The McGraw·Hill Companies

DIGITAL SIGNAL PROCESSORS

Architecture, Programming and Applications

Second Edition

About the Authors

B Venkataramani is presently working as Professor in the Department of Electronics and Communication Engineering. He has been a faculty of the National Institute of Technology (previously referred to as Regional Engineering College), Tiruchirappalli, since 1987 . Prior to that, he worked for BEL, Bangalore, and IIT Kanpur for about three years each. He has executed various development and research projects in DSP and VLSI and has authored two books and published/presented several research papers in various international and national journals/conferences respectively. He has guided five PhD scholars and mentored several students in presenting their M Tech and MS theses. His research interests include design of FPGA as well as P-DSP based speech-recognition systems, software-defined radio and sensor networks.

M Bhaskar is currently working as Associate Professor in the Department of Electronics and Communication Engineering and teaches the subjects of low power VLSI and DSP architecture, programming and applications. He has been a faculty of the National Institute of Technology (previously referred to as Regional Engineering College), Tiruchirappalli, since 1997. He has authored one book and published/presented several research papers in various international and national journals/conferences respectively. He has guided several M Tech students in presenting projects on ‘C54X and ‘C6X based system designs. He has also executed several development projects in DSP. His research interests include design of FPGA based coprocessors for P-DSP and design of low-power interconnects for high-speed VLSI.

The McGraw·Hill Companies

DIGITAL SIGNAL PROCESSORS

Architecture, Programming and Applications

Second Edition

B Venkataramani

Professor
Department of Electronics and Communication Engineering
National Institute of Technology
Tiruchirappalli, Tamil Nadu

M Bhaskar

Associate Professor
Department of Electronics and Communication Engineering
National Institute of Technology
Tiruchirappalli, Tamil Nadu

Tata McGraw Hill Education Private Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Published by Tata McGraw Hill Education Private Limited,
7 West Patel Nagar, New Delhi 110 008

Digital Signal Processors, 2e

ISBN (13): 978-0-07-070256-1
ISBN (10): 0-07-070256-X

Vice President and Managing Director—McGraw-Hill Education, Asia Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Manager—Sponsoring: SEM & Tech Ed: *Shalini Jha*
Assoc. Sponsoring Editor: *Suman Sen*
Development Editor: *Manish Choudhary*
Executive—Editorial Services: *Sohini Mukherjee*
Sr Production Manager: *P L Pandita*

Dy Marketing Manager—SEM & Tech Ed: *Biju Ganesan*

General Manager—Production: *Rajender P Ghansela*
Asst. General Manager—Production: *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at Text-o-Graphics, B1/56 Arawali Apartment, Sector 34, Noida 201 301 and
printed at Sheel Printers Pvt. Ltd., D-132, Hoisery Complex, Phase-II, Noida 201 305

Cover Printer: Rashtriya Printers

RAXQCRQZDRZAX

The McGraw·Hill Companies

Dedicated to
Our Parents

The McGraw·Hill Companies

CONTENTS

PREFACE

Brief Overview

Digital Signal Processors (DSPs) are microprocessors specifically designed to handle Digital Signal Processing tasks and are deployed in a variety of applications like hard-disk controllers, cellular phones, speech-recognition systems, image processing, wireless communication systems, and so on. They are replacing conventional microprocessors in several applications. DSPs from companies such as Analog Devices, Motorola and Texas Instruments are deployed in all these applications.

This book presents details of DSPs from Texas Instruments (TI) in greater depth as compared to the DSPs from other vendors. The TI processors are used in major universities, institutes and in the industry. TI has been donating DSP kits and literature to universities periodically under the University Program called UNITI. The individual institutions have supplemented this with their own funding and have set up DSP labs.

Courses on Digital Signal Processing have undergone a gradual change during the last decade. The focus is shifting gradually from the design of DSP systems and algorithms to efficient implementation of the systems and algorithms. To facilitate this, the subject of DSP Architecture and Programming is now included by many leading institutions in the main curriculum. However, students have to generally rely on the data manuals of various companies for their study since formal textbooks are not readily available. This book has fulfilled the student's requirements and has been used extensively in universities and leading institutions.

Aim of the Revision

The first edition of this text was published eight years back in 2002. Since then, a lot of new DSPs have evolved due to continuous research and development in this field.

The objective of this revision is to include the recent developments in the field of Digital Signal Processors including TMSC6X Series and FPGA based system design methodology. We also aim to bridge the gap in topical coverage in the current edition and improve the pedagogical features to meet the students' requirements.

New to this Edition

In the revised edition, the introductory chapter is expanded with more real-world applications. This includes power spectrum estimation, orthogonal frequency division multiplexing, algorithm for the computation of 1D and 2D discrete wavelet transforms and JPEG2000.

Some of the digital signal processors such as 55X and 6X were treated in brief in the first edition. Since a large number of systems are implemented using these processors, a more detailed treatment of these chapters are given in this edition.

The last chapter in the previous edition had a brief introduction on the FPGA based system. However, FPGAs are now deployed in many high-speed applications such as network routers and front ends of software-defined as well as cognitive radio. In view of these, more details of the FPGA based system design including implementation of system on programmable chips are presented in this revised edition.

In order to illustrate the use of FPGAs and PDSPs in Digital radio receiver, a separate chapter is devoted for the presentation of the design details of the various blocks of a radio receiver with digital hardware and the case study of a software-defined spread spectrum transmitter and receiver is presented. The pedagogy is also refreshed with inclusion of new review questions, multiple choice questions and new programs. The chapter on 'Motorola DSP563XX Processors' is uploaded on the website.

To summarise, the changes made to this edition are the following:

New Chapters

❖ TMS320C6X Assembly Language Instructions (Chapter 14)
❖ Architecture and Application Programs of TMS320C55X (Chapter 16)
❖ FPGAs in Telecommunication Applications (Chapter 18)

New Topical Inclusion

❖ Convolution and real time filtering using FFT
❖ OFDM Using FFT
❖ Data Paths in TMS320C6X

Organization of the Book

Chapter 1 presents an overview of DSP principles, algorithms and applications. At the beginning of this chapter, a simple treatment on DSP theory, algorithms and applications is presented for students having no prior knowledge of DSPs. Introduction to DSP architecture and comparison of this with that of µPs, DSPs and RISC processors is given in **Chapter 2. Chapters 3, 4, 5** and **6** present the detailed architecture, addressing modes, instruction sets, and pipelining and application programs on TMS320C5xDSP. The corresponding details on TMS320C3X are presented in **chapters 7, 8,** and **9. Chapter 10** introduces the TMS30C54X and presents a comparison of the features of 5X with that of 54X. The instruction set and addressing modes of 54X are discussed in **Chapter 11.** Application programs on 54X and program development using Code Composer Studio are presented in **Chapter 12.** Architecture, assembly-language instructions, application programs and peripherals of TMS320C6X are given in **chapters 13, 14** and **15** respectively. Architecture of TMS320C55X is explained in **Chapter 16. Chapter 17** gives a list of some of the recent DSP application case studies and introduces an alternate DSP system design approach using programmable logic devices and FPGAs. Examples of architectures of two leading FPGA families and hardcore as well as softcore processors for these families are explained in this chapter. Algorithms for efficient implementation of DSP systems in FPGA are also given here. **Chapter 18** explains the different applications of FPGAs in telecommunication.

Web Supplements

The web supplements can be accessed at, *http://www.mhhe.com/venkataramani/dsp2* and contain the following material:

Instructor Resources

- Solution manual
- PowerPoint lecture slides

Student Resources

- Interactive quiz
- Chapter on *Overview of Motorola DSP563XXX Processors*

Acknowledgements

We are thankful to Texas Instruments for providing the DSP kits on various processors. We thank our alumni in TI who have been helping us to upgrade our DSP laboratory from time to time. A special note of thanks goes to the UNITI coordinator and his team for extending his wholehearted encouragement and necessary approval to use the data books of TI. Motorola was equally cooperative and allowed us to include the material on their DSPs. Hence, we would like to express our gratitude to the Motorola team. Xilinx and Altera also deserve a special mention for letting us use their material for the revised edition.

We are indebted to our former principal, Dr M Arumugam, and former head of the department, Dr N Kalyanasundaram, who extended all the necessary support and encouragement to offer the laboratory course based on Digital Signal Processors not only to our undergraduate and postgraduate students but also to the students from other colleges under the continuing education programme twice a year. The first edition of this book is borne out of the experience we gained by offering this course for five years to the regular students and to the students under the continuing education programme. The projects carried out by them have been helpful in clearing many of the practical implementation issues. We would like to express our gratitude to all our students who in the process of learning have also taught us. In this regard, we would especially like to mention the names of Vaidyanathan, Amudha, Balaji, Arun, Praveen, Karthikeyan, Jeyendran, Radhakrishnan who have contributed to this project in many ways.

Revision of this book has been made possible because of the experience gained by the implementation and study of various systems on both programmable DSPs and FPGAs by undergraduate, graduate, PhD students and project staff in addition to our experience through teaching. We would like to thank the following students and staff who have been responsible for carrying out the above work: Dr V Amudha, Dr G Seetharaman, Dr S Ramasamy, J Manikandan, S Mohanasundaram, A Geethanath, L Govinda Rao, G Chaitanya, T Dhirendrakumar, H Reshmi, D S Prasada Reddy and S Ashish. We would like to thank our Director Dr M Chidambaram for encouraging us to bring out the revised edition of this book.

A number of experts have provided invaluable suggestions and feedback for the revision of this book. Our heartfelt gratitude goes out to all of them.

Finally, we thank the editorial and production staff of Tata McGraw-Hill for the initiative and interest shown by them in bringing out this work in a short span of time.

B Venkataramani

M Bhaskar

Feedback

Tata McGraw-Hill invites comments, views and suggestions from readers, all of which can be sent to *tmh.ecefeedback@gmail.com*, mentioning the title and author's names in the subject line.

The McGraw·Hill Companies

The McGraw·Hill Companies

1 AN OVERVIEW OF DIGITAL SIGNAL PROCESSING AND ITS APPLICATIONS

SIGNALS AND THEIR ORIGIN 1.1

A signal refers to any continuous function $f(\ )$ which is a function of one or more variables like time, space, frequency, etc. Some common examples of signals are the voltage across a resistor, the velocity of a vehicle, light intensity of an image, temperature, pressure inside a system, etc., as a function of time, space or any other independent variable. These signals are processed in order to either monitor or control one or more parameters of a system. Detection of the average, RMS or peak values of a parameter, separation between adjacent peaks or zero crossings are examples of some processing carried out on the signal. In some applications, the processing may be done in order to produce another signal which has better characteristics than the original signal. The processing of the signal is carried out efficiently and with ease if these signals are converted to equivalent electrical voltages or currents using transducers. Hence, the emphasis in this book will be restricted to processing signals in electrical form.

NOISE 1.2

Processing of the signal is made complex by the presence of other signals called noise. The noise signals are generated from man made and natural objects. Electrical appliances/machinery, lightening and thunderstorms are some of the sources of noise. In addition to this, any signal which interferes with the detection of the desired signal may be called as an intereference or noise. The first step in signal processing is to combat the effect of noise. When the noise and the desired signal have different characteristics, the signal can be completely separated from the noise before processing the signal further. Even though this step is an overhead, this may be mandatory. Let us consider the following example to illustrate this.

FILTERS AND NOISE 1.3

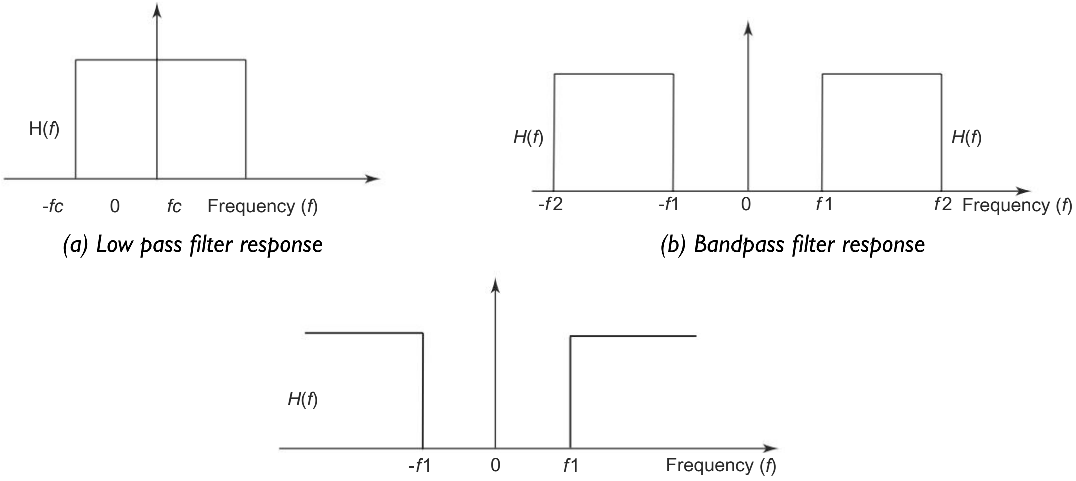
Frequency shift keying is a technique adopted for transmitting binary data from one place to another. For example, the transmitted sinusoidal signal may be chosen to be 1025 Hz or 1225 Hz depending upon whether 0 or 1 is to be transmitted.

On the receiver, the signal frequency is determined in order to decode the transmitted data to be 0 or 1. One of the techniques adopted for determining the frequency (also called *frequency detection*) is to count the number of zero crossings of the signal in a given period of time. This can be done efficiently if the transmitted signal is received without noise. However, one of the common noise which appears at the receiver input is the power supply ripple. A noise of 60/50 Hz frequency may appear at the input to the receiver. Alternately, if the receiver site has any high frequency oscillator, it may get leaked through the power supply lines and will appear at the input to the receiver.

Both these types of noise will make the detector to take wrong decisions. However, its performance can be improved if these two types of noise are removed from the received signal before it is fed to detector. This can be achieved by using a low pass filter for removing the high frequency signal and high pass filter for removing the low frequency power supply ripple. The ideal characteristics of low pass, high pass and band pass filters are shown in Fig. 1.1. These filters may be constructed using discrete components and their frequency response do not have the ideal characteristics. The ideal filters pass the signals in the pass band without any attenuation. The signals in the stop band have infinite attenuation. The transition from pass band to stop band occurs instantaneously as the frequency of the signal is swept. The frequency at which this occurs is called as the cut off frequency *fc*. These filters by themselves qualify to be called as signal processors as they remove the unwanted frequency components. Additional processing may be required to carry out a particular task like frequency detection.

CORRELATORS 1.4

However, separation of the signal from the noise cannot always be achieved using simple filters shown above. The desired signal and the interfering noise may both lie in the same frequency range (bandwidth). In this case, the signal may be retrieved from noise by exploiting its behaviour in the time domain. The effect of the interference signal may be minimized by multiplying the received signal with the replica of all possible transmitted signals with suitable delay and then integrating them over one period (for e.g., 1 bit duration in the case of FSK) of the transmitted signal. This operation is called *correlation*.



(a) Low pass filter response

(b) Bandpass filter response

(c) Highpass filter response

Fig. 1.1

CONVOLUTION AND INVERSE FILTERING 1.5

On its transit to the receiver, the transmitted signal may pass through several systems and each of these systems may modify it. The output $y(t)$ of a linear time invariant (LTI) causal system can be expressed as the convolution of the input $x(t)$, with the impulse response $h(t)$ which is given by the expression:

$$y(t) = \int_{0}^{t} x(\tau)h(t-\tau)d\tau = x(t) * h(t) \tag{1.1}$$

Here $*$ denotes the convolution operation .

A system is said to be linear if the superposition principle is true. In other words if $y_1(t)$ is the response to the input $x_1(t)$ and $y_2(t)$ is the response to the input $x_2(t)$, then for any scalar α, β let the response of the system to the input $\alpha x_1(t) + \beta x_2(t)$ be $y(t)$. If the system is linear then

$$y(t) = \alpha y_1(t) + \beta y_2(t) \tag{1.2}$$

A system is said to be time invariant if a shift in the input causes a corresponding shift in the output. In other words if $y(t)$ is the response of the system to $x(t)$, then for a time invariant system the response to the time shifted input $x(t - \tau)$ is given by $y(t - \tau)$.

A system is said to be causal if the output at any instant is determined only by its present input and its past input/outputs but not by its future inputs. Only causal system can be realised in practice and this requires the impulse response of the system $h(t)$ to be zero for $t < 0$

For simplicity and to a good accuracy, the systems can be modelled to be LTI and causal in majority of signal processing applications. However, there are applications such as image processing system where causality is not true.

In order to nullify the effect of a system on the transmitted signal, the received signal may be passed through another system whose transfer function (Laplace transform of the impulse response) is the inverse of the original system. Inverse filters and equalisers use this principle.

FOURIER TRANSFORM AND CONVOLUTION THEOREM 1.6

The signals may be processed either in the time domain or in the transform domain, e.g., computation of the output $y(t)$ using the convolution integral given by (1.1) is an example of time domain processing. The output may also be computed using transform domain techniques. This can be explained as follows: A signal is said to be of finite energy if

$$\int_{-\infty}^{\infty} |f(\tau)|^2 \, d\tau < \infty \tag{1.3}$$

Any finite energy signal $f(t)$ can be represented using the Fourier transform $F(\omega)$ given by

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-j\omega t} dt \tag{1.4}$$

$f(t)$ can be obtained from $F(\omega)$ using the inverse Fourier transform given by

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{j\omega t} \, d\omega \tag{1.5}$$

When the lower limit in (1.4) and (1.5) are 0 and $j\omega$ is replaced by s, we get the expression for the Laplace transform of $f(t)$ and inverse Laplace transform of $F(s)$ respectively.

Convolution theorem relates $h(t)$, $x(t)$ and the convolved output $y(t)$ to their Fourier and Laplace Transforms. Let the Fourier transform and the Laplace Transform of $(x(t), h(t), y(t))$ be $(X(\omega), H(\omega), Y(\omega))$ and $(X(s), H(s), Y(s))$ respectively.

Then if $y(t) = x(t) * h(t)$ then $Y(\omega) = X(\omega)\, H(\omega)$ and $Y(s) = X(s)\, H(s)$.

Similarly if $Y(\omega) = A(\omega) * B(\omega)$ then $y(t) = a(t).\, b(t)$

and $Y(s) = A(s) * B(s) \Rightarrow y(t) = a(t).\, b(t)$

As mentioned earlier $*$ denotes the convolution operation. $H(s)$, the Laplace Transform of the impulse response $h(t)$ of a causal LTI system is called as the transfer function of the system and is given by

$$H(s) = \frac{Y(s)}{X(s)} \tag{1.6}$$

where $X(s)$, $Y(s)$ are the Laplace transforms of $x(t)$ and $y(t)$ respectively. Laplace Transforms have a no. of applications like studying the stability of the system, solving the differential equations and finding the initial value and final value of a system.

SAMPLING THEOREM AND DISCRETE TIME SYSTEM 1.7

Various signal processing operations explained above can be carried out either directly on the continuous signal or indirectly using the samples of the input signal and impulse responses. Accordingly, they are called as continuous time signal processing and discrete time signal processing respectively. The discrete time signal offers no. of advantages. Firstly, it permits a no. of such signals to be transmitted using the same channel by sending them at disjoint time intervals. This technique is called as the time division multiplexing. However, in order to transmit the information without any loss of information the discrete time signal should satisfy the Nyquist's sampling theorem which states:

Any signal bandlimited to a maximum frequency of *fm* can be perfectly reconstructed from its samples if the sampling rate, *fs* is greater than or equal to 2*fm*. If *fs* is equal to 2*fm*, then it is called the *Nyquist rate*.

If the sampling rate is less than 2*fm*, then any signal component *fh* which is greater than *fs*/2 by Δf (i.e., $fh = fs/2 + \Delta f$) gets mapped to a frequency $fs/2 - \Delta f$ after sampling and appears as a low frequency signal. This is called as aliasing. To avoid this, either the sampling rate should be chosen to be above Nyquist rate or the sampler should be preceded by a low pass filter with cut off frequency, $fc = fs/2$.

LINEARITY, SHIFT INVARIANCE, CAUSALITY AND STABILITY OF DISCRETE TIME SYSTEMS 1.8

Some of the properties of the continuous time system discussed in Section 1.5 can be extended for the discrete time system as follows: A discrete time system is said to be LSI if the superposition property holds and a shift in the input causes a corresponding shift in the output.

In other words if $y_1(n)$ is the response of a discrete time system to the input $x_1(n)$ and $y_2(n)$ is the response to the input $x_2(n)$, then for any scalar α, β let the response of the system to the input $\alpha\, x_1(n) + \beta\, x_2(n)$ be $y(n)$. Then if the system is linear

$$y(n) = \alpha\, y_1(n) + \beta\, y_2(n) \tag{1.7}$$

The discrete time system is said to be shift invariant if a shift in the input causes a corresponding shift in the output. In other words if $y(n)$ is the response of the system to $x(n)$, then for a shift invariant system the response to the input $x(n-k)$ is given by $y(n-k)$. A LSI system can also be called as linear time invariant (LTI) system if the shift in the index (e.g., k in $N-k$) corresponds to a different sampling instant.

A discrete time system is causal if the impulse response sequence $h(n) = 0$ for $n < 0$. The system is stable, if a bounded input sequence $x(n)$ i.e a sequence for which $|x(n)| < \eta$ for all n and any arbitrary η results in bounded output sequence $y(n)$. This is achieved if

$$\sum_{k=0}^{\infty} |h(k)| < \infty \tag{1.8}$$

Z TRANSFORM 1.9

Analogous to the Laplace transform for the continuous time system, for the discrete time system, unilateral Z transform is defined. The unilateral Z transform of a sequence $x(n)$ is denoted as $X(z)$ and is given by

$$X(z) = \sum_{n=0}^{\infty} x(n) z^{-n} \tag{1.9}$$

Here, z denotes a complex variable.

The z transform $H(z)$ of the impulse response sequence $h(n)$ of a causal LSI system is called as the system function. $H(z)$ can be used to examine the stability of a discrete time system. For a stable system, the magnitude of the poles of $H(z)$ should be < 1. In other words, the magnitude of the points at which the denominator of $H(z)$ becomes zero should be less than 1, in a system with system function $H(z)$ given by

$$H(z) = \frac{1}{(1 - az^{-1})} \tag{1.10}$$

$|a|$ should be < 1 for the system to be stable.

1.9.1 Additional Properties of *z* transform

1. Linearity If $x(n)$ and $X(z)$ are z transform pairs (denoted as $x(n) \leftrightarrow X(z)$) and $y(n) \leftrightarrow Y(z)$ then

$$(\alpha\, x(n) + \beta\, y(n)) \leftrightarrow (\alpha\, X(z) + \beta\, Y(z))$$

2. Multiplication by Exponential Sequence If $x(n) \leftrightarrow X(k)$, then $a^n x(n) \leftrightarrow X(a^{-1}z)$

3. Initial and Final Value Theorems The initial value $x(0)$ and final value $x(\infty)$ can be evaluated using the Z transform $X(z)$ as follows:

$$x(0) = \lim_{z \to \infty} X(z) \tag{1.11}$$

$$x(\infty) = \lim_{z \to 1} (z-1) X(z) \tag{1.12}$$

4. Differentiation of $X(k)$ If $x(n) \leftrightarrow X(z)$ then

$$nx(n) \leftrightarrow -z\frac{d}{dz}X(z)$$

5. Convolution The convolution sum $y(n)$ of two sequences $x(n)$ and $h(n)$ denoted as $x(n) * h(n)$ is given by

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k) \tag{1.13}$$

If $x(n) \leftrightarrow X(z)$, $h(n) \leftrightarrow H(z)$ and $y(n) \leftrightarrow Y(z)$ and then $(x(n) * h(n)) \leftrightarrow X(z)\,H(z) = Y(z)$. Similarly $(a(n)\,b(n)) \leftrightarrow (A(z) * H(z))$

6. Shifting Property If $X(z)$ is the z transform of the sequence $x(n)$, then the z transform of the sequence $x(n-k)$ is given by $X(z)z^{-k}$. This property can be used for solving a difference equation. For e.g., consider the output sequence $y(n)$ of a LSI system expressed in terms of the input sequence $x(n)$ by the difference equation

$$y(n) = x(n) + a\,y(n-1) \tag{1.14}$$

Taking the z transform of each of the terms of (1.14) and using the shifting property $Y(z)$ is given by

$$Y(z) = X(z) + (az^{-1})\,Y(z)$$

Rearranging the terms we get

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{(1-az^{-1})} \tag{1.15}$$

The impulse response of the system can be found by taking the inverse z transform of (1.15). In general, z transform can be used to convert a difference equation into an algebraic equation involving the z transforms of the sequences.

7. The inverse z transform of a sequence can be obtained either using the power series expansion of $X(k)$ or using partial fraction expansion, e.g., if $X(z)$ is given by (1.10), its power series expansion is given by

$$X(z) = 1 + (az^{-1}) + (az^{-1})^2 + (az^{-1})^3 + \cdots = \sum_{n=0}^{\infty} a^n z^{-n} \tag{1.16}$$

Comparing this with (1.9), it can be concluded that $x(n) = a^n$
Similarly if

$$H(z) = \frac{1}{(1-az^{-1})(1-bz^{-1})} \tag{1.17}$$

Using partial fraction expansion $H(z)$ can be rewritten as

$$H(z) = \frac{a}{(a-b)}\frac{1}{(1-az^{-1})} + \frac{b}{(b-a)}\frac{1}{(1-bz^{-1})} \tag{1.18}$$

Comparing (1.18) with (1.10) it can be concluded that

$$h(n) = \frac{a}{(a-b)}a^n + \frac{b}{(b-a)}b^n \tag{1.19}$$

Z transform can be used to arrive at efficient schemes for implementation of discrete time systems. This is considered in Section 1.13.

FREQUENCY RESPONSE OF LTI DISCRETE TIME SYSTEM 1.10

The response $y(n)$ of a LTI discrete time system with impulse response $h(n)$ for a complex exponential sequence given by

$x(n) = e^{jn\omega}$ for $-\propto < n < \propto$

is given by

$$y(n) = \sum_{m=-\infty}^{\infty} h(m)e^{j(n-m)\omega} \tag{1.20}$$

Let $H(e^{j\omega})$ be defined as

$$H(e^{j\omega}) = \sum_{m=-\infty}^{\infty} h(m)e^{-jm\omega} \tag{1.21}$$

Using (1.21) in (1.20), $y(n)$ is given by

$$y(n) = e^{jn\omega}\sum_{m=-\infty}^{\infty} h(m)e^{-jm\omega} = e^{jn\omega}H(e^{j\omega}) = x(n)H(e^{j\omega}) \tag{1.22}$$

Hence the complex exponential sequence is the eigen function of the discrete LTI system . Since it corresponds to the sampled sinusoid of frequency ω, the response, $H(e^{j\omega})$ is called as the frequency response of the system.

Properties

1. $H(e^{j\omega})$ can be obtained from the transfer function $H(z)$ by evaluating it at $z = e^{j\omega}$. This can be verified by comparing (1.21) with (1.9) .
2. $H(e^{j\omega})$ is periodic in ω with period equal to 2π. In the sampled data system $y(n)$ actually denotes the value $y(t)$ at the nth sampling instant nTs. The dependence of the sampling frequency on the frequency response can be made clear by replacing ω by ωTs. Hence for the sampled data system with sampling interval Ts, the frequency response is given by $H(e^{j\omega Ts})$ and it is periodic with a period of $(2\pi/Ts)$. Since ω is equal to $2\pi f$, when the frequency response is plotted as a function of f, it is periodic with a period of $(1/Ts)$.
3. As $H(e^{j\omega})$ is periodic in ω with period of 2π, (1.21) can be viewed as the Fourier series expansion for $H(e^{j\omega})$ with the Fourier coefficients $h(n)$ given by

$$h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(e^{j\omega})e^{j\omega n}d\omega \tag{1.23}$$

For real values of $h(n)$, $|H(e^{j\omega})|$, the magnitude of the frequency response is symmetric in the interval $(0, 2\pi)$ and the phase response is antisymmetric in this interval.

DIGITAL SIGNAL PROCESSING 1.11

As mentioned in Section 1.7, one of the advantages of discrete time signal is time division multiplexing. The second advantage of discrete time signal is that they can be digitized and they can be processed either using digital hardware or using software. For the digitization, firstly, the discrete time signal which can take any value in the range ($-A_m$, A_m) where A_m denotes the maximum amplitude of the signal is approximated to one of the $2N$ levels which is closest to the signal. The approximation error is called the *quantisation error* and it can be made small by choosing N to be large.

The $2N$ levels may be chosen to be at equal intervals in the range ($-A_m$, A_m) in which case the signal is said to be uniformly quantized. The next step is to represent each of these $2N$ levels by an n bit number. The number of bits, n is given by $\log_2 2N$. Processing the n bit numbers corresponding to the various samples of the signal is called as digital signal processing. It offers a number of advantages compared to processing the continuous time signal directly. The latter approach is also called *analog signal processing* if the signal amplitude range is also continuous.

ADVANTAGES OF DIGITAL SIGNAL PROCESSING (DSP) 1.12

1.12.1 Ease of Processing

One of the requirements in signal processing is to delay the signal by a particular duration. For example in moving target indicator radar, a number of pulses are transmitted one after another and the received signal corresponding to adjacent pulses are to be subtracted. This requires the received signal to be delayed by one pulse repetition interval. For the analog signal processing, this is achieved using acoustic delay line. Increasing or decreasing the delay requires the delay line length to be changed which is cumbersome. On the other hand, for DSP, the samples of the received signal can be stored in memory and they can be read after one pulse repetition interval. Delay can be easily changed without switching in or switching out cables.

1.12.2 Thermal Drift and Reliability

For analog signal processing, circuits consisting of analog components like resistors, capacitors, and op amps are used and their characteristics change with temperature. DSP circuits use digital hardware like adders, multipliers and shift registers whose characteristics show no variation with temperature throughout their operating range. Component aging alters the performance of the analog circuit. For example, the dielectric material of capacitors is particularly prone to aging which changes the impedance and alters the performance. DSP circuits have the same performance throughout their life time.

1.12.3 Repeatability

Component tolerance makes the analog circuit to have different characteristics with different set of components of same nominal value. For example, a resistor with 100 Ω resistor with 10% tolerance can have any value in the range (90, 110) Ω. Accordingly, the circuit behaviour cannot be exactly predicted. This problem can be minimised by using components with smaller tolerance. But this increases the system cost as these components are costly. Typical capacitors have a tolerance of 20% or worse. This makes the characteristics of an analog circuit to be poorly repeatable with new set of components with same nominal values. Digital circuits, however, are inherently repeatable.

1.12.4 Immunity to Noise

In analog signal processing, the signals are allowed to take any value within a particular range and noise can easily alter the magnitude. In DSP, the signals take binary values and for altering a 1 to 0 and vice versa a noise voltage of a large magnitude is required. In digital data transmission, the effect of noise can be completely eliminated by putting repeaters at suitable intervals. However, a repeater used with an analog signal will amplify both signal and noise and will be ineffective in combating the noise.

1.12.5 Programmability

DSP functions can be implemented either using microprocessors/microcontrollers or Programmable Digital Signal Processors. This enables the same hardware configuration to be reprogrammed to perform a very wide variety of signal processing tasks by loading in different software. In the analog case this would call for rewiring/resoldering.

1.12.6 Some Special Signal Processing Techniques

There are some signal processing techniques that cannot be performed by analog systems. Examples of these techniques are linear phase filters, notch filters, adaptive filters, data compression, errol control coding. In control systems, an example is a deadbeat controller used when a very rapid settling time is required.

DSP IN THE SAMPLE AND TRANSFORM DOMAIN 1.13

For a discrete time system, the convolution integral given by eqn. (1.1) reduces to the convolution sum given by

$$y(n) = \sum_{k=0}^{n} x(n-k)h(k) \tag{1.24}$$

where $y(n)$, $x(n)$ and $h(n)$ are the n^{th} sample of output, input and the impulse response of the LTI causal discrete time system and $x(n)$ and $h(n)$ are assumed to be 0 for $n < 0$. If $x(t)$ and $h(t)$ are of finite duration they may be represented faithfully using M, N samples respectively. In this case the output is also of finite duration and can be represented using $M+N-1$ samples. In other words convolution of two sequences of length M, N results in a sequence of length $M+N-1$. For finding $y(n)$, $n+1$ multiplications and n additions are required. For large values of n, this may call for significant computational effort; $y(n)$ may be alternately computed using an indirect approach using the transforms of the input and the impulse response. One of the transform that may be used is the Discrete Fourier transform (DFT). The DFT $X(k)$ of a sequence $x(n)$ of finite length M, for $k = 0, \ldots M-1$ is defined as

$$X(k) = \sum_{n=0}^{M-1} x(n) e^{-j\left(\frac{2\pi}{M}\right)kn} \tag{1.25}$$

The computation of the M DFT coefficients $X(0), \ldots X(M-1)$ requires M^2 complex multiplications. However, using the Fast Fourier Transform (FFT) algorithm, the no. of multiplications required is reduced to $\frac{\mathrm{M}}{2}\log_2\mathrm{M}$.

The inverse DFT (IDFT) is used to find $x(n)$ from $X(k)$ using the expression given by

$$x(n) = \frac{1}{M}\sum_{k=0}^{M-1} X(k)e^{j\left(\frac{2\pi}{M}\right)kn} \tag{1.26}$$

IDFT can also be computed using FFT.

FAST FOURIER TRANSFORM (FFT) 1.14

The direct computation of DFT of a sequence of length N requires N^2 complex multiplications. The number of multiplications can be reduced to $\frac{N}{2}\log_2 N$ using the FFT algorithm. There are two popular FFT algorithms: Decimation in Time (DIT) algorithm and Decimation in Frequency (DIF) algorithm. These two algorithms are dealt in detail in the literature. For the sake of brevity, one of the algorithms, DIT is presented here.

1.14.1 Decimation in Time Algorithm

The first step in this algorithm is to rearrange the sequence in the bit reversed order. In the bit reversed number representation, the binary pattern corresponding to a particular decimal number is obtained by writing the natural binary equivalent of the number in the reverse order so that the most significant bit of the natural binary number becomes the least significant bit of the bit reversed number and vice versa. Using this rule, it can be verified that the binary equivalent of the numbers 0–15 is as shown in Table 1.1.

Let the input sequence be $x(0)$, $x(1)$ ... x $(N–1)$. Let the bit reversed sequence be $x_1(0)$, $x_1(1)$ ... $x_1(N–1)$. The computation of FFT involves log N stages of computation.

In the first stage, 2 point DFT of blocks of every consecutive two elements of $x_1(0)$ ... $x_1(N–1)$ is computed and the resulting sequence is denoted as $X_1(0)$, $X_1(1)$, ... $X_1(N–1)$ as shown in Fig. 1.3. The manner in which $X_1(k)$ is obtained from $x_1(l)$, for k,l = 0, 1, ... N–1 can be explained using the FFT butterfly diagram shown in Fig. 1.2. Let W_N be given by

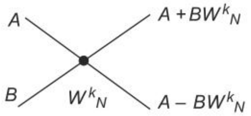


Fig. 1.2 *FFT Butterfly with a twiddle factor of* W_N^k

Table 1.1 *Natural binary numbers and their bit reversed equivalents*

Decimal number	*Natural binary number*	*Bit reversed number*	*Decimal number*	*Natural binary number*	*Bit reversed number*
0	0000	0000	8	1000	0001
1	0001	1000	9	1001	1001
2	0010	0100	10	1010	0101
3	0011	1100	11	1011	1101
4	0100	0010	12	1100	0011
5	0101	1010	13	1101	1011
6	0110	0110	14	1110	0111
7	0111	1110	15	1111	1111

$$W_N = e^{-j(2\pi/N)} \tag{1.27}$$

The multiplication factor $W_N{}^K$ is called the *twiddle factor*. For the 2 point DFT, the value of k is N and hence the twiddle factor is equal to one.

In the second stage, every consecutive block of 4 elements of $X_1(0)$, $X_1(1)$ ... $X_1(N – 1)$ are combined using two FFT butterflies. For n = 0, 1, ... $(N/4 –1)$, the first butterfly is formed between the 4nth element and $(4n + 2)^{th}$ element. The second butterfly is formed between the $(4n + 1)^{\text{th}}$ element and $(4n + 3)^{\text{th}}$ element. The twiddle factor for the first butterfly is W_N^0. The twiddle factor for the second butterfly is $W_N^{N/2}$.

For example, for $N = 8$, the second twiddle factor becomes W_8^4. Let the output of the butterflies at the 2nd stage be $X_2(0), X_2(1), \ldots X_2(N - 1)$.

In the third stage, every consecutive block of 8 elements of $X_2(0), X_2(1) \ldots X_2(N - 1)$ are combined using four FFT butterflies. For $n = 0, (N/8 - 1)$, the first butterfly is formed between the $8n^{\text{th}}$ element and $(8n + 4)^{\text{th}}$ element. The second butterfly is formed between the $(8n + 1)^{\text{th}}$ element and $(8n + 5)^{\text{th}}$ element. The third butterfly is formed between the $(8n + 2)^{\text{th}}$ element and $(8n + 6)^{\text{th}}$ element. The fourth butterfly is formed between the $(8n + 3)^{\text{th}}$ element and $(8n + 7)^{\text{th}}$ element. The twiddle factor for the k^{th} butterfly (for $k = 1, 2, 3, 4$) is obtained as $W_N^{(k-1)N/4}$. For example, for $N = 8$, the four twiddle factors are W_8^0, W_8^2, W_8^4, W_8^6 respectively. Let the resulting output of the butterflies be $X_2(0), X_2(1), \ldots X_2(N - 1)$.

The procedure can be generalized as follows: At the r^{th} stage of FFT computation, every block of consecutive 2^r elements of the output of the previous butterflies $Xr - 1(0), Xr - 1(1) \ldots Xr - 1(N - 1)$ are combined using 2^{r-1} butterflies. For $m = 0, 1, \ldots (N/2^r - 1)$, the k^{th} butterfly is formed between the $(2^r m + k)^{\text{th}}$ element and $(2^r\, m + k + 2^{r-1})^{\text{th}}$ element.

The twiddle factor for the kth butterfly is given by

$$W_N^{(k-1)N/p} \qquad \text{where } p = 2^{r-1}.$$

Twiddle factors at various stages of FFT computation for $N = 4, 8, 16$ are given in Table 1.2. The complete FFT butterfly flow diagram for $N = 8$ is given in Fig. 1.3.

Table 1.2 *Twiddle factors the FFT butterflies for N = 4, 8 16*

	$N = 4$	$N = 8$	$N = 16$
Stage 2 Twiddle factor	$1, W_4^2$	$1, W_8^4$	$1, W_{16}^8$
Stage 3 Twiddle factor		$1, W_8^2$ W_8^4, W_8^6	$1, W_{16}^4$ W_{16}^8, W_{16}^{12}
Stage 4 Twiddle factor			$1, W_{16}^2$ W_{16}^4, W_{16}^6 W_{16}^8, W_{16}^{10} W_{16}^{12}, W_{16}^{14}

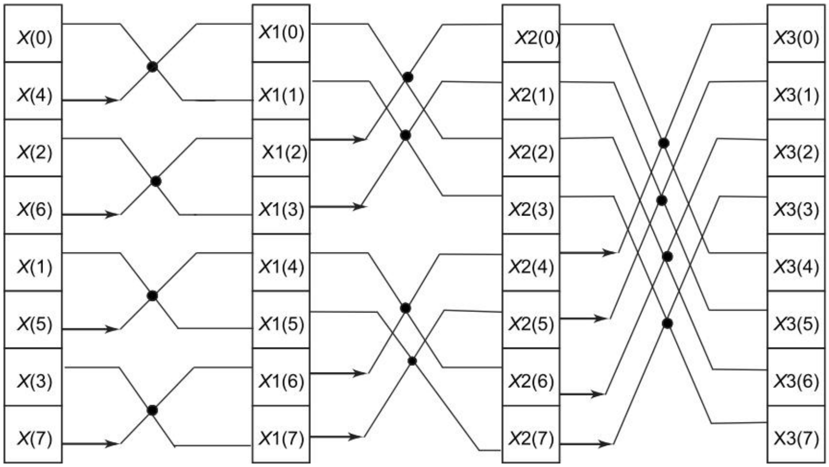


Fig. 1.3 *Decimation-in-time FFT flow diagram for 8 point FFT*

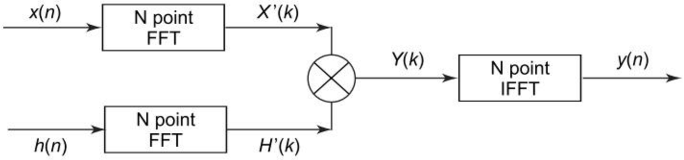


Fig. 1.4 *Convolution using FFT*

1.14.2 Convolution using FFT

The convolution can be done using FFT as shown in Fig.1.4. Let the sequences $x(n)$, $h(n)$ be of length L, M respectively. New sequences $x'(n)$ and $h'(n)$ are formed by appending $M-1$, $L-1$ zeros to $x(n)$ and $h(n)$ respectively at their end. Lengths of both of the sequences become $L+M-1$. Let $N = L+M-1$. The FFT of both of these sequences are found and the corresponding FFT coefficients are multiplied and a new set of coefficients are obtained. ie. $X'(0)$ is multiplied with $H'(0)$, $X'(1)$ is multiplied with $H'(1)$ and so on. The inverse FFT of these coefficients gives the convolution of the sequence $x(n)$ with $h(n)$. The total number of real multiplications required for convolution becomes $4([3N/2]\log_2(N)+N)$.

Using the direct approach the number of multiplications required is LM. This can be verified as follows: Let $L > M$. The first $M-1$ outputs ($y(0) - y(M-1)$) require ($1+2+3+\ldots.\ M-1$) multiplications. The last $M-1$ outputs ($y(n) - y(M+N-2)$) require ($M-1+M-2+\ldots\ 3+2+1$) multiplications respectively. The remaining $L-M+1$ outputs require M multiplications each. Hence, the total no. of multiplications is given by

$$2M \times \frac{M-1}{2} + (L-M+1)\times M = M^2 - M + LM - M^2 + M = LM$$

For small values of L and M, the direct approach is computationally efficient. For large values, the FFT approach is efficient, e.g. for L=M=8, the number of multiplications required using direct, DFT approaches are 64, 444 respectively. For L=M=64, they are 4096, 5888 respectively.

DIGITAL FILTERS 1.15

In Section 1.3, the use of low pass and high pass filters are illustrated. The flatness of the amplitude response in the pass band and the sharpness of the transition from pass band to stop band determines the complexity of the filter or the no. of components required for building the filter. Digital filters use the digital hardware like adder, multiplier and shift registers. The design of digital filters using the transfer function/impulse response of the analog filters has been dealt in detail in several textbooks. (see for e.g., Rabiner, Oppenheim). The digital filters can be classified into broadly two types: Finite Impulse response (FIR) filters and Infinite impulse response (IIR) filters. FIR, IIR filters described below are examples of linear shift invariant system (LSI).

1.15.1 FIR Filters

The output of the FIR filter at the n^{th} sampling instant $y(n)$, can be expressed as the function of the present as well as the past (M–1) input samples and the impulse response sequence $h(k)$ as follows:

$$y(n) = \sum_{k=0}^{M-1} x(n-k)h(k) = \sum_{k=0}^{M-1} x_{n-k}h_k \qquad (1.28)$$

This is a difference equation of order M. It is also referred to as tapped delay line filter with M taps or FIR filter with M taps. Comparing (1.28) with (1.13), it can be seen that FIR filter is a convolver where the present as well as past $M - 1$ input samples are convolved with M impulse response coefficients. Hence, it can also be called as a convolver. It may be noted that the past $M - 1$ samples can be obtained by storing them in $M - 1$ shift registers. Each shift register introduces a delay equal to 1 sampling interval and has a transfer function of z^{-1}.

With this observation a structure for the implementation of the FIR filter can be obtained as shown in Fig. 1.5. This is also referred to as the direct implementation scheme as it uses M multipliers and adder for M products. In this case, the minimum sampling interval T_{sd} is given by

$$T_{sd} = T_M + (M - 1)T_A \tag{1.28a}$$

where T_M, T_A are respectively the time required for one multiplication, addition respectively. An alternative filter given in Fig. 1.6 is obtained by taking the transpose of the filter structure given in Fig. 1.5. The minimum sampling time, T_{st} of the transpose filter is given by

$$T_{st} = T_M + T_A \tag{1.28b}$$

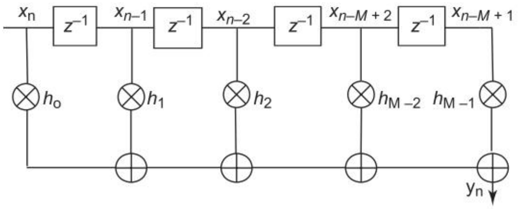


Fig. 1.5 *Direct Implementation scheme for FIR filter*

The transpose of a filter/network is obtained as follows: The direction of the signal flow in each branch of the filter is reversed. The branching points are replaced by adders and adders are replaced by branching nodes. The nodes at which the input is fed and the output is tapped are interchanged. The transfer function of the transpose structure is the same as the original structure. However their other characteristics like the effect of inaccuracy due to finite number of bits used for representing the numbers in a filter (referred to as the finite word length effect) will be different. This structure can also be used for serial/parallel convolution as in this case the input data can be fed serially bit by bit and the output can also be obtained serially.

For implementation either in software or hardware another scheme that can be used is shown in Fig. 1.7. This consists of two circulating registers for storing and circulating M numbers, a multiplier and an accumulator consisting of a register and an adder.

In this case, an M tap filter requires M clock cycles for computation of each output of the filter. If Tc is the clock period, the minimum sampling interval Ts, required for the inputs to this filter is given by

$$T_s = MT_c \tag{1.28c}$$

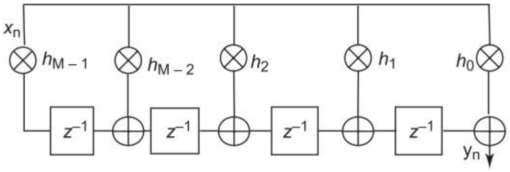


Fig. 1.6 *Serial/parallel convolver*

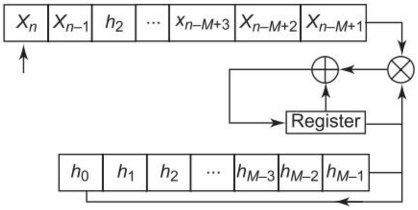


Fig. 1.7 *Direct implementation scheme with single multiplier/adder*

Merits and Demerits of FIR Filters

FIR filters have a numbers of advantages. They are stable and can be made always realizable with suitable delays. The inaccuracy in the output due to the number of bits used for representation of the input samples and the impulse response sequence is smaller in FIR filters than in IIR filters. The inaccuracy that results in both FIR and IIR filter due to the number of bits used for the representation is referred to as finite wordlength effect. For speech processing and data transmission, it is required to have frequency selective filters whose magnitude response is flat and the phase response is linear with frequency in the pass band. Such filters are referred to as linear phase filters and they can be realised using FIR structure. In this case, the impulse response has even symmetry . For example the impulse response of the N tap linear phase FIR filter satisfies

$$h(n) = h(N - 1 - n) \quad 0 \le n < (N/2) \tag{1.29}$$

For example, in an 8 tap linear phase FIR filter, the impulse response coefficients satisfy the conditions:

$$h(7)=h(0),\ h(6)= h(1),\ h(5) =h(2),\ h(4) = h(3) \tag{1.30}$$

This reduces the number of multiplications required for computing (1.28) by half. In hardware implementation of the filter the number of multipliers required is reduced by half.

Another advantage of the FIR filter is that an FIR filter can be designed using well proven techniques for any arbitrary frequency response. IIR filters can be designed only for four basic types like Low pass (LP), High pass (HP), bandpass (BP), Bandstop (BS) and few others. More general filters such as multiband filters are difficult to design using IIR filters but can be designed as FIR filters.

One of the limitations of FIR filter is the need for large number of taps for obtaining filters with sharp cut off characteristics. IIR filters require only less no. of taps compared to FIR filters. More no. of taps require more hardware resources or more computation time. Another disadvantage is the linear phase filter may result in non integral no. of sample delays with frequency and it may not be acceptable in some applications.

1.15.2 Real Time Filtering Using FFT

In real time filters, the input samples to be processed arrive periodically. If an M Tap FIR filter is used for processing, between every sampling interval M multiplications and M-1 additions are to be performed.

Overlap and Save Method

In stead of processing every sample individually, the input sequence may be segmented into overlapping blocks of subsequences of N samples as shown in Fig. 1.8. The no. of samples overlapping between the adjacent block is chosen to be $M - 1$. Let us see what happens when the block of N samples are convolved with the impulse response sequence. The first $M - 1$ outputs would be wrong as we require the present and the past $M - 1$ samples to compute the

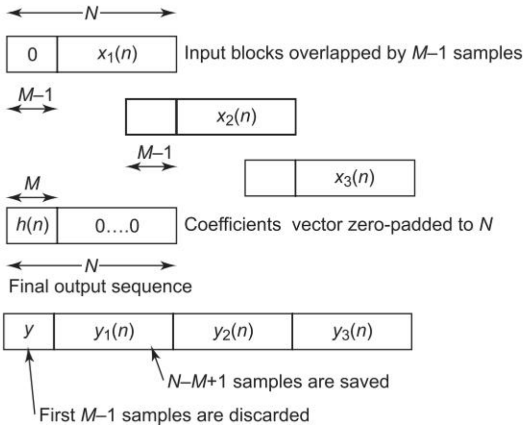

N
0
x1(n)
Input blocks overlapped by M–1 samples
M–1
x2(n)
M–1
x3(n)
M
h(n)
0....0
Coefficients vector zero-padded to N
N
Final output sequence
y
y1(n)
y2(n)
y3(n)
N–M+1 samples are saved
First M–1 samples are discarded

Fig. 1.8 *Real time filtering using overlap and save method*

output. The outputs corresponding to sampling instants M to N would be correct as all the past samples required are contained in the block. The second block contains the samples $N - M + 1$ to $2N - M$. In this block, the outputs corresponding to sampling instants $N - M + 1$ to N would be wrong and the outputs corresponding to sampling instants N to $2N - M$ would be correct. However, the correct outputs corresponding to the instants $N - M + 1$ to N have been already computed using the first block. Hence, in each block, the first $M - 1$ outputs may be discarded and the remaining $N - M + 1$ outputs may be saved. This technique for performing convolution using a block of input samples is called as overlap and save method. The convolution may be efficiently performed using FFT as shown in Section 1.14.2. The following steps are adopted for this purpose.

1. The impulse response sequence $h(n)$ is appended with $N - M$ zeros and the FFT of resulting sequence is found. Denote the FFT coefficients as $H(0) - H(N - 1)$
2. The FFT of the N samples in the i^{th} block $(x_i(0) - x_i(N - 1)$ are found. The FFT coefficients are denoted as $X_i(0) - X_i(N - 1)$
3. Compute $Y_i(k)$ using the relation $Y_i(k) = X_i(k)H(k)$ for $k = 0 - (N - 1)$
4. Compute the inverse DFT of $\{Y_i(0), Y_i(1) \ldots Y_i(N - 1)\}$ and find $y_i(k)$. Discard $y_i(0) - y_i(M - 2)$ and save $y_i(M - 1)$ to $y_i(N - 1)$

Overlap and Add Method

In this method, the input sequence is segmented into non-overlapping blocks of subsequences of L samples as shown in Fig. 1.9.

The samples of the ith block can be written as

$$x_i(n) = x(n + iL) \quad n = 0,1, \ldots L - 1 \qquad (1.31a)$$

The L samples of the i^{th} block are appended with $M - 1$ zeros to form a sequence of length N which is a power of 2. Let us denote this as $x_i'(n)$. $X_i'(k)$, the FFT of this sequence is found. Similarly another sequence $h'(n)$ of length N is formed by appending $L - 1$ zeros to $h(n)$. $H'(k)$, the FFT of this sequence is found. $Y_i(k)$ is computed as the product of $X_i'(k)$ and $H'(k)$. $y_i(n)$ is found by computing inverse FFT of $Y_i(k)$. Each output block contains N output samples. Since the input blocks are non overlapping, out of the N outputs, first $M - 1$ outputs and the last $M - 1$ outputs would be incorrect as at least one sample from the adjacent block is required to get the correct output for these sampling instants. To find the correct output, the outputs corresponding to i^{th} and $(i + 1)^{\text{th}}$ block are aligned as shown in Fig.1.9 such that the last $M - 1$ outputs of i^{th} block overlaps with first $M - 1$ outputs of the $(i + 1)^{\text{th}}$ block. The overlapping outputs are added column wise to get the correct output. L output samples corresponding to i^{th} block are obtained from the N outputs of i^{th} and $(i - 1)^{\text{th}}$ block as follows:

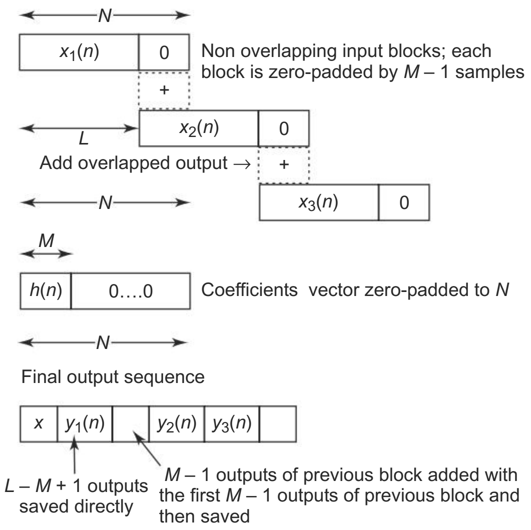

N
x1(n)
0
Non overlapping input blocks; each block is zero-padded by M – 1 samples
+
L
x2(n)
0
Add overlapped output →
+
N
x3(n)
0
M
h(n)
0….0
Coefficients vector zero-padded to N
N
Final output sequence
x
y1(n)
y2(n)
y3(n)
L – M + 1 outputs saved directly
M – 1 outputs of previous block added with the first M – 1 outputs of previous block and then saved

Fig. 1.9 *Real time filtering using overlap and add method*

$$y(iN + k) = y(iN + k) + y(iN - M + 1 + k) \text{ for } k = 0,1, 2, \ldots M–2 \qquad (1.31b)$$

$$y(iN + k) = y(iN + k) \text{ for } k = M - 1, M, \ldots L \qquad (1.31c)$$

This can be verified by finding $y(iN + k)$ for $k = 0$ and $M - 2$ as follows

$$y(iN) = y(iN) + y(\, iN - M + 1) \tag{1.31d}$$

$$y(iN + M - 2) = y(iN + M - 2) + y(\, iN - 1) \tag{1.31e}$$

1.15.3 Design of FIR Filters

The FIR filter may be specified by the frequency response of the filter. This in turn can be specified by the magnitude response and the desired phase response of the filter. A number of techniques have been developed for finding the filter coefficients and computer programs have also been developed to facilitate this. Overview of two of the methods the window method and frequency sampling method is presented next.

Window Method Using the periodicity property of the frequency response, it is shown in section 1.10 that $h(n)$ the impulse response coefficients can be obtained as the Fourier coefficients given by Eqn. (1.23). However, this requires an infinite sequence and the filter will be non causal as $h(n)$ is nonzero for $n < 0$. This series may be truncated to have N coefficients. However this will result in overshoot and ripples in both pass band and stop band and is referred to as *Gibb's oscillation* or phenomena. Another problem with the fourier series technique is its slow convergence. Choosing larger values of N does not automatically reduce the magnitude of the ripples in the transition region from pass to stop band significantly. This problem is overcome by choosing a window sequence $w(n)$ and multiplying it with $h(n)$ obtained using the Fourier series approach. $w(n)$ is chosen so as to obtain a smooth transition from the pass band to the stop band and to be non-zero for $0 \le n \le N - 1$. The resulting $h_w(n)$ is made causal by shifting it by $(N - 1)/2$ towards right. Some of the window functions used are the rectangular window, Von Hann (also referred to as Hanning) window, Hamming window, Blackman window and kaiser window. The expressions for the first three windows can be specified using the parameter α as follows:

$$\begin{aligned} w(n) &= \alpha + (1 - \alpha)\cos\left(2\pi \frac{n}{N-1}\right) \quad \text{for } 0 \le n \le N - 1 \\ &= 0 \text{ otherwise.} \end{aligned} \tag{1.32}$$

The value of α is 1, 0.5, 0.54 respectively for rectangular, Hanning, Hamming windows. The Blackman window function is also of similar form and is given by

$$\begin{aligned} w(n) &= 0.42 - 0.5\cos\left(2\pi \frac{n}{N-1}\right) + 0.08\cos\left(4\pi \frac{n}{N-1}\right) \quad \text{for } 0 \le n \le N{-}1 \\ &= 0 \text{ otherwise.} \end{aligned} \tag{1.33}$$

The frequency response of the window functions have a main lobe and a no. of side lobes. The ripple ratio defined as the ratio of the first side lobe amplitude to the main lobe amplitude in the frequency response of the window function increases with α and the main lobe width decreases with α.

Frequency Sampling Method In this method the value of the frequency response at M points in the interval $(0, (2\pi/Ts))$ is taken. In other words $H(e^{j\omega Ts})$ is evaluated for $\omega = (2\pi k/MTs)$ for $0 \le k \le M - 1$ and are taken to be M DFT coefficients. The inverse DFT of this sequence is taken and is treated as M impulse response samples $h(n)$. Out of these M samples N samples $(N < M)$ is taken to be the impulse response sequence for the FIR filter with N taps. To minimize the Gibb's oscillations, this is combined with one of the window functions discussed in the last section. The choice of the value of N may be done iteratively. Starting with a value of N, the filter frequency response may be evaluated after windowing. If it differs significantly from the desired frequency response, the value of N may be changed. This step is repeated till a satisfactory response is obtained. Several variations of the frequency sampling method is used in practice. For example, instead of choosing all the M DFT coefficients using the frequency response, some of the coefficients in the transition region from pass band to the stop band may be chosen using optimization techniques in order to achieve the required ripple as well as undershoot/overshoot characteristics.

1.15.4 IIR Filters

The output of the IIR filter at the n^{th} sampling instant $y(n)$, can be expressed as the function of the present input and past M-1 input samples as well as the past $N - 1$ output samples and is given by :

$$y(n) = \sum_{k=0}^{M-1} x(n-k)a(k) + \sum_{k=1}^{N-1} y(n-k)b(k) \tag{1.34}$$

where $a(k)$ and $b(k)$ are weight factors for the inputs and past outputs. For ease of notation, the samples are represented using suffixes wherever convenient. In terms of this notation, (1.34) can be written as

$$y_n = \sum_{k=0}^{M-1} x_{n-k}a_k + \sum_{k=1}^{N-1} y_{n-k}b_k \tag{1.35}$$

Since the present output depends on the past outputs, IIR filter is also called as a recursive filter. The Z transform of (1.34) can be used to arrive at different structures for implementation of the filter. For $M = N$ and $b(0) = 1$, the transfer function $H(k)$ of the filter given by (1.34) is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{A(z)}{B(z)} \tag{1.36}$$

$$H(z) = \frac{a_0 + a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + \;\ldots\; a_{N-2}z^{-N+2} + a_{N-1}z^{-N+1}}{1 + b_1z^{-1} + b_2z^{-2} + b_3z^{-3} + \;\ldots\; b_{N-2}z^{-N+2} + b_{N-1}z^{-N+1}} \tag{1.37}$$

where $A(z)$, $B(z)$ are the z transform of the weights $a(k)$ and $b(k)$. Structure for implementation of the filter by given (1.34) – (1.36) is shown in Fig. 1.10. It is called as the direct implementation structure. It can be verified that the top set of adders and delay units with transfer function z^{-1} realises the function $A(k)$ and the bottom units realise the function $1/B(z)$. Transpose of this filter is shown in Fig. 1.11. Both of these filter structures require $2(N - 1)$ delay units, $2(N - 1)$ adders and $2N - 1$ multipliers. The number of delay units can be reduced by realising $1/B(z)$ first and feeding the output of this filter to the filter with transfer function $A(z)$ as shown in Fig. 1.12. Since the delay units in both of these filters have the same

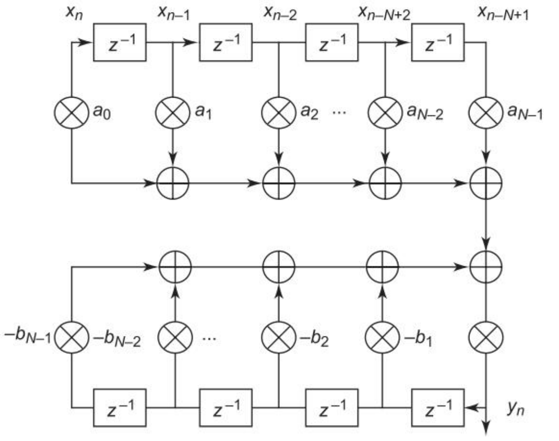


Fig. 1.10 *Direct implementation structure for an IIR filter*

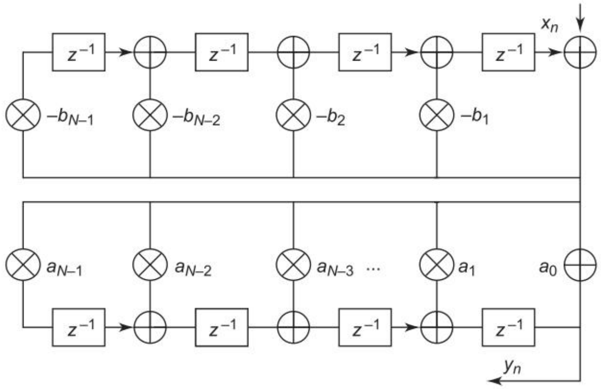


Fig. 1.11 *Direct implementation scheme for the IIR filter using the transpose structure*

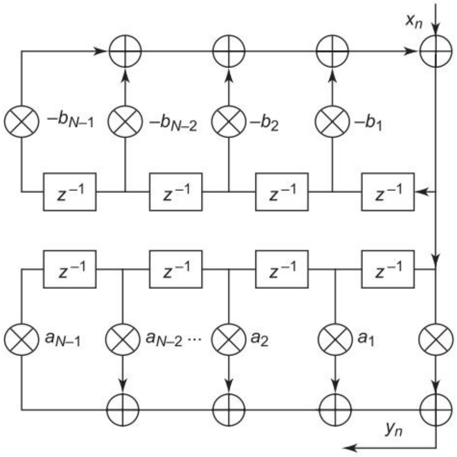


Fig. 1.12 *An alternate implementation scheme for the IIR filter*

inputs, a single delay unit can be shared by both of the filters with the transfer function $1/B(z)$ and $A(z)$. The resulting filter structure is given in Fig. 1.13. A digital filter is said to be canonic if the no. of delay units is equal to the order of the transfer function. Since in the above case, the order of the transfer function is $N - 1$ and the number of delay units used in Fig. 1.13, the filter given by Fig. 1.13 represents a canonic realization. Several other realizations can be obtained by factorizing $H(z)$ or writing it as a sum of a number of transfer functions.

Accordingly a cascade realisation and parallel realisation of the filters are obtained.

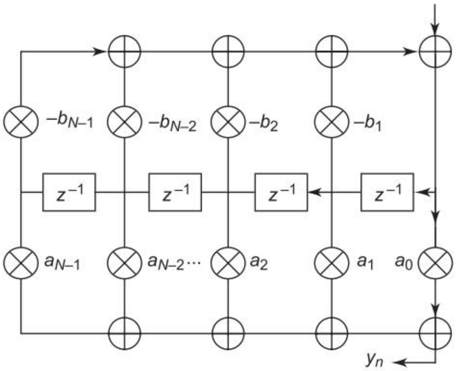


Fig. 1.13 *Canonic realisation scheme for the IIR filter*

As mentioned earlier an IIR requires less hardware and computational effort compared to FIR filter for the same sharpness and flatness in the filter. However, these filters may not always be realisable and may become unstable and are affected more by the finite word length effects. When an IIR scheme exists for an FIR filter, then it requires less memory and less computation time compared to the latter. For example consider an FIR filter with M coefficients in which the n^{th} coefficient is given by

$$h(n) = a^n \tag{1.38}$$

The transfer function of this filter can be written as

$$H(z) = 1 + (az^{-1}) + (az^{-1})^2 + (az^{-1})^{(M-1)} \tag{1.39a}$$

$$= \frac{1-(az^{-1})^M}{1-az^{-1}} \tag{1.39b}$$

(1.39a) represents an FIR filter and it requires $M - 1$ delay units, $M - 1$ adders as well as multipliers. (1.39b) represents an equivalent IIR filter. This requires $M + 1$ delay units, two adders and two multipliers.

1.15.5 Design of IIR Filters

Design of analog filters for LP. HP, BP, and BS has been studied in detail in the past and has been implemented in a large number of communication and consumer electronic circuits. The impulse response of the IIR filters can be obtained from the corresponding analog circuits commonly using one of the techniques impulse invariance method and the bilinear transform method. These two techniques are discussed in brief next.

1.15.6 Impulse Invariance Method

In this method, a digital IIR filter with impulse response $h(n)$ is obtained from the analog filter with impulse response $h_a(t)$ by evaluating $h(n)$ as

$$h(n) = h_a(t) \text{ for } t = nTs\ ;\ N = 0,1,2 \tag{1.40a}$$

where Ts is the sampling interval. However this method is applicable only if the analog filter response is bandlimited such that the frequency response $H(j\omega)$ is given by

$$H(j\omega) = 0 \text{ for } |\,\omega\,| > 1/2Ts \tag{1.40b}$$

If this condition is not satisfied, aliasing occurs; i.e., the signal with frequency f, in the frequency range $1/2Ts$ to $1/Ts$ gets mapped to $(f - 1/2Ts)$. Because of periodic nature of the frequency response of the LTI digital filters, for $m = 1, 2, \ldots$, all the signals in the frequency range $(m/Ts, (2m + 1)/2Ts)$ gets mapped to the frequency range $(0, 1/2Ts)$ in the digital filters. The signal with frequency f in the range $((2m + 1)/2Ts, (2m + 2)/2Ts)$ appears as $f - 1/2Ts$ due to aliasing. Hence eventhough this method is simple it is not applicable for filters whose response is not bandlimited. For example, this method is not suitable for the design of high pass filters. In practice, this method can be adopted if

$$H(j\omega) < 0.01\, H_{\max} \text{ for } |\,\omega\,| > 1/2Ts \tag{1.40c}$$

where $H_{\max}$ denotes the maximum amplitude of the frequency response of the analog filter in the frequency range $(0, 1/2Ts)$.

1.15.7 Bilinear Transform Method

In this method the transfer function $H(z)$ of the digital filter is obtained from the corresponding analog filter transfer function $H(s)$ by using the mapping given by

$H(z) = H(s)$ with

$$s = \frac{2}{Ts}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{1.41}$$

The relationship between the digital frequency ω and the analog frequency Ω can be obtained by substituting $s = j\Omega$ and $z = e^{j\omega Ts}$ in (1.41) as follows:

$$j\Omega = \frac{2}{Ts}\left(\frac{1 - e^{-j\omega Ts}}{1 + e^{-j\omega Ts}}\right) \tag{1.42}$$

$$\Omega = (2/Ts) \tan{(\omega Ts/2)} \tag{1.42a}$$

In this case the upper half of the imaginary axis (or in other words the frequency range $(0, \propto)$ is mapped uniquely within the unit circle corresponding to the z transform. Hence, aliasing does not occur in this case. However, this method introduces distortion in the frequency response of the digital filter designed. Specifically, the phase response of this filter is distorted and hence a filter with linear phase characteristics cannot be obtained using the bilinear transform method.

This is because the bilinear transform results in nonlinear mapping between the the digital frequency ω and the analog frequency Ω as given by (1.42a). This is called as frequency warping.

To overcome this nonlinearity, compensation has to be applied to the filter. One of the technique adopted is to prewarp the cutoff frequencies of the filter.This is called as prewarping. For example, if a digital filter with the cut off frequencies in the pass band as ω_1, ω_2, ω_3, ω_4 is desired, an analog filter with cut off frequencies given by

$$\Omega_i = (2/Ts)\tan(\omega_i\, Ts/2) \text{ for } i = 1, 2, 3,4 \tag{1.42b}$$

is designed. After the warping given by (1.42a), a digital filter with the desired cut off frequencies is obtained. The prewarping can only compensate for the amplitude nonlinearities and the phase nonlinearities cannot be overcome.

FINITE WORD LENGTH EFFECT IN DIGITAL FILTERS 1.16

Due to finite size of the registers used for storing the output of the processing elements like adders and multipliers, overflow can occur in FIR filters if the results exceed the capacity of the registers. These results will normally be rounded off and this gives rise to round off noise. Overflow can be avoided by using floating point arithmetic. However, in filters using fixed point computations, the overflow can be minimized by scaling the impulse response coefficients suitably.

In the case of IIR filters, because of the feed back connection, finite precision arithmetic can lead to oscillations which make the output to *k*eep alternating between two values. Such oscillations are called as limit cycle oscillations. This can occur when the input to the filter is zero for example during silence periods in voice communication. In this case the output *k*eeps oscillating between two small values and is referred to as zero input limit cycles. Another type of oscillation called as overflow limit cycle oscillation occurs when overflow occurs at the output due to the finite precision arithmetic. In this case output can *k*eep oscillating with amplitudes equal to the full supply voltage. Another effect of finite word length is the quantization noise. It can arise due to inadequate no. of bits for representing the input to the filter, output of the filter or the output of the processing elements like multipliers and adders.

POWER SPECTRUM ESTIMATION 1.17

Power spectrum of a stationary signal may be obtained by considering a block of N samples and computing the FFT of these samples. Power spectrum or periodogram is defined as

$$P(k) = \frac{1}{N}\left|X(k)\right|^2 = \frac{1}{N}X(k)X^*(k) \tag{1.43a}$$

where $X(k)$ is the FFT of the input sequence. When the length of the data sequence is large, the power spectrum is estimated by segmenting the data sequence into a number of overlapping segments. Figure 1.14 shows the most commonly used method for the power spectrum estimation using the Welch method. In this method, the averaged periodogram is obtained by segmenting the input into blocks of N samples and multiplying them with a bell shaped window $w(n)$ to reduce spectral leakage. The accuracy of the averaged spectrogram can be increased by increasing the number of samples, M, overlapping between consecutive blocks. Typically M is chosen to be $N/2$. K, the total number of segments of data sequence of length L is given by

$$K = \operatorname{int}\left(\frac{L-N}{M}+1\right) \tag{1.43b}$$

where int(.) is the integer operator. For each of the windowed segment, N point FFT is found and the Power spectrum corresponding to the i^{th} segment is denoted as $P_i(k)$ and is given by

$$P_i(k) = \frac{1}{NP_w}\left|X_i(k)\right|^2 \tag{1.43c}$$

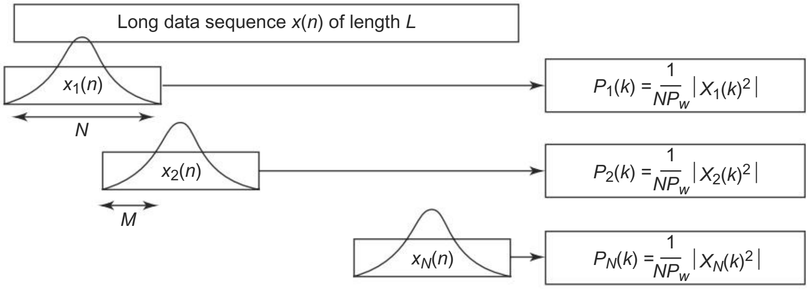

Long data sequence $x(n)$ of length L
$x_1(n)$
N
$x_2(n)$
M
$x_N(n)$
$P_1(k) = \frac{1}{NP_w}|X_1(k)^2|$
$P_2(k) = \frac{1}{NP_w}|X_2(k)^2|$
$P_N(k) = \frac{1}{NP_w}|X_N(k)^2|$

Fig. 1.14 *Computation of power spectrum using Welch algorithm*

This includes the contribution due to the window function. Power of the window function is denoted as P_w and is given by

$$P_w(k) = \frac{1}{N}\sum_{n=0}^{N-1} w^2(n) \tag{1.43d}$$

The averaged power spectrum of the entire data sequence is denoted as $P(k)$ and is given by

$$P(k) = \frac{1}{NKP_w}\sum_{i=1}^{K}\left|X_i(k)\right|^2 \quad k = 0,1,2,\ \dots N-1 \tag{1.43e}$$

Compared to the periodogram obtained using a single window of length L, the computation of periodogram using Welch method (given by (1.43e)) results in more accurate and smoother estimate of power spectrum. Size of the window N has to be chosen as a compromise between accuracy and frequency resolution: Smaller N results in poor frequency resolution but better accuracy; the reverse is true for larger N.

SHORT TIME FOURIER TRANSFORM 1.18

Many real world signals such as speech, radar and sonar signals are non stationary in nature. Their frequency, phase and peak amplitude may vary with time. For example, the chirp signal, also referred to as linear frequency signal (LFM), given by

$$x[n] = A\cos(w_0 n^2) \tag{1.44a}$$

is used in continuous wave radar to determine the range of a target. In this signal, the frequency of the signal linearly increases over a time period. Similarly, the frequency components in the speech signal varies from time to time. In order to evaluate the frequency content of a signal as a function of time, short time fourier transform (STFT) may be used.The STFT of a sequence $x[n]$ is defined by

$$X_{\text{STFT}}(e^{j\omega}, n) = \sum_{m=-\infty}^{m=\infty} x[n-m]w[m]e^{-j\omega m} \tag{1.44b}$$

$w[m]$ is a window sequence. The width of window is chosen so that the signal may be considered to be stationary in that interval. When $w[n]$ is rectangular window, the resulting transform is called as the discrete time fourier transform (DTFT).

In practice, the window sequence of finite length R is chosen. Moreover, the STFT is evaluated at N ($N \geq R$) equi-distant points along the unit circle $\frac{2\pi}{N}k$. The sampled values of the STFT is denoted as

$$X_{\text{STFT}}(k, n) = X_{\text{STFT}}(e^{j\omega}, n)\Big|_{\omega=\frac{2\pi}{N}k} = X_{\text{STFT}}\left(e^{j\frac{2\pi}{N}k}, n\right) \tag{1.44c}$$

$$= \sum_{m=0}^{m=R-1} x[n-m]w[m]e^{-j\frac{2\pi}{N}\text{km}} \qquad 0 \leq k \leq N-1 \tag{1.44d}$$

From (1.44d), it may be noted that $X_{\text{STFT}}[k,n]$ may be obtained by finding DFT of $x[n–m]w[n]$. $X_{\text{STFT}}[k,n]$ is periodic with respect to k with period N. Assuming $w[m]$ to be non zero, $x[n–m]$ can be obtained by computing the inverse DFT of $X_{\text{STFT}}[k,n]$ and is given by

$$x[n-m] = \frac{1}{w[m]N}\sum_{k=0}^{N-1} X_{\text{STFT}}[k,n]e^{j\frac{2\pi}{N}km} \qquad 0 \leq m \leq R-1 \tag{1.44e}$$

As $X_{\text{STFT}}[k,n]$ is a function of 2 variables, it requires a three dimensional plot. In practice, a two dimensional plot known as spectrogram is used to plot $X_{\text{STFT}}[k, n]$. In this plot, the X and Y axis denote the time sample index and frequency sample index respectively. The magnitude of $X_{\text{STFT}}[k, n]$ is indicated by the darkness of the point. Higher the darkness, higher is the amplitude of $X_{\text{STFT}}[k,n]$.

The characteristics of the window function determine the time resolution ($\Delta\tau$) and frequency resolution($\Delta\omega$). If R is small, the window duration is small and STFT has very good time resolution. Two signals of the same frequency but occurring one after another with a delay of $\Delta\tau$ can be perceived as two different signals in the STFT plot. However, small R implies small N and the frequency resolution $\frac{2\pi}{N}$ becomes poor. On the other hand, large window duration results in better frequency resolution but poor time resolution.

MULTIRATE SIGNAL PROCESSING 1.19

Multirate signal processing refers to the techniques adopted for processing a digital signal by sampling it further either using a higher rate than the rate at which the digital samples were obtained or using a lower sampling rate. The sampler used for this purpose is accordingly called as upsampler, downsampler respectively. The down sampler is also referred to as a decimator and a decimator which downsamples the input by a factor of M is denoted by the symbol given in Fig. 1.15.a. The upsampler is also referred to as interpolator and an interpolator which increases the sampling rate by a factor of M is denoted by the symbol given in Fig. 1.15.b. If $x_d(n)$ denotes the output of an M factor decimator of an input signal $x(n)$, then $x_d(n)$ can be written as

$$x_d(n) = x(Mn) \text{ for } N = 1, 2, 3, \ldots \quad (1.45)$$

Similarly if $y_u(n)$ denote the output of an M factor interpolator of a signal $y(n)$, $y_u(n)$ is given by

$$y_u(n) = y(n/M) \text{ for } n = 0, M, 2M, 3M \ldots \quad (1.46)$$
$$= 0 \text{ otherwise}$$

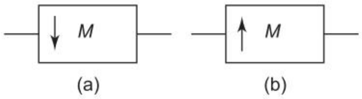


Fig. 1.15 *(a) Downsampler; (b) Upsampler*

1.19.1 Frequency Domain Characterization of the Decimator and Interpolator

The frequency domain characterization of the decimator and interpolator can be obtained using the z transform of $x_d(n)$ and $y_u(n)$ denoted as $X_d(k)$ and $Y_u(k)$ respectively.

$$X_d(z) = \sum_{n=-\infty}^{\infty} x_d(n)z^{-n} = \sum_{n=-\infty}^{\infty} x(Mn)z^{-n} \quad (1.47a)$$

$$= x(0) + x(M)z^{-1} + x(2M)z^{-2} + x(3M)z^{-3} + \cdots \quad (1.47b)$$

Let us define an intermediate sequence $x_a(n)$ as follows:

$$x_a(n) = x(n) \text{ for } n = kM;\ k = 0, 1, 2, \ldots. \quad (1.47c)$$
$$= 0 \text{ otherwise}$$

$x_a(n)$ can be rewritten as

$$x_a(n) = x(n)c(n) \quad (1.47d)$$

where c(n) is a periodic sequence with period M and is defined as:

$$c(n) = [1, 0, 0, 0, \ldots 0] \quad (1.47e)$$

$C(k)$, the DFT of c(n) is given by

$$C(k) = 1 \text{ for } k = 0, 1, 2, \ldots. M–1 \quad (1.47f)$$

Computing the inverse DFT of $C(k)$ and denoting $e^{-j\frac{2\pi}{M}}$ by the symbol W_M we get

$$c(n) = \frac{1}{M}\sum_{k=0}^{M-1} W_M^{-kn} \quad (1.47g)$$

Substituting eqn (1.47g) in (1.47d), $X_a(k)$ can be obtained as

$$X_a(z) = \sum_{n=0}^{\infty} x(n)z^{-n} \frac{1}{M}\sum_{k=0}^{M-1} W_M^{-kn} \quad (1.47h)$$

$$= \frac{1}{M}\sum_{k=0}^{M-1} \sum_{n=0}^{\infty} W_M^{-kn} x(n)z^{-n} \quad (1.47i)$$

$$= \frac{1}{M}\sum_{k=0}^{M-1} \sum_{n=0}^{\infty} x(n)(W_M^k z)^{-n} \quad (1.47j)$$

$$= \frac{1}{M}\sum_{k=0}^{M-1} X(W_M^k z) \quad (1.47k)$$

Using (1.47c) in (1.47b), $X_d(k)$ can be rewritten as

$$X_d(z) = x_a(0) + x_a(M)z^{-1} + x_a(2M)z^{-2} + x_a(3M)z^{-3} + \cdots \tag{1.47l}$$

For $M = 2$, $X_d(k)$ is given by

$$X_d(z) = x_a(0) + x_a(2)z^{-1} + x_a(4)z^{-2} + x_a(6)z^{-3} + \cdots \tag{1.47m}$$

Since $x_a(1)$, $x_a(3)$, $x_a(5)$ ... are zero, Eqn. (1.47m) can be rewritten as

$$\begin{aligned} X_d(z) &= x_a(0) + x_a(1)z^{-\frac{1}{2}} + x_a(2)z^{-\frac{2}{2}} + x_a(3)z^{-\frac{3}{2}} + x_a(4)z^{-\frac{4}{2}} \cdots \\ &= X_a\left(z^{\frac{1}{2}}\right) = X_a\left(z^{\frac{1}{M}}\right) \end{aligned} \tag{1.47n}$$

Using (1.47k) in (1.47n), we get

$$X_d(z) = \frac{1}{M}\sum\nolimits_{k=0}^{M-1} X\left(W^k z^{\frac{1}{M}}\right) \tag{1.47o}$$

Similarly, $Y_u(k)$ can be written as follows:

$$\begin{aligned} Y_u(z) &= \sum_{n=-\infty,}^{\infty} y_u(n)z^{-n} = \sum_{\substack{n=-\infty, \\ n=kM\ k\ =0,1,2,\ \ldots}}^{\infty} y\left(\frac{n}{M}\right)z^{-n} \\ &= y(0) + y(1)z^{-M} + y(2)z^{-2M} + y(3)z^{-3M} + \cdots \\ &= Y\left(z^M\right) \end{aligned} \tag{1.48}$$

1.19.2 Spectrum at the Output of the Downsampler and Upsampler

More insight into the downsampling and upsampling process can be gained by looking at the spectrum of the sampled signal and comparing it with the spectrum of the original signal.

Frequency spectrum at the output of the decimator and interpolator can be obtained by substituting $z = e^{-j\omega}$ in (1.47o) and (1.48). For $M = 2$, the frequency response of the decimator and interpolator are given by (1.49) and (1.50).

$$X_d(e^{-j\omega}) = \frac{1}{2}[X(e^{-j\omega/2}) + X(e^{j\omega/2})] \tag{1.49}$$

$$Y_u(z) = Y(e^{-j2\omega}) \tag{1.50}$$

Let $H(\omega)$, denote the frequency spectrum of a signal bandlimited to f_m. $H_s(\omega)$ denotes the spectrum at the output of sampler which samples the analog input at the rate of fs samples/sec. $X_d(\omega)$ denotes the frequency spectrum at the output of the downsampler which samples the output of the above sampler at the rate of f_d (i.e., f_s/M) samples/sec. $Y_u(\omega)$ denotes the frequency spectrum at the output of the upsampler which samples the output of the above sampler at the rate of Mf_s samples/sec.

For the case, where the input signal is assumed to be a low pass signal bandlimited to $\omega_m = 0.4\pi$ radians and $M = 2$, the spectrum of $H(\omega)$, $H_s(\omega)$ and $H_d(\omega)$ and $H_u(\omega)$ are shown in Fig. 1.16(a), Fig. 1.16(b), Fig. 1.16(c). and Fig. 1.16(e) respectively.

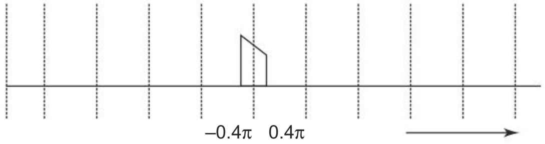


Fig. 1.16(a) *Spectrum of the input signal*

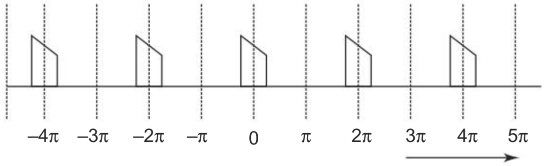


Fig. 1.16(b) *Spectrum of the sampled signal ω*

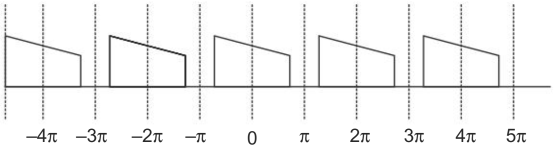


Fig. 1.16(c) *Spectrum at the decimator output for M =2 and ω_M =0.4π*

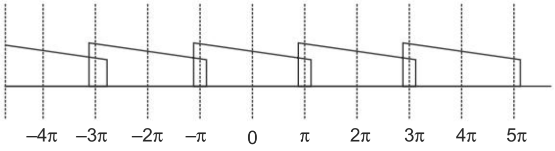


Fig. 1.16(d) *Spectrum at the decimator output for M=2 and ω_M = 0.6π*

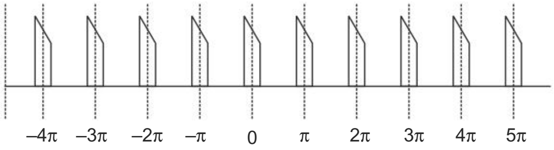


Fig. 1.16(e) *Spectrum at the interpolator output for M =2 and ω_M = 0.4π*

In Fig. 1.16(b). the frequency spectrum given in Fig. 1.16(a). gets replicated at intervals of 2π radians. Fig. 1.16(c) is obtained by stretching the frequency spectrum of the sampled signal by a factor of two along the ω axis and replicating it at intervals of π radians. The magnitude of the spectrum is scaled by a factor of 2. Spectrum at the output of the interpolator is shown in Fig. 1.16(e). This is obtained by compressing the frequency spectrum of the sampled signal by a factor of two along the ω axis and replicating it at intervals of π radians. It may be noted that a signal which is sampled and decimated by 2 can be recovered by interpolating it by 2 and passing it through a low pass filter with cutoff frequency of $\omega_m = 0.4\pi$. Similarly, a signal which is sampled and upsampled by 2 can be recovered by decimating it by 2 and passing it through a low pass filter with cutoff frequency of $\omega_m = 0.4\pi$.

It can be verified from Fig. 1.16(c) that the band of signals centred around 0 radians and its replicas at intervals of π radians will be non overlapping only if $\omega_m \leq 0.5\pi$ radians. Fig. 1.16(d) shows the spectrum at the output of the decimator when $\omega_m = 0.6\pi$ radians. In this case, the replicas at intervals of π overlap and aliasing is said to occur. In this case, the original signal cannot be faithfully recovered from the decimated signal by passing it through an interpolator and a low pass filter.

In general, in order to avoid the aliasing, the input to the decimator should be bandlimited to π/M. A low pass filter, also referred to as antialiasing filter, with cut off frequency $\omega_c = \pi/M$ is used for this purpose.

1.19.3 Subband Coding

Processing a signal after downsampling or upsampling has a no. of advantages. To appreciate it let us consider the example of a voice coder/decoder. When an analog signal is digitized using the sampler followed by the quantizer, the minimum sampling rate required is determined by the maximum frequency component in the signal and the no. of bits used for quantisation is independent of the relative importance of the various frequency components that constitute the signal. Nonuniform quantizers allocate different step sizes for different amplitude ranges of the signal taking into account their probability of occurrence. Low amplitude signals occur more frequently and are more important. They are allocated small step sizes. Large amplitude signals are quantized with large step size.

Quantisation noise and the bit rate required for digitizing an analog signal can be reduced further by examining their frequency content and allocating the no. of bits per sample depending upon their importance. For example, the energy in the speech signal is concentrated more in the low frequency region of (0–1 kHz) and is of decreasing concentration as the frequency is increased. Hence, more number of bits may be allocated in frequency bands where there is more energy concentration and less number of bits may be allocated where there is less concentration. This technique is called as the subband coding. The first step required for subband coding is to separate the incoming signal into separate bands using low pass, band pass and high pass filters. This may be done in the digital domain; i.e., the analog signal may first be sampled and quantized without any regard to the frequency. The resulting digital signal may then be passed through the digital filters to separate the different band of frequencies. The individual band of frequencies may be individually processed further. However they need not be processed at the rate at which the analog signal was sampled. This can be be verified using the sampling theorem for the bandpass signal.

1.19.4 Sampling Theorem for Bandpass Signals and its Application

Bandpass Sampling Theorem A band pass signal can be faithfully reproduced from its samples if it is sampled at the rate $fs = 2(f_1–f_2) = 2B$ if either f_1 or f_2 is a harmonic of $2B$. Here, f_1 and f_2 denote the lowest and highest frequency components of the signal respectively. B denotes the bandwidth. If neither f_1 nor f_2 is a harmonic of $2B$, then fs is chosen to be greater than $2B$ such that this condition is satisfied. In general, the range of fs required for a bandpass signal is $2B \leq fs \leq 4B$

In the above subband encoder, if one of the subband is of bandwidth 0.5 kHz, it need not be sampled at 8 kilosamples/s(KSPS) for further processing but a sampling rate of 1KSPS may be just adequate. In practice, one may choose 2 KSPS as the sampling rate. When the voice signal is sampled at 8 KSPS rate, at the output of the filter corresponding to a sub band, the samples depart at 8 KSPS rate, but all of these samples are not required for processing. In the above case only 1/4th of the samples may be retained and the rest of them may be discarded. This is achieved using a down sampler.

At the receiver, the decoder has to combine the outputs from the individual sub bands and deliver a single contiguous band. Even though the individual bands generate samples at a rate lower than 8 KSPS the signal has to be delivered at the 8 KSPS rate to the play out device. This is achieved by sampling the output of the sub band at a rate higher than the rate at which the sample arrive at its input. This is achieved by inserting zeros between the actual sample values. This process is achieved by the upsampler.

1.19.5 Block Diagram of a Subband Processing System and Filter Banks

The block diagram of a subband processing system which processes M subbands individually using decimation by M and then upsampling by M is shown in Fig. 1.17. The filters $H_1(k)$, $H_2(k)$ … $H_M(k)$ denote the filters used for separating the M bands and are called as analysis filter banks. When the bandwidth of each of the filter band is the same, each of them can be decimated by a factor of M or lower without causing aliasing. In this case the filter banks are called as the decimated filter banks. If the decimation factor is M, then they are called as maximally decimated filter banks and they result in the maximum computational efficiency. The filters $G_1(k)$, $G_2(k)$, … $G_M(k)$ are called as the synthesis filter banks. The analysis and synthesis filter banks shown in Fig. 1.18 require narrow bandpass filters. The design of the analysis filter banks can be simplified by using the tree structure given in Fig. 1.18. The corresponding tree structure for the synthesis filter bank is shown in Fig. 1.19. These structures use only the low pass and high pass filters. The cutoff frequency of the low pass filter is increased as the tree is traversed towards the right. In Fig. 1.18 $H_0(k)$, $H_1(k)$ denote the transfer function of the low pass, high pass filter respectively and it can be verified that

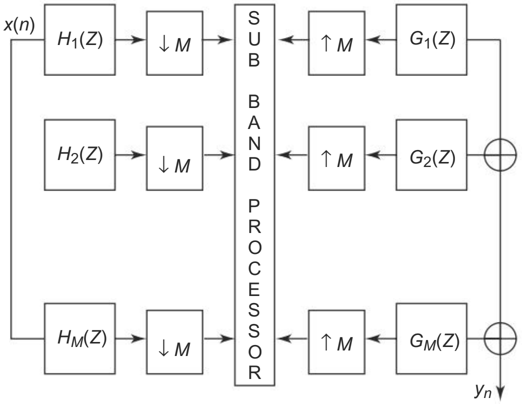


Fig. 1.17 *A subband processor with filter banks*

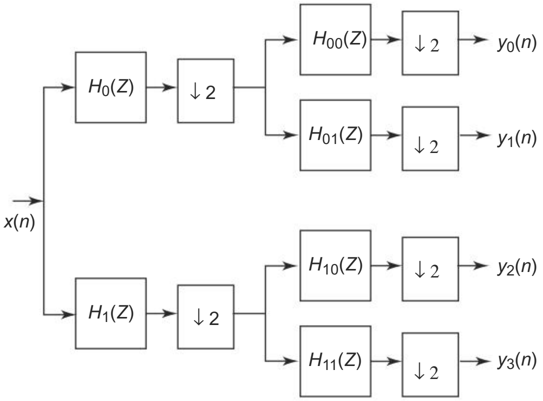


Fig. 1.18 *Tree structured analysis filter bank*

$$H_0(k) = H_1(-z) \tag{1.51}$$

The two channel filter bank which satisfies(1.51) is referred to as a quadrature mirror filter bank (QMF).

The tree structure is obtained by splitting every branch into two. However it may not always be required to split each branch into two for subband processing. For eg. in the subband coding scheme proposed by Crochiere[1983], the analog speech signal is sampled at 8 KSPS and split into 4 bands as shown in Fig. 1.19(a). Each of these bands has a bandwidth of 1 kHz. Next the band 0–1 kHz is split into 2 bands and a pruned tree with 5 bands: 0–0.5, 0.5–1, 1–2, 2–3 and 3–4 kHz is shown in Fig. 1.19(a)

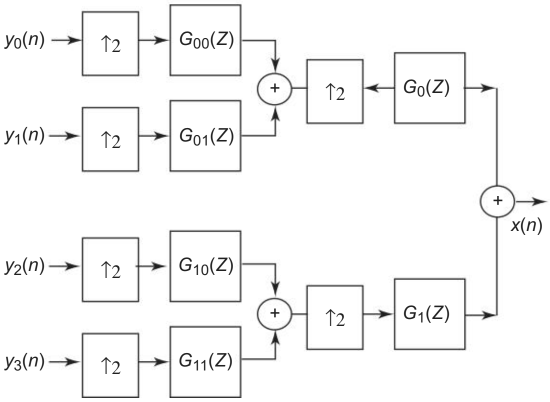


Fig. 1.19 *Tree structured synthesis filter bank*

and coded individually. Quantization of the first two bands with 5 bits/sample, the next two bands with 4 bits/sample and the last band with 3 bits/sample gives a bit rate of 32 kbps. Quantization of the first two bands with 4 bits/sample, the next two bands with 2 bits/sample and the last band with 0 bits/sample gives a bit rate of 16 kbps. With the 8 KSPS sampling 8 bits/samples, the bit rate required is 64 kbps. Hence the decimation/interpolation has facilitated the data compression by a factor of 2–4.

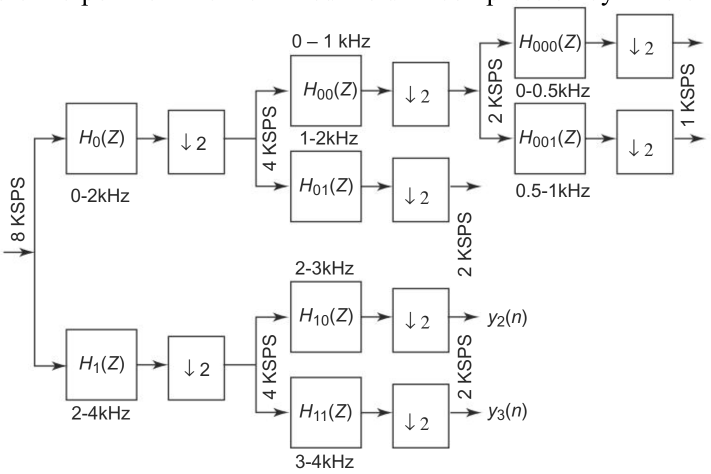


Fig. 1.19(a) *Subband encoder*

1.19.6 Reuse of Digital Filters for Different Subbands

One of the important characteristics of digital filters is that the cut off frequency of the filter cannot be uniquely determined from the filter architectural diagram and the impulse response response coefficients. For example, the cut off frequency of the M tap FIR filter shown in Fig. 1.4 cannot be uniquely determined if the impulse response coefficients h_0, h_1, …$h_{M\text{–}1}$ are specified. This is because, cut off frequency normalized by the sampling rate is used for the design of digital filter. Hence, a low pass filter with cut off frequency of 0.2π radians has the cut off frequency of 1 kHz if the sampling rate is 10 KSPS. The same filter has the cut off frequency of 10 kHz if the sampling rate is 100 KSPS. Hence, the same filter may have different pass band characteristics depending upon the rate at which inputs arrive and the registers (delay units in the filter) are clocked.

In Fig 1.19(a), the normalized cut off frequencies of $H_0(z)$, $H_{00}(z)$ and $H_{000}(z)$ are the same (0.5π). Hence, a single low pass filter with pass band $0 - 0.5\pi$ radians may be time shared or reused for all the three bands. Similarly, $H_1(z)$, $H_{01}(z)$ and $H_{001}(z)$ have the same normalized cut off frequency and have pass bands $0.5\pi - \pi$. Hence, a single filter may be time shared or reused for all the three bands.

Highest frequency band up to which a filter works satisfactorily depends on the filter architecture. The maximum sampling rate is determined by (1.28a) – (1.28c) depending on the filter type used.

1.19.7 Polyphase Filters

Another application of multirate signal processing is design of polyphase filter structures for speeding up the computation. Consider a LTI filter with N filter coefficients whose output is given by

$$y(n) = \sum_{K=0}^{N-1} x(n-k)h(k) \tag{1.52}$$

where $y(n)$, $x(n)$ and $h(n)$ are the n^{th} sample of output, input and the impulse response of the LTI causal discrete time system and $x(n)$ and $h(n)$ are assumed to be 0 for $N < 0$.

Let us find $y(n)$ for n= 70 and 71 for N =6. Using (1.52) we get

$$y(70) = x(70)h(0) + x(68)h(2) + x(66)h(4) + x(69)h(1) + x(67)h(3) + x(65)h(5)$$

$$y(71) = x(71)h(0) + x(69)h(2) + x(67)h(4) + x(70)h(1) + x(68)h(3) + x(66)h(5)$$

Hence (1.52) can be rewritten as

$$y(n) = \sum_{m=0}^{N/2-1} x(n-2m)h(2m) + \sum_{m=0}^{N/2-1} x(n-2m+1)h(2m+1) \tag{1.53}$$

Hence, $y(n)$ can be obtained using two subfilters $H_0(z^2)$, $H_1(z^2)$ as shown in Fig. 1.20. Internal architecture of these filters are shown in Fig. 1.20a and Fig.1.20b respectively. The first filter contains only the even indexed impulse response coefficients ($h(0)$, $h(2)$, $h(4)$). The second filter contains only the odd indexed impulse response coefficients. ($h(1)$, $h(3)$, $h(5)$). When N is even, the 1st filter processes the even samples and the 2nd filter processes the odd samples. The reverse happens when N is odd. Since each of them have $N/2$ taps and their computational complexity is reduced by a factor of two. In other words, the sub filters need to perform only $N/2$ multiplications and $N/2 - 1$ additions within each sampling interval instead of N multiplications and N–1 additions. When the sub filters are implemented in hardware, reduced computational complexity implies that slower and cheaper multipliers and adders can be used for the implementation. This technique can be extended further. Using M–1 delay units and M sub filters, the computational complexity of the individual filters can be reduced by a factor of M. The M individual filters are called as the polyphase filters.

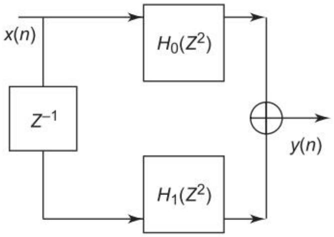


Fig. 1.20 *Linear filtering with 2 polyphase filters*

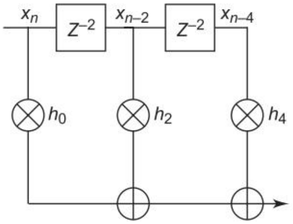


Fig. 1.20(a) *Subfilter with system function $H_0(z^2)$*

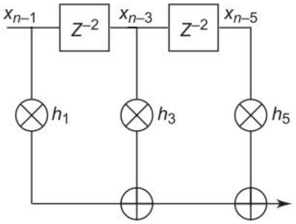


Fig. 1.20(b) *Subfilter with system function $H_l(z^2)$*

1.19.8 Sampling Rate Conversion

Another application of multirate signal processing is the conversion of an analog signal digitized at one sampling rate to another digital signal which is stored and retrieved at another sampling rate. For example CD quality voice is sampled, digitized and stored in the CD at the rate of 44.1 K samples/sec. When this signal is to be stored/read in/from a digital audio Tape (DAT), the samples have to be processed at the rate of 48 K samples/s. Hence for this application sampling rate conversion 44.1 /48 K samples is required. This is achieved by choosing an interpolator with a factor of 160 and a decimator with decimation factor of 147 as shown in Fig. 1.21. In Fig. 1.21, M=160 and N =147. Similar technique can be adopted for playing out a video signal digitized at one sampling rate on a system where the sampling rate is different.

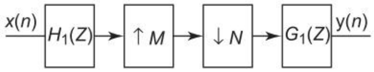


Fig. 1.21 *Sampling rate conversion for CD to DAT data*

1.19.9 Cascade Equivalence

In many multirate systems such as the subband encoder discussed in Section 1.19.6, filters and decimators are used in cascade. Similarly, interpolators and filters are used in cascade. Interpolator and decimator are used in cascade in sampling rate conversion systems. Cascade equivalences, also referred to as Noble identities, enable the order in which these operations are performed to be interchanged resulting in significant reduction on the computational complexity.

1. When M and N are relatively prime, the order in which the interpolation and decimation are performed can be changed without any change in the input output relationship. This is depicted in Fig. 1.22(a)
2. A filter represented in polyphase form in cascade with a decimator can be replaced by a decimator and a polyphase filter in cascade as shown in Fig. 1.22(b). Similarly, an interpolator followed by a filter in poly phase form has an equivalent cascade form as shown in Fig. 1.22(c).

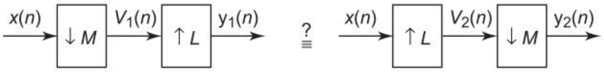


Fig. 1.22(a) *Cascade equivalence for interpolators and decimators*

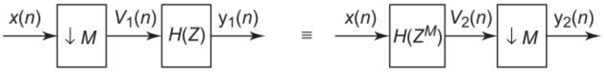


Fig. 1.22(b)

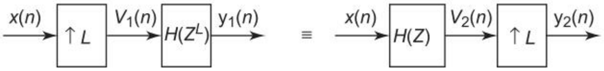


Fig. 1.22(c) *Cascade equivalence for filters and interpolators*

1.19.10 Efficient Implementation of Filters Used with Decimators and Interpolators

In a number of applications such as the subband encoder and computation of discrete wavelet transform(DWT), decimators are used in cascade with low pass and high pass filters. Cascade equivalence discussed in the previous section can be used for the efficient implementation of these filters. Let us consider the implementation of a filter $H(z)$ with N taps using a poly phase filter with M phases. The k^{th} sub filter has the transfer function $H_k(z^M)$. $H(z)$ can be expressed as

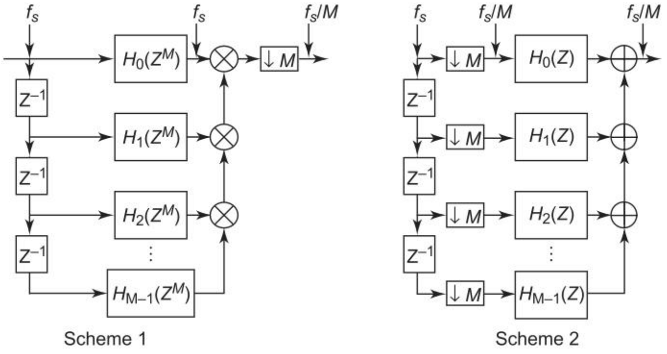
$$H(z) = \sum_{k=0}^{M-1} H_k(z^M) z^{-k} \tag{1.50a}$$

$$H_k(z) = h_k(0) + h_k(1)z^{-1} + h_k(2)z^{-2} + \cdots \tag{1.50b}$$

$$h_k(n) = h(Mn + k) \tag{1.50c}$$

Polyphase filter in cascade with the decimator can be realized using two schemes shown in Fig. 1.23(a). The second scheme is obtained by using the cascade equivalence discussed in the previous section. Let us assume that the sub filters have direct form I realization scheme. In the first scheme, each of the sub filters receive a new input at an interval of *Ts* (sampling rate *fs* = 1/*Ts*) and each sub filter needs to perform one multiplication and (*N*/*M*) – 1 additions within this interval. In the second scheme, each of the sub filters receive a new input at an interval of MTs and each sub filter needs to perform one multiplication and (*N*/*M*) – 1 additions within this interval. Hence, scheme 2 requires multipliers and adders whose speeds are M times smaller. This not only reduces the cost but also the power dissipation.

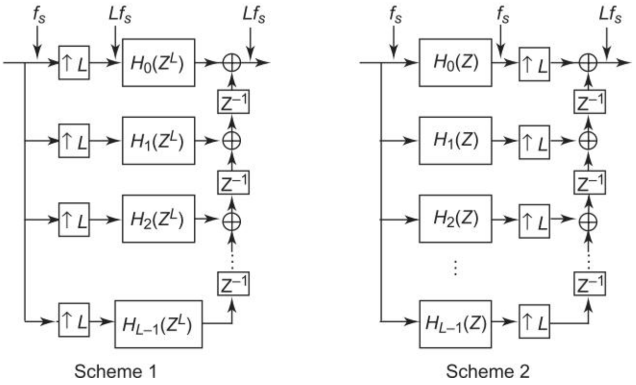
Similarly, interpolators in cascade with low pass/high pass filters are used in synthesis section of sub band coders and in the computation of inverse DWT. The interpolator and the filter using poly phase form have two equivalent implementation schemes as shown in Fig. 1.23(b). Scheme 2 requires slower and cheaper hardware and hence is to be preferred.



Scheme 1

Scheme 2

Fig. 1.23(a) *Efficient implementation of decimator filter*



Scheme 1

Scheme 2

Fig. 1.23(b) *Efficient implementation of interpolator filter*

1.19.11 Realisation of Time Division Multiplexer, Frequency Division Multiplexer and Transmultiplexer Using Mutirate Signal Processing

Fig. 1.24(a) shows the block diagram of a time division multiplexer. This is used in short haul communication. In this case, each of the input is digitized using an A/D converter before multiplexing. Fig. 1.24(b) gives the block diagram of the demultiplexer for the multiplexed data. For long haul communication, frequency division multiplexing is used. It may involve one or more analog exchanges. Fig. 1.25(a) shows the block diagram Frequency division multiplexer realized using interpolators and a bank of synthesis filters. It converts TDM signal into FDM signal. Fig. 1.25(b) shows the block diagram of Frequency division demultiplexer realized using decimators and a bank of analysis filters. It converts the FDM signal into TDM signal. The TDM-FDM-TDM converter is called as transmultiplexer. It takes digitized signal such as voice from M users and send them as FDM signal. $y(n)$ denotes the FDM output. The D/A converter at the output of the FDM signal enables this signal to be processed by analog exchanges. The analog FDM signal originating from the analog exchange is digitized using the A/D before conversion to TDM signal.

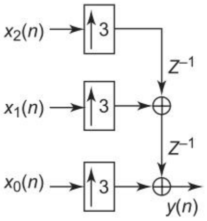


Fig. 1.24(a) *Block diagram of a time division multiplexer*

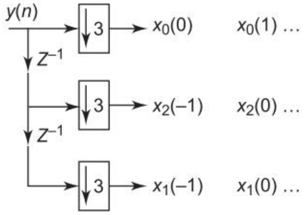


Fig. 1.24(b) *Block diagram of the TDM demultiplexer*

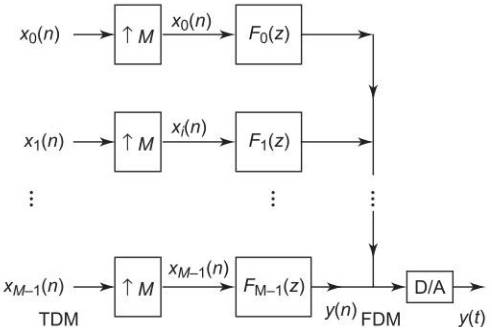


Fig. 1.25(a) *Block diagram of TDM- FDM converter*

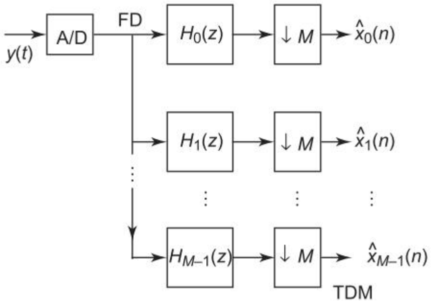


Fig.1.25(b) *Block diagram of FDM-TDM converter*

Let us consider the example of a transmultiplexer for $M = 4$, Fig. 1.26 shows the spectrum of four signals $x_0(n) - x_3(n)$. Fig. 1.27 shows the spectrum at the output of the interpolators. By choosing the synthesis filter $F_0(z) - F_3(z)$ to have equal bandwidth of $\pi/2$ and non overlapping passbands, the multiplexed output $y(n)$ whose spectrum is given in Fig. 1.28 can be obtained. The output of the transmultiplexer $\hat{x}_0(n) - \hat{x}_3(n)$ will be perfect reconstruction of $x_0(n) - x_3(n)$ excepting for a delay by n_0 samples if the analysis filters $H_0(z) - H_3(z)$ and synthesis filters $F_0(z) - F_3(z)$ satisfy the condition

$$F_k(z)\, G_\ell(z) = \delta_{k\ell}\, z^{-n0} \quad k = 0,1..3,\ \ell = 0,1 \ldots 3 \tag{1.51a}$$

where $\delta_{k\ell}$ is the kronecker delta.

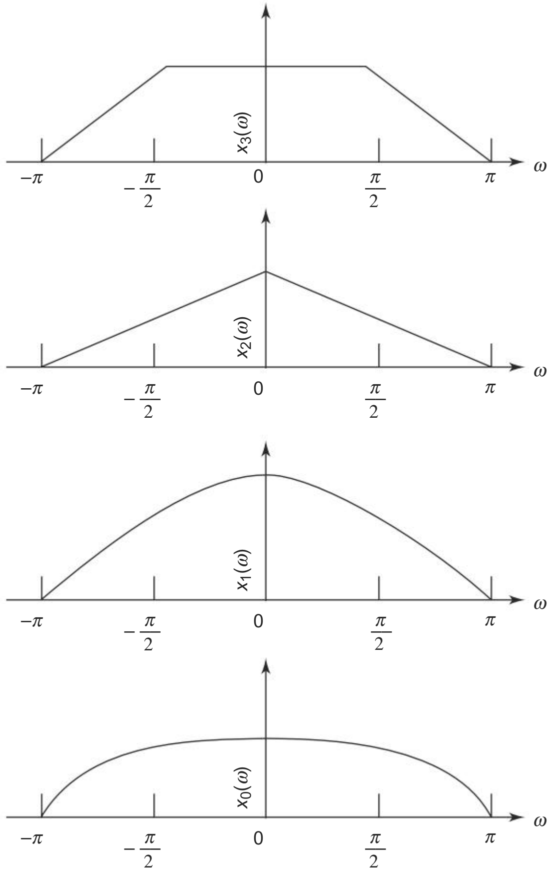


Fig. 1.26 *Spectrum of the input signals to be multiplexed*

1.19.12 Implementation of Orthogonal Frequency Division Multiplexing (OFDM) System

A simplified block diagram of an OFDM system is shown in Fig. 1.29. It uses M orthogonal carriers for transmitting M data streams. For this reason, it is called as discrete multitone transmission system.This differs from the conventional FDM in two aspects:

1. In the conventional FDM, the signal to be multiplexed originate from M different users. Each user is allocated one channel or band of frequencies. In the case of OFDM, the data originating from a single user is split into M substreams, each substream data is sent over one of the M channels.
2. In the conventional FDM, M distinct carriers which are amplitude modulated (single sideband modulation) by the signal from M users need not be harmonically related. M carriers used in the case of OFDM are integral multiple of a fundamental frequency

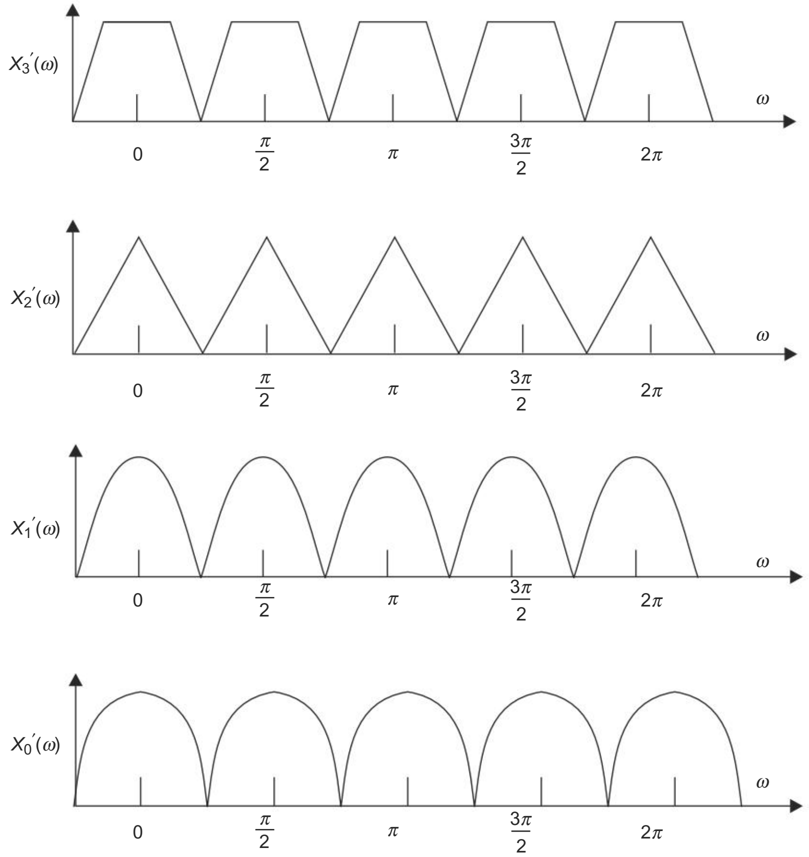


Fig. 1.27 *Spectrum at the output of the interpolators*

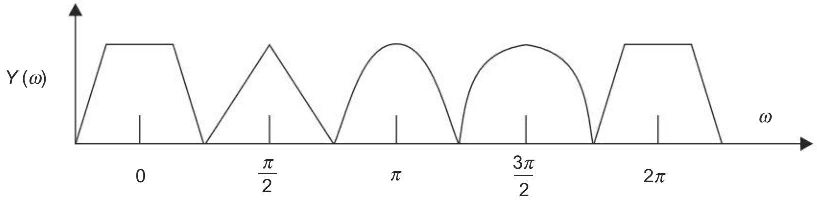


Fig. 1.28 *FDM output*

To understand the operation of the OFDM, let us consider 3 orthogonal carrier signals shown in Fig.1.30. They can be expressed as

$$s_k(t) = \cos(k\omega_0 t) \qquad \text{where } k = 1,2,3$$

By shifting these signals by 90° we can get

$$s_k(t) = \sin(k\omega_0 t)$$

Using these two sets of signals, complex carrier signals can be obtained as follows:

$$s_k(t) = \cos(k\omega_0 t) + j\sin(k\omega_0 t) = e^{+jk\omega_0 t} \tag{1.51b}$$

It may be verified that

$$\int_{0}^{T_0} s_k(t)s_l(t)dt = 0 \quad \text{for } k \neq l \tag{1.51c}$$

where $\omega_0 = \frac{2\pi}{T_0}$. These three orthogonal carriers may be modulated by three message signals $m_1(t) - m_3(t)$ individually and transmitted simultaneously by combining the modulated outputs as follows

$$y(t) = \sum\nolimits_{k=1}^{3} s_k(t)m_k(t) \tag{1.51d}$$

At the receiver, the message signal can be recovered without cross talk by correlating the received signal with the carrier corresponding to the required message signal as follows:

$$m_k(t) = \int_{0}^{T_0} y(t)s_k(t)dt \tag{1.51e}$$

In the discrete time implementation of OFDM, samples of M orthogonal carriers may be multiplied with M data streams, added and transmitted as shown in Fig. 1.29. In Fig. 1.29, $s_k(n)$ is given by

$$s_k(n) = s\left(M\left\lfloor \frac{n}{M} \right\rfloor + k\right) \tag{1.51f}$$

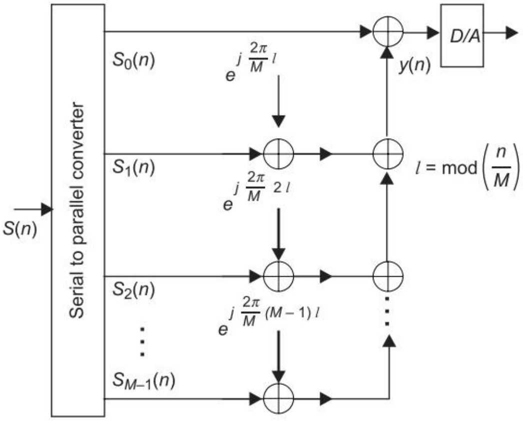

Serial to parallel converter
S(n)
S0(n)
S1(n)
S2(n)
SM–1(n)
$e^{j\frac{2\pi}{M}l}$
$e^{j\frac{2\pi}{M}2l}$
$e^{j\frac{2\pi}{M}(M-1)l}$
y(n)
D/A
$l = \text{mod}\left(\frac{n}{M}\right)$

Fig.1.29 *Block diagram of an OFDM system using orthogonal carriers*

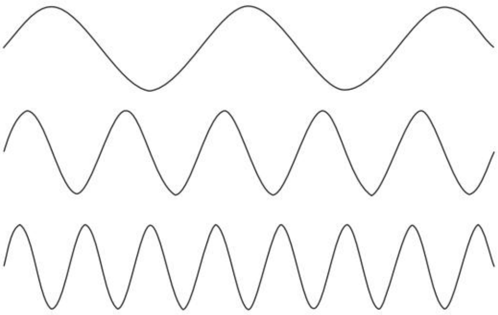


Fig. 1.30 *Orthogonal carriers used for OFDM*

However, OFDM output can be generated without using the multipliers and the samples of M orthogonal complex exponential carriers by using Inverse DFT. To understand this, let us compute $y(n)$ for $n = 0, 1, M–1$.

$$y(0) = s_0(0) + s_1(0) + s_2(0) + \ldots s_{M-1}(0) \tag{1.51g}$$

$$y(1) = s_0(1) + s_1(1)\, W_M^{-1} + s_2(1)\, W_M^{-2} + \cdots s_{M-1}(1) \tag{1.51h}$$

$$y(2) = s_0(2) + s_1(2)\, W_M^{-2} + s_2(2)\, W_M^{-4} + \cdots s_{M-1}(2) \tag{1.51l}$$

$$y(M-1) = s_0(M-1) + s_1(M-1)W_M^{-(M-1)} + s_2(M-1)\, W_M^{-2(M-1)} \ldots s_{M-1}(M-1)\, W_M^{-(M-1)^2} \tag{1.51m}$$

where $W_M^k = e^{-j\frac{2\pi k}{M}}$

Using (1.51f), it may be noted that

$$s_k(0) = s_k(1) = s_k(2) \ldots = s_k(M-1) = s(k) \quad \text{for } k = 0, 1, \ldots M-1$$

Using this (1.51g–1.51m) can be written in the matrix form as follows:

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ \cdots \\ y(M-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & W_M^{-1} & W_M^{-2} & W_M^{-3} & \cdots & W_M^{-(M-1)} \\ 1 & W_M^{-2} & W_M^{-4} & W_M^{-6} & \cdots & W_M^{-2(M-1)} \\ & \cdot\cdot & & & & \\ 1 & W_M^{-(M-1)} & W_M^{-2(M-1)} & W_M^{-3(M-1)} & \cdots & W_M^{-(M-1)^2} \end{bmatrix} \begin{bmatrix} s(0) \\ s(1) \\ s(2) \\ \cdots \\ s(M-1) \end{bmatrix} \tag{1.51n}$$

From (1.51n), it may be noted that $y(0) - y(M–1)$ can be obtained by finding the inverse DFT of the sequence $s(0)$–$s(M–1)$. The IDFT can be computed efficiently using the FFT algorithm.When the input to the IDFT block is real, the IDFT coefficients become complex valued. This requires two channels for transmitting the real and imaginary components. In order to overcome this limitation, the input to the IDFT block is formed as follows:

Let $a_k(t)$, $b_k(t)$ for $k = 1,2, \ldots M–1$ be two real valued data streams sampled at a rate of F_T that are to be transmitted. A new set of complex data stream, $\alpha_k(n)$ of length $N = 2M$ is formed as follows:

$$\alpha_k[n] = \begin{cases} 0 & k = 0, \\ a_k[n] + jb_k[n] & 1 \le k \le \dfrac{N}{2} - 1 \\ 0 & k = \dfrac{N}{2}, \\ a_{N-k}[n] - jb_{N-k}[n] & \dfrac{N}{2} + 1 \le k \le N - 1 \end{cases} \tag{1.51o}$$

Applying the inverse DFT to the above set of N sequences, we get the sequence $u_l(n)$.

$$u_\ell[n] = \frac{1}{N} \sum_{K=0}^{N-1} \alpha_k[n] W_N^{-\ell k}, \qquad \ell = 0,1,\ldots.,N-1 \tag{1.51p}$$

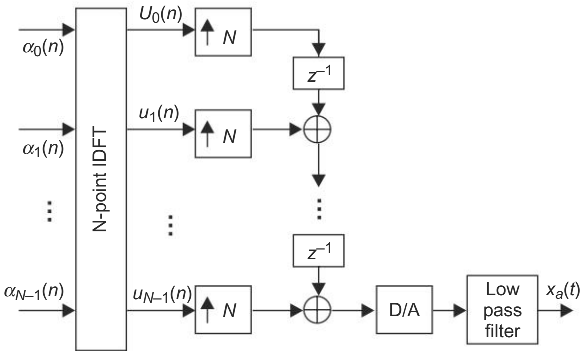


Fig. 1.31 *Block diagram of DMT transmitter using the IDFT block.*

The manner in which $\alpha_k(n)$ is formed ensures that $u_l(n)$ is real valued. The N transform coefficients $(u_l(n))$ may be converted to a serial stream using interpolators and delay elements and converted into analog signal using a D/A converter and a filter. The resulting analog signal $x_a(t)$ is transmitted over the channel. Fig. 1.31 shows the block diagram of the DMT transmitter using the above processes.

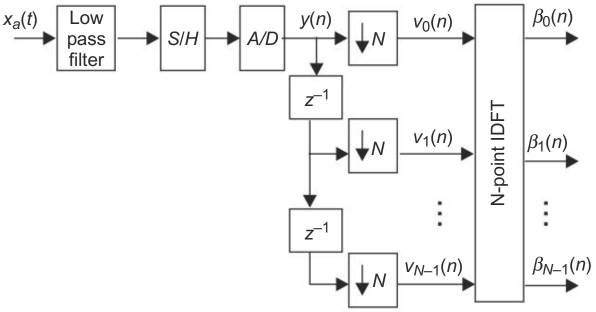


Fig. 1.32 *Block diagram of the DMT receiver*

Fig. 1.32 shows the block diagram of the DMT receiver. At the receiver, the received signal $y_a(t)$ is passed through an antialiasing filter and converted into digital data using a sample and hold circuit and an A/D converter. A sampling rate NF_T is used for both A/D, D/A and S&H converters. The data stream is converted to parallel form using the decimator and delay elements. The resulting N parallel streams $v_k(n)$ are applied to the DFT block and the DFT coefficients $\beta_k(n)$ are generated using the equation:

$$\beta_k[n] = \sum_{\ell=0}^{N-1} v_\ell[n] W_N^{-\ell k}, \qquad k = 0,1,. \, . \, ., N-1 \tag{1.51q}$$

If we assume the frequency response of the channel to have flat passband, A/D and D/A converters to be linear and assume the analog reconstruction and antialiasing filters to be ideal low pass filters, we can assume $y[n] = x[n]$. From the manner in which the interleaving and deinterleaving is performed in Fig. 1.31 and Fig. 1.32 respectively, the sequences at the receiver can be shown to be related to that of the transmitter as follows:

$$v_k(n) = u_{\text{k-1}}(n-1) \qquad k = 1,2, \ldots N-2 \tag{1.51r}$$

$$v_0(n) = u_{N-1}(n) \tag{1.51s}$$

$$\beta_{\text{k}}(n) = \alpha_{k-1}(n-1) \qquad k = 1,2, \ldots N-2 \tag{1.51t}$$

$$\beta_0(n) = \alpha_{N-1}(n) \tag{1.51u}$$

1.19.13 Applications of DMT

Digital Subscriber Loop (DSL) Modems The frequency response of the twisted pair used for carrying the signal from the telephone exchange to the subscriber premises is not flat if a large frequency band is considered.

Figure 1.33 shows the magnitude of the Frequency response of a band limited channel. If a high speed data is to be directly transmitted over the channel, a complex equalizer is required for perfect reconstruction. However, if the datastream is split into a number of substreams and DMT is used for transmission, then frequency band occupied by each of the subcarriers and sidebands after modulation by each substream occupies only a fraction of the entire bandwidth. If the subchannels are of smaller bandwidth as shown in Fig. 1.32, the frequency response can be assumed to be flat within this band. In this case, the equalizer required becomes merely a gain adjustment scheme for each of the substream. Hence, the attenuation of the channel for each of the substream data can be found by sending a test data. By providing sufficient amplification, imperfections in the

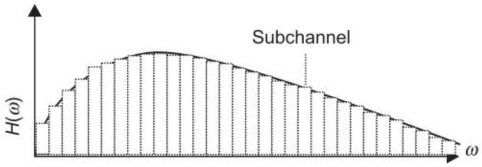


Fig. 1.33 *Frequency response of a band limited channel*

channel can be compensated. Moreover, if the signal strength is low in some of the sub channels, data rate can be reduced in those sub channels and error control coding can be adopted to reduce the errors. DSL modems use DMT scheme. The data transmission rate required from an internet server to the client at customer premises is much higher compared to that required from the client to the server. For this reason, the modem used in the digital subscriber loop is called as asymmetric digital subscriber loop (ADSL) modem. The data transmitted from server to client is called as downstream data and the data rate is 24Mbps in ADSL2. The data transmitted in the reverse direction is called as upstream data and the data rate is 1.3 Mbps in ADSL2.

Modems for Wireless LANs The advantages of using MCM holds good for the wireless channel as well. In addition to this, it can also remove ambiguity due to multipath propagation. For example, if the data transmission rate is 50 Mbps, bit time is 20 ns. If the propagation delay between two paths is of the order of 20ns, at the receiver, both n^{th} and $(n+1)^{th}$ bits will be received simultaneously through two paths and it may become difficult to demodulate the data. If the substream data rate is 1 Mbps, the difference in propagation delays will have negligible effect on the demodulation. The signal strength in the channel decreases with frequency and also with distance. In order to minimize the errors, error control coding may be used and in this case it is called as coded OFDM.

DISCRETE WAVELET TRANSFORM 1.20

Our eyes and ears can resolve low frequency signals better than the high frequency signals. Hence, we require better frequency resolution at low frequencies. The high frequency signals are normally of shorter duration and hence we require better time resolution at high frequency. In order to analyze both high and low frequency components of a signal properly, we require multi resolution capability or ability to provide different resolutions at different frequency bands. STFT does not have the multi resolution capability.

The continuous wavelet transform and discrete wavelet transform have multiresolution property and are extensively used for the analysis and classification of signals.

The continuous wavelet transform of a function $f(t)$ is defined as

$$WT(f(t),\tau,s)=\int_{-\infty}^{\infty} f(t)\psi\left(\frac{t-\tau}{s}\right)dt \tag{1.52a}$$

where $\psi\left(\frac{t-\tau}{s}\right)$ is called as a small wave or mother wavelet. τ and s are called as the translation and scaling parameter. Examples of typical wavelets used for analysis are shown in Fig. 1.34. Since wavelet is of finite duration, the parameter τ determines the time interval overwhich the mother wavelet is nonzero. The parameter s determines the width of the interval over which the wavelet is non zero. The discrete time version of CWT is called as the discrete wavelet transform. Every mother wavelet does not have a DWT representation. For a CWT to have a DWT, it should be realizable using a two channel filter bank as shown in Fig. 1.35 . Another equivalent condition for the existence of DWT is to test whether the scaling equation given by

$$\phi(t)=\sum_{n} g(n)\phi(2t-n) \tag{1.52b}$$

exists, where the actual wavelet is computed with

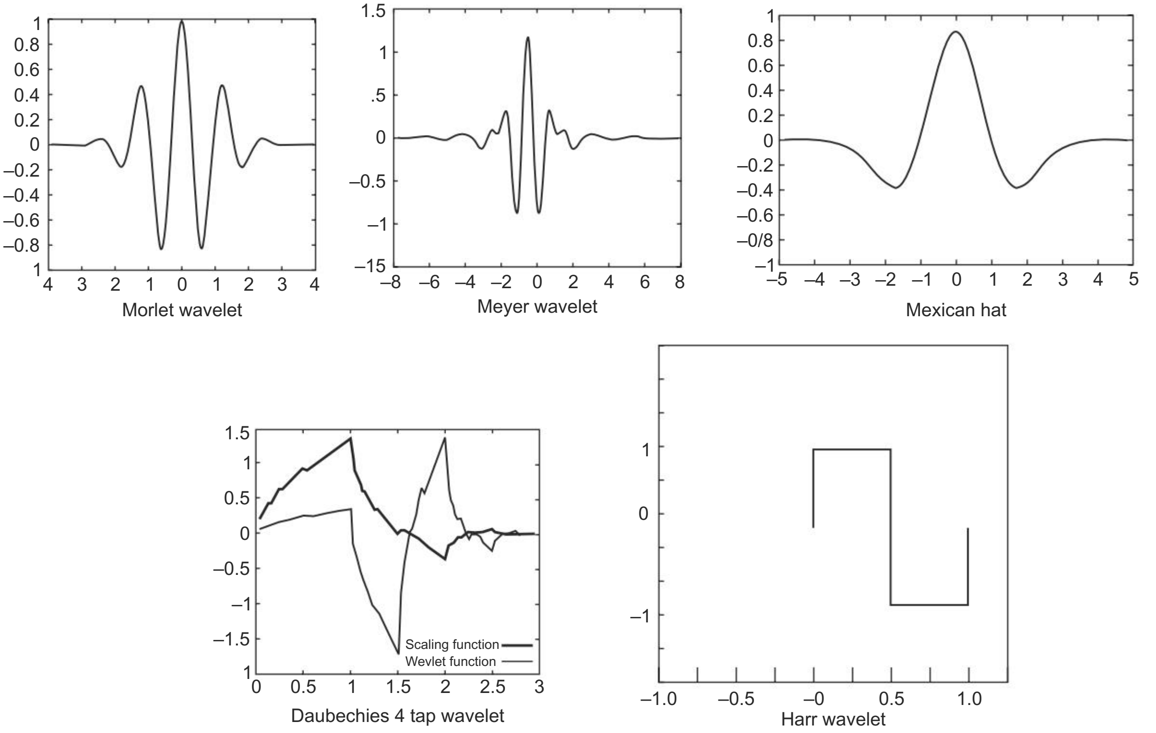


Fig. 1.34 *Some mother wavelets*

$$\psi(t) = \sum_{n} h(n)\phi(2t - n) \tag{1.52c}$$

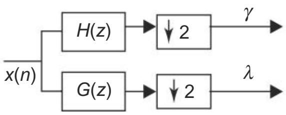


Fig. 1.35 *Filter bank for the computation of 1 D DWT*

Here, $g(n)$ is a low pass filter and $h(n)$ is a high pass filter. In Fig.1.34a, the decimated outputs of low pass and high pass filters are denoted as λ and γ and are called as scaling coefficient and wavelet coefficient.

1.20.1 Discrete Wavelet Analysis of Signals Using Filter Banks

In discrete wavelet analysis, a signal is represented using a set of basis functions called as the wavelets. These basis functions are obtained by shifting and dilating(scaling) the mother wavelet sequence $h(n)$. If a signal $x(k)$ is represented using m basis functions given by

$$h_i(2^{i+1}\, n - k) \text{ for } (\, 0 \le i \le m{-}1, -\infty < k < \infty \,)$$

the 1 dimensional discrete wavelet transform (DWT) is defined as

$$y_i(n) = \sum_{-\infty}^{\infty} x(k)\, h_i(2^{i+1} n - k) \qquad \text{for } 0 \le i \le m - 2 \tag{1.52d}$$

$$y_{m-1}(n) = \sum_{-\infty}^{\infty} x(k) h_{m-1}\, (2^{m-1} n - k) \qquad \text{for } i = m - 1 \tag{1.52e}$$

$y_i(n)$ are called as the wavelet coefficients. The inverse discrete wavelet transform is computed by the expression

$$x(n)=\sum_{i=0}^{m-2}\sum_{k=-\infty}^{\infty} y_i(k)f_i(n-2^{i+1}k)+\sum_{k=-\infty}^{\infty} y_{m-1}(k)f_{m-1}(n-2^{m-1}k) \qquad (1.52f)$$

where $f_i(n-2^{i+1}k)$ are designed such that (1.52f) perfectly reconstructs the original signal $x(n)$. It may be noted that the computation of the DWT and IDWT coefficients are similar to convolution operations. They can be calculated recursively as a series of convolutions and decimations using filter banks considered in section 1.19.6.

For e.g., let us consider the computation of the DWT for $m = 4$ using filter banks. The wavelet coefficients are given by

$$y_0(n)=\sum_{-\infty}^{\infty} x(k)h_0(2n-k) \qquad (1.52g)$$

$$y_1(n)=\sum_{-\infty}^{\infty} x(k)\,h_1(4n-k) \qquad (1.52h)$$

$$y_2(n)=\sum_{-\infty}^{\infty} x(k)\,h_2(8n-k) \qquad (1.52i)$$

$$y_3(n)=\sum_{-\infty}^{\infty} x(k)h_3(8n-k) \qquad (1.52j)$$

They can be computed using the analysis filter banks with the decimators given in Fig.1.36.

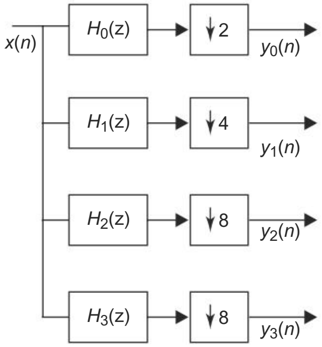


Fig. 1.36 *Analysis filter banks for discrete wavelet transform*

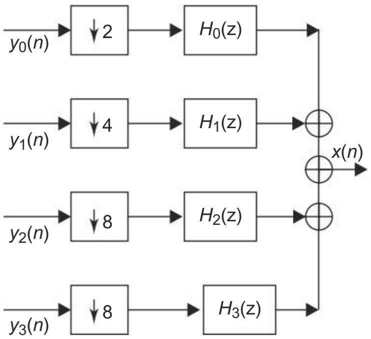


Fig. 1.37 *Synthesis filter bank for Inverse DWT*

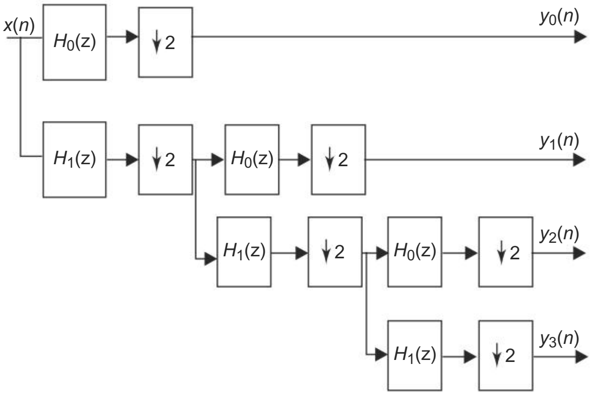


Fig. 1.38 *Octave band filter bank for DWT*

The signal $x(n)$ can be reconstructed by computing the inverse DWT using the interpolators and synthesis filter bank as shown in Fig.1.37. Discrete wavelet transform processes M samples at a time and generates M transform coefficients. When M is chosen to be of the form $M = 2^m$ the wavelet coefficients can be computed using the tree structured filter bank shown in Fig. 1.38. The tree has m nodes or level at which a low pass filter and a high pass filter is used. As the tree is traversed from the root towards the child nodes, the cut off frequency of the low pass filter decreases. However, the tree structure for the computation of DWT differs from that given in Fig.1.18. In this case in the tree the branch corresponding to the output of the low pass filter alone is split into branches; the high pass output is not split further.

Such a special kind of pruned – tree filter bank is called as the octave band filter bank. The resulting tree structure for 3 level 1D DWT (i.e., $m = 3$) is shown in Fig.1.38.

1.20.2 DWT for Two Dimensional Signals

The wavelet theory can be extended for the analysis of 2 dimensional signals and can be used for the compression of 2D signals such as image data. Fig.1.39 shows the filter bank used for the computation of one level 2D DWT. In this figure, the image matrix is converted into a vector and the resulting input samples $x(n)$ are passed through the 2 stages of analysis filters. In the first stage it is passed through the low pass and high pass horizontal filters ($h(n)$ and g(n)) and are sub sampled by two. In the second stage, the output of these two filters are processed by low pass and high pass vertical filters. Because of the decimation by a factor of 2 at the output of both horizontal and vertical filters, if an image matrix of size 512X512 is fed as input to the one level 2D DWT, all the four sub bands (LL1, LH1, HL1, HH1) contain 256X256 transform coefficients. In the 2 level 2D DWT, LL1 component alone is passed through horizontal and vertical filters. In the 3 Level DWT, LL2 component is passed through horizontal and vertical filters. The size of the various sub bands corresponding to 3 level DWT is shown in Fig.1.40. For image processing applications, wavelet transform is increasingly used. They have a number of advantages over the other transforms. One of the advantages is that a subset of the transform coefficients represents a coarse form of the image and can be displayed without computing the inverse transform. This has an attractive application in Progressive Image transmission (PIT) scheme used for Internet applications. It has proved its efficiency in many other fields ranging from general-purpose multimedia to medical imaging. As a consequence, it is a part of the JPEG 2000 standard for still image coding.

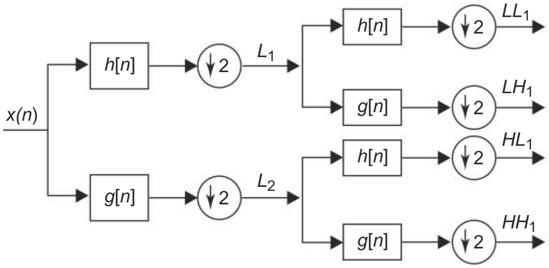


Fig. 1.39 *Block diagram of one level 2D DWT*

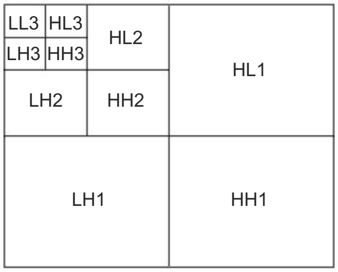


Fig. 1.40 *Subbands corresponding to 3 level 2D DWT*

1.20.3 Condition for Perfect Reconstruction of Signals Using 2 Channel Filter Banks

Two channel analysis and synthesis filters banks are cascaded in applications such as computation of wavelet transform and reconstruction of the signal from the wavelet coefficients. Such a filter bank along with decimator and interpolator is shown in Fig.1.41. The filters at the analysis section and synthesis section may be chosen to satisfy the QMF property. The z transform of the output sequence $y(n)$ can be obtained from the z transform of the input sequence $x(n)$ as follows:

Let $v_0[n]$ and $v_1[n]$ denote the outputs of the analysis filters. $u_0[n]$ and $u_1[n]$ denote the outputs of the analysis filters after decimation by 2. $\hat{v}_0[n]$ and $\hat{v}_1[n]$ denote the outputs of the interpolators. $H_k(z)$ and $G_k(z)$ denote the transfer function of the analysis and synthesis filters respectively (k takes the values 0 and 1 corresponding to the two filters in the filter bank). The z transform of the outputs of the analysis filter ($V_k(z)$) can be written as

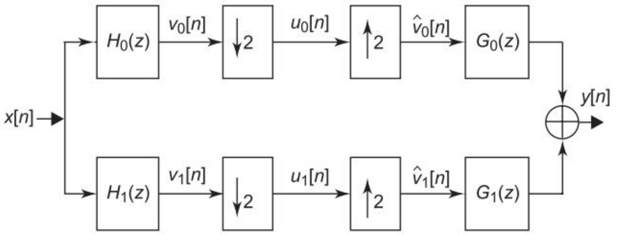


Fig. 1.41 *Two channel filter bank with decimator and interpolator*

$$V_k(z) = H_k(z)X(z) \tag{1.53}$$

Using (1.47o) and (1.48) respectively, the outputs of the decimators ($U_k(z)$) and the z transforms of the outputs of the interpolators ($\hat{V}_k(z)$ can be written as.

$$\hat{V}_k(z) = \frac{1}{2}\left[\{V_k(z) + V_k(-z)\} = \frac{1}{2}\{H_k(z)X(z) + H_k(-z)X(-z)\}\right] \tag{1.54a}$$

$$(\hat{V}_k(z) = U_k(z)_2 \tag{1.54b}$$

The z transform of the final output ($Y(z)$) obtained by combining the outputs of the synthesis filter is given by

$$Y(z) = G_0(z)\hat{V}_0(z) + G_1(z)\hat{V}_1(z) \tag{1.55}$$

$$Y(z) = \frac{1}{2}\{H_0(z)G_0(z) + H_1(z)G_1(z)\}X(z) + \{H_0(-z)G_0(z) + H_1(-z)G_1(z)\}X(-z) \tag{1.56}$$

The second term in (1.56) is the aliasing component due to sampling rate conversion. It can be made zero by properly choosing transfer function of the analysis and synthesis filter bank. One of the choices is

$$H_1(z) = H_0(-z)$$

$$G_0(z) = H_1(-z) \qquad G_1(z) = -H_0(-z)$$

1.20.4 Efficient Implementation of 2 Channel Filter Banks

Two channel filter bank shown in Fig. 1.35 and Fig. 1.39 may be efficiently implemented using poly phase structure with 2 phases and using cascade equivalence. A more efficient implementation scheme for wavelet transform is proposed by Swelden [1998] and is referred to as the lifting scheme. This uses a polyphase structure for the analysis filter and uses a transformation which permits the use of integer wavelet and scaling coefficients instead of floating-point coefficients. The lifting scheme requires less hardware complexity and memory.

For obtaining the lifting scheme, first express the low pass filter G(z) and high pass filter H(z) in polyphase form with two phases as follows:

$$G(z) = G_0(z^2) + z^{-1}G_1(z^2) \tag{1.57a}$$

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2) \tag{1.57b}$$

The $\lambda(z)$ and $\gamma(z)$, the transform of scaling coefficients and wavelet coefficients can be written as

$$\lambda(z) = G_0(z^2)X_0(z^2) + z^{-1}G_1(z^2)X_1(z^2) \tag{1.57c}$$

$$\gamma(z) = H_0(z^2)\,X_0(z^2) + z^{-1}H_1(z^2)\,X_1(z^2) \tag{1.57d}$$

where $X_0(z^2)$, $X_1(z^2)$ denote the z transform of the even indexed and odd indexed input samples respectively. (1.57a) and (1.57b) can be written in the matrix form as follows

$$\begin{bmatrix}\lambda(z)\\ \gamma(z)\end{bmatrix} = \begin{bmatrix}G_0(z^2) & z^{-1}G_1(z^2)\\ H_0(z^2) & z^{-1}H_1(z^2)\end{bmatrix}\begin{bmatrix}X_0(z^2)\\ X_1(z^2)\end{bmatrix} \tag{1.57e}$$

$$= P(z)\begin{bmatrix}X_0(z^2)\\ X_1(z^2)\end{bmatrix} \tag{1.57f}$$

$P(z)$ is called as the polyphase matrix. It is factored into the following form to obtain the lifting structure:

$$[P(z)] = \begin{bmatrix}G_0(z^2) & G_1(z^2)\\ H_0(z^2) & H_1(z^2)\end{bmatrix} = \prod_i \begin{bmatrix}K & 0\\ 0 & \frac{1}{K}\end{bmatrix}\begin{bmatrix}1 & s_i(z)\\ 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\ t_i(z) & 1\end{bmatrix} \tag{1.57g}$$

The resulting filter structure is shown in Fig.1.42. The advantage of this structure is that the low pass and high pass filter outputs are computed using a number of sub filters of smaller length. This can result in upto 50% reduction in the number of multiplications.

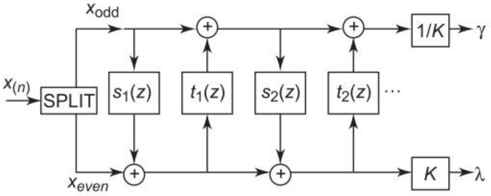


Fig. 1.42 *Computation of 1 level 1D DWT using lifting scheme*

ADAPTIVE FILTERS 1.21

As mentioned in Section 1.3, inverse filters and equalizers are used in a numbers of applications. The impulse response coefficients of these filters may have to be varied with respect to time as the original degradation mechanism itself may vary w.r.t time. For example, the charactersitics of the channel through which the signal is transmitted and reaches the receiver may change with time. A filter in which the filter coefficients are adapted to ensure that the desired signal is obtained as faithfully as possible is called an adaptive filter. If $d(n)$ is the desired signal and $y(n)$ is the output of the filter, the error $e(n)$ at the output of the filter at the nth sampling instant is given by

$$e(n) = d(n) - y(n)$$

One of the methods used for choosing the filter coefficients of the adaptive filter is to choose the coefficients so as to minimise the mean square error $E[e^2(n)]$; $e(n)$ is a random variable which can take any value between -2^{N-1} to 2^{N-1} where N is the no. of bits used for representing a no. in the filter. Knowing the probability distribution of $d(n)$ and $y(n)$, the mean square error (MSE) can be computed. One of the popularly used technique for adaptive filter uses the FIR structure given in Fig. 1.43 and is called as the Wiener filter. The filter coefficients are adapted so as to minimize the MSE. If $h^{(i)}_n$ denotes the nth filter coefficient at the i^{th} iteration, then its value at the $(i+1)^{\text{th}}$ iteration $h^{(i+1)}_n$ is given by

$$h^{(i+1)}_n = h^{(i)}_n + e(n)\alpha x(n) \qquad (1.58)$$

the value of α should be chosen to be in the range

$$0 < \alpha \leq [1/\ (N\,E(x^2(n))]$$

where $x(n)$ is the input to the adaptive filter and $E(x^2(n))$ denotes the average input signal power. The algorithm used above for the adaptation is called as the least mean square algorithm (LMS).

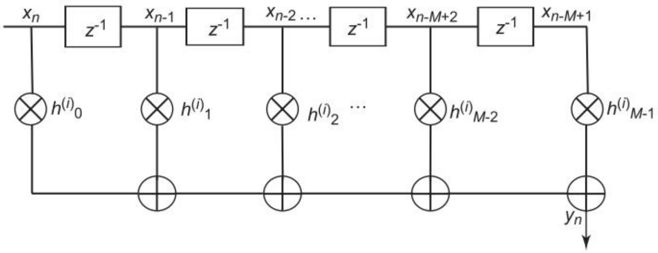


Fig. 1.43 *Adaptive FIR filter*

In addition to the adative equalizers and inverse filters which may require adaptive filters, the adaptive filters are used in a number of applications. For example an application may require the ambient noise present to be cancelled in preference to the signal. This for example, could be used in an audio system used in an automobile. Since the background noise depends on the objects present, the noise generated would also change with time. The LMS algorithm can be used to adjust the filter coefficients to reduce the interference. The noise can be modelled as autoregressive or autoregressive moving average process. They are equivalent to outputs of FIR and IIR filters respectively.

IMAGE DATA COMPRESSION 1.22

1. 22.1 Discrete Cosine Transform and JPEG Standard

One of the popular methods used for compression of image data is using a combination of sample domain and transform domain techniques. An image may be considered to be consisting of a large number of picture elements called as pixels. The image intensity corresponding to each of these pixels is digitized and processed further for either storage or transmission to another point. The no. of pixels which constitutes an image depends on the image resolution used. One of the standards used is to assume the image to have 512 × 512 pixels/ frame. On the other hand the VGA standard assumes a resolution of 640 × 480 pixels/frame. Real time transmission of images requires 25 frames to be transmitted/sec. For monochrome images with a resolution of *512X512* this requires a transmission rate of about 50 Mbps assuming 8 bit accuracy/pixel. For colour images, the transmission rate becomes 150 Mbps. This calls for a large bandwidth and hence, the image data has to be compressed for transmitting it using a moderate BW. For compression of images two commonly used standards are the Joint Photographic Expert Group (JPEG) and Motion Picture Experts Group (MPEG). JPEG is used for still images and MPEG is used for moving images. They both make use of discrete cosine transform for image data compression.

An orthogonal transform is said to be efficient if it is able to pack the energy in the signal in the form of a set of samples in the first few transform coefficients. This is useful for image transmission as well as storage as only less no. of coefficients need to be considered compared to the original signal. For the image data which can be approximated by a first order Markov process to a good accuracy, DCT is an efficient transform. $X(k)$, the k^{th} transform coefficient of 1 Dimensional DCT of a sequence $x(n)$ of length M is given by

$$X(k) = a(k)\sum_{n=0}^{M-1} x(n)\cos\left[(2n+1)k\pi/2M\right] \tag{1.59a}$$

Similarly $x(n)$, the nth data in the sampled sequence, can be expressed in terms of the DCT coefficients as follows:

$$X(n) = \frac{1}{M}\sum_{K=0}^{M-1} a(k)X(k)\cos[(2n+1)k\pi\,/\,2M] \tag{1.59b}$$

where $\alpha(k)$ is $1/\sqrt{2}$ if $k = 0$ and is 1 if $k \neq 0$

The M point DCT and IDCT can be computed using 2M point DFT of a sequence $y(n)$ obtained as follows:

$$\begin{aligned} y(n) &= x(n) \textit{ for } 0 \le n < M \\ &= x(2M - n - 1) \text{ for } M \le n < 2M \end{aligned} \tag{1.59c}$$

Let the k^{th} DFT coefficient of $y(n)$ be $Y(k)$. The DCT coefficient $X(k)$ is given by

$$\begin{aligned} X(k) &= \alpha(k)Y(k)\, e^{-j\,(k\pi/2M)} \text{ for } 0 \le k < M \\ &= 0 \text{ otherwise} \end{aligned} \tag{1.60}$$

The M point IDCT of the transform coefficients $X(k)$ may be obtained by computing the Inverse DFT of the 2M DFT coefficients $Y(k)$ given by

$$\begin{aligned} Y(k) &= X(k) \quad \text{for} \quad 0 \le k < \text{M} \\ &= 0 \quad \text{for} \quad k = M \\ &= -X(2M-1-k) \text{ for } M+1 \le k < 2M \end{aligned} \tag{1.61}$$

If $y(n)$ denotes the inverse DFT coefficients of $Y(k)$, then $x(n)$ is obtained from $y(n)$ as follows:

$$x(n) = y(n) \text{ for } 0 \le n < M \tag{1.62}$$

Since DFT can be computed using FFT, the computational complexity for DCT and IDCT are of the order of $M \log_2 2M$.

Image data is normally an $M{\times}N$ 2D array. This may be converted to an 1 D array and the 1 D DCT may be computed for data compression. Alternately, $X(k, l)$, the MN 2D DCT coefficients of the 2 D image data array x(m,n) may be computed using the 2D DCT equation given by

$$X(k,l) = \alpha(k,l)\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)\cos\frac{(2m+1)k\pi}{2M}\cos\frac{(2n+1)l\pi}{2N} \tag{1.63}$$

The 2D image data array elements $x(m,n)$ can be obtained from the 2D DCT coefficients $X(k, l)$ using the inverse DCT and are given by

$$x(m,n) = \sum_{K=0}^{M-1}\sum_{l=0}^{N-1} X(k,l)\alpha(k,l)\cos\frac{(2m+1)k\pi}{2M}\cos\frac{(2n+1)l\pi}{2N} \tag{1.64}$$

$$\begin{aligned} \text{where } \alpha(k, \text{l}) &= 1/\sqrt{2} \text{ if } k = 0 \text{ and } l = 0 \\ &= 1\ k \neq 0 \text{ and } l \neq 0 \end{aligned} \tag{1.65}$$

To reduce the computational and storage requirements, the MXN image data array is split into square blocks of size $L{\times}L$ where $(L < N)$.

DCT for each of these square blocks may be computed either using the 2D DCT or using 1D DCT. This also permits the computation of DCT using a no. of processors simultaneously. DCT coefficients may be truncated to be zero when their value is below a threshold for data compression. For this purpose either threshold coding or zonal coding may be used. See Jain [1995]. for more details.

Further, the DCT coefficients matrix becomes almost like an upper triangular matrix with almost identical elements along the off diagonals. This property is used to achieve further data compression using run length encoding. In this scheme, a sequence of data which has identical value is encoded into two n bit numbers where the first number denotes the frequency of repetition of the number and the second number denotes the actual value of the number.

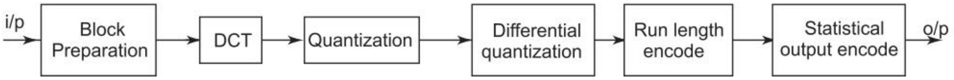


Fig. 1.44 *Block diagram of the JPEG coding scheme*

Block diagram of the JPEG coding scheme is shown in Fig. 1.44. Let the image resolution be 640 × 480 pixels. The function performed by each of the blocks is as follows:

Block Preparation The given colour image matrix of size 640 × 480 is split into 3 matrices one into 4800 luminance (Y) matrices each of size 8 × 8, and the remaining into two 1200 matrices each of size 8 × 8. They are called as the in phase (I) and Quadrature (Q) matrices. The Red, Green, Blue (RGB) components are mapped into Y, I, Q components using the equation given in (1.66).

$$\begin{aligned} Y &= 0.30R + 0.59G + 0.11B \\ I &= 0.60R - 0.28G - 0.32B \\ Q &= 0.21R - 0.52G + 0.31B \end{aligned} \tag{1.66}$$

This is done in order to achieve data compression. Our eye has poor resolution to colour (chrominance) information compared the intensity (Y) information. Hence the full image matrix is mapped into Y matrix. However the I and Q matrices representing the chrominance information is obtained by reducing the image matrix size by a factor of 2 along the X and Y coordinates.

Discrete Cosine Transform and Quantization DCT is computed for each of the 8 × 8 matrices corresponding to the Y, I and Q components. The DCT has very good energy compaction. Hence in the DCT coefficients of an 8 × 8 matrix, only the elements corresponding to the left hand top corner has significant values as shown in Fig. 1.45. The higher order coefficients along the rows and column correspond to high frequency components and they need to be reproduced with only less no. of levels. This is because our eye has a poor high frequency response. Different coefficients are allocated different no. of bits based on their importance. One such allocation scheme is shown in Fig. 1.46.

Using this allocation scheme, the image matrix coefficients are modified as shown in Fig. 1.47. For coding this image either the zonal coding or the threshold coding can be employed. The number of bits required for transmission is reduced further by encoding only the difference in the amplitude corresponding to the DC component in the successive blocks along the vertical and horizontal direction. This corresponds to the top left corner element of the 8 × 8 transform matrix. This is called as the differential quantization and is used only for the DC coefficients.

	90	50	24	16	16	32	0
102	86	46	20	8	16	0	0
62	48	36	16	16	16	0	0
24	20	16	12	24	32	0	0
16	16	16	0	0	0	0	0
16	16	32	32	0	0	0	0
32	32	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig. 1.45 *DCT coefficients of a 8X8 matrix*

8	8	7	6	5	4	3	2
8	8	7	6	5	4	3	2
7	7	7	6	5	4	3	2
6	6	6	6	5	4	3	2
5	5	5	5	5	4	3	2
4	4	4	4	4	4	3	2
3	3	3	3	3	3	3	2
2	2	2	2	2	2	3	2

Fig. 1.46 *Quantization table for the DCT coefficients*

160	90	25	6	2	1	1	0
102	86	23	5	1	1	0	0
31	24	18	4	2	1	0	0
6	5	4	3	3	2	0	0
2	2	2	0	0	0	0	0
1	1	2	2	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig. 1.47 *Quantized DCT coefficients*

Runlength Encoding The quantized matrix is converted in to a linear vector using the zig zag pattern. This helps to combine the 0s using run length encoding explained above.

Source Coding The resulting vector is further compressed using Huff man coding which allocates more number of bits for those amplitudes whose probability of occurrence is less and vice versa.

1.22.2 JPEG 2000 Standard

The JPEG2000 standard for still image compression uses wavelet transform. JPEG2000 is targeted for a no. of applications such as Internet , facsimile, Printing , Scanning, Digital photography, Remote Sensing, Medical imaging, E-commerce and Digital libraries. JPEG2000 is aimed at providing the following features:

Low Bit-rate Performance It should give acceptable quality below 0.25 bpp. Remote sensing and networking require this feature.

Provision for both Lossless and Lossy Compression

Progressive Transmission The standard allows progressive transmission which enables images to be reconstructed with increasing pixel accuracy and resolution depending upon the size of the file transmitted.

Region of Interest Coding It can preferentially allocate more bits to the regions of interest (ROIs) as compared to the non-ROI ones. For example, the area corresponding to the face may be allocated more bits than the rest of the body of a picture of a person.

Comparison of the Performance of JPEG2000 and JPEG Compression Schemes

The codestream obtained after compression of an image with JPEG 2000 is scalable. It can be decoded in a number of ways: by truncating the codestream at any point, one may obtain a representation of the image at a lower resolution, or signal-to-noise ratio. By ordering the codestream in various ways, applications can achieve significant performance increases. However, this is achieved at the cost of higher computational complexity.

At compression ratios less than 25:1 or so, the JPEG performs better numerically than the wavelet coders. At compression ratios above 30:1, JPEG performance rapidly deteriorates, while wavelet coders degrade gracefully well beyond ratios of 100:1.

The block diagram JPEG2000 encoder and decoder are shown in Fig. 1.48 and Fig. 1.49 respectively. A brief description of the various blocks of the encoder are as follows:

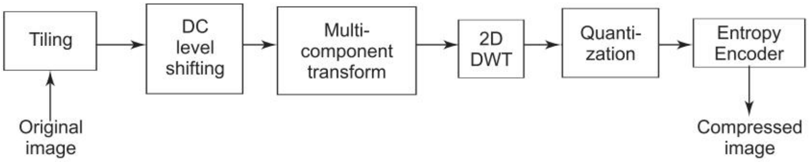


Fig. 1.48 *Block diagram JPEG2000 encoder*

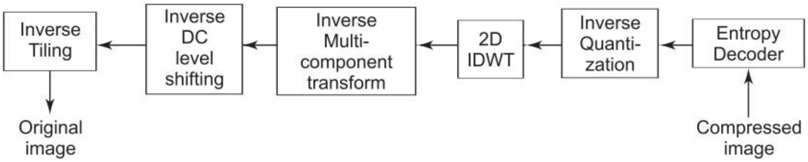


Fig. 1.49 *Block diagram JPEG2000 decoder*

Tiling The input image is partioned into a number of non overlapping blocks if the image is large. Each of these blocks is called a tile. Tile size of 256 × 256 or 512 × 512 is chosen in order to simplify the VLSI implementation of the encoder. Smaller tiles create more boundary artifacts and degrade the compression efficiency.

DC Level Shifting The pixels in the image are normally stored as unsigned integers. For mathematical computations, it is necessary to convert them into two's complement form. The purpose of the DC level shifting is to ensure that the input image samples have a dynamic range that is approximately centred around zero.

Multi-component Transform This is effective in reducing the correlations amongst the multiple components in a multicomponent image. This increases the compression performance. For this purpose, images in the RGB color space are transformed to another color space, leading to three *components* that are handled separately. There are two possible choices:

Irreversible Color Transform (ICT) uses the well known YC_BC_R color space. It is called "irreversible" because it has to be implemented in floating or fixed point and causes round-off errors.

Reversible Color Transform (RCT) uses a modified YUV color space that does not introduce quantization errors, so it is fully reversible. Proper implementation of the RCT requires that numbers are rounded as specified that cannot be expressed exactly in matrix form.

2D DWT

JPEG 2000 uses two types of wavelet transforms:

Irreversible This is implemented using a 9/7 Daubechies filter, whose analysis and synthesis filter coefficients are shown in Table 1.3

It is said to be "irreversible" because it introduces quantization noise that depends on the precision of the decoder.

Reversible It uses the biorthogonal 5/3 Daubechies filter whose analysis and synthesis filter coefficients are shown in Table 1.4. It uses only integer coefficients, so the output does not require rounding (quantization) and so it does not introduce any quantization noise. It is used in lossless coding.

The wavelet transforms are implemented using either the lifting scheme or convolution.

Table 1.3 *Daubechies 9/7 analysis and synthesis filter coefficients*

Analysis Filter Coefficients		
i	*Low-pass filter g(i)*	*High-pass filter h(i)*
0	0.6029490182363579	1.115087052456994
1±	–0.2668641184428723	0.5912717631142470
2±	–0.07822326652898785	–0.05754352622849957
3±	0.01686411844287495	–0.09127176311424948
4±	0.02674875741080976	

Synthesis Filter Coefficients		
i	*Low-pass filter g(i)*	*High-pass filter h(i)*
0	1.115087052456994	0.6029490182363579
1±	0.5912717631142470	-0.2668641184428723
2±	-0.05754352622849957	-0.07822326652898785
3±	-0.09127176311424948	0.01686411844287495
4±		0.02674875741080976

Table 1.4 *Daubechies 5/3 analysis and synthesis filter coefficients*

	Analysis Filter Coeficients		*Synthesis Filter Coefficients*	
i	*Low-pass filter g(i)*	*High-pass filter h(i)*	*Low-pass filter g(i)*	*High-pass filter h(i)*
0	6/8	1	1	6/8
1±	2/8	–1/2	1/2	–2/8

Quantization After the wavelet transform, the coefficients are scalar-quantized to reduce the number of bits required to represent them, at the expense of a loss of quality.

Entropy Coding A two step coding scheme (Tier 1 and Tier 2 coding) is adapted in JEPEG2000. In Tier 1 coding, Embedded block coding with optimized truncation (EBCOT) algorithm is used for compressing the quantized transform coefficients. Each sub band LL,LH,HL,HH is partitioned into small code blocks of size 32 × 32 or 64 × 64 . The EBCOT algorithm generates a highly scalable bit stream for each code block B_i. The bit stream associated with B_i may be independently truncated to a predetermined lengths depending upon the specified distortion. Next a post compression rate distortion step is used to truncate the bit stream of each code block optimally so as to minimize the distortion subject to the bit rate contraint. Tier 2 coding is used to efficiently represent the layer and block summary information for each code block.

LINEAR PREDICTIVE CODER AND SPEECH COMPRESSION 1.23

Another application of digital signal processing is the data compression of the speech signal using the linear predictive coder. For this purpose the human vocal tract is modelled as a linear time varying filter. The voiced sounds(for example vowels) are produced by exciting this filter with quasi periodic impulses. These impulses are called quasi periodic as periods of the impulses are not exactly equal and the amplitude of the impulses are not exactly equal. The unvoiced sounds (for example S, SH) are produced by exciting the above filter with white noise. The time varying filter is normally approximated by a linear time invariant filter over short time intervals. One of the common models used for the vocal tract is the linear predictive coder which is a LTI filter over a short term interval where the output at the nth sample y(n) is predicted based on the past p samples and is given by

$$y(n) = a_1 y(n–1) + a_2 y(n–2) + \cdots a_p y(n–p) + u(n)$$

The error $u(n)$ in turn may be obtained by exciting another linear LTI filter fed with another exciting signal. In the European standard for cellular telephone GSM 6.10, the output rate of the speech coder used is 13 kbps and the input to the coder is voice signal sampled at the rate of 8 Kilo samples with 13 bits/samples. Hence the compression factor is 8. To achieve this, samples corresponding to every 20 msec are used to compute the LPC model parameters and the model parameters corresponding to excitation signal including the LTI filter used if any for generating the excitation. Additional details on estimating the LPC model parameters may be obtained from Rabiner and Juang [1993].

Review Questions

1.1 Find the output $y(n)$ of a linear shift invariant system with unit sample (impulse) response $h(n)$ given by $h(0) = 3$, $h(1) = 2$, $h(2) = 1$ and $h(n) = 0$ for all other value of n if it is fed with an input $x(n)$ which is non zero for only for $n = 0$ and 1. $x(0) = 2$ and $x(1) = 1$.

1.2 Show that the output $y(n)$ of a causal linear time invariant discrete time system for the input $x(n)$ given by $x(n) = e^{j\omega n}$ is $y(n) = x(n)\ H(e^{j\omega})$ where $H(z)$ is the system (transfer) function of the LTI system.

1.3 Find the frequency response of the LSI causal system given by $y(n) = x(n) + a\,y(n-1)$ for $a < 1$ and sketch the magnitude and the phase response of this system. Find also its impulse response. Show that the phase response of this system denoted as $\angle H(e^{j\omega})$ is given by $\angle H(e^{j\omega}) = \omega - tan^{-1}(sin\ \omega / (a - \cos\ \omega))$

1.4 Find the frequency response of the LSI causal system given by $y(n) = x(n) + a\,y(n-2)$ *for* $a < 1$. Find also its impulse response.

1.5 The magnitude of the frequency response of the LSI causal system given by $y(n) = x(n) - ax(n-1) + $ by $(n-1)$ is independent of frequency and $a \neq b$. Find the relationship between a and b. Such a system is called as an all pass system.

[Hint: Assume the magnitude of the numerator of the frequency response to be k times the magnitude of the denominator.]

1.6 Determine the impulse response coefficients of a digital low pass filter with $[H(e^{j2\pi fTs}) = 1$ for $f < fc$ and is 0 *for* $fc < f < fs/2]$.

where $fs = 1/Ts$ is the sampling rate of the digital filter.

1.7 Determine the impulse response coefficients of a digital filter with $H(e^{j2\pi fTs})$ given in Fig. 1.

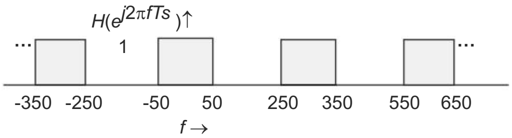


Fig. I *Frequency response of the filter*

1.8 Determine the impulse response coefficients of a digital low pass filter with $[H(e^{j\omega}) = 1$ for $\omega < \omega c$ and is 0 for $\omega c < \omega < \pi$

1.9 Determine the impulse response coefficients of a digital filter with $H(e^{j\omega})$ given in Fig. 2

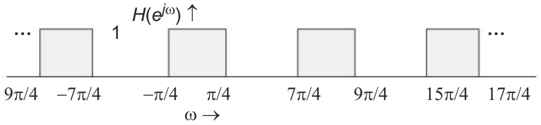


Fig. 2 *Frequency response of the filter*

1.10 Sketch the frequency spectrum of $f(t) = \sin 2\pi(1250)\,t$. Determine the minimum sampling rate required to reconstruct the signal from its samples. What should be the cut off frequency of the reconstruction (smoothing) filter? If the signal is sampled is at the rate of 2000 samples/s what would be the output of the smoothing filter .

1.11 Sketch the frequency spectrum of

$$f(t) = \sin 2\pi(1000)t + \sin 2\pi(1250)t$$

Determine the minimum sampling rate required to reconstruct the signal from its samples. What should be the cut off frequencies of the reconstruction (smoothing) filter? If the signal is sampled is at the rate of 2000 samples/s what would be the output of the smoothing filter .

1.12 Let $f(t)$ be $\sin 2\pi(1000)t + \sin 2\pi(1250)t$ and $y(t) = [f(t)]^2$. Determine the minimum sampling rate required to reconstruct y(t) from its samples. What should be the cut off frequencies of the reconstruction (smoothing) filter?

1.13 A continuous time signal is to be filtered to remove frequency components in the range 10 kHz < f < 25 kHz. The maximum frequency component present in the signal is 50 kHz. If the signal is sampled at the Nyquist rate what range of digital frequencies should be rejected by the filter. Sketch the magnitude characteristics of the ideal digital filter required for this purpose.

1.14 Find the impulse response of a system whose system function is given by

$$X(z) = \frac{2}{(1-0.5z^{-1})(1+0.5z^{-1})}$$

1.15 Find the impulse response of a system whose system function is given by

$$X(z) = \frac{(3-z^{-2})}{(1-0.25z^{-1})(1-0.75z^{-1})}$$

1.16 The impulse response of a discrete time system is given by $h(n) = a^{-n}$ for $0 \le n \le 15$ and 0 otherwise. Find the system function of this filter.

1.17 The impulse response of a discrete time system is given by $h(n)$ = [1, 0.2, 0.04, 0.008, 0.0016, 0, 0, 0, ...]. Find the system function of this filter.

1.18 For a sampling rate of 10KSPS and cutoff frequency of 1 kHz, a low pass filter requires 18 taps. Under what condition will the above filter work as a low pass filter with cut off frequency of 100 kHz? Using the same idea, the above filter was proposed to be used as a low pass filter with cut off frequency of 10 MHz. However the filter was found to be not satisfactory. What could be the reason? (Hint: critical path delay)

1.19 Sketch the frequency response of QMF. How are the impulse response of the two filters related?

1.20 The impulse response of a low pass filter is given by the sequence { 0.9, 0.6, 0.3, 0.1, 0.03, 0.008}. If a high pass filter is implemented as the QMF, find its impulse response.

1.21 (a) Find the system function of the following LTI causal systems

(i) $y_1(n) = x_1(n) + 0.25\, x_1(n-1) + 0.5\, y_1(n-1)$

(ii) $y_2(n) = x_2(n) + 0.5\, x_2(n-1) + 0.75\, y_2(n-1)$

(b) If $x_2(n) = y_1(n)$, find the system function of the LTI system whose input is $x_1(n)$ and the output is $y_2(n)$

(c) If $x_1(n) = y_2(n)$, find the system function of the LTI system whose input is $x_2(n)$ and the output is $y_1(n)$

(d) If the systems given in (a) are cascaded, find the system function of an equivalent LTI system.

(e) If the systems given in (a) are connected in parallel, find the system function of an equivalent system.

1.22 Find the 3 point DFT of the sequence given by $x(n) = 1$ for $n = 0,1,2$

1.23 Find the 8 point DFT of the sequence given by $x(n) = 1$ for $n = 0,1,2$ and $x(n) = 0$ for $n = 3, 4, \ldots 7$.

1.24 Find the 8 point DFT of the infinite sequence given by $a(n) = (0.5)^n$ for $n = 0, 1, 2, \ldots.$

1.25 Find the 16 point DFT of the infinite sequence given by $h(n) = (0.25)^n$ for $n = 0, 1, 2, \ldots.$

1.26 Show that the computation of the DFT of a N point sequence is equivalent to a multiplication of matrix by a vector. What is the $(m,n)^{th}$ element of this matrix.

1.27 It is required to design a low pass digital filter with cut off frequency of 0.5π.

(a). If the sampling period is 1 sec, what should be the cut off frequency of the corresponding analog filter if the digital filter is designed using (i) impulse invariant technique, and (ii) bilinear transform method.

(b). If the sampling period is 0.1 msec, what should be the cut off frequency of the corresponding analog filter if the digital filter is designed using (i) impulse invariant technique (ii) bilinear transform method.

1.28 It is required to design a low pass digital filter with cut off frequency of 0.5π. If the sampling period is 0.01 msec, what should be the cut off frequency of the corresponding analog filter if the digital filter is designed using (i) impulse invariant technique (ii) Bilinear transform method.

1.29 Determine the impulse response coefficients of a digital filter whose frequency response is given by

$H(e^{j\omega}) = 0$ for $|\omega| < .85\pi$ and is 1 for $.85\pi < |\omega| < \pi$

1.30 (a) Obtain the impulse response of the analog filter whose transfer function is given by

$$H(s) = \frac{1}{s - 0.5}$$

(b) If the sampling rate used is $1/T$ what is the impulse response of the equivalent digital filter obtained using the impulse invariant method.

(c) What is the transfer function of the equivalent digital IIR filter?

1.31 The transfer function of an analog filter is given by

$$H(s) = \frac{1}{s - a}$$

(a) What is the transfer function of the equivalent digital IIR filter if bilinear transform is used to convert the analog filter to the equivalent digital filter? Assume the sampling rate to be 1/T.

(b) What is the impulse response of the equivalent digital IIR filter?

1.32 Obtain the impulse response of the analog filter whose transfer function is given by

$$H(s) = \frac{1}{s - 0.75}$$

If the sampling rate used is 10000 samples/sec what is the impulse response of the equivalent digital filter obtained using the impulse invariant method.

1.33 Determine the impulse response coefficients of a digital FIR filter whose frequency response is given by

$H(e^{j\omega}) = 0$ for $|\omega| < .75\pi$ and is 1 for $.75\pi < |\omega| < \pi$

using a filter of order 8 and windowing using Black man window given by

$\omega(n) = 0.42 - 0.5\cos(2\pi n/N{-}1) + 0.08\cos(4\pi n/N{-}1)$ for $0 \le n \le N{-}1 = 0$ otherwise.

1.34 A digital FIR filter of order 8 is to be designed using the frequency sampling method. The frequency response of the filter is given by

$$\begin{aligned} H(e^{j\omega}) &= 1 \text{ for } |\omega| < .3\pi \\ &= 0 \text{ for } .3\pi < |\,\omega\,| < \pi \end{aligned}$$

Determine the impulse response coefficients.

1.35 Show that the frequency response function $H(e^{j\omega})$ of a linear phase filter with M (odd) taps and with impulse response $h(n)$, $n = 0, \ldots N{-}1$ symmetrical about $(N{-}1)/2$ is given by

$$H(e^{j\omega}) = e^{-j\omega(N-1)/2}\sum_{n=0}^{(N-1)/2} a(n)\cos\omega n$$

where $a(0) = h[(N{-}1)/2]$ and $a(n) = 2h[(N{-}1)/2 - n]$ for $n = 1,..\,(N{-}1)/2$

1.36 Show that the frequency response function $H(e^{j\omega})$ of a linear phase filter with M (even) taps and with impulse response $h(n)$, $n = 0, \ldots N{-}1$ symmetrical about (N−1)/2 is given by

$$H(e^{j\omega}) = e^{-j\omega(N-1/2)}\sum_{n=1}^{N/2} b(n)\cos\left[\omega(n-1/2)\right]$$

where $b(n) = 2h[N/2 - n]$ for $n = 1, \ldots N/2$

1.37 Show that the frequency response function $H(e^{j\omega})$ of a linear phase filter with M (odd) taps and with impulse response $h(n)$, $n = 0, \ldots N{-}1$ anti symmetrical about $(N{-}1)/2$ is given by

$$H(e^{j\omega}) = e^{-j\omega(N-1)/2}e^{j\pi/2}\sum_{n=1}^{(N-1)/2} c(n)\sin(\omega n)$$

where $c(n) = 2h[(N{-}1)/2 - n]$ for $n = 1, \ldots (N{-}1)/2$ and $h[(N{-}1)/2] = 0$

1.38 Show that the frequency response function $H(e^{j\omega})$ of a linear phase filter with M (even) taps and with impulse response $h(n)$, $n = 0, \ldots N{-}1$ symmetrical about $(N{-}1)/2$ is given by

$$H(e^{j\omega}) = e^{-j\omega(N-1)/2}e^{j\pi/2}\sum_{n=1}^{N/2} d(n)\sin\left[\omega(n-1/2\right]$$

where $b(n) = 2h[N/2 - n]$ for $n = 1, \ldots N/2$

1.39 Realize the system given by the transfer function

$$X(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4}$$

in direct form I and direct form II.

1.40 Realize the system given by the transfer function

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4}}{1 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4}}$$

in (a) direct form I (b) the transpose of direct form I (c) canonic form

1.41 State and explain the three cascade equivalence relations for the multirate DSP systems using block diagrams.

1.42 With a diagram, explain how a poly phase filter with three phases followed by decimator by a factor of three can be efficiently implemented using noble identity. Compare the computational complexity with the direct implementation approach.

1.43 Show how the computational complexity of an FIR filter of order 21 can be reduced by using 3 polyphase filters.

1.44 Find the output of the systems consisting of (i) a decimator by 3 followed by a interpolator by 3 (ii) an interpolator by 3 followed by a decimator by 3. If the sequence 0, 1,2,3,4,5, ... is fed to both of the systems, what will be the output of both of the systems?

1.45 Draw the block diagram of a sub band encoder for speech signal. Explain how it can be implemented using filter bank.

1.46 A multirate system is used to convert the sampling rate from 32 to 48 KSPs. Show how this system can be efficiently implemented using noble identity.

1.47 Derive the condition for the perfect reconstruction of 2 channel filter bank.

1.48 If transfer function of the high pass filter is $\frac{1}{2}(1 - z^{-1})$, find the T.F. of other filters at analysis and synthesis sides for perfect reconstruction of a 2 channel filter bank.

1.49 A low pass signal bandlimited to $\pm\pi/4$ radians is decimated by a factor of 4. Sketch the magnitude of the frequency response at the output of the decimator.

1.50 Derive the z transform of the output of decimator by a factor of M.

1.51 A 64 tap filter followed by decimation by 4 is efficiently implemented using poly phase scheme and noble identity. The filters use multipliers and adders with computation time of 8ns, 2ns. Compare the maximum sampling rates of the original filter and the one using the polyphase structure and noble identity. Assume Direct Form I realization.

1.52 A low pass signal bandlimited to $\pm\pi/3$ radians is decimated by a factor of M. What is the maximum value of M for which the aliasing does not occur. Sketch the magnitude of the frequency response at the output of the decimator corresponding to this M.

1.53 An interpolator by 4 followed by 128 tap filter is efficiently implemented using poly phase scheme and noble identity. The filters use multipliers and adders with computation time of 8ns, 2ns. Compare the maximum sampling rates of the original filter and the one using the polyphase structure and noble identity. Assume Direct Form II realization.

1.54 Write the equation for the computation of short time Fourier transform (STFT). How is its frequency spectrum plotted? What are the limitations of STFT?

1.55 Given STFT, how is the original function $f(n)$ obtained?

1.56 What is meant by spectrogram? Give an example plot.

1.57 What is the limitation of STFT? How does the frequency resolution and time resolution vary with window size?

1.58 Explain how STFT can be found using DFT/FFT.

1.59 What is the relationship between R and N of STFT? ($N > R$) What do R and N denote?

1.60 The down stream data corresponding to ADSL modem is split into no. of sub streams (e.g., using serial to parallel converter) and each substream data modulates a separate carrier. How is this scheme superior compared to the case where the entire data stream modulates a single carrier?

1.61 Why is multicarrier modulation scheme preferred for wireless LAN?

1.62 Explain how the DFT techniques can be used for transmission of discrete multi tone modulated signals.

1.63 802.11a WLAN systems employ a 64 point transform with *52* of the subcarriers actually used for *Carrying* user data from 16-QAM alphabet. 16 symbol duration is used as guard time between adjacent block of data.The symbol rate used is 20 Msymbols/sec What is the bit rate corresponding to user data if (i) no error correction code is used (ii) A convolution code with coding efficiency of 0.75 is used?

1.64 State the three differences between conventional FDM scheme and OFDM.

1.65 What is meant by (i) coded OFDM (ii) Multirate OFDM? What are their applications?

1.66 Draw the block diagram of OFDM transmitter and OFDM receiver

1.67 State the relationship between the sequence at the output of IFFT block and input of FFT block used with the OFDM transmitter and receiver.

1.68 How is the input to the IFFT block formed from the input data stream?

1.69 Write the equation for continuous time wavelet transform. What is the condition to be satisfied for a mother wavelet to have DWT?

1.70 What are the two advantages of DWT over DCT for image Data compression

1.71 State two differences between basis functions used for wavelet transform and Fourier series expansion.

1.72 Draw the block diagram of 3 level ID DWT. Explain how a single LPF & HPF may be used at all levels.

1.73 State the Heisenberg's uncertainty principle w.r.t. time & freq. resolution. Which function meets the bound?

1.74 Sketch the waveforms of any three wavelets.

1.75 Show that the computation of wavelet transform is equivalent to analyzing a function using window functions of varying durations.

1.76 Explain why different mother wavelets are used for wavelet transform unlike the fourier series/FT?

1.77 Write the equation for continuous time wavelet transform.

1.78 Does DWT exist corresponding to every CWT?

1.79 Explain how scaling function is obtained from mother wavelet?

1.80 Explain how scaling function and wavelet function at one level is related to scaling function at another level

1.81 Draw the block diagram of 1 level 2D DWT. Qualitatively explain how 2D DWT of an image of size 512 x 512 is computed. What is the size of each of the transform coeff. matrices?

1.82 What are the two advantages of DWT over DCT for image Data compression?

1.83 Show how a three level ID DWT consisting of 4 subband outputs can be obtained using a filter bank consisting of 4 filters (Use Noble Identity). In the filters used which parameter is the same for all the filters?

1.84 Write the polyphase matrix for the filter bank with 2 filters.

1.85 Explain the lifting scheme for the computation of 1 D DWT.

1.86 What is meant by (i) lifting (ii) dual lifting blocks?

1.87 What are the two steps which are repeatedly carried out to derive the lifting structure? What are the two advantages of lifting scheme over the conventional scheme?

1.88 Show how Discrete cosine Transform (DCT) of an 8 point sequence can be computed using a 2N point FFT.

1.89 Show how the DCT coefficients of an eight point sequence can be computed using a bank of filters.

Self Test Questions

1.1 $y(n)$ denotes the output of a linear shift invariant system with unit sample (impulse) response $h(n)$ given by $h(0) = 3$, $h(1) = 2$, $h(2) = 1$ and $h(3) = 0.5$. $h(n) = 0$ for all other value of n. This system is fed with an input $x(n)$ which is non zero for only for $n = 0$, 1 and 2 with $x(0) = 2$, $x(1) = 1$ $x(2) = 0.5$. The minimum value of n ($n > 0$) for which $y(n) = 0$ is

(a) 8 (b) 7 (c) 6 (d) 5

1.2 Which of the following systems are causal?

(a) $y(n) = x(n) + 0.25\,x(n-1) + 0.5\,x(n-2)$

(b) $y(n) = x(n) + 0.25\,x(n-3) + 0.75\,y(n-1)$

(c) $y(n) = x(n) + 0.5\,x(n-1) + 0.5\,y(n-1) + 0.8\,y(n-2)$

(d) $y(n) = x(n+1) - 0.5\,x(n-1) + 0.8\,x(n-1)$

1.3 The minimum sampling rate required to reconstruct the signal $f(t) = \sin 2\pi f(1000)t$ from its samples is ——— Hz.

(a) 500 (b) 1000

(c) 2000 (d) none of the above

1.4 The cut off frequency of the reconstruction (smoothing) filter for a signal sampled at a frequency of *fs* is ———.

(a) *fs*/2 (b) fs

(c) 2*fs* (d) *fm* i.e. the signal max. frequency

1.5 The signal $f(t) = \sin 2\pi f(1000)t$ is sampled is at the rate of 1800 samples/sec. The frequency of the signal at the output of the smoothing filter is ——— Hz.

(a) 1000 (b) 900 (c) 1100 (d) 800

1.6 Speech is digitized using a sampling rate of 8 kHz. An antialiasing filter with cut off frequency of 3.4 kHz is preceded by the sampler. The loss of the speech signals in the frequency range 3.4–20 kHz due to antialiasing filter introduces a degradation in the signal quality. On the otherhand sampling without the antialiasing filter also introduces degradation in the signal quality. Which of the following statements are true.

(a) Degradation with antialiasing filter is less

(b) Degradation without antialiasing filter is less

(c) Degradation with or without the filter is the same.

1.7 The no. of stages of FFT computations required for the computation of the DFT of a 512 point sequence is ———

(a) 9 (b) 8 (c) 7 (d) 6

1.8 At the 5th stage of FFT computation of a 512 point FFT, the no. of distinct twiddle factors used is 2———.

(a) 2 (b) 3 (c) 4 (d) 5

1.9 Which of the the following properties are true for an IIR filter designed using bilinear transform method.

(a) Requires the use of antialiasing filter

(b) Requires prewarping the filter cutoff frequencies

(c) Not suited for the design of high pass filters

(d) Results in unique mapping from analog to digital frequencies.

1.10 Which of the the following properties are true for an IIR filter designed using impulse invariant technique .

(a) Requires the use of antialiasing filter

(b) Requires prewarping the filter cutoff frequencies

(c) Not suited for the design of high pass filters

(d) Results in unique mapping from analog to digital frequencies.

1.11 Which of the following properties are true for the linear phase FIR filter with even no. of coefficients and symmetric impulse response.

(a) not suited for high pass filter but has real magnitude response

(b) suited for high pass filter and has real magnitude response

(c) has imaginary magnitude response

(d) suited for differentiators and hilbert transformers

1.12 Which of the following properties are true for the linear phase FIR filter with odd no. of coefficients and symmetric impulse response.

(a) not suited for high pass filter but has real magnitude response

(b) suited for high pass filter and has real magnitude response

(c) has imaginary magnitude response

(d) suited for differentiators and hilbert transformers

1.13 Which of the following characteristics are true for a half band filter

(a) In a filter with M taps, the value of the coefficients of the odd taps is zero.

(b) The frequency response of the filter is antisymmetric w.r.t to the frequency fs/4 where fs is the sampling frequency.

(c) The ripple in the pass band is equal to that in the stop band.

(d) These filters are used in multirate systems

Table I *Impulse response coefficients of 4 filters*

Filter no	*h(0)*	*h(1)*	*h(2)*	*h(3)*	*h(4)*	*h(5)*	*h(6)*
I	2	6	7	6	2	0	0
II	2	5	8	–5	–2	0	0
III	2	3	4	4	3	2	0
IV	2	0	3	0	2	0	0

1.14 Among the filters given in Table 1, the filter which has imaginary magnitude response and is suited for differentiators and hilbert transformers is

(a) I (b) II (c) III (d) IV

1.15 Among the filters given in Table 1, the filter which is not suited for high pass filter but has real magnitude response is

(a) I (b) II (c) III (d) IV

1.16 Among the filters given in Table 1, the filter which is suited for high pass filter and has real magnitude response

(a) I (b) II (c) III (d) IV

1.17 Among the filters given in Table 1, the filter whose frequency response is antisymmetric w.r.t to the frequency $fs/4$ is.

(a) I (b) II (c) III (d) IV

1.18 The no. of multiplications required for performing the convolution of two sequences with identical length 8 using the direct method is

(a) 256 (b) 120 (c) 128 (d) 64

1.19 The no. of multiplications required for performing the convolution of two sequences with identical length 8 using the indirect method using FFT is

(a) 256 (b) 120 (c) 192 (d) 185

1.20 A multirate system is required for converting the sampling rate from 48 K samples to 42.1 K samples, the interpolation factor, decimation factor to be used is

(a) 147, 160 (b) 160,147 (c) 480, 421 (d) 421,480

1.21 The z transform of the input to a decimator by factor 4 is $1+z^{-1}+z^{-2}+\mathrm{z}^{-3}$. The z transform at the output is ———.

2 INTRODUCTION TO PROGRAMMABLE DSPs

The programmable digital signal processors (P-DSPs) are designed with features that are specifically required for digital signal processing applications. The conventional microprocessors are meant for general purpose applications and hence they do not have these features. However, an advanced microprocessor or a RISC processor may use some of the techniques adopted in P-DSPs or may even have instructions that are specifically required for DSP applications. They may have performances close to that of a P-DSP for certain operations. For example, the DEC Alpha 21064 computes a 1024 point complex FFT in 480 μs, as compared to the Analog device ADSP 21060 that takes about 460 μs to carry out the same operation. However in terms of low power requirement, cost, real time I/O capability and availability of high speed on-chip memories, the P-DSPs have an advantage over the advanced microprocessors and the RISC processors. In this chapter some of the features specifically required for performing digital signal processing operations efficiently are discussed in detail.

MULTIPLIER AND MULTIPLIER ACCUMULATOR (MAC) 2.1

One of the most common operations required in digital signal processing applications is array multiplication. For example, convolution and correlation require array multiplication. In Chapter 1, it was shown how the array multiplication can be done using a single multiplier and adder. The implementation scheme is reproduced in Fig. 2.1. One of the important requirements of these array multipliers is that they have to process the signals in real time. Before the next sample of the input signal arrives at the input to the array, the array multiplication should be completed. This requires the multiplication as well as accumulation to be carried out using hardware elements. There are two approaches to solve this problem. A dedicated MAC unit may be implemented in hardware, which integrates multiplier and accumulator in a single hardware unit. This approach is adopted by the Motorola DSP processor DSP5600X. The other approach is to have multiplier and accumulator separate. For example, in the Texas Instruments DSP processor, 320C5X, the output of the multiplier is stored into the product register. The content of this in turn can be added to accumulator register ACC in the central ALU. In both of the

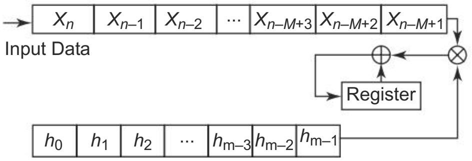


Fig. 2.1 *Implementation of convolver with single multiplier/adder*

above approaches, the MAC operation can be completed in one clock cycle. The presence of H/W multipliers and/or multiplier accumulator is one of the mandatory requirements of a P-DSP.

In Fig. 2.1, y_n , the output at the nth sampling instant, is obtained by multiplying the array $\boldsymbol{x_n} = [x_n\ x_{n-1}\ x_{n-2} \cdots , x_{n-M+3}\ x_{n-M+2}\ x_{n-M+1}]$ corresponding to the present and the past $M - 1$ samples of the input with the array $\mathbf{h} = [h_0\ h_1\ h_2 \ldots h_{M-3}\ h_{M-2}\ h_{M-1}]$ corresponding to the impulse response sequence. To obtain y_{n+1}, the input signal array $\boldsymbol{x_{n+1}}$ is multiplied with the array $\mathbf{h}$. The vector $\boldsymbol{x_{n+1}}$ is obtained by shifting the array $\boldsymbol{x_n}$ towards right so that the $(n + 1)^{\text{th}}$ sample of the input data x_{n+1} becomes the first element and all the elements of $\boldsymbol{x_n}$ are shifted towards right by 1 position so that the i^{th} element of $\boldsymbol{x_n}$ becomes the $(i + 1)^{\text{th}}$ element of $\boldsymbol{x_{n+1}}$. Instead of shifting the elements of $\boldsymbol{x_n}$ towards right all at a time after finishing the vector multiplication, each of the elements may be shifted separately soon after the MAC operation that uses these elements is over. For example, after obtaining the product $x_{n-M+1}\ h_{M-1}$ the element x_{n-M} may be made to be equal to x_{n-M+1}. Similarly, after obtaining the product $x_{n-M+2}\,h_{M-2}$ the element x_{n-M+1} may be made equal to x_{n-M+2} and so on.

This is achieved in P-DSP by using a special instruction called MACD multiply accumulate with data shift. For example, TMS320C5X has the instruction MACD pma, dma, which multiplies the content of the program memory pma with the content of the data memory with address dma and stores the result in the product register. The content of product register is added to the accumulator before the new product is stored. Further, the content of dma is copied to the next location whose address is dma + 1.

MODIFIED BUS STRUCTURES AND MEMORY ACCESS SCHEMES IN P-DSPs 2.2

It may be noted that the MAC operation with data move (i.e. the MACD instruction) requires four memory accesses per instruction cycle. (An instruction cycle is the time that elapses since an instruction is fetched till the particular instruction completes execution including the time taken for writing the result into a register or memory. Many of the instructions in P-DSPs including the MACD instruction require only one processor clock period/instruction cycle. In the conventional microprocessors one instruction cycle corresponds to several clock periods.) The four memory accesses/clock period required for the MACD instructions are as follows:

1. Fetch the MACD instruction from the program memory
2. Fetch one of the operands from the program memory
3. Fetch the second operand from the data memory
4. Write the content of the data memory with address dma into the location with the address dma + 1

The relatively static impulse response coefficients are stored in the program memory and the samples of the input data are stored in the data memory. If the MACD instruction is to be executed in a machine with *Von Neumann architecture,* it requires four clock cycles. This is because in the Von Neumann architecture shown in Fig. 2.2 there is a single address bus and a single data bus for accessing the program as well as data memory area. One of the ways by which the number of clock cycles required for the memory access can be reduced is to use more than one bus for both address and data. For example in the *Harvard architecture* shown in Fig. 2.3, there are two separate buses for the

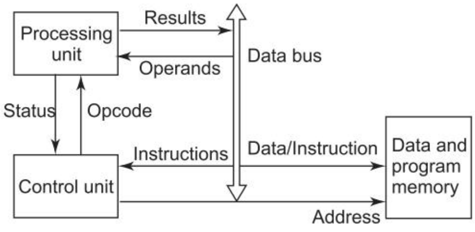


Fig. 2.2 *Von Neumann architecture*

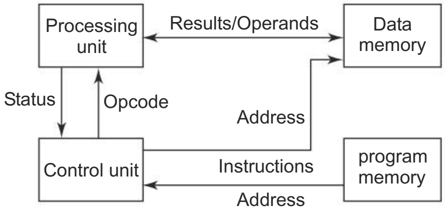


Fig. 2.3 *Harvard architecture*

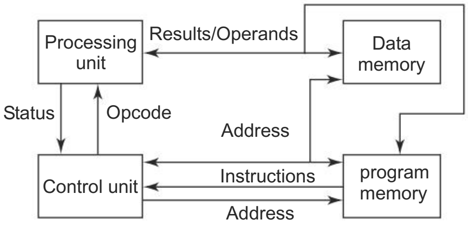


Fig. 2.4 *Modified harvard architecture*

program and data memory. Hence the content of program memory and data memory can be accessed in parallel. The instruction code can be fed from the program memory to the control unit while the operand is fed to the processing unit from the data memory. The processing unit consisting of the registers and processing elements such as MAC units, multiplier, ALU, shifter, etc., are also referred to as data path. The P-DSPs follow the modified Harvard architecture shown in Fig. 2.4. One set of bus is used to access a memory that has both program and data and another that has data alone. Data can also be transferred from one memory to another. The modified Harvard architecture is used in several P-DSPs, for example P-DSPs from Texas Instruments and Analog devices.

With the Harvard architecture, the number of memory accesses/clock cycle was shown to be two. This can be increased further by using more number of buses. For example, by using three separate address and data buses, the number of memory accesses/clock cycle can be increased to three. Motorola DSP5600X, DSP96002, etc. have three separate buses. TMS320C54X has four address buses.

Since the cost of an IC increases with the number of pins in the IC, extending a number of buses outside the chip would unduly increase the price. Hence the P-DSP's use multiple buses only for connecting the on-chip memory to the control unit and data path. For accessing off-chip memory only a single bus is used for accessing both the program memory and data memory. Because of this, any operation that involves an off-chip memory is slow compared to that using the on-chip memory.

MULTIPLE ACCESS MEMORY 2.3

The number of memory accesses/clock period can also be increased by using a high speed memory that permits more than one memory access/clock period. For example, the DARAM, the dual access RAM, permits two memory access/clock period. Multiple access RAM may be connected to the processing unit of the P-DSP by using the Harvard architecture. For example DARAM connected to a P-DSP with two independent data and address buses can be used to achieve four memory accesses/ clock period.

MULTIPORTED MEMORY 2.4

Another technique that is adopted for increasing the number of accesses/clock period is to use multiported memory. For example the dual port memory has two independent data and address buses as shown in Fig. 2.5 and hence two memory accesses can be achieved in a clock period. Multiported memories dispense with the need for storing the program and data in two different memory chips in order to permit simultaneous access to both program and data memory. However, one of the major limitations of the dual-ported memory is the increase in the cost compared to two single port

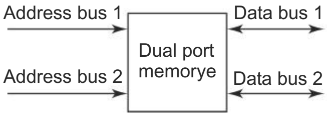


Fig. 2.5 *Block diagram of a dualported memory*

memory of the same total capacity. This is because of the increased number of pins and larger chip area required for the dualported memory. Larger number of I/O pins require a larger and more expensive package and a larger die size.

Some P-DSPs combine the modified Harvard architecture with the dualported memories. For example, the Motorola DSP 561XX processors have a singleported program memory and a dualported data memory. Hence one program memory access and two data memory accesses can be achieved per clock period.

VLIW ARCHITECTURE 2.5

Another architecture used for P-DSPs, for example in TMS320C6X, is the very long instruction word (VLIW) architecture. These P-DSPs have a number of processing units (data paths). In other words, they have a number of ALUs, MAC units, shifters, etc. The VLIW is accessed from memory and is used to specify the operands and operations to be performed by each of the data paths. As shown in Fig. 2.6, the multiple functional units share a common multiported register file for fetching the operands and storing the results. Parallel random access by the functional units to the register file is facilitated by the read/write cross bar. Execution of the operations in the functional units is carried out concurrently with the load/store operation of data between a RAM and the register file.

The performance gains that can be achieved with VLIW architecture depends on the degree of parallelism in the algorithm selected for a DSP application and the number of functional units. The throughput will be higher only if the algorithm involves execution of independent operations. For example, in Fig. 2.1, by using eight functional units, the time required for convolution can be reduced by a factor of 8 compared to the case where a single functional unit is used.

However, it may not always be possible to have independent stream of data for processing. Further the number of functional units is also limited by the hardware cost for the multiported register file and cross bar switch.

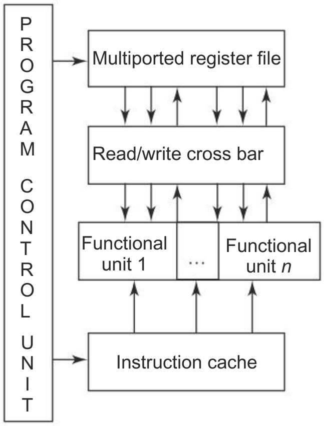


Fig. 2.6 *Block diagram of the VLIW architecture*

PIPELINING 2.6

One of the approaches adopted for increasing the efficiency of the advanced microprocessors as well as P-DSPs is instruction pipelining. An instruction cycle starting with the fetching of an instruction and ending with the execution of the instruction including the time storage of the results can be split into a number of microinstructions. Execution of each of the microinstructions is also referred to as one phase of an instruction. For example, an instruction cycle requiring four microinstructions can be said to be in four phases as follows:

1. Fetch phase in which the instruction is fetched from the program memory
2. Decode phase in which the instruction is decoded
3. Memory read phase in which the operand required for the execution of the instruction may be read from the data memory

4. Execution phase in which execution as well as the storage of the results in either one of the registers or memory is carried out

Each of the above microinstructions may be carried out separately by four functional units. Let us assume that each of the above four phases take equal time for completion. In this case in a conventional microprocessor with no pipelining, each of the functional units is busy only 25% of the time. This is because only one instruction is processed at the CPU at a time. Figure 2.7 shows when each of the functional unit is busy when a program containing three instructions I1, I2, I3 is executed.

Value of *T*	Fetch	Decode	Read	Execute
1	I1			
2		I1		
3			I1	
4				I1
5	I2			
6		I2		
7			I2	
8				I2
9	I3			
10		I3		
11			I3	
12				I3

Fig. 2.7 *Instruction cycles of processor with no pipelining*

Value of *T*	Fetch	Decode	Read	Execute
1	I1			
2	I2	I1		
3	I3	I2	I1	
4	I4	I3	I2	I1
5	I5	I4	I3	I2
6	I6	I5	I4	I3
7	I7	I6	I5	I4
8	I8	I7	I6	I5
9	I9	I8	I7	I6
10		I9	I8	I7
11			I9	I8
12				I9

Fig. 2.8 *Instruction cycles of a processor with pipelining*

The functional units can be kept busy almost all the time by processing a number of instructions simultaneously in the CPU. For example, in a machine with four functional units, four instructions I1, I2, I3 and I4 can be processed simultaneously as shown in Fig. 2.8. When I1 enters the decode phase I2 can enter the opcode fetch phase. When I1 enters the operand read phase I2 enters the decode phase and I3 enters the opcode fetch phase. When I1 enters the execute phase I2 enters the operand read phase I3 enters the decode phase and I4 enters the opcode fetch phase. The pipeline is fully loaded now and all the functional units have useful work to do. The instructions that follow I4 keep the functional units busy till the program is exited. Let *T* denote the time required for each phase of the instruction. One clock cycle of the processor corresponds to *T*. In a period of 12*T* only three instructions can be executed in a machine without pipelining. In the same period nine instructions can be executed as shown in Fig. 2.8. Hence the throughput is increased by a factor of 3 in this case.

It may be noted that the initial latency of a machine with four phases is 4*T*. Hence for executing a program with *N* instructions, the time required for execution is $(N + 4)T_s$. With a non-pipelined machine, the time required for executing *N* instructions is 4*NT*.

Instruction pipeline shown in Fig. 2.8 corresponds to a highly optimistic case. In the case of processors requiring single clock cycle for execution for each of the instructions in the program, the throughput shown in Fig. 2.8 can be achieved. This is normally achieved with restricted instruction set computers (RISC). However in complex instruction set computers (CISC), there are also instructions with multiple word requiring multiple clock cycles for execution. In this case all the functional units cannot be kept busy all the time. For example, in the case of call and branch instructions of a P-DSP, four phases or *T* states are required for the call/branch instruction to exit execution phase. By that time two more single word instructions or one double instruction enters the instruction pipeline. These instructions should not be executed. Hence two words have to be flushed out of the instruction pipeline before the instructions are fetched starting from the new program address.

To overcome this problem, some of the P-DSPs have special branch/call and return instructions called as delayed branch/call/return instructions. When the delayed branch instruction is executed, the program branches to the new program address only after the two 1 -word instructions or the single 2-word instruction following the branch instruction are completely executed. Similarly, when the delayed call instruction is executed, the program calls to the subroutine only after the two 1-word instructions or the single 2-word instruction following the call instruction are completely executed. When the delayed call/branch/return instructions are executed, there is no need for flushing the pipeline and maximum throughput is obtained. Examples of pipeline operation of delayed as well as undelayed branch/call instructions are given in Chapter 4.

The throughput efficiency of the pipeline may also be reduced because of conflicts between the instructions in the instruction pipeline in different phases. This happens if the same memory is used to store the data and program and there is only a single address bus for addressing both the program and data memory. This is true in the case of off-chip memory. For example, an instruction in fetch phase may try to fetch the instruction code from a memory chip that is also accessed by another instruction that is in the operand read phase. To avoid the conflict, the operand read phase will be done first and the opcode fetch phase will be repeated till there is no conflict again.

The number of instructions that are processed simultaneously in the CPU, also referred to as depth of the instruction pipeline, differs in different families of P-DSPs. The pipeline depths of some of the P-DSPs are given in Table 2.1.

Table 2.1 *Instruction pipeline depth of some P-DSPs*

P-DSP Name/family	*Pipeline Depth*
Analog devices	2
Motorola DSP5600X	3
Tl TMS320C5X	4
Tl TMS 320C54X	6

SPECIAL ADDRESSING MODES IN P-DSPs 2.7

In addition to the addressing modes such as direct, indirect and immediate supported by the conventional microprocessors, P-DSPs have special addressing modes that permit single word/instruction format and thereby speed up the execution by making effective use of the instruction pipelining. Further there are also special addressing modes such as cyclic addressing and bit reversed addressing that are specifically tailored for DSP applications. The details of these addressing are presented next.

2.7.1 Short Immediate Addressing

This permits the operand to be specified using a short constant that forms part of a single word instruction. The length of the short constant depends on the instruction type and the P-DSP. For example in the case of Tl TMS320C5X, an 8-bit constant can be specified as one of the operands in the single word instructions for addition, subtraction, AND, OR, XOR, etc.

2.7.2 Short Direct Addressing

This permits the lower order address of the operand of an instruction to be specified in the single word instruction. In the Tl TMS320 DSPs, the higher order 9 bits of the memory are stored in the data page pointer and only the lower 7 bits are specified as a part of the instruction. Each contiguous block of 128 words is referred to as one page in the Tl DSPs. The argument in the instruction specifies only the location within the current page. In the Motorola DSP5600X, short direct memory addressing permits a 6-bit address to be specified in the instruction.

2.7.3 Memory-mapped Addressing

The CPU registers and the I/O registers of the P-DSPs are also accessible as memory location. This is achieved by storing them in either the starting page or the final page of the memory space. For example, in TMS320C5X, page 0 corresponds to the CPU registers and I/O registers. In the case of Motorola DSP5600X, the last page of the memory space containing 64 locations is used as the memory map for the CPU and I/O registers. When these registers are accessed using memory mapped addressing modes, the higher address bits are not taken from the data page pointer and instead made to be 0 in the case of TI DSPs and made to be 1 in Motorola DSPs.

2.7.4 Indirect Addressing

In P-DSPs this addressing mode has a number of options. This permits an array of data to be processed in P-DSP to be efficiently fetched and stored. The address of the operands can be stored in one of the registers called *indirect address registers.* In the case of TI processors, the indirect address registers are called *auxiliary registers* ARs. Any of these registers can be updated when the operand fetched using these registers are being executed. This is made possible by having an additional ALU in the CPU core specifically for the indirect address registers or ARs. The ARs may be incremented or decremented either in steps of 1 or in steps specified by the content of an offset register. In the case of TI processors, the offset register is called an *INDX register.* In the P-DSPs from analog devices it is called the *modifier register.* The content of the indirect address registers may also be updated by a constant using bit reversed addressing mode explained in the next section. In the TI 5X processors the new address computed by the auxiliary ALU is not used for fetching the operand for the current instruction that is being decoded and is executed. It is used for fetching the operand that uses the indirect addressing mode next with this particular AR. For this reason, the indirect addressing mode used in TI 5X P-DSPs is called *indirect addressing mode* with post-increment/decrement. In Motorola DSP563XX, the updated indirect address register content may also be used to fetch the operand for the current instruction. Hence this mode is called the indirect addressing mode with pre-increment/ decrement. In TI TMS320C54X processors both post-increment/decrement and pre-increment/dec-rement operations are supported.

2.7.5 Bit Reversed Addressing Mode

The bit reversed number representation is explained in Section 1.14. The binary pattern corresponding to a particular decimal number is obtained by writing the natural binary equivalent of the number in the reverse order so that the most significant bit of the natural binary number becomes the least significant bit of the bit reversed no and vice versa.

For the computation of the FFT, the data is to be arranged in the bit reversed order and 2-point DFT of the resulting sequence is to be computed first. In the bit reversed addressing mode, when a 16-point FFT is to be computed, 2-point DFT of X(0) and X(8) is to be found. Similary 2-point DFT of X(4) and

X(12) and so on. It may be noted from Table 1.1 that the value 0, 8, 4, 12 corresponds to the consecutive numbers in the bit reversed number representation. In the bit reversed addressing mode, the address is incremented/decremented by the number represented in the bit reversed form.

2.7.6 Circular Addressing

In real time processing of signals, the input signal is continuously stored in the memory. The processed data is stored in another memory space continuously and may be written onto the output device. In this case input as well as output program will be simple. However, since the input as well as output memory space will be finite in size, the entire memory space would be exhausted after processing the input signal for some time, if the data is written into the memory by using linear addressing mode. One way to overcome this problem is to keep checking whether the range of either the input or the output memory space is exceeded. In that case, the new data is to be stored starting from the beginning of the particular memory space. However, checking this condition is an overhead that can be overcome using the circular addressing mode. In this mode, the memory can be organised as a circular buffer with the beginning memory address and the ending memory address corresponding to this buffer defined by the programmer. In the circular addressing mode, when the address pointer is incremented, the address will be checked with the ending memory address of the circular buffer. If it exceeds that, the address will be made equal to the beginning address of the circular buffer.

ON-CHIP PERIPHERALS 2.8

The P-DSPs have a number of on-chip peripherals that relieve the CPU from routine functions. Further they also help to reduce the chip count on the DSP system based around P-DSP. Some of the on-chip peripherals in the P-DSPs and their functions are as follows.

2.8.1 On-chip Timer

Two of the common applications of the timers are generation of periodic interrupts to the P-DSPs and generation of the sampling clocks for the A/D converters. The timer mode can be programmed by the P-DSPs. The timers can generate a single pulse or a periodic train of pulses. They can also generate a single square wave or a periodic square wave. The period of the timer is also made programmable.

2.8.2 Serial Port

This enables the data communication between the P-DSP and an external peripheral such as A/D converter, D/A converter or an RS232 C device. These ports normally have input and output buffers so that the P-DSP writes or reads from the serial port in parallel form and the serial port sends and receives data to the peripherals in serial form. They also generate interrupts when the serial port output buffer is empty or the input buffer is full, These devices have parallel to serial and serial to parallel converter inbuilt into them. The shift clock can be fed either from the P-DSP or an external device can supply it. The serial ports can operate either in the asynchronous mode or in the synchronous mode. In the asynchronous mode, the transmit data and receive data lines alone are used for communication and bit clock is transmitted from either end. In the case of synchronous mode, both bit clock and a frame sync signal that indicates the beginning of the first bit of the data transmitted using synchronous mode is transmitted from the serial port to the I/O device and also from I/O port to the serial port. Example, of the two signals with respect to the transmitted data is shown in Fig. 2.9.

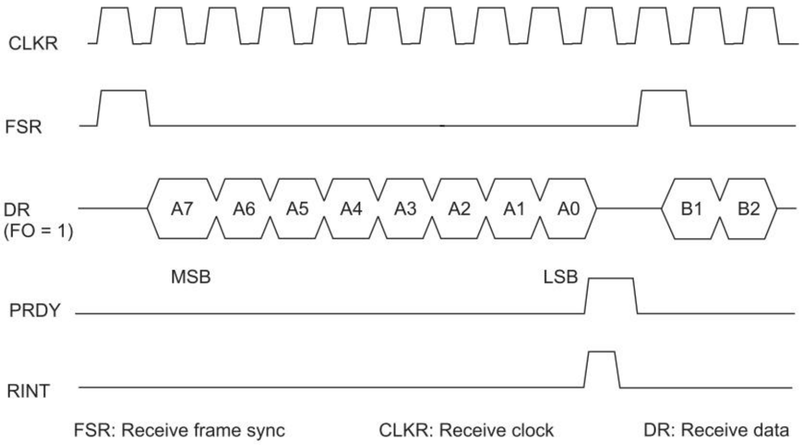


Fig. 2.9 *Burst mode serial port receive operation*

2.8.3 TDM Serial Port

The P-DSPs have a special serial port called TDM serial port. This permits a P-DSP to communicate with other devices or P-DSPs by using time division multiplexing (TDM). One of the devices can generate the frame sync pulse that indicates the beginning of a TDM frame and bit clock, the duration for which a bit is to be transmitted. As shown in Fig. 2.10 the TDM frame is split into a number of equal slots and each slot can be allotted for one of the devices.

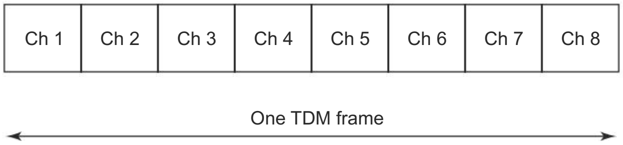


Fig. 2.10 *TDM frame with 8 time slots*

For example, in Fig. 2.10, there are 8 slots/frame and is referred to as a TDM with eight channels. In each of the slots, a number of bits may be transmitted by a channel. The TDM serial port normally uses four lines for the purpose of serial communication. They are

TFRM:	the frame sync signal
TClock:	the bit clock
TADD:	The address of the serial device that is outputting data in a particular TDM slot
TDAT:	The data transmitted into the TDM channel by the authorised device

The signals TADD and TDAT are bidirectional and are tristate controlled so that only one of the devices transmit the data and address in these lines at a time. Any one of the devices can generate the TFRM and clock signals and they are used by the other devices as a reference. A scheme where eight devices are interconnected using the TDM serial port is shown in Fig. 2.11. An example of how TI TMS320C5X can be configured to be one of the devices is shown in Fig. 2.12. An example, of each of the devices outputting a 16-bit data (D15 - DO) in its slot and also the address of the device (A0-A15), which is supposed to receive this data, is shown in Fig. 2.13.

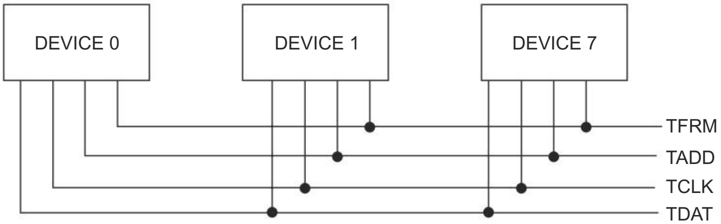


Fig. 2.11 *Interconnecting 8 devices using TDM serial using 4-biPbus*

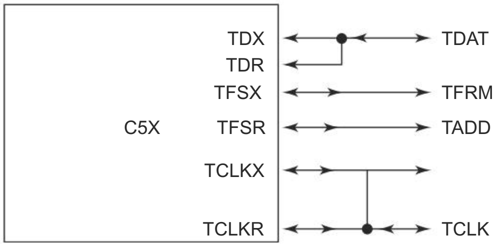


Fig. 2.12 *TMS320C5X configured to be one of TDM devices*

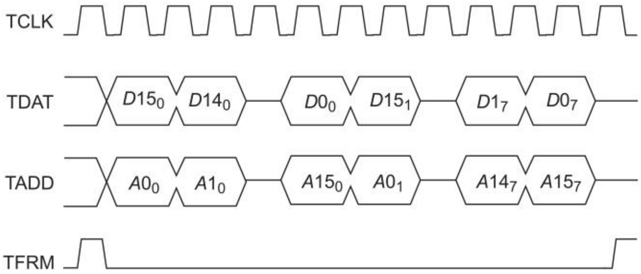


Fig. 2.13 *Data transfer using TDM channel*

2.8.4 Parallel Port

Parallel ports enable communication between the P-DSP and other devices to be faster compared to the serial communication by using a number of lines in parallel. In addition, they also have additional lines, which are for strobing or for handshaking purposes. The P-DSPs have two approaches for assigning lines for parallel port. In one approach used by the TI, the data bus itself is used for parallel ports. This is achieved by allocating a specific address space for I/O and whenever this address space is addressed using the I/O instructions, the parallel port signals including the handshaking signals are sent over the data bus. In another approach, separate lines are dedicated for parallel ports including the handshaking signals.

2.8.5 Bit I/O Ports

The P-DSPs have additional I/O ports that are single bit wide. These port bits may be individually set, reset or read. These bits are normally used for control purposes but they can also be used for data transfer. There are no handshaking signals for these I/O ports. Some of these bits are also used for conditional branching or calls. For example, in TI processors there are instructions such as branch if I/O zero.

2.8.6 Host Port

The P-DSPs also have a special parallel port normally 8-bit or 16-bit wide called the *host port* that enables them to communicate with a microprocessor or PC, which is called as a host. In addition to data communication, the host can generate interrupts and also cause the P-DSP to load a program from ROM to the RAM on reset. Almost all the P-DSPs including the ones from Analog devices, Motorola and TI have host ports.

2.8.7 Comm Ports

These are parallel ports that are used for interprocess communication between a number of identical P-DSP in a multiprocessor system. For example, a multiprocessor system may be built using a number of TI TMS320C4X. For the purpose of communication of the data between these processors six comm ports each of width 8 bits is provided. Since the data to be processed may be 32 or more number of bits, the P-DSPs have provision for splitting the data in streams of 8 bits and also assemble the 8 bits into words of 32 bits. Analog devices DSP ADSP 2106X has 6 comm ports each of which is 4 bits wide.

2.8.8 On-Chip A/D and D/A Converters

Some of the P-DSPs targeted towards voice applications such as cellular telephones and tapeless answering machines have A/D and D/A converters inside the P-DSP. For example, Motorola DSP 561XX and Analog devices ADSP 21MSP5X both have the A/D and D/A on chip and permit effective sampling rates of about 8 kHz.

2.8.9 P-DSPs with RISC and CISC

P-DSPs may be implemented using either the RISC processor or the CISC processor. For example, TI TMS320C6X P-DSPs uses RISC processor and a large number of P-DSPs from Analog devices, Motorola and TI make use of CISC. For example, TI TMS32054X and the Motorola DSP563XX and analog devices ADSP 2100X make use of CISC. TI TMS320C8X has a RISC and four P-DSPs with CISC in a single core. The relative advantages of each of these processors are as follows:

RISC Advantages

The chip area dedicated to the realisation of the control unit is considerably reduced because of the reduced number of instructions. About 20% of the chip area may be used for the control unit in RISC. In CISC processors about 30 - 40% of the chip area is used up for the control unit. Therefore in a RISC there is more area available for incorporating other features.

As a result of considerable reduction in the control area, the CPU registers and the data paths (processing units) can be replicated and the throughput of the processor can be increased by applying pipelining and parallel processing.

In a RISC, all the instructions are of uniform length and take the same time for execution. Hence the dummy periods or hold periods in the instruction pipeline is reduced to the minimum. This increases the computational speed.

A simpler and smaller control unit in RISC has fewer gates. This reduces the propagation delay and increases the speed. Reduced number of instructions, formats and addressing modes result in simpler and smaller decoder, which, in turn, increase the speed.

In RISC processors, the delayed branch and call instructions can be effectively used and they improve the speed.

HLL support Writing the programs in C and C++ relieves the programmer from learning the instruction set of a P-DSP and instead concentrate on the application. It increases the throughput of the programmer. Since RISC has a smaller number of instructions, the compiler for any HLL is shorter and simpler. The availability of a relatively large number of CPU registers permits a more efficient code optimisation by maximising the use of CPU registers over slower memories.

CISC Advantages

Some of the advantages of RISC also turn out to be disadvantages when viewed from a different perspective. The CISC processors have a very rich instruction set that even support high level language constructs similar to "if condition true then do", "for" and "while". The P-DSPs with CISC also have instructions specifically required for DSP applications such as MACD, FIRS, etc. This makes the application program written in the assembly language to be shorter and easy to follow. Since RISC has a smaller number of instructions, implementation of a single CISC instruction might require a number of instructions in RISC. This increases the memory required for storing the program and the traffic between CPU and memory is increased. This is on the one hand increases the computation time and on the other hand makes the program difficult to debug.

The HLL compilers are costly by several orders of magnitude compared to the P-DSPs themselves. For P-DSP with RISC architecture, compilers are essential. For most of the low cost applications, DSP platforms without the compilers are preferred. Hence a majority of P-DSPs are CISC based. The P-DSP manufacturers have tried to keep the codes for the new processors upward compatible with the older processors. This makes the learning curve steeper.

The relative disadvantages of each of these architectures are diminishing. By making the RISC processors applicable for larger and larger applications, the cost of the chip per se and the compiler costs are being brought down. The HLL compilers for the CISC processors are also becoming as efficient as hand assembly and the costs are coming down. Hence the distinction between the two in terms of cost and debugging efficiency is likely to narrow down further. The code composer studio from TI permits the programming in HLL as well as assembly language in a single development environment so that the best features of both the HLL and assembly language programming can be used by the programmer.

Review Questions

2.1 Explain why a MAC operation is implemented in hardware in programmable DSPs.

2.2 Explain how convolution is performed using a single MAC unit.

2.3 Explain the difference between a MAC instruction and MAC with data shift instruction. When is the latter instruction preferred?

2.4 Explain the difference between Von Neumann and Harvard architecture for the computer. Which architecture is preferred for DSP applications and why?

2.5 Explain why the P-DSPs have multiple address and data buses for internal memory and peripherals but have only a single address and data bus for the external memory and peripherals?

2.6 Explain the different techniques adopted for increasing the number of memory accesses/instruction cycle.

2.7 Explain how a higher throughput is obtained using the VLIW architecture. Give an example, of a DSP that has VLIW architecture.

2.8 Explain what is meant by instruction pipelining. Explain with an example, how pipelining increases the throughput efficiency.

2.9 Explain how delayed branch/call instructions are superior to the undelayed branch/call instructions.

2.10 Explain the memory mapped addressing mode used in P-DSPs.

2.11 What are the different ways in which the operand for instructions can be specified using indirect addressing mode.

2.12 What is meant by bit reversed addressing mode? What is the application for which this addressing mode is preferred?

2.13 What is meant by circular addressing mode? What is the application for which this addressing mode is preferred?

2.14 Mention some applications of on-chip timer in P-DSPs.

2.15 Distinguish between the synchronous and asynchronous mode of operation of serial ports.

2.16 Explain the operation of TDM serial ports in P-DSPs.

2.17 What is the use of host ports in P-DSPs? How do they differ from the comm ports?

2.18 List the relative merits and demerits of RISC and CISC processors.

Self Test Questions

2.1 The features in which PDSP is superior to advanced microprocessors is———.
(a) Low cost (b) Low power
(c) Computational speed (d) Real time I/O capability

2.2 In modified Harvard architecture for fetching the content of program and data memory, a separate bus is used for ——— memory and a single bus is used for ——— memory.

2.3 Number of memory accesses/clock /period that can be achieved using on chip DARAM of a P-DSP is ———.
(a) 1 (b) 2 (c) 3 (d) 4

2.4 VLIW architecture differs from conventional P-DSP in which of the following aspects:
(a) Instruction cache
(b) Number of functional units
(c) Use pipelining
(d) A single word fetched from memory has a number of instructions

2.5 A P-DSP has four pipeline stages and uses four phase clock. The number of clock cycles required for executing a program with 25 instruction is ———.
(a) 29 (b) 28 (c) 25 (d) 26

2.6 The number of instruction cycles required for executing a program in a microprocessor with no pipelining is ———.
(a) 1 (b) 2 (c) 3 (d) 4

2.7 The addressing mode that is convenient for FFT computation is———.
(a) Indirect addressing (b) Circular mode
(c) Bit reversed addressing (d) Memory mapped

2.8 The addressing that permits the content in internal register of the CPU & I/O to be accessed as memory location is———.
(a) Indirect addressing (b) Circular mode
(c) Bit reversed addressing (d) Memory mapped

2.9 The serial port that permits the data from a number of I/O devices to be sent using a single serial port is called———.
(a) Comm port
(b) Host port
(c) Time division multiplexing
(d) Bit I/O port

2.10 Which of the following characteristics are true for a RISC processor?
(a) Smaller control unit
(b) Small instruction set
(c) Short program length
(d) Less traffic between CPU & memory

3 ARCHITECTURE OF TMS320C5X

INTRODUCTION 3.1

Leading manufacturers of integrated circuits such as Texas Instruments (TI), Analog Devices and Motorola manufacture the digital signal processor (DSP) chips. These manufacturers have developed a range of DSP chips with varied complexity. The underlying concepts are broadly the same. Some of these concepts are discussed in Chapter 2. In order to give a feel for the design of systems with DSP chips, in this chapter, some details on the design of systems using the TMS320C5X DSP chip (denoted in brief as 5X) manufactured by TI are given.

The TMS320 DSP family consists of two types of single-chip DSPs: 16-bit fixed-point and 32-bit floating-point. These DSPs possess the operational flexibility of high-speed controllers and the numerical capability of array processors. Combining these two qualities, the TMS320 processors are inexpensive alternatives to custom fabricated VLSI and multichip bit-slice processors. TMS320C5X belongs to the fifth generation of the TI's TMS320 family of DSPs. The first five generations of TMS320 family are C1X, C2X, C3X, C4X and C5X. The C1X, C2X, C2XX and C5X are 16-bit fixed-point processors. Instruction sets of the higher generation fixed-point processors are upward compatible to the lower generation fixed-point processors. For example C5X can execute the instructions of both C1X and C2X. The 54X is upward compatible with 5X. C3X and C4X are 32-bit floating-point processors and C4X is upward compatible with C3X instruction set. The sixth generation C6X devices feature VelociTI™, an advanced very long instruction word (VLIW) architecture developed by TI and can execute 1600 MIPS. The eighth generation C8X devices, have, on a single piece of silicon, a number of advanced DSPs (ADSPs) and a RISC master processor. Typical application of the above families of TI DSPs are as follows:

C1X, C2X, C2XX, C5X, C54X: toys, hard disk drives, modems, cellular phones and active car suspensions

C3X: filters, analysers, hi-fi systems, voice mail, imaging, bar-code readers, motor control, 3D graphics or scientific processing

C4X: parallel-processing clusters in virtual reality, image recognition telecom routing, and parallel-processing systems.

C6X: wireless base stations, pooled modems, remote-access servers, digital subscriber loop systems, cable modems and multichannel telephone systems

C8X: video telephony, 3D computer graphics, virtual reality and a number of multimedia applications

The TI DSP chips have IC numbers with the prefix TMS320. If the next letter is C (e.g. TMS320C5X), it indicates that CMOS technology is used for the IC and the on-chip non-volatile memory is a ROM. If it is E (e.g. TMS320E5X) it indicates that the technology used is CMOS and the on-chip non-volatile memory is an EPROM. If it is neither (e.g. TMS3205X), it indicates that NMOS technology is used for the IC and the on-chip non-volatile memory is a ROM. Under C5X itself there are three processors, ‘C50, ‘C51 and ‘C5X, that have identical instruction set but have differences in the capacity of on-chip ROM and RAM. The characteristics of some of the TMS320 family DSP chips are given in Table 3.1.

The instruction set of TMS320C5X and other DSP chips is superior to the instruction set of conventional microprocessors such as 8085, Z80, etc., as most of the instructions require only a single cycle for execution. The multiply accumulate operation used quite frequently in signal processing applications such as convolution requires only one cycle in DSP.

Table 3.1 *Characteristics of some of the TMS320 family DSP chips*

	′C15	*′C25*	*′C30*	*′C50*	*′C541*
Cycle time (ns)	200	100	60	50	25
on chip RAM	4K	4K	4K	2K	5K
Total memory	4K	128K	16M	128K	128K
Parallel ports	8	16	16M	64K	64K

Architecture of TMS320C5X DSPs The block diagram of the internal architecture of C5X is shown in Fig. 3.1. The 320C5X DSPs are said to have advanced Harvard architecture because they have separate memory bus structures for program and data and have instructions that enable data transfer between the program and data memory area.

BUS STRUCTURE 3.2

Separate program and data buses allow simultaneous access to program instructions and data, providing a high degree of parallelism. For example, while data is multiplied, a previous product can be loaded into, added to or subtracted from the accumulator and, at the same time, a new address can be generated. Such parallelism supports a powerful set of arithmetic, logic and bit-manipulation operations that can all be performed in a single machine cycle. In addition, the ′C5X includes the control mechanisms to manage interrupts, repeated operations and function calling. The ′C5X architecture has four buses and their functions are as follows:

Program bus (PB) It carries the instruction code and immediate operands from program memory space to the CPU.

Program address bus (PAB) It provides addresses to program memory space for both reads and writes.

Data read bus (DB) It interconnects various elements of the CPU to data memory space.

Data read address bus (DAB) It provides the address to access the data memory space. The program and data buses can work together to transfer data from on-chip data memory and internal or external program memory to the multiplier for single-cycle multiply/accumulate operations.

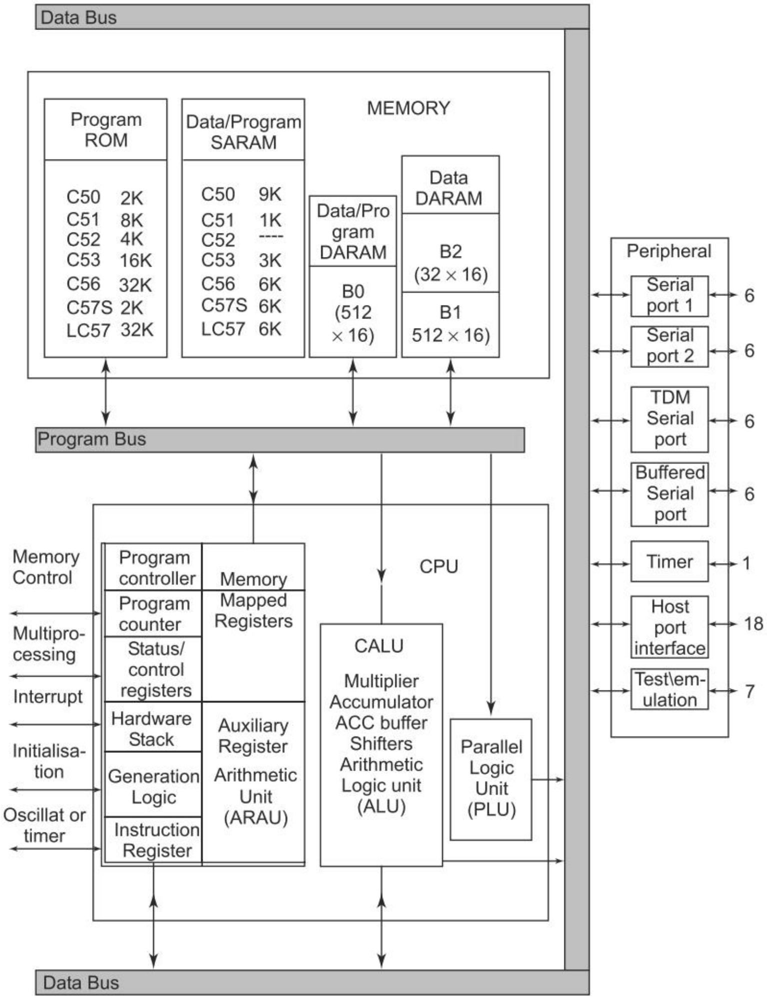


Fig. 3.1 *Internal architecture of C5X*

CPU registers (except STO and ST1), peripheral registers and I/O ports occupy data memory space. Some of the registers/execution units in the CPU of C5X DSP processors and their functions are as follows.

CENTRAL ARITHMETIC LOGIC UNIT (CALU) 3.3

It consists of the following elements: (16xl6)-bit parallel multiplier, arithmetic logic unit (ALU), accumulator (ACC), accumulator buffer (ACCB), product register (PREG) each with 32 bits and 0-16-bit left barrel shifter and right barrel shifter.

One of the operands for the ALU operation comes from ACC. The result of operations performed in central ALU are stored in ACC. Either the higher order word or lower order word of ACC can be loaded from memory. A 32-bit register denoted as ACCB is used for temporary storage of ACC. The hardware multiplier unit in the C5X processors performs 16 x 16 multiplication of numbers represented in 2's complement form. The 32-bit PREG holds the result of multiplication. The 16-bit temporary register 0 (TREG0) holds the multiplicand. The other operand for the multiplication can be specified using one of the addressing modes.

0-16-bit left barrel shifter and right barrel shifter in CALU permit the contents of memory to be left shifted by 0 to 16 bits before they are either fed to ALU or stored from ALU to memory. The CPU registers ACC and PREG can also be shifted using these shifters. In this case they require two cycles. A 5-bit register TREG1 specifies the number of bits by which the scaling shifter should shift either the incoming data to one of the CPU registers or vice versa. When the incoming data to CPU is left shifted by the scaling shifter the LSBs are filled with 0.

AUXILIARY REGISTER ALU (ARAU) 3.4

It consists of eight 16-bit auxiliary registers (ARs) AR0-AR7, a 3-bit auxiliary register pointer (ARP) and an unsigned 16-bit ALU. ARAU calculates indirect addresses by using inputs from ARs, 16-bit index register (INDX) and auxiliary register compare register (ARCR). The ARAU can autoindex the current AR while the data memory location is being addressed and can index either by ± 1 or by the contents of the INDX. As a result, accessing data does not require the CALU for address manipulation; therefore, the CALU is free for other operations in parallel. This makes the instructions to be executed faster compared to the conventional microprocessors. For example, let us consider the following sequence of 8085 instructions:

```
M0V A,M
INX H
```

These instructions enable the accumulator to be loaded using indirect addressing mode and HL register used as the address pointer is incremented. These two instructions can be replaced by a single 5X instruction LACC *+, 0.

Further, any one of the auxiliary registers can be used as the address pointer and incremented by the above instruction. The register that will be used is specified by the content of the ARP.

The auxiliary registers AR0-AR7 may also be used as the general purpose registers for holding the operands for arithmetic and logical operations in CALU. Some of the other registers of ARAU and their functions are as follows:

INDEX REGISTER (INDX) 3.5

The 16-bit INDX is used by the ARAU as a step value (addition or subtraction by more than 1) to modify the address in the ARs during indirect addressing. For example, when the ARAU steps across a row of a matrix, the indirect address is incremented by 1. However, when the ARAU steps down a column, the address is incremented by the dimension of the matrix. The ARAU can add or subtract the value stored in the INDX from the current AR as part of the indirect address operation. INDX can also map the dimension of the address block used for bit-reversal addressing.

AUXILIARY REGISTER COMPARE REGISTER (ARCR) 3.6

The 16-bit ARCR is used for address boundary comparison. The CMPR instruction compares the ARCR to the selected AR and places the result of the compare in the TC bit of ST1.

BLOCK MOVE ADDRESS REGISTER (BMAR) 3.7

The 16-bit BMAR holds an address value to be used with block moves and multiply/accumulate operations. This register provides the 16-bit address for an indirect-addressed second operand.

BLOCK REPEAT REGISTERS (RPTC, BRCR, PASR, PAER) 3.8

All these registers are 16-bit wide. Repeat counter register (RPTC) holds the repeat count in a repeat single-instruction operation and is loaded by the RPT and RPTZ instructions. Block repeat counter register (BRCR) holds the count value for the block repeat feature. This value is loaded before a block repeat operation is initiated. Block repeat program address start register (PASR) indicates the 16-bit address where the repeated block of code starts. The block repeat program address end register (PAER) indicates the 16-bit address where the repeated block of code ends. The PASR and PAER are loaded by the RPTB instruction.

PARALLEL LOGIC UNIT (PLU) 3.9

It performs Boolean operations or the bit manipulations required of high-speed controllers. The PLU can set, clear, test or toggle bits in a status register control register, or any data memory location. The PLU allows logic operations to be performed on data memory values directly without affecting the contents of the ACC or PREG. Results of a PLU function are written back to the original data memory location.

MEMORY-MAPPED REGISTERS 3.10

The ‘C5X has 96 registers mapped into page 0 of the data memory space. All ‘C5X DSPs have 28 CPU registers and 16 input/output (I/O) port registers but have different numbers of peripheral and reserved registers. Since the memory-mapped registers are a component of the data memory space, they can be written to and read from in the same way as any other data memory location. The memory-mapped registers are used for indirect data address pointers, temporary storage, CPU status and control, or integer arithmetic processing through the ARAU.

PROGRAM CONTROLLER 3.11

The program controller contains logic circuitry that decodes the instructions, manages the CPU pipeline, stores the status of CPU operations and decodes the conditional operations. Parallelism of architecture lets the ′C5X perform three concurrent memory operations in any given machine cycle: fetch an instruction, read an operand and write an operand. The program controller consists of the following elements:

16-bit program counter (PC)

16-bit status registers ST0, ST1, processor mode status register (PMST) and circular buffer control register (CBCR)

(8 x 16)-bit hardware stack

Address generation logic

Instruction register

Interrupt flag register and interrupt mask register

15 – 13	12	11	10	9	8 – 0
ARP	OV	OVM	1	INTM	DP

Fig. 3.2(a) *Status register 0 (ST0) bit assignment*

SOME FLAGS IN THE STATUS REGISTERS 3.12

The status registers can be stored into data memory and loaded from data memory, thereby allowing the 'C5X status to be saved and restored for subroutines. The ST0 and ST1 each have an associated 1-level deep shadow register stack for automatic context-saving when an interrupt trap is taken. These registers are automatically restored upon a return from interrupt.

The bit assignment details for ST0 and ST1 are given in Fig. 3.2. Significance of the various bits of ST0 and ST1 are as follows:

ARP (Auxiliary Register Pointer) These bits select the AR to be used in indirect addressing. When the ARP is loaded, the previous ARP value is copied to the auxiliary register buffer (ARB) in ST1.

OV (Overflow) flag bit This bit indicates that an arithmetic operation overflow in the ALU.

OVM (Overflow Mode) bit This bit enables/disables the accumulator overflow saturation mode in the ALU.

INTM (Interrupt Mode) bit This bit globally masks or enables all interrupts. The INTM bit has no effect on the non-maskable $\overline{\text{RS}}$ and $\overline{\text{NMI}}$ interrupts.

DP (Data Memory Page Pointer) bits These bits specify the address of the current data memory page. The DP bits are concatenated with the 7 LSBs of an instruction word to form a direct memory address of 16 bits.

15 – 13	12	11	10	9	8 – 7	6	5	4	3 – 2	1 – 0
ARB	CNF	TC	SXM	C	11	HM	1	XF	11	PM

Fig. 3.2(b) *Status register 1 (ST1) bit assignment*

ARB Auxiliary Register Buffer

This 3-bit field holds the previous value contained in the ARP in ST0. Whenever the ARP is loaded, the previous ARP value is copied to the ARB, except when using the LST #0 instruction. When the ARB is loaded using the LST #1 instruction, the same value is also copied to the ARP. This is useful when restoring context (when not using the automatic context save) in a subroutine that modifies the current ARP.

CNF On-chip RAM configuration control bit This 1-bit field enables the on-chip dual-access RAM block 0 (DARAM B0) to be addressable in data memory space or program memory space. The CNF bit can be modified by the LST #1 instruction. If CNF is 0, the on-chip DARAM block 0 is mapped into data memory space. The CNF bit can be cleared by a reset or the CLRC CNF instruction. When CNF is 1, the on-chip DARAM block 0 is mapped into program memory space. The CNF bit can be set by the SETC CNF instruction.

TC Test/control flag bit This 1-bit flag stores the results of the ALU or parallel logic unit (PLU) test bit operations. The status of the TC bit determines if the conditional branch, call and return instructions are to be executed.

SXM Sign-extension mode bit This 1-bit field enables/disables sign extension of an arithmetic operation. The SXM bit does not affect the operations of certain arithmetic or logical instructions; the ADDC, ADDS, SUBB or SUBS instruction suppresses sign extension, regardless of SXM.

C Carry bit This 1-bit field indicates an arithmetic operation carry or borrow in the ALU. The single-bit shift and rotate instructions affect the C bit.

HM Hold mode bit This 1-bit field determines whether the central processing unit (CPU) stops or continues execution when acknowledging an active $\overline{\text{HOLD}}$ signal.

XF pin status bit This 1-bit field determines the level of the external flag (XF) output pin.

PM Product shift mode bits This 2-bit field determines the product shifter (P-SCALER) mode and shift value for the PREG output into the ALU. Table 3.2 gives the PM bits and the function performed.

Table 3.2 *PM bits and the function performed*

PM bits	*Function*
b1 b0	P-SCALER mode for PREG output
0 0	No shift
0 1	Left-shifted 1 bit; LSB zero-filled
1 0	Left-shifted 4 bits; 4 LSBs zero-filled
1 1	Right-shifted 6 bits; sign extended; 6 LSBs lost. The product is always sign extended, regardless of the value of the SXM bit

ON-CHIP MEMORY 3.13

The ′C5X architecture contains a considerable amount of on-chip memory to aid in system performance and integration:
Program Read-Only Memory (ROM)
Data/Program Dual-Access RAM (DARAM)
Data/Program Single-Access RAM (SARAM)

The ′C5X has a total address range of 224K words x 16 bits. The memory space is divided into four individually selectable memory segments: 64K-word program memory space, 64K-word local data memory space, 64K-word I/O ports and 32K-word global data memory space.

3.13.1 Program ROM

All ‘C5X DSPs carry a 16-bit on-chip maskable programmable ROM (see Fig. 3.1 for sizes). Some of the ‘C5X DSPs have boot loader code resident in the on-chip ROM, and the other ′C5X DSPs offer the boot loader code as an option. This memory is used for booting program code from slower external ROM or EPROM to fast on-chip or external RAM. Once the custom program has been booted into RAM, the boot ROM space can be removed from program memory space by setting the MP/$\overline{MC}$ bit in the processor mode status register (PMST). The on-chip ROM is selected at reset by driving the MP/$\overline{MC}$ pin low. If the on-chip ROM is not selected, the ‘C5X devices start execution from off-chip memory.

3.13.2 Data/Program Dual-Access RAM

All ′C5X DSPs carry a 1056-word x 16-bit on-chip dual-access RAM (DARAM). The DARAM is divided into three individually selectable memory blocks: 512-word data or program DARAM block B0, 512-word data DARAM block B1 and 32-word data DARAM block B2. The DARAM is primarily intended to store data values but, when needed, can be used to store programs as well. DARAM blocks B1 and B2 are always configured as data memory; however. DARAM block B0 can be configured by software as data or program memory.

DARAM improves the operational speed of the ‘C5X CPU. The CPU operates with a 4-deep pipeline. In this pipeline, the CPU reads data on the third stage and writes data on the fourth stage. Hence, for a given instruction sequence, the second instruction could be reading data at the same time the first instruction is writing data. The dual data buses (DB and DAB) allow the CPU to read from and write to DARAM in the same machine cycle.

3.13.3 Data/Program Single-Access RAM

Almost all ′C5X DSPs carry a 16-bit on-chip single-access RAM (SARAM) of sizes varying from 1-9K (16–bits) words. Code can be booted from an off-chip ROM and then executed at full speed once it is loaded into the on-chip SARAM. The SARAM can be configured by software as data memory, as program memory or combination of both data memory and program memory. The SARAM is divided into 1K- and/or 2K-word blocks contiguous in address memory space. All ′C5X CPUs support parallel accesses to these SARAM blocks. However, one SARAM block can be accessed only once per machine cycle. In other words, the CPU can read from or write to one SARAM block while accessing another SARAM block.

3.13.4 On-Chip Memory Protection

The ′C5X DSPs have a maskable option that protects the contents of on-chip memories. When the related bit is set, no externally originating instruction can access the on-chip memory spaces.

ON-CHIP PERIPHERALS 3.14

All ′C5X DSPs have the same CPU structure; however, they have different on-chip peripherals connected to their CPUs. The ‘C5X DSP on-chip peripherals available are as follows:

Clock Generator
Hardware Timer
Software-Programmable Wait-State Generators
Parallel I/O Ports
Host Port Interface (HPI)
Serial Port
Buffered Serial Port (BSP)
Time-Division Multiplexed (TDM) Serial Port
User-Maskable Interrupts

3.14.1 Clock Generator

The clock generator consists of an internal oscillator and a phaselocked loop (PLL) circuit. The clock generator can be driven internally by a crystal resonator circuit or driven externally by a clock source.

The PLL circuit can generate an internal CPU clock by multiplying the clock source by a specific factor and so a clock source with a frequency lower than that of the CPU can be used.

3.14.2 Hardware Timer

A 16-bit hardware timer with a 4-bit prescaler is available. This programmable timer clocks at a rate that is between 1/2 and 1/32 of the machine cycle rate (CLKOUT1), depending upon the timer's divide-down ratio. The timer can be stopped, restarted, reset or disabled by specific status bits. Three registers control and operate the timer. The timer counter register (TIM) gives the current count of the timer. The timer period register (PRD) defines the period for the timer. The 16-bit timer control register (TCR) controls the operations of the timer.

3.14.3 Software-Programmable Wait-State Generators

Software-programmable wait-state logic is incorporated in ‘C5X DSPs allowing wait-state generation without any external hardware for interfacing with slower off-chip memory and I/O devices. This feature consists of multiple wait-state generating circuits. Each circuit is user-programmable to operate in different wait states for off-chip memory accesses.

3.14.4 Parallel I/O Ports

A total of 64K I/O ports are available, 16 of these ports are memory-mapped in data memory space. Each of the I/O ports can be addressed by the IN or the OUT instruction. The memory-mapped I/O ports can be accessed with any instruction that reads from or writes to data memory. The $\overline{\text{IS}}$ signal indicates a read or write operation through an I/O port. The ′C5X can easily interface with external I/O devices through the I/O ports while requiring minimal off-chip address decoding circuits.

3.14.5 Host Port Interface (HPI)

The HPI is available on the ′C57S and ′LC57. It is an 8-bit parallel I/O port that provides an interface to a host processor. Information is exchanged between the DSP and the host processor through on-chip memory that is accessible to both the host processor and the ‘C57.

3.14.6 Serial Port

Three different kinds of serial ports are available: a general-purpose serial port, a time-division multiplexed (TDM) serial port and a buffered serial port (BSP). Each ′C5X contains at least one general-purpose, high-speed synchronous, full-duplexed serial port interface that provides direct communication with serial devices such as codecs, serial analog-to-digital (A/D) converters and other serial systems. The serial port is capable of operating at up to one-fourth the machine cycle rate (CLKOUT1). The serial port transmitter and receiver are double-buffered and individually controlled by maskable external interrupt signals. Data is framed either as bytes or as words.

Five 16-bit registers (SPC, DRR, DXR, XSR, RSR) control and operate the serial port interface. The serial port control (SPC) register contains the mode control and status bits of the serial port. The data receive register (DRR) holds the incoming serial data, and the data transmit register (DXR) holds the outgoing serial data. The data transmit shift register (XSR) controls the shifting of the data from the DXR to the output pin. The data receive shift register (RSR) controls the storing of the data from the input pin to the DRR.

3.14.7 Buffered Serial Port (BSP)

The BSP is available on the ′C56 and ′C57 devices. It is a full-duplexed, double-buffered serial port and an autobuffering unit (ABU). The BSP provides flexibility on the data stream length. The ABU supports high-speed data transfer and reduces interrupt latencies. The BSP has a 2K-word buffer, which resides in the ‘C5X internal memory. Five BSP registers control and operate the BSP.

3.14.8 TDM Serial Port

The TDM serial port available on the ‘C50, ‘C51 and ‘C53 devices is a full-duplexed serial port that can be configured by software either for synchronous operations or for time-division multiplexed operations. The TDM serial port is commonly used in multiprocessor applications.

3.14.9 User-Maskable Interrupts

Four external interrupt lines ($\overline{\text{INT1}}$ – $\overline{\text{INT4}}$) and five internal interrupts, a timer interrupt and four serial port interrupts are user maskable. When an interrupt service routine (ISR) is executed, the contents of the program counter are saved on an 8-level hardware stack, and the contents of 11 specific CPU registers, ACC, ACCB, PREG, ST0, ST1, PMST, TREG0, TREG1, TREG2, INDX and ARCR, are saved in one deep stack (shadow registers). When a return from interrupt instruction is executed, the CPU registers’ contents are restored.

Review Questions

3.1 Mention few applications of each of the families of TI DSPs

3.2 What are the different buses of TMS320C5X and their functions?

3.3 List the functional units in CALU of 5X and explain the source and destination of operands of each of these units.

3.4 List the various registers used with the ARAU and their functions.

3.5 What is meant by memory mapped register? How is it different from a memory?

3.6 List status register bits of 5X and their functions.

3.7 Distinguish between the dual-access RAM and single-access RAM used in the on-chip memory of 5X.

3.8 List the on-chip peripherals in 5X and their functions.

3.9 What are the various interrupt types supported by 5X?

3.10 Draw the internal architecture diagram of 5X and indicate the various blocks.

Self Test Questions

3.1 The 320C5X DSPs are said to have advanced Harvard architecture because

(a) they have separate memory bus structures for program and data

(b) they have instructions that enable data transfer between the program and data memory area

(c) they have same memory bus structures for program and data

(d) the contents of program memory cannot into the data memory or vice versa

3.2 The central ALU of C5X DSP processors have ——— bit ALU and one of the operands for the ALU operation comes from ———.

(a) 32,ACC (b) 16,ACC (c) 32,ACCB (d) 16,ACCB

3.3 The result of operations performed in central ALU are stored in ———.

(a) ACC (b) ACCB (c) TREG0 (d) PREG

3.4 The ALU register whose either higher order word or lower order word can be loaded from memory is.

(a) ACC (b) ACCB (c) TREG0 (d) PREG

3.5 The ——— bit register used for temporary storage of accumulator is ———.
(a) 32, PREG (b) 32, ACCB (c) 16, TREG0, (d) 32, ACC

3.6 The ——— permits execution of logical operations on data without affecting the contents of ACC.
(a) parallel logic unit (b) auxiliary ALU
(c) central ALU

3.7 The hardware multiplier unit in the C5X processors perform multiplication of ——— times ——— bit represented in ——— complement form.
(a) 16, 16, 1s (b) 8,8 1s (c) 16, 16, 2s (d) 8, 8, 2s

3.8 ——— holds the result of multiplication and is ——— bit wide.
(a) PREG, 32 (b) PREG, 16
(c) TREG0, 16 (d) TREG0, 32

3.9 The register in which the multiplicand is stored before multiplication is performed is ——— and is ——— bit wide.
(a) PREG, 32 (b) PREG, 16
(c) TREG0, 16 (d) TREG0, 32

3.10 ——— permits the contents of memory to be left shifted by 0-16 bits before they are either fed to ALU or stored from ALU to memory.
(a) Scaling shifter (b) ALU
(c) PLU (d) Auxiliary ALU

3.11 The register that specifies the number of bits by which the scaling shifter should shift either the incoming data to one of the CPU registers or vice versa is ——— and is ——— bit wide.
(a)TREG1,4 (b) TREG1, 5 (c) TREG2, 5 (d) TREG2, 4

3.12 When the incoming data to CPU is left shifted by the scaling shifter the LSBs are filled with ———
(a) 0 (b) 1 (c) LSB before shifting

3.13 The bit of status register ST1, which determines whether the MSBs of the bits left shifted by the scaling shifter is zero, are sign extended is ———.
(a) SXM (b) TC (c) OV (d) OVM

3.14 In the hardware stack of C5X processors ——— bit numbers can be stored.
(a) 16, 16 (b) 16, 8 (c) 8, 8 (d) 8, 16

3.15 The bit of status register 0 (ST0) that becomes 1 if overflow occurs from an ALU operation is ———
(a) SXM (b) OV (c) OVM (d) TC (e) C

3.16 The bit of ST0 that determines whether the ACC is replaced with either largest positive or negative number or left unmodified is ———.
(a) SXM (b) OV (c) OVM (d) TC (e) C

3.17 The bit of ST1 that is used for testing whether a particular memory is zero or not or for comparing one register against another register memory is ———.
(a) SXM (b) OV (c) OVM (d) TC (e) C

3.18 The bit of ST1 that becomes 1 if either addition generates a carry or subtraction results in borrow is ———.
(a) SXM (b) OV (c) OVM (d) TC (e) C

3.19 The status register bit that determines whether multiplier's 32-bit product is left shifted by 0, 1, 4 or right shifted by 6 with sign extension before it is transferred/added to the ACC is ———.
(a) PM (b) CNF (c) HM (d) XF
(e) INTM

3.20 The RAM configuration control bit that indicates whether the on-chip reconfigurable dual-access RAM is mapped to data space or program space is ———.
(a) PM (b) CNF (c) HM (d) XF
(e) INTM

3.21 The bit of status register that determines whether the processor halts the internal operation while acknowledging a hold or not is ———.
(a) PM (b) CNF (c) HM (d) XF
(e) INTM

3.22 The ——— bit of the status register indicates the status of the general purpose output pin.
(a) PM (b) CNF (c) HM (d) XF
(e) INTM

3.23 The pointers that are contained in the status register 0 are ———.
(a) ARP (b) DP (c) ARB (d) IPTR
(e) INTM

3.24 The pointers that are contained in the status register 1 are ———.
(a) ARP (b) DP (c) ARB (d) IPTR
(e) INTM

3.25 If ——— bit is set to 0, all unmasked interrupts are enabled. Otherwise all the maskable interrupts are disabled.
(a) ARP (b) DP (c) ARB (d) IPTR
(e) INTM

4 TMS320C5X ASSEMBLY LANGUAGE INSTRUCTIONS

The ′C5X instruction set supports numerically intensive signal processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. The C5X instruction set is a superset of the ′C1X and ′C2X instruction sets and is source-code upward compatible with both devices.

ASSEMBLY LANGUAGE SYNTAX 4.1

For programming in the assembly language the C5X assembler assumes the following assembly language syntax. A source statement can contain four ordered fields. The general syntax for source statements is as follows:

[label] [:] mnemonic [operand list][;comment]

They in turn follow these guidelines:

- All statements must begin with a label, a blank, an asterisk or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- Labels may be placed either before the instruction mnemonic on the same line or on the preceding line in the first column.
- One or more blanks must separate each field. Tab characters are equivalent to blanks.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column must begin with a semicolon.

The following types of operands are permitted:

0 ≤ dma ≤ 127	dma: Data Memory Address
0 ≤ pma ≤ 65535	pma: Program Memory Address
0 ≤ shift ≤ 15	
0 ≤ shift2 ≤ 7	
0 ≤ *n* ≤ 7	***n***:AR no.
0 ≤ *k* ≤ 255	***k***: Short Constant
0 ≤ *lk* ≤ 65535	*lk*: Long Constant

ind: {**+*–*0+ *0– *BR0+ *BR0–}

Operands can be constants or assembly-time expressions that refer to memory, I/O ports, register addresses, pointers, shift counts and a variety of other constants.

The complete list of the mnemonics of the various instructions supported by 5X and a brief description of each of these instructions are given in Appendix A4. More detailed explanation of these instructions with examples is given in the TMS30C5X user reference manual. In this chapter, in Section 4.2 the various addressing modes supported by C5X are discussed. In Sections 4.3-4.5 some of the most commonly used C5X instructions are explained individually with examples. In chapter 6 application programs which make use of the instructions explained in the above sections are presented.

ADDRESSING MODES 4.2

C5X processors can address 64K words of program memory and 96K words of data memory. C5X supports the following six addressing modes:

Direct addressing
Memory-mapped register addressing
Indirect addressing
Immediate addressing
Dedicated-register addressing
Circular addressing

The details of each of these addressing modes are considered next.

4.2.1 Direct Addressing

The data memory used with C5X processors is split into 512 pages each of 128 words long. The data memory page pointer (DP) in ST0 holds the address of the current data memory page. In the direct addressing mode of C5X, only lower-order 7 bits of the address are specified in the instruction. The upper 9 bits are taken from the DP as shown in Fig. 4.1.

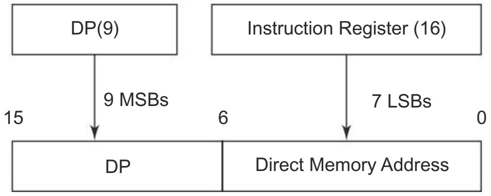


Fig. 4.1 *16-bit data memory address bus (DAB)*

4.2.2 Memory-Mapped Register Addressing

The RAM area in page 0 is used for storing some of the registers, interrupt vector addresses and so on. These locations can be accessed by specifying the actual address or by the register name, (e.g., the AR0 can either be denoted by the actual memory location (10h) used for storing its value or by the symbol AR0). Since these memory locations can be interchangeably used with the register names, the registers corresponding to page 0 are referred to as *memory-mapped registers* (MMRs).

With memory-mapped register addressing, the MMRs can be modified without affecting the current data page pointer value. In addition, any scratch pad RAM (DARAM B2) location or data page 0 can also be modified. The memory-mapped register addressing mode operates like the direct addressing mode, except that the 9 MSBs of the address are forced to 0 instead of being loaded with the contents of the DP. This allows the memory-mapped registers of data page 0 to be modified directly without the overhead of changing the DP or auxiliary register. The following instructions operate in the memory-mapped register addressing mode. Using these instructions does not affect the contents of the DP:

LAMM—Load accumulator with memory-mapped register
LMMR—Load memory-mapped register
SAMM—Store accumulator in memory-mapped register
SMMR—Store memory-mapped register

Example 4.1 The instruction LMMR AR0, #1500h loads AR0 with the content of the location 1500h as shown in Fig. 4.2. Let the content of AR0 and the data memory location 1500h be 2345h and 6789h, respectively, before executing the instruction. After executing the instruction their contents become 6789h and 6789h.

Before execution of LMMR AR0, # 1500 h after execution.

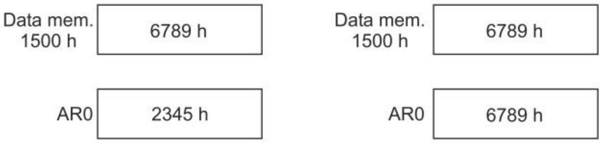


Fig. 4.2 *Memory-mapped register addressing example 4.1*

Before execution of SMMR ARO, #1500h After execution.

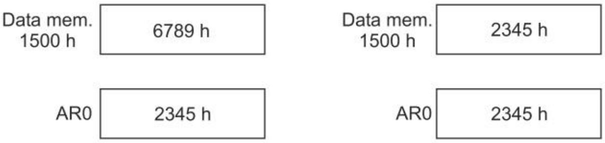


Fig. 4.3 *Memory-mapped register addressing example 4.2*

The SMMR does the reverse operation.

Example 4.2 Let the content of AR0 and the data memory location be 2345h and 6789h, respectively, before executing the instruction SMMR AR0, # 1500h. After executing the instruction their contents become 2345h and 2345h as shown in Fig. 4.3.

LAMM * loads lower 16 bits of ACC, i.e., ACCL from the location pointed by the lower-order 7 bits of the current AR. The higher 16 bits ACCH is filled with 0.

Example 4.3 Let the content of ARP, AR1, ACC, the value of data memory locations 25h and 825h be as shown in Fig. 4.4. After execution of the LAMM * instructions, the register contents are as shown on the right hand column in Fig. 4.4. It can be seen that the value in data memory location 25h is loaded into ACC. 25h corresponds to the lower-order 7 bits of AR1 and the higher bits of PAB are made to be 0 as the MMR corresponds to page 0.

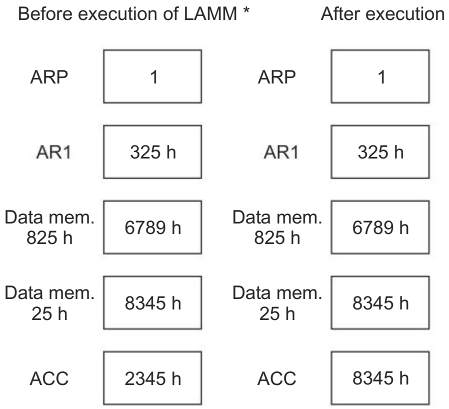


Fig. 4.4 *Memory-mapped register addressing example 4.3*

4.2.3 Immediate Addressing

The immediate addressing mode can be used to load either a 16-bit constant or a constant of length 13, 9 or 7. Accordingly it is referred to as long immediate or short immediate addressing mode. This mode is indicated by the symbol #. For e.g., ADD # 56h adds 56h to ACC. Similarly ADD # 4567h adds 4567h to ACC.

4.2.4 Indirect Addressing

The ARs AR0-AR7 are used for accessing data, using indirect addressing mode. In the indirect addressing mode, out of the eight ARs the one which is currently used for accessing data is denoted by the register ARP. The contents of ARP can be temporarily stored in the ARB register. The indirect addressing mode of C5X permits the AR used for the addressing to be updated automatically either after or before the operand is fetched. Hence a separate instruction is not required to update the AR. However, if required, the contents of an AR can be incremented or decremented by any 8-bit constant using SBRK and ADRK instructions, (e.g., SBRK #k, ADRK #k subtracts, adds the constant k from/ to the AR pointed by ARP).

In the indirect addressing mode, the manner in which the memory address is computed and the manner in which the AR is altered after the instruction depends on the instruction. This is indicated to the assembler by the symbols *, *+, *–,*0+ , *0–, *BR0+ and *BR0–. The symbol used to indicate the indirect addressing mode and the action taken after executing the instruction are given in Table 4.1.

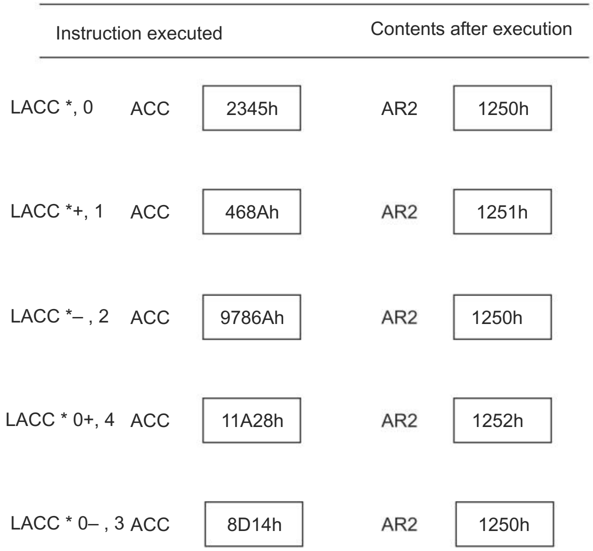
Table 4.1 *Various options in the indirect addressing mode of 5X*

Symbol	*Value of AR pointed by ARP after instruction execution*
*	AR unaltered
*+	AR incremented by 1
*–	AR decremented by 1
*0+	AR incremented by the content of INDX
*0–	AR decremented by the content of INDX
*BR0+	AR incremented by the content of INDX with reverse carry propagation
*BR0–	AR decremented by the content of INDX with reverse carry propagation

The details on addition/subtraction of INDX with AR with reverse carry propagation is given in Section 4.2.5.

Example 4.4 Let the value of ARP, AR2 and INDX register be 2, 1250h and 2h, respectively, and the content of the data memory location 1240h–1260h be filled with the data 2345h. Let SXM be 0. The value of ACC and AR2 after the following sequence of LACC (load accumulator with shift) instructions are executed is shown in Fig. 4.5.

```
LACC *, 0
LACC *+, 1
LACC *–, 2
LACC *0+, 4
LACC *0–, 3
```



Instruction executed	Contents after execution			
LACC *, 0	ACC	2345h	AR2	1250h
LACC *+, 1	ACC	468Ah	AR2	1251h
LACC *– , 2	ACC	9786Ah	AR2	1250h
LACC * 0+, 4	ACC	11A28h	AR2	1252h
LACC * 0– , 3	ACC	8D14h	AR2	1250h

Fig. 4.5 *Contents of ACC and AR after execution of program in example 4.4*

4.2.5 Bit-Reversed Addressing

In the bit-reversed addressing mode, INDX specifies one-half the size of the FFT. The value contained in the current AR must be equal to 2^{n-1}, where n is an integer, and the FFT size is 2^n. An AR points to the physical location of a data value. When INDX is added to the current AR, using bit-reversed addressing, addresses are generated in a bit-reversed fashion.

Example 4.5 Assume that the ARs are eight bits long, that AR2 represents the base address of the data in memory ($0110\ 0000_2$) and that INDX contains the value $0000\ 1000_2$. When the MAC (Multiply Accumulate) instruction MAC 0FF00h, *BR0+ is repeatedly executed eight times, the value of AR2 is modified as given in Table 4.2.

Table 4.2 *Bit-reversed addresses example 4.5*

Instruction executed	*Value of AR2*
MAC 0FF00h,*BR0+ ;	AR2 = 0110 0000 (0th value)
MAC 0FF00h,*BR0+ ,	AR2 = 0110 1000 (1st value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 0100 (2nd value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 1100 (3rd value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 0010 (4th value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 1010 (5th value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 0110 (6th value)
MAC 0FF00h,*BR0+ ;	AR2 = 0110 1110 (7th value)

4.2.6 Immediate Addressing

In immediate addressing, the instruction word(s) contains the value of the immediate operand. The ‘C5X has both 1-word (8-bit, 9-bit and 13-bit constant) short immediate instructions and 2-word (16-bit constant) long immediate instructions. Table 4.3 lists the instructions that support immediate addressing.

Table 4.3 *Instructions that support immediate addressing*

	Short Immediate (1-word)		
8-bit constant	*9-bit constant*	*13-bit constant*	
ADD	LDP	MPY	
ADRK			
LACL			
LAR			
RPT			
SBRK			
SUB			
	Long immediate (2-word) 16-bit constant		
ADD	AND	APL	CPL
LACC	LAR	MPY	OPL
OR	RPT	RPTZ	
SPLK	SUB	XOR	XPL

4.2.6.1 Short Immediate Addressing

In short immediate instructions, the operand is contained within the instruction machine code.

Example 4.6

ADD#0FFh

In this example, the lower 8 bits are the operand and are added to the ACC by the CALU.

4.2.6.2 Long Immediate Addressing

In long immediate instructions, the operand is contained in the second word of a 2-word instruction There are two long immediate addressing modes: one-operand instructions and two-operand instructions.

4.2.6.3 Long Immediate Addressing with Single/No Data Memory Access

Example 4.7

ADD #1234h

In this example, the second word (1234h) of the 2-word instruction is added to the ACC by the CALU.

4.2.6.4 Long Immediate Addressing with Dual Data Memory Access

The long immediate addressing also could apply for a second data memory access for the execution of the instruction.

Example 4.8

BLDD #2345h, 012h

In this example, the source address (operand1) is fetched via PAB, and the destination address (operand2) uses the direct addressing mode. Bits 15 through 8 of machine code1 contain the opcode. Bit 7, with a value of 0, defines the addressing mode as direct, and bits 6 through 0 contain the dma

4.2.7 Dedicated-Register Addressing

The dedicated-registered addressing mode operates like the long immediate addressing mode, except that the address comes from one of two special-purpose memory-mapped registers in the CPU: the block move address register (BMAR) and the dynamic bit manipulation register (DBMR) The advantage of this addressing mode is that the address of the block of memory to be acted upon can be changed during execution of the program.

The syntax for dedicated-register addressing can be stated in one of two ways:

Specify BMAR by its predefined symbol:

Example 4.9

BLDD BMAR, DAT100; DP = 0

If BMAR contains the value 200h, then the content of data memory location 200h is copied to data memory location 100 on the current data page.

Exclude the immediate value from a parallel logic unit (PLU) instruction:

Example 4.10

OPL DAT10; DP = 6

If DBMR contains the value 00FF0h and the address 030Ah contains the value 01h, then the content of data memory location 030Ah is ORed with the content of the DBMR. The resulting value 0FFF is stored back in memory location 030Ah.

4.2.8 Circular Addressing

Many algorithms such as convolution, correlation and finite impulse response (FIR) filters can use circular buffers in memory to implement a sliding window, which contains the most recent data to be processed. The ′C5X supports two concurrent circular buffers operating via the ARs. The following five memory-mapped registers control the circular buffer operation:

CBSR1—Circular buffer 1 start register
CBSR2—Circular buffer 2 start register
CBER1—Circular buffer 1 end register
CBER2—Circular buffer 2 end register
CBCR—Circular buffer control register

The 8-bit CBCR enables and disables the circular buffer operation. To define circular buffers, the start and end addresses are loaded into the corresponding buffer registers first; next, a value between the start and end registers for the circular buffer is loaded into an AR. The corresponding circular buffer enable bit in the CBCR should be set.

LOAD/STORE INSTRUCTIONS 4.3

Mnemonics and brief description of some of the load and store instructions of C5X are as follows:

LACB	Load ACC To ACCB
LACC	Load data memory value, with left shift, to ACC
	Load long immediate, with left shift, to ACC
	Load data memory value, with left shift of 16, To ACC
LACL	Load data memory value to ACCL; zero ACCH
	Load short immediate to ACCL; zero ACCH
LACT	Load data memory value, with left shift specified by TREG1, to ACC
LAMM	Load contents of memory-mapped register to ACCL; zero ACCH
SACB	Store ACC in ACCB
SACH	Store ACCH, with left shift, in data memory location
SACL	Store ACCL, with left shift, in data memory location
SAMM	Store ACCL in memory-mapped register
LAR	Load data memory value to ARX
SAR	Store ARX in data memory location
LDP	Load data memory value to DP bits
MAR	Modify AR
SPLK	Store long immediate in data memory location
LPH	Load data memory value to PREG high byte
LT	Load data memory value to TREG0
PAC	Load PREG, with shift specified by PM bits, to ACC
SPH	Store PREG high byte, with shift specified by PM bits, in data memory location
SPL	Store PREG low byte, with shift specified by PM bits, in data memory location
LST	Load data memory value to ST0
	Load data memory value to ST1
SST	Store ST0 in data memory location
	Store ST1 in data memory location

The instructions LACC, SAMM, LAMM, SMMR and LMMR have already been explained in Section 4.2.

The ARP can be loaded either using the MAR instruction or LST # 0, LST #1 instruction as follows:

When ST0 is loaded using the LST #0, it modifies the content of ARP but leaves the ARB in STl unaltered. On the other hand, if ARB is modified using LST #1, it makes ARP to be equal to ARB.

Example 4.11 Let the value of DP and the content of dma 400h be 7 and 3E00h, respectively. After executing the LST #0, 00h instruction, the contents of ST0 and ST1 are as shown in Fig. 4.6.

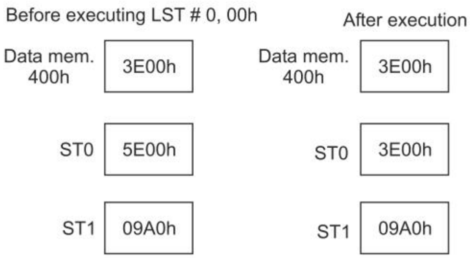


Fig. 4.6 *Loading ARP using LST #0 instruction*

Example 4.12 Let the value of DP and the content of dma 400h be 7 and 0580h, respectively. After executing the LST #1, 00h instruction, the contents of ST0 and ST1 are as shown in Fig. 4.7.

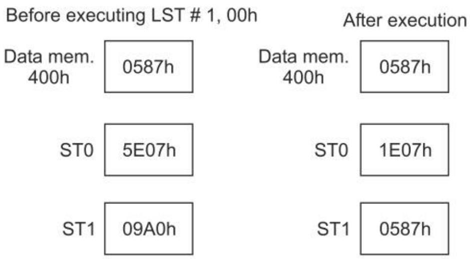


Fig. 4.7 *Loading ARB using LST #0 instruction*

ST0 and STl can also be loaded using indirect addressing mode. In this case if the next ARP value is specified in the instruction as an argument, it is ignored. The value of ARP is determined only by the value loaded into ST0/ST1.

Example 4.13 Let the value of ARP be 2 and the content of AR2, 0400h be 0400h, 0587h respectively. After executing the LST #1, *, AR3 instruction, the contents of ST0 and ST1 are as shown in Fig. 4.8.

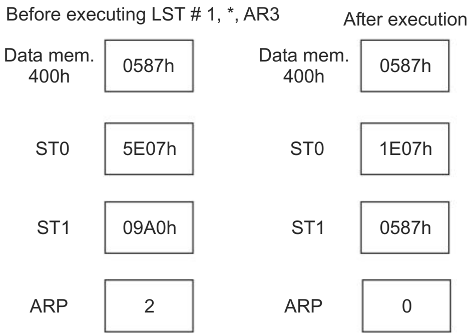


Fig. 4.8 *Loading ARP using indirect addressing*

The ARP can be modified using the MAR *+, ARn instruction. In this case, content of the current AR is incremented and the new value of ARP becomes *n*.

Example 4.14 Let the present value of ARP be 5. When the Instruction MAR *+, AR3 executed, the value of ARP becomes 3. The content of AR5 is incremented.

ADDITION/SUBTRACTION INSTRUCTIONS 4.4

In the addition/subtraction instructions of C5X, one of the operand is ACC. The other operand can be PREG, ACCB or the content of memory fetched using one of the addressing modes. For the ADD and SUB instructions alone, the number fetched from memory, using dma, indirect addressing and immediate addressing with long constant, can be shifted left in the scaling shifter by 0-16 before performing the required operation.

Example 4.15 Let the initial content of ACC be 1234h. After executing the instruction ADD #2345h, 2, the content of ACC is as shown in Fig. 4.9.

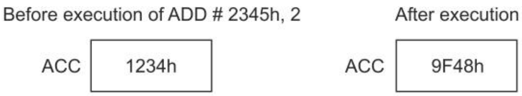


Fig. 4.9 *Addition using immediate addressing, example 4.14*

In this case the data 2345h is left shifted by two positions before it is added to ACC.

In the case of indirect addressing mode, for the ADD/SUB instructions, the third arguement (if it is present) denotes the new value to which ARP is to be updated after executing the instruction.

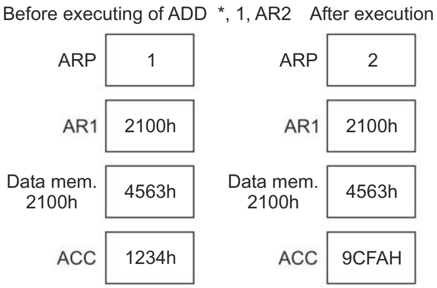


Fig. 4.10 *Addition with shifting and changing ARP in a single instruction*

Example 4.16 Let the initial content of ARP, AR1, ACC and data memory location 2100h be 2, 2100h, 1234h and 4563h, respectively; after executing the instruction ADD *, 1, AR2, the content of these registers and memory location are as shown in Fig. 4.10. In this case after the instruction is executed, the content of ARP becomes 2.

Examples of the some more ADD/SUB instructions are given next.

Example 4.17 ADD 55h, 2 ACC is added with the content of data memory with dma 55h in the current page after shifting it left by two positions.

Example 4.18 ADD #23h; ACC is added with the immediate constant 23h.

Example 4.19 SUB 55h, 2; ACC is subtracted with the content of data memory with dma 55h in the current page after shifting it left by two positions.

Example 4.20 SUB *, 2; ACC is added with the content of location pointed by the current AR after shifting it left by two positions.

The mnemonics of some of the other addition/subtraction instructions of 5X are as follows: ADCB, ADDB, ADDC, ADDS, ADDT, SBB, SBBB, SUBB, SUBC, SUBS, SUBT.

Description of these instructions is given in Appendix A4.1.

MOVE INSTRUCTIONS 4.5

The data move (DMOV) instruction copies the data from one memory location to the next higher location. It can use either direct or indirect addressing mode. For e.g., DMOV 45 copies the contents of location 45h to 46h.

Some more move instructions of C5X are as follows:

BLDP: Block move data from data memory to program memory

BLDD: Block move data from one data memory to another

BLPD: Block move data from program memory to data memory

TBLR: Block move data from program memory to data memory; the program memory address is contained in ACC lower order word. The dma can be specified using either direct addressing or indirect addressing.

TBLW: Block move data from data memory to program memory; The program memory address is contained in ACC lower order word. The dma can be specified using either direct addressing or indirect addressing.

BLDP and BLPD For the block move instruction BLDP the program memory address for the transfer is specified using block move address register (BMAR).

Example 4.21 Let the value of dma page pointer DP and the BMAR be 8, 2850. After executing the instruction BLDP 00h, the contents of the memory locations and BMAR are as shown in Fig. 4.11.

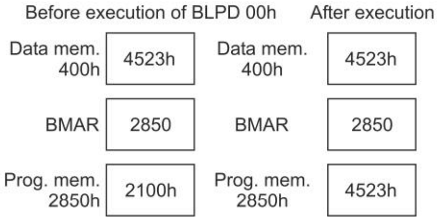


Fig. 4.11 *Block move from data to program memory*

The data memory address is specified using direct memory address in Example 4.21. The dma may also be specified using indirect addressing. For example, BLDP * is a valid instruction.

For the block move instruction BLPD, the program memory address for the transfer may be specified using BMAR. The pma may also be specified using immediate addressing mode. For the instruction BLPD, either direct addressing or indirect addressing may be used for specifying the dma.

Example 4.22 Let the value of dma page pointer DP and the BMAR be 8, 2850h. After executing the instruction BLDP 00h, the contents of the memory locations and BMAR are as shown in Fig. 4.12.

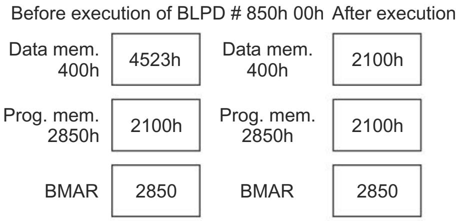


Fig. 4.12 *Block move with BMAR from program to data memory*

Example 4.23 Let the value of dma page pointer DP be8. After executing the instruction BLPD #850h, 00h, the contents of the memory locations are as shown in Fig. 4.13.

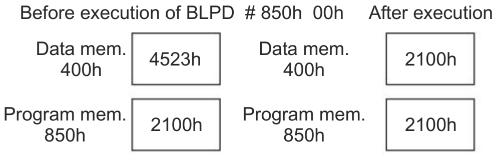


Fig. 4.13 *Block move with immediate addressing from program to data memory*

BLDD Instruction This instruction is used for moving data from one data memory to another. This permits a variety of combinations for specifying the source and destination addresses. BMAR may be used to specify either the source address or the destination address. Alternately one of the addresses may be specified using immediate addressing and the other address may be specified using either direct or indirect addressing. The various possible BLDD instructions and the addressing mode for source and destinations are given in Example 4.24.

Example 4.24 Let the value of DP and ARP be 8 and 2 and the content of AR2 and BMAR be 2800 h and 2900h, respectively. Specify the addressing modes and the addresses for the source and destination for the following instructions:

```
BLDD #400, 25h
BLDD #400h, *+
BLDD 45h,#400h
BLDD *+, #456, AR4
```

BLDD BMAR,*
BLDD BMAR, 25h
BLDD 00h, BMAR
BLDD *+, BMAR

The addressing modes and the addresses for the source and destination for the above instructions are given in Table 4.4.

Table 4.4 *Addressing modes for the various instruction types of BLDD*

Instruction	*Source*		*Destination*	
	Address	*Addressing mode*	*Address*	*Addressing mode*
BLDD # 400, 25h	400h	Imm.	425h	Direct
BLDD #400h, *+	400h	Imm.	2800h	Ind.
BLDD 45h, #450h	445h	Direct	450h	Imm.
BLDD *+, #456, AR4	2800h	Ind.	456h	Imm.
BLDD BMAR, *	2900h	(BMAR)	2800h	Ind.
BLDD BMAR, 25h	2900h	(BMAR)	425h	Direct
BLDD 00h, BMAR	400h	Direct	2900h	BMAR
BLDD *+, BMAR	2800h	Ind.	2900h	BMAR

MULTIPLICATION INSTRUCTIONS 4.6

The mnemonic for some of the instructions which multiply two 16-bit numbers are as follows:

MPY:	Multiply numbers in 2's complement form
MPYU:	Multiply unsigned numbers
MPYA:	Multiply and add the product to ACC
MPYS:	Multiply and subtract the product from ACC
MADD:	Multiply and add the product to ACC address of an operand given by BMAR
MPYA:	Multiply and add the product to ACC and move the on-chip RAM by one word.

In all the above multiply instructions one of the operand is taken from TREG0 and the other operand is specified using one of the addressing modes. In the MACD instruction, a data memory value is multiplied by a program memory value. The previous value of the product register is shifted and added to the ACC before the PREG is loaded with the product. The number of shifts is specified by the PM status bits. This instruction is useful for performing convolution of one array with another.

Example 4.25 Let the initial content of the data memory locations 315h and 316h, program memory location FF0Fh and the content of the registers TREG0, PREG and ACC be 45h, 28h, 8h, 46h, 0025 4235h, 0234 5678h, respectively. Let the value of data memory page pointer DP be 6 and the value of PM be 0 (i.e., no shift by the prescaler). After executing the instruction MACD 0FF0Fh, 15h, the content of the memory locations and registers are as shown in Fig.4. 14.

Before execution of MACD 0FF0Fh, 15h		After execution	
Data mem. 315h	45h	Data mem. 315h	45h
Data mem. 316h	28h	Data mem. 316h	45h
Program mem. FF0Fh	8h	Program mem. FF0Fh	8h
TREG0	1234h	TREG0	45h
PREG	4563h	PREG	0045h
ACC	1234h	ACC	5797h

Fig. 4.14 *Multiply accumulate and data move in a single instruction*

The data memory address from which one of the operand is obtained is 315h. This is because higher-order data memory address bus (DAB) bits 15 – 8 is 0000 0011 0 corresponding to DP 6. The lower-order bits specified in the instruction is 15h, i.e., 001 0101. The complete address in binary form is 0000 0011 0001 0101 and it corresponds to 0315h.

It may be noted from Example 4.24 that the previous value of PREG is added to ACC before the new value of the product is stored in PREG. The same thing is true if MAC is used in place of MACD. When either of these instructions are used for the convolution, when the last data is multiplied by the multiplier, accumulator will contain only the sum upto previous data. Hence to add the product corresponding to the last data, one more addition operation is required. This is achieved using the instruction APAC which adds the product register to the ACC.

The instructions ZAP and ZPR can be used for initialisation in this case. ZAP makes both ACC and PREG to be 0. ZPR makes PREG to be 0.

C5X has instructions for finding square of a number. The SQRA instruction adds the shifted value of PREG to ACC and stores the square of the content of data memory location in PREG. The SQRS instruction subtracts the shifted value of PREG from ACC and stores the square of the content of data memory location in PREG. Number of shifts for both these instructions are determined from the PM status bits. Brief description of all the multiply instructions is given in Appendix A4.4

4.6.1 Shift/logical Instructions

One of the operands for the AND, OR and XOR instructions is ACC. The other operand can be the content of a memory location specified using direct addressing or indirect addressing. Alternately a long constant can be specified using immediate addressing. In this case the constant may be shifted by 0–15 bits towards left before performing the operation.

AND	ANDing the ACC with a long constant k, the content of a dma or pma
OR	ORing the ACC with a long constant k, the content of a dma or pma

Example 4.26 Let the initial content of ACC be 1234h. After executing the instruction AND #2345h, 2, the content of ACC is as shown in Fig. 4.15.

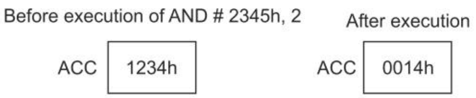


Fig. 4.15 *AND operation using immediate addressing, example 4.25*

In this case the data 2345h is left shifted by two positions before it is ANDed with ACC.

Example 4.27 Let the initial content of ARP, AR1, ACC and data memory location 2100h be 2, 2100h, 1234h and 4563h, respectively; after executing the instruction OR *,AR2, the content of these registers and memory location is as shown in Fig. 4.16.

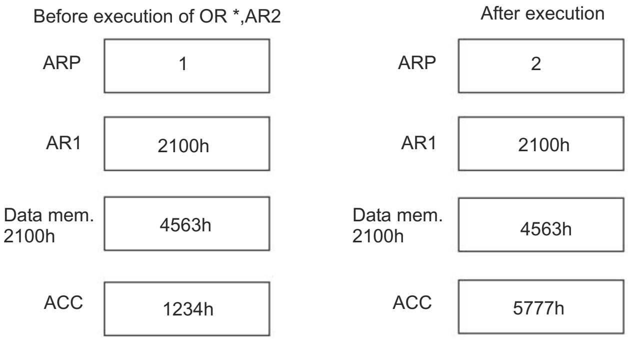


Fig. 4.16 *ORing using indirect addressing, example 4.26*

Example 4.28 Let the initial content of DP, ACC and data memory location FFD0h be 511, 1234h and 4563h, respectively; after executing the instruction XOR 50h, the content of these registers and memory location is as shown in Fig. 4.17.

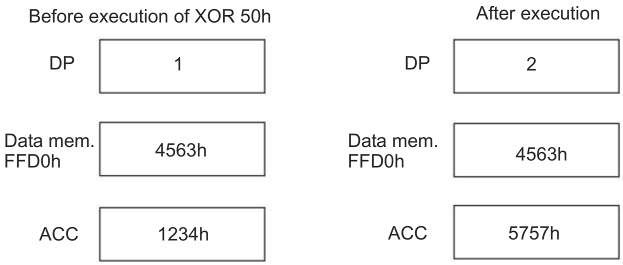


Fig. 4.17 *XORing using direct addressing, example 4.27*

Mnemonics for some of the other shift/logical instructions are as follows:

ANDB	ANDing ACC with ACCB
XORB	EX ORs the ACC with ACCB
ROLB	rotate both ACC and ACCB left once
ROL	rotate ACC left once
RORB	rotate both ACC and ACCB right once
ROR	rotate ACC right once

BSAR *n* Rotates ACC right by *n*(1–16) bits
EXAR exchange the contents of ACC, ACCB
NEG find 2's complement of ACC
CMPL find 1's complement of ACC
BIT copy bit *n* of a memory onto TC
BITT copy bit *n* of a memory onto TC.
n is given by the 4 LSBs of TREG2

Status bits affected by the above logical/shift instructions are given in Table 4.5.

Table 4.5 *Status bits affected/unaffected by the shift/logical instructions*

Mnemonic	*Status bit affected*	*Status bits not affected*
AND	All bits	
ANDB	None	
BIT, BITT	TC	
BSAR	None	
CMPL		C
EXAR	None	
NEG	C and OV	
OR		C
ORB	None	
ROL ROLB	C	
ROR, RORB	C	
XOR		C
XORB	None	

THE NORM INSTRUCTION 4.7

This instruction is useful for converting a fixed point number into a floating point number. The number to be converted is stored in ACC. In a sign extended number some of the most significant bits denote the sign extended bits and only the remaining bits denote the magnitude. Every time NORM instruction is executed, it removes an extra bit in ACC, which denotes the bit corresponding to sign extension. By repeated use of the NORM instruction, the ACC can be made to contain only the magnitude. The exponent is stored in the current AR. This instruction adjusts the value of ACC, current AR and TC bit as follows:

```
If ACC = 0 TC is set to 1;
Else if ((ACC(31) XOR ACC(30)) = 0)
    then
        {TC is set to 0;
        current AR is modified as specified by the
        NORM instruction;
        ACC is shifted left once;
        }
    Else TC is made 0;
```

The value of AR can be modified using one of the following schemes:
*, *+, *-, *0+, *0-, *BR0+, *BR0-

Example 4.29 Let the initial value of ARP, AR2 and ACC be 3, 00, and 0FFFF F002h, respectively. After executing the norm instruction NORM *+, their contents are shown in Fig. 4.18.

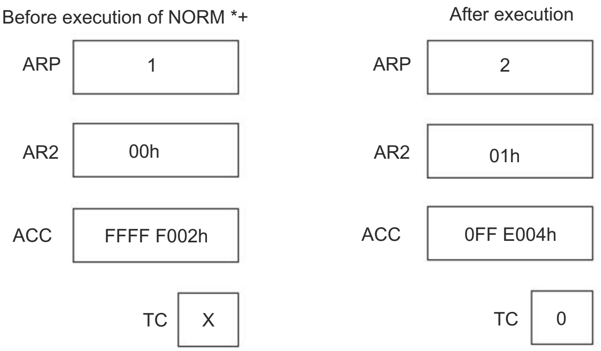


Fig. 4.18 *NORM instruction incrementing AR if ACC has extra sign bits*

Example 4.30 Let the initial value of ARP, AR2 and ACC be 3, 0Fh and 0FFFF F002h, respectively. After executing the norm instruction NORM *-, their contents are shown in Fig. 4.19.

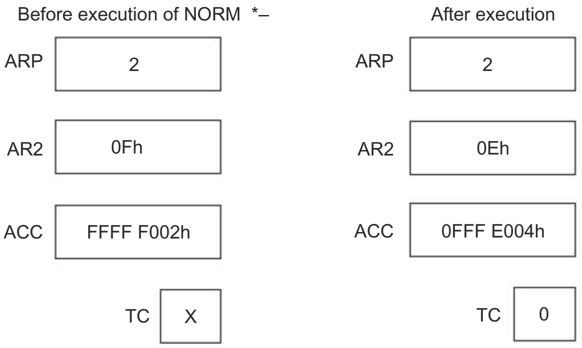


Fig. 4.19 *NORM instruction decrementing AR if ACC has extra sign bits*

PROGRAM CONTROL INSTRUCTIONS 4.8

4.8.1 Branch and Call Instructions

The C5X instruction set has both conditional and unconditional branch and call instructions. The branch and call instructions of C5X permit more than one condition to be tested using a single instruction. Branching occurs only if all the conditions are satisfied. The conditions which can be tested and the corresponding condition codes are given in Table 4.6.

Table 4.6 *Condition codes for 5X*

Conditions	*Condition codes*
ACC = 0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C = 0	NC
C = 1	C
OV = 0	NOV
OV=1	OV
TC = 0	NTC
TC = 1	TC
BIO low	BIO
Unconditionally	UNC

Example 4.31 BCND PGM1FFh, LEQ, OV In this case branch occurs to program memory address 1FFh only if ACC £ 0 and OV= 1. Otherwise branching does not occur. The ¢C5X performs speculative fetching by reading two additional instruction words. Hence if the conditions are not met, the fetched instructions would be executed. Otherwise these two instruction words are discarded.

The mnemonics for some of the program control instructions are as follows:

B	branch unconditionally
BACC	branch unconditionally to the address given by ACC
BCND	branch conditionally
BANZ	branch conditionally if ARn not zero
CALA	call a subroutine using indirect addressing
CALL	call a subroutine unconditionally
CC	call a subroutine conditionally

The branch instruction requires four cycles when branching occurs; one for the B instruction to enter the execute phase, one for fetching the branch address, two more for flushing out the two 1-word or one 2-word instruction which enter the instruction pipeline after the branch instruction. (Instruction pipelining in C5X is explained in Chapter 5.) The same is true with the call instructions. They also require four cycles out of which the last two cycles are required for flushing out the pipeline.

However the delayed branch and call instructions permit the call and branching to be carried out in two clock cycles. The one 2-word instruction or two 1-word instructions following the delayed branch/call instruction are fetched from program memory and executed before the branch/call is carried out. Hence instruction pipeline need not be flushed out after the call/branch instruction and the execution can resume from the branch address. Mnemonics of some of the delayed branch/call instructions are as follows:.

BACCD	delayed branch to program memory location specified by ACCL

BANZD	delayed branch to program memory location if AR not zero
BCNDD	delayed branch conditionally to program memory location
BD	delayed branch unconditionally to program memory location
CALAD	delayed call to subroutine addressed by ACCL
CALLD	delayed call to subroutine unconditionally
CCD	delayed call to subroutine conditionally

4.8.2 PUSH and POP Instructions

The PUSH instruction pushes the values down one level in the seven lower locations of the stack. The contents of the accumulator low byte (ACCL) are copied to the top of the stack (TOS). The values on the stack are pushed down before the ACC value is copied. The hardware stack is last-in, first-out with eight locations. If more than eight pushes (CALA, CALL, CC, INTR, NMI, PSHD, PUSH or TRAP instructions) occur before a POP, the first data values written are lost with each succeeding push. The PSHD instruction pushes a data memory location to the top of the stack instead of ACC after pushing the contents of the stack one level down. The POP and POPD instruction does the reverse operations. POP instruction pops the top of the stack to ACC and POPD instruction pops the top of the stack to a data memory. When the stack is popped, the bottom word is left unaffected and hence the bottom two words contain the same values.

4.8.3 RET Instruction

When this instruction is executed, the contents of the TOS are copied to the program counter (PC). The stack is popped one level after the contents are copied. The RET instruction is used with the CALA, CALL and CC instructions for subroutines.

Example 4.32 Let the RET instruction be stored in PGM address 96h.After executing the RET instruction, the contents of PC and stack are modified as shown in Table 4.7.

Table 4.7 *Contents of PC and stack In example 4.31*

Before executing RET		*After execution*
PC	96h	137h
Stack	137h	451h
	451h	751h
	751h	212h
	212h	3F3h
	3F3h	454h
	454h	16Eh
	16Eh	16Eh
	16Eh	16Eh

Note that in C5X, the stack is of size 8. Like the conditional and delayed call/branch instructions, there are also conditional and delayed return instructions. The mnemonics of these instructions are as follows:

RETCD	delayed return from subroutine conditionally
RETD	delayed return from subroutine

4.8.4 Repeat Instructions

The RPT instruction can be used to execute a single instruction repeatedly to a maximum no. of 65535 times. The iteration count can be specified using direct addressing, indirect addressing, short as well as long immediate addressing. Hence the following RPT instructions are valid

```
RPT 20
RPT *
RPT #7
RPT #2345h
```

If *n* is the content of the memory location or immediate constant, the operation is repeated *n*+1 times.

The RPTB instruction can be used to execute a block of instructions repeatedly to a maximum number of 65535 times.

For the RPTB instruction, the register block repeat count register (BRCR) determines the number of times a block of instructions is repeatedly executed. It should be loaded before the RPTB instruction is used. The PASR and PAER registers give the starting and ending address of the block of instructions.

For example, in the instruction RPTB 1500h, the PAER is loaded with 1500h and the block of instructions till the program memory address 1500h is executed. If the instructions following RPTB till the instruction in 1500h are to be executed *n* times, BRCR should be loaded with *n*–1.

Example 4.33

```
        SPLK #9h, BRCR
        RPTB 1500h
        LACC *+
        SUB 30h
        ADDB
1500h   LACB
```

In this example, the block starting at LACC instruction and ending with LACB is executed 10 times.

The instruction RPTZ # k clears both ACC and PREG and then executes the next instruction *k* times.

Only some of the instructions of C5X can be repeated. The instructions which can be repeatedly executed using repeat instructions are indicated in **bold face** form in the Appendix 4.

PERIPHERAL CONTROL 4.9

4.9.1 IN and OUT Instructions

The IN instruction of C5X reads a 16-bit number from input port and stores it in the data memory location. The OUT instruction of C5X reads a 16-bit number from data memory and writes it onto the output port. The data memory address could be given either using direct or indirect addressing I/O ports can either be the ports (PA0-PA15) inside C5X or ports outside. The ports PA0-PA15 are memory mapped to the locations 50h-5Fh in data page 0. Examples of valid IN and OUT instructions are as follows:

Example 4.34 IN 20h, PA5; Let DP = 7; In this case, the data is read from PA5 (i.e. data memory location 55h) and stored in data memory location 420h.

Example 4.35 IN *,100h; In this case, the data is read from I/O port with address 1000h and stored in the dma pointed by the current AR.

Example 4.36 OUT *,PA7; In this case, the data is read from data memory location pointed by the current AR and stored in PA7 (i.e. data memory location 57h in page 0).

Example 4.37 OUT 30h, 1000h; Let DP = 06h. In this case, the data is read from the data memory location 330h and stored in I/O port with address 1000h.

4.9.2 Instructions Used with Interrupts

As mentioned in Chapter 3, the core of C5X consists of the following on-chip devices: serial port, TDM serial port, timer, software programmable wait state generators, I/O ports and divide by one clock circuit. They are controlled using memory mapped registers.

The C5X devices have four external maskable interrupts $\overline{\text{IN T4}}$–$\overline{\text{INT1}}$ and a non-maskable interrupt (NMI) . Internal interrupts are generated by the serial port (RINT and XINT), the timer (TINT), the TDM port (TRNT and TXNT) and the software interrupt instructions (TRAP, NMI and INTR).

Interrupt priorities are set so that $\overline{\text{RS}}$ has the highest priority and $\overline{\text{INT4}}$ has the lowest priority. $\overline{\text{NMI}}$ has the second highest priority. The memory-mapped register interrupt mask register (IMR) is used for masking external and internal interrupts. The bit assignment for the IMR is shown in Fig. 4.20.

Lowest ⟶ Priority ⟶ Highest

15 – 12		11	10 – 9	8	7	6	5	4	3	2	1	0
RESD	HINT	RESD	$\overline{\text{INT4}}$	TXNT	TRNT	XINT	RINT	TINT		$\overline{\text{INT3}}$	$\overline{\text{INT2}}$	$\overline{\text{INT1}}$

Fig. 4.20 *Bit assignment for interrupt mask register (IMR)*

The IMR is accessible with both read and write operations. A 1 in bit positions 15 through 0 of the IMR enables the corresponding interrupt provided the interrupt enable flag INTM is 0. INTM =1 disables all interrupts. The instructions SETC INTM and CLRC INTM are used for this purpose. The IFR register indicates the interrupts which are pending. The bit assignment for the IFR is shown in Fig. 4. 21.

RESD	HINT	RESD	$\overline{\text{INT4}}$	TXNT	TRNT	XINT	RINT	TINT	$\overline{\text{INT3}}$	$\overline{\text{INT2}}$	$\overline{\text{INT1}}$

Fig. 4.21 *Bit assignment for interrupt flag register (IFR)*

An IFR bit is cleared by reset, servicing of the corresponding interrupt or by writing a 1 to the corresponding bit. Writing a 0 to a specific bit has no effect. All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR.

When the CPU accepts an interrupt, before fetching the instruction at the interrupt vector location, it stores each of the following registers ACC, ACCB, PREG, ST0, ST1, PMST, TREG0, TREG1, TREG2, INDX and ARCR in one deep stack (shadow registers). The PC is stored in an 8 deep hardware stack. These registers are popped back when the return from interrupt instructions RETE and RETI are executed. RETE also clears INTM but RETI does not do that. The following list gives a list of instructions used with interrupts.

IDLE wait till an unmasked interrupt or reset
IDLE2 enter low power mode and wait till
an unmasked interrupt or reset occurs
NMI execute the ISR starting at 0024h
NOP increment PC; perform nothing else
XC n, [] execute the next *n* instructions if the condition [ ] is met else it executes NOPs for the Next *n* instructions
TRAP Execute the ISR Starting at 0022h
INTR *K* execute a software ISR starting at an address
depending on the value of *k*
k is not the starting address of ISR
RETE return from ISR and clear INTM
RETI return from ISR but do not clear INTM

When the IDLE instruction is executed, the CPU enters the lower power wait state and comes out of it only if an interrupt occurs. After it comes out, it will execute the ISR if the INTM = 0. Otherwise it skips the ISR and executes the instruction following IDLE.

When the IDLE2 is executed, the CPU behaves the same way as for IDLE. In addition to that, the peripherals also become inactive.

When interrupts are disabled (INTM =1) and an interrupt causes an IDLE or IDLE2 instruction to be exited, none of the IFR bits are cleared (including the IFR bit that caused the IDLE or IDLE2 to be exited). The only event, other than reset or clearing the IFR bits directly in software, that can cause an IFR bit to be cleared is actually taking the interrupt trap when the ISR is entered. Therefore, if an interrupt causes an IDLE or IDLE2 instruction to be exited when interrupts are disabled, the corresponding IFR bit is not cleared; whereas, if interrupts are enabled and the ISR is entered, the IFR bit is cleared.

The INTR instruction allows any ISR to be executed through the software. An INTR interrupt for the external interrupts $(\overline{\text{INT1}} - \overline{\text{INT4}})$ executes like an external interrupt.

The NMI instruction has the same affect as a hardware non-maskable interrupt (NMI). The NMI instruction transfers program control to program memory location 24h. Interrupts are globally disabled (INTM =1), but key registers are not saved into context shadow registers.

The TRAP instruction transfers program control to program memory location 22h. The TRAP instruction disables interrupts (INTM =1), but key registers are not saved into context shadow registers.

APPENDIX 4

INSTRUCTION SET SUMMARY

A4.1 Accumulator Memory Reference Instructions

Mnemonic	*Description*
ABS	Absolute value of ACC; zero carry bit
ADCB	Add ACCB and carry bit to ACC
ADD	Add data memory value, with left shift, to ACC
	Add data memory value,with left shift of 16, to ACC
	Add short immediate to ACC
	Add long immediate, with left shift, to ACC
ADDB	Add ACCB to ACC
ADDC	Add data memory value and carry bit to ACC with sign extension suppressed
ADDS	Add data memory value to ACC with sign extension suppressed
ADDT	Add data memory value, with left shift specified by TREG1, to ACC
AND	AND data memory value with ACCL; zero ACCH
	AND long immediate, with left shift, with ACC
	AND long immediate, with left shift of 16, with ACC
ANDB	AND ACCB with ACC
BSAR	Barrel-shift ACC right
	CMPL 1s complement ACC
CRGT	Store ACC in ACCB if ACC > ACCB
CRLT	Store ACC in ACCB if ACC < ACCB
EXAR	Exchange ACCB with ACC
LACB	Load ACC to ACCB
LACC	Load data memory value, with left shift, to ACC
	Load long immediate, with left shift, to ACC
	Load data memory value, with left shift of 16, to ACC
LACL	Load data memory value to ACCL; zero ACCH
	Load short immediate to ACCL; zero ACCH
LACT	Load data memory value, with left shift specified by TREG1, to ACC
LAMM	Load contents of memory-mapped register to ACCL; zero ACCH
NEG	Negate (2s complement) ACC
NORM	Normalise ACC
OR	OR data memory value with ACCL
	OR long immediate, with left shift, with ACC
	OR long immediate, with left shift of 16, with ACC
ORB	OR ACCB with ACC
ROL	Rotate ACC left 1 bit
ROLB	Rotate ACCB and ACC left 1 bit
ROR	Rotate ACC right 1 bit
RORB	Rotate ACCB and ACC right 1 bit

SACB	Store ACC in ACCB
SACH	Store ACCH, with left shift, in data memory location
SACL	Store ACCL, with left shift, in data memory location
SAMM	Store ACCL in memory-mapped register
SATH	Barrel-shift ACC right 0 or 16 bits as specified by TREG1
SATL	Barrel-shift ACC right as specified by TREG1
SBB	Subtract ACCB from ACC
SBBB	Subtract ACCB and logical inversion of carry bit from ACC
SFL	Shift ACC left 1 bit
SFLB	Shift ACCB and ACC left 1 bit
SFR	Shift ACC right 1 bit
SFRB	Shift ACCB and ACC right 1 bit
SUB	Subtract data memory value, with left shift, from ACC
SUBB	Subtract data memory value and logical inversion of carry bit from ACC with sign extension suppressed
SUBS	Subtract data memory value from ACC with sign extension suppressed
SUBT	Subtract data memory value, with left shift specified by TREG1, from ACC
XOR	Exclusive-OR data memory value with ACCL
XORB	Exclusive-OR ACCB with ACC
ZALR	Zero ACCL and load ACCH with rounding
ZAP	Zero ACC and PREG

A4.2 Auxiliary Registers and Data Memory Page Pointer Instructions

Mnemonic	*Description*
ADRK	Add short immediate to AR
CMPR	Compare AR with ARCR as specified by CM bits
LAR	Load data memory value to ARx
LDP	Load data memory value to DP bits
MAR	Modify AR
SAR	Store ARx in data memory location
SBRK	Subtract short immediate from AR

A4.3 Parallel Logic Unit (PLU) Instructions

Mnemonic	*Description*
APL	AND data memory value with DBMR, and store result in data memory location
	AND data memory value with long immediate and store result in data memory location
CPL	Compare data memory value with DBMR
	Compare data memory value with long immediate
OPL	OR data memory value with DBMR and store result in data memory location
OR	Data memory value with long immediate and store result in data memory location
SPLK	Store long immediate in data memory location
XPL	Exclusive-OR data memory value with DBMR and store result in data memory location

	Exclusive-OR data memory value with long immediate and store result in data memory location
LPH	Load data memory value to PREG high byte
LT	Load data memory value to TREG0

A4.4 TREG0, PREG and Multiply Instructions

Mnemonic	*Description*
LTA	Load data memory value to TREG0; add PREG, with shift specified by PM bits, to ACC
LTD	Load data memory value to TREG0; add PREG, with shift specified by PM bits, to ACC; and move data
LTP	Load data memory value to TREG0; store PREG, with shift specified by PM bits, in ACC
LTS	Load data memory value to TREG0; subtract PREG, with shift specified by PM bits, from ACC
MAC	Add PREG, with shift specified by PM bits, to ACC; load data memory value to TREG0; multiply data memory value by program memory value and store result in PREG
MACD	Add PREG, with shift specified by PM bits, to ACC; load data memory value to TREG0; multiply data memory value by program memory value and store result in PREG; and move data
MADD	Add PREG, with shift specified by PM bits, to ACC; load data memory value to TREG0; multiply data memory value by value specified in BMAR and store result in PREG; and move data
MADS	Add PREG, with shift specified by PM bits, to ACC; load data memory value to TREG0; multiply data memory value by value specified in BMAR and store result in PREG
MPY	Multiply data memory value by TREG0 and store result in PREG Multiply short immediate by TREG0 and store result in PREG Multiply long immediate by TREG0 and store result in PREG
MPYA	Add PREG, with shift specified by PM bits, to ACC; multiply data memory value by TREG0 and store result in PREG
MPYS	Subtract PREG, with shift specified by PM bits, from ACC; multiply data memory value by TREG0 and store result in PREG
MPYU	Multiply unsigned data memory value by TREG0 and store result in PREG
PAC	Load PREG, with shift specified by PM bits, to ACC
SPAC	Subtract PREG, with shift specified by PM bits, from ACC
SPH	Store PREG high byte, with shift specified by PM bits, in data memory location
SPL	Store PREG low byte, with shift specified by PM bits, in data memory location
SPM	Set product shift mode (PM) bits
SQRA	Add PREG, with shift specified by PM bits, to ACC; load data memory value to TREG0; square value and store result in PREG
SQRS	Subtract PREG, with shift specified by PM bits, from ACC; load data memory value to TREG0; square value and store result in PREG
ZPR	Zero PREG

A4.5 Branch and Call Instructions

Mnemonic	*Description*
B	Branch unconditionally to program memory location
BACC	Branch to program memory location specified by ACCL
BACCD	Delayed branch to program memory location specified by ACCL
BANZ	Branch to program memory location if AR not zero
BANZD	Delayed branch to program memory location if AR not zero
BCND	Branch conditionally to program memory location
BCNDD	Delayed branch conditionally to program memory location
BD	Delayed branch unconditionally to program memory location
CALA	Call to subroutine addressed by ACCL
CALAD	Delayed call to subroutine addressed by ACCL
CALL	Call to subroutine unconditionally
CALLD	Delayed call to subroutine unconditionally
CC	Call to subroutine conditionally
CCD	Delayed call to subroutine conditionally
INTR	Software interrupt that branches program control to program memory location
NMI	Nonmaskable interrupt and globally disable interrupts (INTM =1)
RET	Return from subroutine
RETC	Return from subroutine conditionally
RETCD	Delayed return from subroutine conditionally
RETD	Delayed return from subroutine
RETE	Return from interrupt with context switch and globally enable interrupts (INTM = 0)
RETI	Return from interrupt with context switch
TRAP	Software interrupt that branches program control to program memory location 22h
XC	Execute next instruction(s) conditionally

A4.6 I/O and Data Memory Instructions

Mnemonic	*Description*
BLDD	Block move from data to data memory
	Block move from data to data memory with destination address long immediate
	Block move from data to data memory with source address in BMAR
	Block move from data to data memory with destination address in BMAR
BLDP	Block move from data to program memory with destination address in BMAR
BLPD	Block move from program to data memory with source address in BMAR
	Block move from program to data memory with source address long immediate
DMOV	Move data in data memory
IN	Input data from I/O port to data memory location
LMMR	Load data memory value to memory-mapped register
OUT	Output data from data memory location to I/O port
SMMR	Store memory-mapped register in data memorylocation

TBLR	Transfer data from program to data memory with source address in ACCL
TBLW	Transfer data from data to program memory with destination address in ACCL

A4.7 Control Instructions

Mnemonic	*Description*
BIT	Test bit BITT Test bit specified by TREG2
CLRC	Clear overflow mode (OVM) bit
	Clear sign extension mode (SXM) bit
	Clear hold mode (hM) bit
	Clear test/control (TC) bit
	Clear carry (C) bit
	Clear configuration control (CNF) bit
	Clear interrupt mode (INTM) bit
	Clear external flag (XF) pin
IDLE	Idle until non-maskable interrupt or reset
IDLE2	Idle until non-maskable interrupt or reset, low-power mode
LST	Load data memory value to ST0
	Load data memory value to ST1
NOP	No operation
POP	Pop top of stack to ACCL; zero ACCH
POPD	Pop top of stack to data memory location
PSHD	Push data memory value to top of stack
PUSH	Push ACCL to top of stack
RPT	Repeat next instruction specified by data memory value
	Repeat next instruction specified by short immediate
	Repeat next instruction specified by long immediate
RPTB	Repeat block of instructions specified by BRCR
RPTZ	Clear ACC and PREG; repeat next instruction specified by long immediate
SETC	Set overflow mode (OVM) bit
	Set sign extension mode (SXM) bit
	Set hold mode (hM) bit
	Set test/control (TC) bit
	Set carry (C) bit
	Set external flag (XF) pin high
	Set configuration control (CNF) bit
	Set interrupt mode (INTM) bit
SST	Store ST0 in data memory location
	Store ST1 in data memory location

Review Questions

4.1 Explain how the memory address of the operand is obtained in (i) direct addressing mode and (ii) MMR addressing mode.

4.2 What are the different ways in which the auxiliary register pointer can be updated in 5X?

4.3 Explain the immediate addressing mode of C5X with examples.

4.4 Give a brief account of the load/ store instructions of 5X.

4.5 Let the content of ARP, AR1, ACC and the value of data memory locations 25h and 825h be as shown in Fig. 4.22. After execution of the SAMM * instruction, What are the contents of the above registers and memory locations?

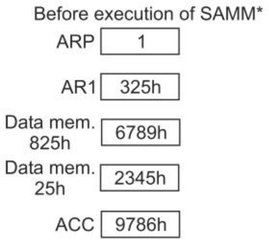


Fig. 4.22 *Content of ARP, AR1, ACC and value of data memory locations 25h and 825h*

4.6 Explain the arithmetic instructions of C5X.

4.7 What are the instructions of C5X which are used for block transfer?

4.8 Give the list of the mnemonics of the shift/logical expressions of C5X and explain them in brief.

4.9 Explain how NORM instruction of C5X is used?

4.10 Explain how the delayed and undelayed call and branch instructions of C5X are different in their operation.

4.11 How is the repeat count determined in the RPT & RPTB instructions of C5X?

4.12 Explain the use of the interrupt mask register and interrupt flag register of C5X.

Self Test Questions

4.1 The data memory used with C5X processors is split into ——— pages each of ——— words long.
(a) 512, 128 (b) 256, 256 (c) 128, 512 (d) 1024, 64

4.2 The register which holds the address of the current data memory page is ———.
(a) DP (b) ARP (c) ARB

4.3 No. of words of program memory, data memory that can be addressed by C5X processors are ——, ——.
(a) 64K,64K (b) 64K, 96K (c) 96K, 64K (d) 96K, 96K

4.4 The memory-mapped direct addressing mode is used to access data in page ———.
(a) 1 (b) 0 (c) 511 (d) 512

4.5 The no. of registers which can be used for accessing data using indirect addressing mode is ———.
(a) 16, 16 (b) 16, 8 (c) 8, 8 (d) 8, 16

4.6 The registers used for indirect addressing of memory are called ———.
(a) auxiliary registers (ARs)
(b) block move address register (BMAR)
(c) TREGn
(d) index register (INDX)

4.7 In the indirect addressing mode, out of the eight ARs, the one which is currently used for accessing data is denoted by the register ———.
(a) ARB (b) ARP (c) DP (d) BRCR

4.8 The register which is used for storing the contents of ARP temporarily is
(a) ARB (b) DP (c) TREG1 (d) TREG2
(e) TREG3

4.9 When an operand for an instruction is accessed using the indirect addressing mode and the content of the AR used for accessing the data is to be left unaltered after the instruction is executed, the addressing mode is specified by the symbol ———.
(a) # (b) * (c) *– (d) *+

4.10 When an operand for an instruction is accessed using the indirect addressing mode and the content of the auxiliary register used for accessing the data is to be decremented after the instruction is executed, the addressing mode is specified by the symbol ———.

(a) # (b) * (c) *– (d) *+

4.11 The 16-bit register used with indirect addressing mode for testing whether an increment/decrement operation of an AR has exceeded a particular value or not is ———.

(a) ARCR (b) ARP (c) ARB (d) INDX

4.12 The 16-bit register used for incrementing/decrementing the ARn in steps larger than 1 is ———.

(a) ARCR (b) ARP (c) ARB (d) INDX

4.13 The contents of ARs are decremented/incremented using ———.

(a) central ALU (b) auxiliary ALU (c) PLU

4.14 When an operand for an instruction is accessed using the indirect addressing mode and after the data is fetched, the content of the AR used for accessing the data is to be decremented by the number in INDX register, the addressing mode is specified by the symbol ———.

(a) *0+ (b) *0– (c) *BR0+ (d) *BR0-

4.15 When an operand for an instruction is accessed using the indirect addressing mode and after the data is fetched, the content of the AR used for accessing the data is to be incremented by the number in INDX register with reverse carry propagation, the addressing mode is specified by the symbol ———.

(a) *0+ (b) *0– (c) *BR0+ (d) *BR0–

4.16 The AR ALU (ARAU) performs ——— arithmetic on ——— numbers.

(a) unsigned, 16 (b) signed, 16

(c) signed, 32 (d) unsigned, 32

4.17 The symbol used to indicate the immediate address mode for the operand is ———.

(a) $ (b) * (c) # (d) *-(e)*+

4.18 In the dedicated register addressing mode, the register whose contents are used if an immediate operand is unspecified is ———.

(a) ARCR (b) BMAR (c) DBMR (d) ACCB

4.19 Before the instruction SBRK #5H is executed, the contents of ARP, AR3 and AR5 are 3H, 1058H and 1000H, respectively. After the execution of the instruction, the content of AR3 is ———.

(a) 5H (b) 1053H (c) 1058H (d) 1000H

4.20 Assume that the contents of ACC, ARP, AR3 and locations 0045H, 40C5H are 1000H, 3, 40C5H and 2400,2300H, respectively, initially. When the instruction LAMM * is executed, the content of ACC is ———.

(a) 2400H (b) 2300H (c) 40C5H (d) 0003H

4.21 The mnemonic for the instruction used to move a word from data memory to program memory is ———.

(a) BLDD (b) BLDP (c) BLPD (d) TBLR
(e) TBLW

4.21 The mnemonic for the instruction used to move a word from data memory to program memory and in which the program memory address is contained in ACC lower order word is ———.

(a) BLDD (b) BLDP (c) BLPD (d) TBLR
(e) TBLW

4.22 The mnemonic for the instruction which multiplies two 16-bit numbers represented in 2's complement form is ———.

(a) MPYA (b) MPY (c) MPYU (d) MPYS
(e) MADD (f) MADS

4.23 The mnemonic for the instruction which loads zero into PREG is ———.

(a) SQRA (b) SQRS (c) ZPR (d) ZAP

4.24 The mnemonic for the instruction which makes the program to branch unconditionally is ———.

(a) B (b) BACC (c) BANZ (d) BCND

4.25 Using the RPT *#k* instruction, the maximum no. of times a single instruction can be repeatedly executed is ———.

(a) 65535 (b) 255 (c) 256 (d) 65536

4.26 The mnemonic for the instruction which executes next *k* instructions repeatedly and which clears both ACC and PREG before starting the execution of the block of instructions is ———.

(a) RPTB #*k* (b) RPTB #(*k*–1)

(c) RPTZ #*k* (d) RPTZ #(*k*–l)

4.27 The IN instruction of C5X reads a ——— number from input port and stores it in ———.

(a) 8, ACC (b) 8, memory

(c) 16, ACC (d) 16, memory

4.28 The mnemonic for the instruction which forces the program being executed to wait until an unmasked interrupt or reset occurs is ———.

(a) NOP (b) IDLE (c) IDLE2 (d) XC

4.29 The mnemonic for the instruction which executes the next *n* instructions if the condition specified with the instruction is met else it executes NOPs for the next *n* instructions is ——— *n* conditions.

(a) XC (b) RPT (c) RPTB (d) RPTZ

5 INSTRUCTION PIPELINING IN C5X

PIPELINE STRUCTURE 5.1

C5X permits four operations, viz., Fetching, Decoding, Reading and Execution to be performed simultaneously using a 4-phase clock. This allows four instructions to be processed simultaneously in the CPU. When the first instruction is in the execute phase, the second instruction can be in the read phase, the third instruction can be in the decode phase, and the fourth instruction can be in the fetch phase.

The functions performed in the four phases of the ′C5X pipeline are as follows:

Fetch (F): This phase fetches the instruction words from memory and updates the program counter (PC).

Decode (D): This phase decodes the instruction word and performs address generation and ARAU updates of auxiliary registers.

Read (R): This phase reads operands from memory, if required.
If the instruction uses indirect addressing mode, it will read the memory location pointed at by the ARP before the update of the previous decode phase.

Execute (E): This phase performs any specify operation, and, if required, writes results of a previous operation to memory.

PIPELINE OPERATION 5.2

The pipeline is essentially invisible to the user except in some cases, such as AR updates, memory-mapped accesses of the CPU registers, the NORM instruction and memory configuration commands. Furthermore, the pipeline operation is not protected. The user has to understand the pipeline operation to avoid the pipeline conflict by arranging the code. The following sections show how the pipeline operation and the pipeline conflict affect the result.

NORMAL PIPELINE OPERATION 5.3

5.3.1 Instructions with Single Word and Two Words

When a program segment involves single-word single-cycle instructions executing with no wait state, there is perfect overlapping in the pipeline, where all four phases operate in parallel. In the case of programs involving two-word instructions, it is not possible to keep all the functional units busy all the time and the operation performed by some of the units may be dummy operations resulting in no productivity. Examples 5.1 and 5.2 illustrate these facts.

Example 5.1 Consider the following program involving only single-word single-cycle instructions:

```
ADD *+
SAMM  TREG0
MPY *+
SQRA *+, AR2
```

When this program is executed, the instruction pipeline is loaded as shown in Table 5.1.

Table 5.1 *Pipeline operation of 1 -word instruction*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[SAMM]	ADD			
2	[MPY]	SAMM	ADD		
3	[SQRA]	MPY	SAMM	ADD	
4		SQRA	MPY	SAMM	ADD
5			SQRA	MPY	SAMM
6				SQRA	MPY
7					SQRA

Example 5.2 Consider the following program involving two-word two-cycle instructions in addition to single-word single-cycle instructions:

```
ADD # 2500h
SAMM  TREG0
MPY *+
SQRA *+, AR2
```

When this program is executed, the instruction pipeline is loaded as shown in Table 5.2.

Table 5.2 *Pipeline operation with a 2 -word instruction*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[2500h]	ADD			
2	[SAMM]	2500h	ADD		
3	[MPY]	SAMM	Dummy	ADD	
4	[SQRA]	MPY	SAMM	Dummy	ADD
5		SQRA	MPY	SAMM	Dummy
6			SQRA	MPY	SAMM
7				SQRA	MPY
					SQRA

From Table 5.2 it can be verified that the decode, read and execute functional units are doing unproductive operations in cycles 3, 4 and 5 respectively.

5.3.2 Pipeline Operation with Branch and Call Instructions

The branch instruction requires two cycles when branching does not occur and four cycles when branching occurs; one for the B instruction to enter the execute phase, one for fetching the branch address, two more for flushing out the one two-word or two 1-word instructions which enter the instruction pipeline after the branch instruction. The same is true with the call instructions. They require two cycles when the subroutine is not called. When the subroutine is called they also require four cycles out of which the last two cycles are required for flushing out the pipeline.

However, the delayed branch and call instructions permit the call and branching to be carried out in two clock cycles. The one 2-word instruction or two 1-word instructions following the delayed branch/call instruction are fetched from program memory and executed before the branch/call is carried. Hence instruction pipeline need not be flushed out after the call/branch instruction and the execution can resume from the branch address. Examples 5.3-5.6 illustrate these facts.

Example 5.3 When the following program

```
           ZAP
           BPGM1250h
           ADD *
           SACL *+
           MAC 4500h,25h
PGM1250h:  LACC *+
```

is executed, the instruction pipeline is loaded in different cycles as shown in Table 5.3.

Table 5.3 *Instruction pipeline for branch instruction, Example 5.3*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[B]	ZAP			
2	[1250h]	B	ZAP		
3	[ADD*]	125Oh	B	ZAP	
4	[SACL*+]	ADD*	Dummy	B	ZAP
5	[LACC*-]	SACL*+	Dummy	Dummy	B
6		LACC*+	Dummy	Dummy	Dummy
7			LACC*+	Dummy	Dummy
8				LACC*+	Dummy
9					LACC*+

In Table 5.3, [B], [1250h], etc., denote the program memory (PGM) address where the branch instruction, the next operand, etc., are stored.

When the instruction B enters the execute phase, the ADD and SACL instructions enter the decode and fetch phases, respectively. However, since these two instructions are not to be executed, they are dummy phases and these instructions have to be flushed out of the instruction pipeline. This requires two cycles. Hence branch requires four cycles (one for the B instruction to enter the execute, one for fetching the branch address and two more for flushing out the unwanted instructions.)

The same is true with the call instructions. They also require four cycles out of which the last two cycles are required for flushing out the pipeline. However the delayed branch and call instructions permit the call and branching to be carried out in two clock cycles. The one 2-word instruction or two 1-word instructions following the delayed branch/call instruction are fetched from program memory and executed before the branch/call is carried. Hence instruction pipeline need not be flushed after the call/branch instruction and the execution can resume from the branch address.

Example 5.4 When the following program

```
            ZAP
            BD PGM1250h
            ADD *
            SACL *+
            MAC 4500h, 25h
PGM1250h:   LACC *+
```

is executed, the instruction pipeline is loaded in different cycles as shown in Table 5.4.

Table 5.4 *Instruction pipeline for delayed branch instruction*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[BD]	ZAP			
2	[1250h]	BD	ZAP		
3	[ADD*]	1250h	BD	ZAP	
4	[SACL*+]	ADD*	Dummy	BD	ZAP
5	[LACC *+]	SACL*+	ADD*	Dummy	BD
6		LACC*+	SACL*+	ADD*	Dummy
7			LACC*+	SACL*+	ADD*
8				LACC*+	SACL*+
9					LACC*+

The ADD and SACL instructions are in the instruction pipeline when the delayed branching BD occurs to PGM1250h and the LACC instruction enters the fetch phase. After these two instructions are executed, LACC instruction enters the execute phase.

Example 5.5 When the following program

```
            SACL *+
            CC PGM1250h, GT, NOV
            LDP 127h
            SUB 20h
            MAC 4500h,25h
PGM1250h:   LACC *+
```

is executed, the instruction pipeline is loaded in different cycles (as shown in Table 5.5) if the condition GT, NOV is not met,

Table 5.5 *Instruction pipeline for CC instruction when condition not met*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[CC]	SACL			
2	[1250h]	CC	SACL		
3	[LDP]	1250h	CC	SACL	
4	[SUB]	LDP	Dummy	CC	SACL
5	[MAC]	SUB	LDP	Dummy	CC
6		MAC	SUB	LDP	Dummy
7			MAC	SUB	LDP
8				MAC	SUB
9					MAC

(i.e., subroutine is not called). If the condition is met, the instruction pipeline in different cycles is as shown in Table 5.6.

In Table 5.5, the CC instruction requires only two cycles as the condition is not met. The instructions LDP, SUB and MAC which enter the pipeline are executed one after another.

Table 5.6 *Instruction pipeline for CC instruction when condition met*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[CC]				
2	[1250h]	SACL			
3	[LDP]	CC	SACL		
4	[SUB]	1250h	CC	SACL	
5	[LACC]	LDP	Dummy	CC	SACL
6		SUB	Dummy	Dummy	CC
7		LACC	Dummy	Dummy	Dummy
8			LACC	Dummy	Dummy
9				LACC	Dummy
					LACC

In Table 5.6, the CC instruction requires four cycles as the condition is met. The instructions LDP and SUB which enter the pipeline are flushed out of pipeline in the last two cycles. After four cycles the first instruction in the subroutine (LACC) is executed.

5.3.3 Pipeline Operation with Return Instructions

The unconditional return instruction RET requires four clock cycles of which the last two cycles are required for flushing out the pipeline. The conditional return instruction RETC requires four clock cycles if the condition is met. Out of the four cycles the last two cycles are required for flushing out the pipeline. If the condition is not met, it requires only two cycles. Delayed return instructions RETD and RETCD require only two clock cycles and they do not require the pipeline to be flushed out.

5.3.4 Pipeline Operation on ARAU Memory-Mapped Registers

Auxiliary register arithmetic unit (ARAU) updates of the ARs occur during the decode (D) phase of the pipeline. Hence the ARs are updated in the decode phase itself when the indirect addressing mode is used with arguments such as *+, *–, *0+, etc. This enables any instruction which follows this instruction to have the correct address when it enters the data read phase. This is desirable because when one instruction enters the execute phase the next instruction enters the data read phase.

However, when the ARs are modified using load, store, using memory-mapped addressing (e.g., LAR, SAMM, LMMR, SACL, or SPLK, etc), they are modified only in the execute (E) phase of the pipeline. Therefore, the use of ARs for the next two instructions after a memory-mapped load of the AR is prohibited. This means that the next two instructions after a memory-mapped load of the AR should not use this AR. Modifications to the index register (INDX) and auxiliary register compare register (ARCR) also occur in the E phase of the pipeline. Therefore, any AR updates using the INDX or the ARCR must take place at least two cycles after a load of these registers.

In Example 5.6, a case where SAMM instruction is immediately followed by instructions which use AR is presented. In this case unexpected results are obtained. In Example 5.7, this program is modified so that the AR is used only after two cycle elapses after the AR is loaded. This is achieved by using two NOPs. In this case correct results are obtained.

Example 5.6 Assume that the contents of the memory location 164h, 165h, 166h, 167h, 168h are as follows:

(164h) = 90h, (165h) = 80h, (166h) = 60h,
(167h) = 40h, and (168h) = 30h.
When the following program

```
LAR  AR1,#167h
LACC #164h
SAMM AR1
LACC *+
ADD *+
```

is executed the instruction pipeline at various clock cycle is as shown in Table 5.7.

Table 5.7 *Instruction pipeline with SAMM followed by illegal instructions*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*	*ACC*	*ARl*
1	[LACC]	LAR				XX	XX
2	[164h]	LACC	LAR			XX	XX
3	[SAMM]	164h	LACC	LAR		XX	XX
4	[LACC]	SAMM	Dummy	LACC	LAR	XX	167h
5	[ADD]	LACC	SAMM	Dummy	LACC	164h	167h
6		ADD	LACC	SAMM	Dummy	164h	168h
7			ADD	LACC	SAMM	164h	164h
8				ADD	LACC	40h	164h
9					ADD	70h	164h

In Example 5.6 what was expected was to add the content of location 164h, i.e., 90h, with that of 165h, i.e., 80h, and store the result in ACC. But what actually happens is that the content of location 167h is added with that of location 168h and the result is stored in ACC. This is because SAMM instruction is followed immediately by LACC *+ and ADD *+ both of which make use of ARl for indirect addressing. Unexpected results are obtained because of the the following reasons:

When the instruction LACC + enters the decode phase in cycle 6 the value of AR1 is 167h and this is used for accessing the data in its data read phase in cycle 6. Further even though LACC *+ enters the instruction pipeline after the SAMM instruction, it modifies AR1 to be 168h in its decode phase in clock cycle 6 itself. SAMM modifies the value of AR1 only in its execute phase in clock cycle 7.

In 7th clock cycle the ADD *+ enters the decode phase. At that time the value of AR1 is 168h and hence it uses this address when it enters the data read phase in cycle 8. It also modifies AR1 to 169h in cycle 7. But SAMM also tries to modify AR1 to 164h. In this conflict only SAMM succeeds.

Example 5.7

Assume that the contents of the memory location 164h, 165h, 166h, 167h, 168h are as follows:

(164h) = 90h, (165h) = 80h, (166h) = 60h,
(167h) = 40h, and (168h) = 30h.
When the following program

```
LAR  AR1,#167h
LACC #164h
SAMM AR1
NOP
NOP
LACC *+
ADD *+
```

is executed the instruction pipeline at various clock cycles is as shown in Table 5.8.

Table 5.8 *Instruction pipeline with SAMM followed by illegal instructions*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*	*ACC*	*AR1*
1	[LACC]	LAR				XX	XX
2	[164h]	LACC	LAR			XX	XX
3	[SAMM]	164h	LACC	LAR		XX	XX
4	[NOP]	SAMM	Dummy	LACC	LAR	XX	167h
5	[NOP]	NOP	SAMM	Dummy	LACC	164h	167h
6	[LACC]	NOP	Dummy	SAMM	Dummy	164h	168h
7	[ADD]	LACC	Dummy	Dummy	SAMM	164h	164h
8		ADD	LACC	Dummy	Dummy	164h	165h
9			ADD	LACC	Dummy	164h	166h
10				ADD	LACC	90h	166h
11					ADD	110h	166h

In Example 5.7 what was expected was to add the content of location 164h, i.e., 90h, with that of 165h, i.e., 80h, and store the result in ACC. This actually happens as the SAMM instruction is followed immediately by two NOP instructions which do not make use of AR1. Expected results are obtained because of the the following reasons:

When the instruction LACC *+ enters the decode phase in cycle 8, the value of AR1 is 164h and this is used for accessing the data in its data read phase in cycle 9. The SAMM instruction modifies AR1 to 164h in its execute phase in clock cycle 7 itself. In this case since the decode phase of LACC *+ occurs after the execute phase of SAMM, no problem arises. In the decode phase of LACC *+, the value of AR1 is modified to 165h.

In 9th clock cycle, the ADD *+ enters the decode phase. At that time the value of AR1 is 165h and hence it uses this address when it enters the data read phase in cycle 10. It also modifies AR1 to 166h in cycle 9.

5.3.5 Pipeline Conflicts

When more than one pipeline stage requires processing on the same resource, such as memory and CPU registers, a pipeline conflict occurs. There is no priority between these four phases and unexpected results are obtained when pipeline conflict occurs. Therefore, conflict between these four phases should be avoided in order to get the correct results.

Since the 'C5X only has one set of external address and data buses, a bus conflict occurs between instruction fetch (F), operand read (R), and execute (E) write phases if both program and data memory are external. While the bus conflict is occurring, a dummy operation can be inserted to eliminate the bus conflict. Example 5.8 shows pipeline operation with a bus conflict and a dummy operation.

Example 5.8 In the following program assume that the DP and the current AR point to memory in the external memory space.

```
LDP 120h
LACC 50h
ADD *+
SACL *+
NOP
LACC 51h
```

The instruction pipeline when this program is executed is shown in Table 5.9.

Table 5.9 *Pipeline operation with external bus conflicts*

Cycle	*PC*	*Fetch*	*Decode*	*Read*	*Execute*
1	[LACC]	LDP			
2	[ADD]	LACC	LDP		
3	[SACL]	ADD	LACC	LDP	
4	[SACL]	Dummy	ADD	LACC	LDP
5	[NOP]	Dummy	Dummy	ADD	LACC
6		SACL	Dummy	Dummy	ADD
7		NOP	SACL	Dummy	Dummy
8			Dummy	SACL	Dummy
9				Dummy	SACL
10					Dummy

In the operand read (R) phase of LACC, a bus conflict occurs with the fetch of SACL. Therefore, a dummy fetch operation is inserted. In the next fetch (F) phase, the SACL has a bus conflict with the ADD operand read (R) phase. Therefore, the fetch of SACL is delayed again by one cycle. Two dummy instruction fetches are inserted between ADD and SACL because of this delay. A similar situation occurred in the execute (E) phase of SACL. Since external memory writes take three cycles, during the execution of SACL any instruction fetch or operand read access on the external bus will be delayed for three cycles.

Review Questions

5.1 Explain when a program can make all the four functional units in the CPU to do productive work in all the cycles.

5.2 Explain with reference to the C5X instruction pipeline why the branch and call instructions require four clock cycles for program control transfer

5.3 When are the unexpected results obtained when the load/store instruction of a AR is contained in a C5X program? How is the problem overcome?

5.4 Explain how pipeline conflicts occur when the program memory and data memory spaces of a program are contained in external memory space.

5.5 Execute the program in Example 5.6 in a C5X system (e.g. starter kit) and verify that the AR1 and ACC are as shown in Table 5.7.

5.6 Execute the program in example 5.7 in a C5X system (e.g. starter kit) and verify that the AR1 and ACC are as shown in Table 5.8.

5.7 Draw the table showing the content of the instruction pipeline when each of the following programs are executed.

Program 1

```
LDP     20h
LACC    10h
SUB     30h
SACL    11h
CMPL
SACL    12h
```

Program 2

```
LAR     1200h
LDP     20h
ZAP
ADD     30h
ADD     *+
SACL    #1300h
NEG
```

Program 3

```
        LDP      20
        LACC     30h
        SUb      #30h
        BCND     YY, GT
        B   YY
XX      NEG
YY      SACL     30h
```

Program 4

```
        MAR *, AR1
        LDP 20h
        LACC 10h
        SUB 30h
        SACL 11h
        CALL   YY
XX      LACC *+1
        ADD *+
YY      MAR *+, AR2
```

Program 5

```
        MAR *, AR1
        LDP 20h
        LACC 10h
        SUB 30h
        SACL *
        RET
XX      LACC *  +1
        ADD *+
```

Program 6

```
        LAR ARI,    1200h
        LAR AR2,    1300h
        LAR AR3,    02h
        MAR *,      ARI
XX      LACC *+,    AR2
        SACC *+,    AR3
        BANZ XX,    AR1
        ADD *-
        ADD *-
```

Program 7

```
LAR    ARl,#167h
LACC   #164h
SAMM   AR2
LACC   *+
ADD    *+
SACL   *+
LAMM   AR2
```

Program 8

```
LAR    ARl,#167h
LACC   #164h
SAMM   AR1
NOP
LACC   *+
ADD    *+
SACL   *+
LAMM   AR1
```

Self Test Questions

5.1 The program containing ——— instructions will ensure perfect overlapping of the operations in the four stages of the instruction pipeline of 5X.
(a) single-word single-cycle
(b) both single-word and double-word
(c) delayed branch
(d) delayed call

5.2 The number of clock cycles required for flushing out the pipeline in the case of execution of branch instruction B begin is ———.
(a) 0 (b) 1 (c) 2 (d) 3

5.3 The number of clock cycles required for flushing out the pipeline in the case of execution of delayed branch instruction BD BEGIN is ———.
(a) 0 (b) 1 (c) 2 (d) 3

5.4 The number of single-cycle instructions which can be executed after the delayed branch instruction of 5X before the execution begins the new branch address is ———.
(a) 0 (b) 1 (c) 2 (d) 3

5.5 The number of double-cycle instructions which can be executed after the delayed branch instruction of 5X before the execution begins the new branch address is ———.
(a) 0 (b) 1 (c) 2 (d) 3

5.6 Which of the following instructions does not require the instruction pipeline to be flushed out before executing additional instructions?
(a) CC cond when cond true
(b) CC cond when cond false
(c) RETC cond when cond true
(d) RETC cond when cond false

5.7 In which phase of the instruction pipeline, the AR is modified when the instruction LACC *+ is executed?
(a) Fetch (b) Decode (c) Read (d) Execute

5.8 In which phase of the instruction pipeline, AR is modified when the instruction LAR AR0 #1000h is executed?
(a) Fetch (b) Decode (c) Read (d) Execute

5.9 In 5X programs using indirect addressing mode, to ensure proper operation the number of single-word instructions that should be inserted between the instruction which loads an AR and an instruction which fetches the operand using this AR is ———.
(a) 1 (b) 2 (c) 3 (d) 4

5.10 An external memory used with 5X requires three clock cycles, the number of dummy operations carried out by the 5X CPU to avoid pipeline conflict is ———.
(a) 1 (b) 2 (c) 3 (d) 4

6 APPLICATION PROGRAMS IN C5X

In this chapter, some application programs on 5X are given which illustrate the diverse applications to which the P-DSPs can be used. For testing these programs with real time inputs and to verify that the expected results occur, we require a suitable experimental set-up. This should have a provision for loading the program into the P-DSP, digitise the analog input and store it into the P-DSP memory and execute the program which simultaneously processes the data and keep communicating with the input/output units when required. For this purpose it will be assumed that the experimental set-up/development environment used is the TI's DSP starter kit (DSK) for 5X. The programs given here would also work in other 5X-based kits. However, the external memory used, if any, and the A/D and D/A converters used in the other kits have to be taken into account for making this program work in the other kits. However, many of the other kits also use the same A/D and D/A converter and no external memory is used. Hence, the programs given in this chapter would work without any modification in the other kits as well.

′C50-BASED DSP STARTER KIT (DSK) 6.1

6.1.1 Block Diagram of the DSK

Figure 6.1 depicts the block diagram of the C50-based DSK starter kit. The host interface permits the application program to be downloaded from a host (Personal Computers, e.g.) to the P-DSP. PC communications are via the RS-232 port on the DSK board. The 2K bytes of on-chip PROM of C50 contain the kernel program for boot loading. This facilitates execution as well as debugging of the programs. The analog interface permits the audio input signals in the frequency range 0-9.6 kHz to be digitised using the analog interface circuit (AIC) TLC32040 AIC on the board. The AIC also permits the processed signal to be converted into an analog signal. The AIC has the following characteristics:

- Single-chip digital-to-analog (D/A) and analog-to-digital (A/D) conversion with 14 bits of dynamic range
- Variable D/A and A/D sampling rate and filtering
- The AIC interfaces directly to the ‘C50 serial port
- The master input clock to the AIC is provided by a 10.368 MHz timer output from the ‘C50
- The maximum sampling rate of the AIC is limited to 19.2 kilo samples/s.
- The AIC is hard-wired for 16-bit word mode operation

Additional details on AIC TLC32040 are given in Section 6.4.3.

In addition to the above, DSK also has an emulation interface. All pins of the ‘C50 are connected to the external I/O interfaces. The external I/O interfaces include four 24-pin headers, a 4-pin header and a 14-pin header.

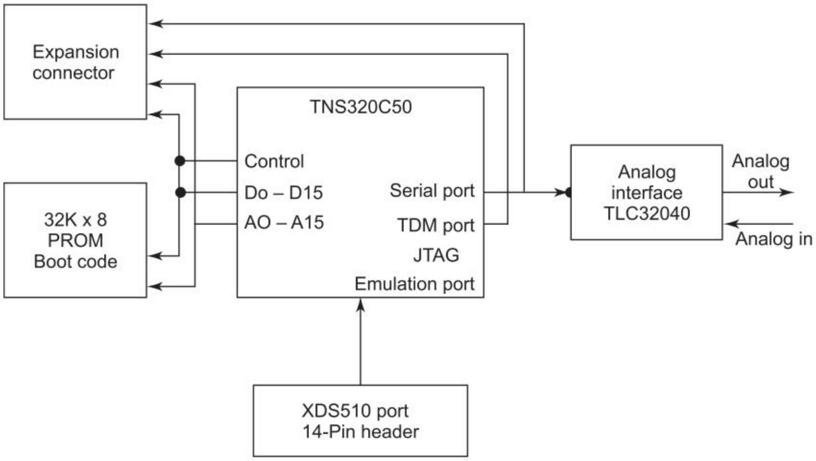


Fig. 6.1 *C5X DSK block diagram*

6.1.2 Memory in DSK

The ′C5X DSK does not have any external memory on the board. However, the 10K on-chip RAM of the ′C50 provides enough memory for most DSP application programs. The kernel program is contained in the 2K, 8-bit PROM. The PROM is only for DSK boot loading and cannot be accessed after boot loading, as this portion of the on-chip memory is reserved for the kernel program. Figure 6.2 shows the memory map of the ′C5X DSK.

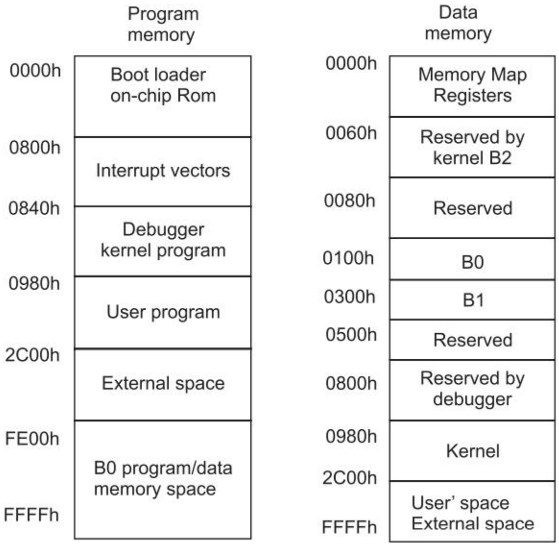
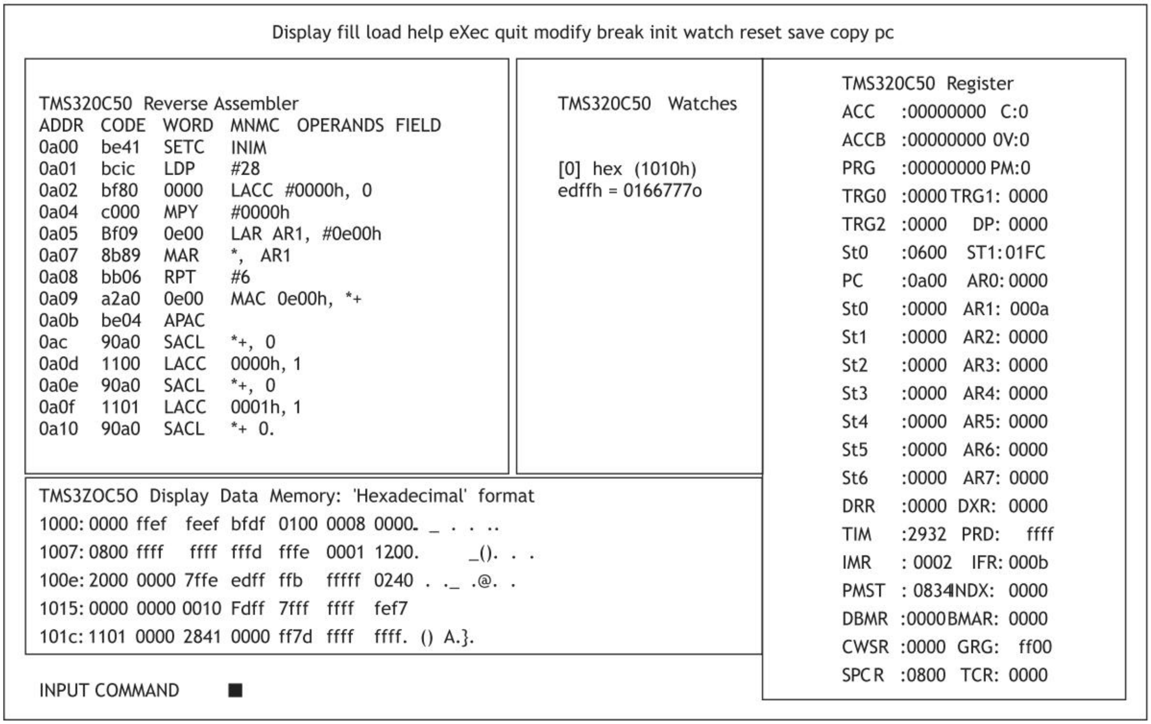


Fig. 6.2 *Memory map for C5X DSK*

The on-chip, dual-access, random-access-memory (DARAM) B2 is reserved as a buffer for the status registers. The single-access, random-access-memory (SARAM) is configured as program and data memory. The kernel program is stored in this area from 0X840h-0X980h. If the kernel program performs an overwrite, a reset signal is required to let the DSK reload the kernel program. Since the kernel program is stored in the SARAM, this on-chip memory cannot be configured as data memory only (RAM = 0). The interrupt vectors are allocated, starting from 0X800h. The IPTR in the PMST register should not be modified by the programmer. B0 may be configured as either program or data memory, depending on the value of the CN bit in status register ST1.

6.1.3 Development Environment in DSK

The ‘C5X DSK has a PC windows-oriented debugger that makes it easy to develop and debug software code. The DSK communicates with the PC using the XF and BIO pins through the RS-232 serial port. Figure 6.3 shows the display of the debugger screen. The DSK has its own assembler.



Display fill load help eXec quit modify break init watch reset save copy pc

```
TMS320C50  Reverse Assembler
ADDR  CODE  WORD  MNMC  OPERANDS  FIELD
0a00  be41  SETC  INIM
0a01  bcic  LDP   #28
0a02  bf80  0000  LACC  #0000h,  0
0a04  c000  MPY   #0000h
0a05  Bf09  0e00  LAR  AR1,  #0e00h
0a07  8b89  MAR   *,   AR1
0a08  bb06  RPT   #6
0a09  a2a0  0e00  MAC  0e00h,  *+
0a0b  be04  APAC
0ac   90a0  SACL  *+,  0
0a0d  1100  LACC  0000h, 1
0a0e  90a0  SACL  *+,  0
0a0f  1101  LACC  0001h, 1
0a10  90a0  SACL  *+  0.
```

```
TMS320C50   Watches

[0]  hex  (1010h)
edffh = 0166777o
```

```
TMS320C50  Register
ACC    :00000000   C:0
ACCB   :00000000 0V:0
PRG    :00000000 PM:0
TRG0   :0000 TRG1:  0000
TRG2   :0000    DP:  0000
St0    :0600   ST1: 01FC
PC     :0a00   AR0: 0000
St0    :0000   AR1:  000a
St1    :0000   AR2:  0000
St2    :0000   AR3:  0000
St3    :0000   AR4:  0000
St4    :0000   AR5:  0000
St5    :0000   AR6:  0000
St6    :0000   AR7:  0000
DRR    :0000  DXR:   0000
TIM    :2932   PRD:     ffff
IMR    : 0002   IFR: 000b
PMST   : 0834NDX:   0000
DBMR   :0000BMAR:  0000
CWSR   :0000  GRG:    ff00
SPCR   :0800  TCR:  0000
```

```
TMS3ZOC5O  Display  Data  Memory:  'Hexadecimal'  format
1000: 0000 ffef  feef  bfdf  0100  0008  0000. _  .  .  ..
1007: 0800  ffff   ffff  fffd  fffe  0001  1200.       _().  .  .
100e: 2000  0000  7ffe  edff  ffb   fffff  0240  .  ._  .@.  .
1015: 0000  0000  0010  Fdff  7fff  ffff   fef7
101c: 1101  0000  2841  0000  ff7d  ffff   ffff.  ()  A.}.
```

```
INPUT COMMAND    ■
```

Fig. 6.3 *TMS320C5X debugger screen display*

6.1.4 Assembling a Program

For programming in the assembly language the C5X assembler assumes the following assembly language syntax. A source statement can contain four ordered fields. The general syntax for source statements is as follows:

[label] [:] mnemonic [operand list][;comment]

They in turn follow these guide lines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon

- Labels are optional; if used, they must begin in column 1
  Labels may be placed either before the instruction mnemonic on the same line or on the preceding line in the first column
- one or more blanks must separate each field. Tab characters are equivalent to blanks
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column must begin with a semicolon

The following types of operands are permitted:

0 < dma < 127	dma: Data Memory Address
0 < pma < 65535	pma: Program Memory Address
0 < shift < 15	
0 < shift2 < 7	
0 < *n* < *1*	*n:* AR no.
0 < k < 255	k: Short Constant
0 < lk < 65535	lk: Long Constant

ind: {*, *+, *-, *0+, *0-, *BR0+, *BR0-}

Operands can be constants or assembly-time expressions that refer to memory, I/O ports, register addresses, pointers, shift counts and a variety of other constants.

The mnemonics for the various instructions of C5X is given in Appendix A4 of Chapter 4. In addition to this the assembler of C5X permits a set of assembler directives or pseudo instructions which simplifies the programming in assembly language. For example, the .ps assembler directive specifies the program memory address from where the program should be loaded and the execution should begin. Similarly the .mmregs directive permits the memory mapped registers to be referred by their short names instead of their actual memory address. The list of assembler directives is given in Appendix 6.1. The list of memory mapped registers and their actual memory addresses is given in Appendix 6.2.

Even though there are many assembler directives in Appendix 6.1, only some of them are more frequently used. Some of these pseudo instructions are given in Table 6.1. Sample programs in assembly language are given in Program 6_1a.asm and Program 6_1b.asm. To create assembly source file, any ASCII program editor can be used. The file name should have the extension **.asm.**

Table 6.1 *Some most commonly used pseudoinstructions*

Pseudo instruction	*Description*
.mmregs	Includes memory map registers
.ps XXXX	Assemble into program memory address XXXX
.entry	Initialise the starting address of the program counter when loading a file
.include "yy.asm"	This enables a list of instructions in program yy.asm to be inserted in the place where this directive appears
•end	Program end

Program 6_1a.asm

```
Pseudo instruction                    Description
.mmregs                               ;includes memory map registers
.ps 0a00h                             ;assemble with origin as program memory
```

```
                                            ;address as 0a00h
            .entry                          ;initialise the value of the program
                                            ;counter when loading a file
            LACC #1000h                     ;accumulator loaded with the constant 1000h
            LACC #0004h,4                   ;the constant 4h is left shifted by four
                                            ;bits and leaded to accumulator
            LAR AR0,#1000h                  ;AR0 loaded with constant 1000h
loop:       b loop                          ;infinite loop
            .end                            ;program end
```

Program 6_la.asm may be rewritten using the .include directive as shown in Program 6_lb.asm; include directive is useful if a no. of instructions are commonly used in a no. of programs. For example, some of the initialisation instructions may be commonly required for a no. of programs. These common instructions may be put in a single file and this file can be included wherever required. The .include directive reduces the length of the source program but the object code is not reduced. This is because at the time of assembly, wher ever .include statement occurs, the instructions in the included file are inserted and assembled as if they appeared in the file being assembled.

Program 6_lb.asm

```
            .mmregs                         ;includes memory map registers
            .ps 0a00h                       ;assembles with origin of program memory
                                            ;address as 0a00h
            .entry                          ;initialise the program counter when
                                            ;loading a file
            .include “init.asm”             ;3 lines corresponding to init.asm are
                                            ;inserted here at the time of assembly
loop        b loop                          ;infinite loop
            . end                           ;program end
```

init.asm

```
            LACC #1000h                     ;accumulator loaded with the constant
                                            ;1000h
            LACC #0004h,4                   ;the constant 4h is left shifted by
                                            ;four bits and loaded to accumulator
            LAR AR0,#1000h                  ;AR0 loaded with constant 1000h
```

The program written in source code using mnemonics and pseudo instructions has to be assembled to generate the program in the object code of C5X. The object code is loaded into the DSK using the debugger and it can be executed using the debugger. The command for invoking the assembler when preparing a program for debugging is:

dsk5a [filename(s)][-options]

dsk5a : The command that invokes the assembler,

filenames : One or more assembly language source files. Filenames are not case sensitive.

-options : affect the way the assembler processes input files.

Options and filenames can be specified in any order on the command line. Table 6.2 lists the assembler options.

Table 6.2 *Summary of assembler options*

Option	*Description*
–k	Generates an output file regardless of errors or warnings
–I	Generates a temporary file containing a list of any unresolved opcodes or symbols
asm	Allows the user to define assembler statements from the command line

Program 6_la. asm can be assembled using the command **dsk5a program 6_1a**

6.1.5 Using the DSK Debugger

After successfully assembling the program, the object code is to be loaded into the DSK by invoking the debugger using the command:

```
dsk5d ci
```

if DSK is connected to serial communication port i (comi) of PC. The legal values of ci are cl, c2, c3 and c4 corresponding to coml, com2, com3, com4. For example, if DSK is connected to serial communication port 2 (com2) of PC, the command entered should be

```
dsk5d c2
```

The default setting is cl. In that case the command **dsk5d** would suffice. After entering the dsk5d command, the display similar to the one shown in Fig. 6.3 appears on the PC screen. It can be seen that the debugger screen has 6 sections/windows. The details presented in each of the windows are as followed:

Command List Window Displays the list of debugger commands. For each of the commands only a single character needs to be entered for invoking. The character to be entered is shown in boldface form.

Reverse Assembler Window It shows the disassembly of the code presently stored in the program memory specified by the user. Default program memory address is taken as 0a00h.

Watch Window In this, the content of one or memory locations may be displayed as the program is being executed.

Register Window The content of various internal registers of C5X is displayed.

Command Window This permits the command to be entered.

Dialog (box) Window This permits the additional parameters required for specifying a command. It is also used for displaying as well as modifying any one of the C5X registers and the memory locations.

Normally debugging phase starts with loading a file with extension .DSK from the PC to the C5X using the load command. Then this file is executed using the execute command. The execution may be done either using single-step mode or using break points. In the learning phase it is helpful to execute the program in single-step mode and examine the register window to verify that the various registers are altered by the DSK as expected. After gaining confidence in programming, breakpoints may be set at points which are critical in the program. More details on using the various commands of the DSK can be obtained from the 5X DSK user's guide [1996].

PROGRAMS FOR FAMILIARISATION OF THE ADDRESSING MODES 6.2

6.2.1 Immediate Addressing Mode

Program 6_2.asm gives an example of an assembly language program in which the instruction word contains the value of the immediate operand. The immediate operand is used for the accumulator (ACCU), auxiliary registers (AR0 & AR1) load, memory mapped register store, addition, subtraction and multiply operations. It is also possible in some instructions to left shift the immediate operand defined by the shift code followed by the operand and the respective operations can be performed. For the memory mapped registers there is no need to specify their hexadecimal address; the abbreviation of the respective registers can be written in the instruction.

Program 6_2.asm Immediate addressing mode

```
label           Mnemonic              Comments
                .mmregs               ;includes memory map registers
                .ps 0a00h             ;assemble with origin of the program
                                      ;memory address as 0a00h
                .entry                initialise the starting value of the
                                      ;program counter when loading a file
                LACC #1000h           ;value 1000h loaded into accumulator
                LACC #1111h,3         ;the constant 1111h left shifted by 3
                                      ;bits & loaded into ACCU. The contentof
                                      ;ACCU after execution is 8888h
                LAR AR0,#1000h        ;AR0 is loaded with the constant 1000h
                LAR AR1,#1100h        ;1100h is loaded into AR1
                ADD #00FFh            ;FFh is added to the content of ACCU and
                                      ;the result is stored into ACCU
                ADD #0011h,2          ;0011h left shifted by 2 bits & added to
                                      ;ACCU
                SPLK #10h,TREG0       ;10h is stored into register TREG0
                MPY #0010h            ;0010h is multiplied with the content of
                                      ;TREG0 and the result stored into the
                                      ;product register (PREG)
                SUB #0022h            ;0022h is subtracted from ACCU
                SUB #0011h,3          ;0011h left shifted by 3 bits & sub
                                      ;tracted from the content of ACCU
                .end                  ;program end
```

6.2.2 Direct Addressing Mode

In direct addressing mode, the instruction word contains the location of the operand in a particular data page. The starting address of the operand page is indicated by the data page pointer (DP). It is loaded using the LDP instruction. Program 6_3.asm gives an example illustrating how the operations such as load accumulator and ARs, store the operands in data memory, add, subtract and multiply can be performed in direct addressing mode. It is also possible in some instructions to left shift the data memory content defined by the shift code in the instruction and the respective operations can be performed.

Program 6_3.asm Direct addressing mode

```
label        Mnemonic             Comments
             .mmregs              ;includes memory map registers
             .ps 0a00h            ;origin of the program 0a00h
             .entry               ;program counter initialised
             LDP #20h             ;the data page no. 20h (32) is loaded into
                                  ;the data page pointer (DP)
             LACC 10h             ;content of 20h (32) page 10 th location
                                  (i.e. ;content of data memory address
                                  (dma) ;(1010h) is loaded into ACCU
             LACC 5h,2            ;content of dma 1005h is left shifted by 2
                                  ;bits and then loaded into ACCU
             LDP #22h             ;DP loaded with 22h (dma page starting
                                  ;address 1100h)
             LAR AR0,15h          ;AR0 loaded with content of dma 1115h
             SACL 15h             ;ACCU low byte stored into dma 1115h
             SACL 20h,3           ;ACCU low byte left shifted by 3 bits &
                                  ;stored into dma 1120h
             SAMM AR7             ;ACCU low byte stored into AR7 in
                                  ;page 0. DP remains unaffected
             LDP #12h             ;the data page no. 12h loaded into DP
             ADD 25h              ;the content of dma 0925h added to
                                  ;ACCU & the result stored into ACCU
             ADD 7h,2             ;the content of dma 0907h left shifted by
                                  ;2 bits & added to ACCU
             SUB 10h              ;the content of 0910h is subtracted from
                                  ;the content of ACCU
             SUB 12h,2            ;the content of 0912h is left shifted by 2
                                  ;bits & subtracted from ACCU
             SPLK #10h,TREG0      ;constant 10 stored into TREG0
             MPY 15h              ;content of 0915h is multiplied with
                                  ;TREG0 & the result stored into PREG
             .end                 ;program end
```

6.2.3 Indirect Addressing Mode

The address of the operand in indirect addressing mode is the content present in the current AR. The current AR is indicated by the auxiliary register pointer (ARP). Program6_4.asm gives an example for loading and storing of various registers and data memory locations, as well as performing arithmetic operations such as add, subtract and multiply. This program also illustrates how INDX register is used to modify the address of the operand and how bit-reversed addressing can be used.

Program 6_4.asm Instructions using indirect addressing mode

```
label           Mnemonic                  Comments
                .mmregs                   ;includes memory map registers
                .ps 0a00h                 ;origin of the program 0a00h
                .entry                    ;initialise the program counter
                LAR AR0,#1000h
                LACC *                    ;content of dma pointed by AR0
                                          ;(i.e. 1000h) is loaded into ACCU
                LACC *,4,AR1              ;content of dma 1000h left shifted by 4
                                          ;bits & loaded into ACCU, ARP points to
                                          ;auxiliary register 1
                LAR AR1 #1010h
                SACL *                    ;ACCU low byte stored into the dma
                                          ;pointed by AR1 i.e. 1010h
                SACL *+,2,AR0             ;ACCU low byte is left shifted by 2 bits &
                                          ;stored into dma pointed by AR1, AR1 is
                                          ;incremented by one , ARP points to AR0
                LACC *-,2,AR1             ;ACCU loaded with the 2 bits left shifted
                                          ;content of dma pointed by AR0, AR0
                                          ;decremented ARP points to AR1
                LACC *0+                  ;ACCU loaded with the content of dma
                                          ;pointed by AR1, AR1 is incremented by
                                          ;the content of INDX register
                LACC *BR0+                ;ACCU loaded with the content of dma
                                          ;pointed by AR1 and the content of INDX
                                          ;register added to AR1 with reverse carry
                                          ;propagation
                ADD *+,0,AR0              ;content of dma pointed by AR1 is added to
                                          ;the content of ACCU, AR1 is incremented
                                          ;and ARP points to AR0
                SUB *-,2                  ;content of dma pointed by AR0 left
                                          ;shifted by 2 bits & subtracted from the
                                          ;content of ACCU, the result is stored
                                          ;into ACCU, AR0 is decremented by 1.
                SPLK #10h,TREG0
                MPY *                     ;content of dma pointed by AR0 is
                                          ;multiplied with the content of TREG0 and
                                          ;the result is stored into PREG, the
                                          ;content
                                          ;of AR0 & ARP are not modified
                .end                      ;program end
```

Data Block Move In this example two data blocks are considered in the data memory space whose starting addresses are 1000h and 1100h respectively and each of them have 10 data values. The indirect addressing mode is used to exchange the data content present in these locations. Two ARs are used to indicate the address values of the blocks. The data value from one block is loaded into ACCU and exchanged with the contents of ACCB. Then the content from the other block is loaded into ACCU and stored in the starting address of the first block. Now the first block data which is present in ACCB is loaded into ACCU and stored in the starting address of the second block. The ARs can be incremented and repeated operation will exchange the data block contents. The resulting program is given in Program 6_5.asm. This program may be modified to copy the content of data memory to program memory and vice versa.

Program 6_5.asm Block move using indirect addressing

```
label          Mnemonic                       Comments
               .mmregs
               .ds 1000h
               .word 1h,2h,3h,4h,5h
               .word 6h,7h,8h,9h,0ah;         ;10 data values are stored in block 1
               .ds 1100h
               .word 11h,12h,13h,14h,15h      ;10 data
               .word 16h,17h,18h,19h,20h      ;values are stored into block 2
               .ps 0a00h
n              .set 09h                       ;size of the block is assigned to the variable n
               .entry
               LAR AR0,#1000h
               LAR AR1,#1100h
               LAR AR2,#n                     ;block size is loaded in AR2
loop           LACC *,0,AR1                   ;block 1 data value is loaded into ACCU
               EXAR                           ;content of ACCU and ACCB are exchanged
               LACC *,0,AR0                   ;block 2 data value is loaded into ACCU
               SACL *+,0,ARl                  ;block 2 data value is stored in block 1
               LACB                           ;ACCB content loaded back to ACCU
               SACL *+,0,AR2                  ;block 1 data value is stored in block 2
               BANZ loop,AR0                  ; branch to loop until content of AR2
                                              ;decrements to zero
               NOP
               NOP
               .end
```

6.2.4 Circular Addressing Mode

To define circular buffers, start and end addresses are loaded into the corresponding buffer registers, a value between start and end address is loaded into an AR. Next the AR which is used for circular addressing and the circular buffer which is used for the addressing are chosen by programming the appropriate bits in the CBCR. In the example shown in Program 6_6.asm, the start and end addresses are 1000h & 1003h respectively. The auxiliary register AR0 and circular buffer 1 are used for the circular addressing. For this 0008h is to be loaded into CBCR register. When the instructions using indirect addressing mode are used, the content of AR specified in CBCR (i.e. AR0) is incremented. When its value is greater than CBER1, the CBSR1 address is loaded into the auxiliary register AR0.

Program 6_6.asm Program for circular addressing mode

```
label        Mnemonic                Comments
             .mmregs
             .ps 0a00h
             .entry
             SPLK  #1000h, CBSR1     ;load 1000h into the circular buffer
                                     ;start address register 1
             SPLK  #1003h, CBER1     ;load 1003h into the circular buffer end
                                     ;address register 1
             SPLK  #08h, CBCR        ;load 8h into CBCR to activate circular
                                     ;buffer 1 & choose AR0 as the auxiliary
                                     ;register for circular buffer 1
             LACC #AR0, #1000h       ;the start address of the buffer is loaded in AR0
             LACC *+                 ;ACCU loaded from 1000h, AR0 =1001h
             LACC *+                 ;ACCU loaded from 1001h, AR0 =1002h
             LACC *+                 ;ACCU loaded from 1002h, AR0 =1003h
             LACC *+                 ;ACCU loaded from 1003h, AR0 =1000h
             .end                    ;program end
```

PROGRAM FOR FAMILIARISATION OF ARITHMETIC INSTRUCTIONS 6.3

6.3.1 Finding the Sum of *n* Integers

The example given in Program 6_7.asm illustrates how the series 1, 2, 3, ..., *n* may be generated and stored in data memory. It also shows how its sum is calculated. The indirect addressing mode is used for storing and retrieving the numbers.

Program 6_7.asm Finding the sum of *n* integers

```
Label        Mnemonic                Comments
             .mmregs
             .ds 1000h
             .ps 0a00h
             .entry
             LAR AR0,#10h            ;'n' of the sequence loaded; here n =10
             LAR AR1,#1000h          ;starting address (1000h) from where the
                                     ;sequence is stored is loaded into AR1
```

```
                LACC #1h                ;ACCU is initialised with value 1
                MAR *,AR1               ;modify auxiliary register pointer to AR1
loop:           SACL *+,0,AR0           ;sequence generated & stored from data
                                        ;memory address 1000h
                ADD #1h                 ;increment ACCU by 1
                BANZ loop.AR1           ;branch to loop on AR0 not zero; AR0 is
                                        decremented & ARP points to AR1
                LAR AR0,#10h
                LAR AR1,#1000h
                ZAP                     ;zero ACCU and product register
loop1:          MAR *,AR1               ;loop1 starts here: it finds the sum
                ADD *+,0,AR0            ;content of dma starting from 1000h added
                                        ;to ACCU
                BANZ loop1,AR1          ;branch to loop1 on AR0 not zero
                                        ;AR0 is decremented & ARP points to AR1
                .end                    ;program end
```

The value of n is stored in one of the ARs (AR0). First the series is generated and stored by using a loop. Using another loop the sum is calculated. Each time the loop is executed once, the AR having the value of n is decremented once its value decrements to zero the execution completes.

6.3.2 Generation of Fibonacci Series

The generation of the numbers corresponding to Fibonacci series is achieved using Program 6_8.asm. If the first two numbers of the sequence are assumed, the next number in the series is the sum of the previous two numbers. For example, $x(0) = 0$, $x(l) = 1$ then $x(i) = x(i-1) + x(i-2)$, where $i>1$. The length of the sequence 'n' is stored into one of the AR (AR1). The previous two numbers are needed for the generation of the next number. One number is stored in ACCU and another number is stored into the accumulator buffer ACCB. The sum of these two registers give the next number in the series. Each time the next number is calculated, the content of AR1 is decremented, once the content of AR1 decrements to zero the generation of the additional numbers in the sequence is stopped.

Program 6_8.asm **Generating the Fibonacci series numbers**

```
Label           Mnemonic                Comments
                .mmregs
                .ds 1000h
                .ps 0a00h
                .entry
                LAR AR0,#1000h          ;starting dma (1000h) stored into AR0
                LAR AR1,#10h            ;sequence length (N =10) loaded into AR1
                LACC #0h                ;zero ACCU
                EXAR                    ;contents of ACCU & ACCB exchanged
                LACC #01h               ;load the constant 1 into ACCU
```

```
loop:              MAR *, AR0              ;modify ARP to AR0
                   SACL *,0                ;store ACCU low byte in dma pointed by AR0

                   ADDB                    ;the content of ACCB added to ACCU
                   EXAR                    ;contents of ACCU & ACCB exchanged
                   SACL *+,0,AR1           ;store the ACCU low byte in dma pointed
                                           ;by AR0, increment the content of AR0 and
                                           ;ARP points to AR1
                   BANZ loop,AR0           ;branch to loop if AR1 non-zero AR1
                                           ;decremented & ARP points to AR0
                   .end                    ;program end
```

6.3.3 Convolution Using MAC and MACD Instructions

The convolution of the following two sequences of length N and M is performed using MAC and MACD instructions in the following examples.

Example 6.1 Convolve two sequences $x(n)$, $y(m)$ of length N and M respectively for $N = 5$, $M = 3$ and $x(n)$, $y(m)$ given by

$X(n)$ = 1, 2, 3, 2, 1 where $0 \le n \le 4$
$Y(m)$ = 3, 4, 5 where $0 \le m \le 2$

Solution The sequence after convolution is

$z(l) = x(n) * y(m)$ = 03, 0A, 16, 1C, 1A, E, 05 where $0 \le l \le 6$

6.3.3.1 Convolution Using MAC Instruction

Program 6_9.asm gives the listing of the program for convolving two sequences $x(n)$ and $y(m)$ given in example 6.1. One of the sequences is loaded into program memory space and another one in data memory space using .word assembler directive. For one of the sequences ($x(n)$ in the above example) padding of zeros is not required, but for the other sequence, ($y(m)$ in the above example) padding of zeros is necessary. The sequence which is to be stored into data memory should be padded with zeros. The number of zeros to be padded is $N - 1$, at the beginning and at the end of the $y(m)$ sequence. The starting address of the $x(n)$ sequence is indicated in the MAC instruction (0a00h). The starting address of the $y(m)$ sequence is to be loaded into an AR (AR0). Since zeros are padded, the starting address for convolution is not equal to the address which is indicated in the .ds assembler directive (1000h), but it is $(N - 1)$ locations ahead of it, i.e. $(1000 + (N - 1))$. The length of the convolved sequence is

Program 6_9.asm **Convolution using MAC**

```
Label              Mnemonic                Comments
                   .mmregs
                   .ps 0a00h
                   .word 1h,2h,3h,2h,1h    ;x(n) stored from pma 0a00h
                   .ds 1000h               ;origin of dma is 1000h
                   .word 0h,0h,0h,0h       ;y(m) stored from dma 1000h after
```

```
                  .word 3h,4h,5h               ;padding 4 zeros at the beginning
                  .word 0h,0h,0h,0h
                  .entry
                  LAR AR0,#1004h               ;actual data starts only at 1004h
                  LAR AR1,#1050h               ;starting address for result: 1050h
                  LAR AR2,#07h                 ;length of the output sequence
loop:             ZAP
                  MAR *,AR0
                  RPT #5h                      ;execute the instruction followed by RPT
                                               ;instruction 5 times
                  MAC 0a00h,*-                 ;the PREG content is added to ACCU.
                                               ;contents of pma given by PFC & dma multiplied
                                               ;& product loaded into PREG pma is
                                               ;incremented & dma pointer decremented
                                               ;(see note 1)
                  MAR *,AR1                    ;AR pointed to output dma
                  SACL *+,0,AR0                ;one result stored
                  ADRK #7h                     ;add 7 to the current AR i.e. AR0
                  MAR *,AR2
                  MAR *-
                  BANZ loop
                  .end
```

$N + M - 1$ ($l = 7$). This is loaded into an AR (AR2). The constant that is to be put in RPT instruction is N. The constant that is to be added to AR0 each time in the execution of the loop is $N + 2$. The convolved sequence is stored in the data memory location. Its starting address is indicated in an AR (AR1). The same procedure can be extended for any sequence and its convolution can be obtained.

Note 1: When the MAC instruction is executed for the first time, the pma given in the MAC instruction is copied into the prefetch counter (PFC) for the program memory and the contents of pma given by PFC are multiplied with that of the dma. After this operation PFC is incremented and dma pointer is incremented /decremented depending upon whether the MAC instruction has *+ or *- as the argument for the indirect addressing mode. When the MAC instruction is repeatedly executed, when the MAC instruction is executed for the second time onwards, the pma given in the MAC instruction is not copied into PFC. The pma address is directly taken from the PFC.

6.3.3.2 Convolution Using MACD Instruction

Program for the convolution of the two sequences given in Example 6.1 using MACD instruction is given in Program 6_10.asm. As in Program 6_9.asm, out of the two sequences $x(n)$ and $y(m)$, one is loaded into program memory space and another one in data memory space using .word assembler directive. For both the sequences $x(n)$ and $y(m)$, padding of zeros is necessary. For $x(n)$ sequence zeros are to be padded at the end only. But for the $y(m)$ sequence zeros are to be padded at the beginning and at the end. The number of zeros to be padded for $x(n)$ sequence is $(M - 1)$, whereas for $y(m)$ it is $(N - 1)$.

The starting address of the $x(n)$ sequence is indicated in the MACD instruction (0a00h). For MACD instruction the end address of the $y(m)$ sequence is to be loaded into an AR (AR0 in this program).

Since zeros have been padded, to find the end address of dma $M + 2(N - 1)$ is added to the address indicated in the .ds assembler directive. The length of the convolved sequence is $N + M - 1$ ($l = 7$). This is loaded into an AR (AR2). The constant that is to be put in the RPT instruction is ($N + M - 1$). The convolved sequence is stored in the data memory. Its starting address is indicated by an AR (AR1). Each time the loop is executed, the end address of the sequence in data memory space is reloaded. Since convolution sum computation starts with the last value of the resultant sequence, the resultant sequence is stored from 1050h,104Fh to 104Ah using *– option in SACL instruction. The same procedure can be extended to any sequence and its convolution can be obtained. **MACD instruction is useful for real time convolution as the input data array is shifted towards right (to higher memory location) and the new input data can be inserted at the beginning of the dma.**

Program 6_10.asm Convolution using MACD instruction

```
Label           Mnemonic                Comments
                .mmregs
                .ps 0a00h
                .word 1h,2h,3h,2h       ;x(n) sequence is stored in program
                .word 1n,0h,0h          ;memory from the address 0a00h
                .ds 1000h               ;origin of data memory: 1000h
                .word 0h,0h,0h,0h       ;y(n) sequence is stored in data
                .word 3h, 4h,5h         ;memory with 4 zeros added at the
                .word 0h, 0h,0h,0h      ;beginning and end of the sequence
                .entry
                LAR AR1,#1050h          ;output sequence length
                LAR AR2,#07h            ;output sequence length
loop:           LAR AR0,#100ah          ;the end address of the sequence y(n)
                                        ;stored in DM space is loaded into AR0
                ZAP
                MAR *,AR0
                RPT #7
                MACD 0a00h,*-           ;the PREG content is added to ACCU.
                                        ;contents of pma given by PFC& dma multiplied
                                        ;& product loaded into PREG PFC is
                                        ;incremented & dma pointer decremented.
                                        ;content of dma pointed by ARO moved to
                                        ;dma+l (see note 2);
                MAR *,AR1
                SACL *-,0,AR2
                BANZ loop,AR0
                .end
```

Note: When the MACD instruction is executed for the first time, the pma given in the MACD instruction is copied into the PFC for the program memory and the contents of pma given by PFC are multiplied with that of the dma. After this operation PFC is incremented and dma pointer is incremented/ decremented depending upon the whether the MAC instruction has *+ or *– as the argument for the indirect addressing mode. When the MACD instruction is repeatedly executed, when the MACD instruction is executed for the second time onwards, the pma given in the MACD instruction is not copied into PFC. The pma address is directly taken from the PFC. The content of data memory is copied to the next higher location each time the MACD instruction is executed.

PROGRAMS IN C5X FOR PROCESSING REAL TIME SIGNALS 6.4

In many DSP applications, the signals to be processed are in analog form. The analog signals are first converted into digital data and then the required processing is done using P-DSPs. After processing, the digital data are once again converted back to analog signals. So it is essential that an A/D converter and a D/A converter are interfaced with the P-DSPs. The digital data to the A/D and D/A converter from/to the P-DSP may be fed using either parallel ports or serial ports. Accordingly they are called as parallel A/D, D/A and serial A/D, D/A respectively. In this section, interfacing the analog interface circuit (AIC) TLC320C40 to the serial port of C5X is considered. This AIC consists of a serial A/D and D/A converter.

The sampling clock to the AIC is generated using the on-chip timer in C5X. The details on programming the timer is presented first. Next, programming the serial port of C5X is considered. Finally the details on the AIC and its programming are considered.

6.4.1 The on-chip Timer in C5X and Programming its Mode

The timer is an on-chip down counter that can be used to periodically generate CPU interrupts. The timer mode is programmed using the timer control register (TCR). The TCR diagram is shown in Fig. 6.4. The significance of some of the frequently used bits of the TCR is shown in Table 6.3. The remaining bits of TCR may be chosen to be 0.

d15 – d12	d11	d10	d9 – d6	d5	d4	d3 – d0
Reserved	Soft	Free	PSC	TRB	TSS	TDDR

Fig. 6.4 *The timer control register diagram*

Table 6.3 *Significance of some of the Bits of the TCR*

TCR bits	*Parameter name*	*Value on reset*	*Description*
d9-d6	PSC	—	Timer prescaler counter bits. These bits specify the count for the on-chip timer. When the PSC is decremented past 0 or the timer is reset, the PSC is loaded with the contents of the TDDR, and the TIM is decremented
d5	TRB	—	Timer reload bit. This bit resets the on-chip timer. When the TRB is set, the TIM is loaded with the value in the PRD and the PSC is loaded with the value in the TDDR. The TRB is always read as a 0
d4	TSS	0	Timer stop status bit. This bit stops or starts the on-chip timer. At reset, the TSS bit is cleared and the timer immediately starts timing. TSS = 0 The timer is started TSS = 1 The timer is stopped
d3-d0	TDDR	0000	Timer divide-down register bits. These bits specify the timer divide-down ratio (period) for the on-chip timer. When the PSC bits are decremented past 0, the PSC is loaded with the contents of the TDDR

The timer is decremented by one at every CLKOUTl cycle. A timer interrupt (TINT) is generated each time the counter decrements to zero. The timer provides a convenient means of performing periodic I/O or other functions. The timer interrupt rate, $\text{TINT}_{\text{rate}}$, is given by

$$\text{TINT}_{\text{rate}} = [t_c *(\text{TDDR} + 1)*(\text{PRD} + 1)]^{-1}$$

where t_c is period of CLKOUTl of C5X. TDDR is a 4-bit register and its value is loaded by writing into the lower order 4 bits of TCR. PRD is a 16-bit memory-mapped register. For the timer interrupt to cause the DSP to branch to the interrupt service routine, the mask bit corresponding to TINT should be unmasked in the IMR. An example program for programming the on-chip timer is given in Program 6.11_asm.

Program 6_11.asm Programming the on-chip timer

```
Timerinit       Splk #020h, tcr           DSP on-chip timer control register
                                          ;programmed to run in free running
                                          ;mode
                Splk #01h, prd            ;timer period register programmed to
                                          ;divide CPU clock by 2
                Ret
```

6.4.2 5X Serial Port Block Diagram and Its Operation

The full duplex on-chip serial port in C5X provides direct communication with serial devices such as codecs and serial A/D converters. The block diagram of the on-chip serial port of C5X is given in Fig. 6.5. Three signals each are used to connect the transmit pins of the transmitting device with the

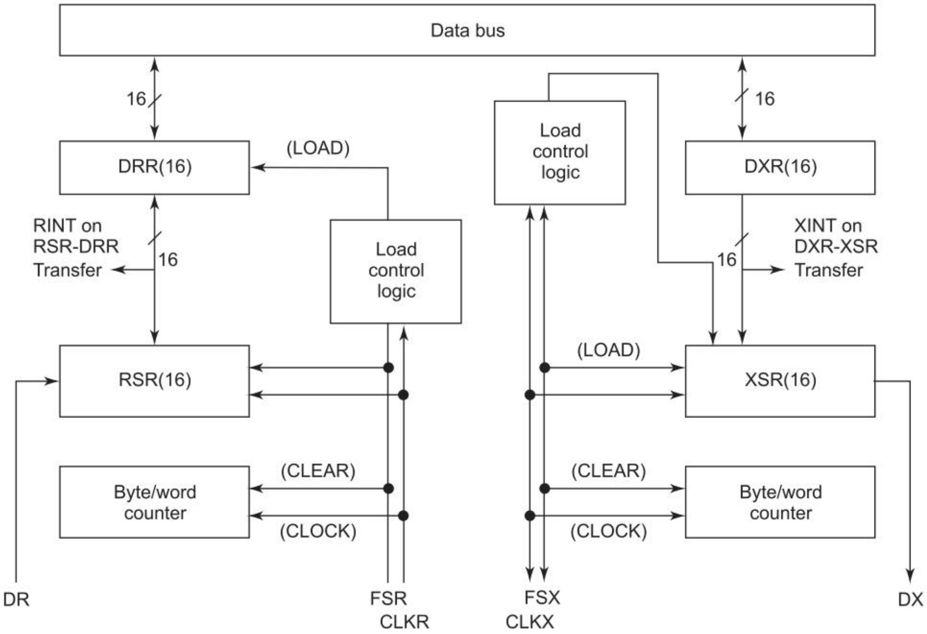


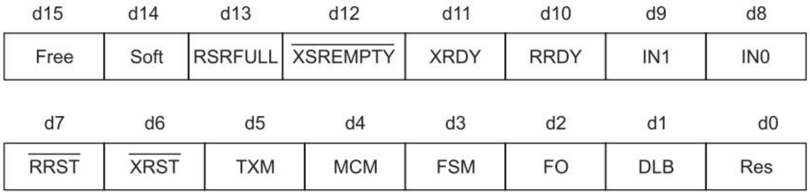
Fig. 6.5 *Serial port interface block diagram*

receiving device and vice versa for data transmission. The pins used for serial communication and their functions are given in Table 6.4.

Table 6.4 *The pins used for serial communication and their functions*

Pin	*Function*
CLKX	Transmit clock signal
DX	Transmit serial data signal
FSX	Transmit frame synchronisation signal
CLKR	Receive clock signal
RX	Receive serial data signal
FSR	Receive frame synchronisation signal

The serial port operates through three memory-mapped registers: serial port control (SPC) register, data transmit register (DXR) and data receive register (DRR) and two other registers: transmit shift register (XSR) and receive shift register (RSR). The details on the bits allocated for various functions of SPC register are given in Fig. 6.6. The description of the bits of SPC is given in Table 6.5.



d15	d14	d13	d12	d11	d10	d9	d8
Free	Soft	RSRFULL	$\overline{\text{XSREMPTY}}$	XRDY	RRDY	IN1	IN0

d7	d6	d5	d4	d3	d2	d1	d0
$\overline{\text{RRST}}$	$\overline{\text{XRST}}$	TXM	MCM	FSM	FO	DLB	Res

Fig. 6.6 *Serial port control (SPC) register diagram*

A transmit is executed by writing data to XSR when XSR is empty (the last word is serially shifted out through the DX pin). The XSR manages the shifting of the data and allows another write to DXR as soon as the DXR to XSR copy is completed. Upon completion of this copy, a 0–1 transition occurs on the XRDY bit in the SPC and generates the serial port transmit interrupt (XINT) and signals that DXR is ready for a new word. The process is similar on the receive side. Data from the DR pin is shifted into the RSR, which copies it into the DRR from which it may be read. Upon completion of the RSR–DSR copy, a 0–1 transition occurs on the receive ready (RRDY) bit in the SPC and generates the serial port receive interrupt (RINT). Thus while a data is being received or transmitted serially another 8- or 16-bit data can be either read or written into the serial port. The signals FSR and FSX initiate the serial shifting of the data at the beginning of every frame in burst mode and for the first frame in the synchronous transfer mode for the shift registers used at the receive and transmit side respectively.

Table 6.5 *The description of the Bits of serial port control register*

Bitname	*Description*
RSRFULL	Receive Shift Register Full. This bit indicates whether the receiver has experienced overrun. Overrun occurs when RSR is full and DRR has not been read since the last RSR-to-DRR transfer.

(Contd.)

Table 6.5 *(Contd.)*

$\overline{\text{XSREMPTY}}$	Transmit Shift Register Empty. This bit Indicates whether the transmitter has experienced underflow. Underflow occurs when XSR is empty and DXR has not been loaded since the last DXR-to-XSR transfer.
XRDY	Transmit Ready. A transition from 0 to 1 of the XRDY bit indicates that the DXR contents have been copied to XSR and that DXR is ready to be loaded with a new data word. A transmit interrupt (XINT) is generated upon the transition. This bit can be polled in software instead of using serial port interrupts.
RRDY	Receive Ready. A transition from 0 to 1 of the RRDY bit indicates that the RSR contents have been copied to the DRR and that the data can be read. A receive interrupt (RINT) is generated upon the transition. This bit can be polled in software instead of using serial port interrupts.
IN1	Input 1. This bit allows the CLKX pin to be used as a bit input.
IN0	Input 0. This bit allows the CLKR pin to be used as a bit input.
$\overline{\text{RRST}}$	Receive Reset. This signal resets and enables the receiver.
$\overline{\text{XSRT}}$	Transmitter Reset. This signal is used to reset and enable the transmitter.
TXM	Transmit Mode. This bit configures the FSX pin as an input (TXM = 0) or as an output (TXM=1).
MCM	Clock Mode. This bit specifies the clock source for CLKX. MCM = 0 CLKX is taken from the CLKX pin. MCM = 1 CLKX is driven by an on-chip clock source.
FSM	Frame Sync Mode. This bit specifies whether frame synchronisation pulses (FSX and FSR) are required after the initial frame sync pulse for serial port operation. FSM = 0 Continuous mode. FSM = 1 Burst mode.
FO	Format. This bit specifies the word length of the serial port transmitter and receiver. FO = 0 The data is transmitted and/or received as 16-bit words. FO = 1 The data is transferred as 8-bit bytes. The data is transferred with the MSB first.
DLB	Digital Loopback Mode. This bit can be used to put the serial port in digital loopback mode.
Res	Reserved.

6.4.2.1 Serial Port Initialisation

To have effective communication between DSP and AIC, it is necessary to initialise the serial port of the DSP.

The serial port initialisation involves activating the interrupts and programming the serial port control register (SPC). For the DSP to transmit and receive data from the AIC, the serial port receive interrupt (RINT), transmit interrupt (XINT) and external interrupt $\overline{\text{INT2}}$ are to be set. Whenever a particular interrupt is to be enabled, all the maskable pending interrupts are globally disabled. This is done by setting the INTM bit of status register 0 (ST0) to be 1. For the required interrupt to be enabled, the control word is transferred to the interrupt mask register (IMR), then the interrupts are enabled by clearing the INTM bit. The control word 32h written into IMR will enable both transmit and receive interrupt from the serial port.

The SPC register is programmed for the desired operation of the serial port. The control word of 0008h written into SPC disables the loop back mode of serial port, selects 16-bit word length for data transfer and enables burst mode of data transfer between DSP and AIC. Further, the CLKX is taken from CLKX pin, external frame synchronisation is used and the serial port transmitter and receiver are enabled.

The data transfer between the serial port and the CPU can be achieved without any wait states. So the program/data wait register (PDWSR) and wait state control registers (CWSR) are cleared. A sample initialisation program for serial port is given in Program6_12.asm.

Program 6_12.asm Program for initialisation of serial port

```
spc_init      lacc #0008h          ;SPC programmed to transmit in burst
                                   ;mode, 16-bit data transfer mode
              sacl spc
              lacc #00c8h          ;the Rx and Tx reset signals generated
              sacl spc
              setc intm            ;intm bit in interrupt mask register set
                                   ;& all unmaskable interrupts disabled
              splk #32h, imr       ;32h to imr to enable both Tx & Rx intr
              splk #0h,cwsr        ;clear wait-state control register
              splk #0h,pdwsr       ;clear program/data wait state control
                                   ;register zero wait state programmed
              clrc intm            ;all unmasked interrupts are enabled
              Ret
```

6.4.3 An Overview of the Analog Interfacing Circuit (AIC)

TLC320C40 and TLC320C41 are single monolithic CMOS chips consisting of 14-bit resolution A/D and D/A converters and four microprocessor compatible serial port modes. These AICs also consist of a BP switched-capacitor antialiasing input filter and a LP switched-capacitor output-reconstruction filter.

These devices offer numerous combinations of master clock input frequencies and conversion/ sampling rates, which can be changed via digital processor control. The devices can transmit and receive synchronously as well as asynchronously. When the communication is happening synchronously it can interface two SN74299 serial-to-parallel shift registers. These serial-to-parallel shift registers can then interface in parallel to the DSPs or external FIFO circuitry.

A flexible control scheme is provided so that the functions of this integrated circuit can be selected and adjusted coincidentally with signal processing via software control. A selectable, auxiliary, differential analog input is provided for applications where more than one analog input is required. The functional block diagram of the IC is given in the Fig. 6.7.

6.4.3.1 Terminal Functions

The abbreviations of the terminals of the AIC are given below.

ANLG GND	Analog ground for all internal analog circuits
AUX IN+	Non-inverting auxiliary analog input
AUX IN–	Inverting auxiliary analog input
DGTL GND	Internal digital ground for logic circuits
DR	Data receive, used to transmit ADC output bits to DSP serial port

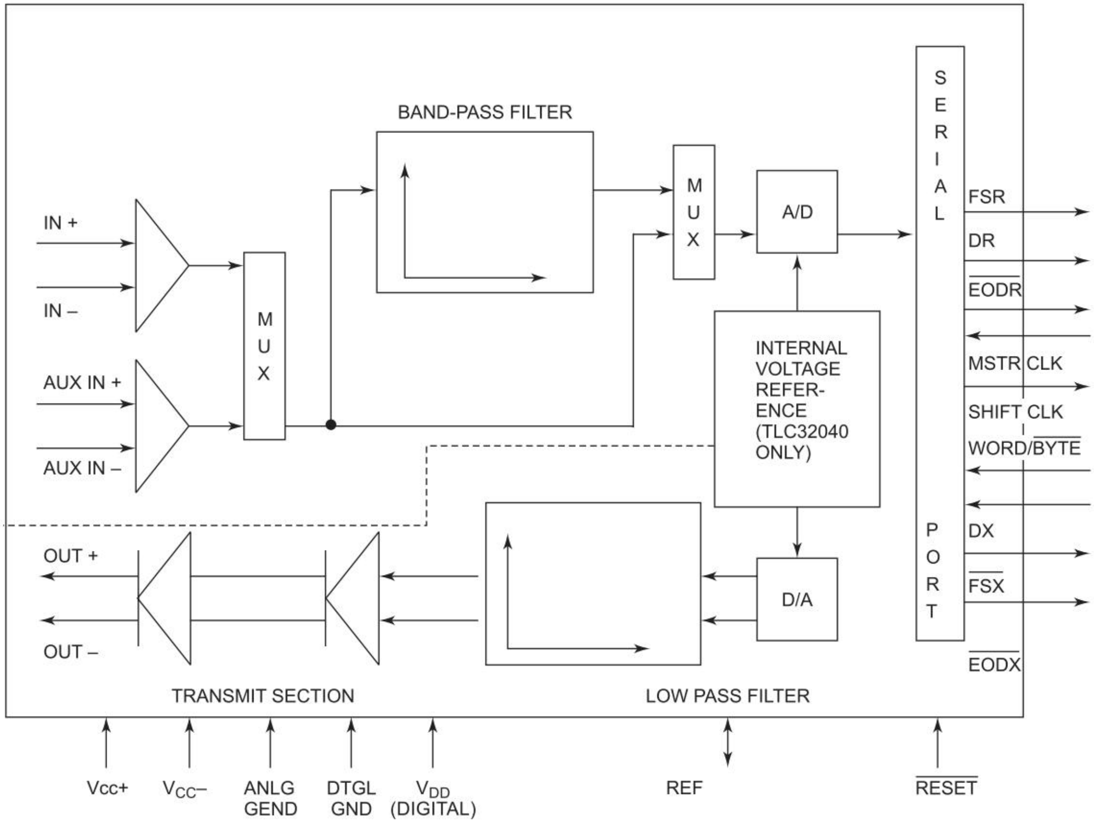


Fig. 6.7 *Functional block diagram of AIC*

DX Data transmit, used to receive the input bits, timing and control information from the DSP

$\overline{\text{EODR}}$ End of data receive

$\overline{\text{EODX}}$ End of data transmit

$\overline{\text{FSR}}$ Frame synchronisation receive

$\overline{\text{FSX}}$ Frame synchronisation transmit

IN+ Non-inverting input to analog input amplifier stage

IN– Inverting input to analog input amplifier stage

MSTRCLK Master clock. This is used to derive all the key logic signals of AIC. This signal is fed from DSP

OUT+ Non-inverting output to analog output power amplifier stage

OUT– Inverting output to analog output power amplifier stage

REF Internal reference voltage for the TLC320C40

$\overline{\text{RESET}}$ Active low reset, the reset function is provided to initialise the TA,TA',RA,RA',TB,RB and control registers in the AIC. This reset function initiates serial communication between DSP and AIC with an 8-kHz conversion rate for 5.184 MHz master clock when the control register bits of AIC are set with default value.

SHIFT CLK Shift clock. Shift clock signal is obtained by dividing master clock frequency by four.
*V*dd Digital supply voltage, 5 V ± 5%
*V*cc+ Positive analog supply voltage, 5 V ± 5%
*V*cc– Negative analog supply voltage, –5 V ±5%
WORD/$\overline{\text{BYTE}}$ A 16-bit (word) or 8-bit (byte) is transmitted or received in conjunction with the control register

The following sections give the description of AIC functional blocks.

6.4.3.2 Analog Input and Output

Two sets of analog inputs are provided. Normally, the IN+ and IN- input set is used; however, the auxiliary input set, AUX IN+ and AUX IN-, can be used if a second input is required. The auxiliary input can be enabled or disabled by bit d4 of the AIC control register. Each input set can be operated in either differential or single-ended modes, since sufficient common-mode range and rejection are provided. The gain for the IN+, IN-, AUX IN+ and AUX IN- inputs can be programmed to be either 1, 2 or 4 using the control bits d7 and d6 in the AIC control register. An analog output power amplifier is provided at the output circuitry. Both non-inverting and inverting outputs are brought out of the chip. The amplifier can drive loads directly in either a differential or a single-ended configuration.

6.4.3.3 A/D and D/A Filters

The input filter comprises seventh-order LP and fourth-order Cc-type (chebyshev/elliptical transitional) high filters whereas the output is seventh-order LP filter. The input filter is preceded and the output filter is followed by a continuous time filter to eliminate any possibility of aliasing caused by sampled data filtering at the input end and to eliminate images of the digitally encoded signal at the output end. Both input and output filter are followed by fourth-order equalisers. When no filtering is desired, the entire composite filter can be switched out of the signal path. The A/D BP filter can be selected or bypassed using the bit d0 present in the control register.

The A/D filter is a switched-capacitor and the switched-capacitor filter clock frequency of 288 kHz is generated in several options by programming the TX counter A. The A/D conversion rate is then attained by frequency dividing the 288 kHz switched-capacitor clock with TX counter B. The unwanted aliasing is prevented because the A/D conversion rate is an integral submultiple of the band pass switched-capacitor filter sampling rate, and the two rates are synchronously locked. It is to be noted that when the BP filter clock frequency is not 288 kHz the filter transfer function is frequency scaled by the ratio of the actual clock frequency to 288 kHz.

The D/A LP filter is also a switched-capacitor filter which operates at the clock frequency of 288 kHz. The 288 kHz clock signal is obtained by programming the RX counter A. The D/A conversion rate is obtained by frequency dividing the 288 kHz filter clock with RX counter B. Similar to A/D converter the frequency scaling happens when the filter clock frequency is not 288 kHz. A continuous time filter is provided at the output of the D/A converter to attenuate the switched-capacitor clock feedthrough.

6.4.3.4 Internal Timing Configuration

All the internal timing signals of AIC are derived from the master clock input. The shift clock signal is obtained by dividing the master clock input four times and this is used for the data communication between AIC and DSP. The internal timing configuration of the AIC is given in Fig. 6.9.

Shift clock frequency = master clock frequency/4
Switched-capacitor filter (SCF) = master clock frequency/(2 x contents of counter A)
Clock frequency
conversion frequency = SCF clock frequency/contents of counter B

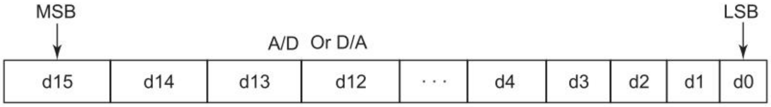


Fig. 6.8 *AIC DR or DX word bit pattern*

Table 6.6 *The LSB two bits of DX data format and their significance for primary communication*

d1 d0	*Function*
0 0	TX and RX counter A's are loaded with TA, RA B's are loaded with TB and RB register values
0 1	The TX and RX counter A's are loaded with the TA+TA' and RA+RA' register values. The TX and RX counter B's are loaded with the TB and RB register values
1 0	The TX and RX counter A's are loaded with the TA-TA' and RA-RA' register values. The TX and RX counter B's are loaded with the TB and RB register values
1 1	TX and RX counter A's are loaded with TA and RA register values. The TX and RX counter B's are loaded with the TB and RB register values. A secondary transmission starts immediately after four shift clock cycles to program the AIC to operate in the desired configuration

The switched-capacitor filter frequency 288 kHz is derived both for the A/D conversion BP filter and D/A LP filter from the master clock frequency by frequency dividing with TX counter A and RX counter A respectively. The TX and RX counter A's can be loaded with the values of TA and RA register (5 bits) contents during every conversion period. Both counters can also be loaded with the values of TA and RA register contents subtracted or added with the contents of TA' and RA' registers (6 bits). This can be selected with the d1 and d0 bits of DX word bit pattern as shown in Table 6.6. The AIC DR or DX word bit pattern is given in Fig. 6.8.

The D/A and A/D conversion frequencies are derived from the 288 kHz switched-capacitor clock frequency by frequency dividing with TX counter B and RX counter B respectively. The TX counter B and RX counter B are loaded with the values of TB and RB registers (6 bits) during every conversion period.

6.4.3.5 AIC Serial Port Modes and its Registers

There are four modes of transmission possible using the serial port in AIC. The WORD/$\overline{\text{BYTE}}$ pin present in AIC, in conjunction with bit d5 in the AIC control register, is used to establish one of the four serial modes. The active high/low signal given to the pin WORD/$\overline{\text{BYTE}}$ selects transmission of 16-bits (word)/8-bits (byte). The 0/1 stored in the bit d5 of the AIC control register selects asynchronous/synchronous mode of transmission between DSP and AIC. The selection of these possible modes is listed in Table 6.7.

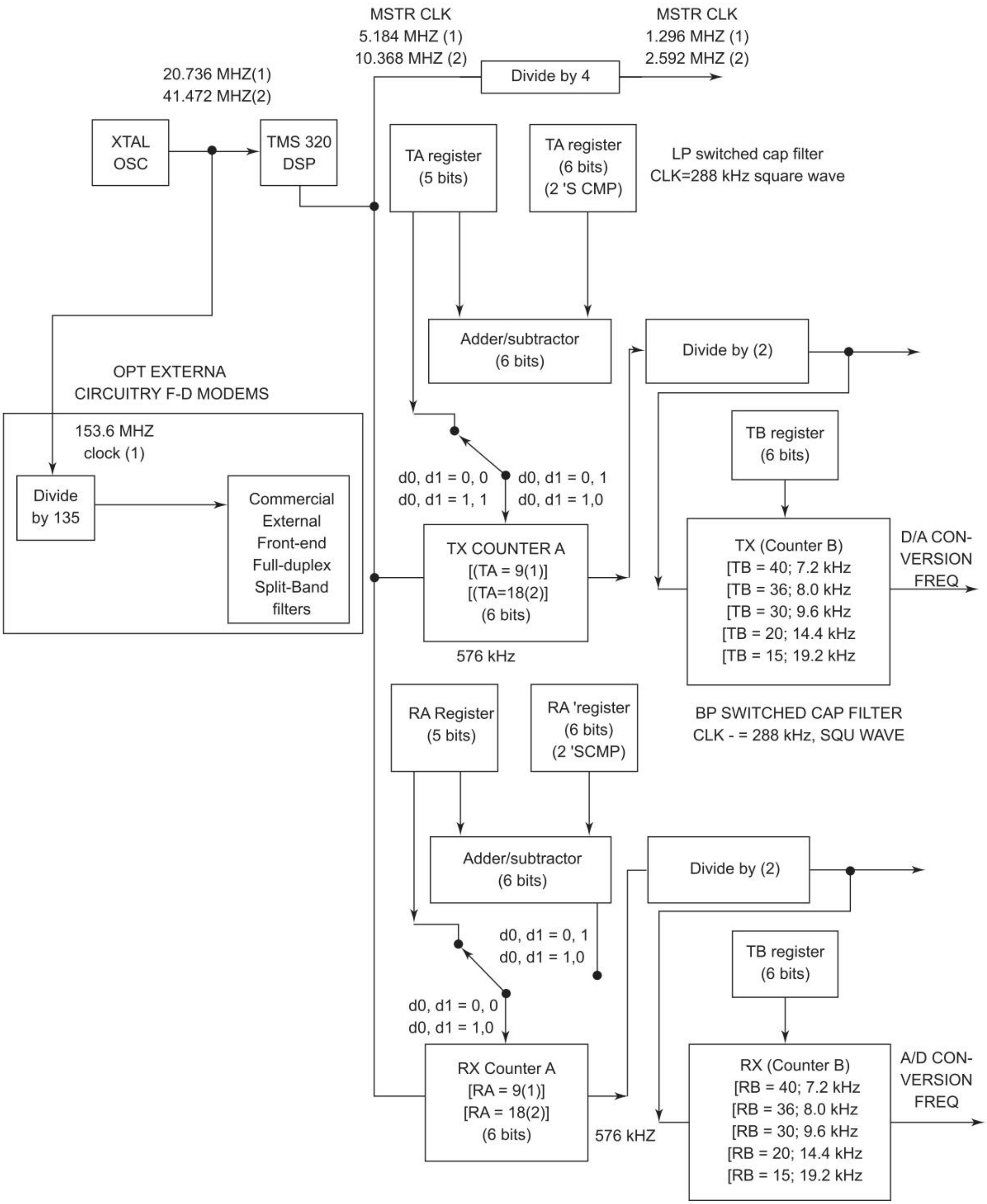


Fig. 6.9 *Internal timing configuration of AIC*

Table 6.7 *The AIC serial port modes*

AIC mode	*Word/ $\overline{\text{Byte}}$ pin level*	*Bit d5 of AIC*
Asynchronous transmission of 8 bits (byte)	0 (low level)	0
Synchronous transmission of 8 bits (byte)	0 (low level)	1
Asynchronous transmission of 16 bits (word)	1 (high level)	0
Synchronous transmission of 1 6 bits (word)	1 (high level)	1

For all the four modes, two communication protocols exist, namely, the primary and secondary communication protocols. In the primary communication protocol, the data transfer between AIC and DSP is performed, whereas in secondary communication the TA, RA, TB, RB and AIC control registers are programmed to operate in the desired configuration.

The d1 and d0 both bits set to 1 in the DX data word format in primary communication initiates the secondary communication after a four shift clock cycles. During the secondary communication the counter registers A's, B's and AIC control register are programmed. The d1 and d0 bits of the DX data format are used to load various registers of AIC as shown in Table 6.8.

Table 6.8 *The LSB two bits of DX data format and its significance for secondary communication*

d1 d0	*Function*
0 0	The TA and RA register are loaded with the 5-bit value given in the AIC initialisation routine
0 1	The TA' and RA' register are loaded with the 6-bit value given in the AIC initialisation routine
1 0	The TB and RB register are loaded with the 5-bit value given in the AIC initialisation routine
1 1	The AIC control register is loaded with the 5-bits value given in the initialisation routine

The data format of DX word during the secondary communication to load various counter registers and AIC control register is given in Figs 6.10-6.13. Table 6.9 gives the TA and TB bits for different master clocks. Table 6.10 shows how the functions of AIC are programmed by selecting the AIC control register bits.

Table 6.9 *The TA and RA Bits for Different Master Clocks*

Master clock	*Content of TA and RA register*
5.184 MHz	9 (01001)
10.368 MHz	18 (10010)

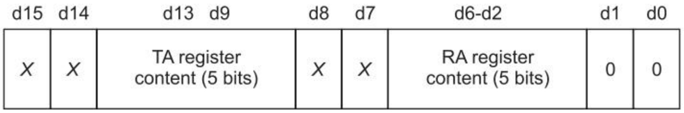


Fig. 6.10 *Data format for programming TA and RA registers*

d15	d14 d9	d8	d7-d2	d1	d0
X	TA′ register content (6 bits)	X	RA′ register content (6 bits)	0	1

Fig. 6.11 *Data format for programming TA' and RA.' registers*

d15	d14 d9	d8	d7 d2	d1	d0
X	TB register content (6 bits)	X	RB register content (6 bits)	1	0

Fig. 6.12 *Data format for programming TB and RB registers*

d15 d8	d7 d6 d5 d4 d3 d2	d1	d0
X X X X X X X X	A/C control register bits(6 bits)	0	1

Fig. 6.13 *Data format for programming the AIC control registers*

Table 6.10 *AIC control register bits and the function programmed*

d2	0/1 deletes/inserts the highpass filter
d3	0/1 disables/enables the loopback function
d4	0/1 disables/enables the AUX IN+ and AUX IN-
d5	0/1 asynchronous/ synchronous transmit and receive sections
d6	0/1 gain control bits
d7	0/1 gain control bits

6.4.3.6 AIC Serial Port Operation and Reset Function

After power has been applied to the AIC, a negative going pulse from DSP is sent to AIC RESET pin to initialise the AIC registers. The default values are loaded into the counter A's and counter B's and the control register. For the master clock frequency of 5.184 MHz the A/D and D/A conversion rate of 8 kHz and for 10.368 MHz a conversion rate of 14.4 kHz is selected. The AIC control register is loaded with the value 111001 and the synchronous primary communication between DSP and AIC starts. If the AIC is to be programmed to the desired operation then all the registers present in the AIC are to be programmed before starting the primary communication. The programming of the registers is given as AIC initialisation subroutine. It is necessary to call the subroutine before the primary communication is initiated.

6.4.3.7 AIC Secondary Communication Protocol Routine

This routine is called by the main program for AIC initialisation. The program required for the secondary communication is given in Program6_13.asm. The TA, RA, TB, RB and AIC control register contents that are to be loaded during the secondary communication of the AIC are defined using the word assembler directive in the main program. In Program 6_13. asm, first the DSP on-chip timer is programmed to generate the master clock signal for the AIC. For that the timer control register (TCR) and the timer period registers (PRD) are loaded with the appropriate control words. The value 20h transferred to TCR, makes timer to operate in the free running mode. The constant 01h loaded into the

PRD, frequency divides the CPU clock by two so as to get the master clock signal of 10.368 MHz in the TMS320C50 starter kit. The required program is given in timerinit.asm. Then the (SPC) register is programmed for the burst mode of transmission and the receive and transmit reset signals are generated. The global memory allocation register (GREG) is programmed to allocate 0000h to 7fffh locations as global memory. The required program for these functions is given in spc_init.asm. The wait state of 0.5 ms is introduced before programming the AIC. Program for inserting the required delay is given in delay.asm. Next the control words for TA, RA, TB, RB and AIC control registers are generated as per the data format described for these registers and the subroutine AIC2 given at the end of Program6_13.asm actually transmits the control words to the respective registers.

Program 6_13.asm Program for secondary communication

```
        Splk #020h,tcr                        ;DSP on-chip timer control register
                                              ;programmed to run in free running mode
        Splk #01h,prd                         ;timer period register programmed to
                                              divide CPU clock by 2
```

timeinit.asm: Initialisation of on-chip timer

```
        spc_init:  Lacc #0008h                ;SPC programmed to transmit in burst
                   Sacl spc                   ;mode, 16-bit data transfer mode
                   Lacc #00c8h                ;the Rx and Tx reset signals generated
                   Sacl spc
                   Sacl #0080h
                   Sach dxr
                   Sacl greg                  ;global memory allocation register
                                              ;Programmed
```

Spc__init.asm: Initialisation of serial port and global memory

```
        delay:     Lar ar0,#0ffffh            ;AR for rpt instruction initialized
                   Rpt #10000                 ;next instruction executed 10000 times
                   Lacc *,0,ar0               ;to introduce delay of 0.5 ms
```

delay.asm: Delay routine

```
                   .mmregs
                   .ds 1000h
                   .ps 0a00h
ta                 .word 12h                  ;control word for TA & RA to generate
                                              ;switched capacitor filter (SCF) clock
ra                 .word 12h                  ;frequency of 288 kHz from 10.368 MHz
                                              ;master clock signal
tb                 .word 40                   ;control word for TB & RB to generate A/D &
                                              ;D/A conversion rate of 7.2 kHz
rb                 .word 40                   ; from SCF clock.
aic_ctr            .word 30h                  ;30-BP filter is not included in the
                                              ;input path, 31-BP filter is included
                   .entry
```

```
aicint:         .include “timeinit.asm”      ;on-chip timer initialised
                .include “spc_init.asm”      ;serial port initialised
                .include “delay.asm”         ;delay of 0.5 ms introduced
                ldp # ta                     ;control word for TA register is obtained
                ssxm
                Lacc ta,9
                Add ra,2
                call aic2                    ;control word loaded into aic ta register
                ldp #tb                      ;control word for TB register is obtained
                Lacc tb,9
                Add rb,2
                Add #02h
                Call aic2                    ;control word loaded into aic tb register
                ldp #aic _ctr                ;control word for AIC control register
                                             ;read
                Lacc aic_ctr,2
                Add #03h
                Call aic2                    ;AIC control register written
                ret
;Assignment of values for TA, RA, TB, RB, AIC control register completed
aic2:           Ldp #0h
                Idle
                Sach dxr
                Add #06h,15                  ;initialisation of secondary communica-
                                             ;tion
                Idle
                Sach dxr                     ;AIC reset
                Idle
                Sacl dxr                     ;control word transmission happens here
                Zap
                Idle
                Sacl dxr                     ;control words transferred from DSP to AIC
                Ret
                . end
```

The initialisation of timer, serial port and the delay routine are given in separate files in order to improve the readability of the program. They are inserted in Program6_13.asm by the include statements.

6.4.3.8 Interfacing the DSP and AIC

The interfacing diagram of the TMS320C50 and TLC320C40 is given in Fig. 6.14. The timer clock out signal is fed as master clock to AIC and the shift clock signal generated by the AIC is applied to clock receive and transmit pins of DSP. The receive and transmit frame synchronisation signals from DSP and AIC are interconnected. Similarly the DX and DR pins are interconnected.

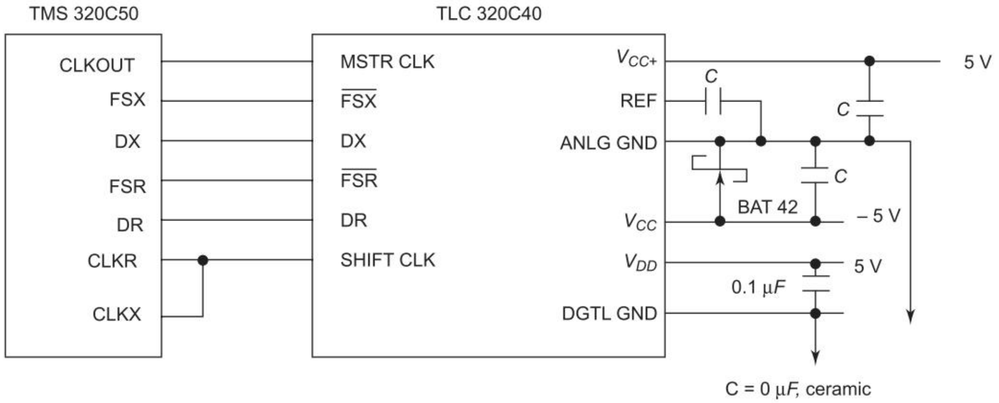


Fig. 6.14 *AIC interface to TMS320C50*

Waveform Generation using AIC Programs for generating various **waveforms** such as square, ramp, triangular and trapezoidal are given in this section.

6.4.3.9 Square Wave

For square wave generation the positive and negative amplitudes and T_{on} and T_{off} time periods are to be fixed. For this the following symbol declaration is used. The values are set for these symbols using .set assembler directive. In the source program it is enough to mention only these symbols wherever required.

Amp+ve: positive amplitude	T_{on}: on period of the wave
Amp-ve: negative amplitude	T_{off}: off period of the wave

Table 6.11 *Parameters for generating different square waves*

Amp -ve	*Amp +ve*	*Ton*	*Toff*	*Type of square wave generated*
9fffh	7fffh	0f00h	0f00h	Sqr wave with 50% duty cycle & 0 dc component
9fffh	7fffh	7f00h	5f00h	Sqr wave with duty cycle >50% & 0 dc component
9fffh	7fffh	3f00h	7f00h	Sqr wave with duty cycle <50% & 0 dc component
0000h	7ffeh	0f00h	0f00h	+ve clamped sqr wave with 50% duty cycle
9ffeh	0000h	0f00h	0f00h	-ve clamped sqr wave with 50% duty cycle

The maximum positive and negative values that can be assigned are 7FFFh and 9FFFh respectively. Similarly the maximum and minimum Ton and Toff periods are 7FFFh and 0F00h. Table 6.11 lists the various types of square waveforms that can be generated by choosing various values for Amp+ve, Amp-ve, T_{on} and T_{off}. The amplitude and time period can be varied to any value by properly selecting the constants within the range given. Two programs are given for square wave generation.The first-program Program6_14.asm uses RPT instruction to generate the square wave, whereas in the second program Program6_15.asm two loops are present; one loop is executed for positive amplitude and another one is executed for the negative amplitude. The time periods are loaded into the ARs and decremented each time the loop is executed.

Spc initl.asm Serial port initialisation for Transmit mode

```
spc initl          Setc intm                    ;intm bit in the interrupt mask register
                                                ;set, unmasked interrupts disabled
                   Splk #22h,imr                ;control word 22h stored in IMR
                   Splk #0h,cwsr                ;clear wait-state control register
                   Splk #0h,pdwsr               ;clear program/data wait-state control
                                                ;register
                   Clrc intm                    ;all unmasked interrupts enabled
```

The serial port is initialised in spc_initl.asm, with the control word 22h being transferred to IMR for serial port transmit (Tx) mode [refer Section 6.4.2.1] and for the A/D and D/A conversion rates of AIC the default values are used [refer Section 6.4.3.5].

Program 6_14.asm Square wave generation using repeat instruction

```
Label              Mnemonic                     Comments
                   .mmregs
                   .ds 1000h
                   .ps 0a00h
                   .entry
amp+ve             .set 7fffh                   ;7FFFh is chosen for Amp+ve
amp-ve             .set 9fffh                   ;9FFFh is chosen for Amp-ve
ton                .set 0f00h                   ;0F00h is set for Ton
toff               .set 0f00h                   ;0F00h is set for Toff
                   .entry
                   .include spc_init1.asm
                                                ;serial port for Tx mode
loop               lar ar0,#ton
                   lar ar1,#toff
                   Lacc #amp+ve
                   And #0fffch                  ;last 2 bits of DX data format to be 0 fo
                                                ;primary commn between DSP and AIC
                   rpt #ton
                   Samm dxr                     ;constant in ACCU low byte loaded into DXR
                   Lacc # amp-ve
                   And #0fffch
                   rpt # toff
                   Samm dxr
                   b loop
                   .end
```

6.4.3.10 Ramp Signal Generation

Program6_16.asm is used to generate ramp signal. The amplitude and the slope of the ramp are set in the assembler directive using ‘amp’ and ‘step’ symbols. The amplitude value is loaded in the AR compare register (ARCR). The ACCU is reset and step-size value is added with the content of the ACCU. After addition the content of ACCU is transmitted to DXR. The step size is also added to an AR (AR0 is used here) and compared with ARCR. Once the AR value reaches the amplitude value in the ARCR, the ACCU is reset and once again the execution starts in a loop.

Program 6_15.asm **Square wave generation with two loops**

```
Label                 Mnemonic                      Comments
                      .mmregs
                      .ds 1000h
                      .ps 0a00h
                      .entry
amp+ve                .set 7fffh                    ;7FFFh is chosen for Amp+ve
amp-ve                .set 9fffh                    ;9FFFh is chosen for Amp-ve
ton                   .set 0f00h                    ;0F00h is set for Ton
toff                  .set 0f00h                    ;0F00h is set for Toff
                      .entry
                      .include
                      spc initl.asm
                                                    ;serial port initialised for Tx mode
loop                  Lar ar0,#ton
                      Lar ar1,#toff
                      Lacc #amp+ve
                      And #0fffch
                      Mar *,ar0
loop1                 Samm dxr
                      Nop                           ;2 nops introduced to avoid pipeline
                                                    ;conflict when using indirect
                      Nop                           ;addressing followed by memory
                                                    ;mapped register write operation
                      Banz loop1,*-
                      Lacc #amp-ve
                      And #0fffch                   ;last 2 bits of DX data format to be
                                                    ;0 for primary communication
                                                    ;between DSP and AIC
                      Mar *,ar1
loop2                 Samm dxr
                      Nop
                      Nop
                      Banz loop2,*-
                      B loop
                      .end
```

Program 6_16.asm Ramp signal generation

```
Label       Mnemonic              Comments
            .mmregs
            .ds 1000h
Amp         .set 7fffh
Step        .set 010h             ;min 4h, max 10h
            .ps 0a00h
            .entry
            .include spc_init1.asm
loop        Lacc #amp
            Lar ar0,#0h
            Samm arcr             ;amplitude stored into ARCR register
            Lacc #0h
loop1       Add #step             ;step size added to ACCU
            And #0fffch           ;last 2 bits of DX data format to be 0 for
                                  ;primary communication between DSP & AIC
            Samm dxr
            Nop
            Nop
            Adrk #step            ;the step size is added to the auxiliary
                                  ;register AR0
            Bcnd loop1,tc         ;content of AR0 compared with ARCR.
                                  ;if content of AR is less.TC flag bit is
                                  ;set,
                                  ;if not TC is cleared. If TC set, branch to
                                  ;loop1, if not execute next instruction
            b loop                ;unconditional branch to loop
            .end
```

6.4.3.11 Triangular Wave Generation

Program6_17.asm is used to generate a triangular wave signal. The required amplitude and slope of the wave is set in the assembler directive section using the symbols ‘amp’ and ‘step’. The amplitude value is loaded into the ARCR. The ACCU is reset and step-size value is added with the content of the ACCU. After addition the content of ACCU is transmitted to DXR. The step size is also added to an AR (AR0 is used here) and compared with ARCR. Once the AR value reaches the amplitude value in the ARCR, the loop1 execution stops and loop2 execution starts. The ARCR is loaded with the constant zero. The step size is subtracted from the content of ACCU and then it is transferred to DXR. The content of AR is also decremented by step size. Once the AR content reaches zero the loop2 execution stops and loopl execution starts. Alternate execution of loop1 and loop2 generates the triangular wave form.

Program 6_17.asm Triangular wave generation

```
Label           Mnemonic                  Comments
                .mmregs
                .ds 1000h
amp             .set 7fffh
step            .set 010h
                .ps 0a00h
                .entry
                .include spc_init1.asm
loop            Lacc lamp
                Lar ar0,#0h
                Samm arcr                 ;Maximum amplitude stored into ARCR
                Lacc #0h
loop1           Add #step
                And #0fffch               ;last 2 bits of DX data format to be 0 for
                                          ;primary communication between DSP & AIC
                Samm dxr
                Nop
                Nop
                Adrk #step
                Bcnd loop1,tc             ;content of AR0 compared with ARCR.
                                          ;if content of AR is less.TC flag bit is set,
                                          ;if not TC is cleared. If TC set branch to
                                          ;loopl, if not execute next instruction
                Splk #0h,arcr             ;the minimum amplitude is stored in
                                          ;the ARCR register
loop2           Sub #step
                And #0fffch
                Samm dxr
                Nop
                Nop
                Sbrk #step
                Bcnd loop2,tc             ;content of AR0 compared with ARCR.
                                          ;if content of AR is less.TC flag bit is set,
                                          ;if not TC is cleared, If TC set branch to
                                          loop2, if not execute next instruction
                Nop
                Nop
                b loop
                .end
```

6.4.3.12 Capture and Display (Without AIC Initialisation)

Program 6_18.asm is used for digitising an analog signal, store it in DSP memory and convert it back to analog waveform. A sine wave signal is fed to the analog input pin of AIC, it is converted into digital word, stored in the accumulator, transmitted back to AIC, converted back to analog signal and displayed in the CRO. It is verified that the input and output signals are the same and also the Nyquist theorem is verified. In this example the AIC control register and A/D and D/A conversion rate control registers are loaded with the default values. The control word 32h transferred to interrupt mask register (IMR) will enable both the transmit and receive interrupts. If the data value loaded into the ACCU is left shifted and transmitted to the D/A converter of AIC, the amplitude of the output analog signal will increase. Similarly if the data samples are stored in the data memory and alternate samples are sent to AIC, the frequency can be modified.

Program 6_18.asm Capture and display of waveforms with default values for timer, SPC and AIC registers

```
Label           Mnemonic                Comments
                .mmregs
                .ps 0a00h
                .entry
                Setc intm
                Splk #32h,imr           ;control word 32h enables RINT, XINT and
                                        ; INT2
                Splk #0h,pdwsr
                Splk #0h,cwsr
                Clrc intm
                Nop
                Nop
loop:           Lamm drr                ;data word receive and transmit loop
                Nop                     ;(Primary communication)
                Nop
                And #0fffch
                Samm dxr
                Idle
                B loop
                .end
```

As per the Nyquist theorem if the baseband signal maximum frequency is fm, if the sampling of the baseband signal is done at a rate fs (≥ 2fm), then the original signal can be reconstructed back. In this sample program the sampling rate is selected as 14.4 kHz. It can be seen that when the input frequency is < 7.2 kHz, the input signal and the output signal are the same. Otherwise aliasing occurs.

6.4.3.13 Capture and Display (With AIC Initialisation)

In Program 6_19.asm, the DSP serial port is initialised and the A/D and D/A conversion rates are software programmed using the AIC initialisation routine [refer Section 6.4.3.7]. The conversion rate is selected as 7.2 kHz with master clock signal frequency of 10.368 MHz. After initialising the interrupts, it is enough to execute the AIC initialisation routine once, then the data conversion will be decided based on the AIC register values set in the initialisation routine. The programs timeinit.asm, spc_init.asm, delay.asm and aic_init.asm are given in Section 6.4.3.7.

Program 6_19.asm Capture and display of waveforms with values for timer, SPC and IC registers specified by the user

```
Label            Mnemonic                  Comments
                 .mmregs
                 .ds 1000h
                 .ps 0a00h
ta               .word 12h                 ;control word for TA & RA to generate
                                           ;switched-capacitor filter (SCF) clock
ra               .word 12h                 frequency of 288 kHz from 10.368 MHz
                                           ;master clock signal
tb               .word 40                  ; control word for TB & RB to generate A/D &
                                           ;D/A conversion rate of 7.2 kHz
rb               .word 40                  ;from SCF clock.
ai c_ctr         .word 30h                 ;30-BP filter is not included in the
                                           ;input path, 31-BP filter is included
                 .entry
                 Setc intm                 ;serial port interrupt and AIC initialisation
                 Splk #32h,imr             ;control word 32h enables RINT, XINT &
                                           ;INT2
                 Splk #0h,pdwsr
                 Splk #0h,cwsr
                 Clrc intm
                 Call aicint               ;call AIC initialisation routine
                 Nop
                 Nop
loop:            Lamm drr                  ;data word receive and transmit loop
                 Nop                       ;(primary communication)
                 Nop
                 And #0fffch
                 Samm dxr
                 Idle
                 B loop
aicint:          .include “aic_int.asm”    ;program for AIC initialisation inserted here
                 .end
```

6.4.3.14 FSK Generation

Frequency shift keying (FSK) technique is used in digital communication systems for transmission of digital data by changing the carrier frequency in accordance with the data to be transmitted. For example, in the binary data transmission system, the input signals to the digital communication system will be 1 s and 0s. To transmit this data through the communication channel, FSK technique may be used.

In binary FSK system two different carrier frequency signals are transmitted through the channel based on the binary information. For the binary 1, carrier signal of one frequency is transmitted and for logic 0 another signal of different frequency is transmitted. The two sine wave signals may be generated by using the look up table approach. The samples of the sine waves corresponding to a bit duration may be stored in the look up table and transmitted to the AIC. The AIC converts these samples into analog signals.

In Program 6_20.asm used for the generation of FSK signal, the two different carrier signals are assumed to be captured (digitised) and stored in two different data files using the capture and display assembly language program given in Program6_17.asm or Program6_18.asm. Based on the binary information the sine wave samples present in the look up table are transmitted to the output.

6.4.3.15 Generating Sine Wave Look Up Table Using Capture Program

The sampling rate of the AIC (TLC320C40) can be varied using the contents present in the TB and RB counter registers (refer Chapter AIC initialisation). Select a 1 kHz sine wave as input signal and the TB and RB register values as 15. This will enable a sampling rate of 19.2 kHz for A/D and D/A conversion. Execute the capture and display program, where the captured sample values will be stored in the data memory space. It can be noted that 18 sample values are stored for each input cycle of the sine wave. If the sampling rate selected is 8 kHz (TB & RB = 36) the number of sample values that get stored are 8 per cycle. Similarly for rest of the sampling rate, corresponding sample values that will be stored. It is to be noted that if the input frequency doubles, the samples stored reduce to half of the previous input.

The sampled data values can be stored into a data file using the save option available in the TMS320C5X DSK debugger. The number of sample values to be stored in a data file depends on the number of cycles needed to be transmitted. For example, 1-kHz input with 19.2 kHz sampling rate stores 18 sample values. If four cycles are to be reproduced, 72 sample values are to be stored in a data file. As FSK generation needs two different frequencies, two signals of different frequency can be captured and stored in two data files. With the different sampling rate conversion it is also possible to generate various frequencies with one frequency signal being captured. For example, the input samples can be captured at the 19.2 kHz conversion rate and if it is displayed at the 9.6 kHz rate, at half the capture rate, the frequency can be increased. Similarly any frequency in multiples of one frequency can be easily generated by changing the A/D and D/A conversion rates.

In the following example 1 kHz and 2 kHz input signals are captured and stored in two data files 'sinlk.dat' and 'sin2k.dat'. These files containing the sine wave sample values are included in the FSK generation program, The subroutines aic_init used for the initialisation of the AIC is given in Program aic_init.asm in Section 6.4.3.7 and is included at the end of Program6_20.asm.

Program 6_20.asm **FSK generation program**

```
                .mmregs
                .ds 1000h
                .include 'sin1k.dat'         ;include 1 kHz sine wave look up table in
                                             ;data
                                             ;memory space starting address 1000h
                .ds 1200h
                .include 'sin2k.dat          ;include 2 kHz sine wave look up table in
                                             ;data
                                             ;memory space starting address 1200h
                .ps 0a00h
ta              .word 12h
ra              .word 12h
tb              .word 15
rb              .word 15
```

```
aic_ctr         .word 31h               ;30-bpf not inluded, 31-bpf included
                .entry
                setc intm
                splk #32h,imr
                splk #0h,pdwsr
                splk #0h,cwsr
                clrc intm
                call aicint
loop1           lamm drr                ;content of drr register loaded into
                                        ;accumulator
                idle
loop            sfr                     ;right shift the content of accumulator
                exar                    ;exchange content of ACCU with ACCB
                Bcnd loop3,nc           ;if there is no carry branch to loop3,
                                        ;else execute next instruction
                lar ar0,#1000h          ;load the starting address of the look up
                                        ;table 1 (1000h) in AR0
                lar ar1,#53             ;the no. of sample values to be
                                        transmitted (54) is loaded in AR1
                mar *,ar0
loop2           lacc *+,0,ar1           ;loop2 transmits 1 kHz sine wave
                                        ;samples to output
                and #0fffch
                samm dxr
                idle
                Banz loop2,ar0
                lacb                    ;load the ACCB content back to ACCU
                Bcnd loop1.eq           ;branch to loopl if ACCU content is zero,
                                        ;to get new sample from drr
                b loop                  ;branch to loop to know the next bit
                                        ;information of ACCU
loop3           lar ar2,#1200h          ;load the starting address of the look
                                        ;up table 2 (1200h) in AR2
                lar ar3,#26             ;the no. of sample values to be
                                        ;transmitted (27) is loaded in AR3
                mar *,ar2
loop4           lacc *+,0,ar3           ;loop4 transmits 2 kHz sine wave sample
                                        ;to output
                and #0fffch
                samm dxr
                idle
                Banz loop4,ar2
                lacb
                Bcnd loop1.eq
                b loop
aicint:         .include "aic_int.asm"  ;program for AIC initalisation inserted
                                        ;here
                .end
```

6.4.3.16 The Shine Wave Look Up Tables For 1 kHz and 2 kHz

The sine wave samples captured using the AIC routine will have hexadecimal values. It is needed that .word assembler directive be included in the beginning of the sample values and a lower case h at the end to indicate to the assembler that the sample values are hexadecimal values. It is also important that the sample values having their MSBs logic 1, zero to be put in front of the sample values. The look up tables given in Tables 6.12 and 6.13 contain the samples of 1 kHz and 2 kHz, for three cycles of input stored at the 19.2 kHz conversion rate of AIC.

Table 6.12 *Lookup table for 1 kHz signal*

First 18 samples	*Second 18 samples*	*Third 18 samples*
.word 0ff60h	.word 0ff58h	.word 0ff54h
.word 0fe1ch	.word 0fe1 8h	.word 0fe14h
.word 0fd1ch	.word 0fd18h	.word 0fd14h
.word 0fc74h	.word 0fc74h	.word 0fc74h
.word 0fc40h	.word 0fc44h	.word 0fc44h
.word 0fc88h	.word 0fc8ch	.word 0fc90h
.word 0fd44h	.word 0fd48h	.word 0fd4ch
.word 0fe54h	.word 0fe58h	.word 0fe60h
.word 0ffa0h	.word 0ffa8h	.word 0ffa8h
.word 00fch	.word 00fch	.word 00fch
.word 0238h	.word 0240h	.word 0240h
.word 0334h	.word 0338h	.word 0338h
.word 03d0h	.word 03cch	.word 03d0h
.word 03fch	.word 03f8h	.word 03fch
.word 03b0h	.word 03b0h	.word 03ach
.word 02fch	.word 02fch	.word 02f8h
.word 01f0h	.word 01f4h	.word 01ech
.word 00b0h	.word 00b0h	.word 00ach

Table 6.13 *Lookup table for 2kHz signal*

First 9 samples	*Second 9 samples*	*Third 9 samples*
.word 0038h	.word 0060h	.word 0084h
.word 02b8h	.word 02d4h	.word 02ech
.word 0400h	.word 0404h	.word 0404h
.word 0374h	.word 0360h	.word 034ch
.word 015ch	.word 0134h	.word 0110h
.word 0feach	.word 0fe88h	.word 0fe68h
.word 0fcb4h	.word 0fca0h	.word 0fc90h
.word 0fc5ch	.word 0fc64h	.word 0fc6ch
.word 0fdd0h	.word 0fdech	.word 0fe08h

6.4.3.17 FIR Filter Implementation

This section discusses the details on Program 6_21.asm used for realising a FIR LP filter in real time. The input, output relation of the FIR filter may be expressed using the convolution expression given by

$$y[n] = \sum_{k=0}^{M-1} h[k] \times [n-k]$$

where $y[n]$ is the output and $x[n]$ is the input sequence, $h(n)$ is the impulse response sequence corresponding to the filter and M is the number of filter coefficients. The filter coefficients may be generated using various methods, such as using a C program or using MATLAB . For these programs, some of the parameters to be specified are given in Table 6.14. Values of the parameters used for the FIR LP filter in Program6_21.asm are also given in Table 6.14. It is important that to get good filter response, the coefficients are represented in Q15 data format.

Table 6.14 *Parameters for the FIR Filter*

Parameter name	*Parameter used for the filter in Program6_20.asm*
Type of filter	Low pass
Cutoff frequency	1 kHz
Sampling rate	9 kHz
No. of filter taps (filter coefficients)	81

The filter coefficients are inserted into the FIR filter assembly program using the .include assembler directive. The real time samples of the input signals are captured by initialising the AIC with 9 kHz A/D conversion rate. The captured samples are stored in the data memory and convolved with the filter coefficients using the MACD instruction. Each time the MACD instruction is repeated in a loop, the first sample stored in data memory gets right shifted by one location. The process is repeated for '*n*' number of sample values. Each time the convolution is completed, the resultant MSB 16 bits of accumulator are written to scratch pad RAM with left shift of one bit to remove the sign bit and then it is transmitted to the AIC output. The LSB 16 bits are discarded.

To start with, the data memory locations corresponding to the input samples have to be filled with 0s. Subsequently, these locations are filled with the input sample one after another every time one convolution of the input sequence with the filter coefficient is completed. The manner in which the convolution is required to be performed for the real time filter is different from the way it is performed in the non real time case explained in Section 6.3.3.2. In Section 6.3.3.3 two sequences of length M and N are convolved and an output sequence of length $M + N - 1$ is generated. However, in the present case every time a new sample arrives, this along with the past 80 samples is convolved with the filter sequence and a single output value is generated. This process is continued infinitely. The number of times the MACD instruction is repeated is 81, as the filter sequence length is 81. The filter coefficients corresponding to the LP filter used for this program is given in Appendix 6.3. It may be noted that this program can be used for the implementation of HP filter as well as BP filters. The values of Ta and Tb should be chosen to correspond to the sampling rate used. The repeat count should be chosen so as to be equal to the no. of taps –1 used for the filter.

Program 6_21.asm

```
                    .mmregs
                    .ps 1200h
                    .include 'coeff.dat'        ; include the filter coefficients
                    .ds 1000h
                    .include 'pad.dat'          ; padding of 0s for the second sequence
                    .ps 2000h
ta                  .word 12h                   ;sampling rate chosen as 9 kHz
ra                  .word 12h
tb                  .word 15
rb                  .word 15
aic_ctr             .word 30h                   ; 30-bpf no, 31-bpf yes
                    .entry
                    setc intm
                    splk #32h,imr
                    splk #0h,pdwsr
                    splk #0h,cwsr
                    clrc intm
                    ssxm                        ;set sign extension mode bit
                    call aicint
                    nop
                    nop
                    lar ar0,#1000h              ;starting address of the data sequence
                                                ;being
                                                ;stored is loaded into AR0
loop1               lamm drr
                    nop
                    nop
                    sacl *,0,ar2
                    nop
                    nop
                    lar ar2,#1050h              ;end address of data sequence loaded into
                                                ;AR2
                    zap
                    rpt #050h
                    macd 1200h,*-               ;convolution & data move operation is
                    apac                         ;performed
                    sach output,1                ;ACCU high bits left shifted by 1 bit to
                                                ;remove sign extension bit, & then stored
                                                ;into
                                                ;scratch pad RAM (location 127 of data
                                                ;page zero)
                    lacc output                 ;accu loaded with scratch pad ram content
                    sfl
                    and #0fffch
```

```
samm dxr                        ;convolved data transmitted to serial
                                ;port
idle
mar *,ar0
b loopl
.include "aic_init.asm"         ;AIC initialisation routine in-
                                ;serted here.
.end
```

pad.dat

```
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
.WORD  0,0,0,0,0,0,0,0,0,0
```

6.4.3.18 Study of Periodic Frequency Response of the Digital Filters

In Chapter 1, it was mentioned that the digital filters have periodic frequency response with period equal to the sampling frequency. In the LP filter considered in Section 6.4.3.17, assume that the continuous time switched-capacitor BP filter is not included (aic_ctr = 30h) at the input to the AIC. In this case if the input to the AIC is greater than fs/2 (i.e. 4.5 kHz) aliasing would occur. Since the LP filter designed has a cutoff frequency of 1 kHz, only those aliased components which lie within ±1 kHz from nf_s would be passed by the LP filter. Hence if the frequency of input signal to the filter is increased, it will have a periodic response with period equal to fs (9 kHz in the above example). Hence if an input signal of frequency 108 kHz is fed to the digital LP filter discussed above, the output is not zero, it is as strong as the input itself. This is because of aliasing. The periodicity property of the digital filter may also be used to determine the sampling rate of the AIC for different values of Ta and Tb. With the filter coefficients chosen in Section 6.4.3.18, vary the values of Ta and Tb. The LP filter cutoff frequency will not be 1000 Hz as the sampling rate is changed. The output will be zero for frequency f/X. As the input frequency is increased further, the output would start rising again at fcl. As the frequency is increased further, the output amplitude rises and reaches the full amplitude. With further increase in frequency, the output would become zero again at $fx2$. The sampling frequency fs is then given by ($fx1$+$fx2$)/2. This is because

$$f_{x1} = f_s - f_x \text{ and } f_{x2} = f_s + f_x$$

If the switched-capacitor BP filter is included at the input to the AIC, aliasing would not occur. A LP filter in this case really behaves like a LP filter. In this case for all frequencies greater than fc., the cutoff frequency of the LP filter, the output would be zero.

APPENDIX 6

A6.1 ASSEMBLER DIRECTIVES

Mnemonic and syntax	*Description*
(a) Directives that define sections	
.data	Assemble into data memory
.ds [address]	Assemble into data memory (initialise data address)
.entry [address]	Initialise the starting address of the program counter when loading a file
.ps [address] .text	Assemble into program memory (initialise program address)
.text	Assemble into program memory
(b) Directives that reference other files	
.copy [“] filename[“]	Include source statements from another file
.include [“] filename[“]	Include source statements from another file
(c) Conditional assembly directives	
.else	Optional conditional assembly
.endif	End conditional assembly
.if well-defined expression	Begin conditional assembly
(d) Directives that initialise constants (data and memory)	
bfloat value$_1$ [,..., value$_n$]	Initialise a 16-bit, 2s-complement exponent and a 32-bit, 2s-complement mantissa—an unpacked floating-point number
.byte value$_1$ [,..., value$_n$]	Initialise one or more successive words in the current section
.double value$_1$ [,...,value$_n$]	Initialise a 64-bit, IEEE double-precision, floating-point constant
.efloat value$_1$ [,..., value$_n$]	Initialise a 16-bit, 2s-complement exponent and a 16-bit, 2s-complement mantissa—a less accurate unpacked floating-point number
.float value$_1$ [,..., value$_n$]	Initialise a 32-bit, IEEE single-precision, floating-point constant
.int value$_1$ [,..., value$_n$]	Initialise one or more 16-bit integers
.long value$_1$ [,..., value$_n$]	Initialise one or more 32-bit integers
.lqxx value$_1$ [,..., value$_n$]	Initialise a 32-bit, signed 2s-complement integer whose decimal point is displaced *xx* places from the LSB
.qxx value$_1$ [,..., value$_n$]	Initialise a 16-bit, signed 2s-complement integer whose decimal point is displaced *xx* places from the LSB
.space size in bits	Reserve size bits in the current section; note that a label points to the beginning of the reserved space
.string “stringl” [,..., string *n*”]	Initialise one or more text strings
.tfloat value$_1$ [,..., value$_n$]	Initialise a 32-bit, 2s-complement exponent and a 64-bit, 2s-complement mantissa; note that the initialised integers are in unpacked form
.word value$_1$ [,..., value$_n$]	Initialise one or more 16-bit integers

(e) Miscellaneous directives

.end	Program end
.listoff	End source listing (overrides the -1 assembler option)
.liston	Restart the source listing (overrides the -1 assembler option)
.set	Equate a value with a local symbol
.mmregs	Enter memory-map registers into symbol table

A6.2 MEMORY-MAPPED REGISTERS AND THEIR ADDRESSES

Address name	*DEC*	*HEX*	*Description*
	0-3	0-3	Reserved
IMR	4	4	Interrupt mask register
GREG	5	5	Global memory allocation register
IFR	6	6	Interrupt flag register
PMST	7	7	Processor mode status register
RPTC	8	8	Repeat counter register
BRCR	9	9	Block repeat counter register
PASR	10	A	Block repeat program address start register
PAER	11	B	Block repeat program address end register
TREG0	12	C	Temporary register used for multiplicand
TREG1	13	D	Temporary register used for dynamic shift count
TREG2	14	E	Temporary register used as bit pointer in dynamic bit test
DBMR	15	F	Dynamic bit manipulation register
AR0	16	10	Auxiliary register 0
AR1	17	11	Auxiliary register 1
AR2	18	12	Auxiliary register 2
AR3	19	13	Auxiliary register 3
AR4	20	14	Auxiliary register 4
AR5	21	15	Auxiliary register 5
AR6	22	16	Auxiliary register 6
AR7	23	17	Auxiliary register 7
INDX	24	18	Index register
ARCR	25	19	Auxiliary register compare register
CBSR1	26	1A	Circular buffer 1 start register
CBER1	27	IB	Circular buffer 1 end register
CBSR2	28	1C	Circular buffer 2 start register
CBER2	29	ID	Circular buffer 2 end register
CBCR	30	IE	Circular buffer control register
BMAR	31	IF	Block move address register
DRR	32	20	Data receive register
DXR	33	21	Data transmit register
SPC	34	22	Serial port control register
	35	23	Reserved

TIM	36	24	Timer register
PRD	37	25	Period register
TCR	38	26	Timer control register
	39	27	Reserved
PDWSR	40	28	Program S/W wait-state register
IOWSR	41	29	I/O S/W wait-state register
CWSR	42	2A	S/W wait-state control register
	43-47	2B-2F	Reserved
TRCV	48	30	TDM data receive register
TDXR	49	31	TDM data transmit register
TSPC	50	32	TDM serial port control register
TCSR	51	33	TDM channel select register
TRTA	52	34	Receive/transmit address register
TRAD	53	35	Received address register
	54-79	36-4F	Reserved

A6.3

The values of the filter coefficients for the LP filter with a cutoff frequency of 1 kHz and a sampling rate of 9 kHz is computed using MATLAB and the coefficients are as follows.

Values of coefficients after scaling		*Actual filter coefficients*	
.word	0	40	0.0000
.word	-157	39	-0.0048
.word	-261	38	-0.0080
.word	-268	37	-0.0082
.word	-170	36	-0.0052
.word	0	35	-0.0000
.word	180	34	0.0055
.word	301	33	0.0092
.word	310	32	0.0095
.word	198	31	0.0060
.word	0	30	0.0000
.word	-211	29	-0.0065
.word	-354	28	-0.0108
.word	-367	27	-0.0112
.word	-236	26	-0.0072
.word	0	25	-0.0000
.word	255	24	0.0078
.word	431	23	0.0132
.word	451	22	0.0138
.word	292	21	0.0089
.word	0	20	0.0000
.word	-323	19	-0.0098
.word	-551	18	-0.0168
.word	-584	17	-0.0178

```
.word     -383                16    -0.0117
.word        0                15    -0.0000
.word      438                14     0.0134
.word      763                13     0.0233
.word      827                12     0.0252
.word      557                11     0.0170
.word        0                10     0.0000
.word     -681                 9    -0.0208
.word    -1240                 8    -0.0378
.word    -1417                 7    -0.0432
.word    -1022                 6    -0.0312
.word        0                 5    -0.0000
.word     1533                 4     0.0468
.word     3307                 3     0.1009
.word     4960                 2     0.1514
.word     6131                 1     0.1871
.word     6554                 0     0.2000
.word     6131                 1     0.1871
.word     4960                 2     0.1514
.word     3307                 3     0.1009
.word     1533                 4     0.0468
.word        0                 5    -0.0000
.word    -1022                 6    -0.0312
.word    -1417                 7    -0.0432
word     -1240                 8    -0.0378
.word     -681                 9    -0.0208
word         0                10     0.0000
word       557                11     0.0170
word       827                12     0.0252
word       763                13     0.0233
word       438                14     0.0134
word         0                15    -0.0000
word      -383                16    -0.0117
word      -584                17    -0.0178
word      -551                18    -0.0168
word      -323                19    -0.0098
word         0                20     0.0000
word       292                21     0.0089
word       451                22     0.0138
word       431                23     0.0132
word       255                24     0.0078
word         0                25    -0.0000
word      -236                26    -0.0072
word      -367                27    -0.0112
word    -354
word    -211
word       0
word     198
word     310
word     301
word     180
word       0
word    -170
word    -268
word    -261
word    -157
word       0
```

28	-0.0108
29	-0.0065
30	0.0000
31	0.0060
32	0.0095
33	0.0092
34	0.0055
35	-0.0000
36	-0.0052
37	-0.0082
38	-0.0080
39	-0.0048
40	0.0000

Review Questions

6.1 What are the addresses of the program memory address space and data memory address space in the on-chip memory of C50 in the DSP starter kit where user programs and data may be stored?

6.2 In the C5X DSK, can the data be stored in the space 3000-37FF?

6.3 What is the command for invoking the DSK assembler, debugger for 5X kit?

6.4 What is the use of the .mmregs DSP assembler directive?

6.5 If the program given in Program6_2.asm is executed in single-step mode, what are the values of AR0, ARl, ACC, TREG0 and PREG after each of the instructions are executed?

6.6 If the program given in Program6_3.asm is executed in single-step mode, what are the values of DP, AR0, AR7, ACC, TREG0 and PREG after each of the instructions are executed?

6.7 If the program given in Program6_4.asm is executed in single-step mode, what are the values of AR0, ARl, ACC, TREG0 and PREG after each of the instructions are executed?

6.8 Write an assembly language program in C5X to transfer a block of 256 16-bit words of data from program memory address space 2000h to data memory address space starting with 2000h.

6.9 Write an assembly language program in C5X to transfer a block of 256 16-bit words of data from data memory address space 2000h to program memory address space starting with 2500h.

6.10 Explain with an example how the circular addressing mode is useful for real time processing of signals. Write a program which illustrates the operation of the circular addressing mode.

6.11 Write an assembly language program in C5X which generates the first 20 numbers corresponding to the infinite sequence 5, 3, 8, 11, 19, ...

6.12 Write an assembly language program in C5X to convolve two sequences of length 32 and 48 respectively.

6.13 Write an assembly language program in C5X to convolve two sequences of length 32 and 48 respectively. One of the sequences should be shifted by one position towards right (to higher memory location) after the convolution is completed.

6.14 Explain with a block diagram how the data transmission/reception is achieved using the on-chip serial port of C5X. How is the data transmission/reception rate programmed?

6.15 Explain how the data transmission to an external device is achieved through the on-chip C5X serial port using (a) polling and (b) interrupt.

6.16 If the data transmission through the serial port of C5X is achieved using interrupt, what should be the control word for the interrupt mask register (IMR) if (a) transmission alone is required and (b) both serial transmission as well as reception are required. Give the program required for setting the IMR in both of the cases. What determines the data transmission/reception rate?

6.17 How does the on-chip timer of C5X affect the operation of the serial port?

6.18 What is the maximum frequency of the signal which may be digitised using AIC TLC320C40?

6.19 How is the data transfer between the AIC and the 5X processor achieved in the DSK kit.

6.20 For the AIC TLC320C40, the value of Ta is chosen to be 24h to obtain the switched capacitor frequency of 288 kHz. What is the value of the master clock frequency fed from the C5X to AIC?

6.21 For a sampling rate of 9 kHz, what should be the value of Tb of AIC, if the switched capacitor frequency is 288 kHz?

6.22 Explain how the values of the registers Ta, Tb, Ra and Rb of the AIC determine the mode of operation of the AIC?

6.23 What is the function performed by the switched capacitor BP filter used in AIC? What happens if it is not included?

6.24 In the AIC initialisation routine why is the global memory register programmed?

6.25 Distinguish between the primary and secondary communication modes of AIC? How is the type of communication mode chosen through the program?

6.26 When a data is written into DXR of serial port when is it ANDed with 0FFFCh and why is it ANDed?

6.27 What is the use of the idle instruction in the AIC initialisation routine?

6.28 Explain how a waveform may be digitised and converted back to analog signal in real time using the 5X DSK and AIC TLC320c40. Explain any two methods for outputting a signal whose frequency is (i) half that of the signal applied to the AIC input and (ii) double that of the input signal (*Hint:* You may do this by choosing either the Tx and Rx rates to be same or choose them to differ by a factor of 2).

6.29 A program given for waveform generation using C5X (e.g., Program6_14.asm-Program6_16.asm), the AIC is not initialised. What parameters determine the period of the AIC? How can this period be varied without changing the ton and toff values?

6.30 A digital LP filter is implemented in C5X and it uses the AIC TLC320C40. The AIC is programmed so as to exclude the switched capacitor BP filter. Explain how the A/D conversion rate of the AIC may be determined by finding the frequency response of this filter.

6.31 A digital LP filter is implemented in C5X and it uses the AIC TLC320C40. The output of this filter is zero at frequencies close to DC. What could be the reason for this? In case non-zero output is required for frequencies of the order of 100 Hz, what should be done?

6.32 Explain how the FSK signal may be demodulated using a C5X program.

Self Test Questions

6.1 The assembler directive of C5X which permits the memory-mapped registers to be denoted by their names instead of their actual memory address is ———.
(a) .ps (b) .mmregs (c) .entry (d) .include

6.2 The assembler directive of C5X which is similar to ORG (origin) instruction of 8085 assemblers is ———.
(a) .ps (b) .mmregs (c) .entry (d) .include

6.3 The assembler directive of C5X which indicates the location from where the program will be executed is ———.
(a) .ps (b) .mmregs (c) .entry (d) .include

6.4 The assembler directive of C5X which permits a main program to be split into more than one modules and assemble them individually is ———.
(a) .ps (b) .mmregs (c) .entry (d) .include

6.5 If the debugger is invoked by typing in the command dsk5d, the com port to which the DSK is assumed to be connected to PC is ———.
(a) coml (b) com2 (c) com3 (d) com4

6.6 In Program 6_la.asm, a constant is loaded to ACC using the instruction LACC #0004h, 4. The actual value of the constant loaded to ACC is ———.
(a) 0004h (b) 0008h (c) 0010h (d) 0020h
(e) 0040h

6.7 In Program 6_2.asm, a constant is added to ACC using the instruction ADD #001 lh, 2. The actual value of the constant added to ACC is ———.
(a) 001 lh (b) 0022h (c) 0044h (d) 0088h

6.8 When the instruction MPY 10h is executed, one of the operands for the multiplier is taken from the CPU register ———.
(a) ACC (b) PREG (c) TREG0 (d) TREGl

6.9 When the instruction MPY 10h is executed, the output of the multiplier is stored into the CPU register ———.
(a) ACC (b) PREG (c) TREG0 (d) TREGl

6.10 When the instruction BANZ loopl, AR1 is executed which of the following operations are performed?
(a) branching occurs to loop 1 if the current AR is non-zero.
(b) current AR content is decremented before branching
(c) branching occurs if AR1 is non-zero
(d) ARP made to point to AR1 after branching

6.11 The timer of C5X is an on-chip-counter and interrupt (TINT) is generated each time the counter reaches the state ———.
(a) down, all I's (b) down, all 0's
(c) up, all I's (d) up, all 0's

6.12 The signals used to connect the transmit pins of the serial port of C5X with the receiving device for data transmission are ———.
(a) CLKX (b) DX (c) FSX (d) CLKR
(e) RX (f) FSR

6.13 The signals used to connect the receive pins of the serial port of C5X with the transmitting device data reception are ———.
(a) CLKX (b) DX (c) FSX (d) CLKR
(e) RX (f) FSR

6.14 The registers of on-chip serial port of C5X which cannot be directly accessed by the CPU are
(a) SPC (b) DXR (c) DRR (d) XSR
(e) RSR

6.15 The registers of on-chip serial port of C5X which shifts the data serially in or out are
(a) SPC (b) DXR (c) DRR (d) XSR
(e) RSR

6.15 Writing the data into ——— of serial port of C5X generates the XINT.
(a) SPC (b) DXR (c) DRR (d) XSR
(e) RSR

6.16 Writing the data into ——— of serial port of C5X generates the RINT.
(a) SPC (b) DXR (c) DRR (d) XSR
(e) RSR

6.17 The signals which initiate the serial shifting of the data in the on-chip serial port at the beginning of every frame in burst mode and for the first frame in the synchronous transfer mode for the shift registers used at the receive and transmit side respectively are ———.
(a) CLKX (b) DX (c) FSX (d) CLKR
(e) RX (f) FSR

6.18 In the AIC, the registers which are used to obtain the switched capacitor frequency of 288 kHz are
(a) TX counter A, TX counter B
(b) TX counter A, RX counter A
(c) RX counter A, RX counter B
(d) TX counter B, RX counter A

6.19 In the AIC, the registers which are used to obtain the required sampling frequency are ———.
(a) TX counter A, TX counter B
(b) TX counter A, RX counter A
(c) RX counter A, RX counter B
(d) TX counter B, RX counter B

6.20 To program the AIC, the secondary communication is initiated by choosing the d0, dl bits of the data transmitted through the DX pin in the primary communication mode as ———, ———.
(a) 0, 0 (b) 1, 0 (c) 0, 1 (d) 1,1

6.21 256 samples of a sine wave of frequency 1000 Hz are stored in data memory organised as a circular buffer of size 256. The data is read one after another from this buffer and transmitted through the DX pin infinitely. The frequency at the output of the AIC is ——— Hz.
(a) 1000 (b) 500 (c) 2000 (d) 4000

6.22 256 samples of a sine wave of frequency 1000 Hz are stored in data memory organised as a circular buffer of size 256. The data is read one after another from this buffer and the alternate data is transmitted through the DX pin infinitely. The frequency at the output of the AIC is ——— Hz.
(a) 1000 (b) 500 (c) 2000 (d) 4000

6.23 256 samples of a sine wave of frequency 1000 Hz are stored in data memory organised as a circular buffer of size 256. The data is read one after another from this buffer and each data is transmitted twice through the DX pin infinitely. The frequency at the output of the AIC is ——— Hz.
(a) 1000 (b) 500 (c) 2000 (d) 4000

6.24 Sampling frequency of the AIC is programmed to be 8 kHz. A LP filter with cutoff frequency of 4 kHz is implemented using the kit. The antialiasing filter at the input section of AIC is not enabled. If a signal of frequency 9 kHz is fed to the AIC, the output of the LP filter has a frequency of ———kHz.
(a) 9 (b) 1 (c) 5 (d) 7

7 ARCHITECTURE OF TMS320C3X

INTRODUCTION 7.1

The TMS320C3X series of digital signal processors (DSPs) are high-performance CMOS 32-bit floating-point devices in the TMS320 family of single-chip DSPs. The ′C3X devices integrates both system control and math-intensive functions on a single controller. This system integration allows fast, easy data movement and high-speed numeric processing performance. Extensive internal busing and a powerful DSP instruction set provide the devices with the speed and flexibility to execute upto 60 million floating-point operations per second (MFLOPS) and 30 million fixed-point instructions per second (MIPS). The devices also feature a high degree of on-chip parallelism that allows users to perform up to 11 operations in a single instruction.

AN OVERVIEW OF TMS320C3X DEVICES 7.2

The ′C3X family consists of three members: the ′C30, ′C31 and ′C32. These processors can perform parallel multiply and arithmetic logic unit (ALU) operations on integer or floating-point data in a single cycle. These processors consist of the following:

general-purpose register file,
program cache,
dedicated auxiliary register arithmetic units (ARAU),
internal dual-access memories,
direct memory access channel (DMA) supporting concurrent I/O,
large address space,
multiprocessor interface,
internal and externally generated wait states,
external interface ports,
timers,
serial ports and
multiple-interrupt structure.

The speed, memory and list of peripherals for the ′C3X family processors are given in Table 7.1. The ′C30 is the first member of the ′C3X generation. It differs from the ′C31 and ′C32 by offering more ROM (4K), RAM (2K), a second serial port, and a second external bus. The ′C31 and ′LC31 are the second members of the ′C3X generation. They are low-cost 32-bit floating-point DSPs which have a boot-loader program, 2K RAM, single external port, single serial port and are available in 3.3-V operation (′LC31). The ′C32 is the newest member of the ′C3X generation. They are enhanced versions of the ′C3X family and the lowest cost floating-point processors. These enhancements include a variable-width memory interface, two-channel DMA coprocessor with configurable priorities, flexible boot loader and a relocatable interrupt vector table.

Table 7.1 *TMS320C3X family processor details*

Device name	*Frequency cycle time (MHz/ns)*	*Memory (words)*		*Peripherals*
		On-chip	*Off-chip*	

′C30
(5 V)
	27/75
	33/60
	40/50
	50/40
		RAM = 2K
		ROM = 4 K
		Cache = 64
			16Mx32
			8Kx32
				Serial port = 2
				DMA channel = 1
				Timers = 2

′C31
(5 V)
	27/75
	33/60
	40/50
	50/40
	60/33
		RAM =2K
		ROM is Boot loader
		Cache = 64
			16Mx32	Serial port = 1
				DMA channel = 1
				Timers = 2

′LC31
(3.3 V)
	33/60
	40/50
		RAM =2K
		ROM is Boot loader
		Cache = 64
			16Mx32	Serial port = 1
				DMA channel = 1
				Timers = 2

′C32
(5 V)
	40/50
	50/40
	60/33
		RAM =512K
		ROM is Boot loader
		Cache = 64
			16Mx
			32/16/8
				Serial port = 1
				DMA channel = 2
				Timers = 2

It may be noted from Table 7.1 that in the ′C3X family processors the memory configuration is the same for ′C30 and ′C31 but the speed of the DSPs and the peripheral present are different. The ′C3X supports a wide variety of system applications from host processor to dedicated coprocessor. High-level language is implemented more easily through a register-based architecture, large address space, powerful addressing modes, flexible instruction set and well-supported floating-point arithmetic.

INTERNAL ARCHITECHTURE 7.3

The block diagram of TMS320C3X is given in Fig. 7.1. The ′C3X processors have the following four major blocks.

1. Central Processing Unit (CPU)
2. Memory unit (RAM, ROM and Cache)
3. Peripherals (Serial ports, Timer, etc.) and
4. DMA controller

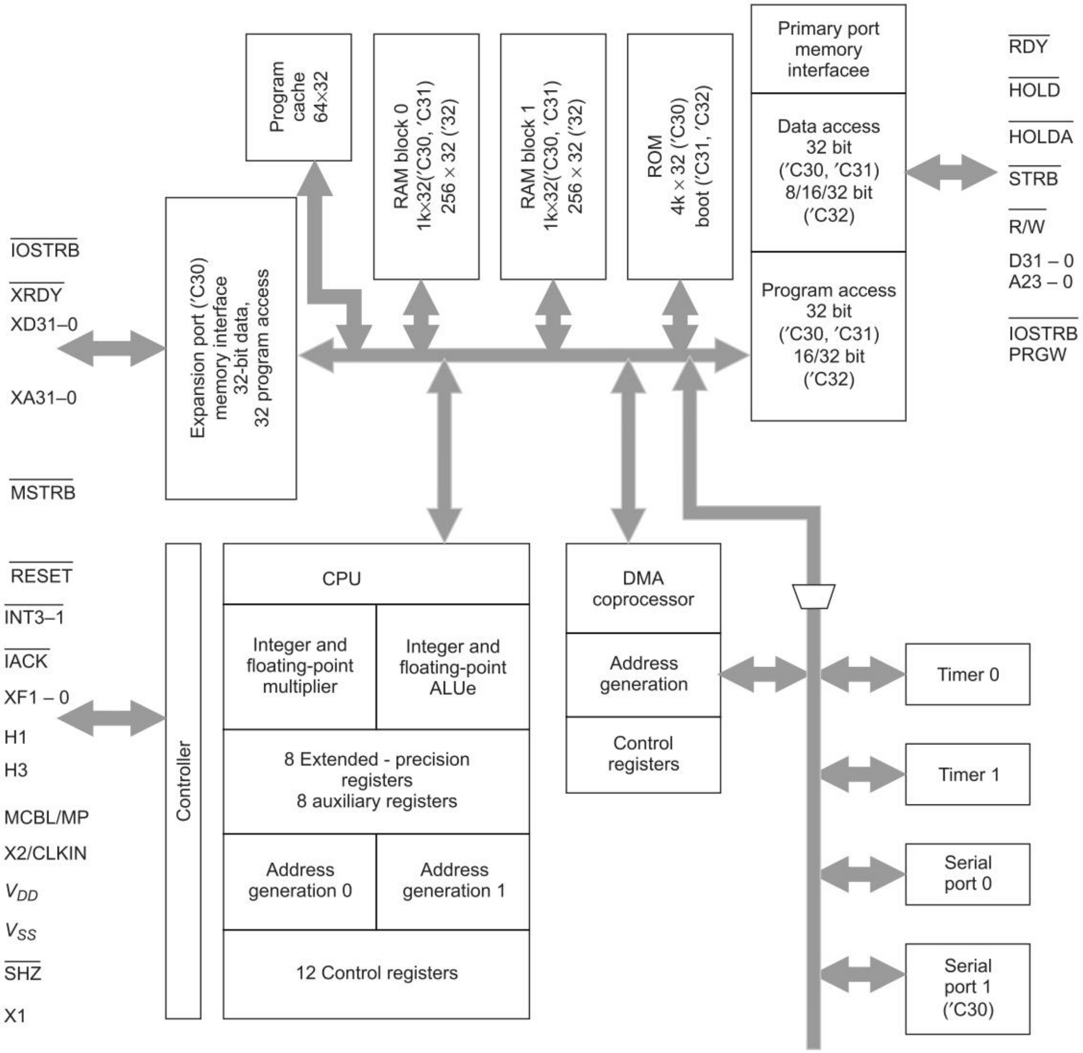

Program cache 64×32
RAM block 0 1k×32('C30, 'C31) 256 × 32 ('32)
RAM block 1 1k×32('C30, 'C31) 256 × 32 ('32)
ROM 4k × 32 ('C30) boot ('C31, 'C32)
Primary port memory interfacee
Data access 32 bit ('C30, 'C31) 8/16/32 bit ('C32)
Program access 32 bit ('C30, 'C31) 16/32 bit ('C32)
RDY
HOLD
HOLDA
STRB
R/W
D31 – 0
A23 – 0
IOSTRB
PRGW
IOSTRB
XRDY
XD31–0
XA31–0
MSTRB
Expansion port ('C30) memory interface 32-bit data, 32 program access
RESET
INT3–1
IACK
XF1 – 0
H1
H3
MCBL/MP
X2/CLKIN
V_{DD}
V_{SS}
SHZ
X1
Controller
CPU
Integer and floating-point multiplier
Integer and floating-point ALUe
8 Extended - precision registers 8 auxiliary registers
Address generation 0
Address generation 1
12 Control registers
DMA coprocessor
Address generation
Control registers
Timer 0
Timer 1
Serial port 0
Serial port 1 ('C30)

Fig. 7.1 *The TMS320C3X block diagram*

CENTRAL PROCESSING UNIT (CPU) 7.4

The ′C3X devices have a register-based CPU architecture. The CPU consists of the following units:

- Internal buses (CPU1/CPU2 and REG1/REG2)
- Floating-point/integer multiplier
- Arithmetic logic unit (ALU)
- 32-bit barrel shifter
- Auxiliary register arithmetic units (ARAUs)
- CPU register file
- The various CPU components of ′C3X processors are shown in Fig. 7.2.

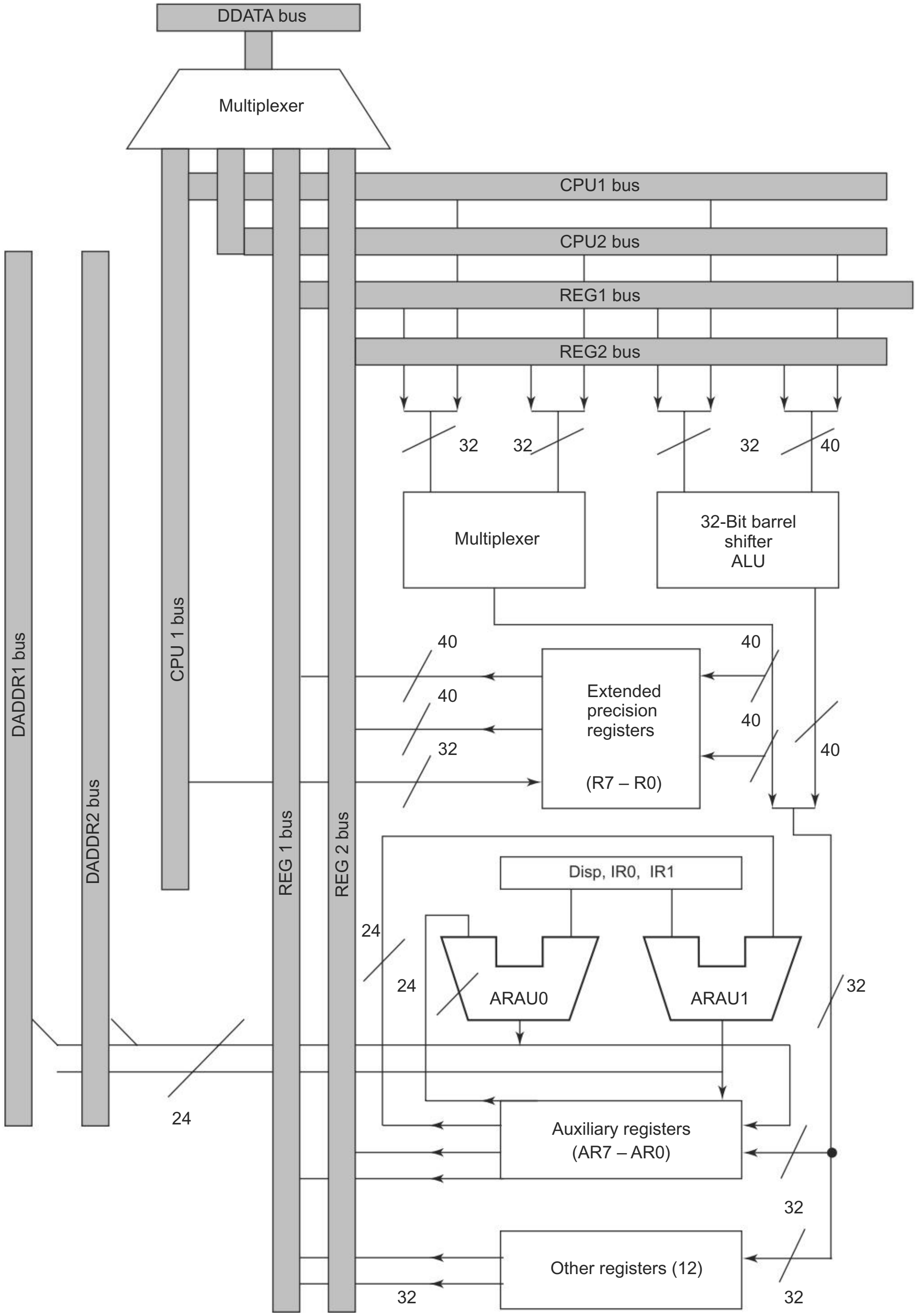


Fig. 7.2 *CPU components of 'C3X*

7.4.1 CPU Internal Buses

The CPU unit consist of four buses, CPU1, CPU2, REG1 and REG2. The data bus DDATA carries data to the CPU over the CPU1 and CPU2 buses. The CPU1 and CPU2 buses can carry two data memory operands to the multiplier, ALU and the register file for every machine cycle. Internal to the CPU are the two buses REG1 and REG2, these buses can carry two data values from the register file to multiplier and ALU for every machine cycle. The CPU internal buses are shown in the Fig. 7.2.

7.4.2 Floating-Point/Integer Multiplier

The multiplier performs multiplications on 24-bit integer and 32-bit floating-point values in a single cycle. The ′C3X implementation of floating-point arithmetic allows for floating-point or fixed-point operations at speeds upto 33 ns per instruction cycle. To get further higher speed, parallel instructions can be used. The parallel instructions will perform a multiply and an ALU operation in a single cycle. When the multiplier performs floating-point multiplication, the inputs are 32-bit floating-point numbers and the result is a 40-bit point floating number, whereas in the case of integer multiplication, the input data is 24 bits and yields a 32-bit result.

7.4.3 Arithmetic Logic Unit (ALU) and Barrel Shifter

The ALU performs single-cycle arithmetic and logical operations. It performs operations on 32-bit integer, 32-bit logical and 40-bit floating-point data. It also performs integer and floating-point conversions in a single cycle. The results of the ALU are always maintained at 32-bit integer or 40-bit floating-point formats. The barrel shifter is used to shift the operands upto 32 bits left or right in a single cycle.

7.4.4 Auxiliary Register Arithmetic Unit (ARAU)

The ′C3X family processors consists of two ARAUs (ARAU0 and ARAU1). These two units can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU and they are used for indirect addressing mode. The ARAUs support address displacement and the displacement can be indicated in the assembly code using the displacement field or the two index register (IR0 and IR1) contents. They also support circular and bit-ieversed addressing modes.

CPU REGISTER FILE 7.5

The ′C3X processors consists of 28 registers in a multiport register. These registers are tightly coupled to the CPU. The list of the registers present in the CPU register file are given below.

(a)	Extended-precision registers	(R7-R0)
(b)	Auxiliary registers	(AR7-AR0)
(c)	Data page pointer	(DP)
(d)	Index registers	(IR1 and IR0)
(e)	Block size register	(BK)
(f)	System stack-pointer	(SP)
(g)	Status register	(ST)
(i)	CPU/DMA interrupt-enable’ register	(IE)
(j)	CPU interrupt flag register	(IF)
(k)	I/O flag register	(IOF)
(1)	Repeat start-address register	(RS)

(m) Repeat end-address register (RE)
(n) Repeat count register (RC)

These registers can be operated on by the multiplier and ALU and can be used as general-purpose 32-bit registers. The eight extended-precision registers are especially suited for maintaining extended-precision 40-bit floating-point results. The eight ARs used for a veriety of indirect addressing modes and also as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack management, processor status, interrupts and block repeat, etc.

7.5.1 The Extended-precision Registers (R7-R0)

All the eight extended precision registers are 40 bits in size. They are capable of storing and supporting operations on 32-bit integer and 40-bit floating-point numbers. If the operands are floating-point numbers, bits 39-0 (40 bits) are used. The extended-precision register floating-point format is illustrated in Fig. 7.3. The bits 39-32 are dedicated to store the exponent *(e)* and bits 31-0 are for the mantissa part of the floating-point number. In the mantissa part of the floating-point number, bit 31 is assigned as sign bit and bits 30-0 are dedicated for the fraction.

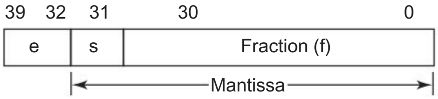


Fig. 7.3 *Extended-precision register floating-point format*

For integer operands, either signed or unsigned, only bits 31-0 (32 bits) are used. Bits 39-32 remain unchanged. The extended-precision register integer format is shown in Fig. 7.4.

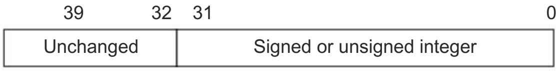


Fig. 7.4 *Extended-precision register integer format*

7.5.2 Auxiliary Registers (AR7-AR0)

There are eight ARs, each 32-bit in size,. They are used for the generation of 24-bit addresses. These registers can be accessed by the CPU and modified by the two ARAUs. They can also be used as loop counters in indirect addressing modes or as 32-bit general purpose registers that can be modified by the multiplier and ALU.

7.5.3 Data-page Pointer (DP)

The data page pointer is a 32-bit register used to point the page of the data being addressed by the direct addressing mode. In ′C3X processors the number of pages present are 256 and each page contains 64K words (each of 32 bits). To point these 256 pages LSB (bits 7-0) of the DP are used and bits 31-8 are reserved (these bits should always be kept zero). The DP format is given in Fig. 7.5. To load DP, LDP instruction is used.

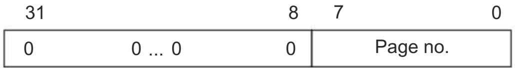


Fig. 7.5 *Data-page pointer format*

7.5.4 Index Registers (IR1,IR0) and Block Size Register (BK)

The index and block size registers are 32-bit each in size. The index registers are used by the ARAU for indexing the address (refer Section 8.2.3) whereas the block size register is used by the ARAU to specify the data block size in circular addressing mode (refer Section 8.2.6).

7.5.5 System Stack Pointer (SP)

The system stack pointer is a 32-bit register, which contains the address of the top of the system stack. The system stack fills from low-memory address to high-memory address. The system stack configuration is shown in Fig. 7.6. The SP always points to the last element pushed onto the stack. A push performs preincrement, whereas a pop postdecrement of the system stack pointer.

The program counter is pushed onto the system stack on subroutine calls, traps and interrupts. It is popped from the system stack on returns. The system stack can be pushed using PUSH & PUSHF and popped by POP & POPF instructions.

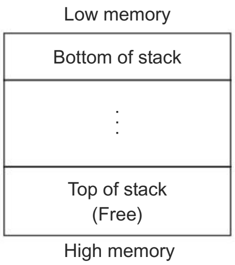
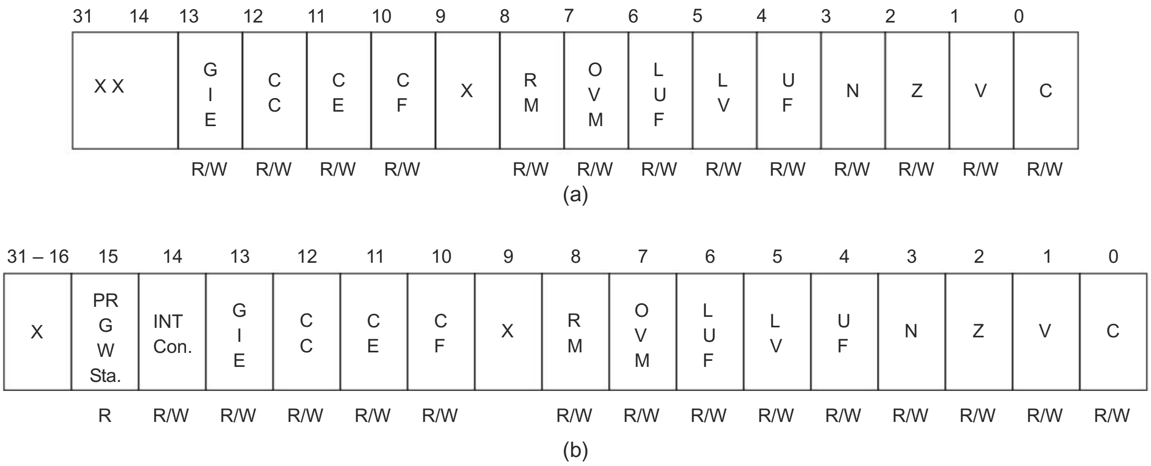


Fig. 7.6 *System stack configuration*

7.5.6 Status Register (ST)

The status register is 32-bit in size, out of which bits 13-10 and 8-0 are used; rest of the bits are reserved (i.e. the reserved bits should always be set zero). The status register format for ′C30, ′C31 and ′C32 is shown in Fig. 7.7. The status register contains global information about the state of the CPU. There are certain conditional flag bits present in the status register. Based on the results of the operations such as load and store, as well as arithmetic and logical functions, these flag bits are set or cleared. The results could be zero, negative, carry, overflow, etc.



31 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X X	GIE	CC	CE	CF	X	RM	OVM	LUF	LV	UF	N	Z	V	C
	R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

(a)

31 – 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	PRGW Sta.	INT Con.	GIE	CC	CE	CF	X	RM	OVM	LUF	LV	UF	N	Z	V	C
	R	R/W	R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

(b)

Fig. 7.7 *(a) The status register format (′C30 and ′C31); (b) The status register format CC32)*

Note: (1) *x - reserved bit, read as 0 and* (2) *R- read, W - write*

It is possible to read the information in the status register bits, it can be written as well. When the status register is loaded, the contents of the source operand replace the current contents of ST bit-for-bit, regardless of the state of any bits in the source operand. The source operand content can be written as such in the status register. This allows the status register to be saved and restored. At system reset, logic 0 is written to this register. Table 7.2 defines the status register bit names and their functions.

Table 7.2 *Status register bits summary*

Bit name	*Function*
C	Carry condition flag
V	Overflow condition flag
Z	Zero condition flag
N	Negative condition flag
UF	Floating-point underflow condition flag
LV	Latched overflow condition flag
LUF	Latched floating-point underflow condition flag
OVM	Overflow flag. The overflow mode flag affects only integer operations If OVM = 0, the overflow mode is turned off If OVM = 1, integer results overflowing in the positive direction are set to the most positive, 2s-complement number (7FFFFFFFh), and integer results overflowing in the negative direction are set to the most negative 32-bit, 2s-complement number (8000 0000h)
RM	Repeat mode flag If RM = 1, the PC is modified in either the repeat-block or repeat-single mode
CF	Cache freeze. Enables or disables the instruction cache. Set CF = 1 to freeze the cache (cache is not updated). When CF = 0, the cache is automatically updated by instruction fetches from external memory
CE	Cache enable. CE enables or disables the instruction cache. Set CE = 1 to enable the cache, Set CE = 0 to disable the cache
CC	Cache clear. CC = 1 invalidates all entries in the cache
CIE	Global interrupt-enable. If GIE = 1, the CPU responds to an enabled interrupt. If GIE = 0, the CPU does not respond to an enabled interrupt
INT Con.	Interrupt configuration('C32 only) Sets the external interrupt signals $\overline{\text{INT3}}$-$\overline{\text{INT0}}$ for level- or edge-triggered interrupts 0: All the external interrupts ($\overline{\text{INT3}}$-$\overline{\text{INT0}}$) are configured as level-triggered interrupts. Multiple interrupts may be triggered when the signal is active for a long period of time 1: All the external interrupts ($\overline{\text{INT3}}$-$\overline{\text{INT0}}$) are configured as edge-triggered interrupts Edge and duration are required for ail interrupts to be recognised
PRGW Sta.	Program width status CC32 only) Indicates the status of the external input PRGW pin 0: Instruction fetches use one 32-bit external program memory read 1: instruction fetches use two 16-bit external program memory reads

7.5.7 CPU/DMA Interrupt Enable (IE) Register and CPU Interrupt Flag (IF) Register

The CPU/DMA interrupt enable (IE) register is a 32-bit register. The CPU IE bits are in locations 10-0 and DMA IE bits are in locations 26-16. Rest of the bits are reserved. A logic 1 set in bit 26 of IE will enable the DMA interrupts, whereas a logic 0 set will disable DMA interrupts and similarly bit 10 of IE can be used to enable/disable interrupts.

The CPU interrupt flag (IF) register is also a 32-bit register. Bits 10-0 is used as IFs. The logic 1 present in an IF register bit indicates that the corresponding interrupt is set, and a logic 0 indicates that it is not set. Whenever an interrupt occurs the IF bits are set logic 1. They may also be set and cleared through software to cause an interrupt or to clear. At reset, logic 0 is written in the IE as well as IF registers. Figure 7.8 shows the format of IE and IF registers.

Interrupt enable register

31 – 27	26	25	24	23	22	21	20	19	18	17	16
X X X	EDINT DMA	ETINT1 DMA	ETINT0 DMA	ERINT1 DMA	EXINT1 DMA	ERINT0 DMA	EXINT0 DMA	EINT3 DMA	EINT2 DMA	EINT1 DMA	EINT0 DMA
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15 – 11	10	9	8	7	6	5	4	3	2	1	0
X X X	EDINT CPU	ETINT1 CPU	ETINT0 CPU	ERINT1 CPU	EXINT1 CPU	ERINT0 CPU	EXINT0 CPU	EINT3 CPU	EINT2 CPU	EINT1 CPU	EINT0 CPU
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Interrupt flag register

31 – 10	10	9	8	7	6	5	4	3	2	1	0
X X X	TINT1 CPU	TINT0 CPU	RINT1 CPU	XINT1 CPU	RINT0 CPU	XINT0 CPU	INT3 CPU	INT2 CPU	INT1 CPU	INT0 CPU	
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

Fig. 7.8 *Interrupt enable and interrupt flag register format*

7.5.8 I/O Flag (IOF) Register

The ′C3X processors have two dedicated external pins XF0 and XF1. The IOF register controls the function of these pins. These pins can be configured for input or output using the IOF register. Figure 7.9 shows the format of an IOF register. This will be useful in interlocked operations of ′C3X processors. Table 7.3 shows the bit-field names and their functions of IOF register.

31 – 8	7	6	5	4	3	2	1	0
X	INXF1	OUTXF1	$\overline{\text{I}}$/OXF1	*X*	INXF0	OUTXF0	$\overline{\text{I}}$/OXF0	*X*
	R	*R/W*	*R/W*		*R*	*R/W*	*R/W*	

X - reserved bit, read as 0, R – read, W – write

Fig. 7.9 *I/O flag register format*

Table 7.3 *IO flag register bits summary*

Bit name	*Function*
$\bar{I}$/OXF0	If 0, XF0 is configured as a general-purpose input pin. If 1, XF0 is configured as a general-purpose output pin
OUTXF0	Data output on XF0
INXF0	Data input on XF0. A write has no effect
$\bar{I}$/OXF1	If 0, XF1 is configured as a general-purpose input pin. If 1, XF1 is configured a general-purpose output pin
OUTXF1	Data output on XF1
INXF1	Data input on XF1. A write has no effect

7.5.9 Block-Repeat Registers (RS, RE) and Repeat-Count (RC)

The block-repeat start address, end address and repeat counter registers are 32-bits in size. The start address register (RS) contains the starting address and end address register (RE) contains the end address of the block of program memory to be repeated, when the CPU is operating in the repeat mode. The repeat-count (RC) register is used to specify the number of times a block of code is to be repeated. If RC has the number *n,* the block is executed $n + 1$ times.

7.5.10 Program Counter (PC) and Instruction Register (IR)

The program counter (PC) and Instruction register (IR) are 32-bit registers and they are not in the register file. The PC contains the address of the next instruction to be fetched. The IR holds the instruction op-code during the decode phase of the instruction. This register is used by the instruction decode control circuitry and is not accessible to the CPU.

MEMORY ORGANISATION 7.6

The total memory space of a ′C3X processor is 16 million (16M) 32-bit words. This space contains Program, data and I/O spaces. The ′C3X processors have on-chip RAM, ROM and cache memories.

The memory organisation and internal buses are shown in Fig. 7.10. In ′C3X there are two RAM blocks, Block 0 and Block 1, (each of IK x 32 bits in ′C30 and ′C31 and each of 256 x 32 bits in ′C32) and a ROM block (4K x 32 bits only in ′C30 and boot loader in ′C31 and ′C32). The RAM and ROM blocks are capable of supporting two CPU accesses (dual access memory) in a single cycle. A 64 x 32 bit instruction cache is present to store often-repeated sections of code. This will reduce the number of off-chip accesses.

7.6.1 Memory Internal Buses

The high performance of ′C3X processors is due to the internal buses and parallelism. The memory buses are program bus, data bus and DMA buses. The separate program bus consists of program address (PADDR) bus and program data (PDATA) bus. There are two data address buses (DADDR1 and DADDR2) and one data (DDATA) bus. The DMA bus contains DMA address (DMAADDR) bus and a DMA data (DMADATA) bus. These buses allow parallel instruction code fetch from program memory, two data accesses and DMA access. These buses connect on-chip memory, off-chip memory and on-chip peripherals. All address buses are 24 bits and data buses are 32bits in size.

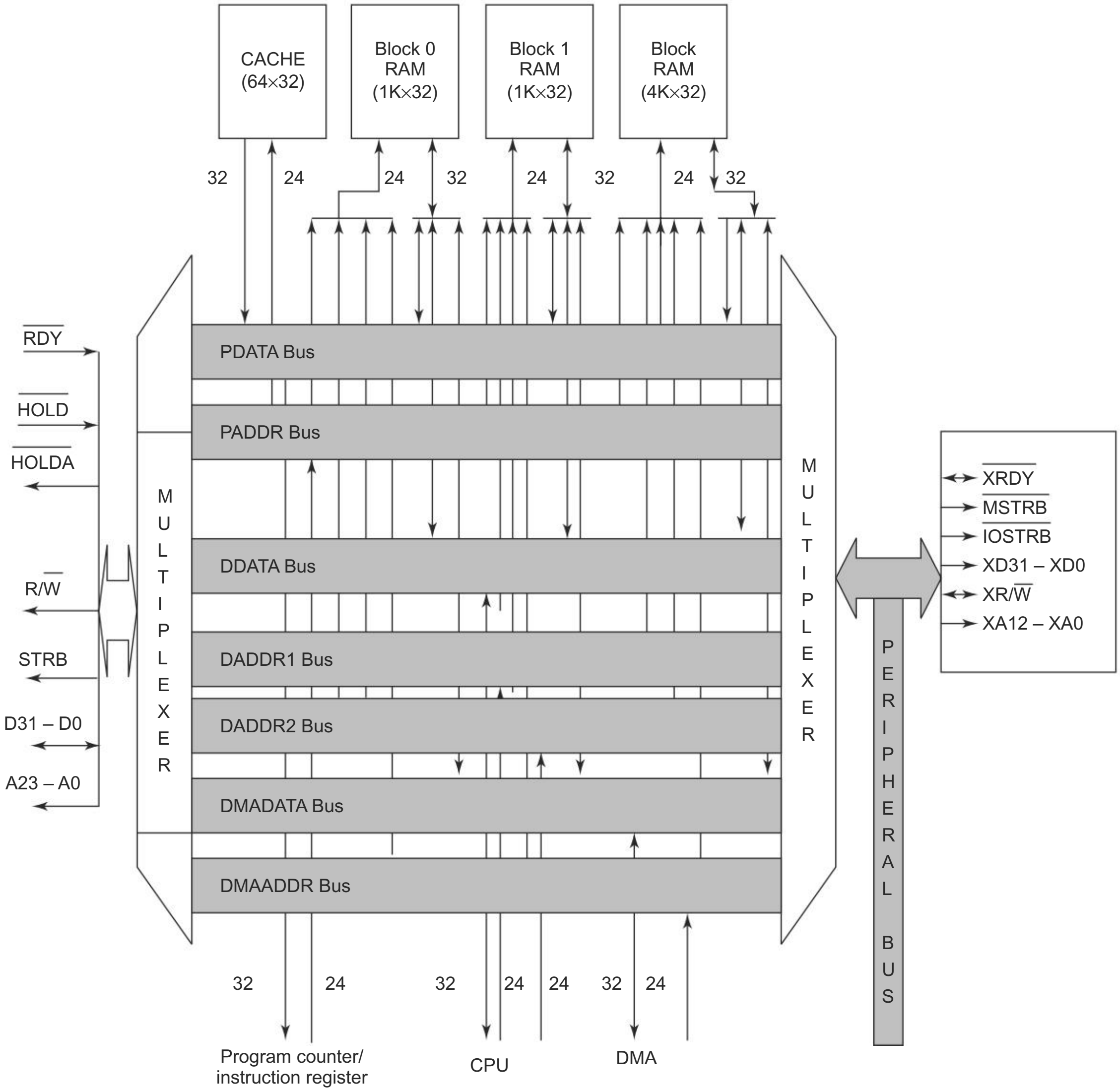


Fig. 7.10 *Memory organisation*

The PC is connected to PADDR bus and the IR is connected to PDATA bus. These buses can fetch a single instruction word for every machine cycle. The two data address buses are connected to the AR unit, which generates the data memory addresses and the data bus is connected to the CPU over CPU1 and CPU2 buses, which carry the data to the CPU.

The DMA controller is connected with a 24-bit address bus (DMAADDR) and a 32-bit data bus (DMADATA). These buses allow the DMA to perform memory access in parallel with the memory accesses occurring from the data and program buses.

7.6.2 Memory Maps

The memory map of ′C3X devices depend on the logic level applied to the external pin MC/$\overline{\mathrm{MP}}$ or MCBL/$\overline{\mathrm{MP}}$. For logic 0 the processor runs in microprocessor mode, whereas for logic 1 it runs in microcomputer mode. The number of address lines for the memory is 24 bits and start and end address of the memory are 000000h-FFFFFFh.

Microprocessor mode In microprocessor mode, the 4K on-chip ROM is not mapped into the ′C3X memory map. The memory maps of ′C30, ′C31 and ′C32 devices are almost similar. The memory map for microprocessor mode of ‘C30, ‘C31 and ‘C32 can be found in C3X User’s guide [1996].

The external memory port with $\overline{\mathrm{STRB}}$ signal active accesses the address locations 000000h-FFFFFFh (8.192M words). In this space, locations 0h-03Fh (192 words) consists of interrupt, trap vectors and reserved locations in ′C30 and ′C31, whereas in ′C32 location 0h alone is containing the reset vector. The locations 800000h-807FFFh (32K words) are reserved in ′C31 and ′C32. In ′C30 the total 32K size is partitioned into four 8K segments. The first and third 8K segments are mapped to the expansion bus and can be accessed when $\overline{\mathrm{MSTRB}}$ & $\overline{\mathrm{IOSTRB}}$ signals are active respectively. The second and fourth 8K segments are reserved locations. In all the three processors, locations 808000h-8097FFh (6K words) are memory-mapped for the peripheral bus registers.

The ′C30 and ′C31 processors have two IK word RAM blocks starting from 809800h to 809FFFh, whereas in ′C32 it has two 256-word blocks starting from 87FE00h to 87FFFFh. The locations 80A000h-FFFFFFh (7.96M words) of ′C30 and ′C31 are mapped to external space and accessed when $\overline{\mathrm{STRB}}$ signal is active. In ′C32 device, locations 809800h-80FFFFh (26K words) and 830000h-87FDFFh (319.5K words) are reserved. The locations 810000h-82FFFFh (128K words), 880000h-8FFFFFh (512K words) and 900000h-FFFFFFh (7.168M words) are mapped to external space and can be accessed when $\overline{\mathrm{IOSTRB}}$, $\overline{\mathrm{STRBO}}$ and STRB1 signals are active.

Microcomputer Mode In microcomputer mode the 4K on-chip ROM in ′C30 and boot loader in ′C31 and ′C32 are mapped into locations 000000h-FFFFFFh. The locations 001000h-7FFFFFh are mapped to external space similar to microprocessor mode. Rest of the locations are similar to the microprocessor mode in all the devices. The memory map of ′C30, ′C31 and ′C32 devices for microcomputer mode can be found in C3X User’s guide [1996]. The locations 000h-0BFh (192 words) in ′C30 are used for reset, interrupt and trap vectors and reserved locations, whereas in ‘C31 the locations 809FC1h-809FFFh (the last 63 words of internal RAM block 1) are used for interrupt and trap branches.

7.6.3 Memory Mapped Registers (Peripherals)

The ′C3X devices have on-chip DMA controller, the peripherals such as timers and serial ports, and primary and expansion ports. All these units have programmable registers. By programming these registers the operations such as DMA transfer, serial port communication, external device interface, etc. can be performed. These peripheral registers are memory mapped from the starting address 808000h to 8097FFh. Each peripheral occupies a 16-word region of the memory map.

CACHE MEMORY 7.7

A 64 x 32-bit instruction cache increases the processor performance. When a section of code is repeatedly accessed by the processor from the off-chip memory, the cache stores the code in the cache. This reduces the number of off-chip accesses needed. Because of this facility the codes can be stored on even slower, low-cost off-chip memories. It is to be noted that the cache also frees the external buses from source code fetches, so that the external bus can be used by the DMA or other system elements. The cache functions automatically, with no user intervention. Only instruction words are fetched from cache, whereas the data reads and writes and bypasses the cache. The instruction fetches from the on-chip memory do not modify the cache.

7.7.1 Cache Control Bits

There are three control bits for the efficient management of the cache. They are cache clear (CC), cache enable (CE) and Cache freeze (CF) bits. These control bits are present in the status register. All the 64 words present in the cache are having a flag bit P. At reset the cache is cleared and zero is written to these three **bits.**

Cache Clear Bit (CC) This bit is used to clear all the entries present in the 64 words of the cache and the P flags. Writing logic 1 to this bit will clear the cache and the CC bit present in the status register is cleared after the cache is being cleared.

Cache Enable Bit (CE) This bit is used to enable or disable the cache. When logic 1 is written in this bit, the cache is enabled and it is used according to the cache algorithm, whereas writing logic 0 disables the cache and there will not be any updates or modifications of the cache entries.

Cache Freeze (CF) When the cache freeze bit is 1, the cache is frozen. The operation of the cache is determined by the CF bit along with the CE bit. Table (7.4) shows the combined effect of CE and CF bits. When the cache is frozen and it is enabled, no modification of the state of the cache is allowed, but the instruction code fetch from the cache is allowed. This function is used to keep frequently used codes stored in the cache and used by the CPU.

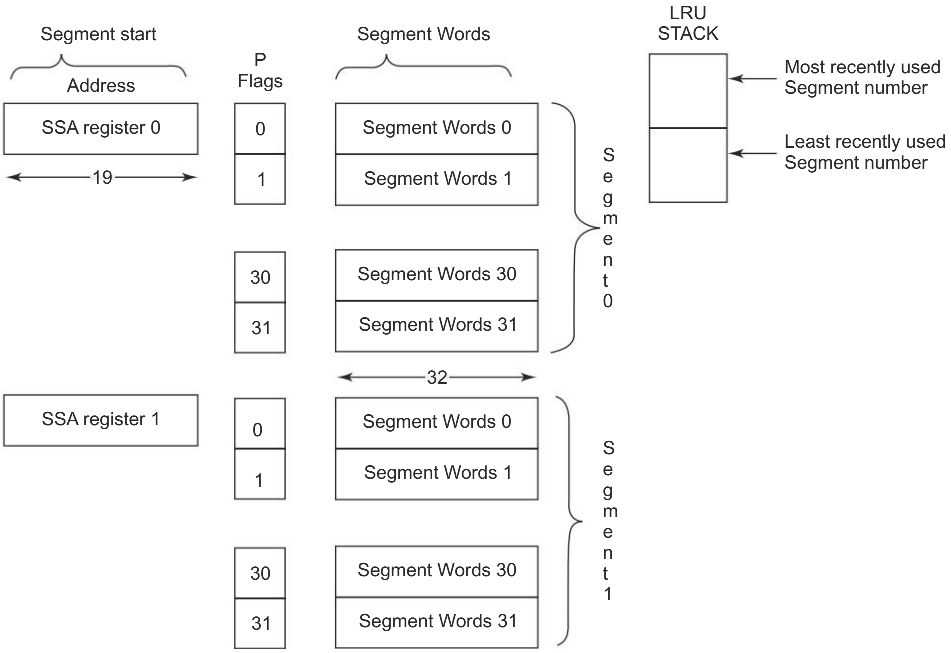


Fig. 7.11 *Program cache architecture*

Table 7.4 *The CE and CF Bit values and its functions*

CE	*CF*	*Function*
0	0	Cache not enabled
0	1	Cache not enabled
1	0	Cache enabled and not frozen
1	1	Cache enabled and frozen

7.7.2 Cache Architecture

The 64 x 32-bit words of cache RAM are divided into two segments. Each segment contains 32 words and each word in the segment is assigned a single flag bit called *P-flag*. There are segment start address (SSA) registers of 19-bit for each segment. The cache architecture is shown in Fig. 7.11. There is a LRU stack, which determines out of the two segments which one is the least recently used segment. Each time the segment is accessed, its segment number (0 or 1) is removed and pushed to the top of the stack. The number at the top is most recently used and number at the bottom of the stack is recently used segment number.

7.7.3 Cache Algorithm

When the CPU requests an instruction word from the external memory, the cache algorithm checks whether the word is already present in the cache. Out of the 24-bit instruction word address, the cache algorithm uses the 19 MSBs of the instruction address to select the segment and 5 LSBs to define the address of the instruction word within the segments. The algorithm compares the 19 MSBs of the instruction address with the two SSA registers.

If neither of the SSA registers match, then the least recently used segment is used for writing the instruction word. All the P flags are cleared in that segment and the SSA register is loaded with the 19 MSBs of the instruction address. The instruction word is fetched from the external memory, copied into the cache and the corresponding P-flag is set. The segment number now accessed is pushed to the top of the stack, thus the other segment number will come to the bottom of the stack.

If there is a match, then the algorithm checks within the segment for the P-flag, because the P-flag present for each segment word indicates whether a word within a particular segment is already preset or not. If the P-flag is set, the instruction word present in the cache is read by the CPU. The segment number now accessed is pushed to the top of the stack, thus the other segment number will come to the bottom of the stack.

If there is no match then the word is read from the external memory and copied into the cache and the corresponding flag bit is set. The segment number now accessed is pushed to the top of the stack, thus the other segment number will come to the bottom of the stack.

PERIPHERALS 7.8

The ′C3X processors peripherals include serial ports, timers and on-chip DMA (Direct Memory Access) controllers. The ′C30 processor has two serial ports and a DMA controller, whereas ‘C31 has only one serial port and ′C32 has one serial port and two DMA controllers. All the three devices have two timers. Figure 7.12 shows the details about the TMS320C3X peripherals. The peripheral bus is used to communicate with the peripherals. The peripheral bus consists of a 24-bit address bus and a 32-bit data bus.

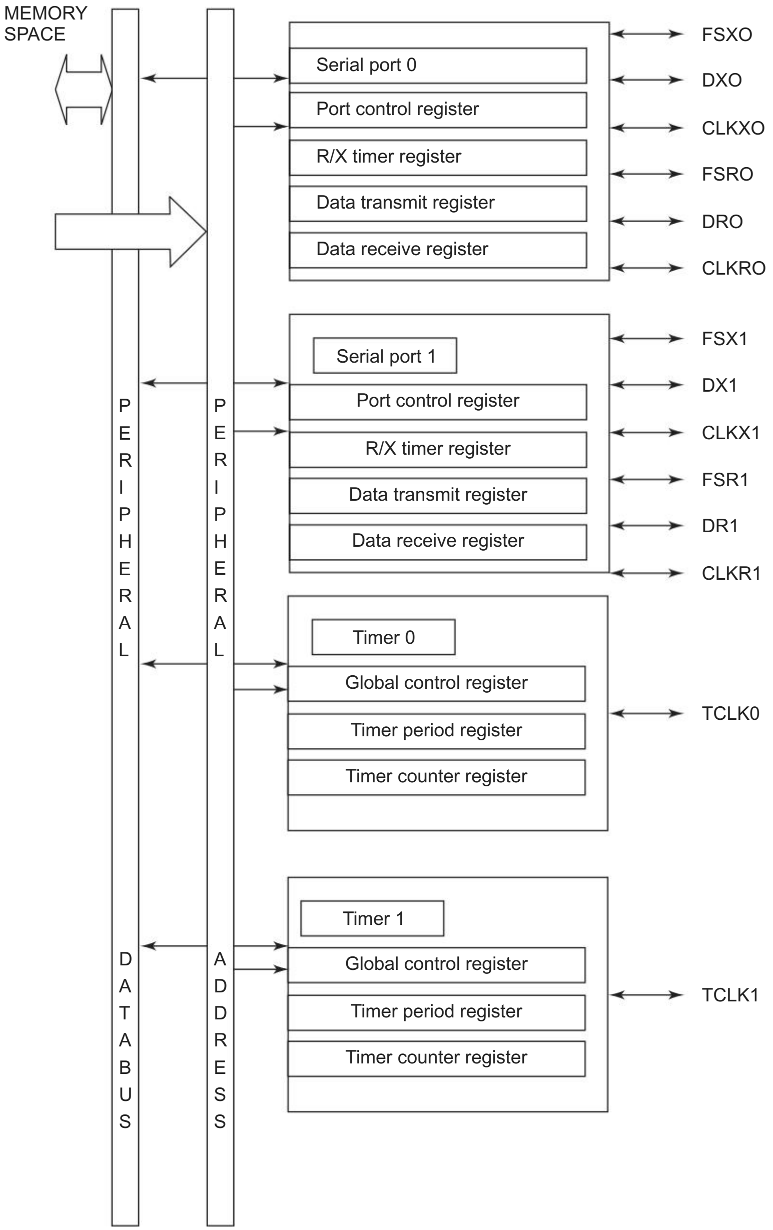


Fig. 7.12 *Block diagram of ′C3X peripherals*

7.8.1 Timers

The ′C3X timers are general-purpose 32-bit timer/event counters. The block diagram of the timer is shown in Fig. 7.13. The timer operates in two signaling modes, internal or external clocking. With internal clock we can use the timer to signal external devices such as A/D converter, D/A converters, etc. When external clock is applied to the timer, it can count the external events and interrupts the CPU after a specified number of events. Each timer has an I/O pin that can be configured as an input, output or general-purpose I/O pin. The three memory-mapped registers, global control register, period register and counter register, are used by the timer. These registers, can be accessed using the memory map address values.

The 32-bit counter present in the timer increments for the rising edge or the falling edge of the input clock. The input clock can be half of the internal clock of'C3X or it can be the external clock signal on TCLKx pin. The counter register holds the value of the counter and its present value is compared with the content of the period register. When the values are equal, the counter is zeroed. This causes the internal interrupt. The pulse generator present generates two types of external clock signals, either pulse or clock.

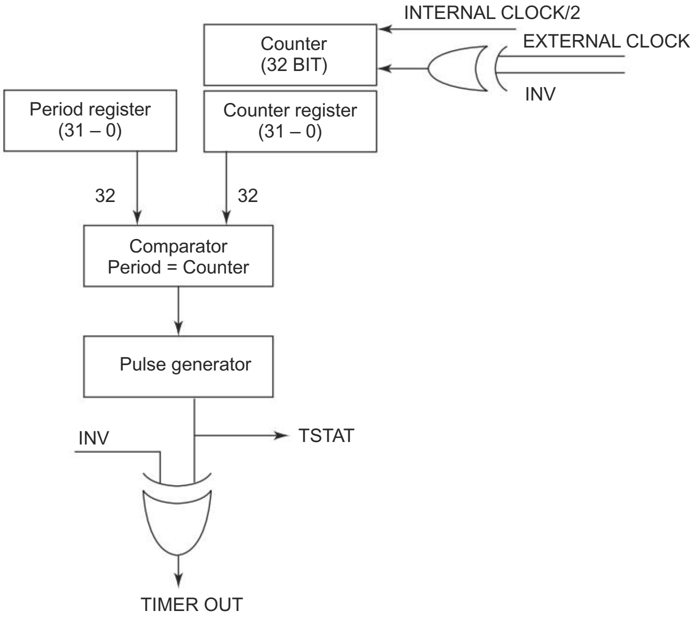


Fig. 7.13 *Timer block diagram*

7.8.2 Serial Ports

The ′C30 and ′32 devices have two serial ports. These serial ports are independent bidirectional serial ports and are identical. The ′C31 device has only one serial port, which is also a bidirectional port. Each serial port has eight control registers, which are memory mapped (refer Section 7.6.3. memory-mapped peripherals), and can be programmed by the user for the desired modes of operation. The block diagram of the serial port is given in Fig. 7.14. The control registers include a global control register, two control registers for the serial I/O pins, three transmit/receive timer registers and one data transmit and receive register. The serial ports can be programmed to transfer 8,16, 24 or 32 bits of data words simultaneously in both directions. The clock signal needed for the serial port can be generated in DSP itself by programming the timer, or it can be externally supplied. The serial port can transmit data words in continuous mode or burst mode.

In the transmit section of the serial port, the data transmit register (DXR) will hold the 32-bit data to be transmitted and it is copied to transmit shift register (XSR), whenever the XSR is empty. Once XSR starts shifting the data bits to the DX pin, the next data word is copied to the DXR register. Once the DXR content is completely copied to XSR register, the transmit ready (XRDY) bit is set in the serial port global control register. This will initiate a serial port transmit interrupt (XINT) and this signal indicates that the DXR is ready to accept new data. The transmit fram synchronisation (FSX) signal will be generated at the beginning of each frame and this initiates the data transfer. All the transmit section operations are synchronised by the transmit clock signal (CLKX).

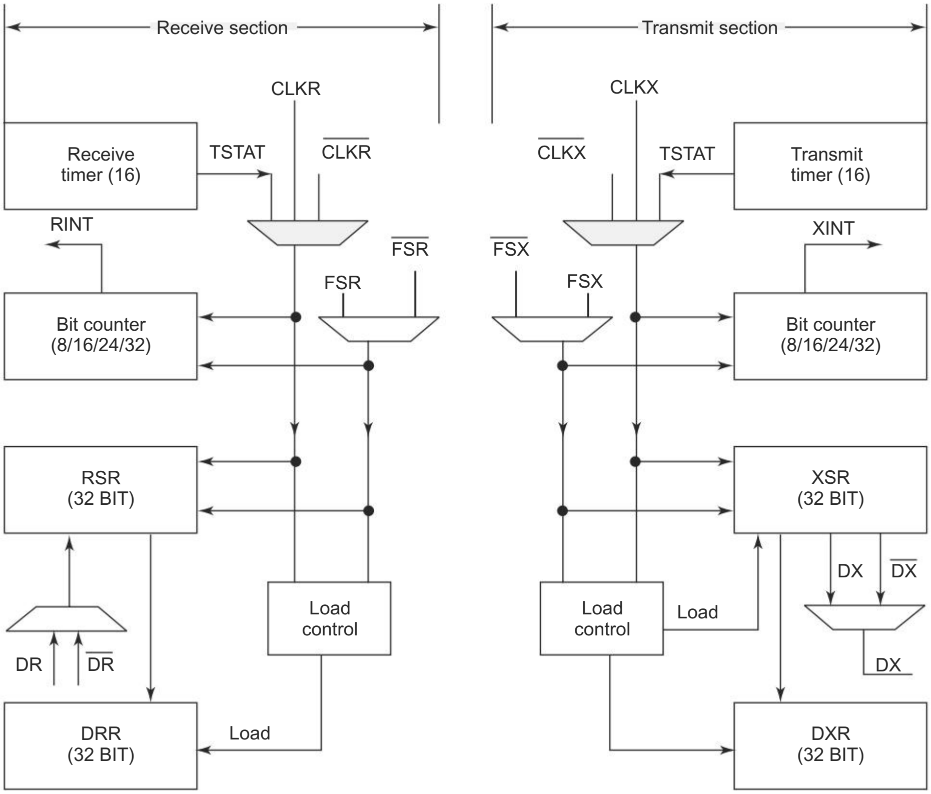


Fig. 7.14 *Serial port block diagram*

At the receive section of the serial port, the data from the DR pin is shifted into receive shift register (RSR). The content of RSR is copied to the data receive register (DRR) and from this register it is read by the CPU. Once the RSR content is completely copied to DRR, the receive ready bit (RRDY) bit in the serial port global control register is set. This initiates the serial port receive interrupt (RINT). Similar to transmit section the receive frame synchronisation (FSR) of the receiver section initiates the receive operation and receiver clock signal CLKR is used to synchronise all the receiver section actions.

7.8.3 DMA Controller

The on-chip DMA controller present in C3X devices can be used to read or write the 32-bit operands in any locations in the memory map. The ′C30 and ′C31 devices have only one DMA controller, whereas ′C32 has two DMA controllers. The DMA operations never interfere with the operations of the CPU and this enables to interface slow external devices such as memory, peripherals, etc., without reducing the throughput to the CPU. The DMA controller consists of address generators, source and destination registers and transfer counter. The block diagram of DMA controller is shown in Fig. 7.15. The DMA controller has its own address bus and data bus and this minimise the conflict with CPU.

The CPU and DMA controller busses can function independently, but when they access the same on-chip or the external memory location the priority is provided. As far as ′C30 and ′C31 devices are concerned, the highest priority is for the CPU access, but in ′C32 devices the user can configure the priorities.

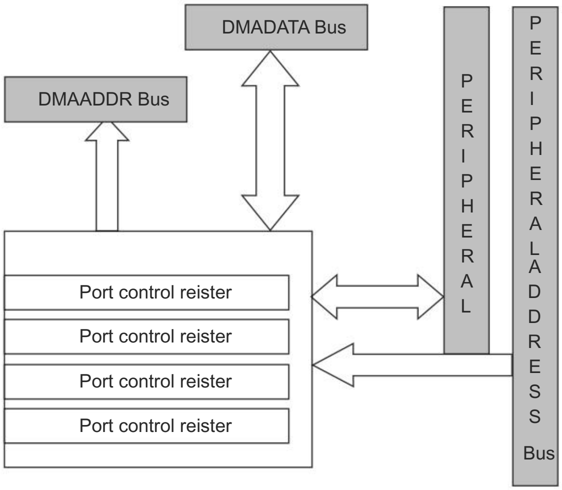


Fig. 7.15 *The block diagram of DMA controller*

Review Questions

7.1 List the processors available in the ‘C3X family.

7.2 What are all the various units present in ‘C3X CPU?

7.3 List the data buses and address buses in the ‘C3X processor.

7.4 What are all the various registers in the ‘C3X register file?

7.5 What is the use of data page pointer?

7.6 What is the use of auxiliary registers?

7.7 List the various flags available in the ‘C3X status register.

7.8 What is the use of interrupt enable and interrupt flag registers?

7.9 What are the various registers used for the repeat operations?

7.10 What are the differences between microprocessor and microcomputer mode of operation of ′C3X processor?

7.11 What is the use of cache memory?

7.12 What are all the bits used for controlling the cache operation?

7.13 What are all the on-chip peripherals available in ′C3X processor?

7.14 What are all the registers to be programmed for activating the serial port?

7.15 What are the memory map registers available for controlling the on-chip timer operation?

Self Test Questions

7.1 The ′C3X family processors are ——— bit floating-point processors.
(a) 8 (b) 16 (c) 24 (d) 32

7.2 The ′C3X family processor suitable for low power application is ———.
(a) ′C30 (b) ′C31 (c) XC31 (d) ′C32

7.3 The on-chip RAM available in the ′C3X family processors is ———.
(a) 2K (b) 4K (c) 8K (d) 64K

7.4 The ′C3X family processors are different in ———.
(a) architecture (b) memory size
(c) speed of operation

7.5 The ′C3X family processors have ———.
(a) onlySARAM (b) onlyDARAM
(c) both SARAM and DARAM

7.6 The no. of address buses present in the ‘C3X family processors are ———

(a) 5 (b) 6 (c) 3 (d) 2

7.7 The no. of data buses present in the ‘C3X family processors are ———

(a) 8 (b) 7 (c) 9 (d) 4

7.8 The multiplier in the ’C3X family processors can perform ——— multiplications.

(a) 24-bit integer
(b) 32-bit floating point
(c) both 24-bit integer and 32-bit floating-point

7.9 The size of the ’C3X processor ALU is ———.

(a) 24 bits (b) 16 bits (c) 32 bits (d) 40 bits

7.10 The no. of ARAUs in the ’C3X processors are ———.

(a) 2 (b) 3 (c) 1 (d) 4

7.11 The no. of index registers in the ’C3X processors are ———.

(a) 3 (b) 1 (c) 2 (d) 4

7.12 The ’C3X processor register file contains ——— registers.

(a) 12 (b) 24 (c) 28 (d) 34

7.13 The no. of data pages and location in each page of data memory of the ’C3X processor are

(a) 512, 128K (b) 256, 64K
(c) 256, 32K (d)512,8K

7.14 The no. of bits of the DP used for direct addressing mode are ———.

(a) LSB9bits (b) LSB 8 bits
(c)LSB12bits (d) all bits of DP

7.15 The no. of ARs used for indirect addressing mode of the ’C3X processor are ———.

(a) 6 (b) 8 (c) 4 (d) 2

7.16 The no. of flag bits present in the status (ST) register of the ’C3X processor are ———.

(a) 4 (b) 6 (c) 9 (d) 10

7.17 Which bit present in the ST register of the ’C3X processor will globally disable and enable the interrupts?

(a) GIE (b) C (c) Z (d) OV

7.18 The purpose of I/O flag register is to

(a) to control the function of XFO and XF1 pins
(b) to control the interrupt operation
(c) to control the read and write operations

7.19 The no. of registers used for the repeat instruction of ’C3X processor are ———.

(a) 2 (b) 3 (c) 4 (d) 1

7.20 The ’C3X processor which is having on-chip ROM is ———.

(a) ’C30 (b) ’C31 (c) LC31 (d) ’C32

7.21 The cache size of’C3X processor is ———.

(a) 64K (b) 64 words (c) 4K (d) 2K

7.22 The address buses of the ’C3X processor are of ——— width.

(a) 32 bits (b) 24 bits (c) 16 bits (d) 40 bits

7.23 The on-chip RAM memory address location of the ’C3X processor is ———.

(a) 808000h-80FFFFh (b) 809800h-809FFFh
(c) 800000h-80FFFFh

7.24 The segment start address register of the ‘C3X processor cache is of ——— size.

(a) 19 bits (b) 20 bits (c) 24 bits (d) 32 bits

7.25 The cache available in the ‘C3X processor is for ———.

(a) program memory
(b) data memory
(c) hpth program and data memory

7.26 The no. of serial ports in ‘C31 and ‘C32 processors are ———.

(a) 1 (b) 2 (c) 3 (d) 0

7.27 The no. of registers which control the timer operation is ———.

(a) 2 (b) 3 (c) 4 (d) 1

7.28 The size of the ’C3X processor timer is ———.

(a) 32 bit (b) 16 bit (c) 24 bit (d) 40 bit

7.29 The no. of control registers for each serial port of the ’C3X processor are ———.

(a) 4 (b) 5 (c) 7 (d) 8

7.30 The no. of DMA controllers in ‘C32 processor are ———.

(a) 1 (b) 2 (c) 3 (d) 4

8 ADDRESSING MODES AND LANGUAGE INSTRUCTIONS OF ′C3X

The TMS320C3X processors support four groups of addressing modes. There are six types of addressing modes which can be used within the groups. These addressing modes allow the access of instruction word and access of data from memory and registers. The ′C3X processors instructions set contains 113 instructions, which are organised into six groups. These assembly language instructions set supports numeric-intensive, signal processing and general purpose applications. In this chapter details of various addressing modes and instructions are discussed with examples.

DATA FORMATS 8.1

The TMS320C3X processors support both integer and floating-point data formats. As far as the integer data format is concerned both the signed and unsigned integer can be used. The floating-point operations make fast, trouble-free, accurate and precise computations. In floating-point data format short floating-point, single-precision and double-precision formats are used.

8.1.1 Integer Formats

The signed and unsigned integer formats have a 16-bit short integer format and a 32-bit single-precision integer format. In short integer format only the least significant 16 bits are used to represent the operands, whereas in single-precision integer format all the 32 bits are used.

Short Integer Format The short integer format is a 16-bit two's complement integer format for immediate integer operands. In the instructions in which this format is used, the least significant 15-bits are used to represent the operand and the most significant bit (MSB) is used as sign bit, whereas in the unsigned short integer format all the 16 bits are used to represent the operands.

In signed short integer format, the most significant 16 bits (31-16) can be used as sign bits and this is called *sign extension of a short integer.* In the case of unsigned short integer the most significant 16-bits are zero filled and it is called *zero fill of an unsigned short integer.* The range of integers that can be represented in short integer format is $-2^{15} \le X \le 2^{15} - 1$.

Single-precision Integer Format In the single-precision integer format, the integer is represented in two's complement format. In this format the MSB (31st bit) is used as sign bit and the rest of the 31 bits (30-0) are used to represent the operand. In the case of unsigned format, all the 32 bits are used to represent the operands. The range of integers that can be represented in short integer format is $-2^{31} \leq X \leq 2^{31} - 1$.

8.1.2 Floating-point Formats

The floating-point data format consists of three fields. They are exponent field (*e*), single sign bit field (*s*) and a fraction field (*f*). The combined sign field and fraction field is referred as *mantissa field.* The exponent field is a two's complement number and mantissa part is a two's complement fraction combined with the sign bit and the implied most significant bit. The floating-point format is shown in Fig. 8.1.

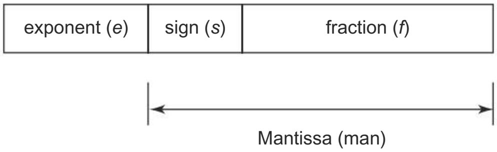


Fig. 8.1 *Floating-point format*

The mantissa represents a normalised 2s-complement number. In a normalised representation, a most significant nonsign bit is implied, thus providing an additional bit of precision. The implied sign bit is used as follows:

- $s = 0$, then the leading two bits of the mantissa are 01
- $s = 1$, then the leading two bits of the mantissa are 10

If the sign bit '*x*' is equal to 0, the mantissa becomes 01.f_2, whereas if '*s*' is 1, the mantissa becomes 10. f_2 , where *f* is the binary representation of the fraction field.

The floating point two's complement number *X* as a function of the fields *e, s* and *f* is given below.

$X = 01.f \times 2e$ if $s = 0$ or if the leading 0 is the sign bit and 1 is the implied most significant bit

$= 10.f \times 2e$ if $s = 1$ or if the leading 1 is the sign bit and
0 is the implied most significant nonsign bit

$= 0$ if e = most negative two's complement value
of the specified exponent field width

As far as the floating-point format is concerned, there are three formats, short floating-point format, single-precision floating-point format and extended precision floating-format, present.

Short Floating-Point Format In the short floating-point format, 16 bits are used to represent the operand. The floating-point numbers are represented with a 4-bit exponent field and a 12-bit mantissa with an implied most significant nonsign bit.

Single Precision Floating-Point Format In the single precision floating-point format, an 8-bit exponent field and a 24-bit mantissa with an implied most significant nonsign bit is used to represent the operands.

Extended Precision Floating-Point Format In the extended precision floating-point format the operands are represented with an 8-bit exponent field and a 32-bit mantissa with an implied significant nonsign bit.

The maximum positive, negative, minimum positive and negative numbers that can be represented using the three floating-point formats are given in Table 8.1.

Table 8.1 *Most, least positive and negative numbers in floating-point format*

	Short floating-point format	*Single precision floating-point format*	*Extended precision floating-point format*
Most, positive	$(2–2^{–11}) \times 2^7 =$ 2.5594×10^2	$(2–2^{–23}) \times 2^{127} =$ 3.4028234×10^{38}	$(2–2^{–23}) \times 2^{127} =$ 3.4028234×10^{38}
Least, positive	$(1 \times 2^{–7}) =$ $7.18125 \times 10^{–3}$	$(1 \times 2^{–127}) =$ $5.8774717 \times 10^{–39}$	$(1 \times 2^{–127}) =$ $5.8774717541 \times 10^{–39}$
Least, negative	$(–1–2^{–11}) \times 2^{–7} =$ $–7.8163 \times 10^3$	$(–1–2^{–23}) \times 2^{–127} =$ $–5.8774724 \times 10^{–39}$	$(–1–2^{–23}) \times 2^{–127} =$ $–5.8774717569 \times 10^{–39}$
Most, negative	$(–2 \times 2^7) =$ $–2.5600 \times 10^2$	$(–2 \times 2^{127}) =$ $–3.4028236 \times 10^{38}$	$(–2 \times 2^{127}) =$ $–3.4028236691 \times 10^{38}$

ADDRESSING MODES 8.2

The TMS320C3X processor supports the following six addressing modes:

(1) Register addressing mode, (2) Direct addressing mode, (3) Indirect addressing mode, (4) Short-immediate addressing mode, (5) Long-immediate addressing mode and (6) PC-relative addressing mode. In this section, the various addressing modes are explained with examples.

8.2.1 Register Addressing

The TMS320C3X processor register file contains eight extended precision registers (R0-R7). These CPU registers contain the operand. The syntax of this addressing mode is

mnemonic src, dst

The *mnemonic* can be any assembly instruction code that support register addressing mode, *src* is the source register and *dst* is the destination register. The registers R0-R7 can be used both for source and destination registers.

Example 8.1 ADD1 R3,R5—This instruction adds the two hexadecimal integer operands present in the registers R3 and R5. The result is stored in the register R5. The content of register R3 is unchanged.

	Before execution		After execution
R3	11223344	R3	11223344
R5	22334455	R5	33557799

Example 8.2 LDF R4,R5—This instruction loads the floating point operand present in the source register R4 to the destination register R5. The content of register R4 is unchanged

	Before execution		After execution
R4	1.23425354e + 01	R4	1.23425354e + 01
R5	2.34353674e + 01	R5	1.23425354e + 01

8.2.2 Direct Addressing

The number of address bits of TMS320C3X processor is 32 bits, and the total addressable data space with 32 bits is 16 M words. The data space of the ′C3X processor is organised in such a way that, there are 256 pages and each page contains 64K words. The data page pointer (DP), a separate 32-bit register holds the value of the data page. The location of the operand in a specific page is given in the instruction code. The data page pointer is to be loaded first before the access of the data from the page using direct addressing.

In direct addressing mode, the least significant 16-bits of the instruction code and the least significant 8-bits of DP are combined to form the 24-bit address of the data operand present in the data space. Figure 8.2 shows the data address formation in direct addressing mode. The syntax for the direct addressing mode is

mnemonic @expr, dst

and the *expr* is the LSB 16-bit value of the operand in a particular page.

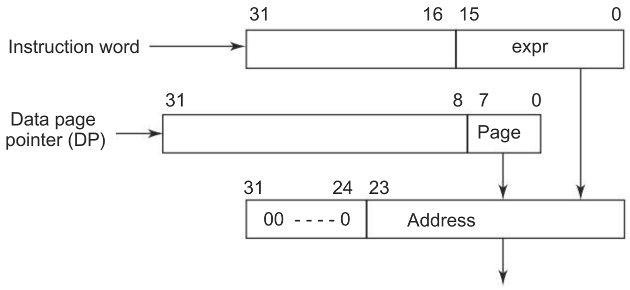


Fig. 8.2 *24-Bit data address formation*

Example 8.3 LDF @1000h,R4—This instruction loads the floating point operand present in the location 1000h of the page 128 (Take content of DP=80h) into the register R4. The address of the operand is 00801000h

	Before execution		After execution
DP	00000080		00000080

Data in the location 00801000h

	2.34353674e + 01		2.34353674e + 01
R4	1.23425354e + 01	R4	2.34353674e + 01

Example 8.4 ADDI @2010h,R5—This instruction adds the integer operand in the data location 602010h(Take content of DP=60h) and the content in the register R5, the result is stored in register R5.

	Before execution		After execution
DP	00000060		00000060

Data in the location 00602010h

	11223344		11223344
R5	23459872	R5	3467CBB6

8.2.3 Indirect Addressing

In indirect addressing mode the address of an operand in memory is specified through the content of an auxiliary register. There are eight auxiliary registers (AR0-AR7) in ′C3X processors, any one of which can be used to specify the address. The auxiliary register is of 32 bits, the LSB 24 bits are used to specify the address; rest of the bits are not modified by the instructions, which load ARs. The two auxiliary register arithmetic units (ARAUs) present are used to perform the arithmetic operations needed for the address displacement. The instruction code format of the indirect addressing mode is given in Fig. 8.3. It consists of three fields, *MOD, ARn* and *disp* fields.

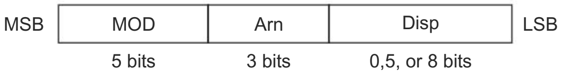


Fig. 8.3 *Indirect addressing instruction code format*

The 5-bit MOD field specifies the type of indirect addressing, the 3-bit ARn field specifies the content of auxiliary register that can be used as the operand address and the disp (displacement) field is used to modify the address value after the current memory access. The displacement of the address is either an explicit unsigned 8-bit integer contained in the instruction word or an implicit displacement of one. The displacement can also be provided by the two index registers IR0 and IR1. The circular and bit-reversed addressing mode is also possible. Tables 8.2, 8.3 and 8.4 lists the indirect addressing MOD field value, its syntax and its operation.

Table 8.2 *Indirect addressing with displacement*

MOD Field	*Syntax*	*Operation*
00000	*+ARn(disp)	Address = Content of ARn + the displacement
00001	*–ARn(disp)	Address = Content of ARn - the displacement
00010	*++ARn(disp)	Address = Content of ARn + the displacement
		New ARn content = Old Content of ARn + the displacement
00011	*––ARn(disp)	Address = Content of ARn - the displacement
		New ARn content = Old Content of ARn - the displacement
00100	*ARn++(disp)	Address = Content of ARn
		New ARn content = Old Content of ARn + the displacement
00101	*ARn––(disp)	Address = Content of ARn
		New ARn content = Old Content of ARn - the displacement
00110	*ARn++(disp)%	Address = Content of ARn
		New ARn content = circular addressing of (old content of ARn + displacement)
00111	*ARn––(disp)%	Address = Content of ARn
		New ARn content = circular addressing of (old content of ARn - displacement)

In indirect addressing with index registers, either the content of index register 0 or index register 1 can be used for displacement. For the MOD fields 01000 – 01111 the content of index register 0 is used and from 10000 – 10111 the content of index register 1 is used. In the following table the syntax is given only for IR0, the same way all the indirect addressing mode is possible with IR1 also.

Table 8.3 *Indirect addressing with index register IR0/IR1*

MOD Field	*Syntax*	*Operation*
01000	*+ARn(IR0)	Address = Content of ARn + the content of index register IR0
01001	*–ARn(IR0)	Address = Content of ARn - the content of index register IR0
01010	*++ARn(IR0)	Address = Content of ARn + the content of index register IR0 New ARn content = Old Content of ARn + the content of index register IR0
01011	*––ARn(IR0)	Address = Content of ARn - the content of index register IR0 New ARn content = Old Content of ARn - the content of index register IR0
01100	*ARn++(IR0)	Address = Content of ARn New ARn content = Old Content of ARn + the content of index register IR0
01101	*ARn––(IR0)	Address = Content of ARn New ARn content = Old Content of ARn - the content of index register IR0
01110	*ARn++(IR0)%	Address = Content of ARn New ARn content = Circular addressing of (Old Content of ARn + the content of index register IR0)
01111	*ARn––(IR0)%	Address = Content of ARn New ARn content = Circular addressing of (Old Content of ARn - the content of index register IR0)

Table 8.4 *Indirect Addressing (special cases)*

MOD Field	*Syntax*	*Operation*
11000	*ARn	Address = content of ARn
11001	*ARn++(IR0)B	Address = content of ARn New ARn content = Old content of ARn + Bit reversed content of index register IR0

Example 8.5 STI R3,*+AR2(3)—This is a store instruction, the integer operand present in register R3 is stored in the data memory location. The address of the data memory location is the sum of the content of AR2 and the displacement (disp = 3). Take the content of AR2 is 00803010h then the address of the operand is 00803013h

	Before execution		After execution
AR2	00803010		00803010
R3	23459872	R3	23459872

Content of the location 00803013h

	Before execution		After execution
	11223344		23459872

Example 8.6 STF R4,*--AR5(5)—The floating point operand present in the register R4 is stored in data memory. The data memory address is the subtraction of displacement (disp=5) from the content of AR5. After execution, the new content of AR5 is the subtraction of displacement from the old content AR5. Take the content of AR5 as 00602020h, then the memory address is 006020 lBh.

	Before execution		After execution
AR5	00602020	AR5	0060201B
R4	2.34353674e + 01	R4	2.34353674e + 01
Content of the location 0060201Bh			
	1.23425354e + 01		2.34353674e + 01

Example 8.7 SUBB *AR2++(4),R6—The integer operand present in the data memory location, pointed by AR2 is subtracted from the content of register R6 and the result is stored in register R6, The data memory address before execution is the content of AR2, the new content of AR2 is the sum of displacement and the old content AR2. Take the content of AR2 as 00701010h.

	Before execution		After execution
AR2	00701010	AR2	00701014
R6	23459872	R6	23459872
Content of the location 00701010h			
	11223344		1223652E

Example 8.8 ADDF *AR6++(IR1),R0—The floating point operand present in the data memory address pointed by AR6 is added with the content of the register R0 and the result is stored in R0. The new content in AR6 is the sum of the old content in AR6 and the content of IR1.

	Before execution		After execution
AR6	00800000	AR6	00801000
IR1	00001000		00001000
Content of the location 0080000h			
	1.23425354e + 01		1.23425354e + 01
R0	2.34353674e + 02	R0	2.46696209e + 02

8.2.4 Immediate Addressing

In immediate addressing mode the operand can be given directly in the instruction. There are two types of immediate addressing modes,

1. Short-immediate addressing and
2. Long-immediate addressing

The syntax of the immediate addressing mode is;

mnemonic expr

The field *expr* can be a 16-bit or 24-bit operand value. In short immediate addressing mode the operand is a 16-bit immediate value contained in the 16 LSBs of the instruction word (expr). The short-immediate operand can be two's complement integer, an unsigned integer, or a floating-point number.

In long-immediate addressing mode the operand is a 24-bit immediate value contained in the 24 LSBs of the instruction word (expr).

Example 8.9 LDI 1000h,R7—The immediate operand 1000h is loaded in to register R7

	Before execution		After execution
R7	00000200	R7	00001000

8.2.5 PC-Relative Addressing

The Program counter (PC) - relative addressing is used for branching. It loads the 16-bit or 24-bit LSBs of the instruction word to the PC register. The syntax of the addressing is;

mnemonic src

The *mnemonic* can be branch, call and repeat instructions codes and *src* is a label or address.

Example 8.10 BR 8000h—When this instruction is in execute phase the address 8000h is loaded in to PC.

	Before execution		After execution
PC	00080200	PC	00008000

8.2.6 Circular Addressing

The digital signal processing algorithms which use convolution and correlation requires the circular addressing. In this section the circular addressing mode is described.

The block size register (BK) specifies the size of the circular buffer(R). The starting address (Top of buffer) and end address (End of buffer) of the circular buffers are computed immediately from buffer length. The index register is used as increment or decrement counter. Figure 8.4 illustrates the relationship between the various registers used in circular addressing.

The length of the block size is loaded in register BK. The address of the top of the buffer is found by filling K LSB bits zero, where K is an integer that satisfies the condition $2^K \geq R$ and the K+l to 31 bits are concatenated from ARn. The address of the bottom of the buffer is found by filling K LSBs of the BK register as such and K+1 to 31 bits of ARn. The index register is loaded with the K LSB bits of ARn.

- The first time the circular buffer is used, the auxiliary register must be pointing to an element in the circular queue.
- The step used must be less than or equal to the block size and it is treated as unsigned integer.

The algorithm for circular addressing is given below

```
if 0 ≤      index +step < BK;
            index = index + step.
Else if     index + step ≥ BK
            index = index + step – BK.
Else if     index + step < 0;
            index = index + step + BK
```

Example 8.11 Assume that the size of the circular buffer is 20, the auxiliary register used is AR3 and its content is 00601000. The length of the buffer is entered in the block size register (BK) and it is 00000014h. The value of *K* bits needed for finding the address of the top and bottom of the buffer is 5 ($2^5 > 20$).

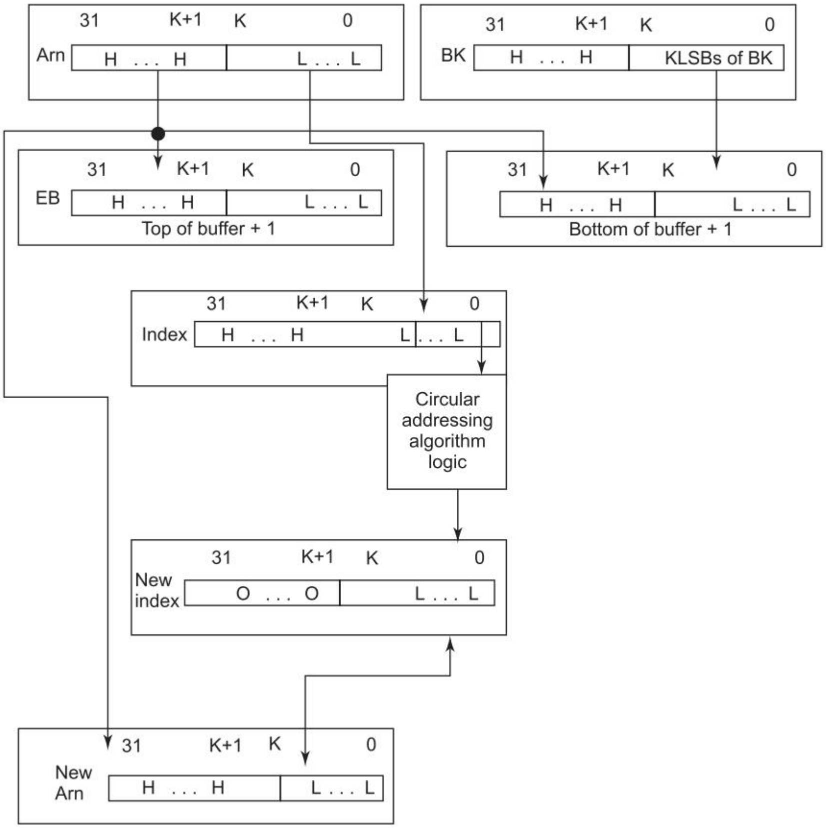


Fig. 8.4 *Address computation in circular addressing*

The top of the buffer is found by filling *K* LSB bits zero and taking K+1 to 31 bits from AR3. The address of the top of the buffer is 00601000h. The bottom of the buffer is found by filling K LSB bits of BK register and *K*+1 to 31 bits from AR3. The address of the bottom of the buffer is 00601014h. The index register IR0 is used as counter. The following instruction code illustrates the circular buffer addressing. The content of AR3 before execution and after execution is given.

		Content of AR3	
		Before execution	After execution
LDI	*AR3++(5), R0	00601000	00601005
LDI	* AR3++(5), R1	00601005	0060100A
ADDI	*AR3++(5), R0	0060100A	0060100F
ADDI	*AR3++(5), R1	0060100F	00601014
SUBB	*AR3++(2), R2	00601014	00601002
SUBB	*AR3 - -(4), R3	00601002	00601012

8.2.7 Bit-Reversed Addressing

Bit-reversed addressing is used for the implementation of FFT algorithms. The base address of bit-reversed addressing must be located on a boundary of the size of the table. For example, if IR0 = 2^{n-1}, the n LSBs of the base address must be 0. The base address of the data in memory must be on a 2^n boundary. One auxiliary register points to the physical location of a data value. IR0 specifies one-half the size of the FFT, that is, the value contained in IR0 must be equal to 2^{n-1}, where n is an integer and the FFT size is 2^n. When you add IR to the auxiliary register by using bit-reversed addressing, addresses are generated in a bit-reversed fashion.

Example 8.12 Consider that AR4 is used for bit-reversed addressing and its address value is 00700100h. The index register IR0 is used, and its values for the following instructions are 0,2,4,8 respectively.

		Content of AR4	
		Before execution	After execution
LDI	*AR4++(IR0)B, R0	00701000	00701000
LDI	*AR4++(IR0)B, R0	00701000	00701004
LDI	*AR4++(IR0)B, R0	00701000	00701002
LDI	*AR4++(IR0)B, R0	00701000	00701001

GROUPS OF ADDRESSING MODES 8.3

There are six types of addressing modes. Since some addressing modes are not appropriate for some instructions, the types of addressing modes are used in four groups as follows 1. General addressing modes (G); 2. Three operand addressing modes (T); 3. Parallel addressing modes (P); 4. Conditional branch addressing modes. This section explains about the groups of addressing modes.

1. General Addressing Modes The general addressing mode includes register addressing, direct addressing, indirect addressing and immediate addressing modes. The general addressing mode uses the general-purpose instructions. The bits 31-29 set 000 indicates the general addressing mode. The bits 22 and 21 of the instruction word specify the general addressing mode.

0 0 - register addressing
0 1 - direct addressing
1 0 - indirect addressing
1 1 - immediate addressing

2. Three-operand Addressing Modes In three-operand addressing modes two source operands and one destination operand are used. The instructions which use three operand addressing mode have the form ADDI3, LSH3, CMPF3, XOR3, etc. The three-operand addressing modes include register and indirect addressing modes. The bits 31-29 set to 001 indicates the three-operand addressing mode. The bits 22 and 21 of the instruction word specify the type of addressing mode to be used for two source operands

	SRC1	SRC2
0 0 -	register	register
0 1 -	indirect	register

1 0 - register indirect
1 1 - indirect indirect

3. Parallel Addressing Modes Some of the ′C3X instructions can occur in pair and those instructions will be executed in parallel. The parallel instructions are indicated with two vertical bars (||) and this includes register and indirect addressing modes. For parallel addressing mode four operands are needed. The instruction code format of the parallel addressing mode is shown in Fig. 8.5 and its field description is given in Table 8.5

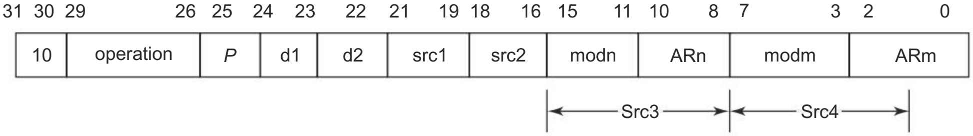


Fig. 8.5 *The instruction code format of parallel addressing mode*

Table 8.5 *Description of parallel addressing mode instruction code format*

Bits	*Description*
31 &30	The value set is 10, this indicates the parallel addressing mode
29–26	These four bits, indicates the type of operation to be performed
25&24	Parallel addressing mode field (P), this specifies how the bits 21-0 are used for addressing the source operands 23&22 These two bits specify the destination of the operands
	d1 If 0, dst is R0. If 1, dst is R1
	d2 If 0, dst is R2. If 1, dst is R3
21–19	This field specifies the source operand 1 (src1) address
18–16	This field specifies the source operand 2 (src2) address
15–8	This fields specifies the source operand 3 (src3) address
	Bits 15-11 (modn) specifies the type of indirect addressing used for src3 and
	Bits 10-8 (ARn) specifies the auxiliary register used for pointing the address of src3
7–0	This fields specifies the source operand 4 (src4) address
	Bits 7-3 (modm) specifies the type of indirect addressing used for src4 and
	Bits 2-0 (ARm) specifies the auxiliary register used for pointing the address of src4

4. Conditional-Branch Addressing Modes The ′C3X processors instruction set includes conditional and branch addressing mode instructions (Bcond, BcondD, DBcond, DBcondD and CBLLcond). These instructions can perform variety of conditional operations. The conditional-branch addressing mode includes only the register and PC-relative addressing modes. The instruction code format of the conditional-branch instruction is shown in Fig. 8.6 and its field description is given in Table 8.6.

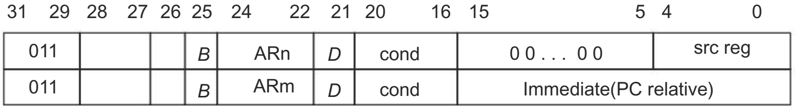


Fig. 8.6 *The instruction code format of parallel addressing mode*

Table 8.6 *Description of parallel addressing mode instruction code format*

Bits	*Description*
31–29	These bits set 011 specify conditional-branch addressing mode
28&27	01 – specifies branch instructions
	10 – specifies call instructions
26	1 – specifies the DBcond branch instruction
	0 – specifies the Bcond branch instruction
25	This bit specifies the addressing mode
	0 – specifies the register addressing mode
	1 – specifies the PC-relative addressing mode
24–22	This field specifies the auxiliary register used
21	This field specifies the branch is a standard branch or a delayed branch
	0 – specifies standard branch
	1 – specifies the delayed branch
20–16	Specifies the condition used in the branch and call instructions
15–0	Specifies the new PC value either in register addressing or in PC-relative addressing

ASSEMBLY LANGUAGE INSTRUCTIONS 8.4

The TMS320C3X assembly instructions set supports numeric-intensive, signal processing and general-purpose applications. There are 113 instructions organised in six groups. All instructions sets are one word and most of them require one cycle to execute. In the instructions set some instructions support floating-point and some use fixed-point operands. The six functional groups of addressing modes are as follows.

- Load and store
- Two-operand arithmetic/logical
- Three-operand arithmetic/logical
- Program control
- Interlocked operations
- Parallel operations

In this section the groups of addressing modes are discussed and some important instructions sets are explained with examples. The general syntax of assembly instructions is as follows.

1. Three operand instructions

   mnemonic src2,src1,dst

The source operands *src2* and *src1* can be accessed by the register and indirect addressing modes, whereas the destination operand *dst* should be accessed only by register addressing mode.

2. Other instructions

   mnemonic src,dst

The source operand *src* can be accessed by general addressing mode (G) and the destination operand *dst* should be accessed only by register addressing mode. But for store instructions the source and destination operand addressing modes are reverse.

Load and Store Instructions The ′C3X processors support 13 load and store instructions. Using these instructions a word can be loaded from memory into a register, stored from register into memory. Certain instructions are used to manipulate data on the system stack. Load instructions can also be conditional instructions.

Two-Operand Instructions There are 35 two-operand arithmetic and logical instructions. Out of this two-operands one is source and another is destination. The source operand can be a memory word, a register, or a part of the instruction word, where as the destination operand is always a register.

Three-Operand Instructions Some arithmetic and logical instructions have three operands. There are 17 such three operand instructions, which allow the processor to read two source operands from memory or from the CPU register file in a single cycle and store the results in destination, which is always a register.

Program-Control Instructions The program-control instruction group consists of repeat instructions, both standard and delayed branch, call and return instructions. There are 17 such instructions, which affect the program flow. Several program-control instructions support conditional operations.

Interlocked-Operation Instructions There are 5 instructions which support interlocked operations. These instructions are used for multiprocessor communication through the external signals (XF0 and XF1) to allow for powerful synchronisation mechanisms. The source address is accessed only through direct and indirect addressing modes and the destination address is accessed only through register addressing.

Parallel-Operations Instructions Some TMS320C3X instructions can occur in pairs that will be executed in parallel. These instructions can perform parallel loading of registers, parallel arithmetic operation, or arithmetic/ logical operations in parallel with a store instruction. Each instruction in a pair is entered as a separate source statement. The second instruction in the pair must be written followed by two vertical bars (||).

8.4.1 Load Instructions

The assembly instructions and description of the various load instructions are listed in Table 8.6. The load instructions can load integer and floating-point values in various addressing modes. It is also possible to load separately the floating-point exponent and mantissa values. The load operation can be performed both in interlocked and parallel operation modes.

Table 8.6

Instruction	*Description*
LDP	Load data page pointer
LDI	Load integer
LDF	Load floating point value
LDM	Load floating point mantissa
LDE	Load floating point exponent
LDII	Load integer, interlocked
LDFI	Load floating point value, interlocked
LDF|| LDF	Load floating point, parallel
LDI || LDI	Load integer value, parallel

Example 8.13 LDP @601000h,DP - This is data page pointer load instruction. The address of the source operand given in the instruction is 24-bits. The MSB 8 bits of the 24-bit source address is loaded into the 8 LSBs of the data page pointer (DP)

	Before execution		After execution
DP	00000010		00000060

Example 8.14 LDI *AR4++(2),R1—The integer operand present in the address location pointed by AR4 is loaded into register R1. The new address content of AR4 is the sum of the old content of AR4 and the displacement (2).

	Before execution		After execution
AR4	00800000	AR4	00800002

Content of the location 00800000h

	23459872		23459872
R1	11223344	R1	23459872

Example 8.15 LDFI *+AR4(1),R5—An interlocked operation is signaled over XF0 and XF1 pins. The floating-point data operand content is loaded into register R5. The address of the data operand is the sum of the content of AR4 and the displacement (1).

	Before execution		After execution
AR4	00602020	AR4	00602020

Content of the location 00602021h

	1.23425354e + 01		1.23425354e + 01
R5	2.34353674e + 02	R5	1.23425354e + 01

Example 8.16 LDI * AR2++(5),R5 | | LDI *-AR4(1),R2—In this instruction two integer operands are loaded into registers in parallel. The. address of the first operand is the content of AR2, the address of the second operand is the subtraction of the displacement (1) from the content of AR4. After execution the new address content in AR2 is the sum of the old content in AR2 and the displacement (5).

	Before execution		After execution
AR2	00602020	AR2	00602025
AR4	00700001	AR4	00700001

Content of the location 00602020h

	23456789		23456789

Content of the location 00700000h

	22446688		22446688
R5	00000000	R5	23456789
R2	22343547	R2	22446688

8.4.2 Store Instructions

The various kinds of store instructions present in ′C3X processor are listed in Table 8.7. The store instructions can store the operands in various addressing modes. The operands can be stored both in parallel and interlocked operations. It is to be noted that the source operand can be accessed only with register addressing mode and the destination operand can be accessed with general addressing modes.

Table 8.7

Instruction	*Description*
STI	- Store Integer operand
STF	- Store floating point operand
STII	- Store integer operand, interlocked
STFI	- Store floating point operand, interlocked
STI || STI	- Store integer operand parallel
STF|| STF	- Store floating point operand parallel

Example 8.17 STF R4, @ 1000h—(Consider the data page pointer value as 80h) This instruction stores the floating point operand present in the register R4 into the data page 128 and the location 1000h (data address 801000h)

	Before execution		After execution
DP	00000080		00000080
R4	2.34353674e + 01	R4	2.34353674e + 01
Content of the location 00801000h			
	1.23425354e + 01		2.34353674e + 01

Example 8.18 STII R2, *AR2++(4)—The integer operand present in the register R2 is loaded in the data memory. The address of the data memory is the content of AR2. An interlocked operation is signaled over pins XF0 and XFl, After execution the new content of AR2 is the sum of the old content of AR2 and the displacement (4).

	Before execution		After execution
R2	23456789	R2	23456789
AR2	00601000	AR2	00601004
Content of the location 00601000h			
	22446688		23456789

Example 8.19 STF R4, *AR4 || STF R3, *-AR3(4)—This is a parallel store instruction. Two floating point operands present in registers R4 and R3 are stored into the data memory. The data memory address of the first operand is the content of AR4 and for the second operand, it is the subtraction of displacement (4) from the current content of AR3 (i.e. 0070000h).

	Before execution		After execution
R4	2.34353674e + 01	R4	2.34353674e + 01
R3	1.23425354e + 01	R3	1.23425354e + 01
AR4	00802000	AR4	00802000
AR3	00700004	AR3	00700004

Content of the location 00802000h

2.46696209e + 02	2.34353674e + 01

Content of the location 00700000h

3.57796412e + 02	1.23425354e + 01

8.4.3 Addition/Subtraction Instructions

The list of addition and subtraction instructions present in ′C3X processor is given in Table 8.8. In addition and subtraction operations there are two operand and three operand instructions. In two operand instructions, one source operand can be added/subtracted with the register content and result is stored in the same register. Where as in three operand instructions, two source operands are added/ subtracted and the result is stored in a different register. It is also possible to add the carry bit and subtract borrow bit with the operands in some instructions.

Table 8.8

Instruction	*Description*
ADDC	Add integers with carry
ADDF	Add floating pointing operands
ADDI	Add integer operands
ADDC3	Add two integer source operands with carry
ADDF3	Add two floating point source operands
ADDI3	Add two integer source operands
SUBB	Subtract integer with borrow
SUBF	Subtract floating point operands
SUBI	Subtract integer operands
SUBRB	Subtract reverse integer with borrow
SUBRF	Subtract reverse floating point value
SUBRI	Subtract reverse integer value
SUBB3	Subtract two source floating point operands with borrow
SUBF3	Subtract two floating point source operands
SUBI3	Subtract two integer source operands

Example 8.20 ADDC R2,R7—This instruction adds the integer operands present in register R2, R7 and the carry bit. The result of addition is stored in register R7.

	Before execution		After execution
	Carry bit C=1		C=1
R2	23456789	R2	23456789
R7	4567A158	R7	68AD08E2

Example 8.21 ADDI *AR4++(5),R1—The integer operand present in the data memory address location pointed AR4 is added with the integer content in register R1. The result is stored in register R1. The new address content of AR4 is the sum of the old content and the displacement (5).

	Before execution		After execution
AR4	00601000	AR4	00601005
Content of the location 00601000h			
	22446688		22446688
R1	23456789	R1	4589CE11

Example 8.22 ADDF3 *AR4, *--AR0(1),R0—The two floating point operands are accessed from the data memory, added and the result is stored in the register R0. The data memory address of the first operand is the content of AR4 and the second is subtraction of the displacement (1) from the current content of AR0 and the same will be the new content in AR0 after execution.

	Before execution		After execution
AR4	00803000	AR4	00803000
AR0	00905006	AR0	00905005
Content of the location 00803000h			
	2.34353674e + 02		2.34353674e + 02
Content of the location 00905005h			
	1.23425354e + 01		1.23425354e + 01
R0	00000000	R0	2.46696209e + 02

Example 8.23 SUBF *AR1—(IR0),R0—The floating point operand present in the data address location pointed by AR1 is subtracted from the content of register R0 and the result is stored in R0. The new address content of AR1 is the subtraction of the index register content IR0 from the old content of AR1.

	Before execution		After execution
AR1	00809880	AR1	00809800
IR0	00000080	IR0	00000080
Content of the location 00809880h			
	1.40500000e + 02		1.40500000e + 02
R0	1.79750000e + 02	R0	3.92500000e + 01

Example 8.24 SUBRI *AR7++(IR1), R7—The integer operand content of the register R7 is subtracted from the data memory content pointed by AR7. This is just the reverse of the rest of the subtract operations, here the destination operand is subtracted from the source operand content. The new content of AR7 is the sum of IR1 content with the old content of AR7.

	Before execution		After execution
AR7	00809000	AR1	00809080
IR1	00000080	IR1	00000080
Content of the location 00809000h			
	00006500		00006500
R7	00003500	R7	00003000

8.4.4 Mulitply Instructions

The two operand and three operand multiply instructions present in ‘C3X processors are listed in Table 8.9. In two operand instructions, the product of the destination operand and the source operand is loaded into the destination register. In three operand instructions, the product of the two source operands is loaded into the destination register.

Table 8.9

Instruction	*Description*
MPYF	Multiply floating point operands
MPYI	Multiply integer operands
MPYF3	Multiply two floating point source operands
MPYI3	Multiply two integer source operands

Example 8.25 MPYI R1,R2—The integer operand present in register R2 is multiplied with the content of register R1 and the result is stored in register R2.

	Before execution		After execution
R1	00003457	R1	00003457
R2	00002356	R2	07397A3A

Example 8.26 MPYF3 *AR6, R5,R0—The floating point operands present in the register R5, the data memory location pointed by AR6 are multiplied and the result is stored in register R0.

	Before execution		After execution
AR6	00809000	AR6	00809000

Content of the location 00809000h

	1.40500000e + 02		1.40500000e + 02
R5	6.28125000e + 01	R5	6.28125000e + 01
R0	00000000	R0	8.82515625e + 03

8.4.5 Logical Instructions

The instructions that support logical operations in ′C3X are listed in Table 8.10. The two operand and three operand logical OR, AND and EXOR operations can be performed. The complement operation can be separately performed for the operands or it can be performed along with the AND operation.

Table 8.10

Instruction	*Description*
AND	Logical-AND operation
ANDN	Logical-AND between destination operand and the complement of the source operand
NOT	Logical complement operation
OR	Logical-OR operation
XOR	Logical exclusive-OR operation
AND3	Logical-AND operation between two source operands
ANDN3	Logical-AND operation between sourcel operand and the complement of the source2 operand
OR3	Logical-OR between two source operands
XOR3	Logical exclusive-OR between two source operands

Example 8.27 ANDN R1,R2—The bitwise logical-AND operation is performed between the (dst) operand in register R2 and the complement value of the content of the (src) register R1. The result of the AND operation is stored in the register R2.

	Before execution		After execution
R1	00002A2B	R1	00002A2B
R2	0002C2D	R2	00000404

Example 8.28 OR *AR2++(5),R7—The bitwise logical OR operation is performed between the data operand in the data memory address location pointed by AR2 and the register content R7. The result is stored in register R7. The new address content in AR2 is the sum of displacement (5) with the old content of AR2.

	Before execution		After execution
AR2	00809000	AR2	00809005

Content of the location 00809000h

	01245000		01245000
R7	00002A2B	R7	01247A2B

Example 8.29 XOR3 *--AR2(1), R4,R2—The bitwise logical exclusive OR operation is performed between data memory operand and the content of register R4. The address of the data memory operand is the subtraction of the displacement (1) from the content of AR2. The result is stored in register R2.

	Before execution		After execution
AR2	00809005	AR2	00809004

Content of the location 00809004h

	000FF5C1		000FF5C1
R4	000FFA32	R4	000FFA32
R2	00000000	R2	00000F33

8.4.6 Conditional Instructions

The TMS320C3X processors support conditional instructions. There are 20 conditions that can be specified in the condition (*cond*) field of any of the conditional instructions. The conditions include, signed and unsigned comparisons, comparisons to 0, and comparisons based on the status of individual flags. The status register contains seven conditional flags. These flags provide the information about the properties of the result of arithmetic and logical instructions. The flag bits and its details are explained in Table 8.11. The conditional codes are based on the status of these flag bits. The list of conditions and its description are given in Table 8.12. The specified condition in the conditional field is true, the respective operation is performed, if it is false the next instruction is executed. The load, branch, call, return and trap instructions can be conditional instructions.

Table 8.11 *The status register flags and its descriptions*

Flag	*Description*
1. Latched Floating-point underflow conditional flag (LUF)	LUF is set whenever UF is set. LUF can be cleared by resetting the processor or by modifying it in the status register.
2. Latched overflow condition flag (LV)	LV is set Whenever V is set. LV can be cleared by resetting the processor or by modifying it in the status register.
3. Floating-point underflow condition flag (UF)	Whenever the exponent of the result is less than or equal to –128, a floating point under flow occurs. UF is set for under flow, if not it is cleared. The output value is set to zero for under flow.
4. Negative condition flag (N)	For logical, integer and floating point operations N is set if the result is negative, and cleared otherwise. Zero is positive.
5. Zero condition flag (Z)	For logical integer and floating point operations, Z is set if the output is 0 and cleared otherwise.
6. Overflow condition flag(V)	For integers if the maximum positive (2^{32-1}) and negative (-2^{32}) numbers are obtained in the result V is set, otherwise it is cleared. For floating point operations, if the exponent of the result is greater than 127, V is set; otherwise it is cleared.
7. Carry flag (C)	When integer addition results a carry or in an integer subtraction, a borrow occurs to the MSB of the output, the C bit is set otherwise it is cleared.

Table 8.12 *Flag conditions and descriptions*

Condition	*Description*
Unconditional Compares	
U	Unconditional
Unsigned Compares	
LO	Lower than
LS	Lower than or same as
HI	Higher than
HS	Higher than or same as
EQ	Equal to
NE	Not equal to
Signed Compares	
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to
EQ	Equal to
NE	Not equal to
Compare to Zero	
Z	Zero
NZ	Not zero
P	Positive
N	Negative
NN	Nonnegative

(Contd.)

Table 8.12 *(Contd.)*

Compare to Condition Flags	
NN	Nonnegative
N	Negative
NZ	Nonzero
Z	Zero
NV	No overflow
V	Overflow
NUF	No underflow
UF	Underflow
NC	No carry
C	Carry
NLV	No latched overflow
LV	Latched overflow
NLUF	No latched floating-point underflow
LUF	Latched floating-point underflow
ZUF	Zero or floating-point underflow

Example 8.30 LDFZ R3,R5—The floating point operand load operation from register R3 to R5 is performed, if the Z flag bit is set, if not this instruction will not be executed.

Before execution		After execution	
Z-Flag bit value =1		Z=1	
R3	1.40500000e + 02	R3	1.40500000e + 02
R5	6.28125000e + 01	R5	1.40500000e + 02

8.4.7 Shift and Rotate Instructions

The operands in TMS320C3X processor can be shifted and rotated right as well as left. The instructions that support the shift and rotate operations are listed in Table 8.13.

Table 8.13

Instruction	*Description*
ASH	Arithmetic shift
ASH3	Arithmetic shift (three operand instruction)
LSH	Logical shift
LSH3	Logical shift (three operand instruction)
ROL	Rotate left
ROLC	Rotate left through carry
ROR	Rotate right
RORC	Rotate right through carry

There are two kinds of shift operations, the arithmetic shift and logical shift. The syntax of the two operand and three operand shift instructions are;

Two-operand instructions	*mnemonic count, dst*
Three-operand instructions	*mnemonic count, src, dst*

In two-operand instructions, if the *count* operand is greater than zero, the left shift operation is done. The *dst* operand is left-shifted by the value of the count operand. The lower order bits, which are shifted in are zero-filled and the higher order bits are shifted out through carry bit. This is same both for the arithmetic and logical shift instructions.

If the count operand is less than zero, the right shift operation is done. The *dst* operand is right shifted by the absolute value of the *count* operand. The higher order bits of the *dst* operand during the right shift are sign extended and the lower order bits are shifted out through the carry bit for the arithmetic shift. But for the logical shift operation the difference is, during the right shift, the higher order bits are zero-filled and the lower order bits are shifted out through the carry bit.

If the count operand is zero, no shift is performed both for arithmetic and logical shift operations and the C bit is set zero. It is to be noted that only seven LSBs of the *count* operand are used for the shift operation. In arithmetic shift both the *count* and *dst* operands are considered to be signed integers. But for logical shift the *count* operand is considered as signed integer and the *dst* operand is considered as the unsigned integer.

As far as the rotate operations are concerned there are two kinds of rotate operations, rotate right and left. The syntax of the rotate instructions is;

mnemonic dst

In rotate left/right operation, the dst operand is left/right rotated by one bit and loaded into dst register. In the case of rotate left, the content of MSB bit is transferred into LSB and for rotate right; the LSB bit is transferred into the MSB bit. In both case, the shifted MSB (rotate left) and LSB (rotate right) bits are copied into carry bit. It is also possible to perform the rotate right and left operations through the content of carry bit.

Example 8.31 ASH R2,R3—The content of the register R3 is shifted based on the content of register R2 and the result is loaded into register R3. Consider the content of register R2 is 08h. The content of register R3 is left shifted by 8 bits and the result is loaded into register R3.

	Before execution		After execution
R2	00000008	R2	00000008
R3	00002458	R3	00245800

Example 8.32 LSH3 *AR2,R0,R2—The register content of R0 is shifted based on the content of data memory location pointed by AR2. The result of the shift operation is loaded into register R2. Consider the content of data memory location value is –8 (FFFFFFF8h). The content of register R0 is right shifted by 8 bits and the result is loaded into register R2.

	Before execution		After execution
AR2	00801000	AR2	00801000

Content of the location 00801000h

	FFFFFFF8		FFFFFFF8
R0	00569800	R0	00569800
R2	00000000	R2	00005698

Example 8.33 ROL R7 - The content of register R7 is rotated left by one bit. The content of MSB bit is transferred to LSB bit and also copied to carry bit.

	Before execution		After execution
R7	82233455	R7	044668AB
	Carry bit C = 0		C = 1

Example 8.34 RORC R2—The content of the register R2 is rotated right through the carry bit. The content of LSB is transferred into carry bit, the content of carry bit is transferred to MSB bit.

	Before execution		After execution
R2	04060040	R2	82030020
	Carry bit C = 1		C = 0

8.4.8 Program Control Instructions

The program control instruction group consists of 17 instructions, which affect the program flow. The standard and delayed branch instructions, call instructions and return instructions are supported. Certain program control instructions support both conditional and unconditional operations. There are two repeat mode instructions, one allows repetition of a single code and the other allows repetition of a block of code.

Branch, Call and Return Instructions The ′C3X processors support branch, call and return instructions. The branch operation can be conditional, unconditional, delayed and decrement & branch instructions. The call operations are conditional and unconditional, and the return instructions are conditional only. The list of branch, call and return instructions is given in Table 8.14.

Table 8.14

Instruction	*Description*
BR	Standard unconditional branch
BRD	Delayed unconditional branch
Bcond	Standard conditional branch
BcondD	Delayed conditional branch
Dbcond	Decrement and standard conditional branch
DBcondD	Decrement and delayed conditional branch
CALL	Call subroutine (unconditional)
CALL cond	Call subroutine conditionally
RETS cond	Return from subroutine conditionally
RETI cond	Return from interrupt conditionally

Due to pipeline operation, the branch operation happens only after four machine cycles. In delayed branch, before branch instruction is being executed, three instructions after the delayed branch instruction are executed. The effect is a single cycle branch. In the case of decrement and branch instruction, the content of the specified auxiliary register is decremented before the branch operation is performed.

The syntax of the branch/ call instructions are *mnemonic src.* The *src* operand is 24-bit unsigned integer, which is the new address to be loaded into the program counter (PC). If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative addressing mode, the assembler generates a displacement. The displacement is label - (PC of branch instruction +1) for the conditional branch/call instructions. It is label - (PC of branch instruction +3) for the delayed conditional branch. The new PC value is the sum of the PC value of the branch/call instruction, the displacement plus one for conditional instructions and plus 3 for the delayed conditional instructions.

Example 8.35 BR 8900h—This is an unconditional branch instruction. When this instruction is executed, the current content of program counter (PC) is pushed to the stack and the new integer value given in the instruction (8900) is loaded into PC.

	Before execution		After execution
PC	00001000	PC	00008900

Example 8.36 BNZD 36h—This branch instruction is executed, if the condition specified in the instruction is true. Since this is a delayed branch instruction the three instructions after the delayed branch instruction are fetched before the PC is modified. The displacement for this instruction is 39h (36+3). This will be added with the current content of PC.

	Before execution		After execution
PC	001000	PC	001039
	Z Flag bit Z = 0		Z = 0

Repeat Instructions The ′C3X processors support two repeat instructions to support zero-overhead looping.

RPTB – Repeat a block of code

RPTS – Repeat a single instruction

These two instructions are four-cycle instructions; these four cycles are needed only in the initial execution of the loop and all subsequent executions of the loop have no overhead.

There are three memory-mapped registers, RS - Repeat start address register, RE - Repeat end address register, RC - Repeat counter register and Two flag bits RM bit and S bit, used for the repeat operations. The RS register holds the address of the first instruction of the block of code to be repeated and RE holds the last address. The RC register contains a value one less than the number of times the block to be repeated. For single repeat instruction both RS and RE will have the same value of the address of the instruction to be repeated.

The repeat mode flag bit (RM), which is present in the status register (ST), specifies whether the processor is running in the repeat mode. The S bit is internal to the processor and cannot be programmed but this bit is necessary to describe the single instruction repeat operation. If the RM flag bit is set, it specifies the repetitions of a black of code. If both RM bit and S bit are set, it indicates single instruction repeat operation. The maximum number of repetitions that is possible in ′C3X processor is 8000 000 lh times.

8.4.9 Low-power Control Instructions

The low-power control instruction group consists of four instructions that can be used for low-power modes. The low-power instructions and its description are given in Table 8.15.

Table 8.15

Instruction	*Description*
IDLE	Idle until interrupt
IDLE2	Low-power idle
LOPOWER	Divide the clock by 16
MAXSPEED	Restore clock to regular speed

The IDLE instruction stops the CPU operations until an interrupt is received. The global interrupt bit is set. The IDLE2 instruction serves the same function as IDLE, but it removes the functional clock input from the internal devices. When LOPOWER instruction is used, the processor continues the execution of instructions at the reduced clock rate. The input clock rate is divided by 16 times. For MAXSPEED, it exits the LOPOWER power down mode and starts the execution with full speed.

APPENDIX 8

INSTRUCTION SET SUMMARY—FUNCTIONAL GROUPS

A8.1 Load, Store, Push and Pop Instructions

Mnemonic	*Description*
LDE	Load floating-point exponent
LDF	Load floating-point value
LDF cond	Load floating-point value conditionally
LDI	Load integer
LDI cond	Load integer conditionally
LDM	Load floating-point mantissa
LDP	Load data page pointer
STF	Store floating-point value
STI	Store integer
POP	Pop integer from stack
POPF	Pop floating-point value from stack
PUSH	Push integer on stack
PUSHF	Push floating-point value on stack

A8.2 Two-operand Instructions

Mnemonic	*Description*
ABSF	Absolute value of a floating-point number
ABSI	Absolute value of an integer
ADDC	Add integers with carry
ADDF	Add floating-point values
ADDI	Add integers ROL Rotate left
AND	Bitwise-logical AND
ANDN	Bitwise-logical AND with complement
ASH	Arithmetic shift
CMPF	Compare floating-point values
CMPI	Compare integers
FIX	Convert floating-point value to integer
FLOAT	Convert integer to floating-point value
LSH	Logical shift
MPYF	Multiply floating-point values
MPYI	Multiply integers

NEGB	Negate integer with borrow
NEGF	Negate floating-point value
NEGI	Negate integer
NORM	Normalize floating-point value
NOT	Bitwise-logical complement
OR	Bitwise-logical OR
RND	Round floating-point value
ROLC	Rotate left through carry
ROR	Rotate right
RORC	Rotate right through carry
SUBB	Subtract integers with borrow
SUBC	Subtract integers conditionally
SUBF	Subtract floating-point values
SUBI	Subtract integer
SUBRB	Subtract reverse integer with borrow
SUBRF	Subtract reverse floating-point value
SUBRI	Subtract reverse integer
TSTB	Test bit fields
XOR	Bitwise-exclusive OR

A8.3 Three-operand Instructions

Mnemonic	*Description*
ADDC3	Add with carry
ADDF3	Add floating-point values
ADDI3	Add integers
AND3	Bitwise-logical AND
ANDN3	Bitwise-logical AND with complement
ASH3	Arithmetic shift
CMPF3	Compare floating-point values
CMPI3	Compare integers
LSH3	Logical shift
MPYF3	Multiply floating-point values
MPYI3	Multiply integers
OR3	Bitwise-logical OR
SUBB3	Subtract integers with borrow
SUBF3	Subtract floating-point values
SUBI3	Subtract integers
TSTB3	Test bit fields
X0R3	Bitwise-exclusive OR

A8.4 Program Control Instructions

Mnemonic	*Description*
B cond	Branch conditionally (standard)
B condD	Branch conditionally (delayed)

BR	Branch unconditionally (standard)
BRD	Branch unconditionally (delayed)
CALL	Call subroutine
CALL cond	Call subroutine conditionally
DB cond	Decrement and branch conditionally(standard)
DB condD	Decrement and branch conditionally(delayed)
IACK	Interrupt acknowledge
IDLE	Idle until interrupt
NOP	No operation
RETI cond	Return from interrupt conditionally
RETS cond	Return from subroutine conditionally
RPTB	Repeat block of instructions
RPTS	Repeat single instruction
SWI	Software interrupt
TRAP cond	Trap conditionally

A8.5 Low-power Control Instructions

Mnemonic	*Description*
IDLE2	Low-power idle
LOPOWER	Divide clock by 16
MAXSPEED	Restore clock to regular speed

A8.6 Interlocked Operations Instructions

Mnemonic	*Description*
LDFI	Load floating-point value, interlocked
LDII	Load integer, interlocked
SIGI	Signal, interlocked
STFI	Store floating-point value, interlocked
STII	Store integer, interlocked

A8.7 Parallel Instructions

(a) Parallel Arithmetic with Store Instructions

Mnemonic	*Description*
ABSF || STF	Absolute value of a floating point
ABSI || STI	Absolute value of an integer
ADDF3 || STF	Add floating-point value (3 operand)
ADDI3 || STI	Add integer (3 operand)
AND3 || STI	Bitwise-logical AND (3 operand)
ASH3 || STI	Arithmetic shift (3 operand)
FIX || STI	Convert floating-point value to integer
FLOAT || STF	Convert integer to floating-point value
LDF || STF	Load floating-point value
LDI || STI	Load integer
LSH3 || STI	Logical shift

MPYF3 || STF	Multiply floating-point value
MPYI3 || STI	Multiply integer
NEGF || STF	Negate floating-point value
NEGI || STI	Negate integer
NOT || STI	Complement
OR3 || STI	Bitwise-logical OR
STF || STF	Store floating-point value
STI || STI	Store integer
SUBF3 || STF	Subtract floating-point value
SUBI3 || STI	Subtract integer
X0R3 || STI	Bitwise-exclusive OR

(b) Parallel Load Instructions

Mnemonic	*Description*
LDF || LDF	Load floating-point value
LDI || LDI	Load integer

(c) Parallel Multiply and Add/Subtract Instructions

Mnemonic	*Description*
MPYF3 || ADDF3	Multiply and add floating-point value
MPYF3 || SUBF3	Multiply and subtract floating-point value
MPYI3 || ADDI3	Multiply and add integer
MPYI3 || SUBI3	Multiply and subtract integer

INSTRUCTION SET SUMMARY—ALPHABETICAL ORDER

Mnemonic	*Description*
ABSF	Absolute value of a floating-point number
ABSI	Absolute value of an integer
ADDC	Add integers with carry
ADDC3	Add integers with carry
ADDF	Add floating-point values
ADDF3	Add floating-point values
ADDI	Add integers
ADDI3	Add integers
AND	Bitwise-logical AND
AND3	Bitwise-logical AND (3-operand)
ANDN	Bitwise-logical AND with complement
ANDN3	Bitwise-logical ANDN (3-operand)
ASH	Arithmetic shift
ASH3	Arithmetic shift (3-operand)
B cond	Branch conditionally (standard)
B condD	Branch conditionally (delayed)
BR	Branch unconditionally (standard)
BRD	Branch unconditionally (delayed)

Mnemonic	Description
CALL	Call subroutine
CALL cond	Call subroutine conditionally
CMPF	Compare floating-point values
CMPF3	Compare floating-point values (3-operand)
CMPI	Compare integers
CMPI3	Compare integers (3-operand)
DB cond	Decrement and branch conditionally (standard)
DB condD	Decrement and branch conditionally (delayed)
FIX	Convert floating-point value to integer
FLOAT	Convert integer to floating-point value
IACK	Interrupt acknowledge
IACK	Toggled low, then high
IDLE	Idle until interrupt
IDLE2	Low-power idle
LDE	Load floating-point exponent
LDF	Load floating-point value
LDF cond	Load floating-point value conditionally
LDFI	Load floating-point value, interlocked
LDI	Load integer
LDI cond	Load integer conditionally
LDII	Load integer, interlocked
LDM	Load floating-point mantissa
LDP	Load data page pointer
LOPOWER	Divide clock by 16
LSH	Logical shift If count
LSH3	Logical shift (3-operand)
MAXSPEED	Restore clock to regular speed
MPYF	Multiply floating-point values
MPYF3	Multiply floating-point value (3-operand)
MPYI	Multiply integers
MPYI3	Multiply integers (3-operand)
NEGB	Negate integer with borrow
NEGF	Negate floating-point value
NEGI	Negate integer
NOP	No operation
NORM	Normalize floating-point value Normalize
NOT	Bitwise-logical complement
OR	Bitwise-logical OR Dreg
OR3	Bitwise-logical OR (3-operand)
POP	Pop integer from stack
POPF	Pop floating-point value from stack
PUSH	Push integer on stack

PUSHF	Push floating-point value on stack
RETI cond	Return from interrupt conditionally
RETS cond	Return from subroutine conditionally
RND	Round floating-point value round
ROL	Rotate left dreg rotated left 1 bit
ROLC	Rotate left through carry
ROR	Rotate right dreg rotated right
RORC	Rotate right through carry
RPTB	Repeat block of instructions
RPTS	Repeat single instruction
SIGI	Signal, interlocked
STF	Store floating-point value
STFI	Store floating-point value, interlocked
STI	Store integer
STII	Store integer, interlocked
SUBB	Subtract integers with borrow
SUBB3	Subtract integers with borrow (3-operand)
SUBC	Subtract integers conditionally
SUBF	Subtract floating-point values
SUBF3	Subtract floating-point values (3-operand)
SUBI	Subtract integers
SUBI3	Subtract integers (3-operand)
SUBRB	Subtract reverse integer with borrow
SUBRF	Subtract reverse floating-point value
SUBRI	Subtract reverse integer
SWI	Software interrupt
TRAP cond	Trap conditionally
TSTB	Test bit fields
TSTB3	Test bit fields (3-operand)
XOR	Bitwise-exclusive OR
XOR3	Bitwise-exclusive OR (3-operand)

Review Questions

8.1 List the various data formats available in 'C3X processor.

8.2 What are all the various addressing modes available in 'C3X processor?

8.3 Explain how the address of the operand is computed in direct addressing mode?

8.4 In indirect addressing mode, what is use of displacement field?

8.5 In indirect addressing mode, what all are the various ways you can modify AR?

8.6 Explain what all are the registers to be programmed for circular addressing?

8.7 Explain, how bit reversed addressing is useful for the computation of FFT?

8.8 List the group of addressing modes in 'C3X processor.

8.9 Explain the various conditional branch instructions available in 'C3X processor.

8.10 Explain about three operand instructions and its functions

8.11 What is meant by interlocked operation? What is the use of this?

8.12 Explain parallel instructions. What are its advantage?

8.13 Explain the various registers that are to be programmed for block repeat operation.

8.14 Explain the various conditions that can be specified in conditional instructions

8.15 Explain the various low power control instructions available in 'C3X processor.

Self Test Questions

8.1 The number of types of addressing modes in 'C3X processor are
(a) 4 (b) 5 (c) 6 (d) 3

8.2 The number of assembly instructions in 'C3X processor are
(a) 120 (b) 130 (c) 113 (d) 100

8.3 In short integer format the number of bits used to represent the operand are
(a) 8 (b) 16 (c) 24 (d) 32

8.4 In single precision and extended precision floating point format the exponent is of size
(a) 4 bits (b) 6 bits (c) 8 bits (d) 12 bits

8.5 The registers used in register addressing mode are
(a) AR0-AR7 (b) R0-R7 (c) IR0&IR1

8.6 The total memory space accessible by direct addressing mode of 'C3X processor is
(a) 64 K (b) 256 K (c) 16 M (d) 8M

8.7 In direct addressing the number of bits of the instruction word used for address computation are
(a) 8 (b) 16 (c) 24 (d) 32

8.8 The symbol used to specify the direct addressing mode is
(a) * (b) #
(c) @ (d) no symbol used

8.9 The registers used to specify the address of the operand in indirect addressing mode are
(a) R0-R7 (b) AR0-AR7 (c) ST, IE and IF

8.10 The number of bits of AR used to specify the operand address in indirect addressing mode are
(a) 16 (b) 8 (c) 24 (d) 32

8.11 In indirect addressing mode the possible maximum displacement of the address using (disp) field is
(a) 4 (b) 32 (c) 256 (d) FFFFh

8.12 For bit reversed addressing mode of'C3X the index register used is
(a) IR0 (b) IR1 (c) both IR0 and IRl

8.13 In immediate addressing mode the operand value can be
(a) 16 bits (b) 24 bits (c) both 16 and 24 bits

8.14 The PC-relative addressing mode is used for
(a) direct memory access (b) indirect memory access
(c) for branch operation (d) for interrupt operation

8.15 In circular addressing mode the block size register (BK) specifies
(a) the size of the buffer
(b) the starting address of the buffer
(c) the end address of the buffer

8.16 In circular addressing the maximum size of the circular buffer can be
(a) 32 (b) 16K (c) 24K (d) 64K

8.17 In circular addressing the registers to be programmed are
(a) BK and ARn (b) BK and IRn (c) ARn and IRn

8.18 Bit reversed addressing is used for
(a) FIR filters (b) FFT computation (c) IIR filters

8.19 In three operand instructions the destination operand (dst) is
(a) data memory location
(b) program memory location
(a) (c) registers R0-R7 (d) register AR0-AR7

8.20 The number of conditions available in conditional instructions are
(a) 10 (b) 4 (c) 20 (d) 8

8.21 The number of conditional flags in ST are
(a) 9 (b) 7 (c) 5 (d) 3

8.22 The number of machine cycles needed for the execution of branch, call and return instructions are
(a) 2 (b) 3 (c) 4 (d) 5

8.23 In repeat instructions, the maximum number of times the repetition is possible is
(a) 8000 0000h (b) 8000 0001h (c) FFFF FFFFh

8.24 The number of low-power control instructions in 'C3X processor are
(a) 2 (b) 3 (c) 4 (d) 5

8.25 In parallel operation instructions the type of addressing mode that can be used are
(a) direct and indirect (b) register and indirect
(c) register and direct

9 APPLICATION PROGRAMS IN C3X

In this chapter, some application programs in TMS320C3X processors are given. To test these programs, assembly language programming tools and certain hardware accessories are needed. The programming tools are used to write the ′C3X assembly language programs and converted into machine language codes. The machine language codes are then loaded into the ′C3X DSP using the hardware accessories. It is necessary to supply the real time signals to the DSP for processing. The real time signals, which are analog in nature are to be digitised and after processing the signal digitally in DSP, they have to be converted back to analog signals. So, along with DSP, a minimum number of devices are to be connected for real time signal processing. In this chapter application program development using the TI’s TMS320C3X starter kit is discussed.

9.1 TMS320C3X STARTER KIT (DSK)

9.1.1 Overview of TMS320C3X Starter Kit

The ′C3X starter kit (DSK) is a low-cost, simple, high-performance stand-alone application development board. The DSK has on-board, industry standard TMS320C31 floating-point processor. This allows us to verify ′C3X codes with full speed and experiment real time signal processing. The block diagram of DSK is shown in Fig. 9.1. The 50 MHz system clock makes the instruction cycle time of ′C31 as 40 ns (25 MHz) and this allows execution of instructions upto 50 MFLOPS and 25 MIPS. A standard or enhanced parallel port interface is used to connect the DSK to host PC and through this interface, the communication is carried out between ′C31 and PC.

The DSK has TLC320C40, an analog interface circuit (AIC). The AIC consists of variable rate analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) with 14-bit dynamic range 20,000 samples per second. There are two standard RCA plug connectors, for analog input and outputs.

All the signals of ′C3X are routed to expansion connectors. The expansion connectors include four 32-pin headers, an 11-pin jumper block, and a 12-pin XDS510 header. This feature gives freedom to design new daughter boards, to create new software on a host PC and to run the software on the DSK board.

The software tool used for development of ′C3X code, downloading and running it on the DSK board are the ′C3X assembler and debugger. The debugger is windows oriented and this simplifies the code development and debugging capabilities.

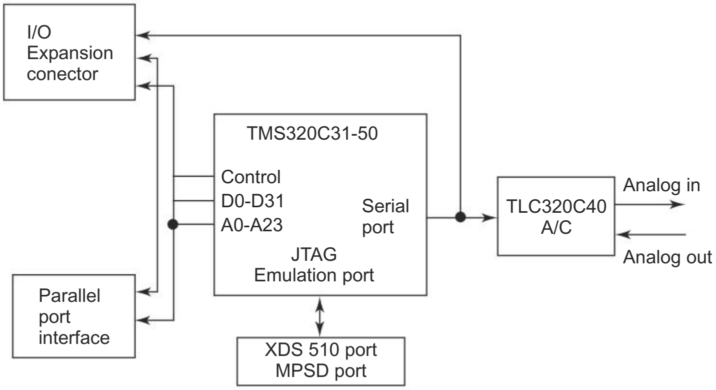


Fig. 9.1 *TMS320C3X starter kit block diagram*

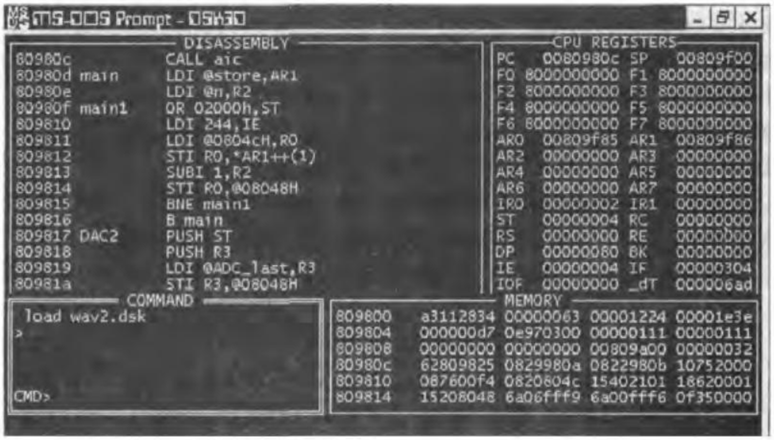
9.1.2 DSK Software Tools

The TMS320C3X starter kit uses the DSK assembler and debugger tools. These tools help to develop, test and refine the ′C3X assembly language programs.

The assembler converts the assembly language codes into machine language codes. This assembler is different from other assemblers because it does not go through the linker phase to create the output file. The DSK uses special directives to assemble code at an absolute address during the assembly phase. The command to run the assembler is **dsk3a [file name] [-options].**

The debugger can load and execute source codes with single-step, with break points and can be run with halt capabilities. The command to invoke the debugger windows is **dsk3d.**

The debugger is used to download the machine language code generated by the assembler on an actual ′C3X DSP. The debugger is a window-oriented interface that reduces the need of complex commands and learning time. The list of the debugger commands and their descriptions is given in Appendix 9A. There are four windows in the debugger display. They are disassembly window, register window, command window and memory window. The debugger display is shown in the Fig. 9.2.



```
DISASSEMBLY
80980c              CALL aic
80980d main         LDI @store,AR1
80980e              LDI @n,R2
80980f main1        OR 02000h,ST
809810              LDI 244,IE
809811              LDI @0804cH,R0
809812              STI R0,*AR1++(1)
809813              SUBI 1,R2
809814              STI R0,@08048H
809815              BNE main1
809816              B main
809817 DAC2         PUSH ST
809818              PUSH R3
809819              LDI @ADC_last,R3
80981a              STI R3,@08048H
```

```
CPU REGISTERS
PC    0080980c SP    00809f00
F0  8000000000 F1  8000000000
F2  8000000000 F3  8000000000
F4  8000000000 F5  8000000000
F6  8000000000 F7  8000000000
AR0   00809f85 AR1   00809f86
AR2   00000000 AR3   00000000
AR4   00000000 AR5   00000000
AR6   00000000 AR7   00000000
IR0   00000002 IR1   00000000
ST    00000004 RC    00000000
RS    00000000 RE    00000000
DP    00000080 BK    00000000
IE    00000004 IF    00000304
IOF   00000000 _dT   000006ad
```

```
COMMAND
 load wav2.dsk
>

CMD>
```

```
MEMORY
809800   a3112834 00000063 00001224 00001e3e
809804   000000d7 0e970300 00000111 00000111
809808   00000000 00000000 00809a00 00000032
80980c   62809825 0829980a 0822980b 10752000
809810   087600f4 0820804c 15402101 18620001
809814   15208048 6a06fff9 6a00fff6 0f350000
```

Fig. 9.2 *C3X debugger display*

The disassembly window's first column consists of the on-chip memory address values, the second column contains the op-codes of the assembly codes and the third column has the assembly source codes. The register window shows the latest values of the register file contents during the execution of the program. The command window is used to write the required commands for the proper operation of the debugger. The memory window is used to see the content of the on-chip memory.

9.1.3 Code Development in DSK

The text editors present in PC can be used to write the assembly source files. The name of the assembly file should have the file extension **.asm.** This is to indicate to the assembler that the input file is a valid assembly file. The syntax for writing the source code is as follows

[label] [:] mnemonic [operand list] [;comments]

Whatever guidelines are used to write assembly language program in TMS320C5X (Section 6.1.4.) are valid for ′C3X. Once the development of assembly file is over, the assembler is invoked using the syntax

dsk3a [file name] [-options]

The assembler takes the assembly file as input file and creates an executable file with file extension .dsk. If there are any errors in the assembly language input file, the valid executable file will not be created. During the assembling process, status of the progress of the assembly is flashed through messages. These messages include the type of errors if any in the program. The software development flow is shown in Fig. 9.3.

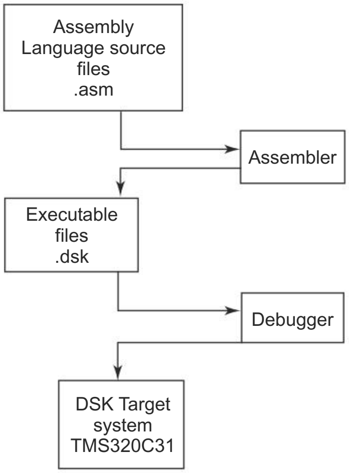

Assembly Language source files .asm
Assembler
Executable files .dsk
Debugger
DSK Target system TMS320C31

Fig. 9.3 *DSK software development flow*

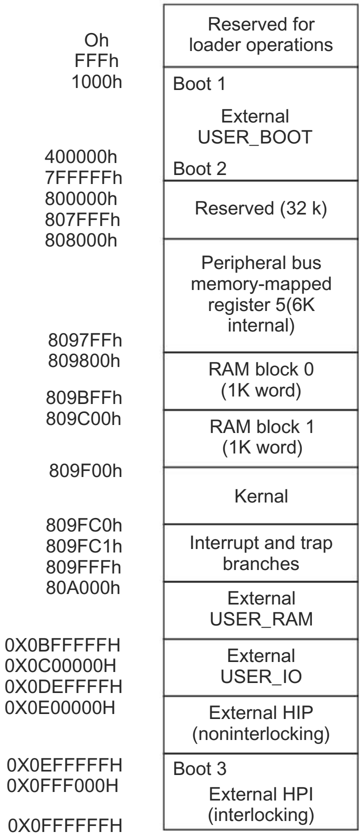


Fig. 9.4 *C3X DSK memory map*

After the successful assembly, the debugger is invoked using the command dsk3d. The executable file (.dsk) is now loaded into the target system (TMS320C31) and the execution of the program is carried out in the ′C31 DSP.

9.1.4 DSK Memory Map

The target system, TMS320C31 DSP has two lK-32-bit word on-chip dual access memory. The executable files created by the assembler are loaded into the on-chip RAM. It is to be noted that the starting address of assembly language program is to be indicated in the beginning of the program using the assembler directive *.start.* The address that can be used for the source code is according to the memory DSK memory map shown in Fig. 9.4. The user can use the on-chip address locations starting from 809800h to 809F00h and the last 256 location of RAM block 1 is used for the DSK kernel, interrupt, and trap tables. If the application program source code is more than 2K in size, external memory space can be used.

9.1.5 Assembler Directives

The assembler directives are used to supply data for the program and to control the assembly process. They assemble the code and data into specified locations. The initialisation of memory space is carried out and they also reserve space for uninitialised variables. The basic assembler directives needed for writing the assembly programs are .start, .sect, .entry, .word, .float and .end. The complete list of assembler directives used in ′C3X programming is given in Appendix 9B. Program 9.1 gives an example which illustrates the basic format of the ′C3X assembly language program.

Program 9.1 Example program for C3X assembler directives

```
Label        Mnemonic              Comments
             .start " example",
             0x809800              ;the starting address of the section example
                                   ;is 809800h (on-chip memory location)
             .sect " example"      ;assemble the code into the named section
a            .word 1000h           ;the integer 1000h is reserved for the variable a
b            .float 10.5           ;the floating-point value 10.5 is reserved
                                   ;for the variable b
             .entry                ;initialise the program counter with start-
                                   ;ing address when the program is loaded into ;DSP
             LDI 10h, R0           ;the integer 10h is loaded into R0
             LDF 10.5, R1          ;the floating-point integer 10.5 is loaded
                                   ;into R1
             ADDI 20h, R2          ;the content of R2 is added with the given
                                   ;integer 20h and result is stored in R2
             .end                  ;program end
```

The .start assembler directive is used to specify the starting address for the section program (here the name of the section is example). For the ′C3X starter kit on-chip RAM starting address is 0x809800h. .sect assembler directive assemble the source code into the named section. The assembler directives .word and .float are used to initialise one or more integers and floating-point constants respectively, .entry assembler directive will end the assembler directive section and after this the ′C3X processor source code will start. This assembler directive initialises the starting address of the program counter when the assembled file is loaded into the DSP starter kit using the ′C3X debugger. The assembler directive .end is to end the program section.

Program 9.2 Example program on immediate addressing mode in C3X

```
Label        Mnemonic              Comments
             .start "immediate"
             ,0x809800             ;the starting address of the section immedi-
                                   ;ate is 809800h (on-chip memory location)
             .sect "immediate"     ;assemble the source code into the named
                                   ;section
             .entry                ;initialise the program counter with start-
```

```
                                                ;ing address when the program is loaded into
                                                ;DSP
        LDI 44h, R0                             ;the integer operand 44h is loaded into R0
        LDI –10h, R1                            ;the two’s complement value of 10h is loaded
                                                ;into R1
        ABSI –34.5, R4                          ;the absolute value of the operand 34.5 is
                                                ;loaded into R4, the sign and the fractional
                                                ;value is discarded
        ADDI 10h, R0                            ;the operand 10h is added with the content of
                                                ;R0 and the result of addition is stored in R0
        SUBI 10h, R0                            ;the operand 10h is subtracted from the
                                                ;content of R0 and the result of subtraction
                                                ;is loaded into R0
        MPYI 10h, R0                            ;the operand 10h is multiplied with the
                                                ;content of R0 and the result is stored in R0
        AND 55h, R0                             ;the bit-by-bit and operation is performed
                                                ;with the content of R0 and the operand 55h and
                                                ;the result is stored in R0
        LDF 10.5, R2                            ;the operand 10.5 is loaded into R2 in float-
                                                ;ing-pointing format
        LDF -20.5, R3                           ;the operand 20.5 is loaded into R3 with the
                                                ;sign-bit set
        ADDF 9.6, R2                            ;the floating-point operand 9.6 is added with
                                                ;the content R2, result is stored back in R2
        SUBF 10.1, R2                           ;the floating-point operand 10.1 is
                                                ;subtracted from the content of R2, the
                                                ;result is loaded back into R2
        MPYF 4.5, R2                            ;the floating-point operand 4.5 is multi-
                                                ;plied with the register content R2, the result
                                                ;is stored back into R2
        LDI 10h, AR0                            ;the operand 10h is loaded into the auxiliary
                                                ;register ARO
        LDI 20h, IR0                            ;the operand 20h is loaded into the index
                                                ;register IR0
        LDI 40h, BK                             ;the operand 40h is loaded into blcck size
                                                ;register BK
        .end                                    ;program end
```

EXAMPLE PROGRAMS FOR ADDRESSING MODES 9.2

9.2.1 Immediate Addressing Mode

The example in Program 9.2 illustrates some instructions which use the immediate operands. The operand can be given in the instruction itself as 16-bits or 24-bits immediate values (short integer or long integer). The immediate operand can be both integer and floating-point values. The immediate operands are loaded into all the registers in the register file. The extended precision register accepts both the integer and floating-point operands, whereas the other registers accept only the integer operands.

The negative numbers are represented in two's complement format. The first set of instructions are using integer operands for the various operations, the second set of instructions use floating point operands and the last set of instructions are to load the various registers in the register file. It is to be noted that for any of the integer operand instructions, the floating-point value is used, the fractional value is discarded and the integer value is used for the operation. If the integer operands are used for the floating-point operand instructions, the integer value is loaded in floating-point format.

9.2.2 Direct Addressing Mode

In direct addressing mode, to access a particular memory location, the page number is to be loaded first in data page pointer (DP) and then the location in that particular page is to be given in the instruction word (LSB 16 bits). This is given as four hexadecimal numbers followed by the symbol after the mnemonic. The start and end values of the page location are 0000h and FFFFh respectively. The example given in Program 9.3 illustrates some instructions that uses direct addressing mode. The on-chip memory available in ′C3X processor is only 2K in size and it is present in page 128 (80h). The CPU and peripheral memory-mapped registers are also present in this page. To access on-chip memory and the memory-mapped registers it is enough to load 80h in DP.

To load the data page value in DP, LDP instruction is used. Followed by the symbol @ 24-bit information (6 hexadecimal integers) is provided. Out of this, the 8 MSB are used to indicate the page number. These 8 MSB are loaded into DP during the execution. The load, store, arithmetic and logic instructions in direct addressing mode are also given in Program 9.3.

Program 9.3 **Example program on direct addressing mode in C3X**

```
Label        Mnemonic             Comments
             .start "direct",     ;the starting address of the section
             0x809850             ;immediate is ;809850h (on-chip memory
                                  ;location)
             .sect "direct"
             .entry
             LDI 10h, R0
             LDI 05h, R1
             LDI 12h, R3
             LDF 10.5, R2
             LDP @800000h         ;the page 128 (80h) is loaded into data page
                                  ;pointer, the 8 MSB of the value given after
                                  ;the symbol @ is used to indicate the page
                                  ;number
             STI R0,@9900h        ;the integer operand available in R0 is
                                  ;stored into the location 9900h of page 80h
             LDI @9905h, R7       ;the integer operand available in location
                                  ;9905h of data page 80h is loaded into R7
             ADDI @9910h, R0      ;the integer operand present in location
                                  ;9910h of data page 80h is added with the
                                  ;content of R0, the result of addition is
                                  ;stored in R0
```

```
ASH @9901h, R3            ;the content of R3 is shifted by the number
                          ;of times the absolute value of integer in the
                          ;location 9901h of data page 80h. If the data
                          ;content is greater than 0, the register
                          ;content R3 is left shifted and it is right
                          ;shifted, if the data content is less than
                          ;zero. When left shifted, the LSBs are zero
                          ;filled and for right shift the MSBs are sign extended
OR @9900h, R2             ;the bit wise logical 0R is performed between
                          ;the content of data memory location 9900h of
                          ;data page 80h and the content of R2
SUBI @9900h, R3           ;the integer operand present in location
                          ;9900h of data page 80h is subtracted from the
                          ;register content of R3 and the result is
                          ;stored in R3
MPYI @9910h, R0           ;the integer operand present in the location
                          ;9910h of data page 80h is multiplied with the
                          ;content of R0 and the result is stored in R0
STF R2,@9902h             ;the floating-point operand present in R2 is
                          ;stored into the data memory location 9902h
                          ;of data page 80h
LDF @9902h, R6            ;the floating-point operand present in the
                          ;data memory location 9902h of data page 80h
                          ;is loaded into R6
ADDF @9902h, R2           ;the floating-point operand present in the
                          ;data memory location 9902h of data page 80h
                          ;is added to the content of register R2 and the
                          ;result is stored in R2
SUBF @9902h, R2           ;the floating-point operand present in the
                          ;data memory location 9902h of data page 80h
                          ;is subtracted from the content of R2 and the
                          ;result is stored in R2
MPYF @9902h, R2           ;the floating-point operand present in the
                          ;data memory location 9902h of data page 80h
                          ;is multiplied with the content of R2 and the
                          ;result is stored in R2
.end
```

9.2.3 Register Addressing Mode

The ′C3X processor CPU has eight extended precession registers R0-R7. These registers are used for the various fixed-point and floating-point operations. In this addressing mode the operations are performed with the content of these eight registers. These registers accept fixed-point and floating-point operands. The example given in Program 9.4 illustrates some instructions which use the register-addressing mode. It is to be noted that the register-addressing mode supports three operand instructions. The two different operands in two different registers can be used for the arithmetic and logic operations. The three-operand instructions are valid for both fixed-point and floating-point operands.

Program 9.4 Example program on register addressing mode in C3X

```
Label          Mnemonic                  Comments
               .start "register",        ;the starting address of the section register
               0x809800                  ;is 809800h
               .sect "register"
               .entry
               LDI 10h, R1
               LDI 20h, R2
               LDF 10.5, R4
               LDF 20.5, R5
               LDI R1, R0                ;fixed-point operand present in R1 is loaded
                                         ;into R0
               LDF R4, R3                ;the floating-point operand present in R4 is
                                         ;loaded into R3
               ADDI R1, R2               ;the fixed-point content present in R1 is
                                         ;added with the content of R2 and the result is
                                         ;stored in R2
               ADDI3 R1, R2, R0          ;the fixed-point contents present in R1 and R2
                                         ;are added and the result is stored in R0
               MPYI R1, R2               ;the fixed-point operand present in R1 is
                                         ;multiplied with the content R2 and the result
                                         ;is stored in R2
               MPYI3 R1, R2, R2          ;the fixed-point operands present in R1 and R2
                                         ;are multiplied and the result is stored in R2
               SUBF R5, R4               ;the floating-point operand present in R5 is
                                         ;subtracted from the content of R4 and the
                                         ;result is stored in R4
               SUBF3 R5, R4, R5          ;the floating-point operand present in R5 is
                                         ;subtracted from the content of R4 and the
                                         ;result is stored in R5
               MPYF R4, R5               ;the floating-point operand present in R4 is
                                         ;multiplied with the content of R5 and the
                                         ;result is stored in R5
               .end
```

9.2.4 Indirect Addressing Mode

In indirect addressing mode the address of the operand is stored in some temporary registers. In ′C3X processors there are two dedicated auxiliary register arithmetic units (AR0 & ARl) with eight auxiliary registers for this addressing mode. This facilitates the processor to generate two data addresses simultaneously. In indirect addressing mode of ′C3X, the AR having address for that instruction is specified in that instruction itself. It also supports the pre- or postdisplacement of eight bits (256 locations) for the address, in the instruction itself using (disp) displacement field. There are two index registers IR0 and IR1. They support the address displacement more than 256 locations and this is called index-addressing mode. The content of index register IR0 alone is used for bit reversed addressing mode.

In indirect addressing mode assembly code syntax, the +/– symbol that comes before the ARn value indicates the predisplacement add/subtract for the address value of the operand. If the symbol ++/–– is present before ARn value, then it is predisplacement add/subtract for the address value of the operand and also the content of AR used in that instruction is modified (add/subtract) by the displacement value. The symbol ++/–– after the ARn value indicates that postdisplacement, the address of the operand is content of AR used in the instruction and the content of AR is modified (add/subtract) by the displacement value. If the index register is used in the displacement filed, the content of index register is used for pre- and postdisplacement for the address of the operand and the content of the AR. The indirect addressing mode supports three operand instructions, in which two source operands can be specified with indirect addressing mode and the destination must be a register (R0-R7). It is important to note that the displacement that can be given in three operand instructions using indirect addressing mode is only 0 and 1. An example on the use of indirect addressing mode is given in Program 9.5.

Program 9.5 Example program on indirect addressing mode in C3X

```
Label          Mnemonic              Comments
               .start “indirect”,
               0x809800
               .sect “indirect”
               .entry
               LDI 10h, R0
               LDI 20h, R1
               LDF 10.5, R2
               LDF 20.5, R3
a              .word 809900h         ;variable ‘a’ is assigned the address value
                                     ;809900h
               LDI @a,AR0            ;value assigned to the variable ‘a’ is loaded
                                     ;into auxiliary register AR0
               STI R0,*AR0           ;integer value in R0 is stored into data
                                     ;memory pointed by auxiliary register AR0.
                                     ;(809900h)
               STI R1, *+AR0(5)      ;the integer content of R1 is stored into the
                                     ;data address location, which is sum of the
                                     ;content of AR0 and the displacement (here it
                                     ;is 5), i.e., 809905h
b              .word 809915h         ;variable ‘b’ is assigned data address value
                                     ;809915h
               LDI @b, AR2           ;the value assigned to the variable ‘b’ is
                                     ;loaded into the auxiliary register AR2,
                                     ;i.e., the address 809915h
               STF R2, *AR2          ;floating-point value in R2 is stored into
                                     ;data memory address pointed by auxiliary
                                     ;register AR2
               STF R3, *-AR2(5)       ;floating-pcint value in R3 is stored into
                                     ;the data memory address (which is the dis-
                                        ;placement value subtracted from the content
                                        ;of auxiliary register AR2, i.e., 809910h)
                ADDI *++AR0(5), R7      ;the integer content present in data memory
                                        ;address location (which is sum of AR0
                                        ;content and the displacement (5)) is added
                                        ;with content of R7. The result is stored in
                                        ;R7 and the content of auxiliary register is
                                        ;incremented by the displacement value given
                                        ;in the instruction
                ADDF *-AR2(5), R6       ;the floating-point operand present in data
                                        ;memory address location (which is the
                                        ;displacement value (5) subtracted from the
                                        ;content of AR2) is added to the content of
                                        ;R6. The result is stored in R6 and the
                                        ;content of AR2 is decremented by the
                                        ;displacement value given in the instruction
                SUBI *AR0-(5), R5       ;the integer operand in the data memory
                                        ;address location, whose value is the content
                                        ;of AR0 is subtracted from the content of R5.
                                        ;The result is stored in R5 and the content of
                                        ;AR0 is decremented by the displacement
                                        ;value (5) given in the instruction
                ADDI3 *AR0++(2),
                *AR2 - (3), R7           ;the integer operands in data
                                        ;memory locations pointed by AR0 and AR2 are
                                        ;added. The result is stored in R7. The
                                        ;content of AR0 is incremented and AR2 is
                                        ;decremented by the value of displacement
                                        ;given in the instruction respectively
ir0             .word 05h               ;the variable ir0 is assigned the integer
                                        ;value 5h
                LDI @IR0, IR0           ;integer value 5h is loaded into index register IR0
                SUBF *AR2-(IR0), R4      ;floating-point operand present in data
                                        ;memory location pointed by auxiliary
                                        ;register AR2 is subtracted from the content
                                        ;of R4. The result is stored in R4 and the
                                        ;content of index register IR0 is subtracted
                                        ;from the content of AR2 and the result is
                                        ;stored in AR2 as new address
                MPYI *AR0++(5),R7       ;integer content present in data memory
                                        ;location pointed by auxiliary register AR0
                                        ;is multiplied with the content of R7. Result
                                        ;is stored in R7 and content of AR0 is
                                        ;incremented by the value of displacement (5)
                                        ;given in the instruction
ir1             .word 10h               ;the variable ir1 is assigned the value 10h
```

```
                LDI @IR1,IR1              ;integer value 10h is loaded into index
                                          ;register ir1
                MPYF *AR2++(IR1),R6       ;the floating-point content of data memory
                                          ;location pointed by auxiliary register AR2
                                          ;is multiplied with the content of R6. The
                                          ;result is stored in R6 and the content of
                                          ;index register is added with content in AR2
                                          ;and the result is stored in AR2
                .end
```

9.2.5 Circular Addressing Mode

In circular addressing mode, the size of the block is to be loaded into the block size register (BK). A data memory address value between start and end address of the block is to be loaded into the AR which is being used for circular addressing. The continuous incrementing of the address value in AR can be performed by some instruction. Once the address value in AR reaches the address value of the end of the block, the start address value is automatically loaded into the AR. The same way decrementing the address will tend to load the end address of the block in AR once the start address of the block is encountered. Program 9.6 gives an example which illustrates the circular addressing mode. The block considered in this example has 6h locations. This is loaded into the register BK and the start address of the block 809900h is loaded into the auxiliary register AR0. The value of AR0 is incremented in the ADDI instruction and once the address in AR0 exceeds 809905h, the content AR0 is loaded with the start address 809900h. In the same way, while decrementing the address in AR0, when it exceeds the start address value 809900h, the end address is loaded into AR0.

Program 9.6 **Example program on circular addressing mode in C3X**

```
                .start “circular”,
                0x809800
                .sect “circular”
                .entry
bk              .word 6h                  ;the variable ‘bk’ is assigned the integer 6h
                LDI @BK, BK               ;the block size register BK is loaded the value
                                          of the block size (6h)
a               .word 809900h             ;the variable ‘a’ is assigned the integer
                                          809900h, the data memory address value
                LDI @a, AR0               ;data memory address value is loaded into
                                          auxiliary register AR0, this is the start
                                          address of the block
                LDI 10h, R0
                LDI 20h, R1               ;the operand address new AR0 value
                ADDI *AR0++(2)%, R2       ;809900h        809902h
                ADDI *AR0++(2)%, R3       ;809902h        809904h
                ADDI *AR0++(2)%, R4       ;809904h        809900h
                ADDI *AR0++(2)%, R5       ;809900h        809902h
```

```
                    ADDI *AR0—(2)%, R6          ;809902h       809900h
                    SUBI *AR0— (1)%, R7         ;809900h       809905h
                    .end
```

9.2.6 Parallel Instructions

The ′C3X processor architecture has four buses in CPU (CPU1, CPU2, REG1 and REG2 buses) and two address generation units (ARU1 and ARU2). This facilitates the CPU to get four operands and can perform two operations in parallel in CPU. Program 9.7 gives an example which illustrates some of the instructions that can be executed in parallel in ′C3X processors. In parallel instructions, only register and indirect addressing modes are used. The three-operand instructions can be used in parallel instructions. If only one three-operand instruction is used all the eight extended precision registers (R0-R7) can be used for destination. If two three-operand instructions are used there will be four source operands and two destination operands. Out of the four source operands, two of them can be register addressed and two of them can be indirect addressed. As far as the two destination operands are concerned, for the first three-operand instruction only registers R0 and R1 can be used and for the second three-operand instruction only registers R2 and R3 can be used.

Program 9.7 Example program on parallel instructions in C3X

```
                    .start “parallel”,
                    0x809800
                    .sect “parallel”
                    .entry
                    Ldi 10h, r1
                    Ldi 10h, r2
                    Ldf 10.5, r4
                    Ldf 20.5, r5
a                   .word 809900h               ;variable ‘a’ is assigned integer value 809900h
                    Ldi @a, ar0                 ;auxiliary register AR0 is loaded the data
                                                memory address value 809900h
b                   .word 809905h               ;variable ‘b’ is assigned integer value 809905h
                    Ldi @b, ar1                 ;the auxiliary register AR1 is loaded the
                                                data memory address value 809905h
                    Sti r0, *ar0++
                    || sti r1, *ar1++           ;the integer content in R0 and R1 is stored into the
                                                data memory location pointed by auxiliary register
                                                AR0 and AR1 respectively. The content of AR0 and AR1
                                                is incremented by the value of displacement (since
                                                the displacement value is not given the default value
                                                (1) will be taken for the increment of data memory
                                                address)
                    ldi *++ar0, r3
```

```
|| ldi *-ar1, r6              ;the integer content present in the data memory
                              location (i.e. the content of AR0 added with the
                              displacement (here it is 1)) is loaded into the register
                              R3 and the content of ARO is incremented by the
                              value of displacement. By the same time the content
                              of data memory location (i.e. the displacement (here
                              it is 1) is subtracted from the content of AR1) is
                              loaded into R6 and also the content of AR1 is
                              decremented by the value of displacement
addi3 *ar0, r2, r3            ;the integer data present in the data memory
|| sti r4, *ar1               location pointed by auxiliary AR0 is added with the
                              content of R2 and the result is stored in R3. By the
                              same time, the content of R4 is stored in the data
                              memory address location pointed by AR1
mpyf3 *ar1,r5,r6              ;the floating-point data present in the data memory
|| stf r2,*ar2++(1)           location pointed by auxiliary register AR1 is
                              multiplied with the register content of R5 and the
                              result is stored in R6. By the same time the
                              floating-point operand present in R2 is stored into the
                              data memory location pointed by AR2 and the
                              content of AR2 is incremented by displacement value
subi3 r0, *ar1, r3            ;the integer content of R0 is subtracted from the
|| sti r5, *ar1-(1)           content of data memory location pointed by AR1 and
                              the result is stored in R3, by the same time the
                              content of R5 is stored in the data memory location
                              pointed by AR1 and the content of AR1 is
                              decremented  by the displacement value
mpyi3 r1, r2, r0              ;the integer content present in R1 and R2 are
|| addi3 *ar1, *ar0, r3       multiplied, the result is stored in R0 and at the same
                              time the integer operands present in data memory
                              locations pointed by auxiliary register AR1 and AR0
                              are added and the result is stored in R3
.end
```

GENERATION AND FINDING THE SUM OF SERIES 9.3

The two examples given in Programs 9.8 and 9.9 illustrate the generation of some sequences and finding their sum. The number of values to be calculated in the sequence is assigned for the variable ‘n’. This value can be loaded in any one of the extended precision registers R0-R7, used as counter and decremented. The sequence values generated are stored in the data memory; the starting address from which the sequence is to be stored is assigned to the variable ‘*a*’. Each time the sequence value is generated, the sum is calculated and accumulated. Once the generation of sequence for the given value of ‘*n*’ is completed, the accumulated sum is stored in the data memory.

(a) Generation and finding the sum of series 1+2 + 3 + 4 + ··· + n

Program 9.8 Example C3X program for sum of *n* integers

```
                 .start "series"
                 0x809900
                 .sect "series"
n                .word 10h           ;the number of the sequence values to be
                                     ;computed is assigned for 'n'
                 .entry
a                .word 809a00h       ;the starting address from which the gener-
                                     ;ated sequence is to be stored is assigned
                 LDI @a, AR0         ;the address is loaded into auxiliary
                                     ;register AR0
                 LDI @n, R2          ;the number of sequence values is loaded
                                     ;in R2
loop             ADDI 1h, R0         ;the integer 1h is added to R0
                 ADDI3 R0, R1, R1    ;the new value of the sequence generated
                                     ;in R0 is added with the previous sum present
                                     ;in R1 and accumulated in R1 itself
                 STI R0, *AR0++(1)   ;the new value of the sequence generated
                                     ;in R0 is stored in the data memory location
                                     ;pointed by AR0 and the content of AR0 is
                                     ;incremented by one
                 SUBI 1h, R2         ;the content of R2 is used as counter, it is
                                     ;decremented by one for each sequence value
                                     ;being generated, added and accumulated
                 STI R1, *AR0        ;the sum of the sequence present in R1 is
                                     ;stored in data memory address location
                                     ;pointed by AR0
                 .end
```

(b) Generation and finding sum of series $1^2 + 2^2 + 3^2 + 4^2 + 5^2 + \cdots + n^2$

Program 9.9 Example C3X program for sum of squares of *n* integers

```
                 .start "series2"
                 0x809800
                 .sect "series2"
                 .entry
n                .word 10h
                 .entry
a                .word 809a00h
                 LDI @a, AR0
```

```
                   LDI @n, R2
loop               ADDI 1h, R0
                   MPYI3 R0, R0, R1         ;the content of R0 is multiplied with the
                                            ;content of R0 and stored in R1, i.e. the
                                            ;square of the integer is computed
                   ADDI3 Rl, R3, R3         ;the content of R1 (squared value) is added
                                            ;with the content of R3 and the result is
                                            ;stored in R3. This is done for finding the
                                            ;sum of the sequence
                   STI R1, *AR0++(1)        ;the sequence values present in R1 are stored
                                            ;in data memory location pointed by AR0 and
                                            ;the content of AR0 is incremented by a value one
                   SUBI 1h, R2 BNZ loop
                   STI R3, *AR0             ;the sum of the sequence values present in R3
                                            ;is stored in the address location pointed by AR0
                   .end
```

(c) Fibonacci Series Generation

The generation of the Fibonacci series numbers is explained in the following example. The first two numbers are assumed; the next number in the series is the sum of the previous two numbers. For example, $x(0) = 0$, $x(1) = 1$, then $x(i) = x(i - 1) + x(i - 2)$ where $i > 1$. The initial two values of the sequence are assigned to the variables '*b*' and '*c*' and they are stored in two registers R0 and Rl respectively. The number of sequence values to be computed (n) and the data memory address (a) to store the computed sequence are stored in register R7 and AR0 respectively. The sequence value is computed in register R2 and it is stored in the data memory location pointed by AR0, each time the loop is executed. The instructions for generating the series are given in Program 9.10.

Program 9.10 **C3X program for generation of Fibonacci series**

```
                   .start "series3",
                   0x809900
                   .sect "series3"
n                  .word 8h
a                  .word 809950h
b                  .word 0h
c                  .word 1h
                   .entry
                   LDI @n, R7
                   LDI @b, R0
                   LDI @c, R1
                   LDI @a, AR0
                   STI R0, *AR0++
                   STI R1, *AR0++
loop                ADDI3 R0, R1, R2
                   STI R2, *AR0++
                   LDI R1, R0
                   LDI R2, R1
                   SUBI 1h, R7
                   BNZ loop
                   .end
```

CONVOLUTION OF TWO SEQUENCES 9.4

In DSP applications, one of the operations, which is commonly used, is convolution. Programs 9.11 and 9.12 give two examples which illustrate the method used for convolution of fixed-point and floating-point sequences. The assembly program steps for the both fixed-point and floating-point convolution are same. The difference in fixed-point and floating-point sequence convolution is that in fixed-point convolution the fixed-point assembly instructions are used and in floating point convolution the floating-point assembly instructions are used.

The two sequences, which are to be convolved, are stored in on-chip memory location in different sectors (sectors *x* and *y*). The number of sequence values in these sectors is '*n*' and '*m*' respectively. The data memory address of one sequence is incremented and that of the other one is decremented for the convolution operation. The sequence which is incremented is not padded with zeros (*n* values), whereas the sequence which decrements for convolution is padded with zeros (*m* values). The number of zeros to be padded for the sequence having *m* values is $n - 1$ at the beginning and at the end.

The number of values in the resultant sequences is $n + m - 1$ values. The basic function needed for the convolution operation is multiply and accumulate. The ′C3X processor supports three operand multiply instruction. Using this instruction, two data sequence values from memory are fetched, multiplied and stored in registers R0–R7. The add instruction followed by the multiply instruction can be used for the convolution operation. For finding each convolution sum, the multiply and accumulate of the sequence values is to be repeated. The number of times it is to be repeated is '*n*'. The starting address of the two sequences and the starting address in which the convolved sequence is stored are stored in ARs. The number of sequence values in the convolved sequence and the number of times the multiply and accumulate operation is to be performed for each convolution sum are stored in extended precession registers.

(a) Convolution of Sequence (Integer Values)

Program 9.11 **C3X program for convolution of two integer sequences**

```
                    .start “x”, 0x809900        ;starting address for segment ‘x’ is 809900h
                                                ;(on-chip memory location)
                    .sect “x”
                    .word 3h, 4h, 5h            ;integer values of the sequence ‘x’ are
                                                ;stored from 809900h
                    .start “y”, 0x809910        ;the starting address for segment ‘y’ is
                                                ;809910h
                    .sect “y”
                    .word 0h, 0h, 1h, 2h, 3h, 2h, 1h, 0h, 0h
                                        ;integer values of sequence 'y' are stored
                                        ;from address 809910h
                    .start "convolution",  ;the starting address for the segment
                    0x809800            ;'convolution' is 809800h
                    .sect "convolution"
                    .entry
xs                  .word 809900h       ;staring address of the sequence 'x' is
                                        ;assigned to the variable xs
ye                  .word 809912h       ;the end address of the sequence 'y' is
                                        ;assigned to the variable ye
zs                  .word 809950h       ;the starting address in which the convolu-
                                        ;tion result is to be stored is assigned to
                                        ;variable zs
n                   .word 7h             ;the number of sequence values in the result-
                                        ;ant sequence is assigned to variable n
rpt                 .word 2h            ;the number of times the multiply and accumu-
                                        ;late operation is to be repeated for each
                                        ;convolution sum is assign to the variable rpt
                    LDI @ye, AR1        ;end address of sequence y is loaded in AR1
                    LDI @zs, AR2        ;the start address from which the result of
                                        ;the convolved sequences to be stored is
                                        ;loaded in AR2
                    LDI @n, R6          ;the number of sequence values in resultant
                                        ;sequence is loaded in R6
loop1               LDI @xs, AR0        ;start address of the sequence x is loaded in
                                        ;auxiliary register AR0
                    LDI 0h, R1          ;the register content R1 is cleared
                    LDI @rpt, RC        ;the number of times the multiply and accumu-
                                        ;late operation is to be repeated is loaded
                                        ;into repeat counter register
                    RPTB loop           ;the block repeat operation starts, the
                                        ;block 'loop' is repeated until the repeat
                                        ;counter register decrements to zero
                    MPYI3 *AR0++,       ;the integer operands in data memory address
                    *AR1-, R0           ;location pointed by AR0 and AR1 are multi-
                                        ;plied, stored in register R1, the content of
                                        ;AR0 is incremented and AR1 is decrement by
                                        ;one
                    ADDI R0, R1         ;content of R0 is added with the content of R1
                                        ;and the result is stored in R1
loop                NOP
                    STI R1, *AR2++      ;the convolution sum present in R1
                    ADDI 4h, AR1        ;the integer 4h is added to the content AR1,
                                        ;this is done to point the next value of
                                        ;sequence 'x'
```

```
           SUBI 1h, R6                   ;the content of R6 is decremented by one, this
                                         ;done each time one convolution sum is com-
                                         ;puted loaded into repeat counter register
           BNZ loop1                     ;branch to label address loop1 if the content
                                         ;of R6 is not zero, if zero go to next instruction
           .end
```

(b) Convolution of Sequences (Floating-point Values)

Program 9.12 C3X program for convolution of 2 real number sequences

```
           .start "x", 0x809900
           .sect "x"
           .float 3.1, 4.1, 5.1          ;the floating-point value of the sequence
                                         ;'x' stored from the address 809900h
           .start "y", 0x809910
           .sect "y"
           .float 0.0, 0.0, 1.1,         ;the floating-point sequence y' is stored
           2.1, 3.1, 2.1, 1.1, 0.0, 0.0  ;from the address 809910h
           .start "convolution",
           0x809800
           .sect "convolution"
           .entry
xs         .word 809900h
ye         .word 809912h
zs         .word 809950h
n          .word 7h
Rpt        .word 2h
           LDI @ye, AR1
           LDI @zs, AR2
           LDI @n, R6
loop1      LDI @xs, AR0
           LDI 0h, R1
           LDI @rpt, RC
           RPTB loop
           MPYF3 *AR0++, *AR1-, R0       ;the floating-point operands in data memory
                                         ;address location pointed by AR0 and AR1 are
                                         ;multiplied, stored in R1, the content of AR0
                                         ;is incremented and AR1 is the floating-point
                                         ;content in R0 is R1 and the result is stored in
                                         ;R1
           ADDF R0, R1
loop       NOP
           STF R1, *AR2++                ;store the floating-point convolution sum in
                                         ;R1 into the data memory location pointed by
                                         ;AR2 and the content of AR2 is incremented by
```

```
                              ;one
        ADDI 4h, AR1
        SUBI 1h, R6
        BNZ loop1
        .end
```

PROCESSING REAL TIME SIGNALS WITH C3X KIT 9.5

The real time signals are commonly in analog form. For processing these signals, they are converted into digital signals and stored in memory. The processing of these real time digital samples is carried out in programmable DSPs. After processing, the digital signals are converted back to analog signals. This requires a analog to digital converter (A/D) and a digital to analog converter (D/A) which work either at same or different sampling and conversion rates respectively.

The C3X starter kit is provided with TLC320C40, an analog interface circuit IC (AIC), 14 bit, for audio applications. This programmable AIC has both A/D and D/A converters. The bandwidth of the IC is from 200 Hz to 19.2 kHz. The operation and programming of this AIC is given in Section 6.4.3.

This AIC is interfaced with the standard serial port of C3X and the clock signal needed for TLC320C40 is generated from the on-chip timer of C3X processor. To have the real time signal to be stored in the on-chip memory of the DSP, the AIC initialisation routine is to be executed first. The AIC initialisation routine consists of the following steps:

(a) the on-chip timer initialisation
(b) the on-chip serial port initialisation
(c) the AIC reset
(d) programming the control words of AIC
(e) interrupt processing

In this section, first the details of on-chip timer and serial port are discussed, and then programming details of AIC are given.

9.5.1 TMS320C3X On-chip Timer

The two on-chip timers present in C3X are programmable 32-bit timer. This timer is used to generate clock signals for the external devices such as A/D and D/A converters or it can generate interrupt signal for DMA transfer. The C3X timer can be operated in two signaling modes, one with the internal clock and the other with the external clock. Each timer has an I/O pin that can be programmed as an input clock to timer, an output clock signal, or a general purpose I/O pin.

The functional block diagram of the timer is shown in Fig. 9.5. The three memory-mapped registers present in each timer are used to determine the function. The three memory-mapped timer registers are global control register (GCR), period register (PR) and counter register (CR).

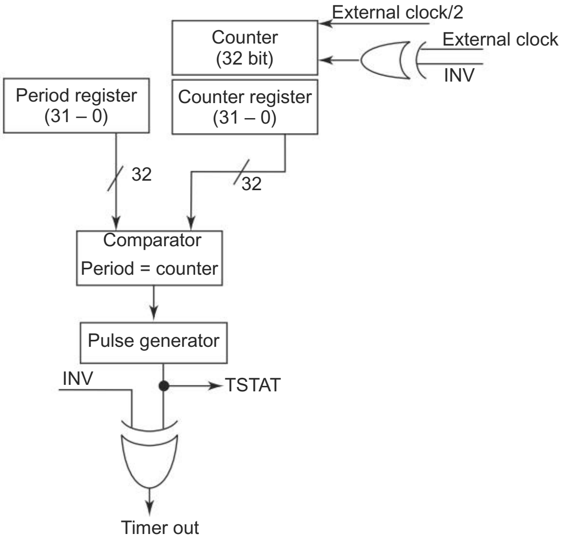


Fig. 9.5 *The functional block diagram of the C3X timer*

The timer global control register determines the operating mode of the timer, monitors the timer status and controls the function of the I/O pin of the timer. The period register specifies the timer's signaling frequency. The counter register contains the current value of the incrementing counter. The on-chip memory address locations of the timer registers are given in Table 9.1. For the required mode of timer operation, the control word for the global control register, and for the required frequency, the count value of the period register are stored in the respective memory-mapped address locations.

Table 9.1 *Memory-mapped address locations of timer registers*

Register name	*On-chip memory address of Timer 0*	*On-chip memory address of Timer 1*
Global control register	808020h	808030h
Timer counter register	808024h	808034h
Timer period register	808028h	808038h

9.5.2 Timer Global Control Register

The timer global control register is a 32-bit register that contains the global and port control bits for the timer. The bits 3–0 are the port control bits; bits 11–6 are the timer global control bits. Fig. 9.6 shows the format of timer global control register. The various bits of the timer global control register and its functions are given in Table 9.2. At reset all bits are set to 0 except the DATIN bit.

31 – 12	11	10	9	8	7	6	5 4	3	2	1	0
XX	TSTAT	INV	CLK SRC	C/$\overline{P}$	$\overline{HLD}$	GO	XX	DAT IN	DAT OUT	I/$\overline{O}$	FUNC
	R	R/W	R/W	R/W	R/W	R/W		R	R/W	R/W	R/W

Fig. 9.6 *Timer global control register*

9.5.3 Timer Operation and Timer Modes

The timer can operate both for internal and external clock signals. Depending upon the timer output frequency, the divide ratio of the input clock is loaded into the timer period register. For each input clock pulse, the timer counter (32-bits) increments. The increment operation can be programmed for leading edge or the falling edge of the input clock. The counter register (32-bits) holds the current count value of the counter. The period register content and the counter register content are compared in the comparator. If the values are equal, an internal interrupt is generated, the pulse generator generates a pulse, the counter is reset and starts incrementing and this action is repeated. The pulse generator can generate two types of external clock signals, the pulse or the clock signal.

Table 9.2 *Global control register bits and its functions*

Bit Name	*Function*
FUNC	Function controls the function of TCLK If FUNC = 0, TCLK is configured as a general-purpose digital I/O port If FUNC = 1, TCLK is configured as a tinner pin

(Contd.)

Table 9.2 *(Contd.)*

$I/\overline{O}$	Input/output If FUNC = 0 and CLKSRC = 0, TCLK is configured as a general-purpose I/O pin If I/O = 0, TCLK is configured as a general-purpose input pin If I/O = 1, TCLK is configured as a general-purpose output pin
DATOUT	Data output Drives TCLK when the ′C3X is in I/O port mode. You can use DAT-OUT as an input to the timer
DATIN	Data input on TCLK or DATOUT. A write has no effect
$G\overline{O}$	Go resets and starts the timer counter When GO = 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The GO bit is cleared on the same rising edge GO = 0 has no effect on the timer
$\overline{HLD}$	Counter hold signal When this bit is 0, the counter is disabled and held in its current state. If the timer is driving TCLK, the state of TCLK is also held. The internal divide-by-2 counter is also held so that the counter can continue where it left off when $\overline{HLD}$ is set to 1. You can read and modify the timer registers while the timer is being held. RESET has priority over $\overline{HLD}$. The effect of writing to GO and HOLD is shown below GO $\overline{HLD}$ Result 0 0 All timer operations are held. No reset is performed 0 1 Timer proceeds from state before write 1 0 All timer operations are held, including zeroing of the counter. The GO bit is not cleared until the timer is taken out of hold 1 1 Timer resets and starts
$C/\overline{P}$	Clock/pulse mode control When C/P = 1, clock mode is chosen, and the signaling of the TSTAT flag and external output has a 50% duty cycle When C/P = 0, the status flag and external output will be active for one H1 cycle during each timer period
CLKSRC	Clock source. This bit specifies the source of the timer clock When CLKSRC = 1, an internal clock with a frequency equal to one-half of the H1 frequency is used to increment the counter. The INV bit has no effect on the internal clock source When CLKSRC = 0, you can use an external signal from the TCLK pin to increment the counter. The external clock is synchronised internally, thus allowing external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency. This is less than f(H1)/2
INV	Inverter control bit If an external clock source is used and INV = 1, the external clock is inverted as it goes into the counter. If the output of the pulse generator is routed to TCLK and INV = 1, the output is inverted before it goes to TCLK. If INV = 0, no inversion is performed on the input or output of the timer. The INV bit has no effect, regardless of its value, when TCLK is used in I/O port mode
TSTAT	Timer status bit. This bit indicates the status of the timer. It tracks the output of the uninverted TCLK pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect

The timer can receive its input and send its output in several different modes, depending upon the setting of CLKSRC, FUNC and I/$\overline{\text{O}}$. The four timer modes of operation are defined in Table 9.3. The timer output frequency depends on the input frequency and the count value in the period register. The following equations can be used to calculate the output frequency of the timer either in clock mode or in pulse mode.

f(pulse mode) = *f*(timer input clock)/period register

f(clock mode) = *f*(timer input clock)/(2X period register)

Table 9.3 *Timer modes*

CLKSRC	*FUNC*	*Timer mode*
1	0	The timer input comes from the internal clock. The internal clock is not affected by the INV bit in the global control register. In this mode, TCLK is connected to the I/O port control, and TCLK can be used as a general-purpose I/O pin
1	1	The timer input comes from the internal clock, and the timer output goes to TCLK. This value can be inverted using INV, and you can read in DATIN the value output on TCLK
0	0	The timer is driven according to the status of the I/O bit If I/O = 0, the timer input comes from TCLK If I/O = 1, TCLK is an output pin
0	1	TCLK drives the timer If INV = 0, all 0-to-1 transitions of TCLK increment the counter If INV = 1, all 1-to-0 transitions of TCLK increment the counter

9.5.4 Timer Initialisation

The on-chip timer can be used to generate clock signals for external devices. By programming the global control register, timer period register and timer counter register, the required frequency can be obtained from the timer output pin of the timer. In C3X starter kit the timer input is 25 MHz and it is from internal clock. The period register is prograrnmed to divide this input clock by a factor of two. The initialisation routine for the timer is given in Program 9.13. First the on-chip timer memory-map register address values and the timer period register count value are set to variables. Then the control word for the global control register and count value for the period register are loaded into respective registers using these memory-map address values.

Program 9.13 **Initialisation routine for the C3X timer**

```
Tgcr          .set 0x808020          ;memory-map address value for the timer global
                                     ;control register
Tcount        .set 0x808024          ;memory-map address value for the counter
                                     ;register
Tprd          .set 0x808028          ;memory-map address value for the timer period
                                     ;register
```

```
Tperiod      .set 2              ;the count value for the period register
aic          Ldp Tgcr            ;the starting address of the memory-map regis-
                                 ;ter is loaded into data page pointer
             Ldi 0, R0           ;integer 0 is loaded into R0
             Sti R0, @Tgcr       ;0 is stored into timer global control register

             Sti R0, @Tcount     ;zero is stored into timer counter register
             Ldi Tperiod, R0     ;the count value (2) for the period register is
                                 ;loaded into period register
             Sti R0, @Tprd       ;the count value is stored into period register
             Ldi 0x2C1, R0       ;the control word 2C1 for the global control
                                 ;register is loaded into R0
             Sti R0, @Tgcr       ;the control word is stored into timer global
                                 ;control register
```

SERIAL PORT 9.6

The C3X processor has two independent bidirectional serial ports and they are identical. A set of eight control registers is available for each serial port. These registers are memory-mapped registers and they are global-control register, two control registers for the six serial I/O pins, three receive/transmit timer registers, data transmit register and data receive register. The serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word simultaneously in both-directions. The clock signal for each serial port can be fed internally, via the serial port timer or externally, via a supplied clock.

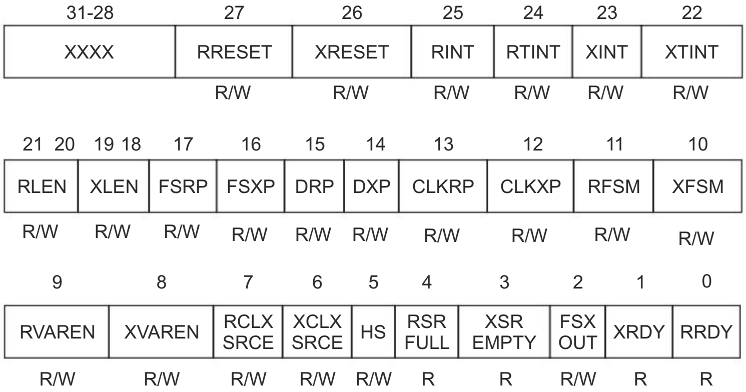
The global control register controls the global functions of the serial port and determines the serial-port operating mode. Two port-control registers control the functions of the six serial-port pins. The transmit buffer contains the next complete word to be transmitted. The receive buffer contains the last complete word received. Three additional registers are associated with the transmit/receive sections of the serial-port timer. The operation of the serial port is discussed in detail in section 7.7.2. The memory-mapped address values of the serial port registers are given in Table 9.4.

Table 9.4 *Memory-mapped address values for serial port registers*

Register name	*On-chip memory address of Timer 0*	*On-chip memory address of Timer 1*
Serial port global control	808040h	808050h
FSX/DX/CLKX port control	808042h	808052h
FSR/DR/CLKR port control	808043h	808053h
R/X timer control	808044h	808054h
R/X timer counter	808045h	808055h
R/X timer period	808046h	808056h
Data transmit	808048h	808058h
Data receive	80804Ch	80805Ch

9.6.1 Serial Port Global Control Register

The serial port global control register is a 32-bit register. It contains the global control bits for the serial port. The format of the global control register is shown in Fig. 9.7. The various bits and their functions are given in Table 9.5.



31-28	27	26	25	24	23	22
XXXX	RRESET	XRESET	RINT	RTINT	XINT	XTINT
	R/W	R/W	R/W	R/W	R/W	R/W

21 20	19 18	17	16	15	14	13	12	11	10
RLEN	XLEN	FSRP	FSXP	DRP	DXP	CLKRP	CLKXP	RFSM	XFSM
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

9	8	7	6	5	4	3	2	1	0
RVAREN	XVAREN	RCLX SRCE	XCLX SRCE	HS	RSR FULL	XSR EMPTY	FSX OUT	XRDY	RRDY
R/W	R/W	R/W	R/W	R/W	R	R	R/W	R	R

Fig. 9.7 *Serial port global control register*

Table 9.5 *Bit Functions of Serial-port Global-control Register*

Bit name	*Function*
RRDY	Receive ready flag. If RRDY = 1, the receive buffer has new data and is ready to be read. A three H1/H3 cycle delay occurs from the loading of DRR to RRDY = 1. The rising edge of this signal sets RINT. If RRDY = 0, the receive buffer does not have new data since the last read. RRDY = 0 at reset and after the receive buffer is read
XRDY	Transmit ready flag. If XRDY = 1, the transmit buffer has written the last bit of data to the shifter and is ready for a new word. A three H1/H3 cycle delay occurs from the loading of the transmit shifter until XRDY is set to 1. The rising edge of this signal sets XINT. If XRDY =0, the transmit buffer has not written the last bit of data to the transmit shifter and is not ready for a new word
FSXOUT	Transmit frame sync configuration FSXOUT = 0 configures the FSX pin as an input FSXOUT = 1 configures the FSX pin as an output
XSREMPTY	Transmit-shift register empty flag If XSREMPTY = 0, the transmit-shift register is empty If XSREMPTY = 1, the transmit-shift register is not empty Reset or XRESET causes this bit to = 0
RSRFULL	Receive-shift register full flag If RSRFULL = 1, an overrun of the receiver has occurred. In continuous mode, RSRFULL is set to 1 when both RSR and DRR are full. In noncontinuous mode, RSRFULL is set to 1 when RSR and DRR are full and a new FSR is received. A read causes this bit to be set to 0. This bit can be set to 0 only by a system reset, a serial-port receive reset (RRESET = 1) or a read. When the receiver tries to set RSRFULL to 1 at the same time that the global register is read, the receiver dominates, and RSRFULL is set to 1. If RSRFULL = 0, no overrun of the receiver has occurred

(Contd.)

Table 9.5 *(Contd.)*

HS	Handshake. If HS = 1, the handshake mode is enabled. If HS = 0, the handshake mode is disabled.
XCLK SRCE	Transmit clock source If XCLK SRCE = 1, the internal transmit clock is used If XCLK SRCE = 0, the external transmit clock is used
RCLK SRCE	Receive clock source If RCLK SRCE = 1, the internal receive clock is used If RCLK SRCE = 0, the external receive clock is used
XVAREN	Transmit data rate mode. Specifies a fixed or variable data rate mode when transmitting. With a fixed data rate, FSX is active for at least one XCLK cycle and then goes inactive before transmission begins. With variable data rate, FSX is active while all bits are being transmitted. When you use an external FSX and variable data rate signaling, the DX pin is driven by the transmitter when FSX is held active or when a word is being shifted out
RVAREN	Receive data rate mode Specifies a fixed or variable data rate mode when receiving. If RVAREN = 0 (fixed data rate), FSX is active for at least one RCLK cycle and then goes inactive before reception begins. If RVAREN = 1 (controlled data rate), FSX is active while all bits are being received
XFSM	Transmit frame sync mode Configures the port for continuous mode operation or standard mode operation. If XFSM = 1 (continuous mode), only the first word of a block generates a sync pulse, and the rest are transmitted continuously to the end of the block. If XFSM = 0 (standard mode), each word has an associated sync pulse
RFSM	Receive frame sync mode Configures the port for continuous mode operation or standard mode operation. If RFSM = 1 (continuous mode), only the first word of a block generates a sync pulse, and the rest are received continuously to the end of the block. If RFSM = 0 (standard mode), each word received has an associated sync pulse
CLKXP	CLKX polarity. If CLKXP = 0, CLKX is active high If CLKXP = 1, CLKX is active low.
CLKRP	CLKR polarity. If CLKRP = 0, CLKR is active (high) If CLKRP = 1, CLKR is active (low)
DXP	DX polarity. If DXP = 0, DX is active (high) If DXP = 1, DX is active (low)
DRP	DR polarity. If DRP = 0, DR is active (high) If DRP = 1, DR is active (low)
FSXP	FSX polarity, if FSXP = 0, FSX is active (high) If FSXP=1, FSX is active (low)
FSRP	FSR polarity. If FSRP = 0, FSR is active (high) If FSRP = 1, FSR is active (low)

(Contd.)

Table 9.5 *(Contd.)*

XLEN	Transmit word length. These two bits define the word length of serial data transmitted. All data is assumed to be right justified in the transmit buffer when fewer than 32 bits are specified 0 0—8 bits 1 0—24 bits 0 1—16 bits 1 1—32 bits
RLEN	Receive word length. These two bits define the word length of serial data received. All data is right justified in the receive buffer 0 0—8 bits 1 0—24 bits 0 1—16 bits 1 1—32 bits
XTINT	Transmit timer interrupt enable If XTINT = 0, the transmit timer interrupt is disabled If XTINT = 1, the transmit timer interrupt is enabled
XINT	Transmit interrupt enable If XINT = 0, the transmit interrupt is disabled If XINT = 1, the transmit interrupt is enabled
RTINT	Receive timer interrupt enable If RTINT = 0, the receive timer interrupt is disabled If RTINT = 1, the receive timer interrupt is enabled
RINT	Receive interrupt enable If RINT = 0, the receive interrupt is disabled If RINT = 1, the receive interrupt is enabled
XRESET	Transmit reset. If XRESET = 0, the transmit side of the serial port is reset. To take the transmit side of the serial port out of reset, set XRESET to 1. Do not set XRESET to 1 until at least three cycles after RESET go inactive. This applies only to system reset. Setting XRESET to 0 does not change the contents of any of the serial-port control registers. It places the transmitter in a state corresponding to the beginning of a frame of data. Resetting the transmitter generates a transmit interrupt. Reset this bit during the time the mode of the transmitter is set. You can toggle XFSM without resetting the global control register
RRESET	Receive reset. If RRESET = 0, the receive side of the serial port is reset. To take the receive side of the serial port out of reset, set RRESET to 1. Do not set RRESET to 1 until at least three cycles after RESET go inactive. This applies only to system reset. Setting RRESET to 0 does not change the contents of any of the serial-port control registers. It places the receiver in a state corresponding to the beginning of a frame of data. Reset this bit at the same time that the mode of the receiver is set. You can toggle without resetting the global control register

9.6.2 Serial Port Signal Control Register

The serial port needs frame synchronisation and clock signals for transmission and reception. These signals are FSX and FSR, CLKX and CLKR respectively. Similarly, there are two pins for data transmission and reception, DX and DR respectively. The two port-control registers control the functions of these six pins. The format of these registers and functionality are the same; the difference is that one register controls the functions of transmission and the other controls the functions of reception. The format of the port-control register is shown in Fig. 9.8. The bit name and its functionality are given in Table 9.6.

31-12	11	10	9	8	7
XXX ... XXX	FSX DATIN	FSX DATOUT	FSX I/O	FSX FUNC	DX DATIN
	R	R/W	R/W	R/W	R

6	5	4	3	2	1	0
DX DATOUT	DX I/O	DX FUNC	CLKX DATIN	CLKX DATOUT	CLKX I/O	CLKX FUNC
R/W	R/W	R/W	R	R/W	R/W	R/W

Fig. 9.8 *FSX/DX/CLKXport-control register*

Table 9.6 *Bit functions of FSX/DX/CLKX port-control register*

Bit name	*Function*
CLKX FUNC	Clock transmit function. Controls the function of CLKX If CLKX FUNC = 0, CLKX is configured as a general-purpose digital I/O port. If CLKX FUNC = 1, CLKX is configured as a serial-port pin
CLKX I/O	Clock transmit input/output mode If CLKX I/O = 0, CLKX is configured as a general-purpose input pin. If CLKX I/O = 1, CLKX is configured as a general-purpose output pin
CLKX DATOUT	Clock transmit data output Data output on CLKX when configured as general-purpose output
CLKX DATIN	Clock transmit data input Data input on CLKX when configured as general-purpose input A write has no effect
DX FUNC	DX function. DXFUNC controls the function of DX If DXFUNC = 0, DX is configured as a general-purpose digital I/O port. If DXFUNC = 1, DX is configured as a serial-port pin
DX I/O	DX input/output mode If DX I/O = 0, DX is configured as a general-purpose input pin If DX I/O = 1, DX is configured as a general-purpose output pin
DX DATOUT	DX data output. Data output on DX when configured as general-purpose output.
DX DATIN	DX data input. Data input on DX when configured as general-purpose input. A write has no effect
FSX FUNC	FSX function. Controls the function of FSX If FSX FUNC = 0, FSX is configured as a general-purpose digital I/O port. If FSX FUNC = 1, FSX is configured as a serial-port pin
FSX I/O	FSX input/output mode If FSX I/O = 0, FSX is configured as a general-purpose input pin If FSX I/O = 1, FSX is configured as a general-purpose output pin
FSX DATOUT	FSX data output. Data output on FSX when configured as general-purpose output
FSX DATIN	FSX data input. Data input on FSX when configured as general-purpose input. A write has no effect

9.6.3 Serial Port Initialisation

In the C3X starter kit, the TLC320C40 audio codec IC is connected to the serial port0 of C3X DSP. Through the serial port, the digital samples obtained from the AIC A/D converter are received; also after processing, the samples are transmitted to the D/A converter in the AIC. To have the serial communication between the AIC and the DSP, it is necessary to initialise the serial-port control register. The routine used to initialise the serial-port global control register, FSX/DX/CLKX port control register and FSR/DR/CLKR port control register is given in Program 9.14. The memory-mapped address values and the control word for the registers are assigned to some variables. Then the control words are loaded into the respective registers using the memory-mapped address.

Program 9.14 Program for initialisation of C3X serial port registers

```
Sgcr          .set 0x808040         ;memory-map address value for the serial port
                                    ;global control register
Sxpctrl       .set 0x808042         ;memory-map address value for the FSX/DX/CLKX
                                    ;port control register
Srpctrl       .set 0x808043         ;memory-map address value for the FSR/DR/CLKR
                                    ;port control register
Sxdata        .set 0x808048         ;memory-map address value for the data transmit
                                    ;register
Srdata        .set 0x80804C         ;memory-map address value for the data receive
                                    ;register
Sgcrval       .word0x0E970300       ;the control word for serial port global
                                    ;control register
Sxpctrlval    .word0x00000111       ;the control word
                                    ;for FSX/DX/CLKX port control register
Srpctrlval    .word0x00000111       ;the control word for FSX/DX/CLKX port control
                                    ;register
ldi           @Sxpctrlval, R0       ;the control word for FSX/DX/CLKX port
                                    ;control register is loaded into R0
sti           R0,@Sxpctrl           ;the control word is stored into FSX/DX/CLKX
                                    ;port control register
ldi           @Srpctrlval, R0       ;the control word for FSR/DR/CLKR port control
                                    ;register is loaded into R0
sti           R0,@Srpctrl           ;the control word is stored into
                                    ;FSR/DR/CLKR port control register
ldi           0,R0                  ;integer 0 is loaded into R0
sti           R0,@Sxdata            ;zero is stored into the data transmit
                                    ;register
ldi           @Sgcrval, R0          ;the control word for the serial port control
                                    ;register is loaded into R0
sti           R0, @Sgcr             ;the control word is stored Into the serial
                                    ;port control register
```

9.6.4 Analog Interface Circuit Programming

The AIC contains programmable registers such as TA register, RA register, TB register, RB register and AIC control register. The sampling rate for A/D converter, conversion rate to D/A converter and the various parameters of the AIC can be programmed by loading control word into these programmable registers. The secondary communication protocol executed from the DSP to AIC will load the control words to AIC registers. The details of the various registers in AIC and the operation of AIC are discussed in Chapter 6 (Section 6.4.3). In this section the assembly code for resetting the AIC and programming the registers present in AIC are given for C3X processor.

AIC Reset The XF0 pin of DSP is connected to TLC320C40 AIC reset pin. The XF0 pin can be programmed as I/O pin through the I/O flag register. The XF0 pin is configured as output pin and zero volts is set at this pin for 64 machine cycles. This will reset the AIC; after reset the voltage in XF0 pin is made logic high. The assembly source code for the AIC reset operation is given in the following Program.

```
Label           Mnemonic                Comments
                ldi 0, R0               ;integer zero is loaded into R0
                sti R0, @Sxdata         ;zero is stored into serial port data transmit
                                        ;register, i.e. the register content is cleared
                RPTS 0x040              ;repeat 65 times the following instruction
                LDI 2, I0F              ;integer 2 is loaded into 1/0 flag register,
                                        ;this will reset the AIC
                RPTS 0x40               ;repeat 65 times the following instruction
                LDI 6, I0F              ;integer 6 is loaded into I/0 flag register,
                                        ;this will allow AIC to run
```

Programming AIC Control Registers The sampling rate of A/D converter and the conversion rate of D/A converter are selected using the control words loaded into TA, RA, TB and RB register. The various programmable functions of the AIC are selected using the control word loaded into AIC control register. As discussed in Chapter 6 (Section 6.4.3), the various control words for these registers are computed. The AIC secondary communication protocol routine will load all these control words into their respective registers. The AIC routine for loading the control words from DSP to AIC registers is given in Program 9.15 and it invokes the secondary communication protocol which is given in Program 9.16.

Program 9.15 **Program for loding the control words in the AIC of C3X kit**

```
Label           Mnemonic                Comments
Ta              .set 9                  ;TA register value
Tb              .set 15                 ;TB register value
Ra              .set 9                  ;RA register value
Rb              .set 15                 ;RB register value
TAcon           .word (TA«9) + (RA«2)+0 ;control word for loading both TA
                                        ;and RA registers above the set value
TBcon           .word (TB«9) + (RB«2)+2 ;control word for loading both TB
                                        ;and RB registers, above the set value
```

```
AICcon      .word 11010111b        ;control word for AIC control register
            Ldi @AICcon, R0        ;the control word for AIC control
                                   ;register is loaded into R0
            Call AIC2nd            ;call the AIC secondary
                                   ;communication protocol routine
            Ldi @TBcon, R0         ;the control word to load TB and RB
                                   ;register is loaded into R0
            Call AIC2nd            ;call the AIC secondary communication
                                   ;protocol routine
            Ldi @Tacon, R0         ;the control word to load TA and RA
                                   ;register is loaded into R0
            Call AIC2nd            ;call the AIC secondary communication
                                   ;protocol routine
```

Program 9.16 AIC secondary communication protocol routine

```
Label       Mnemonic               Comments
AIC2nd      Ldi @Sxdata, R1        ;the data available in data transmit
                                   ;register of the serial port is loaded
                                   ;into R1
            Sti R1, @Sxdata        ;the content in R1 is stored into data
                                   ;transmit register
            Idle
            Ldi @Sxdata, R1        ;the data present in data transmit
                                   ;register is loaded into R1
            Or 3, R1               ;logical OR operation between the
                                   ;content of R1 and integer 3 is
                                   ;performed
            Sti R1, @Sxdata        ;the content of R1 stored into data
                                   ;transmit register, this will enable
                                   ;the secondary communication
            Idle
            Sti R0, @Sxdata        ;the control word in R0 transmitted to
                                   ;the respective register in AIC
            Idle
            Andn 3, R1             ;logical 0R operation between the
                                   ;content of R1 and complement of integer
                                   ;3 is performed
            Sti R1, @Sxdata        ;the original content of R1 is stored
                                   ;into data transmit register
            Ldi @Srdata, R0        ;the content in data receive register
                                   ;is loaded into R0
            Rets                   ;return to main program
```

Interrupt Processing The signals used by the AIC interface to interrupt the DSP are the transmit interrupt (XINT), receive interrupt (RINT) and the hardware interrupt $\overline{\text{INT2}}$. The corresponding bits in the interrupt enable register (IE) of C3X processor are to be enabled. Whenever the new word is to be loaded into IE register, all the pending interrupts are to be cleared. This is done by globally disabling all the maskable interrupts using global interrupt enable (GIE) bit present in the status register. In C3X the transmit and receive interrupt need certain delay time; this is introduced by the two delay routines ADC and DAC.

CAPTURE AND DISPLAY OF SINE WAVE 9.7

The sine wave signal is fed to the RCA jack present in the C3X starter kit. It is digitised using the A/D converter in the AIC and transmitted to the DSP on-chip data memory via the serial port of the DSP. After storing the samples the samples are once again read from the data memory and transmitted back to the AIC D/A converter. The output of the signal obtained from the DSP can be compared with the input signal in a CRO. By varying the sampling rate and conversion rate this exercise can be repeated and the sampling theorem can be verified. The complete assembly source code along with the initialisation routine is given in Program 9.17A-9.17I.

Program 9.17A Capture and display main routine

```
Label           Mnemonic                    Comments
                .start “capture”,
                0x809802                    ;interrupt service routine seg-
                                            ;ment start address
                .sect “capture”
Tgcr            .set 0x808020               ;on-chip timer memory-map regis-
                                            ter
Tcount          .set 0x808024               ;address values
Tprd            .set 0x808028
Tperiod         .set 2
Sgcr            .set 0x808040
Sxpctrl         .set 0x808042
Srpctrl         .set 0x808043
Sxdata          .set 0x808048
Srdata          .set 0x80804C
ta              .set 9
tb              .set 15
ra              .set 9
rb              .set 15
gie             .set 0x2000                 ;control word to enable global
                                            interrupt enable bit in status
                                            ;register
TAcon           .word (TA<<9) + (RA<<2)+0
```

```
TBcon          .word (TB<<9) + (RB<<2)+2
AlCcon         .word 11010111b
sgcrval        .word 0x0E970300
sxpctrlval     .word 0x00000111
srpctrlval     .word 0x00000111
ramp           .word 0
adclast        .word 0
store          .word 809a00h
n              .word 50
               .entry
               call aic
main           ldi @store, ar1
               ldi @n, r2
main1          or gie.ST               ;disable all the interrupts
               ldi 0xF4, IE            ;enable XINT/RINT/INT2 interrupts
               ldi @Srdata, r0
               sti r0, *ar1++
               subi 1h, r2
               sti r0, @Sxdata
               bnz main1
               b main
```

Program 9.17B Interrupt service routine for RINT

```
dac            push ST
               push R3
               ldi @adclast, R3
               sti R3, @Sxdata
               pop R3
               pop ST
               reti
```

Program 9.17C Interrupt service routine for XINT

```
adc            push ST
               push R3
               Ldi @Srdata, R3
               sti R3, @adclast
               pop R3
               pop ST
               Reti
```

Program 9.17D Timer initialisation for capture and display

```
aic        ldp Tgcr
           ldi 0, R0
           sti R0, @Tgcr
           sti R0, @Tcount
           ldi Tperiod, R0
           sti R0, @Tprd
           ldi 0x2C1, R0
           sti R0, @Tgcr
```

Program 9.17E Serial port initialisation for capture and display

```
ldi @Sxpctrval, R0
sti R0, @Sxpctrl
ldi @Srpctrval, R0
sti R0, @Srpctrl
ldi 0, R0
sti R0, @Sxdata
ldi @Sgcrval, R0
sti R0, @Sgcr
```

Program 9.17F AIC RESET

```
LDI 0x10, IE
Andn 0x34, IF
ldi 0, R0
sti R0, @Sxdata
RPTS 0x040
LDI 2, IOF
Rpts 0x40
LDI 6, IOF
```

Program 9.17G Program to load control words for aic initialisation

```
Ldi @AICcon, R0
Call prog_AIC
Ldi @TBcon, R0
Call prog_AIC
ldi @TAcon, R0
Call prog_AIC
B main
```

Program 9.17H Secondary communication routine for capture and display

```
AICsecond       ldi @Sxdata, R1
                sti R1, @Sxdata
                Idle
                ldi @Sxdata, R1
                or 3, R1
                sti R1, @Sxdata
                idle
                sti R0, @Sxdata
                idle
                andn 3, R1
                sti R1, @Sxdata
                ldi @Srdata, R0
                rets
```

Program 9.17I Interrupt delay routine for capture and display

```
                .sect "intvectors"
                B dac
                B adc
```

APPENDIX 9A

TMS32OC3X STARTER KIT DEBUGGER TOOL COMMANDS

A9.1 Keyboard Commands

F1	Help screen
F2	40-bit hex display
F3	FLOAT display
F4	Source/DASM debug toggle
F5	Run
F6	Display breakpoints
F7	Clear all breakpoints
F8	Single step
F9	Toggle DASM window size
F10	Step over function
shift+F8	Force single step
shift+F10	Force function step
ALT+D	Select disassembly window
ALT+M	Select memory window

A9.2 Editing a Command

Page up	Selects first command in buffer
Page down	Selects last command in buffer
Up/down	Move through the command buffer
Left/right	Move cursor
Home	Move cursor to start
End	Move cursor to end
Shift+end	Erase remaining command
Insert	Insert mode
Delete	Delete char
Tab	Re-executes last used command
Enter	Executes command

A9.3 Debugger Commands

? <exp>	What is the value of <exp>
HELP	Display HELP
SS	Single step
XN n	Single step *n* times
STEP <n>	Single step <*n*> times
FSTEP <n>	Step through functions <*n*> times
XG addr	Single step until <addr>
GO addr	Run until <addr>
RUN	Execute with breakpoints
RUNF	Execute without breakpoints
MEM <addr>	View mem @<addr>, 32-bit hex
MEMX <addr>	”, 32-bit hex
MEMI <addr>	”, signed
MEMD <addr>	”, signed
MEML <addr>	”, signed
MEMUI<addr>	”, unsigned
MEMUD<addr>	”, unsigned
MEMUL<addr>	”, unsigned
MEMU<addr>	”, unsigned
MEMF <addr>	”, TMS float
MEMQxx <addr>	“ (xx is int), Qxx format
DASM addr	Disassemble from <addr>
MM addr	Modify memory at <addr>
MM addr, leng, val	Fill memory with <value>
SB addr	Set breakpoint at <addr>
CBaddr	Clear breakpoint at <addr>
CB	Clear all breakpoints
DB	Display breakpoints
LF <file>	Load file (defaults to last)
LOAD<file>	Load file

RELOAD <file>	Load file
SLOAD<file>	Load symbols
BLOAD<file>	Load binary only
FILE2HEX<file>	.dsk|.out to HEX
DSK2HEX<file>	.dsk to HEX
COFF2HEX<file>	.out to HEX
DSK2COFF file	.dsk to COFF
MEM2HEX file, a, 1	Mem at address, length to HEX
MEM2COFF file, a, 1	Mem at address, length to HEX
MAXFLEN length	Max out file length (no '=')
FLF <file>	Fast load from HEX file
SCLEAR or SC	Clear symbols
RESET	DSK reset
QUIT or EXIT	Quit debugger
DOS	DOS prompt
DOS <dos_exec>	Execute a DOS program
EDIT <name>	EDIT <name> (DOS editor)
DSK3 A <name>	DSK3A assemble <name>
FLOAT	CPU display in float format
REG40	CPU display in 40-bit hex
SYMBOLS or SYM	View symbol table
MOVE/MOV src, dst, n	Move src->dst *n<256* times
XON/XOFF	Enable PG6 extended opcodes
CMDxx 'cmd string'	Copies 'cmd string' to buffer xx
DASM0-DASM3	Set DASM window display mode
PAUSE	Pauses a TAKE file
END	End a take file
TAKE <filename>	Load commands from a file
CREAD <filename>	Default is CMDFILE.SAV
CLOAD <filename>	
CSAVE <filename>	Save DSK3D context to a file
CSAVEALL<filename>	'ALL' saves all on-chip memory
CSAVEALLHEX <filename>	To HEX (default) or COFF file
CSAVEALLCOFF <filename>	Default name is FULLSAVE.HEX
SAVE name, addr, leng, t	Saves 'leng' memory at 'addr'
<t>ypes	
L or LONG	- ASCII long
U or UNSIGNED	- ASCII unsigned long
F or FLOAT	- ASCII float
X or H or HEX	- ASCII hexadecimal
B or BIN	- Binary byte
C or CHAR	- Binary byte
I or INT	- Binary integer
W or WORD	- Binary long word
D or DASM	- DASM listing

APPENDIX 9B

TMS320C3X ASSEMBLER DIRECTIVES

Mnemonic and syntax	Description
(1) Directives that define sections	
.data	Assemble source code into data memory
.sect " section name"	Assemble source code into a named (initialised) section
.text	Assemble source code into program memory
(2) Directives that initialise constants (data and memory)	
.byte value 1 [,..., value *n*]	Initialise one or more 8-bit integers
.fill size in words	Reserve size words in the current section; note that a label points to the beginning of the reserved space
.float expression	Initialise a 32-bit TMS320C3X floating-point constant
.float 16 expression	Initialise a 16-bit TMS320C3X floating-point constant
.float8 expression	Initialise an 8-bit TMS320C3X floating-point constant
.ieee expression	Initialise one or more 32-bit, IEEE single-precision, floating-point constants
.int value 1 [,..., value *n]*	Initialise one or more 16-bit integers
.long value 1[, ... , value *n]*	Initialise one or more 32-bit integers
.pfloatl6	Initialise 16-bit TMS320C3X floating-point constants into a single word
.pfloat8	Initialise 8-bit TMS320C3X floating-point constants into a single word
.q xx value 1 [,..., value *n]*	Initialise a 16-bit, signed 2s-complement integer, whose decimal point is displaced xx places from the LSB
.space size in words	Reserve size words in the current section; note that a label points to the beginning of the reserved space
.string "string 1 " [,..., "string *n*"]	Initialise one or more text strings
.word value 1 [, ... , value *n*]	Initialise one or more 32-bit integers
(3) Directives that reference other files	
.copy ["] filenamef"]	Include source statements from another file
.include " filename"	Include source statements from another file
(4) Directives that enable conditional assembly	
.else	Optional conditional assembly
.endif	End conditional assembly
.if well-defined expression	Begin conditional assembly
.loop [well-defined expression]	

Begin repeatable assembly of a code block.
The loop count is determined by the well defined expression

.endloop End .loop code block

(5) Directives that modify the section program counter (SPC)

.align [size in bytes] Align the SPC on a boundary specified by size in bytes, which must be a power of 2; default to byte boundary

.entry [address] Initialise the starting address of the SPC when loading a file

(6) Directives that define symbols at assembly time

.set value Equate a value with a local symbol

.sdef value Equate a value with a local symbol multiple times

(7) Miscellaneous directives

.brstart "section name", *n* Align the named section to the next 2*n* address boundary

.end Program end

.start "section name", address Links the named section to start assembling at the location address

Review Questions

9.1 What are the basic blocks present in the C3X starter kit?

9.2 What are the software tools needed for the C3X starter kit? Explain them.

9.3 Explain the starter kit memory map. Why is it needed?

9.4 Explain the syntax for writing the assembly codes in C3X tools.

9.5 What are the programming steps needed to activate the circular buffer in C3X?

9.6 What are the limitations in parallel instructions?

9.7 How MAC operation can be performed in C3X DSPs?

9.8 What are the programming steps needed to capture real time signal in C3X DSP?

9.9 Explain the operation of C3X timer.

9.10 Explain the programming steps needed to initialise the timer.

9.11 List the memory-map registers present in the serial port of C3X.

9.12 Explain the programming steps needed to start communication through serial port.

9.13 How AIC can be reset from the C3X DSP?

9.14 What are all the interrupts to be enabled in AIC interface with DSP? Why?

9.15 List the assembler directives used to initialise constants in C3X processor.

Self Test Questions

9.1 The C3X DSP present in C3X starter kit is ———.
(a) ′C30 (b) ′C31 (c) ′C32

9.2 The crystal oscillator present in C3X starter kit is ———.
(a) 20 MHz (b) 40 MHz (c) 25 MHz (d) 50 MHz

9.3 The interface between C3X starter kit to PC is through ———.
(a) serial port (b) HPI (c) parallel port

9.4 The C3X debugger is based on ——— operating system.

(a) DOS (b) windows (c) unix

9.5 The number of windows in C3X starter kit debugger are ———.
(a) 4 (b) 2 (c) 3 (d) 5

9.6 The starting address location from which the assembly codes can be written in C3X starter kit is ———.
(a) 800000h (b) 809800h (c) 809900h (d) FF0000h

9.7 The end address location up to which the assembly codes can be written in C3X starter kit is ——— .
(a) 809800h (b) 809A00h (c) 809900h (d) 809F00h

9.8 In C3X starter kit the least 256 locations of on-chip RAM block 1 are used for ———.
(a) DSK kernel (b) Interrupt
(c) trap table (d) all of these

9.9 The assembler directive that is used to assign starting address location to a program section is ———.
(a) .start (b) .sect (c) .entry (d) .word

9.10 The assembly directive used to assign floating-point values for a variable is ———.
(a) .word (b) .int (c) .float (d) .string

9.11 The symbol used in assembly code to indicate immediate addressing mode in C3X is ———.
(a) # (b) @ (c) $ (d) *
(e) none of these

9.12 In immediate addressing mode the maximum size of operand can be ———.
(a) 8 bits (b) 16 bits (c) 24 bits (d) 32 bits

9.13 The symbol used in assembly code to indicate direct addressing mode in C3X is ———.
(a) @ (b) # (c) $ (d) *

9.14 In direct addressing mode the number of bits used to specify the address is ———.
(a) 8 bits (b) 16 bits (c) 24 bits (d) 32 bits

9.15 The symbol used in assembly code to indicate indirect addressing mode in C3X is ———.
(a) # (b) @ (c) $ (d) *

9.16 The symbol used in assembly code to indicate circular addressing mode in C3X is ———.
(a) % (b) @ (c) # (d) *

9.17 In three-operand instructions the addressing mode used are ———.
(a) register, direct (b) indirect, direct
(c) register, indirect

9.18 In parallel instructions the addressing mode used are ———.
(a) register, indirect (b) indirect, direct
(c) register, direct

9.19 The size of AIC present in C3X starter kit is ———.
(a) 14 bits (b) 16 bits (c) 8 bits (d) 10 bits

9.20 In timer initialisation, the value set in the timer period register is ———.
(a) 2 (b) 4 (c) 16 (d) 24

9.21 The number of registers to be programmed in serial port initialisation routine is ———.
(a) 2 (b) 3 (c) 4 (d) 5

9.22 The pin used for AIC reset in C3X starter kits is ———.
(a) BR (b) XF0 (c) XF1 (1) HLD

9.23 The number of AIC control registers to be programmed in AIC initialisation routine of C3X is
(a) 5 (b) 4 (c) 6 (d) 3

9.24 The number of interrupts to be set in AIC initialisation routine is ———.
(a) 3 (b) 4 (c) 2 (d) 5

9.25 The maximum sampling rate at which the audio signals can be sampled in AIC of C3X starter kit is ———.
(a) 9.2 kHz (b) 14.4 kHz (c) 8 kHz (d) 19.2 kHz

10 AN OVERVIEW OF TMS320C54X

INTRODUCTION 10.1

The ′54X has been developed to be almost upward compatible to the earlier fixed-point processors from TI such as 1X, 2XX and 5X and at the same time has been built with more advanced features and more DSP application specific logic in its hardware and instruction set so as to make it widely applicable. ′54X has a large no. of instructions compared to its predecessors and a variety of ways in which the application programs can be written, viz., using assembly language, algebraic instructions, high level language or combinations of these. ′54X is easy to learn and use if it is seen as an attempt to remove some of the limitations of its predecessor, for example, 5X. The simplest approach would be to migrate from 5X to 54X. Almost all the programs written in 5X can be made to work in 54X without any major modification. For the majority of instructions of 5X there is an equivalent instruction in 54X. There are programs which take a 5X assembly language program and convert it to an equivalent program in assembly language with 54X syntax. Appendix 10.1 gives a summary of C54X instructions. Appendix 10.2 gives the list of 54X instructions and the equivalent 5X instructions. This may be used to translate the 5X program to 54X program either manually or with the aid of a program (see, e.g., Code Composer Studio in Chapter 12).

One of the advantages of 54X for the users who have used 5X before is that the on-chip memory and peripherals in 5X and 54X are almost identical. For example, the on-chip timer and synchronous serial port in both 5X and 54X are identical in operation. The programs used for initialising the on-chip timers and serial ports work without any modification.

In the next sections some of the features of 54X are reviewed and only those features which differ from those of the 5X DSP or which are new in 54X are discussed in some detail.

ARCHITECTURE OF 54X 10.2

A quick review of the architecture of 54X is achieved by comparing the features of 5X with 54X as shown in Table 10.1. From this table it can be verified that some of the registers of 54X are identical to that of 5X.

Many of the on-chip peripherals are also identical. However, the actual memory map addresses of the CPU registers and the on-chip peripherals of 54X and 5X are not identical. This should be taken care while porting the 5X program to 54X platform. The block diagram of TMS320C54X internal hardware

Table 10.1 *Comparison of the features of 5X and 54X*

Description	*5X*	*54X*
Name of program bus	One, PB	One, PB
Name of the data bus	One, DB	DB and CB (for Read) EB (for write)
Name of address buses	PAB, DAB	PAB, CAB, DAB, EAB
MainALU	32-bit ALU	40-bit ALU
Accumulators	32-bit ACC	40-bit ACCA and ACCB
Barrel shifter	0-16-bit left shift 0-16-bit right shift	40 bit: 0-31 left shift 0-15 right shift
Multiplier	16 X 16 bit	17 X 17 bit
Adder	32 bit	40 bit
Auxiliary Register ALU	ARAU	ARAU0&ARAU1
Auxiliary registers	AR0-AR7	AR0-AR7
Stack pointer (SP)	Not available	16 **bit:** SP
Circular buffer register	Two 16-bit start & end register.	16-bit BK
Status registers	16-bit PMST, ST0, ST1	16-bit PMST, ST0, ST1
Block repeat registers	16-bit BRCR,PASR,PAER	16-bit BRC, RSA,REA
Program counter	16-bit PC	16-bit PC
Extended progm memory	Not available	7-bit XPC
Interrupt registers	16-bit IMR and IFR	16-bit IMR and IFR
General purpose I/O	$\overline{\text{BIO}}$ and XF	Same as that of 5X
Wait state generator	PDWSR	SWWSR
Hardware timer	16-bit timer	Same as that of 5X
Clock generator	PLL based	Same as that of 5X
Synchronous serial port	Full duplex and double buffered	Same as that of 5X
TDM serial ports	Upto 7 devices using TDM can communicate serially	Same as that of 5X
Buffered serial port	Standard 5X serial port with additional autobuffering unit	Same as that of 5X
Host port interface	8-bit standard HPI	8-bit standard HPI or enhanced 8-bit and 16-bit HPI
Multichannel buffered serial port including internal programmable clock and other advanced features	Not available	Available
On-chip ROM for look up table for A law, u. law companding, sine wave generation	Not available	Available

is shown in Fig. 10.1. It consists of the CPU containing the various functional units such as ALU, MAC unit, EXP encoder, barrel shifter, memory mapped registers, system control interface, peripheral interface,

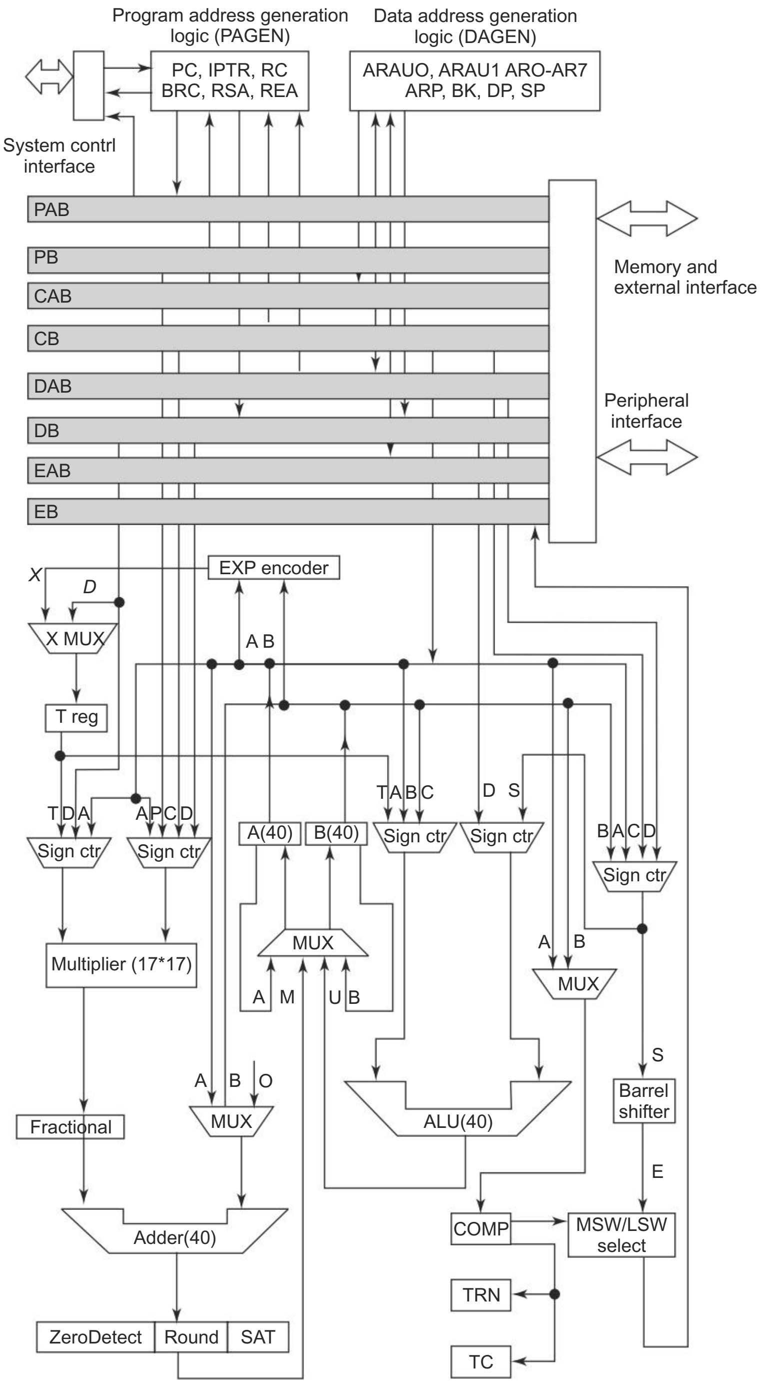


Fig. 10.1 *Block diagram of 54X internal hardware*

memory and external interface, program address generation logic (PAGEN) and data address generation logic (DAGEN) and eight 16-bit buses which interconnect these units. Details of each of these units are considered next.

′54X BUSES 10.3

The ′54x architecture is built around eight major 16-bit buses (four program/data buses and four address buses): The program bus (PB) carries the instruction code and immediate operands from program memory. Three data buses (CB, DB and EB) interconnect to various elements, such as the CPU, data address generation logic, program address generation logic, on-chip peripherals and data memory. The CB and DB carry the operands that are read from data memory. The EB carries the data to be written to memory. Four address buses (PAB, CAB, DAB and EAB) carry the addresses needed for instruction execution.

The ′54X can generate upto two data-memory addresses per cycle using the two ARAUs (ARAU0 and ARAU1). The PB can carry data operands stored in program space (for instance, a filter coefficient table) to the multiplier and adder for multiply/accumulate operations or to a destination in data space for data move instructions (MVPD and READA). The capability to read one coefficient from program memory and two data values from the data memory using ARAU0 and ARAU1 enables the operation $[x(i) + x(N\text{-}1\text{-}i)] \times h(i)$ required for the symmetric FIR filter in single cycle. The 54X instruction FIRS is used for this purpose.

The ′54x also has an on-chip bidirectional bus for accessing on-chip peripherals; this bus is connected to DB and EB through the bus exchanger in the CPU interface. Accesses that use this bus can require two or more cycles for reads and writes, depending on the peripheral's structure.

INTERNAL MEMORY ORGANISATION 10.4

The ′54x memory is organised into three individually selectable spaces: program, data and I/O space. All ′54X devices contain both random access memory (RAM) and read only memory (ROM). Among the devices, two types of RAM are represented: dual-access RAM (DARAM) and single-access RAM (SARAM). The DARAM and SARAM may be configured either as data memory or program/data memory. Table 10.2 shows how much ROM, DARAM and SARAM are available on the different ′54X devices. The ′54X also has 26 CPU registers plus peripheral registers that are mapped in data memory space.

Table 10.2 *Program and data memory on the TMS320C54x devices*

Memory type	*′541*	*′542, ′543*	*′545, ′546*	*′548*	*′549*	*′5402*	*′5410*	*′5420*
ROM:	28K	2K	48K	2K	16K	4K	16K	0
Program	20K	2K	32K	2K	16k	4k	16k	0
Program/data	8K	0	16K	0	16K	4K	0	0
DARAM†	5K	10K	6K	8K	8K	16K	8K	32K
SARAM†	0	0	0	24K	24K	0	56K	168K

† TheDARAM and SARAM may be configured as data memory or program/data memory.

10.4.1 On-ChipROM

The on-chip ROM is part of the program memory space and, in some cases, part of the data memory space. The amount of on-chip ROM available on each device varies, as shown in Table 10.2. On devices with a small amount of ROM (2K words), the ROM contains a boot loader, which is useful for booting to faster on-chip or external RAM. On devices with larger amounts of ROM, a portion of the ROM may be mapped into both data and program space.

10.4.2 On-Chip Dual-Access RAM (DARAM)

The DARAM is composed of several blocks. Because each DARAM block can be accessed twice per machine cycle, the CPU can read from and write to a single block of DARAM in the same cycle. The DARAM is always mapped in data space and is primarily intended to store data values. It can also be mapped into program space and used to store program code.

10.4.3 On-Chip Single-Access RAM (SARAM)

The SARAM is composed of several blocks. Each block is accessible once per machine cycle for either a read or a write. The SARAM is always mapped in data space and is primarily intended to store data values. It can also be mapped into program space and used to store program code.

10.4.4 On-Chip Memory Security

The ′54X maskable memory security option protects the contents of on-chip memories. When this option is chosen, no externally originating instruction can access the on-chip memory spaces.

10.4.5 Memory-Mapped Registers

The data memory space contains memory-mapped registers for the CPU and the on-chip peripherals. These registers are located on data page 0, simplifying access to them. The memory-mapped access provides a convenient way to save and restore the registers for context switches and to transfer information between the accumulators and the other registers.

CENTRAL PROCESSING UNIT (CPU) 10.5

The ′54X CPU is common to all the ′54X devices. The block diagram of Internal hardware of 54X is given in Fig. 10.1. The 54X CPU contains:

40-Bit Arithmetic Logic Unit (ALU)
Two 40-Bit Accumulator Registers
Barrel Shifter Supporting a -16 to 31 Shift Range
Multiply/Accumulate Block
16-Bit Temporary Register (T)
16-Bit Transition Register (TRN)
Compare, Select and Store Unit (CSSU)
Exponent Encoder

The CPU registers are memory-mapped, enabling quick saves and restores. Table 10.3 gives the list of memory-mapped CPU registers and their functions are as follows:

10.5.1 Interrupt Registers (IMR, IFR)

The interrupt mask register (IMR) individually masks off specific interrupts at required times. The interrupt flag register (IFR) indicates the current status of the interrupts.

10.5.2 Status Registers (ST0 and ST1)

The status registers ST0 and ST1 contain the status of the various conditions and modes for the ′54X devices. ST0 contains the flags (OVA, OVB, C and TC) produced by arithmetic operations and bit manipulations, in addition to the DP and the ARP fields. ST1 reflects the status of modes and instructions executed by the processor. ST0 and ST1 contain the status of various conditions and modes; PMST contains memory-setup status and control information. Because these registers are memory-mapped, they can be stored into and loaded from data memory; the status of the processor can be saved and restored for subroutines and interrupt service routines (ISRs). The individual bits of the ST0 and ST1 registers can be set or cleared with the SSBX and RSBX instructions. For example, the sign-extension mode is set with SSBX 1, SXM, or reset with RSBX 1, SXM. The ARP, DP and ASM bit fields can be loaded using the LD instruction with a short-immediate operand. The ASM and DP fields can also be loaded with data-memory values by using the LD instruction. The ST0 bits are shown in Fig. 10.2 and described in Table 10.3. The ST1 bits are shown in Fig. 10.3 and described in Table 10.5.

Table 10.3 *CPU memory-mapped registers*

Add	*Name*	*Description*
0	IMR	Interrupt mask register
1	IFR	Interrupt flag register
2-5	—	Reserved for testing
6	ST0	Status register 0
7	ST1	Status register 1
8	AL	Accumulator A low word (bits 15-0)
9	AH	Accumulator A high word (bits 31-16)
A	AG	Accumulator A guard bits (bits 39-32)
B	BL	Accumulator B low word (bits 15-0)
C	BH	Accumulator B high word (bits 31-16)
D	BG	Accumulator B guard bits (bits 39-32)
E	T	Temporary register
F	TRN	Transition register
10	AR0	Auxiliary register 0
11	AR1	Auxiliary register 1
12	AR2	Auxiliary register 2
13	AR3	Auxiliary register 3
14	AR4	Auxiliary register 4
15	AR5	Auxiliary register 5
16	AR6	Auxiliary register 6
17	AR7	Auxiliary register 7
18	SP	Stack pointer
19	BK	Circular-buffer size register
1A	BRC	Block-repeat counter
1B	RSA	Block-repeat start address
1C	REA	Block-repeat end address
1D	PMST	Processor mode status register
1E	XPC	Program counter extension register (′548/9/02/10/20)
1F	-	Reserved

15 – 13	12	11	10	9	8 – 0
ARP	TC	C	OVA	OVB	DP

Fig. 10.2 *Status register 0 (ST0) diagram*

Table 10.4 *Status register 0 (ST0) bit summary*

Name	*Function*
ARP	Auxiliary register pointer. This 3-bit field selects the auxiliary register to be used in indirect single-operand addressing. ARP must always be set to zero when the DSP is in standard mode (CMPT = 0)
TC	Test/control flag. TC stores the results of the arithmetic logic unit (ALU) test bit operations. TC is affected by the BIT, BITF, BITT, CMPM, CMPR, CMPS and SFTC instructions. The status (set or cleared) of TC determines if the conditional branch, call, execute and return instructions execute
C	Carry: set to 1 if the result of an addition generates a carry; it is cleared to 0 if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition and is set after a subtraction, except for an ADD or SUB with a 16-bit shift
OVA	Overflow flag for accumulator A . OVA is set to 1 when an overflow occurs in either the ALU or the multiplier's adder and the destination for the result is accumulator A
OVB	Overflow flag for accumulator B . OVB is set to 1 when an overflow occurs in either the ALU or the multiplier's adder and the destination for the result is accumulator B
DP	Data-memory page pointer . This 9-bit field is concatenated with the seven LSBs of an instruction word to form a direct-memory address of 1 6 bits for single data-memory operand addressing. This operation is done if the compiler mode bit in ST1 (CPL) = 0

15	14	13	12	11	10	9	8	7	6	5	4 – 0
BRAF	CPL	XF	HM	INTM	0	OVM	SXM	C16	FRCT	CMPT	ASM

Fig. 10.3 *Status register 1 (ST1) diagram*

Table 10.5 *Status register 1 (ST1) bit summary*

Name	*Function*
BRAF	Block-repeat active flag. BRAF indicates whether a block repeat iscurrently active. BRAF = 0, the block repeat is deactivated. BRAF is cleared when the block-repeat Counter (BRC) decrements below 0. BRAF = 1, the block repeat is active. BRAF is automatically set when an RPTB instruction is executed
CPL	Compiler mode. CPL indicates which pointer is used in relative direct addressing: CPL = 0, the relative direct-addressing mode using the data page pointer (DP) is selected. CPL = 1, the relative direct-addressing mode using the stack pointer (SP) is selected
XF	XF status. XF indicates the status of the external flag (XF) pin, which is a general-purpose output pin. The SSBX instruction can set XF and the RSBX instruction can reset XF
HM	Hold mode. HM indicates whether the processor continues internal execution when acknowledging an active HOLD signal: HM = 0, the processor continues execution from internal program memory but places its external interface in the high-impedance state. HM = 1, the processor halts internal execution

(Contd.)

Table 10.5 *(Contd.)*

Name	*Function*
INTM	Interrupt mode. INTM globally masks or enables all interrupts. INTM = 0, all unmasked interrupts are enabled. INTM = 1, all maskable interrupts are disabled. The SSBX instruction sets INTM and the RSBX instruction resets INTM. INTM is set to 1 by reset or when a maskable interrupt trap is taken (INTR or external interrupts). INTM is cleared to 0 when a RETE or RETF instruction (return from interrupt) is executed. INTM does not affect the nonmaskable interrupts (RS and NMI). INTM cannot be set by memory-write operations
0	Always read as 0
OVM	Overflow mode. OVM determines what is loaded into the destination accumulator when an overflow occurs: OVM = 0, an overflowed result from either the ALU or the multiplier's adder overflows normally in the destination accumulator. OVM = 1, the destination accumulator is set to either the most positive value (00 7FFF FFFFh) or the most negative value (FF 8000 0000h) upon encountering an overflow. The SSBX and RSBX instructions set and reset OVM, respectively
SXM	Sign-extension mode. SXM determines whether sign extension is performed: SXM = 0, sign extension is suppressed. SXM = 1, data is sign extended before being used by the ALU
C16	Dual 16-bit/double-precision arithmetic mode. CT6 determines the arithmetic mode of the ALU's operation: C16 = 0, the ALU operates in double-precision arithmetic mode. C16 = 1, the ALU operates in dual 16-bit arithmetic mode
FRCT	Fractional mode. When FRCT is 1, the multiplier output is left-shifted by 1 bit to compensate for an extra sign bit
CMPT	Compatibility mode. CMPT determines the compatibility mode for the ARP: CMPT = 0, ARP is not updated in indirect addressing mode with a single data-memory operand. ARP must always be set to 0 when the DSP is in this mode. CMPT = 1, ARP is updated in indirect addressing mode with a single data-memory operand, except when the instruction is selecting auxiliary register 0 (AR0).
ASM	Accumulator shift mode. The **5-bit** ASM field specifies a shift value within a -16 through 15 range and is coded as a 2s-complement value.

10.5.3 Processor Mode Status Register (PMST)

The PMST register is loaded with memory-mapped register instructions such as STM. The PMST bits are shown in Fig. 10.4 and described in Table 10.6.

15-7	6	5	4	3	2	1	0
IPTR	MP/$\overline{MC}$	OVLY	AVIS	DROM	CLKOFF	SMUL+	SST+

Fig. 10.4 *Processor mode status register (PMST) diagram. + only on the LP devices; reserved bits on all other devices*

Table 10.6 *Processor mode status register (PMST) bit summary Name Function*

Name	*Function*
IPTR	Interrupt vector pointer. The 9-bit IPTR field points to the 128-word program page where the interrupt vectors reside
MP/$\overline{MC}$	MP/$\overline{MC}$ pin microprocessor/ microcomputer mode MP/$\overline{MC}$ enables/disables the on-chip ROM to be addressable in program memory space. MP/$\overline{MC}$ = 0, the on-chip ROM is enabled and addressable. MP/$\overline{MC}$ = 1, the on-chip ROM is not available

(Contd.)

Table 10.6 *(Contd.)*

OVLY	RAM overlay. OVLY enables on-chip dual-access data RAM blocks to be mapped into program space. The values for the OVLY bit are: OVLY = 0, the on-chip RAM is addressable in data space but not in program space. OVLY = 1, the on-chip RAM is mapped into program space and data space. Data page 0 (addresses Oh to 7Fh), however, is not mapped into program space
AVIS	Address visibility mode. AVIS enables/disables the internal program address to be visible at the address pins. AVIS = 0, the external address lines do not change with the internal program address. Control and data lines are not affected and the address bus is driven with the last address on the bus. AVIS = 1, this mode allows the internal program address to appear at the pins of the ′54X so that the internal program address can be traced
DROM	Data ROM. DROM enables on-chip ROM to be mapped into data space. DROM = 0, the on-chip ROM is not mapped into data space. DROM = 1, a portion of the on-chip ROM is mapped into data space
CLKOFF	CLOCKOUT off. When the CLKOFF bit is 1, the output of CLKOUT is disabled and remains at a high level
SMUL†	Saturation on multiplication. When SMUL = 1, saturation of a multiplication result occurs before performing the accumulation in a MAC or MAS instruction
SST†	Saturation on store. When SST = 1. saturation of the data from the accumulator is enabled before storing in memory. The saturation is performed after the shift operation

†Only on the LP devices; reserved bits on all other devices.

10.5.4 Accumulators A and B

The ′54X devices have two 40-bit accumulators: accumulator A and accumulator B. Each accumulator is memory-mapped and partitioned into accumulator low word (AL, BL), accumulator high word (AH, BH) and accumulator guard bits (AG, BG). Accumulator A and accumulator B can be configured as the destination registers for either the multiplier/adder unit or the ALU. In addition, they are used for MIN and MAX instructions or for the parallel instruction LD||MAC, in which one accumulator loads data and the other performs computations. Each accumulator is split into three parts, as shown in Figs 10.5 and 10.6.

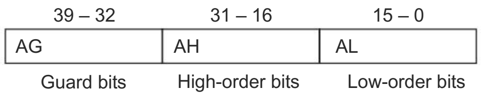


Fig. 10.5 *Accumulator A*

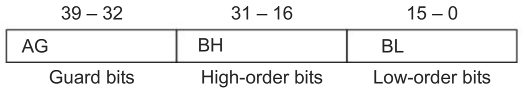


Fig. 10.6 *Accumulator B*

The guard bits are used as a headmargin for computations. Headmargins allow the prevention of some overflow in iterative computations such as autocorrelation. AG, BG, AH, BH, AL and BL are memory-mapped registers that can be pushed onto and popped from the stack for context saves and restores by using PSHM and POPM instructions. These registers can also be used by other instructions that use memory-mapped registers (MMR) for page 0 addressing. The only difference between accumulators A and B is that bits 32-16 of A can be used as an input to the multiplier in the multiplier/ adder unit.

10.5.5 Temporary (T) Register

The temporary (T) register has many uses. For example, it may hold

(a) one of the multiplicands for multiply and multiply/accumulate instructions;

(b) a dynamic (execution-time programmable) shift count for instructions with shift operation such as the ADD, LD and SUB instructions;

(c) a dynamic bit address for the BITT instruction, and

(d) branch metrics used by the DADST and DSADT instructions for ACS operation of Viterbi decoding. In addition, the EXP instruction stores the exponent value computed into T register, and then the NORM instruction uses the T register value to normalise the number.

10.5.6 Transition (TRN) Register

The 16-bit transition (TRN) register holds the transition decision for the path to new metrics to perform the Viterbi algorithm. The CMPS (compare select max and store) instruction updates the contents of TRN register on the basis of the comparison between the accumulator high word and the accumulator low word.

10.5.7 Auxiliary Registers (AR0-AR7)

The eight 16-bit ARs (AR0-AR7) can be accessed by the CPU and modified by the ARAUs. The primary function of the ARs is to generate 16-bit addresses for data space. However, these registers can also act as general-purpose registers or counters.

10.5.8 Stack-Pointer (SP) Register

The 16-bit stack-pointer (SP) register contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The stack is manipulated by interrupts, traps, calls, returns and the PSHD, PSHM, POPD and POPM instructions. Pushes and pops of the stack predecrement and postincrement, respectively, the 16-bit value in the stack pointer.

10.5.9 Circular-Buffer Size Register (BK)

The ARAUs use 16-bit circular-buffer size register (BK) in circular addressing to specify the data block size. For information on BK and circular addressing, see Section 11.1.5.7.

10.5.10 Block-Repeat Registers (BRC, RSA and REA)

The 16-bit block-repeat counter (BRC) register specifies the number of times a block of code is to repeat when a block repeat is performed. The 16-bit block-repeat start address (RSA) register contains the starting address of the block of program memory to be repeated. The 16-bit block-repeat end address (REA) register contains the ending address of the block of program memory to be repeated.

10.5.11 Program Counter Extension Register (XPC)

This is available on ′548/9/02/10/20. The program counter extension register (XPC) contains the upper 7 bits of the current program memory address. This allows access of upto 8192K words of program memory. In all these devices, the no. of address lines is increased to 23. They have six extra instructions for addressing the extended program space.

ARITHMETIC LOGIC UNIT (ALU) 10.6

The 40-bit ALU, shown in Fig. 10.8, implements a wide range of arithmetic and logical functions, most of which execute in a single clock cycle. After an operation is performed in the ALU, the result is usually transferred to a destination accumulator (accumulator A or B). Instructions that perform memory-to-memory operations (ADDM, ANDM, ORM and XORM) are exceptions.

10.6.1 ALU Input

ALU input takes several forms from several sources. The X input source to the ALU is either of two values:

- The shifter output (a 32-bit or 16-bit data-memory operand or a shifted accumulator value)
- A data-memory operand from data bus DB

The Y input source to the ALU is any of three values:

- The value in one of the accumulators (A or B)
- A data-memory operand from data bus CB
- The value in the T register

When a 16-bit data-memory operand is fed through data bus CB or DB, the 40-bit ALU input is constructed in one of two ways:

If bits 15 through 0 contain the data-memory operand, bits 39 through 16 are zero filled (SXM = 0) or sign-extended (SXM = 1). If bits 31 through 16 contain the data-memory operand, bits 15 through 0 are zero filled, and bits 39 through 32 are either zero filled (SXM = 0) or sign extended (SXM =1).

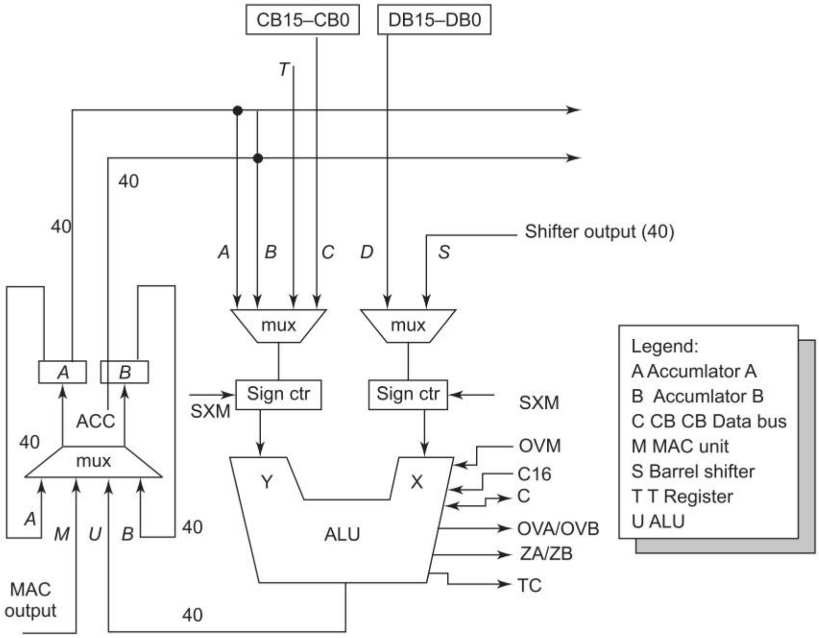


Fig.10.7 *'C54X arithmetic logic unit functional diagram*

10.6.2 Overflow Handling

The ALU saturation logic prevents a result from overflowing by keeping the result at a maximum (or minimum) value. This feature is useful for filter calculations. The logic is enabled when the overflow mode bit (OVM) in status register ST1 is set. When a result overflows:

If OVM = 0, the accumulators are loaded with the ALU result without modification

If OVM = 1, the accumulators are loaded with either the most positive 32-bit value (00 7FFF FFFFh) or the most negative 32-bit value (0FF 8000 0000h), depending on the direction of the overflow

The overflow flag (OVA/OVB) in status register ST0 is set for the destination accumulator and remains set until one of the following occurs:

A reset is performed

A conditional instruction (such as a branch, a return, a call or an execute) is executed on an overflow condition.

The overflow flag (OVA/OVB) is cleared

The accumulator may also be saturated by using the SAT instruction, regardless of the value of OVM.

10.6.3 The Carry Bit

The ALU has an associated carry (C) bit that is affected by most arithmetic ALU instructions, including rotate and shift operations. The C bit supports efficient computation of extended-precision arithmetic operations. The C bit is not affected by loading the accumulator, performing logical operations or executing other nonarithmetic or control instructions, so it can be used for overflow management. Two conditional operands, C and NC, enable branching, calling, returning and conditionally executing according to the status (set or cleared) of the C bit. Also, the RSBX and SSBX instructions can be used to load the C bit. The C bit is set on a hardware reset.

10.6.4 Dual 16-Bit Mode

For arithmetic operations, the ALU can operate in a special dual 16-bit arithmetic mode that performs two 16-bit operations (for instance, two additions or two subtractions) in one cycle. This mode is selected by setting the C16 field of ST1. This mode is especially useful for the Viterbi add/compare/select operation (see Section 10.9, Compare, Select and Store Unit (CSSU)).

BARREL SHIFTER 10.7

The barrel shifter is used for scaling operations such as prescaling an input data-memory operand or the accumulator value before an ALU operation; performing a logical or arithmetic shift of the accumulator value; normalising the accumulator; postscaling the accumulator before storing the accumulator value into data memory. The SXM bit controls signed/unsigned extension of the data operands; when the bit is set, sign extension is performed. Some instructions, such as LDU, ADDS and SUBS operate with unsigned memory operands and do not perform sign extension, regardless of the SXM value. The shift count determines how many bits to shift. Positive shift values correspond to left shifts, whereas negative values correspond to right shifts. The shift count is specified as a 2s-complement value in several ways, depending on the instruction type,

MULTIPLIER/ADDER UNIT 10.8

The ′54X CPU has a 17-bit × 17-bit hardware multiplier coupled to a 40-bit dedicated adder. This multiplier/adder unit provides multiply and accumulate (MAC) capability in one pipeline phase cycle. The multiplier/adder unit is shown in Fig. 10.8.

The multiplier can perform signed, unsigned and signed/unsigned multiplication with the following constraints:

For signed multiplication, each 16-bit memory operand is assumed to be a 17-bit word with sign extension. For unsigned multiplication, a 0 is added to the MSB (bit 16) in each input operand. For signed/unsigned multiplication, one of the operands is sign extended, and the other is extended with a 0 in the MSB (zero filled). The multiplier output can be shifted left by 1 bit to compensate for the extra sign bit generated by multiplying two 16-bit 2s-complement numbers in fractional mode. (Fractional mode is selected when the FRCT bit = 1 in ST1.) The adder in the multiplier/adder unit contains a zero detector, a rounder (2s complement) and overflow/saturation logic. Rounding consists of adding 2^{15} to the result and then clearing the lower 16 bits of the destination accumulator. Rounding is performed in some multiply, MAC and multiply/subtract (MAS) instructions when the suffix R is included with the instruction. The LMS instruction also rounds to minimise quantisation errors in updated coefficients. The adder's inputs come from the multiplier's output and from one of the accumulators. Once any multiply operation is performed in the unit, the result is transferred to a destination accumulator (A orB).

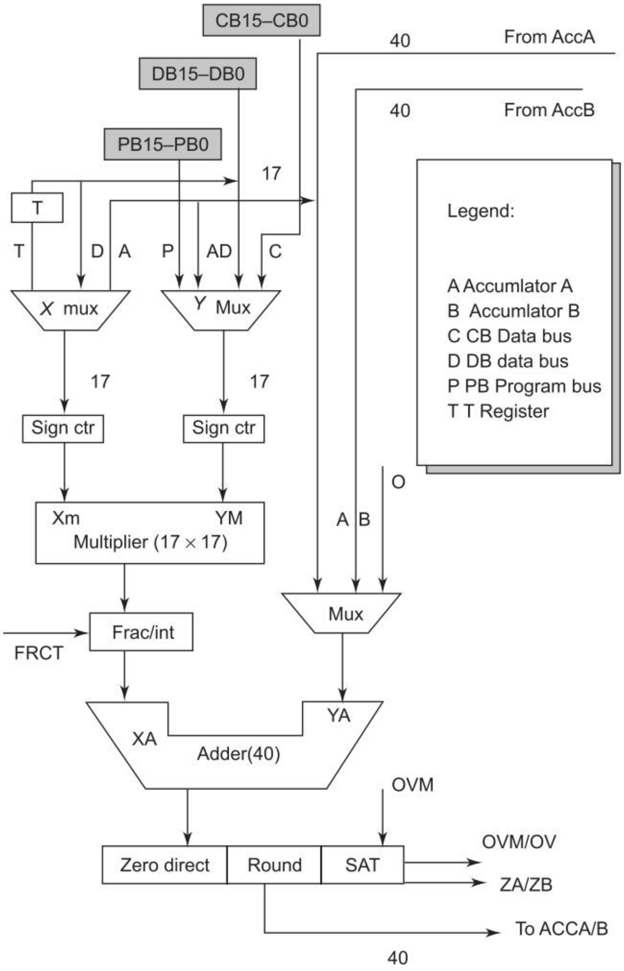


Fig. 10.8 *C54X multiplier and adder functional diagram*

10.8.1 Multiplier Input Sources

Sources for the multiplier inputs are as follows:

The XM input source to the multiplier is any of the following values:

The T Register

A Data-memory Operand from Data Bus DB

Accumulator A Bits 32-16

The YM Input source to the multiplier is any of the following values:

A Data-memory Operand from Data Bus DB

A Data-memory Operand from Data Bus CB

A Program-memory Operand from Program Bus PB

Accumulator A bits 32-16

Since bits A(32-16) can be an input to the multiplier, some sequences that require storing the result of one computation in memory and feeding this result to the multiplier can be made faster. For some application-specific instructions (FIRS, SQDST, ABDST and POLY), the contents of accumulator A can be computed by the ALU and then input to the multiplier without any overhead.

COMPARE, SELECT AND STORE UNIT (CSSU) 10.9

The compare, select and store unit (CSSU) is an application-specific hardware unit dedicated to add/ compare/select (ACS) operations of the Viterbi operator. Figure 10.9 shows the CSSU, which is used with the ALU to perform fast ACS operations. The CSSU allows the ′54X to support various Viterbi butterfly algorithms used in equalisers and channel decoders. The add function of the Viterbi operator in Fig. 10.10 is performed by the ALU. This function consists of a double addition function (Metl±Dl and Met2± D2). Double addition is completed in one machine cycle if the ALU is configured for dual 16-bit mode by setting the C16 bit in ST1. With the ALU configured in dual 16-bit mode, all the long-word (32-bit) instructions become dual 16-bit arithmetic instructions. T is connected to the ALU input (as a dual 16-bit operand) and is used as local storage in order to minimise memory access.

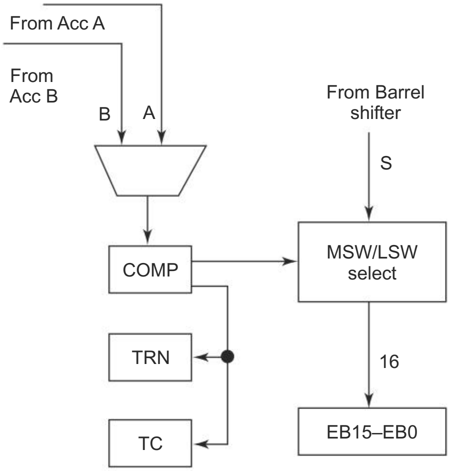


Fig. 10.9 *Compare, select and store unit (CSSU)*

The CSSU implements the compare and select operation via the CMPS instruction, a comparator and the 16-bit transition (TRN) register. This operation compares two 16-bit parts of the specified accumulator and shifts the decision into bit 0 of TRN. This decision is also stored in the TC bit of ST0. Based on the decision, the corresponding 16-bit part of the accumulator is stored in data memory. TRN register contains information of the path transition decisions to new states. This information can be used for a back-tracking routine that finds the optimal path, which results in decoding the code.

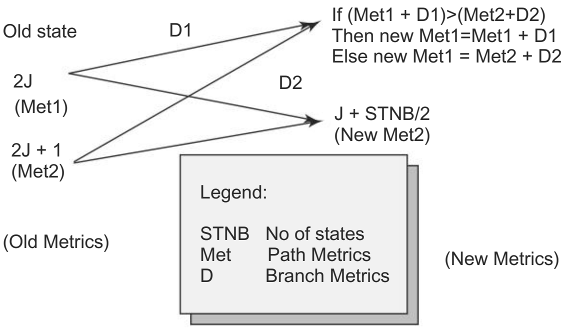


Fig. 10.10 *Viterbi operator*

EXPONENT ENCODER 10.10

The exponent encoder is an application-specific hardware device dedicated to supporting the EXP instruction in a single cycle. With the EXP instruction, the exponent value in the accumulator can be stored in T as a 2s-complement value within a -8 through 31 range. The exponent is defined as the number of leading redundant bits - 8, which corresponds to the number of shifts required in the accumulator to eliminate non-significant sign bits. This operation results in a negative value when the accumulator value exceeds 32 bits. The EXP and NORM instructions use the exponent encoder to normalise the accumulator's contents efficiently. NORM supports shifting the accumulator value by the number of bits specified in T in a single cycle. A negative value in T produces a right shift of the accumulator's contents, which normalises any value beyond the 32-bit range of the accumulator.

THE C54X PIPELINE 10.11

The C54X instruction pipeline consists of six levels: prefetch, fetch, decode, access, read and execute. At each of the levels, an independent operation occurs. Because these operations are independent, from one to six instructions can be active in any given cycle, each instruction at a different stage of completion. Typically, the pipeline is full with a sequential set of instructions, each at one of the six stages. When a PC discontinuity occurs, such as during a branch, call or return, one or more stages of the pipeline may be temporarily unused.

ON-CHIP PERIPHERALS 10.12

All the ′54X devices have the same CPU, but different on-chip peripherals are connected to their CPUs. The ′54X devices have these on-chip peripheral options:

General-purpose I/O Pins (BIO and XF)
Software-programmable Wait-State Generator
Programmable Bank-switching Logic
Host Port Interface (HPI)
Hardware Timer
Clock Generator
Serial Ports
Synchronous Serial Ports
Buffered Serial Ports
Time-division Multiplexed (TDM) Serial Ports

10.12.1 General-Purpose I/O Pins

Each ′54X device has two general-purpose I/O pins: BIO and XF. BIO is an input pin that can be used to monitor the status of external devices. XF is a software-controlled output pin that allows you to signal external devices.

10.12.2 Software-Programmable Wait-State Generator

The software-programmable wait-state generator extends external bus cycles upto seven machine cycles to interface with slower off-chip memory and I/O devices. The software wait-state generator is incorporated without any external hardware. For off-chip memory accesses, from zero to seven wait states can be specified within the software wait-state register (SWWSR) for each 32K-word block of program and data memory, and for the 64K-word block of I/O space.

10.12.3 Programmable Bank-Switching Logic

The programmable bank-switching logic can automatically insert one cycle when an access crosses memory bank boundaries inside program memory or data memory. One cycle can also be inserted when an access crosses from program memory to data memory. This extra cycle prevents bus contention by allowing memory devices to release the bus before other devices start driving the bus. The size of a memory bank for bank switching is defined by the bank switching control register (BSCR).

10.12.4 Host Port Interface

The host port interface (HPI) is an 8-bit parallel port that provides an interface to a host processor. Information is exchanged between the ′54X and the host processor through ′54X on-chip memory that is accessible to both the host processor and the ′54X.

10.12.5 Hardware Timer

The ′54X features a 16-bit timing circuit with a 4-bit prescaler. The timer counter is decremented by 1 at every CLKOUT cycle. Each time the counter decrements to 0, a timer interrupt is generated. The timer can be stopped, restarted, reset or disabled by specific status bits.

10.12.6 Clock Generator

The clock generator consists of an internal oscillator and a phase-locked loop (PLL) circuit. The clock generator can be driven internally by a crystal resonator circuit or externally by a clock source. The PLL circuit can generate an internal CPU clock by multiplying the clock source by a specific factor; thus, a clock source with a lower frequency than that of the CPU should be used.

10.12.7 Serial Ports

The serial ports on the ′54X vary by device, but four types of serial ports are represented: synchronous, buffered, multichannel buffer (McBSP) and time-division multiplexed (TDM). Table 10.7 gives the number of each type of serial ports on the various ′54X devices.

Table 10.7 *Serial port interfaces on the TMS320C54X devices*

Serial ports	*′541*	*′542, ′543*	*′545, ′540*	*′548/′549*	*′5402*	*′5410*	*′5420*
Synchronous	2	0	1	0	0	0	0
Buffered	0	1	1	2	0	0	0
McBSP	0	0	0	0	2	3	6
TDM	0	1	0	0	0	0	0

Synchronous Serial I/O Ports The synchronous serial ports are high-speed, full-duplexed serial ports that provide direct communication with serial devices such as codecs, analog-to-digital (A/D) converters and other serial systems. When more than one synchronous serial port resides on a ′54X, these ports are identical but independent. Each synchronous serial port can operate at upto one-fourth the machine cycle rate (CLKOUT). The synchronous serial port transmitter and receiver are double buffered and individually controlled by maskable external interrupt signals. Data is framed either as bytes or as words.

10.12.8 Buffered Serial Ports

A buffered serial port (BSP) is a synchronous serial port that is enhanced with an autobuffering unit and is clocked at the full CLKOUT rate. It is full-duplexed and double-buffered to offer flexible data stream length. The autobuffering unit supports high-speed transfers and reduces the overhead of servicing interrupts.

10.12.9 TDM Serial Ports

A time-division multiplexed (TDM) serial port is a synchronous serial port that is enhanced to allow time-division multiplexing of the data. It can be configured for either synchronous operations or for TDM operations and is commonly used in multiprocessor applications.

EXTERNAL BUS INTERFACE 10.13

The ′54X can address upto 64K words of data memory, 64K words of program memory (8M words in the ′548/9/02/10/20), and 64K words of 16-bit parallel I/O ports. Accesses to either external memory or

I/O ports take place through the external interface. Individual space-select signals, $\overline{DS}$, $\overline{PS}$ and $\overline{IS}$ allow the selection of physically separate spaces. The interface's external ready input signal and software-generated wait states allow the processor to interface with memory and I/O devices of many different speeds. The interface's hold modes allow an external device to take control of the ′54X buses; in this way, an external device can access the resources in the program, data and I/O spaces. External memory can be accessed by most ′54X instructions. However, accessing I/O ports requires the use of special instructions: PORTR and PORTW.

DATA-ADDRESSING 10.14

The ′54X offers seven basic data addressing modes:

Immediate Addressing uses the instruction to encode a fixed value.

Absolute Addressing uses the instruction to encode a fixed address.

Accumulator Addressing uses accumulator A to access a location in program memory as data.

Direct Addressing uses 7 bits of the instruction to encode the lower 7 bits of an address. The 7 bits are used with the data page pointer (DP) or the stack pointer (SP) to determine the actual memory address.

Indirect Addressing uses the ARs to access memory.

Memory-Mapped Register Addressing uses the memory-mapped registers without modifying either the current DP value or the current SP value.

Stack Addressing manages adding and removing items from the system stack. During the execution of instructions using direct, indirect or memory-mapped register addressing, the data-address generation logic (DAGEN) computes the addresses of data-memory operands.

PROGRAM ADDRESS GENERATION LOGIC (PAGEN) 10.15

Program memory is usually addressed on a ′54X device with the program counter (PC). With some instructions, however, absolute addressing may be used to access data items that have been stored in program memory. The PC, which is used to fetch individual instructions, is loaded by the program-address generation logic (PAGEN). Typically, the PAGEN increments the PC as sequential instructions are fetched. However, the PAGEN may load the PC with a non-sequential value as a result of some instructions or other operations. Operations that cause a discontinuity include branches, calls, returns, conditional operations, single-instruction repeats, multiple-instruction repeats, reset and interrupts. For calls and interrupts, the current PC is saved onto the stack, which is referenced by the SP.

When the called function or interrupt service routine is finished, the PC value that was saved is restored from the stack via a return instruction.

APPENDIX 10

54X INSTRUCTION SET SUMMARY AND TRANSLATION FROM 5X TO 54X INSTRUCTIONS

A10.1

TMS320C54X	*Instruction Summary*
ABDST	Absolute distance
ABS	Absolute value of accumulator

ADD	Add to accumulator
ADDC	Add to accumulator with carry
ADDM	Add long-immediate value to memory
ADDS	Add to accumulator with sign-extension suppressed
AND	And with accumulator
ANDM	And memory with long immediate
B[D]	Branch unconditionally
BACC[D]	Branch to location specified by accumulator
BANZ[D]	Branch on auxiliary register not zero
BC[D]	Branch conditionally
BIT	Test bit
BITF	Test bit field specified by immediate value
BITT	Test bit specified by TREG
CALA[D]	Call subroutine at location specified by accumulator
CALL[D]	Call unconditionally
CC[D]	Call conditionally
CMPL	Complement accumulator
CMPM	Compare memory with long immediate
CMPR	Compare auxiliary register with ARO
CMPS	Compare select max and store
DADD	Double precision/dual mode add to accumulator
DADST	Double precision load with TREG add/dual 16-bit load with TREG add/subtract
DELAY	Memory delay
DLD	Long word load to accumulator
DRSUB	Double precision/dual 16-bit subtract from long word
DSADT	Long load with TREG add/dual 16-bit load with TREG subtract/add
DST	Store accumulator in long word
DSUB	Double precision/dual 16-bit subtract from accumulator
DSUBT	Long-word load with TREG subtract/dual 16-bit load with TREG subtract
EXP	Accumulator exponent
FIRS	Symmetrical finite impulse response filter
FRAME	Stack pointer immediate offset
IDLE	Idle until interrupt
INTR	Software interrupt
LD	Load accumulator with shift
LD	Load TREG/DP/ASM/ARP
LDM	Load memory-mapped register
LD||MAC[R]	Multiply accumulate with/without rounding and parallel load
LD||MAS[R]	Multiply subtract with/without rounding and parallel load
LDR	Load memory value in accumulator high with rounding
LDU	Load unsigned memory value
LMS	Least mean square
LTD	Load TREG and insert delay
MAC[R]	Multiply accumulate with/without rounding

MACA[R]	Multiply by accumulator A and accumulate
MACD	Multiply by program memory and accumulate with delay
MACP	Multiply by program memory and accumulate
MACSU	Multiply signed by unsigned and accumulate
MAR	Modify auxiliary register
MAS[R]	Multiply and subtract
MASA[R]	Multiply by accumulator A and subtract
MAX	Accumulator maximum
MIN	Accumulator minimum
MPY[R]	Multiply
MPYA	Multiply by accumulator A
MPYU	Multiply unsigned
MVDD	Move data from data memory to data memory with X, Y addressing
MVDK	Move data from data memory to data memory with destination addressing
MVDM	Move data from data memory to memory-mapped register
MVDP	Move data from data memory to program memory
MVKD	Move data from data memory to data memory with source addressing
MVMD	Move data from memory-mapped register to data memory
MVMM	Move data from memory-mapped register to memory-mapped register
MVPD	Move data from program memory to data memory
NEG	Negate accumulator
NOP	No operation
NORM	Normalisation
OR	OR with accumulator
ORM	OR memory with constant
POLY	Polynomial evaluation
POPD	Pop top of stack to data memory
POPM	Pop top of stack to memory-mapped register
PORTR	Read data from port
PORTW	Write data to port
PSHD	Push data-memory value onto stack
PSHM	Push memory-mapped register onto stack
RC[D]	Return conditionally
READA	Read data memory addressed by accumulator A
RESET	Software reset
RET[D]	Return
RETE[D]	Enable interrupts and return from interrupt
RETF[D]	Enable interrupts and fast return from interrupt
ROL	Rotate accumulator left
ROLTC	Rotate accumulator left using TC
ROR	Rotate accumulator right
RPT	Repeat next instruction
RPTB[D]	Block repeat
RPTZ	Repeat next instruction and clear accumulator

RSBX	Reset status register bit
SACCD	Store accumulate conditionally
SAT	Saturate accumulator
SFTA	Shift accumulator arithmetically
SFTC	Shift accumulator conditionally
SFTL	Shift accumulator logically
SQDST	Square distance
SQUR	Square
SQURA	Square and accumulate
SQURS	Square and subtract
SRCCD	Store block repeat counter conditionally
SSBX	Set status register bit
ST	Store TREG, TRN, or immediate value into memory
STH	Store accumulator high into memory
STL	Store accumulator low into memory
STLM	Store accumulator low into memory-mapped register
STM	Store immediate value into memory-mapped register
ST||ADD	Store accumulator with parallel add
ST||LD	Store accumulator with parallel load
ST||MAC[R]	Parallel store and multiply accumulator with/without rounding
ST||MAS[R]	Parallel store and multiply and subtract
ST||MPY	Parallel store and multiply
ST||SUB	Parallel store and subtract
STRCD	Store TREG conditionally
SUB	Subtract from accumulator
SUBB	Subtract from accumulator with borrow
SUBC	Subtract conditionally
SUBS	Subtract from accumulator with sign-extension suppressed
TRAP	Software interrupt
WRITA	Write memory data addressed by accumulator A
XC	Execute conditionally
XOR	Exclusive-or with accumulator
XORM	Exclusive-or memory with constant

*Highlighted instruction are non-repeatable instructions.

A10.2 Instruction Set of C54X

In this appendix, to facilitate the quick understanding of the instruction set of 54X, various registers, bit fields and instructions of 5X and their equivalents in 54X are given. The registers, bit fields and instructions which are new in 54X are also given. Some of the 5X registers, bit fields and instructions don't have an equivalent in 54X. The entries where a "- " appears in the tables in this appendix denote that there is no 54X equivalent for the corresponding 5X register/bit field/instruction.

CPU Register Mapping Table A 10.2.1b gives the list of C5X registers and their equivalent 54X registers. Even where there is an equivalent register, bit positions and/or functionality may change between

′C5X and ′C54X. Table A 10.2. la gives the list of CPU registers which are not in 5X but are present in 54X.

Table A10.2.1a *New ′C54X CPU registers/fields*

′C54X	*′C54X register description*
AG	A Register guard band
BG	B Register guard band
BK	Block size register
SP	Stack pointer
TRN	Transition register
XPC	Extended addressing register (548/9/02/10/20 only)

Table A10.2.1b *CPU register mapping*

′C5X	*′C54X*	*′C5X field description*
ACC	A	Accumulator
ACCB	B	Accumulator buffer
AR0	AR0	Auxiliary register 0
AR1-AR7	AR1-AR7	Auxiliary register 1-7
ARCR	—	Auxiliary register compare register
BMAR	—	Block-move address register.
BRCR	BRC	Block repeat counter register.
CBER1	—	Circular buffer end address 1
CBER2	—	Circular buffer end address 2
CBSR1	—	Circular buffer start address 1
CBSR2	—	Circular buffer start address 2
DBMR	—	Dynamic bit manipulation register
GREG	—	Global memory allocation register
IFR	IFR	Interrupt flag register
IMR	IMR	Interrupt mask register
INDX	—	Indirect addressing index register
PAER	REA	Block repeat end address
PASR	RSA	Block repeat start address
PMST	PMST	Processor-mode-status register
PREG	—	Product register
RPTC	—	Repeat-counter register
STO	STO	Status register 0
ST1	ST1	Status register 1
TREGO	T	Temporary register 0
TREG1	—	Temporary register 1
TREG2	—	Temporary register 2

Peripheral Register Mapping Peripheral Register Mapping between 5X and 54X is given in Table A10.2.2b. The new peripheral registers in 54X are given in Table A 10.2.2a. Peripheral registers differ according to the original and target processor family members. No single processor will contain all registers in Table A 10.2.2a or A10.2.2b.

Table A10.2.2a *New 54X peripheral registers*

'C54X	*'C54X register description*
ARR[0,1]	Address receive register (BSP)
AXR[0,1]	Address transmit register (BSP)
BSCR	Bank switch control register
BDXR[0,1]	Transmit register (BSP)
BDRR[0,1]	Receive register (BSP)
BKR[0,1]	Receive buffer-size register (BSP)
BKX[0,1]	Transmit buffer-size register (BSP)
BSPC [0,1]	Serial port control (BSP)
SPCE [0,1]	Serial port control extension (BSP)

Table A10.2.2b *Peripheral register mapping*

'C5X	*'C54X*	*'C5X field description*
ARR	ARR	Address receive register (BSP)
AXR	AXR	Address transmit register (BSP)
BKR	BKR	Receive buffer-size register (BSP)
BKX	BKX	Transmit buffer-size register (BSP)
BDXR	BDXR	Transmit register (BSP)
BDRR	BDRR	Receive register (BSP)
BSPC	BSPC	Serial port control (BSP)
CWSR	—	Wait-state control register
DRR	DRR	Transmit register(SP)
DXR	DXR	Receive register(SP)
IOWSR	IOWSR	IO wait-state register
HPIC	HPIC	Host port control register
PA[0-15]	[0x50-0x5F]	IN or OUT operand
PDWSR	SWWSR	Software wait-state control register
PRD	PRD	Timer period register
SPC	SPC	Serial port (SP) control
SPCE	SPCE	Serial port control extension (BSP)
TCSR	TCSR	Channel select register (TDM-SP)
TCR	TCR	Timer control register
TDXR	TDXR	Transmit data register (TDM-SP)
TIM	TIM	Timer counter register
TRAD	TRAD	Received address register (TDM-SP)
TRCV	TRCV	Receive data register (TDM-SP)
TRTA	TRTA	RXATX address register (TDM-SP)
TSPC	TSPC	Serial port control (TDM-SP)

CPU Bit Field Mapping The assembler and instruction sets allow for some bit fields to be accessed and modified explicitly as well as implicitly. Table A 10.2.3b gives CPU Bit-field Mapping between 5X and 54X. Table A 10.2.3a gives the list of bit fields which are unique to 54X.

Table A10.2.3a *New 'C54X bit fields*

'C54X	*'C54X field description*
ASM	Accumulator shift mode
C16Dual	16-bit ALU arithmetic mode
CLKOFF	Disable CLKOUT bit
CMPT	ARP compatibility
CPL	Compiler mode
DROM	Data ROM enable
OVB	Overflow flag for B accumulator
SMUL	Saturation on multiplication

Table A10.2.3b *CPU bit-field mapping*

'C5X	*'C54X*	*'C5X field description*
ARB	—	Auxiliary register pointer buffer
ARP	ARP	Auxiliary register pointer
AVIS	AVIS	Address visibility
BRAF	BRAF	Block repeat active flag
C	C	Carry
CAR1	—	Circular buffer auxliary register 1
CAR2	—	Circular buffer auxiliary register 2
CENB1	—	Circular buffer enable 1
CENB2	—	Circular buffer enable 2
CNF	—	On-chip RAM configuration
DP	DP	Data page pointer
HM	HM	Hold mode bit
% INTM	INTM	Global interrupt mask bit
IPTR	IPTR	Interrupt vector table pointer
MPNMC	MPNMC	Microprocessor/Microcontroller
NDX	—	Enable INDX register
OV	OVA	Overflow flag
OVLY	OVLY	Internal RAM overlay
OVM	OVM	Overflow mode
PM	FRCT	Product mode PM = 0,1 only
RAM	—	Program RAM enable
SXM	SXM	Sign extension mode
TC	TC	Test control bit
TRM	—	Enable multiple T registers
XF	XF	External flag

Data, Program and I/O Addressing Modes In the assembly language, the symbols used for indicating the addressing modes/operand addresses in 5X and their equivalents in 54X assembly language are given in Table A10.2.4b. Table A10.2.4a gives the syntax for the new 54X addressing modes/operands.

Table A10.2.4a *Syntax for new 'C54X data and program addressing modes*

'C54X	*'C54X addressing description*
*SP0	Stack pointer relative
*+ARx.	Indirect with preincrementation
*ARx-%	Indirect with modulo circular addressing
*ARx +%	Indirect with modulo circular addressing
*ARx -0%	Indirect with modulo circular addressing and offset
*ARx +0%	Indirect with modulo circular addressing and offset
*ARx (Ik)	Indirect with long immediate offset and no modify
*+ARx(lk)	Indirect with long immediate offset and premodify
*+ARx (lk)%	Indirect with long immediate offset, premodify and circular
Xmem,Ymem	1 or 2 indirect operands per instruction
*(lk)	Absolute addressing
[XPC]	Far program addressing

Table A10.2.4b *Data, program and I/O addressing modes mapping*

'C5X	*'C54X*	*'C5X addressing mode*
Dma	Dma	Direct
Dma,shift	dma,shift	Direct with shift
*	*AR[ARP]	Indirect
*+	*AR[ARP]+	Indirect with increment modify
*–	*AR[ARP]-	Indirect with decrement modify
*0+	*AR[ARP]+0	Indirect with index modify
*0-	*AR[ARP]-0	Indirect with index modify
*BR0+	*AR[ARP]+0B	Indirect with bit-reverse modify
*BR0-	*AR[ARP]-0B	Indirect with bit-reverse modify
*,shift	*AR[ARP]	Shift indirect with shift
*,ARP	*AR[ARP]	Indirect with ARP modify
*,shift	ARP*AR[ARP]	Shift indirect with shift and ARP modify
#k	#k	Short immediate (8, 9, 13 bit)
#lk	#lk	Long immediate (16 bit)
MMR	MMR	Memory mapped register
Dmad	Dmad	Data memory address
[ACC]	[A]	Accumulator program addressing
Pmad	Pmad	Program memory address
PAx.	PA	Port address

Conditional Code Mapping The combinations of conditional codes that can be used on the ′C5X are more flexible than that of the ′C54X (i.e. mixing control and signed conditions). If this ′C54X criteria is breached, then an error will be reported by the assembler. Table A10.2.5b gives the mapping between 5X and 54X condition codes. Table 10.2.5a gives the list o^f new condition codes in 54X.

Table A10.2.5a *New ′C54X conditional codes*

′C54X	*′C54X condition description*
BOV	Overflow detected (B)
BNOV	No Overflow detected (B)
BEQ	B = 0
BNEQ	B <> O
BLT	B < 0
BLEQ	B <= 0
BGT	B > 0
BGEQ	B => 0
NBIO	BIO signal high

Table A10.2.5b *Conditional code mapping*

′C5X	*′C54X*	*′C5X condition description*
EQ	AEQ	ACC = 0
NEQ	ANEQ	ACC <> 0
LT	ALT	ACC < 0
LEQ	ALEQ	ACC <= 0
GT	AGT	ACC > 0
GEQ	AGEQ	ACC => 0
C	C	Carry = 1
NC	NC	Carry = 0
OV	AOV	Overflow detected
NOV	ANOV	No Overflow detected
IO	BIO	BIO signal low
TC	TC	Test control = 1
NTC	NTC	Test control = 0
UNC	UNC	Unconditional

Accumulator Source Instructions Table A10.2.6b gives the list of instructions in 5X that do not use Data or Program memory as the source of the operation. The accumulators are the only source.

Table A10.2.6a *New ′C54X accumulator source instructions*

′C54X	*′C54X instruction description*
ADD [ASM]	Add with fixed shift or using ASM register
AND	AND with fixed shift

(Contd.)

Table A10.2.6a *(Contd.)*

LD [ASM]	Load with fixed shift or using ASM register
OR	OR with fixed shift
RND	Round accumulator (2^{15})
ROLTC	Rotate register left with TC shifted into LSB
SAT	Saturate accumulator
SFTC	Shift register left if 2 sign bits
SUB [ASM]	Sub with fixed shift or using ASM register
XOR	XOR with fixed shift

Table A10.2.6a gives the list of new instructions of 54X with the accumulator as the source.

Table A10.2.6b *Accumulator source instructions*

'C5X	*'C54 X*	*'C5 X instruction description*
ABS	ABS A	Absolute value of ACC
ADCB	ADDC/ADD	ACC = ACC + ACCB + C
ADDB	ADDA, B, A	ACC = ACC + ACCB
ANDB	ANDA,B,/A	ACC = ACCB & ACC
BSARk(k=1...16)	LDA, -k, A	ACC ≫ k (barrel shift)
CMPL	CMPL A	Complement ACC
CRGT	MAX A	ACC = Max (ACC, ACCB), set C
CRLT	MINA	ACC = Min (ACC, ACCB), set C
EXAR	—	Exchange ACCB and ACC(3)
LACB	LDB,A	ACC = ACCB
NEG	NEC A	Negate ACC
NORM *[+/-]	EXPA	
NORM A	Normalise ACC	
ORB	OR A, B,	A OR ACCB with ACC
ROL	ROL A	Rotate ACC ≪ 1
ROLB	ROL, ROL	Rotate ACCB and ACC ≪ 1
ROR	ROR A	Rotate ACC ≫ 1
RORB	ROR, ROR	Rotate [ACCB I ACC] ≫ 1
SACB	LDA, B	ACCB = ACC
SATH	—	ACC ≫ 16 if T[4:4] = 1
SATL	—	ACCL ≫ T[3:0]
SBB	SUBB,A	ACC = ACC - ACCB
SBBB	SUBB/SUB	ACC = ACC - ACCB -B
SFL	SFTLA,1	ACC ≪ 1
SFLB	SFTL, ROL	[ACC I ACCB] ≪ 1
SFR	SFTA, SFTA	ACC ≫ 1
SFRB	SFTA, SFTA,	ROL [ACC I ACCB] ≫ 1
XORB	XORB, A	ACC = ACCB XOR ACC
ZAP	LD # 0, A	ACC = P = 0

Accumulator and Memory Source Instructions Instructions of 5X and its near equivalents in C54X that use accumulators, program and data memory as sources are given in Table A10.2.7b. The new instructions of 54X are given in Table A10.2.7a.

Table A10.2.7a *New ′C54X accumulator and memory source instructions*

′C54X	*′C54X instruction description*
ABDST Xmem, Ymem	AB5 distance of 2 memory values
ADD Xmem, Ymem	Add 2 data memory operands
BIT	Test bit in memory location
CMPS	Compare, select, store
DADD	32-bit add with memory
DADST	32-bit add/sub with T
DLD	Load 32-bit value
DRSUB	32-bit reverse sub with memory
DSADT	32-bit sub/add with T
DST	32-bit store to memory
DSUB	32-bit sub with memory
DSUBT	32-bit sub with T
LD (ASM)	Load with ASM shift
SACCD	Conditional store of accumulator
SRCCD	Conditional store of BRC
ST || LD	Parallel store, load
ST || ADD	Parallel store, add
ST || SUB	Parallel store, sub
STH/L (ASM)	Store with ASM shift
STRCD	Conditional store of T
SUB Xmem, Ymem	Sub 2 data memory operands

Table A10.2.7b *Accumulator and memory source instructions*

′C5X	*′C54X*	*′C5X instruction description*
ADD dma[,shift]	ADD dma[,shift], A	ACC+=dma[≪ shift]
ADD #k	ADD #lk, A	ACC +=lk
ADD #lk[,shift]	ADD #lk[,shift],A	ACC +=lk [≪ shift]
ADD dma,16	ADD dma,16, A	ACC +=dma ≪ 16
ADDC dma	ADDC dma, A	ACC +=(dma + C)
ADDS dma	ADDS dma, A	ACC +=(unsigned)dma
ADDT dma	ADDT dma,TS, A	ACC +=dma ≪ T
AND dma	AND dma, A	ACC =dma & ACC
AND #lk[,shift]	AND #lk[,shift],A	ACC =lk[,≪shift]&ACC
AND #lk,16	AND #lk,16, A	ACC =lk≪16l ACC
LACC dma[,shift]	LD dma[,shift], A	ACC =dma [«shift]

(*Contd.*)

Table A10.2.7b (Contd.)

LACC #lk[,shift]	LD #lk[,shift],A	ACC =lk[≪ shift]
LACC dma,16	LD dma,16, A	ACC =dma ≪ 1 6
LACL #k	LD #k, A	ACCL =k
LACL dma	LDU dma, A	ACCL =(dma + C)
LACT dma	LD dmaJS, A	ACC =dma ≪ T
LAMM mmr	LDM mmr, A	. ACC =mmr
OR dma	OR dma, A	ACC =dma I ACC
OR #lk[,shift]	OR #lk[,shift],A	ACC =lk[,≪ shift] I ACC
OR #lk,16	OR #lk,16	ACC =lk≪16 I ACC
SACH dma[,shf]	STHA, dma[,shift]	Dma = ACCH≪ shf
SACL dma[,shf]	STLA, dma[,shift]	Dma = ACCL≪ shf
SAMM mmr	STLM A,mmr	mmr = ACCL
SUB dma[,shift]	SUB dma[,shift],A	ACC= dma [≪ shift]
SUB #k	SUB #lk, A	ACC= lk
SUB #lk[,shift]	SUB #lk[,shift],A	ACC= lk [« shift]
SUB dma,16	SUB dma,16, A	ACC= dma ≪ 1 6
SUBB dma	SUBB dma, A	ACC= (dma + C)
SUBC dma	SUBC dma, A	ACC= dma (conditional)
SUBS dma	SUBS dma, A	ACC= (unsigned)dma
SUBT dma	SUB dmaJS, A	ACC= dma ≪ T
XOR dma	XOR dma, A	ACC= dmaXORACC
XOR #lk[,shift]	XOR #lk[,shift],A	ACC= lk[,≪ shift] XORACC
XOR #lk,16	XOR #lk,16,A	ACC= lk≪ 16XORACC
ZALR dma	LDR dma, A	ACC= 0, ACCH=dma

Auxiliary Register and Data Page Pointer Instructions

Table A10.2.8a Auxiliary register and data page pointer instructions

'C5X	'C54X	'C5X instruction description
ADRKk	MAR*+AR[ARP](+k)	Add short immediate to AR[ARP]
CMPR [0,1,2,3]	CMPR #10,1,2,3], AR[ARP]	Compare AR[ARP] with ARCR
LAR ARx, dma	MVDK dma,Arx	Load ARx. from memory
LAR ARx, #k	STM #k, Arx	Load ARx. with short immed.
LARARx, #lk	STM #lk, Arx	Load ARx. with long immed.
LDP dma	LDdma, DP	Load DP from memory
LDP#k	LD#k, DP	Load DP with 9bit immediate
MAR *, ARP	—	Modify ARP
MAR *[+/-] [,ARP]	MAR*AR[ARP][+/-]	Modify auxiliary register
SAR ARx, dma	MVKD ARx, dma	Store Auxiliary register to mem.
SBRKk	MAR*+AR[ARP](-k)	Sub short immediate from AR[ARP]

T Register, P Register and Multiply Instructions

Table A10.2.9a *T register, P register and multiply instructions*

'C5X	'C54X	'C5X Instruction Description
APAC	—	Add PREG to ACC(1)
LPH dma	—	Load PREG from data mem(1)
LT dma	LD dma, T	Load T from data mem
LTA dma	LD dma, T	Load T from data mem & ACC +=P
LTD dma	LD dma, T	Load T from data mem & ACC +=P & dmov
LTP dma	LD dma, T	Load T from data mem & ACC=P
LTS dma	LD dma, T	Load T from data mem & ACC-=P
MAC pma, dma	MACP dma, pma, A	Multiply & accumulate
MACD pma, dma	MACD dma, pma, A	Multiply & accumulate & dmov
MADD dma	—	Multiply & accumulate & dmov using BMAR(2)
MADS dma	—	Multiply & accumulate & dmov using BMAR(2)
MPY dma	MPY dma, A	Multiply signed
MPY#k	MPY#lk, A	Multiply signed with short immed.
MPY#lk	MPY#lk, A	Multiply signed with long immed.
MPYA dma	MPY dma, A	Multiply & accumulate
MPYS dma	MPY dma, A	Multiply & accumulate
MPYU dma	MPYU dma, A	Multiply unsigned
PAC	—	ACC = PREC(1)
SPAC	—	ACC-=PREG(1)
SPH dma	—	Store PREG hi to data mem.(1)
SPL dma	—	Store PREG lo to data mem.(1)
SPM0	RSBX FRCT	PREG shift count = 0
SPM 1	SSBX FRCT	PREG shift count = 1
SPM [2,3]	—	PREG shift count = 4,-6
SQRAdma	SQURdma,A	Square & accumulate
SQRSdma	SQURdma,A	Square & accumulate
ZPR	—	Zero PREG(1)

Note. (1) *PREG is invalid register for 54X.*
(2) *BMAR is invalid register for 54X.*

Table A10.2.9b *New 'C54X T register, P register and multiply instructions*

'C54X	'C54X instruction description
FIRS	Symmetrical filter operation
LD || MAC	Load and MAC
LD || MACR	Load and MACR
LD || MAS	Load and MAS
LD || MASR	Load and MASR
LMS	Least mean squares filter

(Contd.)

Table A10.2.9b *(Contd.)*

MACR	Multiply/accumulate with rounding
MACA[R]	Multiply/accumulate with AH as input [& rounding]
MACSU	Multiply/accumulate signed/unsigned
MASR	Multiply/accumulate with rounding
MASA[R]	Multiply/accumulate with AH as input [& rounding]
MPYA	Multiply with AH as input
MPYR	Multiply & rounding
POLY	Polynomial operation
SQDST	Square distance
SQUR	Square with A register input
SQUR[A,S]	Square with accumulate
ST || MAC	Store and MAC
ST || MACR	Store and MACR
ST || MAS	Store and MAS
ST || MASR	Store and MASR
ST || MPY	Store and MPY

Branch, Call and Return Instructions

Table A10.2.10a *Branch, call and return instructions*

C5X	*'C54X*	*'C5X instruction description*
B pma	B pma	Branch direct translation
B pma, *[+/-][,ARP]	BD pma	Branch with AR update
BACC	BACC	Branch on ACC
BACCD	BACCD	Delayed branch on ACC
BANZ pma	BANZ pma,*ARx-	ARx conditional branch
BANZ pma,*[+/-], ARP	BANZ pma,*ARx[+/-]	ARx conditional branch
BANZD pma	BANZ pma,*ARx-	ARx conditional delayed branch
BANZD pma, *[+/-], ARP	BANZ pma,*ARx[+/-]	ARx conditional delayed branch
BCND pma,conds	BC pma, conds	Conditional branch
BCNDD pma,conds	BCD pma, conds	Conditional delayed branch
BD pma	BD pma	Delayed branch
BD pma[,*,ARP]	BD pma	Delayed branch with AR update
CALA	CALA A	Call on ACC
CALAD	CALAD A	Delayed call on ACC
CALL pma	CALL pma	Call
CALL pma*[+/-][,ARP]	CALLD pma	Call with AR update
CALLD pma	CALLD pma	Delayed call
CALLD pma[,*,ARP]	CALLD pma	Delayed call with AR update
CC pma, conds	CC pma, conds	Conditional call

(Contd.)

Table A10.2.10a *(Contd.)*

CCD pma, conds	CCD pma, conds	Conditional delayed call
INTRk	INTRk+15	Software interrupt
NMI	INTR1	Non-maskable interrupt
RET	RET	Return
RETC conds	RC conds	Conditional return
RETD	RETD	Delayed return
RETE	RETE	Return from interrupt with enable
RETI	RET	Return from interrupt
TRAP	TRAP 2	Software TRAP

Table A10.2.10b *New 'C54X branch, call, return instructions*

'C54X	*'C54X instruction description*
FB[D]	Far branch (548/9)
FBACC[D]	Far branch on ACC(548/9)
FCALA[D]	Far call on ACC(548/9)
FCALL[D]	Far call (548/9)
FRET[D]	Far return (548/9)
FRETE[D]	Far return with INTM enable (548/9)
RESET	Software reset
RETF[D]	Fast return

Program Control Instructions Instructions that modify the program counter in a non-sequencial manner, but which does not use branch instructions.

Table 10.2.11a *New 'C54X program control instructions'*

C54X	'C54X instruction description
FRAME	Modify stack pointer by immediate

Table A10.2.11b *Program control instructions*

'C5X	*'C54X*	*'C5X instruction description*
POP	POPM AL	Pop low ACC from stack (1)
POPD dma	POPD dma	Pop data memory from stack
PSHD dma	PSHD dma	Push data memory to stack
PUSH	PUSHMAL	Push low ACC to stack
RPT #k	RPT #k	Single repeat(short Imm.)
RPT #lk	RPT #lk	Single repeatdong Imm.)
RPT dma	RPT dma	Single repeat(dma)
RPTB	RPTB	Block repeat
RPTZ #lk	RPTZ A, #lk	Single repeat with ACC clear
XC 1,conds	XC 1,conds	Execute conditional 1
XC 2, conds	XC 2,conds	Execute conditional 2

Note. (1) *Stack for 'C54Xis RAMbased and must have SP initialized.*

I/O and Data Memory Operations

Table A10.2.12a *I/O and data memory operations*

′C5X	*′C54X*	*′C5X instruction description*
BLDD #addr, dma	MVKD #addr, dma	Data to data
BLDD dma, #addr	MVDK dma, #addr	Data to data reversed
BLDDBMAR, dma	—	Data to data using BMAR (1)
BLDD dma, BMAR	—	Data to data using BMAR reversed (1)
BLDP dma	—	Data to program using BMAR (1)
BLPD #pma, dma	MVPD pma, dma	Program to data
BLPD BMAR,dma	—	Program to data using BMAR (1)
DMOV dma	DELAY dma	Data move
IN dma, PA	PORTR PA, dma	I/O port to data
LMMR mmr, #addr	MVDM #addr, mmr	Data to memory-mapped register
LMMR *, #addr	MVDM #addr, *AR[ARP]	Data to memory-mapped register
OUT dma, PA	PORTW dma, PA	Data to I/O port
SMMR mmr, #addr	MVMD mmr, #addr	Memory-mapped register to data
SMMR *, #addr	MVMD AR[ARP],#addr	Memory-mapped register to data
TBLR dma	READA dma	Program to data using ACC
TBLW dma	WRITA dma	Data to program using ACC

Note. (1) *BMAR is invalid register for 54X.*

Table A10.2.12b *New ′C54X I/O and data memory operations*

′C54X	*′C54X instruction description*
ADDM	Add long immediate to data memory
MVDD	Move Xmem to Ymem
MVMM	Move Mmr to Mmr

Miscellaneous Control Instructions

Table A10.2.13a *Miscellaneous control instructions*

′C5X	*′C54X*	*′C5X instruction description*
BIT dma, bit	BITF dma, #1«(15-bit)	Test bit (immediate)
BITT dma	BITT dma	Test bit (TREG)
CLRC bit	RSBX bit	Clear bit
IDLE	IDLE1	Idle
IDLE2	IDLE 2	Idle 2 (low-power mode)
LST #0, dma	MVDM dma, STO	Load status register
LST #1, dma	MVDM dma, ST1	Load status register
LST #0, *[+/-...]	MVDK*ARx[+/-], STO	Load status register
LST #1, *[+/-...]	MVDK*ARx[+/-], ST1	Load status register
NOP	NOP	No operation
SETC bit	SSBXbit	Set bit
SST #0, dma	MVMD STO, dma	Store status register
SST #1, dma	MVMDST1, dma	Store status register
SST #0, *[+/-...]	MVKDST0,*ARx [+/-..]	Store status register
SST #1, *[+/-...]	MVKDST1,*ARx[+/-..]	Store status register

Review Questions

10.1 What are the functional blocks present in 54X but not in 5X? Explain the function performed by each of them.

10.2 Compare the multiplier units in 54X and 5X.

10.3 Compare the address and data buses of 54X and 5X.

10.4 The data stored in data memory is 16 bits long. But the multiplier requires 17 bit data. Explain how the MSB is generated?

10.5 Explain the two ways in which the accumlator may be loaded with the most positive value or most negative value when overflow occurs.

10.6 Explain how the rounding operation is carried by the adder/multiplier units of 54X.

10.7 What is the use of guard bits in accumulators?

10.8 What is the use of the following registers of 54X?

10.9 (a) T (b) TRN (c) BIC (d) XPC

10.10 Explain the operation of CSSU of 54X and explain its use with an application.

10.11 Explain the operation of the exponent encoder in 54X.

Self Test Questions

10.1 No. of Auxiliary Register ALU(s) in 54X is ——— and number of data buses which can be used for reading data from data memory is ———.
(a) 1, 1 (b) 1, 2 (c) 2, 1 (d) 2, 2

10.2 Which of the following are available in 54X but not in 5X?
(a) SP (b) 16 bit timer (c) XPC (d) 8-bit HPI

10.3 Number of data bus in 54X is ———.
(a) 1 (b) 2 (c) 3 (d) 4

10.4 Number of address bus in 54X is ———.
(a) 1 (b) 2 (c) 3 (d) 4

10.5 is used to store the result of adder units in the ALU of 54X.
(a) Accumlator A (b) Accumlator B
(c) Either A or B (d) T Register

10.6 ——— is used to store the result of multiplier units in the ALU of 54X.
(a) Accumlator A (b) Accumlator B
(c) Either A or B (d) PREG
(e) T register

10.7 In the LD || MAC parallel instruction, the register where the MAC result is stored is ———.
(a) A (b) B (c) A or B (d) T register

10.8 Bits 32-16 of ——— can be used as an input to the multiplier in 54X.
(a) A (b) B
(c) Either A or B (d) Neither A or B

10.9 Which of the following registers are present in 5X but not in 54X?
(a) PC (b) SP (c) PREG (d) INDX

10.10 The max no. of wait states that software wait-state generator can produce is ———.
(a) 1 (b) 4 (c) 7 (d) 8

11 TMS320C54X ASSEMBLY LANGUAGE INSTRUCTIONS

The ′54X has a number of instructions and addressing modes which are tailored for various DSP applications such as FFT computation, symmetric FIR filtering, adaptive filter (least mean square algorithm) and Viterbi decoding. While some of the instructions and registers of its predecessors such as ′2X, ′24X and ′5X are retained, some additional registers, instructions and addressing modes for both data and program are introduced in ′54X. To facilitate a quick overview and understanding of the instructions of ′54X with its predecessor, a summary of the instructions of ′54X and the comparison of ′5X registers, bit fields and instruction and their equivalent in ′54X are given in Appendix 10. A list of the new registers, bit fields and instructions in ′54X is also given in Appendix 10. In this chapter details on the various addressing modes for the operands in the ′C54X instructions and a brief account of the new instructions and instruction syntax in ′C54X is presented. Additional details on these topics can be obtained from TMS320C54X DSP CPU and Peripherals user manual.

DATA ADDRESSING IN ′C54X 11.1

C54X offers seven basic addressing modes:

Immediate Addressing
Absolute Addressing
Accumulator Addressing
Direct Addressing
Indirect Addressing
Memory-Mapped Register Addressing
Stack Addressing

11.1.1 Immediate Addressing

In immediate addressing, the operand required for an instruction is specified in the instruction word itself. The number of bits of the operand can be 3, 4, 8 or 9 bits in length in short immediate addressing requiring 1 instruction word. In long immediate addressing the operand is 16 bits long and the instruction word is of length 2. Table 11.1 lists the ′54X instructions in which the operand may be specified using immediate addressing. In this table length of the operand for each type of instruction is also specified. The immediate addressing is indicated by using the hash (No.) symbol (#) immediately preceding the value or symbol to indicate that it is an immediate value. For example, to load accumulator A, with the value 55h in hexadecimal, the instruction syntax used is LD #55h, A

Table 11.1 *C54X instructions using immediate addressing modes and the operand length*

3&4-bit constant	*8-bit constant*	*9-bit constant*	*16-bit constant*
LD	FRAME LD	ADD	ORM
	LD	ADDM	RPT
	RPT	AND	RPTZ
		ANDM	ST
		BITF	STM
		CMPM	SUB
		LD	XOR
		MAC	XORM
		OR	

11.1.2 Absolute Addressing

In the absolute addressing mode the complete address of the operand is explicitly specified in the instruction word itself. In this addressing mode, the content of neither the data page pointer/stack pointer nor any of the registers including the accumulators and ARs are used for finding the address. Absolute addresses are always encoded with a length of 16 bits, so instructions that encode absolute addresses are always at least two words in length. There are four types of absolute addressing:

Dmad addressing

- MVDK Smem, dmad
- MVDM dmad, MMR
- MVKD dmad, Smem
- MVMD MMR, dmad

pmad addressing

- FIRS Xmem, Ymem, pmad
- MACD Smem, pmad, src
- MACP Smem, pmad, src
- MVDP Smem, pmad
- MVPD pmad, Smem

PA addressing

- PORTR PA, Smem
- PORTW Smem, PA

***(lk) addressing**

- This is used with all instructions that support the use of a single data-memory (Smem) operand.

11.1.2.1 Dmad Addressing

Data-memory address (dmad) addressing uses a specific value to specify an address in data space. The syntax for dmad addressing uses a symbol or a number to specify an address in data space. For example, to copy the value contained at the address labeled data1 in data space to the memory location in data space pointed to by AR3, the syntax used is MVKD datal, *AR3. In this example, the address referenced by datal is the dmad value.

11.1.2.2 pmad Addressing

Program-memory address (pmad) addressing uses a specific value to specify an address in program space. The syntax for pmad addressing uses a symbol or a number to specify an address in program space. For example, to copy a word in the program-memory location labeled COEFF to a data-memory location specified by AR7, the instruction syntax is MVPD COEFF, *AR7-. In this example, the address referenced by COEFF is the pmad value.

11.1.2.3 PA Addressing

Port address (PA) addressing uses a specific value to specify an external I/O port address. The syntax for PA addressing uses a symbol or a number to specify the port address. For example, to copy a value from the I/O port at port address FIFO to a data-memory location pointed to by AR5, the instruction used is PORTR FIFO, *AR5. In this example, FIFO refers to the port address.

*11.1.2.4 *(lk)Addressing*

*(lk) addressing uses a specific value to specify an address in data space. The syntax for *(lk) addressing uses a symbol or a number to specify an address in data space. For example, to load accumulator A with the value contained in address BUFFER in data space, the instruction used is LD *(BUFFER), A. The syntax for *(lk) addressing allows all instructions that use Smem addressing to access any location in data space without changing the DP or initialising an AR. When this form of absolute addressing is used, the length of the instruction is extended by one word. For example, a l-word instruction would become a 2-word instruction or a 2-word instruction would become a 3-word instruction. The addition of one word to an instruction affects its usability in delay slots. Instructions using the *(lk) form of absolute addressing cannot be used with repeat single instructions (RPT, RPTZ).

11.1.3 Accumulator Addressing

Accumulator addressing uses the accumulator content as an address. This addressing mode is used to address program memory as data. Two instructions allow the accumulator to be used as an address: READA Smem and WRITA Smem. READA transfers a word from a program-memory location specified by accumulator A to a data-memory location specified by the single data-memory (Smem) operand of the instruction. WRITA transfers a word from a data-memory location specified by the Smem operand of the instruction to a program-memory location specified by accumulator A. In repeat mode, an increment may be used to increment accumulator A.

In most ′54X devices, the program-memory location is specified by the lower 16 bits of accumulator A. However, because the 548/9/02/10/20 has 23 address lines, the program-memory location in an 548/9/02/10/20 is specified by the lower 23 bits of accumulator A.

11.1.4 Direct Addressing

In direct addressing mode, the instruction contains the lower seven bits of the data-memory address (dma). The 7-bit dma is an address offset that is combined with a base address, with the data-page pointer (DP) or with the stack pointer (SP) to form a 16-bit data-memory address. Using this form of addressing, any of the 128 locations in a page can be accessed in random order without changing the DP or the SP. Direct addressing is not the only method of offset addressing. However, the advantage of this mode is that it encodes each instruction and address into a single word. Either DP or SP can be combined with the dma offset to generate the actual address. The compiler mode bit (CPL), located in status register ST1, selects which method is used to generate the address: When CPL = 0, the dma field is concatenated with the 9-bit DP field to form the 16-bit data-memory address. When CPL = 1, the dma field is added (positive offset) to SP to form the 16-bit data-memory address. The syntax for direct addressing uses a symbol or a number to specify the offset value. For example, to add the contents of the memory location VALUE1 to accumulator B, provided that the correct base address is in DP (CPL = 0) or SP (CPL = 1), the instruction to be used is ADD VALUE 1, B. The lower seven bits of the address of VALUE 1 are stored in the instruction word. Figure 11.1 shows the opcode format for instructions that use direct addressing. Table 11.2 describes the bits of the direct-addressing instruction. Figure 11.2 illustrates how the 16-bit data address is formed.

15 - 8	7	6 - 0
Opcode	*I* = 0	dma

Fig. 11.1 *Direct-addressing instruction format*

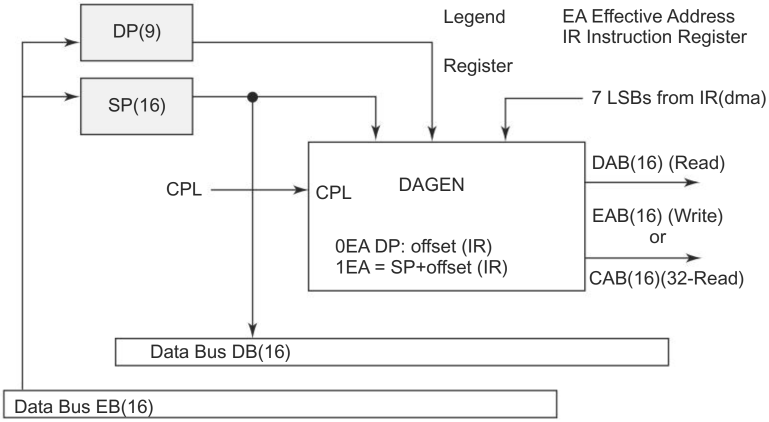


Fig. 11.2 *Direct addressing block diagram*

Table 11.2 *Direct-addressing instruction bit summary*

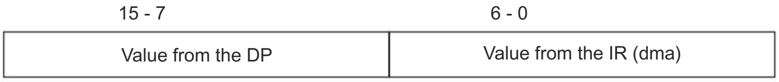
Bit	*Name*	*Function*
15-8	Opcode	This 8-bit field contains the operation code for the instruction
7	I = 0	The addressing mode used by the instruction is the direct addressing mode
6-0	Dma	This 7-bit field contains the data-memory address offset for the instruction

11.1.4.1 SP-Referenced Direct Addressing

In SP referenced direct addressing, the 7-bit dma in the instruction register is added as a positive offset to the SP to form the effective 16-bit data-memory address. The SP points to any address in memory. The dma points to the specific location on the page, allowing you to access a contiguous 128-word (2^7 – 1) block in memory from any base address. SP can also add or remove items from the stack.

11.1.4.2 DP-Referenced Direct Addressing

In DP-referenced direct addressing, the 7-bit dma in the instruction register is concatenated with the 9-bit DP to form the address. Figure 11.3 shows how the two values make up the resulting address.



15 - 7	6 - 0
Value from the DP	Value from the IR (dma)

Fig. 11.3 *DP-referenced direct address*

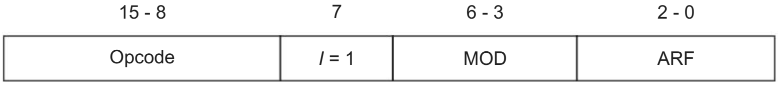
DP-referenced direct addressing divides memory into 512 pages, because the DP's range is from 0 to 511 ($2^9 - 1$). Each page has 128 addressable locations, because the dma ranges from 0 to 127 ($2^7 - 1$). In other words, the DP points to 1 of 512 possible 128-word data-memory pages; the dma points to the specific location within that page. The only difference between an access to location 0 on page 1 and to location 0 on page 2 is the value of the DP. The DP is loaded by the LD instruction.

11.1.5 Indirect Addressing

In indirect addressing, any location in the 64K-word data space can be accessed via a 16-bit address contained in an AR. The ′54X has eight 16-bit ARs (AR0–AR7). Indirect addressing is used mainly when there is a need to step through sequential locations in memory in fixed-size steps. When memory is addressed with indirect addressing, the AR and the address can be optionally modified by a decrement, an increment, an offset or an index. Special modes offer circular and bit-reversed addressing. A circular buffer size register (BK) is used with circular addressing. The AR0 register is used for indexed and bit-reversed addressing modes in addition to being used to point to memory as the other ARs do. Indirect addressing can also be used to access two data-memory locations with one instruction. All four possible operations can be performed With the two independent memory locations, viz, (Read, Read), (Write, Read), (Write, Write) and (Read, Write).

11.1.5.1 Single-Operand Addressing

Figure 11.4 shows the indirect-addressing instruction format for a single data-memory (Smem) operand.



15 - 8	7	6 - 3	2 - 0
Opcode	*I* = 1	MOD	ARF

Fig. 11.4 *Indirect-addressing instruction format for a single data-memory operand*

In Fig. 11.4, MOD defines the type of indirect addressing. ARF defines the AR used for addressing. ARF depends on the compatibility mode bit (CMPT) in status register ST1:

CMPT - 0 Standard mode. In standard mode, ARF always specifies the AR regardless of the value in ARP. ARP is not updated. ARP must always be set to zero when the DSP is in this mode.

CMPT = 1 Compatibility mode. In compatibility mode, ARP selects the AR if ARF = 0. Otherwise, ARF selects the AR and the ARF value is loaded into ARP when the access is completed. *AR0 in the assembly instruction indicates the AR selected by ARP in compatibility mode.

11.1.5.2 ARAU and Address-Generation Operation

Two ARAUs (ARAU0 and ARAU1) operate on the contents of the ARs. The ARAUs perform unsigned, 16-bit AR arithmetic operations. Some addresses can be obtained by premodifying the AR. The ARs can be:

loaded with an immediate value using the STM instruction,
loaded via the data bus by writing to the memory-mapped ARs,
modified by the indirect addressing field of any instruction that supports indirect addressing,
modified by the modify auxiliary register (MAR) instruction and
used as loop counters using the BANZ[D] instruction.

Normally, STM or MVDK is used to load ARs. Both of these instructions allow the next instruction to use the new value in the register. Other instructions that load a new value into an AR produce a pipeline latency.

11.1.5.3 Single-Operand Address Modifications

The addresses used for fetching the operands using indirect addressing mode may be modified before or after they are accessed, or left unchanged. The addresses can be modified by incrementing or decrementing the address by 1, adding a 16-bit offset or indexing with the value in AR0. These three types of actions combined with taking the action either before or after the access, plus the ways of leaving the address unchanged make a total of 16 addressing types, each assigned to a value of MOD, the 4-bit modification field in the encoding of an instruction using indirect addressing. Table 11.3 lists the operand syntax and the description of each addressing type.

Table 11.3 *Indirect addressing types with a single data-memory operand*

Operand syntax	*Function*	*Description*[†]
*ARx	addr = ARx	ARx contains the data memory address

*ARx-	addr = ARx
	ARx = ARx - 1
		After access, the address in Arx is decremented[‡]

*ARx+	addr = ARx
	ARx = ARx + 1
		After access, the address in ARx is incremented[‡§#]

*+ARx	addr = ARx + 1 ARx = ARx + 1	Before access, the address in ARx is incremented

*ARx-0B	addr = ARx
	ARx = B(ARx–AR0)
		After access, AR0 is subtracted from ARx with rc-propagation

*ARx-0	addr = ARx
	ARx = ARx-AR0
		After access, AR0 is subtracted from Arx

*ARx+0	addr = ARx
	ARx = ARx + AR0
		After access, AR0 is added to Arx

*ARx+0B	addr = ARx
	ARx = B(ARx + AR0)
		After access, AR0 is added to ARx with re propagation

*ARx-%	addr = ARx
	ARx = circ(ARx-1)
		After access, the address in ARx is decremented with circular addressing[‡]

*ARx-0%	addr = ARx
	ARx=circ(ARx-AR0)
		After access, AR0 is subtracted from ARx with circular addressing

*ARx+%	Addr = ARx
	ARx = circ(ARx + 1)
		After access, the address in ARx is incremented with circular addressing[‡]

(Contd.)

Table 11.3 *(Contd.)*

*ARx+0%	Addr = ARx
	ARx=circ(ARx + AR0)
		After access, AR0 is added to ARx with circular addressing

*ARx(lk)	Addr = ARx + Ik
	ARx = ARx
		The sum of ARx and the 1 6-bit long offset (Ik) is used as the data-memory address. ARx is not updated

*+ARx(lk)	Addr = ARx + Ik
	ARx = ARx + Ik
		The address in ARx is incremented before its use and added to the signed 16-bit long offset (Ik). It is then used as the data-memory address

*+ARx(lk)%	Addr = circ(ARx + lk)
	ARx = circ(ARx + lk)
		The address in ARx is incremented before its use and added to a signed 1 6-bit long offset (Ik) with circular addressing. It is then used as the data-memory address[§]

*(lk)	addr = Ik	An unsigned 16-bit long offset (Ik) is used as the absolute address of data memory (absolute addressing)

[†] ARx is used as the data-memory address unless otherwise specified.
[‡] Increment/decrement value is 1 for 16-bit word access and 2 for 32-bit word access.
[§] This mode is not allowed in memory-mapped register addressing.
[#] This mode is allowed only for write accesses.

11.1.5.4 Increment/Decrement Address Modifications

While an AR is being used, the AR can be modified by incrementing or decrementing its value. The syntaxes for using the AR without modification, postdecrementing the AR by 1, postincrementing the AR by 1, and preincrementing the AR by 1 are shown in Table 11.3 in the first four entries. Preincrementing (*+ARx) is supported only in instructions that access operands in a write operation.

11.1.5.5 Offset Address Modifications

Offset addressing is a type of indirect addressing in which a predetermined off-set, or step size, is added to the contents of an AR. There are two options for offset addressing. In both cases, a 16-bit long offset, which is part of the instruction, is added to the value in the AR and the result is used to address a location in data memory. In the first case, the AR is not updated. In the second case, the AR is updated with the new address. This type of addressing is useful in accessing a specific element of an array or structure, especially when the AR is not updated. When the AR is updated, this type of addressing is especially useful for stepping through an array in fixed-size steps. The syntaxes for offset addressing of an AR without and with updating the AR using offset addressing are shown in Table 11.3 in line 13 and 14, respectively. It may be noted that instructions using offset addressing cannot be repeated using the repeat single instruction. Premodification by a 16-bit word offset (*+ARx(lk)) uses an extra cycle because the instruction code has two or three words. The last word is the offset.

11.1.5.6 Indexed Address Modifications

Indexed addressing is a type of indirect addressing in which the contents of AR0 are added to, or subtracted from, any other AR, ARx. Indexed addressing differs from offset addressing in that the index or step size can be determined during code execution. Because the index is determined during code execution, step sizes can easily be adjusted. Indexed addressing also offers an advantage over offset addressing: it does not require an additional word for the instruction. The syntaxes for subtracting AR0 from ARx and for adding AR0 to ARx are shown in Table 11.3.

11.1.5.7 Circular Address Modifications

The use of circular addressing is explained in Chapter 2. Many algorithms, such as convolution, correlation and FIR filters, require the implementation of a circular buffer in memory. In these algorithms, a circular buffer is a sliding window containing the most recent data. As new data comes in, the buffer overwrites the oldest data. The key to the implementation of a circular buffer is the implementation of circular addressing. The circular-buffer size register (BK) specifies the size of the circular buffer. A circular buffer of size *R* must start on a *N*-bit boundary (that is, the *N* LSBs of the base address of the circular buffer must be 0), where *N* is the smallest integer that satisfies $2^N > R$. The value *R* must be loaded into BK. For example, a 51-word circular buffer must start at an address whose six LSBs are 0 (that is, $\text{XXXX XXXX XX00 0000}_2$), and the value 51 must be loaded into BK. In some applications, however, it may be possible to use bit-reversed addressing to place a 2*N* buffer on a 2*N* boundary and offer the effect of circular addressing.

The effective base address (EFB) of the circular buffer is determined by zeroing the *N* LSBs of a user-selected AR (ARx). The end of buffer address (EOB) of the circular buffer is determined by replacing the *N* LSBs of ARx with the *N* LSBs of BK. The index of the circular buffer is simply the *N* LSBs of ARx and the step is the quantity being added to or subtracted from the AR. The following three rules are used when circular addressing is to be used:

- Place the first (lowest) address of the circular buffer on a 2^N boundary where 2^N is larger than the circular buffer size.
- Use a step less than or equal to the circular buffer size.
- The first time the circular queue is addressed, the AR must point to an element in the circular queue.

Circular addressing typically uses a decrement or an increment by one or a decrement or an increment by an index. Premodification by a 16-bit word offset (*+ARx(1k)%) requires an extra code word so that the instruction code has two or three words. The last word is the offset. An instruction using indirect-offset addressing cannot be repeated using a single repeat operation. The syntaxes for each of the five types of circular addressing are shown in Table 11.3.

11.1.5.8 Bit-Reversed Address Modifications

Bit-reversed addressing enhances execution speed and program memory for FFT algorithms that use a variety of radixes. In this addressing mode, AR0 specifies one half of the size of the FFT. The value contained in AR0 must be equal to 2^{N-1} , where *N* is an integer, and the FFT size is 2^N. An AR points to the physical location of a data value. When AR0 is added to the AR using bit-reversed addressing, the address is generated in a bit-reversed fashion, with the carry bit propagating from left to right, instead of the normal right to left. The syntaxes for each of the two bit-reversed addressing modes are shown in Table 11.3.

11.1.5.9 Dual-Operand Address Modifications

Dual data-memory operand addressing is used for instructions that perform two reads or a single read and a parallel store (indicated by two vertical bars, ||) at the same time. These instructions are all one word long and operate in indirect addressing mode only. Two data-memory operands are represented by Xmem and Ymem: Xmem is a read operand with access through the D bus. Store instructions, for example STH and STL with shift operation, change Xmem to a write operand. Ymem is used as a read operand in instructions with dual reads (accessed through the C bus) or as a write operand in instructions with a parallel store (accessed through the E bus). Figure 11.5 shows the indirect-addressing instruction format for a dual data-memory operand. Table 11.4 describes the bits of the instruction.

15 - 8	7 - 6	5 - 4	3 - 2	1 - 0
Opcode	Xmod	Xar	Ymod	Yar

Fig. 11.5 *Indirect-addressing instruction format for dual data-memory operands*

Table 11.4 *Indirect-addressing instruction bit summary—dual data-memory operands*

Name	*Function*
Opcode	This field contains the operation code for the instruction
Xmod	Defines the type of indirect addressing mode used for accessing the Xmem operand
Xar	Xmem AR selection field defines the AR that contains the address of Xmem
Ymod	Defines the type of indirect addressing mode used for accessing the Ymem operand
Yar	Ymem AR (AR) selection field defines the AR that contains the address of Ymem

Because only two bits are available for selecting each AR in this mode, only four of the ARs can be used, AR2-AR5. (For Xar or Yar AR selected is: 00 for AR2, 01 for AR3, 10 for AR4 and 11 for AR5). Hence the dual data-memory operand addressing uses four ARs (AR2-AR5). The ARAUs, together with these registers, provide the capability to access two operands in a single cycle.

In instructions that perform dual-operand reads, if the auxiliary register specified by the Yar field accesses one of the memory-mapped registers, the value read will not represent the contents of the register. Table 11.5 shows the different ways in which the address can be modified in dual-operand indirect addressing mode.

Table 11.5 *Indirect addressing types with a dual data-memory operand*

Operand syntax	*Function*	*Description*[†]
*ARx	addr = ARx	ARx contains the data memory address

*ARx-	addr = ARx
	ARx = ARx – 1
		After access, the address in ARx is decremented

*ARx+	Addr = ARx
	ARx = ARx + 1
		After access, the address in ARx is incremented

*ARx+0%	Addr = ARx
	ARx=circ(ARx + AR0)
		After access, AR0 is added to ARx with circular addressing[‡]

[†] ARx is used as the data-memory address unless otherwise specified.

[‡] Size of the circular buffer is specified in circular buffer size register BK.

11.1.5.10 Single-Operand Instructions using the Dual-operand Format

Some instructions with only one data-memory operand use dual data-memory operand addressing so that they fit in a single word for single-cycle execution. In these instructions, only Xmem is available and the Xmod and Xar fields define the addressing mode for the operand. Four single-operand instructions can be executed in a single cycle:

BIT Xmem, BITC
SACCD src, Xmem, cond
SRCCD Xmem, cond
STRCD Xmem, cond

Five instructions with optional shift also support this type of addressing for single-word, single-cycle execution:

ADD Xmem, SHFT, src
LD Xmem, SHFT, dst
STH src, SHFT, Xmem
STL src, SHFT, Xmem
SUB Xmem, SHFT, src

11.1.5.11 TMS320C2X/C2XX/C5XCompatibility (ARP) Mode

ARP can be used in indirect addressing. This allows the AR to be defined by ARP to ease code translation from a ′C2X/C2XX/C5X device. With CMPT = 1 and ARF = 0, ARP is used to determine which AR is used to address memory. In using ARP, the ′54X differs from the ‘C5X in that when the ′54X uses the AR pointed to by ARP, the ′54X does not update the ARP with the same instruction. Table 11.6 shows the assembler syntax for the ′C2X/C2XX/C5X compared to the ′54X.

Table 11.6 *Assembler syntax comparison for TMS320C2X/C2XX/C5X and ′54X*

Syntax for ′C2X/C2XX/C5X	*Syntax for ′54X*	*Syntax for ′C2X/C2XX/C5X*	*Syntax for ′54X*
*	AR0	*+	*AR0+
*0-	*AR0–0	*BR0–	*AR0–0B
*	*AR0–	*BR0+	*AR0+0B
*0+	*AR0+0		

11.1.6 Memory-Mapped Register Addressing

Memory-mapped register addressing is used to modify the memory-mapped registers without affecting either the current data-page pointer (DP) value or the current stack-pointer (SP) value. Because DP and SP do not need to be modified in this mode, the overhead for writing to a register is minimal. Memory-mapped register addressing works for both direct and indirect addressing. In this mode, the addresses are generated by forcing the nine MSBs of data-memory address to 0, regardless of the current value of DP or SP when direct addressing is used. When indirect addressing is used, the seven LSBs of the current AR value are used for the lower order address.

In indirect addressing, the nine MSBs of the AR are forced to 0 after the operation. For example, if ARl is used to point to a memory-mapped register in memory-mapped register addressing mode and it contains a value of FF25h, then AR1 points to the timer period register (PRD), since the seven LSBs of AR1 are 25h and the address of the PRD is 0025h. After execution, the value remaining in AR1 is 0025h.

In addition to registers, any scratch-pad RAM located on data page 0 can be modified by using memory-mapped register addressing. Only eight instructions can use memory-mapped register addressing:

LDM MMR, dst
MVDM dmad, MMR
MVMD MMR, dmad
MVMM MMRx,MMRy
POPM MMR

PSHM MMR
STLM src, MMR
STM # lk, MMR

11.1.7 Stack Addressing

The system stack is used to automatically store the program counter during interrupts and subroutines. It can also be used at your discretion to store additional items of context or to pass data values. The stack is filled from the highest to the lowest memory address. The processor uses a 16-bit memory-mapped register, the SP, to address the stack. SP always points to the last element stored onto the stack. Four instructions access the stack using the stack addressing mode:

PSHD pushes a data-memory value onto the stack.
PSHM pushes a memory-mapped register onto the stack.
POPD pops a data-memory value from the stack.
POPM pops a memory-mapped register from the stack.

A push predecrements and a pop postincrements the address in the SP. Figure 11.6 shows an example of the stack and SP before and after a push of X2 into the stack (PSHD X2). The content of dmad X2 is stored in the location pointed by SP-1 in this case.

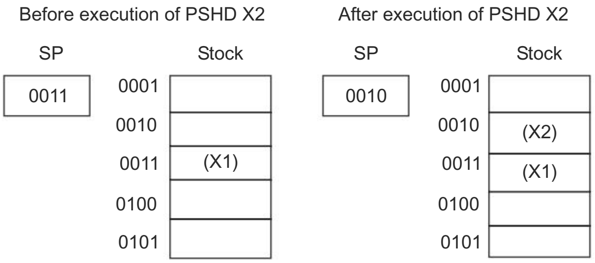


Fig. 11.6 *Stack and SP before and after executing a PSHD instruction*

Other operations also affect the stack and the SP. The stack is used during interrupts and subroutines to save and restore the PC contents. When a subroutine is called or an interrupt occurs, the return address is automatically saved in the stack using a push operation. Instructions used for subroutine calls and interrupts are CALA[D], CALL[D], CC[D], INTR and TRAP. When a subroutine returns, the return address is retrieved from the stack using a pop-operation and loaded into the PC. Instructions used for returns from subroutines are RET[D], RETE[D], RETEF[D] and RC[D]. The FRAME instruction also affects the stack. This instruction adds a short-immediate offset to the SP. The stack is also used in SP-referenced direct addressing.

11.1.8 Data Types

There are two basic data types for accessing memory in the ′54X: 16-bit and 32-bit. Most instructions can access 16-bit data. Accessing 32-bit data, however, requires the use of the special instructions listed in Table 11.7.

Table 11.7 *Instructions with 32-bit word operands*

Instruction	*Syntax*	*Description*
DADD	DADD Lmem, src [,dst]	Double-precision add/dual 16-bit add to accumulator
DADST	DADST Lmem, dst	Double-precision load with T add/dual 16-bit load with T add/subtract
DLD	DLD Lmem, dst	Long-word load to accumulator

(Contd.)

Table 11.7 *(Contd.)*

DRSUB	DRSUB Lmem, src	Double-precision subtract/dual 16-bit subtract from long word
DSADT	DSADT Lmem, dst	Long load with T subtract/dual 16-bit load with T subtract/add
DST	DST src, Lmem	Store accumulator in long word
DSUB	DSUB Lmem, src	Double-precision subtract/dual 16-bit subtract from accumulator
DSUBT	DSUBT Lmem, dst	Long load with T subtract/dual 16-bit load with T subtract

For a 16-bit operand access, a 16-bit word is read from data memory through the D bus and written to data memory through the E bus. For a 32-bit operand access, both the C (for most-significant word) and the D (for least-significant word) buses are used for a read. However, because only the E bus is used for a write, the write operation (DST instruction) is executed in two cycles. With 32-bit accesses, the first word accessed is treated as the most-significant word (MSW), while the second word accessed is the least-significant word (LSW). If the first word accessed is at an even address, then the second word is at the next (higher) address. If the first word accessed is at an odd address, then the second word is at the previous (lower) address.

ARITHMETIC INSTRUCTIONS 11.2

In the ′C54X instructions, the source of the operand is specified first and the destination is specified as the last parameters. The mnemonics for the arithmetic instructions of ′C54X are given in Table 11.8.

Table 11.8 *The mnemonics for the arithmetic instructions of ′C54X*

ABDST	ABS	ADD	ADDC	AD DM	ADDS
SUB	SUBB	SUBC	SUBS		
EXP	FIRS	LTD	MAC[R]	MACA[R]	MACD
MACP	MACSU	MAS[R|	MASA[R]	MAX	MIN
MPY[R]	MPYA	MFYU	NEG	NORM	POLY
SAT	SQDST	SQUR	SQURA	SQURS	LMS

11.2.1 Instructions for Finding the Absolute Value

The syntax and description of instructions ABDST and ABS are as follows:

ABDST: Absolute Distance Syntax ABDST Xmem, Ymem

Description: ABDST calculates the absolute value of the distance between two vectors, Xmem and Ymem. The absolute value of A(32-16) is added to the accumulator B. The content of Ymem is subtracted from Xmem, and the result is left-shifted 16 bits and stored in accumulator A. If the fractional mode bit FRCT =1, the absolute value is multiplied by two.

Example ABDST *AR2+, *AR3+—The absolute value of A(32-l6) is added to the accumulator B. The content of location pointed by AR3 is subtracted from the content of location pointed by AR2 and the result is left-shifted 16 bits and stored in accumulator A. The contents of AR2 and AR3 are incremented.

ABS - Absolute Value of Accumulator Syntax ABS src [,dst]

Description: The ABS instruction calculates the absolute value of the source accumulator and loads it into the destination accumulator. If no destination is given, ABS loads the absolute value into the source accumulator.

Example

1. ABS A, B - Absolute value of A is stored into B
2. ABS A - Absolute value of A is stored into A

11.2.2 Instructions for Addition and Subtraction

ADD and SUB Instructions These two instructions have a similarity w.r.t syntax and the manner in which the source and destinations for operands and the result are specified. Table 11.9 gives the list of the different ways in which the operands can be specified for both of these instructions. The ADD instruction adds a 16-bit value to the contents of the selected accumulator or to a 16-bit Xmem operand in dual data-memory operand addressing mode. The 16-bit value to be added is one of the following:

The content of a single data-memory operand
The content of a dual data-memory operand
A 16-bit long-immediate operand
The shifted value in the source accumulator

If a destination is specified, ADD stores the result in the destination accumulator. If no destination is specified, ADD stores the result in the source accumulator.

Most of the second operands can be shifted. For a left shift, low-order bits are cleared and high-order bits are sign-extended if SXM is 1, otherwise they are cleared. For a right shift, the high order bits are signextended if SXM is 1, otherwise they are cleared.

Example

1. ADD *AR2+, 12, A—The content of the location pointed by AR2 is read, shifted towards left by 12 bits and added to A register. The result is stored in A. AR2 is incremented by 1 after its content is used for fetching the operand.
2. ADD B, -5, A-B register is read, the value obtained is shifted towards right by 5 bits and added to A register. The result is stored in A.
3. ADD #2345h, 7, B, A—The constant 2345h is left shifted by seven bits and added to B.The result is stored in A.

Table 11.9 *Syntax for ADD and SUB instructions ′C54X*

ADD - Add to accumulator syntax	*SUB - Subtract from accumulator syntax*
1: ADD Smem, src	1: SUB Smem, src
2: ADD Smem, TS, src	2: SUB Smem, TS, src
3: ADD Smem, 16,src [,dst]	3: SUB Smem, 16, src [,dst]
4: ADD Smem, [SHIFT,] src [,dst]	4: SUB Smem, [SHIFT,] src [,dst]
5: ADD Xmem, SHIFT,src	5: SUB Xmem, SHIFT, src
6: ADD Xmem, Ymem, dst	6: SUB Xmem, Ymem, dst
7: ADD #lk, [SHIFT,] src[,dst]	7: SUB #lk, [SHIFT,]src[,dst
8: ADD #lk,16, src[,dst]	8: SUB #lk, 16, src [,dst]
9: ADD src [, SHIFT] [,dst]	9: SUB src [, SHIFT] [,dst]
10: ADD src, ASM [,dst]	10: SUB src, ASM [,dst]

The SUB instruction subtracts a 16-bit value from the contents of the selected accumulator or from a 16-bit Xmem operand in dual data-memory addressing mode. The 16-bit value to be subtracted is one of the following:

The content of a single data-memory operand
The content of a dual data-memory operand
A 16-bit long-immediate operand
The shifted value in the source accumulator

If a destination is specified, the SUB instruction stores the result in the destination accumulator. If no destination is specified, SUB stores the result in the source accumulator.

Most of the second operands can be shifted. For a left shift, low-order bits are cleared and high-order bits are sign-extended if SXM is 1, otherwise they are cleared. For a right shift, the high order bits are sign-extended if SXM is 1, otherwise they are cleared.

Example

1. SUB *AR1+, 14, A—The content of the location pointed by AR1 is'read, shifted towards left by 14 bits and subtracted from A register. The result is stored in A. AR1 is incremented by 1 after its content is used for fetching the operand.
2. SUB A, -8, B—A register is read, the value obtained is shifted towards right by eight bits and subtracted from B register. The result is stored in B.
3. SUB #2345h, 8, A, B—The constant 2345h is left shifted by eight bits and subtracted from A. The result is stored in B.

Mnemonics for some additional addition and subtraction instructions and their syntaxes are given in Table 11.10. More details of these instructions are considered next.

Table 11.10 *Mnemonics and syntax for some additional addition and subtraction instructions of 'C54X*

Mnemonic and description	*Syntax*
ADDC—Add to accumulator with carry	ADDC Smem, src
SUBB—Subtract from accumulator with borrow	SUBB Smem, src
ADDS—Add to accumulator with sign-extension suppressed	ADDS Smem, src
SUBS—Subtract from accumulator with sign-extension suppressed	SUBS Smem, src
SUBC—Subtract conditionally	SUBC Smem, src

ADDC and SUBB Instructions The ADDC Instruction adds the 16-bit single data-memory operand and the carry bit to the source accumulator. ADDC stores the result in the source accumulator. Sign-extension is suppressed regardless of the value of the SXM bit.

Example

ADDC *+AR2 (15h), A—15h is added to the content of AR2 first. Then the content of location pointed by AR2 is added to A register with carry.

The SUBB instruction subtracts the contents of the single data-memory operand and the logical inverse of the carry bit from the source accumulator without sign-extension.

Example

1. SUBB DAT5, A—The content of the fifth location in the current page and the logical inverse of the carry bit are subtracted from the accumulator A without sign-extension.
2. SUBB * AR1+, B—The content of the location pointed by AR1 and the logical inverse of the carry bit are subtracted from the accumulator A without sign-extension. The content of AR 1 is incremented after this operation.

ADDS and SUBS Instructions The ADDS instruction adds the 16-bit single data-memory operand to the source accumulator and stores the result in the source accumulator. Sign-extension is suppressed regardless of the value of the SXM bit.

Example ADDS *AR2-, B—Content of the location pointed by AR2 is added to accumulator B and the result is stored in B without sign extension. AR2 is decremented after this operation.

The SUBS instruction subtracts the content of the 16-bit single data-memory operand Smem from the content of the source accumulator. Smem is considered a 16-bit unsigned number regardless of the value of SXM. The result is stored in the source accumulator.

Example SUBS * AR2-, B—Content of the location pointed by AR2 is subtracted from accumulator B and the result is stored in B without sign extension. AR2 is decremented after this operation.

Adding a Long Immediate Constant The ADDM (Add long-immediate value to memory) instruction adds the 16-bit single data-memory operand to the 16-bit long-immediate memory value. ADDM stores the result in the data-memory location specified by Smem.

Syntax ADDM lk, Smem

Example ADDM 0123Bh, *AR4+—The 16-bit constant 0123B is added to the content of the location pointed by AR4 and the result is stored back in that location. After this instruction AR4 is incremented.

Instruction for Division The instruction SUBC (subtract conditionally) may be used for division. The SUBC instruction subtracts the 16-bit single data-memory operand Smem, left-shifted 15 bits, from the content of the source accumulator. If the result is greater than 0, it is shifted 1 bit left and 1 is added to the result. The SUBC instruction stores the result in the source accumulator. Otherwise, SUBC shifts the contents of the source accumulator 1 bit left and stores the result in the source accumulator. SUBC assumes that the divisor and the dividend are both positive. The SXM bit will affect this operation:

If SXM = 1, the divisor must have a 0 value in the MSB.
If SXM = 0, any 16-bit divisor value produces the expected results.

The dividend, which is in the accumulator, must initially be positive (that is, bit 31 must be 0) and must remain positive following the accumulator shift, which occurs in the first portion of the SUBC instruction. Note that SUBC affects OVA or OVB (depending on the source accumulator) but is not affected by OVM; therefore, the accumulator does not saturate on positive or negative overflows when executing this instruction.

Example SUBC DAT2, A—Let the content of DP be 06 and A be 00 0000 0004h. The content of 0302h (i.e. 0000 0011 0 000 0010_2) be 0001h. After shifting this by 15 bits towards left, we get 1000h. When this is subtracted from A, the result is negative. Hence A register is left shifted by 1 bit. The result is 0008h.

By repeated use of SUBC instruction a no. can be divided by the other.

11.2.3 Multiply Instructions

MPY[R]—Multiply The MPY instruction multiplies the contents of TREG or a data-memory value by a data-memory value or an immediate value. TREG is loaded with the Smem or Xmem value in the read phase of the first or second execution cycle.

MPYR rounds the result of the MPY operation by adding 2^{15} to the result and then clearing bits 15-0.

Syntax

1: MPY[R] Smem, dst
2: MPY Xmem, Ymem, dst
3: MPY Smem, #lk, dst
4: MPY #lk,dst

MPYA—Multiply by Accumulator A The MPYA instruction multiplies the high part (bits 32-16) of accumulator A by a single data-memory operand or by TREG. TREG is updated in the read phase.

Syntax

1: MPYA Smem
2: MPYA dst

MPYU—Multiply Unsigned The MPYU instruction multiplies the unsigned contents of TREG by the unsigned contents of the single data-memory operand and stores the result in destination accumulator. The multiplier acts as a signed 17 × 17-bit multiplier for this instruction with the MSB of both operands cleared to 0. The MPYU instruction is particularly useful for computing multiple-precision products, such as multiplying two 32-bit numbers to yield a 64-bit product.

Syntax MPYU Smem, dst

11.2.4 Multiply and Accumulate Instructions

MACP—Multiply by Program Memory and Accumulate This instruction is equivalent to the MAC instruction of 5X. The MACP instruction multiplies data-memory value by program-memory value, adds the product to the source accumulator and stores the result in the same accumulator. The data-memory value is copied into TREG. When this instruction is repeated, the program-memory address (in the program address register PAR) is incremented by 1. Once the repeat pipeline is started, the instruction becomes a single-cycle instruction.

Syntax MACP Smem, pmad, src

Example MACP * AR3-, COEFFS, A—Content of the location pointed by AR3 and the content of the program memory location COEFFS are multiplied and then added to A. AR3 is decremented after this operation. If this instruction is preceded by a repeat instruction PAR is incremented by 1.

MACD—Multiply by Program Memory and Accumulate with Delay This instruction is equivalent to the MACD instruction of 5X. The MACD instruction multiplies a data-memory value by a program-memory value, adds the product to the source accumulator and stores the result in that accumulator. The data-memory value is copied into TREG and into the next address following the Smem address. When this instruction is repeated, the program-memory address (in the program address register PAR) is incremented by 1. Once the repeat pipeline is started, the instruction becomes a single-cycle instruction.

Syntax MACD Smem, pmad, src

LTD—Load TREG and Insert Delay The LTD instruction copies the content of a data-memory location into TREG, and into the address following this data-memory location. When data is copied, the content of the address location remains the same. This function is useful for implementing a Z delay in DSP applications. This function is also contained in the DELAY and MACD instructions.

Syntax LTD Smem

MACSU—Multiply Signed by Unsigned and Accumulate The MACSU instruction multiplies an unsigned data-memory value (Xmem) by a signed data-memory value (Ymem) and adds the result to the source accumulator. The 16-bit unsigned value Xmem is stored in TREG. The TREG is updated with the unsigned value (Xmem) in the read phase.

Syntax MACSU Xmem, Ymem, src

MAC[R]—Multiply Accumulate With/Without Rounding The MAC[R] instruction multiplies and adds with or without rounding. The result is stored in the destination accumulator, if specified, or in the source accumulator. For syntaxes 2 and 3, the data-memory value after the instruction is stored in TREG. TREG is updated during the read phase. The MACR instruction rounds the result of the MAC operation by adding 2^{15} to the result and clearing the 16LSBs (bits 15-0) to 0.

Syntax

1: MAC[R] Smem, src
2: MAQ[R] Xmem, Ymem, src [,dst]
3: MAC #lk, src [,dst]
4: MAC Smem, #lk, src [,dst]

MACA[R]—Multiply by Accumulator A and Accumulate The MACA[R] instruction multiplies the high part of accumulator *A* (bits 32-16) by a single data-memory operand or by TREG. MACA[R] then adds 2^{15} the product to the source accumulator and stores the result in the destination accumulator, if specified, or in the source accumulator. A(32-16) is used as a 17-bit operand for the multiplier. The MACA[R] instruction rounds the result of the MACA operation by adding 2^{15} to the result and clearing the 16 LSBs of the destination accumulator (bits 15-0).

Syntax

1: MACA[R] Smem [,B]
2: MACA[R] T, src [,dst]

11.2.5 MAX and MIN Instructions

The MAX (accumulator maximum) instruction compares the contents of the accumulators and stores the maximum value in the destination accumulator. If the maximum value is in accumulator A, the carry bit is cleared to 0; otherwise, it is set to 1.

Syntax MAX dst

The MIN (accumulator minimum) instruction compares the contents of the accumulators and stores the minimum value in the destination accumulator. If the minimum value is in accumulator A, the carry bit is cleared to 0; otherwise, it is set to 1.

Syntax MIN dst

11.2.6 Instructions for Squaring

SQUR—Square The SQUR instruction squares the single data-memory operand or the high part of accumulator A (bits 32-16) and stores the result in the destination accumulator. TREG is unaffected when accumulator A is used.

Syntax

1: SQUR Smem. dst
2: SQURA, dst

SQURA—Square and Accumulate The SQURA instruction stores the content of data-memory location Smem, i.e. *(Smem) in TREG. Then, SQURA squares content Smem and adds the result to the source accumulator. The result is stored in the source accumulator.

Syntax SQURA Smem, src

SQURS—Square and Subtract The SQURS instruction stores the data-memory value in TREG. Then, it squares Smem and subtracts the product from the source accumulator. The result is stored in the source accumulator.

Syntax SQURS Smem, src

SQDST—Square Distance Used in repeat single mode, the SQDST instruction computes the square distance between two vectors. The high part of accumulator A (bits 32-16) is squared and the product is added to accumulator *8.* The result is stored in accumulator *B.* Ymem is subtracted from Xmem and the difference is shifted 16 bits left. The result is stored in accumulator A. The value to be squared A(32-16) is the value of the accumulator before the subtraction is executed by the SQDST instruction. *Syntax* SQDST Xmem, Ymem

11.2.7 Some Special Instructions

POL Y—Polynominal Evaluation The POLY instruction shifts the content of the single data-memory operand (Smem) 16 bits to the left and stores the result in accumulator *B.* In parallel, POLY multiplies bits 32-16 of accumulator A by the content of TREG, adds the product to accumulator *B,* rounds the result of this operation and stores the final result in accumulator *A.* This instruction is useful for polynominal evaluation to implement computations that take one cycle per monomial to execute.

Syntax POLY Smem

SA T—Saturate Accumulator Regardless of the OVM value, SAT allows the saturation of the content of the accumulator on 32 bits. If the MSB is 0 and if the value of the accumulator is greater than 00 7FFF FFFF, accumulator content is saturated to the value 00 7FFF FFFF. If the MSB is 1 and the value of the accumulator is greater than 80 7FFF FFFF, accumulator content is saturated to the value FF 8000 0000. Otherwise the accumulator is left unchanged. *Syntax* SAT src

EXP—Accumulator Exponent The EXP instruction computes the exponent value, which is a signed 2s-complement value in the -8 to +31 range, and stores the result in TREG. The exponent is computed by calculating the number of leading bits in the source accumulator and subtracting 8 from this value. The number of leading bits is equivalent to the number of left shifts needed to eliminate the significant bits from the 40-bit source accumulator with the exception of the sign bit. The source accumulator is not modified after this instruction.

The result of subtracting 8 from the number of leading bits produces a negative exponent for accumulator values that have significant bits in the guard bits (the eight MSBs of the accumulator used in error detection and correction). *Syntax* EXP src

Example EXP A

	Before execution	After execution
A	FF FFFF FFCB	FF FFFF FFCB
T	xxxx	19

The no. of leading ones excluding the sign bit is 33. Subtracting 8 (guard bits), the no. of leading bits become 25. The hex equivalent of this is 19.

	Before execution	After execution
B	07 86001234	07 86001234
T	XXXX	FFFC

In this case the significant bits are in guard bits. The no. of leading bits excluding the sign bit is 4. Subtracting 8 from this we get -4. 2's complement representation of –4 is FFFC.

NORM—Normalisation The NORM instruction allows single-cycle normalisation of the accumulator once the EXP instruction, which computes the exponent of a number, had executed. The shift value is defined by TREG (5-0) and coded as a 2s-complement number. For the normalisation, the shifter needs the shift value (in TREG) in the read phase; the normalisation is executed in the execution phase.
Syntax NORM src [,dst]

Example NORM A

	Before execution	After execution
A	FF FFFF F001	FF 8008 0000
T	0013	0013

The 19 (13h) leading ones excluding the sign bit and the guard bits are removed and *A* has a single sign bit after the normalisation.

FIRS-—Symmetrical Finite Impulse Response Filter FIRS is useful to implement symmetrical FIR filters. The FIRS instruction multiplies accumulator A(32-16) with a program-memory value addressed by pmad (program memory address) and adds the result to the value in accumulator *B.* At the same time, it adds the memory operands Xmem and Ymem, shifts the result left 16 bits and loads this value into accumulator A. In the next iteration, pmad is incremented by 1. Once the repeat pipeline is started, the instruction becomes a single-cycle instruction.
Syntax FIRS Xmem, Ymem, pmad

Example FIRS *AR3+, *AR4+, COEFFS—Accumulator A(32-16) is multiplied with the program memory with address COEFFS and adds the result to the value in accumulator *B*. The content of the locations pointed by AR3 and AR4 are added, shifted towards left by 16 bits and then loaded into ACC *A*. AR3 and AR4 are incremented. In the repeat mode the content of the next pma (COEFFS +1) is mutiplied with A(32-16).

MOVE INSTRUCTIONS OF ′54X 11.3

Mnemonics and syntax for the move instructions of ′C54X are given in Table 11.11. In this table, either the source or the destination address is specified as dmad for some of the instructions. This corresponds to 16-bit long immediate constant. When these instructions are used in the repeat mode, the dmad is loaded in address register corresponding to E bus (EAR) and it is incremented by one each time the instruction is executed. Similarly, either the source or the destination address is specified as pmad for some other instructions. This corresponds to 16-bit long immediate constant. When these instructions are used in the repeat mode, the pmad is loaded in address register corresponding to P bus (PAR) and it is incremented by one each time the instruction is executed.

READA and WRITA use the lower 16 bits of A for the program memory address (pma). They can also be used in repeat mode. In this case, in the first time, content of A will be copied to PAR and PAR is incremented each time it is executed in the repeat mode.

When the address is specified as Xmem, Ymem, the address is specified using indirect addressing mode. Smem can use both direct and indirect addressing mode.

As mentioned earlier, in ′54X, the address of the destination is indicated as the last parameter in the instruction.

Table 11.11 *Mnemonics and syntax for the move instructions of ′C54X*

Mnemonic	*Description*	*Syntax*
MVDM	Move data from data memory to memory-mapped register	MVDM dmad, MMR
MVMD	Move data from memory-mapped register to data memory	MVMD MMR, dmad
MVMM	Move data from memory-mapped register to memory-mapped register	MVMM MMR1, MMR2
MVDD	Move data from data memory to data memory with X, Y addressing	MVDD Xmem, Ymem
MVDK	Move data from data memory to data memory with destination addressing	MVDK Smem, dmad
MVKD	Move data from data memory to data memory with source addressing	MVKD dmad, Smem
MVDP	Move data from data memory to program memory	MVDP Smem, pmad
MVPD	Move data from program memory to data memory	MVPD pmad, Smem
READA	Read data memory addressed by accumulator A: pma given by the lower 16 bits A to dma specified by Smem	READA Smem
WRITA	Write memory data addressed by accumulator A: dma specified by Smem to pma given by lower 16 bits of A	WRITA Smem
DELAY	Memory delay: Copy the content of a single data-msmory location into the next higher address	DELAY Smem

LOAD/STORE INSTRUCTIONS OF '54X 11.4

In the syntaxes for the load instructions, Smem denotes a single memory operand using either direct or indirect addressing. Xmem denotes a dual-memory operand using indirect addressing (in this case only AR2-AR5 can be used as pointers). The symbols src and dst denote either A or B. 1k denotes a 16-bit long immediate constant.

LD Load Accumulator with Shift The LD instruction loads the accumulator with a data-memory value or an immediate value. This instruction supports different shift quantities. This instruction also supports accumulator-to-accumulator moves with shift. The syntax for this instruction is as follows:

Syntax

1: LD Smem, dst
2: LD Smem, TS, dst
3: LD Smem, 16, dst
4: LD Smem [,SHIFT], dst
5: LD Xmem, SHIFT, dst
6: LD #K,dst
7: LD #lk [,SHIFT], dst
8: LD #lk, 16, dst
9: LD src, ASM [,dst]
10: LD src [,SHIFT] [,dst]

Load TREG/DP/ASM/ARP The LD instruction loads a value into TREG or into the following fields of the status register: DP, ASM and ARP. The value loaded can be a single data-memory operand or a constant.

Table 11.12 *Mnemonics and syntax for some load instructions of 'C54X*

LDM	Load ACC from a memory-mapped register	LDM MMR, dst
LDU	Load ACC with unsigned memory value	LDU Smem, dst
ST	Store TREG, TRN, or immediate value Into memory	ST T, Smem
		ST TRN, Smem
		ST #lk, Smem
STH	Store accumulator high Into memory	STH src, Smem
		STH src, ASM, Smem
		STH src, SHIFT, Xmem
		STH src[,SHIFT], Smem
STL	Store accumulator low into memory	STL src, Smem
		STL src, ASM, Smem
		STL src, SHIFT, Xmem
		STL src [,SHIFT],Smem
STLM	Store accumulator low into memory mapped register	STLM src, MMR
STM	Store immediate value into memory mapped register	STM #lk, MMR

1: LD Smem, T
2: LD Smem, DP
3: LD #k9,DP
4: LD #k5,ASM
5: LD #k3,ARP
6: LD Smem, ASM

LDR—Load Memory Value in Accumulator High with Rounding The LDR instruction loads the data-memory value into the high part of the destination accumulator (bits 31-16). The data-memory value is rounded by adding 1/2 LSB, that is, 2^{15} to this value (8000h) and clearing the 15 LSBs of the accumulator to 0. Bit 15 of the accumulator is set to 1.

Syntax LDR Smem, dst

The Mnemonics and syntax for some of the other load instructions of ′C54X are given in Table 11.12.

LOGICAL INSTRUCTIONS 11.5

The AND, OR and XOR instructions have identical syntaxes as shown in Table 11.13. Description and syntax for some of the other logical instructions are given in Table 11.14.

Table 11.13 *Syntax for the AND, OR and XOR instructions*

Mnemonic	*Description*	*Syntax*
AND	And with accumulator	1: AND Smem, src
		2: AND #lk[,SHIFT],src[,dst]
		3: AND #lk, 16, src [,dst]
		4: AND src [,SHIFT] [,dst]
OR	OR with accumulator	1: OR Smem, src
		2: OR #lk [SHIFT] src [,dst]
		3: OR #lk, 16, src[,dst]
		4: OR src[,SHIFT] [, dst]
XOR	Exclusive-OR with accumulator	1: XOR Smem, src
		2: XOR #lk [SHIFT] src[,dst]
		3: XOR #lk, 16, src [,dst]
		4: XOR src[,SHIIFT] [,dst]

Table 11.14 *Description and syntax for some of the logical instructions*

Mnemonic	*Description*	*Syntax*
ANDM	And memory with long immediate	ANDM #lk, Smem
ORM	OR memory with 16-bit constant	ORM #lk, Smem
XORM	Exclusive-OR memory with constant	XORM #lk, Smem
CMPL	Find 1 ‘s complement of accumulator	CMPL src [,dst]
NEG	Find 2’s complement of accumulator	NEG src [,dst]
CMPM	Compare memory with long immediate	CMPM Smem, #lk
CMPR	Compare auxiliary register with AR0	CMPR CC, ARx

CMPS—Compare Select Max and Store The CMPS instruciion compares the two 16-bit signed values located in the high and low parts of the source accumulator (considered 2s-complement values). CMPS stores the maximum value in the single data-memory location, stores the result in the transition (TRN) register after shifting left one bit and sets or clears the test control (TC) bit. This instruction does not follow the standard pipeline operation. The comparison is performed in the read phase; thus, the src value, is the value one cycle before the CMPS instruction executes. The TRN register and the TC bit are updated during the execution phase.

Syntax CMPS src, Smem

Rotate and Shift Instructions Syntax for the rotate and shift instructions of ′54X are given in Table 11.15. The manner in which the LSB, MSB and carry bits are affected by the rotate instructions are given in Table 11.16. In this table the bits with suffix of old denotes the bit before the rotate instruction is executed. For example carry$_{old}$ denotes the carry bit before the rotate instruction is executed.

Table 11.15 *Syntax for the rotate and shift instructions of ¢54X*

Mnemonic	*Description*	*Syntax*
ROL	Rotate accumulator left	ROL src
ROLTC	Rotate accumulator left using TC	ROLTC src
ROR	Rotate accumulator right	ROR src
SFTA	Shift accumulator arithmetically	SFTA src, SHIFT [,dst]
SFTC	Shift accumulator conditionally	SFTC src
SFTL	Shift accumulator logically	SFTL src, SHIFT [,dst]

Table 11.16 *The effect of the rotate instructions on LSB, MSB and carry*

Instruction	*Carry bit*	*MSB*	*LSB*	*Guard bits*	*TC*
ROL	MSB$_{old}$	(MSB-1)$_{old}$	Carry$_{old}$	0	X
ROLTC	MSB$_{old}$	(MSB-1)$_{old}$	TC$_{old}$	0	X
ROR	LSB$_{old}$	Carry$_{old}$	(LSB+1)$_{old}$	0	X

ROL rotates each bit of the accumulator to the left by one bit, shifts the value of the carry bit into the LSB of the accumulator, shifts the value of the MSB of the accumulator into the carry bit and clears the accumulator's guard bits. ROR rotates each bit of the accumulator to the right by one bit, shifts the value of the carry bit into the MSB of the accumulator, shifts the value of the LSB of the accumulator into the carry bit and clears the accumulator's guard bits. The ROLTC instruction (rotate accumulator left with TC) rotates the accumulator to the left and shifts the TC bit into the LSB of the accumulator. In SFTA and SFTL, the shift count is defined as -16 SHIFT 15. SFTA is affected by the SXM bit. When SXM = 1 and SHIFT is a negative value, SFTA performs an arithmetic right shift and maintains the sign of the accumulator. When SXM = 0, the MSBs of the accumulator are zero filled. SFTL is not affected by the SXM bit; it performs the shift operation for bits 31-0, shifting 0s into the MSBs or LSBs, depending on the direction of the shift. SFTC performs a 1-bit left shift when both bits 31 and 30 are 1 or both are 0. This normalises 32 bits of the accumulator by eliminating the most significant nonsign bit.

CONTROL INSTRUCTIONS 11.6

Branch Call and Return Instructions of ′54X The syntax for the branch, call and return instructions of ′54X are given in Table 11.17. In this table, the instructions which are indicated with a suffix of [D] imply that there are instructions with the suffix and without the suffix. For example, B[D] implies that there are two instructions B and BD. Similarly, CALA[D] refer to two instructions CALA and CALAD. The instructions with the suffix D, for example, BD, BACCD, CALAD, RETD, etc. denote the delayed branch, call and return instructions. If the branch is a delayed branch (specified by the D suffix), the two 1-word instructions or the one 2-word instruction following the branch instruction are fetched from program memory and executed before the branch is taken. For a delayed call (specified by the D suffix), the next two instruction words are fetched and executed before the call. If the return is delayed (specified by the D suffix), the two 1-word instructions or one 2-word instruction following the RETD instruction is fetched and executed before executing the return. For the conditional instructions, for example, BC, the word [,cond] denote the additional conditions that may be tested before the action (either branch, call or return) is taken. The codes used for describing various conditions in ′54X are given in Section A 10.2.5. However, the conditions which may be simultaneously tested are restricted in ′54X. For example, let us consider the instruction CC[D]: Call Conditionally.

Table 11.17 *′54X Instructions for branch, call and return*

B(D]	*Branch unconditionally*	*B{D] pmad*
BACC[D]	Branch to location specified by accumulator	BACC[D] src
BANZ[D]	Branch on auxiliary register not zero	BANZ[D] pmad, Sind
BC[D]	Branch conditionally	BC[D] pmad, cond [,cond] [,cond]
CALA[D]	Call subroutine at location specified by accumulator	CALA[D] src
CALL[D]	Call unconditionally	CALL[D] pmad
CC[D]	Call conditionally	CC[D] pmad, cond [,cond] [,cond]
RC[D]	Return conditionally	RC[D] cond [,cond] [,cond]
RET[D]	Return	RET[D]
RETE[D]	Enable interrupts and return from interrupt	RETE[D]
RETF[D]	Enable interrupts and fast return from interrupt	RETF[D]

The CC[D] instruction tests multiple conditions before passing control to another section of the program. CC[D] can test the conditions individually or in combination with other conditions. But the users can only combine conditions from one group from Table 11.18 as follows:

Group 1: Up to two conditions may be selected. Each of these conditions must be from a different category (category A or B); two conditions from the same category cannot be selected. For example, we can test EQ and OV at the same time but we cannot test GT and NEQ at the same time.

Group 2: Up to three conditions may be selected. Each of these conditions must be from a different category (category A, B or C); we cannot have two conditions from the same category. For example, we can test TC, C and BIO but we cannot test NTC, C and NC at the same time. The same rule is also applicable for the conditional branch and return instructions.

Table 11.18 *Groups and categories for conditional instructions*

Group 1		*Group 2*		
Category A	*Category B*	*Category A*	*Category B*	*Category C*
EQ	OV	TC	C	BIO
NEQ	NOV	NTC	NC	NBIO
LT				
LEQ				
GT				
GEQ				

Stack and Stack Pointer Control Instructions The mnemonics for the push and pop instructions are given in Table 11.19. These instructions are used with the subroutines.

Table 11.19 *Push & pop instructions*

PSHD	Push data-memory value onto stack	PSHD Smem
PSHM	Push memory-mapped register onto stack	PSHM MMR
POPD	Pop top of stack to data memory	POPD Smem
POPM	Pop top of stack to memory mapped register	POPM MMR
FRAME	Add an 8-bit short immediate signed no. to stack pointer	FRAME K

CONDITIONAL STORE INSTRUCTIONS 11.7

There are three conditional store instructions. In each of them one of the conditions "cond", where the condition codes are given in Section A 10.2.5, is tested for storing.

SRCCD—Store Block Repeat Counter Conditionally If the condition is true, SRCCD stores the content of the block repeat counter (BRC) in Xmem. If the condition is false, the instruction reads Xmem and writes the value in Xmem back to the same address; thus, Xmem remains the same. Regardless of the condition, Xmem is always read and updated. *Syntax* SRCCD Xmem, cond

SACCD—Store Accumulator Conditionally If the condition is true, the SACCD instruction stores the source accumulator left-shifted by ASM -16. The shifted no. is stored in the memory location designated by Xmem. If the condition is false, the instruction reads Xmem and writes the value in Xmem back to the same address; thus, Xmem remains the same. Regardless of the condition, Xmem is always read and updated. *Syntax* SACCD src, Xmem, cond

STRCD—Store TREG Conditionally If the condition is true, the STRCD instruction stores the content of TREG into the Xmem location. If the condition is false, the instruction reads Xmem and writes the value in Xmem back to the same address; thus, Xmem remains the same. Regardless of the condition, Xmem is always read and updated.
Syntax STRCD Xmem, cond

REPEAT INSTRUCTIONS OF ′54X 11.8

Only some of the instructions of ′54X are repeatable. The instructions which cannot be repeated are highlighted with † symbol *(Refer to the ¢54X instruction set summary)* in Table A 10.1.

RPT—Repeat Next Instruction The repeat counter (RC) is loaded with the number of iterations when RPT is executed. The number of iterations (*n*) is obtained from a 16-bit single data-memory operand or an 8- or 16-bit constant. The instruction following the RPT instruction is repeated *n* + 1 times. We cannot access RC while it decrements.

Syntax

1: RPT Smem
2: RPT #k
3: RPT #lk

RPTB[D]—Block Repeat The RPTB[D] instruction allows a block of instructions to be repeated the number of times specified by the memory-mapped BRC. BRC must be loaded before the execution of an RPTB instruction. When the RPTB is executed, the Block-Repeat Start Address Register (RSA) is loaded with PC + 2 (PC + 4 if delayed) and the Block-Repeat.End Address Register (REA) is loaded with the program-memory address (pmad).

The RPTB instruction is interruptible. Single-instruction repeat loops (RPT and RPTZ) can be included as part of RPTB blocks. To nest instructions we need to ensure that the BRC, RSA and REA registers are appropriately saved and restored and the block repeat active flag (BRAF) is properly set.

In the RPTBD instruction, which specifies a delayed block repeat, the two 1-word instructions or the one 2-word instruction following the RPTB is fetched and executed before the execution of the RPTBD instruction.

Block repeat can be deactivated by clearing the BRAF bit.

Syntax RPTB[D] pmad

RPTZ—Repeat Next Instruction and Clear Accumulator The RPTZ instruction clears the destination accumulator and repeats the instruction following RPTZ *n* + 1 times, where *n* is the value in the repeat counter (RC). The RC value is obtained from the long-immediate constant.

Syntax RPTZ dst, #1k

INSTRUCTIONS FOR BIT MANIPULATIONS 11.9

BIT—Test Bit The BIT instruction copies the specified bit of the dual data-memory operand into the TC bit of status register STO.

Syntax Bit Xmem, BITC

If *n*th bit is to be copied, BITC should be specified as 15-n.

Example BIT * AR5+, 15-12—This copies bit 12 of the location pointed by AR5 into TC.

BITF—Test Bit Field Specified by Immediate Value The BITF instruction tests the specific bit or bits of the data memory value. If the specified bit is 0, the TC bit in status register ST0 is cleared to 0; otherwise, TC is set to 1. The Ik constant is a mask for the bit or bits tested.

Syntax BITF Smem, Ik

Example

1. BITF-DAT3, 00FFh This tests 8 LSBs of the third location in the current data page
2. BITF DAT5, 0800h This tests lower order eleventh bit of the fifth location in the current data, page

BITT—Test Bit Specified by TREG The BITT instruction copies the specified bit of the data-memory value into the TC bit of status register ST0. The four LSBs of TREG contain a bit code that specifies which bit is copied.

The bit address corresponds to (15-TREG(3-0)). The bit code corresponds to the contents of TREG(3-0).

Syntax BITT Smem

RSBX—Reset Status Register Bit The RSBX instruction clears the specified bit in status register 0 or 1 to a logic 0. N designates the status register to modify and SBIT (0<= SBIT <= 15) specifies the bit to be modified. The name of a field in a status register (Sname) can be used as an operand instead of the N and SBIT operands.

Syntax

1. RSBX N, SBIT
2. RSBX Sname

Example

1. RSBX SXM—SXM means: *n*=1 and SBIT =8
2. RSBX 1, 8

SSBX—Set Status Register Bit The SSBX instruction sets the specified bit in status register 0 or 1 to a logic 1. N designates the status register to modify and SBIT (0<= SBIT <= 15) specifies the bit to be modified. The name of a field in a status register (Sname) can be used as an operand instead of the N and SBIT operands.

Syntax

1. SSBX N, SBIT
2. SSBX Sname

SOME SPECIAL CONTROL INSTRUCTIONS 11.10

MAR—Modify Auxiliary Register The MAR instruction modifies the contents of the selected AR as specified by Smem. In compatibility mode (CMPT = 1), it modifies the AR content as well as the ARP value. In compatibility mode, if ARX = AR0 or ARX is null, then X is 0. Null is represented by an operand with only an asterisk (*). If CMPT = 0, the AR is modified but ARP is not.

Syntax MAR Smem

NOP—No Operation No operation is performed. Only the program counter is incremented. NOP is useful to create pipeline and execution delays.

Syntax NOP

RESET—Software Reset The RESET instruction performs a nonmaskable software reset that can be used at any time to put the ′C54X into a known state. When the RESET instruction is executed, the operations listed in the execution section occur. The MP/$\overline{\text{MC}}$ pin is not sampled during this software reset. The initialisation of IPTR and the peripheral registers is different from the initialisation using -RS. This instruction is not affected by INTM; however, it sets INTM to 1 to disable interrupts. PC is loaded with the content of IPTR after left shifting it by seven bits. IFR (interrupt Flag register) is cleared to 0. The content of the status registers ST0 and ST1 are initialised to the values given in Table 11.20 and 11.21.

Syntax RESET

Table 11.20 *The content of ST0 after software RESET*

ARP	TC	C	OVA	OVB	DP
000	1	1	0	0	0000 0000 0

Table 11.21 *The content of ST1 after software RESET*

BRAF	CPL	XF	HM	INTM	OVM	SXM	C16	FRCT	CMPT	ASM
0	0	1	0	1	0	1	0	0	0	00000

INTR—Software Interrupt The INTR instruction transfers program control to the interrupt vector specified by *K*. (0<=K< = 31). This instruction allows the user software to execute any interrupt service routine.

During execution of the instruction, the content of PC is incremented by one and pushed onto the stack. Then, the interrupt vector specified by *K* is loaded in the PC and the interrupt service routine for this interrupt is executed. The corresponding bit in the IFR is cleared and interrupts are globally disabled (INTM = 1). Note that the interrupt mask register (IMR) has no effect on the INTR instruction. INTR is executed regardless of the value of INTM.

Syntax INTR K

TRAP—Software Interrupt The TRAP instruction is a software interrupt that transfers program control to an interrupt service routine specified by K. The TRAP instruction pushes the program counter plus 1 onto the data-memory location addressed by SP. This enables a return instruction to retrieve the pointer to the instruction after the TRAP from the data-memory location addressed by SP. The TRAP instruction is not maskable. It is not affected by INTM nor does it affect INTM.

Syntax TRAP K (0<=K<=31)

IDLE—Idle Until Interrupt The IDLE instruction forces the program being executed to wait until an unmasked interrupt or reset occurs. The PC is incremented once. The device remains in an idle state (power-down mode) until it is interrupted.

The idle state is exited after an unmasked interrupt, even if INTM = 1. If INTM = 1, the program continues executing at the instruction following the idle. If INTM = 0, program branches to the corresponding interrupt service routine. The interrupt is enabled by the IMR, regardless of the INTM value. The following options, indicated by the value of *K,* determine the type of interrupts that can release the device from idle:

K = 1, Peripherals, such as the timer and the serial ports, are still active. The peripheral interrupts as well as reset and external interrupts release the processor from idle mode.

K =2, Peripherals, such as the timer and the serial ports, are inactive. Reset and external interrupts release the processor from idle mode. Because interrupts are not latched in idle mode as in normal device operation, they must be low for a number of cycles to be acknowledged.

K = 3, Peripherals, such as the timer and the serial ports, are inactive and the PLL is halted. Reset and external interrupts release the processor from idle mode. Because interrupts are not latched in idle mode as in normal device operation, they must be low for a number of cycles to be acknowledged.

Syntax IDLE K

Example

1. IDLE 1—The. processor idles until a reset or unmasked interrupt occurs.
2. IDLE 2—The processor idles until a reset or unmasked external interrupt occurs.
3. IDLE 3—The processor idles until a reset or unmasked external interrupt occurs.

XC—Execute Conditionally The execution of XC depends on the value of *N* and the selected conditions:

If *N* = 1 and the condition(s) is met, the 1-word instruction following the XC instruction is executed.

If *N* = 2 and the condition(s) is met, the 2-word instruction or the two 1-word instructions following the XC instruction are executed.

If the condition(s) is not met, one or two NOPs are executed depending on the value of *N.*

The XC[D] instruction tests multiple conditions before executing. XC[D] can test the conditions individually or in combination with other conditions. But only the conditions from either group 1 or group 2 in Table 11.18 may be combined. In a group only one condition may be chosen from each category in Table 11.18.

The XC instruction and the two instruction words following XC are uninterruptible.

The conditions tested are sampled two full cycles before the XC instruction is executed. Therefore, if the two instructions before XC are single-cycle instructions, their execution will not affect the condition of XC. If the two instructions before XC affect the condition being tested, the interrupt operation using XC can cause undesirable results.

Syntax XC N, cond [,cond][,cond|

I/O INSTRUCTIONS OF '54X 11.11

PORTR—Read Data From Port The PORTR instruction reads a 16-bit value from an external I/O port into the specified data-memory location. The -$\overline{\text{IS}}$ signal goes low to indicate an I/O access, and the IOSTRB and READY timings are explicit I/O read. PA designates the 16-bit immediate address of an I/O port.

Syntax PORTR PA, Smem

PORTW—Write Data to Port The PORTW instruction writes a 16-bit value from the specified data-memory location to external I/O port (PA). The -$\overline{\text{IS}}$ signal goes low to indicate an I/O access, and the IOSTRB and READY timings are explicit I/O write. PA designates the 16-bit immediate address of an I/O port.

Syntax PORTW Smem, PA

Table 11.22 *Syntax and description of parallel instructions of ′54X*

LD||MAC[R]	Multiply accumulate with/without rounding and parallel load	LD Xmem, dst || MAC[R] Ymem [,dst2]
LD||MAS[R]	Multiply subtract with/without rounding and parallel load	LDXmem, [,dst] || MAS[R] Ymem [,dst2]
ST||ADD	Store accumulator with parallel add	ST src, Ymem || ADD Xmem, dst
ST||LD	Store accumulator with parallel load	1: ST src, Ymem || LD Xmem, dst 2: ST src, Ymem || LD Xmem, T
ST||MAC[R]	Store accumulator with parallel multiply accumulate with/without rounding	ST src, Ymem || MAC[R] Xmem, dst
ST||MAS[R]	Store accumulator with parallel multiply subtract with/without rounding	ST src, Ymem || MAS[R] Xmem, dst
ST||MPY	Parallel store and multiply	ST src, Ymem || MPY Xmem, dst
ST||SUB	Parallel store and subtract	ST src, Ymem || SUB Xmem, dst

PARALLEL INSTRUCTIONS 11.12

Parallel instructions of ′54X are single word instructions and two instructions can be specified in a single word. The syntax and descriptions of the parallel instructions of ′54X are given in Table 11.22. The LD||MAC[R] instruction is explained in detail. The same notation is followed for the other parallel instructions.

LD||MAC[R]—Multiply Accumulate With/Without Rounding and Parallel Load The LD||MAC[R] is a single word instruction which multiplies a dual data-memory operand by the contents of TREG and adds the result of the multiplication to dst2. In parallel, it loads the higher part of the destination accumulator (bits 31-16) with a dual data-memory operand. The LD||MACR instruction rounds the result of the MAC operation by adding 2^{15} to the result and clearing the LSBs (15-0).

Syntax LD Xmem, dst
||MAC[R] Ymem [,dst2]
dst: either A or B; dst2: If dst = A, then dst2 = B; if dst = B, then dst2 = A

LMS INSTRUCTION 11.13

The LMS instruction is used to execute the least mean square (LMS) algorithm. The dual data-memory operand Xmem is added to accumulator A and shifted left 16 bits. The result is rounded by adding 2^{15} to the high part of the accumulator (bits 31-16). The final result is stored in accumulator A. In parallel, Xmem and Ymem are multiplied and the result is added to accumulator *B*. Xmem and Ymem are multiplied and the result is added to accumulator *B*. Xmem does not overwrite TREG; therefore, TREG always contains the error value used to update coefficients. *Syntax* LMS Xmem, Ymem

Example

```
LMS*AR3+,*AR4+
```

Review Questions

11.1 What does the following keywords signify in the instruction set of ′54X?

(a) pmad (b) dmad (c) Ik (d) k
(e) smem (f) xmem (g) ymem (h) shift
(I) shift2 (j) src (k) dst

11.2 Can you load *T* register using immediate addressing mode? If so, give an example.

11.3 Give examples for the absolute addressing using
(a) pmad addressing
(b) dmad addressing

11.4 The content of the location 2050h in data memory space is to be copied to ACCU B. What is the instruction to be used if absolute addressing is used.

A	B	DP	SP	1020	1125
45h	33h	20h	1105h	33	23

Fig. 11.7

11.5 The content of some of the registers and data memory location are initialised as shown in Fig 11.7. What is the content of A, *B* after executing instruction ADD 20h, *A, B* assuming CPL bit to be
(a) 0 (b) 1

11.6 Write an ALP which initialises the registers to the values given in Fig. 11.7 using immediate addressing mode.

11.7 Write an ALP which initialises the memory location to the values given in Fig. 11.7 using
(a) Absolute addressing
(b) Direct addressing using DP
(c) Direct addressing using SP
(d) Indirect addressing with AR2, AR3 as memory address pointers

11.8 Assume that AR2 is initialised to 1025h, what is the value of AR2 before and after execution of each of the following instructions
LD *AR2, 20h?
LD *AR2+, 20h
LD *AR2-, 20h
LD *+AR2(2), 20h
LD *AR2(2), 20h

11.9 Write a program for division using repeated use of SUBC instruction. The divisor is in location 1020h.
Hint: Find the number of leading zeros in the divisor, let it be *N*. The number of times SUBC should be repeated *is N* + 1.

11.10 Write a program to find the number of leading zeros in the location 1020h using ROL instruction.

11.11 Write a program to find the number of leading zeros in the location 1020h using EXP instruction.

11.12 How does the multiply instruction of ′54X differ from that of 5X in the manner in which the result is stored?

11.13 When the instruction LTD 20h is executed the contents of which register and memory location are altered?

11.14 In the FIRS instruction where should the coefficients be stored? Where are the input data samples to be stored?

11.15 How does the rounding operation performed in LDR instruction differ from the other instructions which perform rounding such as MPYR, MACR?

11.16 How does the conditional instruction in ′54X differ from those of 5X?

11.17 How does the delayed call instruction differ from the undelayed calls?

11.18 How do the three idle instructions affect the operation of internal hardware of ′54X?

11.19 Explain the operation of parallel instructions of ′54X?

11.20 Explain how an adaptive filter can be implemented using the LMS instruction.

Self Test Questions

11.1 Which of the following instructions has an operand which is of length 3 bits?
(a) LD#05, ARP (b) LD #04, AG(?)
(c) LD #143h, DP (d) AND #1234h, A, A
(e) LD#1234h, B

11.2 Which of the following instructions has an operand which is of length 6 bits?
(a) LD #05,ARP (b) LD #04, AG(?)
(c) LD #143h, DP (d) AND #1234h, A, A
(e) LD #1234h, B

11.3 Which of the following instructions has an operand which is of length 9 bits?

(a) LD#05, ARP (b) LD #09, AG(?)
(c) LD#143h, DP (d) AND #1234h, A, A
(e) LD#1234h, B

11.4 Which of the following instruction does not alter the value of AR2?

(a) LD*AR2, 20h (b) LD *AR2+, 20h
(c) LD *+AR2(2), 20h (d) LD *AR2(2), 20h

11.5 When the ——— bit is ———, the SP is used to compute address in direct addressing mode.

(a) CPL, 0 (b) CPL, 1 (c) CMPT, 0 (d) CMPT, 1

11.6 When the ——— bit is ——— and the ARP = 0, the ARP specifies the AR used for indirect addressing mode

(a) CPL, 0 (b) CPL, 1 (c) CMPT, 0 (d) CMPT, 1

11.7 The AR which is used as index register in ′54X is ———.

(a) AR0 (b) AR2 (c) AR4 (d) AR7

11.8 The mnemonic for the instruction which loads ACCB using indirect addressing mode (AR3) and fetches the operand after incrementing the address by constant 8h is LD ———, B.

(a) *+AR3(8) (b) *AR3(8)
(c) *+AR3(8)% (d) AR3*(8)

11.9 The mnemonic for the instruction which loads ACCB with the content of the data memory location 1000h is ———.

(a) LD *(1000h), B (b) LD #1000h, B
(c) LD 1000h, B (d) None of the above

11.10 The dual data-memory operand addressing mode instructions are ——— word long and can perform ——— read operation/cycle.

(a) 1,2 (b) 2, 1 (c) 2,2 (d) 1, 1

11.11 Which of the ARs cannot be used for specifying the address in the dual operand indirect addressing mode?

(a) AR0-AR1 (b) AR2-AR3 (c) AR4-AR5 (d) AR6-AR7

11.12 The content of DP is 20h. The instruction for ADDing the content of the location 1040h using direct addressing mode after shifting it towards left by five bits to *A* register is ———.

11.13 The instruction for shifting ACC B by eight bits towards left and adding it to *A* register is ———. The result is to be stored in *A*.

11.14 The instruction for shifting ACC B by eight bits towards left and subtracting it to *A* register is ———. The result is to be stored in *A*.

11.15 The instruction for adding the content of the location pointed by AR2 to that pointed by AR3 and storing the result in *A* register is ———.

11.16 The instruction for subtracting the content of location pointed by AR2 from that pointed by AR3 and storing the result in *B* register is ———.

11.17 The instruction for adding the content of location pointed by AR2 to *A* register by the number of bits specified by the ASM bits of ST1 and adding it to *B* is ———. The result should be in *A* register.

11.18 The instruction for subtracting the content of location pointed by AR2 from *A* register by the number of bits specified by the ASM bits of ST1 from *A* is ———. The result should be in *A* register.

11.19 In the ADD #4567, 0, *B,* the content of ——— is added with the constant 4567 and the result is stored in ———.

(a) A,B (b) A, A (c) B,B (d) B,A

11.20 In SUB B, 0, *A* instruction, the content of ——— is subtracted from ——— and the result is stored in ———.

(a) B,A,A (b) A,B,A (c) B,A,B (d) A,B,B

11.21 The content of ACCA is 80 8000 2345h. If the SAT *A* instruction is executed, the content of *A* Reg. becomes ———.

(a) 00 7FFF FFFFh (b) FF 8000 0000h
(c) 80 8000 2345h (d) None of the above

11.22 The content of TREG and location I020h are FFFFh, 8000h. The content of *A* after executing the instruction MPY 20h, *A* is ———.

(a) 00 7FFF 8000 (b) 00 8000 0000
(c) 00 7FFF FFFF (d) FF 8000 0000

11.23 The content of TREG and location 1020h are FFFFh, 8000h. The content of *A* after executing the instruction MPYR 20h, *A* is ———.

(a) 00 7FFF 8000 (b) 00 8000 0000
(c) 00 7FFF FFFF (d) FF 8000 0000

11.24 The instruction for multiplying the contents of the location pointed by AR3 with the program memory 1120h and adding the product to ACC *A* is ———.

11.25 The instruction for multiplying the contents of the location pointed by AR3 with the program memory 1120h and adding the product to ACC *A* is ———. In addition to this, this instruction should also copy the content of location pointed by AR3 to the next higher location, i.e. (AR3)+1.

11.26 The instruction LTD 1000h moves the content of the location 1000h to ——— and to the memory location ———.

(a) TREG, 1002h (b) TREG, 1000h
(c) ACCA, 1002h (d) ACCA, 1000h

11.27 The instruction LTD 10h copies the content of location 10h in the current data page to ——— and the location ——— in the current page.
(a) TREG, 11h (b) TREG, 0Fh
(c) ACCA, 11h (d) ACCA,0Fh

11.28 The ARs which can be used for specifying the address for the data memory values for FIRS instruction are ———.
(a) AR0-AR1 (b) AR2-AR3 (c) AR4-AR5 (d) AR6-AR7

11.29 The FIRS instruction is of length ——— and requires cycles required for execution.
(a) 1, 1 (b) 1, 2 (c) 2, 1 (d) 2, 2

11.30 The FIRS *AR4+, *AR4+, coeff instruction multiplies ——— bits of ——— with the program memory address coeff and adds the result to the value in ———.
(a) (32-16), A, B (b) (15-0), A, B
(c) (32-16), B, A (d) (15-0), B, A

11.31 The FIRS *AR4+, *AR4+, coeff instruction adds the content of location pointed by AR4 & AR5 and loads them into ——— bits of Accumulator ———.
(a) (32-16), A (b) (15-0), B (c) (32-16), B (d) (15-0), A

11.32 The instruction which enables a branch to location 1050h if AR5 is zero and also postdecrements AR5 by 1 is ———.

11.33 The instruction which decrements AR5 first and then branches to 1050h if AR5 is zero, is ———.

11.34 The Max B instruction compares ACC A with ACCB and stores the maximum value in the ——— register. The carry bit is set to 1 if the accumulator ——— has the maximum value.
(a) B,B (b) B,A (c) A,B (d) A, A

11.35 The Min A instruction compares ACC A with ACCB and stores the maximum value in the ——— register. The carry bit is set to 1 if the accumulator ——— has the maximum value.
(a) B,B (b) B,A (c) A,B (d) A, A

12 APPLICATION PROGRAMS IN C54X

In this chapter some application programs in ′C54X are considered. In Chapter 11, the instruction set of ′C54X is discussed. For writing programs in ′C54X, in addition to the knowledge on the instruction set of ′C54X, the interdependencies between the various instructions should also be taken into account. This is because of the instruction pipelining and concurrency of execution of different instructions in different phases in ′C54X. Hence, the details on the instruction pipeline are presented first. Next the details on the assembly language programming and executing the programs using the code composer studio are considered. The details on the execution of the programs using both the ′C54X simulator and ′C5402-based DSP starter kit are discussed. Finally, the application programs in ′C54X are considered. There are three different approaches to write a ′C54X program:

Using Assembly Language
Using the Algebraic Instruction Set
Using the High Level Language C In this chapter only the first approach is discussed.

PIPELINE OPERATION 12.1

The ′C54X CPU has a six-level deep instruction pipeline. The six stages of the pipeline are independent of each other, which allows overlapping execution of instructions. During any given cycle, from one to six different instructions can be active, each at a different stage of completion. The six levels and functions of the pipeline structure are

Program Prefetch Program address bus (PAB) is loaded with the address of the next instruction to be fetched.

Program Fetch An instruction word is fetched from the program bus (PB) and loaded into the instruction register (IR). This completes an instruction fetch sequence that consists of this and the previous cycle.

Decode The contents of the IR are decoded to determine the type of memory access operation and the control sequence at the data-address generation unit (DAGEN) and the CPU.

Access DAGEN outputs the read operand's address on the data address bus, DAB. If a second operand is required, the other data address bus, CAB, is also loaded with an appropriate address. ARs in indirect addressing mode and the stack pointer (SP) are also updated. This is considered the first of the two-stage operand read sequence.

Read The read data operand(s), if any, are read from the data buses, DB and CB. This completes the two-stage operand read sequence. At the same time, the two-stage operand write sequence begins. The data address of the write operand, if any, is loaded into the data write address bus (EAB). For memory-mapped registers, the read data operand is read from memory and written into the selected memory-mapped registers using the DB.

Execute The operand write sequence is completed by writing the data using the data write bus (EB). The instruction is executed in this phase.

12.1.1 Branch, Call and Return Instructions in Pipeline

Similar to ′C5X, in ′C54X, the undelayed call and return instructions (e.g., CALL sub, B loop) require four clock cycles to start execution in the new address specified by the call/branch instruction. Out of this, two clock cycles are required to flush out the pipeline. The delayed call and return instructions (e.g., CALLD sub, BD loop) require four clock cycles to start execution in the new address specified by the call/branch instruction. Out of this 2-clock cycles are used to execute either a single 1-word instruction or two 1-word instructions following the delayed branch/call instructions. The undelayed conditional call and branch (e.g., CC sub, BC loop) require five clock cycles out of which the last two cycles are used to flush out the pipeline. The delayed conditional call and branch instructions (e.g., CCD sub, BCD loop) require five clock cycles to start execution in the new address specified by the call/branch instruction. Out of this two clock cycles are used to execute either a single 2-word instruction or two 1-word instructions following the delayed branch instructions.

The delayed as well as undelayed return instructions (ret and retd) both require five clock cycles. Out of this two clock cycles are used to execute either a single 2-word instruction or two 1-word instructions following the delayed return instructions.

12.1.2 Dual-Access Memory and the Pipeline

The ′C54X features on-chip memory that supports two accesses in a single cycle. This dual-access memory is organised as several independent memory blocks. Simultaneous accesses to different blocks are supported with no conflicts: while one instruction in the pipeline accesses one block, another instruction at the same stage in the pipeline can access a different block without conflict. Furthermore, each memoiy block supports two accesses in a single cycle: two instructions, each in different stages of the pipeline, can access the same block simultaneously. However, a conflict can occur when two simultaneous accesses are performed on the same block. The ′C54X CPU resolves these conflicts automatically. This however introduces one clock cycle latency that is, it requires one additional clock cycle to complete the execution. If a dual-access memory block is mapped in both program and data spaces, an instruction fetch will conflict with a data operand read access if they are performed on the same memory block.

Another conflict arises if a single-operand write instruction is followed by an instruction that does not perform a write access and this instruction is followed by a dual-operand read instruction.

Example

```
STL A, *AR3+
LD #0, A
ADD *AR4+. *AR5+, A; AR3 and AR5 both point to the same dual-access memory block
```

The CPU resolves the conflict by inserting a dummy cycle after the first instruction.

12.1.3 Single-Access Memory and the Pipeline

The ′C54X also features on-chip single-access memory that supports one access per cycle to each memory block. There are two different types of single-access memory that are available on ′C54X devices:

Single Access Read-Write Memory (SARAM)
Single-Access Read-Only Memory (ROM or DROM)

Both types of single-access memory behave similarly in terms of pipelined accesses, with the exception that ROM and DROM cannot be written to. These memory blocks are contiguous in memory with the first block beginning at the start address of SARAM or ROM. Simultaneous accesses with no conflicts are supported by single-access memory as long as the access are to different memory blocks; while one instruction in a pipeline stage accesses one memory block, another instruction can access a different memory block in the same cycle without any conflict.

A conflict can occur when two simultaneous accesses are performed on the same memory block. In case of such a conflict, only one access is performed in that cycle and the second access is delayed until the following cycle. This results in a one-cycle pipeline latency.

A pipeline conflict due to single-access memory may occur in several different situations.

Dual-Operand Instructions Many instructions have two memory operands to read or write data. If both operands are pointing to the same single-access memory block, a pipeline conflict occurs. The CPU automatically delays the execution of that instruction by one cycle to resolve the conflict.

Example

```
MAC *AR2+, *AR3+%, A, B ;This instruction will take two cycles if both
                        ;operands are in the same SARAM or DROM
                        ;block.
```

32-Bit Operand Instructions Instructions that read 32-bit memory operands still take only one cycle to execute, even if their operand is in single-access memory. Single-access memory blocks are designed to allow a 32-bit read to occur in one cycle. Instructions that write 32-bit operands take two cycles to execute.

Example

```
DLD *AR2, A ;This instruction takes only one cycle even
            ;if the operand is in single-access memory
```

Read-Write Conflict If an instruction that writes to a single-access memory block is followed by an instruction that reads from the same single-access memory block, a conflict occurs because both instructions try to access the same memory block simultaneously. In this case, the read access is delayed automatically by one cycle.

Example

```
STL A, *AR1+ ;AR1 and AR3 points at the same SARAM block.
LD *AR3, B   ;This instruction takes one additional
             ;cycle due to a memory access conflict
```

On the other hand, a dual-operand instruction that has a read operand and a write operand does not cause this conflict because the two accesses are done in two different pipeline stages.

Example

```
ST A, *AR2+          ;This instruction does not take any
||ADD *AR3+, B       ;extra cycles, even if AR2/AR3 point
                     ;at the same single-access memory block
```

Code-Data Conflict Another type of memory access conflict can occur when SARAM or ROM is mapped in both program and data spaces. In this case, if instructions are fetched from a memory block and data accesses (read or write) are also performed on the same memory block, the instruction fetch is delayed by one cycle.

Example

```
LD *AR1+, A          ;This read data access delays a
                     ;subsequent instruction fetch
STH A, *AR2          ;This write data access delays a
                     ;subsequent instruction fetch
```

This situation causes significantly higher pipeline latency than the cases described previously. This is because each time there is a read or write access to the memory block, the pipeline is stalled for one cycle. It is generally recommended that each single-access memory block be reserved for either data or program storage to avoid hits each time a data access is made to that block.

12.1.4 Pipeline Latencies

The ′C54X pipeline allows multiple instructions to access CPU resources simultaneously. Because CPU resources are limited, conflicts can occur when one CPU resource is accessed by more than one pipeline stage. Some of these pipeline conflicts are resolved automatically by the CPU by delaying accesses. Other conflicts are unprotected and must be resolved by the programmer. In general, unprotected conflicts are resolved by rearranging instructions or by inserting NOP instruction (no operation performed). They can also be avoided by using only instructions that do not create any pipeline conflicts or by observing necessary delays before certain registers are accessed.

12.1.5 Recommended Instructions for Accessing Memory-Mapped Registers

Unprotected pipeline conflicts can occur when any one of the following memory-mapped registers is accessed:

Auxiliary Registers (AR0-AR7)
Block Size Register (BK)
Stack Pointer (SP)
Temporary Register (T)
Processor Mode Status Register (PMST)
Status Registers (ST0 and ST1)
Block-repeat Counter Register (BRC)
Memory-mapped Accumulator Registers (AG, AH, AL, BG, BH, BL)

However, certain instructions can access these registers without causing pipeline conflicts if the appropriate latency cycles are noted. Table 12.1 lists these instructions. Table 12.1 is valid only if programmers limit themselves to those instructions that are listed in column 2 in order to perform functions listed in column 1. Refer to the TMS320C54X CPU and Peripherals reference manual to find the latency of each individual instruction. Furthermore, this table is provided as a quick reference for pipeline latencies.

Table 12.1 *Recommended instructions for accessing memory-mapped registers*

Category and function	*Instruction(s)*	*Latency and restrictions*
1. Writing to ARx/BK without using an accumulator	STM MVDK MVMM MVMD	ARx update: None BK update: The next word must not use circular addressing
2. Writing to ARx/BK using an accumulator	STLM STH STL Store type *	The next two words (ARx) or three words (BK) must not use the same register. The next instruction must not write to any ARx, BK or SP using STM, MVDK or MVMD
3. Popping ARx/BK from stack	POPM	The next one word (ARx) or two words (BK) must not use the same register. Do not precede a category 3 instruction with any category 2 or 5 instruction that writes to any ARx, BKorSP
4. Writing to SP without using an accumulator	STM MVDK MVMM MVMD	None if CPL = 0. The next one word must not use SP if CPL = 1
5. Writing to SP using an accumulator	STLM STH STL Store type *	The next two (if CPL = 0) or three (if CPL = 1) words must not use SP. The next instruction must not write to ARx, BK or SP using STM, MVDK or MVMD.
6. Writing to T without using an accumulator	STM MVDK LD Smem, T LDSmem, T||ST	None
7. Writing to T using an accumulator	STLM STH STL	The next word must not use T
8. Writing to BRC without using an accumulator	STM MVDK	None
9. Writing to BRC using an accumulator	STLM STH	The next instruction must not be a RPTB[D]

(Contd.)

Table 12.1 (Contd.)

	STL Store type *	
10. Writing to ARP	LD #k, ARP	None
11. Writing to DP	LD #k, DP LDSmem, DP	None
12. Writing to CPL	RSBX SSBX	The next three words must not use direct addressing mode
13. Writing to SXM	RSBX SSBX	The next word must not be affected by SXM status
14. Writing to ASM	LD #k, ASM LD Smem, ASM	None
15. Writing to BRAF	RSBX SSBX	The next five words must not contain the last instruction word in the RPTB loop
16. Writing BRC to memory	SRCCD	The next two words must not contain the last instruction word in the RPTB loop
17. Writing to OVLY, MP/MC, or IPTR	ANDM ORM XORM	The next six cycles must not include an instruction fetch from the on-chip memory's address range. An external-bus cycle may cause additional latency
18. Writing to DROM bit	ANDM ORM XORM	The next three words must not access the DROM's address range. An external-bus cycle may cause additional latency.
19. Calculating an exponent	EXP	The next instruction must not use T
20. Stack manipulation in compiler mode (CPL=1)	FRAME POPM/POPD PSHM/PSHD	The next instruction must not use direct addressing mode (CPL = 1).
21. Reading AG, AH, AL, BG, BH, or BL as memory mapped registers	Any instruction that can read from memory	The previous instruction must not modify accumulator A or accumulator B

*Any other store-type instruction. Refer table 7-5 of CPU and Peripherals reference set, Texas Instruments, 1996 for store-type instructions.

CODE COMPOSER STUDIO 12.2

Code composer studio (CCS) is an advanced tool for development and debugging programs for DSPs such as ′C54X and ′6X family of processors from Texas Instruments. It provides an integrated development environment (IDE) wherein editing source file, executing and debugging program and viewing the graphical output waveform can all be carried out using multiple windows simultaneously. It has a no. of advanced features such as the following:

Advanced editor which permits easy file manipulation and which enables to easily view source, include library files, etc. It automatically tracks file dependencies and edits within the IDE supporting the features such as Editing C and assembly source code together, syntax highlighting, parenthesis and brace matching, find and replace, quick search and context-sensitive help.

CCS has an efficient compiler which saves time by programming in C. It includes integrated code generation tools which supports features like graphically configuring build options, background build, etc.

The debugger within the IDE which permits to inject/extract data signals and customize and automate testing. It gathers profile information on one part of the code while debugging another profile interactively. It supports multiple profile points and helps to identify hotspots and measure performance. It also optimises code and permits visualization of data.

Real-time data exchange (RTDX) tool in CCS enables real-time analysis. This sets up a real-time channel between host and target. It supports a 20KB per second bandwidth on ‘C6000.

DSP/BIOS in the CCS includes a priority-based, preemptive real-time scheduler, It also supports multi-threading and minimal interrupt latency.

A detailed account of the CCS can be obtained from the Manual (Tutorials on TMS320c54X Code Composer Studio).

12.2.1 An Introduction to Debugging Using CCS

The CCS can be invoked by double clicking on the CCS icon. If the CCS version 1.10 is installed, this enables the ′C54X program to be simulated. If the CCS version 1.11 is installed, this enables the ′C54X program to be executed on the 5402 based DSK. CCS version 1.11 may also be used without the kit. In this case when the system displays the message.

Failure to initialise target DSP Host port 378h, Click on the ignore option. It will be assumed that the CCS is properly installed and hence the installation details would not be considered here. The procedure for executing a assembly language program is discussed here.

After invoking the CCS, a window similar to that shown in Fig. 12.1 appears.

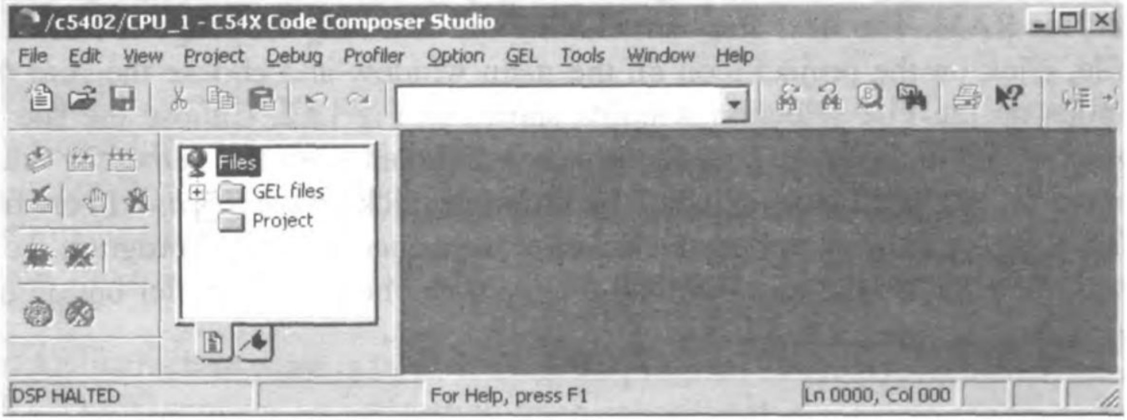
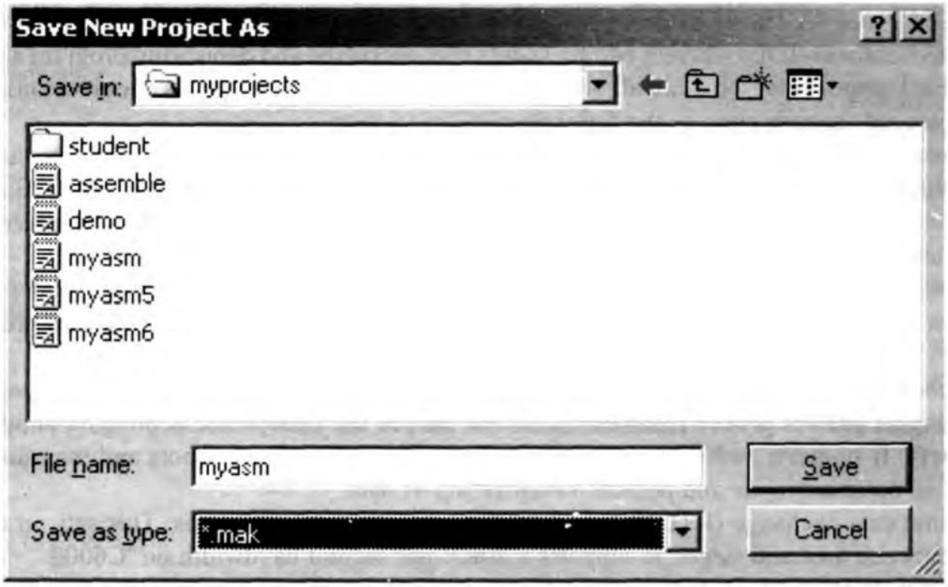

/c5402/CPU_1 - C54X Code Composer Studio
File Edit View Project Debug Profiler Option GEL Tools Window Help
Files
GEL files
Project
DSP HALTED
For Help, press F1
Ln 0000, Col 000

Fig. 12.1 *Code composer studio main window*

First a project is created by clicking on project This pops up a menu with a no. of options. Click on the option new. This opens up the file selection window similar to one shown in Fig. 12.2.



Save in: myprojects

File name: myasm

Save as type: *.mak

Fig. 12.2 *File selection window of code composer studio*

The file type and file name are specified in this window. Choose the file type as .mak and enter a file name, for example, *myasm* and then enter *save.* Now the new project has been created and entered into the menu with heading **files.**

Under this menu, if the icon project is clicked, it can be verified that the new project file with the name as myasm.mak.is included If you click on myasm.mak, the list of files in this batch file is displayed. To this mak file, the hello.cmd file has to be included to enable the assembly language program to be linked after it is assembled. To include this file, click on the project icon on the main window and choose the *Add files* to project option. This pops up the file selection window similar to one shown in Fig. 12.2. Choose the file type as *.cmd* and file name as *Hello* and then enter *open.* The *Hello.cmd file* may be edited to configure the on-chip and on-board RAM anywhere within the permitted memory map for RAM in 5402. Click on the *Hello.cmd* in the **files** window to see its contents. It contains the memory map details such as the internal data memory (IDATA) and the external data memory (EDATA), their origin and the length. This may be edited to suit our requirement within the space allocated for RAM. The next step is including the assembly language file to be executed. To include this file, click on the project icon on the main window and choose the *Add files* to project option. This pops up the File selection window shown in Fig. 12.2. Choose the file type as *.asm,* choose the folder where the required *asm* file is stored and then enter file name, for example, *firs* and then enter *open.* Now the *asm* file is included. Next double click on the *firs.asm* in the files window to edit this file. This file has to be compiled and linked to the project to run the program. To do this, in the files window, single click on the asm file, and then double click the compiler button on the tool bar shown in Fig. 12.3.

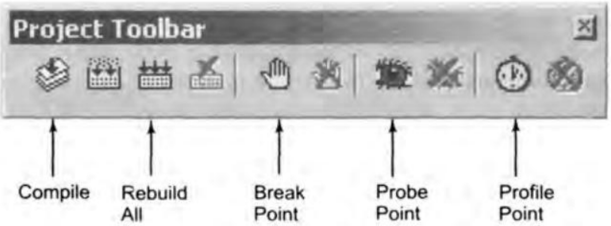
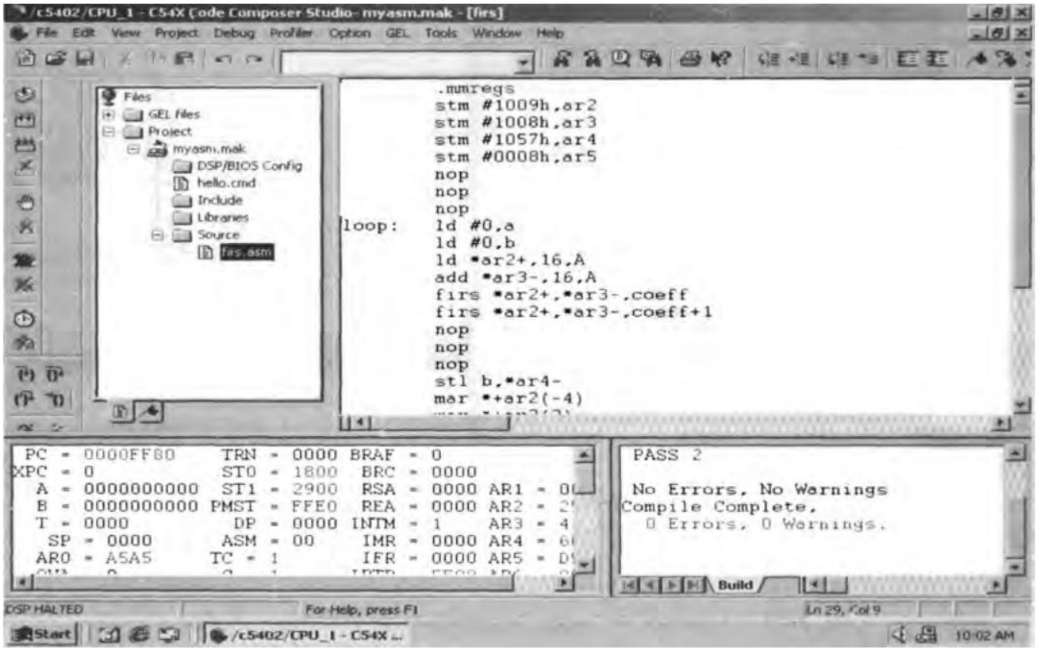


Fig. 12.3 *CCS project tool bar*

The *firs.asm* file is compiled and the compiler reports if there are errors in the program. If there are errors, correct the errors in the source file by double clicking on the firs.asm in the **files** window and repeat the above process. Figure 12.4 shows a sample of the CCS display after a file is compiled. The .asm file is also listed in this figure.



```
            .mmregs
            stm #1009h,ar2
            stm #1008h,ar3
            stm #1057h,ar4
            stm #0008h,ar5
            nop
            nop
            nop
loop:       ld #0,a
            ld #0,b
            ld *ar2+,16,A
            add *ar3-,16,A
            firs *ar2+,*ar3-,coeff
            firs *ar2+,*ar3-,coeff+1
            nop
            nop
            nop
            stl b,*ar4-
            mar *+ar2(-4)
```

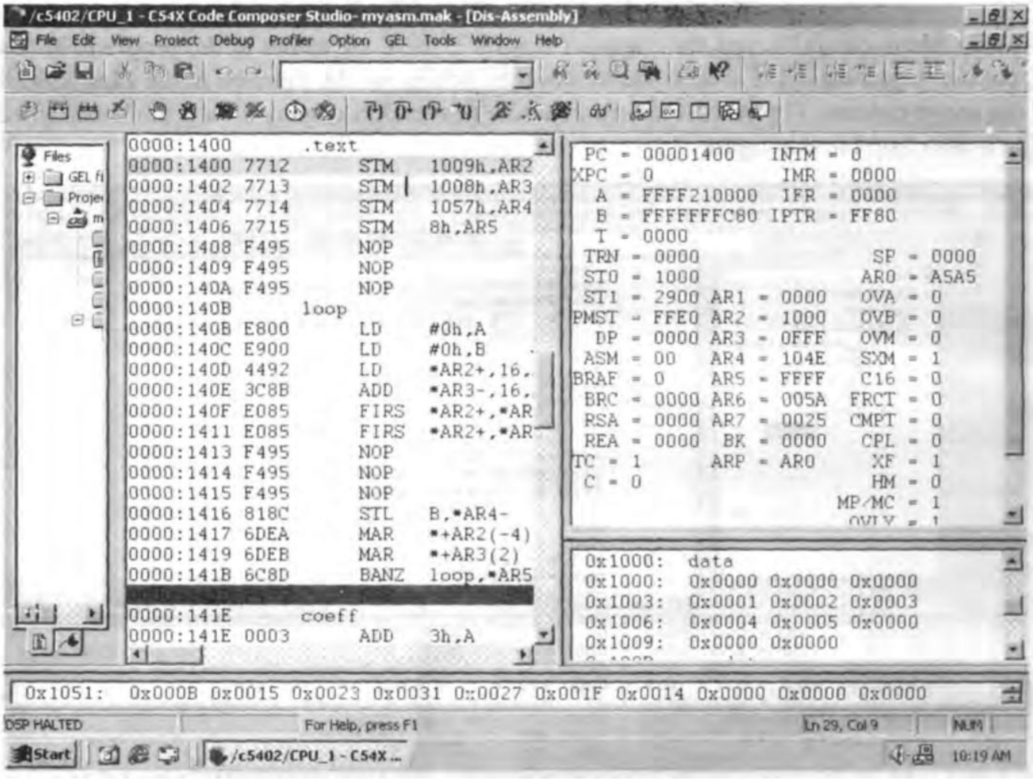
```
 PC  = 0000FF80      TRN  = 0000 BRAF = 0
XPC  = 0             ST0  = 1800  BRC = 0000
  A  = 0000000000    ST1  = 2900  RSA = 0000 AR1 = 0
  B  = 0000000000   PMST  = FFE0  REA = 0000 AR2 = 2
  T  = 0000           DP  = 0000 INTM = 1    AR3 = 4
   SP  = 0000        ASM  = 00    IMR = 0000 AR4 = 6
  AR0  = A5A5        TC  = 1      IFR = 0000 AR5 = D
```

```
 PASS  2

 No Errors, No Warnings
Compile Complete,
  0 Errors, 0 Warnings.
```

Fig. 12.4 *A sample of the compiler output of the CCS*

When the program is successfully compiled, it is linked by double clicking the *build all* button in the tool bar window. Now the assembly language program is ready to be executed. To execute this program enter the *file* option in the main window and click on the *load program.* Note the program is loaded and disassembly of the program loaded appears in the disassembly window. To verify the correctness of the operation and to debug the program, the registers and memory may have to be viewed. To view them, click on the *view* option in the main window and choose the *CPU registers.* This pops up the CPU register window, move it to a convenient place so that the disassembly window is visible. Next click on the *view* option in the main window and choose the *memory* option. This opens up another window where the details of the contents of the chosen memory area is displayed. Move it to a convenient place so that the disassembly window is visible. Next step is setting break points. Place the cursor in the disassembly window at the instruction upto which the program should be run. Now click on the icon for the break point in the project tool bar shown in Fig. 12.3. Now the program is ready to be run. Figure 12.5 shows a sample debugger screen in the CCS. It displays two memory areas, the contents of CPU registers and the disassembly of the program to be executed along with the points where the break points are set. Now click on the *debug* button followed by *run;* the program will be executed till the point at which the break point is set. Alternately, the program may be executed using single stepping. Press the control key F8 for this.


/c5402/CPU_1 - C54X Code Composer Studio- myasm.mak - [Dis-Assembly]
File Edit View Project Debug Profiler Option GEL Tools Window Help
Files
GEL fi
Proje
Start
/c5402/CPU_1 - C54X ...
10:19 AM

```
0000:1400          .text
0000:1400 7712          STM   1009h,AR2
0000:1402 7713          STM   1008h,AR3
0000:1404 7714          STM   1057h,AR4
0000:1406 7715          STM   8h,AR5
0000:1408 F495          NOP
0000:1409 F495          NOP
0000:140A F495          NOP
0000:140B          loop
0000:140B E800          LD    #0h,A
0000:140C E900          LD    #0h,B
0000:140D 4492          LD    *AR2+,16,
0000:140E 3C8B          ADD   *AR3-,16,
0000:140F E085          FIRS  *AR2+,*AR
0000:1411 E085          FIRS  *AR2+,*AR
0000:1413 F495          NOP
0000:1414 F495          NOP
0000:1415 F495          NOP
0000:1416 818C          STL   B,*AR4-
0000:1417 6DEA          MAR   *+AR2(-4)
0000:1419 6DEB          MAR   *+AR3(2)
0000:141B 6C8D          BANZ  loop,*AR5

0000:141E          coeff
0000:141E 0003          ADD   3h,A
```

```
 PC = 00001400    INTM = 0
XPC = 0            IMR = 0000
  A = FFFF210000   IFR = 0000
  B = FFFFFFFC80 IPTR = FF80
  T = 0000
 TRN = 0000                 SP = 0000
 ST0 = 1000                ARO = A5A5
 ST1 = 2900 AR1 = 0000     OVA = 0
PMST = FFE0 AR2 = 1000     OVB = 0
  DP = 0000 AR3 = 0FFF     OVM = 0
 ASM = 00   AR4 = 104E     SXM = 1
BRAF = 0    AR5 = FFFF     C16 = 0
 BRC = 0000 AR6 = 005A    FRCT = 0
 RSA = 0000 AR7 = 0025    CMPT = 0
 REA = 0000  BK = 0000     CPL = 0
TC = 1      ARP = AR0       XF = 1
 C = 0                      HM = 0
                         MP/MC = 1
                          OVLY = 1
```

```
 0x1000:  data
 0x1000:  0x0000 0x0000 0x0000
 0x1003:  0x0001 0x0002 0x0003
 0x1006:  0x0004 0x0005 0x0000
 0x1009:  0x0000 0x0000
```

```
0x1051:  0x000B 0x0015 0x0023 0x0031 0::0027 0x001F 0x0014 0x0000 0x0000 0x0000
```

```
DSP HALTED          For Help, press F1          Ln 29, Col 9          NUM
```

Fig. 12.5 *A sample debugger screen of code composer studio*

This completes the lists of steps required for creating a new project and running it. When a project is already created, whenever CCS is invoked afresh, the project has to be opened before it can be edited or executed.

The above example introduces only some of the features of the CCS. It does not consider features such as compiling and running a C program, displaying the output of the program in graphical form, etc. For details of the more advanced features such as these, the CCS tutorial may be referred. The CCS also has a *help* button in the main window. It provides a wealth of information, such as the details on the DSK, the component lay out, circuit diagram, details on the components used, software used, viz., CCS, tutorial on using the CCS, ′C54X instruction set with examples including timing details, the details on the CPU and peripheral registers, memory map and so on.

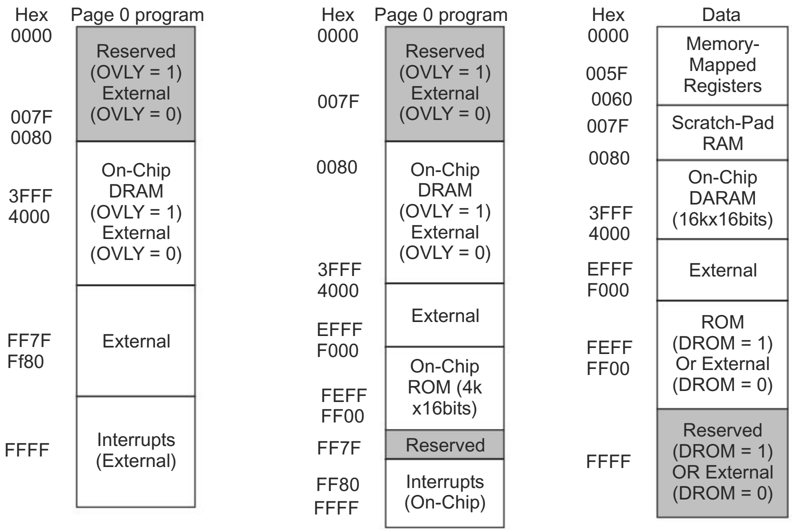


Fig. 12.6a *Memory map for the ′5402*

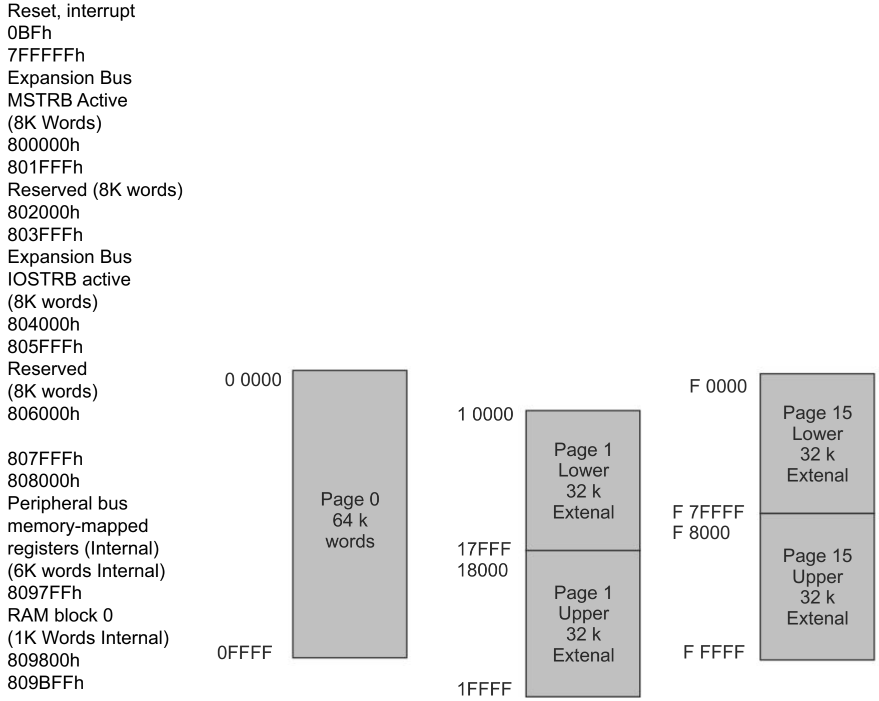


Fig. 12.6b *Extended program memory for ′C5402*

AN OVERVIEW OF THE ′C5402-BASED DSK 12.3

′C5402-based DSP starter kit (DSK) is available with the CCS as the development and debugging tool. The details on the on-chip memory and peripherals on TMS320C5402 is given in Table 12.2. The on-chip memory is augmented by two external RAM in VC5402-based DSK operating at 100 MHz CPU clock. It has 64K 16-bit words of external one wait-state SRAM and 256K words of FLASH memory. The memory map of 5402 is given in Fig. 12.6(a). The on-chip and the off-chip RAM can be configured to lie anywhere within the permissible address space. The memory map of the DSK or the CCS simulator as well as the actual address space allocated for each of the on-chip and off-chip RAM may be edited using the *option* in the main window of CCS and by selecting *memory* map in this menu or by editing the hello.cmd file attached to the project. As mentioned earlier the CCS gives complete details about the DSK.

Table 12.2 *Program and data memory and serial ports on the TMS320C5402 devices*

Memory type	*Size*	*Serial ports*	*No.*
ROM:	4K	Synchronous	0
Program	4k	Buffered	0
Program/data	4K	McBSP	2
DARAM†	16K	TDM	0
SARAM†	0		

†The dual-access RAM (DARAM) and single-access RAM (SARAM) may be configured as data memory or program/data memory.

The DSP interfaces to external SRAM, FLASH memory and an expansion memory interface connector through its 16-bit external memory interface (EMIF). The DSP’s EMIF is also routed to an expansion memory interface connector to allow a daughter board to be used.

The external SRAM and FLASH devices on the board are +3.3 V devices. The expansion memory connector is able to support both +3.3 V and +5 V devices since it uses +5V-tolerant buffers. The amount of external data memory available depends on the setting of the DROM bit. If DROM = 0, then the region from 0x4000h to 0xFFFFh (48K words) is external memory (FLASH or SRAM). (As described below, a paging mechanism exists for accessing more than the 32K-word blocks of either SRAM or FLASH when used in data space.) If DROM = 1, then external data memory is available only from 0x4000h to 0xEFFFh. (See the 5402 data sheet for additional details of its internal memory map and external memory interface.)

Whether on-board or daughterboard data memory is accessed depends on the DMSEL control register bit or the DMSEL DIP switch setting. If DMSEL = 0 (default), then on-board data memory is used. If DMSEL=1, then daughterboard memory is available starting at 0x8000 (32k blocks with same page register feature as on board).

The amount of external program memory available will depend on the setting of the OVLY bit and the MP/$\overline{\text{MC}}$# DIP switch setting. If the OVLY bit is 0 and MP/$\overline{\text{MC}}$# = 0, then the program memory region from 0x0000 to 0xEFFF (60K words) is mapped to the external memory, either FLASH or SRAM depending on the state of the FLASHENB control register bit. On power-up, the FLASHENB bit is set to allow booting from the FLASH device. Software may then clear this bit to enable the one wait-state SRAM into the same memory space. If MP/$\overline{\text{MC}}$# = 0, then the region from 0xF000 to 0xFFFF is reserved for on-chip ROM and interrupt vectors and the external program memory is not available in page 0, but may still be available in other pages, depending on the setting of the OVLY bit.

If MP/$\overline{MC}$# = 1 and OVLY - 0, the region from 0x0000h to 0xFFFFh is decoded for external memory, either FLASH or SRAM. In the case of 70-ns FLASH memory (FLASHENB = 1), with 100 MHz bus speed, seven wait states will be required via the 5402's internal wait-state generator. In the case of SRAM (FLASHENB = 0), one wait state will be required at 100 MHz bus speed. Since the SRAM and the FLASH share the same wait-state generator, the 5402's internal wait-state generator will have to be changed from one wait state when accessing SRAM to seven wait states when accessing FLASH. If MP/ $\overline{MC}$ = 1 and OVLY = 1, only the region from 0 x 4000h to 0xFFFFh is mapped to external memory.

The DSK has two TLC320AD50C analog interface circuit on board. The AIC provides high resolution low speed signal conversion from D/A and A/D using oversampling delta sigma modulation. This device consists of two serial synchronous conversion paths, one for each direction, and is connected to McBSP of the 5402. It includes an interpolation filter before the DAC and the decimation filter after the ADC. It has five user controllable registers using which the operating mode of the AIC can be programmed using the secondary communication. Primary serial communication is used to transmit and receive conversion signal data. The ADC and DAC length can be chosen to be either 15 or 16 bit. In the 15-bit mode, the last bit of the primary serial communication word of DAC is used to request a secondary communication. More details on the AIC can be had from the TI Mixed Signal Products user manual.

The DSK has an analog network interface DAA and an AD50 AIC provide DSP access to a single telephone interface via one of the multichannel buffered serial ports (McBSP0). McBSP0 can optionally be routed to an expansion connector under software control.

In the DSK, microphone and speaker interfaces (via 3.5 mm audio jacks) are provided via a second AD50 AIC that is connected to the DSP's second McBSP (McBSP 1). McBSP 1, along with signals for its two timers and an external interrupt, is also routed to an expansion connector to allow a daughter board to be used. The audio input is AC-coupled and includes an amplifier with fixed 10-dB gain, a stage for conversion from single-ended to differential (prior to being digitised by the TLC320AD50 connected to the DSP's McBSP 1) and passive filtering (between the DSK's audio jacks and the CODEC) for increased performance.

The DSK provides an eight-position DIP switch control for external user options. A push button allows the DSK to be manually reset. A voltage supervisor monitors the internally generated voltage, and will hold the board in reset until the supplies are within operating specifications. The four LEDs on the DSK provide a power-on indicator and three user-controlled indicators.

The DSK provides embedded IEEE Std. 1149.1 (JTAG) emulation that is directly compatible with the code composer debugger. The DSK can also be used with an external XDS510 for the PC via its external JTAG connector. The DSK's embedded JTAG emulation, provided by a test bus controller (TBC), and the DSP's Host Port Interface (HPI) is available via an IEEE-1284 compliant parallel interface.

INTRODUCTION TO C54X ASSEMBLY LANGUAGE PROGRAMMING 12.4

The file names of assembly language programs of ′C54X has extensions as .asm. The TMS320C54X assembly language users manual gives a complete list of directives for the assemblers as well as linker. Only those directives which are required for writing a simple assembly language program in ′C54X are discussed here.

The assembly language is split into three sections:.text section, .data section and .bss section.

The .text section consists of the assembly program which is translated into object code by the assembler and loaded into program memory for execution. By default, .text section is assumed and the assembly language instructions including the pseudo instructions are assembled into the program memory area.

The .data section consists of constants and variables which are initialised and are loaded into the data memory area. The origin of the data memory is determined by the hello.cmd file. It corresponds to the address to which IDATA in the hello.cmd file is initialised.

The .bss section is used to reserve a block of memory which is uninitialised. Some of the assembler directives are listed in Table 12.3.

Table 12.3 *Some assembler directives*

Assembler directive	*Description*
.mmregs	Permits the memory map registers to be referred using the names such as AR0, SP etc.
.include "XX"	Informs the assembler to insert a list of instructions in the file XX to be inserted in this place and assemble it
.end	The end of the assembly language program
.data	Assemble into data memory area
.text	Assemble into program memory area
.equ	Equate a symbol to a constant
.word x, y, ..., z	Reserves 1 6-bit locations and initialises them with values x, y, ..., z. This may be used in both the text section and data section
.space n	Reserve and initialise *n* bits of memory and when a label is used with this directive, the label is assigned the address of the first word of the block reserved
.bes n	Reserve and initialise *n* bits words of memory and when a label is used with this directive, the label is assigned the address of the last word of the block reserved

Program 12.1 contains Example 1.asm. It gives an example of a ′C54X assembly language program. In this program the use of the assembler directives given in Table 12.2 is illustrated. In this program, the operands required for the program are specified using two methods. One method is to use the .data directive and place the data in the data memory area specified in the hello.cmd file. The second method is to specify the operand in the .text section and copy the operands to the required location using block transfer. Program 12.1 adds three vectors of size 5. Two vectors are stored in locations starting from 1000h and 1100h respectively. The third vector is initially stored in the program memory starting from location start. This vector is copied to the data memory area 1500-1504h using the block transfer. Program 12.2 gives the bcopy.asm which is used for copying the block of data from program memory area to the data memory area. The resultant vector is stored in the memory area 1200-1204h. The vector addition is achieved by adding the first element of each vector and storing the result in the first location 1200h. Next the second elements are added and the result is stored in the second location 1201h and so on. The size of vector is specified by the symbol count. By changing the value of this any vector can be added.

Program 12.1 Example1.asm: Vector addition

Label	Mnemonic	Comments
	.mmregs	Permits the use of memory-mapped registers using names instead of their actual memory address
	.text	Assemble into program memory area
Count	.equ 4	Permits the constant 4 to be referred by the symbol count
	Id #0, a	Accumulator A loaded with constant 0
	Stm #1500h, ar1	The register AR1 loaded with the constant 1500h
	Stm #1000h, ar2	The register AR2 loaded with the constant 1000h
	Stm #1100h, ar3	The register AR3 loaded with the constant 1100h
	Stm #1200h, ar4	The register AR4 loaded with the constant 1200h
	Stm #count, ar5	The register AR5 loaded with the constant 4 (count =4)
	.include “bcopy.asm”	
	Stm #1500h, ar1	The register AR1 loaded with the constant 1500h
	Nop	Nop introduced to take care of pipeline latency for stm
loop	Add *ar2+, *ar3+, a	The contents of location pointed by AR2 and AR3 are added and the result is stored in the ACC A higher order word. In a dual-operand indirect-addressing mode, the result is stored only after shifting the result by 16 bits towards left. AR2 and AR3 are incremented by 1
	Add *ar1+, 16, a	The content of location pointed by AR1 is left shifted by 16 bits and added to A register AR1 is incremented
	Sth a, *ar4+	Result contained in higher word of A register is stored in the location pointed by AR4. AR4 is incremented
	Banz loop, *ar5-	If AR5 is not zero, the execution is resumed at the address loop. AR5 is also decremented by 1. Otherwise the execution resumes with the next instruction
	Nop	Nop introduced to enable smooth exit if no branch occurs
start	.word 1, 2, 3, 4, 5	Data required for operation stored in program memory. This data is copied to the data memory

		using the block copy program: bcopy.asm
	.data	This assembler directive causes the following data to be stored into the data memory area
Data1	.word 5, 6, 7, 8, 9	five words are stored. The address corresponding to data1 = 1000h if IDATA =1000h in the Hello.cmd linker file
data2	.bes (0fbh*16)	251 words reserved and the datal is assigned the address of the last word i.e., data2 = 10ffh if data1=1000h
data3	.word 10h, 11h, 12h,	
	.word 13h, 14h	Data3 address = 1100h if datal address is 1000h. Five words are stored here
Data4	.space (0fbh*16)	251 words are reserved, the first address of the reserved space, viz, 1105h is assigned to data4
Data5	.word 0, 0, 0, 0, 0	Address of data5 is 1200h: five words initialised to 0
	.end	Assembler terminates the assembly here

Program 12.2 bcopy.asm

Label	Mnemonic	Comments
	rpt #count	Repeat the next instruction count times
	mvpd start, *ar1+	Copy the data from the program memory with the label start to the data memory pointed by AR1. Increment AR1. In the repeat mode the program memory address is incremented and the corresponding data is copied to the data memory

Program 12.3 gives Example2.asm which performs the vector addition using repeat block instruction. The block of instructions which add a vector element of each of the vectors and store the result is repeated a no. of times equal to the size of the vector. For this purpose, the no. of times the block is to be executed is stored in the Block repeat count (BRC) register. (The no. to be stored is one less than the actual no. of times. For example, in the above case 4 has to be stored to execute the block five times.) The end of the block is specified in the RPTB instruction by specifying the label corresponding to the last instruction in the block.

Program 12.3 Example2.asm: Vector addition using repeat block

Label	Mnemonic	Comments
	.mmregs	Permits the use of memory-mapped registers using names instead of their actual memory address
Count	.equ 4	Permits the constant 4 to be referred by the symbol count
	Id #0, a	Accumulator A loaded with constant 0

```
            Stm #1500h, ar1         The register AR1 loaded with the constant 1500h
            Stm #1000h, ar2         The register AR2 loaded with the constant 1000h
            Stm #1100h, ar3         The register AR3 loaded with the constant 1100h
            Stm #1200h, ar4         The register AR4 loaded with the constant 1200h
            Stm #count, brc         Block repeat count register (BRC) stored with
                                    the count
            .include “bcopy.asm”    Inserts the list of instructions in the file
                                    bcopy.asm here
            Stm #1500h, ar1         The register AR1 loaded with the constant 1500h
            Nop                     Nop introduced to take care of pipeline latency
                                    for stm
            Rptb store              Repeats the block of instructions including the
                                    instruction with the label store.
            Add *ar2+, *ar3+, a     The contents of location pointed by AR2 and AR3 are
                                    added and the result is stored in the ACC A higher
                                    order word. In a dual-operand indirect-addressing
                                    mode, the result is stored only after shifting the
                                    result by 16 bits towards left. AR2 and AR3 are
                                    incremented by 1
            Add *ar1+, 16, a        The content of location pointed by AR1 is left shifted
                                    by 16 bits and added to A register AR1 is incremented
Store       Sth a, *ar4+            Result contained in higher word of A register Is stored
                                    in the location pointed by AR4. AR4 is incremented
                                    This is the last instruction in the repeat block
            Nop                     Nop introduced to enable smooth exit if no branch
                                    occurs
Start       .word 1, 2, 3, 4, 5     Data required for operation stored in program
                                    memory. This data is copied to the data memory
                                    using the block copy program: bcopy.asm
            .data                   This assembler directive causes the following data
                                    to be stored into the data memory area
Data1       .word 5, 6, 7, 8, 9     five words are stored. The address corresponding to
                                    data1 = 1000h if IDATA -1000h in the Hello.cmd linker
                                    file
Data2       .bes (0fbh*16)          251 words reserved and the datal is assigned the
                                    address of the last word, i.e., data2 - 10ffh
                                    if datal=1000h
Data3       .word 10h, 11h, 12h,
            .word 13h, 14h          Data3 address = 1100h if datal address is 1000h.
                                    Five words are stored here
Data4       .space (0fbh*16)        251 words reserved, the first address of the reserved
                                    space, viz., 1105h is assigned to data4
Data5       .word 0, 0, 0, 0, 0     Address of data5 is 1200h:five words initialised to 0
            .end                    Assembler terminates the assembly here
```

APPLICATIONS PROGRAMS IN C54X 12.5

12.5.1 Example Program on Immediate Addressing

Program 12.4: immediate.asm gives an example for loading the various registers and memory locations using immediate addressing.

Program 12.4 immediate.asm:Immediate addressing

```
Label             Mnemonic                 Comments
                  .mmregs                  Permits the use of memory-mapped registers using
                                           names instead of their actual memory address
                  ld #1000h, A             A Register loaded with the value 1000h
                  ld #2000h, B             B Register loaded with the value 2000h
                  ld #0011h, 8, A          The constant 0011h is left shifted by eight bits and
                                           the resulting no. 1100 is stored into A register
                  ld #1000h, 16, B         The constant 1000h is left shifted by 16 bits and the
                                           resulting no. 1000 0000h is stored into B register
                  STM #1000h, AR0          AR0 is loaded with the constant 1000h
                  STM #1100h, AR1          AR1 is loaded with the constant 1100h
                  ADD #00FFh, A            The constant 0ffh is added to A register and the
                                           result is stored in A
                  ADD #0011h, 2, A         The constant 0011h is left shifted by two bits and
                                           the resulting no. 0044h is added to A
                  STM #45h, T              The constant 45h is stored into T register
                  MPY #0010h, A            The content of T register multiplied by the constant
                                           10h and the result is stored into A
                  SUB #0022h, A            The constant 22h is subtracted from A and the result
                                           is stored into A
                  SUB #0011h, 3, A         The constant 11h is left shifted by three bits and
                                           the resulting no. 0088h is subtracted from A. The
                                           result is stored into A
                  STM #110Fh, BK           The constant 110fh is stored into circular buffer
                                           size register BK
                  LD #20h, DP              The data page pointer (DP) is initialised with
                                           the value 20h
                  STM #1100h, SP           The stack pointer is initialised to ll00h
                  St #1000h, 1050h         Constant 1000h is stored into the location 1050h
                  .END
```

12.5.2 Example Program on Direct Addressing

Program 12.5: direct.asm gives an example for loading the various registers and memory locations and specifying the operands for the instructions using direct addressing. This program uses the instructions in fill.asm in order to fill up known contents into the memory location. The fill.asm fills the locations 1000h-1FFFh with the numbers 0000h-0FFFh. This simplifies the debugging procedure. When location 101 0h is read we know that its contents should be 0010h. Further by executing the program in single-step mode, we can study the pipeline latency. This can be achieved by looking at the time when the PC points to the location containing the LD 10h, A instruction and the time the no. of cycles after which A becomes 0010h. Similarly by looking at the PC value corresponding to the instruction LD #20h, DP and the no. of cycles required for the DP to actually become 20h, the delay can be found. When the desired result does not occur, NOPs are inserted immediately after the particular instruction to avoid pipeline conflicts. No. of NOPs depends on the pipeline latency given in Table 12.1 and also the no. of instructions which lie in between the two instructions which have pipeline conflict. One way to avoid the use of NOPs is to insert some instructions which are actually required for the program in between the instructions causing pipeline conflict. However, trying to optimise the code this way may be at the cost of readability of the program. Hence use of NOPs may be preferred.

Program 12.5 direct.asm: Direct addressing

Label	Instruction	Comments
	.MMREGS	Permits the use of names for memory-mapped registers
	.include "fill.asm"	Inserts the instructions in the file fill.asm here at the time of assembly. This program fills the memory 1000-1FFFh with the nos. 0000-0FFFh
	RSBX CPL	Compiler mode bit CPL reset to 0
	LD #20h, DP	DP initialised to page 20h. This corresponds to the page starting with the address 1000h
	STM #1100h, SP	SP initialised to the address 1100h
	LD 10h, A	Content of 1010h is loaded into A register Because of the fill operation the content of A becomes 10h
	LD 5h, 2, B	The content of 1005h is left shifted by 2 bits and loaded into B. Because of the fill operation, B register becomes 0014h
	SSBX CPL	CPL bit changed to 1; SP gives the base address for direct addressing
	STM 15h, AR0	Content of 1115h i.e., (SP+15) is stored into AR0
	STL A, 15h	Content of lower word of A stored in location 1115h
	STL A, 3, 20h	Content of lower word of A stored in location 1120h after left shifting it by three bits
	STLM A, AR7	Content of lower word of A stored into AR7
	LD #30h, DP	DP changed to page no. 30h with base address =1800h
	ADD 25h, A	Content of the location 1125h (i.e., SP+25h) is added to A and the result is stored into A
	ADD 7h, 2, A	Content of the location 1107h (i.e., SP+07h) is left shifted by two bits and added to A and the result is stored into A
	SUB 10h, A	Content of the location 1110h (i.e., SP+10h) is subtracted from A and the result is stored into A
	RSBX CPL	The base register for direct addressing is chosen as DP
	NOP	Nops to take care of pipeline latencies

```
NOP
NOP
ADD 12h, A              Content of the location 1812h (i.e., 1800h+12h) is
                        added to A and the result is stored into A
STM #10h, T             Load the constant 10h into T register
MPY 15h, A              Content of the location 1815h (i.e., 1800h+15h) is
                        multiplied with the T register and the result is
                        stored into A
MPY 16h, B              Content of the location 1816h (i.e., 1800h+16h)
                        is multiplied with the T register and the result is
                        stored into A
.End
```

12.5.3 Example Program on Indirect Addressing

Program 12.6 indirect.asm gives an example program on indirect-addressing mode. The ARs used for specifying the operand may be modified in different values in the auxiliary ALU ARAU while the operand is processed by the main ALU. The no. of ways in which they can be modified are postincrement and decrement with and without indexed addressing. They may also postdecrement or increment the ARs or use circular addressing along with indexed addressing and so on. The modification of AR using circular addressing is considered in Program 12.7. The other options are considered in this program.

Program 12.6 Indirect.asm: Example Program on Indirect Addressing

```
Label          Instruction              Comments
               .MMREGS                  Permits the use of names for memory-
                                        mapped registers
               .Include "Fill.Asm"      Inserts the instructions in the file fill.asm here at the
                                        time of assembly. This program fills the memory
                                        1000-1FFFh with the nos. 0000-0FFFh
               Stm #1000h, Ar0          AR0 intialised to the value 1000h
               Stm # H00h, Ar1          AR1 intialised to the value 1100h
               Ld *Ar0+, A              The content of the address pointed by AR0, i.e.,
                                        address 1000h is loaded into A. Because of the fill
                                        operation A will be loaded with 0; AR0 incremented
                                        by 1
               Ld *Ar0+, 4, A           The content of the address pointed by AR0, i.e.,
                                        address 1001h is loaded into A after left shifting it by
                                        four bits. Because of the fill operation A will be
                                        loaded with 0010h, i.e., 0001 left shifted by 4 bits;
                                        AR0 incremented by 1
               Stl A, *Ar1+             Lower order of A stored into the location pointed by
                                        AR1. AR1 incremented by 1
               Stl A, 5, *+Ar1          Lower order of A stored into the location pointed by
                                        AR1. after left shifting it by five bits. AR1
                                        incremented by 1
```

	Ld *+Ar0, 2,, A	Ar0 is incremented by 1. The content of the location pointed by the new value of AR0 is left shifted by two bits and stored into A
	Ld *Ar1+0, B	The content of address pointed by AR1 is loaded into B register Content of AR0 is added to Ar1
	Ld *Ar1(10h), A	The content of address pointed by AR1 + 10h is loaded into A register Content of AR1 is not altered
	Ld *+Ar1(20h), B	AR1 is incremented by 20h. The content of address pointed by the new value of AR1 is loaded into A register
	Ld *+Ar1(-30h), B	AR1 is decremented by 30h. The content of address pointed by the new value of AR1 is loaded into A register
	Add *Ar1+0, B	The content of address pointed by AR1 is added to B. Content of AR0 is added to Ar1
	Sub *Ar0-, 2, A	The content of location pointed by AR0 is left shifted by two bits and subtracted from A. AR0 is decremented by 1
	Stm #10h, T	Load the constant 10h into T register
	MUL *AR1, A	The content of the location pointed by AR1 multiplied with T register and the result stored into A register
	Stm #1200h, Ar2	AR2 initialised to the value 1200h
	Stm # 1300h, Ar3	AR3 initialised to the value 1300h
	add *ar2+, *ar3+, a	The content of the locations pointed by AR2 is added to the content of the location pointed by AR3, the result is left shifted by 16 bits and stored into A. AR2 and Ar3 are incremented
	.End	End of assembly

12.5.4 Example Program on Filling a Memory Block

Program 12.7, Fill.asm is a program for filling numbers 0000h-0FFFh in the memory area 1000h-1FFFh. This uses indirect-addressing mode. In this program AR0 contains the no. of locations to be filled. A Register contains the data to be filled. It is initialised to be 0 and incremented by 1 after filling one location. This program makes use of BANZ instruction for filling the block. This program can also be modified using RPTB instruction. In this case the BRC has to be initialised to be equal to the number of words to be filled - 1. The indirect-addressing mode is used for filling the location in this program.

Program 12.7 Fill.asm: Program for filling a memory block

Label	Instruction	Comments
	.MMREGS	Permits the use of names for memory-mapped registers
	Stm #400h, Ar0	AR0 initialised to 0400h; It denotes the no. of words to be filled

```
                 Stm #1000h, Ar1         AR1 initialised to 1000h; It holds the address of the
                                         location to be filled
                 Ld #000h, A             A initialised to 0
                 Nop                     Nops to take care of pipeline latency after
                 Nop                     Stm instruction
                 Nop
Loop:            Stl A, *Ar1+            One value filled and the address incremented
                 Add #1h, A              A incremented by 1
                 Banz Loop, *Ar0-        Continues in loop till AR0 is not zero;
                                         AR0 decremented
                 Nop
                 Stm #400h, Ar0          AR0 initialised to 0400h; It denotes
                                         the no. of words to be filled -1
                 Stm #1800h, Ar1         AR1 initialised to 1800h; It holds the
                                         address of the location to be filled
                 Ld #800h,               A initialised to 800h
Loop1:           Stl A, *Ar1+            One value filled and the address incremented
                 Add #1h, A              A incremented by 1
                 Banz Loop1, *Ar0-       Continues in loop till AR0 is not
                                         zero; Ar0 decremented
                 Nop
                 .end
```

12.5.5 Example Program on Circular Addressing

Program 12.8: Circular.asm gives an example where operand is fetched using circular addressing individually and also along with index addressing. The circular buffer is assumed to be of size 3. The circular buffer is initialised to be 1000h. First the manner in which the address gets incremented in the circular-addressing mode is illustrated. Next the circular-addressing mode is combined with indexed addressing and the manner in which the address gets incremented is illustrated. AR0 which is used as the index register is initialised to be 2.

Program 12.8 **Circular addressing: Example on both circular and index addressing**

```
Label            Instruction             Comments
                 .mmregs                 Permits the use of names for memory-
                                         mapped registers
                 Stm #3, Bk              Circular buffer length register BK initialised to 3
                 Stm #1000h, Ar1         AR1 initialised to 1000h
                 Nop                     Nops for taking care of pipeline latency after
                 Nop                     Stm #3, BK instruction
                 Ld *Ar1+%, A            Content of 1000h copied to A, AR1
                                         incremented to 1001h
                 Ld *AR1+%, A            Content of 1001h copied to A, AR1
                                         incremented to 1002h
                 Ld *AR1+%, A            Content of 1002h copied to A, AR1
```

```
                           incremented to 1000h
Ld *AR1+%, A               Content of 1000h copied to A, AR1
                           incremented to 1001h
Ld *Ar1+%, A               Content of 1001h copied to A, AR1
                           incremented to 1002h
Ld *AR1+%, A               Content of 1002h copied to A, AR1
                           incremented to 1000h
Nop                        Nops introduced to observe the changes properly
Nop
Stm #02h, Ar0              AR0 which is used as the index register is
                           initialised to 2
Stm #1000h, Ar1            AR1 initialised to 1000h
                           Following lines use both circular and
                           indexed addressing
Ld *AR1+0%, A              Content of 1000h copied tc A, AR1
                           incremented to 1002h
Ld *AR1+0%, A              Content of 1002h copied to A, AR1
                           incremented to 1001h
Ld *AR1-0%, A              Content of 1001h copied to A, AR1
                           incremented to 1000h
Ld *AR1-0%, A              Content of 1000h copied to A, AR1
                           incremented to 1002h
Ld *AR1+0%, A              Content of 1002h copied to A, AR1
                           incremented to 1001h
Ld *AR1+0%, A              Content of 1001h copied to A, AR1
                           incremented to 1000h
Nop                        Nops introduced to observe the changes properly
Nop
.End
```

12.5.6 Convolution using MACP and MACD Instructions

Convolution operation can be performed using either MACP or MACD instruction. When the convolution is performed using the MACP instruction, the input data-memory area is left unaffected. When MACD is used the input data memory is shifted towards right. In other the content of location N is copied to the location N+1. As mentioned in Chapter 6, the MACD instruction is useful when the real time data is to be processed. In this case by shifting the data to the higher memory location, the input data which is no longer required can be gradually pushed out of the processing window. Program 12.9: conv.asm gives the program which uses the MACP instruction. Program 12.10: convd.asm gives the program which uses the MACD instruction. For convolution the input data sequence is assumed to be of length 3 and is stored into the data-memory area. The impulse response coefficients are assumed to be of length 5 and are stored into the program-memory area. The length of the resulting sequence is of length 3+5-1, i.e., 7 and is stored into the data-memory area. Note that if an instruction is to be repeatedly executed N times using RPT instruction, the no. to be loaded as the argument to RPT instruction is N–1. Hence to perform MACP operation five times the repeat instruction used is RPT #4.

Program 12.9 Conv.asm: Convolution using MACP instruction

Label	Instruction	Comments
	.Mmregs	
	Stm Data+4, Ar2	The address Data+4 denotes the address of the first input data; AR2 is initialised to this value
	Stm #1050h, Ar3	AR3 initialised to the starting address of the output array
	Stm #07h, Ar5	Initialise AR5 with the length of the output array
Loop	Ld #0, B	B initialised to 0
	Nop	
	Rpt #4h	One of the output element found by computing the product of the coefficient array with the input array and
	Macp *Ar2-, Coeff, B	Accumulating the products
	Stl B, *Ar3+	One output element is stored
	Mar *+Ar2(6)	AR2, The data array pointer is made to point to the next element in the input array
	Banz Loop, *Ar5-	Computation of next output element taken up if AR5 is not zero. AR5 is decremented by 1
	Nop	
	Nop	
	Nop	
Coeff	.Word 1h, 2h, 3h, 4h, 5h	The impulse response coefficients stored here
	.Word 0, 0, 0, 0	
	.Data	
Data	.Word 0, 0, 0, 0, 3h, 4h, 5h, 0h	The input data stored here. Four zeros are inserted at the beginning and end of the input sequence
	.Word 0h, 0h, 0h	
	.End	

Program 12.10 Convd.asm: Convolution using MACD instruction

Label	Instruction	Comments
	.Mmregs	
	Stm Data+9, Ar2	The address Data+9 denotes the address of the last element in the input array; AR2 is initialised to this value
	Stm #1056h, Ar3	AR3 initialised to the end address of the output array
	Stm #07h, Ar5	Initialise AR5 with the length of the output array
Loop:	Ld #0, B	B initialised to 0
	Rpt #6h	One of the output element found by computing the product of the coefficient array with the input array and
	Macd *Ar2-, Coeff, B	Accumulating the products. Each of

```
                                        the input data is shifted to one
                                        location higher
                Delay *Ar2              The new data is shifted into left of
                                        the input array
                St1 B, *Ar3-            One output element is stored
                Mar *+Ar2(7)            AR2, The data array pointer is made
                                        to point to the end of the input array again
                Banz Loop, *Ar5-        Computation of next output element
                                        taken up if AR5 is not zero. AR5 is
                                        decremented by 1

                Nop
Coeff           .Word 1h, 2h, 3h, 4h, 5h
                .Word 0, 0, 0, 0
                .Data
Data            .Word 0, 0, 0, 3h, 4h, 5h
                .Word 0h, 0h, 0h, 0h
                .End
```

12.5.7 Convolution using FIRS Instruction

The computational complexity of performing convolution can be reduced by a factor of 2 by using the FIRS instruction if either the input sequence or the impulse response has a symmetry about the middle of the sequence. The linear phase FIR filters can be designed to have symmetry about the middle of the sequence. Program 12.11: firs.asm gives an example where the impulse response is of length 4 and has even symmetry about the middle of the sequence. If the coefficients are denoted as h0, h1, h2, h3 and h4, it is assumed that h0=h3 and h1=h4. The input sequence is assumed to be of length 5. In order to perform the convolution, three zeros are appended before and after the given input sequence and the resulting sequence is stored in the location 1000h-1009h in firs.asm using .data directive. Each element of the convolved output is determined by four input values and four impulse response coefficients. So the input sequence is considered in blocks or windows of four elements. To find out the last element of the convolver, the window consisting of the last non-zero input element and the three padded zeros is considered. In firs.asm it corresponds to the window 1006h-1009h. To find the last element in the output array, the input elements in the locations 1007h and 1008h are to be added and multiplied by the second impulse response coefficient. Next the input elements in the locations 1006h and 1009h are to be added and multiplied by the first impulse response coefficient. These two products are added to get the last element. To get the last but one element, the window is moved towards left by 1 location that is, the window 1005h-1008h is used to find the output element. This process is repeated till all the output elements are found. The impulse response elements are stored in the program memory area starting with label coefficient and the coefficients are stored in the reverse order.

To use the FIRS instruction, the higher word of A should contain the sum of two elements which is to be multiplied by one of the impulse response coefficients. To ensure this, second and third element in the processing window of the input array are added and stored into the higher order word of A. Next the FIRS *ar2+, *ar3- instruction is used to multiply this sum with the content of coefficient and add it to B register. The content of the locations pointed by AR2 and AR3 are added and stored into the higher order word of A, that is, the first and last element of the processing window are added and stored into the higher order word of A. The next FIRS instruction multiplies this sum with the next coefficient, viz., the content of coeff+1 and adds it to B register. The two elements outside the processing window are also added and stored into higher word of A. But this sum is not subsequently used as no FIRS instruction follows this.

Program 12.11 Firs.asm: Convolution using FIRS instruction

```
Label           Instruction                 Comments
                .Mmregs
                stm #1009h, ar2             AR2 initialised to (end address -1) of the input array
                stm #1008h, ar3             AR3 initialised to (end address -2)
                                            of the input array
                stm #1057h, ar4             AR4 initialised to end address of
                                            the output array
                stm #0008h, ar5             AR5 initialised to the length of the
                                            output array
Loop            ld #0, a                    A register initialised to 0
                ld #0, b                    B register initialised to 0
                ld *ar2+, 16, A             The two input values which are to be
                                            multiplied by h1
                add *ar3-, 16, A            are left shifted by 16 bits, added
                                            and stored into A
                Rpt # 1
                firs *ar2+, *ar3-, coeff    First time execution: Higher order word of A
                                            multiplied by the content of the location
                                            coefficient and added to B register. The content of
                                            the locations pointed by Ar2 and Ar3 are added
                                            and stored into the higher order word of A. AR2
                                            is incremented by 1. AR3 is decremented by 1
                                            Second time execution: Higher order word of A
                                            multiplied by the content of the location
                                            coefficient and added to B register. The content of
                                            the locations pointed by Ar2 and Ar3 are added
                                            and stored into the higher order word of A. AR2
                                            is incremented by 1. AR3 is decremented by 1
                stl b, *ar4-                One value of output in lower order word of B
                                            stored into the location pointed by AR4.
                                            AR4 decremented by 1 to point to the next lower
                                            element of output array
                mar *+ar2(-4)               AR2 made to point to the last but one element of
                                            the next input window
                mar *+ar3(2)                AR2 made to point to the second element of
                                            the next input window
                banz loop, *ar5-            Output element computation taken up for the next
                                            input window if Ar5 is not zero. AR5 decremented
                                            by 1
```

```
                 Nop
Coeff            .word 3h, 4h                No. of impulse response coefficients (N) is 4
                 .data
Data             .word 0h, 0h, 0h            Input elements appended with (N-l) zeros at the
                 .word 1h, 2h, 3h, 4h, 5h    beginning and at the end are stored here
                 .word 0h, 0h, 0h
Data1            .space 0f5h*16              245 locations reserved
Data2            .word 1, 2, 3, 4, 5         Address of data2 is 1100h
                 .end
```

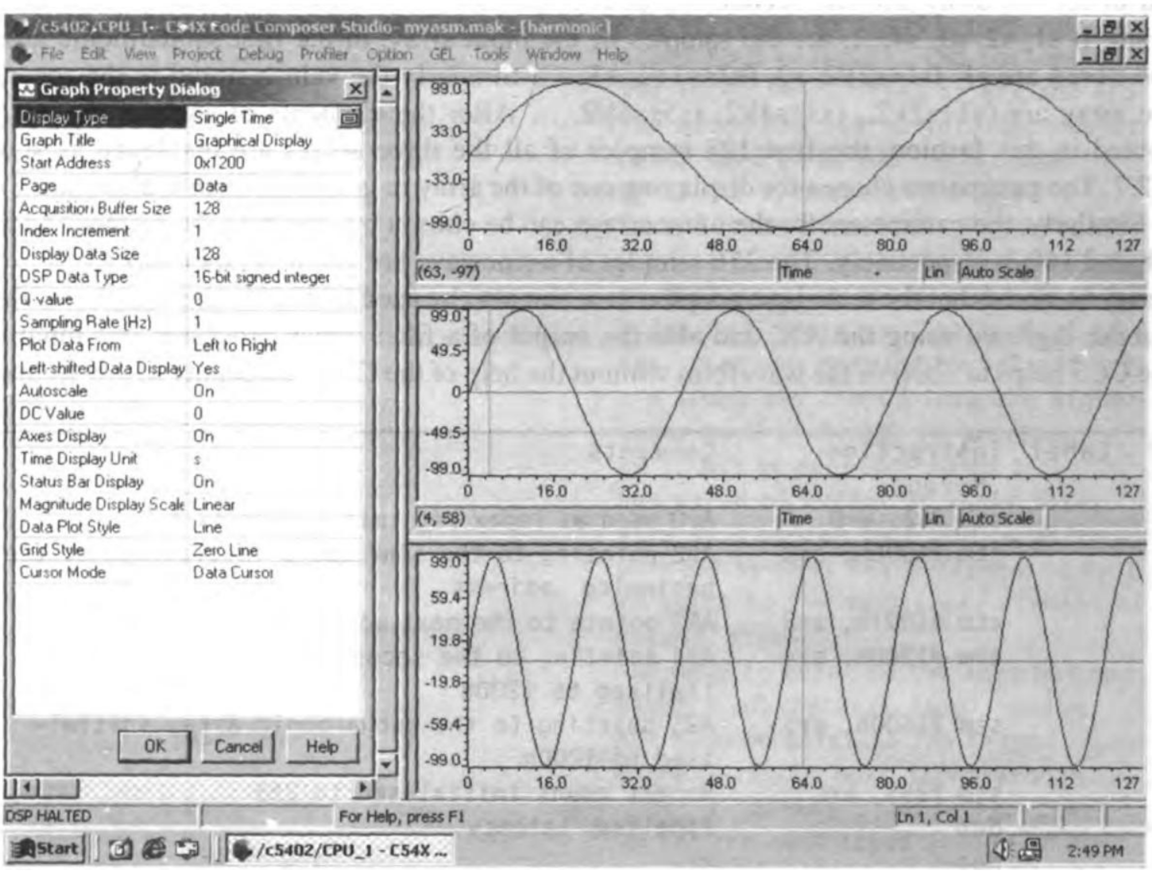
12.5.8 Generation of the Harmonics of a Sinusoidal Signal

Program 12.12: Harmonic.asm gives an example of generation of subharmonic and superharmonic of a sinusoidal signal. This example is taken up to illustrate how the CCS can be used to display information in graphical form. In this program, 256 samples of a sinusoidal signal are stored in one array starting with the address 1000h. The samples corresponding to the first subharmonic, that is, $f_s/2$ is stored in second array and the second harmonic, that is, $2f_s$ is stored in the third array using the program. If s1, s2, s3, s4, ... are the samples of the input array, in the first subharmonic array, the values stored are s1, (s1+s2)/2, s2, (s2+s3)/2, s3, ... Similarly the values stored in the second harmonic array are (sl+s2)/2,, (s3+s4)/2, (s5+s6)/2, ... After the elements of each of the arrays are computed in this fashion, the first 128 samples of all the three arrays are displayed as shown in Fig. 12.7. The parameters chosen for displaying one of the array in graphical form is also shown in Fig. 12.7. Similarly, the parameters for the other arrays can be chosen by changing the memory address as 1000h and 1600h respectively. The 256 samples of a sinewave are given in Appendix 12.

It may be noted that the technique adopted here can also be used to display the samples of the real-time input digitised using the AIC and also the output of a filter which processes this input signal. Hence CCS helps to observe the waveform without the help of the CRO, especially at low frequencies.

Program 12.12 Harmonic, asm

```
Label            Instruction                 Comments
                 .Mmregs
                 stm #02, ar0                Ar0 used as index reg initialised to 2
                 stm #1000h, ar2             AR2 pointing to the sine table initialized to the
                                             beginning address
                 stm #1001h, ar3             AR3 points to the next address in the sine table
                 stm #1200h, ar4             AR4 pointing to the second harmonic array
                                             initialised to 1200h
                 stm #1600h, ar5             AR5 pointing to the subharmonic array
                                             initialised to 1200h
                 stm #255, brc               Repeat count initialised to 255
                 Nop                         Pipeline latency
                 Nop
                 rptb final                  Instructions till final repeated
                 ld *ar2+0, a                Content of location pointed by ar2 copied to A,
                                             AR2 incremented by 2
                 ld *ar3, b                 Content of location pointed by AR3 copied to B
                 st1 a, *ar5+               First value stored in subharmonic array
                 add b, a                   Sum of the first two elements found
                 stl -1, *ar5+              Sum divided by 2 and stored in both the
                 stl -1, *ar4+              Subharmonic and second harmonic array
                 stl *ar5+                  Second element of sine array stored as third
                                            element of subharmonic array
                 ld *ar2, b                 Third element of sine array copied to b
                 add *ar3+0, b              Sum of second and third element found
                 st1 b, -1, *ar5+           Sum divided by 2 and stored as fourth element of
                                            subharmonic array
Final            Nop
                 Nop
                 .data
                 .include "sine.asm"        Contains 256 samples of sinewave
                 .end
```



Display Type	Single Time
Graph Title	Graphical Display
Start Address	0x1200
Page	Data
Acquisition Buffer Size	128
Index Increment	1
Display Data Size	128
DSP Data Type	16-bit signed integer
Q-value	0
Sampling Rate (Hz)	1
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s
Status Bar Display	On
Magnitude Display Scale	Linear
Data Plot Style	Line
Grid Style	Zero Line
Cursor Mode	Data Cursor

Fig. 12.7 *CCS display with graphical outputs*

12.5.9 Programs for Processing Real Time Inputs

One of the ways for feeding the real-time input to kit is through the AIC TLC320AD50C and the multichannel buffered serial port (BSP) also referred to as McBSP. Some of the features of the AIC AD50 are listed in Sec. 12.3.

The BSP has a full-duplex, double-buffered serial port interface and an autobuffering unit (ABU). BSP can be programmed either as a standard ′54X serial port or as a standard serial port with enhanced features. The operation of the ′54X standard serial port is identical to that of the ′5X serial port discussed in Chapter 6. The block diagram of the standard ′54X serial port is the same as that given in Fig. 6.6. The BSP has its own dedicated memory-mapped 16-bit data transmit, data receive and serial port control registers (BDXR, BDRR and BSPC) and the significance of the various bits of these registers is the same as that of the (DXR, DRR and SPC) registers of the standard serial port given in Figs 6.7-6.12. The BSP has an additional control register, the BSP control extension register (BSPCE), for implementing its enhanced features and controlling the ABU. The BSP has four interrupt pins WXINT, WRINT, BXINT and BRINT. The operation of BXINT and BRINT is identical to that of XINT and RINT of the standard serial port in the standard mode. WXINT and WRINT interrupts are generated each time a word is transmitted or received. They in turn generate the interrupt BXINT and BRINT depending upon the operating mode of ABU.

The enhanced features that the BSP offers include the capability to generate programmable rate serial port clocks, select positive or negative polarities for clock and frame sync signals and to perform transfers of 10- and 12-bit words, in addition to the 8- and 16-bit transfers offered by the serial port. Additionally, BSP implements the capability to specify that frame sync signals be ignored until instructed otherwise, and provides a dedicated operating mode which facilitates its use with PCM interfaces. The BSPCE contains the control and status bits that are used in the implementation of these enhanced BSP features and the ABU.

The ABU of BSP allows the serial port section to read/write directly to ′54X internal memory independent of the CPU. This results in a minimum overhead for serial port transfers and faster data rates. The ABU has a set of circular addressing registers, each with corresponding address generation units. Memory for transmit and receive buffers resides within a special 2K word block of ′54X internal memory. This memory can also be used by the CPU as general purpose storage, however, this is the only memory block in which autobuffering can occur. Using autobuffering, word transfers occur directly between the serial port section and the ′54X internal memory automatically using the ABU

Program 12.13 Capture_Display.C

```
#include <type.h>
#include <board.h>
#include <codec.h>
#inc1ude <mcbsp54.h>
HANDLE hHandset;
void main()        S16 data
{
  if (brd_init(100))
  return;                          /* Initialise the board for 100MHz */
  hHandset = codecopen(HANDSETCODEC);
                                   /* Open Handset Codec */
  codecdacmode(hHandset, C0DEC_DAC_15BIT);
                              /* Set DAC in 15-bit mode */
  codec_adc_mode(hHandset, C0DEC_ADC_15BIT);
                              /* Set ADC in 15-bit mode */
  codecai ngai n(hHandset, C0DEC_AIN_6dB);
                              /* 6dB gain on analog input to ADC */
  codec_aout_gain(hHandset, C0DEC_A0UT_MINUS_6dB);
                              /* -6dB gain on analog output from DAC */
  codec_sample_rate(hHandset,SR_16000);
                              /*16KHz sampling rate */
while (1)
{
while (!MCBSP_RRDY(HANDSET_CODEC)) {};
                              /* Wait for sample from handset */
  (data = *(volatile u16*)DRR1_ADDR(HANDSET_C0DEC));
  *(volatile u16*)DXR1_ADDR(HANDSET_C0DEC) = data;
                              /*Read sample from & write back to handset
  codec*/
  }
}
```

-embedded address generators. The length and starting addresses of the buffers within the 2K block are programmable, and a buffer empty/full interrupt can be generated to the CPU. Buffering can also be halted using the autodisabling capability. The BSP autobuffering capability can be separately enabled for the transmit and receive sections. When autobuffering is disabled (standard mode), data transfers with the serial port section occur under software control in the same fashion as with the standard ′54X serial port. In this mode, the ABU is transparent, and the WXINT and WRINT interrupts are sent to the CPU as transmit interrupt (BXINT) and receive interrupt (BRINT). When autobuffering is enabled, the BXINT and BRINT interrupts are only generated to the CPU each time half of the buffer is transferred. Initialisation of the devices in the 5402 DSK is greatly simplified by the CCS. The 5402 DSK is equipped with a no. of header files such as McBSP54.h, board.h and codec.h which contain a no. of Macro function routines. These functions can be called by passing the appropriate parameter. The initialisation routine may be written in C, compiled and verified for the proper operation. For example, Program 12.13: Capture_Display.C gives the program for capture and display of an audio signal from an oscillator and display the same in CRO. To compile and download the code to the DSK, first a project such as AD50.mak is opened and the file Capture_Display.C is added to this project. Rest of the procedure is the same as that used for a asm file used in the previous examples. After verification of the expected result in the DSK, the initialisation routine may be used as a part of an assembly language program. This can be achieved by converting the C program to the equivalent asm file. For example, to convert the above C file to the asm file, the command to be used, under command prompt, is

```
c1500 Capture_Display.C -ss
```

This generates the asm file Capture_Display.asm. (The path has to be properly chosen so that the header files are also accessible) This asm file can be edited as per the user requirement. For example, this routine may be modified to store the data in a circular buffer (either the ABU or some other buffer). The data in the buffer may be processed and the result may be stored in another circular buffer. The content of this buffer may be displayed on real-time basis. The data may obtained from the AIC using either

DMA or interrupt so that some other operation such as modulation/demodulation of FSK signal may also be carried out on real time basis.

APPENDIX 12

The 256 samples of a sinusoidal signal are given in this appendix.

```
Label   Mnemonic
        .word 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046, 0x0050
        .word 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x0046, 0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1
        .word 0xffe2, 0xffd3, 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e
        .word 0xff9d, 0xff9e, 0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3
        .word 0xffe2, 0xfff1, 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046
        .word 0x0050, 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x0046, 0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1
        .word 0xffe2, 0xffd3, 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e
        .word 0xff9d, 0xff9e, 0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3
        .word 0xffe2, 0xfff1, 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046
        .word 0x0050, 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x0046, 0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1
        .word 0xffe2, 0xffd3, 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e
        .word 0xff9d, 0xff9e, 0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3
        .word 0xffe2, 0xfff1, 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046
        .word 0x0050, 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x0046, 0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1
        .word 0xffe2, 0xffd3, 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e
        .word 0xff9d, 0xff9e, 0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3
        .word 0xffe2, 0xfff1, 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046
        .word 0x0050, 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x0046, 0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1
        .word 0xffe2, 0xffd3, 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e
        .word 0xff9d, 0xff9e, 0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3
        .word 0xffe2, 0xfff1, 0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046
        .word 0x0050, 0x0059, 0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059
        .word 0x0050, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1, 0xffe2, 0xffd3
        .word 0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e, 0xff9d, 0xff9e
        .word 0xffa1, 0xffa7, 0xffb0, 0xffd3, 0xffe1, 0xfff1, 0x0000, 0x000f
        .word 0x001e, 0x002d, 0x003a, 0x0046, 0x0050, 0x0059, 0x005f, 0x0062
        .word 0x0063, 0x0062, 0x005f, 0x0059, 0x0050, 0x0046
```

Review Questions

12.1 How does the conditional call instructions of C54X differ from the unconditional call instructions w.r.t. the no. of cycles required for execution?

12.2 What should be the value of IDATA in the hello. cmd if the data memory is required to have the starting address as 1500h?

12.3 The data memory is initialised to have the address 1000h. Explain how four values 43h, 44h, 45h and 46h can be stored in the memory space 1050-1053h using (a) .bes directive (b) .space directive (c) bkpd instruction.

12.4 Explain the sequence of steps required for running a C54X assembly program in code composer studio.

12.5 Explain how a C54X program may be run using break points in CCS environment?

12.6 What is the modification required in Program 12.1 if it is used to add three vectors each of size 15?

12.7 What is the modification required in Program 12.1 if the RPTB instruction is to be used instead of the banz instruction. Of the two approaches which one would consume less clock cycles for execution?

12.8 Write a C54X assembly language program to find the sum of the series 1 + 2 + 3 + ... 1000.

12.9 Write a C54X assembly language program to find the sum of the series $1^2 + 2^2 + 3^2 + \ldots 100^2$

12.10 Write a C54X program to read a sequence of 256 samples of a sine wave signal from a file and store them in a circular buffer of size 80. Every time the buffer reaches the end address, the content of the buffer should be displayed on the screen.

12.11 Write a C54X program to generate the FSK signal using the stored sample values of two sinusoidal signals.

12.12 Write a C54X program to decode the FSK signal.

12.13 Write a program to initialise the McBSP of 54X and test its operation using the loop back mode.

12.14 Write a program to initialise the A1C AD50 and digitise the sinusoidal signal fed to the AIC input and store 256 samples in the address space 1000-10FFh.

12.15 Write a program to initialise the AIC AD50 and digitise the sinusoidal signal fed to the AIC input, divide it by two and display the same in the CRO by converting the signal back to analog signal.

12.16 Write a program to digitise the signal fed to AIC and display the same in the CRO by converting the signal back to analog signal.

12.17 Write a program which processes the input data from the AIC on real-time basis and display the LP filtered output in the CRO. Assume the LP filter to have 81 taps and to be stored in the program memory area.

12.18 Write a program which processes the input data from the AIC on real-time basis and display the LP filtered output in the CCS output screen. Assume the LP filter to have 81 taps and to be stored in the program-memory area.

12.19 The impulse response coefficients of a 5-tap filter are initially [1.5, 1.3, 1.0, 0.5, 0.3]. An adaptive filter is to be designed which adapts the filter coefficients till the coefficients become [1, 0, 0, 0, 0]. Write a C54X program using the LMS instruction to implement this filter. The input samples may be assumed to be fixed and of length 7.

12.20 Explain with an example how convolutional codes can be generated using a C54X program.

12.21 With a simple example illustrate how the convolutional codes may be decoded using the Viterbi algorithm. Write a C54X program for the Viterbi decoder.

12.22 Write a C54X program to find the Eucludian distance between two vectors of dimension 10.

12.23 Write a C54X program to find the Hamming distance between two vectors of dimension 10.

12.24 Write a C54X program to find the vector which has the minimum Hamming distance to a given vector of dimension 10 among five vectors.

12.25 Write a program in C54X to multiply a 3X3 matrix by a vector.

Self Test Questions

12.1 Which of the following call instructions require five clock cycles for execution?
(a) CCsub (b) CALL sub (c) CCD sub (d) CALLDsub

12.2 Which of the following call instructions executes either a single 2-word instruction or two 1-word instructions before executing the subroutine?
(a) CCsub (b) CALL sub (c) CCD sub (d) CALLDsub

12.3 Which of the following branch instructions require four clock cycles for execution?
(a) Bloop (b) BDloop (c) BCD loop (d) BCloop

12.4 Which of the following branch instructions executes either a single 2-word instruction or two 1-word instructions before branching to loop?
(a) Bloop (b) BDloop (c) BCD loop (d) BCloop

12.5 Which of the following instructions require two clock cycles for execution?
(a) Any two memory operand instructions in which the two operands reside in the same block of S ARAM
(b) Any instruction that reads a 320-bit operand from DARAM
(c) Any instruction that reads a 32-bit operand from SARAM
(d) Any instruction that writes a 32-bit operand to SARAM

12.6 The rate at which the data can be exchanged between the host and the target DSP board using the RTDX feature of CCS is.
(a) 20 Kbytes/s (b) 20 Kbits/s
(c) 64 Kbytes/s (d) 64 Kbits/s

12.7 The external SRAM capacity in C5402 kit is words and requires ——— wait states.
(a) 64K, 1 (b) 64K, 7 (c) 256K, 1 (d) 256, 7

12.8 The external Flash memory capacity in 5402 kit is ——— Mwords and requires ——— wait states.
(a) 64K, 1 (b) 64K, 7 (c) 256K, 1 (d) 256, 7

12.9 In the C5402 kit there are ——— AD50 AICs and C5402 has ——— enhanced buffered multichannel serial ports on chip.
(a) 1, 1 (b) 1,2 (c) 2, 1 (d) 2,2

12.10 ——— C54X assembler directive reserves and initialises *n* bits of memory and when a label is used with this directive, the label is assigned the address of the first word of the block reserved.
(a) .bes (b) .space (c) .word (d) .data

12.11 ——— C54X assembler directive reserves and initialises *n* bits of memory and when a label is ——— used with this directive, the label is assigned the address of the last word of the block reserved.
(a) .bes (b) .space (c) .word (d) .data

12.12 ——— C54X assembler directive which enables the code to be assembled into data memory area.
(a) .bes (b) .space (c) .word (d) .data

12.13 For displaying the waveform corresponding to 256 samples, the acquisition buffer size and display data size in the view window of CCS should be chosen as ———, ———.
(a) 128, 256 (b) 128, 128 (c) 256, 256 (d) 256, 128

12.14 If any one of the following pairs of instructions are executed sequentially, unexpected results may occur. One way to avoid this is to separate the pairs of instructions by a no. of nops or instructions do not use the registers which are modified in the first instruction of the pair. How many nops are required between each of the pairs of instructions to ensure that expected results occur?
(a) STM #1000h, AR0
LD *AR0, A
(b) STL A, *AR7+
STM #1000h, AR3
(c) STLM A, AR0
LD *AR0-, A
(d) STM #08h, BK
LD *AR2+%, A
(e) STM #10h, T
MUL 20h, A
(f) STL A,T
MUL 20h, A
(g) STM #10h, BRC
RPTB start
(h) STLM A, BRC
RPTB start
(i) RSBX CPL
LD 20h, A
(j) RSBX CPL
STL A, 20h
(k) SSBX CPL
LD 20h, A
(l) SSBX CPL
STL B, 20h

13 ARCHITECTURE OF TMS320C6X

INTRODUCTION 13.1

The TMS320C6X DSPs use the VelociTI^{TM} architecture, the first DSPs to use advanced VLIW (Very Large Instruction Word) architecture to achieve high performance through increased instruction parallelism. This makes the ′C6X DSPs an excellent choice for multichannel and multifunction applications.

The conventional VLIW architecture consists of multiple execution units running in parallel performing multiple instructions during a single clock cycle. The VelociTI architecture is a highly deterministic architecture having reduced code size, flexibility of code and data type and zero overhead in branching.

The TMS320C62X, TMS320C64X and TMS320C67X are the family of DSPs in the ′C6X generation. The ′C62X and ′C64X devices are fixed point and ′C67X devices are floating point DSPs. In ′C6X DSPs ′C62X and ′C64X processors are code compatible, ′C62X and ′C67X processors are code compatible.

The ′C6X devices execute up to eight 32-bit instructions per cycle with an execution speed of up to 6000 million instructions per second (MIPS). The ′C6X CPU consists of eight functional units, two multiplier and six ALUs and some general purpose registers. The CPU of ′C62X and ′C67X device consists of 32 general purpose registers of 32-bit size, where as ′C64X devices have 64 general purpose registers of 32-bit size.

FEATURES OF ′C6X PROCESSORS 13.2

- Advanced VLIW CPU with eight functional units, including two multipliers and six ALUs
- Executes up to eight instructions per cycle allows to develop effective RISC like code
- Instruction packing reduces code size, program fetches and power consumption
- Conditional execution of all instructions
- Efficient code execution on independent functional units
- Supports 8/16/32- bit data formats
- 40-bit arithmetic operations, saturation and normalization operations
- Field manipulation and instruction extract, set, clear and bit counting operations
- The ′C67X device has hardware support for single precision (32-bit) and double precision (64-bit) IEEE floating point operations and also 32 X 32 bit integer multiplication with 32 or 64 – bit results.

- The ′C64X device multiplier can perform two 16 X16 bit or four 8 X 8 bit multiplications per cycle, quad 8-bit and dual 16-bit instruction set extensions with data flow support, memory access for non-aligned 32-bit and 64-bit, special communication-specific instruction useful in realizing error-correcting codes, bit count and rotate hardware.

INTERNAL ARCHITECTURE 13.3

The block diagram of TMS320C6X devices is given in Fig. 13.1. The ′C6X devices contains 32-bit CPU, on-chip program, data memory and on-chip peripherals. The on-chip memory has cache either for program space or for both program and data space. The ′C6X devices have peripherals such as external memory interface (EMIF), direct memory access controller (DMA), timers, multi-channel buffered serial ports (McBSP), host port interface (HPI) and power down logic.

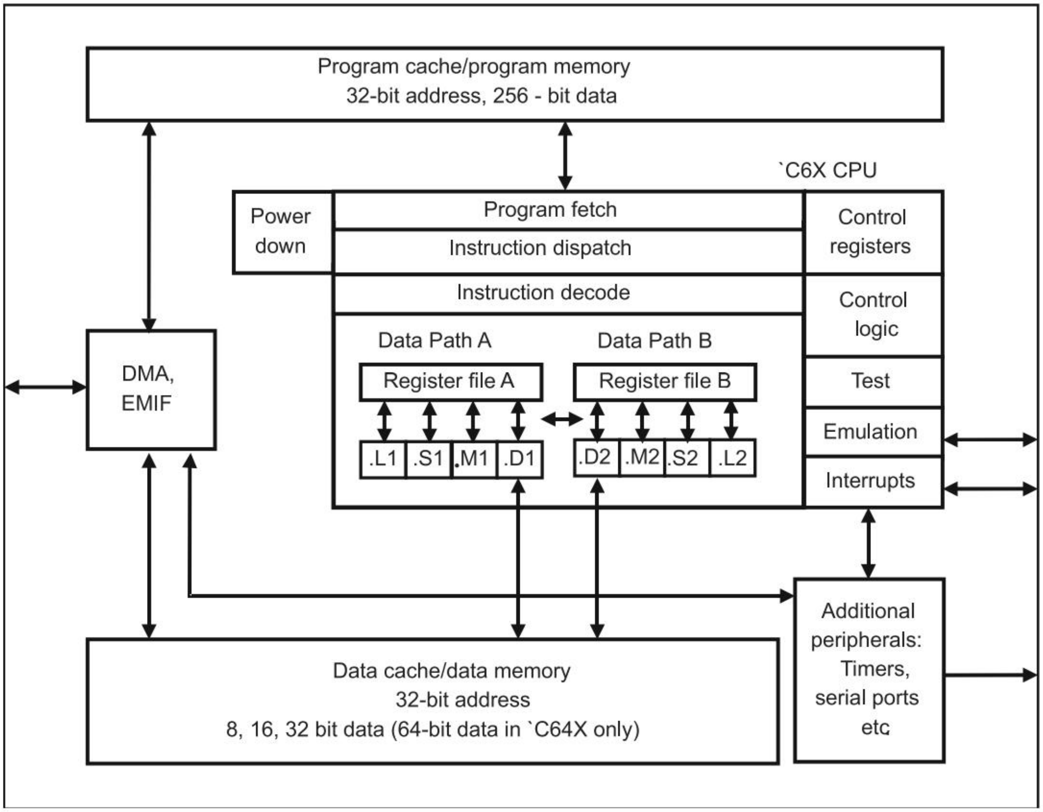


Fig. 13.1 *Internal Architecture of TMS320C62X/′C64X/′C67X Devices*

CPU 13.4

The central processing unit of ′C6X device is 32-bit size. The block diagram of ′C6X CPU is given in Fig. 13.2. The CPU contains the following units:

(a) Program fetch unit

(b) Instruction dispatch unit
(c) Instruction decode unit
(d) Two data paths, each data path consists of four functional units
(e) Register file for each data path
(f) Control registers
(g) Control logic
(h) Test, emulation and interrupt logic

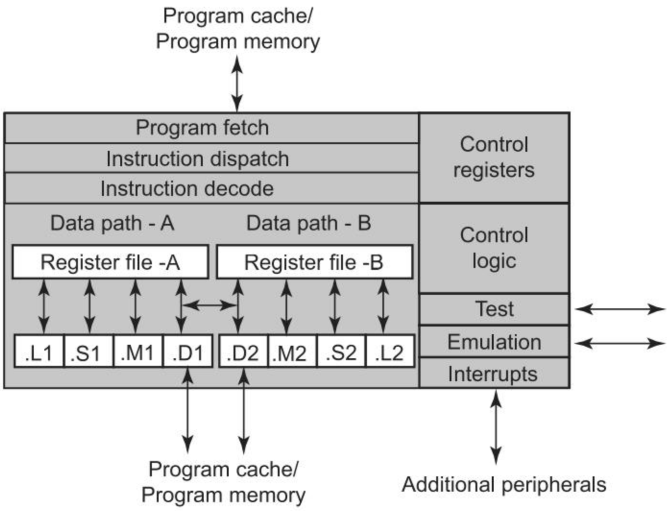


Fig. 13.2 *CPU Unit of TMS320C6X DSP*

The functional units shaded in Fig. 13.2. are common to all ′C6X devices. The ′C6X CPU is based on advanced VLIW architecture, which accepts eight 32-bit instructions (the instruction fetch packet size is 256 bits) at a time. The program fetch unit generates the addresses of eight instructions and sends it to the program memory for each fetch packet. Once the contents of the program memory read occurs, the fetch packet is received at the CPU.

The instruction dispatch unit receives the fetch packet and splits it into execute packets. The instructions in the execute packet (eight instructions) are assigned to the appropriate eight functional units in the data path. During the instruction decode, the source registers, destination registers and associated paths are decoded for the execution of the instructions in the functional units. Finally the instructions are executed by the functional units.

The register file (A&B) of all the ′C6X devices contain 32 numbers of 32-bit registers (16 register for each data path) except ′C64X devices. The ′C64X device register file has 64 numbers of 32-bit registers with 32 registers for each data path.

The ′C6X CPU contains eight functional units, six arithmetic and logic units and two multipliers (.L1, .L2, .S1, .S2, .M1, .M2, .D1 and .D2.). These functional units can be divided into two groups of four. The L, S & D units are arithmetic and logic units (ALU), and the M unit is a multiplier unit. Each data path has almost identical functional units.

GENERAL-PURPOSE REGISTER FILES 13.5

There are two general-purpose register files A and B in ′C6X CPU data paths. In ′C62X/′C67X devices each register file contains 16 numbers of 32-bit registers, the registers A0-A15 for register file-A and B0-B15 for register file-B. The ′C64X devices have double the number of general-purpose registers as that are in ′C62X/′C67X processors. There are 32 numbers of 32-bit registers for each data path, where A0-A31 for register file-A and B0-B31 for register file-B. The general-purpose registers can be used for handling data; data address pointers or condition registers.

The ′C62X/′C67X general-purpose register files supports packed 16-bit, 40-bit fixed point data and 64-bit floating point data types. The packed data type can store four 8-bit values or two 16-bit values in a single 32-bit register or four 16-bit values in 64-bit register. The values larger than 32 bits, such as 40-bit fixed point and 64-bit floating point are stored in register pairs. The storage scheme for 40-bit long data in register pair is shown in Fig. 13.3. In register pairs (A3:A2), 32 LSBs of data are placed in an even numbered register (A2) and the remaining 8-bit or 32 MSBs in the next upper register (A3), which is always odd numbered register. The ′C64X register file has this facility by supporting packed 8-bit and 64-bit floating point data types. For 40-bit and 64-bit data, there are 16 valid register pairs in ′C62X/′C67X and 32 valid register pairs in ′C64X core. The valid register pairs in ′C6X devices are given in Table 13.1.

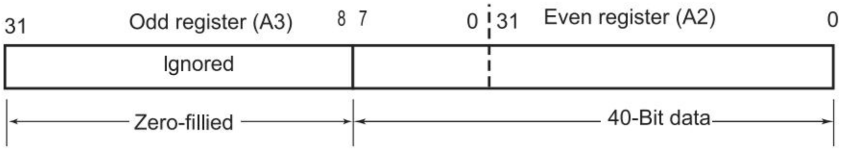


Fig. 13.3 *Storage Scheme for 40-Bit Data in a Register Pair*

Table 13.1 *Valid Register Pairs in ′C6X CPU Register files*

Device Family	*Register Pairs*	
	Data path – A	*Data path – B*
′C62X/′C64X/′C67X	A1 : A0	B1 : B0
	A3 : A2	B3 : B2
	A5 : A4	B5 : B4
	A7 : A6	B7 : B6
	A9 : A8	B9 : B8
	A11 : A10	B11 : B10
	A13 : A12	B13 : B12
	A15 : A14	B15 : B14
′C64X only	A17 : A16	B17 : B16
	A19 : A18	B19 : B18
	A21 : A20	B21 : B20
	A23 : A22	B23 : B22
	A25 : A24	B25 : B24
	A27 : A26	B27 : B26
	A29 : A28	B29 : B28
	A31 : A30	B31 : B30

FUNCTIONAL UNITS AND OPERATION 13.6

The ′C6X CPU consists of eight functional units, .L1, .S1, .M1, .D1, .L2, .S2, .M2 and .D2. These eight functional units of ′C6X devices are divided into two groups, one group for each data path. Each functional unit in one data path is almost identical to the corresponding unit in the other data path and arranged as mirror image to each functional unit. The .L, .S and .D units are arithmetic and logic unit, .M unit is a multiplier unit. The fixed point operations performed in ′C6X processor functional units and the bit size of the operation are given in Table 13.2.

The .L unit performs arithmetic and logical operations, other operations like compare and count are performed in this unit. The .S unit is used for arithmetic and logical operations as well as for branch, shift, constant generation and move operations. The .D unit does add and subtract operations. The .D unit is a dedicated unit for the load, store operations, linear and circular address calculations. The .M unit is dedicated unit to perform multiply operations. The functional units in ′C64X processor supports Dual 16-bit and Quad 8-bit functional operations pertaining to their units apart from the normal operations.

The operations performed by ′C67X processor functional units are given in Table 13.3. The .L unit performs arithmetic operations and .S unit does the compare operations. The .M unit can do 32 x32 bit fixed-point multiply operations and floating point operations. The .D unit is used to load and store double words with 5-bit constant offset.

Table 13.2 *Functional units of ′C6X and its fixed point operations*

Name of the Unit *Type of operation*	*.L unit*	*.S unit*	*.M unit*	*.D unit*
Arithmetic operation	32/40 bit operation ***Dual 16 bit, Quad 8 bit arithmetic and min/max operations****	32-bit operation ***Dual 16 bit, Quad 8 bit saturated arithmetic operations****	—	32-bit add & subtract operations only
Logical operation	32-bit operations	32-bit operations	—	***32-bit logical operations****
Multiply operations	—	—	16x16 multiply operations***16x32, Quad 8x8, Dual 16x16 multiply operations***	—
Shift operations	***Byte shifts****	32/40 bit shift operations ***Byte shifts, Dual 16 bit shift operation****	***Variable shift operations****	—
Compare operations	32/40 bit operations	***Dual 16 bit, Quad 8 bit compare operations ****	—	—
Branch operations	—	Yes	—	—
Load and Store operations	—	—	—	Load and stores with 5-bit constant offset(15-bit constant offset in .D2 only)
Linear and circular address calculation	—	—	—	Yes
Constant generation	***5 bit constant generation****	Yes	—	***5 bit constant generation****
Count operations	32/40 bit count operations	—	—	—
Move operations	Register to register only ***16-bit move operations****	16-bit move operations	— ***Register to register only****	Register to register only***16-bit move operations****

**** - additional operations performed by the functional units in ′C64X processors.***

Table 13.3 *Functional units of ′C6X and its floating point operations*

Name of the unit	*Type of floating point operation*
.L unit	Arithmetic operations
.S unit	Compare, square-root and Absolute value operations
.M unit	32x32 bit Fixed point multiply operations and Floating point multiply operations
.D unit	Load double word with 5-bit constant offset

DATA PATHS 13.7

The ′C6X CPU has two data paths, Data path – A and Data path – B. The data paths of ′C62X, ′C67X and ′C64X devices are shown in Fig. 13.4., Fig. 13.5., and Fig. 13.6. respectively.

13.7.1 Register File Data Paths

Most of the data lines in the CPU data path are 32-bit wide but some support 40-bit (long operands) and 64-bit (double word operands) lines. The functional units ending in 1 (.L1, .S1, .M1 and .D1) have access to register file A, and functional units ending in 2 (.L2, .S2, .M2 and .D2) to register file B.

Each functional unit has two 32-bit ports for reading source operands **src1** and **src2** from the respective register files. The .L and .S units have an extra 8-bit line for 40-bit **long src** operand reads.

Each functional unit has its own 32-bit write port into the respective register file for destination **dst** operands except .M unit of ′C64X. The ′C64X multiplier unit can return up to a 64-bit result, so an extra 32-bit write port is available to the register file. The same way us the read port, .L and .S units have an extra 8-bit line for 40-bit **long dst** operand writes. Since each unit has its own port for operand read and writes, when performing 32-bit operations all the eight functional units can be used in parallel every machine cycle.

13.7.2 Register File Cross Paths

The ′C6X processors functional units can read and write the operands directly from their respective register files using its own data paths. The register files are connected to the opposite side functional units through 1X and 2X cross paths. These cross paths allow the functional units from one data path to access 32-bit operand from the opposite side register file. The functional units of data path –A read their source operands from register file B via 1X cross path and the 2X cross path allows the functional units of data path –B to read the source operand from register file A.

The six functional units (.L1, .L2, .S1, .S2, .M1 and .M2) out of the eight units of ′C62X and ′C67X processors, have access to the opposite side register file via cross path. In .S1, .S2, .M1 and .M2 units **src2** operand is selectable between the cross path and the same side register file path but in the case of .L1 and .L2 units, both **src1** and **src2** operands are selectable.

In ′C64X processor, all the eight functional units have access to the register file of the opposite side through cross path. In ′C64X also .L1 and .L2 units both **src1** and **src2** operands are selectable between the cross path and the same side register file path but in the case of other six functional units only **src2** operand is selectable.

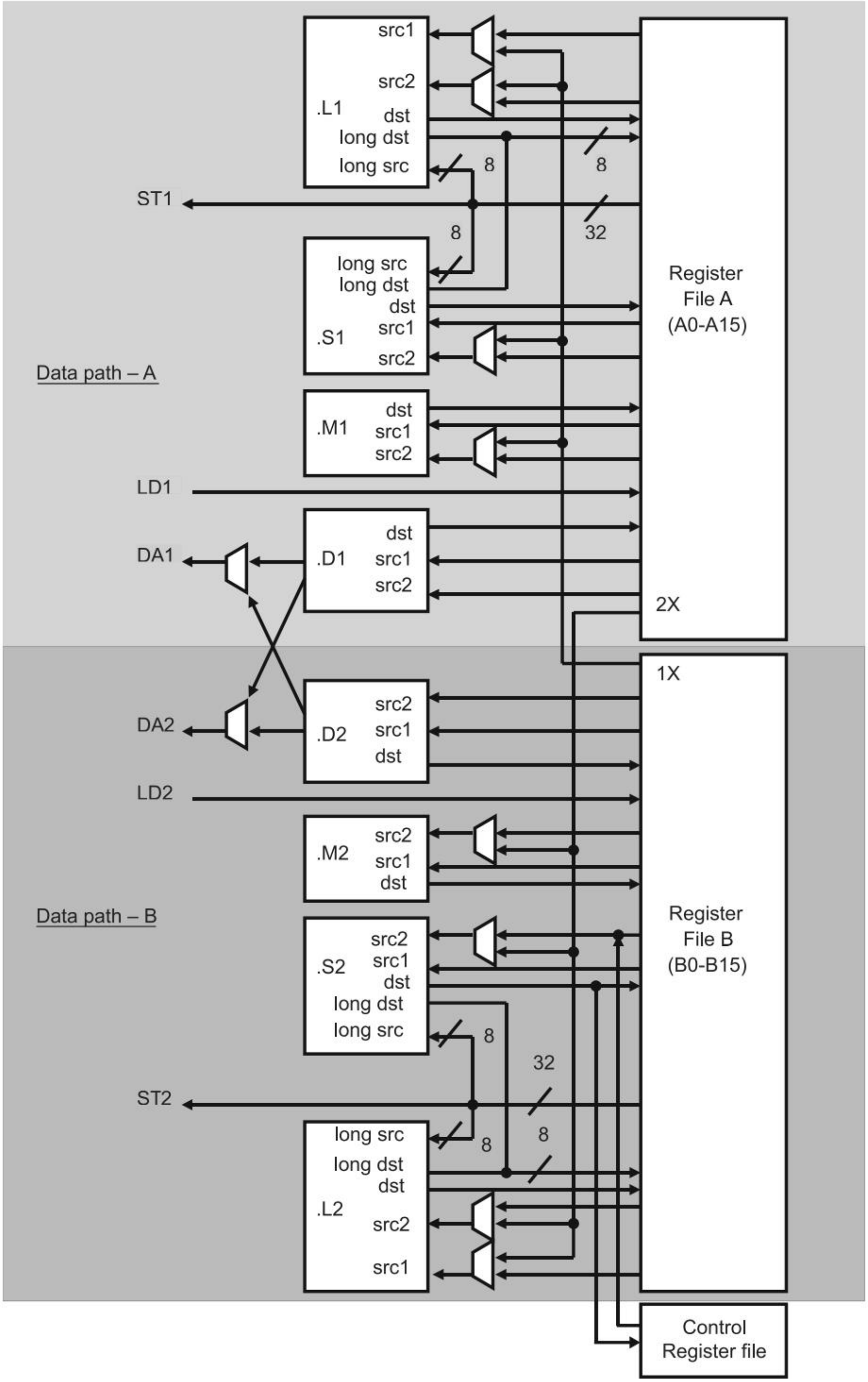


Fig. 13.4 *TMS32C62X CPU Data Paths*

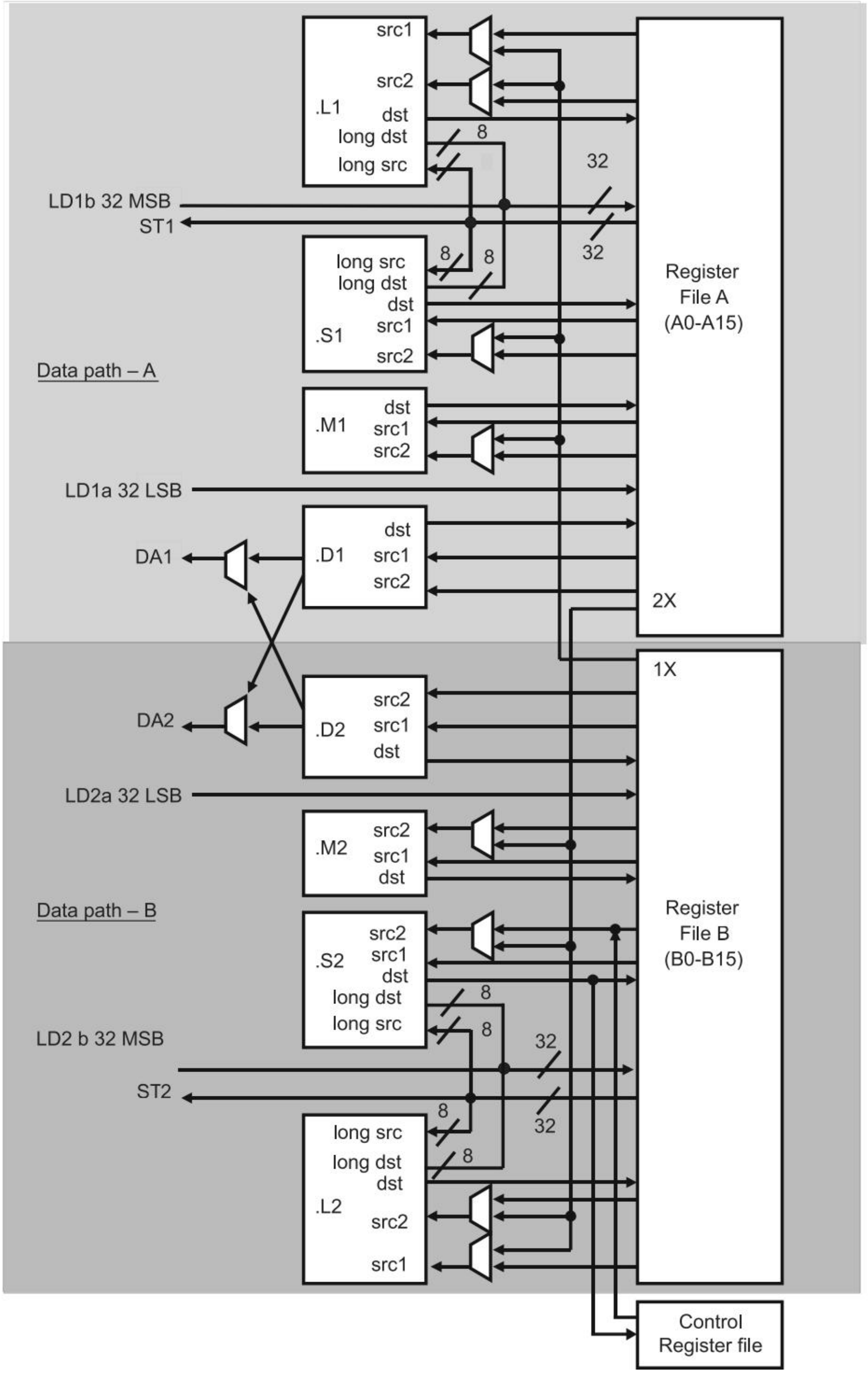


Fig. 13.5 *TMS32C67X CPU Data Paths*

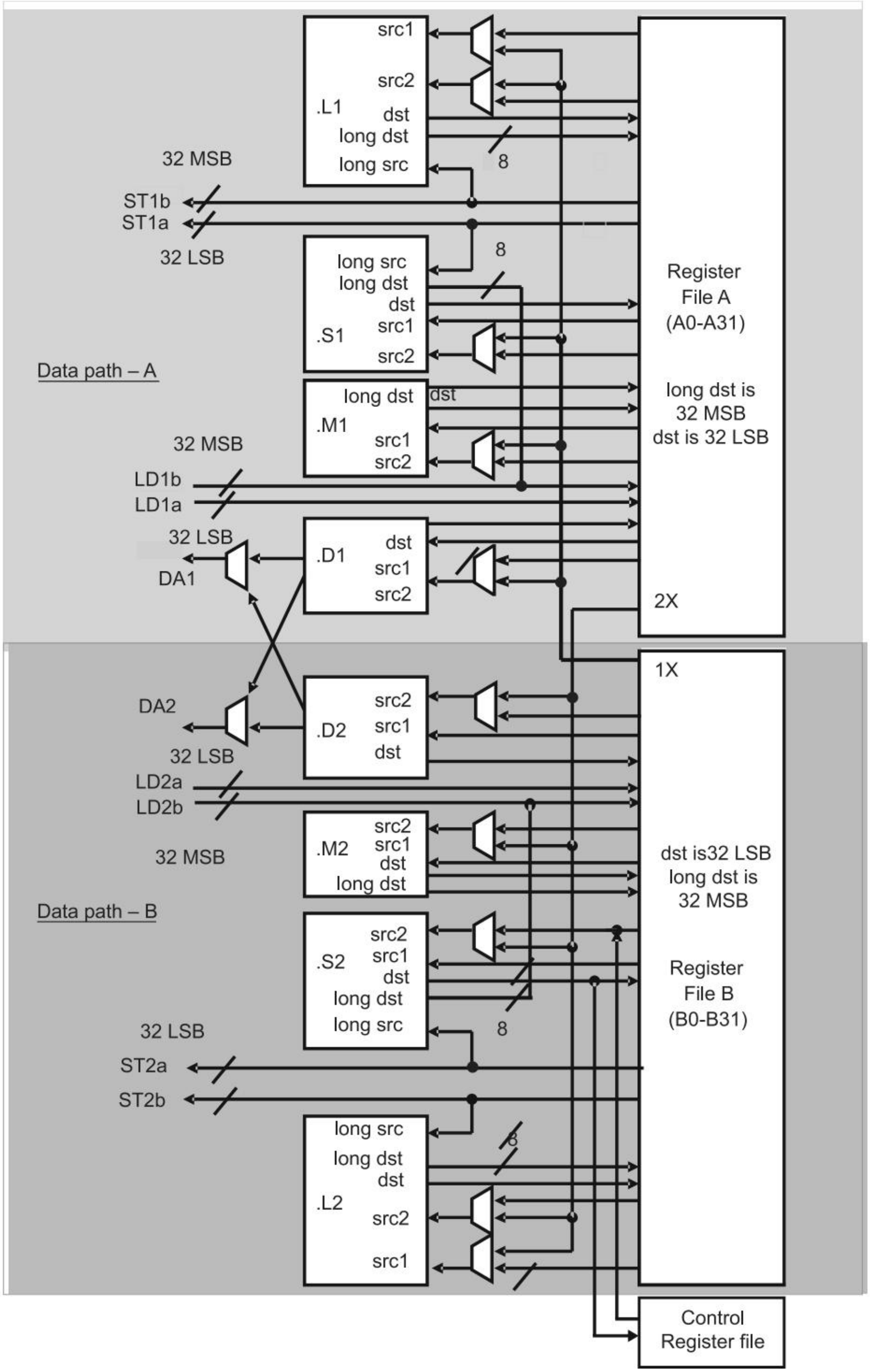


Fig. 13.6 *TMS32C64X CPU Data Paths*

13.7.3 Register File Memory Access Paths

In order to access data from memory to CPU register files, ′C6X CPU has address paths, data load and store paths. The DA1 and DA2 the address paths, LD1 and LD2 the data load paths and ST1 and ST2 the data store paths are used for memory access.

The DA1 and DA2 address paths are 32-bit size and are connected to .D unit of the respective data paths. The paths allow addresses generated by any one path to access data to or from any register. The DA1 and DA2 resources and their associated data paths are specified as T1 and T2 respectively in the instruction set. It is important to note that there is cross path for the address buses, the address generated in .D1 and .D2 units can have access to DA2 and DA1 paths (opposite paths) respectively.

The ′C62X processor has two 32-bit paths for loading data from memory to register file, LD1 for register file A and LD2 for register file B, but both ′C64X and ′C67X processors have additional 32-bit load paths (LD1a and LD1b, LD2a and LD2b) for register files A and B. This allows CPU to load simultaneously two 32-bit (64-bit) values in register files A and B.

As for as the store path is concerned, both ′C62X and ′C67X processors have two 32-bit paths to store data values from register file to memory. The ′C64X has additional 32-bit store paths ST1a and ST1b and ST2a and ST2b for register files A and B. The′C64X processor alone supports double word load and store instructions. The size of memory access paths in C6X processors are given in Table 13.4.

Table 13.4 *Size of memory access paths in ′C6X processors*

Data path type	*′C62X*		*′C64X*		*′C67X*	
	Size	*Number*	*Size*	*Number*	*Size*	*Number*
Address path	32-bit	2 (DA1 and DA2)	32-bit	2 (DA1 and DA2)	32-bit	2 (DA1 and DA2)
Load path	32-bit	2 (LD1 and LD2)	64-bit	2 (LD1a, LD1b and LD2a, LD2b)	64-bit	2 (LD1a, LD1b and LD2a, LD2b)
Store path	32-bit	2(ST1 and ST2)	64-bit	2 (ST1a, ST1b and ST2a, ST2b)	32-bit	2 (ST1 and ST2)

CONTROL REGISTER FILE 13.8

The control register file of ′C6X processor contains ten control registers common to ′C62X, ′C64X and ′C67X. The .S2 unit alone can read and write to control register file. The control registers are generally accessed by the **MVC** (Move between the Control file and Register file) instruction but some of the control register bits are specially accessed in other ways. For example, the global interrupt enable bit, maskable interrupt bits and interrupt flag bits are accessed in different way. The list of control registers common to ′C6X processors and their description is given in Table 13.5.

13.8.1 Addressing Mode Register (AMR)

The eight registers A4-A7 and B4-B7 of the CPU register file can be used for linear and circular addressing. The Addressing Mode Register (AMR) specifies the addressing mode; it consists of mode select fields and block select fields. The various fields of the AMR are shown in Fig. 13.7. A 2-bit field, mode select filed for each register in AMR selects the address modification mode between linear or circular mode.

The 5-bit field, block size field BK0 and BK1 is used to select the block size of the circular buffer in circular addressing. The 2-bit field in AMR also specifies which BK (block size) field is to be used for a circular buffer. The mode select field encoding is given in Table 13.6. The calculation of block size for circular addressing based on the 5-bit block size fields in BK0 and BK1 is given below.

Block size in bytes = $2^{(N+1)}$

where, N is the 5-bit value in BK0 and BK1

The buffer must be aligned on a byte boundary equal to the block size. The reserved portion of AMR is always 0 and AMR is initialized to 0 at reset.

Table 13.5 *Control registers common to ′C6X processors*

Register Name	*Abbreviation*	*Description*
Addressing Mode Register	AMR	Specifies linear or circular addressing for eight registers A4-A7 and B4-B7. Also used to select size of the circular buffer in circular addressing
Control Status Register	CSR	Contains important control and status bits of the processor
Program Counter, E1 phase	PCE1	Contains the address of the fetch packet that is in the E1 phase of pipeline
Interrupt Flag Register	IFR	Contains the status of INT4-INT15 and NMI maskable interrupts
Interrupt Set Register	ISR	Used to manually set maskable pending interrupts
Interrupt Clear Register	ICR	Used to manually clear maskable pending interrupts
Interrupt Enable Register	IER	Used to enable/disable the individual maskable interrupts
Interrupt Service Table Pointer	ISTP	Points to the beginning of the interrupts service table
Interrupt Return Pointer	IRP	Contains the address to be used to return from a maskable interrupt
Nonmaskable interrupt Return Pointer	NRP	Contains the address to be used to return from a non-maskable interrupt

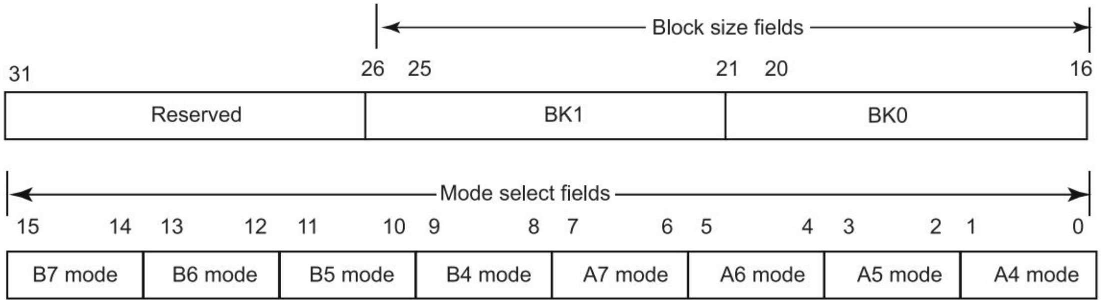


Fig. 13.7 *Address Mode Register (AMR) fields*

Table 13.6 *AMR Mode select field encoding*

Mode select bits	*Description of mode*
0 0	Linear modification of address (default at reset)
0 1	Circular addressing using the BK0 field
1 0	Circular addressing using the BK1 field
1 1	Reserved

13.8.2 Control Status Register (CSR)

The Control Status Register (CSR) of ′C6X contains control and status bits of the processor. The various fields of the CSR are given in Fig. 13.8 and the functions of each field are listed in table 13.7. The bits 0-7 and 10-15 are both readable and writable, but bits 8, 9 and 16-31 are only readable. During reset of the processor, 16 LSB bits are reset to zero; the 16 MSB bits containing Revision ID and CPU ID are fixed for a particular processor.

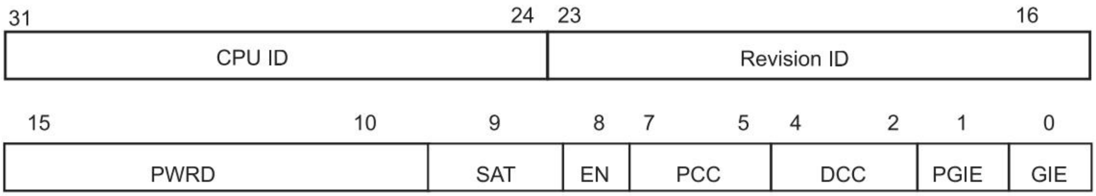


Fig. 13.8 *Control Status Register (CSR) fields*

Table 13.7 *Control Status Register field functions*

Field Name	*Functions of the field*
CPU ID	CPU ID defines which family of CPUs: CPU ID = 00h - ′C62X family of processors CPU ID = 02h - ′C67X family of processors CPU ID = 04h - ′C64X family of processors
Revision ID	Revision ID defines silicon version of the CPU
PWRD	Control power down modes; the values are always read as zero
SAT	The saturate bit. Bit is set only by the functional units when it performs saturate and can be cleared only by the MVC instruction
EN	Endian bit1 = little endian , 0 = big endian
PCC	Program cache control mode
DCC	Data cache control mode
PGIE	Previous GIE bit; saves GIE when an interrupt is taken.
GIE	Global Interrupt Enable bit. Used to enable (1) and disable (0) all the maskable interrupts

13.8.3 Control Register File Extensions

The ′C67X and ′C64X processors contain additional control registers. The ′C67X processor contains three configurations registers to support floating point operations. These registers specify the desired floating-point rounding mode for the .L, .S and .M units. The ′C67X additional control registers and its functions are given in Table 13.8. There in only one additional control register in ′C64X, the Galois Field Polynomial Generator Function Register (GFPGFR). This GFPGFR register along with the Galois Filed Multiply hardware in ′C64X can be used for Reed Solomon encode and decode functions. The GFPGFR register contains 8-bit (0-7) polynomial generator field (POLY) and 3-bit (24-26) field size field (SIZE), remaining bits are reserved. The Galois Field Multiply on ′C64X processor is performed using GMPY4 instruction. The GMPY4 instruction performs four parallel operations on 8-bit packed data on the .M unit. All Galois Multiplies for fields of the form GF (2^m) can be programmed using Galois Field Multiplier in ′C64X. The value of m can range between 1 and 8 using any generator polynomial.

Table 13.8 *Control Register File Extensions in ′C67X*

Register Name	*Abbreviation*	*Description*
Floating-point adder configuration register	FADCR	Specifies underflow mode, rounding mode, not a number (NaN) and other exceptions for the .L unit
Floating-point auxiliary configuration register	FAUCR	Specifies underflow mode, rounding mode, not a number (NaN) and other exceptions for the .S unit
Floating-point multiplier configuration register	FMCR	Specifies underflow mode, rounding mode, not a number (NaN) and other exceptions for the .M unit

Review Questions

13.1 What is VLIW architecture?

13.2 List the processors in ′C6X family. Which processors are code compatible?

13.3 What are the blocks present in the CPU of ′C6X?

13.4 What is register file? What is the size of register files in ′C6X processors?

13.5 What is ′C6X register pair? Explain its use.

13.6 List the various functional units in ′C6X CPU.

13.7 What are the functions performed by .L unit?

13.8 Explain the functions performed by .S unit?

13.9 What are the different multiply operations performed by .M unit?

13.10 List the functions performed by .D unit?

13.11 How many data paths are in ′C6X register file?

13.12 What is register file cross path? What is its use?

13.13 How many data paths are in ′C6X to access memory? What is its size?

13.14 List the control registers common to ′C6X family of processors.

13.15 What are the fields in the addressing mode register? Explain the functions of each field.

13.16 List the additional control registers in ′C64X and ′C62X processors.

Self Test Questions

13.1 The ’C6X processor is based on ——— architecture
(a) Modified Harvard (b) Advanced Harvard
(c) Veloci TI (d) Davinci

13.2 The fixed point devices in ’C6X processors are
(a) ’C62X (b) ’C62X and ’C64X
(c) ’C67X (d) ’C64X

13.3 The floating point devices in ’C6X processors are
(a) ’C62X (b) ’C62X and ’C64X
(c) ’C67X (d) ’C64X

13.4 The number of functional units in ’C6X CPU is
(a) 2 (b) 8 (c) 4 (d) 16

13.5 The size of the ’C6X CPU is
(a) 16-bit (b) 32-bit (c) 40-bit (d) 64-bit

13.6 The number of general purpose register files in ’C6X CPU is
(a) 2 (b) 3 (c) 4 (d) 8

13.7 The number of register in ’C62X and ’C67X CPU register file is
(a) 16 (b) 32 (c) 40 (d) 64

13.8 The number of register in ’C64X CPU register file is
(a) 16 (b) 32 (c) 40 (d) 64

13.9 The number of ALU units in ’C6X CPU is
(a) 8 (b) 4 (c) 6 (d) 2

13.10 The number of multiplier units in ’C6X CPU is
(a) 8 (b) 4 (c) 6 (d) 2

13.11 The ’C6X CPU accepts ——— instructions at a time
(a) 8 (b) 4 (c) 6 (d) 2

13.12 Which units of the following are ALU units in ’C6X CPU?
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.13 The ——— functional unit of ’C6X is used for 32/40 bit shift operation
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.14 The functional unit of ’C6X that can be used for branch operation is
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.15 ——— functional unit of ’C6X is used to load and store the data values.
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.16 The functional unit of ’C6X used for linear and circular addressing is ———
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.17 ——— functional units is used for 32/40 bit compare operations
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.18 The ___ functional unit of ’C6X is used for 32/40 bit count operation
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.19 The number of data paths from ’C6X register file to functional units is
(a) 16 (b) 32 (c) 24 (d) 40

13.20 The number of cross paths in ’C6X register file is
(a) 4 (b) 2 (c) 6 (d) 8

13.21 ——— functional units of ’C62X and ’C67X is not having cross path access
(a) .L unit (b) .S unit (c) .M unit (d) .D unit

13.22 ——— ’C6X processor has all the memory access paths with 32-bit.
(a) ’C62X (b) ’C62X and ’C64X
(c) ’C62X and ’C67X (d) ’C64X

13.23 The ’C6X processor having both load and store paths with 64-bit is ———
(a) ’C62X (b) ’C62X and ’C64X
(c) ’C62X and ’C67X (d) ’C64X

13.24 The ’C6X processor having load path with 64-bit is ———
(a) ’C62X (b) ’C62X and ’C64X
(c) ’C64X and ’C67X (d) ’C64X

13.25 The number of control registers common to ’C6X family of processor is __
(a) 10 (b) 16 (c) 8 (d) 3

13.26 ——— numbers of additional control registers are in ’C67X processor.
(a) 10 (b) 16 (c) 8 (d) 3

14 TMS320C6X ASSEMBLY LANGUAGE INSTRUCTIONS

In the ′C6X family of DSPs ′C62X and ′C64X, ′C62X and ′C67X processors are code compatible. All the fixed point instruction sets of C62X processor are valid for ′C64X and ′C67X processors. The ′C67X is a floating-point device; there are certain instructions unique to it which do not execute on the fixed point devices. Similarly, ′C64X with additional functionality to the ′C62X devices has some unique instructions. This chapter describes about the assembly language instructions corresponding to functional units of the CPU, addressing modes, parallel, and conditional operations. Also, details about the fixed point instructions common to the ′C62X, ′64X and ′C67X devices as well as ′C67X floating-point instructions are described.

FUNCTIONAL UNITS AND ITS INSTRUCTIONS 14.1

The ′C6X devices have six ALU (.L1, .L2, .S1, .S2, .D1 and .D2 units) and two multiplier units (.M1 and .M2 units). The ALU units can perform basic arithmetic and logical operations; apart from that each unit has special functions as listed in Table 13.2. The multiplier unit can perform only multiply operations.

14.1.1 Instructions to .L Functional unit

The .L unit (Basic ALU unit) performs 32/40 bit arithmetic, 32-bit logical, 32/40 bit compare and 32/40 bit count operations. The .L unit of ′C64X processor is used to do dual 16-bit, quad 8-bit arithmetic, byte shifts and 5-bit constant generation operations. The fixed point instructions of .L unit common to ′C62X, ′C64X and ′C67X processors are given in Table 14.1.

14.1.2 Instructions to .S Functional unit

The .S unit (Shift and Branch unit) is a dedicated unit to perform 32/40 bit shift operations and branch operations. It is also used to perform 32-bit arithmetic, 32-bit logical operations and constant generation operations. The ′C64X processor .S unit performs dual 16-bit, quad 8-bit arithmetic operations, dual 16-bit shift operation, dual 16-bit and quad 8-bit compare operations. The fixed point instructions those are common to ′C62X, ′C64X and ′C67X processors for .S unit are given in Table 14.2.

Table 14.1 *Assembly Language Instructions for .L Function unit*

Type of operations	*Mnemonic*	*Description*
Arithmetic operations	ABS	Integer absolute value with saturation
	ADD/ADDU	Signed/unsigned integer addition operation without saturation
	SADD	Integer addition operation with saturation to result size
	SSUB	Integer subtraction operation with saturation to result size
	SUB/SUBU	Signed/unsigned integer subtraction operation without saturation
	SUBC	Conditional integer subtract and shift operation
	NEG	Negate operation (Pseudo-operation)
Logical operations	AND	Bitwise AND operation
	NOT	Bitwise NOT operation
	OR	Bitwise OR operation
	XOR	Bitwise XOR operation
Compare operations	CMPEQ	Integer compare operation for equality
	CMPGT/ CMPGTU	Signed/unsigned integer compare operation for greater than
	CMPLT/ CMPLTU	Signed/unsigned integer compare operation for less than
Other operations	NORM	Normalize integer operation
	MV	Move from register to register operation(Pseudo-operation)
	LMBD	Left most bit detection operation
	SAT	Saturate a 40-bit integer to a 32-bit integer operation
	ZERO	Zero a register (pseudo-operation)

Table 14.2 *Assembly Language Instructions for .S Function unit*

Type of operations	*Mnemonic*	*Description*
Arithmetic operations	ADD	Signed integer addition operation without saturation
	ADDK	Integer addition operation using signed 16-bit constant
	ADD2	Two 16-bit integer addition on upper and lower register halves
	SUB/SUBU	Signed/unsigned integer subtraction operation without saturation
	SUB2	Two 16-bit integer subtractions on upper & lower register halves
	NEG	Negate operation (Pseudo-operation)
Logical operations	AND	Bitwise AND operation
	NOT	Bitwise NOT operation
	OR	Bitwise OR operation
	XOR	Bitwise XOR operation
Shift operations	SHL	Arithmetic shift left operation
	SHR	Arithmetic shift right operation
	SHRU	Logical shift right operation

(Contd.)

Table 14.2 *(Contd.)*

	SSHL	Shift left with saturation operation
Branch operations	B disp	Branch operation using a displacement
	B reg	Branch operation using a register
	B NRP	Branch operation using NMI return pointer
	B IRP	Branch operation using interrupt return pointer
Move operations	MV	Move from register to register operation(Pseudo-operation)
	MVC	Move between control file and the register file operation
	MVK	Move a 16-bit signed constant into a register and sign extend
	MVKH/ MVKLH	Move 16-bit constant into the upper/lower bits of a register
Other operations	CLR	Clear a bit field operation
	EXT/EXTU	Extract and sign-extend/zero-extend a bit field operation
	SET	Set a bit field operation
	ZERO	Zero a register (Pseudo-operation)

14.1.3 Instructions to .M Functional unit

The .M unit (Multiply unit) is a dedicated unit, which performs 16x16 bit multiply operations. In the ′C64X processor, in .M unit 16×32, dual 16×16 and quad 8×8 multiply operations can be performed. The fixed point instructions of .M unit common to ′C62X, ′C64X and ′C67X processors are given in Table 14.3.

Table 14.3 *Assembly Language Instructions for .M Function unit*

Type of operations	*Mnemonic*	*Description*
Multiply operations	MPY/MPYU/MPYUS/MPYSU	Signed/unsigned integer multiply of 16LSB X 16 LSB operation
	MPYH/MPYHU/MPYHUS/MPYHSU	Signed/unsigned integer multiply of 16MSB X 16 MSB operation
	MPYHL/MPYHLU/MPYHULS/MPYHSLU	Signed/unsigned integer multiply of 16MSB X 16 LSB operation
	MPYLH/MPYLHU/MPYLUHS/MPYLSHU	Signed/unsigned integer multiply of 16LSB X 16 MSB operation
	SMPY/SMPYHL/SMPYLH/SMPYH	Integer multiply with left shift and saturation operation

14.1.4 Instructions to .D Functional unit

The .D unit (Data access unit) is a dedicated unit for memory access. The linear and circular address generation, load and store operations with 5-bit constant offset are performed by this unit. The load and store operations of .D2 unit alone can have 15-bit constant offset. Apart from the data access, .D unit is used to do only 32-bit add and subtract operations. The .D unit of ′C64X is used to perform 32-bit logical operations and 5-bit constant generation. The fixed point instructions of .D unit common to ′C62X, ′C64X and ′C67X processors are given in Table 14.4.

Table 14.4 *Assembly Language Instructions for .D Function unit*

Type of operations	*Mnemonic*	*Description*
Arithmetic operations	ADD	Signed integer addition operation without saturation
	ADDAB/ADDAH/ ADDAW	Integer addition using byte/half word/word addressing mode
	SUB	Signed integer subtraction operation without saturation
	SUBAB/SUBAH/ SUBAW	Integer subtraction using byte/half word/word addressing mode
Load store operations	LDB/LDBU/ LDH/ LDHU/ LDW	Load byte/half word/word from memory with 5-bit/15-bit unsigned constant offset or register offset
	STB/STH/STW	Store byte/half word/word to memory with 5-bit/15-bit unsigned constant offset or register offset
Other operations	MV	Move from register to register operation(Pseudo-operation)
	ZERO	Zero a register (Pseudo-operation)

ADDRESSING MODES 14.2

The addressing modes of ′C62X, ′C64X and ′C67X are

(i) Register addressing mode
(ii) Linear addressing mode or (Indirect addressing mode)
(iii) Circular addressing mode

All the functional units (.L, .S, .M and .D) with all registers in the register file (A0-A15 and B0-B15) are used to perform Register addressing. For linear and circular addressing mode, .D unit alone is used. All the registers of the register file are used for linear addressing mode, but for circular addressing the registers A4-A7 are used by the .D1 unit and registers B4-B7 are used by the .D2 unit

14.2.1 Register Addressing Mode

The register file of ′C62X and ′C67X contains 32 registers and of ′C64X contains 64 registers. The content of these registers are used as operand. The syntax of the assembly language instruction for register addressing mode is given below. The instruction contains four fields, the mnemonic, functional unit, source operands and the destination operand.

mnemonic .unit src1, src2, dst

The mnemonic filed is for the assembly codes like ADD, MPY and SUB etc that support register addressing mode. For the .unit field, any of the eight functional units is specified depending upon the operation performed as per the Table 14.1 to 14.4. The source operands (*src1, src2)* and destination operand (*dst)* are the registers of the register file.

Example 14.1 ADD .L1 A1,A2,A3 - This instruction adds the hexadecimal signed integer operands in register A1 and A2. The result is stored in register A3. The content of register A1 and A2 are unchanged. The functional unit used is .L1 and the registers of path A are used for both source and destination operands.

	Before execution		After execution
A1	11223344	A1	11223344
A2	33445566	A2	33445566
A3	22222222	A3	446688AA

In the above example, to perform add operation the functional units .S1and .D1 are also used. For the source and destination operands, registers from register file A alone are to be used. Same way, to do add operation in register path B, the functional units .L2, .S2 and .D2 are used. The source and destination operand registers are to be used only from register file B (B1-B15 registers). For the arithmetic and logic instructions, the source and destination operand can be specified with same register of the register file.

Example 14.2 ADD .S2 B1,B2,B2 - This instruction adds the hexadecimal signed integer operands in register B1 and B2. The result is stored in register B2 itself after addition. The content of register B1 is unchanged. The functional unit used is .S2 and the registers of path B are used for both source and destination operands.

	Before execution		After execution
B1	3456789A	B1	3456789A
B2	11112222	B2	45679ABC

The data path of ′C6X architecture has cross paths between path A and B (1X &2X). This cross path is used to access one of the source operand from the opposite path. The destination operand cannot use the cross path.

Example 14.3 ADD .L1X A1,B2,A2 - This instruction adds the hexadecimal signed integer operands in register A1 and B2. The result is stored in register A2. The content of register A1 and B2 are unchanged. The functional unit used is .L1 and the registers of path A are used for the source operand (A1) and destination operand (A2). The source operand B2 is obtained through cross path from register file B.

	Before execution		After execution
A1	22221111	A1	22221111
B2	33332222	B2	33332222
A2	44444444	A2	55553333

14.2.2 Linear Addressing Mode

The linear addressing mode uses .D (.D1 and .D2) unit alone, along with all the registers of the register file. The load instruction, store instruction, add and subtract with addressing mode instructions can use linear addressing mode. These instructions are of three kinds, byte access, half word access and word access. The syntax of the linear addressing mode type instruction is given below. The instruction contains four fields, the mnemonic, functional unit, mode field and destination field.

mnemonic .unit mode field, dst

The mnemonic field uses load, store, add and subtract with addressing mode instructions only (LDB(U)/ LDH(U)/ LDW, STB(U)/ STH(U)/ STW, ADDAB/ ADDAH/ ADDAW/ADDAD & SUBAB/ SUBAH/SUBAW). For the unit field, the .D1 and .D2 units are used. The mode field specifies the type of address access and address modification type. The destination field (*dst*) can use any of the register in the register file. The different types of mode fields that are used in linear addressing mode are given in Table 14.5. The register containing the base address of the operand is denoted as **baseR**. The offset (displacement) from the base address specified in some register is represented as **offsetR**. Instead of using register to specify the offset, a 5-bit unsigned constant can be used as an offset, which is denoted as **ucst5**. The registers used for **baseR** and **offsetR** are must be in the same register file. The destination (***dst***) register can be from the opposite register file through cross path.

Table 14.5 *Address generation Option for Mode field in Linear addressing mode*

Mode field Syntax	*Address modification performed*
*+baseR[offsetR/ucst5]	Positive offset from baseR specified by offsetR/ucst5
*-baseR[offsetR/ucst5]	Negative offset from baseR specified by offsetR/ucst5
*++baseR[offsetR/ucst5]	Pre-increment from baseR specified by offsetR/ucst5
*––baseR[offsetR/ucst5]	Pre-decrement from baseR specified by offsetR/ucst5
*baseR++[offsetR/ucst5]	Post increment from baseR specified by offsetR/ucst5
*baseR––[offsetR/ucst5]	Post decrement from baseR specified by offsetR/ucst5

The offset value specified in the offset register (**offsetR)** or the 5-bit unsigned constant given in the instruction is left shifted by 0, 1 or 2 for the byte, halfword and word access instructions respectively. Then, to find the address of the operand the following procedure is used:

(i) The shifted offset value is added or subtracted from the value in the base register (**baseR**) for *+ or *- mode fields respectively. The added or the subtracted value from the content of the base register is the address of the operand to be accessed from memory. The content of the base register is unchanged.

(ii) For *++ or *–– mode fields, the address of the operand is calculated as mentioned in (i), but the content of the base register increments or decrements by the shifted offset value respectively before accessing the memory (pre-increment/pre-decrement). The address of the operand is incremented or decremented value from the base register content.

(iii) In the case of *baseR++ or *baseR––, the address of the operand is calculated as mentioned in (i), but the content of the base register increments or decrements by the shifted offset value respectively after accessing memory (post increment/post decrement). The address of the operand is the content of the base register, after accessing the address changes as per the address modification syntax.

Example 14.4 LDW .D1 *+A0[1],A1 – This instruction loads a hexadecimal word from memory to register A1. The address of the memory is the base address value in register A0 added with the 5-bit constant offset given in brackets left shifted by two times. If the base address is 500h, the given offset 1 is left shifted by two times is 4, the address of the memory to be accessed is 504h. The content of A0 is unchanged after access.

	Before execution		After execution
A0	00000500	A0	00000500
B1	11111111	A1	3456789A
504h	3456789A	504h	3456789A

Example 14.5 LDW .D1 *++A0[A4],A1 - This instruction loads a hexadecimal word from memory to register A1. The address of the memory is the base address value in register A0 added with the content of offset register A4 given in brackets left shifted by two times. If the base address is 500h, the content of offset register A4 is say 4, then it is left shifted by two is 10h (16). The content of A0 is incremented to 510h before accessing the memory. The address of the memory to be accessed is 510h. The content of offset register A4 is unchanged after access.

	Before execution		After execution
A0	00000500	A0	00000510
A4	00000004	A4	00000004
A1	34587698	A1	55667788
510h	55667788	510h	55667788

Example 14.6 LDW .D1 *A0++[2],A1 - This instruction loads a hexadecimal word from memory to register A1. The address of the memory is the base address value in register A0. After accessing the memory the new address in register A0 is the content of A0 added with the content of offset given in brackets left shifted by two times. If the base address is 500h, the address of the memory to be accessed is 500h. If the offset given is say 2, the two times left shifted value is 8h. Then the new address in A0 is the register value A0 added with the left shifted value i.e. 508h.

	Before execution		After execution
A0	00000500	A0	00000500
B1	76234589	A1	99887766
500h	99887766	500h	99887766

14.2.3 Circular Addressing Mode

In circular addressing mode .D1 unit of register path-A and .D2 unit of register path-B is used. The registers A4-A7 of path-A and B4-B7 of path-B can be used for circular addressing. To activate the circular buffer the corresponding mode select bits (two bit field), the size of the block size (BK0/BK1, 5-bit field) are to be loaded in Address Mode Register (AMR) as given in section 13.8.1.

The load instruction, store instruction, add with addressing mode and subtract with addressing mode instructions can use circular addressing mode. These instructions are of three kinds, byte access, half word access and word access. The syntax of the circular addressing mode for load and store instructions is given below.

mnemonic .unit mode field, dst

The instruction contains four fields, the mnemonic, functional unit, mode field and destination field. The mnemonic field can be load and store instructions as described in Section 14.2.2. The unit field has to be .D1 or .D2 unit. The mode field specifies the type of address modification, the different types of address modification that are used in circular addressing mode is given in Table 14.5. In circular addressing mode the base register (**baseR)** specified in the mode field must be only the registers A4-A7 and B4-B7, the destination **(dst**) register can be any of the registers of the register file.

The offset value specified in the offset register **(offsetR)** or the 5-bit unsigned constant given in the instruction is left shifted by 0, 1 or 2 for the byte, halfword and word access instructions respectively. The address increment/decrement for the shifted offset value happens up to the end address/start address of the circular buffer; once it is reached, the address is wrapped around to the start/end address of the circular buffer.

Example 14.7

For circular addressing mode, register A4 is used. To specify the block size, BK0 field in AMR register is used. The two bit mode field for A4 is 01 and the 5-bit field to specify the block size in BK0 is 01, hence the control word for AMR is 00010001h. The size of the block is 2^{1+1} = 4. If the starting address of the memory is 0x0100h, the circular buffer boundary is from 0x0100h to 0x0103h. Content of memory locations 0100h-0103h is $\boxed{44332211}$

```
MVK      .S1    0X0001,A0      ;move the two bit mode field value to LSB of A0
MVKLH    .S1    0X0001,A0      ;move the 5-bit BK0 value to MSB of A0
MVC      .S     2X A0,AMR      ;move the control word from A0 to AMR register
MVK      .S1    0X0100,A4      ;the register A4 is loaded with the start address of
                               the buffer 0x0100h
LDB      .D1    *A4++[1], A1   ;load byte from the address of the memory pointed
NOP 4                          by A4 register to A1 register, increment content
                               of A4 by one. Followed by that is 4 no operations
                                    Before executions      After execution
                                    A4   00000100          A4   00000101
                                    A1   00000000          A1   00000011
LDB      .D1    *A4++[1], A1        A4   00000101          A4   00000102
NOP 4                               A1   00000011          A1   00000022
LDB      .D1    *A4++[1], A1        A4   00000102          A4   00000103
NOP 4                               A1   00000022          A1   00000033
LDB      .D1    *A4++[1], A1        A4   00000103          A4   00000100
NOP 4                               A1   00000033          A1   00000044
```

In this example, the memory address increments by one location for each load byte instruction, once it reaches the end of the buffer 0x0103h, the next content in A4 is 0x100h. The data access happens circularly between 0x0100h to 0x0103h address locations.

The syntax of the circular addressing mode instruction for add with addressing mode and subtract with addressing mode case is given below.

mnemonic .unit src2, src1, dst

The mnemonic field can be add and subtract with addressing mode instructions given in Section 14.2.2. The source operand ***src2*** should be registers A4-A7 and B4-B7 of the respective data paths. The source operand ***src1*** can be any register in the register file and the destination operand ***dst*** should be the same register used for source operand **src2**.

The content of source operand ***src1*** in the instruction is left shifted by 0, 1 or 2 for the byte, half-word and word access instructions respectively. The shifted content of ***src1*** is added/subtracted from the content of ***src2,*** if the added/subtracted content is exceeding the circular buffer boundary, the content

src2 is wrapped around with in the buffer size, the result is available in the destination register ***dst.*** The content of src2 is always within the circular buffer size.

Example 14.8 For circular addressing mode, register B5 is used. To specify the block size, BK1 field in AMR register is used. The two bit mode field for B5 is 10 and the 5-bit field to specify the block size in BK1 is 03, hence the control word for AMR is 00600800h. The size of the block is 2^{3+1} = 16. If the starting address of the memory is 0x0100h, the circular buffer boundary is from 0x0100h to 0x010Fh.

```
MVK      .S2      0X0800,B0    ; move the two bit mode field value to LSB of B0
MVKLH    .S2      0X0060,B0    ; move the 5-bit BK1 value to MSB of B0
MVC      .S2      B0,AMR       ; move the control word from B0 to AMR register
MVK      .S1X     0X0100,B5    ; the register B5 is loaded with the start address of
                                 the buffer 0x0100h using cross path
MVK      .S2      0x0002,B1    ;the register B1 is loaded with the value 02h
ADDAH    .D1      B5,B1,B5     ; the content of B1 is left shifted by one (04h),
                                added with the content of B5(0100h), result
                                stored in B5(0104h). The content of B1 is
                                unchanged
                                     Before executions     After execution

                                   B5  | 00000100 |     B5  | 00000104 |
ADDAH    .D1      B5,B1,B5         B5  | 00000104 |     B5  | 00000108 |
ADDAH    .D1      B5,B1,B5         B5  | 00000108 |     B5  | 0000010C |
ADDAH    .D1      B5,B1,B5         B5  | 0000010C |     B5  | 00000100 |
ADDAH    .D1      B5,B1,B5         B5  | 00000100 |     B5  | 00000104 |
```

In this example, the content of B5 increments by a value of 04h for each time ADDAH instruction is executed. Once the content of B5 exceeds the end value of the circular buffer 0x010Fh, it is wrapped around to the first value 0x0100h. The register B5 content increments four values within the circular buffer size.

FIXED POINT INSTRUCTIONS 14.3

In this section, the fixed-point instruction details those are common for ′C62X, ′C64X and ′C67X processors are given. The syntax of the instruction, the functional unit details and the addressing modes of the instruction are given with examples.

14.3.1 Move Instructions

The move instructions are used to move the contents between registers of the register file, control register file to register files and also to move 16-bit constant into the lower and upper bits of the registers of the register file. To move contents between registers all .L, .S and .D units are used and to move values between control register file and register file, .S2 unit alone is used. To move 16-bit constant to registers of the register file, .S1 and .S2 units are used, but in ′C64X processors, for the same 16-bit constant move operation all .L, .S and .D units are used. The register file cross path is not accessible for MVK instruction. The addressing mode used for move instructions is only register addressing mode. The instruction and its description of the ′C6X processors move instructions are listed in Table 14.6.

Table 14.6 *Move Instructions of ′C6X processor*

Instruction	*Functional unit*	*Description*
MV	.L1 or.L2,.S1 or.S2, .D1 or .D2	Move value from one register to another register in register file
MVC	.S2 only	Move value between control register file and register file
MVK	.S1 or .S2(all .L, .S and .D units in ′C64X only)	Move a 16-bit constant into lower 16-bits of a register and sign extended
MVKLH	.S1 or .S2	Move a 16-bit constant into upper 16-bits of a register
MVKH	.S1 or .S2	Move upper 16-bit value of 32-bit constant to upper 16-bits of a register

Example 14.9 MV .S1 A1,A2 - Move register to register instruction. The content of register A1 is moved to register A2, the content of register A1 is unchanged and the functional unit used is .S1

	Before executions		After execution
A1	22334455	A1	22334455
A2	20408754	A2	22334455

MV .L1X B1,A3 - Move register to register instruction using the cross path. The content of register B1 is moved to register A3, the content of B1 is unchanged and the functional unit used is .L1

	Before executions		After execution
A3	30504321	A3	547698AB
B1	547698AB	B1	547698AB

Example 14.10 MVC .S2 A1,AMR - Move value between control register file and register file instruction. The content of register A1 is moved to Address mode register (AMR) in control register file, the content of register A1 is unchanged and the functional unit used is .S2

	Before executions		After execution
A1	00020005	A1	00020005
AMR	00400001	AMR	00020005

MVC .S2 AMR,B2 - The content of AMR is moved to register B2, the content of AMR is unchanged and the functional unit used is .S2

	Before executions		After execution
AMR	00020005	AMR	00020005
B2	20408754	B2	00020005

Example 14.11 MVK .S1 0x1223,A1 - Move the 16-bit constant to lower 16-bit of register in register file. The 16-bit constant 1223h is moved to lower 16-bits of register A1 and the functional unit used is .S1

	Before executions		After execution
A1	00020005	A1	00021223

MVK .S2 -0x012,B2 - The negative 16-bit constant -012h is moved to lower 16-bit of register B2 and the sign bit is extended to MSB bits. The 2-s complement value of 012h (FFEDh) appears as result in register B2 lower 16-bit.The MSB bits are sign extended and functional unit used is .S2

	Before executions		After execution
B2	00050002	B2	FFFFFFED

Example 14.12 MVKLH .S1 0x3344,A2 – Move the lower 16-bit constant to upper 16-bit of register in register file. The 16-bit constant 3344h is moved to upper 16-bit of register A2, the lower 16-bits are unchanged and functional unit used is .S1

	Before executions		After execution
A2	00220055	A2	33440005

MVKH .S2 0x44552233,B2 – The upper 16-bit of the 32-bit constant is moved to upper 16-bit of register B2. The upper 16-bit value 4455h is moved to register B2 upper 16-bit, lower 16-bit are unchanged. The functional unit used is .S2

	Before executions		After execution
B2	20404252	B2	44554252

14.3.2 Load/Store Instructions

The load and store instructions are used for the memory to register file and register file memory data transfer through load and store data paths respectively. The various types of load/store instructions are based on byte, half- word and word access. For load and store instructions linear and circular addressing modes are used (Sections 14.2.2, 14.2.3). The offset value given in the instruction is scaled by left-shift of 0, 1 or 2 for byte, half-word or word access respectively, It is added or subtracted from the base register content based on the address modification specified in the instruction. The functional units used for load and store operations with register offset or 5-bit constant offset are .D1 and .D2 units. For 15-bit constant offset type of instructions, .D2 unit alone is used and the base register that could be used in the instruction is B14 and B15 only. The different load and store instructions of ′C6X processors and its description are listed in Table 14.7.

Table 14.7 *Load Instructions of ′C6X processor*

Instruction	*Functional unit*	*Description*
LDB/STB	.D1 or .D2	Load byte from memory to register in register file/Store byte from register in register file to memory
LDBU	.D1 or .D2	Load byte unsigned from memory to register in register file
LDH/STH	.D1 or .D2	Load half word from memory to register in register file/ Store half word from register in register file to memory
LDHU	.D1 or .D2	Load half word unsigned from memory to register in register file
LDW/STW	.D1 or .D2	Load word from memory to register in register file/ Store wrod from register in register file to memory

(For 15-bit constant offset, functional unit used is .D2 only)

Example 14.13 LDB .D1 *A0,A1 – Load byte instruction. The byte content of memory location, who's address is present in base address register A0 is loaded into register A1, the sign bit is extended to MSB bits of register A1. The memory address is 100h, the byte content of 100h location is 44h. The value 44h is moved to LSB of A1 register and MSB bits are zero filled. The content of register A0 and 100h location are unchanged; the functional unit used is .D1

	Before executions		After execution
A1	11111111	A1	00000044
A0	00000100	A0	00000100
100h	11223344	100h	11223344

LDH .D1 *+A0[2],A2 – Load Half-word instruction with positive offset. To calculate the address of the memory to be accessed, the 5-bit constant offset given in the instruction is left shifted ones and added to base register content A0. The content of register A0 is unchanged. The half-word content of the memory address is moved to register A2. The offset value 2 left shifted once is 4; the content of base register is 100h, hence the address of memory is 104h. The half-word content of memory (104h) 8899h is loaded into register A2 LSB, the MSBs are sign extended as FFFF. The functional unit used is .D1

	Before executions		After execution
A2	11111111	A2	FFFF8899
A0	00000100	A0	00000100
104h	44558899	104h	44556677

LDW .D1 *++A0[1],B2 – Load word instruction with pre-increment. The 5-bit constant offset given in the instruction is left shifted two times and added to base register content A0. The content of base register A0 is pre-incremented and it is the address of the memory to be accessed. The word content of memory address is moved to register B2. The offset value 1 left shifted twice is 4. If the content of base register A0 is 100h, the new content of A0 is 104h and the memory address is also 104h. The word content of memory (104h) 44558899h is loaded into register B2. The content of 104h location is unchanged, the functional units used is .D1 along with the cross path.

	Before executions		After execution
B2	11111111	B2	44558899
A0	00000100	A0	00000104
104h	44558899	104h	44558899

STW .D2 B3,*B1--[B0] – Store word instruction with post-decrement. The content of register B3 is stored in memory. The address of the operand is the content of base register B1 and the offset is specified in offset register B0. The content of offset register B0 is left shifted by two times and subtracted from the content of base register B1 and that is the new content in base register B1. If the content of register B3 is 00004578h, the content of base register B1 is 100h the content of register B3 is stored in the memory location pointed by register B1. The content of base register B0 is 1h, left shifted twice is 4h, which subtracted from 100h is 0FCh that is the new content in B1. The content of B0 and B3 are unchanged and the functional unit used is .D2

	Before executions		After execution
B3	00004578	B3	00004578
B1	00000100	B1	000000FC
B0	00000001	B0	00000001
100h	11223344	100h	00004578

14.3.3 Add Instructions

The various types of add instructions of ′C6X processor and its description are given in Table 14.8. All add instructions use register addressing mode except addition using addressing mode instructions (ADDAB/ADDAH/ADDAW). Addition with addressing mode instructions use circular addressing mode (Section 14.2.3). Signed integer addition with and without saturation, unsigned integer addition, 16- bit constant and two 16-bit integer addition operations can be performed using add instructions. To perform signed integer addition operation, all .L, .S and .D functional units are used. For unsigned integer addition and integer addition with saturation, .L1 and .L2 units are used. To add 16-bit constant and two 16-bit integers, .S1 and .S2 units are used. Integer addition with addressing mode instructions uses .D1 and .D2 units.

Table 14.8 *Add Instructions of 'C6X processor*

Instruction	*Functional unit*	*Description*
ADD	.L1 or.L2, .S1 or .S2, .D1 or .D2	Signed integer addition without saturation
ADDU	.L1 or .L2	Unsigned integer addition without saturation
SADD	.L1 or .L2	Integer addition with saturation to result size
ADDK	.S1 or .S2	Integer addition using singed 16-bit constant
ADD2	.S1 or .S2	Two 16-bit integer additions on upper and lower register halves
ADDAB/ADDAH/ ADDAW	.D1 or .D2	Integer Byte/ Half-word/Word addition using addressing mode

Example 14.14 ADD .D1 31,A0,A1 - Five bit signed constant (-31 to 31) add instruction. The given five bit signed constant is added to the content of register A0 and the result is stored in register A1. The 5-bit constant 31 is added to register A0 content 00008754h and the result 00008773h is loaded in register A1. The functional unit used is .D1, the register content A0 is unchanged.

	Before executions		After execution
A0	00008754	A0	00008754
A1	00020005	A1	00008773

Example 14.15 ADD .L1 A0,A1,A2 - Signed 32-bit integer add instruction. The signed integer content of register A0 and A1 are added, the result is stored in register A2. If the 32-bit positive integers 00045566h in A0 and 00076655h in A1, they are added, the result 000BBBBBh is stored in register A2. The content of registers A0 and A1 are unchanged, the functional unit used is .L1

	Before executions		After execution	
A0	00045566	A0	00045566	+284006
A1	00076655	A1	00076655	+484949
B0	12348765	B0	000BBBBB	+768955

ADD .S2X A0,B1,B2 - If the 32-bit positive integer in register A0 is 00045566h and the negative integer in register B1 is FFFFC742h, they are added and the result 00041CA8h is stored in register B2. The content of registers A0 and B1 are unchanged, the functional unit used is .S2 with the cross path.

	Before executions		After execution	
A0	00045566	A0	00045566	+284006
B1	FFFFC742	B1	FFFFC742	- 14526
B2	12348765	B2	00041CA8	+269480

Example 14.14 SADD .L1 A1,A2,A3 - Signed integer add instruction with saturation. The signed integer content of register A1 and A2 are added; the added result is stored in register A3, if there is no saturation. If the result is saturated, for positive integer the maximum positive number (7FFF FFFF) and for negative integer the negative number (8000 0000) is loaded in register A3 respectively. The SAT bit in CSR register is set. The functional unit used is .L1; the content of registers A1 and A2 are unchanged.

	Before executions		After execution
A1	7D007D00	A1	7D007D00
A2	13881388	A2	13881388
A3	12247666	A3	7FFF FFFF

Example 14.17 ADDU .L1 A5,A6,A9:A8 – Unsigned 32-bit add instruction. The unsigned 32-bit contents in register A5 and register A6 are added and the resultant 40-bit content is loaded in A9:A8 register pair. If the 32-bit integer in register A5 is 00087654h and in register A6 is FFFF4332h, both of them are added and the resultant 40 bit content 10007B986h is loaded in A9:A8 register pair. The functional unit used is .L1, the register contents A5 and A6 are unchanged.

	Before executions			After execution		
A5	00087654		A1	00087654		+554580
A6	FFFF4332		A6	FFFF4332		+4294918962
A9:A8	00020005	00020005	A9:A8	00000001	0007B986	+4295473542

Example 14.18 ADDK .S1 2345,A1 – A 16-bit signed constant add instruction. The signed 16-bit constant given in the instruction is added with the content of register A1 and the result stored in A1. If the 16-bit positive constant is 2345 (0929h), the content in register A1 is 00015432h, they are added, the result 00015D5Bh is stored in register A1. The functional unit used is .S1

	Before executions		After execution
A1	00015432	A1	00015D5B

ADD2 .S2 B1,B2,B3 – Two 16-bit integer add instruction on upper and lower register halves. The upper and lower halves content of register B1 are added to the upper and lower halves content of register B2, the result is stored in upper and lower halves of register B3 respectively. If the content of register B1 is 00347698h, register B2 is 03127654h, the upper and lower halves are added, the result 0346ECECh stored in register B3. The content of registers B1 and B2 are unchanged, the functional unit used is .S2.

	Before executions		After execution
B1	00347698	B1	00347698
B2	03127654	B2	03127654
B3	00000544	B3	0346ECEC

14.3.4 Subtract Instructions

The various types of subtract instructions of ′C6X processor and its descriptions are given in Table 14.9. All subtract instructions use register addressing mode except subtraction using addressing mode instructions (SUBAB/SUBAH/SUBAW). Subtraction with addressing mode instructions use circular addressing mode (Section 14.2.3). Signed integer subtraction with and without saturation, unsigned integer subtraction, conditional integer subtraction and two 16-bit integer subtraction operations can be performed using subtract instructions. To perform signed integer subtraction, all .L, .S and .D functional units are used. For unsigned integer subtract operation .L and .S units are used. .L units are used for conditional integer subtract and integer subtract with saturation operations. To subtract two 16-bit integers, .S1 and .S2 units are used. Integer subtraction with addressing mode instructions uses .D1 and .D2 units. The SUB, SUBU, SSUB, SUB2 and subtraction using addressing mode instruction operation modes are same us the respective add instructions, except that the operands are subtracted rather than addition. The conditional subtract instruction SUBC is used for signed and unsigned integer division operation.

Table 14.9 *Subtract Instructions of 'C6X processor*

Instruction	*Functional unit*	*Description*
SUB	.L1 or.L2, .S1 or.S2, .D1 or .D2	Signed integer subtraction without saturation(For .D units src1 is subtracted from src2)
SUBU	.L1 or .L2, .S1or .S2	Unsigned integer subtraction without saturation
SUBC	.L1 or .L2	Conditional integer subtract and shift used for division
SSUB	.L1 or .L2	Integer subtraction with saturation to result size
SUB2	.S1 or .S2	Two 16-bit integer subtractions on upper and lower register halves
SUBAB/SUBAH/ SUBAW	.D1 or .D2	Integer Byte/ Half-word/Word subtraction using addressing mode

Example 14.19 SUBC .L1 A1,A2,A3 - Conditional subtract and shift operation. The content of register A2 is subtracted from the content of register A1. If the subtraction result is ≥ 0, then the result is left shifted by one bit and 1 is added to LSB bit and the final value is loaded in register A1. Else the subtracted result is less than zero, the content of register A1 is left shifted by one bit and the shifted value is loaded in register A1.

(i) If the register content A2 is 00000404h and A1 is 00002222h, the A2 content is subtracted from A1 content. The result 00001E1E which is > 0 is left shifted by 1 bit (00003C3C) and 1 is added to LSB bit, the final result 00003C3D is loaded in register A3.

	Before executions		After execution
A1	00002222	A1	00347698
A2	00000404	A2	00000404
A3	12243333	A3	00003C3D

(ii) If the register content A2 is 00002424h and A1 is 00002222h, the A2 content is subtracted from A1 content. The result is less than zero. The content of register A1 is left shifted by 1 bit, the result 00004444h is loaded in register A3. The content of register A1 and A2 are unchanged and the unit used is .L1.

	Before executions		After execution
A1	00002222	A1	00347698
A2	00002424	A2	00002424
A3	12243333	A3	00004444

14.3.5 Multiply Instructions

All multiply instructions of 'C6X use register addressing mode. The various multiply instructions of 'C6X processor and its descriptions are given in Table 14.10. The multiply instructions are of signed and unsigned type. Multiplication operation can be performed on 16 LSBs, 16 MSBs, 16 LSBs with 16 MSBs and vice versa on the register file register contents. Integer multiplication with left shift and saturation can also be performed on lower and higher order register contents of the register file. To perform multiplication .M1 and .M2 units are used. The MPY and MPYSU instructions supports signed 5 bit constant multiplication.

Table 14.10 *Multiply Instructions of 'C6X processor*

Instruction	*Functional unit*	*Description*
MPY/MPYU	.M1 or .M2	Signed/Unsigned integer multiplication on 16 LSBs
MPY**H**/MPY**H**U		Signed/Unsigned integer **multiplication on 16 MSBs**
MPY***LH***/MPY***LH***U	.M1 or .M2	Signed/Unsigned integer multiply on ***16 LSBs*** and **16MSBs**
MPY**H*L***/MPY**H*L***U		Signed/Unsigned integer multiply on **16 MSBs** and ***16LSBs***
MPYUS/MPYSU	.M1 or .M2	US-unsigned and signed/ SU-signed and unsigned integer multiplication on 16 LSBs
MPYHUS/MPYHSU		US-unsigned and signed/ SU-signed and unsigned integer multiplication on 16 MSBs
MPYLUHS/MPYLSHU	.M1 or .M2	Unsigned 16 LSBs and signed 16 MSBs /signed 16 LSBs and unsigned 16 MSBs multiplication
MPYHULS/MPYHSLU		Unsigned 16 MSBs and signed 16 LSBs/signed 16 MSBs ans unsigned 16 LSBs multiplication
SMPY/SMPYH	.M1 or .M2	Integer multiplication with left shift and saturation on 16 LSBs /16 MSBs
SMPYHL/SMPYLH		16MSBs and 16 LSBs/16 LSBs & 16 MSB s

Example 14.20 MPYU .M1 A1,A2,A3 - Unsigned integer multiply instruction. The Unsigned 16-bit number present in 16 LSBs of registers A1 and A2 are multiplied and the result is stored in register A3. If the content of register A1 is 56003442h and register A2 is 23451122h, the 16 LSBs are multiplied and the result 037F52C4h is stored in register A3. The content of register A1 and A2 are unchanged and the functional unit used is .M1

	Before executions			After execution		
A1	56003442	13378	A1	56003442	13378	16 LSB value
A2	23451122	4386	A2	23451122	4386	16 LSB value
A3	00007689		A3	037F52C4	58675908	Product value

MPYHL .M1 A4,A5,A6 - Signed integer multiply instruction on 16 MSBs and 16 LSBs of registers. The signed 16-bit number present in 16 MSBs of registers A4 and 16 LSBs of register A5 are multiplied and the result is stored in register A6. If the content of register A4 is FFA13344h and register A5 is 48480044h, the 16 MSBs (FFA1h) and 16 LSBs (0044h) are multiplied and the result FFFFE6C4h is stored in register A6. The content of register A4 and A5 are unchanged and the functional unit used is .M1

	Before executions			After execution		
A4	56003442	(-95)	A4	FFA13344	(-95)	16 MSB value
A5	48480044	(68)	A5	48480044	(68)	16 LSB value
A6	00560544		A6	FFFFE6C4	(-6460)	Product value

Example 14.21 SMPYLH .M1 A1,A2,A3 - Integer multiply with left shift and saturation instruction. The signed number in 16 LSBs of register A1 and 16 MSBs of register A2 are multiplied and the result is left shifted by one bit and stored in register A3. If the left shifted result is 8000 0000h, then the result is saturated to 7FFF FFFFh. If the content of register A1 is F023 3344h, register A2 is 8787 4A81h, the 16 LSBs (3344h) and 16 MSBs (8787h) are multiplied and the result E7DF E4DCh is left shifted by one bit and the value CFBF C9B8h is stored in register A3. The content of register A1 and A2 are unchanged and the functional unit used is .M1

	Before executions			After execution		
A1	F0233344	13124	A1	F0233344		16 LSB value
A2	87874A81	-30841	A2	87874A81		16 MSB value
A3	00007689		A3	CFBFC9B8	-809514568	

MPY .M1 14,A1,A2 - Signed 5-bit constant multiply instruction on 16 LSBs of register. The signed 5-bit number in the instruction is multiplied with 16 LSBs of registers A1 and the result is stored in register A2. If the content of register A1 is 2131 3344h, the 16 LSBs (3344h) and 14 (Eh) are multiplied and the result 0002 CDB8h is stored in register A2. The content of register A1is unchanged and the functional unit used is .M1

	Before executions		After execution
A1	21313344	A1	21313344
A2	48480044	A2	0002CDB8

14.3.6 Logical, Shift and Compare Instructions

Like other processors, ′C6X also supports logical operations; arithmetic and logical shift operations signed and unsigned integers compare operations. The list of logical, shift and compare operations of ′C6X are given in Table 14.11. The logical operations are performed by .L and .S units, the shift operations by .S units and compare operations by .L units of the CPU functional units. All the logical operations are bitwise operations that use only register addressing mode. The AND and OR operations support signed 5-bit constant for the source operand, the remaining 27 MSBs are sign extended. The shift and compare operations also use only register addressing mode.

The arithmetic shift supports both left and right shifting of register contents, but in logical shift instruction only right shift is possible. In both arithmetic and logical shift operations, when register is used the 6 LSBs specify the shift amount, the shift value is 0–40 and for immediate value given in the instruction the shift amount is 0–31 (5 bits). In the case of shift left with saturation instruction, the shift amount is 0–31 for both register and immediate types. If the shift amount is greater than 31 bits the result is saturated to 7FFF FFFFh. In all cases the shift value should be an unsigned number.

In the case of compare operations, signed and unsigned integers can be compared for equality, greater than and less than cases. The compare instructions support signed 5-bit constant for the source operand. If the comparison is true, 1 is written else 0 is written in to destination (***dst***) register.

Example 14.22 AND .L1 A1,A2,A3 - Bitwise AND operation instruction. The bitwise AND operation is performed between the contents of register A1 and A2. The result is placed in register A3. If signed 5-bit constant is used as operand, the sign is extended to 32 bits. If the content of register A1 is 7367 5454h, register A2 is 8282 7676h, the bitwise AND operation between the register contents 0202 5454h is loaded in register A3. The functional unit used is .L1, the register content A1 and A2 are unchanged.

	Before executions		After execution
A1	73675454	A1	73675454
A2	82827676	A2	82827676
A3	11224509	A3	02025454

Table 14.11 *Logical, Compare and Shift Instructions of 'C6X processor*

Instruction	*Functional unit*	*Description*
NOT	.L1 or.L2, .S1 or.S2	Bitwise NOT-Pseudo operation
AND	.L1 or.L2, .S1 or.S2	Bitwise AND - Pseudo operation
OR	.L1 or.L2, .S1 or.S2	Bitwise OR - Pseudo operation
NEG	.L1 or.L2, .S1 or.S2	Negate-Pseudo operation
SHL	.S1 or.S2	Arithmetic shift left
SHR	.S1 or.S2	Arithmetic shift right
SHRU	.S1 or.S2	Logical shift right
SSHL	.S1 or.S2	Shift left with saturation
CMPEQ	.L1 or.L2	Integer compare for equality
CMPGT/CMPGTU	.L1 or.L2	Signed/Unsigned integer compare for greater than
CMPLT/CMPLTU	.L1 or.L2	Signed/Unsigned integer compare for less than

Example 14.23 SHR .S2 B1,B2,B3 - Arithmetic shift right instruction. The content of register B1 is right shifted by n-bits specified in register B2, the result is stored in register B3. If the content of register B1 is 7367 5454h, register B2 is 0012h, the content of B1 is right shifted by the content of B2 (12h=18bits) times, the result is stored in register B3. The functional unit used is .S2, the register content B1 and B2 are unchanged.

	Before executions		After execution
B1	73675454	B1	73675454
B2	00000012	B2	00000012
B3	00020005	B3	00001CD9

Example 14.24 CMPGT .L1X A1,B2,A2 - Integer compare for greater than instruction. The contents of register A1 and B2 are compared for greater number. If register A1 content is greater than B2, the comparison is true, 1 is stored in register A2. If content of A1 is less than B2, the comparison is false, 0 is stored in register A2. If the content of register A1 is 7676h, register B2 is 5454h, the content of A1 is greater than B2. The comparison is true, 1 is set in register A2. The functional unit used is .L1 through cross path; the content of registers A1 and B2 are unchanged.

	Before executions		After execution
A1	00007676	A1	00007676
B2	00005454	B2	00005454
A2	00020005	A2	00000001

14.3.7 Branch and other Instructions

The 'C6X processor supports branch operations. The location to which the branch could occur is specified as label (displacement) in the branch instruction or a register in the register file or interrupt return pointer or NMI return pointer can hold it. The branch using displacement instruction is processed by .S1 and .S2 units. All other branch instructions are processed only by .S2 unit. The CLR, SET, EXT, EXTU, LMBD and ZERO instructions are used for accessing the bit fields of registers in register file. These instructions are used to set, clear or extract information about the bit fields. The ABS, NORM and SAT instructions are used for handing sign bits of the operands. The NOP instruction is used in programming to avoid conflicts between the functional units in the pipeline operation. The IDLE instruction is used to halt the processor operation until interrupt occurs. The list of ′C6X branch instructions and other instructions are given in Table 14.12.

Table 14.12 *Branch and other Instructions of ′C6X processor*

Instruction	*Functional unit*	*Description*
B	.S1 or.S2	Branch using displacement/using a register
B IRP/B NRP	.S2	Branch using an interrupt return pointer/ using NMI return pointer
CLR	.S1 or.S2	Clear a bit field
SET	.L1 or.L2	Set a bit field
EXT	.S1 or.S2	Extract and sign-extend a bit field
EXTU	.S1 or.S2	Extract and zero-extend a bit field
LMBD	.L1 or.L2	Leftmost bit detection
ZERO	L1 or.L2, .S1 or.S2, .D1 or .D2	Zero a register (Pseudo-operation)
ABS	.L1 or.L2	Integer absolute value with saturation
NORM	.L1 or.L2	Normalize integer
SAT	.L1 or.L2	Saturate a 40-bit integer to a 32-bit integer
NOP	—	No operation
IDLE	—	Multi-cycle NOP with no termination until interrupt

CONDITIONAL OPERATIONS 14.4

All the instructions in ′C6X can be conditional instructions. The content of registers A1, A2, B0, B1 and B2 are tested for conditional operations. These register contents equal to zero and non zero can be used as conditions. The conditional instructions are represented by square brackets, [ ], surrounding the register tested for condition. If register name alone is present in square bracket [A1] then the condition to be tested is register content being nonzero rather the exclamatory symbol [!A1] is used before the register in square bracket then the condition to be tested is register content being zero. The specified condition register is tested at the beginning of the E1 phase of the pipeline for all instructions. Example 14.25 shows how ′C6X conditional instructions can be written and how it can get executed.

PARALLEL OPERATIONS 14.5

In ′C6X, instruction are fetched eight times to form a fetch packet. The fetch packets are aligned 256 bits (8 words x 32-bits) and the basic format of the fetch packet is shown in Fig. 14.1. The execution of these eight instructions is controlled by the p-bit. The execution of an instruction in the fetch packet in parallel with another instruction is determined by scanning the *p*-bit, bit-0 of an instruction from left to right.

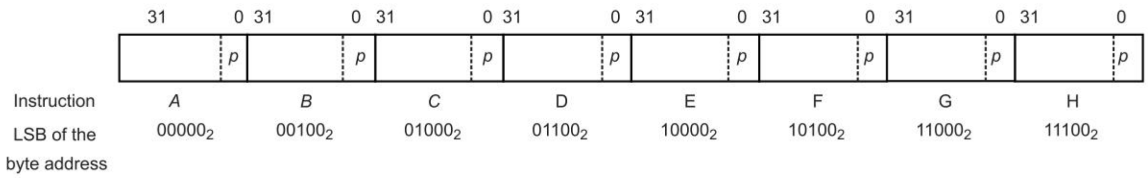


Fig. 14.1 *Basic Format of a Fetch packet*

If the *p*-bit of the instruction i is 1, then the next instruction *i*+1 is to be executed in parallel with instruction *i* in the same machine cycle. If the *p*-bit is zero, then the instruction *i*+1 is executed in the next machine cycle after the execution of instruction *i*. All the eight instructions executing in parallel constitute an execute packet. Each instruction in the execute packet must use a different functional unit. The execute packet cannot be more than eight words, so the last p-bit of the last instruction in a fetch packet is always set to 0. There are three types of execution of the instructions in the fetch packet based on the *p*-bits. They are

(i) Fully serial
(ii) Fully parallel
(iii) Partially serial

In fully serial type of execution, all the *p*-bits are set zero. The eight instructions of a fetch packet are executed serially one after the other in eight machine cycles. For a fully parallel type, all the *p*-bits are set 1 except the last instruction. All the eight instructions of a fetch packet are executed in parallel at the same machine cycle itself. In case of partially serial scheme, p-bit of some instructions are set zero and some with one. The instructions are executed serially one after the other from left to right of the fetch packet until the first *p*-bit with one is detected. Once the *p*-bit with one is detected, that instruction, the next instruction and the successive instructions who's *p*-bits with one are executed parallel until next *p*-bit zero is detected. If the p-bit with zero is sensed then the next instruction will be executed serially and so on. The instruction that is to be executed parallel is represented by the symbol || in the beginning of the opcode. The sample programs illustrating the above concept are given in example 14.25, 14.26 and 14.27.

Example 14.25 Fully serial execution with conditional operation. The following codes are to find the sum of *N* numbers. The instructions are executed one after the other.

The conditional operation is used for branch instruction. Register B0 is used to check for non zero condition. The count N is loaded register B0. The generation of the sequence is done in register A3, summation of *N* number is done in register B1 using ADD instruction. At the end of each summation the count *N* in B0 register is decremented using SUB instruction. The condition for non zero of B0 is checked each time using the representation [B0] and branching to location loop is performed until B0 content becomes zero. On zero of register content B0, execution comes out of the loop.

```
        MVK .S2 05h, B0          ; count N specified in register A1
LOOP    ADD .L1 1,A3,A3          ; generation of sequence in A3
         ADD .L2 A3,B1,B1        ; summation of sequence in A4
         SUB .S2 B0,1,B0         ; decrement of count N in register A1
        [B0] B .S1 LOOP          ; check for non zero of content A1, branch to loop
        NOP 5                    ; no operation 5 times to avoid conflict in pipeline
```

Example 14.26 Partially serial and parallel execution. The codes given in example 14.25 to find the sum on N numbers are modified, written for partially serial and parallel type and is given below. Loading the count value in register B0 is done serially and all other operations are executed in parallel. The parallel operations are performed in .L1, .L2, .S1 and .S2 units.

```
                  MVK .S2 05H, B0
LOOP1            ADD .L1 1,A3,A3
                 || ADD .L2 A3,B1,B1
                 || SUB .S2 B0,1,B0
                 || [B0] B .S1 LOOP1
                 NOP 5
                 NOP
                 NOP
                 NOP
```

Example 14.27 Fully parallel execution. The following codes are executed in parallel. All the eight functional units in ′C6X are used to perform fully parallel execution.

```
LDW .D1 *A4++,A3
|| STW .D2 B3,*B2++
|| ADD .L1 A4,A4,A5
|| ADD .L2 B4,B4,B5
|| MPY .M1 A5,A5,A6
|| MPYH .M2 B5,B5,B6
|| SUB .S1 A6,A5,A7
|| SUB .S2 B6,B5,B7
```

FLOATING POINT INSTRUCTIONS 14.6

The ′C67X floating point DSP supports all the fixed point instructions described in Section 14.3, but it has instructions that are specific to ′C67X. Instructions like 32-bit integer multiply, double word load and floating point addition, subtraction and multiplication are specific instructions for ′C67X processors. This topic describes about those specific instructions.

14.6.1 Data Formats

The ′C67X floating point DSPs support both fixed point and floating point data formats. For the fixed point case, the operands can be signed 32-bit integer values or unsigned 32-bit integer values. As for as the floating point operands are concerned, either Single-Precision (SP) or Double Precision (DP) floating point format can be used. Single-precision floating point operands are 32-bit values stored in a single register, whereas double-precision floating point operands are 64-bit values stored in register pairs (Section 13.5) present in the register file.

The fields of the single-precision floating point format operand are shown in Fig. 14.2. The LSBs 0-22 of the register represent the fraction (mantissa) part (23-bits), the bits 23-30 are used to represent the exponent part (8-bits) and the MSB bit (31st bit) is the sign bit. The floating point fields represent floating point numbers in two ranges, normalized (exponent field is between 0 to 255) and denormalized (exponent field is 0).

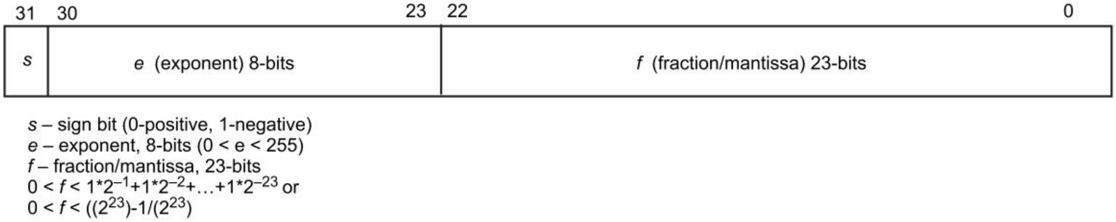

31
30
23
22
0
s
e (exponent) 8-bits
f (fraction/mantissa) 23-bits
s – sign bit (0-positive, 1-negative)
e – exponent, 8-bits (0 < e < 255)
f – fraction/mantissa, 23-bits
0 < f < 1*2^–1+1*2^–2+…+1*2^–23 or
0 < f < ((2^23)-1/(2^23)

Fig. 14.2 *Single-precision Floating point Fields*

The formula to translate the *s*, *e* and *f* fields into single precision floating point number is given below.

Normal range: $-1^{s} * 2^{(e\text{-}127)} * 1.f\ 0 < \mathrm{e} < 255$

Denormalized range: $-1^{s} * 2^{(-126)} * 0.f\ e = 0; f$ - nonzero

The fields of the double-precision floating point format operand are shown in Fig. 14.3. The full even register 32-bits and odd register LSBs 0-19 (20-bits) of the register pair represent fraction (mantissa) part (52-bits), the bits 20-30 of odd register are used to represent the exponent part (11-bits) and the MSB bit (31st bit) of odd register is the sign bit. In double precision format for normalized range the value of exponent is between 0 and 2047 and for denormalized range the value of exponent is 0.

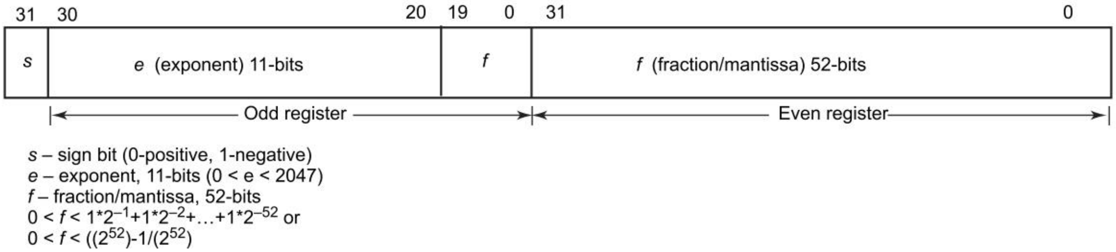

31
30
20
19
0
31
0
s
e (exponent) 11-bits
f
f (fraction/mantissa) 52-bits
Odd register
Even register
s – sign bit (0-positive, 1-negative)
e – exponent, 11-bits (0 < e < 2047)
f – fraction/mantissa, 52-bits
0 < f < 1*2^–1+1*2^–2+…+1*2^–52 or
0 < f < ((2^52)-1/(2^52)

Fig. 14.3 *Double-precision Floating point Fields*

The formula to translate the *s*, *e* and *f* fields into double precision floating point number is given below.

Normal range: $-1^{s} * 2^{(e\text{-}1023)} * 1.f\ 0 < \mathrm{e} < 2047$

Denormalized range: $-1^{s} * 2^{(-1022)} * 0.f\ \mathrm{e} = 0; f$ - nonzero

14.6.2 Data Format Conversion Instructions

The ′C67X processor supports fixed point, single-precision and double-precision floating point data formats. To convert one data format to another data format ′C67X processor supports various data format conversion instructions. The various data format conversion instructions of ′C67X processors are listed in Table 14.13. All the data format conversion instructions are processed by .L units (.L1 and .L2) except single-precision floating point to double precision floating point conversion instruction (SPDP) which is processed by .S units of the CPU. All the data format conversion instructions use register addressing mode.

Table 14.13 *Data Format Conversion Instructions of 'C67X processor*

Instruction	*Functional unit*	*Description*
INTSP/ INTSPU	.L1 or .L2	Convert signed /unsigned integer to single-precision floating point instruction
INTDP/ INTDPU	.L1 or .L2	Convert signed /unsigned integer to double-precision floating point instruction
SPINT	.L1 or .L2	Covert single-precision floating point value to integer instruction
SPTRUNC	.L1 or .L2	Covert single-precision floating point value to integer with truncation instruction
DPINT	.L1 or .L2	Covert double-precision floating point value to integer instruction
DPTRUNC	.L1 or .L2	Covert double-precision floating point value to integer with truncation instruction
SPDP	.S1 or .S2	Covert single-precision floating point value to double -precision floating point instruction
DPSP	.L1 or .L2	Covert double-precision floating point value to single-precision floating point instruction

(SP-Single-Precision, DP- Double-Precision, INT-Integer)

Example 14.28 INTSP .L1 A1,A2 - Signed integer to single-precision floating point conversion instruction. The signed integer content of register A1 is converted to single-precision floating point format and stored in register A2. If integer value content of register A1 is 00007272h (29298), its single precision floating point value 46E4 E400h (2.9298E+4) is loaded in register A2.The functional unit used is .L1 and the content of registers A1 is unchanged.

	Before executions			After execution	
A1	00007272	29298	A1	00007272	
A2	00020005		A2	46E4E400	2.9298E+4

14.6.3 Arithmetic Operation Instructions

The 'C67X processors has arithmetic instructions for single and double precision addition, subtraction and multiplication operations. Integer addition using double word addressing mode (ADDAD) is supported. It also has instructions to perform 32-bit integer multiplications, where the product can be obtained for 32-bits or 64-bits. The list of 'C67X arithmetic instructions are given in Table 14.14. All add and subtract instructions are processed by .L units except ADDAD instruction, which is processed by .D units. The multiply instructions are processed by .M units of the CPU. All the arithmetic instructions use register addressing mode except ADDAD instruction. In case of ADDAD instruction default is linear addressing mode, but if ***src2*** operand is one of the registers A4-A7 or B4-B7, then the mode is circular addressing mode. The ***src1*** operand is left shifted 3 times for double word addressing mode (refer ADDAB/ADDAH/ADDAW in Section14.2.3).

Table 14.14 *Arithmetic Operation Instructions of ′C67X processor*

Instruction	*Functional unit*	*Description*
ADDSP	.L1 or .L2	Single-precision floating point add instruction
ADDDP	.L1 or .L2	Double-precision floating point add instruction
ADDAD	.D1 or .D2	Integer addition using double word addressing mode instruction
SUBSP	.L1 or .L2	Single-precision floating point subtract instruction
SUBDP	.L1 or .L2	Double-precision floating point subtract instruction
MPYSP	.M1 or .M2	Single-precision floating point multiply instruction
MPYDP	.M1 or .M2	Double-precision floating point multiply instruction
MPYI	.M1 or .M2	32-bit integer multiply instruction(32 LSBs of the product is placed in destination)
MPYID	.M1 or .M2	32-bit integer multiply instruction(64-bits of the product is placed in destination register pair)

Example 14.29 ADDSP .L1 A1,A2,A3 – Single-precision floating point add instruction. The single-precision floating point content of register A1 and A2 are added; the result in single-precision floating point format is stored in register A3. If the floating point content of register A1 is 4370 0000h (2.4E+2), register A2 is C453 4000h (-8.45E+2), the added result C417 4000h (-6.05E+2) is stored in register A3. The functional unit used is .L1; the content of registers A1 and A2 are unchanged.

	Before executions			After execution	
A1	43700000	(2.40E+2)	A1	43700000	(2.40E+2)
A2	C4534000	(-8.45E+2)	A2	C4534000	(-8.45E+2)
A3	500F0D18		A3	C4174000	(-6.05E+2)

Example 14.30 SUBDP .L2 B1:B0,B3:B2,B5:B4 – Double-precision floating point subtract instruction. The double-precision floating point content of register pair B3:B2 is subtracted from the content of register pair B1:B0 and the result in double-precision floating point format is stored in register pair B5:B4. If the floating point content of register pair B3:B2 is 8.87634E+3, register pair B1:B0 content is -1.043567E+5, then B3:B2 content is subtracted from B1:B0 content and the result -1.1323304E+5 is stored in register pair B5:B4. The functional unit used is .L2; the content of registers pairs B1:B0 and B3:B2 are unchanged.

Before execution

B1:B0	C0F97A4B	33333333	(-1.043567E+5)
B3:B2	40C1562B	851EB852	(-8.8763E+3)
B5:B4	00000000	00000000	(0.0)

After execution

B1:B0	C0F97A4B	33333333	(-1.043567E+5)
B3:B2	40C1562B	851EB852	(-8.8763E+3)
B5:B4	C0FBA5C0	3D70A3D	(1.1323304E+5)

Example 14.31 MPYI .M1X A1,B1,A2 – 32-bit integer multiply instruction. The 32-bit integer content of register A1 and B1 are multiplied; the lower 32-bits of the product is stored in register A2. If the content of register A1 is 0008 5BDBh, register B1 is 000D B371h, the multiplied result is 72 8609 ACABh. The 32 LSBs of the product (8609 ACABh) alone are stored in register A2. The functional unit used is .M1 with the cross path; the content of registers A1 and B1 are unchanged. If the same operation is performed with MPYID instruction being register pairs used for destination, the entire result of the product is stored in register pairs.

	Before executions			After execution	
A1	00085BDB	547803	A1	00085BDB	547803
B1	C4534000	897905	B1	000DB371	897905
A2	00000000		A2	8609ACAB	491875052715

14.6.4 Compare and Reciprocal Approximation Instructions

The ′C67X processors has instructions for compare and reciprocal approximation operations. The comparisons are performed for equality, less than and greater than cases using single and double-precision floating point data formats. The ***src1*** and ***src2*** operands given in registers are compared, if the comparison case is true, ‘1’ is written in destination register else ‘0’ is written to destination register. The reciprocal and square-root reciprocal approximation operations are performed for single and double-precision floating point data formats. The list of ′C67X compare and reciprocal approximation instructions are given in Table 14.15. The compare and reciprocal approximation instructions are processed by .S1 and .S2 units and all these instructions use register addressing mode.

Table 14.15 *Compare and Reciprocal Approximation Instructions of ′C67X processor*

Instruction	*Functional unit*	*Description*
CMPEQSP	.S1 or .S2	Single-precision floating point compare for equality instruction
CMPEQDP	.S1 or .S2	Double-precision floating point compare for equality instruction
CMPLTSP	.S1 or .S2	Single-precision floating point compare for less than instruction
CMPLTDP	.S1 or .S2	Double-precision floating point compare for less than instruction
CMPGTSP	.S1 or .S2	Single-precision floating point compare for greater than instruction
CMPGTDP	.S1 or .S2	Double-precision floating point compare for greater than instruction
RCPSP	.S1 or .S2	Single-precision floating point reciprocal approximation instruction
RCPDP	.S1 or .S2	Double-precision floating point reciprocal approximation instruction
RSQRSP	.S1 or .S2	Single-precision floating point square-root reciprocal approximation instruction
RSQRDP	.S1 or .S2	Double-precision floating point square-root reciprocal approximation instruction

Example 14.32 CMPGTSP .S1 A3,A4,A5 – Single-precision floating point compare instruction for greater than case. The single-precision floating point content of register A3 and A4 are compared, if the content of register A3 is greater than the content of A4, then ‘1’ is written in register A5. If the floating point content of register A3 is 4608F59Ah (8.7654E+3), register A4 is 45F6 3E66h (7.8798E+3), the comparison is true; hence ‘1’ is written in register A5. The functional unit used is .S1; the content of registers A3 & A4 are unchanged.

	Before executions			After execution	
A3	4608F59A	(8.7654E+3)	A3	4608F59A	(8.7654E+3)
A4	45F63E66	(7.8798E+3)	A4	45F63E66	(7.8798E+3)
A5	00000000		A5	00000001	

Example 14.33 RSQRSP .S1 A1,A2 - Single-precision floating point square-root reciprocal approximation instruction. The square root of the single-precision floating point content of register A1 is obtained and its reciprocal value is stored in register A2 in single-precision floating point format. If the single-precision floating point content of register A1 is 4380 0000h (2.56E+2), it's square root value is 1.6E+1 and its reciprocal value 3D80 0000h (6.25E-2) is stored in register A2. The functional unit used is .S1; the content of registers A1 is unchanged.

	Before executions			After execution	
A1	43800000	(2.56E+2)	A1	43800000	(2.56E+2)
A2	C4534000	(-8.45E+2)	A2	3D800000	(6.25E-2)

14.6.5 Other Instructions

The ′67X processor supports finding absolute value of single and double-precision floating point numbers. The absolute value of source register/register pair content is stored in destination register/register pairs. The functional units used are .S1 and .S2; addressing mode used is register addressing. The ′C67X processors also support loading double word from memory with unsigned constant offset or register offset (refer chapter 14.3.2). The absolute and load double word instructions of ′C67X are given in Table 14.15. The type of addressing mode used for load double word instruction is linear addressing; the functional units used are .D1 and .D2 through LD1b and LD2b 32-bit MSB buses in ′C67X (refer Fig. 13.5).

Table 14.15 *Absolute and Load Double word Instructions of ′C67X processor*

Instruction	*Functional unit*	*Description*
ABSSP	.S1 or .S2	Absolute value of single-precision floating point number
ABSDP	.S1 or .S2	Absolute value of double-precision floating point number
LDDW	.D1 or .D2	Load double word from memory with an unsigned constant offset or register offset

PIPELINE OPERATION 14.7

The ′C6X pipeline operation provides easy way of programming and improves performance. The major phases of ′C6X pipeline

- Fetch
- Decode
- Execute

All instructions require same number of pipeline phases for fetch and decode, but require a varying number of execute phases depending on the type of instruction. The ′C62X/′C64X fixed point processors require less execution phases than the ‘C67X floating point processor. The fetch operation consists of four phases and the decode operation has two phases for all ′C6X processors. But the execute operation of fixed point processors have five phases where as it is ten phases for floating point processors. The ′C6X fixed point and floating point processor pipeline stages are show in Figs 14.4 and 14.5.

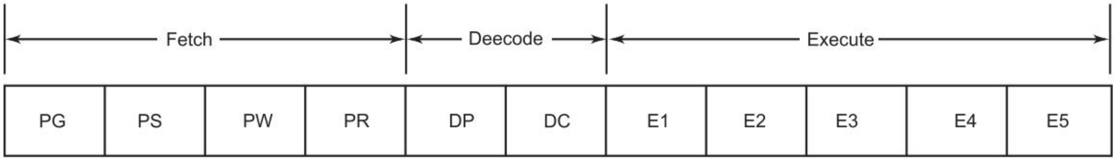


Fig. 14.4 *Fixed point processor (′C62X/′C64X) pipeline stages*

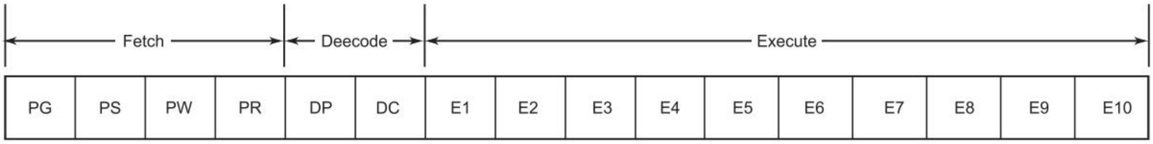


Fig. 14.5 *Floating point processor (′C67X) pipeline stages*

14.7.1 Fetch Operation

The ′C6X processor uses eight instructions in a fetch packet (FP). The eight instructions are fetched from memory through four phases. The fetch phase is subdivided into the following phases.

- Program address generate (PG)
- Program address send (PS)
- Program access ready wait (PW)
- Program fetch packet receive (PR)

Figure 14.6 shows the functional block diagram of fetch phase. During the PG phase the memory addresses corresponding to eight instructions of fetch packet are generated. In PS phase, the addresses are sent to memory and in PW phase the memory read operation is performed. Finally, in PR phase the eight instructions are received at the CPU. The number of execute packets in the fetch packet is based upon instructions written in fully serial, fully parallel and partially serial execution types. If eight instructions of a fetch packet are serial, there are eight execute packets, where as eight instructions are in parallel, there is only one execute packet. In case of partial serial type, the number of execute packets

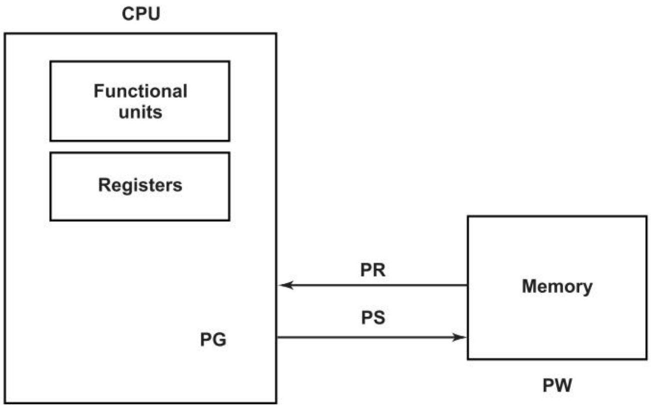


Fig. 14.6 *Functional Block Diagram of ′C6X Fetch phases*

varies between two to seven and depend on the number of instructions that are parallel in the fetch packet.

14.7.2 Decode Operation

In decode phase the fetch packet having eight instructions are split into execute packets, assigned to appropriate functional units and are decoded. The decode phase is subdivided into the following two phases

- Instruction dispatch (DP)
- Instruction decode (DC)

The execute packet consists of one instruction or two to eight parallel instructions. In instruction dispatch phase (DP), the instructions in an execute packet are assigned to the appropriate functional units. In instruction decode phase (DC), the source registers, destination registers and associated data paths are decoded for the execution of the instructions in the eight functional units.

14.7.3 Execute Operation

The execute phase of the pipeline for fixed-point processor is subdivided into five phases (E1-E5) and for floating point processors it is subdivide into ten phases (E1-E10). Different type of instructions require different numbers of execute phases to complete the execution. The execute phases and the operation performed in each phase for fixed point processors are given in Table 14.16 and the same for floating point processors are given in Table 14.17. The pipeline operations of fixed-point processors are categorized into seven instruction types. They are single-cycle, single 16x16 multiply (Two-cycle) and ‘C64X non-multiply, store, ‘C64X extended multiply, load, branch and no operation (NOP) instructions. The pipeline operations of floating-point processors are categorized into fourteen instruction types. They are single-cycle, single 16x16 multiply, store, load, branch, 2-cycle DP, 4-cycle, INTDP, DP compare, ADDDP/SUBDP, MPYI, MPYID, MPYDP and no operation (NOP) instructions. The execute phase in which these instruction categories are executed are shown in Tables 14.16 and 14.17.

INTERRUPTS 14.8

The ′C6X processors have three types of interrupts based on their priorities. First the reset interrupt ($\overline{\text{RESET}}$) which has the highest priority, second the nonmaskable interrupt (NMI) having the second highest priority and third are the twelve makeable interrupts INT4-INT15 having lowest priorities. In ′C6X, eight registers are present that control servicing the interrupts. The list of interrupts and their functions are given in Table 14.18.

The reset interrupt is an active low signal and all other interrupts are active high signal. The reset interrupt must be held low for 10 clock cycles. The nonmaskable interrupt is used to alert the CPU for serious hardware problem such as power failures likely to happen immediately. The twelve maskable interrupts are associated with external devices, on-chip peripherals, software control or in some processor not be available. The ′C6X processors have interrupt acknowledgement signal (IACK) to alert the external hardware that an interrupt has occurred and is being processed and INUMx signals (INUM3-INUM0) to indicate the number of interrupts that is being processed.

When an interrupt occur, the CPU begins to process it and it references interrupt service table (IST). IST is a table containing codes for servicing the interrupts. The IST contains 16 consecutive fetch packets, where each fetch packet contains eight instructions. Instructions of ′C6X is 32-bits, so for eight instructions it occupies 32-bytes of program memory locations for each fetch packet. Hence the address of IST is incremented by 32 bytes (20h) for the next interrupt to be serviced. The interrupt service routine can fit in with these eight instructions.

Table 14.16 *Operation performed in execute phases of ′C6X fixed-point processors*

Execute Phase	*Type*	*Operations performed*
E1	Conditional Instructions	For all instructions, the conditions for the instructions are checked and operands are read.
	Load and store instructions	Address generation is performed and address modifications are written to a register file
	Branch Instructions	Branch fetch packet in PG phase is affected
	Single-cycle instructions	The results are written to a register file
E2	Load instructions	The address is sent to memory
	Store instructions	The address and the data are sent to memory
	Single-cycle instructions	Single-cycle instructions with saturate results, if saturation occurs, set the SAT bit in the control status register (CSR)
	Multiply instructions	For 16x16 multiply instructions, results are written to a register file. In ‘C64X multiply unit, for the non-multiply instructions, results are written to a register file
E3	Store instructions	Data memory accesses are performed
	Multiply instructions	Multiply instructions with saturate results, if saturation occurs, sets the SAT bit in the control status register (CSR)
E4	Load instructions	Data is brought to the CPU boundary
	Multiply instructions	In ‘C64X multiply extensions, results are written to a register file
E5	Load Instructions	The data is written into a register

Table 14.17 *Operation performed in execute phases of ′C6X floating-point processors*

Execute Phase	*Type*	*Operations performed*
E1	Conditional Instructions	For all instructions, the conditions for the instructions are checked and operands are read.
	Load and store instructions	Address generation is performed and address modifications are written to a register file

(Contd.)

Table 14.17 *(Contd.)*

	Branch Instructions	Branch fetch packet in PG phase is affected
	Single-cycle instructions	The results are written to a register file
	DP compare, ADDDP/SUBDP and MPYDP instructions	The lower 32-bits of the source are read. For all other instructions, the source are read
	2-cycle DP instructions	The lower 32-bits of the result are written to a register file
E2	Load instructions	The address is sent to memory
	Store instructions	The address and the data are sent to memory
	Single-cycle instructions	Single-cycle instructions with saturate results, if saturation occurs, set the SAT bit in the control status register (CSR)
	Multiply, 2-cycle DP and DP compare instructions	Results are written to a register file
	DP compare and ADDDP/SUBDP instructions	The upper 32-bits of the source are read
	MPYDP instruction	The lower 32-bits of ***src1*** and the upper 32-bits of ***src2*** are read
	MPYI and MPYID instruction	The sources are read
E3	Store instructions	Data memory accesses are performed
	Multiply instructions	Multiply instructions with saturate results, if saturation occurs, sets the SAT bit in the control status register (CSR)
	MPYDP instruction	The upper 32-bits of ***src1*** and the lower 32-bits of ***src2*** are read
	MPYI and MPYID instruction	The sources are read
E4	Load instructions	Data is brought to the CPU boundary
	MPYI and MPYID instruction	The sources are read
	MPYDP instruction	The upper 32-bits of the source are read
	4-cycle instructions	Results are written to register file
	INTDP instruction	The lower 32-bits of the result are written to a register file
E5	Load Instructions	The data is written into a register
	INTDP instruction	The upper 32-bits of the result are written to a register file
E6	ADDDP/SUBDP instructions	The lower 32-bits of the result are written to a register file
E7	ADDDP/SUBDP instructions	The upper 32-bits of the result are written to a register file
E8	—-	Nothing read or written
E9	MPYI instruction	The result is written to a register file
	MPYDP and MPYID instructions	The lower 32-bits of the result are written to a register file
E10	MPYDP and MPYID instructions	The upper 32-bits of the result are written to a register file

If the interrupt service routine for an interrupt is larger than eight instructions that cannot fit in the IST, an interrupt service fetch packet (ISFP) is used to service an interrupt. The interrupt service fetch packet contains a branch to the interrupt return pointer instruction (B IRP) followed by five no operations (NOP 5) for the branch to reach the execution stage of the pipeline. The additional interrupt service routine code is written from the branched memory location. In both IST and ISFP the interrupt service table pointer (ISTP) register is used to locate the interrupt service routine.

Table 14.18 *Interrupt Control Register and their Functions*

Name of the register	*Abbreviation*	*Functions*
Control status register	CSR	To globally set or disable the maskable interrupts
Interrupt enable register	IER	To enable the makableinterrupts
Interrupt flag register	IFR	Shows the status of interrupts
Interrupt set register	ISR	To set the flags in IFR register manually
Interrupt clear register	ICR	To clear the flags in IFR register manually
Interrupt service table pointer	ISTP	Pointer to the beginning of the interrupt service table
Nonmaskable interrupt return pointer	NRP	Contains the return address used on return from a nonmaskable interrupt. This is accomplished using the B NRP instruction
Interrupt return pointer	IRP	Contains the return address used on return from a maskable interrupt. This is accomplished using the B IRP instruction

To process a maskable interrupt the following conditions are to be satisfied.

- The global interrupt enable bit (GIE) in the control status register is set to 1
- The NMIE bit in the interrupt enable register (IER) is set to 1
- The interrupt enable bit (IE) in the interrupt enable register (IER) for the corresponding interrupt is set to 1

On the above conditions satisfied when an interrupt occurs, the corresponding bit in interrupt flag register (IFR) is set. Based on the priority of the interrupt, the interrupt service table pointer (ISTP) locates the interrupt service routine and the interrupt is processed.

Review Questions

14.1 What are the types of operations performed by .L functional units?

14.2 List the various types of multiply operations performed by .M functional units.

14.3 Which unit is used to process the branch instructions? List the various types of branch instructions in ′C6X.

14.4 What are the various types of load and store operations performed by .D units of ′C6X processor?

14.5 List the addressing modes supported by the ′C6X processor.

14.6 What are the address generation options present in linear addressing mode?

14.7 Explain the operation of circular addressing mode with example.

14.8 What are the various types of move instructions in ′C6X processors?

14.9 List the various types of addition and subtract instructions in ′C6X processors.

14.10 What are the various shift and compare operations supported by ′C6X processors?

14.11 Explain how logical conditional can be defined in ′C6X instructions?

14.12 What are the various instruction execution types in ′C6X? Explain.

14.13 What are the various data formats supported by the ‘C67X processors?

14.14 List the various data format conversion instructions in ‘C67X processors.

14.15 What are the floating point arithmetic operations ‘C67X processor supports?

14.16 Explain the different phases of fetch operation of ′C6X pipeline.

14.17 What are operations performed in decode phase of ‘C67X pipeline.

14.18 List the categories of ′C6X fixed point processor pipeline execute phases

14.19 What are the categories of ‘C67X processor pipeline execute phases?

14.20 List the register present in ′C6X processor to process the interrupts.

14.21 What are the registers in ′C6X register file used for conditional operations?

Self Test Questions

14.1 Arithmetic operations are performed by ——— units.
(a) .L (b) .S (c) . M (d) .L, .S and .D

14.2 The logical operations are processed using ——— units.
(a) .L (b) .D (c) . M (d) .L and .S

14.3 Shift operations are processed by ——— units.
(a) .L (b) .D (c) . M (d) .S

14.4 Compare operations are processed by ——— units.
(a) .L (b) .D (c) . M (d) .S

14.5 ——— units are used to perform move operations
(a) .L (b) .D (c) .S (d) .S and .D

14.6 Multiply operations are processed by ——— units.
(a) .L (b) .D (c) . M (d) .S

14.7 Branch operations are processed by ——— units.
(a) .L (b) .D (c) . M (d) .S

14.8 ——— units are used to perform load and store operations.
(a) .L (b) .D (c) . M (d) .S

14.9 In linear addressing mode the number of address generation options is
(a) 4 (b) 10 (c) 6 (d) 8

14.10 For circular addressing registers used are ———.
(a) A0-A15 (b) B0-B15
(c) A0-A32 (d) A4-A7 and B4-B7

14.11 Instruction to move values between control register file and register file is
(a) MV (b) MVK (c) MVC (d) MVKH.

14.12 Instruction to perform signed 16-bit constant addition operation is ———.
(a) ADD (b) ADDU (c) ADD2 (d) ADDK.

14.13 Instruction to perform two 16-bit addition on upper and lower register halves is ———
(a) ADD (b) ADDU (c) ADD2 (d) ADDK.

14.14 ——— instruction is used to perform division operation.
(a) ADD (b) SUBC (c) SUB (d) SUBU

14.15 For fully parallel type of execution the P-bit set for ——— .
(a) all the eight instructions (b) the fist instruction
(c) the last instruction (d) fist seven instructions

14.16 The condition checked for [!A1] is ———
(a) content of register A1 being non zero
(b) content of register A1 being zero
(c) content of register A1 being negative
(d) content of register A1 being positive

14.17 Integer addition using double word addressing mode instruction is in ——— processor.
(a) ‘C62X (b) ‘C64X
(c) ‘C62X and ‘C64X (d) ‘C67X

14.18 32-bit integer multiply instruction is in ——— processor
(a) ‘C62X (b) ‘C64X
(c) ‘C62X and ‘C64X (d) ‘C67X.

14.19 Reciprocal approximation operations are processed by ——— units
(a) .L (b) .D (c) . M (d) .S

14.20 The number of phases of ′C62X fixed point processor pipeline is ———.
(a) 5 (b) 10 (c) 11 (d) 16

14.21 The number of phases of ′C67X floating point processor pipeline is ———.
(a) 5 (b) 10 (c) 11 (d) 16

15 TMS320C6X APPLICATION PROGRAMS AND PERIPHERALS

In this chapter some application programs for TMS320C6X processors, some details on memory and on-chip peripheral are given. To develop and test ′C6X application codes, programming tools and hardware accessories are needed. The programming tool used is Code Composer Studio (CCS) and TMS320C6X starter kits are used for the implementation. The details about the internal memory resources and the various peripherals like timers, multichannel buffered serial ports, DMA controllers and external memory interface are discussed.

CODE COMPOSER STUDIO (CCS) 15.1

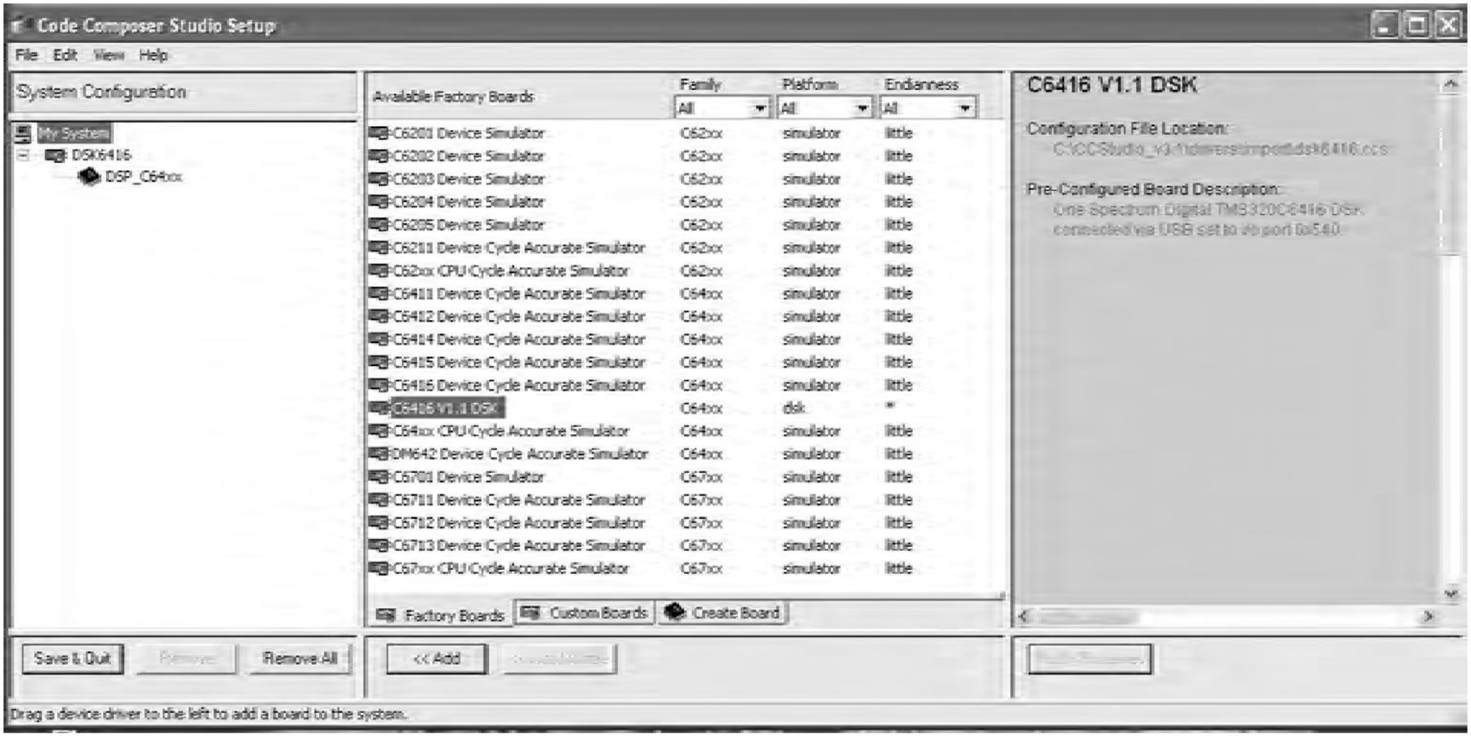
Code composer studio has the basic code generation tools with set of debugging and real-time analysis capabilities. The code composer studio is available in integrate development environment (IDE), which is designed to edit, build and debug ′C6X processor target programs. The steps to do programming in ′C6X processor environment are:

- Setting up the target processor
- Code generation
- Debugging and execution of codes

15.1.1 Setting up the Target Processor

The code composer studio tool of ′C6X platform supports code generation for ′C62X, ′C64X fixed point and ′C67X floating point processors. The CCS tool supports processor simulator mode of operation where there is no target ′C6X processor present. The other modes of operation are using starter kit (DSK) or evaluation module (EVM) in which a particular target ′C6X processor present in the board can be selected. The selection of simulator mode or starter kit or EVM for specific target ′C6X processor can be programmed using the setup option in the CCS tool.

The setup menu of the ′C6X code composer studio is shown in Fig. 15.1. The ′C62X, ′C64X, ′C67X devices supported by the code composer studio tool are listed in the second window. Based on the mode, simulator or DSK or EVM and a device of a family can be selected. The selected device is updated in system configuration window and the details about the selected device are available in third window. After proper selection of the mode and device click the save and quit button at the left bottom corner of the setup window. This completes the setup process of target processor for ′C6X code generation. In this



Available Factory Boards

Available Factory Boards	Family	Platform	Endianness
	All	All	All
C6201 Device Simulator	C62xx	simulator	little
C6202 Device Simulator	C62xx	simulator	little
C6203 Device Simulator	C62xx	simulator	little
C6204 Device Simulator	C62xx	simulator	little
C6205 Device Simulator	C62xx	simulator	little
C6211 Device Cycle Accurate Simulator	C62xx	simulator	little
C62xx CPU Cycle Accurate Simulator	C62xx	simulator	little
C6411 Device Cycle Accurate Simulator	C64xx	simulator	little
C6412 Device Cycle Accurate Simulator	C64xx	simulator	little
C6414 Device Cycle Accurate Simulator	C64xx	simulator	little
C6415 Device Cycle Accurate Simulator	C64xx	simulator	little
C6416 Device Cycle Accurate Simulator	C64xx	simulator	little
C6416 V1.1 DSK	C64xx	dsk	*
C64xx CPU Cycle Accurate Simulator	C64xx	simulator	little
DM642 Device Cycle Accurate Simulator	C64xx	simulator	little
C6701 Device Simulator	C67xx	simulator	little
C6711 Device Cycle Accurate Simulator	C67xx	simulator	little
C6712 Device Cycle Accurate Simulator	C67xx	simulator	little
C6713 Device Cycle Accurate Simulator	C67xx	simulator	little
C67xx CPU Cycle Accurate Simulator	C67xx	simulator	little

C6416 V1.1 DSK

Configuration File Location:
C:\CCStudio_v3.1\drivers\import\dsk6416.ccs

Pre-Configured Board Description:
One Spectrum Digital TMS320C6416 DSK connected via USB set to i/o port 0x540

Drag a device driver to the left to add a board to the system.

Fig. 15.1 *Code composer studio setup for 'C6416 DSK*

chapter the code generation, debugging and execution are carried out using both TMS320C6416 DSK (Starter kit, operating at 720MHz) and TMS320C6713 DSK.

15.1.2 An Overview of the ′C6416 DSK

The ′C6416 Starter kit is a standalone board consisting of the following features. The code composer studio tool present in the host computer communicates with the DSK through an embedded JTAG emulator with a USB host interface.

- TMS320C6416T processor operating at 1GHz
- 16 Mbytes of synchronous DRAM connected to CE0 space of EMIA
- 512 Kbytes of non-volatile Flash memory connected to CE1 space of EMIB
- Software board configuration through registers implemented in CPLD of CE0 space of EMIB
- An AIC23 stereo codec connected to McBSP
- Configured boot options and clock input selection
- External memory, External peripheral and PCI/HPI connectors
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator

The ′C6416 starter kit has TMS320C6416T processor with 1024 K (1M) bytes of the on-chip RAM as unified memory space in the address range 0000 0000h to 000F FFFFh. This space can be used to store the program codes as well as data values. For more memory space applications, the external 16M bytes of DRAM connected to CE0 space of the starter kit can be used. The 512 K bytes of external Flash memory in CE1 space can be used for boot option. The CPLD is used to implement simple logic functions without additional discrete devices. The AIC23 stereo codec is used for input and output audio signals through multi-channel buffered serial port of the processor (McBSP).

15.1.3 Code Generation in CCS

The CCS tool supports ′C6X code generation both in assembly and ‘C’ language. The code generation flow in assembly language is explained with an example in this section. In this example is used to generate an arithmetic series of N numbers, find the summation of the series, the series and the sum to be stored in memory. Invoke the code composer studio tool using the shortcut on the desktop or from the program menu. The code composer studio window will open; the following are the steps to create a new project and build assembly language code.

Step 1: **Creating a new project:** Select Project menu - New project option in CCS, a project selection window will appear. A project name (e.g. series) is to be entered. The default location of the new project being created is in the folder –Code composer studio – Myprojects. Using the browse option, the folder for the new project creation can be altered. A folder in the name of the project will be created and a file in the name of project with the file extension .pjt (series.pjt) will be present in that folder. In the files window of CCS the project name appears.

Step 2: **Creating source file in assembly:** Select File-New - Source file option of CCS. A text edit window will appear; the assembly language code can be entered in the text window. The procedure to enter codes in assembly is common for all the processors (Section 6.1.4). The complete assembly language code for series generation is given in example 15.1. Once the complete code is entered in the text pad, use the save option or shortcut keys of the CCS to save the file in the project folder created (series). Save option window will appear, in which enter the file name with file extension .asm (sumn.asm).

Step 3: **Adding file to the project:** The assembly language file created is to be added to the project. Select Project menu – Add files to the project option in CCS, add files to project window will appear in which select the file type as 'Asm source files, select the asm file in the project folder (sumn.asm). In the files window click the project name, then source folder, added .asm file will appear.

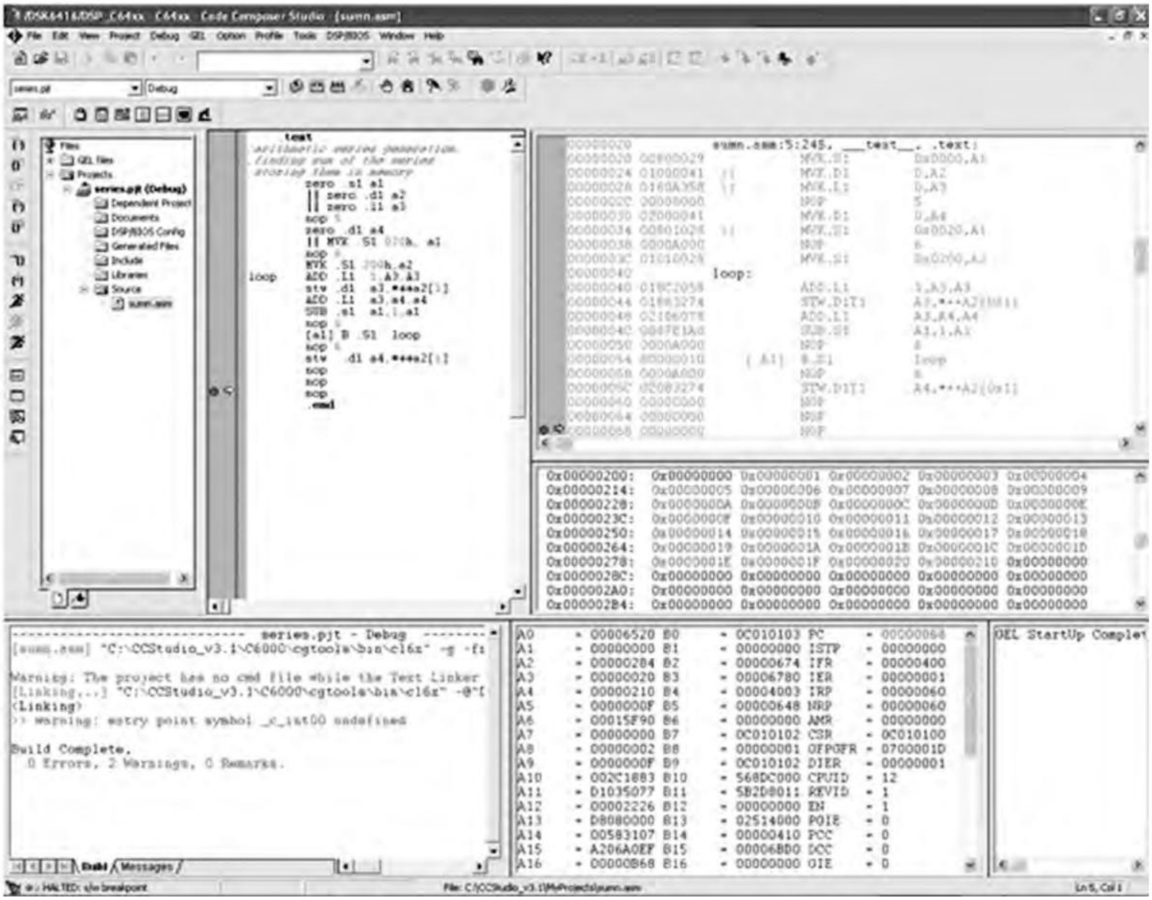
Example 15.1 The assembly code generates the arithmetic series, finds the sum of the series and stores it in memory. The first few instructions initialize the content of registers used to zero. The register A1 specifies the number of values of the series N (20h); register A2 specifies the start address of the memory (0200h) to store the series. The series is generated in register A3, the sum of the series values are accumulated in register A4. The N values of the series are stored in N words (4 bytes) of the memory starting from the next address in register A2 (0204h) and N+1th word the sum of the series is stored. The content of register A1 is used for conditional operation.

```
Label          Mnemonic                Comments
               .text                   ; assemble directive to initialize the program section
                                       (case sensitive)
               ZERO .S1 A1             ; zero the content of registers A1, A2, A3 done in parallel
               || ZERO .D1 A2
               || ZERO .L1 A3
               NOP 5
               ZERO .D1 A4             ;zero the content of register A4, the no. of series-
               || MVK .S1 020h, A1     ;values (20h) entered in A1, done in parallel
               NOP 6
               MVK .S1 200h,A2         ; the start address of the memory (0200h) to store the-
                                       ; sequence is loaded in A2
LOOP           ADD .L1 1,A3,A3         ; the series generation done in A3
               STW .D1 A3,*++A2[1]     ; the values of the series are stored
               ADD .L1 A3,A4,A4        ; the sum of the series is done in A4
               SUB .S1 A1,1,A1          ; decrement the count N
               NOP 6
               [A1] B .S1 LOOP          ; the content of A1 being nonzero condition is tested
               NOP 6
               STW .D1 A4,*++A2[1]     ; the sum of the series is stored in N+1th location
               NOP
               .end                    ;assembler directive to specify the end of section
                                       (case sensitive)
```

Step 4: **Building the code:** Select Project menu – Build option in CCS, a Debug window will appear in CCS. It checks the syntax of the ′C6X assembly code. If the build is successful an .out file in the name of the project is created in Debug folder of the project folder (series.out) else error messages will appear in the debug window. By reading the error messages the correct syntax can be written in the assembly language file. The build option is to be continued till the end of successful build.

Step 5: **Down loading the code in target processor:** Now the .out file is to be down loaded to the on-chip memory of the target processor, for this first the target processor is to be connected. Select Debug menu – Connect option in CCS, a Disassembly window will open in CCS. To load the .out file, Select File menu-Load program option in CCS. A Load program window will popup, click the Debug folder and select the .out file in the name of the project (series.out).

The disassembly window will point the starting address 0000 0000h or the default location of the program counter (PC). The assembly codes developed by the user will be loaded from the starting address 00000020h. The complete assembly codes downloaded can be viewed in the disassembly window from this address.



```
        .text
;arithmetic series generation
;finding sum of the series
;storing them in memory
        zero .s1 a1
        || zero .d1 a2
        || zero .l1 a3
        nop 5
        zero .d1 a4
        || MVK .S1 020h, a1
        nop 6
        MVK .S1 200h,a2
loop    ADD .L1 1,A3,A3
        stw .d1 a3,*++a2[1]
        ADD .L1 a3,a4,a4
        SUB .s1 a1,1,a1
        nop 5
        [a1] B .S1 loop
        nop 6
        stw  .d1 a4,*++a2[1]
        nop
        nop
        nop
        .end
```

```
00000020          sumn.asm:5:245, ___text__, .text:
00000020 00800029          MVK.S1     0x0000,A1
00000024 01000041 ||       MVK.D1     0,A2
00000028 0160A358 ||       MVK.L1     0,A3
0000002C 00008000          NOP        5
00000030 02000041          MVK.D1     0,A4
00000034 00801028 ||       MVK.S1     0x0020,A1
00000038 0000A000          NOP        6
0000003C 01010028          MVK.S1     0x0200,A2
00000040          loop:
00000040 018C2058          ADD.L1     1,A3,A3
00000044 01883274          STW.D1T1   A3,*++A2[0x1]
00000048 02106078          ADD.L1     A3,A4,A4
0000004C 0087E1A0          SUB.S1     A1,1,A1
00000050 0000A000          NOP        5
00000054 80000010   [ A1]  B.S1       loop
00000058 0000A000          NOP        6
0000005C 02083274          STW.D1T1   A4,*++A2[0x1]
00000060 00000000          NOP
00000064 00000000          NOP
00000068 00000000          NOP
```

```
0x00000200:  0x00000000 0x00000001 0x00000002 0x00000003 0x00000004
0x00000214:  0x00000005 0x00000006 0x00000007 0x00000008 0x00000009
0x00000228:  0x0000000A 0x0000000B 0x0000000C 0x0000000D 0x0000000E
0x0000023C:  0x0000000F 0x00000010 0x00000011 0x00000012 0x00000013
0x00000250:  0x00000014 0x00000015 0x00000016 0x00000017 0x00000018
0x00000264:  0x00000019 0x0000001A 0x0000001B 0x0000001C 0x0000001D
0x00000278:  0x0000001E 0x0000001F 0x00000020 0x00000210 0x00000000
0x0000028C:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000002A0:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x000002B4:  0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
```

```
------------------------------  series.pjt - Debug  ---------
[sumn.asm] "C:\CCStudio_v3.1\C6000\cgtools\bin\cl6x" -g -fs
Warning: The project has no cmd file while the Text Linker
[Linking...] "C:\CCStudio_v3.1\C6000\cgtools\bin\cl6x" -@"[
<Linking>
>> warning: entry point symbol _c_int00 undefined

Build Complete,
  0 Errors, 2 Warnings, 0 Remarks.
```

```
A0    = 00006520 B0    = 0C010103 PC     = 00000068
A1    = 00000000 B1    = 00000000 ISTP   = 00000000
A2    = 00000284 B2    = 00000674 IFR    = 00000400
A3    = 00000020 B3    = 00006780 IER    = 00000001
A4    = 00000210 B4    = 00004003 IRP    = 00000060
A5    = 0000000F B5    = 00000648 NRP    = 00000060
A6    = 00015F90 B6    = 00000000 AMR    = 00000000
A7    = 00000000 B7    = 0C010102 CSR    = 0C010100
A8    = 00000002 B8    = 00000001 OFPGFR = 0700001D
A9    = 0000000F B9    = 0C010102 DIER   = 00000001
A10   = 002C1883 B10   = 568DC000 CPUID  = 12
A11   = D1035077 B11   = 5B2D8011 REVID  = 1
A12   = 00002226 B12   = 00000000 EN     = 1
A13   = D8080000 B13   = 02514000 POIE   = 0
A14   = 00583107 B14   = 00000410 PCC    = 0
A15   = A206A0EF B15   = 00006BD0 DCC    = 0
A16   = 00000B68 B16   = 00000000 GIE    = 0
```

```
GEL StartUp Complet
```

Fig. 15.2 *Various windows of CCS for ′C6416*

15.1.4 Execution of ′C6X Codes in Target Processor

The assembly codes which are loaded to the target processor are to be executed and the results can be verified in the CPU register and memory of the processor. To view the results, the register window and memory widows are to be enabled. Select View – Registers – Core registers option in CCS. A new window appears in CCS, in which the content of all the CPU core registers can be viewed. As the same way select View – Memory option in CCS, a new small window will appear for options. Select the address of the memory that is to be viewed (e.g. 0x00000200) and the format in which the data is to be displayed (e.g. 32 Bit Hex – C style), a memory window appears in CCS. In the memory window, click the right button of the mouse; choose Float in main window option to view the memory window along with disassembly window in CCS.

To start execution, the program counter (PC) should point to the starting address of the code. This is being done in two ways; double click the PC in register window, the edit register window will appear, in which enter the start address (0x00000020). The other ways is either in text edit window or in disassembly window, keep the cursor in the first line of the code, right click the mouse button and select Set PC to cursor option or use shortcuts options.

To execute the code, select Debug-Step into option in CCS or shortcut key, the code will be executed line by line. Breakpoint can be introduced using the option in debug menu to the end address of the code or double click the cursor at the lost line of the code in disassembly window. The break points can be introduced to any line of the code and also 'n' number of such break points can be introduced in the program. The 'run' option in debug menu can be used to execute the code in one step. The arithmetic series values and the sum can be viewed in the memory window. The various windows of the CCS along with the result in memory window are shown in Fig. 15.2.

APPLICATION PROGRAMS IN ′C64X 15.2

The assembly language programs for various functions such as convolution, discrete Fourier transform, FIR filter and real time audio signal capture are implemented in TMS320C6416 starter kit. The details about the implementation are given in this section.

15.2.1 Integer Division

The integer division operation is performed using SUBC instruction in ′C6X processor more efficiently. The division operation using SUBC needs the denominator content to be aligned to the numerator content. This can be performed by detecting the left most '1' bit in the denominator using LMBD instruction of ′C6X. The TMS320C6X assembly program to perform unsigned integer division is given in Program 15.1 and signed integer in Program 15.2. The numerator and denominator are stored in two CPU registers (A2 and A3) where denominator must be less than the numerator. Using LMBD instruction, both in numerator and denominators, the left most '1' bit is detected and the result is stored in two new registers (A5 and A6). The difference value of the left most '1' bit detection of the denominator to that of the numerator, say X is the critical value used in the division process. The denominator content is left shifted X bits to align to numerator content. The content of the aligned denominator content is subtracted from numerator using SUBC instruction. It is important to note that X+1 time the SUBC instruction is to be executed to complete the division. After division, both quotient and the remainder will be in a single register in which the numerator is loaded (A2). The quotient of the division can be computed by taking X+1 LSB bits of the numerator register and the remaining MSB bits is used compute the remainder of the division process. In signed integer division, the absolute value of the signed number is obtained and the division is performed same way as unsigned case and at end, sign information is added to the quotient. The Program 15.1 and 15.2 can be used for dividing unsigned and signed integers up to 32-bits respectively.

Program 15.1 Unsigned Integer Division

```
Label           Mnemonic                Comments
                .text                   ; assembler directive to initialize the program section
                zero .s1 a1             ;zero the content of registers A1,A2 and A3
                || zero .d1 a2
```

```
                    || zero .l1 a3
                    zero .s1 a4                ; zero the content of registers A4,A5 and A6
                    || zero .d1 a5
                    || zero .l1 a6
                    zero .s1 a7                ;zero the content of registers A7,A8 and A9
                    || zero .d1 a8
                    || zero .l1 a9
                    zero .s1 a10               ;zero the content of registers A10,A11 and A16
                    || zero .d1 a11
                    mvk .s1 20h,a2             ; 16 LSBs of numerator moved to register A2
                    mvklh .s1 0h,a2            ; 16 MSBs of numerator moved to register A2
                    mvk .s1 03h,a3             ; 16 LSBs of denominator moved to register A3
                    mvklh .s1 0h,a3            ; 16 MSBs of denominator moved to register A3
                    mvk .s1 01h,a4             ; 1 is moved in register A4
                    lmbd .l1 a4,a2,a5          ; left most 1 detection for numerator, result in A5
                    lmbd .l1 a4,a3,a6          ;left most 1 detection for denominator, result in A6
                    sub .s1 a6,a5,a1           ;the difference in left most 1detection, result in A1
                    shl .s1 a3,a1,a3           ;the denominator aligned to numerator by left shifting,
                                               ; the shift value is the content of A1

                    add .l1 1,a1,a1
                    neg .s1 a4,a7
                    shl .s1 a7,a1,a8
                    not .s1 a8,a9
                    mv .l1 a1,a7
loop                subc .s1 a2,a3,a2          ;register A3 content subtracted from A2 content, result
                                               ;in A2 register
                    sub .l1 a1,a4,a1           ;the content of A1 register decremented
                    [a1] b .s2 loop            ;branch to loop for content register A1 non-zero
                    || nop 5                   ; no operations
                    and .d1 a2,a9,a10          ;quotient in A10 register
                    and .d1 a2,a8,a11
                    shr .s1 a11,a7,a11         ; reminder in A11 register
                    .end
```

Program 15.2 Signed Integer Division

```
Label               Mnemonic                   Comments
                    .text                      ; assembler directive to initialize the program section
                    zero .s1 a1                ; zero the content of registers A1-A11,B0 and B1
                    || zero .d1 a2
                    || zero .l1 a3
                    || zero .s2 b0
                    || zero .d2 b1
                    zero .s1 a4
                    || zero .d1 a5
                    || zero .l1 a6
```

```
zero .s1 a7
|| zero .d1 a8
|| zero .l1 a9
zero .s1 a10
|| zero .d1 a11
mvk .s2 -19,b0             ; numerator specified in lsb 16 bits of register B0
mvk .s2 10,b1              ; denominator specified in lsb 16 bits of register B1
abs .l1 b0,a2              ; absolute value of B0 stored in A2
cmplt .l2 b0,a2,b0         ;the sign information is stored in B0
abs .l1 b1,a3              ;absolute value of B1 stored in A3
cmplt .l2 b1,a3,b1         ; the sign information is stored in B1
sub .d2 b0,b1,b0
mvk .s1 01h,a4
lmbd .l1 a4,a2,a5          ; left most 1 detection for numerator, result in A5
lmbd .l1 a4,a3,a6          ;left most 1 detection for denominator, result in A6
sub .s1 a6,a5,a1
shl .s1 a3,a1,a3           ;the denominator aligned to numerator by left shifting,
                           ; the shift value is the content of A1
add .l1 1,a1,a1
neg .s1 a4,a7
shl .s1 a7,a1,a8
not .s1 a8,a9
mv .l1 a1,a7
loop subc .l1 a2,a3,a2     ; subtract operation performed result stored in A2
 sub .l1 a1,a4,a1
[a1] b .s2 loop
|| nop 5
and .d1 a2,a9,a10
[b0] neg .s1 a10,a10       ;quotient in register A10, for signed nos. sign information added
 and .d1 a2,a8,a11
shr .s1 a11,a7,a11         ; reminder in register A11
 .end
```

15.2.2 Convolution Operation

The basic operation to be implemented for signal processing applications is convolution. In ′C6X processor it can be implemented using multiply (MPY) and add (ADD) instruction. The multiply and add instructions are executed in parallel to perform single cycle multiply and accumulate (MAC) operation. In ′C6X there are two multipliers and six ALUs functioning in parallel, so two single cycle MAC operations can be performed simultaneously in path-A and path-B of the CPU paths respectively. The convolution operation can be performed for 8, 16 and 32-bits of data values in ′C6X processor and the assembly language program to perform 8, 16 and 32-bit convolution is given in program 15.3. The 8, 16 and 32 bit data values can be defined using assembler directives .byte, .half and .word respectively. The data values for the sequence can be directly defined in the program or it could be stored in separate data files. The values in the data files can be called in the assembly program using .include or .copy assembler directives. In program 15.3 the two sequence values that are to be convolved are defined using variables **x** and **h** and the number of values in the sequence are defined by the variables **n** and

m respectively. The number of time the convolution output is to be computed is **n+m-1.** While storing the sequence values in memory, padding of zeros for both the sequence **x** and **h** are necessary to avoid garbage values being accessed from memory during convolution operation. For sequence **x**, **m-1** zeros are to be padded after the sequence and **n-1** zeros to be padded before and after the sequence **h** as shown in the program 15.3. The stored values of the two sequences are read from memory one by one using load instruction through path-A and path-B simultaneously, get multiplied and accumulated, and repeated for n+m-1 times to get the first convolution output. The result is stored in memory using store instruction. After the address update of both sequences, the next output is computed and this process continued for n+m-1 times. The conditional operations of 'C6X are used to check the count values. The convolved output can be viewed in memory by invoking the memory window in CCS.

Program 15.3 Convolution operation

```
Label       Mnemonic                   Comments
            .data                      ; assembler directive to initialize the data section
x           .byte 1h,2h,2h,2h,2h,2h,1h ; the values of sequence x defined
                                       ; .byte, .half and .word assembler directives to represent-
                                       ; data in 8, 16 and 32 bit data formats respectively
xpa         .byte 0h,0h,0h,0h          ; m-1 values of zeros padded after the sequence x
hpb         .byte 0h,0h,0h,0h,0h,0h    ; n-1 values of zeros padded before the sequence h
h           .byte 1h,2h,2h,2h,1h       ; the values of sequence h defined
hpa         .byte 0h,0h,0h,0h,0h,0h    ; n-1 values of zeros padded after the sequence h
n           .set 7                     ; the no. of values in sequence x (m)
m           .set 5                     ; the no. of values in sequence h (n)
            .text                      ; assembler directive to initialize the program section
            zero .s1 a1                ; zero the contents of CPU registers
            || zero .d1 a2
            || zero .l1 a3
            zero .s1 a0
            || zero .d1 a4
             || zero .l1 a5
             zero .s2 b2
             || zero .s1 a5
             || zero .d1 a6
             zero .s2 b3
             || zero .l2 b4
             || zero .d1 a7
             mvkl .s1 n+m-1,a7         ; (n+m-1), ((n+m)*2)-2 and ((n+m)*4)-4 for 8,16 and 32 bit-
                                       ; data values respectively. The address displacement after-
                                       ; every convolution output
             ||mvkl .s2 h, b3          ; the start address of the sequence h loaded in register B3
             mvkl .s1 n+m -1,a1        ; the no. of times the convolution output to be computed
             || mvkl .s2 0100h,b5      ; the start address to store result is loaded in register B5
loop1        mvkl .s2 n+m -2,b0        ;the no. of times the multiplication and accumulation to be
                                       ; performed
```

```
              || mvkl .s1 x,a3          ; the start address of the sequence x loaded in register A3
              || zero .d1 a5
              || zero .l1 a6
loop          ldb .d1 *a3++,a4          ; the sequence values x and h are loaded from memory to-
              ||ldb .d2 *b3- -,b4       ; register A4 and B4 respectively. ldb, ldh and ldw instruction-
              ||nop 6                   ; for byte, half word and word load respectively
              mpy .m1 a4,b4,a5          ; multiplication and accumulation of x and h values
              || add .s1 a5,a6,a6
              || sub .s2 b0,1,b0
              || nop 5
              [b0] b loop
             ||nop 7
             sub .s1 a1,1,a1
             || stb .d2 a6,*b5++        ; the convolved output sequence stored in memory. stb, sth-
             ||nop 6                    ; and stw instructions for byte, half word and word store
              add .s2 b3,a7,b3          ; the address update for sequence h
              [a1] b loop1
              ||nop 7
              .end
```

15.2.3 DFT using FFT Algorithm

The fast computation method of Discrete Fourier Transform (DFT) is using Fast Fourier Transform (FFT) algorithm (refer chapter 1.14). The FFT algorithm is based on the symmetry property of the factor W_N^{kn}, where ($W_N^{kn} = e^{-j(2\pi/N)kn}$). The computation of DFT using FFT algorithm is carried out in two methods, Decimation in Time (DIT) and Decimation in Frequency (DIF). The best way to implement DFT either in DIT or DFT method is through butterfly structures. In this chapter 8-point DFT implementation using DIT radix-2 FFT algorithm is presented and it's ′C64X assembly language program is given in Program 15.6. The first module needed in DFT computation using radix-2 algorithm is the rearrangement of input sequence in bit reversed order. In ′C5X, ′C3X and ′C54X processors a specific addressing mode called bit reversed addressing mode is available to perform the bit reversal. But in ′C6X there is no such addressing mode. Hence ′C6X assembly language program to rearrange the input sequence in bit reversed order is given in Program 15.4. The input sequence in DFT computation is real, but the coefficients of FFT and the outputs of the intermediate stage of the butterfly structure are complex, hence the second module required is complex number multiplication. To perform complex multiplication, the ′C6X assembly program is given in Program 15.5.

The FFT coefficients are the twiddle factor W_N^k represented in trigonometry form as cos ($j2\pi k/N$) + j sin ($j2\pi k/N$), where k varies from 0-7 and N is the number of inputs i.e. 8 for 8-point DFT. The cosine and sine function can have values from + 1 to -1 as k varies from 0-7, the fractional values of the coefficients are scaled by an appropriate scaling factor S and rounded off to nearest integers. The scaling factors are selected in powers of 2 ($S=2^x$), because after multiplying the inputs with coefficients in the butterfly structure, the resultant product is to be divided by the scaling factor S to get back the actual value. If the scaling factor is selected in powers of 2, division can be done easily by shift right operation (SHR) for other scaling factors the division program given in program 15.1 and 15.2 can be used. The DFT outputs obtained in the butterfly structure will have error comparing to manual calculation in the path where the twiddle factors are fractional and this error is due to rounding off of the coefficients.

Program 15.4 Bit reversal of input sequence

```
Label       Mnemonic                 Comments
            .data                    ; assembler directive to initialize the data section
x           .byte 1,-1,1,0,2,-1,2,1  ; the input sequence that is to be bit reversed
            .text                    ; assembler directive to initialize the program section
n           .set 8                   ; number of points
l           .equ (n/4)               ; number of swaps
k           .equ (n/2)               ; half point value
            zero .s1 a1              ; zero the register contents
            || zero .d1 a2
            || zero .l1 a3
            zero .s1 a4
            || zero .d1 a5
            || zero .l1 a6
            zero .s1 a7
            || zero .d1 a8
            || zero .l1 a9
            zero .s1 a10
            || zero .d1 a11
            || zero .l1 a12
            mvkl .s1 x,a3            ;the start address of the sequence x loaded in register A3
            mvkl .s1 k,a1
            || mvkl .s2 l,b0
loop        ldb .d1 *++a3[a1],a4
            ||nop 7
            mv .s1 a4,a5
            ||sub a1,1,a0
            ||nop 6
            ldb .d1 *--a3[a0],a4
            || nop 7
            stb .d1 a5,*a3++[a0]
            ||sub a1,2,a0
||nop 7
            stb .d1 a4,*a3--[a0]
            ||sub b0,1,b0
||nop 7
            [b0] b .s2 loop
            ||nop 7
            .end
```

Program 15.5 Complex number multiplication

```
Label       Mnemonic                 Comments
            .data                    ; assembler directive to initialize the data section
re          .byte -1,-1              ; real part of two complex numbers defined
```

```
im          .byte 1,1             ; imaginary part of two complex numbers defined
pro         .byte 0,0             ; multiplication result real and imaginary part output buffer
            .text                 ; assembler directive to initialize the program section
            mvkl re,a3            ; start address of real part loaded in register A3
            mvkl im,a4            ; start address of imaginary part loaded in A4
            mvkl pro,a15          ; start address of output buffer loaded in A15
            ldb *a3++,a5          ; real part loaded in register A5 and A6
            ||nop 7
            ldb *a3++,a6
            ||nop 7
            ldb *a4++,a7          ; imaginary part loaded in register A7 and A8
            ||nop 7
            ldb *a4++,a8
            ||nop 7
            mpy a5,a6,a9          ; real parts multiplied result in a9
            ||nop 4
            mpy a7,a8,a10         ; imaginary parts multiplied result in a10
            ||nop 4
            neg a10,a10           ; sign information of imaginary part product extended
            add a9,a10,a9         ; real part of multiplication obtained
            stb a9,*a15++         ; real part of multiplication stored in memory
            || nop 7
            mpy a5,a7,a9          ; real and imaginary part multiplied
            ||nop 4
            mpy a6,a8,a10         ; real and imaginary part multiplied
            || nop 4
            add a9,a10,a9         ; imaginary part of multiplication obtained
            stb a9,*a15++         ; imaginary part of multiplication stored in memory
            || nop 7
            .end
```

Program 15.6 DFT computation (8-poit) using DIT FFT radix-2 algorithm

```
Label       Mnemonic              Comments
            .data                 ; assembler directive to initialize the data section
core        .byte 8,6,0,-6,-8,-6,0,6  ;real value of coeff. scaled by factor 8
coim        .byte 0,-6,-8,-6,0,6,8,6  ;imaginary value of coeff. scaled by factor 8
xb          .byte 1,2,1,2,-1,-1,0,1   ;input sequence x in bit reversed order
n           .set 8                ;no of inputs to find DFT
h           .set (n/2)
h1          .set h-1
q           .set (n/4)
q1          .set q-1
x2r         .byte 0,0,0,0,0,0,0,0 ;2point butterfly output buffer
x4r         .byte 0,0,0,0,0,0,0,0 ;4point butterfly real and imaginary output value buffer
            .byte 0,0,0,0,0,0,0,0
```

```
x8r             .byte 0,0,0,0,0,0,0,0       ;8point butterfly real and imaginary output value buffer
                .byte 0,0,0,0,0,0,0,0
                .text                       ; assembler directive to initialize the program section
; 2 point butterfly computation
                mvkl .s1 core,a0            ;start address of real part of coeff. loaded in A0
                mvkl .s1 h,a1
                mvkl .s1 xb,a4              ;start address of input sequence loaded in A4
                mvkl .s1 x2r,a15            ;start address of x2r buffer loaded in A15
                ldb *++a0[h],a14            ;load coefficient in register A14
                ||nop 7
loop
                ldb *a4++,a5                        ; load first two inputs in register A5 and A6
                ||nop 7
                ldb *a4++,a6
                || nop 7
                add .l1 a5,a6,a8
                ||mpy .m1 a6,a14,a9
                ||nop 6
                shr a9,3,a9                 ; product divided by a factor 8 by shift right operation
                add a5,a9,a9
                ||stb a8,*a15++             ; two point butterfly output stored in buffer
                ||nop 7
                stb a9,*a15++
                || sub a1,1,a1
                ||nop 7
                [a1] b loop
                ||nop 7
; 4 point butterfly computation
                mvkl x4r,a15                ;start address of x4r buffer loaded in A15
                ||mvkl 1,b2
                mvkl q,a2
loop3
                mvkl core,a0                ;start address of real part of coeff. loaded in A0
                mvkl coim,a1                ;start address of imaginary part of coeff. loaded in A1
                ||mvkl q,b1
loop2           [b2] mvkl .s1 x2r,a3        ;start address of x2r buffer loaded in A3
                [!b2] mvkl .s1 x2r+h,a3     ;start address of the half of the buffer x2r loaded in A3
                mvkl q,b0
loop1           ldb *a0++[q],a14            ;load real part of coefficient in A14
                ||nop 7
                ldb *a1++[q],a13            ;load imaginary part of coefficient in A13
                ||nop 7
                ldb *a3++[q],a4             ; load inputs
                ||nop 7
                ldb *a3- -[q1],a5
                ||nop 7
```

```
                    mpy a5,a14,a6
                    ||nop 4
                    shr a6,3,a6                  ; product divided by a factor 8
                    ||mpy a5,a13,a7
                    || nop 4
                    shr a7,3,a7                  ; product divided by a factor 8
                    ||add a4,a6,a6
                    ||nop 6
                    stb a6,*a15++                ; real part of 4-point butterfly output stored
                    ||nop 7
                    stb a7,*a15++                ; imaginary part of 4-point butterfly output stored
                    || sub b0,1,b0
                    ||nop 7
                    [b0] b loop1
                    ||nop 7
                    sub b1,1,b1
                     [b1] b loop2
                    || nop 7
                    zero b2
                    ||sub a2,1,a2
                    [a2] b loop3
                    ||nop 7
; 8 point butterfly computation
                    mvkl x8r,a15                 ;start address of x8r buffer loaded in A15
                    mvkl core,a0                 ;start address of real part of coeff. loaded in A0
                    mvkl coim,a1                 ;start address of imaginary part of coeff. loaded in A1
                    ||mvkl q,b0
loop5               mvkl x4r,a3                  ;start address of x4r buffer loaded in A3
                    mvkl h,a2
loop4               ldb *a0++,a14                ;load real part of coefficient in A14
                    ||nop 7
                    ldb *a1++,a13                ;load imaginary part of coefficient in A13
                    ||nop 7
                    ldb *a3++,a4                 ; load inputs
                    ||nop 7
                    ldb *a3++(h1+h),a5
                    ||nop 7
                    ldb *a3++,a6
                    ||nop 7
                    ldb *a3—(h1+h),a7
                    ||nop 7
                    mpy a6,a14,a8
                    ||nop 4
                    shr a8,3,a8                  ; product divided by a factor 8
                    ||mpy a7,a13,a9
                    ||nop 4
```

```
shr a9,3,a9                   ; product divided by a factor 8
neg a9,a9
add a8,a9,a9
add a4,a9,a4
stb a4,*a15++                 ; real part of 8 point butterfly output stored
||nop 7
mpy a6,a13,a8
||nop 4
shr a8,3,a8                   ; product divided by a factor 8
||mpy a7,a14,a9
||nop 4
shr a9,3,a9                   ; product divided by a factor 8
add a8,a9,a9
add a5,a9,a5
stb a5,*a15++                 ; imaginary part of 8 point butterfly output stored
|| sub a2,1,a2
|| nop 7
[a2] b loop4
|| nop 7
sub b0,1,b0
[b0] b loop5
||nop 7
.end
```

APPLICATION PROGRAMS IN ′C67X 15.3

The programs in this section are executed in TMS320C6713 starter kit. The programs may be written in assembly language, C language and combination of both. In this section, examples using the last two approaches are presented.

15.3.1 Code Development in C Environment using Code Composer Studio

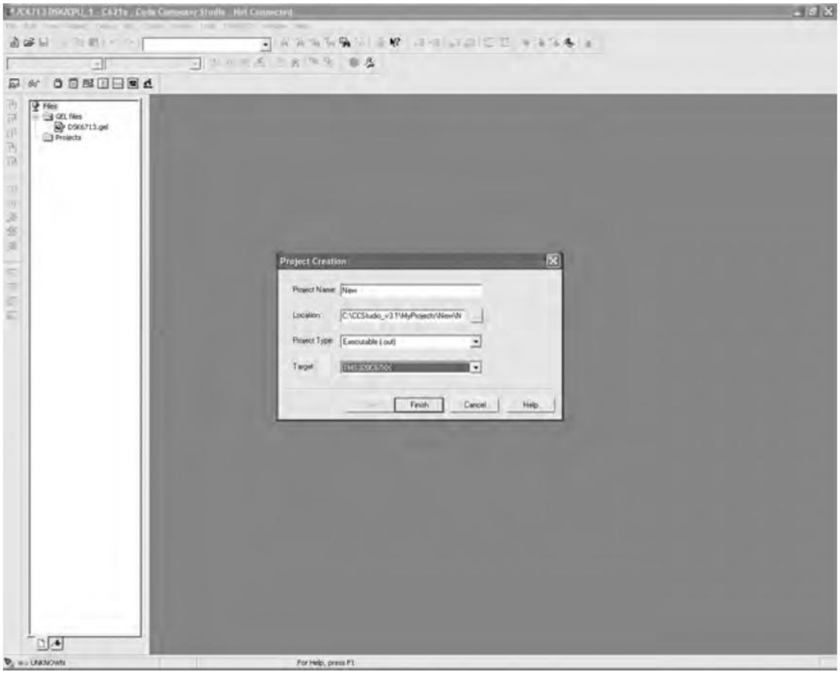
Code Composer Studio (CCS) supports the integrated development environment (IDE) for real - time digital signal processing applications based on the C programming language. It incorporates a C compiler, an assembler, and a linker. It has graphical capabilities and supports real - time debugging. Following are the various file extensions employed by code composer studio:

1. file.pjt : To create and build a project named file.
2. file.c : C source program.
3. file.asm : Assembly source program created by the user, by the C compiler, or by the linear optimizer.
4. file.sa : Linear assembly source program. The linear optimizer uses file.sa as input to produce an assembly program file.asm.
5. file.h : Header support file.
6. file.lib : Library file, such as the run - time support library file rts6700.lib.
7. file.cmd : Linker command file that maps sections to memory.
8. file.obj : Object file created by the assembler.

9. file.out : Executable file created by the linker to be loaded and run on the TMS320C6713 processor.

10. file.cdb : Configuration file when using DSP/BIOS.

The following steps are adopted for code development in C environment. In the Code Composer Studio, Click New Project under the menu Project. Enter the project name, project output and the target processor as shown in Figure 15.3. After completing this, the project will get added in the left side as shown in Figure 15.4. A sample C-code new.c is given in Program 15.7 and the linker command file C6713dsk.cmd file is given in Program 15.8. This is added to the project in order to provide the details about the memory map for the program.



Project Name: New

Location: C:\CCStudio_v3.1\MyProjects\New\N

Project Type: Executable (.out)

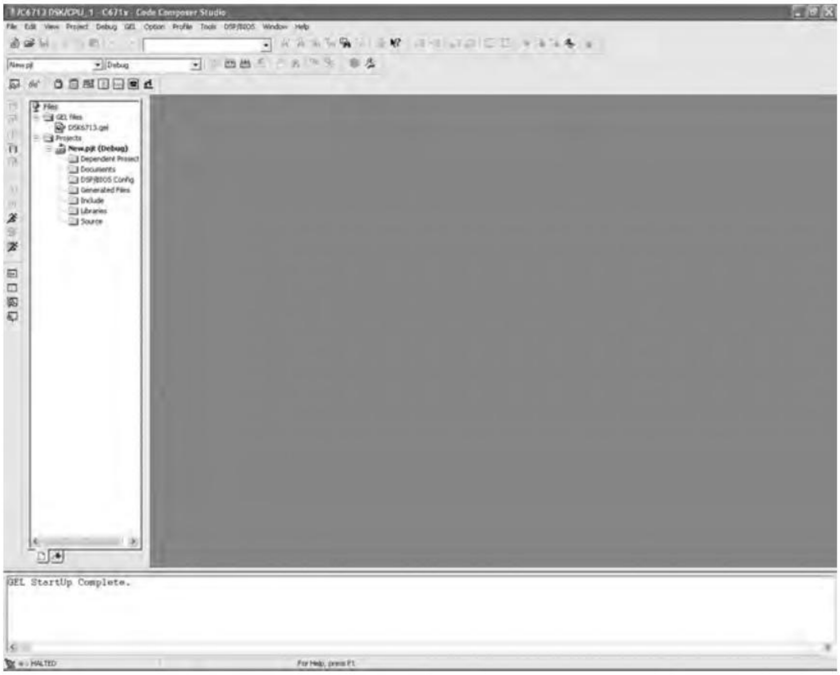
Target: TMS320C67XX

Fig. 15.3

Program 15.7 New.c

```
#include <stdio.h>
#include <math.h>
void main()
{
          int a,b,c;

          a=100;
          b=120;
          c=a+b;
          printf("Sum of %d and %d is %d\n",a,b,c);
}
```



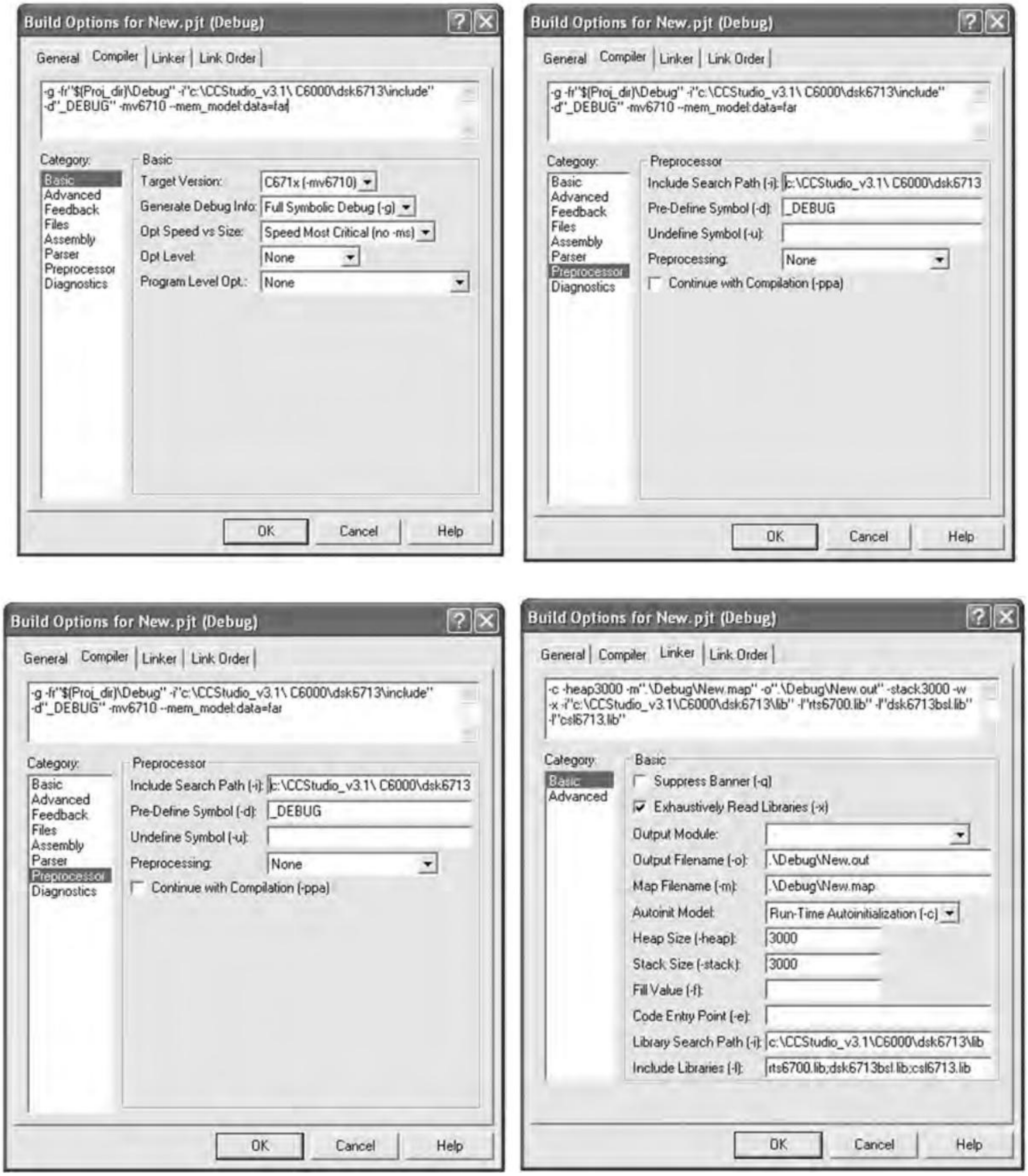
```
GEL StartUp Complete.
```

Fig 15.4

Program 15.8 C6713dsk.cmd Linker command File

```
/*C6713dsk.cmd Linker command file*/
MEMORY
{
IVECS:   org=0h,          len=0x220
IRAM:    org=0x00000220,  len=0x0000A000 /*internal memory*/
SDRAM:   org=0x80000000,  len=0x00100000 /*external memory*/
FLASH:   org=0x90000000,  len=0x00020000 /*flash memory*/
}
SECTIONS
{
  .EXTRAM :>   SDRAM
   .vectors :>  IVECS
   .text :>     IRAM
   .bss :>      IRAM
   .cinit :>    IRAM
   .stack :>    IRAM
   .sysmem :>   SDRAM
   .const :>    IRAM
   .switch :>   IRAM
   .far :>      IRAM
```

```
        .cio :>        SDRAM
        .csldata :>    IRAM
    }
```



Build Options for New.pjt (Debug)

```
-g -fr"$(Proj_dir)\Debug" -i"c:\CCStudio_v3.1\ C6000\dsk6713\include"
-d"_DEBUG" -mv6710 --mem_model:data=far
```

Basic

Target Version: C671x (-mv6710)

Generate Debug Info: Full Symbolic Debug (-g)

Opt Speed vs Size: Speed Most Critical (no -ms)

Opt Level: None

Program Level Opt.: None

Build Options for New.pjt (Debug)

```
-g -fr"$(Proj_dir)\Debug" -i"c:\CCStudio_v3.1\ C6000\dsk6713\include"
-d"_DEBUG" -mv6710 --mem_model:data=far
```

Preprocessor

Include Search Path (-i): c:\CCStudio_v3.1\ C6000\dsk6713

Pre-Define Symbol (-d): _DEBUG

Undefine Symbol (-u):

Preprocessing: None

□ Continue with Compilation (-ppa)

Build Options for New.pjt (Debug)

```
-g -fr"$(Proj_dir)\Debug" -i"c:\CCStudio_v3.1\ C6000\dsk6713\include"
-d"_DEBUG" -mv6710 --mem_model:data=far
```

Preprocessor

Include Search Path (-i): c:\CCStudio_v3.1\ C6000\dsk6713

Pre-Define Symbol (-d): _DEBUG

Undefine Symbol (-u):

Preprocessing: None

□ Continue with Compilation (-ppa)

Build Options for New.pjt (Debug)

```
-c -heap3000 -m".\Debug\New.map" -o".\Debug\New.out" -stack3000 -w
-x -i"c:\CCStudio_v3.1\C6000\dsk6713\lib" -l"rts6700.lib" -l"dsk6713bsl.lib"
-l"csl6713.lib"
```

Basic

□ Suppress Banner (-q)

☑ Exhaustively Read Libraries (-x)

Output Module:

Output Filename (-o): .\Debug\New.out

Map Filename (-m): .\Debug\New.map

Autoinit Model: Run-Time Autoinitialization (-c)

Heap Size (-heap): 3000

Stack Size (-stack): 3000

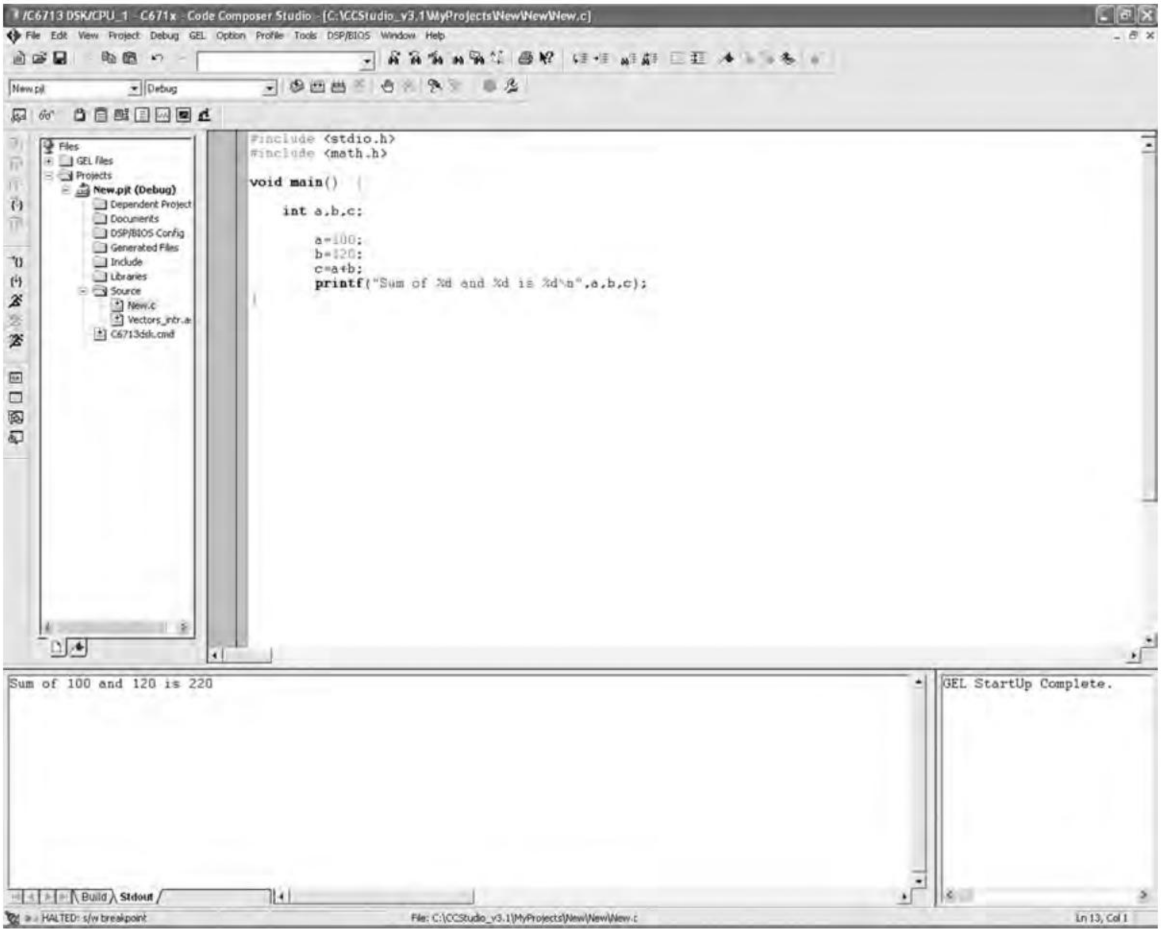
Fill Value (-f):

Code Entry Point (-e):

Library Search Path (-i): c:\CCStudio_v3.1\C6000\dsk6713\lib

Include Libraries (-l): rts6700.lib;dsk6713bsl.lib;csl6713.lib

Fig. 15.5 *Build Options*



```
#include <stdio.h>
#include <math.h>

void main()  {

     int a,b,c;

         a=100;
         b=120;
         c=a+b;
         printf("Sum of %d and %d is %d\n",a,b,c);
}
```

```
Sum of 100 and 120 is 220
```

```
GEL StartUp Complete.
```

```
HALTED: s/w breakpoint        File: C:\CCStudio_v3.1\MyProjects\New\New\New.c        Ln 13, Col 1
```

Fig. 15.6 *CCS IDE with output window*

The code generation tools underlying CCS, that is, C compiler, assembler, and linker, have a number of options associated with each of them. These options must be set appropriately before attempting to build a project. Once set, these options will be stored in the project file. Figure 15.5 shows the build options set for the new.pjt. After setting the build options and adding necessary files, goto Debug→Connect for connecting the target board with CCS. Then goto Project→Rebuild All to build the entire project. After building the entire project, goto File→Load program and select the new.out to be loaded onto the target board. Then goto debug→ Run the project to see the result on the output window as shown in Fig. 15.6.

15.3.2 Computation of the 8- point DFT using FFT Algorithm in C Environment

The DSK6713 kit has an on-board Audio Codec (TLV320AIC23), which can be configured for speech input and speech output. The sampling frequency for speech input can be varied from 8KHz to 96KHz using software. More details about the AIC23 parameters which can be modified using software are provided in dsk6713_aic23.h. The c6713dskinit.c file comprises functions necessary for speech input and speech output and it includes the function input_sample(). This function given in Program 15.9. is used to capture speech input through microphone interface provided in the kit.

Program 15.9 Program to capture speech input through microphone interface

```
Uint32 input_sample()
{
        short CHANNEL_data;
        if (poll) while(!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE));//if ready to receive
         AIC_data.uint=MCBSP_read(DSK6713_AIC23_DATAHANDLE);     //read data
        CHANNEL_data=AIC_data.channel[RIGHT];
        AIC_data.channel[RIGHT]=AIC_data.channel[LEFT];
        AIC_data.channel[LEFT]=CHANNEL_data;
        return(AIC_data.uint);
}
```

The C-program for the computation of 8- point DFT using FFT algorithm is given in Program 15.10. The speech input through the microphone interface is sampled at the rate of 8 KSPS, digitized and stored in a data file ‘samplefft.txt’. The 8- point DFT is computed and the DFT coefficients are printed on the output window of CCS.

Program 15.10 Computation of 8-Point DFT in ′C6713 using C code

```
#include “DSK6713_aic23.h”
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define noof_stages 3 /* the no. of butter fly stages - 3*/
#define noof_samples 8 /* the no. of inputs 8*/
#define PI 3.14159
struct complex {
        float real;
        float imag;
};
struct buffer {
        struct complex data[1][20];
};
#pragma DATA_SECTION(real_buffer,”.EXTRAM”)
struct buffer real_buffer;
FILE *f1;
void fft (struct buffer *, int , int );
/* Main Program */
void main()
{
        int k;
        float sample;
        int sn,sm;
        sn=noof_samples;
        sm=noof_stages;

        printf(“Input\n”);
        f1=fopen(“samplefft.txt”,”r”);
        for(k=0;k<noof_samples;k++)
        {
                fscanf(f1,”%f”,&sample);
                real_buffer.data[0][k].real = ((float)sample);
                fscanf(f1,”%f”,&sample);
                real_buffer.data[0][k].imag = ((float)sample);
                printf(“%f\t%fi\n”,real_buffer.data[0][k].real,real_buffer.data[0][k].imag);
        }
        fclose(f1);
        fft(&real_buffer,sn,sm);
}
/* Function to Compute Fast Fourier Transform */
void fft (struct buffer *input_data, int n, int m) {
        int n1,n2,i,j,k,l;
        float xt,yt,c,s,e,a;
                n2 = n;
                for ( k=0; k<m; k++) {
                        n1 = n2;
                        n2 = n2/2;
                        e = PI/n1;
                        for ( j= 0; j<n2; j++) {
                                a = j*e;
                                c = (float) cos(a);
                                s = (float) sin(a);
                                for (i=j; i<n; i+= n1) {
                                l = i+n2;
xt = input_data->data[0][i].real - input_data->data[0][l].real;
    input_data->data[0][i].real = input_data->data[0][i].real+input_data->data[0][l].real;
yt = input_data->data[0][i].imag - input_data->data[0][l].imag;
    input_data->data[0][i].imag = input_data->data[0][i].imag+input_data->data[0][l].imag;
                        input_data->data[0][l].real = c*xt + s*yt;
                        input_data->data[0][l].imag = c*yt - s*yt;
                        }
                        }
                }
                j = 0;
                for ( i=0; i<n-1; i++) {
                        if (i<j) {
xt = input_data->data[0][j].real;
```

```
    input_data->data[0][j].real = input_data->data[0][i].real; input_data->data[0][i].real = xt;
yt = input_data->data[0][j].imag; input_data->data[0][j].imag = input_data->data[0][i].imag;
                        input_data->data[0][i].imag = yt;
                }
        }
/*      printf("Output\n");
        for ( l=0; l<n; l++)
        {
                printf("%f\t",input_data->data[0][l].real);
                printf("%fi\n",input_data->data[0][l].imag);
        }
*/
        return;
}
```

15.3.3 Estimation of Clock Cycles Required for Code Execution using CCS

The number of clock cycles/machine cycles required to excute the complete program in assembly, C as well as combined assembly and C environment can be estimated using CCS tool. In CCS, first select the option Profile – Clock – Enable and then select the option Profile – Clock – View, a clock icon will appear on the right down corner of the CCS menu bar. The clock can be resetted by double clicking the clock icon. Once the project file is downloaded to the target processor, the PC will set to the starting point of the program code (default value of the start address is 0000 0020h). Break point can be introduced at the last line of the code. (To introduce break point refer section 15.1.4). Select the Debug-run option in the CCS tool or use the shortcut key to run the code from the current point of the program counter (PC) to the address where break point is introduced. The count shown in the clock icon of CCS tool is the measure of number of clock cycles required to execute the block of code from the starting address to the address where the break point is introduced. In the same way by introducing breakpoints at any other place of the program, the clock cycle count required to execute any block of the program can be computed.

15.3.4 Comparison of the Number of Clock Cycles Required for the Computation of 8 Point DFT in both Assembly Language and C Environment

The number of clock cycles required for the computation of DFT using assembly language program given in section 15.2.3 (Program 15.6) and the C program given in 15.3.2 (Program 15.10) are evaluated and compared in this section. For the evaluation of the number of clock cycles required for the computation of DFT using Program 15.6, the number of clock cycles required for bit-reversing the input sequence is computed using CCS tool as mentioned in section 15.3.3. The 8,16,32 and 64 input sequences take 119, 197, 353 and 671 machine cycles respectively for bit-reversing. The clock cycle count for the computation of 8-point DFT using Program 15.6 is also evaluated. The number of clock cycles required to compute 2-point, 4-point and 8-point butterfly outputs are 220, 585 and 565 respectively. For the complete computation of the 8-point DFT using assembly code the number of clock cycles evaluated using CCS tool in ′C6416 starter kit is 1,479.

The number clock cycles required for 8-point DFT computation using C-code in ′C6713 starter kit is also evaluated and the value is 14,739 clock cycles. Hence, the number of clock cycles required for computing 8 point DFT using programming in C environment is larger by a factor of 10. In addition to non optimality of the C compiler, the type of processor used for the implementation also contributes to the difference. The C6416 processor used for the assembly language programming in section 15.2.4 is a fixed point processor where as the C6713 processor used executing the program in C environment is a floating point processor. The floating point processor in general requires more cycles than the fixed point processor. It may be noted CCS can be used for both of these processors to develop programs in assembly language, C language or combinations of both.

15.3.5 Mixing Assembly Language and C Language

The programs in assembly language can be optimized by efficient use of the architecture of the processor and hence require less number of clock cycles for execution. However, this requires the designer to learn the assembly language of the processor. On the other hand, the length of the source program in assembly language is larger compared to that of C language and hence requires more time for development, debugging and testing. Moreover, for programming in C language, the designer need not know either the architecture or the assembly language of the processor. In order to combine the advantages of both, assembly language program may be used for implementing the functions which are computation intensive and can be invoked from the C environment.

15.3.6 Different Ways of Invoking Assembly Language in C-code

In this section, different ways of invoking assembly language in C-code is illustrated on a TMS320C6713 floating point DSP processor using Code Composer Studio software.

There are four ways of invoking assembly language in C-code for DSP programming:

- callable assembly
- intrinsic functions
- linear assembly
- inline assembly

The above-mentioned approaches are illustrated with an example application which requires computing the Euclidean distance for input with two variables. The C-code for the computation of Euclidean distance is given in Program 15.11.

Program 15.11 **C Code for the computation of Eucludian distance**

```
#include<math.h>
#include<stdio.h>
main( )
{
        float x1,x2,y1,y2,dx,dy,e;
        dx = x1-x2;
        dy = y1-y2;
        e = sqrt(dx*dx + dy*dy);
}
```

Callable Assembly The callable assembly approach uses the C source code, which calls an externally declared user defined assembly language function. The C-code can be re-written using callable assembly as shown in Program 15.12.

Program 15.12 C Code for the computuation of Eucludian distance using callable assembly language function:

```
#include<math.h>
#include<stdio.h>
extern float errasm(float,float,float,float);
main( )
{
          float x1,x2,y1,y2,e;
           e = errasm(x1,y1,x2,y2);
}
```

In program 15.12, errasm() is a function called by c-code which is written in assembly language and saved as errasm.asm. The errasm.asm is given by Program 15.13.

Program 15.13 errasm.asm Code

```
          .def _errasm

_errasm:           SUBSP .L1     A4,A6,A7;
                   NOP           5
                   SUBSP .L2     B4,B6,B7;
                   NOP           5
                   MV .S1        A7,A8;
                   MV .S2        B7,B8;
                   MPYSP .M1     A7,A8,A5;
                   NOP           5
                   MPYSP .M2     B7,B8,B5;
                   NOP           5
                   ADDSP .L1X    A5,B5,A9;
                   NOP           5
                   RSQRSP .S1    A9,A6;
                   NOP           5
                   RCPSP .S1     A6,A4;
                   NOP           5
                   B             B3;
                   NOP           3
                   .end
```

Intrinsic Functions Intrinsics are special functions that map directly to inline C6x instructions. For example, int _mpy() is equivalent to the assembly instruction MPY to multiply the 16LSBs of two numbers. The above-mentioned C- code example can be written using intrinsic functions as shown in Program 15.14

Program 15.14 C-code with Intrinsic functions:

```
#include<ieeef.h>
#include<fastrts67x.h>
#include<stdio.h>
main( )
{
float x1,x2,y1,y2,dx,dy,e;
dx = x1-x2;
dy = y1-y2;
e = _rcpsp(_rsqrsp(dx*dx + dy*dy));
}
```

In the above C-code, two intrinsic functions are used. float _rcpsp(float src) computes the approximate 32-bit float reciprocal and float _rsqrsp(float src) computes the approximate 32-bit float square root reciprocal.

Linear Assembly Linear assembly code is a cross between assembly and C. It uses the syntax of assembly code instructions such as ADD, SUB, and MPY, but with operands/registers as used in C. The above-mentioned C- code example can be written using linear assembly as shown in Program 15.15.

Program 15.15 C Code for linear assembly

```
#include<math.h>
#include<stdio.h>
extern float err(float,float,float,float);
main( )
{
float x1,x2,y1,y2,e;
 e = err(x1,y1,x2,y2);
}
```

In program 15.15, err() is a function called by c-code which is written in linear assembly and saved as err.sa file. The linear assembly code for function err() is given in program 15.16.

Program 15.16 Linear asm Code

```
             .def _err
_err:                 .cproc    zc,zcs,msf,msfs
            .reg      x,y,z,w,d1,d2,r;
            mv        zc,x
            mv        zcs,y
            mv        msf,z;
            mv        msfs,w;
```

```
subsp     x,y,d1;
mpysp     d1,d1,y;
subsp     z,w,d2;
mpysp     d2,d2,w;
addsp     y,w,x;
rsqrsp    x,y;
rcpsp     y,r;
 .return  r
.endproc
```

Inline Assembly An inline assembly code can be used with the *asm* statement within a C program. For example, asm(“ MVK 0x0040,B6”). The above-mentioned C- code example can be written using inline assembly as shown in Program in 15.17.

Program 15.17 C Code with Inline assembly

```
#include<math.h>
#include<stdio.h>
main( )
{
float x1,x2,y1,y2,dx,dy,e;
dx = x1-x2;
 dy = y1-y2;
asm(“ mpysp .m1 a4,a4,a6”);
asm(“ NOP 5”);
asm(“ mpysp .m2 b4,b4,b6”);
asm(“ NOP 5”);
asm(“ addsp .l1x a6,b6,a5”);
asm(“ NOP 5”);
asm(“ rsqrsp .s1 a5,a6”);
asm(“ NOP 5”);
asm(“ rcpsp .s1 a6,a4”);
asm(“ NOP 5”);
e=getans();
asm(“ NOP 5”);
}
```

The number of clock cycles required for the computation of Eucludian distance using different approaches of invoking assembly using C-code is reported in Table 15.1. It may be observed from Table 1 that a pure C source code takes a longer execution time followed by inline assembly, callable assembly, intrinsic functions and linear assembly based approaches.

Table 15.1 *Number of Clock cycles required for computation of Eucluduian distance*

Approach	*No. of. Clock Cycles*	*Accel. Factor*
C-code	1378	—
In-line Assembly	466	2.957
Callable Assembly	411	3.353
Intrinsic Functions	401	3.436
Linear Assembly	371	3.714

INTERNAL MEMORY 15.4

The internal memory configuration varies between the different ′C6X processors. The TMS320C620X/ TMS320C670X family processors have separate on-chip program and data memories. The internal program memory can be accessed by the CPU or it can be operated as program cache. The size of internal program memory is 64 K bytes of RAM and it can accommodate 16K 32-bit instructions. The CPU accesses this program memory space through program memory controller. The program memory controller performs CPU and DMA (Direct Memory Access) requests to internal program memory, performs CPU requests to external memory through external memory interface (EMIF) circuit and also manages the internal program memory when it is configured as cache.

The size of internal data memory is 64 K bytes of RAM. Both CPU and DMA controller can access this data memory space through data memory controller. The data memory controller connects CPU to external memory and on-chip peripherals through EMIF and peripheral bus controller respectively. The ′C6202 processor has 2x128 K bytes of internal program memory blocks out of which one 128K bytes block can be used as program cache.

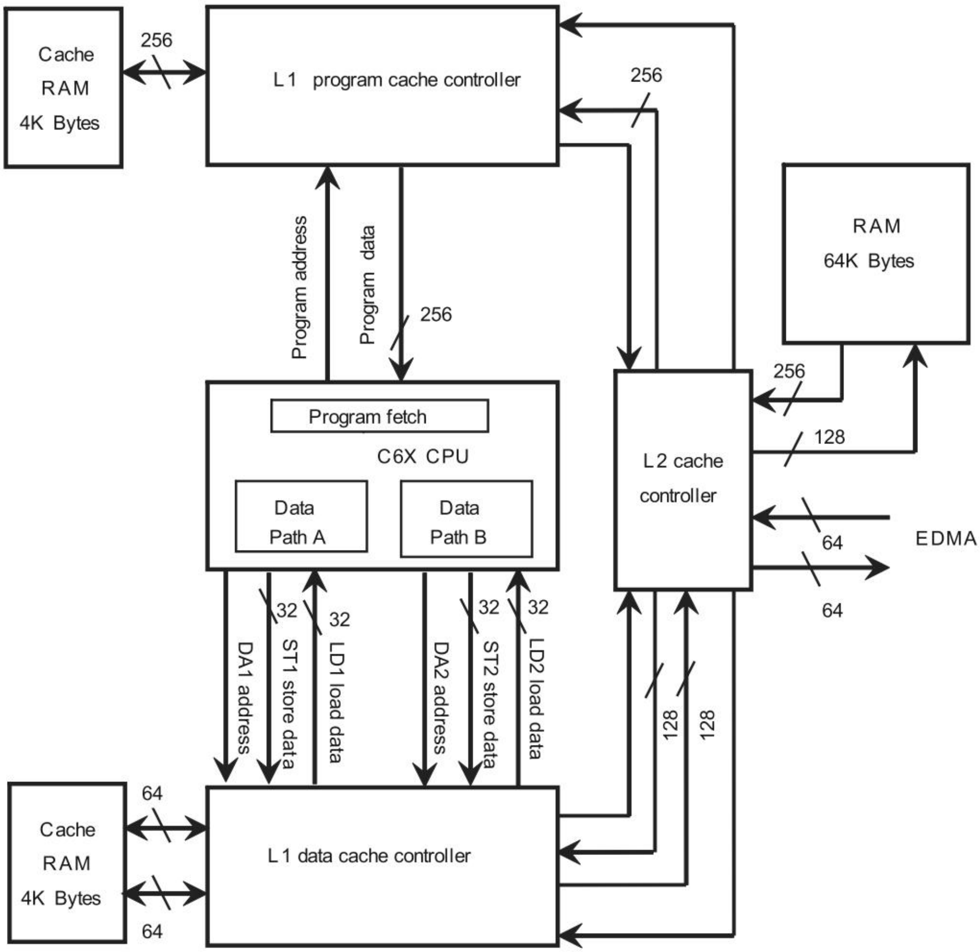
The ′C621X/′C671X family processors have cache-based internal memory architectures. They are provided with two level memory architecture for internal program and data busses. The first level of internal memory is with separate level-one program (L1P) cache and data cache (L1D) each of size 4K bytes. The program and data cache spaces are not included in the memory map and are enabled at all times. The level-one cache memories are accessible only by the CPU.

The program cache controller interfaces the CPU to the L1P cache. A 256 wide path is provided from to the CPU to allow a continuous stream of eight 32-bit instructions for maximum performance. The 4K L1P cache is organized as a 64 line direct mapped cache with a 64 byte line size.

The data cache controller provides interface between the CPU and L1D cache. The L1D is a dual-ported memory. This allows simultaneous access by both paths of the CPU (Path A and B). The L1D, 4K cache is organized as a 64 set 2-way set associative cache with a 32 byte line size. The second level of internal memory is 64K bytes of RAM that is shared by both program and data memory space with L2 cache controller. The internal memories and bus connections between the CPU and various controllers are shown in Fig. 15.7.

First the L1P and L1D caches are accessed, on a miss to either L1D or L1P; the request is passed to L2 controller. The L2 controller facilitates the following accesses

- The CPU and the enhanced direct memory access (EDMA) controller accesses to the internal memory, and performs the necessary arbitration
- The CPU data access to the EMIF



Note: i) For ′C67X processors – LD1 & LD2 data bus size - 64-bits
ii) For ′C64X processors – LD1, LD2, ST1& ST2 data bus size - 64-bits.

Fig. 15.7 *Internal Memory Block diagram of ′C6X processors*

- The CPU access to on-chip peripherals
- Sends a request to EMIF for an L2 data miss.

On request to L2 service, the service depends on the operation mode of L2, which is set in the Cache Configuration Register Fields (CCFG). This is a memory mapped register, whose memory map address is 0184 0000h. The format of the CCFG is shown in Fig. 15.8, and the various L2 modes are shown in the Table 15.2. The L2 memories are organized as four 64 bit wide banks.

31	30 10	9	8	7 3	2 0
P	Reserved	IP	ID	Reserved	L2MODE
RW,+0	R, +x	W,+0	W,+0	R, +0 0000	RW, +000

Fig. 15.8 *Format of Cache Configuration Register (CCFG)*

Table 15.2 *Cache configuration register field description*

Field	*Description*
L2 MODE	L2 Operation modes 000b – 64 K bytes RAM 001b – 16 K bytes 1-way cache/48 K bytes mapped RAM 010b – 32 K bytes 2-way cache/32 K bytes mapped RAM 011b – 48 K bytes 3-way cache/16 K bytes mapped RAM 111b – 64 K bytes 4-way cache
ID	Invalidate L1D ID =0 - normal L1D operation, ID = 1 – All L1D lines invalidated
IP	Invalidate L1P IP =0 - normal L1P operation, IP = 1 – All L1P lines invalidated
P	L2 Requestor Priority P=0, CPU accesses prioritized over enhanced DMA accesses P=1, Enhanced DMA accesses prioritized over CPU accesses

EXTERNAL MEMORY 15.5

On L2 data miss, the L2 controller sends a request to external memory interface (EMIF). The memory attribute register (MAR) can be programmed to turn on caching of each of the external chip enable (CE) spaces. In this way, a single word reads to external mapped devices are performed. Without this feature any external read would always read an entire L2 line of data. Each of the four CE spaces is dived in to four ranges, each of which maps the least significant bit of an MAR register. If an MAR register is set, the corresponding address range is cached by L2. At reset, MAR registers are set to 0. To begin caching data in the L2, the initialization of the appropriate MAR register to 1 is necessary. The MAR defines the cacheability for the EMIF only. Addresses accessed by the EMIF which are not defined by the MAR register are always cacheable. The following Table 15.3 shows the various CE spaces and the corresponding MAR registers to access that space.

All the memory space base address registers, word count registers and the fifteen memory attribute registers are memory mapped registers starting from the location 0184 0000h to 0184 82CCh. Before the memory access appropriate registers are to be initialized.

ON-CHIP PERIPHERALS 15.6

The ′C6X processors programmable on-chip peripherals are listed below.

- Two 32-bit timers
- Two Multichannel buffered serial ports (McBSPs)
- Direct memory access (DMA)/Enhanced Direct memory interface (EDMA)
- External memory interface (EMIF)
- Host-Port Interface (HPI)
- Boot configuration
- Interrupt selector
- Expansion bus

- Power down logic

All ′C6X processors have two McBSPs, but ‘6202 processor has three McBSPs. The ′C620X/′C670X family processors have DMA controllers where as ′C621X/′C671X processors are with EDMA controllers. The Expansion bus is available only in ′C6202 processor but HPI is not available in it. All other peripheral devices are available in all ′C6X processors. These peripherals are configured via a set of memory-mapped control registers. The peripheral bus controller performs the arbitration for accesses of on-chip peripherals. The boot configuration is interfaced through external signals only and the power down logic is accessed directly by the CPU. The block diagram of ′C6X processor with all on-chip peripherals are shown in Fig. 15.9.

Table 15.3 *MAR Registers and its corresponding CE space address range*

MAR	*Address Range Enabled*	*CE space*
15	B300 0000h – B3FF FFFFh	CE3
14	B200 0000h – B2FF FFFFh	CE3
13	B100 0000h – B1FF FFFFh	CE3
12	B000 0000h – B0FF FFFFh	CE3
11	A300 0000h – A3FF FFFFh	CE2
10	A200 0000h – A2FF FFFFh	CE2
9	A100 0000h – A1FF FFFFh	CE2
8	A000 0000h – A0FF FFFFh	CE2
7	9300 0000h – 93FF FFFFh	CE1
6	9200 0000h – 92FF FFFFh	CE1
5	9100 0000h – 91FF FFFFh	CE1
4	9000 0000h – 90FF FFFFh	CE1
3	8300 0000h – 83FF FFFFh	CE0
2	8200 0000h – 82FF FFFFh	CE0
1	8100 0000h – 81FF FFFFh	CE0
0	8000 0000h – 80FF FFFFh	CE0

15.6.1 Timers

The ′C6X devices have two 32-bit general purpose timers that are used to time events, count events, generate pulses, interrupt CPU and send synchronization event to DMA. The timer operation can be configured through three memory mapped registers namely timer control register, timer period register and timer counter register. The ′C6X processor on-chip timer block diagram is given in Fig. 15.10. The timer control register (TCR) is programmed to select the different modes of operation of timer; the timer period register contains the number of timer input clock cycle to count and the timer counter register increments when it is enabled to count. The timer counter register resets to 0 when the count reaches the count value in the period register.

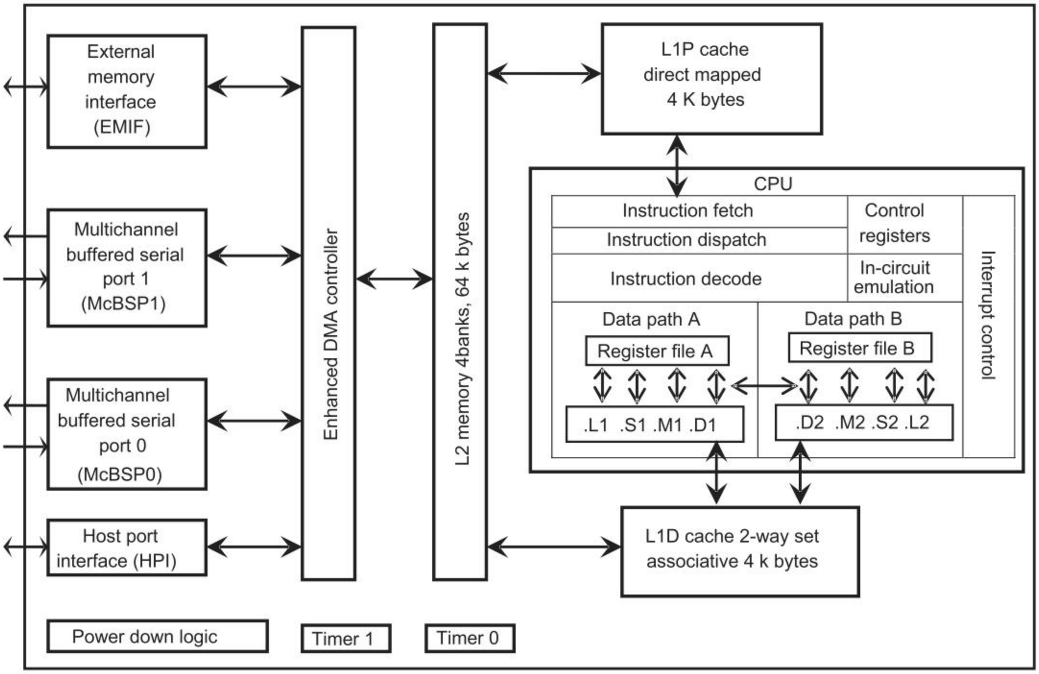

External memory interface (EMIF)
Multichannel buffered serial port 1 (McBSP1)
Multichannel buffered serial port 0 (McBSP0)
Host port interface (HPI)
Enhanced DMA controller
L2 memory 4banks, 64 k bytes
L1P cache direct mapped 4 K bytes
CPU
Instruction fetch
Instruction dispatch
Instruction decode
Control registers
In-circuit emulation
Interrupt control
Data path A
Register file A
.L1 .S1 .M1 .D1
Data path B
Register file B
.D2 .M2 .S2 .L2
L1D cache 2-way set associative 4 k bytes
Power down logic
Timer 1
Timer 0

Fig. 15.9 *TMS320C621X/ 'C671X block diagram with on-chip peripherals*

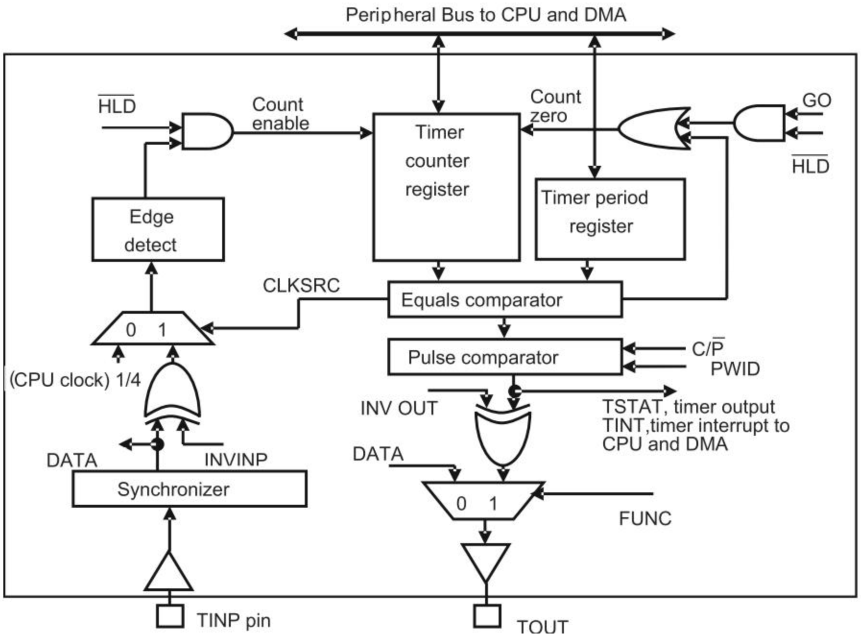


Fig. 15.10 *TMS320C6X Timer Block digaram*

The timer has two signaling modes, clock mode and pulse mode which can be selected by C/$\overline{P}$ bit in TCR. The timer has an input pin TINP and an output pin TOUT and these pins can function as timer clock input and output. These pins can also be configured for general purpose I/O pins respectively using FUNC bit in TCR. The timer functions with both internal clock signal from the CPU and also from the external clock, the clock source can be selected by CLKSRC pin in TCR. The start of the timer, holding it and resetting it are performed with GO and $\overline{\text{HLD}}$ pins in TCR. The frequency of the timer output when operated in clock and pulse modes are given below.

f_{clock} = f(clock source)/ (2* timer period register)
f_{pulse} = f(clock source)/ timer period register

15.6.2 Multichannel Buffered Serial Port (McBSP)

The multichannel buffered serial port is based on the standard serial port interface available in earlier TI processors. The McBSP has the following features:

- Provides full-duplex communication
- Multichannel transmit and receive up to 128 channels
- Data selection size of 8,12,16,20,24 and 32 bits
- Independent framing and clocking for receive and transmit
- External shift clock or an internal programmable frequency shift clock for data transfer
- 8-bit data transfer with the option of LSB or MSB first
- Programmable polarity for both frame synchronization and data clocks
- Double-buffered registers, which allow continuous data transmission
- Auto buffering capability through 5-channel DMA controller
- µ-law and A-law companding
- Direct interface to industry standard codecs, A/D, D/A converters, analog interface chips, T1/ E1, MVIP, H.100, SCSA framers, IOM-2, AC97, IIS complaint devices and SPI™ devices

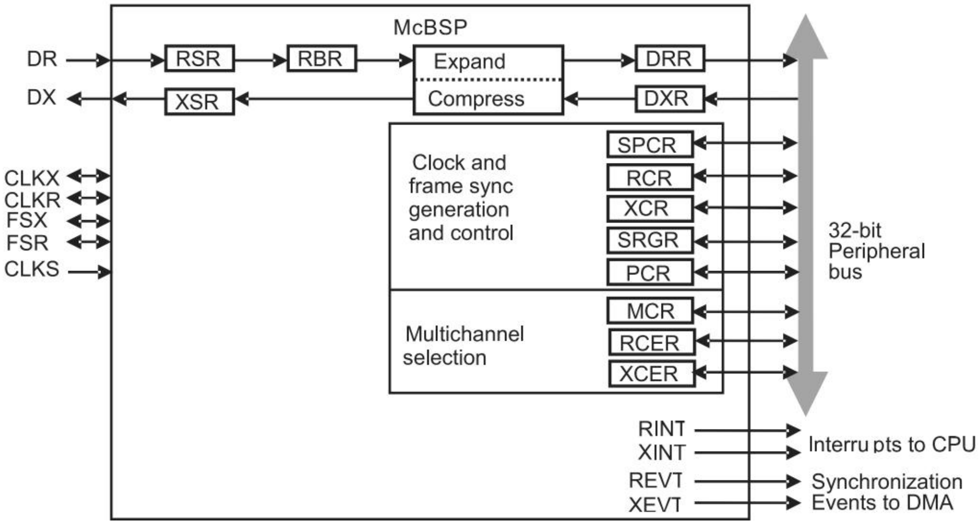


Fig. 15.11 *Multichannel buffered serial port (McBSP) block diagram*

The McBSP consists of two paths, a data path and a control path which is used to connect to external devices. The block diagram of McBSP is shown in Fig. 15.11. There are thirteen memory mapped registers for each McBSPs present in the processor and these registers are accessed via 32-bit peripheral bus. The list of registers and its memory mapped address are given in Table 15.4. The different modes of operation of McBSP are programmed through a 32-bit serial port control register (SCR).

The data communication in McBSP is through data transmit (DX) and data receiver (DR) pins. The clocking and frame synchronization are via CLKX, CLKR, FSX and FSR pins. Either CPU or DMA controller reads the received data from data receiver register (DRR) and also the data to be transmitted is written in data transmit register (DXR). The data transmit shift register (XSR) shifts out the data in DXR to DX pin and the same way the data received in DR pin is shifted into receive shift register (RSR) and copied into the receive buffer register (RBR) and then copied to DRR. The received data is read by the CPU or DMA controller.

Table 15.4 *McBSP memory mapped registers*

Memory mapped register address			*Abberivation*	*Register Name*
McBSP0	*McBSP1*	*McBSP2(in ′C6202 only)*		
—	—	—	RBR	Receive buffere register
—	—	—	RSR	Receiver shift register
—	—	—	XSR	Transmit shift register
018C 0000	0190 0000	01A4 0000	DRR	Data receiver register
018C 0004	0190 0004	01A4 0004	DXR	Data transmit register
018C 0008	0190 0008	01A4 0008	SPCR	Serial port control register
018C 000C	0190 000C	01A4 000C	RCR	Receive control register
018C 0010	0190 0010	01A4 0010	XCR	Transmit control register
018C 0014	0190 0014	01A4 0014	SRGR	Sample rate generator register
018C 0018	0190 0018	01A4 0018	MCR	Multichannel control register
018C 001C	0190 001C	01A4 001C	RCER	Receiver channel enable register
018C 0020	0190 0020	01A4 0020	XCER	Transmit channel enable register
018C 0024	0190 0024	01A4 0024	PCR	Pin control register

Note: RBR, RSR and XSR registers are not directly accessible via CPU or DMA controller

15.6.3 DMA/EDMA Controller

The direct memory access (DMA) controller is available in ′C620X/′C670X devices. The DMA controller transfers data between regions in the memory map without affecting the operation of CPU. The DMA controller is used to move data to and from internal memory, internal peripherals or external devices to occur in the background of CPU operation. The DMA controller has four independent programmable channels, allowing four DMA operations and also there is a fifth auxiliary channel to service requests from the host port interface (HPI). The DMA controller can access the following regions in the memory map.

- On-chip data memory
- On-chip program memory, if it is mapped into memory space rather than being used as cache
- On-chip peripherals
- External memory via EMIF
- Expansion memory via expansion bus

The enhance DMA (EDMA) controller is available in ′C621X/′C671X devices. The EDMA controller performs block transfer of data to/from internal memory, transfer requests from peripherals and between external memory spaces in parallel to CPU intensive operations. The EDMA controller has enhancements than DMA controller and it provides 16 channels with programmable priority and the ability to link data transfers. The EDMA operations are controlled by eight memory mapped EDMA control registers.

15.6.4 External Memory Interface (EMIF)

The external memory interface (EMIF) of ′C6X devices support a glueless interface to a variety of external devices. It can be used to interface synchronous as well as asynchronous devices such as SRAM, DRAM, ROM, FIFOs, FPGAs and external shared memory devices. The ′C620X/′C670X EMIF services requests of external bus from four requesters:

- The on-chip program memory controller that services CPU program fetches
- The on-chip data memory controller that services CPU data fetches
- The on-chip DMA controller
- An external shared-memory device controller

If multiple requests arrive at the same time, the EMIF prioritizes them and performs the necessary number of operations. The ′C621X/′C671X device services requests of the external bus from two requesters:

- An enhanced DMA controller
- An external shared-memory device controller

15.6.5 Host-Port Interface (HPI)

The host-port interface is a16-bit wide parallel port through which a host processor can directly access the CPU's memory space. The host device functions as a master to the interface, which increases the ease of access. The host and CPU can exchange information via internal or external memory. The host also has direct access to memory-mapped peripherals. The connectivity to the CPU's memory space is provided through the DMA controller. Both the host and CPU can access the HPI control register (HPIC). The host can access HPI address register (HPIA), HPI data register (HPID) and HPIC using the external data interface control signals.

15.6.6 Boot Configuration

The ′C6X devices use variety of boot configurations to determine what action the devices are to perform after the reset signal is initialized. Each ′C6X device has some or all of the following boot configuration options:

- Selection of memory map- to determine whether internal or external memory is mapped at address zero
- Selection of type of external memory mapped address zero, if external memory map is selected
- Selection of boot process used to initialize the memory at address zero before the CPU is released from reset.

The external pins BOOTMODE [4:0] are used to select the boot configuration. The values of the BOOTMODE are latched during the low period of $\overline{\text{RESET}}$

15.6.7 Interrupt Selector

The ′C6X peripheral set has up to 32 interrupt sources. The CPU has only 12 interrupts available for use. The interrupt selector allows the user to choose and prioritize the 12 of the 32 for the system needs.

The interrupt selector also allows to effectively change the polarity of the external interrupt inputs. The $\overline{\text{RESET}}$ and NMI are the non-maskable interrupts. The CPU interrupts are maskable. To mask the interrupts the global interrupt enable bit (GIE) in the control status register (CSR) is set to 1. To enable an interrupt the respecive bit in the interrupt enable register (IE) is set to 1. When the corresponding interrupt occurs, the bit in the interrupt flag register (IFR) is set and the CPU starts processing the interrupt.

15.6.8 Expansion Bus

The expansion bus is available only in ′C6202 processor. The expansion bus is 32-bit wide bus that is used to interface different types of asynchronous peripherals, asynchronous and synchronous FIFOs, PCI bridge chips and other external masters. The expansion bus offers a flexible bus arbitration scheme.

15.6.9 Power-down Logic

In CMOS logic circuits, power dissipation can be reduced by decreasing the switching from one logic state to another. By preventing some or all of the chip's logic from switching, significant power can be reduced without losing the data or operational context. PD1, PD2 and PD3 are three power-down modes available to perform this function. The PD1 mode blocks the internal clock inputs at the boundary of the CPU, preventing most of its logic from switching. PD1 effectively shuts down the CPU. The PD2 mode halts the entire on-chip clock structure at the output of the PLL. The PD3 mode is like PD2 mode but also disconnects the external clock source (CLKIN) from reaching PLL. In addition to these power-down modes, the IDLE instruction provides low CPU power consumption by executing continuous NOPs. The IDLE instruction terminates only upon servicing an interrupt.

Review Questions

15.1 List the steps to do programming in ′C6X tool.

15.2 What are the basic features of ′C6416 starter kit?

15.3 Explain the memory resources available in ′C6416 DSK.

15.4 What are the steps involved in ′C6X code generation using CCS tool?

15.5 Which instruction is used for division? How?

15.6 Explain the internal memory details of ′C6X processors.

15.7 For what operations L2 controller is used?

15.8 List the on-chip peripheral in ′C6X processors.

15.9 Explain the operation of ′C6X timer.

15.10 What are the features of McBSP?

15.11 For what interfaces McBSP is used?

15.12 List the signals used for clocking and frame synchronization of ′C6X McBSP.

15.13 What regions of memory map of ′C6X DMA controller can be used?

15.14 Explain the uses of EMIF.

15.15 What is the use of interrupt selector?

15.16 Why power-down logic is needed? Explain the ′C6X power-down logics.

Self Test Questions

15.1 The operating frequency of 'C6416 starter kit is ____
(a) 200 MHz (b) 720 MHz (c) 1GHz (d) 800 MHz

15.2 The size of on-chip RAM in 'C6416T processor is ___
(a) 64 K words (b) 16 K words
(c) 1024 K bytes (d) 64 K bytes

15.3 The Max. operating frequency of 'C6416T processor is ___
(a) 200 MHz (b) 720 MHz (c) 1GHz (d) 800 MHz

15.4 The size of external DRAM in 'C6416 starter kit is ____
(a) 1024 K bytes (b) 512 K bytes
(c) 8 M bytes (d) 16 M bytes

15.5 The size of flash memory in 'C6416 starter kit is ____
(a) 1024 K bytes (b) 512 K bytes
(c) 8 M bytes (d) 16 M bytes

15.6 The name of extension given for a project in CCS is ___
(a) .pjt (b) .mak (c) .asm (d) .out

15.7 The file extension name an assembly language file should have is ___
(a) .pjt (b) .mak (c) .asm (d) .out

15.8 The executable file name extension for a project is ____
(a) .pjt (b) .mak (c) .asm (d) .out

15.9 The starting memory address of 'C6X where the code is down loaded is ____
(a) 0x0000 0000h (b)0x0000 0200h
(c) 0x0000 0020h (d) 0x0000 F000h

15.10 The instruction used to perform division in 'C6X processor is ___
(a) ADDH (b) SUB (c) SUBC (d) ADDU

15.11 The 'C6X instruction used for the left most bit detection is ___
(a) LDB (b) LDHU (c) LMBD (d) LDH

15.12 ___ instruction is used to perform convolution in 'C6X processor.
(a) MPY & ADD (b) MPY & SUB
(c) MAC (d) MACD

15.13 The size of on-chip memory in all 'C6X processors except 'C6202 is ___
(a) 64 K words (b) 16 K words
(c) 1024 bytes (d) 64K bytes

15.14 The size of on-chip memory in 'C6202 processor is ___
(a) 64 K byes (b) 128 K bytes
(c) 256 K bytes (d) 64 K words

15.15 The size of program & data cache in 'C6X processor is ___
(a) 2K bytes (b) 2 K words
(c) 4 K bytes (d) 4K words

15.16 The number of external memory space in 'C6X processor is ___
(a) 2 (b) 5 (c) 4 (d) 3

15.17 The no. of McBSP in 'C6202 processor is ____
(a) 2 (b) 5 (c) 4 (d) 3

15.18 The expansion bus is available in ___ processor
(a) 'C6201 (b) 'C6202 (c) 'C6211 (d) 'C6711

15.19 The 'C6X processor without HPI is ___
(a) 'C6201 (b) 'C6202 (c) 'C6211 (d) 'C6711

15.20 The no. of on-chip timers in 'C6X processors is ___
(a) 2 (b) 5 (c) 4 (d) 3

15.21 The no. of channels the McBSP can transmit and receive is ___
(a) 64 (b) 128 (c) 32 (d) 200

15.22 The no. of DMA channels in 'C6X processor is __
(a) 2 (b) 5 (c) 4 (d) 3

15.23 The no. of EDMA channels in 'C6X processor is ___
(a) 4 (b) 8 (c) 16 (d)13

15.24 The Max. no. of interrupt sources present in 'C6X processor is ___
(a) 12 (b) 13 (c) 32 (d) 28

15.25 The no. of interrupt sources the CPU can use in 'C6X processor is ___
(a) 12 (b) 13 (c) 32 (d) 28

15.26 The no. of power-down logic modes in 'C6X processor is ___
(a) 4 (b) 2 (c) 5 (d) 3

16 ARCHITECTURE OF TMS320C55X PROCESSORS

INTRODUCTION 16.1

The architecture of TMS320C55X digital signal processor is based on the architecture of the earlier ′C54X processor and is source compatible with ′C54X. The ′C55X processor architecture is optimized for power efficiency, low system cost and best performance for low power applications. The ′C55X processor offers a cost effective solution in personal and portable processing applications and low power digital communication infrastructure. Compared to ′C54X processor, ′C55X processor has high end performance and dissipates one-sixth the core power dissipation of ′C54X.

The ′C55X core delivers twice the cycle efficiency that of ′C54X through a dual-MAC architecture with parallel instructions, additional accumulators, ALUs and data registers. The ′C55X device instructions are variable byte lengths ranging in size from 8 bit to 48 bits. With this feature, ′C55X devices can reduce control code size per function, hence reduced memory requirements and lower the system cost. The typical applications for ′C55X device are given below:

- Wireless handsets and personal communication systems
- Portable audio players
- Personal medical devices such as hearing aids
- Digital cameras
- Internet application
- Power efficient multichannel telephony systems

FEATURES OF ′C55X PROCESSORS 16.2

The ′C55X devices have advanced features that provide processing efficiency, low-power dissipation and ease of use. Some of the key features of the device are listed below:

- A 32x 16 instruction buffer queue
- Two 17-bit x 17-bit multiply and accumulate (MAC) units
- One 40-bit ALU
- One 16-bit ALU
- One 40-bit barrel shifter
- Four 40-bit accumulators

- Twelve independent buses
- User configurable IDLE domains

The features of ′C55X device are compared with ′C54X device and are listed in Table 16.1. The processing power and superior code density of ′C55X is its efficient implementation. The ′C55X processor uses variable length instruction encoding to achieve optimal code density and efficient bus usage. Multiple computational units are included to carry out computations in parallel, hence reduction in number of cycles required per operation. The dual MAC units perform two 17-bit x 17-bit MAC operations in a single cycle where the 40-bit ALU can be used to perform arithmetic and logic operations on 32-bit data or can be used to perform dual 16-bit operations. A second 16-bit ALU is used for general purpose arithmetic operations, which further increases the parallelism and adds flexibility. The ′C55X device is based on modified Harvard architecture, which has one program bus and three independent read and two independent write buses. The three read buses are used to bring operands simultaneously to various computational units. The high degree of parallelism and efficient instruction encoding maximize the overall processor efficiency without sacrificing its performance.

Table 16.1 *Comparison of features in ′C54X and ′C55X devices*

Feature	*′C54X*	*′C55X*
Multiply and accumulate unit (MAC)	1	2
Arithmetic and logic unit (ALU)	1 (40-bit)	1(40-bit)1(16-bit)
Accumulators	2	4
Auxiliary register unit (ARU)	2 (16-bit)	3 (24-bit)
Auxiliary registers	8	8
Program bus	1 (PB)	1 (PB)
Data Read buses	2 (CB and DB)	3 (BB, CB and DB)
Data Write buses	1 (EB)	2 (EB and FB)
Program word size	16 bits	8/16/24/32/40/48 bits
Data word size	16 bits	16 bits
Data registers	0	4
Memory space	Separate Program/Data space	Unified space

CPU ARCHITECTURE OF ′C55X 16.3

The CPU of ′C55X device consists of four important units, they are

- Instruction buffer unit (I – unit)
- Program flow unit (P – unit)
- Address data flow unit (A – unit)
- Data computational unit (D – unit)

The instruction buffer unit buffers and decodes the instructions of the application program. This unit has the decode logic that interprets the variable length instructions. The instruction buffer unit maintains a constant stream of tasks for the various computational units.

The program flow unit keeps track of the execution point of the program. This unit consists of hardware units used for efficient looping as well as dedicated hardware for branching, conditional execution and pipeline protection. This unit helps to reduce the number of processor cycles needed for program control changes such as branches and subroutine calls.

The address flow unit has the address pointers for data accesses during program execution. This unit has dedicated hardware units for managing the five data buses which keeps the data flowing to the various computational units. The address flow unit has an additional general purpose ALU for simple arithmetic operations.

The data computation unit is the heart of the DSP and it performs the arithmetic computations on the data being processed. It consists of two MAC units, the main ALU and the accumulator registers. It also consists of barrel shifter, rounding & saturation control unit and dedicated hardware to perform Viterbi algorithm. The block diagram of ′C55X CPU is given Fig. 16.1.

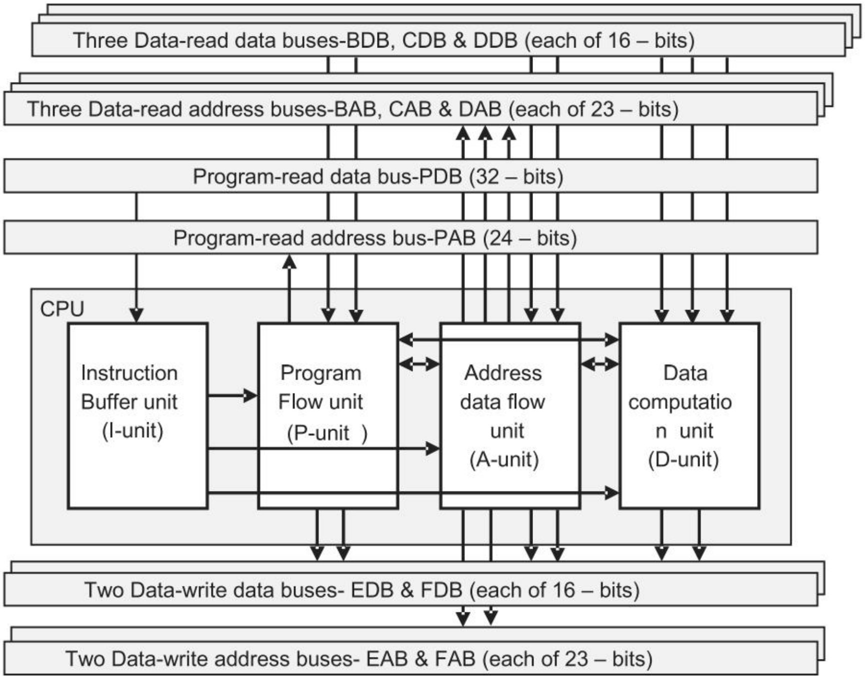

Three Data-read data buses-BDB, CDB & DDB (each of 16 – bits)
Three Data-read address buses-BAB, CAB & DAB (each of 23 – bits)
Program-read data bus-PDB (32 – bits)
Program-read address bus-PAB (24 – bits)
CPU
Instruction Buffer unit (I-unit)
Program Flow unit (P-unit)
Address data flow unit (A-unit)
Data computatio n unit (D-unit)
Two Data-write data buses- EDB & FDB (each of 16 – bits)
Two Data-write address buses- EAB & FAB (each of 23 – bits)

Fig. 16.1 *TMS320C55X Processor CPU Diagram*

16.3.1 Instruction Buffer Unit (I-unit)

The instruction buffer unit of ′C55X brings instruction stream from memory into the CPU. During each CPU cycle, the I-unit receives four bytes of program code from 32-bit program bus and places it in the instruction buffer queue. The instruction buffer queue can hold up to 64 bytes of code at a time and it is used to maintain continuous program flow to CPU. When the CPU is ready to decode the instruction, one to six bytes of code that were previously received in the queue are transferred to the instruction decoder. After decoding, the I-unit passes the data to P-unit, A-unit and D-unit. The block diagram of the I-unit is given Fig. 16.2.

In addition to helping with the pipelining of instructions, the instruction buffer queue with the help of local repeat instruction, is used to repeat or loop a block of codes stored in the queue. This techniques is extremely efficient in both performance and power consumption because once the code is loaded into the queue, no additional fetches are required to execute the loop. Another benefit of the instruction buffer queue is that it can perform speculative fetching of instructions while a condition is being tested for conditional program flow control instructions such as conditional branch, conditional call or conditional return. This minimizes the overhead due to program flow discontinuities by preventing the need to flush the pipeline. The cycles that are wasted to a pipeline flush are converted to useful processing cycles.

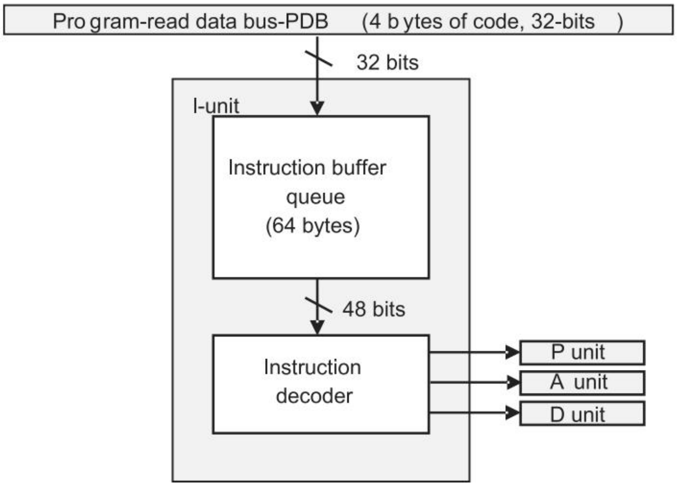

Pro gram-read data bus-PDB (4 b ytes of code, 32-bits)
32 bits
I-unit
Instruction buffer queue (64 bytes)
48 bits
Instruction decoder
P unit
A unit
D unit

Fig. 16.2 *Instruction Buffer Unit (I-unit) Diagram*

The instruction decoder identifies the instruction boundaries so that it can decode 8, 16, 24, 32, 40 bit instructions. It determines whether the CPU has been instructed to execute two instructions in parallel. The instruction decoder sends the decoded execution commands and immediate values to the P-unit, the A-unit and the D-unit. Even though the decoder typically decodes not more than 6 bytes at a time; there are cases in which it decodes 7 bytes for a single instruction such as that in k23 absolute addressing mode.

16.3.2 Program Flow Unit (P-Unit)

The program flow unit generates all program-space addresses for instruction fetches from program memory and also it controls the sequtence of instructions executed in a program. The P-unit directs operations such as hardware loops, branches and conditional execution. This unit also includes the logic for managing the instruction pipeline and four status registers to control and monitor various features of CPU. The block diagram of the P-unit is given in Fig. 16.3.

The program address generation logic in P-unit generates 24-bit addresses for instruction fetches from program memory and with 24-bit program memory address 16 M bytes of program code can be accessed. The P-unit normally generates sequential addresses using the program counter; however this logic also generates non sequential addresses for program control operations such as branches, calls, returns, repeat operations, and conditional operations and interrupt servicing. Once an address is generated, the memory access is done through program-read address bus (PAB). The program address generation logic can accept immediate data from I-unit and register values from D-unit.

The P-unit registers used for program flow, repeat operations, interrupts and the status registers are given below:

- Program flow registers

Program counter	- PC
Return address register	- RETA
Control flow context register	- CFCT

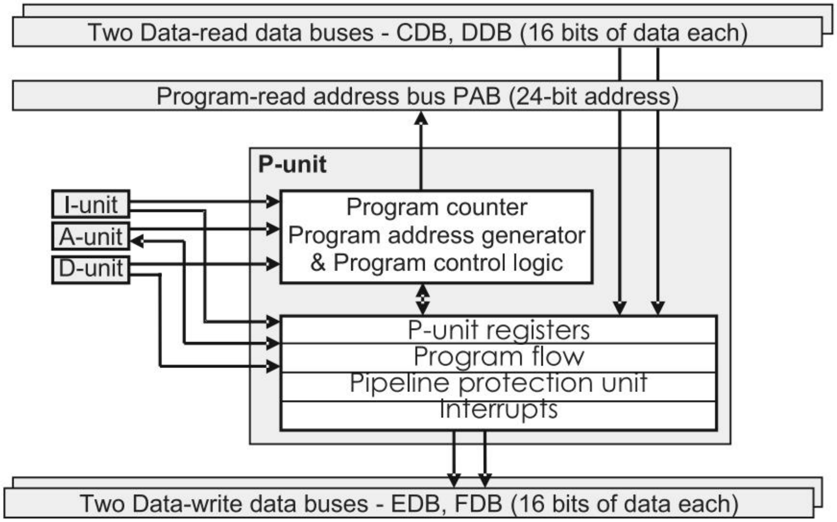

Two Data-read data buses - CDB, DDB (16 bits of data each)
Program-read address bus PAB (24-bit address)
P-unit
I-unit
A-unit
D-unit
Program counter
Program address generator
& Program control logic
P-unit registers
Program flow
Pipeline protection unit
Interrupts
Two Data-write data buses - EDB, FDB (16 bits of data each)

Fig. 16.3 *Program Flow Unit (P-unit) Diagram*

• Black-repeat registers	
Block-repeat counters 0 and 1	- BRC0 and BRC1
BRC1 save register	- BRS1
Block-repeat start address registers 0 and 1	- RSA0 and RSA1
Block-repeat end address registers 0 and 1	- REA0 and REA1
• Single-repeat registers	
Single-repeat counter register	- RPTC
Computed single-repeat register	- CSR
• Interrupt registers	
Interrupt flag registers 0 and 1	- IFR0 and IFR1
Interrupt enable registers 0 and 1	- IER0 and IER1
Debug interrupt enable registers 0 and 1	- DBIER0 and DBIER1
• Status registers	
Status registers 0, 1, 2 and 3	- ST0_55, ST1_55, ST2_55 & ST3_55

16.3.3 Address Data Flow Unit (A-Unit)

The address data flow unit contains all the logic and registers necessary to generate addresses for read and write accesses to data-space and I/O space. This unit can generate addresses for the three data-read address buses and the two data-write address buses. The A-unit also contains 16-bit ALU unit that can perform arithmetical, logical, shift and saturation operations. The block diagram of the A-unit is given in Fig. 16.4.

The data-address generation unit (DAGEN) generates all addresses for accessing data-space and I/O space. While generating addresses, it can accept immediate values from the I-unit and register values from A-unit. The P-unit indicates to DAGEN, whether to use linear or circular addressing for an instruction that uses and indirect addressing mode. The DAGEN unit contains three auxiliary register units (ARUs) with eight auxiliary registers (XAR0-XAR7) to do indirect addressing mode access and five circular buffers to do circular addressing mode access.

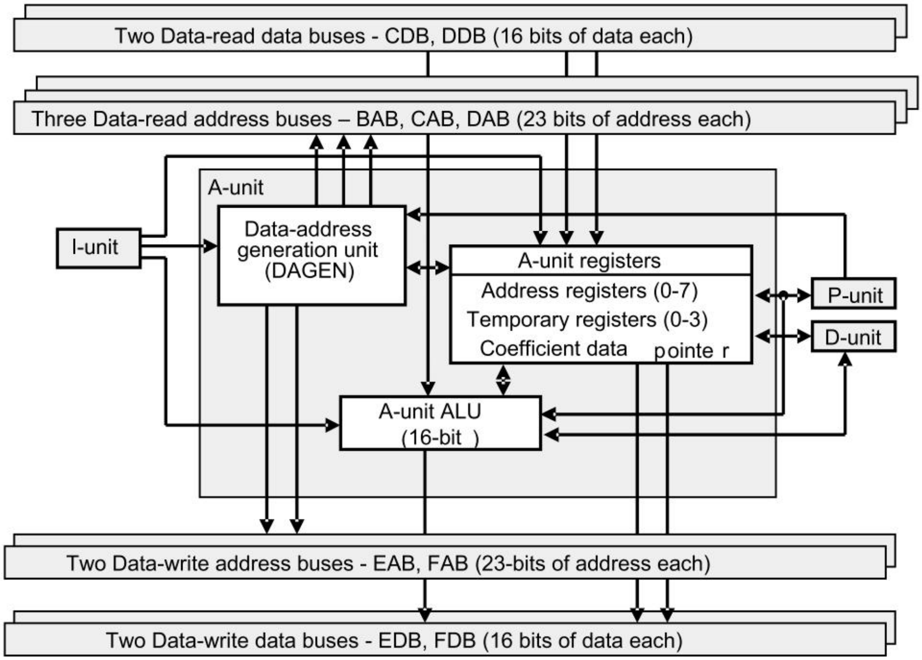

Two Data-read data buses - CDB, DDB (16 bits of data each)
Three Data-read address buses – BAB, CAB, DAB (23 bits of address each)
A-unit
I-unit
Data-address generation unit (DAGEN)
A-unit registers
Address registers (0-7)
Temporary registers (0-3)
Coefficient data pointe r
P-unit
D-unit
A-unit ALU (16-bit)
Two Data-write address buses - EAB, FAB (23-bits of address each)
Two Data-write data buses - EDB, FDB (16 bits of data each)

Fig. 16.4 *Address-Data Flow Unit (A-unit) Diagram*

The 16-bit ALU present in A-unit accepts immediate values from the I-unit and communicates bidirectionally with memory, I/O space, A-unit registers, D-unit registers and P-unit registers. The A-unit ALU performs the following operations:

- Performs additions, subtractions, comparisons, Boolean logic operations, signed and logical shift operations and absolute value calculations
- Tests, sets, clears and complements A-unit register bits and memory bits
- Modifies and moves register values
- Rotates register values
- Moves certain results from the shifter to an A-unit register

The list of A-unit registers is given below. All these registers can accept immediate data from I-unit and can accept or provide data to P-unit registers, D-unit registers and data memory. Within A-unit the registers have bidirectional connections with DAGEN unit and the 16-bit ALU.

• Data page registers	
Data page registers	- DPH and DP
Peripheral data page register	- PDP
• Pointer registers	
Coefficient data page registers	- CDPH and CDP
Stack pointer registers	- SPH, SP and SSP
Auxiliary registers	- XAR0-XAR7
• Circular buffer registers	
Circular buffer size registers	- BK03, BK47 and BKC
Circular buffer start address registers	- BSA01, BSA23, BSA45, BSA67 and BSAC

- Temporary registers
  Temporary registers 0, 1, 2 and 3 - T0-T3

16.3.4 Data Computation Unit (D-unit)

The data computational unit is the primary computational unit of the CPU where data is processed. The D-unit consists of two MAC (17-bitx17-bit) units, 40-bit ALU with four 40-bit accumulators (AC0-AC3) and shift registers. The three data-read buses feed these computational units. The block diagram of the D-unit is show in Fig. 16.5.

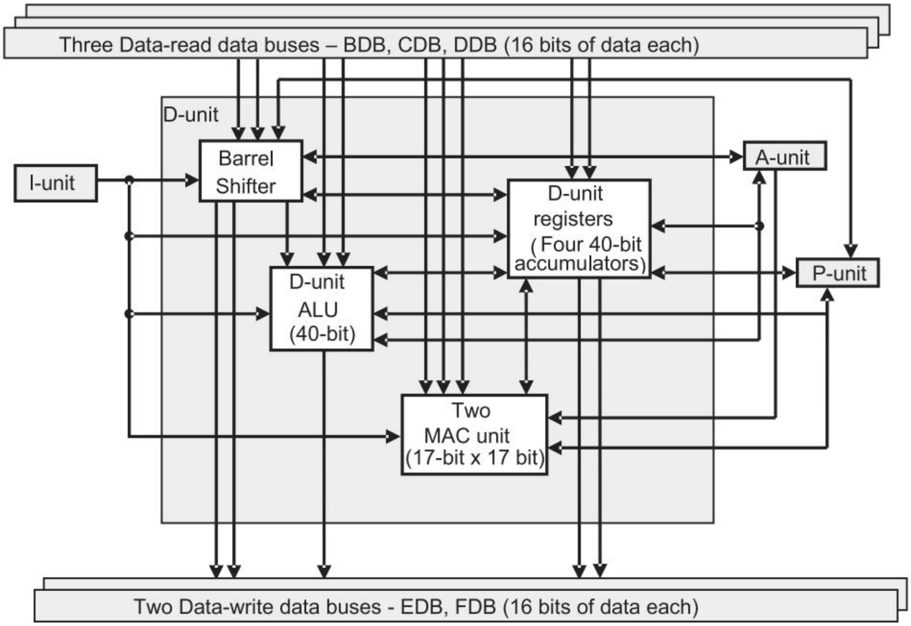

Three Data-read data buses – BDB, CDB, DDB (16 bits of data each)
D-unit
I-unit
Barrel Shifter
A-unit
D-unit registers (Four 40-bit accumulators)
P-unit
D-unit ALU (40-bit)
Two MAC unit (17-bit x 17 bit)
Two Data-write data buses - EDB, FDB (16 bits of data each)

Fig. 16.5 *Data Computation Unit (D-unit) Diagram*

The two MACs support multiplication and addition/subtraction operations. In a single cycle each MAC unit can perform fractional or integer 17-bit x 17-bit multiplication and a 40-bit addition or subtraction with optional 32/40-bit saturation. The four accumulators receive the results of the MACs. The MACs accept immediate values form I-unit, accept data values from memory, I/O space and A-unit registers. It can communicate bidirectionally with D-unit registers and P-unit registers.

The 40-bit ALU accepts immediate values from the I-unit and communicates bidirectionally with memory, I/O space, A-unit registers, D-unit register and P-unit registers. It can also receive results from shifter. The D-unit performs the following actions:

- Performs additions, subtractions, comparisons, rounding, saturation, Boolean logic operations and absolute value calculations
- Performs two arithmetical operations simultaneously when dual 16-bit arithmetic instruction is executed.
- Tests, sets, clears and complements D-unit register bits
- Moves register values

The D-unit shifter accepts immediate value from I-unit and communicates bidirectionally with memory, I/O space, A-unit registers, D-unit registers and P-unit registers. It also sends the shifted value to the D-unit ALU and A-unit ALU. The shift count value can be read from one of the temporary registers (T0-T3) or it can be represented as a constant in the instruction. The shifter can perform the following operations:

- Shifts 40-bit accumulator content up to 31-bits to the left and 32-bits to the right
- Shifts 16-bit immediate values up to 15-bits to the left
- Rotates register values
- Normalizes the accumulator values
- Extracts and expands bit fields and perform bit counting

The D-unit registers are Accumulators 0, 1, 2 and 3 (AC0-AC3) and Transition registers TRN0 and TRN1

16.3.5 Internal Address and Data Buses

The ′C55X CPU has one program bus and five data buses. The size of the program- read address bus is 24-bits and program-read data bus is 32-bits. Out of five data buses three buses are data-read buses and two are data-write buses. All the data-read & data-write address buses are of size 23-bits and data-read & data-write data buses are of 16-bits in size. The parallel bus architecture enables up to 32-bit program read, three 16-bit data reads and two 16-bit data write per CPU clock cycle. The functions of 12 CPU buses are given in Table 16.2.

Table 16.2 *Functions of ′C55X CPU buses*

Name of the Bus(es)	*Width*	*Function*
Program-read address bus (PAB)	24-bits	Carries a 24-bit byte address to read instructions from program space
Program-read data bus (PDB)	32-bits	Carries 4-bytes (32-bits) of program code from program memory to CPU
Data-read address buses(CAB & DAB)	23-bits	Carries 23-bit data memory address to read data values. DAB carries address for a read from data space or I/O space. CAB carries second address for dual operand read.
Data-read data buses(CDB & DDB)	16-bits	Carries 16-bit data value from memory to CPU. DDB carries data value from data space or I/O space. CDB carries second value during long data reads or dual data reads
Data-read address bus (BAB)	23-bits	Carries 23-bit data memory address for coefficient reads to internal memory only. This bus is used for instructions that use coefficient addressing mode.
Data-read data bus (BDB)	16-bits	Carries 16-bit coefficient data value from internal memory to CPU. BDB is not connected to external memory.
Data-write address buses(EAB & FAB)	23-bits	Carries 23-bit data memory address to write a data value. EAB carries address for a write to data space or I/O space. FAB carries a second address for dual data writes.
Data-write data buses(EDB & FDB)	16-bits	Carries 16-bit data value from CPU to memory. EDB carries data value to data space or I/O space. FDB carries a second data value during long data writes and dual data writes.

MEMORY ARCHITECTURE 16.4

The ′C55X processor has 16M bytes of unified memory for program/data space and a 64K words (16-bits) of separate memory for I/O space. The 24-bit program memory address is used to read program code from memory. The program memory access is byte access. For accessing the data space 23-bit data memory address is used. The data memory access is word access (16-bits). In both program memory and data memory access the address busses carry 24-bit values, but during the data space access, the least significant bit (LSB) on the address but is forced to 0.

The ′C55X processor uses byte addresses to fetch instructions of size 8, 16, 24, 32, 40 and 48 bits. When instructions are stored in program memory, the user need not have to align them but the instruction fetches are aligned to even-address 32-bit boundaries.

The CPU uses word addresses to read or write data values of size 8, 16 or 32-bit values. The address that needs to be generated for a particular value depends on how it is stored within the word boundaries in data space.

The data-space is divided into 128 data pages where each data page has 64K addresses. An instruction to access a location in the data page concatenates a 7-bit data page value and a 16-bit offset from instruction. On data page 0, 96 addresses are reserved for memory-mapped registers (MMR).

I/O space is separate from program/data space and is available only accessing registers of the peripherals on the DSP. The word address is used access the 64K locations. The CPU uses the bus DAB for data reads and the bus EAB for data writes to I/O space.

The ′C55X processor has an on-chip boot loader, which provides option for transferring code and data from an external source to the RAM inside the processor at power up/reset.

ADDRESSING MODES 16.5

The ′C55X processors support the following three types of addressing modes that are used to access data memory, memory-mapped registers, register bits and I/O space:

- Register addressing mode
- Immediate addressing mode
- Absolute addressing mode
- Direct addressing mode
- Indirect addressing mode

16.5.1 Register Addressing Mode

The ′C55X processor has four accumulators AC0-AC3 and four temporary registers T0-T3. The register addressing mode uses these eight registers for source (src) as well as destination (dst) operands to perform operations. Examples for register addressing mode are given below:

(i) ADD ACx,ACy
(ii) ADD Tx,Ty
(iii) MPY Tx,ACx,ACy
(iv) ADDR ACx,ACy
(v) ADDV ACx,ACy

16.5.2 Immediate Addressing Mode

The ′C55X processor supports immediate addressing mode. The 4-bit unsigned constant (k4)/8-bit signed constant (k8)/16-bit signed constant can be used as source operand, the destination can be any of the accumulators (AC0-AC3). For some instructions the destination can be temporary registers (T0-T3). Examples for the immediate addressing mode are given below. The symbol used to represent immediate addressing is # after the mnemonic.

```
ADD #k4,dst
ADD #k16,src,dst
MPYK #k8,ACx,ACy
MPYK #k16,ACx,ACy
```

In this addressing mode, the immediate operand can be left shifted by 4-bit (1-16) shift value specified in the instruction and the respective operation can be performed, example for prescaled immediate addressing mode instruction is given below:

```
ADD #k16<<#shift,ACx,ACy
```

16.5.3 Absolute Addressing Mode

The absolute addressing mode is used to access a memory location by supplying all or part of an address as a constant in an instruction. There are three modes of absolute addressing available, they are:

- k16 absolute addressing
- k23 absolute addressing
- I/O absolute addressing

k16 Absolute Addressing The k16 absolute addressing mode uses the source operand syntax *abs16(#k16), where k16 is a 16-bit unsigned constant and the destination can be any of the accumulators. The 7 MSBs of data page register (DPH) and k16 specified in the source operand are concatenated to form a 23-bit data-space address. The 7 MSBs of DPH are the MSBs and k16 value is the LSBs of the data-space address. This mode can be used to access a memory location or a memory-mapped register. Example for this addressing mode is given below:

```
ADD *abs16(#1000h),ACx
```

k23 Absolute Addressing The k23 absolute addressing mode uses the source operand syntax *(#k23), where k23 is a 23-bit unsigned constant and the destination can be any of the accumulators. The 23-bit constant specified in the instruction is used as the data-space address. This mode can be used to access a memory location or a memory-mapped register. Example for this addressing mode is given below:

```
ADD *(#201000h),ACx
```

I/O Absolute Addressing In the case of mnemonic instruction type, the I/O addressing mode is provided by the syntax port(#k16) by port() operand qualifier and for the algebraic instruction set it is provided by the syntax *port(#k16), where k16 is a 16-bit unsigned constant. Since the I/O space is 64K words 16-bits are used to specify the I/O space address.

16.5.4 Direct Addressing Mode

The direct addressing mode of ′C55X is to access the data-space using data page register (DPH) & stack register (SPH) and to access I/O space using peripheral data page register (PDP). The direct addressing mode has the following modes:

- DP direct addressing
- SP direct addressing
- Register-bit direct addressing
- PDP direct addressing

The direct addressing mode with DPH and SPH are mutually exclusive and the mode selection depends on the compiler mode bit (CPL) in status register ST1_55.

16.5.4.1 DP Direct Addressing Mode

The DP direct addressing mode uses the extended data page pointer (XDP) for data-space address calculation. The concatenation of DPH and DP is called the extended data page register (XDP).

XDP (23-bits) = DPH (7-bits):DP(16-bits)

To access the data-space of ′C55X device 23-bit data address is to be generated. At run-time, the 23-bit data address is generated as given below:

Data-space address (23-bits) = DPH:(DP+Doffset)

The data-space is divided into 128 pages and each page has 64K addresses. To select one of the 128 main pages (0-127), 7-bits are required and it is obtained from MSBs of DPH register. The remaining 16-bit LSBs of the address is obtained by the summation of the value in the data page register (DP) and the 7-bit offset (Doffset) calculated by the assembler.

The value of DP can be from 0000h – FFFFh. The 7-bit offset value is calculated by the assembler as given below:

Doffset = (Daddr- current data page value-DP) & 7Fh

Where Daddr is the 16-bit value specified in the direct addressing mode instruction. The current data page pointer (DP) value is subtracted from Daddr value, to the result bit wise AND operation is performed with 7Fh to obtain the Doffset value. Example for direct addressing mode is given below:

```
AMOV #012000h,XDP   ;the main data page 01 loaded in DPH, 2000h loaded in DP
ADD @2010h,AC0      ;Daddr specified in the instruction is 2010h
```

Doffset = (Daddr-DP) & 7Fh = (2010-2000) & 7Fh = 0010h **(& - bit wise AND operation)**

23-bit data address = DPH:(DP+Doffset) = 012010h

The symbol used for direct addressing mode is @ after the mnemonic, followed by 16-bit Daddr value is specified. The content of data memory address 012010h is added to the content of AC0, result stored in AC0. Loading DPH and DP can be done individually or XDP can be loaded using instrucions.

16.5.4.2 SP Direct Addressing Mode

The SP direct addressing mode uses the extended data stack pointer (XSP) for data-space address calculation. The concatenation of SPH and SP is called the extended data stack pointer (XSP).

XSP (23-bits) = SPH (7-bits):SP(16-bits)

At run-time, the 23-bit data address is generated using XSP as given below:

Data-space address (23-bits) = SPH:(SP+Doffset)

The 7 MSBs of the SPH register is used to point the main data page, the 16 LSBs of the data address is calculated by the summation of the 16-bit SP value and the 7-bit offset (Doffset) specified in the instruction itself. The offset value can be from 0-127. Loading SPH and SP can be done individually or XSP can be loaded using instrucions. In main page 0, 96 locations are reserved for memory mapped registers (MMR), so in SP direct addressing other than these locations of the main page 0 (00 0060h- 00 FFFFh) are to be used.

16.5.4.3 Register-bit Direct Addressing Mode

The register-bit direct addressing mode is to access bits in accumulators AC0-AC3, temporary registers T0-T3 and the auxiliary registers AR0-AR7 only. The register bit test/set/clear/complement instructions support this addressing mode. The offset in the operand field is specified by the syntax @bitoffset. The offset specified in the instruction is the bit value from the LSB of the respective register used in the instruction. Example for register-bit direct addressing mode is given below:

BSET @5,AC0 ; the LSB 5th bit in Accumulator AC0 is set to 1

16.5.4.4 PDP Direct Addressing Mode

The PDP direct addressing mode is used to access the I/O space. The 64K words of I/O space are divided into 512 pages and each page contains 128 words. To specify the 512 pages 9-bits are required and it is selected using peripheral data page register (PDP). The 7-bit offset (Poffset) required to select a word in a page is specified in the instruction. The concatenation of PDP and Poffset is used to generate the address for the I/O space.

16.5.5 Indirect Addressing Mode

The ′C55X processor supports indirect addressing mode, this addressing mode uses the auxiliary register units present in the address data flow unit (A-unit) along with eight auxiliary registers (AR0-AR7). The indirect addressing mode can be used for linear addressing or circular addressing. The following are the types of indirect addressing modes:

- AR indirect
- Dual AR indirect
- CDP indirect
- Coefficient indirect

16.5.5.1 AR Indirect Addressing Mode

The AR indirect addressing mode uses an auxiliary register ARn (n = 0-7) to point the data address. The size of auxiliary register is 16-bits and it is used to access the data within one data page (64K). The main page can be selected using higher order 7-bits of the ARnH. The concatenation of ARnH and ARn is called extended auxiliary register XARn. The 23-bits of XARn are used to access the data-space. To access a data space location an instruction that loads XARn is used. The ARn register can be individually loaded but not ARnH. The types of addressing mode operands available in this mode depend on the auxiliary register mode switch bit (ARMS) in status register ST2_55. The format of ST2_55 is given in Fig. 16.6.

ARMS=0: **DSP mode – the addressing mode operands are used for the DSP intensive applications**
ARMS=1: **Control mode – the addressing mode operands are used for control system applications**

The addressing mode operands for DSP mode and control mode are given in Table 16.3. The following are the three important points to remember in AR indirect addressing:

(i) The ARn modification linear or circular depends on the configuration bit ARnLC in status register ST2_55 (ARnLC = 0: linear addressing & ARnLC = 1: circular addressing). Normally ARnLC will be zero; to activate circular addressing this bit is to be set 1 with BSET instruction.

(ii) The address increment and decrement are made to 16-bit ARn contents only i.e. only within main data page. The data memory access across the main pages are not possible; to do that full 23-bit in XARn register is to be modified.

(iii) The index addressing and bit-reversed addressing use the content of T0/AR0 register for address increment/decrement. Using the content of T0 or AR0 depends on C54 compatibility mode bit (C54CM) in status register ST1_55 (C54CM = 0: T0 content used and C54CM = 1: AR0 content used). The default value of C54CM bit is 0, hence to use T0 in ′C55X, clear the C54CM bit using BCLR C54CM instruction.

Examples for AR indirect addressing mode are given below:

```
ADD *AR3+,T0,T1          ;the memory content pointed by AR3 added with T0, result in T1
ADD uns(*AR3),AC0,AC     ;the unsigned content pointed by AR3 added with AC0, result in AC1
BCLR C54CM               ;clear the C54CM bit in status register ST1_55.
ADD *(AR3+T0), AC0       ;the memory content pointed by AR3 added with AC0, result in AC0
                         ;the content of AR3 added with content of T0, result stored in AR3
```

15	14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARMS	Res.	DBGM	EALLOW	RDM	Res.	CDPLC	AR7LC	AR6LC	AR5LC	AR4LC	AR3LC	AR2LC	AR1LC	AR0LC

R/W-0 R/W-11 R/W-1 R/W-0 R/W-0 R-0R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0

0-7	– ARnLC	: ARn linear/circular mode configuration bits
8	– CDPLC	: CDP linear/circular mode configuration bit
9	– Reserved	: 0 all ways
10	– RDM	: Rounding mode bit
11	– EALLOW	: Emulation access enable bit
12	– DBGM	: Debug mode bit
13 -14	– Reserved	: 11b all ways
15	– ARMS	: Auxiliary register mode switch

Fig. 16.6 *Format of Status register -ST2_55*

16.5.5.2 AR Indirect Access of Register Bits

The AR indirect addressing mode is also used to access a register bit. The bits of accumulators AC0-AC3, temporary registers T0-T3 and the auxiliary registers AR0-AR7 can be accessed through register bit test/set/clear/complement instructions. The content of ARn contains the bit to be accessed and the corresponding bit is accessed from LSB of the register to be accessed. Example for register-bit access using indirect addressing mode is given below:

BSET AR0,AC0 ;if content of AR0=5,the LSB 5th bit in Accumulator AC0 is set to 1

16.5.5.3 AR Indirect Accesses of I/O Space

The AR indirect addressing mode is also used to access I/O space. The words in I/O space are accessed using 16-bit addresses. When AR indirect addressing mode is used the 16-bit content ARn is used to access the I/O space. The content of ARn is the I/O space memory address.

16.5.5.4 Dual AR Indirect Addressing Mode

The dual AR indirect addressing mode is used to access two data-memory locations using eight auxiliary registers AR0-AR7. The dual addressing mode can be used for linear or circular data accesses and it is based on the ARnLC bit in status register ST2_55. The address increment and decrement are made to 16-bits within main page only. The dual AR addressing mode can be used to two 16-bit data access for a single instruction or for two instructions that are executed in parallel. The syntax of dual operand instruction and parallel instruction is given below:

```
ADD Xmem,Ymem,ACx            ;dual operand access
ADD Smem,dst                 ;parallel instruction, first Smem is Xmem &
|| AND Smem,src,dst          ;second Smem is Ymem
```

The ARMS bit in status register does not affect the dual AR indirect addressing mode. The dual AR addressing mode operands supported in dual AR addressing mode are *ARn, *ARn+,*ARn-, *ARn(T0/AR0), *(ARn+T0/AR0), *(ARn-T0/AR0), *(ARn+T1) and *(ARn-T1) only. Examples for dual AR addressing mode are given below:

```
ADD *AR1,*AR2-,AC0      ;the data memory content pointed by AR1&AR2 are added, result stored
                        ;in AC0, after address generation AR2 content decremented
MPY *AR2,AC0,AC1        ;the multiply and add instructions are executed parallel
|| ADD *AR1,T0,T1       ;the data memory content pointed by AR2 & AR1 are accessed
```

Table 16.3 *Addressing mode operands for AR Indirect addressing mode*

Addressing mode operand	*ModeType*	*Description*
*ARn	DSP mode/ control mode	The 16-bits of ARn used for data memory address generation. ARn is not modified
*ARn+	DSP mode / control mode	The 16-bits of ARn used for data memory address generation, ARn is incremented after address generation (Post increment)
*ARn-	DSP mode / control mode	The 16-bits of ARn used for data memory address generation, ARn is decremented after address generation (Post decrement)
*+ARn	DSP mode only	The 16-bit ARn value is incremented before address generation, the incremented value is used for address generation (Pre-increment)
*-ARn	DSP mode only	The 16-bit ARn value is decremented before address generation, the decremented value is used for address generation (Pre-decrement)
*ARn(T0/AR0)	DSP mode / control mode	Index addressing. The 16-bit ARn content used as a base pointer. The 16-bit signed constant in T0/AR0 is used as an offset from the base pointer
*(ARn+T0/AR0)	DSP mode / control mode	Index addressing (Post increment). The 16-bits of ARn used for address generation. The 16-bit signed constant in T0/AR0 added to ARn after address generation
*(ARn-T0/AR0)	DSP mode / control mode	Index addressing (Post decrement). The 16-bits of ARn used for address generation. The 16-bit signed constant in T0/AR0 subtracted from ARn after address generation
*(ARn+T0B/ AR0B)	DSP mode only	Bit-reversed addressing (Post increment). The 16-bits of ARn used for address generation. The 16-bit signed constant in T0/AR0 added with reverse carry propagation to ARn after address generation to create bit-reversed addressing.
*(ARn-T0B/ AR0B)	DSP mode only	Bit-reversed addressing (Post decrement). The 16-bits of ARn used for address generation. The 16-bit signed constant in T0/AR0 added with reverse carry propagation is subtracted from ARn after address generation to create bit-reversed addressing.
*ARn(T1)	DSP mode only	Index addressing with T1. The 16-bit ARn content used as a base pointer. The 16-bit signed constant in T1 is used as an offset from the base pointer

(Contd.)

Table 16.3 *(Contd.)*

*(ARn+T1)	DSP mode only	The 16-bits of ARn used for data memory address generation. The 16-bit signed constant in T1 is added to ARn after address generation.
*(ARn-T1)	DSP mode only	The 16-bits of ARn used for data memory address generation. The 16-bit signed constant in T1 is subtracted from ARn after address generation.
*ARn(#k16)	DSP mode / control mode	The 16-bit ARn content used as a base pointer. The 16-bit signed constant k16 is used as an offset from the base pointer
*+ARn(#k16)	DSP mode / control mode	The 16-bit signed constant k16 is added to the content of ARn before address generation. The added value is used for address generation
*ARn(short(#k3))	Control modeonly	The 16-bit ARn content used as a base pointer. The 3-bit signed constant k3 is used as an offset from the base pointer

Note: For 16-bit data size ARn increment/decrement for one location (ARn+1) and for 32-bit data size ARn increment/ decrement for two locations (ARn+2)

16.5.5.5 CDP Indirect Addressing Mode

The CDP indirect addressing mode uses coefficient data pointer (CDP) to point data. This addressing mode is similar to AR indirect addressing mode and it can be used to access data space, register bits and I/O space.

Accesses of Data Space The 7-bits required to point the main page is supplied by 7 MSBs of CDPH and 16 LSBs to access the location in a data page is obtained from CDP register. The concatenation of CDPH and CDP is called extended coefficient data pointer (XCDP). The 23-bit XCDP value is the data memory address. The same way as AR indirect addressing mode in this addressing mode also the address modification can be linear or circular. The linear or circular address modification is decided by the CDP configuration bit (CDPLC) in status register ST2_55. Default value is linear addressing; to activate circular addressing CDPLC bit is to be set. The various CDP indirect operands supported in this addressing mode are given in Table 16.4. Examples for CDP indirect addressing mode are given below:

```
ADD *CDP+,AC0,AC1      ; the data memory content pointed by CDP register is added to AC0,
                       ; result stored in AC1
```

The address increments and decrements are made with the main data page only using the 16-bit CDP value.

Table 16.4 *Addressing mode operands for CDP Indirect addressing mode*

Addressing mode operand	*Description*
*CDP	The 16-bits of CDP used for data memory address generation. CDP is not modified after address generation
*CDP+	The 16-bits of CDP used for data memory address generation, CDP is incremented after address generation (Post increment)
*CDP-	The 16-bits of CDP used for data memory address generation, CDP is decremented after address generation (Post decrement)
*CDP(#k16)	The 16-bit CDP content used as a base pointer. The 16-bit signed constant k16 is used as an offset from the base pointer
*+CDP(#k16)	The 16-bit signed constant k16 is added to the content of CDP before address generation. The added value is used for address generation

Note: For 16-bit data size CDP increment/decrement for one location (CDP+1) and for 32-bit data size CDP increment/decrement for two locations (CDP+2)

Accesses of Register Bits The CDP indirect addressing mode can be used to access a register bit. The CDP register contains the bit number to be accessed in the register. The bits of accumulators AC0-AC3, temporary registers T0-T3 and the auxiliary registers AR0-AR7 can be accessed through register bit test/set/clear/complement instructions. Example for CDP indirect access to register bit is given below:

BSET CDP,T0 ; if the content of CDP is 7, the LSB 7th bit of T0 is set to 1

Accesses of I/O Space The CDP indirect addressing mode is used access the I/O space. The 16-bit content of CDP is used to access the entire 64K I/O space.

16.5.5.6 Coefficient Indirect Addressing Mode

The coefficient indirect addressing mode is used only for the data space access with three memory operands per cycle. It uses both dual AR indirect addressing and CDP indirect addressing to access three operands. The three operands are represented as Xmem, Ymem and Cmem, where Xmem and Ymem are accessed by AR indirect addressing through CB & DB data memory read buses and Cmem is accessed by CDP indirect addressing through BB data read bus. This addressing mode supports the following instructions:

- Memory move/initialization
- Multiply
- Multiply and accumulate (MAC)
- Multiply and subtract
- Dual multiply
- Dual multiply and accumulate/subtract

The address generation can be linear or circular according to the pointer configuration bit in status register ST2_55. The increments and decrements of address are within the main data page only. The address modification operands pertaining to dual AR indirect addressing are used. As for as the CDP indirect addressing is concerned only the address modification operands *CDP, *CDP+, *CDP- and *(CDP+T0/AR0) are used. Examples for coefficient indirect addressing mode are given below:

```
MPY *AR2,*CDP,AC0      ;multiply and multiply accumulate instruction are executed in parallel
|| MAC *AR3,*CDP,AC1   ;the data memory pointed by AR2,AR3 &CDP are accessed, result
                       ;stored in respective accumulators
```

For parallel instructions either the symbol || or :: can be used in ′C55X. The BB bus is not connected to external memory; hence Cmem operand access through BB must in internal memory.

16.5.5.7 Circular Addressing Mode

The AR indirect addressing mode and CDP indirect addressing mode can be used for circular addressing by setting the ARnLC and CDPLC configuration bits in ST2_55 to 1 respectively. The ′C55X processor can be configured for five circular buffers. The size of the circular buffer is defined by one of the three registers BK03, BK47 and BKC. To specify the start address of the circular buffer one of the five registers BSA01, BSA23, BSA45, BSA67 and BSAC are used. The ARn register or CDP register can be used as a pointer for the circular buffer. The ARn & CDP pointer registers and the corresponding register to specify the start address and buffer size are given in Table 16.5. The five circular start address registers and three buffer size registers are each 16-bits in size.

Each address within the circular buffer is 23-bits; the 7 MSBs are used to access the main data page and 16-LSBs to access the locations in the page. To set the main data page ARnH and CDPH are loaded. To access the locations in the page ARn register and CDP registers are loaded.

Table 16.5 *Pointers, Buffer start address & Buffer size registers for Circular addressing*

Circular address pointer	*Buffer start address register*	*Buffer size register*
AR0	BSA01	BK03
AR1	BSA01	BK03
AR2	BSA23	BK03
AR3	BSA23	BK03
AR4	BSA45	BK47
AR5	BSA45	BK47
AR6	BSA67	BK47
AR7	BSA67	BK47
CDP	BSAC	BKC

The steps for implementing the circular buffer are as follows:

(i) Load the size of the circular buffer in an appropriate buffer size register based on the pointer used. For example, if the buffer size is 16 and the pointer used is AR2, the value 16 is to be loaded in register BK03.

(ii) Set the appropriate circular buffer configuration bit ARnLC or CDPLC in ST2_55 using BSET instruction. For AR2 pointer to be configured in circular addressing mode, use BSET AR2LC instruction.

(iii) Load main data page value in the appropriate XARn or XCDP registers. For example, if the main data page is 7 and the circular pointer being AR2, the MSB 7-bits of XAR2 is to be loaded with 7. If CDP is used as pointer 7 MSBs of XCDP register is to be loaded.

(iv) Load the appropriate buffer start address register. For example for AR2 pointer the buffer start address is to be loaded in register BSA23. The concatenation of the 7-bit main data page information and 16-bit buffer start address (BSA) defines the 23-bit start address of the buffer in data memory.

(v) Load the selected pointer ARn or CDP, with a value from 0 to (buffer size -1). For buffer size 16, AR2 being pointer, load a value less than 15 in AR2.

In circular buffer the address increment and decrements must be within the 16-bit content of ARn or CDP. Example program to initialize and access the circular buffer is given below. The program is access the array of 16 numbers present in the memory location 021000h to 02100Fh repeatedly. The size of buffer is 16, pointer used is AR2. The data page value is 02.

Example (Initializing and accessing the circular buffer in 'C55X)

```
.text
MOV #16, BK03          ; the size of the buffer 16 loaded in BK03 register
BSET AR2LC             ; AR2LC bit is set for AR2 circular addressing mode
AMOV #020000h, XAR2    ; the main data page value 02 loaded in 7-bit AR2H
MOV #1000h,BSA23       ;the start address of the buffer 1000h loaded in BSA23
MOV #0000h,AR2         ;0h is loaded in register AR2
ADD *AR2+,AC0  ⎫       ;the address pointed by AR2 is accessed, the content of-
 .             ⎪       ;memory location 021000h is added to AC0 content,-
 . 15 instructions⎨     ;result in AC0. The content of AR2 incremented to-
 .             ⎪       ;1001h. After the 16th instruction the address in AR2-
ADD *AR2+,AC0  ⎭       ;will point the first location 1000h
 .end
```

16.5.6 Memory Mapped Register Access

The first page of the data memory space is mapped to 96 CPU registers of ′C55X. The memory mapped registers can be accessed using immediate addressing, absolute addressing, direct addressing, indirect addressing modes and using memory mapped register access qualifier.

MMR access through immediate addressing: The #k4 and #k16 immediate addressing operands can be used to access the MMR registers.

```
MOV #9h,BKC ; the immediate value 9h is loaded in register BKC
MOV #0FFFFh,AR1 ;the immediate value FFFFh is loaded in register AR1
```

MMR access through Absolute addressing: The k16 absolute addressing mode with DPH=0 and k23 absolute addressing mode can be used to access MMR registers.

Examples

```
MOV *abs(#10h),T0 ;the 10h location of first page is mapped to AR0, the content
     ;of AR0 is moved to T0 register
MOV *(#ACOL),T1 ;The 16 LSBs of accumulator AC0 is moved to T1 register
```

MMR access through Direct addressing: The DP direct addressing mode with CPL bit =0 & DPH=0 can be used to access MMR registers. Example for MMR access

Example

```
MOV @AC0L,T2 ; the 16 LSBs of accumulator AC0 is moved to T2 register
```

MMR access through indirect addressing: The AR indirect addressing and CDP indirect addressing modes can be used to access MMR registers with ARnH and CDPH values being zero respectively. The content of ARn and CDP must contain the address of the memory mapped register present in the first main page.

Examples

```
MOV #08h,AR2 ;the memory mapped address for AC0L register is 08h, loaded in AR2
MOV *AR2,T2 ;the 16 LSBs of AC0L is moved to register T2
MOV #0Bh,CDP ;the memory mapped address for AC1L register is 0Bh, loaded in CDP
MOV *CDP+,T3 ;the 16 LSBs of AC1L is moved to register T3
```

MMR access through access qualifier: The memory mapped register access qualifier mmap() can be used in direct addressing mode to access memory mapped registers. This access qualifier forces the address-generation unit (DAGEN) to make DPH =0, CPL = 0 and DP = 0 only for the memory mapped register access instruction.

Example

```
MOV mmap(@AC0L),T2; the 16 LSBs of accumulator AC0 is moved to T2 register
```

ASSEMBLY LANGUAGE INSTRUCTIONS 16.6

The ′C55X processor has assembly language instructions to perform effectively arithmetic operations, logical operations, data move operations, compare operations, program control operations, bit manipulation operations, parallel operations, extended auxiliary register operations and other miscellaneous operations. In the instruction syntax src & dst operands correspond to registers (AC0-AC3 & T0-T3). The Smem, Xmem, Ymem, Lmem corresponds to direct/indirect addressing mode operands. The Cmem corresponds to CDP indirect addressing mode/coefficient indirect addressing mode. The ACx & ACy operands correspond to accumulators (AC0-AC3).

16.6.1 Arithmetic Operations

The ′C55X processor has two ALU units, one 40-bit ALU in D-unit and another 16-bit ALU in A-unit. Both ALUs can be used to perform arithmetic operations (addition/subtraction). The 40-bit D-unit ALU is used, when the destination operand in the instruction is an accumulator, The 16-bit ALU is used, when the destination operand specified in the instruction is an auxiliary register or temporary register. The syntax and description of addition/subtraction instructions performed by the ALUs are given in Table 16.6.

Table 16.6 *Addition/subtraction Instructions of ′C55X*

Syntax	*Description*
ADD/SUB src,dst	The addition/subtraction operation performed between the content of two registers
ADD/SUB #k4,dst	The 4-bit unsigned constant specified in the instruction is added to/ subtracted from the destination register
ADD/SUB #k16,dst	The 16-bit unsigned constant specified in the instruction is added to/ subtracted from the destination register
ADD/SUB Smem,src,dst	The content of memory location (Smem) is added to/subtracted from the source register (src) content, result stored in destination register (dst)
SUB src,Smem,dst	The source register (src) content is subtracted from the memory location (Smem) content, result stored in destination register
ADD/SUB ACx<<Tx,ACy	The content of accumulator ACx is left shifted by the content of temporary register Tx and added to/subtracted from the accumulator ACy, result stored in ACy.
ADD/SUB ACx<<#shiftw,ACy	The content of accumulator ACx is left shifted by the 6-bit value (shiftw) and added to/subtracted from the accumulator ACy, result stored in ACy. Maximum of 31-bits can be shifted.
ADD/SUB #k16<<#16,ACx,ACy	The sixteen bit signed constant (k16) is left shifted by 16-bits and added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
ADD/SUB #k16<<#shft,ACx,ACy	The sixteen bit signed constant (k16) is left shifted by 4-bit value (shft) and added to/subtracted from the content of accumulator ACx, result stored in accu. ACy. Maximum of 15-bits can be shifted
ADD/SUB Smem<<Tx,ACx,ACy	The content of the memory location (Smem) is left shifted by the content of Tx and added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
ADD/SUB Smem<<#16,ACx,ACy	The content of the memory location (Smem) is left shifted by 16-bits and added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
SUB ACx,Smem<<#16,ACy	The accumulator ACx content is subtracted from the content of memory location (Smem) left shifted by 16-bits, result stored in ACy
ADD uns(Smem),Carry, ACx,ACy	The content of the memory location (Smem) and the carry bit are added to the content of accu. ACx, result stored in accu. ACy
SUB uns(Smem),Barrow, ACx,ACy	The logical compliment of carry bit & the content of memory location (Smem) are subtracted from the content of accu. ACx, result stored in accu. ACy

(Contd.)

Table 16.6 *(Contd.)*

ADD/SUB uns(Smem),ACx,ACy	The unsigned content of the memory location (Smem) is added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
ADD/SUB uns(Smem),<<#shiftw ACx,ACy	The unsigned content of the memory location (Smem) is left shifted by 6-bit value (shiftw) and added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy. Maximum of 31-bit can be shifted.
ADD/SUB dbl(Lmem),ACx,ACy	The long word (two memory locations) of the memory location (Lmem) is added to/subtracted from accumulator ACx, result stored in accumulator ACy
SUB ACx,dbl(Lmem),ACy	The content of accumulator ACx is subtracted from the long word (two memory locations) of the memory location (Lmem), result stored in accumulator ACy
ADD/SUB Xmem,Ymem,ACx	The content of memory location (Ymem) is left shifted by 16-bits and added to/subtracted from the content of memory location (Xmem) left shifted by 16-bits, result stored in accumulator ACx
ADD #k16,Smem	The signed 16-bit constant (k16) is added to the content of memory location (Smem), result stored back into the same memory location (Smem)

The multiplication operations are performed in D-unit multiply and accumulate unit (MAC). The D-unit has two MAC units, hence two multiply operations can be performed simultaneously in ′C55X processor. In multiply instructions, when the accumulators AC0-AC3 are used for source operands (src), the MSB 17-bits (32-16) are used for the multiplication and for other source operands, the sign extended to 17-bits. The product is 32-bits and the multiply instructions of ′C55X processor are given in Table 16.7.

Table 16.7 *Multiply Instructions of ′C55X*

Syntax	*Description*
SQR ACx,ACy	The 17 MSBs of accumulator ACx multiplied with itself (ACx *ACx), result stored in accumulator ACy
MPY ACx, ACy	The 17 MSBs of accumulator ACx & ACy are multiplied, result stored in accumulator ACy
MPY Tx,ACx, ACy	The content of temporary register Tx, sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied, result stored in accumulator ACy
MPY #k8,ACx, ACy	The signed 8-bit constant (k8), sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied, result stored in ACy
MPY #k16,ACx, ACy	The signed 16-bit constant (k16), sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied, result stored in ACy
SQRM Smem,ACx	The content of memory location (Smem), sign extended to 17-bits is multiplied with itself (Smem*Smem), result stored in accumulator ACx
MPYM Smem, Cmem,ACx	The content of memory location (Smem), sign extended to 17-bits and the content data memory operand (Cmem) addressed through coefficient addressing mode, sign extended to 17-bits are multiplied, result stored in accumulator ACx
MPYM Smem, ACx, ACy	The content of memory location (Smem), sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied, result stored in accumulator ACy
MPYMK Smem,#k8, ACx	The content of memory location (Smem), sign extended to 17-bits and the signed 8-bit constant (k8), sign extended to 17-bits are multiplied, result stored in accumulator ACx

(Contd.)

Table 16.7 *(Contd.)*

MPYM uns(Smem),uns(Ymem),ACx	The unsigned content of data memory locations (Smem) and (Ymem) are extended to 17-bits, multiplied and result stored in accu. ACx
MPYM Smem,Tx, ACx	The content of memory location (Smem), sign extended to 17-bits and the content of temporary register Tx, sign extended to 17-bits are multiplied, result stored in accumulator ACx

The MAC unit in D-unit can be used for multiply accumulate (MAC) and multiply subtract (MAS) instructions. The multiplication is carried out for the operands sign extended to 17-bits and the product is of size 32-bits. The product is sign extended to 40-bits, when accumulation or subtraction is performed after multiply operation. The multiply and accumulate/subtract instructions of ′C55X processor are given in Table 16.8.

Table 16.8 *Multiply and Accumulate/subtract Instructions of ′C55X*

SQA/SQS ACx,ACy	The 17 MSBs of accumulator ACx is multiplied with itself (ACx*ACx). The product is added to/subtracted from the content of accumulator ACy, result stored in accumulator ACy
SQAM/SQSM Smem, ACx,ACy	The content of memory location (Smem), sign extended to 17-bits is multiplied with itself (Smem*Smem). The product is added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
MAC ACx,Tx,ACy	The 17 MSBs of accumulator ACx and the content of temporary register Tx, sign extended to 17-bits are multiplied. The product is added to the content of accu. ACy, result stored in accu. ACy
MAC ACy,Tx,ACx,ACy	The 17 MSBs of accumulator ACy and the content of temporary register Tx, sign extended to 17-bits are multiplied. The product is added to the content of accu. ACx, result stored in accu. ACy
MAS Tx,ACx,ACy	The content of temporary register Tx, sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied. The product is subtracted from the content of accumulator ACy, result stored in accumulator ACy
MACK Tx,#k8,ACx,ACy	The content of temporary register Tx, sign extended to 17-bits and the 8-bit constant (k8), sign extended to 17-bits are multiplied. The product is added to the content of accumulator ACx, result stored in accumulator ACy
MACK Tx,#k16,ACx,ACy	The content of temporary register Tx, sign extended to 17-bits and the 16-bit constant (k16), sign extended to 17-bits are multiplied. The product is added to the content of accumulator ACx, result stored in accumulator ACy
MACM/MASM Smem,Cmem,ACx	The content of memory location (Smem), sign extended to 17-bits and the content data memory operand (Cmem) addressed through coefficient addressing mode, sign extended to 17-bits are multiplied. The product is added to/subtracted from the content of accumulator ACx, result stored in accumulator ACx
MACM/MASM Smem,ACx,ACy	The content of memory location (Smem), sign extended to 17-bits and the 17 MSBs of accumulator ACx are multiplied. The product is added to/subtracted from the content of accumulator ACy, result stored in accumulator ACy

(Contd.)

Table 16.8 *(Contd.)*

MACM/MASM Smem, Tx,ACx,ACy	The content of memory location (Smem), sign extended to 17-bits and the content of temporary register Tx, sign extended to 17-bits are multiplied. The product is added to/subtracted from the content of accumulator ACx, result stored in accu. ACy
MACMK Smem,#k8,ACx,ACy	The content of memory location (Smem), sign extended to 17-bits and the signed 8-bit constant (k8), sign extended to 17-bits are multiplied. The product is added to the content of accumulator ACx, result stored in accumulator ACy
MACM /MASM uns(Xmem), uns(Ymem),ACx,ACy	The unsigned content of data memory locations (Smem) and (Ymem) are extended to 17-bits and multiplied. The product is added to/subtracted from the content of accumulator ACx, result stored in accumulator ACy
MACM **uns(Xmem),uns(Ymem),ACx>>#16,ACy**	The unsigned content of data memory locations (Smem) and (Ymem) are extended to 17-bits and multiplied. The product is added to the content of accumulator ACx, which is right shifted by 16-bits with sign extended and result stored in accu. ACy
MACMZ Smem, Cmem, ACx (Equivalent to multiply accumulate and data move in ′C54X-MACD)	The content of memory location (Smem), sign extended to 17-bits and the content data memory operand (Cmem) addressed through coefficient addressing mode, sign extended to 17-bits are multiplied. The product is added to the content of accumulator ACx, result stored in accumulator ACx. **This instruction can be in parallel with the delay memory instruction. The content of memory location (Smem) is copied into the next higher address.**

The 40-bit signed shift operations can be performed in D-unit shifter. The accumulator content can be shifted left by a range of 1 to +31 and right by a range of 1 to 32. The right shift value is indicated by negative number and if the shift value is out of range then the shift is saturated to -32 or +31. The signed shift instructions of ′C55X processors are given in Table 16.9.

Table 16.9 *Signed shift Instructions of ′C55X*

Syntax	*Description*
SFTS dst, #-1	The content of destination register (dst) is right shifted by 1-bit. The registers are accumulators, auxiliary and temporary registers.
SFTS dst, #1	The content of destination register (dst) is left shifted by 1-bit. The registers are accumulators, auxiliary and temporary registers.
SFTS ACx,Tx,ACy	The content of accumulator ACx is shifted by the content of temporary register Tx, result stored in accumulator ACy.
SFTSC ACx,Tx,ACy	The content of accumulator ACx is shifted by the content of temporary register Tx, result stored in accumulator ACy. The shifted out bit is stored in CARRY status bit
SFTS ACx,#shiftw,ACy	The content of accumulator ACx is shifted by the 6-bit value (shiftw), result stored in accumulator ACy. Max. of 31-bits can be shifted right and 32-bits right
SFTSC ACx,#shiftw,ACy	The content of accumulator ACx is shifted by the 6-bit value (shiftw), result stored in accumulator ACy. The shifted out bit is stored in CARRY status bit

The 40-bit D-unit ALU can perform dual 16-bit arithmetic operations. The dual 16-bit arithmetic instructions are used to perform an addition or subtraction. In dual 16-bit mode, lower 16 bits of both ALU and accumulator are separated from their higher 24 bits. Arithmetic operations are carried out separately in lower and upper part of ALUs in the same machine cycle. The syntax of dual 16-bit arithmetic instructions are given in Table 16.10.

Table 16.10 *Dual 16-bit arithmetic Instructions of ′C55X*

Syntax	*Description*
ADDSUB Tx,Smem,ACx	The addition & subtraction operations performed parallel in one cycle. The content of temporary register Tx is added to the content of memory location (Smem), result stored in higher order bits of accumulator ACx (39-16). Also the content of Tx is subtracted from the content of memory location (Smem) & the result stored in lower order bits of accu. ACx(15-0)
ADDSUB Tx,dual(Lmem),ACx	The addition & subtraction operations performed parallel in one cycle. The content of temporary register Tx is added to the content of memory location pointed by Lmem, result stored in higher bits of accu. ACx (39-16). Also the content of Tx is subtracted from the content of memory location Lmem+1 & the result stored in lower order bits of ACx (15-0)
ADD/SUB dual(Lmem),ACx, ACy	Two parallel additions/subtractions are performed. The memory content pointed by Lmem is added to/subtracted from the higher order bits of accu. ACx (39-16), result stored in higher order bits of accu. ACy (39-16). Also, the memory content pointed by Lmem+1 is added to/ subtracted from the lower order bits of accu. ACx(15-0) result stored in lower order bits of accu. ACy(15-0).
ADD/SUB dual(Lmem),Tx,ACx	Two parallel additions/subtractions are performed. The memory content pointed by Lmem is added to/subtracted from the higher order bits of temporary register Tx (39-16), result stored in higher order bits of accu. ACx (39-16). Also, the memory content pointed by Lmem+1 is added to/subtracted from the lower order bits of temporary register Tx (15-0) result stored in lower order bits of accu. ACx(15-0).
SUBADD Tx,Smem,ACx	The subtraction &addition operations performed parallel in one cycle. The content of temporary register Tx is subtracted from the content of memory location (Smem), result stored in higher order bits of accumulator ACx (39-16). Also, the content of Tx is added to the content of memory location (Smem) & the result stored in lower order bits of accu. ACx(15-0)
SUB ACx,dual(Lmem),ACy	Two parallel subtractions are performed. The higher order bits of accu. ACx (39-16) are subtracted from the memory content pointed by Lmem, result stored in higher order bits of accu. ACy (39-16). Also, the lower order bits of accu. ACx(15-0) is subtracted from the memory content pointed by Lmem+1, result stored in lower order bits of accu. ACy(15-0).
SUB Tx,dual(Lmem),ACx	Two parallel subtractions are performed. The higher order bits of Tx (39-16) are subtracted from the memory content pointed by Lmem, result stored in higher order bits of accu. ACy (39-16). Also, the lower order bits of Tx(15-0) is subtracted from the memory content pointed by Lmem+1, result stored in lower order bits of accu. ACy(15-0).
SUBADD Tx,dual(Lmem),ACx	The subtraction & addition operations performed parallel in one cycle. The content of temporary register Tx is subtracted from the content of memory location pointed by Lmem, result stored in higher bits of accu. ACx (39-16). Also the content of Tx is added to the content of memory location Lmem+1 & the result stored in lower order bits of ACx (15-0)

The ′C55X processor supports conditional addition or subtraction, conditional subtraction and conditional shift operations based on the test control flag bits TC1 & TC2 in status register ST0_55. These conditional operations are performed in D-unit ALU. The conditional arithmetic operations are given in Table 16.11.

Table 16.11 *Conditional Arithmetic Instructions of 'C55X*

Syntax	*Description*
ADDSUBCC Smem,ACx,TCx,ACy	The content of memory location (Smem) left shifted by 16-bits is added to the content of accu. ACx, result stored in accu. ACy – **If TCx = 1**The content of memory location (Smem) left shifted by 16-bits is subtracted from the content of accu. ACx, result stored in accu. ACy – **If TCx = 0**
ADDSUBCC Smem,ACx,TC1, TC2,ACy	The instruction performs subtraction, addition and move operation based on TC1 & TC2 bits. **If TC2=1,** The content of accu. ACx is moved to accu. ACy. **If TC2=0 & TC1=0**, The content of memory location (Smem) left shifted by 16-bits is subtracted from the content of accu. ACx, result stored in accu. ACy. **If TC2=0 & TC1=1,** The content of memory location (Smem) left shifted by 16-bits is added to the content of accu. ACx, result stored in accu. ACy
ADDSUB2CC Smem, ACx,Tx, TC1, TC2,ACy	The instruction left shifts the content of Smem by 16-bits or by the content of Tx based TC2 bits and addition or subtraction operation is performed based on TC1 bits. **TC2=0, If TC1=0**, the content memory location Smem is left shifted by Tx content and subtracted from the content of ACx, result stored in ACy. If **TC1=1** addition operation is performed instead of subtraction. **TC2=1, If TC1=0**, the content memory location Smem is left shifted by 16-bits and subtracted from the content of ACx, result stored in ACy. If **TC1=1** addition operation is performed instead of subtraction
SUBC Smem, ACx,ACy	Conditional subtract instruction. The content of memory location (Smem) is left shifted 15-bits and subtracted from the content of ACx. If the result is greater than 0, the result is left shifted by 1-bit and 1 is added and stored in ACy. Else the content of ACx is left shifted by 1-bit and stored in ACy. This instruction is used for division operation.
SFTCC ACx, TCx	Conditional shift operation. The sign bits are extracted from the bit positions of 31and 30 of ACx. If the sign bit information is present, the content of ACx is left shifted by 1-bit and TCx is cleared to 0. If the sign bit information is not present the content of ACx is not shifted and TCx bit is set to 1.

16.6.2 Logical Operations

The bitwise logic operations like AND, OR, XOR, NOT, negate and logical shift are supported by 'C55X processor. It also supports bit field counting and rotate left and rotate right operations. In logical operations, if the destination operand is an accumulator, the operation is performed in D-unit. If the destination operand is an auxiliary register or a temporary register or memory, the A-unit is used. The syntax and description of logic instructions are given in Table 16.12.

Table 16.12 *Logical Instructions of 'C55X*

Syntax	*Description*
NOT src,dst	The content of source (src) operand is complimented and stored in destination (dst). 1's compliment of src stored in destination.
AND/OR/XOR src,dst	The bit wise AND/OR/XOR operations are performed between source and destination operand registers, result stored in destination.
AND/OR/XOR #k8,src,dst	The bit wise AND/OR/XOR operations are performed between the 8-bit value (k8) and the source (src) register, result stored in destination register

(Contd.)

Table 16.12 *(Contd.)*

AND/OR/XOR #k16,src,dst	The bit wise AND/OR/XOR operations are performed between the 16-bit value (k16) and the source (src) register, result stored in destination register
AND/OR/XOR Smem,src,dst	The bit wise AND/OR/XOR operations are performed between the content of memory location (Smem) and the source (src) register, result stored in destination register
AND/OR/XOR ACx<<#shiftw,ACy	The bit wise AND/OR/XOR operations are performed between the content of accu. ACy and the accu ACx content left shifted by 6-bit value (shifw), result stored in accu. ACy.
AND/OR/XOR #k16<<#16,ACx,ACy	The bit wise AND/OR/XOR operations are performed between the content of accu. ACx and the 16-bit value (k16) left shifted by 16-bits, result stored in accu. ACy.
AND/OR/XOR #k16<<#shft,ACx,ACy	The bit wise AND/OR/XOR operations are performed between the content of accu. ACx and the 16-bit value (k16) left shifted by 4-bit value (shft) , result stored in accu. ACy.
AND/OR/XOR **#k16,Smem**	The bit wise AND/OR/XOR operations are performed between the content of memory location (Smem) and the 16-bit value (k16), result stored back in memory location (Smem)
NEG src,dst	The 2's compliment for the content of the source register is stored in dst.
BCNT ACx,ACy,TCx,Tx	The bitwise AND operation is performed between the contents of accu. ACx & ACy. The number of bits set to 1 in the result is evaluated and stored in temporary register Tx. If the number of bits is even TCx bit cleared & if the number bits is odd TCx bit is set to 1.
SFTL dst,#1	The content of destination operand (dst) register is logically shifted left by 1-bit. The shifted out bit stored in CARRY status bit.
SFTL dst,#-1	The content of destination operand (dst) register is logically shifted right by 1-bit. The shifted out bit stored in CARRY status bit.
SFTL ACx,Tx,ACy	The content of the accu. ACx is left shifted by the content of Tx and the result is stored in accu. ACy. The shifted out bit stored in CARRY status bit.
SFTL ACx,#shiftw,ACy	The content of the accu. ACx is left shifted by 6-bit value (shiftw) and the result is stored in accu. ACy. The shifted out bit stored in CARRY status bit. Max. 31-bits can be shifted.
ROL Bitout,src,Bitin,dst	Bitwise rotate left the accumulator content (src). Bitin & Bitout are shift in one bit and shifted out bit respectively during rotation. Both TC2 & CARRY status bit can be used for bitin & bitout. The rotated result is stored in destination register.
ROR Bitin,src,Bitout,dst	Bitwise rotate right the accumulator content (src). Bitin & Bitout are shift in one bit and shifted out bit respectively during rotation. Both TC2 & CARRY status bit can be used for bitin & bitout. The rotated result is stored in destination register.

16.6.3 Move Operations

The ′C55X processor has the following three categories of move instructions:

- Move instructions to access accumulators, auxiliary and temporary registers.
- Move instructions to access CPU registers other than registers mentioned in (i).
- Move instructions to perform memory-to-memory move operations.

(i) Accumulator, Auxiliary and Temporary Register Move Instructions

In ′C55X processor, assembly language instructions are available to swap the content of accumulators, auxiliary registers and temporary registers. Instructions are there to move, load and store values in these registers. The register swap instructions in Table 16.13, the register move instructions in Table 16.14, the register load instructions in Table 16.15 and register store instructions in Table 16.16 are given.

Table 16.13 *Register Swap Instructions of 'C55X*

Syntax	*Description*
SWAP ARx,Tx	Parallel move between auxiliary registers (ARx) and temporary registers (Tx). The content of ARx moved to Tx and the content of Tx moved to ARx. Only the auxiliary registers AR4-AR7 are used for this operation.
SWAP Tx,Ty	Parallel move between temporary registers. The content of Tx moved to Ty and the content of Ty moved to Tx
SWAP ARx,ARy	Parallel move between auxiliary registers (ARx). The content of ARx moved to ARy and the content of ARy moved to ARx. Only the auxiliary registers AR0-AR3 are used for this operation.
SWAP ACx,ACy	Parallel move between accumulators (ACx). The content of ACx moved to ACy and the content of ACy moved to ACx.
SWAPP ARx,Tx	Parallel move between two adjacent auxiliary registers (ARx) and two adjacent temporary registers (Tx). The content of two adjacent ARx moved to two adjacent Tx and the content of two adjacent Tx moved to two adjacent ARx. Only the auxiliary registers AR4-AR7 are used for this operation
SWAPP T0,T2	Parallel move between two adjacent temporary registers. The content of T0 moved to T2 and the content of T1 moved to T3. Also, the content of T2 moved to T0 and the content of T3 moved to T1
SWAPP AR0,AR2	Parallel move between two adjacent auxiliary registers (ARx). The content of AR0 moved to AR2 and the content of AR1 moved to AR3. Also, The content of AR2 moved to AR1 and the content of AR3 moved to AR1. Only the auxiliary registers AR0-AR3 are used for this operation.
SWAPP AC0,AC2	Parallel move between two adjacent accumulators (ACx). The content of AC0 moved to AC2 and the content of AC1 moved to AC3. Also, The content of AC2 moved to AC1 and the content of AC3 moved to AC1.
SWAP4 AR4,T0	Parallel move between four auxiliary register to temporary registers. The content of auxiliary registers AR3, AR5, AR6 & AR7 are moved to temporary registers T0, T1, T2 & T3 respectively and also its reciprocal move operation performed. Only the auxiliary registers AR4-AR7 are used for this operation

Table 16.14 *Register Move Instructions of 'C55X*

Syntax	*Description*
MOV src,dst	The source register content is moved to destination register content
MOV HI(ACx),TAx	The 16 MSBs of the accumulator ACx are moved to temporary or auxiliary register
MOV TAx, HI(ACx)	The temporary or auxiliary register content is moved to 16 MSBs of accu. ACx

Table 16.15 *Register Load Instructions of 'C55X*

Syntax	*Description*
MOV #k4, dst	The 4-bit unsigned constant (k4) is loaded to the destination register(dst).
MOV #-k4, dst	The 2's complement value of the 4-bit unsigned constant (k4) is loaded to destination register (dst).
MOV #k16, dst	The 16-bit signed constant (k16) is loaded to destination register (dst).
MOV Smem, dst	The content of memory location (Smem) is loaded to destination register.
MOV uns(high_byte(Smem)),dst	The high byte content (15-8) of memory location (Smem) is loaded to destination register (dst).

(Contd.)

Table 16.15 *(Contd.)*

MOV uns(low_byte(Smem)),dst	The low byte content (7-0) of memory location (Smem) is loaded to destination register (dst).
MOV #k16 << #16, ACx	The 16-bit signed constant (k16) is left shifted by 16-bits and loaded to accumulator ACx
MOV #k16 << #shft, ACx	The 16-bit signed constant (k16) is left shifted by the 4-bit value (shft) and loaded to accumulator ACx
MOV Smem << Tx,ACx	The content of memory location (Smem) is left shifted by the content Tx and loaded to accumulator ACx.
MOV**low_byte(Smem) << #SHIFTW, ACx**	The low byte content (7-0) of memory location (Smem) is left shifted by 6-bit shift value (shifw) and loaded to accumulator ACx.
MOV**high_byte(Smem) << #SHIFTW, ACx**	The high byte content (15-8) of memory location (Smem) is left shifted by 6-bit shift value (shifw) and loaded to accumulator ACx.
MOV Smem << #16, ACx	The high byte content of memory location (Smem) is left shifted by 16-bits and loaded to accumulator ACx.
MOV uns(Smem), ACx	The content of memory location (Smem) is zero extended to 40-bits and loade to accumulator ACx. If syntax 'uns' is not used no zero extension.
MOV **uns(Smem) << #SHIFTW, ACx**	The content of memory location (Smem) is zero extended to 40-bits and left shited by 6-bit value (shiftw). The result is loade to accumulator ACx. If syntax 'uns' is not used no zero extension.
MOV dbl(Lmem),ACx	The long word load instruction. The content of memory location pointed by Lmem and Lmem+1 are loaded to accumulator ACx. The operand is sign extended to 40-bits.
MOV Xmem, Ymem, ACx	Dual 16-bit load instruction. The memory content pointed by Xmem is loaded to lower order bits (15-0) of accu. ACx. The memory content pointed by Ymem is sign extened to 24-bits and loaded to higher order bits (40-16) accumulator ACx.
MOV dbl(Lmem), pair(HI(ACx))	The memory content point by Lmem is loaded to the higher order bits (31-16) of accumulator ACx and the memory content pointed by Lmem+1 is loaded to the higher order bits (31-16) of the next accumulator ACx+1.
MOV dbl(Lmem), pair(LO(ACx))	The memory content point by Lmem is loaded to the lower order bits (15-0) of accumulator ACx and the memory content pointed by Lmem+1 is loaded to the lower order bits (15-0) of the next accumulator ACx+1.
MOV dbl(Lmem), pair(TAx)	The memory content point by Lmem is loaded to the lower order bits (15-0) of temporary register Tx and the memory content pointed by Lmem+1 is loaded to the lower order bits (15-0) of the next temporary register Tx+1.

Table 16.16 *Register Store Instructions of 'C55X*

Syntax	*Description*
MOV src, Smem	The 16 LSBs of source register (src) is stored to the memory location pointed by Smem.
MOV src, high_byte(Smem)	The low byte content (7-0) of the source register (src) is stored to high byte of the memory (15-8) location pointed by Smem.

(Contd.)

Table 16.16 *(Contd.)*

MOV src, low_byte(Smem)	The low byte content (7-0) of the source register (src) is stored to low byte of the memory (7-0) location pointed by Smem.
MOV HI(ACx), Smem	The higher part of accu. ACx (31-16) is stored to the memory location pointed by Smem.
MOV rnd(HI(ACx)), Smem	The higher part of accu. ACx (31-16) is rounded and stored to memory location pointed by Smem.
MOV ACx << Tx, Smem	The content of accumulator ACx is left shifted by the content of temporary register Tx and 16 LSBs of ACx are stoted to memory location pointed by Smem.
MOV rnd(HI(ACx << Tx)), Smem	The content of accumulator ACx is left shifted by the content of temporary register Tx and rounded (rnd-optional). The higher part of ACx (31-16) is stoted to memory location pointed by Smem.
MOV ACx << #SHIFTW, Smem	The content of accu. ACx is left shifted by the 6-bit shift value (shiftw) and 16 LSBs of ACx is stored to the memory location pointed by Smem.
MOV HI(ACx << #SHIFTW), Smem	The content of accu. ACx is left shifted by the 6-bit shift value (shiftw) and the higher part of ACx (31-16) is stored to the memory location pointed by Smem.
MOV rnd(HI(ACx << #SHIFTW)), Smem	The content of accu. ACx is left shifted by the 6-bit shift value (shiftw) and rounded (rnd –optional). The higher part of ACx (31-16) is stored to the memory location pointed by Smem.
MOV uns(rnd(HI(saturate(ACx)))), Smem	The unsigned content of accu. ACx is rounded (rnd-optional). For shift/rounding overflow 40-bit ACx content is saturated (saturate-optional) The higher part of ACx (31-16) is stored to the memory location pointed by Smem.
MOV **uns(rnd(HI(saturate (ACx << Tx)))), Smem**	The unsigned content of accu. ACx is left shifted by the content of temporary register Tx and rounded (rnd-optional). For shift/rounding overflow 40-bit ACx content is saturated (saturate-optional) The higher part of ACx (31-16) is stored to the memory location pointed by Smem.
MOV **uns((rnd(HI(saturate(ACx<<#SHIFTW)))),Smem**	The unsigned content of accu. ACx is left shifted by the 6-bit value (shiftw) and rounded (rnd-optional). For shift/rounding overflow 40-bit ACx content is saturated (saturate-optional) The higher part of ACx (31-16) is stored to the memory location pointed by Smem.
MOV ACx, dbl(Lmem)	The long word store. The 16 MSBs of the accumulator ACx (31-16) are stored to memory location pointed by Lmem and 16 LSBs of accumulator ACx are stored to memory location pointed by Lmem+1
MOV uns(saturate(ACx)), dbl(Lmem)	The unsigned content of accumulator ACx is saturated (optional) on overflow. The 16 MSBs of the accumulator ACx (31-16) are stored to memory location pointed by Lmem and 16 LSBs of accumulator ACx are stored to memory location pointed by Lmem+1

(Contd.)

Table 16.16 *(Contd.)*

MOV ACx >> #1, dual(Lmem)	The higher part of accumulator ACx (31-16) is right shifted by 1-bit and stored to memory location pointed by Lmem. The 16 LSBs of accumulator ACx is right shifted by 1-bit and stored to the memory location pointed by Lmem+1.
MOV pair(HI(ACx)), dbl(Lmem)	The higher part of accumulator content ACx (31-16) is stored to memory location pointed by Lmem. The next accumulator (ACx+1) higher part content (31-16) is stored to memory location pointed by Lmem+1
MOV pair(LO(ACx)), dbl(Lmem)	The 16 LSBs of accumulator ACx (15-0) is stored to memory location pointed by Lmem. The 16 LSBs (15-0) of the next accumulator (ACx+1) is stored to memory location pointed by Lmem+1
MOV pair(TAx), dbl(Lmem)	The content of temporary register Tx is stored to memory location pointed by Lmem. The next temporary register (Tx+1) is stored to memory location pointed by Lmem+1
MOV ACx, Xmem, Ymem	The lower part of accumulator (15-0) is stored to memory location point by Xmem. The higher part of accumulator (31-16) is stored to memory location pointed by Ymem.

(ii) CPU Register Access Instructions

Instructions to access CPU registers other than accumulators, auxiliary registers and temporary registers are available. There are CPU register move, load and store instructions.

CPU Register Move Instructions Instructions are used to move temporary or auxiliary register values (TAx) to CPU registers BRC0, BRC1, CDP, CSR, SP and SSP and vice versa. The A-unit is used to perform these operations. The syntax for CPU register move instruction is given below:

```
MOV TAx, Register      ; the content of temporary or auxiliary register is moved to
                       ; specified CPU register
MOV Register, TAx      ; the content of specified CPU register is moved to temporary or
                         auxiliary register
```

CPU Register Load Instructions Instructions to load unsigned integers of size kx (k7, k9, k12 and k16) to destination CPU registers are available. Also instructions to load the content of memory location to destination CPU registers are present. The destination CPU registers are BK03, BK47, BKC, BRC0, BRC1, CSR, DPH, PDP, BSA01, BSA23, BSA45, BSA67, BSAC, CDP, DP, SP, SSP, TRN0, TRN1 and RETA. The CPU register load instructions are given in Table 16.17.

Table 16.17 *CPU Register Load Instructions of 'C55X*

Syntax	*Description*
MOV #k7, DPH	The unsigned 7-bits are loaded to 7 MSBs (DPH) of extended data page pointer XDP
MOV #k9,PDP	The unsigned 9-bits are loaded to peripheral data page pointer PDP
MOV #k12,Register	The unsigned 12-bits are loaded to destination register. The destination registers are BK03, BK47, BKC, BRC0, BRC1 and CSR
MOV #k16,Register	The unsigned 12-bits are loaded to destination register. The destination registers are BSA01, BSA23, BSA45, BSA67, BSAC, CDP, DP, SP and SSP

(Contd.)

Table 16.17 *(Contd.)*

MOV Smem,Register	The content of memory location (Smem) is loaded to destination register. The destination registers are BK03, BK47, BKC, BRC0, BRC1, CSR, DPH, PDP, BSA01, BSA23, BSA45, BSA67, BSAC, CDP, DP, SP, SSP, TRN0 and TRN1
MOV dbl(Lmem), RETA	The long word (32-bits) content of memory location (Lmem) is loaded to destination register RETA. The LSB 24-bits are loaded to RETA register and MSB 8-bits to CFCT register

CPU Registers Stored Instructions Instructions to store the CPU register content to memory is supported by ′C55X. The CPU registers having this option are BK03, BK47, BKC, BRC0, BRC1, CSR, DPH, PDP, BSA01, BSA23, BSA45, BSA67, BSAC, CDP, DP, SP, SSP, TRN0, TRN1 and RETA. The syntax for the CPU register store instructions are given below:

```
MOV Register, Smem     ; the register content is stored in memory location Smem
                       ; the registers are above mentioned registers except RETA
MOV RETA, dbl(Lmem) ; The 24-bits of RETA and 8-bits of CFCT are stored in two
                       ; data memory locations pointed by Lmem
```

(iii) Memory-to-Memory Move Instructions

Instructions to move 8-bit or 16-bit signed constant to memory locations are available in ′C55X. Also instructions to move data values from one memory location to another memory location are available. The memory-to-memory move instructions are given in Table 16.18.

Table 16.18 *Memory-to-memory move Instructions of ′C55X*

Syntax	*Description*
MOV Cmem, Smem	The data memory content addressed by coefficient data page poinet (CDP) is copied to the memory location pointed by Smem.
MOV Smem, Cmem	The data memory location pointed by Smem is copied to the memory location addressed by coefficient data page pointer (CDP)
MOV #k8, Smem	The 8-bit signed constant (k8) is stored to memory location pointed by Smem.
MOV #k16, Smem	The 16-bit signed constant (k16) is stored to memory location pointed by Smem.
MOV Cmem, dbl(Lmem)	The long word move, two consecutive memory locations are moved. The data memory location addressed by coefficient data page pointer (CDP) is moved to memory location pointed by Lmem. The next memory location pointed by CDP+1 is moved to the memory location pointed by Lmem+1
MOV dbl(Lmem), Cmem	The long word move, two consecutive memory locations are moved. The data memory location pointed by Lmem is moved to memory location addressed by coefficient data page pointer (CDP). The next memory location pointe by Lmem+1 is moved to memory location addressed by CDP+1
MOV dbl(Xmem), dbl(Ymem)	Two consecutive memory locations are moved using dual addressing mode. The data memory content pointed by Xmem and Xmem+1 are moved to memory locations Ymem and Ymem+1 respectively.
MOV Xmem, Ymem	The data memory content pointed by Xmem is moved to memory content pointed by Ymem.
DELAY Smem	The data move instruction. The instruction copies the content of memory location point by Smem to its higher address Smem+1. The content of memory location pointed by Smem remains the same.

16.6.4 Compare Operations

The ′C55X processor supports register compare, maximum compare, minimum compare and memory compare operations. The memory comparison is performed in A-unit ALU and other comparisons are performed in D-unit or A-unit ALU units. If the comparison is true the test control flag bit TCx is set 1, otherwise it is cleared to 0. There are two test control flags TC1 and TC2 in status register ST0_55 of ′C55X processor and any one can be used. The syntax and description of compare instructions are given in Table 16.19.

Table 16.19 *Compare Instructions of ′C55X*

Syntax	*Description*
MAX src, dst	The maximum comparison instruction. The content of source register (src) and destination register (dst) are compared. If the content of source is greater than the destination, the source content is stored to destination and the CARRY status bit is cleared to 0. If the content of source is less than the destination, the destination register content remains the same and the CARRY status flag bit is set to 1.
MIN src, dst	The minimum comparison instruction. The content of source register (src) and destination register (dst) are compared. If the content of source is less than the destination, the source content is stored to destination and the CARRY status bit is cleared to 0. If the content of source is greater than the destination, the destination register content remains the same and the CARRY status flag bit is set to 1.
CMP Smem = = #k16, TCx	Memory compare instruction. The content of memory location pointed by Smem is compared to the 16-bit signed constant (k16). If they are equal the TCx status bit in status register is set to 1 else it is cleared to 0
CMP src RELOP dst, TCx **(RELOP operators** **= = - equal** **< - less than** **>= - greater than or equal to** **!= - not equal to)**	RELOP refers relational operator. The content of source register and destination register are compared. If the relation operator (RELOP) specified in the instruction is true the TCx status bit in status register is set to 1 else the TCx bit is cleared to 0.
CMPAND/CMPOR src RELOP dst, TCy, TCx	The content of source register & destination register are compared. If the relation operator (RELOP) specified in the instruction is true, TCx status bit in status register is set to 1 else the TCx bit is cleared to 0. The comparison result of TCx bit ANDed/ORed with TCy bit and result is updated in TCx.
CMPAND/CMPOR src RELOP dst, !TCy, TCx	The content of source register & destination register are compared. If the relation operator (RELOP) specified in the instruction is true, TCx status bit in status register is set to 1 else the TCx bit is cleared to 0. The comparison result of TCx bit ANDed/ORed with the complement of TCy bit and result is updated in TCx.

16.6.5 Program Control Operations

The program control operations of ′C55X include instructions to perform single repeat, block repeat, branch, call, return, compare and branch, software interrupts and execute conditional operations. The single repeat, branch, call and return instructions can be executed conditionally or unconditionally. The various test conditions that can be used for the ′C55X processor conditional instructions are given below:

(i) Conditions testing of accumulator, auxiliary and temporary register content

The content of registers are tested for equal to zero, not equal to zero, greater than, greater than or equal to zero, less than or less than or equal to zero. The syntax to represent the conditions in instructions is given below:

== #0 - the content equal to zero   !==#0 - the content not equal to zero
< #0 - the content less than zero   <=#0 - the content less than or equal to zero
> #0 - the content greater than zero   >=#0 - the content greater than or equal to zero

Examples

ACx <#0 - the content accumulator less than zero
*ARx >#0 - the content of auxiliary register greater than zero
Tx !==#0 - the content of temporary register not equal to zero

(ii) Conditions testing accumulator overflow

The accumulator over flow flag bit (ACOVx) in status register can be tested for 1 or 0. The syntax for testing overflow is:

overflow(ACx) - overflow flag bit tested for 1 & !overflow(ACx) - overflow flag bit tested for zero

(iii) Conditions testing CARRY status bit

The CARRY status bit in status register can be tested for 1 or 0. The syntax for testing carry bit is: CARRY – carry flag bit tested for 1 & ! CARRY – carry flag bit tested for 0

(iv) Conditions testing test control flags TC1 & TC2

The test control flag bits TC1 and TC2 can be independently tested for 1 or 0. The syntax for testing TCx bit is: TCx – test control flag tested for 1 & !TCx – test control flag tested for 0

The TC1 and TC2 bits can be combined with AND (&), OR (|) and XOR(^) logical bit combinations. The syntax is given below:

AND TC1 & TC2   TC1 & !TC2 OR TC1 | TC2   TC1 | !TC2   XOR TC1 ^ TC2   TC1 ^ !TC2
!TC1 & TC2   !TC1 & !TC2   !TC1 | TC2   !TC1 | !TC2   !TC1 ^ TC2   !TC1 ^ !TC2

The program control instructions of ′C55X are given in Table 16.20.

Table 16.20 *Program Control Instructions of ′C55X*

Syntax	*Description*
Single Repeat Instructions	
RPT CSR	The next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified in computed single repeat register (CSR) + 1 time.
RPTADD CSR, TAx	The next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified in computed single repeat register (CSR) + 1 time. The content of CSR is incremented by the content of temporary or auxiliary register content TAx.
RPT #k8	The next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified by the 8-bit immediate value k8 + 1 time.
RPT #k16	The next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified by the 16-bit immediate value k16 + 1 time.

(Contd.)

Table 16.20 *(Contd.)*

RPTADD/RPTSUB CSR, #k4	The next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified in computed single repeat register (CSR) + 1 time. The content of CSR is incremented (ADD)/decremented (SUB) by the unsigned 4-bit value (k4) specified in the instruction.
RPTCC #k8, cond	Conditional single repeat instruction. If the condition specified in the conditional operator field is true, the next instruction or the next two paralleled instructions followed by the RPT instruction is executed by the number of times specified by the 8-bit immediate value k8 + 1 time.
Block repeat instructions	
RPTB pmad	Repeat block instruction. The block of instructions defined can be executed by number of times block repeat counter register (BRCx) + 1 time. There are two block repeat counter registers BRC0 & BRC1. If no loop has been already detected BRC0 content is used to repeat the block. If one level loop is detected BRC1 content is used to repeat the block.
RPTBLOCAL pmad	Nested repeat block instruction. The block of instructions defined can be executed by number of times block repeat counter register (BRCx) + 1 time. If no loop has been already detected BRC0 content is used to repeat the block. If one level loop is detected BRC1 content is used to repeat the block.
Branch instructions	
B ACx	Branch to 24-bit program memory address defined by lower order bits of accumulator ACx (23-0)
B L7/ L16/P24	Branches to a program address defined by a program address label assembled into Lx/P24
BCC I4/L8/L16/P24, cond	Conditional branch instruction. Branches to a program address defined by a program address label assembled into I4/Lx/P24, if the condition specified in the conditional operator filed (cond) is true
BCC L16, ARn_mod != #0	Branch on auxiliary register content not zero instruction. Branch to program memory address specified as a 16-bit signed offset, L16, relative to PC, if the auxiliary register address modification content (ARn_mod) not zero.
BCC L8, src RELOP #k8	Compare and branch instruction. The comparison operation between the source register content and the 8-bit signed value (k8) is performed. If the comparison is true, the branch to program memory address specified as a 8-bit signed offset (L8), relative to PC.
Call instructions	
CALL ACx/L16/L24	Call subroutine program address defined by the content of the 24 lowest bits of the accumulator ACx/a program address label assembled into L16 or P24.
CALLCC L16/P24, cond	Conditional call instruction. A subroutine call occurs to the program address defined by the program address label assembled into L16 or P24, if the condition specified in the conditional operator field (cond) is true.
RET	Return instruction. The program counter is loaded with the return address of the calling subroutine.
RETCC cond	The program counter is loaded with the return address of the calling subroutine, if the condition specified in the conditional operator field (cond) is true.
RETI	The program counter is loaded with the return address of the interrupted task.

(Contd.)

Table 16.20 *(Contd.)*

INTR #k5	This instruction passes control to a specified interrupt service routine (ISR) and interrupts are globally disabled. The ISR address is stored at the interrupt vector address defined by the content of an interrupt vector pointer (IVPD or IVPH) combined with the 5-bit constant (k5).
RESET	This instruction performs a nonmaskable software reset that can be used any time to put the device in a known state.
TRAP #k5	This instruction passes control to a specified interrupt service routine (ISR). The ISR address is stored at the interrupt vector address defined by the content of an interrupt vector pointer (IVPD or IVPH) combined with the 5-bit constant (k5). This instruction is not maskable.
XCC label,cond	This instruction evaluates a single condition defined by the conditional field (cond) and allows to control the execution flow of an instruction, or instructions, from the address phase to the execute phase of the pipeline.
XCCPART label,cond	This instruction evaluates a single condition defined by the conditional field (cond) and allows to control the execution flow of an instruction, or instructions, from the execute phase of the pipeline
IDLE	Power-down mode instruction. This instruction forces the program being executed to wait until an interrupt or a reset occurs.
NOP	No operation. The program counter is incremented by 1 byte.
NOP_16	No operation. The program counter is incremented by 2 byte.

16.6.6 Bit Manipulation Operations

The bit manipulation instructions are to bit set or clear the status registers ST0_55 to ST3_55 and bit test/set/clear/complement the registers and memory. Instructions are also available for bit field comparison, bit field expand and bit field extract. The bit manipulation instructions are given in Table 16.21.

Table 16.21 *Bit Manipulation Instructions of ′C55X*

Syntax	*Description*
BCLR # k4,STx_55	Bit clear instruction. This instructions clear to 0 a single bit, as defined by a 4-bit immediate value (k4) in the selected status register
BSET #k4,STx_55	Bit set instruction. This instructions set to 1 a single bit, as defined by a 4-bit immediate value (k4) in the selected status register
BCLR f-name	The bit field name (f-name) is cleared to 0 in the selected status register. The name of the bit to be cleared in status register can be specified directly.
BSET f-name	The bit field name (f-name) is set to 1 in the selected status register. The name of the bit to be cleared in status register can be specified directly
BTST Baddr, src,TCx	A single bit of the source register location as defined by the bit addressing mode (Baddr) is tested. The tested bit is copied into the selected TCx status bit.
BNOT Baddr,src	A single bit of the source register location as defined by the bit addressing mode (Baddr) is complemented.
BCLR Baddr,src	A single bit of the source register location as defined by the bit addressing mode (Baddr) is cleared to 0.

(Contd.)

Table 16.21 *(Contd.)*

BSET Baddr,src	A single bit of the source register location as defined by the bit addressing mode (Baddr) is set to 1.
BTSTP Baddr,src	The two consecutive bits of the source register location as defined by the bit addressing mode (Baddr) and Baddr + 1are tested. The tested bits are copied into status bits TC1 and TC2.
BTST src,Smem,TCx	A single bit in the memory location pointed by Smem is tested as defined by the content of the source operand (src). The tested bit is copied into the selected TCx status bit.
BNOT src,Smem	A single bit in the memory location pointed by Smem is complemented as defined by the content of the source operand (src).
BCLR src,Smem	A single bit in the memory location pointed by Smem is cleared to 0 as defined by the content of the source operand (src).
BSET src,Smem	A single bit in the memory location pointed by Smem is set to 1 as defined by the content of the source operand (src).
BTSTSET #k4,Smem,TCx	A single bit in the memory location pointed by Smem is tested as defined by the 4-bit immediate value (k4). The tested bit is copied into status bit TCx. The selected bit in the memory location pointed by Smem is set to 1.
BTSTCLR #k4,Smem,TCx	A single bit in the memory location pointed by Smem is tested as defined by the 4-bit immediate value (k4). The tested bit is copied into status bit TCx. The selected bit in the memory location pointed by Smem is cleared to 0.
BTSTNOT #k4,Smem,TCx	A single bit in the memory location pointed by Smem is tested as defined by the 4-bit immediate value (k4). The tested bit is copied into status bit TCx. The selected bit in the memory location pointed by Smem is complemented.
BTST #k4,Smem,TCx	A single bit in the memory location pointed by Smem is tested as defined by the 4-bit immediate value (k4). The tested bit is copied into status bit TCx.
BAND Smem,#k16,TCx	Bit field comparison instruction. The 16-bit immediate value (k16) is ANDed with the content of memory location point by Smem. If the result is equal to zero, the TCx bit cleared to 0, else TCx bit is set to 1.
BFXPA #k16,ACx,dst	Bit field expand instruction. The 16-bit immediate value (k16) is scanned from the least significant bit (LSB) to the most significant bit (MSB). According to the bit set to 1 in the bit field mask, the 16 LSBs of the source accumulator (ACx) bits are extracted & separated with 0 toward the MSBs. The result is stored in dst.
BFXTR #k16,ACx,dst	Bit field extract instruction. The 16-bit immediate value (k16) is scanned from the least significant bit (LSB) to the most significant bit (MSB). According to the bit set to 1 in the bit field mask, the corresponding 16 LSBs of the source accu.(ACx) bits are extracted and packed toward the LSBs. The result is stored dst.

16.6.7 Parallel Operations

The ′C55X processor supports instruction execution in parallel. The instructions can perform the following parallel operations:

- Multiply and store
- Multiply and accumulate (MAC), and store
- Multiply and subtract (MAS), and store
- Addition and store
- Subtraction and store
- Load and store
- Multiply and accumulate (MAC), and load
- Multiply and subtract (MAS), and load

- Dual multiply
- Multiply and accumulate (MAC) and multiply
- Multiply and multiply and accumulate (MAC)
- Multiply and subtract (MAS) and multiply
- Dual multiply and accumulate (MAC)
- Dural multiply and subtract (MAS)
- Multiply and subtract (MAS) and multiply and accumulate (MAC)
- Auxiliary register modification and multiply
- Auxiliary register modification and multiply and accumulate (MAC)
- Auxiliary register modification and multiply and subtract (MAS)
- Multiply and accumulate (MAC) and add (Least mean square-LMS and Finite Impulse Response -FIRS)
- Multiply and accumulate (MAC) and subtract (Finite Impulse Response – FIRS and Square Distance-SQDST)

The syntax and description of parallel operation instructions are given in Table 16.22.

Table 16.22 *Parallel Instructions of 'C55X*

Syntax	*Description*
MPYM Xmem,Tx,ACy:: **MOV HI(ACx << T2),Ymem**	The content of memory location pointed by Xmem and the content of temporary register Tx are sign extended to 17-bits and multiplied. In parallel, the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.
MACM Xmem, Tx, ACy:: **MOV HI(ACx << T2),Ymem**	The content of memory location pointed by Xmem and the content of temporary register Tx are sign extended to 17-bits, multiplied, the product is added to the content of ACy and the result stored in ACy. In parallel, the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.
MASM Xmem, Tx, ACy:: **MOV HI(ACx << T2),Ymem**	The content of memory location pointed by Xmem and the content of temporary register Tx are sign extended to 17-bits, multiplied, the product is subtracted from the content of ACy and the result stored in ACy. In parallel, the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.
ADD Xmem << #16, ACx, ACy:: **MOV HI(ACy << T2), Ymem**	The content of memory location pointed by Xmem is left shifted by 16-bits, added to the content of ACx and the result stored in ACy. In parallel, the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.
SUB Xmem << #16, ACx, ACy:: **MOV HI(ACy << T2), Ymem**	The content of accu. ACx is subtracted from the content of memory location pointed by Xmem left shifted by 16-bits and the result stored in ACy. In parallel the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.

(Contd.)

Table 16.22 *(Contd.)*

MOV Xmem << #16, ACy:: MOV HI(ACx << T2), Ymem	The content of memory location pointed by Xmem is left shifted by 16-bits and stored in ACy. In parallel, the higher order bits of accu. ACx (31-16) is left shifted by the content of T2 and stored in memory location pointed by Ymem.
MACM Xmem, Tx, ACx:: MOV Ymem << #16, ACy	The content of memory location pointed by Xmem and the content of temporary register Tx are sign extended to 17-bits, multiplied, the product is added to the content of ACx and the result stored in ACx. In parallel, the content of memory location pointed by Ymem is left shifted by 16-bits and stored in ACy.
MASM Xmem, Tx, ACx:: MOV Ymem << #16, ACy	The content of memory location pointed by Xmem & the content of temporary register Tx are sign extended to 17-bits, multiplied, the product is subtracted from the content of ACx and the result stored in ACx. In parallel, the content of memory location pointed by Ymem is left shifted by 16-bits stored in ACy.
MPY uns(Xmem), uns(Cmem),ACx:: MPY uns(Ymem), uns(Cmem), ACy	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACx. In parallel, The unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACy.
MAC uns(Xmem),uns(Cmem), ACx:: MPY uns(Ymem), uns(Cmem),ACy	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx and the result stored in accu. ACx. In parallel, The unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACy.
MAS uns(Xmem),uns(Cmem), ACx:: MPY uns(Ymem),uns(Cmem), ACy	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACx and the result stored in accu. ACx. In parallel, The unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACy.
AMAR Xmem:: MPY uns(Ymem),uns(Cmem), ACx	The content of auxiliary register is modified as specified by the content of memory location pointed by Xmem. In parallel, The unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACx.

(Contd.)

Table 16.22 *(Contd.)*

MAC uns(Xmem),uns(Cmem), ACx:: **MAC uns(Ymem),uns(Cmem), ACy**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy and the result stored in accu. ACy.
MAS uns(Xmem),uns(Cmem), ACx:: **MAC uns(Ymem),uns(Cmem), ACy**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACx and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy and the result stored in accu. ACy.
AMAR Xmem:: **MAC uns(Ymem),uns(Cmem), ACx**	The content of auxiliary register is modified as specified by the content of memory location pointed by Xmem. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx and the result stored in accu. ACx.
MAS uns(Xmem),uns(Cmem), ACx:: **MAS uns(Ymem),uns(Cmem), ACy**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACx and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACy and the result stored in accu. ACy.
AMAR Xmem :: **MAS uns(Ymem),uns(Cmem), ACx**	The content of auxiliary register is modified as specified by the content of memory location pointed by Xmem. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACx and the result stored in accu. ACx.
MAC uns(Xmem),uns(Cmem), ACx >> #16:: **MAC uns(Ymem),uns(Cmem), ACy**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx right shifted by 16-bits and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy and the result stored in accu. ACy.

(Contd.)

Table 16.22 *(Contd.)*

MPY uns(Xmem),uns(Cmem), ACx:: **MAC uns(Ymem),uns(Cmem), ACy >> #16**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied, result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy right shifted by 16-bits and the result stored in accu. ACy.
MAC uns(Xmem), uns(Cmem), ACx >> #16:: **MAC uns(Ymem),uns(Cmem), ACy >> #16**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx right shifted by 16-bits and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy right shifted by 16-bits and the result stored in accu. ACy.
MAS uns(Xmem),uns(Cmem), ACx:: **MAC uns(Ymem), uns(Cmem), ACy >> #16**	The unsigned values in memory location pointed by Xmem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is subtracted from the content of ACx and the result stored in accu. ACx. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACy right shifted by 16-bits and the result stored in accu. ACy.
AMAR Xmem:: **MAC uns(Ymem),uns(Cmem), ACx >> #16**	The content of auxiliary register is modified as specified by the content of memory location pointed by Xmem. In parallel, the unsigned values in memory location pointed by Ymem and memory location addressed by coefficient data page pointer (CDP) are sign extended to 17-bits and multiplied. The product is added to the content of ACx right shifted by 16-bits and the result stored in accu. ACx.
AMAR Xmem, Ymem, Cmem	Two auxiliary register modify & one CDP modify operation executed in one cycle. The content of two auxiliary registers is modified as specified by the content of memory location pointed by Xmem & Ymem respectively. The content of coefficient data page pointer is modified as addressed by coefficient data page pointer CDP addressing mode.
LMS Xmem,Ymem, ACx, ACy **(MAC and Add)**	The content of memory locations pointed by Xmem & Ymem is sign extended to 17-bits and multiplied. The product is added to the content of accu. ACy and the result stored in ACy. In parallel, the content of memory location pointed by Xmem is left shifted by 16-bits and added to the content of accumulator ACx. The added result is rounded and stored in accumulator ACx.

(Contd.)

Table 16.22 *(Contd.)*

FIRSADD/FIRSSUB **Xmem, Ymem, Cmem, ACx, ACy** **FIRSADD - MAC and Add** **FIRSSUB - MAC and Subtract**	The higher order bits of accu. ACx (32-16) is multiplied with the content of memory location addressed by coefficient addressing mode Cmem, sign extended to 17-bits. The product is added to the content of accu. ACy and result stored in ACy. In parallel, the content of memory locations pointed by Xmem & Ymem are left shifted by 16-bits and added. The added result is stored in accumulator ACx.
SQDST Xmem, Ymem, ACx, ACy **(MAC and Subtract)**	The higher order bits (32-16) of the accumulator ACx is multiplied with itself (ACx*ACx – squaring) and added to the content of accumulator ACy, result stored in ACy. In parallel, the content of memory location pointed by Ymem left shifted by 16-bits is subtracted from the content of memory location pointed by Xmem left shifted by 16-bits. The subtracted result is stored in accumulator ACx.

16.6.8 AR, Tx, SP and XAR Register Access Operations

Instructions to modifty the conent of auxiliary registes (ARs), temporary registers (Tx), stack pointers (SP) and extended data page pointers (XDP) are available. The access of ARs, Tx and SP are performed in A-unit ALU and XDP access is done by D-unit ALU. Arithmetic operations can also be performed in ARs, Tx and SP registers. The instructions to access ARs, Tx, SP and XAR are given in Table 16.23.

Table 16.23 *AR, Tx , SP and XAR access Instructions of 'C55X*

Syntax	*Description*
AADD/ASUB TAx, TAy	Addition/subtraction between two temporary registers or auxiliary registers. The content of Tx/ARx is added to/subtracted from the content of Ty/ARy and the result is stored in Ty/ARy.
AMOV TAx, TAy	The content of temporary register Tx or auxiliary register ARx it moved to Ty or ARy content respectively.
AADD/ASUB #k8, TAx	The 8-bit unsigned constant (k8) is added to/subtracted from the content of temporary or auxiliary register and the result stored in Tx or ARx respectively.
AMOV #k8, TAx	The 8-bit unsigned constant (k8) is moved to temporary or auxiliary register.
AMOV #D16, TAx	The absolute data address signed constant D16 is loaded to temporary or auxiliary registers.
AMAR Smem	The auxiliary register is modified as specified by Smem
AADD #k8, SP	The 8-bit signed constant (k8) is sign extended to 16-bits and added to the content of stack pointer (SP)
MOV xsrc, xdst	Extended auxiliary register move instruction. The source and destination registers could be ACx (40-bits), XARx(23-bits), XSP(23-bits), XSSP(23-bits), XDP(23-bits), or XCDP(23-bits). If the source register is accumulator ACx, the lower order bits (22-0) is moved to destination pointer registers. If the source register is 23-bit pointer register and the destination register is accumulator ACx, the higher order bits of ACx (39-23) are extended with zero.
AMAR Smem, XAdst	The 23-bit destination register (XARx, XSP, XSSP, XDP, or XCDP) content is modified as per the effective address specified by the Smem operand field.

(Contd.)

Table 16.23 *(Contd.)*

AMOV #k23, XAdst	The 23-bit unsigned constant (k23) is loaded into the 23-bit destination register (XARx, XSP, XSSP, XDP, or XCDP).
MOV dbl(Lmem), XAdst	The lower 7-bits of the memory content pointed by Lmem and 16-bits of the memory content pointed by Lmem+1 are loaded into the 23-bit destination register (XARx, XSP, XSSP, XDP, or XCDP).
MOV XAsrc, dbl(Lmem)	The content of the 23-bit source register (XARx, XSP, XSSP, XDP, or XCDP) is stored to 32-bit data memory location addressed by data memory operand Lmem. The 7 MSBs of the register are stored into 7-LSBs of the memory location pointed by Lmem with 9-MSB of memory filled with zero and 16 LSBs of the register are stored into the memory location pointed by Lmem+1

PIPELINE OPERATION 16.7

The ′C55X instruction pipeline is a protected pipeline that has the following two decoupled segments:

- Fetch pipeline segment and
- Execution pipeline segment

16.7.1 Fetch Pipeline Segment

The first segment of the ′C55X pipeline is fetch pipeline. In this segment, the program memory address is passed to program memory; the 32-bit instruction packet is fetched from memory and placed in the 64 byte instruction buffer queue (IBQ). Then 48-bit instruction packet is fed to second pipeline segment. The phases of the fetch pipeline segment are Prefetch1, Prefetch2, Fetch and Predecode and are shown in Fig. 16.7(a). The operations performed in each phases of the fetch pipeline segment is given in Table 16.5.

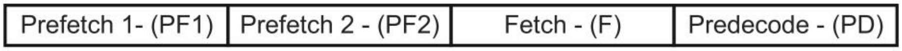


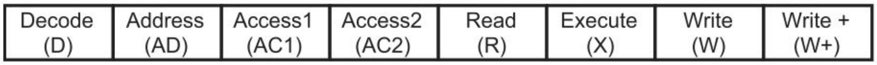
Fig. 16.7(a) *First Segment of ′C55X pipeline (Fetch Pipeline)*

Table 16.5 *Fetch pipeline segment phase operations*

Pipeline phase	*Description of the operation*
Prefetch1 – PF1	The 24-bit program memory address is passed to memory
Prefetch 2 – PF2	Wait for memory to respond
Fetch – F	The 32-bit instruction packet is fetched from memory and placed in the instruction buffer queue (IBQ)
Predecode – PD	The instructions in the IBQ are pre-decoded. Identifying where the instructions begin and end. Identifying parallel instructions.

16.7.2 Execution Pipeline Segment

The second segment of the pipeline phase is execution pipeline. In this segment the instructions are decoded, the data accesses are performed and computations are completed. The execution pipeline segment has eight phases namely decode, address, access1, access2, read, execute, write and write+ and are shown in Fig. 16.7(b). The operations performed in each phase of execute pipeline is given in Table 16.24.



Decode (D)	Address (AD)	Access1 (AC1)	Access2 (AC2)	Read (R)	Execute (X)	Write (W)	Write + (W+)

Fig. 16.7(b) *Second Segment of ′C55X pipeline (Execution Pipeline)*

Table 16.24 *Execution pipeline segment phase operations*

Pipeline phase	*Description of the operation*
Decode - D	The six byte instruction is read from the instruction buffer queue (IBQ). The instruction pair or a single instruction is decoded. The instructions to the appropriate CPU functional units are dispatched. The STx_55 bits associated with data address generation is read.
Address - AD	Registers involved in data address generation are read and address modification in registers is performed. The operations pertaining to A-unit 16-bit ALU are performed. For the conditional branch instruction using ARx, the content of ARx is decremented and braches on ARx being non zero. The condition of the XCC instruction is evaluated as an exception.
Access1 (AC1)	The data memory addresses are sent on the appropriate CPU address buses for memory read operation.
Access2 (AC2)	One cycle is allowed for memories to respond to read requests.
Read (R)	The data from memory and memory mapped register (MMR) addressed registers are read and passed to CPU. The A-unit registers are read, when executing specific D-unit instructions. The conditions of conditional instructions are evaluated except the exception cases.
Execute (X)	The registers that are not MMR addressed are read /modified. The individual register bits are read/modified. The conditions are set. The condition of the XCCPART and RPTCC instructions are evaluated as an exception.
Write (W)	The data write to MMR addressed registers or to peripheral registers are performed. From the perspective of CPU the data memory write operation is performed.
Write+ (W+)	From the perspective of memory, the data memory write operation is performed.

16.7.3 Pipeline Protection

The ′C55X pipeline is a protected pipeline. In an unprotected pipeline, when multiple instructions are executed simultaneously in the pipeline, where read and write at the same location lead to pipeline conflicts. The ′C55X has a mechanism that automatically protect against pipeline conflicts by adding inactive cycles between instructions that would cause conflicts. The pipeline protection cycles are inserted based on the following two rules:

- If an instruction is a write instruction to a location, but the previous instruction has not yet read from that location, extra cycles are inserted so that the read occurs first then the write can happen.
- If an instruction is a read instruction from a location, but the previous instruction has not yet written to that location, extra cycles are inserted so that the write occurs first followed by that read can happen.

INTERRUPTS 16.8

The ′C55X processor supports up to 32 interrupts. Some interrupts can be triggered by software or hardware, where others can be triggered only by software. All the interrupt in ′C55X are placed in two categories, nonmaskable and maskable interrupts. The maskable interrupts can be blocked through software.

The $\overline{\text{RESET}}$ and $\overline{\text{NMI}}$ are the two nonmaskable interrupts. These two interrupts can be triggered both by hardware and software. There are 30 maskable interrupts, in which IV2-IV29 are hardware or software interrupts and SIV30 and SIV31 are only software interrupts. The IV24 interrupt is bus error interrupt (BERR), IV25 is data log interrupt (DLOG) and IV26 is real-time operating system interrupt (RTOS). The interrupts IV27-IV29 are reserved.

The interrupt vector pointer IVPD and IVPH points up to 32 interrupt vectors in program space. IVPD points 256 byte program page for interrupts 0-15 and 24-31 and IVPH points to the 256 byte program page for interrupts 16-23. The registers used to enable the maskable interrupts are interrupt enable registers IER0 and IER1 and debug interrupt enable registers DBIER0 and DBIER1. The INTM bit is used to globally enable/disable all the maskabale interrupts.

PERIPHERALS 16.9

The ′C55X DSPs have the following common on-chip peripherals in all the devices

- Clock generator with PLL
- General purpose timer
- Multichannel serial ports (McBSP)
- Direct memory access controller (DMA)
- External memory interface (EMIF)
- Host port interface (HPI)
- Power management/Idle configurations

They also have the following certain other on-chip peripherals in some devices of ′C55X.

- Watchdog timer
- Analog-to-digital converter (ADC)
- Real-time clock (RTC)
- Instruction cache (IC)
- Inter-integrated circuit (I2C) module
- Universal asynchronous receiver/transmitter (UART)
- Universal serial bus module (USB)
- Multimedia card/SD card controller

The list of peripherals in various ′C55X DSPs and its quantity are given in Table 16.25.

Table 16.25 *TMS320C55X DSP Peripherals*

Description of the Peripheral	*′C5501*	*‘5502*	*′C5509*	*′C5510*
Clock generator with PLL	1	1	1	1
General purpose timer	2	2	2	2
Multichannel serial ports (McBSP)	2	3	3	3
Direct memory access controller (DMA),External memory interface (EMIF), Host port interface (HPI) & Power management/Idle configurations	1	1	1	1
Watchdog timer &Inter-integrated circuit (I2C) module	1	1	1	—
Instruction cache (IC)	1	1	—	1

(Contd.)

Table 16.25 *(Contd.)*

Universal asynchronous receiver/ transmitter (UART)	1	1	—	—
Analog-to-digital converter (ADC),Real-time clock (RTC) and Universal serial bus module (USB)	—	—	1	—
Multimedia card/SD card controller	—	—	2	—

Clock Generator with PLL The clock generator accepts an input clock at the CLKIN pin and enables to produce an output clock of desired frequency. The output clock signal is passed to CPU, peripherals and other modules inside ′C55X processor. A digital phase-lock loop (PLL) is included in the clock generator and it can be enabled or bypassed by programming clock mode register (CLKMD). The frequency of the input clock signal is divided by 1, 2 or 4 in PLL bypass mode. In PLL enable mode, the input clock signal is multiplied by a factor decided by PLL MULT and PLL DIV bits in the CLKMD register. The CPU clock can also be passed through a programmable clock divider to the CLKOUT pin. The frequency of CLKOUT pin depends on the CLKDIV bits of the system register (SYSR) and the divide value can be varied from 1 to 14 times of the CPU clock frequency.

General Purpose Timer The ′C55X processors have two identical 20-bit software programmable timers. These timers are used to generate periodic clock signal for the devices outside the processor and to generate periodic interrupts. The general-purpose timer has up to a 20-bit dynamic range provided by two counters, a 4-bit prescaler counter and a 16-bit main counter. The timer has two count registers PSC and TIM along with two corresponding period registers TDDR and PRD of size 4-bits and 16-bits respectively. The operation is same as that of timer in ′C5X device except that the PSC and TDDR bits are present separately in timer prescaler register (PRSC) instead of in timer control register (TCR) and more control bits are introduced in TCR.

Multichannel Serial Ports (McBSP) The ′C55X DSPs have 2/3 high-speed, multichannel buffered serial ports that allow direct interface to other C55x DSPs, codecs, and other devices in a system. The operation and features of McBSP in ′C55X device is same as that in ′C6X devices (refer chapter 15.5.2).

External Memory Interface (EMIF) The ′C55X EMIF controls all data transfers between the DSP and external memory. The EMIF provides a glueless interface to the following three types of memory devices:

- Asynchronous devices, including ROM, flash memory and asynchronous SRAM.
- Synchronous burst SRAM (SBSRAM)
- Synchronous DRAM (SDRAM)

The EMIF also supports the following types of accesses

- 32-bit instruction fetches for the CPU or the instruction cache
- 8, 16 and 32-bit data accesses for the CPU or the DMA controller

Host Port Interface (HPI) The HPI enables an external host processor to directly access a portion of the memory in the memory map of the ′C55X DSP through a 16-bit-wide parallel port. The host and the DSP can exchange information via memory internal or external to the DSP and within the address reach of the HPI. The HPI uses 20-bit addresses, where each address is assigned to a 16-bit word in memory.

The DMA controller handles all HPI accesses. Through the DMA controller, one of two HPI access configurations can be chosen. In one configuration, the HPI shares internal memory with the DMA channels. In the other configuration, the HPI has exclusive access to the internal memory. The HPI cannot directly access other peripherals' registers. If the host requires data from other peripherals, then the data must be moved to memory first, either by the CPU or by activity in one of the six DMA channels. Same way data from the host must be transferred to memory before being transferred to other peripherals.

The HPI allows two modes for passing data and addresses, the nonmultiplexed mode and the multiplexed mode. In nonmultiplexed mode the host processor is provided with separate address and data buses, but in multiplexed mode it provides a single bus to transport address and data information. The following three HPI registers that a host can be used to access the memory of the DSP:

- HPI data register –HPID
- HPI address register – HPIA
- HPI control register - HPIC

Watchdog Timer The watchdog timer available in certain devices of ′C55X and it is to prevent a system from locking up if the software becomes trapped in loops with no controlled exit. It provides an automatic mechanism for recovery from application software error conditions by counting down for a pre-defined number of cycles and used to trigger an interrupt or a DSP reset.

The watchdog timer consists of a prescaler up to 16- bit resolution followed by a 16-bit main counter, which provides a counter up to 32-bit dynamic range. The watchdog timer is disabled after reset allowing the application software to configure it before it is enabled. After the watchdog timer is enabled, it cannot be disabled without a DSP reset or timeout condition due to software error.

Analog-to-Digital Converter (ADC) The on-chip ADC is available in ‘VC5507 and ′C5509 devices. The 10-bit successive approximation ADC converts an analog input signal to a digital value for use by the DSP. There are four input AIN0-AIN3 and the ADC can sample one of the inputs with a maximum sampling rate of 21.5KHz and generates a 10-bit digital representation. This ADC is suitable for sampling analog signals that change at a slow rate.

Real-Time Clock (RTC) The RTC provides a time reference and the capability to generate time-based alarms to interrupt the DSP. The current date and time is tracked in a set of counter registers that update once per second. The time can be represented in 12-hour or 24-hour mode. The calendar and time registers are buffered during reads and writes so that updates do not interfere with the accuracy of the time and date. Alarms are available to interrupt the CPU at a particular time, or at periodic time intervals, such as once per minute or once per day. In addition, the RTC can interrupt the CPU every time the calendar and time registers are updated, or at programmable periodic intervals. The real-time clock (RTC) provides the following features:

- 100-year calendar up to year 2099
- Counts seconds, minutes, hours, day of the week, date, month, and year with leap year compensation
- Binary-coded-decimal (BCD) representation of time, calendar, and alarm
- 12-hour clock mode (with AM and PM) or 24-hour clock mode
- Second, minute, hour, day, or week alarm interrupt
- Update cycle interrupt
- Periodic interrupt

- Single interrupt to the DSP CPU
- Supports external 32.768-kHz crystal or external clock source of the same frequency
- Separate isolated power supply

Instruction Cache (IC) Some ′C55X devices have instruction cache. The instructions can reside in internal memory or external memory. When instructions reside in external memory, the instruction cache (I-Cache) can improve the overall system performance by buffering the most recent instructions accessed by the CPU. The CPU status register ST3_55 contains three cache control bits for enabling, freezing, and flushing the I-Cache. To configure the I-Cache and to check its status, the CPU accesses a set of registers in the I-Cache. For storing instructions, the I-Cache has the following:

- One 2-way cache. The 2-way cache uses 2-way set associative mapping and holds up to 16K bytes. It has 512 sets, two lines per set, four 32-bit words per line. In the 2-way cache, each line is identified by a unique tag.
- Two RAM sets. These two banks of RAM are available to hold blocks of code. Each RAM set holds up to 4K bytes. It has 256 lines, four 32-bit words per line. Each RAM set uses a single tag to identify a continuous range of memory addresses that is represented in the RAM set. Before enabling the I-Cache, configure the I-Cache to use zero, one, or both RAM sets.

Inter-Integrated Circuit (I2C) Module The I2C module provides an interface between one of the ′C55X DSPs and devices compliant with Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1 and connected by way of an I2C-bus. External components attached to this 2-wire serial bus can transmit/receive 1to 8-bit data to/from the ′C55X DSP through the I2C module.

Universal Asynchronous Receiver/Transmitter (UART) The UART peripheral is based on the industry-standard TL16C550 asynchronous communications element. The UART can be placed in an alternate FIFO mode and relieves the CPU of excessive software overhead by buffering received and transmitted characters. The receiver and transmitter FIFOs, store up to 16 bytes including three additional bits of error status per byte for the receiver FIFO. The UART performs serial-to-parallel conversions on data received from a peripheral device and parallel-to-serial conversion on data received from the CPU. The CPU can read the UART status at any time. The UART includes control capability and a processor interrupt system that can be tailored to minimize software management of the communications link. The UART includes a programmable baud generator capable of dividing the UART input clock by divisors from 1 to 65535 and producing a 16X reference clock for the internal transmitter and receiver logic.

Universal Serial Bus Module (USB) Using the USB module, the ′C55X DSP can be used to create a full speed (12Mbps) USB slave device that is compliant with Universal Serial Bus Specification Version 2.0. The ′C55X USB module has the following 16 endpoints:

- Two control endpoints OUT0 and IN0 for *co*ntrol transfers *on*ly.
- Fourteen general-purpose endpoints OUT1-OUT7 and IN1-IN7, for other types of transfers.

Each of these endpoints can support the following:

- Bulk, interrupt and isochronous transfers.
- An optional double-buffer scheme for fast data throughput.
- A dedicated DMA channel.

A DMA controller inside the USB module can pass data between the general-purpose endpoints and the DSP memory while the CPU performs other tasks. The USB DMA controller cannot access the control endpoints.

Multimedia Card/SD Card Controller The multimedia/SD card controller supports both the MultiMediaCard (MMC) protocol and the Secure Digital (SD) Memory Card protocol. The controller has a programmable option for the frequency of operation of the MMC controller and for the clock that controls the timing of transfers between the MMC controller and the memory card. The MMC controller passes data between the CPU or the DMA controller on one side and one or more a memory cards on the other side. The CPU or the DMA controller can read from or write to the control and status registers in the MMC controller. The CPU and/or the DMA controller can store or retrieve data in the DSP memory or in the registers of other peripherals. Data transfers between the MMC controller and a memory card can use one bidirectional data line (for the MMC protocol) or four parallel data lines (for the SD protocol). If multiple cards are connected, the MMC controller uses commands of the MMC/SD protocol to select and communicate with one card at a time.

Review Questions

16.1 What are the key features of TMS320C55X processors?

16.2 Compare the features of TMS320C54X and TMS320C55X processors.

16.3 What are the important units in the CPU of 'C55X processor?

16.4 Explain the function of I-unit.

16.5 List the various buses present in 'C55x processor.

16.6 Explain the opration of program folw unit (P-unit).

16.7 List the registers present in the program flow unit.

16.8 What are the operations performed by the address generation unit (A-unit) ALU?

16.9 What is the function of address generation unit (A-unit)?

16.10 What are the various functional units present in data computation unit (D-Unit)?

16.11 Explain the actions performed by the D-unit.

16.12 Explain the on-chip memory details of 'C55X processor.

16.13 What are the various addrssing modes of 'C55x processor?

16.14 Explain the types of abosulte addressing mode.

16.15 How DP and SP are used for direct addressing mode?

16.16 List the types of indirect addressing mode access.

16.17 Explain the important points to be remembered in indirect addressing mode?

16.18 What are the various ways the address modification can be done in AR indirect addressing mode?

16.19 Explain about CDP indirect addressing mode and its uses.

16.20 How circular buffer can be initialized in 'C55X? Explain.

16.21 Explain different ways memory mapped register (MMR) can be accessed in 'C55X.

16.22 What are the different types of arithmetic instructions in 'C55X?

16.23 Explain about the logical operations in 'C55X.

16.24 What are the different categories of move oprations in 'C55X? Explain.

16.25 Explain about the compar operation in 'C55X processor.

16.26 What are the various test conditions present in 'C55X processor?

16.27 List the different parallel operations that can be executed in 'C55X.

16.28 How AR, SP and CDP registers are accessed? Explain.

16.29 Explain the different phases of 'C55X pipeline.

16.30 List the various interrupts in 'C55x processor.

16.31 What are the peripheral devices present in 'C55X processor?

Self Test Questions

16.1 The bit size of the 'C55X processor is __
(a) 16 (b) 32 (c) 8 (d) 24

16.2 The size of 'C55X CPU is __
(a) 16 (b) 32 (c)40 (d) 24

16.3 The number of ALUs in 'C55X CPU is ___
(a) 1 (b) 2 (c) 4 (d) 6

16.4 The size of the instrcion buffer queue is __
(a) 32x 16 (b) 32 x32 (c) 16x16 (d) 24x16

16.5 The numer of MAC units in 'C55X processor is ___
(a) 1 (b) 2 (c) 4 (d) 6

16.6 The number of accumulators in 'C55X processor is ___
(a) 1 (b) 2 (c) 4 (d) 6

16.7 The number of auxiliary register (ARs) units in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 6

16.8 The number of read buses in 'C55X processor is ___
(a) 2 (b) 3 (c) 4 (d) 6

16.9 The number ofwrite buses in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 6

16.10 What is the size of program read address bus?
(a) 16 (b) 32 (c) 23 (d) 24

16.11 The size of data read address bus is ___
(a) 16 (b) 32 (c) 23 (d) 24

16.12 The number of status registers present in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 6

16.13 The number of block repeat counters (BRC) present in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 6

16.14 The size of XAR and XDP is ___
(a) 16 (b) 32 (c) 23 (d) 24

16.15 The size of program/data space of 'C55X processor is ___
(a) 64K words (b) 16 M words
(c) 16 M bytes (d) 64 K bytes

16.16 The size of I/O of 'C55X processor is ___
(a) 64K words (b) 16 M words
(c) 16 M bytes (d) 64 K bytes

16.17 In absolute addressing mode the size of unsigned constants used are ___
(a) 8 & 16 (b) 16 &23 (c) 4 & 8 (d) 16 &24

16.18 In AR indirect addressing the linear/circular address modification depends on __ bit.
(a) ARnLC (b) CDPLC (c) C54CM (d) ARMS

16.19 In CDP indirect addressing linear/circular address modification depends on __ bit.
(a) ARnLC (b) CDPLC (c) C54CM (d) ARMS

16.20 The number of circular buffers present in 'C55X processor is ___
(a) 3 (b) 5 (c) 4 (d) 6

16.21 The numer of memory mapped CPU registers in 'C55X processor is___
(a) 80 (b) 96 (c) 120 (d) 128

16.22 The number of test control flag bits in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 1

16.23 The number of phases in fetch pipeline segment of 'C55X is ___
(a) 4 (b) 6 (c) 8 (d) 5

16.24 The number of phases in execute pipeline segment of 'C55X is ___
(a) 4 (b) 6 (c) 8 (d) 5

16.25 The number of nonmaskable interrupts in 'C55X processor is ___
(a) 3 (b) 2 (c) 16 (d) 1

16.26 The number of maskable interrupts in 'C55X processor is ___
(a) 8 (b) 16 (c) 30 (d) 32

16.27 The number of on-chip timers in 'C55X processor is ___
(a) 3 (b) 2 (c) 4 (d) 1

16.28 The size of the timer in 'C55X processor is ___
a) 16 (b) 32 (c) 24 (d) 20

16.27 The maximum sampling rate for the ADC in 'C55X processor is ___
(a) 8 KHz (b) 19.2 KHz (c) 21.5 KHz (d) 44.1 KHz

17 RECENT TRENDS IN DSP SYSTEM DESIGN

In the previous chapters, some of the simple applications such as waveform generation, convolution of sequences, Symmetric FIR filter are used to illustrate the use of programmable DSPs. The real world DSP applications are more sophisticated than them. To appreciate the computational requirements as well as the storage requirements for using the P-DSPs for these advanced applications, thorough understanding of the various issues relating to the particular application is required. For example, to design an echo canceller scheme for cellular telephone applications, the various sources of echo, the models used for the echo , the strength of the desired signal, the extent to which the echo needs to be cancelled should be known. One may not always have the luxury of implementing a filter whose filter coefficients and the sampling rates are completely specified. Since diverse applications require the knowledge on diverse fields, to keep the treatment simple, these applications are not discussed in detail in this book. However, a brief description of some of the real world applications where the P-DSPs are deployed is given in Section 17.1.

In the recent past, DSP systems are also being built around an alternate approach based on Field Programmable Gate arrays (FPGA). An introduction to this approach is presented in Section 17.2-17.10. This approach is also compared with that using the P-DSPs in Section 17.11.

AN OVERVIEW OF THE APPLICATION NOTES ON DSP SYSTEMS 17.1

Texas instruments official web site www.ti.com contains rich information on the TI DSPs, their applications and interfacing details. To access the documents pertaining to a particular digital signal processor, the following steps may be followed:

- Select technical documents on home page
- In the product type, select *Digital signal processors & ARM processor platforms*
- Then choose the name of processor such as *TMS320c54X* and document type required (such as application notes)

The web site includes details on the TI DSP processors, application notes on programming these devices, interfacing details, details on the C compiler, and no. of application reports. A brief summary of some of these application reports are given next. Some of these reports are also available in Papamichalis et al [1991].

The report on Telecommunication applications with TMS30c5X has an exhaustic account of the theory and implementation of various applications using the 5X processor. This describes how the systems for speech synthesis, error-correction coding, baseband modem, etc. required for Digital Cellular Systems can be implemented using general-purpose DSPs. This report also includes an application paper on U.S. Digital Cellular vocoder implementation using 5X. Details of Concatenated coding schemes used to provide protection against bit errors using both the CRC and convolutional codes is discussed in this report. Details on Forward error correcting coding (FEC) schemes and implementation of forward error-correction technique used for V.32 Modems is also discussed in this report.

The implementation of the U.S. Digital Cellular IS-54 standard modem for mobile phones using 5X is discussed in this report. Additional topics discussed in this report include the following:

- U.S. digital cellular error-correction coding algorithm on TMS320c5x DSPs
- Viterbi implementation on the TMS320c5x for V.32 modems
- Automated dialing of cellular telephones using speech recognition
- Channel equalization for the IS-54 digital cellular system with the TMS320c5x
- Digital voice echo canceler implementation on the TMS320c5x
- DSP-Based Handprinted Character Recognition

In the report, *DSP solutions for Telephony and data/fax modem,* a detailed account of the principle of operation of Telephony as well as Modems is presented first. Next the implementation details of voice mail systems using TI DSPs for the following operations

- Tone detection and generation
- DTMF generation and detection
- Voice compression and decompression using ADPCM

and implementation of functions like Line echo cancellation, acoustic echo cancellation required for Full duplex speakerphone and modem applications are discussed in detail. Another application discussed in this report is transmission of Caller Identification (CID) information from the telephone company, via the local loop, to the subscriber's CID unit.

All the above applications can also be implemented in TMS320C54x processors. The report, *Viterbi Decoding Techniques in the TMS320C54x Family,* gives an outline of the theory of convolutional coding and decoding and explains the programming techniques for Viterbi decoding in the Texas Instruments (TI) TMS320C54x family of digital signal processors (DSPs). Some of the other applications discussed in the C5000 website includes the following

- Extended Precision IIR Filter Design on the TMS320C54x
- TMS320C54x Digital Filters
- Fast Fourier Transform Algorithms of Real-Valued Sequences with TMS320 Family
- Implementation of the Double-Precision Complex FFT for the TMS320C54x DSP
- Overflow Avoidance Techniques in Cascaded IIR Filter Implementation on TMS320 DSPs
- Implementation of a Software UART on TMS320C54x Using I/O Pins
- TMS320C5000 DMA Applications

Some of the applications on 3X discussed in the website is listed below.

Some of them are also available on the above CD.

- Engine knock detection using spectral analysis with TMS320c25 or TMS320c30 dsps
- Enhanced control of an ac motor using fuzzy logic and a TMS320 dsp
- Integrated automotive signal processing and audio system using a TMS320c3x dsp
- FFT, DCT, and Other Transforms on TMS320C30

- A DSP-Based Three-Dimensional Graphics System
- Adaptive Active Noise Control for Headphones Using the TMS320C30 DSP
- Adaptive Filters With TMS320C25 or TMS320C30
- CELP Speech Coder for TMS320C30 Using SPOX
- Implementing a Fast 3-D Vision Sensor With Multiple TMS320C31 DSPs
- Integrated Automotive Signal Processing and Audio System Using a TMS320C3x DSP

AN OVERVIEW OF OPEN MULTIMEDIA APPLICATIONS PLATFORM(OMAP) 17.2

The OMAP architecture is based on a combination of Texas Instruments TMS320 DSP core (such as 55x, 64x) and high performance RISC processor such as ARM925T, ARM Cortex A8 and A9 CPU. It is targeted for 2.5G and 3G wireless systems requiring advanced video and speech processing tasks such as encoding and decoding video/audio data, data compression, motion compensation, pixel Interpolation, speech recognition and synthesis. OMAP combines both DSP and RISC core in a single IC in order to gain the maximum benefits from both. The RISC architecture is well suited for execution of control instructions commonly required for Operating System (OS), man-machine Interfaces and OS applications. DSP is best suited for signal processing applications, such as MPEG4 video, speech recognition, and audio playback. A comparative benchmarking study [Jamil Chaoui,2001], using StrongARM™, ARM9E™ and C55x™ DSP shows that signal processing task such as echo cancellation, MP3, MPEG4 and JPEG decoding executed on these RISC machines requires three times more cycles compared to that required for execution on a C55x™ DSP. In terms of power consumption, tests show that a given signal-processing task executed on such a RISC engine consumes more than twice the power required to execute the same task on a C55x DSP architecture.

Hence, battery life is much greater when such tasks are executed on a DSP.The OMAP architecture's use of two processors provides this kind of power consumption benefits. At the same time, it allows the DSP to gain support from the RISC processor. For instance, a single C55x DSP can process, in real time, a full videoconferencing application (audio and video at 15 images/sec.), using only 40 percent of the available computational capability. Therefore, 60 percent of the capacity can be employed to run other applications concurrently. At the same time, in the OMAP dual-core architecture, the ARM processor stands ready to handle any other application requirements or can be suspended, thus saving battery life. As a result, the mobile user can enjoy access to popular OS applications (Word™, Excel™, etc.) while also engaging a videoconferencing application.

Both processors utilize an instruction cache to reduce the average access time to instruction memory and eliminate power-hungry external accesses. In addition, both cores have a memory management unit (MMU) for virtual-to-physical memory translation and task-to-task memory protection. The OMAP core contains two external memory interfaces and one internal memory port. The external memory interfaces support direct connection to synchronous DRAMs and to standard asynchronous memories, such as SRAM, FLASH, or burst FLASH devices. The latter interface is typically used for program storage. The OMAP core also contains numerous interfaces to connect to peripherals or external devices from either the DSP or GPP.

To support common operating system requirements, the OMAP architecture includes several peripherals, timers, general purpose input/ output interfaces (I/Os), UART, and watchdog timers. These are the minimum peripherals required in the system; other peripherals can be added on the TI peripheral bus (TIPB) interfaces.

OMAP supports high-level operating systems (OSs), such as Linux and Windows CE. OMAP has a number of on chip hardware accelerator such as graphics accelerator and image processing hardware for DCT, IDCT, Pixel Interpolation and Motion Estimation. OMAP has four families OMAP1 – OMAP4. The type of RISC CPU, DSP and on chip hardware accelerators in an OMAP device depends on the family. Many of these OMAP devices such as that belonging to OMAP2 are available only for high volume manufacturers such as the vendors of Internet Tablets and mobile phones. Table 17.1 gives a sample list of devices belonging to different families. It may be noted that some OMAP devices do not contain 320x family DSP. Devices such as OMAP3530 also contain DAC. Functional block diagram of OMAP1510 and OMAP3530 are shown in Fig.17.1 and 17.2 respectively.

Table 17.1 *Features of some OMAP devices*

Device No.	*RISC CPU*		*DSP*		*h/w accelerator*
	No.	*speed*	*No.*	*speed*	
OMAP1510	ARM925T	168 MHz	C55x	200 MHZ	Dct, Idct, Pixel Interpolation, Motion Estimation
OMAP2431	ARM1136	330 MHz	C64x	220 MHz	—
OMAP2420	ARM1136	330 MHz	C55x	220 MHz	PowerVR MBX GPU
OMAP3410	ARM Cortex A8	600 MHz	C64x	430MHz	PowerVR SGX 530 GPU
OMAP3530	ARM Cortex A8	720 MHz	C64x	520 MHz	PowerVR SGX 530 GPU
OMAP4430	ARM Cortex A9	720 MHz	—	—	PowerVR SGX 540 GPU, DSP ISP, IAV3

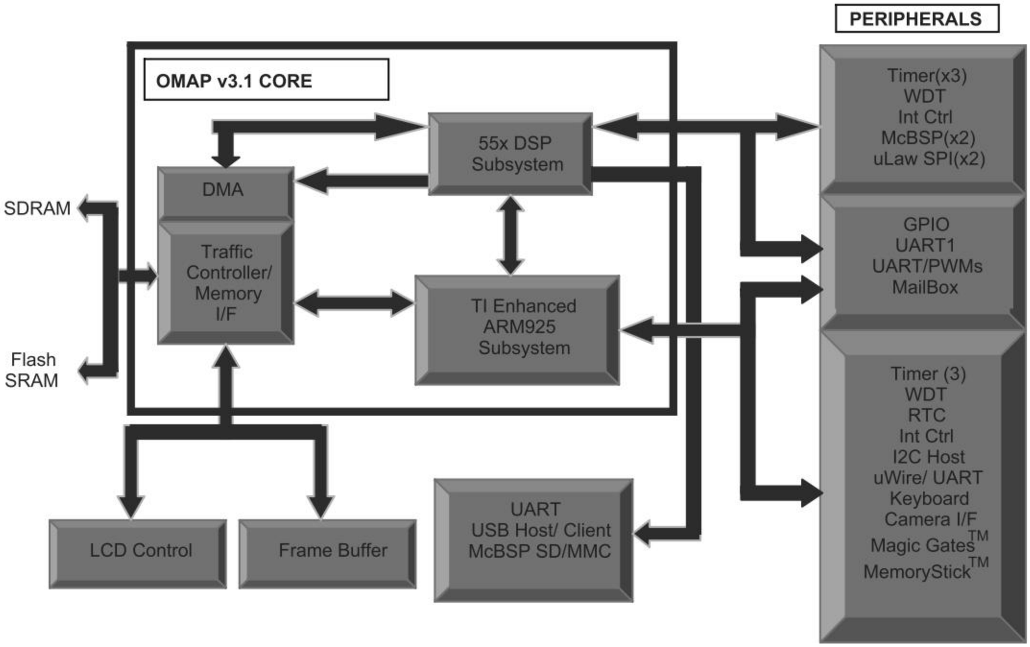

PERIPHERALS
OMAP v3.1 CORE
55x DSP Subsystem
DMA
SDRAM
Traffic Controller/ Memory I/F
TI Enhanced ARM925 Subsystem
Flash SRAM
Timer(x3) WDT Int Ctrl McBSP(x2) uLaw SPI(x2)
GPIO UART1 UART/PWMs MailBox
Timer (3) WDT RTC Int Ctrl I2C Host uWire/ UART Keyboard Camera I/F Magic Gates™ MemoryStick™
LCD Control
Frame Buffer
UART USB Host/ Client McBSP SD/MMC

Fig. 17.1 *Functional block diagram of OMAP 1510 (Courtesy of Texas Instruments inc.)*

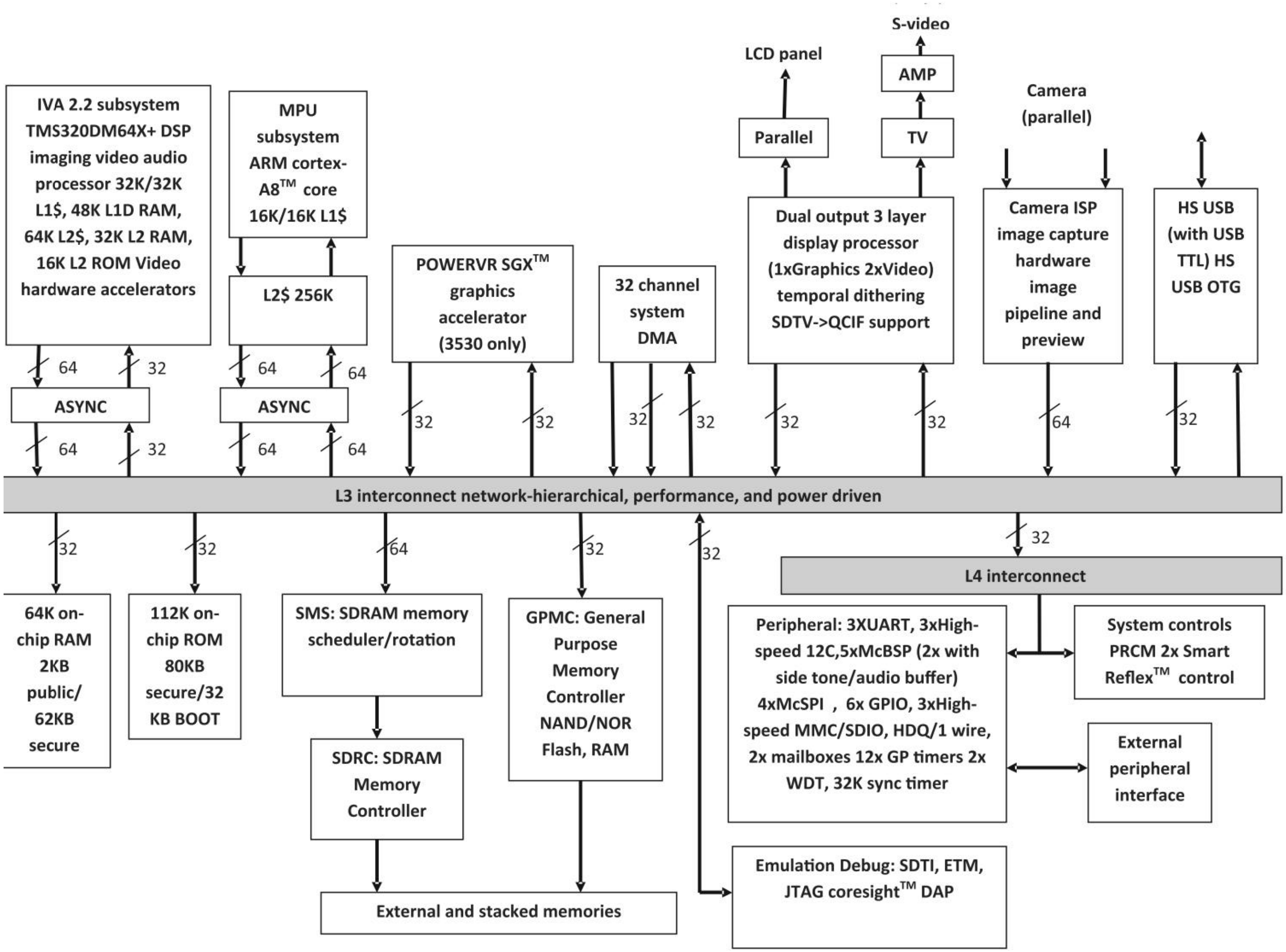

IVA 2.2 subsystem TMS320DM64X+ DSP imaging video audio processor 32K/32K L1$, 48K L1D RAM, 64K L2$, 32K L2 RAM, 16K L2 ROM Video hardware accelerators
MPU subsystem ARM cortex-A8™ core 16K/16K L1$
L2$ 256K
ASYNC
ASYNC
64
32
64
64
POWERVR SGX™ graphics accelerator (3530 only)
32 channel system DMA
Dual output 3 layer display processor (1xGraphics 2xVideo) temporal dithering SDTV->QCIF support
Parallel
LCD panel
TV
AMP
S-video
Camera (parallel)
Camera ISP image capture hardware image pipeline and preview
HS USB (with USB TTL) HS USB OTG
L3 interconnect network-hierarchical, performance, and power driven
L4 interconnect
64K on-chip RAM 2KB public/ 62KB secure
112K on-chip ROM 80KB secure/32 KB BOOT
SMS: SDRAM memory scheduler/rotation
SDRC: SDRAM Memory Controller
GPMC: General Purpose Memory Controller NAND/NOR Flash, RAM
External and stacked memories
Peripheral: 3XUART, 3xHigh-speed 12C,5xMcBSP (2x with side tone/audio buffer) 4xMcSPI , 6x GPIO, 3xHigh-speed MMC/SDIO, HDQ/1 wire, 2x mailboxes 12x GP timers 2x WDT, 32K sync timer
System controls PRCM 2x Smart Reflex™ control
External peripheral interface
Emulation Debug: SDTI, ETM, JTAG coresight™ DAP

Fig. 17.2 *Functional block diagram of OMAP5330 (Courtesy of Texas Instruments inc)*

The OMAP architecture abstracts the implementation of the DSP software architecture from the GPP environment. This is achieved defining an interface scheme that allows the GPP to be the system master. This interface scheme is called the DSP/BIOS™ Bridge and consists of a set of APIs that includes device driver interfaces. The most important function of the DSP/BIOS Bridge is providing communications between GPP applications and DSP tasks. The application developers develop programs on the OMAP platform as if they were developing on a single RISC processor. The environment provided for development allows the application developer to call the localized functions for video, audio, speech, etc. and to develop in the traditional manner on platforms such as the PC. The high-level application developer does not require any awareness of the DSP or DSP/BIOS Bridge API. The DLL and driver developers actively use the DSP/BIOS Bridge API to: Initiate and control tasks on the DSP, Exchange messages with the DSP Stream data to and from the DSP and Perform status queries.

EVOLUTION OF FPGA BASED DSP SYSTEM DESIGN 17.3

The digital signal processors have completed one complete cycle. In the 1970's, some of the DSP systems like filters and equalizers were built in hardware using multipliers, shift registers and adders. However, towards the end of 1970's, processing the signal in transform domain appeared to be attractive and several computationally efficient algorithms such as FFT were invented which enabled several DSP problems to be solved using computers. With the advent of fast and cost effective computers, this trend continued in 1980's. Some of the special purpose hardware such as multipliers and multiply accumulate units as well as memory were embedded with the microprocessors in the programmable DSPs considered in the previous chapters. The availability of very high density FPGAs since 1990's have again attracted attention of the designers in pure hardware oriented solutions in preference to the processor oriented solutions. For example, one of the FPGA manufacturers, Xilinx introduced Virtex E family of FPGAs with 3.2 million gates and 622 MHz differential I/O performance in the Fourth quarter of 1999 (see for eg. Data source [2000]). In the first quarter of 2009, Xilinx introduced the Virtex 6 FPGA which has 760,000 logic cells, 38 Mb block RAM, 6.5 Gbps serial transceiver and support for Microblaze for soft core processor. In this section, a brief overview of a few FPGA families and some algorithms used for implementation of systems on FPGAs are presented.

AN INTRODUCTION TO FPGA 17.4

FPGAs are fabricated in Integrated Circuit form and consist of an array of logic cells. Each logic cell may be used individually to implement simple logic functions. Alternately, the outputs of several logic cells may be combined in one or more logic cells to realize complex logic functions. In FPGAs, programmable interconnect elements are used to interconnect the input of the logic cells to either one of the input variables from the IC pad or the output of another logic cell. The programmable interconnect elements may be either antifuses or transistor switches (pass transistors). The state of the interconnect element determines the function performed by the logic cell.

The FPGAs from the companies Actel and Quicklogic use antifuses as the interconnect elements. The antifuses have high impedance in unprogrammed state. Passing a current of the order of 10mA through them convert them to low impedance state permanently. Hence, the antifuse based FPGAs can be programmed only once.

The FPGAs from the companies Altera and Xilinx use transistor switches as the interconnect elements. In this case, by choosing the required logic level to the gate of the transistor switches, they can be switched on or off. The state of the various switches in the FPGA is normally stored in static RAMs (SRAM) and hence they are called SRAM based FPGAs. The function performed by the FPGA is programmed by storing the required pattern in the SRAM. SRAM based FPGAs can be programmed any number of times. Further, the function performed by the FPGAs can be programmed even after it is mounted onto the board. For this reason, they are called as in system programmable. The Antifuse based FPGAs are not in system programmable. Since the content of SRAM is lost when power is switched off, to keep the SRAM content intact, normally the configuration data is stored in a serial PROM and is transferred to the SRAM whenever the FPGA is powered up. By changing the contents of serial PROMs, the function performed by the FPGA can be altered.

Over the years, Xilinx has developed a variety of FPGA families such as XC4000, Spartan, Spartan II, Spartan III, Spartan VI, Virtex, Virtex II, Virtex II PRO, Virtex III , IV, V and VI . They have different speed, power dissipation, cost and complexity. Similarly, Altera has come up with different FPGA families such as Flex, Apex,Acex, Excalibur,Cyclone, Cyclone II, Cyclone III, Cyclone IV, Arria, Arria II, Stratix, Stratix II, Stratix III, Stratix IV FPGAs. As newer families of FPGAs with better features are introduced in the market, supply of devices of older families are gradually discontinued. In the next sections, a brief overview of four FPGA families XC4000, Spartan III, Cyclone III and Virtex II PRO are presented.

17.4.1 An Overview of Xilinx XC4003E FPGA

This belongs to the XC4000 family FPGA from Xilinx. The individual family members differ only in the number of configurable logic blocks, input output blocks, I/O pins, package on which they are offered and their speed. They contain the same building blocks. XC4003E is an SRAM based FPGA containing 100 logic cells. Each of the logic cell of XC4000 family FPGA is referred to as a configurable logic block (CLB). The 100 CLBs in XC4003E is organized as a 2D array of 10X10 CLBs. A simplified functional block diagram of a CLB is given in Fig.17.3. It consists of two 4X1 look up tables(LUT), one 2X1 LUT, two F/Fs and additional circuitry for fast carry logic and control function. The CLB can also be configured as a single 5X1 LUT. Alternately, it can be configured as two 16X1 RAM or a single 32X1 RAM. A CLB in XC4003E can be used to implement 2 four variable Boolean functions, or one 5 variable Boolean function or a limited no. of 6 variable Boolean functions. By using more CLBs, more complex functions can be implemented.

17.4.2 An Overview of Xilinx Spartan 3 Family FPGAs

The Spartan-3 family architecture shown in Fig.17.4 consists of five fundamental programmable functional elements:

Configurable Logic Blocks (CLBs) The CLBs contain RAM-based Look-Up Tables (LUTs) to implement logic and storage elements that can be used as flip-flops or latches. The no. of CLBs in a device varies from 192-8320 CLBs depending the device chosen. CLBs can be programmed to perform a wide variety of logical functions as well as to store data. Each CLB comprises four interconnected slices, as shown in Fig.17.5. These slices are grouped in pairs. Each pair is organized as a column with an independent carry chain.

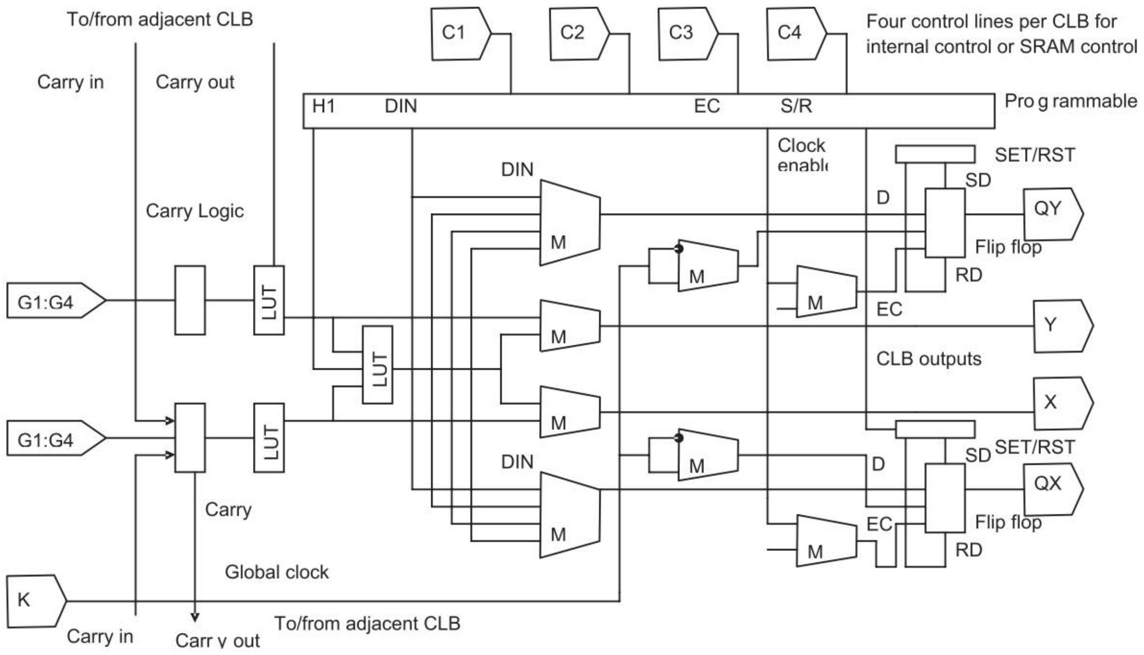


Fig. 17.3 *Function block diagram of XC4000E family CLB (Courtesy of Xilinx, Inc.)*

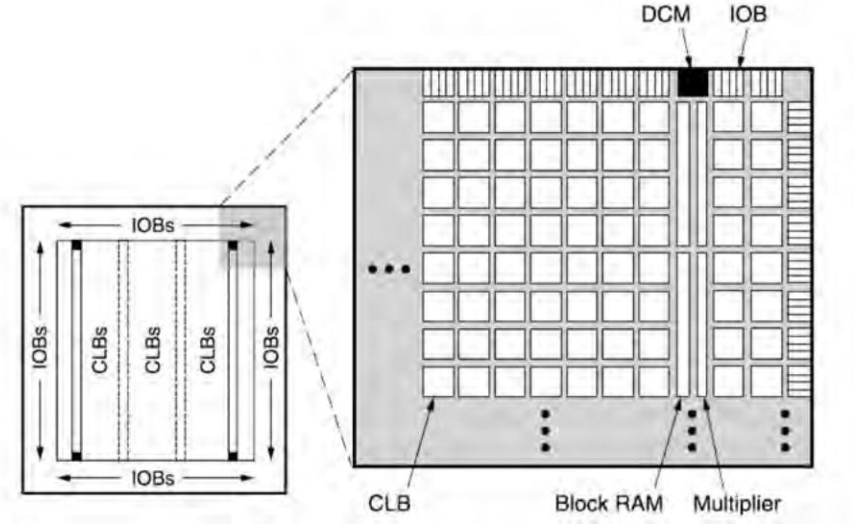


Fig.17.4 *The Spartan-3 family architecture (Courtesy of Xilinx, Inc.)*

The letter 'X' followed by a number identifies columns of slices. The 'X' number counts up in sequence from the left side of the die to the right. The letter 'Y' followed by a number identifies the position of each slice in a pair as well as indicating the CLB row. Fig.17.5 shows the CLB located in the

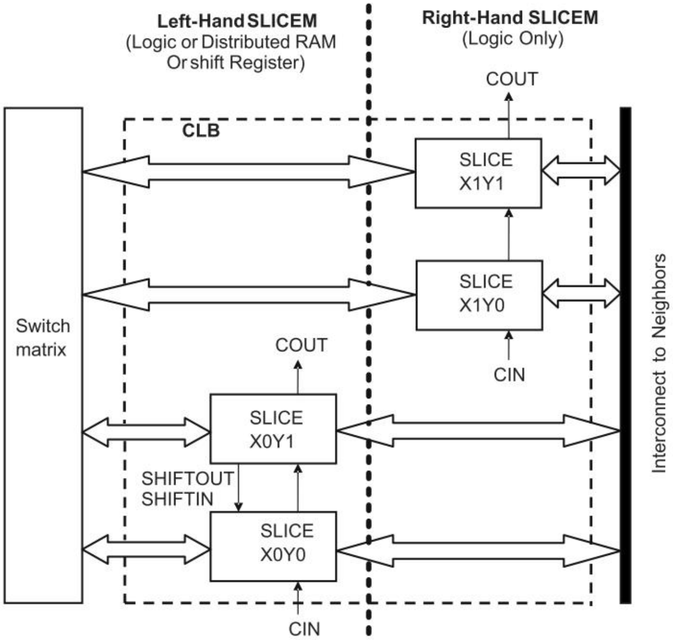


Fig. 17.5 *Organisation of the CLB located in the lower left-hand corner of the die (Courtesy of Xilinx, Inc.)*

lower left-hand corner of the die. Slices X0Y0 and X0Y1 make up the column-pair on the left whereas slices X1Y0 and X1Y1 make up the column-pair on the right. For each CLB, the term "left-hand" (or SLICEM) indicates the pair of slices labeled with an even 'X' number, such as X0, and the term "right-hand" (or SLICEL) designates the pair of slices with an odd 'X' number, e.g., X1.

All four slices have the following elements in common: two logic function generators, two storage elements, wide-function multiplexers, carry logic, and arithmetic gates, as shown in Fig.17.6 . Both the left-hand and right-hand slice pairs use these elements to provide logic, arithmetic, and ROM functions. Besides these, the left-hand pair supports two additional functions: storing data using Distributed RAM and shifting data with 16-bit registers. Fig. 17.6 is a diagram of the left-hand slice. The function generators located in the upper and lower portions of the slice are referred to as "G" and "F", respectively. The storage element, which is programmable as either a D-type flip-flop or a level-sensitive latch, provides a means for synchronizing data to a clock signal, among other uses.

The storage elements in the upper and lower portions of the slice are called FFY and FFX, respectively.

Wide-function multiplexers effectively combine LUTs in order to permit more complex logic operations. Each slice has two of these multiplexers with F5MUX in the lower portion of the slice and FiMUX in the upper portion.

Block RAM All Spartan-3 devices support block RAM, which is organized as configurable, synchronous 18Kbit blocks. The no. of 18K blocks in a device varies from 4-104 depending the device chosen. Block RAM stores relatively large amounts of data more efficiently than the distributed RAM feature described earlier. The aspect ratio i.e., width vs. depth of each block RAM is configurable. Furthermore, multiple blocks can be cascaded to create still wider and/or deeper memories. A choice

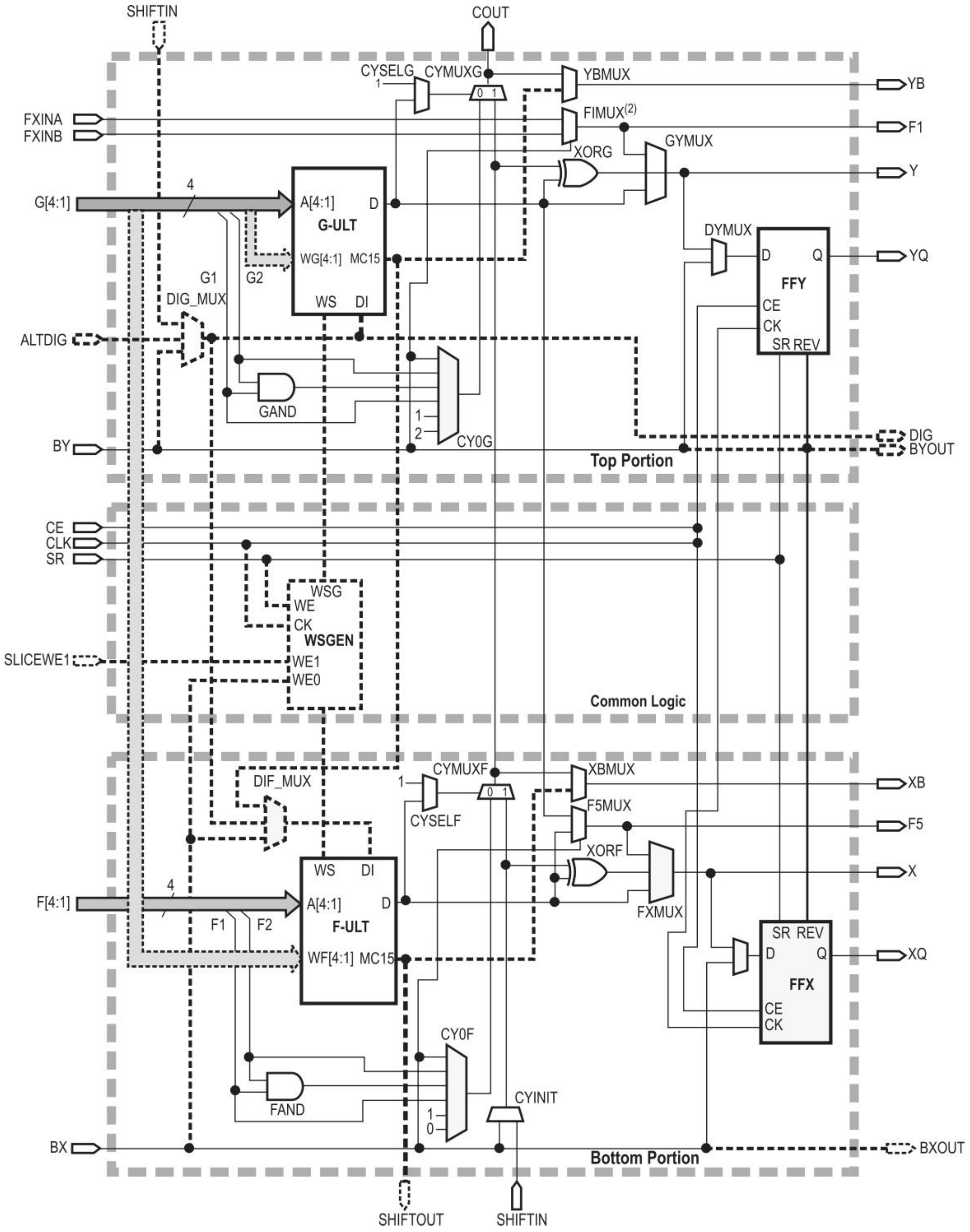


Fig.17.6 *Internal architecture of a left side slice (Courtesy of Xilinx, Inc.)*

among primitives determines whether the block RAM functions as dual- or single-port memory. The XC3S50 has a single column of block RAM embedded in the array. Those devices ranging from the XC3S200 to the XC3S2000 have two columns of block RAM. The XC3S4000 and XC3S5000 devices have four RAM columns.

Digital Clock Manager (DCM) Spartan-3 devices provide flexible, complete control over clock frequency, phase shift and skew through the use of the DCM feature. The DCM supports three major functions:

- Clock-skew Elimination: Clock skew describes the extent to which clock signals may, under normal circumstances, deviate from zero-phase alignment. It occurs when slight differences in path delays cause the clock signal to arrive at different points on the die at different times. This clock skew can increase set-up and hold time requirements as well as clock-to-out time, which may be undesirable in applications operating at a high frequency, when timing is critical. The DCM eliminates clock skew by aligning the output clock signal it generates with another version of the clock signal that is fed back. As a result, the two clock signals establish a zero-phase relationship. This effectively cancels out clock distribution delays that may lie in the signal path leading from the clock output of the DCM to its feedback input.
- Frequency Synthesis: When DCM is provided with an input clock signal, the DCM can generate a wide range of different output clock frequencies. This is accomplished by either multiplying and/or dividing the frequency of the input clock signal by any of several different factors.
- Phase Shifting: The DCM provides the ability to shift the phase of all its output clock signals with respect to its input clock signal.

Input/Output Blocks (IOBs) IOBs control the flow of data between the I/O pins and the internal logic of the device. A ring of IOBs surrounds a regular array of CLBs. Each IOB supports bidirectional data flow plus 3-state operation. Twenty-six different signal standards, including eight high-performance differential standards, are available. Double Data-Rate (DDR) registers are included. The Digitally Controlled Impedance (DCI) feature provides automatic on-chip terminations, simplifying board designs.

Multiplier Blocks accept two 18-bit binary numbers as inputs and calculate the product. Each 18K RAM block is associated with a dedicated multiplier.

Switching Networks The Spartan-3 family features a rich network of traces and switches that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing. Interconnect (or routing) passes signals among the various functional elements of Spartan-3 devices. There are four kinds of interconnect: Long lines, Hex lines, Double lines, and Direct lines. Long lines connect to one out of every six CLBs (see Fig.17.7). Because of their low capacitance, these lines are well-suited for carrying high-frequency signals with minimal loading effects (e.g. skew). If all eight Global Clock Inputs are already committed and there remain additional clock signals to be assigned, Long lines serve as a good alternative. Hex lines connect one out of every three CLBs (see Fig.17.8). These lines fall between Long lines and Double lines in terms of capability: Hex lines approach the high-frequency characteristics of Long lines at the same time, offering greater connectivity.

Double lines connect to every other CLB (see Fig.17.9). Compared to the types of lines already discussed, Double lines provide a higher degree of flexibility when making connections. Direct lines afford any CLB direct access to neighboring CLBs (see Fig.17.10). These lines are most often used to conduct a signal from a "source" CLB to a Double, Hex, or Long line and then from the longer interconnect back to a Direct line accessing a "destination" CLB.

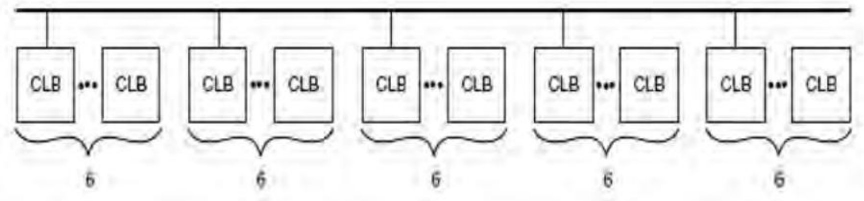


Fig.17.7 *Long lines in a spartan 3 FPGA (Courtesy of Xilinx, Inc.)*

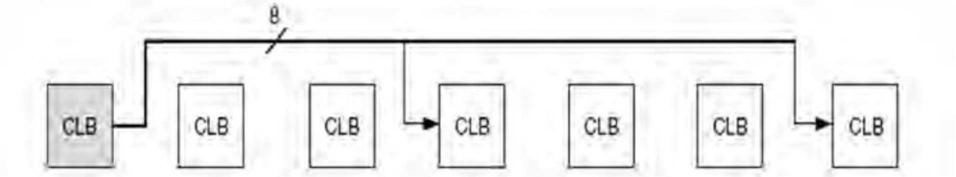


Fig.17.8 *Hex lines in a spartan 3 FPGA (Courtesy of Xilinx, Inc.)*

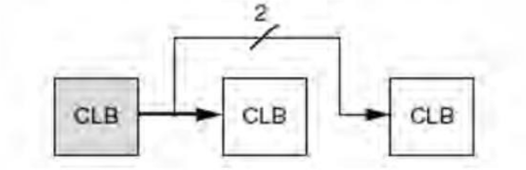


Fig.17.9 *Double lines in a spartan 3 FPGA (Courtesy of Xilinx, Inc.)*

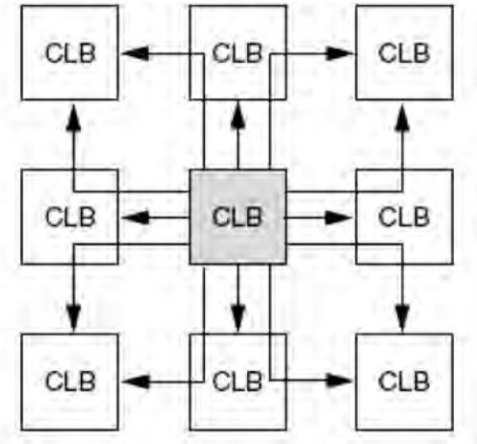


Fig.17.10 *Direct lines in a spartan 3 FPGA (Courtesy of Xilinx, Inc.)*

SRAM Configuration SRAM-based FPGAs are reconfigured by changing the contents of the configuration SRAM. A few pins on the chip are dedicated to configuration. Some additional pins may be used for configuration and later released for use as general-purpose I/O pins. As the FPGAs are reconfigured relatively infrequently, configuration lines are usually bit-serial. However, it is possible to send several bits in parallel if configuration time is important. During prototyping and debugging, we change the configuration frequently. A download cable can be used to download the configuration directly from a PC. When we move the design into production, we do not want to rely on a download cable and a PC. Specialized programmable read-only memories (PROMs) are typically used to store the configuration on the printed circuit board with the FPGA. Upon power-up, the FPGA runs through a protocol on its configuration pins. The EPROM has a small amount of additional logic to supply a clock signal and answer the FPGA's configuration protocol. Many modern FPGAs incorporate their reconfiguration **scan chains** into their testing circuitry. Manufacturing test circuitry is used to ensure that the chip was properly manufactured and that the board on which the chip is placed is properly manufactured. The **JTAG** standard (JTAG stands for Joint Test Action Group) was created to allow chips on boards to be more easily tested. Spartan-3 devices are configured by loading application specific configuration data into the internal configuration memory. Configuration is carried out using a subset of the device pins, some of which are "Dedicated" to one function only, while others, indicated by the term "Dual-Purpose",can be re-used as general-purpose User I/Os once configuration is complete. Depending on the system design, several configuration modes are supported, selectable via mode pins.

The chip can be configured in one of the following modes:

- Master serial mode assumes that the chip is the first chip in a chain (or the only chip). The master chip loads its configuration from an EPROM or a download cable.
- Slave serial mode gets its configuration from another slave serial mode chip or from the master serial mode chip in the chain.
- Master parallel and Slave parallel mode allow fast 8-bit-wide configuration.
- Boundary scan mode uses the standard JTAG pins.

17.4.3 An Overview of Cyclone III Family FPGAs

The Cyclone III family architecture consists of five fundamental programable functional elements:

Logic Elements and Logic Array Blocks The logic array block (LAB) consists of 16 logic elements(LEs) and a LAB-wide control block. An LE is the smallest unit of logic in the Cyclone III device family architecture. Each LE has four inputs, a four-input look-up table (LUT), a register, and output logic. The four-input LUT is a function generator that can implement any function with four variables.

Memory Blocks Each M9K memory block of the Cyclone III device family provides nine Kbits of on-chip memory capable of operating at up to 315 MHz for Cyclone III devices and up to 274 MHz for Cyclone III LS devices. The embedded memory structure consists of M9K memory blocks columns that can be configured as RAM, first-in first-out (FIFO) buffers, or ROM. The Cyclone III device family memory blocks are optimized for applications such as high throughout packet processing, embedded processor program, and embedded data storage.

Embedded Multipliers and Digital Signal Processing Support Cyclone III devices support up to 288 embedded multiplier blocks and Cyclone III LS devices support up to 396 embedded multiplier blocks. Each block supports one individual 18 × 18-bit multiplier or two individual 9 × 9-bit multipliers.

Clock Networks and PLLs Cyclone III device family includes 20 global clock networks. Global clock signals may be driven from dedicated clock pins, dual-purpose clock pins, user logic, and PLLs. Cyclone III device family includes up to four PLLs with five outputs per PLL to provide robust clock management and synthesis. PLLs may be used for device clock management, external system clock management, and I/O interfaces. The Cyclone III device family PLLs can be dynamically reconfigured to enable auto-calibration of external memory interfaces while the device is in operation. This feature enables the support of multiple input source frequencies and corresponding multiplication, division, and phase shift requirements. PLLs in Cyclone III device family may be cascaded to generate up to ten internal clocks and two external clocks on output pins from a single external clock source.

I/O Features Cyclone III device family has eight I/O banks. All I/O banks support single-ended and differential I/O standards. Cyclone III device family supports high-speed differential interfaces such as BLVDS, LVDS, mini-LVDS, RSDS, and PPDS. These high-speed I/O standards in Cyclone III device family provide high data throughput using a relatively small number of I/O pins and are ideal for low-cost applications.

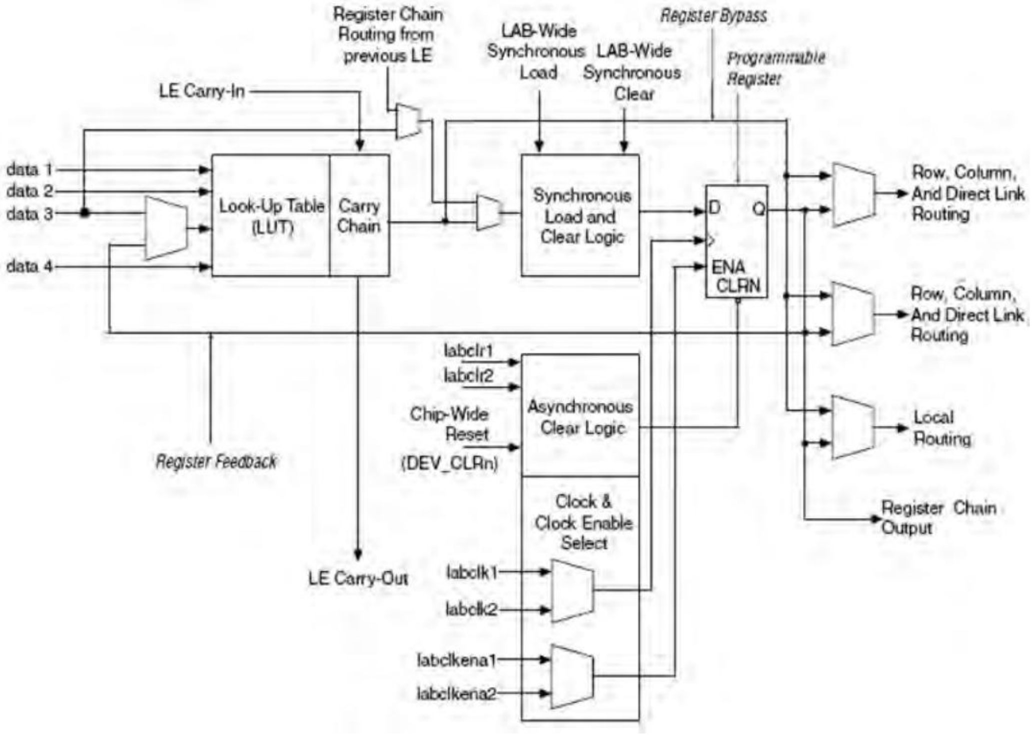

Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Local Routing
Register Feedback
Clock & Clock Enable Select
Register Chain Output
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2

Fig.17.11 *Internal architecture of Cyclone III Logic element (Courtesy of Altera Corporation.)*

Fig.17.11 gives the diagram of the internal architecture of Cyclone III Logic element. LEs are the smallest units of logic in Cyclone III family devices architecture. LEs are compact and provide advanced features with efficient logic usage. Each LE consists of a four-input look-up table (LUT), a programmable register, a carry chain connection and a register chain connection. LUTs can implement any function of four variables. LEs have the ability to drive the following interconnects: Local, Row, Column, Register chain and Direct link. The registers in LE can be either packed together or fedback. The registers can be programmed for D, T, JK, or SR flipflop operation. Each register has data, clock, clock enable, and clear inputs. Signals that use the global clock network, general-purpose I/O pins, or any internal logic can drive the clock and clear control signals of the register. Either general-purpose I/O pins or the internal logic can drive the clock enable. For combinational functions, the LUT output bypasses the register and drives directly to the LE outputs.

Each LE has three outputs that drive the local, row, and column routing resources. The LUT or register output independently drives these three outputs. Two LE outputs drive the column or row and direct link routing connections, while one LE drives the local interconnect resources. This allows the LUT to drive one output while the register drives another output. This feature, called register packing, improves device utilization because the device can use the register and the LUT for unrelated functions. The LAB-wide synchronous load control signal is not available when using register packing.

Cyclone III family devices LEs operate in either Normal mode or Arithmetic mode. In each of these modes, LE resources are used differently. In each mode, there are six available inputs to the LE. These inputs include the four data inputs from the LAB local interconnect, the LE carry-in from the previous LE carry-chain, and the register chain connection. Each input is directed to different destinations to implement the desired logic function. LAB-wide signals provide clock, asynchronous clear, synchronous clear, synchronous load, and clock enable control for the register. These LAB-wide signals are available in all LE modes.

Normal Mode Normal mode is suitable for general logic applications and combinational functions. In normal mode, four data inputs from the LAB local interconnect are inputs to a four-input LUT as shown in Fig. 17.12.

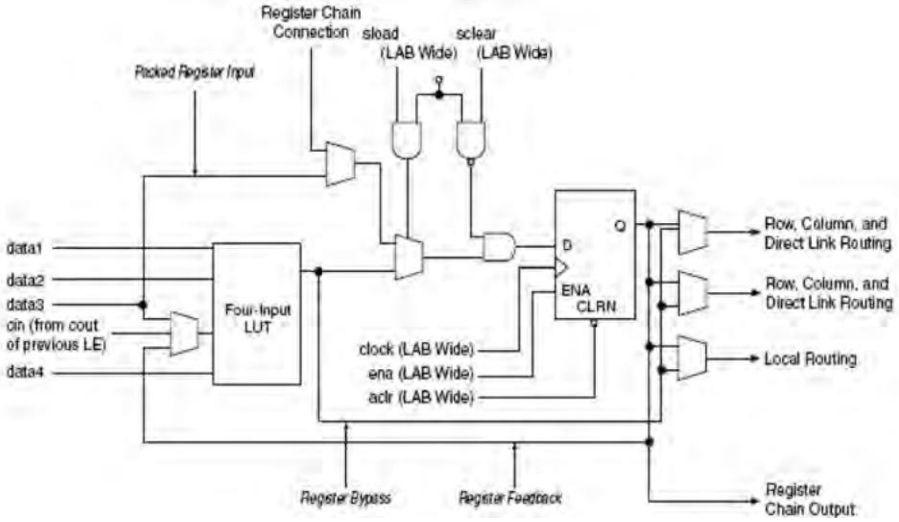


Fig. 17.12 *LE of cyclone III FPGA in normal mode (Courtesy of Altera Corporation.)*

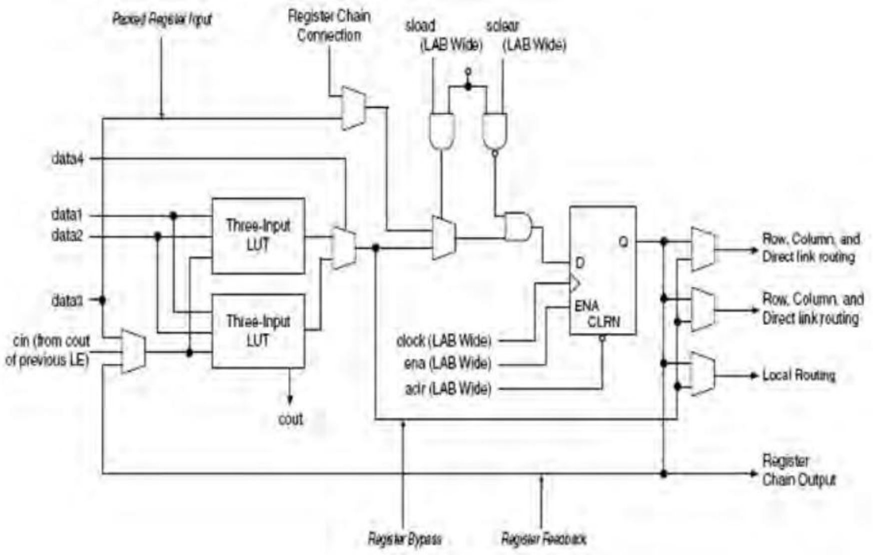


Fig. 17.13 *LE of cyclone III FPGA in arithmetic mode (Courtesy of Altera Corporation.)*

Arithmetic Mode

Arithmetic mode is ideal for implementing adders, counters, accumulators and comparators. Fig. 17.13 gives the diagram of LE of cyclone III FPGA in normal mode. In arithmetic mode, an LE implements a 2-bit full adder and basic carry chain. LEs in arithmetic mode can drive out registered and unregistered versions of the LUT output. Register feedback and register packing are supported when LEs are used in arithmetic mode.

Topology Each LAB consists of 16 LEs, LAB control signals, LE carry chains, Register chains and Local interconnect. The local interconnect transfers signals between LEs in the same LAB. Register chain connections transfer the output of one LE register to the adjacent LE register in an LAB. The Quartus II Compiler places associated logic in an LAB or adjacent LABs, allowing the use of local and register chain connections for performance and area efficiency.

LAB Interconnects

The LAB local interconnect is driven by column and row interconnects and LE outputs in the same LAB. Neighboring LABs, phase-locked loops (PLLs), M9K RAM blocks, and embedded multipliers from the left and right can also drive the local interconnect of a LAB through the direct link connection. The direct link connection feature minimizes the use of row and column interconnects, providing higher performance and flexibility. Each LE can drive up to 48 LEs through fast local and direct link interconnects.

Configuration Cyclone III device family uses SRAM cells to store configuration data. Configuration data is downloaded to Cyclone III device family each time the device powers up. Anyone of the five methods used to program the Stratix 3 FPGA can be used for the cyclone III FPGA

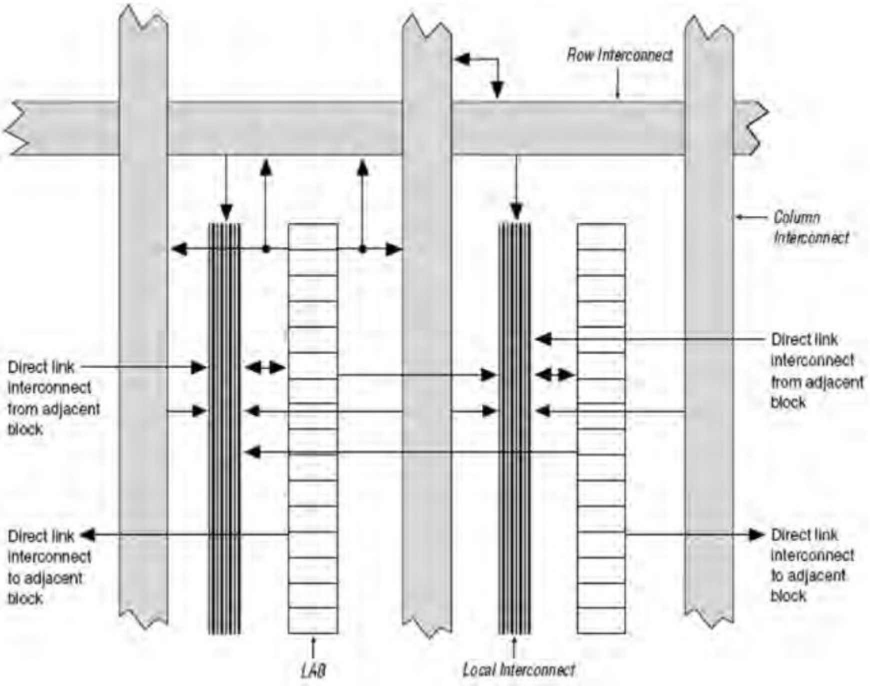

Row Interconnect
Column Interconnect
Direct link interconnect from adjacent block
Direct link interconnect from adjacent block
Direct link interconnect to adjacent block
Direct link interconnect to adjacent block
LAB
Local Interconnect

Fig.17.14 *Interconnect structure of a Logic array block (Courtesy of Altera Corporation.)*

17.4.4 Platform FPGAs

A platform FPGA has all the components necessary to build a complete system and should require few, if any, additional chips. Platform FPGAs include the basic five functional blocks of an advanced FPGA. (CLBs, block RAM, Embedded multiplier, Interconnection network and I/O blocks with different interfaces). It also includes CPUs, high-speed serial interfaces and bus interfaces.

Advantages of Platform FPGAs

Moving more functions onto a single chip generally provides several advantages such as Smaller physical size, higher speed, increase in system complexity, Lower power consumption and Higher reliability. Platform FPGAs provide the best of both worlds viz Microcontrollers and FPGAs. Microcontrollers have a no. of advantages such as availability of the off the shelf device drivers for a variety I/O devices, off the shelf software for different protocol stacks used for serial communication in disks and computer networks, support for implementation of different operating system kernels and support for programming in high level languages. CPUs can be easily programmed, can execute large and complex programs, and can take advantage of pipelining and other design optimizations. On the other hand, FPGAs provide faster computational speed: Techniques such as pipelining and parallel processing can be used to increase the speed as per user requirement. They can also be implemented more efficiently for low sampling rate systems. Their reconfigurability enables their speed and performance to be scaled as the system evolves. FPGA fabrics can handle a wide variety of data widths and can perform specialized operations on that data.

Due to their higher speed, FPGAs were originally proposed as coprocessors for advanced microprocessors and digital signal processors. Integration of both CPU and FPGA fabric in a single chip results in higher speed and lower power dissipation than the off the chip coprocessor approach.

Xilinx Virtex-II Pro family [Xil02] is an example of a platform FPGA. It has the following features:

- One or more IBM Power PC 405DC RISC CPUs.
- Multi-gigabit I/O circuitry.
- Embedded memory.
- An FPGA fabric consisting of CLBs, block RAM, embedded multipliers, DCMs , gigabit transceivers, and user I/O pins. The operation of these functional blocks are similar to that of Spartan III discussed in the previous section.

The largest Virtex-II Pro has four PowerPC CPUs, 125,136 logic cells, 10 Mbits of block RAM, 556 multipliers, 12 clock management blocks, 24 gigabit I/O transceivers, and 1200 user pins.

The gigabit transceiver units, known as Rocket I/O, operates in a range of 622 Mb/s to 3.125 Gb/s. It can be used to implement a variety of standard protocols, such as Fibre Channel, Ethernet, Infiniband, and Aurora. Each transceiver has a Physical Media layer which serializes and deserializes the data. It also has a Physical Coding Layer that contains CRC, elastic buffers, and 8-to-10 encoding/decoding.

PowerPC 405DC core has 32- bit Harvard architecture and operates at over 300 MHz on chip. It supports both 32-bit and 64-bit fixed-point arithmetic. The CPU has both instruction and data caches. The CPU can address a 4 GB address space. Its memory management unit (MMU) provides address translation, memory protection, and storage attribute control. The MMU supports demand-paged virtual memory. The CPU also has a complete interrupt system and a set of integrated timers. The IBM CoreConnect bus is used to connect the PowerPC to other parts of the FPGA.

DESIGN FLOW FOR AN FPGA BASED SYSTEM DESIGN 17.5

As discussed in Section 17.2.1, the content of the SRAM in the FPGA determines the function performed by the FPGA. FPGA has to be programmed so that the system uses the minimum resources (CLBs, F/Fs etc) and performs the required function satisfactorily meeting all the objectives specified. When the system to be implemented is complex, the computer aided design tools (CAD) are used for the design and optimization. Design of Very large Scale Integrated Circuits (VLSI) based system in general and FPGA based systems in particular is achieved using the following 6 steps with the help of CAD tools.

1. Design entry: Specification for the system to be designed is submitted to the CAD tool in this step. This may be in the form of logic diagram or schematic diagram for small systems. In the case of large systems, the design is specified using a hardware description language like VLSI hardware description language (VHDL) or Verilog.
2. Synthesis : The system represented by a schematic or description in HDL is translated to a network of components using parts available in the library. This in turn depends on the technique adopted for the implementation. In the case of FPGA based system design, the model, make and IC no. of the FPGA has to be specified. In the VLSI terminology the standard parts are called as library cells. Synthesis step also optimises the no. of library cells used for the translation.
3. Simulation : This uses the models for the building blocks in the implementation library. This step is used to check whether the logic diagram or the HDL description accurately represents the design requirement. For example, the design specification in terms of truth tables may be translated to logic circuits using traditional methods. This in turn may be expressed using

HDLs. To check the correctness of the translation, input test vectors are fed to the model for the system and the results obtained are compared with the expected results. For a complex system, exhaustic testing of the system for all possible combinations of inputs is impossible due to the large testing times required. An efficient simulator enables the testing with a smaller test vector set with less testing time.

4. Placement and Route (P&R) : A particular application may require only n out of N (n>N) CLBs in an FPGA for the implementation. Depending upon which CLBs are chosen, the performance of the system (propagation delay, maximum operating frequency, output drive capability, loading at the output etc.) may vary. The P&R step is concerned with choosing the correct combination of n CLBs and interconnecting them so that the system meets the required performance characteristics.
5. Post layout simulation and Timing analysis: This step checks whether the system implemented using the library of cells and interconnected in a particular fashion using the P&R step works as per specification. This step is essential as the interconnect delays may become available only after P&R step is completed. If required P&R step may be revisited to alter the performance characteristics.
6. Implementation: In the case of FPGAs, this step is used to translate the design using CLBs and their optimum interconnect pattern determined using steps 2, 4 & 5 above to bit patterns to be stored in the SRAM to program the CLBs and the interconnects appropriately. The resulting bitmap file may be transferred to the SRAM either directly through a download cable or by programming a serial PROM.

The above steps are also adopted for the design of application specific integrated circuits using standard library cells from IC fabricators. In this case, the implementation step generates a bit map file which is used for producing the masks for IC fabrication.

VLSI design has matured significantly and powerful VLSI CAD tools have been developed by a no. of companies like Cadence, Mentorgraphics, Avant, Synopsis and so on. Cost effective CAD tools targeted towards FPGA based system design have also become available.

CAD TOOLS FOR FPGA BASED SYSTEM DESIGN 17.6

The CAD tools developed for use with FPGAs provide a no. of features which enable the design and testing of a system based on FPGAs to be carried out in short period of the order of the few weeks to few months. Some of these features are:

- Availability of freeware: The FPGA vendors such as Altera and Xilinx offer two versions of some of their of CAD tools: One which is priced and another which is free. For example, The Quartus II software from Altera and Integrated Software Environment (ISE) from Xilinx has both the versions. The free version has the same look as the priced version but supports only designs of lower complexity. The procedure for design entry, simulation etc are the same for both the versions. This enables a beginner to learn the features of the tool at his own pace so that very little training is required when he/she is required to design a more complex system with the priced version.
- Lower cost and availability of CAD tools on PC platform: Most of the CAD tools for FPGAs are cheap and can be executed on Personal computers in both windows and linux operating systems. These features enable these tools to be learnt by a large number of designers in a short span of time.

- Availability of Intellectual proprietary cores and reuse of designs : Some of the system blocks such as central processing units (CPUs), input/ouput ports, universal synchronous asynchronous receiver transmitter (USART), FSK modulator/demodulator, microcontroller, convolutional encoder, viterbi decoder, FIR filter, DCT,FFT, DWT, speech and video codec are already designed and tested by the FPGA vendors. They are made available as a library component along with the CAD tools. They can be instantiated in the user program and can be synthesized along with the modules designed by the user. This reduces the time required for designing a system. Further, some of the earlier designs of a user may be reused in a future design.
- The CAD tools also support design entry using different mix of hardware description languages such as Verilog and VHDL. They also support the use of high level language such as C for the design of one part of the system and use HDL for another part of the system. This enables the harware/software partitioning of a system to be efficiently carried out.
- Support for high level DSP system design and Simulation: DSP Builder software from Altera facilitates the use of algorithmic DSP design in the MATLAB software and system integration in the Simulink software. The design can then be ported to hardware description language (HDL) files for use in the Quartus II design software. DSP Builder also produces HDL test bench files that can be used in ModelSim- Altera and other third-party HDL simulators. Similarly, the Xilinx System Generator for DSP is a plug-in to Simulink that enables designers to develop high-performance DSP systems for Xilinx FPGAs. Designers can design and simulate a system using MATLAB, Simulink, and Xilinx library of bit/cycle-true models. The tool will then automatically generate synthesizable Hardware Description Language (HDL) code mapped to Xilinx pre-optimized algorithms. This HDL design can then be synthesized for implementation in Virtex-II Pro Platform FPGAs and Spartan-IIE FPGAs. As a result, designers can define an abstract representation of a system-level design and easily transform this single source code into a gate-level representation. Additionally, it provides automatic generation of a HDL testbench, which enables design verification upon implementation.
- Support for development of system on a programmable chip: For the Altera FPGAs, SoPC Builder automates the task of adding, parameterizing and linking intellectual property cores, including multiple embedded processors, for system-on-a-programmable chip (SoPC) applications. The user defines the switch architecture component interconnect matrix to maximize system performance and then generates the system. SoPC Builder automatically assembles all the hardware components using the high-performance Avalon switch bus architecture. Moreover, a.h header file is automatically created that embodies a software view of the entire system with register and memory maps, as well as pre-defined software routines to control all the IP.

  For the Xilinx FPGAs, the Embedded Development Kit provides the support for the development of SOPC. It consists of four tools:

  Xilinx Platform Studio (XPS) Tool Suite which contains Graphical IDE and command line support for developing hardware platforms for embedded applications. The Base System Builder wizard enables creation of a working embedded system within minutes. XPS also includes other intelligent design wizards to quickly configure the embedded system architecture, buses and peripherals.

  Software Development Kit (SDK) for MicroBlaze and PowerPC which includes GNU C/C++ compiler and debugger; Xilinx Microprocessor Debug (XMD) target server; Data2MEM utility for bitstream loading and updating.

Real-Time Operating System and Embedded OS Support - Provides design support and board support package (BSP) generation for numerous third party suppliers in the Xilinx ecosystem, including vendors such as Wind River, Green Hills, Mentor, LynuxWorks and other embedded industry leaders.

Processing IP and MicroBlaze Soft Processor Core Pre-verified IP catalog, including a wide variety of processing peripheral cores for customizing the embedded systems as well as the flexible MicroBlaze 32-bit soft processing core. The MicroBlaze processor offers memory management and FPU configuration options enabling commercial grade RTOS support, unique for a soft processor.

- Support for Real-time on-chip verification:

  The following tools are available for the Altera FPGAs for this purpose.

  SignalTap® II Logic Analyzer: This embedded logic analyzer uses FPGA resources to sample tests nodes and outputs the information to the Quartus II software for display and analysis.

  SignalProbe: This tool incrementally routes internal signals to I/O pins while preserving results from the last place-and-routed design.

  Logic Analyzer Interface (LAI): This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows us to select which signals are switched onto the I/O pins over a JTAG connection.

  In-System Memory Content Editor: This tool displays and allows the users to edit on-chip memory.

  For the Xilinx FPGAs, ChipScope™ Pro tool provides the support for real-time on-chip verification. It inserts logic analyzer, bus analyzer, and virtual I/O low-profile software cores directly into the design. It allows us to view any internal signal or node, including embedded hard or soft processors. Signals are captured at or near operating system speed and brought out through the programming interface, freeing up pins for our design. Captured signals can then be analyzed through the included ChipScope Pro Logic Analyzer.

SOFTCORE PROCESSORS 17.7

The Power PC in a Virtex II PRO IC is called as hardcore processor as it is pre fabricated and its hardware architecture cannot be altered through programming. Instead of using a CPU fabricated and integrated to the FPGA fabric in a single chip, We may also implement a CPU into any FPGA by programming and configuring a part of the FPGA fabric. Such a processor is called as a softcore processor or IP CPU. Many Verilog or VHDL models for CPUs are available on the Internet. One advantage of using an IP CPU is that we can modify the CPU's architecture to add instructions or modify the cache organization. However, an IP CPU requires a lot of logic elements and we need a fairly large FPGA to have room left over for non-CPU logic or memory.The relative merits of softcore processors and hardcore processors are as follows:

Advantages

- Soft processors are far more flexible than hard processors. Soft processors can be enhanced with custom hardware to extend the instruction set with custom instructions and coprocessors targeted to the application using the processor. This can significantly enhance the performance of the soft processor in targeted applications.

- Design utilizing soft processors can be migrated to the latest FPGA fabric and onto any device in an FPGA family. This portability and design reuse extends the life cycle of soft-processor based designs.
- Soft processors can be deeply embedded into the system architecture for integrated control of the hardware architecture.
- Designs can be implemented by utilizing more than one processor to increase the overall system performance. While this is possible with hard processors, in software defined radio-based systems this can be changed from configuration to configuration of the modem.

Disadvantages

- Soft processors cannot reach the same clock rate as hard processors, generally resulting in lower performance than hard processors (unless custom instructions and coprocessors are used to enhance the soft processors).
- Soft processors do not have the power saving features available in many wireless-focused hard processors such as the ARM processor.
- Soft processors do not have the installed base of software available for mainstream processors such as the ARM processor. In general, a soft processor will not have a standardized wireless protocol stack available from a third party vendor.

Let us consider two examples of two popular soft core processors.

17.7.1 Softcore Processors from Xilinx

The Embedded Development Kit (EDK) from Xilinx, includes the soft processor core such as Microblaze and picoblaze and a standard set of peripherals. The kit includes a complete set of GNU-based software tools including the compiler, assembler, debugger, and linker. More details of MicroBlaze softcore processor(SCP) is considered next.

It is a standard 32-bit RISC Processor with Harvard architecture. Figure 17.15 shows a typical MicroBlaze SCP with its peripherals. The 32 by 32-bit registers are lookup table (LUT) RAM based. It guarantees a very short register access time. For memory, either the on-chip block RAM or off-chip memory can be used. The access time to the on-chip block RAM is minimal because there are dedicated routing resources to access them. The MicroBlaze SCP can be customized for any application. Its barrel shifter, divide unit, data cache, instruction cache, and the FSL bus system are optional. The sizes of the caches are configurable from 2 to 64 Kbytes. Standard peripherals are provided as well and are Core Connect compatible. Consequently, they can be integrated in an embedded design very easily. These peripherals are either free, such as the memory controller, UART, interrupt controller, and timer, or commercial cores such as the Ethernet controller, gigabit Ethernet controller, PCI, HDLC, etc.

Generally, there are two ways to integrate a customized IP core into a MicroBlaze-based embedded soft processor system. One way is to connect the IP on the On-chip Peripheral Bus (OPB). The OPB is part of the IBM Core ConnectTM on-chip bus standard. The second way is to connect the user IP to the MicroBlaze dedicated Fast Simplex Link (FSL) bus system. If the application is time-critical, the user IP should be connected to the FSL bus system; otherwise, it can be connected as a slave or master on the OPB. If the customized core is connected to the dedicated FSL interface, it is then possible to use predefined C functions to use the user core in the application software.

MicroBlaze contains eight input and eight output FSL interfaces. The FSL channels are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces on MicroBlaze are 32 bits wide. Further, the same FSL channels can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information. The performance of the FSL interface can reach up to 300 MB/sec. This throughput depends on the target device itself. The FSL bus system is ideal for MicroBlaze-to-MicroBlaze or streaming I/O communications. The FSL bus is driven by one Master and drives one Slave. Fig.17.16 shows the principle of the FSL bus system and the available signals.

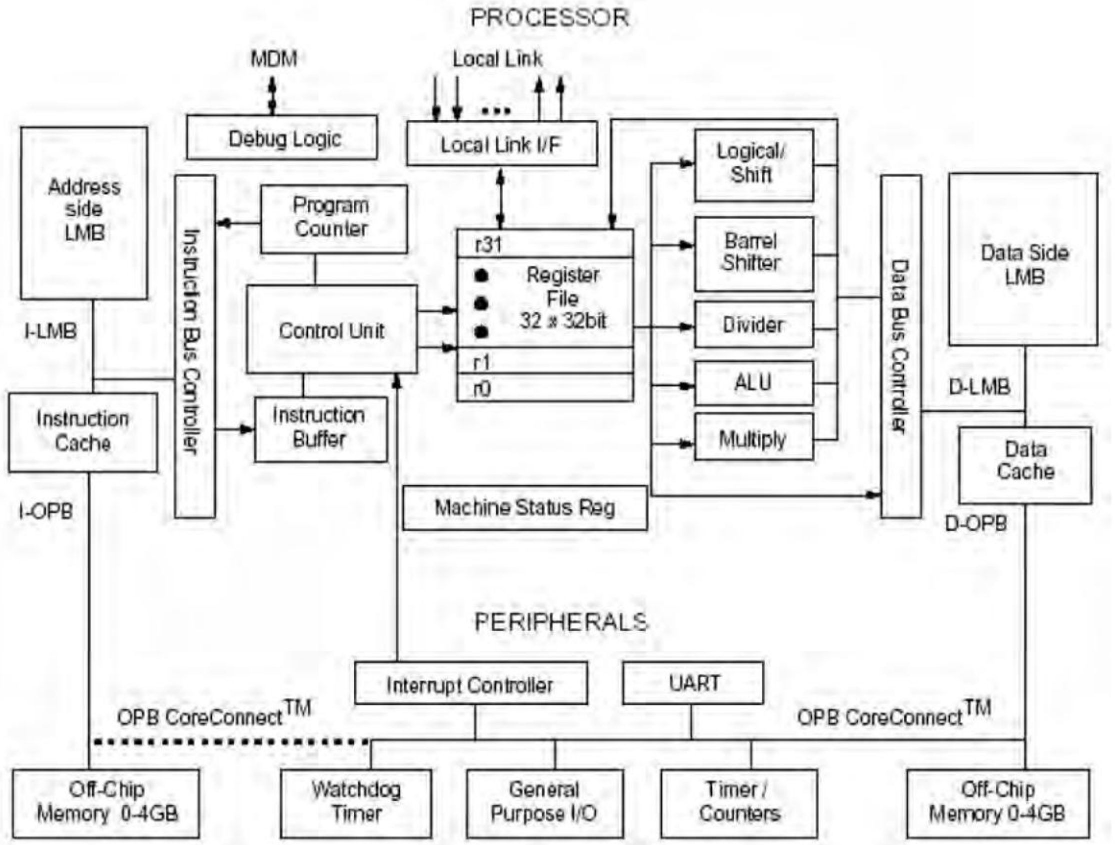


Fig.17.15 *Architecture of Microblaze processor (Courtesy of Xilinx, Inc.)*

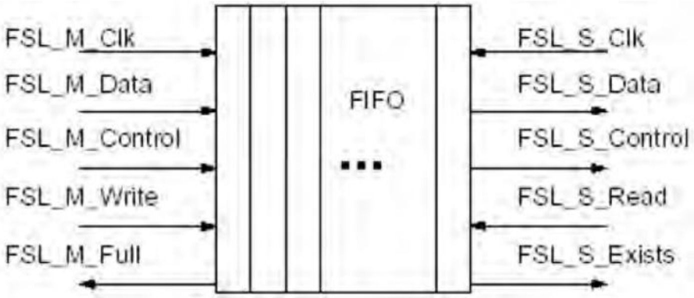


Fig.17.16 *FSL system bus (Courtesy of Xilinx, Inc.)*

FSL peripherals may be created as a Master or a Slave to the FSL bus. A peripheral connected to the master ports of the FSL bus pushes data and control signals onto the FSL. All peripherals that act as a master to the FSL bus should create a bus interface of the type MASTER for the bus standard FSL in the Microprocessor Peripheral Description (MPD) file. A peripheral connected to the slave ports of the FSL bus reads and pops data and control signals from the FSL. All peripherals that are a slave to the FSL bus should create a bus interface of the type SLAVE for the bus standard FSL in the MPD file. The put and get instructions of MicroBlaze can be used to transfer the contents of a MicroBlaze register onto the FSL bus and vice-versa. The FSL bus configuration of MicroBlaze can be used in conjunction with any of the other bus configurations.

Application As an application to demonstrate the use of the FSL interface, a 1-dimension IDCT is used. [Hans04] This DSP application highlights very well the performance win that could be reached. A 1-dimension IDCT realized in software would require a high execution time because the C- program would consist mainly of loops which get executed sequentially by the processor. If the application is implemented as its own hardware module, the execution time requires much fewer clock cycles. The used 1-IDCT core on the FSL interface is an example and needs approximately 150 LUTs and the latency of 64 clock cycles. It may be noted that this IDCT core is used to show how to implement a user core on the FSL interface. The software application writes 8 values from memory to the FSL. The IDCT core gets the data and calculates the result. When the result is available, MicroBlaze reads the data (8 words) back from the FSL. The IDCT core is connected to the FSL interface as shown in Fig. 17.17. For the FSL0 connection, the MicroBlaze is the Master on the FSL bus and the IDCT core is the Slave. Thus, MicroBlaze controls the data sent on the FSL0 bus to the IDCT core. For the FSL1 bus, it is vice versa, and the IDCT core is the Master and the MicroBlaze the Slave. The IDCT controls the data on the FSL1 bus.

Predefined C functions are provided in EDK for integrating the customized user IP in the C/C++ application program.

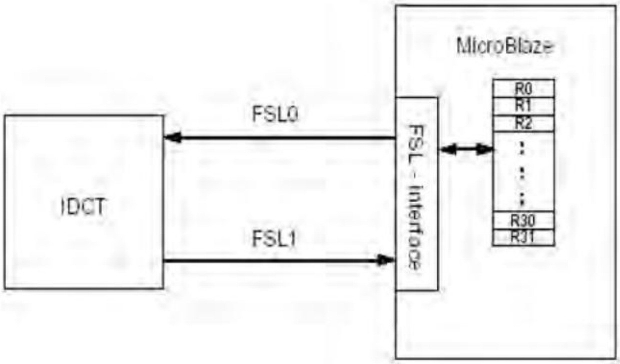


Fig 17.17 *On-dimensional IDCT IP via the FSL interface onto MicroBlaze (Courtesy of Xilinx, Inc.)*

17.7.2 Nios II Processor System

A Nios II processor system includes a processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera ® chip. To facilitate the design of embedded systems using Nios processor, Altera offers System on chip (SOC) kits with different Altera® devices such as APEX, Cyclone and Stratix device which are pre-loaded with a 32-bit Nios/Nios II softcore processor system. The Nios II processor has the following features:

32-bit instruction set, data path, and address space

32 general-purpose registers

32 external interrupt sources

Single-instruction 32 ×32 multiply and divide producing a 32-bit result

Dedicated instructions for computing 64-bit and 128-bit products of multiplication

Single-instruction barrel shifter

Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals

Hardware-assisted debug module enabling processor start, stop, step and trace under integrated development environment (IDE) control

Software development environment based on the GNU C/C++ tool chain and Eclipse IDE

Instruction set architecture (ISA) compatible across all Nios II processor systems

Performance beyond 150 DMIPS

Figure 17.18 shows an example of a Nios II processor reference design available in an Altera Nios II development kit.

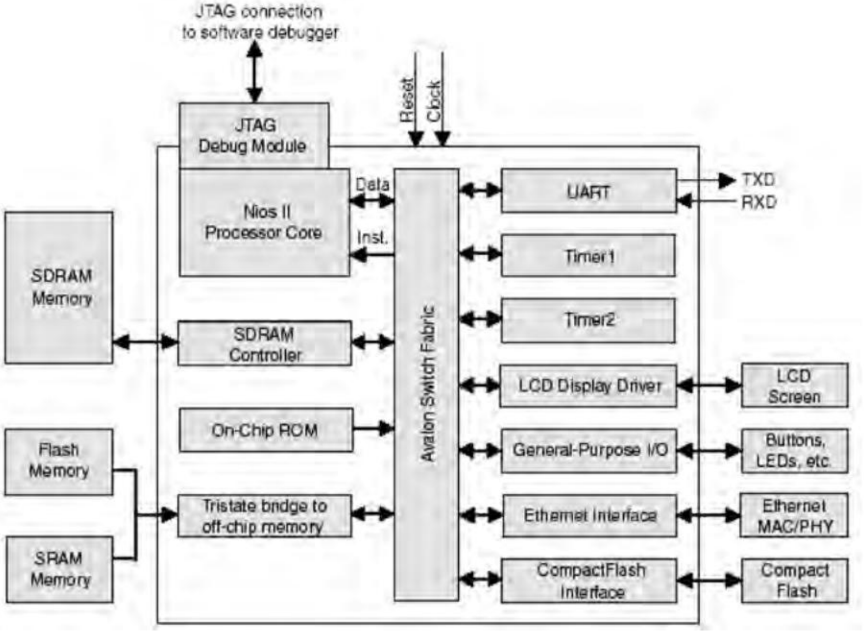


Fig. 17.18 *Nios II processor reference design (Courtesy of Altera Corporation.)*

Custom Peripherals Designers can also create their own custom peripherals and integrate them into Nios II processor systems. For performance-critical systems that spend most CPU cycles executing a specific section of code, it is a common technique to create a custom peripheral that implements the same function in hardware. This approach offers a double performance benefit: the hardware implementation is faster than software; and the processor is free to perform other functions in parallel while the custom peripheral operates on data.

17.7.3 Custom Instructions

Like custom peripherals, custom instructions are a method to increase system performance by augmenting the processor with custom hardware. The soft-core nature of the Nios II processor enables designers to integrate custom logic into the arithmetic logic unit (ALU). Similar to native Nios II instructions, custom instruction logic can take values from up to two source registers and optionally write back a result to a destination register. The interfacing details of the custom logic with the Nios ALU is shown in Fig.17.19. The performance of the Nios processor is enhanced by integrating these blocks with the Nios core using the SOPC builder system development tool. The program to be executed by the Nios core is written in C/C++ and the custom instructions executed by the custom block are defined as software macro and invoked in C/C++ program.

Custom instructions consist of two essential elements, a *Custom logic block,* the hardware that performs the user-defined operation, and a *Software macro, which* allows the system designer to access the custom logic through software code.

The task performed by the custom block may be defined either as single-cycle combinatorial operation or as multi-cycle sequential operation. In both cases, two 32-bit operands may be passed to custom block and a 32-bit result is returned. The CPU clock is made available to the custom block only when it is defined to be sequential.

A 11 bit prefix code may be sent to the custom block along with the other operand(s) through the prefix port. This may be used to specify the type of operation to be performed in the custom block.

To study the efficacy of the custom instruction, 2D DWT of a subimage of size 32×32 is computed using both custom logic and the in-built instructions in section 17.6.2. The computation using custom instruction is found to be faster by a factor of 90.

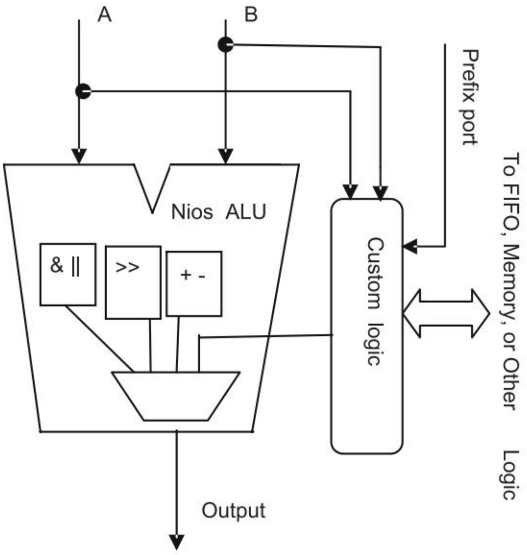


Fig.17.19 *Adding Custom Logic to the Nios ALU*

17.7.4 Hardware/Software Partitioning

In an FPGA based system consisting of both CPU and non CPU logic, it is important to decide about the parts of the application which go into the CPU as software and what parts should go into the non CPU logic of FPGA fabric. This problem is known as **hardware/software partitioning.** As shown in Fig.17.19, we are trying to fit the application into a pre-existing architecture consisting of a CPU and FPGA fabric connected by a bus. We refer to the logic on the FPGA side as an **accelerator;** in contrast, a co-processor would be dispatched by the execution unit of the CPU.

We must partition the application into two pieces, the CPU side of the bus and the FPGA side of the bus. There are many different ways to do this and a careful analysis is required to find a partitioning that results in a higher-performance system.

FPGA BASED DSP SYSTEM DESIGN 17.8

An FIR or IIR filter is built using multipliers, adders and delay elements (shift registers). The efficient implementation of these elements in VLSI circuits in general and FPGAs in particular have been discussed in detail in a no. of works (see for. Eg. Pirsch [1998], Smith[1999], Swartzlander[1987], Weste[1999]). For example, for implementing the multiplier, a no. of algorithms such as shift and add algorithm, serial/parallel algorithm, array multiplication, pipelined array multiplication, Wallace Tree algorithm etc. have been proposed. A simple example of a DSP system using FPGAs is discussed in Section 17.7.1. In addition to direct implementation of these filters, alternate realizations of the DSP system which either operate at higher speed or which require less hardware have been proposed in the recent past. A brief description of some of these approaches are given in Section 17.8

17.8.1 Implementation of Serial/Parallel Convolver using FPGAs

The block diagram of a serial/parallel convolver is given in Fig. 1.5 in Chapter 1. For the sake of simplicity, let us assume that the length of the impulse response sequence M is 4. Fig.17.20 shows the diagram of the convolver for M=4. Let both the input samples and impulse response coefficients be represented as unsigned 4 bit numbers. From Fig. 17.20, it may be noted that four 4×4 multipliers, three 8 bit adders and three 8 bit shift registers are required. This may be verified as follows: When two 4 bit numbers are multiplied, the result is 8 bits long. The output of adder Σ_3 is 9 bit long. The LSB is discarded and the remaining bits are fed to the Shift register. Similarly at the output of the adders Σ_2 and Σ_1, LSBs are discarded.

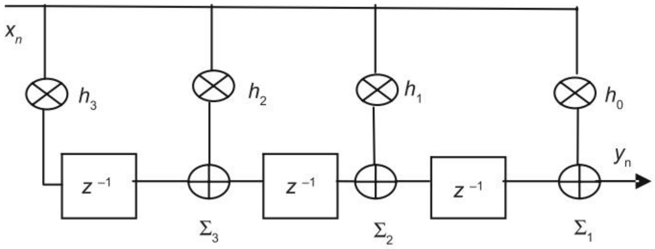


Fig. 17.20 *Serial/parallel convolver for M=4*

Next, consider the implementation of 8 bit adders. For implementing the adders, there are several algorithms such as carry save, carry propagation, ripple carry, serial addition and so on. For example, the ripple carry adder for two 4 bit numbers may be obtained by cascading 4 full adders as shown in Fig. 17.21. Each full adder may be implemented using two 4 input LUTs as shown in Fig.17.22. In Fig.17.22, a0 and b0 denote two bits and Cin denotes the carry in. S0, C0 denote the sum and carry outputs. It may be noted that in Fig.17.22, four input look up tables are used for generating the sum and carry outputs eventhough only three inputs are normally fed to a F/A. The fourth output is chosen arbitrarily as 0 as the CLBs of XC4000E family ICs given in Fig.17.3 has only 4 input LUTs. In Fig.17.20, the output of the adders are stored in a shift registers (which correspond to the delay elements). From Fig.17.3, it may be observed that the output of the LUTs may be taken out of the CLB using two unlatched output lines X, Y and two latched

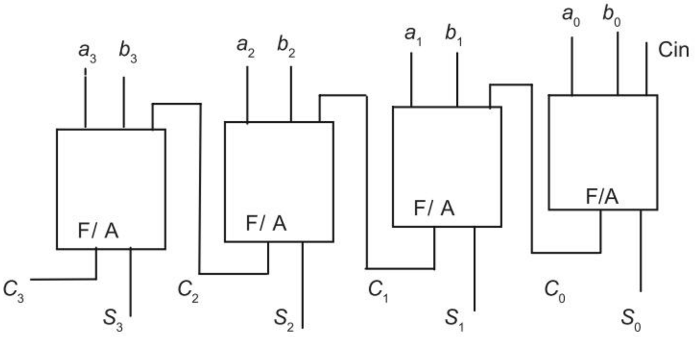


Fig. 17.21 *Four bit ripple carry adder using Full adders*

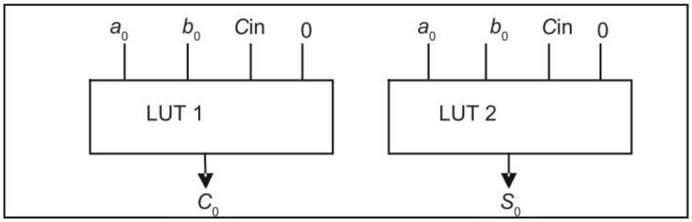


Fig. 17.22 *Full adder implemented using two LUTs*

output lines XQ and YQ. The F/Fs in the CLB themselves can be used for obtaining the shift register. The clock frequency to these F/Fs should be made equal to the sampling rate of the input signal. To implement an 8 bit adder and 8 bit shift register, 8 CLBs are required. To simplify the treatment separate LUTs are used for generating the carry and sum output. However XC4000E ICs have fast carry logic and using this a single LUT can be used to generate both Carry and sum outputs. In this case only 4 CLBs are required for implementing the 8 bit adder and shift register.

Next, consider the implementation of the 4 bit multipliers in FPGAs. For simplicity, consider the serial/parallel multiplier scheme. This scheme is similar to the serial/parallel convolver scheme given in Fig,17.20. For the 4 bit multiplier, the multipliers, adders and registers shown in Fig 17.20 should be replaced by single bit elements. For the adder, the carry output should be fed back to the input after a delay of 1 bit. It may be noted that the single bit multiplier is equivalent to the AND gate. The single bit adder is F/A. As noted earlier, only 3 input LUT is required for F/A. The 4 input LUT can be used to implement both the AND and full adder functions in a single LUT. The resulting diagram of the 4 bit serial/parallel multiplier is shown in Fig.17.23. In Fig.17.23, D denotes the delay elements corresponding to 1 bit time and is implemented using the F/Fs in the CLBs. The output of the LUTs are fed to these F/Fs. Σi , Ci for i = 0-3 denotes the sum and carry outputs of the full adder cum single bit multipliers. b0-b3 and a0-a3 correspond to the 4 bits of the two numbers to be multiplied. a0-a3 are fed serially. These bits have to appended by 4 zeros to ensure carry propagation. This multiplier requires 8 clock cycles for multiplication. The serial shift clock is also used to latch the new carry and sum outputs in the CLBs. It can be observed from Fig.17.23 that 4 CLBs are required for 4 bit multiplication. Similarly, an N bit multiplication requires N CLBs and 2N clock cycles for multiplication.

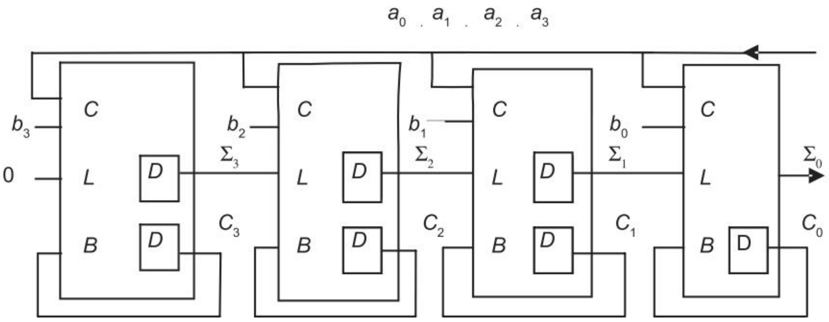


Fig. 17.23 *Implementation of serial/parallel 4 bit multiplier using CLBs*

Using the ripple carry adder and serial parallel multiplier, it may be concluded that the serial/parallel convolver for M=4 and for samples and impulse response coefficients represented using 4 bit numbers, the no. of CLBs required is 28. Out of these, 16 CLBs are required for the 4 multipliers and 12 CLBs are required for the three 8 bit shift registers and adders. Similarly, for a serial/parallel convolver for M=8 and for samples and impulse response coefficients represented using 8 bit numbers, the no. of CLBs required is 120.

17.8.2 Implementation of Convolver using Constant Coefficient Multiplier

When the impulse response coefficients h_0-h_3 in the above convolver are time invariant (constants), the constant coefficient multiplier (KCM) can be used to realize the fastest multiplier and the fastest convolver. KCM uses a ROM for finding the product of a constant and a variable. The variable is fed as address to the ROM which contains the products corresponding to all possible combinations of the operands. For example, if a 4 bit constant is to be multiplied with a 4 bit variable, there are 16 possible products each product having 8 bit. Eight 4 input LUTs may be used to realize this multiplier. Each LUT can be treated as a 16X1 RAM. The F inputs F3-F0 may be treated as the address inputs.The MSB of the address lines to all the LUTs (F3's of all LUTs) are tied together and the MSB of the variable is applied to it. Similarly, LSB of the address lines of all LUTs (F0's of all LUTs) are tied together and the LSB of the variable is applied to it. Similarly, the other address lines of LUTs are connected. Among the 8 LUTs, one of them stores the MSB of the products, next one stores the next bit of the product and so on. The speed of the multiplier is determined by the access time of the 16X1 RAM which is very much faster than actual multiplication time.

When the ROM is implemented using 4 input LUTs, a number of stages of LUTs and adders are required to find the product. The circuit diagram of a 12x12 bit KCM is shown in Fig.17.24. This requires one ROM stage and two stages of addition. The speed of the KCM can be increased further by introducing the pipelining registers at the points denoted in dotted lines in Figure17.24 and this scheme is denoted as pipelined KCM.

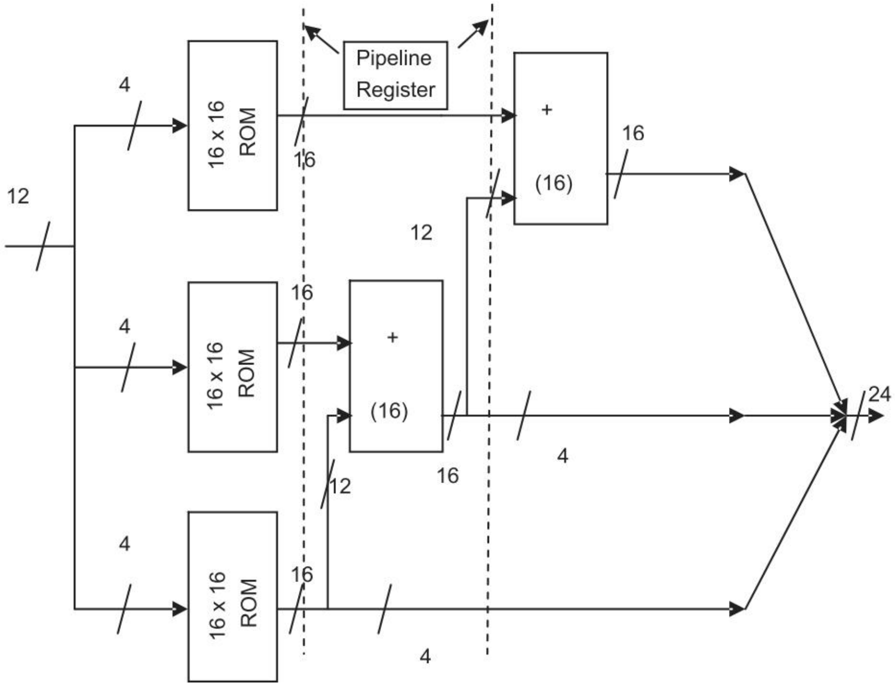


Fig. 17.24 *Block diagram of pipelined 12X12 KCM*

The following observations may be made with regard to the above examples:

- The number of 4 input LUTs/CLB depends on the FPGA family. In the XC4000 family there are two 4 input LUTs/CLB wheras in Spartan 3 family, there are four 4 input LUTs/CLB.
- The arguments given in this section, can be used to find the no. of CLBs required for any convolver using serial/parallel scheme for both convolution and multiplication of the binary numbers.

- The no. of CLBs required would change if the algorithm used for addition and multiplication are different from the ones considered here. However, the no. of CLBs required can be found for anyother mix of adder and multiplier algorithms using the same arguments.
- Eventhough in this example, the multiplier and adders schemes are arrived at manually, in practice, the algorithm for the adder and the multiplier would be specified to the CAD tools using logic and algebraic equations. The CAD tools would translate these equations to actual hardware circuit using CLBs and the interconnects.
- Eventhough, the function performed by each LUT or CLB may be manually forced or programmed, in practice the CAD tool would automatically do this exercise depending upon the requirement.

Taking recourse to CAD tools would result in better optimisation in terms of the no. of CLBs used and propagation delays especially for complex designs.

NEW ALGORITHMS FOR IMPLEMENTATION OF FILTERS IN VLSI 17.9

A number of architectures have been proposed in the literature for the efficient implementation of filters (IIR and FIR), filter banks and computation of transforms such as FFT, DCT, and DWT in both ASICs and FPGAs. [Parhi]. Techniques for either increasing the speed or reducing the power dissipation of systems using pipelining and parallel processing have been proposed in [Parhi]. FIR filters using delta sigma modulators have been proposed in the literature. In this scheme, an FIR filter with N taps and input samples of M bit each is implemented using another FIR filter with input samples with only m bits (m < M). This is achieved by using a prediction filter to predict the input values. Only the prediction error (represented using m bits) is fed as input to the FIR filter. To decrease the prediction error, the input is oversampled. Additional details of this scheme are given in Data source [2000].

In addition to the above, several other schemes such as Interpolated FIR filters, Filters using polynomial transforms etc have been proposed in the literature for efficient implementation of filters in VLSI circuits.

A number of algorithms such as fast convolution algorithms, fast filter algorithms and distributed arithmetic algorithm have been proposed to realize faster systems including filters and transform blocks. For brevity, the distributed arithmetic algorithm alone is presented in more detail in the next section.

DISTRIBUTED ARITHMETIC ALGORITHM 17.10

Distributed Arithmetic (DA) plays an important role in embedding DSP functions in the LUT based FPGAs and enables the FPGAs to achieve performance which is superior to those of programmable DSPs. DA technique is applicable for both Xilinx FPGAs and Altera Flex devices. Distributed Arithmetic is used to perform multiplication with look-up tables. The sum of products also referred to as the vector dot product is required to be computed in a number of applications such as digital filters and computation of fast fourier transform as well as discrete cosine transform. The DA can be optimized for area efficiency, speed efficiency or for both. For efficient implementation of DA on FPGAs, a no. of algorithms such as ROM decomposition technique and offset binary coding (Stanley[1989]) have been proposed in the literature. In this section a brief introduction to DA and its enhancements for FPGAs are considered.

17.10.1 An Overview of the Distributed Arithmetic Algorithm

The output sequence $y(n)$ of a linear, time invariant discrete time system can be expressed as

$$y(n) = \sum_{k=0}^{N-1} x(n-k)h(k) \tag{17.1}$$

where x(n-k) is the sample of the input at the (n-k)th sampling instant and h(k) for k = 0, 1, … N-1 are the samples of the impulse response. The transform coefficients X(n) of DFT and DCT may also be expressed in terms of the samples of the inputs x(n) as :

$$X(n) = \sum_{k=0}^{N-1} h(n,k)x(k) \tag{17.2}$$

where h(n,k) denotes the (n,k)th element of the transform matrix. Both eqn. (17.1) and (17.2) involves the computation of the vector dot product of h and x and can be generalized as the problem of computation of the sum of products given by

$$y(n) = \sum_{k=0}^{N-1} a(n,k)h(k) \tag{17.3}$$

In the case of LTI filters and transform computation, a(n,k) is time invariant and only x(k) varies with time. Computation of y(n) using (17.3) requires N multiplications to be performed every time a new input comes. For large values of N, it would either require a large area for implementation or restrict the maximum sampling rate that can be used. In the direct implementation scheme for the system given by eqn. (17.3), the minimum sampling period Tmin is limited to Tm + (N-1)Ta where Tm, Ta denote the computation time of a multiplier, adder respectively. With the transpose structure, Tmin can be increased to Tm+Ta. But, for large values N, driving N-1 registers with a single input sample synchronously becomes tricky. However, the fact that a(n,k)'s are constants can be used to compute eqn. (17.3) by using the look up tables for multiplication. This can be achieved as follows:

The input samples x(k) may be assumed to be represented in 2's complement representation using W bits where the MSB x(W-1,k) is the sign bit and bits x(W-2,k) to x(0,k) represent the fractional part of the number. x(k) can be written as

$$x(k) = -x(W-1,k) + \sum_{m=1}^{W-1} x(W-1-m,k)2^{-m} \tag{17.4}$$

Substituting eqn (17.4) in (17.3) and interchanging the order of summation w.r.t m and k we get

$$y(n) = -\sum_{k=0}^{N-1} x(W-1,k)a(n,k) + \sum_{m=1}^{W-1}\left[\sum_{k=0}^{N-1} x(W-1-m,k)a(n,k)\right]2^{-m} \tag{17.5}$$

It may be noted that x(m,k) for m= 0,1, … W-1 takes binary values 1 or 0. Hence, the terms inside the square brackets of (17.5) can be computed using ROM with address as the bits x(m,0), x(m,1), … x(m,N-1). For N = 4, the content of the ROM is as shown in Table 17.2. Each location is of width W + log (N) assuming a(n,k) to be represented using W bits. For N=W=4, in each location a 6 bit number is to be stored.

Table 17.2 *Content of DALUT ROM for N=4*

X(m,3)	*x(m,2)*	*x(m,1)*	*x(m,0)*	*Content of ROM*
0	0	0	0	0
0	0	0	1	a(n,0)
0	0	1	0	a(n,1)
0	0	1	1	a(n,1)+ a(n,0)
0	1	0	0	a(n,2)
1	0	0	1	a(n,2)+ a(n,0)
1	0	1	0	a(n,2)+ a(n,1)
…	…	…	…	….
1	1	1	0	a(n,3)+a(n,2)+ a(n,1)
1	1	1	1	a(n,3)+a(n,2)+ a(n,1)+ a(n,0)

17.10.2 Fully Parallel DA Algorithm

To compute y(n) , W ROMs whose contents are identical may be used. The LSBs of all the samples are fed as the address to the 0th ROM. The next bit of all the samples are fed to the 1st ROM address bits. Similarly, the MSB of all the samples are fed as address to the (W-1)th ROM. The contents of the ROMs W-2 to 0 are to be added after shifting their content suitably and the result is to be subtracted with the content of ROM W-1. For W= N =4, the manner in which the ROM contents are to be added is shown in Fig. 17.25. For computing the vector dot product, the scheme shown in Fig.17.25 requires a computation time of one LUT delay and two adder delays. In general, an N tap filter requires a computation time of

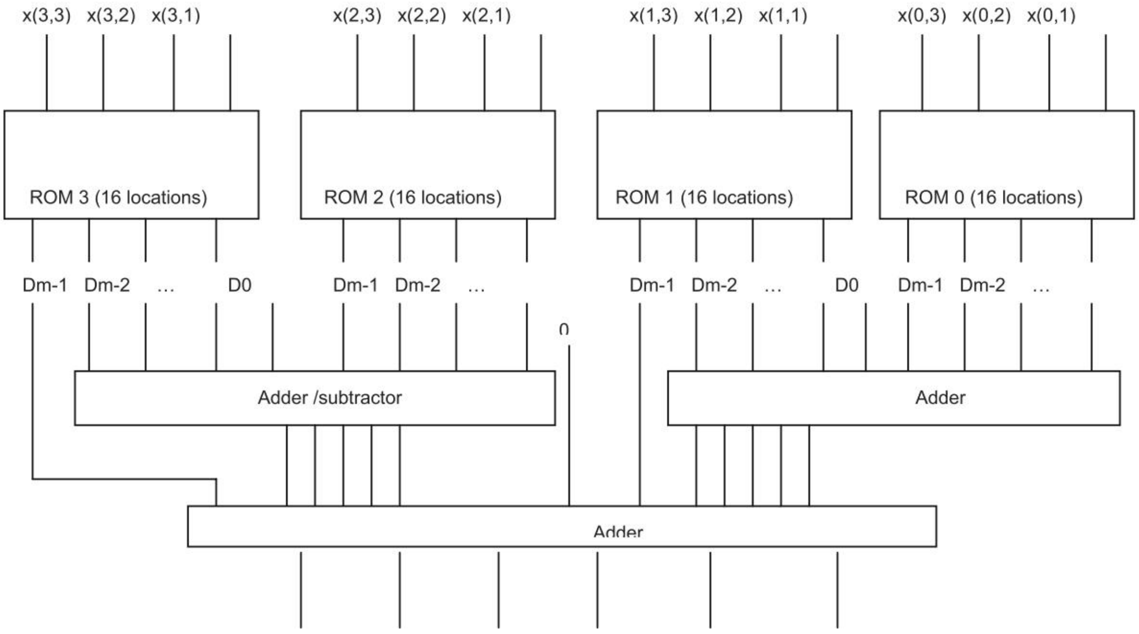


Fig. 17.25 *Fully parallel DA algorithm with W ROMs and W-1 adders*

1 LUT delay and logN adder delays. Parallel addition of the distributed arithmetic LUT (DALUT) contents results in the maximum speed.

17.10.3 Serial DA Algorithm

A single adder and a ROM may also be used for the computation of the dot product. In this case, the samples have to be stored in N shift registers and their contents are to be applied to the ROM one bit after another in parallel from the N registers. The products are to be accumulated and stored in a register. The resulting scheme is shown in Fig. 17.26. This requires W clock cycles for computation of the dot product. The clock period is equal to 1 DALUT delay + 1 adder delay. Note that x_{jk} denotes the k^{th} bit of the j^{th} sample of the input(x_j).

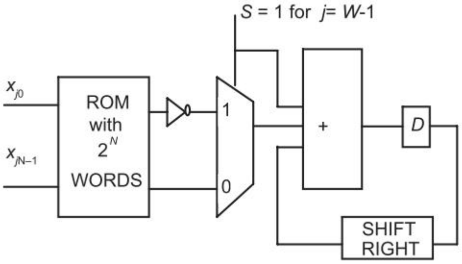


Fig. 17.26 *Serial implementation of Distributed Arithmetic algorithm*

17.10.4 The Speed and Area Tradeoff

By using multiple DALUTs and adders, the speeds in between the above two cases can be obtained at the cost of additional gate count.

17.10.5 ROM Decomposition Technique for DA Algorithm

DA algorithm discussed above can be modified to reduce the size of the ROM required. It can be verified that an N tap filter requires DALUTs with 2^N locations. The exponential growth in the ROM size can be avoided by splitting the N address bits to the ROM into blocks of K address bits each. Now, only K input DALUTs are required and hence the individual ROM size becomes 2^K. Totally N/K such DALUTs are required for computing the output corresponding a particular bit of the input samples. To get the correct output, the outputs of the K input DALUTs have to be added. The serial DA algorithm using the ROM decomposition scheme is shown in Fig. 17.27.

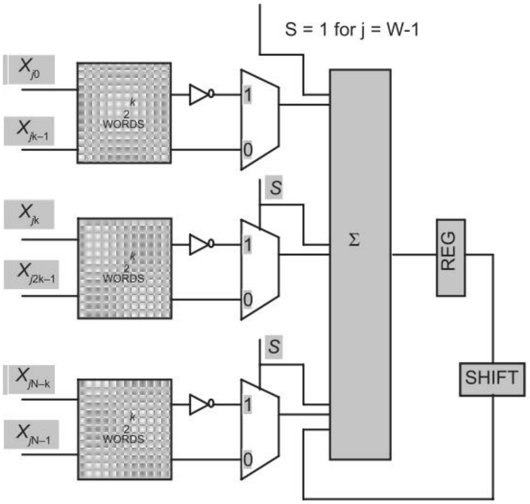


Fig. 17.27 *Distributed Arithmetic using ROM decomposition*

Application to FPGAs The ROM decomposition technique can be gainfully used for not only reducing the ROM size but also for ease of implementation on FPGAs. Xilinx FPGAs such as XC4000, Spartan and Virtex family FPGAs and Altera FPGAs such as Flex, Cyclone and Stratix family devices have 4 input LUTs. Hence, the value of K can be chosen to be 4 for efficient implementation on these FPGAs.

17.10.6 Implementation of Symmetric FIR Filters using DA Algorithm

The symmetry in the impulse response of such filters may be exploited to use DALUTs of size $2^{N/2}$. In this case the input to the DALUT should be appropriately added to generate the correct address to the

DALUT. In an N tap symmetric filter, the samples x(p) and x(N-P) should be added for p = n to n-N/2. The corresponding bits of these numbers should be fed as the input to the DALUT.

17.10.7 Implementation Results

An 8 tap symmetric FIR filter is implemented in [Xcell 28] using XC4003E-3 devices assuming both the impulse response coefficients and input samples to be represented using 8 bit signed numbers. The filter using parallel DA algorithm operates at a maximum frequency of 70MHz and requires 150 CLBs. The gate-efficient serial implementation of the 8 tap filter consumes only 30 CLBs but has a maximum sample rate of 7 MHz. We have an interesting tradeoff : the serial FIR filter uses 1/5 th the number of gates but produces 1/10 th the performance.

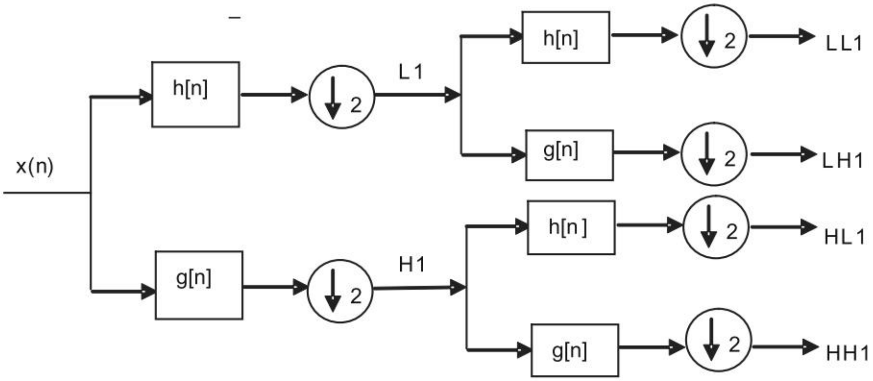


Fig. 17.28 *Block diagram for the computation of one level 2D DWT*

CASE STUDIES 17.11

Two examples of FPGA implementation of 2D DWT one using Xilinx FPGA and another using Altera FPGA with Nios processor and custom instruction are discussed in this section. These examples use both DAA and lifting scheme discussed earlier for the implementation.

17.11.1 Implementation of Image Encoder Using 2D DWT

In this section, implementation of one level 2-D DWT of sub images of size 32x32 on Xilinx Spartan II FPGA (XC2S150PQ208-5) is considered. For the implementation, 9/7 bi-orthogonal lowpass/highpass filters with filter coefficients given in Table 1.3 is assumed. The pixels and filter coefficients are assumed to be of size 11 and 8 bits respectively. 2D wavelet transform is computed using filter banks shown in Fig. 17.28. The input samples x(n) are passed through 2 stages of analysis filters. They are first processed by the low pass (h[n]) and high pass (g[n]) horizontal filters and are sub sampled by two. Subsequently, the outputs (L1, H1) are processed by low pass and high pass vertical filters. The 2D DWT is implemented using both lifting scheme and DAA and their results are compared.

Computation of 2D DWT Using Lifting Scheme The lifting scheme (Swelden [1998]) uses a polyphase structure for the analysis filter. In the lifting scheme, the odd and even input samples are processed using 5 blocks (α, β, γ, δ, ξ_1/ξ_2) in cascade as shown in Fig.17.28. ξ_1, ξ_2 are scaling blocks. Details of α and β blocks are shown in Fig. 17.29 and Fig. 17.30. γ and δ blocks are obtained by replacing the constants α, β with γ, δ. In order to speed up these blocks, the following techniques are adopted:

- Since the blocks are in cascade, the maximum rate at which the input can be fed to the system depends on the sum of the delays in all the four blocks. The speed is increased by introducing pipelining at the output of each of the blocks. In this case, the input rate is determined by the largest delay among all the four blocks.
- The delay in the individual stages is reduced further by using BW-PKCM (pipelined KCM with Baugh-Wooley multiplication algorithm)

Details of the α block implemented using BW-PKCM is shown in Fig.17.31.

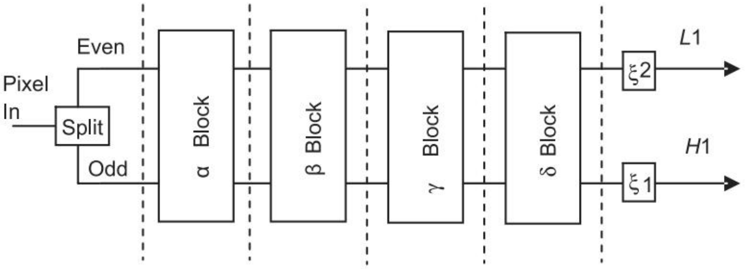


Fig. 17.28 *Simplified block diagram of Lifting Scheme for 9/7 filter*

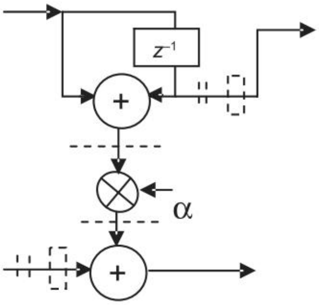


Fig.17.29 *Details of α block*

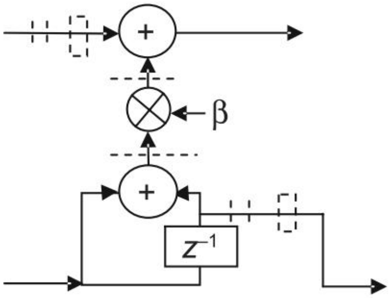


Fig. 17.30 *Details of β block*

Computation of 2D DWT Using Full Parallel DAA Scheme for computation of DWT using DAA for 9/7 Bi-orthogonal filter is shown in Fig. 17.32. This represents the horizontal filter section of Fig. 17.31 with the low-pass and high-pass filters implemented using separate DAAs. The DAA also exploits the symmetry property of the 9/7 filter and uses reduced size DA-LUTs. The vertical filter is also implemented in the same manner.

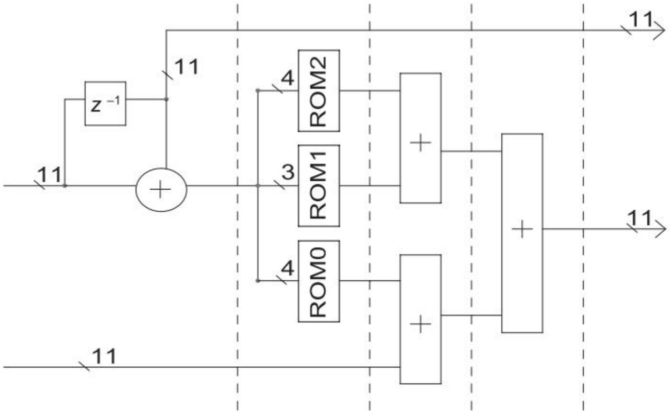


Fig.17.31 *Implementation of α block using BW-PKCM*

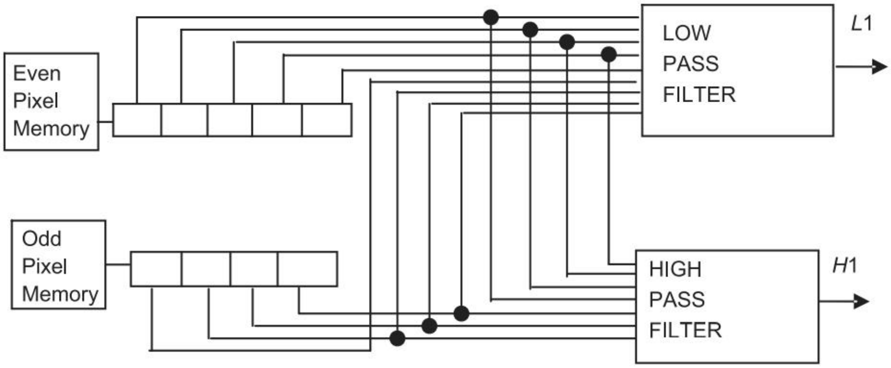


Fig. 17.32 *Computation of DWT using DAA*

The content of DA-LUT is determined assuming multiplication using BW multiplier (Baugh [1973]) for each of the filter coefficients and accumulating the individual products. For a N tap filter, when BW algorithm is used, corresponding to each of the N products, MSBs which are to be ignored, will actually be added and form the higher order $\log_2 N$ bits of the sum of products. For those products which are positive, a one which is to be ignored is actually added to the 2W^th bit of the sum of products. To get the correct result, a one has to be subtracted from the result corresponding to the 2W^th bit of the ROM. For example, assume that both the filter coefficients and the input samples are represented using 4 bit numbers. Let one of the filter coefficient h_0 be a positive number. If the MSB of the corresponding input sample x_0 is one, the product is negative and there will be no carry. Otherwise there would be a carry and to get the correct result, the no. 1000 0000 has to be subtracted from the content of the ROM. This can be achieved by adding the 2's complement of the number to the ROM. Since the sign of the filter coefficient is known apriori, by merely examining the sign bit of the input sample, the correction factor can be determined and correct result can be stored in the ROM.

Computation of 2D DWT for an image of Size 128X128 The image is split into a number of blocks in order to perform the computation of 2D DWT in parallel in a no. of ICs. Further, it reduces the memory required for storing the image and its transform. An overlapping scheme is proposed wherein the image block is formed such that a number of pixels overlapped between adjacent blocks along the vertical and horizontal direction is equal to the order of the filter. For example, for the 9/7 bi-orthogonal filter used for the 2D DWT, the no. of overlap pixels should be equal to 4 on the left and 4 on the right between horizontal blocks. Similarly, between vertical blocks, the number of overlap pixels should be equal to 4 on the top and 4 on the bottom. For the blocks on the boundary, overlapping needs to be done only on the non-boundary edge.

Implementation Results In order to test the functionality and for the purpose of comparison, 9/7 horizontal filters are implemented on SPARTAN II device (XC2S150PQ208–5) using lifting and DAA schemes.

For the implementation on SPARTAN II, the lifting multiplier constants (α, β, γ, δ, ξ_1, ξ_2) are assumed to be of 8 bits each. For the DAA, the filter coefficients are assumed to be of 11 bits for both high pass and lowpass filters. The horizontal filters are implemented using DAA BW algorithm. The lifting scheme is implemented using the BW-PKCM. The simulation results obtained for both the cases are found to be matching. Area as well as speed corresponding to both schemes are tabulated in Table 17.3.

Table 17.3 *Performance of 9/7 biorthogonal filters*

Name of the scheme using 11x8 multipliers	*Area (No. Of slices)*	*Speed in MHz*
Lifting with BW-PKCM	377	136
DAA with BW	397	136

From Table 17.3, it may be observed that the lifting scheme with BW-PKCM requires about 5% less area but has the same speed as that of the DAA scheme.

To verify the correctness and efficacy of the schemes proposed for the computation of 2D DWT, the overlap method proposed in the last section is used. The implementation of the one level 2D DWT for image block of size 32x32 is carried out using lifting scheme with BW-PKCM and the results are tabulated in Table 17.4. For storing the image input, outputs of the horizontal filter and the outputs of the vertical filters, the block RAMs are configured suitably. The image is loaded into the block RAMs through the user constraint file (UCF) of the implementation tool.

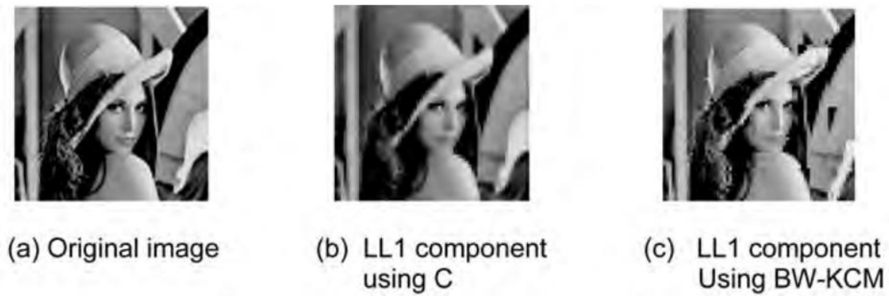
Table 17.4 *Implementation report on 1 level 32x32 2D DWT*

Scheme	*Area (no. of slices)*	*Speed in MHz*
Lifting with BW-PKCM	1197	100

The 2D DWT for the image is also computed using a C program. Lena image of size 128x128 with blocks (sub-images) of size 32x32 pixels is used for the computation of the 2D DWT. This is carried out using both high-level language C and hardware approach using FPGA. The Lena image shown in Fig. 17.33a is obtained by compressing the image dimension by a factor of 4 along both dimensions. Totally 36 image blocks are used for the 128x128 image.

For implementation in C language, the lifting multiplier constants (α, β, γ, δ, ξ_1, ξ_2) are declared as "double" type (64 bits) variables. The pixel intensities are declared as "short" type (16 bits) variables. The analysis filter output obtained corresponding to 36 image blocks are merged suitably and LL1 component of the image obtained using software (C program) and FPGA are shown in Fig. 17.33b and 17.33c respectively.

From these figures, it may be concluded that the LL1 components obtained through the FPGA implementation match well with that obtained using C.



(a) Original image

(b) LL1 component using C

(c) LL1 component Using BW-KCM

Fig. 17.33 *Orginal image and LL1 components of image*

17.11.2 System on Chip Implementation of 2D DWT Using Lifting Scheme

System on chip (SOC) kit based on Altera® APEX20K200E device is used for the implementation of 2D DWT. 2D DWT is computed using Nios softcore processor with and without custom instructions and the results are compared. The custom instructions are added to the Nios core as discussed in section 17.6. The sequential blocks require more than two clock cycles for correct operation. (The number of clock periods is chosen to be the number of clock cycles required for the sequential block + 1). The combinatorial blocks require 1 clock cycle. However, due to the overheads involved, for every call to a custom block, the Nios CPU spends at least 7 CPU cycles. For less computation intensive tasks, it would be preferable to make the Nios CPU to wait during every call to the custom block. For highly computation intensive tasks, it would be desirable to make the CPU concurrently working.

To ensure the concurrency between the Nios processor and custom logic, the scheme shown in Fig.17.34 is adopted. The execution of the custom instruction is split into a number of phases: write operand and operand number, issue start or reset signal, and read the results. Each phase of the instruction requires only 2-3 clock cycles. When a custom instruction is to be executed, first, the input data is fed into the input register file one after another. Next, a reset signal and start signal are issued to start the computation. The counter stops the computation process after a predetermined number of clock cycles. Using the read phase, the processor can read the result.

While adding custom instructions in SOPC builder, the number of cycles after which we need to process the result of custom block, is given as either 2 or 3.

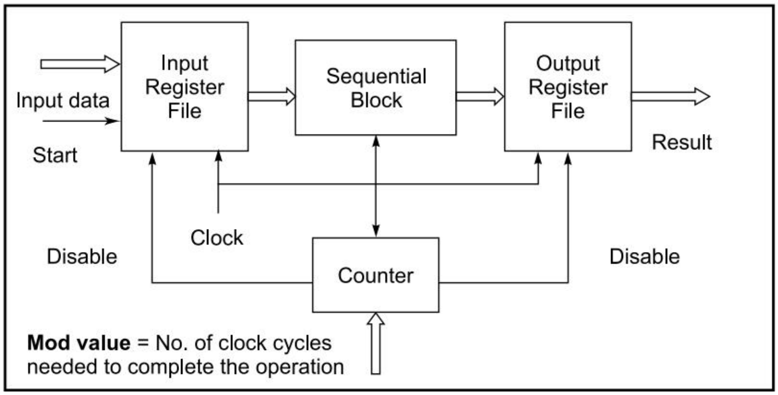


Fig. 17.34 *Scheme for concurrent execution of custom logic with Nios core*

As the input register file and output register file are the integral part of the custom logic, there is no need for separate arbitration for writing the data and reading the result of the custom logic. But the overhead here is additional memory needed to store the input data and output data in register files.

Implementation of the 2D DWT on the APEX FPGA The 2D DWT scheme given in Fig.1.39 is implemented on the APEX20K200 using the lifting blocks with 9/7 biorthogonal filters and BW-PKCM multipliers. The lifting multiplier constants (α, β, γ, δ, ξ_1, ξ_2) are assumed to be of 8 bits each and the input samples are assumed to be of 11 bits. For 2D DWT, image block of size 32x32 is assumed. The input image and the outputs of the horizontal filters as well as vertical filters are assumed to be stored in the block RAMs. For the horizontal filters, the even and odd inputs are applied from two block RAMs of size 512 X11. The result is written into 4 block RAMs of size 256X1. For the vertical filters, the inputs are applied from these four RAM blocks. For testing, the image is assumed to be loaded into the block RAMs using memory initialization file. However, the block RAMs may also be loaded using the data ports of the custom instruction. The block RAMs are realized using RAM library function *altsyncram.* The library function specifies that two clock cycles are required for either read or write operation. However, it has been found that write operation is satisfactory even with a single clock cycle.

From the synthesis and timing simulation using Quartus II, the critical path delay of the lifting blocks is found to be 20ns. The design is integrated with the Nios core and downloaded to the APEX device and tested. The 2D DWT is computed using the internal clock of frequency 66.66 MHz. These results matched with that obtained using a C program.

The 2D DWT block added as a custom block to Nios CPU and downloaded to the APEX device is studied under the following two cases:

- Declare the DWT block as a sequential block and run the DWT instruction in the foreground. The no. of clock cycles required for block is specified as 808. This includes 512 cycles required for the horizontal filters and 256 cycles required for the vertical filters. In this case, the Nios processor keeps waiting till the result is returned by the custom block.
- Declare the DWT block as a sequential block and run the DWT instruction in the background. The no. of clock cycles required for the block is specified as 3. The custom block is invoked with three prefix codes, one for clearing the internal registers, second for starting the computation of DWT and the third for latching any three words of the result in the output block RAM into the result port of custom block. The address of the RAM location to be read is specified through the prefix port. In this case, the Nios processor does not wait till the DWT is computed. It waits only for 3 clock cycles.

The results obtained with case 1 matched with the results obtained in case 2. 2D DWT is also computed using the in-built instruction set of Nios. The number of CPU clocks for both the cases are tabulated in Table 17.5. The custom instruction for 2D DWT is found to be faster by a factor of 90 compared to the implementation using C.

Table 17.5 *Computation time for 2D DWT for 32X32 sub-image*

Function	*No. of CPU clock cycles for Software approach*	*Equivalent CPU clock cycles for custom block*
2D- DWT	73280	814

The reading and writing of the block RAMs may be carried out concurrently with the computation of DWT by running the custom instruction in the background. Using dataa and datab outport ports of Nios core, 64 bits can be sent to the custom block for every custom call. In addition to this, the prefix port may also be used for sending data. If 2 bits of the prefix port are dedicated for sending the data, 66 bits can be sent to the custom block at a time. A 32x32 image requires 1024 eleven bit data to be transferred to the custom block. This can be achieved using 171 CPU calls to the custom block. Simultaneously about 512 words of 11 bit results can be read. To read the remaining 512 words additional 171 CPU calls are required. The area required for the 2D DWT and Nios Core are given in Table 17.6.

Table 17.6 *Resource utilization for 2D DWT and Nios Core for 32X32 sub-image*

Function	*No. of logical elements*	*No. Memory bits*
2D- DWT	3,705	34,688
CPU core	2,672	26,496
Available	8,320	1,06,496

COMPARISON OF THE PERFORMANCES OF THE SYSTEMS DESIGNED USING FPGAS AND DIGITAL SIGNAL PROCESSORS 17.12

Digital signal processors enjoyed a number of advantages over the FPGAs in the past, such as availability of

- efficient C compilers which enabled faster development and debugging of programs
- off the shelf subroutines for a variety of applications
- Interface with Matlab
- Good debugging environment such as code composer studio

But they are no longer valid. This is because the techniques for designing the system using FPGA have also matured over the past decade. The CAD tools for FPGA based systems support all the above features as discussed in section 17.5. However, FPGAs and digital processors have some distinct advantages of their own.

The digital signal processors have the following advantages over the FPGAs: for systems with moderate speed requirements, they require lower cost and lower power dissipation for implementation. They have power down modes which enable them to be power efficient.

On the other hand, FPGAs are suitable for systems which require very small sampling rates as well as the ones which require very high sampling rates. FPGAs offer the the flexibility to the users in choosing the word size. Adders and multipliers can be of any size. For example, when an application requires a 12 bit adder or a multiplier, the FPGA based design builds only the adder or the multiplier of the required size. In the case of digital signal processors, the word size is dictated by the size of the ALU. In the case of P-DSP,the size of the multiplier and adder may be fixed to be of 32 bits wide. Hence hardware of higher capacity is wasted for a smaller task. Further, the no. of adders and multipliers available in P-DSP is limited. In the case of FPGA based circuits, a no. of multipliers and adders can be realized in a single FPGA. Hence, parallel processing can be done to speed up the process. For example, a P-DSP operating at 100 MHz may be able to process only signals sampled at 10 MHz and below due to overheads involved in handling the repeat statements and loops. In the case of FPGAs operating at 100 MHz, the input sampling rates can be as high as 100 MHz.

In applications such as software defined radio, FPGAs outperform the digital signal processors because of the availability of a large no. of embedded multipliers in their core. For example, let us consider the implementation of 256 tap root-raised cosine (RRC) filter for processing inputs sampled at the rate of 200MSP. The total no. of MACs to be performed by the filter is 51.2 giga MACs/sec. This filter can be realized using a single Virtex-II Pro device. For example, XC2VP70 contains 328 multipliers and each multiplier can operate at a speed of 200MHz concurrently. (we require only 256 multiplier for the filter). On the other hand, the number of MAC units in a digital signal processor is limited. Let us consider a processor with 4 MAC units and clock speed of 600 MHz. Each of the processors can provide only 2.4 giga MACs/sec. Hence, to realize the above filter, we require 22 processors. This solution is area and power intensive and is costlier than the alternate approach using FPGAs. Hence, FPGAs are suited for very high speed applications. They may also be used as coprocessors to the P-DSPs and the advantages of both the approaches can be combined.(see for example Pirch[1997], Bosi[1999]

It may be noted that FPGAs are used to build prototypes of large digital systems and test them with minimum time. If the resulting system is to be used in large volumes, the design in the FPGA is translated to an ASIC and the cost of the system can be brought down further.

Review Questions

17.1 List the different tools used for front end and back end design of VLSI and briefly explain their function

17.2 Explain how interconnects are realized using (i) Antifuse (ii) SRAM (iii) EPROM.

17.3 Discuss the important features of the CLB in an FPGA such as XC4000.

17.4 How many CLBs are required to implement (i) 32X16 RAM (ii) 32 bit adder using XC4000.

17.5 How many CLBs are required to implement (i) 32X16 RAM (ii) 32 bit adder using Spartan III FPGA.

17.6 List the relative merits and demerits of antifuse and SRAM cells.

17.7 Explain the advantages of hw/sw partitioning of a system?

17.8 Why is a serial PROM used with Xilinx FPGAs?

17.9 Compare the features of XC4000 and Virtex 2 FPGAs

17.10 With an example show how FPGA outperforms P-DSPs for the implementation of high speed filters.

17.11 Give examples of hardcore and softcore processors realized on FPGAs. Discuss their relative merits.

17.12 Distinguish between distributed RAMs and block RAMs. Give examples of FPGAs which contain these RAMs.

17.13 Which FPGA requires serial EPROM? Why?

17.14 What is meant by in system programmability?

17.15 Which are the FPGA companies which use SRAM technology?

17.16 State anyone merit and demerit of using ACTEL FPGAs.

17.17 Explain the terms i antifuse ii Insystem programmability.

17.18 List the companies which develop FPGAs using antifuse technology and those using SRAM technology

17.19 With an example show how an FPGA baseband processor outperforms programmable DSPs

17.20 The highest frequency component that can be processed by an N tap filter implemented on P-DSP with MIPs rating of 100 MIPs is 250 KHz. Find the value of N. The above filter is implemented using the transpose of Direct form I structure on FPGAs. One multiply and accumulate operation requires 10 ns in FPGA. Find the maximum input frequency for which the FPGA based filter would be satisfactory

18 FPGAs IN TELECOMMUNICATION APPLICATIONS

Digital implementation of the various blocks of a communication system such as mixer (frequency changer), filter, modulator and demodulator has a number of advantages over their analog counterparts. Greater accuracy and stability frees digital circuits from the drifts caused by temperature, humidity, pressure and supply voltage changes. The ability to store the samples for longer time, enables repeated processing of the received data and leads to more accurate detection and demodulation performance. The highest sampling rate of the data converters (A/D and D/A) has dictated the portions of the communication system which are implemented using digital circuits.

EVOLUTION OF THE RADIO RECEIVER 18.1

Figure 18.1 shows the block diagram of the 1st generation radio receiver [Gunn]. The analog quadrature mixers translate the received signal to the baseband and further processing is carried out in the digital domain. In the second generation radio receiver, shown in Fig.18.2, the in phase and quadrature signals required for demodulation are generated using digital quadrature mixers. This has a number of advantages over analog quadrature mixers such as better matching of in phase and quadrature channel amplitudes and phases. For multichannel receiver, one set of quadrature mixers is required for each channel. In this case, the mismatch in the amplitude/phase between the in phase and quadrature signals results in coupling between the base band channels and leads to ghosts or images. To circumvent this, processing of the signal at the intermediate frequency using digital techniques have been proposed for applications such as multi channel receivers and transmitters for the base station of a cellular mobile communication system [Dick].

The block diagram of 1st generation and 2nd generation transmitters are shown in Fig.18.3 and Fig.18.4 respectively. In the 1st generation transmitters, modulation of the carrier by an information source is achieved using analog circuits. This is true even when the modulating signal is binary as in the case of BPSK/BFSK or when it takes anyone of the N discrete values as in the case of Quadrature Amplitude modulation (QAM). In the second generation transmitters, the information source modulates an intermediate carrier in digital form. The modulated signal is converted to analog signal and then upconverted to the required transmitter frequency.

Implementation of transmitters with modulators realized using digital technology gives a number of advantages: The carrier frequency and the type of modulation (QPSK, QAM, FSK, AM, FM) can be altered through software. This makes a transmitter to be suitable for multiple carrier frequencies and for multiple modulation schemes. Similarly, a receiver with low intermediate frequency blocks implemented in digital domain, enables it to be suitable for a range of carrier frequencies and a number of modulation schemes. Low intermediate frequency digital receivers have been proposed for a number of applications such as wireless LANs and IFM receivers for military communication. In the digital low IF receivers, the carrier frequency and the modulation scheme can be altered through software and hence these are examples of 1st generation software radio or software defined radio. In an ideal software radio, the A/D and D/A would be directly connected to the antenna and the rest of the blocks would be implemented using digital circuits. The lack of availability of very high speed, cost effective and low power data converters are presently the bottleneck for the implementation of all digital software radio. However, as the operating frequency of the data converters are increased with improvement in the technology and using novel techniques, the IF frequency of the digital receivers are also increased. Hence, implementation of a truly software defined radio is a distinct possibility.

The digital receivers with intermediate frequencies of the order of hundreds of MHz can be implemented only using either ASICs or FPGAs. Since the FPGAs are cost effective for low volume applications and are reconfigurable, they are more suitable for digital receivers. An example of a digital transceiver for wireless LAN is given in Fig.18.5 [Canet04]. In the first block on the transmit section, the input data is converted to 48 parallel data streams. They modulate 48 orthogonal carriers. The modulation type could be BPSK, QPSK, 16-QAM or 64-QAM. For phase tracking, 4 additional orthogonal carriers are used. They are modulated by pilot signals. The multicarrier modulation is efficiently achieved using the IFFT block. The 64-point IFFT coefficients are transmitted sequentially. They are followed by a guard band duration LAN corresponding to 16 samples. They are added to make the system robust to multipath and to prevent inter-symbol interference (ISI) from happening, this prefix can also be employed to have some tolerance for symbol timing. The input data rate is 20 Msymbols/sec. Hence IFFT is computed every 4 μs. (80X 0.05μs).

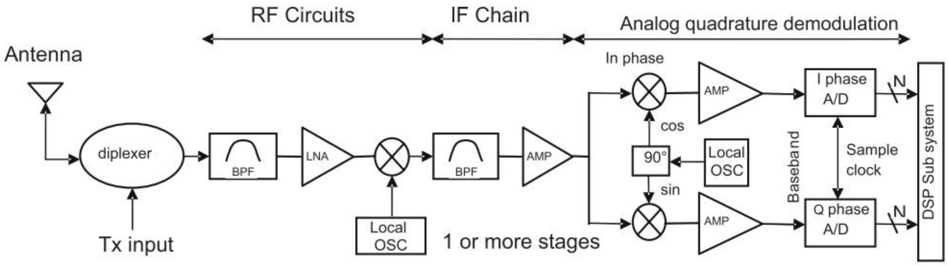


Fig. 18.1 *Block diagram of 1st generation receiver*

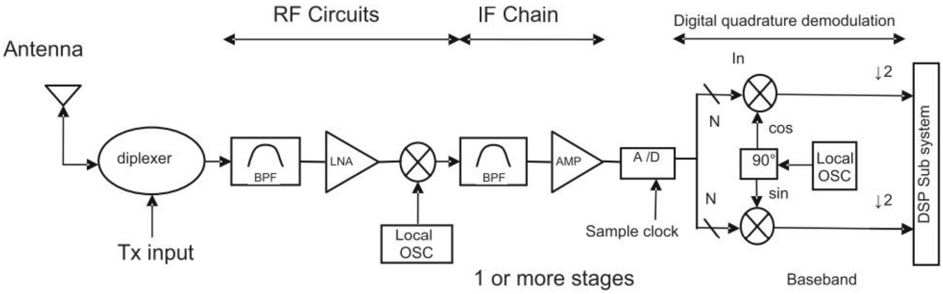


Fig. 18.2 *Block diagram of 2nd generation receiver*

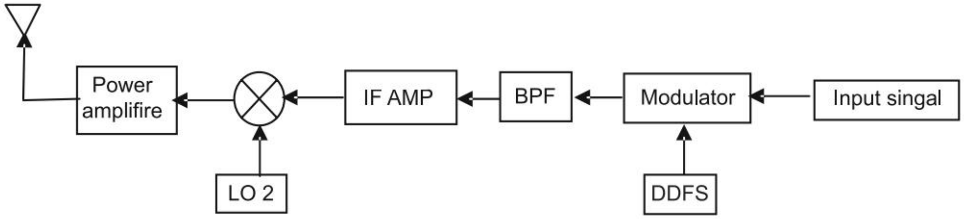


Fig. 18.3 *Block diagram of 1st generation transmitter*

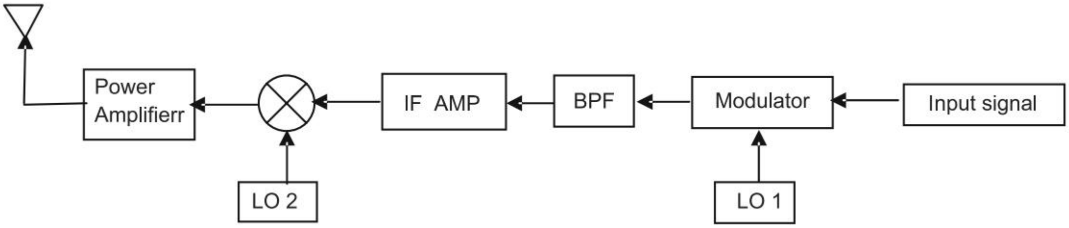


Fig. 18.4 *Block diagram of 2nd generation transmitter*

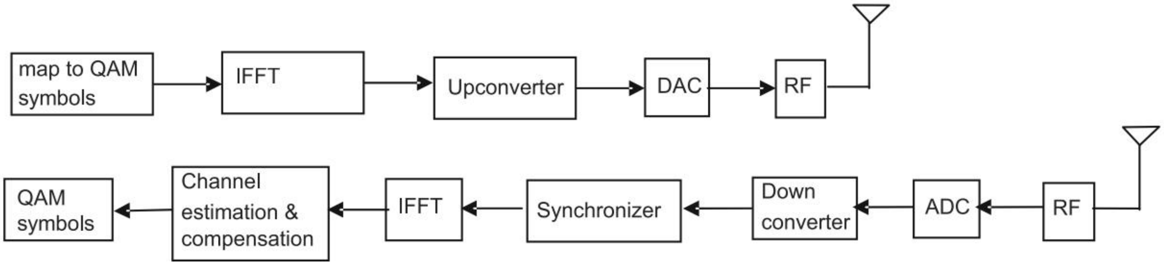


Fig. 18.5 *Block diagram of a digital transceiver for wireless LAN*

The output of the IFFT block has a frequency of 20 MHz and is upconverted to 120 MHz using two interpolation blocks with interpolation factors of 2 and 3 respectively. In Canet04, both filters are implemented using polyphase structure and distributed arithmetic algorithm with a precision of 8 bits.

The received analog signal is first downconverted to an IF of 45 MHz and then undersampled at the rate of 60 MHz resulting in a digital signal of 15 MHz. After digital mixing, the obtained base-band signal is low-pass filtered and decimated by 3 giving a base-band rate of 20 MHz. This is performed using a FIR filter of 8th order realized using a polyphase structure and distributed arithmetic with a precision of 10 bits. After down-conversion, the following stages are applied to the base-band signal: timing synchronization, coarse and fine carrier frequency offset (CFO) estimation and correction, FFT-based OFDM demodulation, channel estimation and compensation.

Several important design techniques used in Canet04 is worth noting: Use of IFFT/FFT for multicarrier modulation/demodulation, polyphase structure and distributed arithmetic for filters, ROM approach for generation of the carriers for the digital quadrature mixers, CORDIC scheme for coarse and fine carrier frequency offset (CFO) estimation as well as correction and undersampling scheme for reducing the sampling rate of the ADC. (Undersampling technique is used in most of the digital receivers and is based on bandpass sampling theorem: The minimum sampling rate – f_s required to sample a bandpass signal with lower, upper pass band frequencies of f_L and f_H respectively is $2(f_H - f_L) < f_s < 4(f_H - f_L)$ where f_s is chosen to be submultiple of either f_H or f_L). All the above blocks are implemented in a single Spartan III FPGA. Techniques used for efficient implementation of these blocks are discussed in more detail in section 18.2-18.4. These techniques are applicable for both ASICs and FPGAs.

DDFS WITH PHASE ACCUMULATOR AND ROM 18.2

Let us consider the implementation of direct digital frequency synthesizer (DDFS) [Lionel2004]. A periodic signal may be generated using a ROM, counter, Digital to Analog (D/A) converter and low pass filter as follows: If the signal is periodic with T, it may be sampled at M equidistant points in a period and the M samples may be stored in a ROM with M locations. The signal can then be regenerated by reading the content of ROM one by one and converting them to analog signal using the D/A converter. A modulo M counter is used for generating the $\log_2 M$ address inputs for the ROM. If M is a power of 2 (ie. $M = 2^N$) and f_s is the sampling rate, then, f_{min}, the fundamental frequency of the signal generated using the ROM is given by

$$f_{\min} = \frac{f_s}{2^N} \tag{18.1}$$

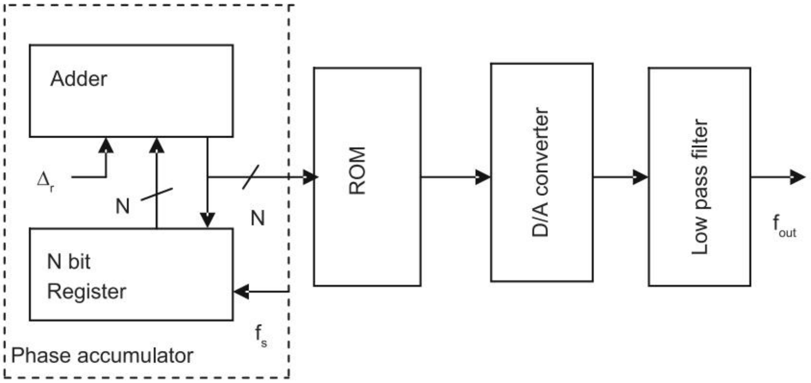


Fig. 18.6 *Block diagram of DDFS with phase accumulator and ROM*

Next, let us consider the case where the counter increments by 2 after every clock cycle. In this case, the modulus of the counter becomes M/2. If the ROM contents are read using the address generated by the above counter, f_{out} , the frequency of the analog signal generated becomes $2f_{min}$. In this case, only the alternate samples in the ROM are read and hence the period of the signal generated is reduced by half. Similarly, if the count is incremented by 3, the modulus of the counter is M/3 and f_{out} is $3f_{min}$. Proceeding in this fashion, higher and higher frequencies can be generated by reducing the modulus of the counter. However, f_{max} , the maximum frequency that can be generated in this fashion is limited to $f_{min}(M/2)$. This is because, for perfect reconstruction of a signal, we require at least 2 samples/period (sampling theorem).

$$f_{\max} = \frac{f_s}{2} \tag{18.2}$$

Figure 18.6 gives the block diagram of the DDFS using the above approach. The variable modulus counter is realized using the phase accumulator block as follows: The register is initially cleared. During the first clock period, the sum output of the adder output is Δ_r. During the next clock period, it is $2\Delta_{r.}$ Similarly, after every clock period, the output of the adder is incremented by Δ_r . After every M/Δ_r clock cycles, the adder output overflows. If Δ_r is 1, it takes M clock cycles for the adder to overflow and become 0. The output f_{out} can be expressed using Δ_r as

$$f_{\text{out}} = \frac{f_s}{2^N}\Delta_r \tag{18.3}$$

If f_i is denoted as the frequency generated when $\Delta_r = i$, it may be noted that $f_{i+1} - f_i = f_3 - f_2 = f_2 - f_1 = f_{min}$. In other words, the resolution of the frequency generator is f_{min}. If the ROM contains the samples corresponding to one period of a sinusoidal signal, incrementing the phase accumulator output by 1 is equivalent to advancing the phase of the sine wave wave by $2\pi/2^N$. In other words, phase resolution that can be achieved is $2\pi/2^N$.

18.2.1 ROM Size and Resolution

In the DDFS scheme with ROM, the resolution of the synthesizer can be chosen to be as small as we require, as it depends only on f_s and N. However, this is achieved at the price of increase in the number of locations and the number of bits to be stored/location. Ideally, the number of bits stored in each location of ROM should be N. This is required in order to ensure that the adjacent samples read, do not have the same value. If they are indeed the same, a ROM with only $2^N/2$ locations needs to be used. In this case, only the most significant N-1 bits of the adder is used as address to the ROM.

Example 18.1 If f_s - the sampling frequency of a DDFS is 1 MHz and the resolution is 1 Hz, the size of ROM computed using (18.1) is 10^6 X 20.

However, the size of the ROM required may be reduced from the following observation: If the DAC used is of a smaller resolution of W bits, a smaller ROM of size (2^W X W) is adequate. In this case, only the most significant W bits of the accumulator are used as the address and the remaining bits are truncated. Since the lower order N-W bits of the accumulator are not used, the address to the ROM changes only at intervals of 2^{N-W} sampling periods (ie. $2^{N-W}/f_s$). Hence, the output of the DAC remains constant for 2^{N-W} sampling periods and it introduces a number of spurious frequencies - f_{spur} , at the output, given by

$$f_{\text{spur}} = \frac{\Delta_r f_s}{2^{N-W}} - kf_s \tag{18.4}$$

where k is an integer. The low pass filter used at the output of DAC is designed to remove these spurious frequency components. For example, for the DDFS of Example 18.1, if a DAC with 8 bits is used, only a ROM of size 2^8 X 8 is required.

Example 18.2 Figure 18.7(a) shows 16 samples corresponding to one period of a sine wave. The 0^{th} and 8^{th} samples are zero valued. For generating the sine wave, a 4 bit phase accumulator and ROM with 16 locations are required. Figure 18.7(b) shows the output of an ideal D/A converter (it has zero conversion time and holds the value till the next sample is applied). Figure18.7(c) shows the output of the ideal D/A converter if a ROM with only 8 locations is used. In this case only the

3 MSB bits of the phase accumulator output is used as the address. It can be seen from Fig. 18.7(c), the DAC levels remain constant for 2 sample periods ($2T_s$). Further, it may be noted that the waveform in Fig. 18.7(c) may be obtained from Fig. 18.7(b) by adding a square wave whose period is $2T_s$ and whose amplitude changes with time. Hence, in addition to the desired frequency, the waveform shown in Fig.18.6.c, contains a sinewave of period $2T_{clk}$ and its harmonics. This is what is to be expected by the application of equation (18.4). In this example, N =4, W =3 and Δ_r =1, f_{out} = $f_s/16$ and fundamental component of $f_{spur} = 8f_{out} = f_s/2$

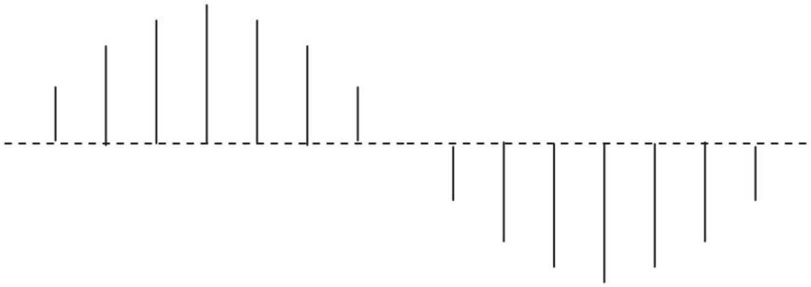


Fig. 18.7(a) *Samples of a sine wave sampled at the rate of 16f_s*

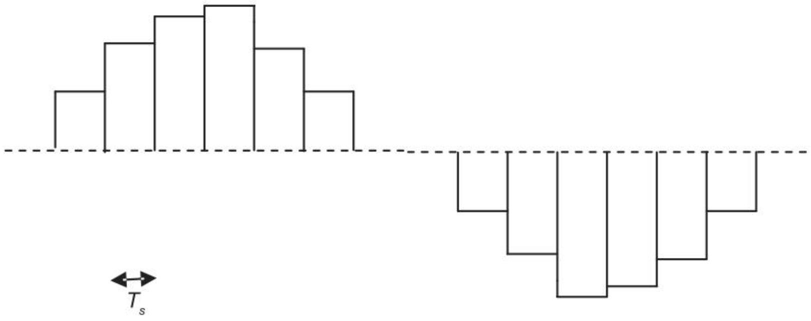


Fig. 18.7(b) *Output of an ideal D/A converter*

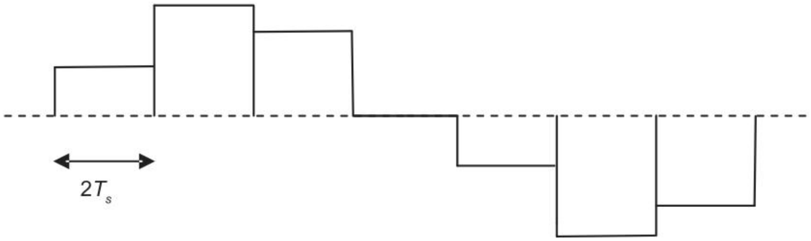


Fig. 18.7(c) *Output of the ideal D/A converter with 1 bit truncations*

In addition to spurious components due to phase truncation, additional spurious components are present when Δ_r is not a power of 2. In this case, when the phase accumulator overflows, it may not lead to 0 output. To understand this, let us consider a phase accumulator with N = 4 and list the consecutive outputs for Δ_r = 2, 3 corresponding to various clock cycles. Table 18.1 gives the outputs for both the cases at different clock cycles.

Table 18.1 *Output of the phase accumulator at different cycles*

Clk cycle no	1	2	3	4	5	6	7	8	9	10
Δ_r =2	0	2	4	6	8	10	12	14	0	2
Δ_r =3	0	3	6	9	12	15	2	5	8	11
Clk cycle no	11	12	13	14	15	16	17	18	19	20
Δ_r =2	4	6	8	10	12	14	0	2	4	6
Δ_r =3	14	1	4	7	10	13	0	3	6	9

When Δ_r =2 , phase accumulator overflows after every 8 clock cycles and starts with the value 0. But when Δ_r =3, the overflow does not occur periodically, it takes 6, 5, 5 clock cycles to overflow. The value of accumulator at these instants are 2,1,0 respectively. At clk number 1, the 1st period of the sinewave starts with phase offset of 0. At the 7th clock cycle, it starts with second period with an offset of 2 ($2\pi/2^4$). At the 12th clock cycle, the third period starts with an offset of $2\pi/2^N$. At the 17th clock, the 4th period starts with an offset of 0. It takes 16 clock cycles for resynchronization to occur. Since different periods start with different phase offsets, spurious frequencies result. The spurious frequency components due to periodic jitter is given by

$$f_{\text{jitter}} = \frac{g.c.d(\Delta_r, 2^N) f_s}{2^N} \tag{18.5}$$

where g.c.d. stands for the greatest common divisor.

18.2.2 ROM Compression Techniques

Compression Using Symmetry of Sine Wave When sinusoidal signals are to be generated, the number of ROM locations can be reduced by a factor of 4 using the symmetry property. From Fig.18.2a , it may be noted that only 4 samples corresponding to the first quarter of the periods needs to be stored. The samples in the 2nd quarter cycle can be obtained using mirror symmetry about $\pi/2$. The samples in the 3rd quarter cycle is obtained by using mirror antisymmetry (sign change as well as mirroring) about π. The samples in the 4th quarter cycle can be obtained using mirror symmetry about $3\pi/2$. For phase accumulator with N bits, only a ROM of $2^N/4$ locations are required for this method resulting in 75% savings in ROM.

Sine-Phase Difference Algorithm

This technique is also used to reduce the storage requirements for the quarter-wave sine function. The idea is to store

$$f(\phi) = \sin(\pi\phi/2) - \phi \tag{18.6}$$

instead of $\sin(\pi\phi/2)$ in the ROM LUT and calculate $\sin(\pi\phi/2)$ from $f(\phi) + \phi$. The sine-phase difference algorithm is shown in Fig.18.8.

The variation in f(ϕ) values is small, and thus a small LUT can be used to represent f(ϕ). sin(πϕ/2) can be easily calculated from f(ϕ). This technique can save as many as two bits of amplitude in the storage of the sine function. Moreover, the sine LUT propagation delay is reduced, increasing the maximum clock frequency of the DDS [Lionel 2004].

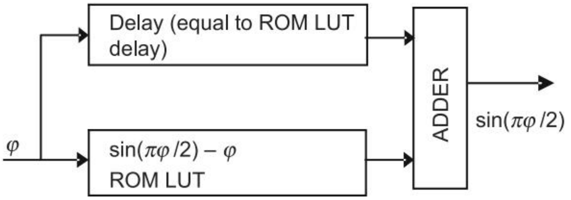


Fig. 18.8 *The Sine phase difference method*

Hutchison and Sunderland Algorithms In the Hutchison algorithm, the values of the sine function in the 1st quadrant is computed using two ROMs - a course ROM and a fine ROM. The given angle θ is split into two parts θ_C and θ_F such that the maximum value of θ_F is of the order of π/128. Then sin(θ) can be approximated as

$$\sin(\theta) = \sin(\theta_C+\theta_F)$$

$$= \sin(\theta_C)\cos(\theta_F) + \cos(\theta_C)\sin(\theta_F) \tag{18.7a}$$

$$= \sin(\theta_C) + \cos(\theta_C)\sin(\theta_F) \tag{18.7b}$$

The Sunderland algorithm is an improvement over the Hutchison algorithm and divides the phase angle into three parts, thus using three ROMs. The final value of the sine function is computed using suitable combinatorial logic at the output of the ROMs. If $\theta=\theta_C+\theta_s+\theta_F$ where θ_C is the coarse angle. θ_S is the Sunderland angle, and θ_F is the fine angle.

$$\sin(\theta) = \sin(\theta_C+\theta_S+\theta_F) \tag{18.8}$$

$$= \sin(\theta_C+\theta_S)\cos(\theta_F) + \cos(\theta_C)\cos(\theta_S)\sin(\theta_F)$$

$$- \sin(\theta_C)\sin(\theta_S)\mathrm{sub}(\theta_F) \tag{18.9}$$

The coarse angles are defined in the first quadrant of a sine wave from 0 to π/2, divided into 2^C equal angles. The Sunderland angle is defined as one of the coarse angles divided into 2^S equal angles. Finally, the fine angle is defined as one of the Sunderland angles divided in to 2^F equal angles. If θ_S and θ_F are small enough so that $\cos(\theta_S) \simeq 1$ and $\sin(\theta_S)\sin(\theta_F) \simeq 0$, then the following approximation can be used

$$\mathrm{Sin}(\theta) = \sin(\theta_C+\theta_S)\cos(\theta_F) + \cos(\theta_C)\sin(\theta_F) \tag{18.10}$$

18.2.3 Generation of Modulated Signal Using DDFS with Phase Accumulator and ROM

The DDFS scheme above can be used for generating a carrier which is modulated using any one of the modulation schemes such as frequency modulation, phase modulation, amplitude modulation and quadrature amplitude modulation. Circuit for the three analog modulation schemes are shown in Fig.18.8. Frequency modulation is performed by directly modulating, Δ_r –the Frequency Control Word, thus no additional hardware is needed to implement this feature. In this case, the modulating signal is digitized and fed to the Δ_r input.

Phase modulation is obtained by adding a phase offset to the phase accumulator output before addressing the ROM look-up table. In hardware, this amounts to incorporating an extra adder stage as shown in Fig. 18.8. The modulating signal is fed as one of the inputs to the 2nd adder preceding the ROM. As observed in section 18.2, incrementing the ROM address by Δ_r corresponds to incrementing the phase of the waveform by $(2\pi/2^N)\Delta_r$. Hence, if binary phase shift keying (BPSK) is required, for modulation by 1 level, the input to the 2nd adder should be (π /2^N)Δ_r so that a fixed phase shift of π is introduced into carrier signal in each of the periods. When it is to be modulated by 0, the input to the 2nd adder should be 0 so that the offset is zero. Similarly, the input to the 2nd adder for different phase shifts can be found for anyother phase modulation scheme.

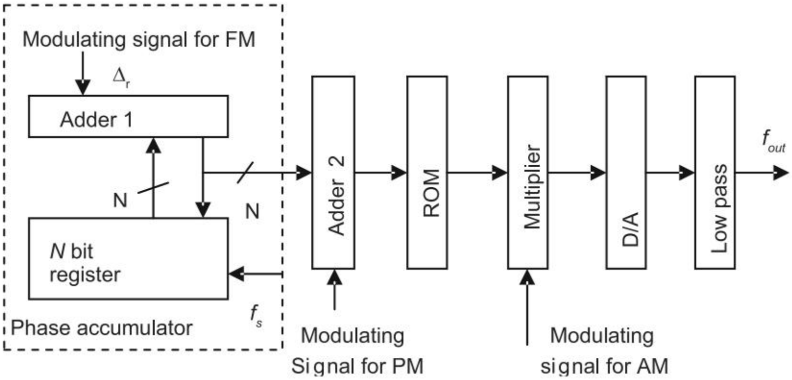


Fig. 18. 9 *Circuit for DDFS with modulation*

The amplitude modulation is achieved by multiplying the output of ROM with the modulating signal and then feeding the output to the DAC. To perform quadrature amplitude modulation, we require two ROMs one which contains the sine values and another which contains the cosine values. The output of these two ROMs are multiplied by the in phase and quadrature modulating signals and then applied to two DACs. This scheme requires 4 multipliers. Only the values of sine and cosine in the interval (0, π/4) needs to be stored into the ROM. Using complementary property of sine and cosine functions, the value of sinθ, for θ lying between π/4 to π/2, is found using the ROM for cos(π/2–θ). Similarly, the value of cosθ, for θ lying between π/4 to π/2, is found using the ROM for sin(π/2 – θ).

COORDINATE ROTATION DIGITAL COMPUTER (CORDIC) ALGORITHM AND ITS APPLICATIONS 18.3

The Coordinate Rotation Digital Computer (CORDIC) algorithms, proposed by Jack Voider[Vol 59] can be used for computation of a wide range of functions including certain trignometric, hyperbolic, linear and logarithmic functions. CORDIC unit uses only shifts and adds to compute these functions. CORDIC algorithm is used in diverse applications such as mathematical coprocessor units, calculators, waveform generators and digital modems.[Jean93]. Different architectures for the implementation of CORDIC unit in FPGAs are considered in [Ray78].

18.3.1 CORDIC ALGORITHM

The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using shifts and adds. In the rotation mode, CORDIC is used for converting one vector in rectangular form to another vector in rectangular form. In the vector mode, it converts a vector in rectangular form to polar form.

Rotation Mode of CORDIC

The CORDIC algorithm for this mode is derived from the general rotation transform

$$x_{fin} = x_{in}\cos\theta - y_{in}\sin\theta \qquad (18.11a)$$

$$x_{fin} = x_{in}\cos\theta + y_{in}\sin\theta \qquad (18.11b)$$

which rotates a vector (x_{in}, y_{in}) in a Cartesian plane by an angle θ to another vector with the coordinates (x_{fin}, y_{fin}). The rotation may be achieved by performing a series of successively smaller elementary rotations θ_0, θ_1. θ_2 … θ_N such that $\theta = \Sigma_0^N\theta_i$. Fig. 18.10, shows the case where a rotation of a vector of magnitude 1 by an angle θ is achieved using three elementary rotations θ_0, θ_1 and θ_2.

Rotation of the vector by an angle can be rewritten as

$$x_{i+1} = x_i\cos\theta_i - y_i\sin\theta_i \qquad (18.12)$$

$$y_{i+1} = y_i\cos\theta_i + x_i\sin\theta_i \qquad (18.13)$$

$$\frac{x_{i+1}}{\cos\theta_i} = x_i - y_i\tan\theta_i \qquad (18.14)$$

$$\frac{y_{i+1}}{\cos\theta_i} = y_i + x_i\tan\theta_i \qquad (18.15)$$

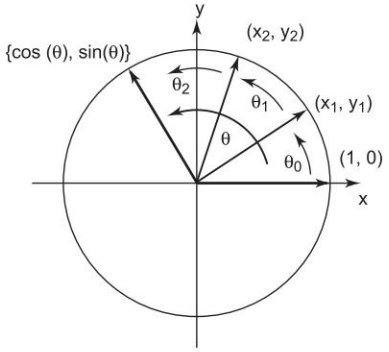


Fig. 18.10 *Rotation of a vector by an angle θ_i using a number of steps*

The computational complexity of (18.14), (18.15) can be reduced by rewriting these equations as

$$x_{i+1} = x_i - y_i\tan\theta_i \qquad (18.16)$$

$$y_{i+1} = y_i - x_i\tan\theta_i \qquad (18.17)$$

$$\left(x_{fin}, y_{fin}\right) = \left(x_N \Big/ \prod_1^N \cos\theta_i ,\; y_N \Big/ \prod_1^N \cos\theta_i\right) \qquad (18.18)$$

and performing the division by cos together for all the N iterations by dividing the value of (x_N,y_N) by $\prod_1^N\cos\theta_i$. Further, the value of for i =1, 2, .. N is chosen such that tan is 2^{-i} . The values of angles for i = 0- 9 are given in Table 18.2.

Table 18.2 *The values of $\theta_i = \tan^{-1}(2^{-i})$*

I	θ_i	tan θ_i
0	45	1
1	26.5	0.5
2	14	0.25
3	18.1	0.125
4	3.57	0.0625
5	1.78	0.03125
6	0.895	0.015625
7	0.4476	0.0078125
8	0.2238	0.00390625
9	0.1119	0.001953125

This reduces the multiplication by the tan θ_i to simple shift operation. As the iteration increases θ_i becomes smaller and smaller. We may terminate the iteration when the difference between θ & $\Sigma_1^N\theta_i$ becomes very small for some value of N. The remaining angle by which the vector needs to be rotated after the completion of i iterations is indicated by the parameter z_{i+1} defined by equation (18.19).

$$z_{i+1} = z_i - \theta_i \qquad (18.19a)$$

$$z_0 = \theta \qquad (18.19b)$$

θ_i is considered to be positive when the rotation required is anticlock wise and is negative otherwise. To approximate an arbitrary angle using θ_i of the form $\tan^{-1}(2^{-i})$, θ_i may have to be chosen to be negative for some values of i. For example, to approximate 50, we have to choose as 45, 26.5, –14, –18.1 in the first four iterations.(The sum of these angles is 50.4). The sign (sgn) of z_i indicates whether in the next iteration, the rotation has to be anticlockwise or clockwise. Since, tan is $+2^{-i}$ when is positive and -2^{-i} otherwise, the iterative equations may be rewitten as

$$\delta_i = \text{sgn}\,(z_i) \qquad (18.20)$$

$$x_{i+1} = x_i - \delta_i\, y_i\, 2^{-i} \qquad (18.21)$$

$$y_{i+1} = y_i - \delta_i\, x_i\, 2^{-i} \qquad (18.22)$$

$$z_{i+1} = z_i - \delta_i \tan^{-1}(2^{-i}) \qquad (18.23)$$

The computation of $\prod_1^N \cos\theta_i$ may be simplified as follows: Since cos $\theta_i = 1$ for very small values of θ_i, $\prod_1^N \cos\theta_i$ may be computed for N=6 and may be used for any other value of N > 6. For N=6, K $= \prod_1^6 \cos\theta_i = 0.6073$.

18.3.2 Vector Mode

In this mode, an initial vector with the x, y coordinates of $(x_{\text{in}}, y_{\text{in}})$ is rotated such that its y coordinate becomes zero. The procedure used for rotation may be adopted for vector mode with the following modifications : Rotation is carried out along clock wise direction (so that the y coordinate can be made 0); The total angle by which the vector has been rotated from the initial position after i rotation is indicated by the parameter z_{i+1} and z_0 is defined as 0. The resulting equations are given in (18.24) – (18.27). Equation (18.24) is obtained as follows: When y_i is positive, the rotation should be along the clockwise direction in the next iteration. When it is negative, it has to be along the anticlockwise direction.

$$\delta_i = \text{sgn}\,(z_i) \qquad (18.24)$$

$$x_{i+1} = x_i - \delta_i\, y_i\, 2^{-i} \qquad (18.25)$$

$$y_{i+1} = y_i - \delta_i\, x_i\, 2^{-i} \qquad (18.26)$$

$$z_{i+1} = z_i - \delta_i \tan^{-1}(2^{-i}) \qquad (18.27)$$

As i becomes large, $y_i \to 0$ and x_{fin}, the magnitude of the vector after N iterations and z_{fin}, the angle of the vector are obtained as

$$x_{fin} = x_N \Big/ \prod_N^1 \cos\theta_i \qquad (18.28)$$

$$z_{fin} = \tan^{-1}\left(\frac{y_0}{x_0}\right) \tag{18.29}$$

18.3.3 Applications of CORDIC in Rotation Mode

DDFS Using CORDIC with Phase Accumulator Fig. 18.11 shows the circuit diagram of DDFS using CORDIC. The expression for the minimum and maximum frequencies generated and the frequency resolution are the same as that for DDFS with ROM. The output of the phase accumulator is fed to the θ input of the CORDIC and is mapped such that $2^N = 2\pi$. Unlike the ROM approach, this generates two carriers in quadrature simultaneously. Compared to the ROM approach, CORDIC approach requires lower area but the highest frequency that can be generated is lower than that of ROM approach. This is because of the larger computation time required for rotation compared to the time required for reading from ROM. This can be explained as follows: If CORDIC requires M iterations for rotation, M stages of rotation blocks may be used. The 1st block rotates the input vector by π/4 (i.e. $\tan^{-1}$ (2^{-0}), the 2nd block by $\tan^{-1}$ (2^{-1}) and (i+1)th block rotates the input vector by angle $\tan^{-1}(2^{-i})$.

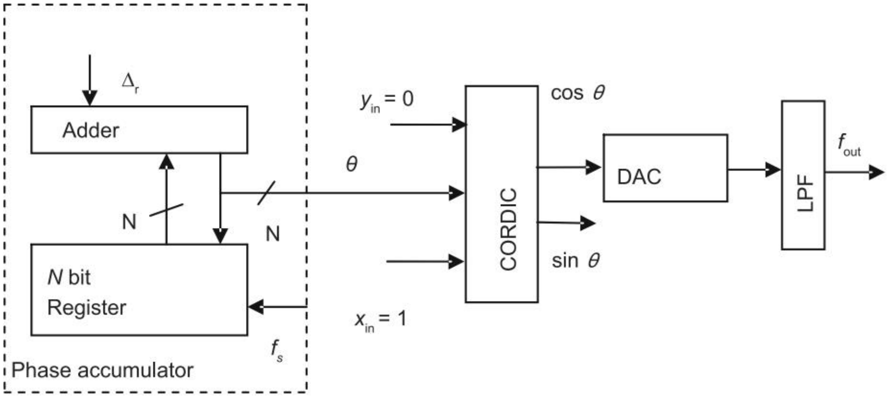


Fig. 18.11 *Circuit diagram of DDFS using CORDIC*

Table 18.3 *Number of iterations required (M) for different sizes of adder (N)*

N	*tan(2^{-N}) computed using*		*M : number of iterations*
	library function in C	*CORDIC*	
4	0.414213	0.412651	13
5	0.198912	0.199332	14
6	0.098491	0.097918	12
7	0.049127	0.048959	13
8	0.024549	0.024479	14
9	0.012272	0.012240	15
10	0.006136	0.006120	16
11	0.003068	0.003060	17
12	0.001534	0.001530	18
13	0.000767	0.000765	19

Such a scheme is referred to as unrolled CORDIC and is shown for M= 4, in Fig. 18.12. In this case, the time required for rotation by an angle θ, is the sum of the delays in the M blocks. Pipelining may be used for speeding up the rotation operation. For this purpose, pipelining registers may be used in between the adjacent blocks. Number of iteration required depends on the frequency resolution. Table 18.3 gives the number of iterations required (M) for different sizes of adder (N) in the phase accumulator. The iteration is terminated when the values of tan(2^{-N}) computed using the library function in C and that computed using CORDIC differ by less than 1%.

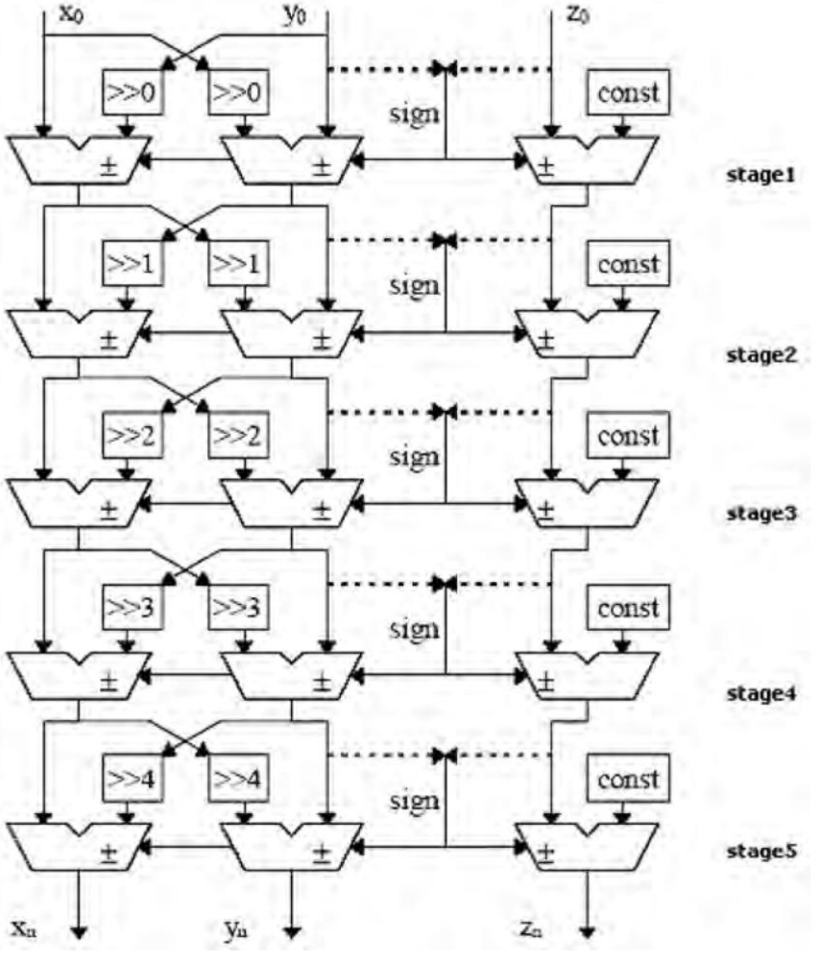


Fig. 18.12 *Block diagram of unrolled CORDIC unit*

DDFS Using CORDIC as Universal Modulator The universal modulator is obtained from the DDFS circuit of Fig. 18.11 by including an adder at the output of the phase accumulator and the resulting circuit is shown in Fig.18.13. (x_{in}, y_{in}) denote the inputs for amplitude modulation. $\phi(t)$ *and* Δ_r denote the inputs for phase and frequency modulation respectively. When (x_{in}, $y_{in,}$ $\phi(t)$) are chosen as (1,0,0) and Δ_r is chosen as a constant, the circuit shown in Fig.18.13, generates two unmodulated carrier signals in quadrature (ie. *sin*ω*t* and *cos*ω*t*). When modulating signals are applied, modulated carriers are obtained. When amplitude modulation is required, the modulating input is applied to the x_{in} input and y_{in} is made 0. For the generation of carrier with quadrature amplitude modulation, the two modulating signals are applied to x_{in} and y_{in} respectively. For generating the carrier with phase, frequency modulation, the modulating inputs are fed to $\phi(t)$ *and* Δ_r respectively. When $\phi(t)$ *and* Δ_r are binary inputs, the carrier modulated with BPSK and BFSK are obtained. When (x_{in}, y_{in}) are binary valued (1's and 0's), the QPSK signal is obtained at the outputs.

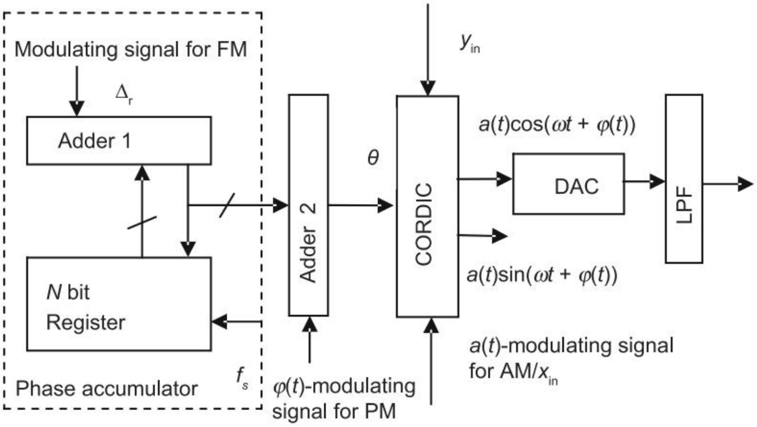


Fig. 18.13 *CORDIC as universal modulator*

18.3.4 Applications of CORDIC in Vector Mode

CORDIC as Universal Demodulator If the carrier frequency, amplitude and phase of the received signal are f_i , 2b(t) and θ(t) respectively, then received signal r(t), is given by

$$r(t) = 2b(t)\sin(\omega_i t + \theta(t)) \qquad (18.30)$$

One of the approaches for demodulating the above signal is the generation of the in-phase I(t) signal and quadrature signal Q(t) given by

$$I(t) = b(t)\sin[2\pi(f_i - f_o)t + \theta(t)] \qquad (18.31)$$

$$Q(t) = b(t)\cos[2\pi(f_i - f_o)t + \theta(t)] \qquad (18.32)$$

using a local oscillator of frequency is f_o with known initial phase.

The in-phase and quadrature signals may be fed to the two inputs of CORDIC operated in rotation mode. The magnitude of the vector gives the demodulated signal corresponding to amplitude modulation as

$$x_{fin} = \sqrt{I^2(t) + Q^2(t)} = b(t) \qquad (18.33)$$

Similarly, z_{fin}, the angle of the vector gives the phase shift introduced into the carrier and is given by

$$z_{fin} = \tan^{-1}\left(\frac{Q(t)}{I(t)}\right) = \tan^{-1}\left(\frac{\sin[2\pi(f_i - f_o)t + \theta(t)]}{\cos[2\pi(f_i - f_o)t + \theta(t)]}\right) = (f_i - f_o)t + \theta(t) \qquad (18.34)$$

Differentiating (18.34), the instantaneous frequency of the carrier can be found.

Generation of in Phase and Quadrature Signals Using the quadrature mixers shown in Fig. 18.14, I(t) and Q(t) may be generated. In this scheme, the input signal is divided into two in-phase paths and the local oscillator signals applied to the two mixers are 90-deg out of phase. The outputs of the two mixers are

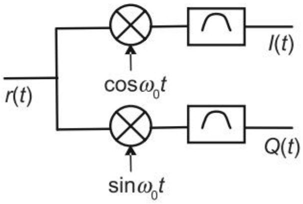


Fig. 18.14 *Generation of in-phase and quadrature signals*

$$V_{\text{if1}} = 2b(t)\sin(2\pi f_i t + \theta(t)) \cos(2\pi f_o t) \quad (18.35)$$

$$= b(t)\{\sin[2\pi(f_i - f_o)t + \theta(t)] + \sin[2\pi(f_i + f_o)t + \theta(t)]\} \quad (18.36)$$

$$v_{\text{if2}} = 2b(t)\sin(2\pi f_i t + \theta(t)) \sin 2\pi f_o t) \quad (18.37)$$

$$= b(t)\{\cos[2\pi(f_{i\,\_}\,f_o)t + \theta(t)]\text{-}\cos[2\pi(f_i + f_o)t + \theta(t)]\} \quad (18.38)$$

The desired I & Q components are obtained by using low pass filters which filter out the high-frequency signals represented by the f_i+f_0 terms in (18.36) and (18.38).

The I & Q components can be generated without using quadrature mixers using a scheme called as **special sampling scheme or double nyquist rate sampling scheme**.[Pellon92] In this case, if the local oscillator frequency is f_0, then r(t) is sampled at a rate of $f_s = 4f_0$. If $V_{\text{if1,}}$ V_{if2} and r(t) are sampled at a rate of $f_s = 4f_0$, then

$$t = nTs = n /4f_0 \quad (18.39)$$

$$2\pi t f_0 = 2\pi\ (nTs)f_0 = 2n\pi\ /4 = n\pi/2 \quad (18.40)$$

If we substitute (18.40) in (18.32), (18.35) and (18.37), the sequence of outputs at different sampling instants are as shown in Table 18.4.

From Table 18.4, it can be verified that the in phase and quadrature components can be generated by alternately passing r(t) to one of the two channels and inserting zeros alternately to each channel as shown in Fig. 18.15. The desired I & Q components are obtained by using low pass filters which filter out the high-frequency signals represented by the f_i+f_0 terms in (18.36) and (18.38). Let us denote the value of r(t), I(t), Q(t) at t = nTs as r(n), I(n) and Q(n) respectively. Then I(n) and Q(n) can be written as

I(n) = r(0), 0, -r(2), 0, r(4), ….

Q(n) = 0, r(1), 0, -r(3), 0, r(5), ….

What happens if the local oscillator frequency f_0 is chosen to be same as the the input frequency f_i? In this case also, f_s may be chosen as $4f_0$. Then, r(t), I(t) and Q(t) at various sampling instants are as shown in Table 18.5.

Hence, without using the mixers, the in-phase and quadrature components can be generated. The price paid is the use of higher sampling rate.

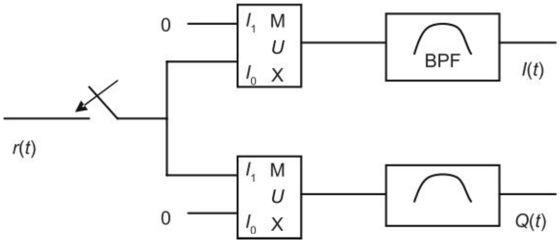


Fig. 18.15 *Generation of in phase and quadrature signals using special sampling scheme*

Table 18.4 *Samples of the received signal, in phase & quadrature channels for $f_0 \neq f_i$*

Sample at t = tn = nTs for n =	V_{if2}	V_{if1}	*R(t)*
0	0	$b(0)\sin(\theta)$	$b(0)\sin(\theta)$
1	$b(t_1)\sin(2\pi f_i t_1 +)$	0	$b(t_1)\sin(2\pi f_i t_1 +)$
2	0	$-b(t_2)\sin(2\pi f_i t_2 = \theta)$	$-b(t_2)\sin(2\pi f_i t_2 + \theta)$
3	$-(t_3)\sin(2\pi f_i t_3 + \theta)$	0	$b(t_3)\sin(2\pi f_i t_3 + \theta)$
4	0	$b(t_4)\sin(2\pi f_i t_4 + \theta)$	$b(t_4)\sin(2\pi f_i t_4 + \theta)$

Table 18.5 *Samples of the received signal, in phase & quadrature channels for $f_0 = f_i$*

Sample at t = tn =	V_{if2}	V_{if1}	*R(t)*
0	0	$b(0)\sin(\theta)$	$b(0)\sin(\theta)$
1	$b(t_1)\cos(\theta)$	0	$b(t_1)\cos(\theta)$
2	0	$-b(t_2)\sin(\theta)$	$-b(t_2)\sin(\theta)$
3	$-b(t_3)\cos(\theta)$	0	$b(t_3)\cos(\theta)$
4	0	$b(t_4)\sin(\theta)$	$b(t_4)\sin(\theta)$

18.3.5 Application of CORDIC in Carrier Recovery Circuits

The coherent demodulators give 1 dB better signal to noise ratio than the non coherent demodulators but require the phase of the incoming carrier signal to be tracked. For carrier phase recovery, two common techniques are the phase locked loop and the Costas loop. Block diagrams of these two techniques are given in Fig. 18.14 and Fig. 18.15 respectively. Note that the phase detectors can be implemented using multipliers. In Fig. 18.14, if the input signal is *r*(*t*) and the local oscillator signal is c(t), then the output of the multiplier m(t) can be written as

$$r(t) = \sin(2\pi f_i t + \theta(t)) \qquad (18.41)$$

$$c(t) = \cos(2\pi f_i t) \qquad (18.42)$$

$$\begin{aligned} m(t) &= \sin(2\pi f_i t + \theta(t))\cos(2\pi f_i t) \\ &= (1/2)\ \{\sin[\theta(t)]+\sin[2\pi(2f)t + \theta(t)]\} \qquad (18.43) \end{aligned}$$

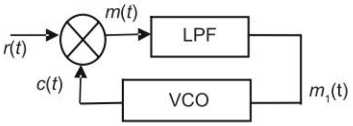


Fig. 18.16 *Carrier recovery using PLL*

The second term of (18.43) can be removed using the low pass filter and the output of the low pass filter m1(t) for small values of θ(t) is

$$m1(t) = (1/2)\theta(t) \qquad (18.44)$$

In the case of costas loop, a quadrature mixer is used to generate the inphase and quadrature signal and they are multiplied to get the phase of the carrier. If $m_I(t)$, $m_Q(t)$ denote the output of

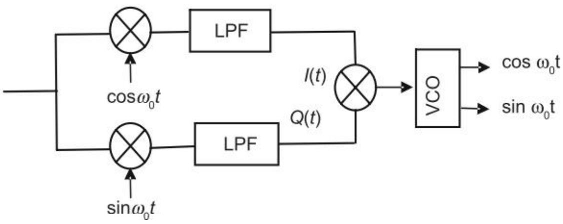


Fig. 18.17 *Carrier recovery using costas loop*

the inphase and quadrature channel outputs after passing through the low pass filter, the output of the third phase detector (multiplier) for small values of θ(t), mp(t) can be written as follows:

$$m_I(t) = \sin[\theta(t)] \tag{18.45}$$

$$m_Q(t) = \cos[\theta(t)] \tag{18.46}$$

$$m_P(t) = \sin[\theta(t)] \cos[\theta(t)] = (1/2) \sin[2\theta(t)] = [\theta(t)] \tag{18.47}$$

Hence, the Costas loop gives an output double that of PLL at the cost of increase in hardware complexity. However, the Costas loop may be implemented using CORDIC in vector mode. This scheme requires only one multiplier and does not require the two carrier signals $\sin(2\pi f_i t)$ and $\cos(2\pi f_i t)$ in quadrature.

18.3.6 Subsampling Receivers

In narrow band systems such as commercial FM, the bandwidth required for the required station is about 200KHz whereas the carrier frequency lies in the range 88-108 MHz. In such systems, the band pass sampling theorem can be used to choose the sampling rate so as recover the modulating signal instead of the carrier. In the IF sampling or subsampling receiver for the commercial FM, the sampling rate may be chosen to be 80MSPS . This introduces aliasing and the aliased signal (8-28MHz) contains the required frequency bands. If this is downsampled by a factor of 200, 8 MHz will be mapped to 0 Hz and 8.1 MHz, the carrier corresponding to the first channel is aliased to 100KHz. Since the sampling rate is 400 KSPS, this corresponds to double nyquist rate for the modulated signal at 100KHz. Hence, inphase and quadrature signals can be generated without using using multipliers as shown in Fig. 18.15. Note that the wireless LAN discussed in section 18.1 also uses subsampling scheme.

18.3.7 Efficient implementation of low pass filters

Low pass filters required for I/Q channel generation needs to operate at very high sampling rates. A special case of the filter given by (1.39a) is called as Moving average filter or boxcar filter when a =1. This becomes a multiplier free low pass filter and is preferred for systems with very high sampling rates. It is used in a variety of applications such as oversampling A/D converters and in wireless systems for conversion of signals sampled at IF and RF rates to baseband rate. MA filter is usually realized using a cascade of integrator I(z) and a comb filter C(z) with transfer functions given by

$$I(z) = \frac{1}{1 - z^{-1}} \tag{18.48}$$

$$C(z) = 1 - z^{-M} = 1 - z^{-LN} \tag{18.49}$$

The boxcar filter using the cascaded integrator comb (CIC) filter is shown in Fig. 18.18 where L and N are integers and M=LN.

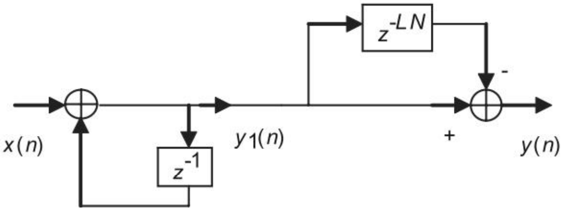


Fig. 18.18 *Boxcar filter using a cascade of comb and integrators*

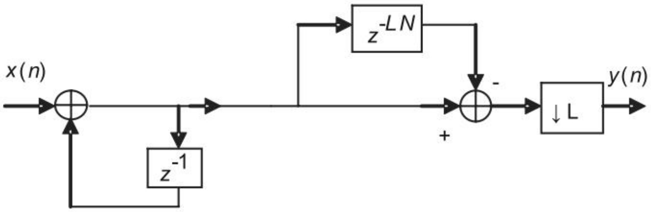


Fig. 18.19(a) *CIC filter followed by decimator by factor L*

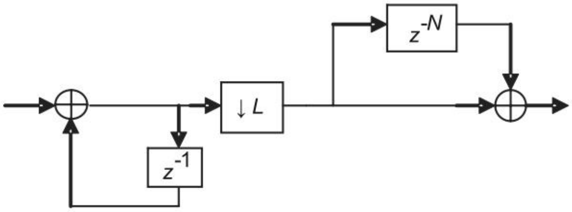


Fig. 18.19(b) *Efficient implementation of CIC followed by decimator*

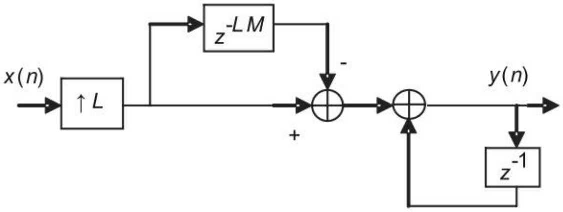


Fig. 18.20(a) *CIC filter preceded by interpolator by factor L*

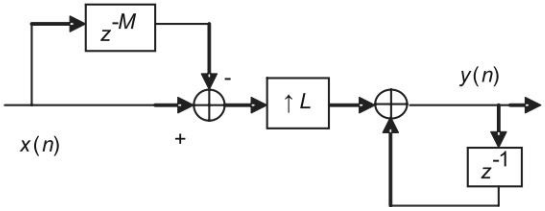


Fig. 18.20(b) *Efficient implementation of CIC filter preceded by interpolator*

The low pass filter may be used either with decimator or interpolator. In this case the order of interpolation/decimation may be interchanged with comb filter in order to reduce the complexity of comb filter using the noble identity. In downsampling systems using CIC filters, integrators are connected first, followed by decimators and the comb filter as shown in Fig. 18.19. In upsampling systems, comb filter is connected first followed by interpolator and the integrator as shown in Figs 18.20(a) and 18.20(b). The CIC filters using the noble identity for efficient realization is called as Hogenauer filter.

Normally a single stage boxcar filter may not be adequate to provide sufficient out of band attenuation and hence number of stages of comb and integrators are used in cascade. A CIC filter with K has stages in cascade has the transfer H(z) given by

$$I(z)=\frac{1}{1-z^{-1}} \tag{18.50}$$

A K stage CIC filter may be converted to a Hogenauer filter by connecting all K stages of integrators first, followed by decimator by a factor L and K stages of comb section with delay of L samples. Figures 18.21(a) and 18.21(b) show a 2 stage CIC filter.

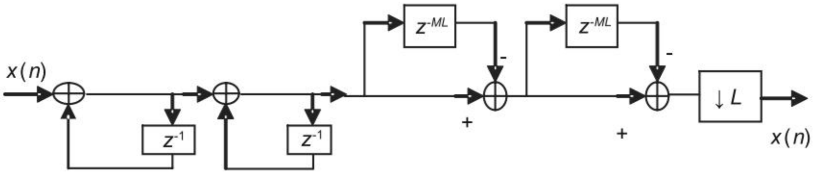


Fig. 18.21(a) *Two stage CIC filter followed by decimator*

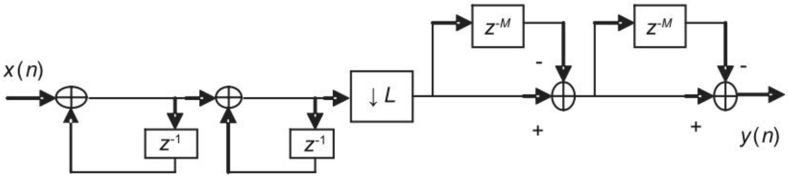


Fig. 18.21(b) *Two stage Hogenauer filter*

Size of the Adders for CIC Filters As CIC filter computes the moving average of present and past M-1 samples, the adder size should n+ $\log_2 M$ for inputs of n bit wide in order to avoid overflow. This can be verified as follows: If two n bit numbers are added, the sum should be represented using n+1 bits. If four n bit numbers are added, the sum should be represented using n+2 bits. Hence, if M, n bit numbers are added, the sum should be represented using n+$\log_2 M$ bits. Similarly, if K stages of CIC filters are used, the word size should be n+K$\log_2 M$ bits in order to avoid overflow. Sampling rate reduction by a factor of few hundreds is not uncommon in A/D converters and digital radio recivers. Let us consider an example:

Example 18.3 Find the word size required for a single stage of 256 tap CIC filter whose inputs are 8 bit wide. The CIC filter is succeeded by decimation by a factor of 128. If 4 such CIC filters are cascaded and realized using Hogenauer filter, find the word size.
Here, n=8, M =256.
Hence, word size required for single stage= 8+$\log_2 256$ = 16
Word size required for 4 stages = 8+4$\log_2 256$ = 40.

From example 18.3 it may be noted that for very sampling rates and high undersampling factors, speed of the adders may become the bottleneck in the system. The conventional adders such as ripple carry adders become unsuitable for such applications. Adders using residue number systems where no carry propagation is required from LSB to MSB is preferred for these applications.

Arithmetic Using Residue Number System(RNS) Any integer less than M can be represented using an L digit number in the residue number system. Let M be represented as a product of L integers

m_1, m_2, … m_L which are relatively prime. In other words $M = \prod_{i=1}^{L} m_i$. Any number X < M can be represented in RNS form as X_L … X_2 X_1 where X_k = Xmod(m_k).

The rules for performing three primitive operations of addition, subtraction and multiplication on numbers represented in RNS arithmetic are the same:

If □ denotes any one of the primitive operators, for any two integers X and Y which are less than M, Z = (X□Y)modM = Z_L … Z_2 Z_1 where Z_k = (X_k□Y_k)mod(m_k).

Example 18.4 Perform 53+ 45 using RNS arithmetic. Let us choose a three digit representation where M = 5X7X8=280= $m_3m_2m_1$.

X = X_3X_2 X_1 = 53mod5 53mod7 53mod8 = 345

Similarly, Y= Y_3Y_2 Y_1= 035

Z= Z_3Z_2 Z_1 = (3+0)mod5 (4+3)mod7 (5+5)mod8 = 302

Representation of 98 (53+45) in RNS is 302. Hence the result is correct.

Example 18.4 Perform 13X15 using RNS arithmetic. Let us choose the three digit representation where M = 5X7X8=280= $m_3m_2m_1$.

X = X_3X_2 X_1 = 13mod5 13mod7 13mod8 = 365

Similarly, Y= Y_3Y_2 Y_1= 017

Z= Z_3Z_2 Z_1 = (3X0)mod5 (6X1)mod7 (5X7)mod8 = 063

Representation of 195 (13X15) in RNS is 063. Hence the result is correct.

Addition and multiplication operations required for each digit of RNS in examples 18.4 and 18.5 can be carried out in parallel. Moreover, these operations can be carried using ROM or look up tables and hence these operations can be performed fast.

CASE STUDY OF AN FPGA BASED DIGITAL RECEIVER 18.4

In this section, the design and implementation details of a reconfigurable spread spectrum communication receiver is presented.

18.4.1 Introduction to spread spectrum system

Spread spectrum is a technique whereby the bandwidth of a signal is spread such that it has the least probability of intercept and has the least interference with the signals operating in the same frequency band. Some of the benefits that can accrue simultaneously by spreading the spectrum are Anti-jamming, Anti-interference, Low probability of intercept, Multiple user random access communications with selective addressing capability and

High resolution ranging. More details of the spread spectrum communication system may be found in [Raymond82], [Dixon76]. Details of implementation of spread spectrum systems on FPGA or combination of FPGA and DSP have been already reported in the literature [Cong 2001] [Yanxin2001]. Design and FPGA implementation of a chaotic frequency hopping (FH) sequence generator for asynchronous CDMA system is considered in [Cong 2001]. A hybrid system consisting of both FPGA and TI DSP is used for implementing the spread spectrum system in [Yanxin2001]. In this paper, baseband coding and disordering of data is done on the FPGA and the operating frequency of 5 MHz . In this section, a scheme which uses a single FPGA for modulation/demodulation, frequency hopping, synchronization and frequency synthesis is presented.

Among various spreading techniques [Dixon76], frequency hopped spread spectrum system (FHSS) is chosen for our implementation. In FHSS, the frequency of the carrier signal is hopped from one value to another using a pseudorandom sequence known to the transmitter and intended receiver only. FHSS requires less complex circuitry and consumes less power.

Depending upon the rate at which the hops occur, there are two basic characterizations of frequency hopping:

1. Slow – frequency hopping, in which the symbol rate is an integer multiple of the hop rate.
2. Fast – frequency hopping, in which the hop rate is an integer multiple of the symbol rate.

In order to reduce the complexity of the receiver, slow frequency hopping system is chosen for the implementation.

18.4.2 Synchronization of the Transmitter and Receiver

In spread-spectrum systems, the transmitter and receiver use identical PRBS sequence for hopping the carrier frequency. A synchronizer is required at the receiver to allow the regeneration of the duplicate of the chipping waveform used at the transmitter.

The block diagram of synchronization circuit for the all digital FHSS system is shown in Fig.18.16. It consists of an ADC, a mixer for down conversion, a narrow band pass filter, a peak detector, a comparator and a numeric controlled oscillator (NCO). The BPF has a centre frequency of f_o and a bandwidth equal to twice the hopping rate ($B = 2f_h$). The NCO consists of a pseudo random binary sequence (PRBS) generator and direct digital frequency synthesizer (DDFS). Frequency of the DDFS is controlled by the output of the PRBS generator. All the blocks in the synchronizer excepting ADC are implemented using the FPGA. The clock input to the PRBS generator is enabled by the comparator output denoted as ON/OFF control. When it is ON, clock pulses at the hopping rate f_h are delivered to the PRBS generator. The PRBS generators and DDFS at the transmitter and receiver are identical.

The synchronization is achieved as follows: Let the receiver synthesizer generate a carrier of frequency $f_o+f_{j,}$ while that of the transmitter is $f_k \neq f_j$. The mixer output i.e., the difference frequency $f_o +(f_j - f_k)$ is outside the passband of the band pass filter. Hence, the peak detector output becomes zero and the output of comparator is in OFF state. As long as the received frequency is not equal to f_j, the PRBS generator has a fixed output and the receiver synthesizer remains camped at $f_o + f_j$.

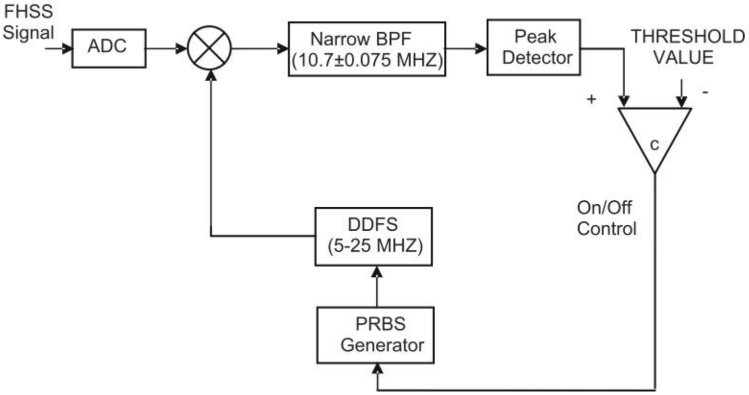


Fig. 18.22 *Clamp and wait synchronization scheme*

When the carrier frequency of received signal becomes f_j, the difference signal lies within the passband of the BPF and the peak detector output rises above the threshold. The comparator output changes to 1 (ON state) and the numeric controlled oscillator is turned on. This enables the receiver PRBS generator to advance through its states in synchronism with the transmitter generator.

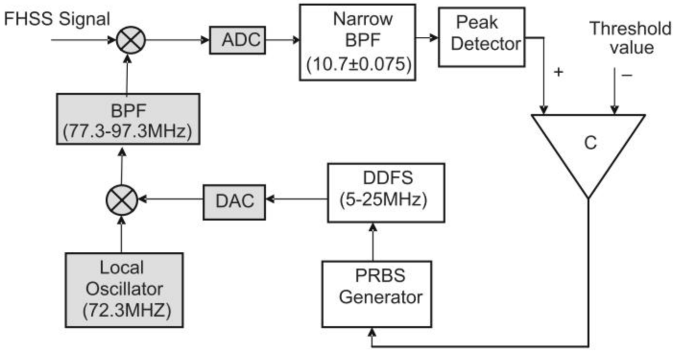


Fig. 18.23 *Synchronization scheme with analog interface*

In a practical system, the carrier frequencies required for the FHSS system is very much above what can be generated using DDFS. Hence, an analog down-conversion scheme is required. The synchronization circuit with analog down conversion scheme is shown in Fig. 18.23.

In Fig. 18.23, all the analog components are indicated by shaded blocks. Once the synchronization is achieved, the incoming FM/FHSS signal, in the range 88-108MHz is down-converted to 10.7MHz (standard IF frequency). The down-converted signal is fed to digital FM Demodulator, which retrieves the message signal.

18.4.3 Block Diagram of FHSS Transmitter and Receiver

The FHSS system is designed to operate in the commercial FM band of 88MHz-108MHz. It is difficult to generate the carrier signal in this band directly using FPGA due to their speed limitations. Hence, the FPGA is used to generate an FHSS signal in the band 5-25 MHz using CORDIC algorithm. This signal is upconverted to the FM band using the analog blocks as shown in Fig. 18.24.

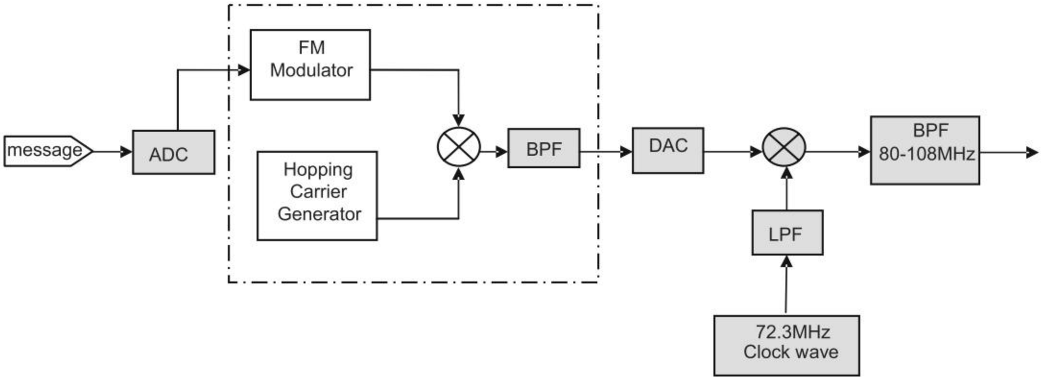


Fig. 18.24 *Frequency modulated and frequency hopped Transmitter*

The receiver consists of an analog mixer which down converts the incoming signal in the band 88-108MHz to 10.7MHz. It also consists of synchronization loop which is described in section 18.4.2. As soon as synchronization is achieved, the FM demodulator is activated which uses the 10.7MHz IF signal to generate message output. Figure 18.25 shows complete architecture for the receiver.

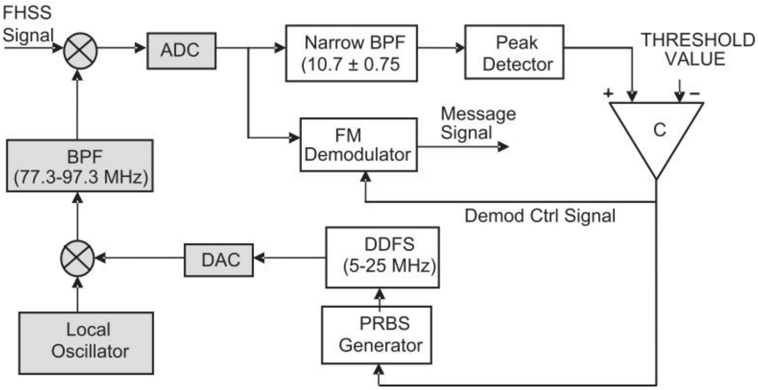


Fig. 18.25 *Block diagram of FHSS Receiver*

18.4.4 Implementation Details

The FHSS transmitter is implemented on Xilinx® Virtex-4® DSP development board using Xilinx® ISE®8.1i and receiver is implemented on Altera® Stratix-II ® DSP development board using Quartus II® 6.0.This has been done to show the compatibility of design across the various FPGA families.

For generating frequency modulated signal, amplitude of carrier signal is fixed and the message signal is given to $\Delta_{r.}$ (see Fig.18.9). To generate a smooth carrier signal of frequency 10.7 MHz, the clock to DDFS is chosen as 200 MHz. Frequency modulator is designed so as to get a peak to peak frequency deviation of less than 150 KHz which is the standard frequency deviation used in FM systems. In order to hop as well as to up-convert the FM signal, DDFS and PRBS Generator are connected to the analog blocks as shown in Fig.18.22. The total number of channels used for hopping and the bandwidth of each channel are chosen to be 31 and 645 KHz respectively. The outputs of FM modulator and FHSS transmitter are as shown in Fig. 18.25 and Fig. 18.26 respectively. These waveforms are obtained using Xilinx SignalTap II® Embedded Logic Analyzer.

Due to the delay incurred in the synchronization loop at the receiver, the time duration for each state of PRBS generator is chosen as 10micro seconds, i.e., the clock frequency for the LFSR is set as 100 KHz. The frequency synthesizer output varies from 5 to 25 MHz depending upon the PRBS output.

For the frequency modulator, the Intermediate Frequency (IF) is chosen to be 10.7MHz. In order to simplify the receiver, ADC is sampled at the rate of 4 X 10.7MHz i.e., 42.8MHz. The bandpass filters are implemented using IP cores from the Altera® and Xilinx®. The narrow bandpass filter in Fig. 18.17 is centered at an IF frequency of 10.7 MHz with band width equal to twice the frequency deviation i.e., from 10.625 – 10.775 MHz. The PRBS generator clamps at a given seed value until it receives the corresponding frequency from the transmitter. The digital peak detector detects the peak value of the BPF output, which is used to compare with a threshold value. When the peak value exceeds the threshold value, the comparator makes the on/off signal to be high which indicates that the receiver PRBS generator and transmitter PRBS generator are synchronous to each other. The control signal is used as the enable signal for FM Demodulator. Hence, FM Demodulator starts working only when synchronization is finished. The output at FM demodulator is shown in Fig. 18.25. The output is captured and displayed using Chipscope Pro®8.1i. It has been verified that receiver and transmitter are operating in synchronization with each other.

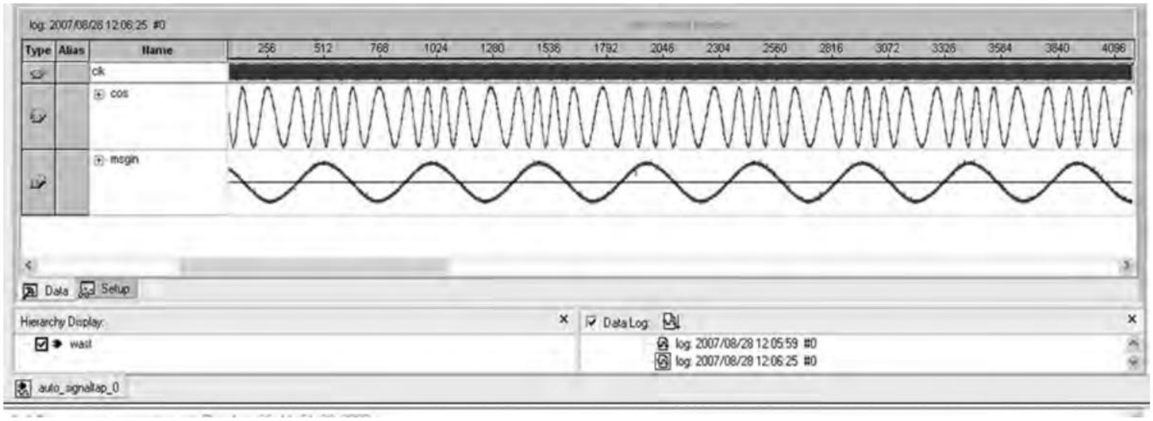
Table 18.6 *FPGA Resources required for the Transmitter*

Family	*Virtex-4*
Device	XC4VSX35F668-10
4-I/P LUT	5,742
Slice Flip Flops	4,578
DCM	1
DSP48s	37

Table 18.7 *FPGA Resources required for the Receiver*

Family	*Stratix II*
Device	EP2S60F1020C4
ALUT	3,712
Registers	3444
PLL	1
Embedded DSP blocks	2

The FPGA at the receiver and transmitter side is interfaced with the suitable analog interface. For the transmitter, DAC on the Virtex-4 board can work upto 160MSPS and hence is used for the FHSS transmitter. *ADC on the Stratix II board can operate up to 1*25MSPS and hence is used for the FHSS receiver. Eventhough we have used the frequency modulation for the FHSS system, other modulation scemes such as FSK, PSK, QPSK can be easily implemented as the modulation type is determined based on the point at which the input is applied to the CORDIC block. The waveform obtained for the BPSK modulation is shown in the Fig.18.26. The summaries of the resources required for the transmitter and receiver are given in Table 18.6 is given in Table 18.7.



log: 2007/08/28 12:06:25 #0

log: 2007/08/28 12:05:59 #0
log: 2007/08/28 12:06:25 #0

Fig. 18.26 *Digital FM Generation for 100KHz signal –Signal Tap II Logic Analyzer output (channel 1.ADC clock, Channel 2.Digital Frequency modulated wave. Channel .3. message signal)*

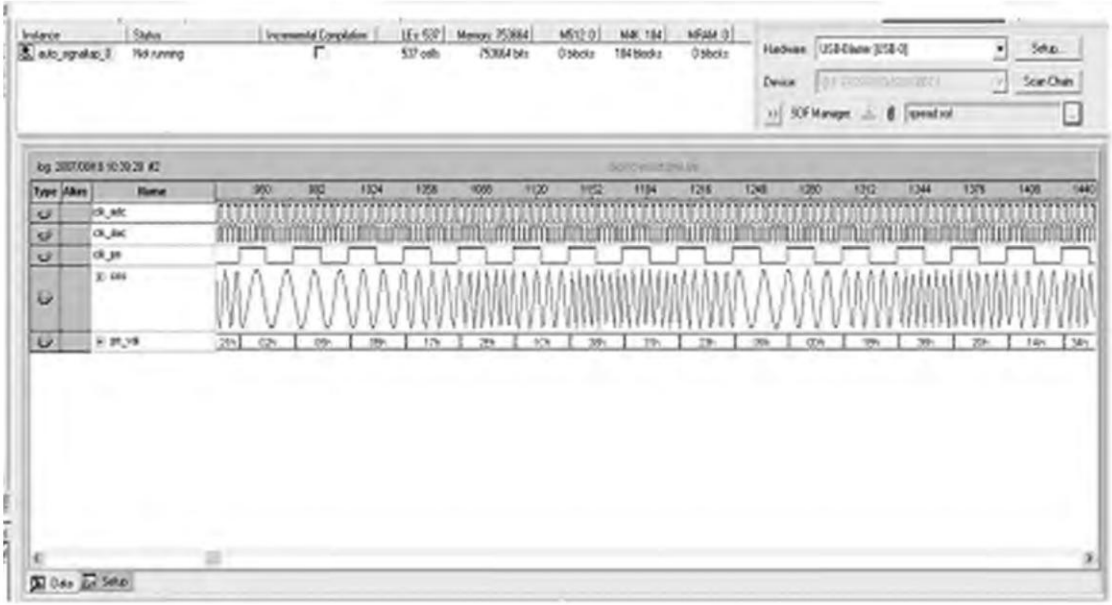


Fig. 18.27 *Digital FHSS Generation –Signal Tap II Logic Analyzer output (channel 1.ADC clock, Channel 2. DAC clock Channel 3.Digital Frequency hopped wave. Channel .4. PN generator)*

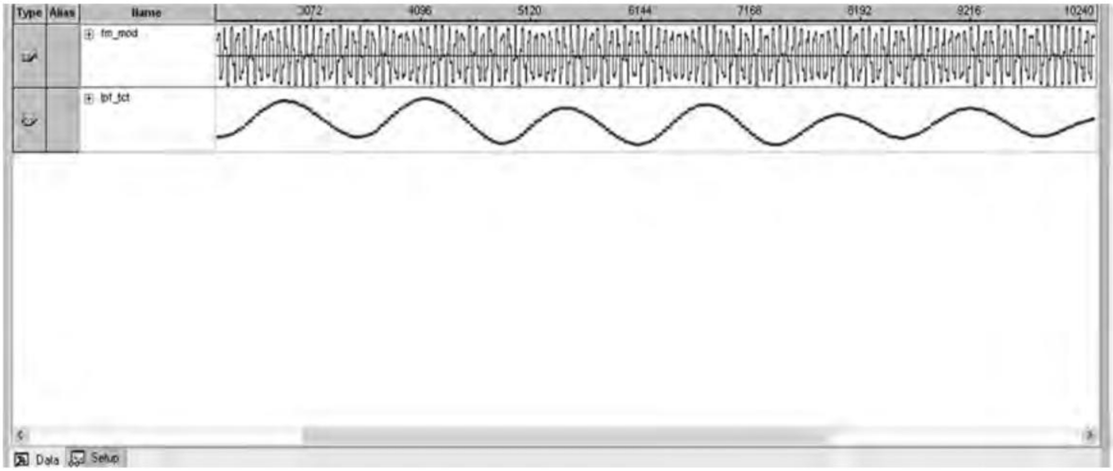


Fig. 18.28 *Digital FM demodulation (Channel 1. FM signal channel.2 100KHz FM demodulated signal)*

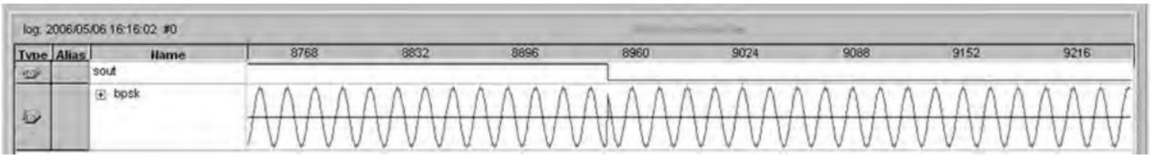


Fig. 18.29 *BPSK modulation using CORDIC*

Review Questions

18.1 Draw the block diagram of 1st generation and 2nd generation transmitter and explain its operation.

18.2 Draw the block diagram of 1st generation and 2nd generation receiver and explain its operation.

18.3 Draw the block diagram of a digital transceiver for wireless LAN and and explain its operation.

18.4 List the important characteristics of software defined radio.

18.5 What are the bottlenecks in realizing an ideal software defined radio.

18.6 Discuss the relative merits and demerits of analog frequency synthesis over DDFS.

18.7 Compare the digital and analog methods for carrier generation .

18.8 Explain a technique for DDFS. Explain how the frequency resolution and dynamic range of the output signal can be independently selected.

18.9 Explain how the following with reference to DDFS with phase accumulator and ROM (i) frequency resolution (ii) ROM Size (iii) phase jitter (iv) spurious frequency components

18.10 A DDFS scheme employing phase accumulator method uses 32 bit accumulator and has the input clock frequency as 100MHz. Find the minimum and maximum frequencies of sine wave which can be generated.

18.11 What is the size of the ROM required for the phase accumulator method for DDFS if the signal in the range 1 KHz to 1 GHz is to be generated? Explain how the no. of locations required can be reduced by a factor of 256. How does this affect of the shape of the waveform generated? How can this effect be minimized?

18.12 A DDFS scheme employing phase accumulator method uses 10 bit accumulator and has the input clock frequency as 200MHz. Find the frequency resolution. Find the lowest frequency of DDFS for which periodic jitter occurs.

18.13 A DDFS scheme employing phase accumulator method has frequency resolution of 1 Hz and the input clock frequency as 100MHz. Find the size of the accumulator and ROM required.

18.14 Write the expression relating the spurious frequency components to the no. of bits of phase accumulator used as address to look up table.

18.15 What is meant by spur (spurious components)? Explain an example when these components are generated by the carrier generation scheme.

18.16 Explain anyone technique for DDFS ROM data compression

18.17 Explain the Sunderland technique for DDFS ROM data compression

18.18 List any three techniques for ROM data compression for DDFS.

18.19 Write the equations used for computing the rectangular coordinates of a vector iteratively with CORDIC algorithm , given its initial position and the angle through which it is to be rotated. Explain how this technique can be used for generating the sin of an angle.

18.20 Explain how CORDIC may be used as universal demodulator. Explain how CORDIC may be used for generation of (i)AM (ii) BPSK.

18.21 Compare the DDFS with ROM approach and DDFS using CORDIC approach with regard to the highest frequency which can be generated. Explain the reason for your observation.

18.22 What is the test used for terminating the iterations in the case of CORDIC scheme operating in (a) rotation mode (b) vector mode . How is the angle used for each rotation chosen in the case of CORDIC? When is the angle considered to be positive?

18.23 Show how BPSK signal can be generated using DDFS with ROM approach. Compare the universal modulator implemented using DDFS with ROM approach and that using CORDIC.

18.24 Explain how CORDIC algorithm can be used for noncoherent demodulation of FM signal. What should be the initial value of "z_i" for this application?

18.25 Draw the block diagram of unrolled CORDIC scheme. What are its advantages over rolled CORDIC scheme?

18.26 Explain how QPSK signal can be generated using CORDIC approach.

18.27 Compare the relative merits of unrolled and rolled CORDIC

18.28 Explain how the amplitude modulated signal can be demodulated using CORDIC.

18.29 Explain how CORDIC may be used to generate the carrier required for coherent demodulation.

18.30 Explain the special sampling scheme for down conversion of both narrow band and wide band signals.

18.31 State the advantages of the following (i) special sampling (ii) down sampling.

18.32 What are the properties of digital Hilbert transformer? What are the advantages of generating I & Q channel using Hilbert transform?

18.33 Draw the block diagram of a 1st order digital Hilbert transformer with symmetric filter coefficients for a narrow band signal.

18.34 What is meant by undersampling receiver? When is it applicable?

18.35 What are advantages of digital technique for I/Q channel generation over analog techniques?

18.36 Sketch the frequency response of a Hilbert transformer if it is fed with a DSB signal? What is the nature of impulse response coefficients of a Hilbert transformer?

18.37 Explain how the special sampling scheme can be used for down conversion of a signal with unknown input frequency.

18.38 In the FPGA based radio/baseband processor, how a 128 tap filter with sampling rate of 200 MSPS may be realized? How a A/D converter and DDR interface in the Virtex II Pro may be used to downconvert 1GHz signal to 200 MHz?

18.39 Explain how a modulated signal with carrier frequency of 1 GHz can be demodulated using downsampling scheme and DDR interface in FPGAs.

18.40 An AM signal with carrier frequency of 1 MHz and BW 10 KHz is downconverted using special sampling scheme and demodulated using CORDIC. Explain how the hardware required for the implementation is minimized by this approach.

18.41 What is the transfer function of a VCO?

18.42 Compare the magnitude of the control voltage obtained in VCO with that obtained using costas loop. Justify your answer.

18.43 Compare the two schemes for coherent demodulation of BPSK.

18.44 What is meant by CIC filter?

18.45 Write the transfer function of a CIC filter used to realize an M tap boxcar filter.

18.46 Explain why multistage CIC filters are preferred for implementation of high speed low pass filters?

18.47 Write the transfer function of a low pass filter realized using K stages M tap boxcar filter. The boxcar filter is realized using CIC filter.

18.48 What is meant by Hogenauer filter? What are its advantages?

18.49 Draw the block diagram of Hogenauer filter if a K stage CIC filter is succeeded by a decimator by a factor of L.

18.50 Draw the block diagram of Hogenauer filter if a K stage CIC filter is preceded by an interpolator by a factor of L.

18.51 State any two applications of CIC filter.

18.52 Explain how a moving average filter with 64 taps can be realized as CIC filter. If the input to this filter are 12 bit wide, what should be the size of the adders if the overflow is to be avoided. If 8 such moving average filters are cascaded to improve the performance of the filter, what should be the size of the adders for no overflow.

18.53 What is the advantage of residue number system? Show how the addition of two numbers 34_D & 54_D is carried out using a 4 digit RNS. Assume the largest number to be represented as 199_D.

18.54 Explain the adder using RNS with an example. Explain why RNS is preferred for high speed applications

18.55 Show how the multiplication of two numbers 25_D & 20_D is carried out using a 3 digit RNS.

18.56 Explain how ROM can be used to perform RNS arithmetic. Illustrate this technique by considering the addition of two three digit RNS numbers.

18.57 State the Chinese remainder theorem.

18.58 Explain how an RNS number can be converted to a decimal number using CRT.

18.59 A matched filter with 64 taps is to be implemented using (i) FPGA with 128 dedicated multipliers (ii) Programmable DSP with 4 MAC units. The maximum clock frequency used for both FPGA and P-DSP is 200MHz. The input signal is sampled at the rate of 200 MSPs. Compare the hardware complexity for both the approaches.

18.60 Explain why slow frequency hopping is used for the digital implementation of FHSS.

18.61 Draw the block diagram of the synchronizer for the all digital FHSS system and explain its operation.

18.62 Explain why the digital FHSS requires analog circuits at the transmitter and receiver.

18.63 Draw the block diagram of digital FHSS (a) transmitter (b) receiver

ANSWERS TO SELECTED QUESTIONS

Chapter 1

Review Questions

1.5 $b = 1/a$

1.7 $h(n) = (\sin 2\pi f_c n)/\pi n$

1.8 $h(n) = (\sin \omega c n)/\pi n$

1.10 $f_s = 2500$ Hz; $fc = fs/2 = 1250$ Hz. Output signal :$\sin 2\pi f(750)t$.

1.11 $f_s = 500$ samples/sec; $f_{c1} = 1000$ Hz, $f_{c2} = 1250$Hz;
output = $\sin 2\pi(1000)t$ for $f_s = 2000$ samples/sec

1.12 $f_s = 5000$ samples/sec; $f_c = 2500$Hz

1.13 $0.2\pi \le \omega \le 0.5\pi$,
$|H(e^{j\omega})| = 0$ for $0.2\pi \le \omega \le 0.5\pi$, and 1 otherwise

1.14 $h(n) = 0.5^n + (-0.5)^n$

1.15 $h(n) = 1.5[3(0.75)^n - (0.25)^n] - 0.5\,[3(0.75)^{n-2} - (0.25)^{n-2}]$

1.16 $H(z) = \dfrac{1-(az)^{-16}}{1-(az)^{-1}}$

1.18 (a) (i) $\dfrac{1+0.25z^{-1}}{1-0.5z^{-1}}$ (ii) $\dfrac{1+0.5z^{-1}}{1-0.75z^{-1}}$

(b) $\dfrac{(1+0.25z^{-1})(1+0.5z^{-1})}{(1-0.5^{-1})(1-0.75z^{-1})}$

(c) same as that for (b)

(d) same as that for (b)

(e) $\dfrac{(1+0.25z^{-1})}{(1-0.5^{-1})} + \dfrac{(1+0.5z^{-1})}{(1-0.75z^{-1})}$

1.19 $X(k) = e^{-j(2\pi k/3)}\,[1 + 2\cos(2\pi k)/3)]$ for $k = 0,1,2$

1.20 $X(k) = e^{-j(2\pi k/8)}\,[1 + 2\cos(2\pi k)/8)]$ for $k = 0,1, \ldots 7$.

1.21 $X(k) = [1 - 0.5e^{-j(2\pi k/8)}]^{-1}$ for $k = 0, 1, 2 \ldots 7$.

1.23 $e^{-j(2\pi\,\mathrm{mn/N})}$

1.24 (a) (i) 0.5π, (ii) 2 (b) (i) 5000π, (ii) 20,000

1.27 (a) $h(t) = u(t)e^{0.5t}$

$h(n) = h(nT) = u(n)e^{0.5nT}$ where $u(n) = l$ for $n \ge 0$

$H(z) = [1 - e^{0.5T} z^{-1}]^{-1}$

1.28 (a) $H(z) = \dfrac{T}{(2 - aT)}\dfrac{(1 + z^{-1})}{(1 - cz^{-1})}$ where $c = \dfrac{(2 + aT)}{(2 - aT)}$

(b) $h(n) = [T/(2 - aT)]\,(1 + c)c^{n-1}$

1.30 $h(n) = -(\sin .75\pi n)/\pi n$; $h_a(n) = h(n)w(n)$, $N = 8$

Self-Test Questions

1.1 c	1.2 a,b,c	1.3 c	1.4 a	1.5 d
1.6 a	1.7 b	1.8 c	1.9 b, d	1.10 a, c
1.11 a	1.12 b	1.13 b, c, d	1.14 b	1.15 c
1.16 a	1.17 d	1.18 b	1.19 d	1.20 a
1.21 1				

Chapter 2

Self-Test Questions

2.1 d	2.2 program, data	2.3 c	2.4 b	2.5 d
2.6 d	2.7 c	2.8 d	2.9 c	2.10 b

Chapter 3

Self-Test Questions

3.1 a,b	3.2 a	3.3 a	3.4 a	3.5 b
3.6 b	3.7 c	3.8 a	3.9 c	3.10 a
3.11 b	3.12 a	3.13 a	3.14 a	3.15 b
3.16 c	3.17 d	3.18 e	3.19 a	3.20 b
3.21 c	3.22 d	3.23 a, b	3.24 c	3.25 e

Chapter 4

Self-Test Questions

4.1 a	4.2 a	4.3 b	4.4 b	4.5 b
4.6 a	4.7 b	4.8 a	4.9 b	4.10 c
4.11 a	4.12 d	4.13 b	4.14 b	4.15 c
4.16 a	4.17 c	4.18 b,c	4.19 d	4.20 a
4.21 b	4.22 d	4.23 b	4.24 c	4.25 d
4.26 a	4.27 c	4.28 d	4.29 b	4.30 a

Chapter 5

Self-Test Questions

5.1	a	5.2	c	5.3	c	5.4	c	5.5	b
5.6	b, d	5.7	b	5.8	d	5.9	b	5.10	c

Chapter 6

Self-Test Questions

6.1	a	6.2	a	6.3	c	6.4	d	6.5	a
6.6	e	6.7	c	6.8	c	6.9	b	6.10	a
6.11	b	6.12	b	6.13	e	6.14	e,f	6.15	e,f
6.16	b	6.17	f	6.18	c,f	6.19	b	6.20	d
6.21	d	6.22	a	6.23	c	6.24	b	6.25	b

Chapter 7

Self-Test Questions

7.1	d	7.2	c	7.3	a	7.4	c	7.5	b
7.6	b	7.7	c	7.8	c	7.9	d	7.10	a
7.11	c	7.12	c	7.13	b	7.14	b	7.15	b
7.16	c	7.17	a	7.18	a	7.19	b	7.20	a
7.21	b	7.22	b	7.23	b	7.24	a	7.25	a
7.26	a	7.27	b	7.28	a	7.29	d	7.30	b

Chapter 8

Self-Test Questions

8.1	c	8.2	c	8.3	b	8.4	c	8.5	b
8.6	c	8.7	b	8.8	c	8.9	b	8.10	c
8.11	c	8.12	a	8.13	c	8.14	c	8.15	a
8.16	d	8.17	a	8.18	b	8.19	c	8.20	c
8.21	b	8.22	c	8.23	b	8.24	c	8.25	b

Chapter 9

Self-Test Questions

9.1	b	9.2	d	9.3	c	9.4	b	9.5	c
9.6	b	9.7	d	9.8	d	9.9	b	9.10	c
9.11	e	9.12	c	9.13	a	9.14	c	9.15	d
9.16	a	9.17	c	9.18	a	9.19	a	9.20	a
9.21	b	9.22	b	9.23	a	9.24	a	9.25	d

Chapter 10

Self-Test Questions

10.1 d 10.2 a, c 10.3 c 10.4 d 10.5 c

10.6 c 10.7 c 10.8 a 10.9 c, d 10.10 c

Chapter 11

Self-Test Questions

11.1 a 11.2 b 11.3 c 11.4 a, d 11.5 b

11.6 d 11.7 a 11.8 a 11.9 a 11.10 a

11.11 a, d 11.12 ADD 40h,5, A

11.13 ADDB,8,A 11.14 SUB B, 8, A

11.15 ADD*AR2, *AR3,A 11.16 SUB *AR2, *AR3, B

11.17 ADD A, ASM,B 11.18 SUB A, ASM 11.19 c

11.20 a 11.21 b 11.22 a 11.23 b

11.24 MACP*AR3,1120h,A 11.25 MACD *AR3,1120h,A

11.26 a 11.27 a 11.28 b, c 11.29 a 11.30 a

11.31 a 11.32 BANZ 1050h, AR5+

11.33 BANZ 1050h, *+AR5 11.34 a 11.35 c

Chapter 12

Self-Test Questions

12.1 a, c 12.2 c, d 12.3 a, b 12.4 b, c 12.5 a, d

12.6 a 12.7 a 12.8 d 12.9 d 12.10 b

12.11 a 12.12 d 12.13 c

12.14

(a) 0 (b) 1 (c) 2 (d) 3 (e) 0

(f) 1 (g) 0 (h) 1 (i) 3 (J) 3

(k) 3 (1) 3

Chapter 13

Self-Test Questions

13.1 c 13.2 b 13.3 c 13.4 b 13.5 c

13.6 a 13.7 b 13.8 d 13.9 c 13.10 d

13.11 a 13.12 a,b,d 13.13 b 13.14 b 13.15 d

13.16 d 13.17 a 13.18 a 13.19 c 13.20 b

13.21 d 13.22 a 13.23 d 13.24 c 13.25 a

13.26 d

Chapter 14

Self-Test Questions

14.1 d	14.2 d	14.3 d	14.4 a	14.5 c
14.6 c	14.7 d	14.8 b	14.9 c	14.10 d
14.11 c	14.12 d	14.13 c	14.14 b	14.15 d
14.16 b	14.17 d	14.18 d	14.19 d	14.20 c
14.21 d				

Chapter 15

Self-Test Questions

15.1 b	15.2 c	15.3 c	15.4 d	15.5 b
15.6 a	15.7 c	15.8 d	15.9 c	15.10 c
15.11 c	15.12 a	15.13 a	15.14 c	15.15 c
15.16 c	15.17 d	15.18 b	15.19 b	15.20 a
15.21 b	15.22 c	15.23 c	15.24 c	15.25 ????
15.26 d				

Chapter 16

Self-Test Questions

16.1 a	16.2 c	16.3 b	16.4 a	16.5 b
16.6 c	16.7 a	16.8 d	16.9 b	16.10 d
16.11 c	16.12 c	16.13 b	16.14 c	16.15 c
16.16 a	16.17 b	16.18 a	16.19 b	16.20 b
16.21 b	16.22 b	16.23 a	16.24 c	16.25 b
16.26 c	16.27 b	16.28 d	16.29 c	

BIBLIOGRAPHY

Allen, J., “Computer Architectures for Digital Signal Processing”, *Proceedings of the IEEE,* Vol. 73, No. 5, May, 1985.

Altera Digital Library 2001, Version 1, Altera International, Hong Kong.

Amy, M., *Digital Signal Processing Applications Using the ADSP-2100 Family,* Vol. 1, Englewood Cliffs, Prentice-Hall, 1990.

Bateman A., and Yates, W., *Digital Signal Processing Design,* Computer Science Press, New York.

Bosi, B., Bois, G., and Savaria Y., “Reconfigurable Pipelined 2-D Convolvers for Fast Digital Signal Processing”, *IEEE Transaction on VLSI Systems,* Vol. 7, pp 299-308, Sept. 1999.

Bruce Jorgens*, “*The new Virtex-E 3.2-Million Gate, *High-bandwidth* FPGA Family” ,Xcell Journal, Xilinx, CA, USA, Issue 34, pp 5-9, 1999.

Canet, M.J. Vicedo, F. Almenar, V. Valls, J. Dpto. “FPGA implementation of an IF transceiver for OFDM-based WLAN”, IEEE Workshop on Signal Processing Systems, 2004. SIPS 2004.13-15, pp 227-232, Oct. 2004.

Chassaing, R., *Digital Signal Processing with C and the TMS320C30,* Wiley, New York, 1992.

Chassaing, R., *Digital Signal Processing with the TMS320C25,* Wiley, New York, 1990.

Chassaing, R., *Digital Signal Processing: Laboratory Experiments Using C and the TMS320C3I DSK,* Wiley, New York 1998.

Chris Dick and Fred Harris, “Implementing narrow-band FIR filters using FPGAs”, IEEE International Symposium on Circuits and Systems. ISCAS ’96., Vol.2, 289-292, 1996.

Chris Dick, Fred Harris, Michael Rice, “FPGA Implementation of Carrier Synchronization for QAM Receivers”, Journal of VLSI Signal Processing 36, 57–71, Kluwer Academic Publishers, 2004.

Chris Dick, Harris, F.J., “Configurable logic for digital communications: some signal processing perspectives”, IEEE Communications Magazine , Volume 37, Issue 8, pp 107-111, Aug. 1999.

Code Composer, User’s Guide, Texas Instruments, Texas, 1999.

Crochiere, R. E., and Rabiner, L. R., *Multirate Signal Processing,* Englewood Cliffs, NJ, Prentice-Hall, 1983.

Cyclone III Device Handbook, Volume 1, Altera San Jose, CA, USA, July, 2009.

Dahnoun, N., *Digital Signal Processing Implementation using the "TMS320C6000(tm) DSP Platform,* Prentice-Hall-Pearson Education Ltd, 2000.

Data Source CDROM, Xilinx, Rev. 2, Fourth quarter, 2000.

El-Sharkawy, *Digital Signal Processing Applications with Motorola's DSP56002 Processor,* Englewood Cliffs, NJ, Prentice-Hall, 1997.

Haddad, R. A., and Parsons T. W., *Digital Signal Processing,* Computer Science Press, New York, 1991.

Hans-Peter Rosinger, "Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel", Xilinx Application Notes XAPP529, May, 2004.

Ifeachor, E. C, and Jervis, B. W., *Digital Signal Processing: A practical approach,* Addison Wesley, 1993.

Jain, A. K., *Digital Image Processing,* Prentice-Hall of India, New Delhi, 1985.

James E. Gunn, Kenneth S. Barron, and William Ruczczyk, "A Low-Power DSP Core-Based."

James Tsui, "Digital Techniques for wide band transmission", Artech house publishers, (1995).

Jamil Chaoui, "OMAP™ : Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals", Dedicated Systems Magazine pp 34-39 Q2, 2001.

Jean Duprat, Jean-Michel Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation. IEEE Trans. Computers 42(2): 168-178 (1993).

Jere, M, *Digital Signal Processing Applications Using the ADSP-2100 Family,* Vol. 2, Prentice-Hall, Englewood Cliffs, NJ, 1995.

Kehtarnavaz N. and Simsek, B., *C6X-Based Digital Signal Processing,* Prentice-Hall, Eaglewood Cliffs, NJ, 1990.

Kehtarnavaz N. and Keramat, M., *DSP System Design Using the TMS320C6000,* Prentice-Hall, Eaglewood Cliffs NJ, 2000.

Lakshminarayanan, G., Venkataramani, B., Senthilkumar, K. P, Sasitharan, M., and Kottapalli, V. A. K., "Design and implementation of a FPGA based wavepipelined fast convolver", *Proceedings of the IEEE Conference TENCON 2000,* Kuala Lumpur, Malaysia, Sept. 24-27, Vol. III, pp 212-217, 2000.

Lapsley, P., Bier, J., Shoham, A., and Shoham A. Lee, E., *DSP Processor Fundamentals,* S. Chand & Company, New Delhi, 1997.

Lee, E. A., *Programmable DSP Architectures,* Part I, IEEE ASSP Magazine, Oct. 1988.

Lee, E. A., *Programmable DSP Architectures,* Part II, IEEE ASSP Magazine, Jan. 1989.

Leopold E. Pellon, "A Double Nyquist Digital Product Detector for Quadrature Sampling" IEEE transactions on signal processing. vol. 40, NO. 7. pp 1670-1681, July, 1992.

Lin, K. S., *Digital Signal Processing with TMS320 Family,* Vol. I, Prentice-Hall, Englewood Cliffs, NJ, 1987.

Ling Cong, and Wu Xiaofu, "Design and Realization of an FPGA-Based Generator for Chaotic Frequency Hopping Sequences" IEEE Transactions On Circuits And Systems—I: Fundamental Theory And Applications, Vol. 48, No. 5, May, 2001.

Lionel Cordesses, "Direct Digital Synthesis: A Tool for Periodic Wave Generation", IEEE Signal Processing Magazine, pp 50-54, July, 2004.

Marven, C, and Ewers, G., *A Simple Approach to Digital Signal Processing,* Wiley Interscience, 1996.

McClellan, Schafer, J. R., Yoder, M, *DSP First: A Multimedia Approach,* Prentice-Hall, New York, 1998.

MicroBlaze Processor Reference Guide: EDK 10.1i UG081, Xilinx, CA, USA

N. Kehtarnavaz and M. Keramat, DSP System Design: Using the TMS320C6000, Printice Hall, 2001.

Oppenheim, A. V, and Schafer, R. W., *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

Panos, P. (ed.), *Digital Signal Processing with the TMS320 Family,* Vol. 3, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Parhi, K. K., *VLSI Digital Signal Processing Systems,* Wiley, New York, 1999.

Pirsch, P., *Architectures for Digital Signal Processing,* Wiley, New York, 1998.

R. C. Dixon, Spread Spectrum Systems. New York: Wiley, 1976.

Rabiner, L., and Gold, *Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

Rabiner, L., and Juang, B., *Fundamentals of Speech Recognition,* Prentice-Hall. Englewood Cliffs, NJ, 1993.

Ray Andraka, "A survey of cordic algorithms for FPGA based computers", International symposium on Field Programmable Gate Arrays" Proceedings, pp. 191-200, 1998.

Raymond L. Pickholtz, Donald L. Schilling, and Laurence B. Milstein, "Theory of Spread Spectrum Communications –A Tutorial" IEEE Transactions On Communications, Vol. 5, May, 1982

S.K.Mitra, Digital Signal Processing (3/e), Tata McGraw Hill, 2006.

S.M. Kuo and B.H. Lee, Real-Time Digital Signal Processing, Implementations, Applications and Experiments with the TMS320C55X, John Wiley and Sons, 2001.

Smith, *Application Specific Integrated Circuits,* Addison Wesley, Longman, Singapore, 1999.

Software Radio Architecture," IEEE Journal On Selected Areas In Communications, VOL. 17, NO. 4, April, 1999.

Sorensen, H. V., and Chen, J., *A Digital Signal Processing Laboratory Using the TMS320C30,* Prentice-Hall, Englewood cliffs, NJ, 1997.

Spartan-3 FPGA Family data sheet, DS099, Xilinx, CA, USA Dec. 2009.

Stanley A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review", IEEE ASSP magazine, pp 4-19, July, 1989.

Steven A. Tretter, *Communication System Design Using DSP Algorithms,* Plenum Publishing New York, 1995.

Swartzlander, E., *Systolic Signal Processing Systems,* Marcel Dekker, 1987.

Technology Innovations, Texas Instruments, Vol. 8, March, 2001.

TMS320 DSP Solutions CD, Texas Instruments, Feb. 1997.

TMS320 Floating-Point DSP Assembly Language Tools User's Guide, Texas Instruments, Texas, 1996.

TMS320C3X User's Guide, Texas Instruments, Texas, USA, 1996.

TMS320C5000 Digital Signal Processing Teaching Kit, Texas Instruments, Texas, 2000.

TMS320C54X Code Composer Studio Tutorial, Texas Instruments, Texas, 1999.

TMS320C54X DSP CPU and Peripheral Reference Set, Vol. I, Texas Instruments, Texas, 1999.

TMS320C54X Assembly Language Tools User's Guide, Texas Instruments, Texas, 1996.

TMS320C54X DSP Mnemonic Instruction Set Reference Set, Vol. 2, Texas Instruments, Texas, 1996.

TMS320C54X DSP Reference Set, Vol. 4: Applications Guide, 1996.

TMS320C55X DSP Assembly Language Tools User's Guide, Texas Instruments, Texas, 2004.

TMS320C55X DSP CPU Reference Guide, Texas Instruments, Texas, 2004.

TMS320C55X DSP Mnemonic Instruction Set Reference Guide, Texas Instruments, Texas, 2002.

TMS320C55X DSP Peripheral Overview Reference Guide, Texas Instruments, Texas, 2002.

TMS320C55X Technical Overview, Texas Instruments, Texas, 2000.

TMS320C5X Digital Signal Processing Teaching Kit, Texas Instruments, Texas, 1997.

TMS320C5X User's, Guide, Texas Instruments, Texas, 1996.

TMS320C5X DSP Starter Kit User's Guide, Texas Instruments, Texas, 1996.

TMS320C6000 Assembly Language Tools User's Guide, Texas Instruments, Texas, 2000.

TMS320C6000 Code Composer Studio Tutorial, Texas Instruments, Texas, 1999.

TMS320C6000 CPU and Instruction Set, Reference Guide, Texas Instruments, Texas, 2000.

TMS320C6000 Peripherals Reference Guide, Texas Instruments, Texas, 1999.

TMS320C6000, Integer Division, Application report, Texas Instruments, Texas, 2000.

Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Data Sheet, DS083, Xilinx, CA, USA Nov. 2007.

Volder, J. E. "The CORDIC trigonometric computing technique" , *IRE Trans. on Electronic Computers,* vol. EC-8, no. 3, pp. 330-4, September, 1959.

Weste, N. H. E., and Eshraghian, K., *Principles of CMOS VLSI Design,* Addison Wesley Longman, Singapore, 1999.

www.altera.com

www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf

www.xilinx.com

Xcell Journal, Xilinx, CA, USA, Issue 39, Spring 2001.

Xilinx XC4000 application notes, "The Fastest Filter in the West",xilinx/documents/apps/4000

Z. X. Li. J. G. Yanxin, "Software Radio Technology in Spread Spectrum Communication", Communication Technology Proceedings, 2000. WCC - ICCT 2000. International Conference on Volume 2 page(s): 1117-1120.

INDEX