الجمهـــورية الجزائـــــرية الديمقـــــراطية الشعـــــــبية
People's Democratic Republic of Algeria
وزارة التعـــــليم العـــــالي والبحـــث العلـــمي
Ministry of Higher Education and Scientific Research
جـــامعة محمد بوضـــياف – بالمســـــيلة
Mohamed BOUDIAF University – M'Sila

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila
1985

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila
1985

كلية التكنولوجيا
قسم الالكترونيك

Faculty of Technology
Department of Electronics

الرقم: 260/ ق.إ/ك .ت /2026

المسيلة في: 2026/04/20

شهادة إدارية

بعد الإطلاع على التقارير الايجابية الواردة من السادة الخبراء أعضاء لجنة دراسة المطبوعة الجامعية والاتية أسماؤهم:

- دارنفاد وردة — أستاذ محاضر "أ" — جامعة الإخوة منتوري - قسنطينة 1
- قرماط نوبيل — أستاذ — جامعة محمد بوضياف – المسيلة
- بختي الهادي — أستاذ — جامعة محمد بوضياف – المسيلة

صادق أعضاء اللجنـــة العلميـــة على قبول المطبوعة البيداغوجية المنجزة باللغة **الإنجليزية** مع إمكانية إتخاذها سندا في تدريس طلبة السنة الأولى ماستر تخصص أنظمة مضمنة في ميدان علوم و تكنولوجيا و أن تعتمد في أي تقييم للمسار العلمي للأستاذ المعني **بن تومي ميلود** (أستاذ محاضر قسم "أ"– جامعة محمد بوضياف - المسيلة) تحت عنوان :

"Digital Signal Processor (DSP)"

رئيس اللجنة العلمية

رئيس القسم

طباخ مصطفى

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

Mohamed BOUDIAF University

Faculty of technology

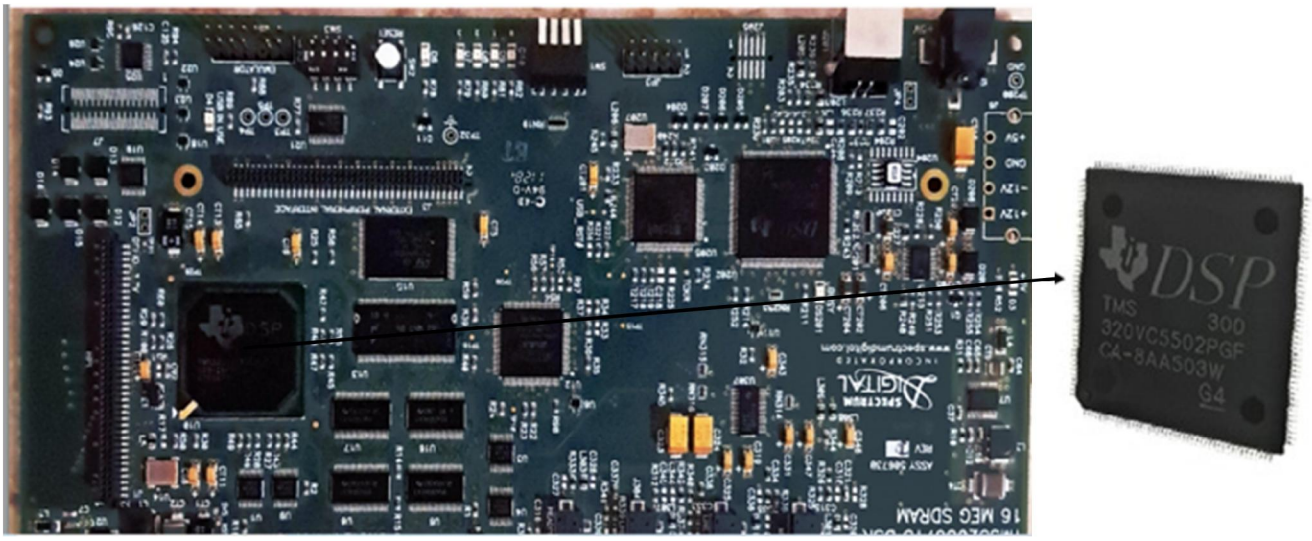
Department of electronic

Course Handouts

Digital Signal Processor (DSP)

Presented By :

Mr. **BENTOUMI Miloud**



Academic Year: 2025/2026

PREFACE

The TMS320C6000 family of DSP processors, as suggested in the second semester syllabus of the first year of the Master's program specializing in Embedded Systems Electronics, was introduced by Texas Instruments (TI) to meet high-performance requirements in signal processing applications. The objective of this course material is to provide the elements for implementation and optimization.

computationally intensive signal processing algorithms on the TMS320C6x family of DSP (Digital Signal Processors).

This course handout is written in such a way that it can be used as a textbook for DSP courses taught at many universities across the country. The presented manual is primarily written for those already familiar with DSP concepts and interested in designing DSP systems based on TI C6x DSP products. Note that much of the information in this handout appears in TI C6000 DSP family textbooks. However, this information has been restructured, modified, and condensed for use in teaching a semester-long DSP course.

Consequently, this course material can be used as a self-study guide for implementing algorithms on C6x DSPs. The chapters are organized to create a close correlation between the topics and the students if used as course materials for a DSP course. Knowledge of the C programming language is required to understand and execute the DSP mechanisms.

Table of Contents

Chapter 1: General Information on DSP Processors

1.1 Introduction

In general, sensors generate analog signals in response to various physical phenomena that occur in an analog fashion (i.e., in continuous time and amplitude). Signal processing can be performed in the analog or digital domain. To process an analog signal in the digital domain, a digital signal must be generated by sampling and quantizing (digitizing) the analog signal. Therefore, unlike an analog signal, a digital signal is discrete in both time and amplitude. The digitization process is performed using an analog-to-digital (A/D) converter.

Digital signal processing (DSP) involves manipulating digital signals to extract useful information. Although an increasing amount of signal processing is performed in the digital domain, there remains a need to interface with the analog world we live in. Analog-to-digital (A/D) and digital-to-analog (D/A) data converters are the devices that enable this interface. **Figure 1.1** illustrates the main components of a DSP system, including A/D, DSP, and D/A devices.

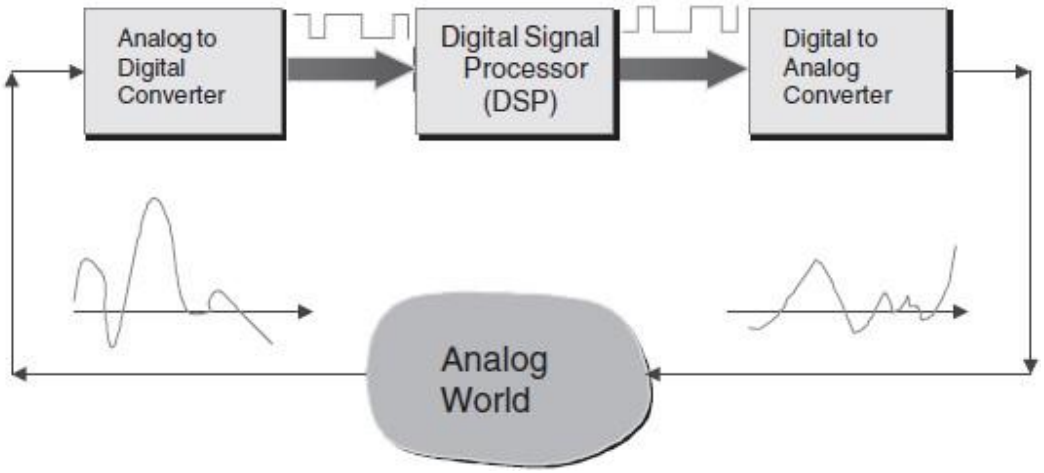


Fig. 1.1. Main components of a DSP system

There are many reasons why one might want to process an analog signal digitally by converting it to a digital signal. The primary reason is that digital processing enables programmability. The same DSP hardware can be used for many different applications simply by changing the code residing in memory. Another reason is that digital circuits provide a more stable and tolerant output than analog circuits, for example, when subjected to temperature changes. Furthermore, the advantage of operating in the digital domain can be inherent. For example, a linear phase filter or a sharp-cut band-stop filter can only be implemented using digital signal processing techniques, and many adaptive systems are only achievable in a practical product through digital signal manipulation. In essence, the digital representation (0s and 1s) allows voice, audio, image, and video data to be processed in the same way for error-tolerant digital transmission and storage.

Therefore, digital processing, and thus digital signal processors (also known as DSPs), are expected to play a major role in the next generation of telecommunications infrastructure, including 3G and 4G wireless lines (third and fourth generation), cables (cable modems) and telephone lines (digital subscriber line - DSL modems).

Digital signal processing can be implemented on various platforms such as a DSP processor, a custom very large-scale integrated circuit (VLSI), or a general-purpose microprocessor. Some of the differences between a DSP and a single-function VLSI implementation are as follows:

1. The implementation of DSPs offers great application flexibility, as the same DSP hardware can be used for different applications. In other words, DSP processors are programmable. This is not the case for a hardwired digital circuit.

2. DSP processors are economical because they are mass-produced and can be used for many applications. A custom VLSI chip is typically designed for a single application and a specific customer.

3. In many situations, new features constitute a software upgrade on a DSP processor that does not require new hardware. Furthermore, bug fixes are generally easier to implement.

4. Often very high sampling rates can be achieved by a custom chip, whereas there are sampling rate limitations associated with DSP chips due to their peripheral constraints and architecture design.

DSP processors share certain common characteristics that also distinguish them from general-purpose microprocessors. Some of these characteristics are as follows:
They are optimized to handle the repetition or looping of common operations in signal processing algorithms. Relatively speaking, DSP instruction sets are smaller and optimized for signal processing operations, such as multiplication and accumulation, in a single cycle.

2. Digital signal processors (DSPs) enable specialized addressing modes, such as indirect and circular addressing. These are efficient addressing mechanisms for implementing many signal processing algorithms.

3. DSPs have appropriate peripherals that allow for efficient input/output (I/O) interfacing with other peripherals.

4. In DSP processors, it is possible to perform multiple memory accesses in a single instruction cycle. In other words, these processors have a relatively high bandwidth between their central processing units (CPUs) and memory.

The majority of the DSP market share belongs to real-time, cost-effective embedded systems, such as cell phones, modems, and disk drives. Real-time means completing processing within the allowed or available time between samples. This available time, of course, depends on the application. As illustrated in Figure 1-2, the number of instructions required for an algorithm to run in real time must be less than the number of instructions that can be executed between two consecutive samples.

For example, for audio processing operating at a sampling rate of 44.1 kHz, corresponding to a sampling interval of approximately 22.6 μs, the number of instructions must be less than nearly 4500, assuming an instruction cycle time of 5 ns. Real-time processing has two aspects: a) the sampling rate and b) the system latencies (delays). Typical sampling rates and latencies for several different applications are shown in **Table 1.1**.

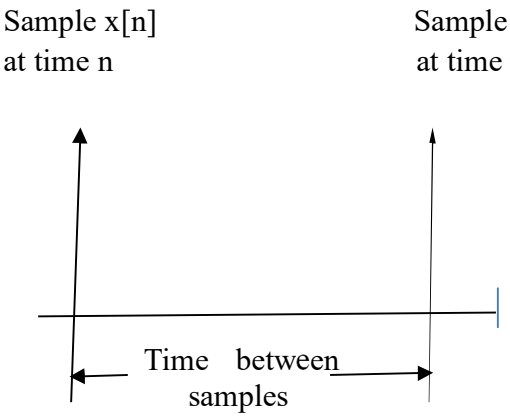


Fig. 1.2: Maximum number of instructions to be followed in real time = time between samples / instruction cycle time.

Table 1.1 Typical sampling rates and latencies for certain applications

Application	I/O sampling rate	Latency
Instrumentation	1 Hz	*Depends on the system
Order	> 0.1 kHz	*Depends on the system
Voice	8 kHz	< 50 ms
Audio	44.1 kHz	*< 50 ms
Video	1–14 MHz	*< 50 ms

In many cases, latency is not a concern; for example, a TV signal depends more on synchronization with the audio than on latency. In each of these cases, latency depends on the application.

1.2 Examples of DSP systems

To enable the reader to appreciate the usefulness of DSPs, an example of currently used DSP systems is presented here.

In recent years, the wireless market has experienced considerable growth.

Figure 1.3 illustrates a DSP wireless communication system for cellular phones.

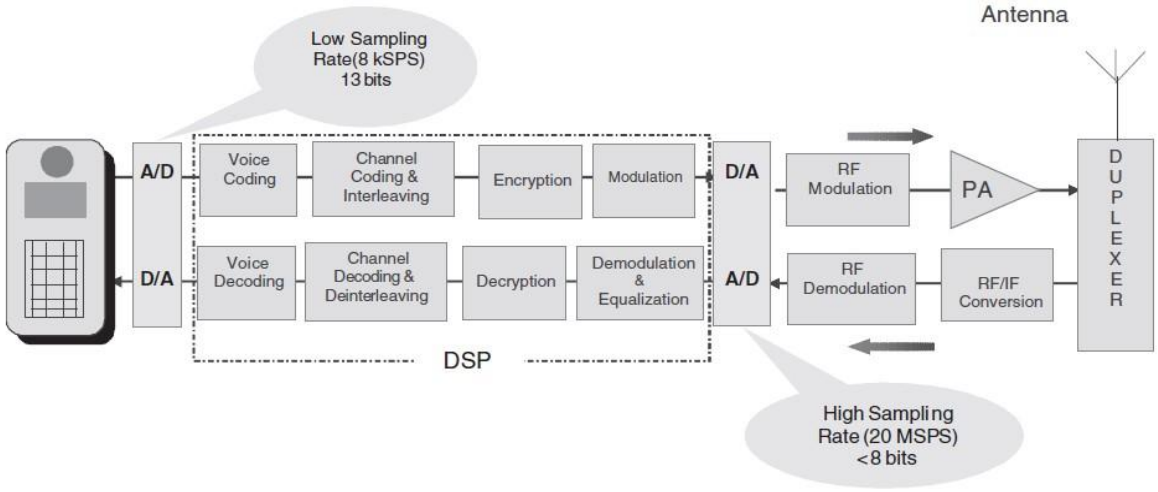


Fig. 1.3 Wireless communication DSP system for cell phones

As this figure shows, there are two sets of data converters. On the voice band side, a low sampling rate (e.g., 8 KSPS [kilo samples per second]) and a high-resolution converter (e.g., 13 bits) are used, while on the RF modulation side, a relatively high bit rate (e.g., 20 MSPS) and a low-resolution converter (e.g., 8 bits) are used. System designers prefer to integrate more functionality into DSPs rather than analog components to reduce the number of components and thus the overall cost. This strategy of increased integration into DSPs depends on specifications that are feasible for low power consumption in portable devices.

Chapter 2: Fixed and Floating-Point Arithmetic

2.1 Fixed or floating point

An important characteristic that distinguishes different DSP processors is whether their processors perform fixed-point or floating-point arithmetic. In a fixed-point processor, numbers are represented and manipulated in integer format. In a floating-point processor, in addition to integer arithmetic, floating-point arithmetic can be handled. This means that numbers are represented by a combination of a mantissa (or fractional part) and an exponent, and the CPU has the hardware necessary to manipulate both parts. Consequently, floating-point processors are generally more expensive and slower than fixed-point processors.

In a fixed-point processor, the dynamic range of numbers must be considered, as a much narrower range of numbers can be represented in integer format compared to floating-point format. For most applications, this concern can be practically ignored when using a floating-point processor. Consequently, fixed-point processors generally require more coding effort than their floating-point counterparts.

2.2 Representation of numbers in Q format on fixed-point DSPs

The decimal value of a two's complement number $B = b_{N-1}b_{N-2} \ldots b_1b_0$, $b_i \epsilon \{0, 1\}$, is given by:

$$D(B) = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \ldots + b_12^1 + b_02^0 \qquad (2.1)$$

The two's complement representation allows a processor to perform addition and subtraction of integers using the same hardware. When using an unsigned integer representation, the sign bit is treated as an extra bit.

In this way, only positive numbers can be represented.

There is a limitation to the dynamic range of the preceding integer representation scheme. For example, in a 16-bit system, it is not possible to represent numbers greater than +215 - 1 = 32767 and less than -215 = -32768. To overcome this limitation, numbers are normalized between -1 and 1. In other words, they are represented as fractions. This normalization is performed by the programmer by moving the implicit or imaginary bit point (note that there is no physical memory allocated at this point) as shown in Figure 6-1. In this way, the fractional value is given by:

$$F(B) = -b_{N-1}2^0 + b_{N-2}2^1 + \ldots + b_1 2^{-(N-2)} + b_0 2^{-(N-1)}$$

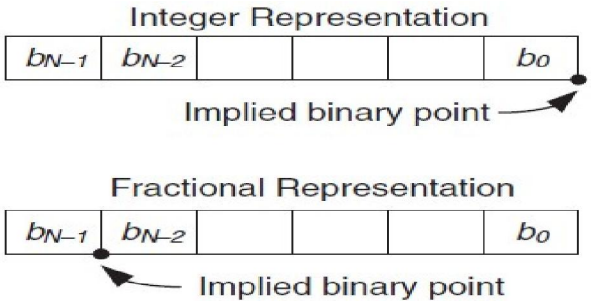


Fig. 2.1 Number representation

This representation scheme is called Q format or fractional representation. The programmer must keep track of the implicit bit point when manipulating numbers in Q format. For example, consider two numbers in Q-15 format, given that we have 16 bits of wide memory. Each number consists of a sign bit plus 15 fractional bits. When these numbers are multiplied, a number in Q-30 format is obtained (the product of two fractions is always a fraction), with bit 31 being the sign bit and bit 32 another sign bit (called the extended sign bit). If there are not enough bits available to store all 32 bits and only 16 bits can be stored, it makes sense to store the most significant bits. This is achieved by storing the top

portion of the 32-bit product register by performing a 15-bit right shift (SHR). In this way, the product would be stored in Q-15 format. (See Figure 2.2.)

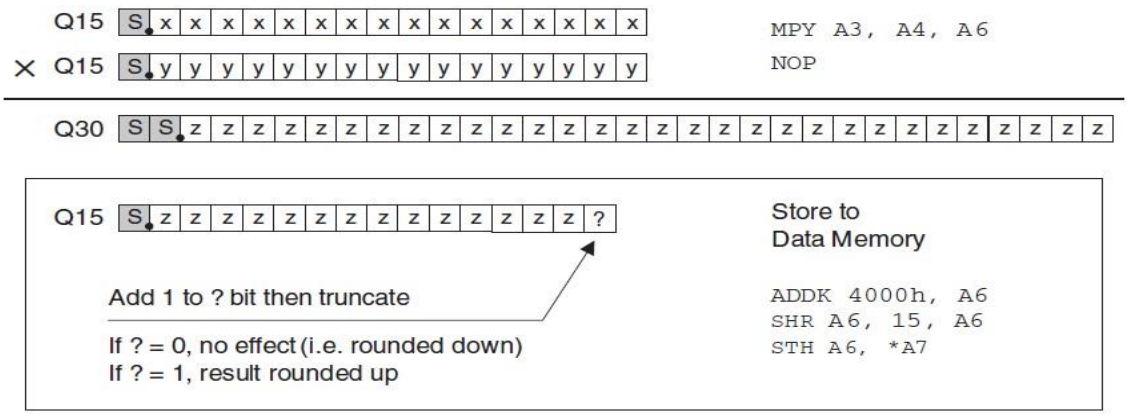


Fig. 2.2 Multiplication and Number Storage Q-15.

Based on the two's complement representation, a dynamic range of $-(2^{N-1}) \leq D(B) < 2^{N-1} - 1$ can be obtained, where N denotes the number of bits. For illustrative purposes, consider a 4-bit system where the most negative number is -8 and the most positive number is 7. The decimal representations of the numbers are shown in **Figure 2.3.** Note how the numbers change from most positive to most negative with the sign bit.

Since only whole numbers between -8 and 7 can be represented, it's easy to see that any multiplication or addition resulting in a number greater than 7 or less than -8 will cause an overflow. For example, when 6 is multiplied by 2, we get 12. Therefore, the result is above the representation limits and will be wrapped around 1100, or -4.

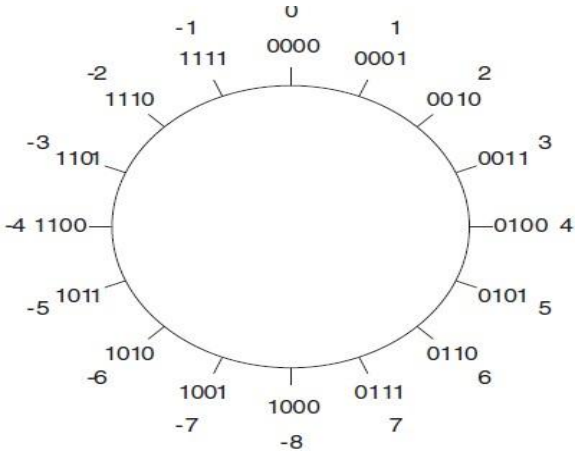


Fig. 2.3: 4-bit binary representation

The Q-format representation solves this problem by normalizing the dynamic range between -1 and 1. Any resulting multiplication will be within the limits of this dynamic range. Using a Q-format representation, the dynamic range is divided into 2^N sections, where $2^{-(N-1)}$ is the size of a section. The most negative number is always -1, and the most positive number is $1 - 2^{-(N-1)}$.

The following example illustrates the difference between the two representation schemes. As shown in **Figure 2.4**, multiplying 0110 by 1110 in binary is equivalent to multiplying 6 by –2 in decimal, resulting in –12, a number exceeding the dynamic range of the 4-bit system. Based on the Q-3 representation, these digits correspond to 0.75 and –0.25, respectively. The result is –0.1875, which falls within the fractional range. Note that the hardware generates the same 1s and 0s; what differs is the interpretation of the bits.

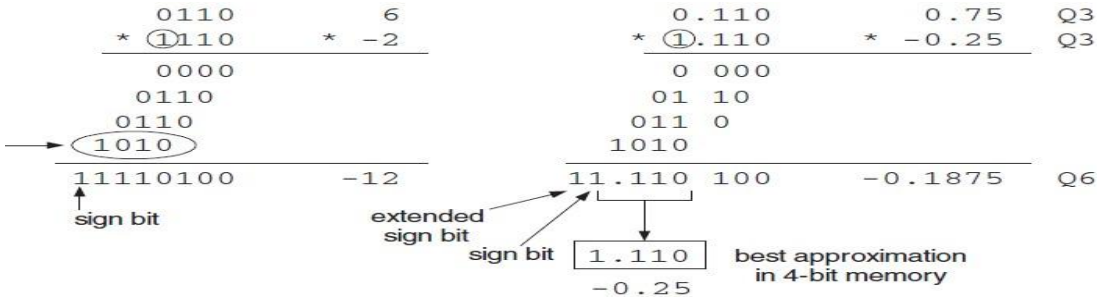


Fig. 2.4 Binary and fractional multiplication

When multiplying QN numbers, remember that the result will consist of 2N fractional bits, a sign bit, and one or more extended sign bits. Depending on the data type used, the result must be shifted accordingly. If two Q-15 numbers are multiplied, the result will have a width of 32 bits, with the MSB being the extended sign bit followed by the sign bit. The imaginary decimal point will be after the 30th bit.

A right shift of 15 is therefore necessary to store the result in a 16-bit memory location as a Q-15 number. It should be understood that some precision is lost, of course, as a result of removing the smaller fractional bits. Since only 16 bits can be stored, the shift allows the higher-precision fractional bits to be retained. If 32-bit storage capacity is available, a left shift of 1 can be performed to remove the extended sign bit and store the result as a Q-31 number.

To better understand a potential loss of precision when manipulating numbers in Q format, consider another example where two Q12 numbers corresponding to 7.5 and 7.25 are multiplied. As shown in Figure 2.5, the resulting product must be shifted 4 bits to the left to store all the fractional bits corresponding to the Q12 format.

However, this results in a product value of 6.375, which is different from the correct value of 54.375. If the product is stored in a lower-precision Q format, for example in Q8 format, the correct product value can be stored.

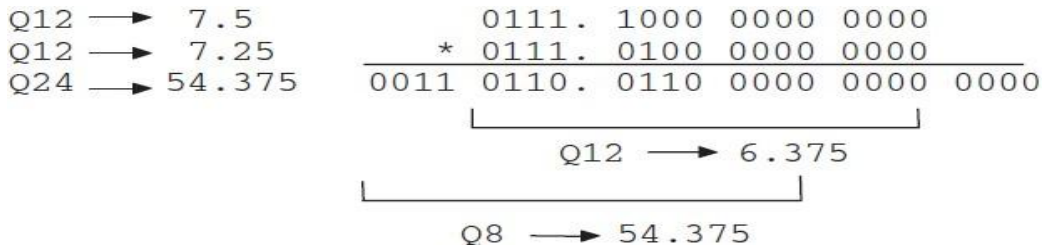


Fig. 2.5 Example of loss of precision in Q format

Although the Q format solves the problem of overflow in multiplication, addition and subtraction still present an issue. When adding two Q15 numbers, the sum exceeds the Q15 representation range. To resolve this problem, the scaling approach, discussed later, must be used.

2.3 Representation of floating-point numbers

The IEEE 754 standard for representing floating-point numbers is the most commonly used standard in DSP processors. It uses a single-precision format with 32 bits and a double-precision format with 64 bits.

The IEEE 754 (Institute of Electronic and Electrical Engineers) single-precision standard is as follows: the first bit (s) is the sign of the number; this is followed by 8 bits representing e, the exponent in code exceeding 127. In this code, 127 represents 0, any number less than 127 is considered negative, and any number between 127 and 255 is positive. Exponents ranging from -126 (00000001) to +127 (11111110) can thus be expressed. The exponents -127 and +128 are reserved for the representations of 0 and ∞, respectively. The following

23 bits represent the fractional part f of the normalized mantissa. This is a normalized mantissa m of the form

$$m = 1,f \text{ such that } 1 \leq m < 2 \qquad (2.2)$$

Since the first bit of such a normalized mantissa is always 1, it doesn't need to be written, saving one bit and achieving 24-bit precision. This representation therefore requires a total of 32 bits, or 4 bytes. This represents a decimal precision on the order of 7 significant digits and can express numbers ranging from approximately

10^{-38} to 10^{+38}.

The value of the number is therefore given by the expression:

$$N = (-1)^s \times 2^{(e-127)} \times 1,f \qquad (2.3)$$

where s is the sign of the mantissa, and e is the exponent as written in the following format:

0	100 0011 1	101 1000 0000 0000 0000 0000
sign	e = exponent	fractional part f
s	exceeding 127 of the normalized mantissa	

(+) = 135

e - 127 = 8

Fig. 2.6: Representation of 432 in single-precision IEEE floating-point

The number 432 = $(110110000)_2$ = 1.10110000 x 2^8 can therefore be written in single precision:

$43D80000_{IEEE}$. Here, hexadecimal notation is used as an abbreviation for the binary representation of the IEEE number. The subscript IEEE is used to clearly indicate that this is not a hexadecimal number.

Examples:

$+0 = 00000000_{IEEE}$,

$-0 = 80000000_{IEEE}$,

$+1 = 1.0 \times 20 = 3F800000_{IEEE}$,

$+2 = 1.0 \times 21 = 40000000_{IEEE}$,

$+\infty = 1.0 \times 2128 = 7F800000_{IEEE}$

$-\infty = FF800000_{IEEE}$.

In double precision, 8 bytes, or 64 bits, are used, of which 11 are used for the exponent (in codes exceeding 1023) and 52 for the fractional part of the mantissa. This results in a precision equivalent to 15 significant figures in decimal and exponents ranging from 10^{-308} to 10^{+307}.

The value of a number N in double precision is therefore given by the expression:

$$N = (-1)^s \times 2^{(e-1023)} \times 1,f \qquad (2.4)$$

In extended precision, 10 bytes, or 80 bits, are used, of which 15 are used for the exponent (in codes exceeding 16,384) and 63 for the fractional part of the mantissa. This results in a precision equivalent to 19 significant digits in decimal and exponents ranging from 10^{-493} to 10^{+4931}.

The value of a number N in extended precision is therefore given by the expression:

$$N = (-1)^s \times 2^{(e-16383)} \times 1,f \qquad (2.5)$$

2.4 Sampling

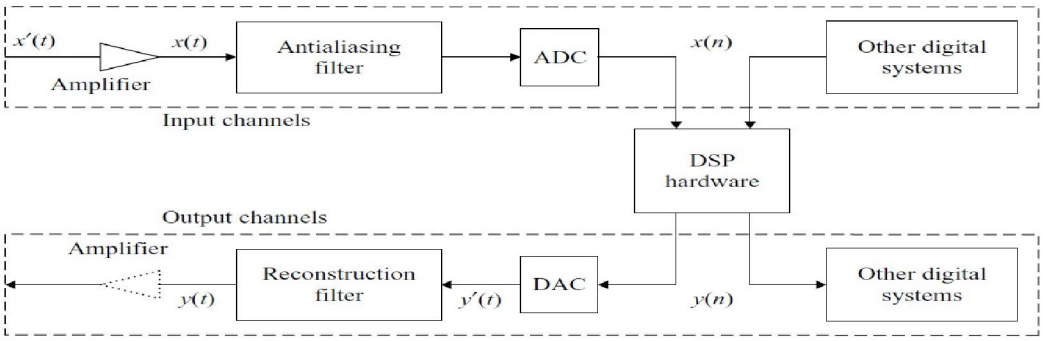


Fig. 2.7 Basic functional diagram of a real-time DSP system

As shown in Figure 2.7, the ADC converts the analog signal x(t) into a digital signal x(n). Analog-to-digital conversion, commonly called digitization, includes the processes of sampling (time digitization) and quantization (amplitude digitization), as illustrated in **Figure 2.8**. The sampling process represents an analog signal as a sequence of values. The basic sampling function can be performed with an ideal "sample-and-hold" circuit, which maintains the level of the sampled signal until the next sample is taken.

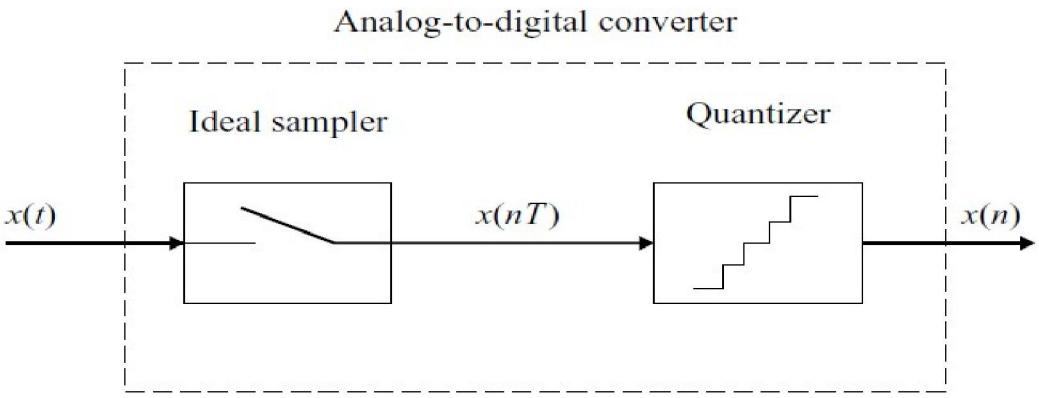


Fig. 2.8: Block diagram of ADC

The quantization process approximates a waveform by assigning a number to each sample. Therefore, the analog-to-digital conversion will perform the following steps:

1. The band-limited signal x(t) is sampled at uniformly spaced time intervals nT, where n is a positive integer and T is the sampling period in seconds. This sampling process converts an analog signal into a discrete-time signal x(nT) with a continuous amplitude value.

2. The amplitude of each discrete-time sample is quantized into one of two levels B, where B is the number of bits the ADC must represent for each sample. The discrete amplitude levels are represented (or encoded) as distinct binary words x(n) with a fixed word length B.

The reason for this distinction is that these processes introduce different distortions. The sampling process results in aliasing or bending distortion, while the encoding process results in quantization noise. As shown in Figure 2.2, the sampler and quantizer are integrated on the same chip. However, high-speed ADCs typically require an external sampling and holding device.

An ideal sampler can be viewed as a switch that opens and closes periodically every Ts (seconds).

The sampling period is defined as:

$$T = \frac{1}{f_s} \qquad (2.6)$$

where f_S is the sampling frequency (or sampling rate) in hertz (or cycles per second). The intermediate signal x(nT) is a discrete-time signal with a continuous value (a number with infinite precision) at a discrete time nT, n = 0, 1, ..., ∞, as illustrated in **Figure 2.9**. The

analog signal x(t) is continuous in time and amplitude. The sampled discrete-time signal x(nT) is continuous in amplitude, but is only defined at discrete sampling instants t = nT.

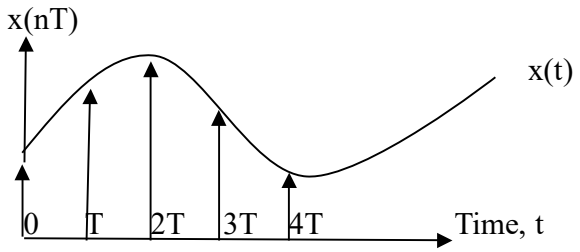


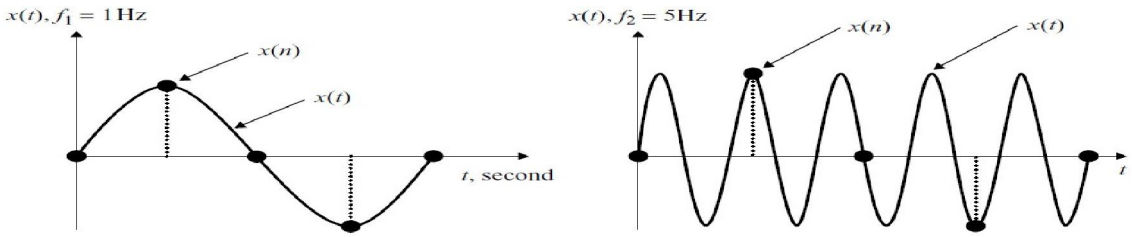
Fig. 2.9 Example of an analog signal x(t) and a discrete-time signal x(nT): In order to accurately represent an analog signal x(t) by a discrete-time signal x(nT), the sampling frequency f_S must be at least twice the maximum frequency component (f_M) in the analog signal x(t). That is,

$$f_S \geq 2f_M$$

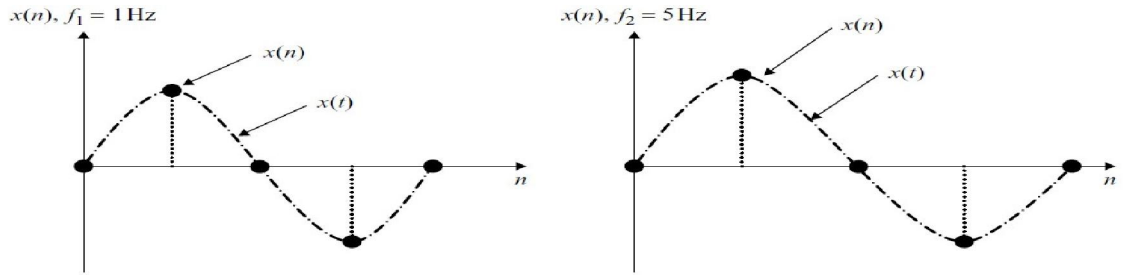
where f_M is also called the bandwidth of the signal x(t). This is Shannon's sampling theorem, which states that when the sampling frequency is greater than twice the highest frequency component contained in the analog signal, the original signal x(t) can be perfectly reconstructed from the corresponding discrete-time signal x(nT).

The minimum sampling rate $f_S = 2\ f_M$ is called the Nyquist rate. The frequency $f_N = f_S / 2$ is called the Nyquist frequency or aliasing frequency. The frequency range $[-f_S/2, f_S/2]$ is called the Nyquist interval. When an analog signal is sampled at f_S, frequency components above $f_S/2$ are aliased into the frequency range $[0, f_S/2]$. The aliased frequency components overlap with the original frequency components in the same range. Consequently, the original analog signal cannot be recovered from the sampled data. This undesirable effect is called aliasing.

Example 1: Consider two sinusoidal waves with frequencies $f_1 = 1$ Hz and $f_2 = 5$ Hz that are sampled at $f_S = 4$ Hz, rather than 10 Hz according to the sampling theorem. The analog waveforms are shown in **Figure 2.10(a)**, while their digital samples and reconstructed waveforms are shown in **Figure 2.4(b)**. As the figures show, we can reconstruct the original waveform from the digital samples for the sinusoidal wave with frequency $f_1 = 1$ Hz. However, for the original sinusoidal wave with frequency $f_2 = 5$ Hz, the reconstructed signal is identical to the sinusoidal wave with frequency 1 Hz. Therefore, f_1 and f_2 are assumed to be related to each other; that is, they cannot be distinguished by their discrete-time samples.



(a) Original analog waveforms and digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz.



(b) Digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz and reconstructed waveforms.

Fig. 2.10 Example of the aliasing phenomenon: (a) original analog waveforms and digital samples for $f_1 = 1$ Hz and $f_2 = 5$ Hz; (b) digital samples of $f_1 = 1$ Hz and $f_2 = 5$ Hz and reconstructed waveforms in **Figure 2.10 (b)**.

As the figures show, we can reconstruct the original waveform from the digital samples for the sinusoidal wave with frequency f_1 = 1 Hz. However, for the original sinusoidal wave with frequency f_2 = 5 Hz, the reconstructed signal is identical to the sinusoidal wave with frequency 1 Hz. Therefore, f_1 and f_2 are said to be aliased, meaning they cannot be distinguished by their discrete-time samples.

Example 2: The sampling frequency range required by DSP systems is wide, ranging from approximately 1 GHz in radar to 1 Hz in instrumentation. Given a sampling rate for a specific application, the sampling period can be determined. Some real-world applications use the following sampling frequencies and periods:

1. In the International Telecommunication Union (ITU) speech compression standards, the sampling rate of ITU-T G.729 and G.723.1 is f_S = 8 kHz, therefore the sampling period T = 1/8000 s = 125 μs. Note that 1 μs = 10^{-6} s.

2. Broadband telecommunication systems, such as ITU-T G.722, use a sampling rate of f_S = 16 kHz, therefore T = 1/16,000 s = 62.5 μs.

3. In audio CDs, the sampling frequency is f_S = 44.1 kHz, therefore T = 1/44 100 s = 22.676μs.

4. High-fidelity audio systems, such as the AAC (Advanced Audio Coding) standard, MPEG2 (Moving Picture Experts Group), and the MP3 audio compression standard (layer MPEG 3) and Dolby AC-3 have a sampling rate of f_S = 48 kHz, and therefore
T = 1/48000 s = 20.833μs. The sampling frequency for MPEG-2 AAC can reach 96 kHz.

Example 3:

We consider the two analog signals: $x_1(t)=\cos 2\pi(10)t$ And $x_2(t) = \text{Cos}2\pi\ (50)t$

Which are sampled at a sampling rate of $F_S = 40$ Hz. The corresponding discrete-time signals or sequences are:

$x_1(n) = \cos 2\pi\ (10/40)n = \cos(\pi/2)n$

$x_2(n) = \cos 2\ \pi\ (50/40)n = \cos(5\ \pi\ /2)n = \cos(2\ \pi\ n + \pi\ /2n) = \cos(\pi\ /2)n \Rightarrow$

$x_1(n) = x_2(n)$.

We say that the frequency $F_2 = 50$ Hz is a fold of the frequency $F_1 = 10$ Hz at a sampling rate of 40 samples per second.

Example 4: Consider the analog signal: $x_a(t) = A \cos(2\pi Ft+\theta) = 3\cos 100\pi t$

a) Determine the minimum sampling rate needed to avoid aliasing.
b) Suppose the signal is sampled at a rate $F_S = 200$ Hz. What is the discrete-time signal obtained after sampling?
c) Assuming the signal is sampled at a sampling rate of $F_S = 75$ Hz, what is the resulting discrete-time signal?
d) What is the frequency $0 < F \leq F_{S/2}$ of a sinusoid that gives samples identical to those obtained in C)?

Solution :

a) The frequency of the analog signal is F = 50 Hz. Therefore, the required sampling rate to avoid aliasing is $F_S = 100$ Hz.
b) If the signal is sampled at $F_S = 200$ Hz, the discrete-time signal is: x(n) = 3cos

(100/200) πn = 3cos (π/2)n.

c) If the signal is sampled at F_S = 75 Hz, the discrete-time signal is: x(n) = 3cos (100/75) πn = 3cos (4/3) πn = 3cos (2π-2/3π)n = 3cos (2/3) πn.

d) For the sampling rate of F_S = 75 Hz, we have F = fF_S = 75f, the frequency of the sinusoid in c) is f = 1/3 ⇒ F = 25 Hz.

2.5 Uniform Quantification

The ADC assumes that the input values cover a full-scale range, say R. Typical values for R are between 1 and 15 volts. Since the quantized sampled value xQ(nT) is represented by bits B, it can only take one of the 2B possible quantization levels. If the spacing between these levels is the same across the entire range R, then we have a uniform quantizer. The spacing between the quantization levels is called the quantization width or quantizer resolution.

For uniform quantization, the resolution is given by:

$$Q = \frac{R}{2^B} \qquad (2.7)$$

The number of bits required to achieve a required resolution of Q is therefore

$$B = log_2 \frac{R}{Q} \qquad (2.8)$$

Most ADCs can accept bipolar inputs, meaning that the sampled values are within the symmetrical range:

$$-\frac{R}{2} \leq x(nt) < \frac{R}{2} \qquad (2.9)$$

For unipolar inputs, $0 \leq x(nT) < R$

In practice, the input signal x(t) must be preconditioned to fall within the quantizer's full-scale range. **Figure 2.11** shows the quantization levels of a 3-bit quantizer for bipolar inputs.

If the conversion is unipolar, the output quantity is always of the same sign and can therefore take values between 0 and 2^n-1.

If we have, for example, a unipolar 12-bit-10-volt DAC. If the conversion is bipolar, the output quantity can be negative or positive, and can therefore take values between -2^{n-1} and 2^{n-1} -1.

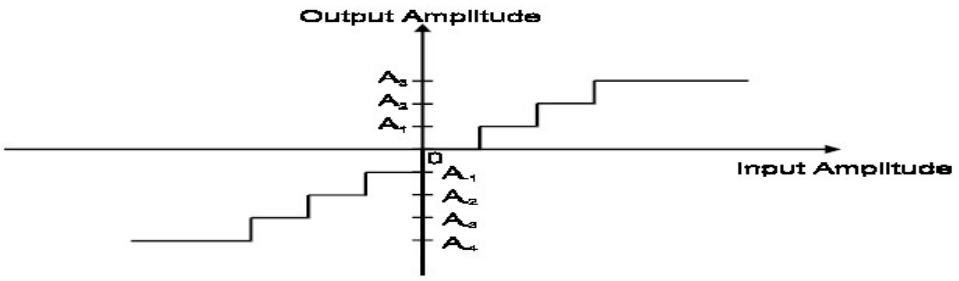


Fig. 2.11 A 3-bit uniform quantizer transfer function

We now discuss a method for representing the sampled discrete-time signal x(nT) as a binary number with a finite number of bits. This is the process of quantization and encoding. If the word length of an ADC is B bits, there are 2B different values (levels) that can be used to represent a sample. If x(n) lies between two quantization levels, it will be either rounded or truncated. Rounding replaces x(n) with the value of the nearest quantization level, while truncation replaces x(n) with the value of the lower level. Since rounding produces a less biased representation of analog values, it is widely used by ADCs.

Therefore, quantization is a process that represents a sample with an analog value x(nT) whose nearest level corresponds to the digital signal x(n).

We can use 2 bits to define four equally spaced levels (00, 01, 10 and11) to classify the signal into the four sub-ranges as illustrated in **Figure 2.12**.

In this figure, the symbol "o" represents the discrete-time signal x(nT) and the symbol "•" represents the digital signal x(n). The spacing between two levels of

Consecutive quantization is called the quantization width, step size, or resolution. If the spacing between these levels is the same, then we have a uniform quantizer. For uniform quantization, the resolution is given by dividing a full-scale range by the number of quantization levels, 2^B.

In **Figure 2.12**, the difference between the quantized number and the original value is defined as the quantization error, which appears as noise in the converter output. It is also called quantization noise, which is assumed to be uniformly distributed random variables. If a B-bit quantizer is used, the signal-to-noise ratio (SQNR) is approximated by SQNR ≈ 6B dB.

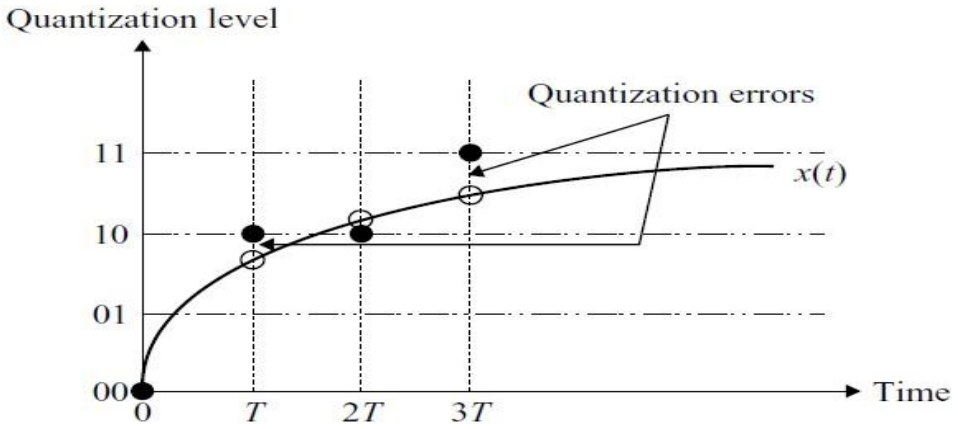


Fig. 2.12: Digital samples using a 2-bit quantizer

Quantification error is the difference between the actual sampled value and the quantified value. Mathematically, it is

$$e(nT) = x(nT) - x_Q(nT) \qquad (2.10)$$

or its equivalent,

$$e(n) = x(n) - x_Q(n) \qquad (2.11)$$

If x (n) lies between two levels of quantification, it will either be rounded or truncated.

Rounding replaces x(n) with the value of the nearest quantization level. Truncation replaces x(n) with the value of the level below.

For rounding, the error is given by:

$$-\frac{Q}{2} \prec e \prec \frac{Q}{2} \qquad (2.13)$$

For truncation, the error is error is given by

$$0 \leq e < Q \qquad (2.14)$$

It is clear that rounding produces a less biased representation of analog values.

The average:

$$\bar{e} = \frac{1}{Q}\int_{\frac{-Q}{2}}^{\frac{Q}{2}} e\,de = 0 \qquad (2.15)$$

This means that on average half of the values are rounded up and half are rounded down.

The root mean square (RMS) value of the error gives us an idea of the average power of the error signal. It is given by:

$$\bar{e} = \frac{1}{Q}\int_{\frac{-Q}{2}}^{\frac{Q}{2}} e^2 de = \frac{Q^2}{12} \qquad (2.16)$$

The mean squared error is therefore:

$$e_{rms} = \sqrt{\overline{e^2}} = \frac{Q}{\sqrt{12}} \qquad (2.17)$$

The quantization signal-to-noise ratio is:

$$\text{SQNR} = 20 \log10 [R/Q] = 20 \log10 (2^B) = 20B \log_{10}2 = 6B \text{ db} \qquad (2.18)$$

Thus, if we increase the number of bits in the ADC by one bit, the quantization signal-to-noise ratio improves by 6 dB. The equation above gives us the dynamic range of the quantizer.

Example :

The dynamic range of the human ear is approximately 100 dB. If a digital audio system is required to match this dynamic range, it will be necessary to

$$100/6 = 16.67 \text{ bits}$$

A 16-bit quantizer will achieve a dynamic range of 96 dB.
If the highest frequency the human ear can hear is 20 kHz, a sampling rate of at least 40 kHz is required. If the actual sampling rate is 44 kHz, the bit rate of this system will be

$$16 \text{ x } 44 = 704 \text{ kbits/sec}$$

This is the typical bit rate of a compact disc player.

Since the quantization error is a random number within the given range, it is usually modeled as a random signal (or noise) with a uniform distribution, as shown in **Figure 2.13**.

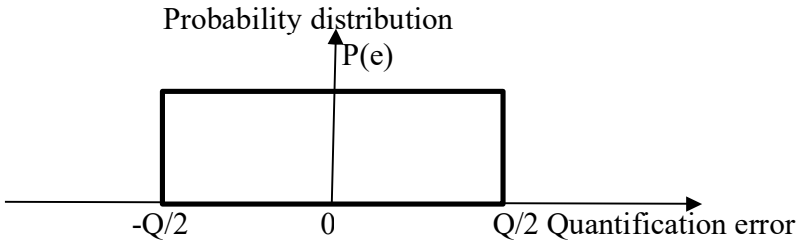


Fig. 2.13 Uniform distribution of quantization error

2.6 Binary coding of quantization levels

Consider an n-bit binary number representing a full-scale interval R. In other words, the interval R is quantized into 2^n quantization levels. If R is unipolar, the quantized value x_Q lies in the interval [0, R]. If it is bipolar, x_Q lies in the interval [−R/2, R/2].

The n-bit model is represented by a vector b = [bn-1, bn-2, …, b1, b0] where b_{n-1} is called the most significant bit (MSB) and b_0 is the least significant bit (LSB). There are many ways to use this n-bit model to encode x_Q. The three most common ways are:

- Unipolar natural binary:

$$x_Q = R(b_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n}) \quad (2.19)$$

- Bipolar shifted binary:

$$x_Q = R(b_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n} -0.5) \quad (2.20)$$

- Bipolar complement to 2:

$$x_Q = R(Б_{n-1}2^{-1}+b_{n-2}2^{-2}+ \dots +b_1 2^{-(n-1)}+b_0 2^{-n} -0.5) \quad (2.21)$$

Here, $Б_{n-1}$denotes the 2's complement of B_{n-1}.

Example :

For R=2V and 3-bit quantization (8 levels), the correspondence between the binary representation and the quantized value is:

b2 b1 b0	Natural binary	Offset Binary	Complement to 2
111	1.75	0.75	-0.25
110	1.50	0.50	-0.50
101	1.25	0.25	-0.75
100	1.00	0.00	-1.00
011	0.75	-0.25	0.75
010	0.50	- 0.50	0.50
001	0.25	- 0.75	0.25
000	0.00	-1.00	0.00

The natural unipolar binary representation codes the levels between 0 and 2 V. The shifted binary and the two's complement code the levels between -1 V and 1 V.

2.7 Non-uniform quantification

One of the assumptions we made when analyzing the quantization error is that the amplitude of the sampled signal is uniformly distributed across the entire range. This assumption may not hold true for some applications. For example, speech signals are known to have a wide dynamic range. Spoken speech (e.g., vocal sounds) can have amplitudes that cover the entire full-scale range, while non-soft speech (e.g., consonants such as fricatives) typically has much smaller amplitudes. Furthermore, the average person

only speaks about 60% of the time when they are talking. The remaining 40% is silence with a negligible signal amplitude.

If uniform quantization is used, louder sounds will be accurately represented. However, softer sounds will likely occupy only a small number of quantization levels with similar binary values. This means we wouldn't be able to distinguish softer sounds. Consequently, analog speech reconstructed from these digital samples will be almost as intelligible as the original.

To circumvent this problem, non-uniform quantization can be used. Lower amplitudes are assigned more quantization levels, while higher amplitudes have fewer levels. This quantization scheme is illustrated in **Figure 2.14**.

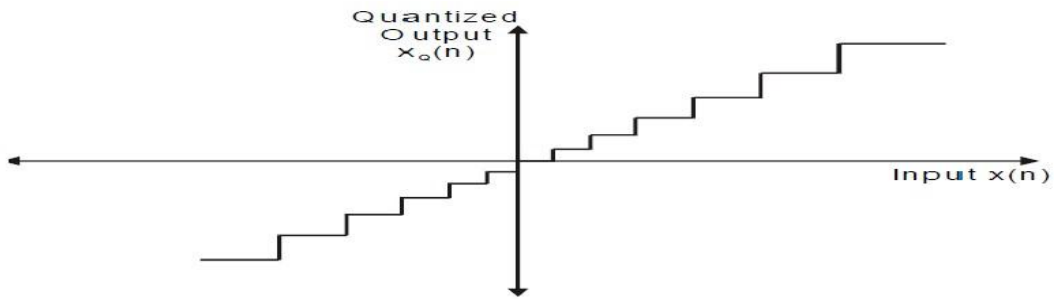


Fig. 2.14 Non-uniform quantification

Alternatively, a uniform quantizer can still be used, but the input signal is first compressed by a system with an input-output relationship (or transfer function) similar to that illustrated in **Figure 2.15**.

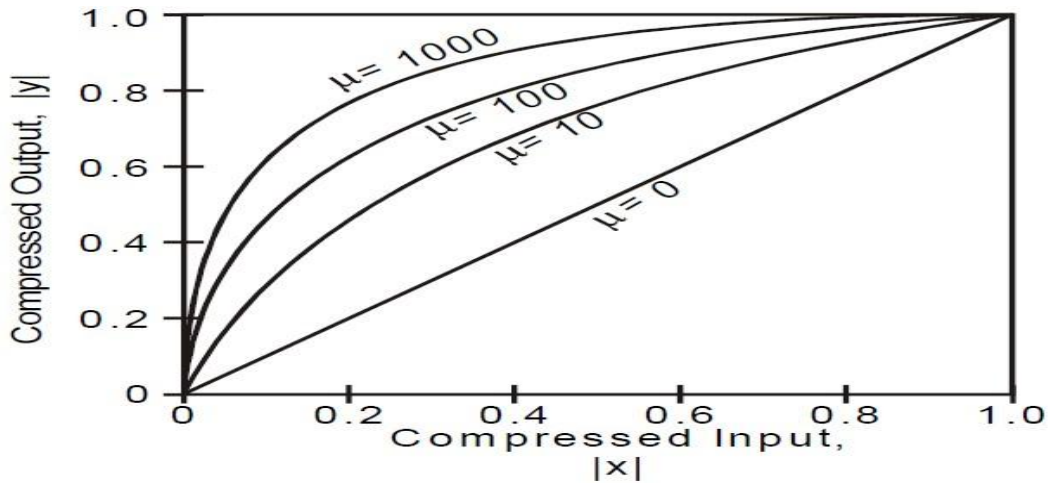


Fig. 2.15: Compression law characteristics μ

Higher amplitudes in the input signal are compressed, effectively reducing the number of levels allocated to them. Lower amplitude signals are stretched (or amplified non-uniformly), causing them to occupy a large number of quantization levels. After processing, the output signal is inversely expanded. The system that expands the signal has an input-output relationship that is the inverse of the compressor. The expander expands high amplitudes and compresses low amplitudes. The entire process is called companding (COMpressing and expanding).

Companding is widely used in public telephone systems. There are two distinct compression-expansion schemes. In Europe, A-law compression is used, and in the United States, μ-law compression is used.

The μ-law compression characteristic is given by the formula:

$$y = y_{max} \frac{\ln\left[1+\mu\left(\frac{|x|}{x_{max}}\right)\right]}{\ln(1+\mu)} sgn(x) \qquad (2.22)$$

Where

$$\text{sgn(x)} = \begin{cases} +1 \; if \; x \geq 0 \\ -1 \; if \; x < 0 \end{cases}$$

Here, x and y represent the input and output values, and x_{max} and y_{max} are the maximum input and output values, respectively. μ is a positive constant. The North American standard specifies μ by 255. Note that μ = 0 corresponds to a linear input-output relationship (i.e., uniform quantization). The compression characteristic is illustrated in **Figure 2.15**.

The compression of A law is given by

$$y = \begin{cases} y_{max} \dfrac{A\left(\frac{|x|}{x_{max}}\right)}{1+lnA} \; sgn(x), & 0 < \dfrac{|x|}{x_{max}} \leq \dfrac{1}{A} \\ y_{max} \dfrac{1+\ln\left[A\left(\frac{|x|}{x_{max}}\right)\right]}{1+lnA} \; sgn(x) & \dfrac{1}{A} < \dfrac{|x|}{x_{max}} < 1 \end{cases} \tag{2.23}$$

Here, A is a positive constant. The European standard specifies A as being 87.6. **Figure 2.16** graphically shows the characteristic.

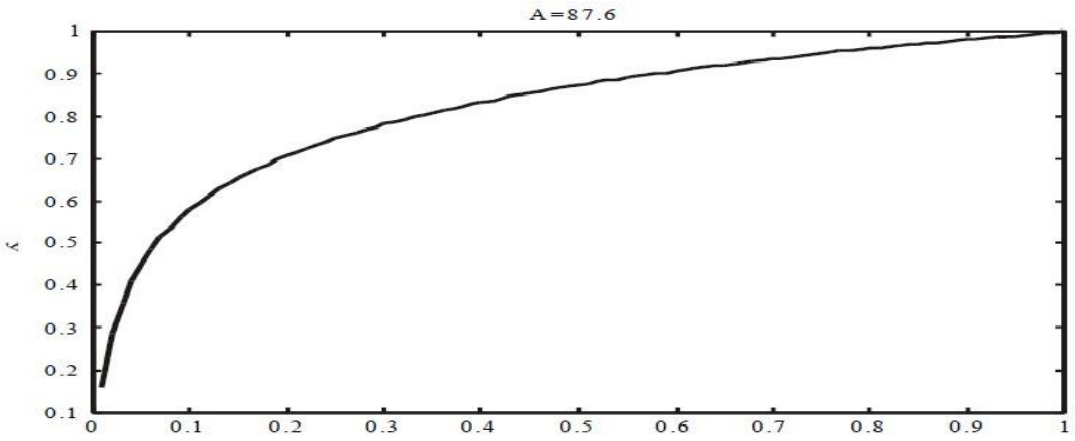


Fig. 2.16 The compression characteristics of law A

Chapter 3: TMS320 C6x DSP Architecture

The TMS320C6x family of processors, manufactured by Texas Instruments, is designed for speed. They are built for millions of instructions per second (MIPS), such as those used in third- and fourth-generation (3G and 4G) wireless and digital imaging. There are many versions of processors in this family, differing in instruction cycle time, speed, power consumption, memory, peripherals, packaging, and cost. For example, the fixed-point C6416-600 version runs at 600 MHz (1.67 ns cycle time), delivering a peak performance of 4800 MIPS. The floating-point C6713-225 version runs at 225 MHz (4.4 ns cycle time), delivering a peak performance of 1350 MIPS.

Figure 3-1 shows a schematic diagram of the generic C6x architecture. The 6x central processing unit (CPU) consists of eight functional units divided into two sides: A and B. Each side has one .M unit (used for the multiplication operation), one .L unit (used for Logical and arithmetic operations), an .S unit (used for branching, bit manipulation, and arithmetic operations), and a .D unit (used for loading, storing, and arithmetic operations). Some instructions, such as ADD, can be performed by multiple units. There are sixteen 32-bit registers associated with each side. Interaction with the CPU must occur through these registers.

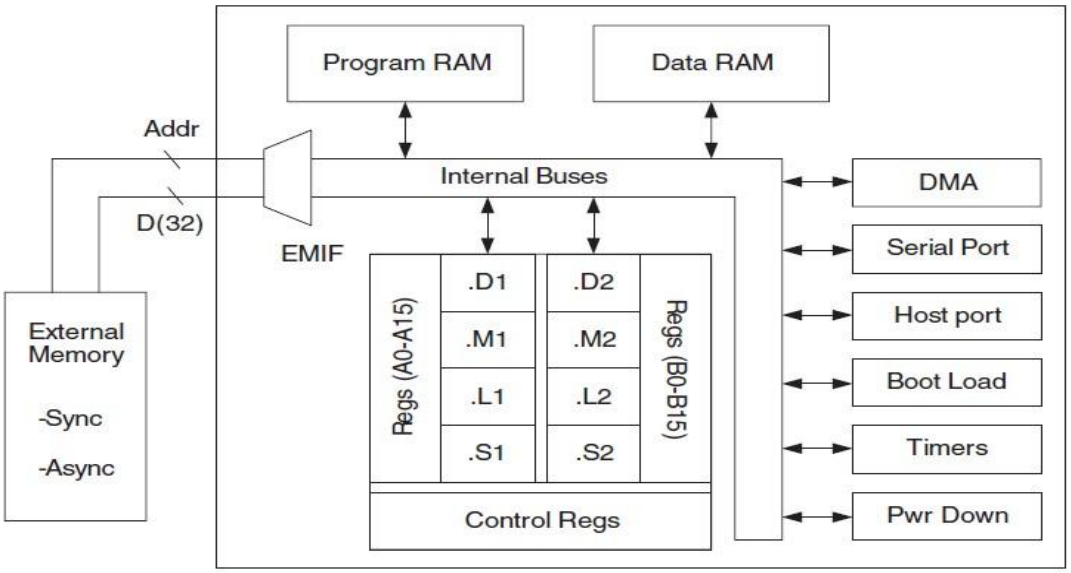


Fig. 3.1 Generic architecture of the C6x

As shown in **Figure 3-2**, the internal buses consist of a 32-bit program address bus, a 256-bit program data bus capable of holding eight 32-bit instructions, two 32-bit data address buses (DA1 and DA2), two 32-bit data buses for loading (64-bit for the C64 version) (LD1 and LD2), and two 32-bit data storage buses (64-bit for the floating-point version) (ST1 and ST2). In addition, there is a 32-bit Direct Memory Access (DMA) data bus and a 32-bit DMA address bus. Off-chip memory is accessed via a 20-bit address bus and a 32-bit data bus.

The peripherals of a typical C6x processor include an external memory interface (EMIF), a Direct Memory Access (DMA), a bootloader, a multi-channel buffered serial port (McBSP), a host port interface (HPI), a timer, and a power-down unit (PDU). EMIF provides the delay required to access external memory. The DMA allows data to move from one memory location to another without interfering with CPU operation. The bootloader starts code from off-chip or HPI memory to internal memory. McBSP provides a high-speed, multi-channel serial communication link. HPI allows a host to access internal

memory. The timer provides two 32-bit counters. The power-down unit is used to conserve power for extended periods when the CPU is idle.

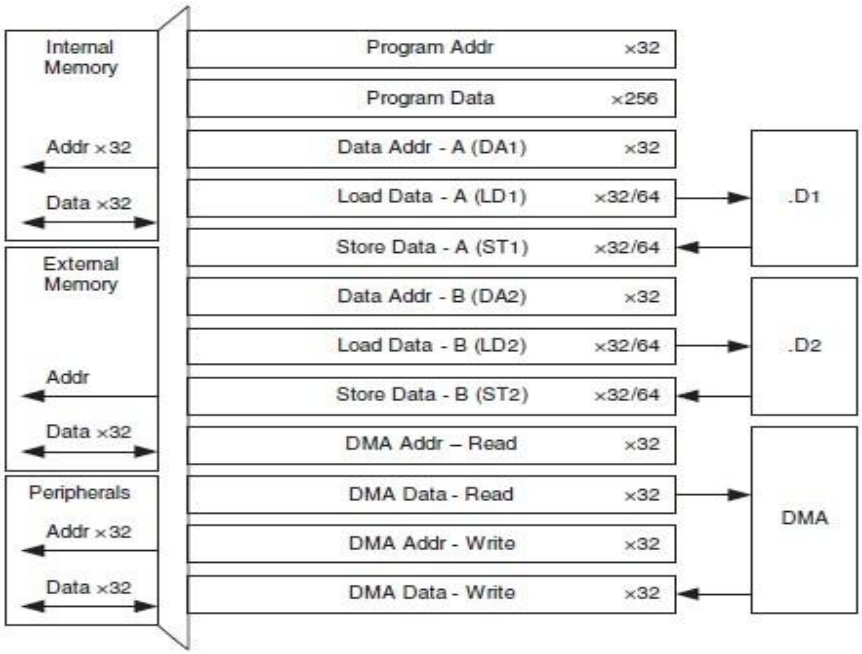


Fig. 3.2 C6x internal buses

3.1 Operation on the control unit (CPU)

Example of a Dot Product:

As shown in Figure 3.1, the C6x processor is split into two data paths, data path A (or 1) and data path B (or 2). An effective way to understand how the processor works is through an example. **Figure 3.3** shows the assembly code for a 40-point dot product y (between two vectors

a and x), $y = \sum_{n=1}^{40} a_n * x_n$

At this point, it is worth mentioning that the assembler is not case-sensitive (i.e., instructions and registers can be written in lowercase or uppercase).

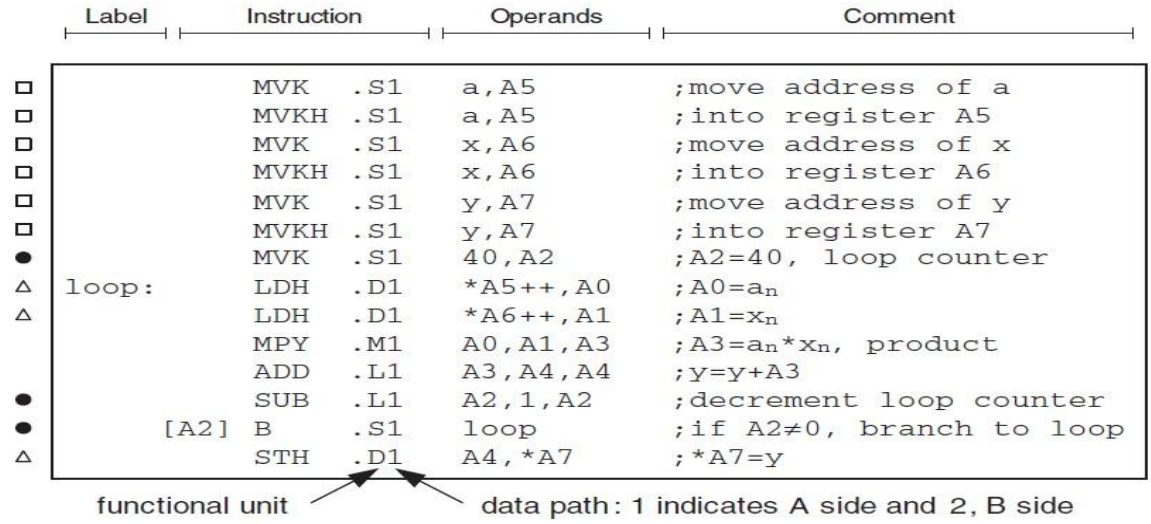


Fig. 3.3 Dot product assembler code

The registers assigned to a_n, x_n, loop counter, product, y, & a[n] (address of an), & x[n] (address of x_n), and & y[n] (address of y_n) are shown in **Figure 3.4**. In this example, only the functional units and registers on the A side are used.

A loop is created by the instructions indicated by • 's. First, a loop counter is set up using the MVK constant move instruction. This instruction uses the .S1 unit to place the constant 40 into register A2. The start of the loop is indicated by the loop label and the end by a SUB subtraction instruction to decrement the loop counter followed by a B branch instruction to return to the loop.

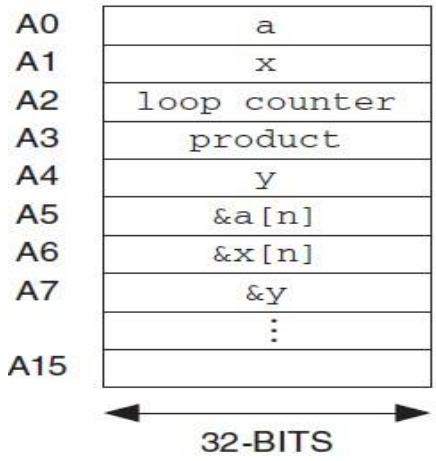


Fig. 3.4 Registers on side A

Subtraction is performed by unit .L1 and branching by unit .S1. The square brackets around the branch instruction indicate that it is a conditional instruction. All C6x instructions can be conditional based on a zero or non-zero value in one of the registers: A1, A2, B0, B1, and B2. The syntax [A2] means "execute the instruction if A2 ≠ 0" and [!A2] means "execute the instruction if A2 = 0". Following these instructions, the loop is repeated 40 times.

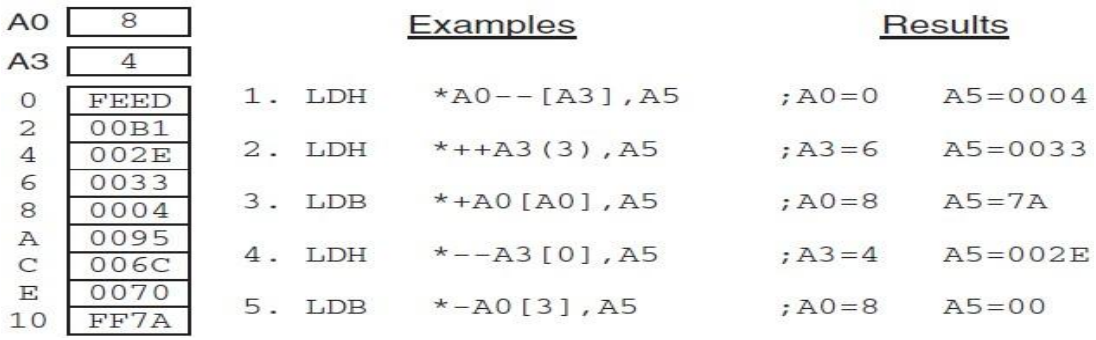
Since interaction with the functional units occurs via the registers on the A-side, these registers must be configured to start the loop. Instructions marked with an asterisk (*) indicate the instructions needed to do this. MVK and MVKH are used to load the addresses of an, xn, and y into registers A5, A6, and A7, respectively. These instructions must be executed in the order shown to load the lower 16 bits of the full 32-bit address first, followed by the upper 16 bits. These registers are used as pointers to load an and xn into registers A0 and A1, and to store y from register A4 (instructions marked with Δ). The C programming language notation * is used to indicate that a register is being used as a pointer.

Depending on the data type, one of the following load instructions can be used: LDB bytes (8 bits), LDH half-words (16 bits), or words (LDW 32 bits). Here, the data is assumed to be half-words. Loading/storage is performed by the .D1 unit, as .D units are the only units capable of interacting with data memory.

Note that pointers A5 and A6 must be post-incremented (C notation) so that they point to the next values for the next iteration of the loop. When registers are used as pointers, there are several ways to perform pointer arithmetic. These include pre- and post-increment/decrement options by a certain offset value, where the pointer is modified before or after it is used (for example, *A1 [disp] and *A1++ [disp]). Additionally, a pre-shift option can be performed without modifying the pointer (for example, *+A1 [disp]). The offset within square brackets specifies the number of data elements (depending on the data type), while the offset within parentheses specifies the number of bytes. These pointer offset options are listed in Figure 3.5 with some examples.

Syntax	Description	Modified pointer
*R	Pointer	No
*+R[disp]	+Pre-shift	No
*-R[disp]	– Pre-shift	No
*++R[disp] *--R[disp]	Pre-increment	Yes
	Pre-decrement	Yes
*R++[disp]	Post-increment	Yes
*R--[disp]	Post-decrement	Yes

[disp] specifies # elements - size in W, H or B
(disp) specifies # bytes
(a)



(b)

Fig. 3.5(a) Pointer offsets, (b) pointer examples
(Note: the instructions are independent and not sequential)

Finally, the MPY and ADD instructions within the loop perform the dot product operation. The MPY instruction is executed by the .M1 unit and ADD by the .L1 unit.

3.2 Implementation of the sum of products (SOP)

It has been shown that the SOP is the key element of most DSP algorithms.

Let's write the code for this algorithm and discover the architecture of the C6000 at the same time.

$$y = \sum_{n=1}^{N} a_n * x_n = a_1 * x_1 + a_2 * x_2 + \dots\dots\dots\dots a_N * x_N \qquad (2.24)$$

Two basic operations are required for this algorithm.

(1) Multiplication

(2) Addition

So let's implement the SOP algorithm! The implementation in this module will be done in assembly language.

Multiplying a1 by x1 is done in assembly using the following instruction:

MPY a1, x1, Y

This instruction is executed by a multiplier called ".M". The .M unit performs multiplications in the hardware:

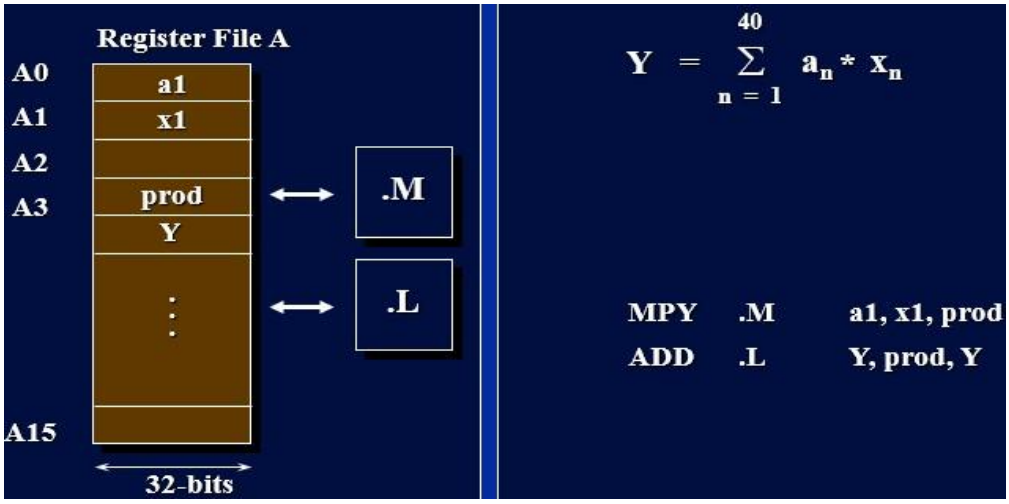
MPY .M a1, x1, Y

Noticed: the 16-bit by 16-bit multiplier provides a 32-bit result.

The 32-bit by 32-bit multiplier provides a 64-bit result.

MPY .M a1, x1, prod

ADD .L Y, prod, Y



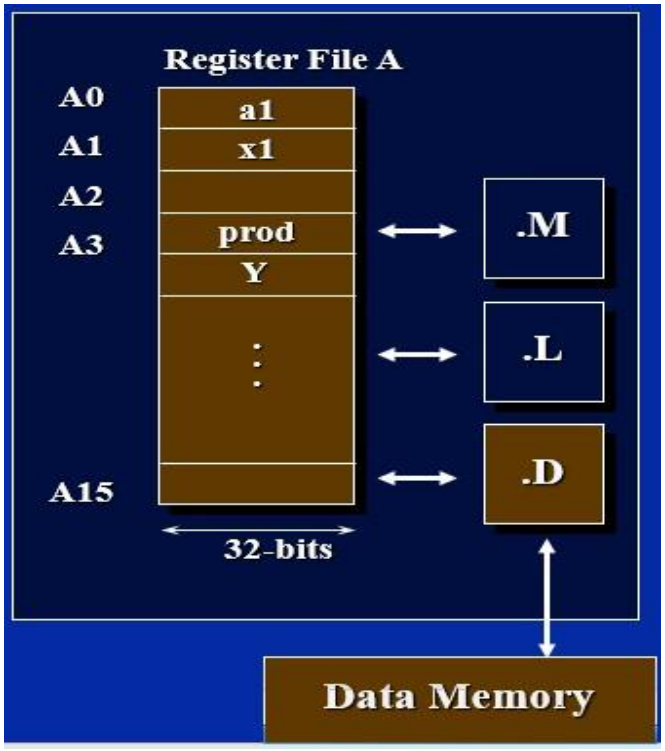
RISC processors such as the C6000 use registers to hold operands, so let's modify this code. We'll correct this by replacing a, x, prod, and Y with the registers shown above.

MPY .M A0, A1, A3

ADD .L A4, A3, A4

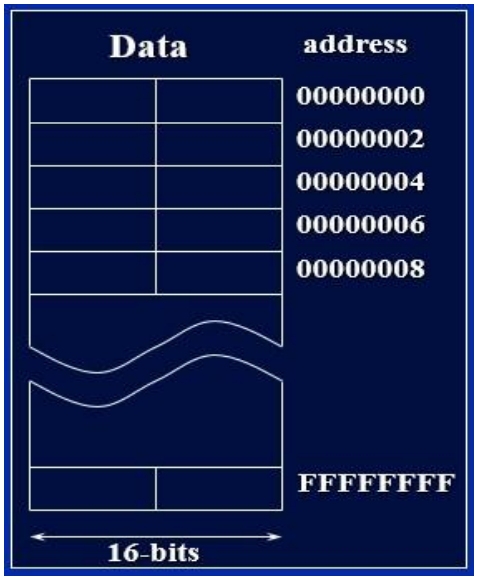
Registers A0, A1, A3, and A4 contain the values to be used by the instructions. The register file A contains 16 registers (A0-A15) with a width of 32 bits.

How are operands loaded into registers? Operands are loaded into registers by loading them from memory using the .D unit.



It should be noted at this point that the only way to access memory is via the .D drive. Which instruction(s) can be used to load operands from memory into registers? The LOAD instruction

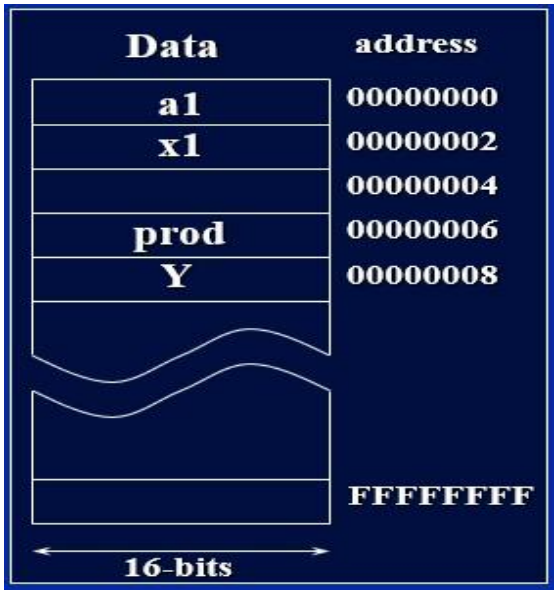
Before using the LOAD unit, you should know that this processor is byte-addressable, meaning that each byte is represented by a unique address. Addresses are also 32 bits wide.



The syntax for the load instruction is as follows:

LD *Rn, Rm

Where: Rn is a register that contains the address of the operand to be loaded and Rm is the destination register.



The question now is how many bytes will be loaded into the destination register?

The answer is that it depends on the instruction you choose:

LDB: loads one byte (8 bits)

LDH: loads half a word (16 bits)

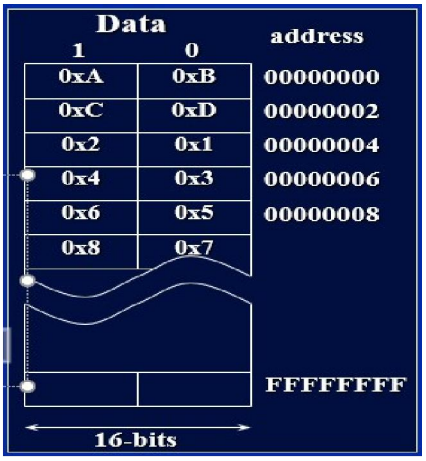
LDW: loads a word (32 bits)

LDDW: loads a double word (64 bits)

Noticed: LD alone does not exist.

Example:

If we assume that A5 = 0x4, then:
(1) LDB *A5, A7; gives A7 = 0x00000001
(2) LDH * A5, A7; gives A7 = 0x00000201
(3) LDW *A5, A7; gives A7 = 0x04030201
(4) LDDW *A5, A7: A6; gives A7: A6 = 0x0807060504030201



If the data is only accessible via the load instruction and the .D unit, how can we load the register pointer Rn in the first place?

The MVKL instruction will allow the movement of a 16-bit constant in a register as shown below:

MVKL.? a, A5

('a' is a constant or a label)

How many bits make up a complete address? 32 bits

So why doesn't the instruction allow a 32-bit shift?

(All instructions have a width of 32 bits)

To solve this problem, another instruction is available: MVKH

Finally, to move the 32-bit address to a register, we can use:

MVKL a, A5

MVKH a, A5

Always use MVKL then MVKH, see the following examples:

Example 1:

A5 = 0x87654321

MVKL 0x1234FABC, A5	MVKH 0x1234FABC, A5
A5 = 0xFFFFFABC (sign extension)	A5 = 0x1234FABC; OK

Example 2:

MVKH 0x1234FABC, A5	MVKL 0x1234FABC, A5
A5 = 0x12344321	A5 = 0xFFFFFABC; False

```
MVKL pt1, A5

MVKH pt1, A5

MVKL pt2, A6

MVKH pt2, A6
LDH .D *A5, A0

LDH .D *A6, A1

MPY .M A0, A1, A3
```

```
ADD .L A4, A3, A4
```

pt1 and pt2 point to certain locations in the data memory.

So far, we have only implemented the SOP for a single cycle, that is

$$Y = a1 * x1$$

Let's create a loop so that we can implement the SOP for N cycles.

With C6000 processors, there are no dedicated instructions such as block repeat. The loop is created using the B instruction.

What are the steps to create a loop?

Create a label that you connect to.

2. Add a branch instruction, B.

3. Create a loop counter.

4. Add an instruction to decrement the loop counter.

5. Make the branch conditional based on the value of the loop counter.

```
MVKL .S pt1, A5

MVKH .S pt1, A5

MVKL .S pt2, A6
```

```
                MVKH .S pt2, A6 MVKL .S
count, B0 loop LDH .D *A5, A0
                LDH .D *A6, A1
                MPY .M A0, A1, A3
                ADD .L A4, A3, A4
                 SUB .S B0, 1, B0
                 B.S loop
```

What is the syntax for making the statement conditional?

[Condition] Instruction Label for example. [B1] B

Loop

(1) The condition can be one of the following registers: A1, A2, B0, B1, B2.

(2) Any instruction can be conditional.

The condition can be inverted by adding the exclamation mark "!" as follows: [! condition]

of instruction Label for example:

[!B0] B loop ;branch if B0 = 0

[B0] B loop ;branch if B0 != 0

This code now performs the following operations:

a0 * x0 + a1 * x1 + a2 * x2 + ... + aN * xN

The A7 pointer has not been initialized.

```
                MVKL .S pt1, A5

                MVKH .S pt1, A5

                MVKL .S pt2, A6

                MVKH .S pt2, A6

                MVKL .S2 pt3, A7 (Pointer A7 is now initialized.)

                MVKH .S2 pt3, A7

               MVKL .S count, B0
loop           LDH .D *A5, A0

                LDH .D *A6, A1

                MPY .M A0, A1, A3

                ADD .L A4, A3, A4

                SUB  .S  B0,  1,  B0  [B0]  B  .S

loop

                STH .D A4, *A7
```

A4 is used as an accumulator, so it must be reset.

(MIPS = IPC * f * 106)...

3.3 Pipelined Control Unit

Generally, executing an instruction requires several steps. These steps are essentially fetching, decoding, and executing. If these steps are performed sequentially, not all processor resources, such as multiple buses or functional units, are fully utilized.

To increase throughput, DSP processors are designed to be pipelined. This means that the preceding steps are performed simultaneously. **Figure 3.6** illustrates the difference in processing time for three instructions executed on a serial or non-pipelined CPU and a pipelined CPU. As can be seen, a pipelined CPU requires fewer clock cycles to execute the same number of instructions.

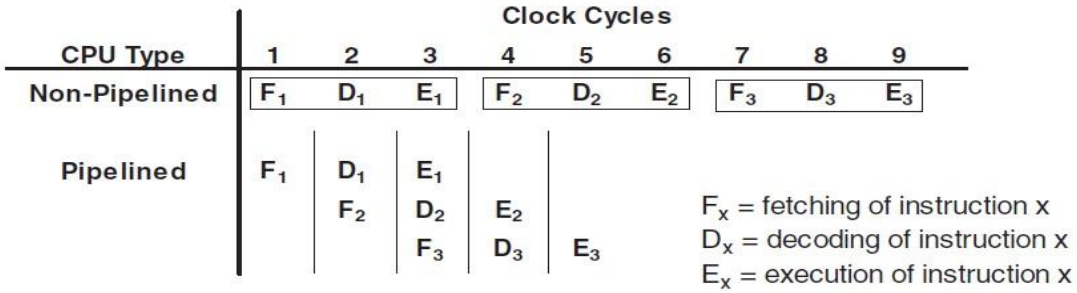


Fig. 3.6 Pipelined and non-pipelined control unit

On the C6x processor, fetching consists of four phases, each requiring one clock cycle.

This process involves generating the search address (F1), sending the address to memory (F2), waiting for the data (F3), and reading the opcode from memory (F4). Decoding consists of two phases, each requiring one clock cycle. These are sent to the appropriate functional units (designated D1) and decoding (D2).

Due to the delays associated with instruction multiplication (MPY - 1 delay), loading (LDx - 4 delays), and branching (B - 5 delays), the execution step can comprise up to six phases

(designated E1 to E6), accepting a maximum of 5 delays. Therefore, as shown in Figure 3.7, step F consists of four phases, step D of two, and step E of six possible substeps or phases.

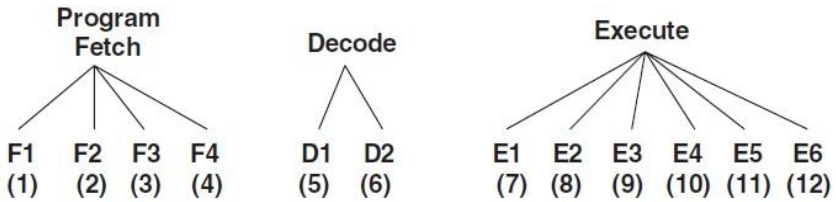


Fig. 3.7 The stages of the pipeline

When the result of one instruction is used by the next, an appropriate number of NOPs (no operations or delays) must be added after the multiplication (one NOP), loading (four NOPs or NOP 4), and branching (five NOPs or NOP 5) to allow the pipeline to function correctly. Therefore, for the example above to run on the C6x processor, the appropriate NOPs, as shown in Figure 3.8, must be added after the MPY, LDH, and B instructions.

```
        MVK.S1 40,A2 loop: LDH .D1 *A5++,A0
        LDH .D1 *A6++,A1
        NOP 4
        MPY .M1 A0,A1,A3
        NOP
        ADD .L1 A3,A4,A4
        SUB .L1 A2,1,A2
[A2] B .S1 loop NOP 5
        STH .D1 A4, *A7
```

Fig. 3.8 Pipelined code with inserted NOPs

Figure 3.9 illustrates an example of a pipeline situation that requires the addition of a NOP.

The plus signs indicate the number of substeps or latencies required to execute the instruction. In this example, it is assumed that the addition operation is performed before

one of its operands is made available from the previous multiplication operation, hence the need to add a NOP after the MPY. Later, we will see that, as part of code optimization, NOPs can be reduced or eliminated, leading to improved efficiency.

Search search Program	Deco ge	Execution						Term ined
F1- F4	D1- D2	E 1	E 2	E 3	E 4	E 5	E 6	
MPY								

MPY is wanted (fetched).

	MPY							
ADD								

MPY is decoded and ADD is fetched.

		MPY						
	ADD							

MPY is executed and ADD is decoded.

			MPY					
		ADD						

MPY is still running while ADD is also running.

				→	→	→	→	MPY
				→	→	→	→	ADD

The two instructions finish at the same time; the result of MPY is not used in the ADD instruction.

(a)

Search search Program	Deco ge	Execution						Term ined
F1-F4	D1-D2	E1	E2	E3	E4	E5	E6	
MPY								

MPY is wanted (fetched).

	MPY							
NOP								

MPY is decoded and NOP is searched for.

		MPY						
	NOP							
ADD								

MPY is executed, NOP is decoded and ADD is searched for.

			MPY					
		NOP						
	ADD							

MPY is still running while NOP blocks the pipeline and ADD is being decoded.

				→				MPY
		ADD						

MPY se termine, ADD est exécuté en utilisant le résultat de MPY.

				→				ADD

ADD is ending.

(b)

Fig. 3.9 (a) Multiplication then addition, and (b) need for NOP insertion.

3.4 Search Package (FP)

The C6x architecture is based on the Very Long Instruction Word (VLIW) architecture. In such an architecture, several instructions are captured and processed simultaneously. This is called a search packet (FP). (See **Figure 3.10**).

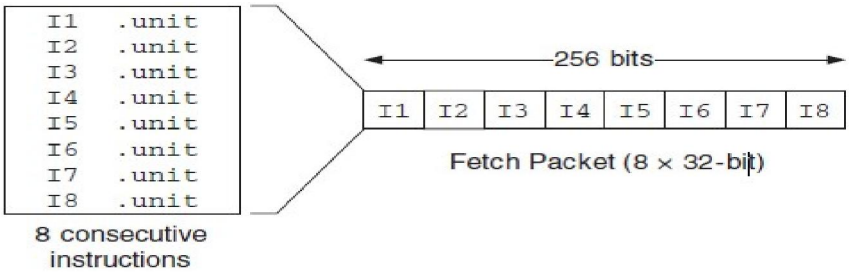


Fig. 3.10: C6x fetch packet: the C6x searches for eight 32-bit instructions in each cycle.

The C6x uses VLIW, allowing eight instructions to be captured simultaneously from on-chip memory onto its 256-bit program data bus. The original VLIW architecture was modified by TI (Texas Instruments) to allow multiple so-called execution packets (EPs) to be included in the same search packet, as shown in **Figure 3.11**. An EP is a group of parallel instructions. Parallel instructions are denoted by double-pipe symbols (||) and, as their name suggests, are executed together or in parallel.

The instructions of an EP move together through each stage of the pipeline. This VLIW modification is called VelociTI. Compared to VLIW, VelociTI reduces code size and increases performance when instructions reside on an external chip.

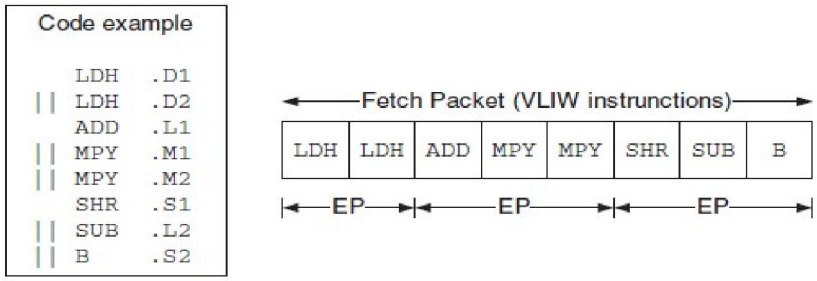


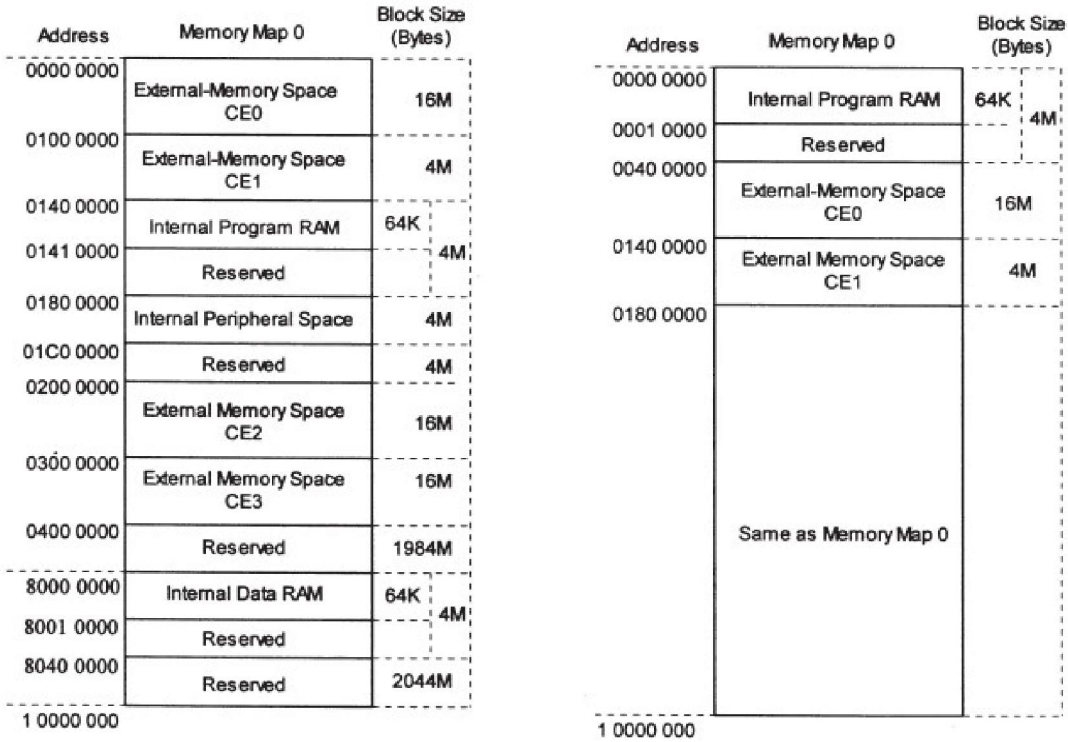
Fig. 3.11: A search package containing three execution packages.

Chapter 4: Memory Management

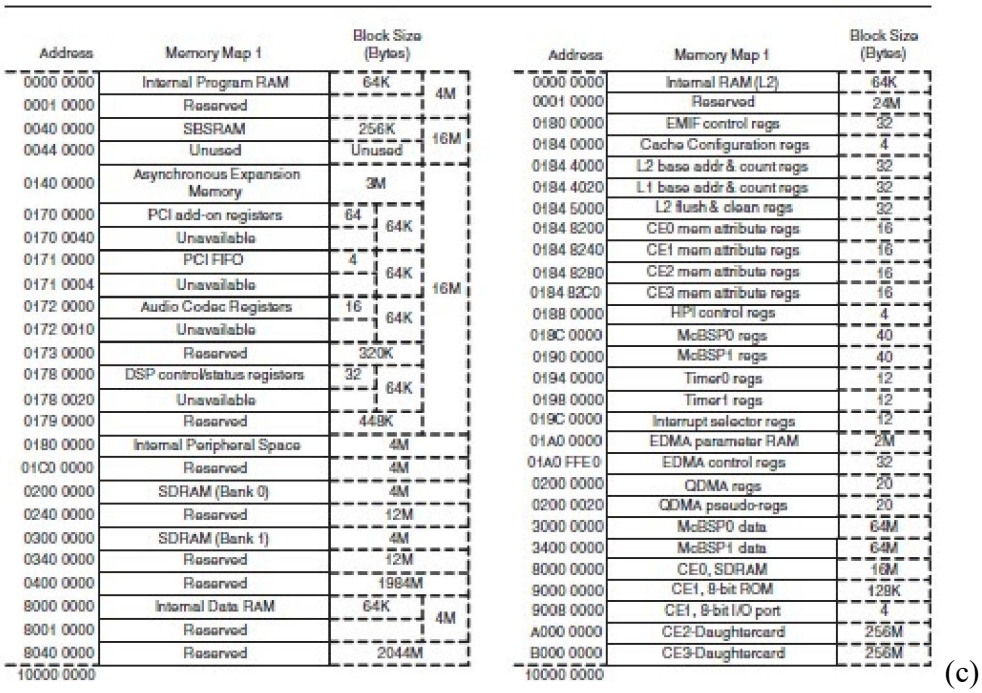
The external memory used by a DSP processor can be static or dynamic. Static memory (SRAM) is faster than dynamic memory (DRAM), but it is more expensive because it takes up more space on the silicon. DRAM also needs to be refreshed periodically.

A good compromise between cost and performance is achieved by using SDRAM (Synchronous DRAM). Synchronous memory requires synchronization,
unlike asynchronous memory, which does not do this.

Since the address bus is 32 bits wide, the total memory space is 2^{32} = 4 GB. On the EVM (EValuation Module = DSK), this space is divided, according to a memory card, into internal program memory (PMEM), internal data memory (DMEM), internal peripherals, and external memory spaces named CE0, CE1, CE2, and CE3. There are two memory card configurations: memory card 0 and memory card 1. **Figures 4.1(a)** and **4.1(b)** illustrate these two memory cards. On the DSK (DSP Starter Kit), there is no separation between the internal program and data memory. For the practical exercises in this handout, the EVM board is configured according to its memory card 1, as shown in **Figure 4.1(c)**, and the DSK board according to its memory card 1, as shown in **Figure 4.1(d)**.



(a) (b)



(d) (c)

Fig. 4.1: (a) C6x 0 memory card, (b) card 1, (c) EVM 1 card, and (d) DSK 1 card.

External memory ranges CE0, CE1, CE2, and CE3 support synchronous (SBSRAM, SDRAM) or asynchronous (SRAM, ROM, etc.) memory, accessible in bytes (8 bits), half-words (16 bits), or words (32 bits). On-chip devices and control registers are mapped to the memory space.

The internal data memory is organized into memory banks so that two loads or stores can be performed simultaneously. As long as data is accessed from different banks, no conflict occurs. However, if data is accessed from the same bank within an instruction, a memory conflict occurs and the CPU is blocked for a cycle.

If a program is stored in internal memory or on the chip, it should be executed from there to avoid the delays associated with accessing external memory or the chip. If a program is too large to fit in internal memory, most of its time-consuming portions should be placed in internal memory for efficient execution. For repetitive code, it is recommended to configure internal memory as a cache.

This allows access to external memory as rarely as possible, thus avoiding the delays associated with such access.

4.1 In-memory data alignment

The C6x allows addressing bytes, halfwords, or words. Consider a Word-format representation of memory, as shown in Figure 4.2. There are four byte boundaries, two halfwords (or shorts), and one word-per-word boundary. The C6x always accesses data on these boundaries according to the specified data type; that is, it always accesses aligned

data. When specifying an uninitialized variable section `.usect`, it is necessary to specify the alignment as well as the total number of bytes.

The examples shown in Figure 4.3 demonstrate the data alignment for constants and variables.

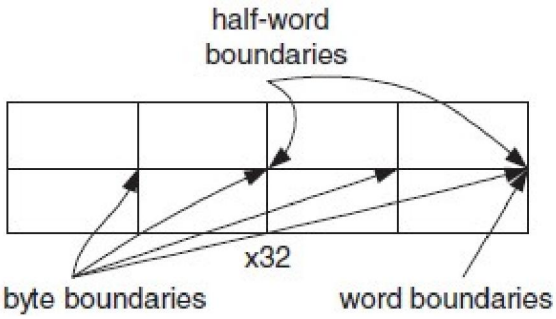


Fig. 4.2 Data limitations

Constants are automatically aligned

```
    .sect "my_const"
A   .byte   11h
B   .short  2222h
C   .int    33333333h
```

Variables need an alignment field

```
;label    .usect  "section",  #bytes,  alignment
d         .usect  "vars",     1,       1
ee        .usect  "vars",     2          ;byte alignment by default
ffff      .usect  "vars",     4,       4
g_array   .usect  "vars",     8,       2
```

22	22	--	11
33	33	33	33
--	e	e	d
ff	ff	ff	ff
g1		g0	
g3		g2	

Note 1: vars and `my_const` sections are assumed contiguous.

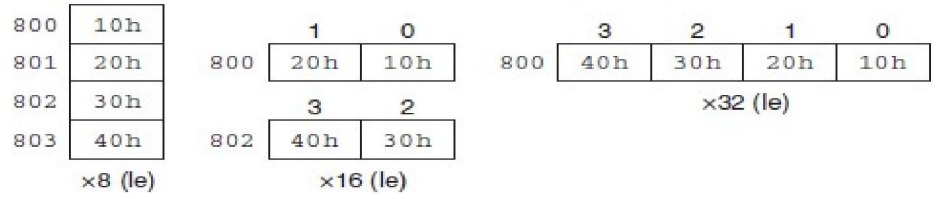
Note 2: First declare words, then shorts and bytes to save memory space.

Fig. 4.3 Examples of aligning constants and variables.

Data in memory can be organized in little-endian or big-endian format. Little-endian (le) means that the least significant byte is stored first. Figure 4.4(a) shows the storage of .int 40302010h in little-endian format for byte, half-word, and word access addressing. In big-endian (be) format, illustrated in Figure 4.4(b), the most significant byte is stored first. Little-endian is the format normally used in most applications.

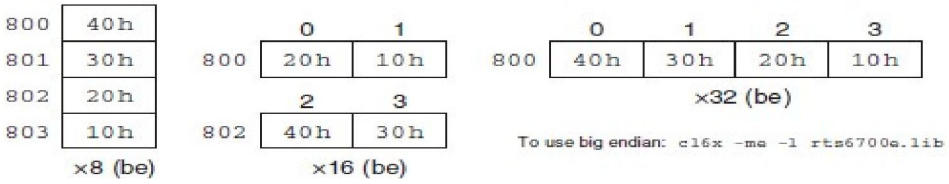
Additional examples of data alignment are shown in **Figure 4.4(c)**, based on the little-endian data format appearing in **Figure 4.4(a)**.

Little endian (default - LSB first)

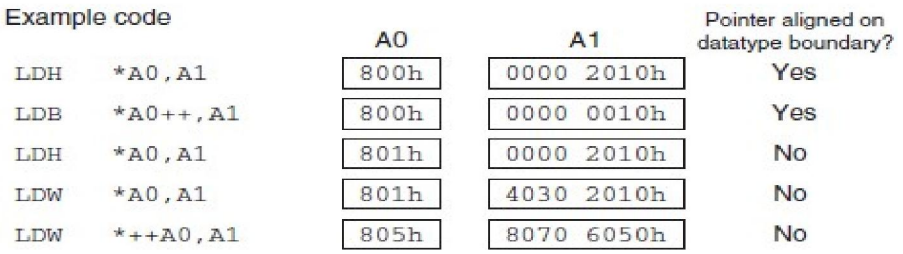


(has)

Big endian (MSB first)



(b)

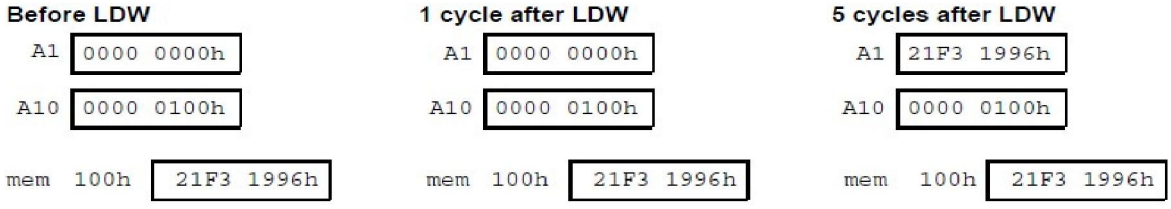


(c)

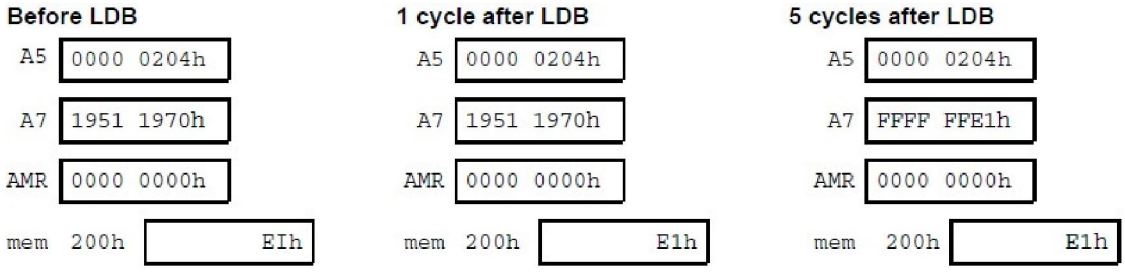
Fig. 4.4 (a) Little endian, (b) big endian, and (c) more examples of data alignment.

4.2 Examples of alignments

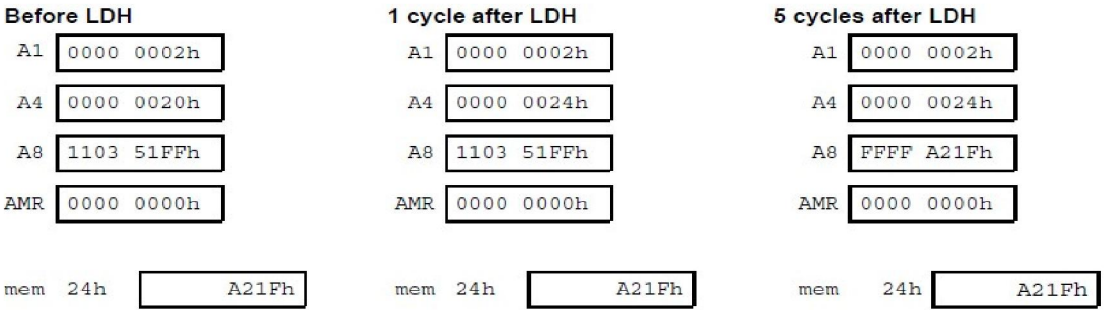
Example 1: LDW .D1 *A10,A1



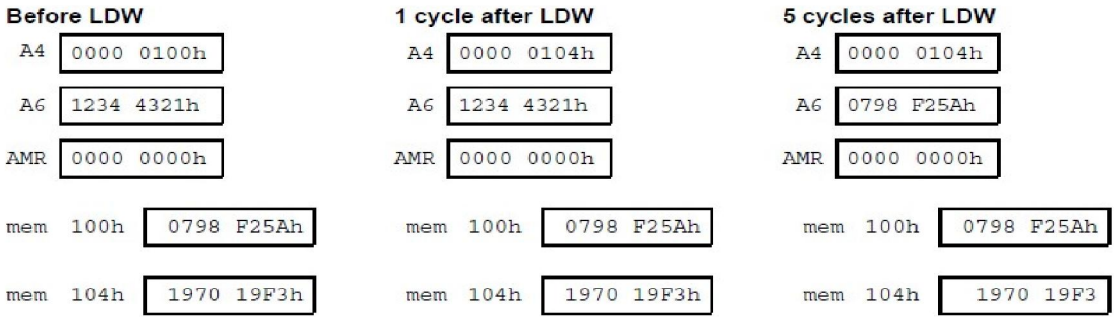
Example 2: LDB .D1 *–A5[4],A7



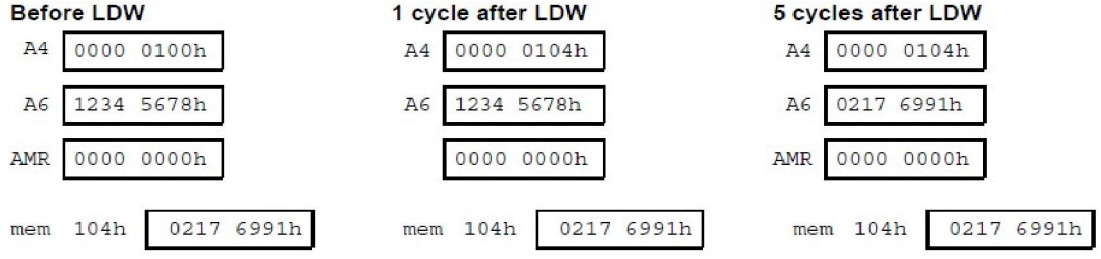
Example 3: LDH .D1 *++A4[A1],A8



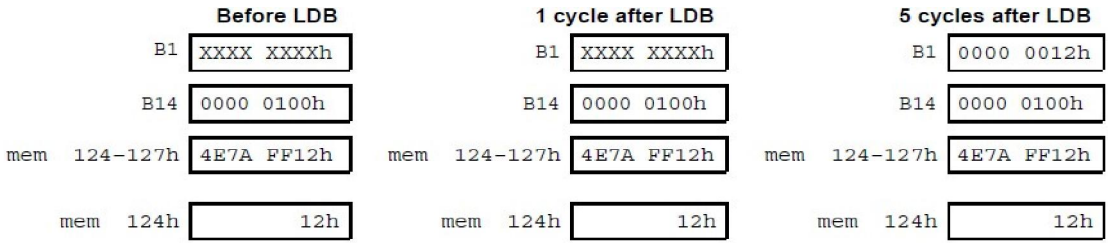
Example 4: LDW .D1 *A4,++[1],A6



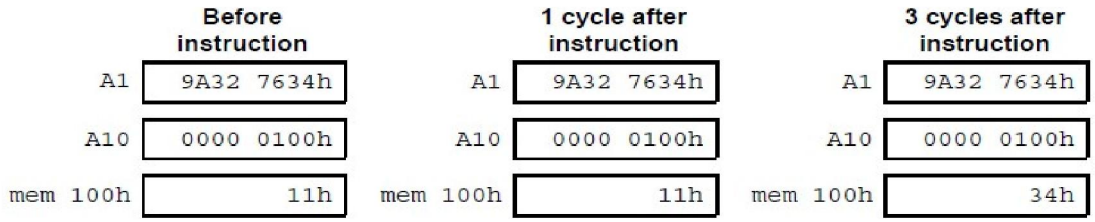
Example 5: LDW .D1 *++A4(4),A6



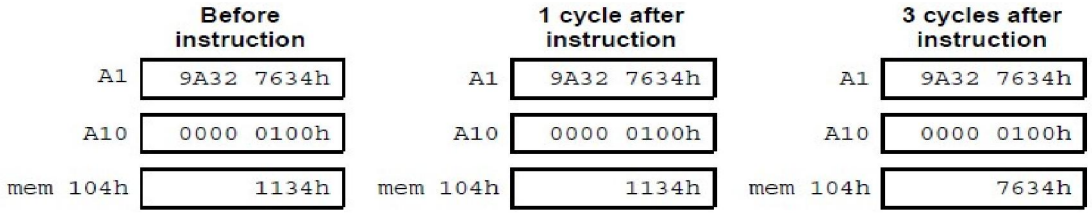
Example 6: LDB .D2 *+B14[36],B1



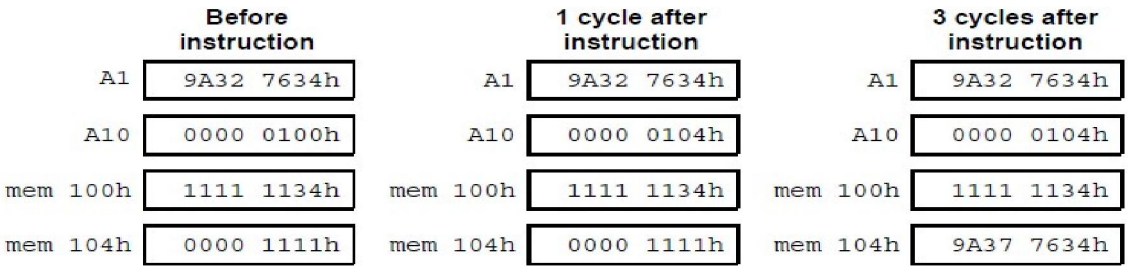
Example 1: STB .D1 A1,*A10



Example 2: STH .D1 A1,*+A10(4)



Example 3: STW .D1 A1,***A10[1]



Example 4: STH .D1 A1,*A10—[A11]

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 2634h	9A32 2634h	9A32 2634h
A10	0000 0100h	0000 009Ch	0000 009Ch
A11	0000 0004h	0000 0004h	0000 0004h
mem 9Ch	0000h	0000h	0000h
mem 100h	0000	0000h	2634h

Exercise 5: STB .D2 B1,*+B14[40]

	Before instruction	1 cycle after instruction	3 cycles after instruction
B1	1234 5678h	1234 5678h	1234 5678h
B14	0000 1000h	0000 1000h	0000 1000h
mem 1028h	42h	42h	78h

Chapter 5: Development Environment: 'Code Composer Studio' (CCS)

5.1 Software and hardware requirements

The software tool required to generate TMS320C6x executable files is called Code Composer Studio (CCS). CCS integrates assembler, linker, compiler, simulator, and debugger utilities. In the absence of a target board, which allows running an executable file on a real C6x processor, the simulator can be used to verify code functionality using data already stored in a data file. However, when using the simulator, an interrupt service routine (ISR) cannot be used to read signal samples from a signal source. To process signals in real time on a real C6x processor, a DSP starter kit (DSK) or an evaluation module (EVM) board is required for code development. Recommended test equipment includes a function generator, oscilloscope, microphone, boom box, and cables with audio connectors.

A DSK card can easily be connected to a host PC via its parallel or USB port. The signal interface with the DSK card is via its two standard audio jacks. An EVM card must be installed in a full-length PCI slot inside a host PC. The signal interface with the EVM card is via its three standard audio jacks.

Knowledge of the C programming language is assumed to be required to complete the practical exercises.

5.2 Software Tools

Most DSP processors can be programmed in C or assembly language. While writing programs in C requires less effort, the resulting efficiency is typically lower than that of programs written in assembly language.

Efficiency means having the fewest instructions or instruction cycles possible while making maximum use of the chip's resources.

In practice, the process begins with coding in C to analyze the behavior and functionality of an algorithm. Then, if the required processing rate is not achieved using the C compiler's optimizer, the time-consuming parts of the C code are identified and converted to assembly language, where the entire code is rewritten. In addition to C and assembly, the C6x allows writing code in linear assembly. Figure 5-1 illustrates the code efficiency versus coding effort for three types of source files on the C6x: C, linear assembly, and hand-optimized assembly. As can be seen, linear assembly offers a good compromise between code efficiency and coding effort.

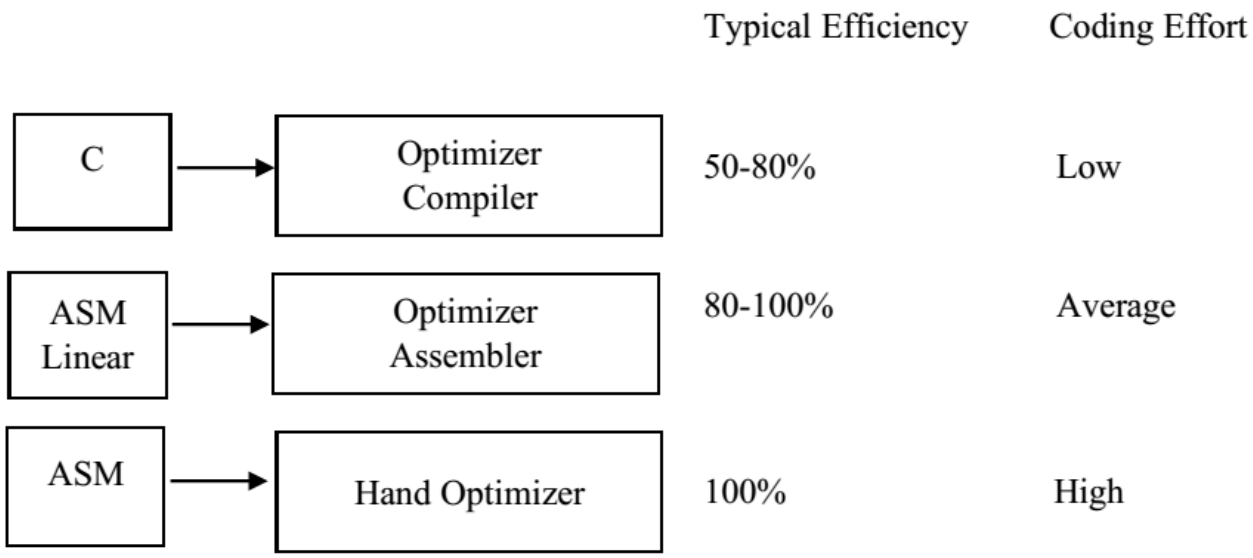
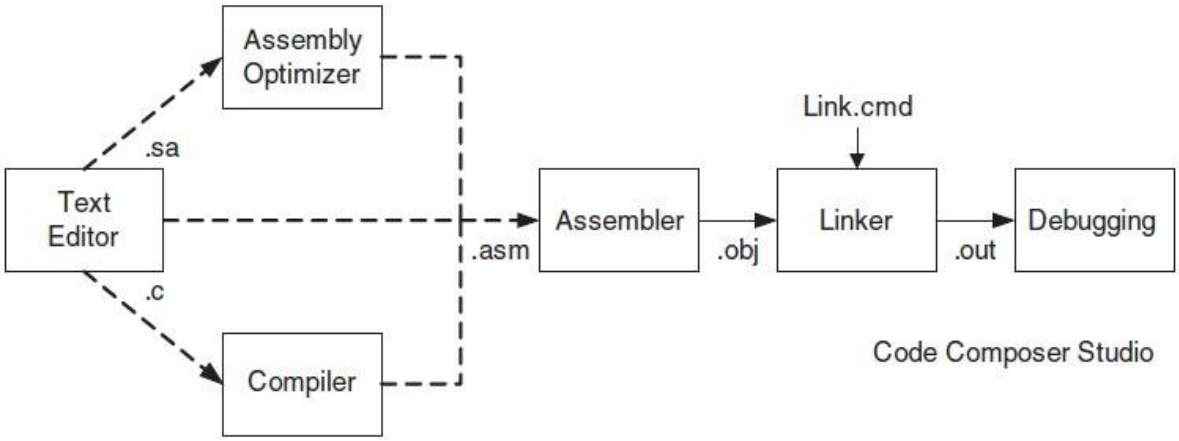


Fig. 5.1 Code efficiency and coding effort.

Figure 5.2 shows the steps to follow to go from a source file (extension .c to C, .asm for assemblers and .sa for linear assembler) to an executable file (extension .out).



.c = C source file

.sa = linear assembly source file

.asm = assembly source file

.obj = object file

.out = executable file

.cmd = linker command file

Fig. 5.2:C6x software tools

Figure 5.3 lists the .c and .sa versions of the dot product example to show what they look like. The assembler is used to convert an assembler file into an object file (extension .obj). The assembler optimizer and compiler are used to convert a linear assembler file and a C file, respectively, into an object file. The linker is used to combine object files, as specified by the linker command file (extension .cmd), into an executable file. All assembly, linking, compilation, and debugging steps have been integrated into an integrated development environment (IDE) called Code Composer Studio (CCS or CCStudio). CCS provides an easy-to-use graphical user environment for creating and debugging C and assembler code on various target DSPs.

```
void main()
{
    y = DotP((int *) a, (int *) x, 40);
```

```
}

int DotP(int *m, int *n, short count)
{ int sum, i;
    sum = 0;

    for(i=0;i<count;i++)
        sum += m[i] * n[i];

    return(sum);
  }
```

(a)

```
            .title "dot product"
            .def dotp
            .sect code
 dotp: .proc A4,B4,A6,B6,A8,B3
             .reg a, ai, b,bi,r,prod,sum,c,ci,i;
            MV A4,c
            MV B4,b
            MV A6,a
            MV B6,r MV
A8,i loop: .trip 40
            LDH *a++, ai
            LDH *b++,bi
            MPY ai,bi,prod
            SHR prod,15,sum
           ADD ai,sum,ci
           STH ci, *c++
      [i] SUB i,1,i
      [i] B loop
            .endproc B3
```

(b)

Fig. 5.3. (a) .c (b) .sa version of an example of a dot product.

5.3 C6x DSK / EVM Target Cards

When a DSK or EVM card is available, an executable file can be run on a real C6x processor. In the absence of such cards, CCS can be configured to simulate the execution process. As shown in Figure 5.4, the C6713 DSK card is a DSP system that includes a C6713 DSP chip operating at 225 MHz with 4/4/256 KB of memory for the L1D data

cache/L1P program cache/L2 memory, respectively, 8 MB of onboard SDRAM (Synchronous Dynamic RAM), 512 KB of flash memory, and a 16-bit AIC23 stereo codec with a sampling rate of 8 kHz to 96 kHz. The DSK C6416 card includes a C6416 DSP chip operating at 600 MHz with 16/16/1024 KB of memory for the L1D data cache / L1P program cache / L2 cache, respectively, 16 MB of onboard SDRAM, 512 KB of flash memory, and an AIC23 codec. The DSK C6711 card includes a C6711 DSP chip operating at 150 MHz with 4/4/64 KB of memory for the L1D data cache / L1P program cache / L2 cache, 16 MB of onboard SDRAM, 128 KB of flash memory, a 16-bit AD535 codec with a fixed 8 kHz sampling rate, and a daughterboard interface to which a PCM3003 audio daughterboard can be connected to change the sampling rate.

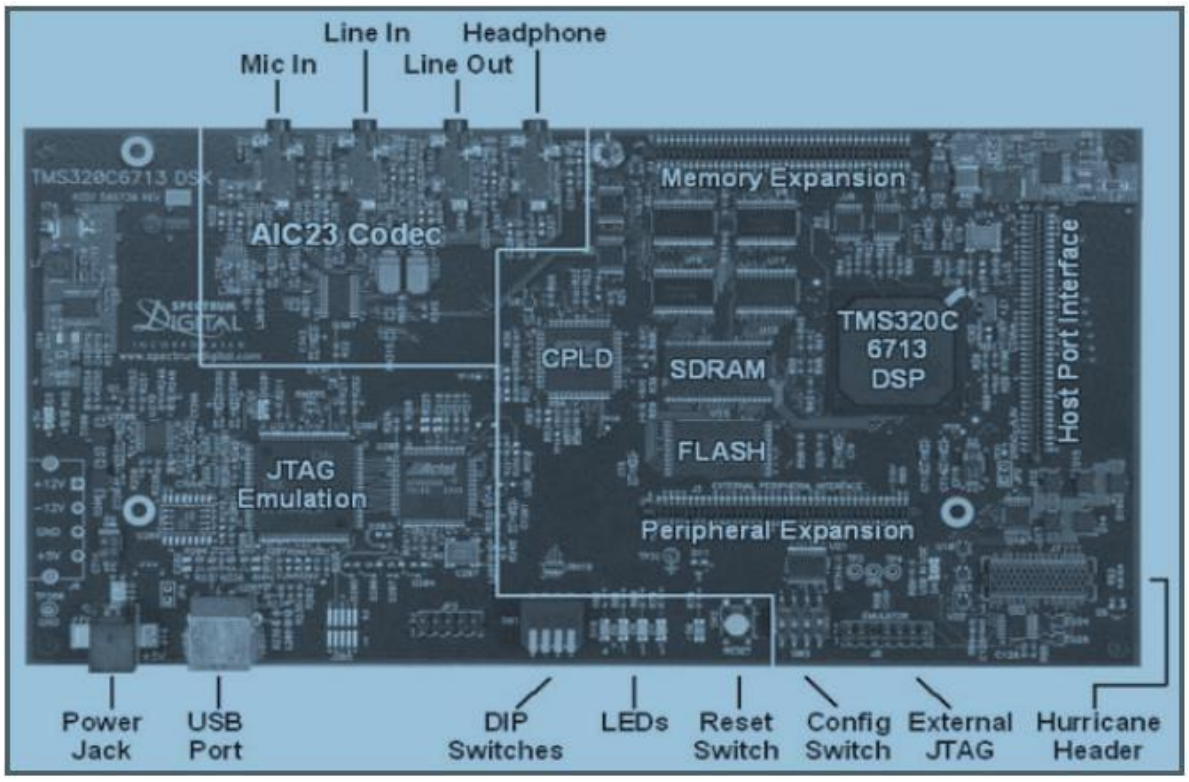
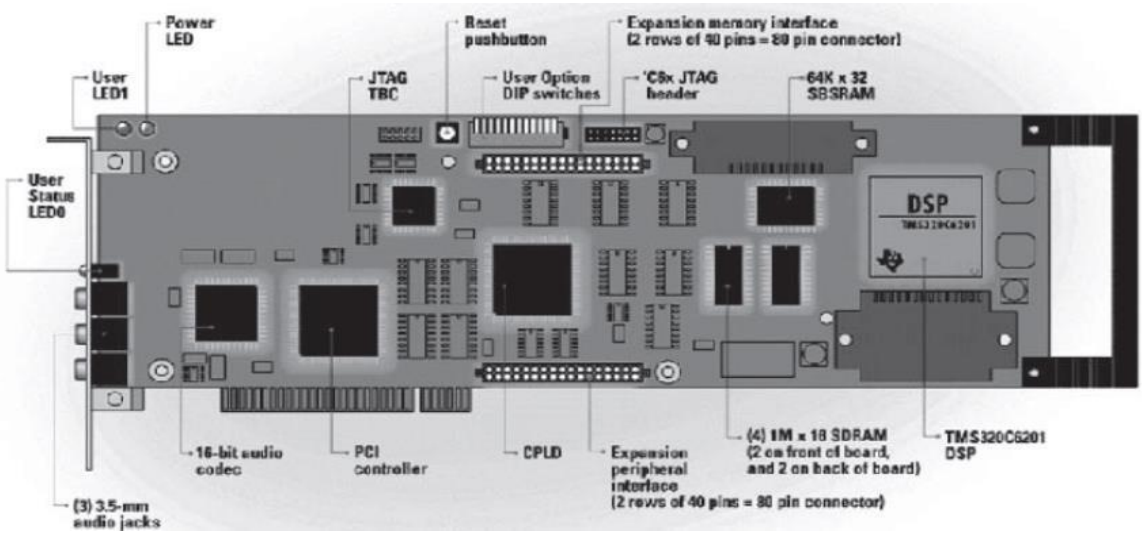
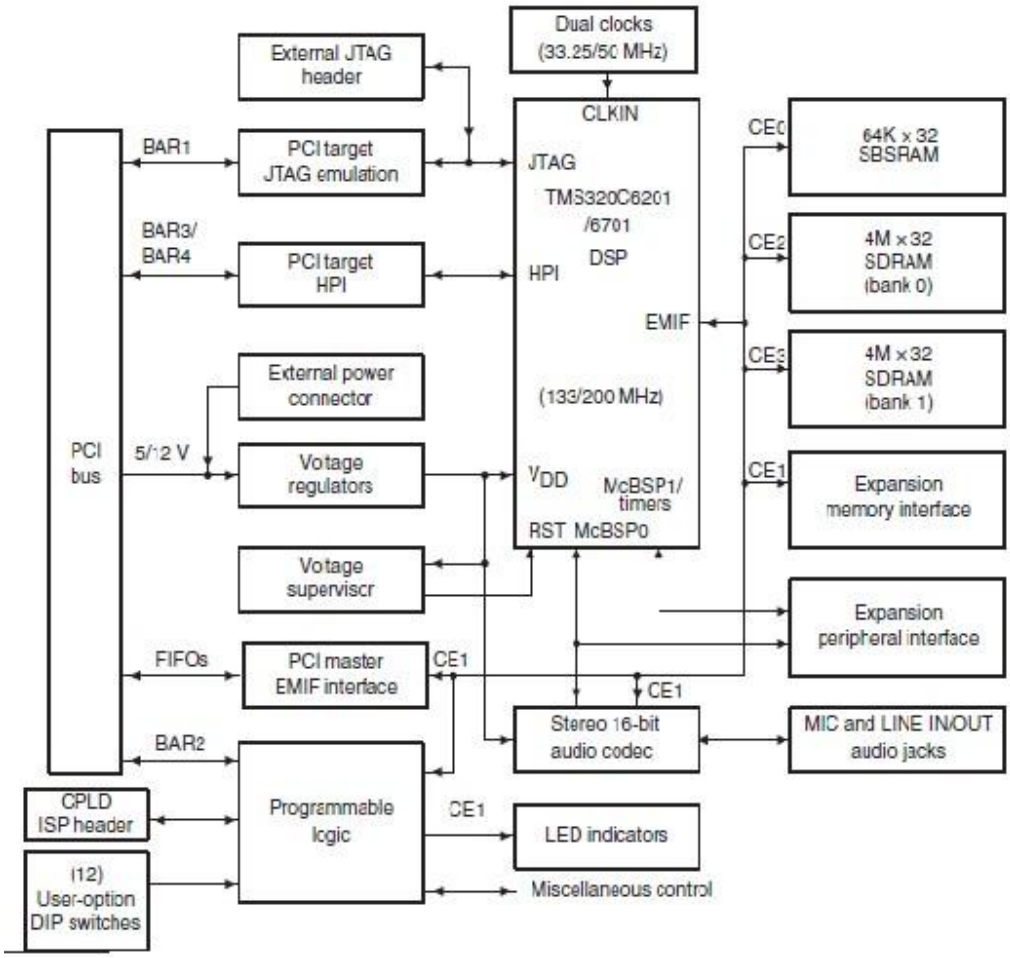


Fig. 5.4. DSK card C6713.

As shown in Figure 5.5(a), the EVM C6701/C6201 board is a DSP system that includes a C6701 (or C6201) chip, external memory, A/D capacitors, and PC host interface components. The EVM board's functional diagram is shown in Figure 4-5(b). The board has a 16-bit CS4231A codec with a sampling rate that can be changed from 5.5 kHz to 48 kHz.



(a)



(b)

Fig. 5.5. (a) EVM C6201/C6701 card, (b) its functional diagram.

The memory residing on the EVM card consists of 32 MB of SDRAM operating at 100 MHz and 256 KB of SBSRAM (Synchronous Static Burst RAM) operating at 133 MHz, a faster but more expensive memory compared to SDRAM. A voltage regulator on the board is used to provide 1.8 V or 2.5 V for the C6x core, 3.3 V for its memory and peripherals, and 5 V for the audio components.

5.4 Assembler file

Similar to other assembly languages, the C6x assembler consists of four fields: label, instruction, operands, and comment. (See Figure 3.3.) The first field is the label field.

Labels must begin in the first column and must start with a letter. A label, if present, indicates a name assigned to a specific memory location that contains an instruction or data. A mnemonic or directive constitutes the instruction field. It is optional for the instruction field to include the functional unit that executes that particular instruction. However, to make code more understandable, assigning functional units is recommended. If a functional unit is specified, the data path should be indexed by 1 for side A and 2 for side B. A parallel instruction is indicated by a double pipe symbol (||), and a conditional instruction by a register appearing in parentheses within the instruction field. As the name operand suggests, the operand field contains the arguments of a statement. Statements require two or three operands. Except for store statements, the destination operand must be a register. One of the source operands must be a register, and the other must be either a register or a constant. Following the operand field is an optional comment field which, if specified, must begin with a semicolon (;).

5.5 Guidelines

Directives are used to specify sections of assembler code and to declare data structures. It's important to note that assembler instructions appearing as directives do not produce any executable code. They simply control the assembler's execution process. Some widely used assembler directives are:

The .sect "name" directive defines a section of code or data named "name".

Directive .int, .long or .word, which reserves 32 bits of memory initialized to a value.

The .short or .half directive, which reserves 16 bits of memory initialized to a value.

The .byte directive, which reserves 8 bits of memory initialized to a value.

Note that in the TI Common Object File Format (COFF), the directives .text, .data, and .bss are used to indicate code, initialized constant data, and uninitialized variables, respectively. Other commonly used directives include .set, to assign a value to a symbol; .global or .def, to declare a symbol or module as global so that it can be recognized externally by other modules; and .end, to signal the termination of assembler code. The .global directive acts as a .def directive for defined symbols and as a .bss directive for other symbols.

.ref for undefined symbols.
At this point, it is worth mentioning that the C compiler creates various sections indicated by the directives .text, .switch, .const, .cinit, .bss, .far, .stack, .sysmem, .cio. Figure 5.6 lists some common compiler sections.

Section name description
.text Code
.switch Tables for switching instructions
.const Global and static string literals
.cinit Initial values for global/static variables
.bss Global and static variables
.far Global and static declared further
.stack Stack (local variables)
.sysmem Memory for mallc fcns (heap)
.cio Buffers for stdio functions

Fig. 5.6. Common sections of the compiler.

5.6 Compilation Utility

The CCS generation function can be used to perform the entire compilation, assembly, and linking process in a single step by enabling the cl6x utility and specifying the correct options. The following command demonstrates how this

This utility is used in CCS to create the source files file1.c, file2.asm, and file3.sa: cl6x -gs file1.c file2.asm file3.sa -z -o file.out -m file.map -l rts6700.lib

The -g option adds debugger-specific information to the object file for debugging purposes. The -s option provides a C and assembler interlist. For file1.c, the C compiler is called; for file2.asm, the assembler; and for file3.sa, the assembly optimizer (linear assembler). The -z option calls the linker, placing the executable code in file.out if the -o option is used. Otherwise, the default file a.out is created.

The -m option provides a map file (file.map), which includes a list of all section addresses, symbols, and labels. The -l option specifies the runtime support library rts6700.lib for file linking on the C6713 processor. **Table 5.1** lists some frequently used options.

Options	Description	AVERAGE
-mv6700	Generates code C67x (C62x is the default value)	Comp/Asm
-g	Enables symbolic debugging at the src level	Comp/Asm
--mg	Allows minimal debugging to enable profiling	Compiler
-s	Writes C instructions to the assembler list	Compiler
-o	Invoke optimizer (-o0, -o1, -o2 / -o, -o3)	Compiler
-pm	Combine all C source files before compiling them	Compiler
-mt	No aliases used	Compiler
-ms	Reduces the code size (-ms0 / -ms, -ms1, -ms2)	Compiler
-z	Call the link editor	Link editor
-o	Output file name	Link editor
-m	Map file name	Link editor
-c	C variables Auto-Init (-cs disables auto-init)	Link editor
-L	Link libraries -in (small -L)	Link editor

Table 5.1. Common compiler options

The compiler allows calling four levels of optimization using -o0, -o1, -o2, and -o3. Large-scale debugging and optimization cannot be performed simultaneously because they

conflict; that is, in debugging, information is added to improve the debugging process, while in optimization, information is minimized or removed to improve code efficiency. In principle, the optimizer modifies the flow of the C code, making program debugging very difficult.

As Figure 4-9 shows, a good programming approach would be to first verify that the code works correctly using the compiler without optimization.

(option -gs). Next, use full optimization to generate efficient code (option -o3). It is recommended to take an intermediate step in which optimization is performed without interfering with source-level debugging (option -go). This intermediate step can recheck the code's functionality before performing full optimization. Note that full optimization can modify memory locations outside the scope of the C code. These memory locations should be declared "volatile" to avoid compilation errors.

1. Compile without optimization. (Get the code running!) cl6x -g -s file.c -z
2. Compile with a certain optimization.
   (Check the code's functionality again)
   cl6x -g -o file.c -z
3. Compile with all optimizations.
   (Generate efficient code) cl6x -o3 -pm file.c -z

Fig. 5.7. Principal programming

To further optimize C code, it is recommended to use intrinsics whenever possible. Intrinsics are functions similar to mathematical functions that are part of the runtime support library. Intrinsics allow the C compiler to access hardware directly while

preserving the C environment. For example, instead of using the multiply * operator in C, the intrinsic _mpy() can be used to tell the compiler to use the C6x instruction MPY. Figure 5.8 shows the intrinsic version of the C dot product. A list of C6x intrinsic features is provided in Appendix A of [1]122.

```
short DotP(int *m, int *n, short count)
{
        short i, productl, producth, suml = 0, sumh = 0;

        for(i=0; i<count; i++)
        {
                productl = _mpy(m[i],n[i]); // _mpy intrinsic
producth = _mpyh(m[i],n[i]); // _mpyh intrinsic suml += productl;
                sumh += producth;
        }
        suml += sumh;
return(suml); }
```

Fig. 5.8. Intrinsic version of the C code for the dot product (dot.product)

5.7 Code Initialization

All programs begin with a reset initialization code. Figure 5.9 illustrates both the C and assembly versions of a typical reset initialization code. This initialization is designed to start at a previously defined initial location. Upon power-up, the system always goes to the reset location in memory, which normally includes a branch instruction at the beginning of the code to be executed. The reset code shown in Figure 5.9 brings the program counter to a globally defined memory location named init or _c_int00.

"**ASM » « C »**

vectors.asm

```
        .ref init
        .sect “vectors” rst
MVK .s2 init,B0 MVKH
.s2 init,B0
        B .s2 B0
        NOP
        NOP
        NOP
        NOP
        NOP
```

cvectors.asm

```
        .global _c_int00
        .sect “vectors” rst
B _c_int00
        NOP; additional NOP's
        NOP; to create a
        NOP ;fetch packet
        NOP
        NOP
        NOP
        NOP
```

Fig. 5.9. Reset code

As shown in Figure 5.10, when writing in assembly language, initialization code is required to create initialized data and variables, and to copy initialized data into corresponding variables. Initialized values are specified using the .byte, .short, or .int directives. Uninitialized variables are specified using the .usect directive.

The first, second, and third arguments of this directive specify the section name, the size in bytes, and the data alignment in bytes, respectively. Before calling the main function or subroutine, another part of the initialization code is usually required to configure the registers and pointers and to move the data to the appropriate memory locations.

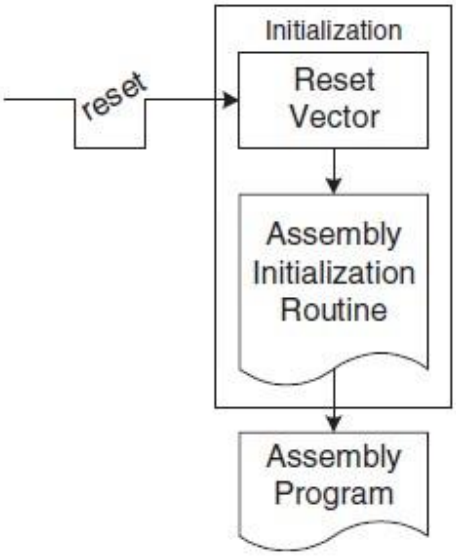


Fig. 5.10. Initializing the assembler

Figure 5.11 shows the initialization code for the dot product example (dotproduct), in which initialized data values appear for three initialized data arrays labeled table_a, table_x, and table_y. Additionally, three variable sections named a, x, and y are declared. The second part of the initialization code copies the initialized data into the corresponding variables. The configuration code for calling the dot product routine is also shown in this figure.

```
        .def    init
        .ref    dotp

;Data initialization
;Initialize tables

        .sect   "init_tables"

table_a .short  40,39,38,37,36,35,34,33,32,31,30,29,28,27   ;Initialize table_a array with values
        .short  26,25,24,23,22,21,20,19,18,17
        .short  16,15,14,13,12,14,10,9,8,7,6,5,4,3,2,1
table_x .short  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15         ;Initialize table_x array with values
        .short  16,17,18,19,20,21,22,23,24,25,26,27,28,29
        .short  30,31,32,33,34,35,36,37,38,39,40
table_y .short  0                                           ;table_y = 0

;Variable declaration
a       .usect  "var", 80, 2                                ;define variables
x       .usect  "var", 80, 2
y       .usect  "var", 2, 2

;Initialization to copy data into variables
        .sect   "init_code"
init    mvk     .s1     table_a, A0     ;move address of table_a to register A0
        mvkh    .s1     table_a, A0
        mvk     .s2     a,B0            ;move address of a to register B0
        mvkh    .s2     a,B0
        mvk     .s2     40,B1           ;create a counter in register B1, B1=40
loop_a  ldh     .d1     *A0++,A1        ;load an element from the address pointed by A0 into A1
        sub     .l2     B1,1,B1         ;decrement counter
        nop             3
        sth     .d2     A1,*B0++        ;store the element to address pointed by B0
   [B1] b       .s2     loop_a          ;branch back to loop_a
        nop             5               ;required latency
init_x  mvk     .s1     table_x, A0     ;move address of table_x into register A0
        mvkh    .s1     table_x, A0
        mvk     .s2     x, B0           ;move address of x into register A0
        mvkh    .s2     x, B0
        mvk     .s2     40, B1          ;create a counter
loop_x  ldh     .d1     *A0++,A1        ;load an element from the address pointed by A0 into A1
        sub     .l2     B1,1,B1         ;decrement counter
        nop             3
        sth     .d2     A1,*B0++        ;store element to address pointed by B0
   [B1] b       .s2     loop_x          ;branch back to loop_x
        nop             5
init_y  mvk     .s1     table_y, A0     ;repeat above procedure for table_y
        mvkh    .s1     table_y, A0
        mvk     .s2     y, B0
        mvkh    .s2     y, B0
        ldh     .d1     *A0, A1
        nop             4
        sth     .d2     A1, *B0
```

(a)

```
;Setup for calling dotp

start   mvk     .s1     a,A4            ;move a into register A4
        mvkh    .s1     a,A4
        mvk     .s2     x,B4            ;move x into register B4
        mvkh    .s2     x,B4
        mvk     .s1     40,A6           ;create a counter in A6, A6=40
        b       .s1     dotp            ;branch to routine dotp
        mvk     .s2     return, B3      ;store return address in B3
        mvkh    .s2     return, B3
        nop             3

;return from dotp here
return  mvk     .s1     y, A0           ;move y into register A0
        mvkh    .s1     y, A0
        sth     .d1     A4, *A0         ;store the result of dotp (returned in A4) to y

;infinite loop
end     b       .s1     end             ;infinite loop
        nop             5
```

(b)

```
;dotp
        .def    dotp

;A4 = &a, B4 = &x, A6 = 40 (iteration count) , B3 = return address

dotp    mv              A6,B0       ;move A6 to B0 (third argument passed from calling function)
        zero            A2          ;zero the sum register A2

loop    ldh     .d1     *A4++,A5    ;load an element from the location pointed by A4 into A5
        ldh     .d2     *B4++,B5    ;load an element from the location pointed by B4 into B5
        nop             4
        mpy     .mlx    A5,B5,A5    ;A5=B5*A5
        nop
        add     .l1     A2,A5,A2    ;A2 += A5
   [B0] sub     .l2     B0,1,B0     ;decrement counter B0
   [B0] b       .s1     loop        ;branch back to loop
        nop             5

        mv              A2,A4       ;move result in A2 to return register A4
        b       .s2     B3          ;branch back to calling address stored in B3
        nop             5
```

(c)

Fig. 5.11. (a) Initialization code for the dot product example, (b) configuration code to call the dot product routine, and (c) dot product routine.

Regarding C coding, the C compiler uses boot.c from the runtime support library to perform initialization before calling main(). The -c option enables boot.c to automatically initialize variables. This is illustrated in **Figure 5.12**.

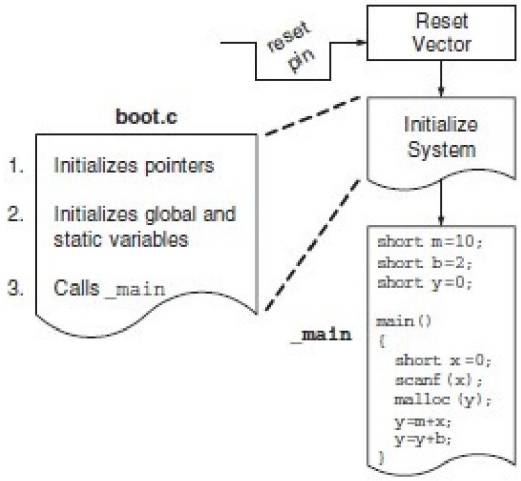


Fig. 5.12. Initializing the C language

5.8 Code Composer Studio (Lab1)

Code Composer Studio™ (CCStudio or CCS) is a useful integrated development environment (IDE) that provides an easy-to-use software tool for creating and debugging programs. In addition, it allows for real-time analysis of application programs.

Figure 5.13 shows the phases associated with the CCS code development process. During configuration, CCS can be set up for different target DSP boards (e.g., C6711 DSK, C6416 DSK, C6701 EVM, C6xxx Simulator). The version used throughout this manual is based on CCS version 2.2.

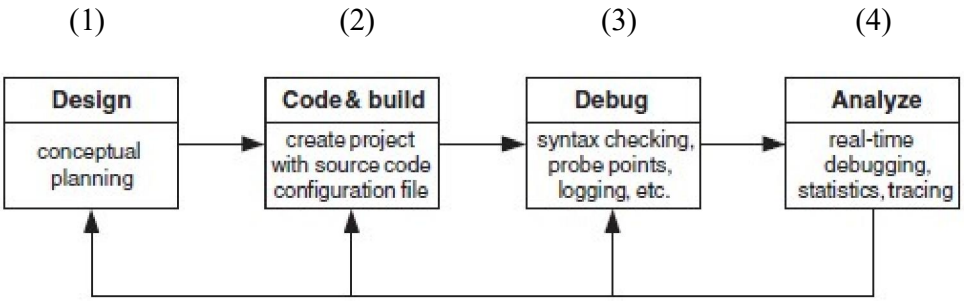


Fig. 5.13. CCS code development process.

(1) Design: Conceptual planning

(2) Code & Construction: Create a project with a source code configuration file

(3) Debugging: syntax checking, probe points, logging, etc.

(4) Analysis: real-time debugging, statistics, tracing

CCS provides a file management environment for creating application programs. It includes a built-in editor for editing C and assembly files. For debugging purposes, it provides breakpoints, data monitoring and graphing capabilities, a profiler for comparative analysis, and probe points for streaming data to and from the target DSP. This tutorial introduces the basic CCS features necessary for creating and debugging an application program.

This lab demonstrates how a simple multi-file algorithm can be compiled, assembled, and linked using CCS. First, multiple data values are written consecutively to memory. Then, a pointer is assigned to the beginning of the data so that it can be treated as an array. Finally, simple functions are added in C and assembly to illustrate how function calls work. This method of placing data in memory is easy to use and can be employed in applications where constants need to be in memory, such as filter coefficients and FFT twist factors. Problems related to debugging and comparative analysis are also addressed in this lab.

5.9 Project Creation (Lab 1.1)

Let's consider all the files needed to create an executable file; that is, the source files .c (c), .asm (assembler), .sa (linear assembler), a linker command file .cmd, a header file .h, and the appropriate library files .lib. The CCS code development process begins with creating

a so-called project to easily integrate and manage all these files required to generate and run an executable. The Project View panel is located on the left side of the window.

CCS provides a simple mechanism for doing this. In this panel, a project file (with the .prj extension) can be created or opened to contain not only the source and library files, but also the compiler, assembler, and linker options for generating an executable file. Therefore, it is not necessary to type command lines for compilation, assembly, and linking, as was the case with the original software development tools.

To create a project, choose the Project → New menu item in the CCS menu bar. This brings up the Project Creation dialog box, as shown in Figure 5.14. In the dialog box, navigate to the working folder, which in this case is assumed to be C:\ti\myprojects, and type a project name in the Project Name field. Then click the Finish button for CCS to create a project file called lab1.prj. All the files needed to create an application must be added to the project.

Fig. 5.14. Creation of a new project.

CCS provides a built-in editor for creating source files. Some of the editor's features include color syntax highlighting, marking C blocks in parentheses and curly braces, matching

parentheses to curly braces, control indentation, and search/replace/find capabilities. It's also possible to add files to the project from Windows Explorer using drag and drop. An editor window appears when you select File → New → Source File. For this lab, let's enter the following assembler code into the editor window:

```
.sect ".mydata"
.short 0
.short 7
.short 10
.short 7
.short 0
.short -7
.short -10
.short -7
.short 0
.short 7
```

This code consists of declaring 10 values using `.short` directives. Note that any of the in-memory data allocation directives can be used to do the same. Assign a section named `.mydata` to the values using a `.sect` directive. Save the created source file by choosing the File → Save menu item. This brings up the Save As dialog box, as shown in Figure 5.15. In the dialog box, go to the Save as type field and select Assembly Source Files (* Select .asm from the dropdown list. Then, go to the File name field and type initmem.asm. Finally, click Save to save the code to an assembly source file named initmem.asm.

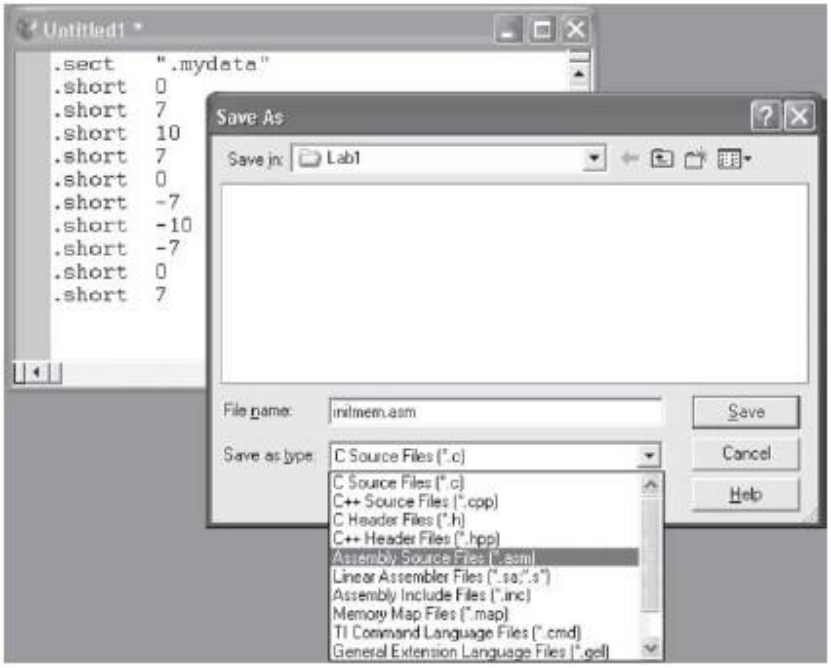


Fig. 5.15. Creating a source file

In addition to the source files, a linker command file must be specified to create an executable file and to conform to the memory specifics of the

The target DSP on which the executable file will run. A linker batch file can be created by choosing File → New → Source File. For this lab, let's use the batch file shown in Figure 5.16.

This file can also be downloaded from the supplied CD-ROM. This linker batch file is configured for the DSK memory card. Since our intention is to place the array of values defined in initmem.asm into memory, a space that will not be overwritten by the compiler should be selected. The external memory space CE0 can be used for this purpose. Let's assemble the data at memory address 0x80000000 (0x denotes hex) located at the beginning of CE0. To do this, assign the section named .mydata to MYDATA by adding .mydata> MYDATA to the SECTIONS section of the linker batch file, as shown in Figure

5.16. Save the linker window to a linker batch file by choosing File → Save or pressing Ctrl+S. The Save As dialog box will appear. Access the Save as type field and select TI Command Language Files (*.cmd) from the drop-down list. Then type lab1.cmd in the File name field and click Save.

Now that the source file initmem.asm and the linker command file lab1.cmd are created, they need to be added to the project for assembly and linking. To do this, choose the Project menu item → Add Files to Project. This brings up the Add Files to Project dialog box. In the dialog box, select initmem.asm and click the Open button. This adds initmem.asm to the project. To add lab1.cmd, choose Project → Add Files to Project. Then, in the Add Files to Project dialog box, set the Files of type to Linker Command File (*.cmd) so that lab1.cmd appears in the dialog box. Now, select lab1.cmd and click the Open button. In addition to the initmem.asm and lab1.cmd files, the execution support library file must also be added to the project. To do this, choose Project → Add Files to Project, navigate to the compiler's library folder (here assumed to be the default C:\ti\c6000\cgtools\lib), select Object and Library Files (*.o*, *.l*) in the Files of type box, then select rts6700.lib and click Open. If you are using the TMS320C6211 DSP fixed point, select rts6200.lib instead. For debugging purposes, let's use the following empty C shell program. Create a C source file named main.c, enter the following lines, and add main.c to the project in the same way as described above.

```
 MEMORY
{
        I RA M: o = 00000000h l = 00100000h
       MY DATA: o = 80000000h l = 00000100h
       C E0: o = 80000100h l = 000FFF00h
}

SECTIONS
```

```
{
      .cinit > IRAM
      .text > IRAM
     .stack > IRAM
     .bss > IRAM
    .const > IRAM
   .data > IRAM
   .far > IRAM
  .switch > IRAM
  .sysmem > IRAM
 .tables > IRAM
 .cio > I RAM
 . my data > MYDATA
}
```

Fig. 5.16. Link editor command file for lab 1.

For debugging purposes, let's use the following empty C shell program. Create a C source file called main.c, enter the following lines, and add main.c to the project in the same way as described above.

```
#include <stdio.h>
void main()
{
    printf("BEGIN\n");
printf("END\n"); }
```

After adding all source files, the command file, and the library file to the project, it's time to build the project or create an executable file for the target DSP. To do this, select the Project → Build menu item. CCS compiles, assembles, and links all the project files. Messages about this process are displayed in a panel at the bottom of the CCS window. Once the build process is complete without errors, the lab1.out executable file is generated. It's also possible to perform incremental builds—that is, to rebuild only the files that have changed since the last build—by selecting the Project → Rebuild menu item. The CCS window provides shortcut buttons for frequently used menu options, such as Build and Rebuild All.

Although CCS offers default build options, these can be modified by choosing Project → Build Options. For example, to change the executable file name to test.out, choose Project → Build Options, click the tab

Linkerfrom the links in the Build Options window and type test.out in the Output Filename (-o) field, as shown in Figure 5.17a.

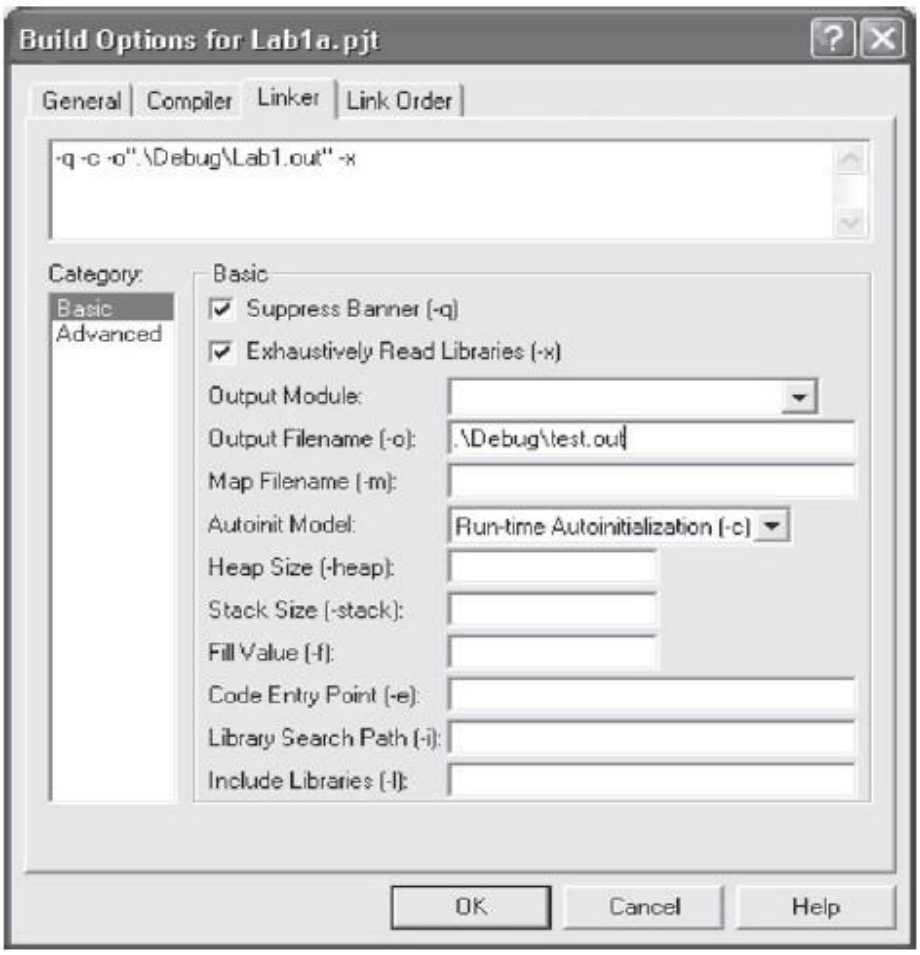
Note that the linker's command file will include test.out when you click on this window.
All compiler, assembler, and linker options are set via the Project → Build Options menu item. Among the many options of the

In the compiler shown in Figure 5.17b, pay particular attention to the optimization level options. There are four optimization levels (0, 1, 2, 3), which control the type and degree of optimization. The level 0 optimization option simplifies the control flow graph, allocates variables to registers, eliminates unused code, and simplifies expressions and statements. Level 1 optimization performs all the level 0 optimizations, removes unused assignments, and eliminates local common expressions. Level 2 optimization performs all the level 1 optimizations, plus software pipelining, loop optimizations, and loop unwinding. It also eliminates global common subexpressions and unused assignments. Finally, level 3 optimization performs all the level 2 optimizations, removes all functions that are never called, and simplifies functions with return values that are never used. It also calls small functions inline and reorders function declarations.

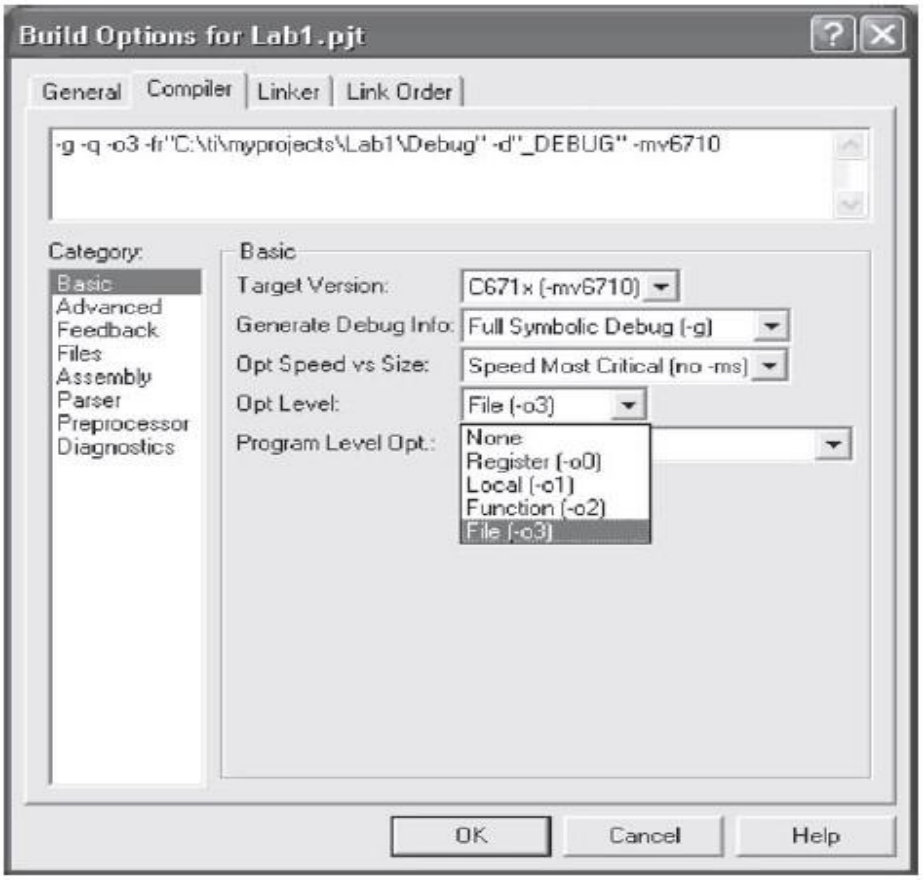
Note that in some cases, debugging is not possible due to optimization. Therefore, it is recommended to debug your program first to ensure it is logically correct before performing any optimization. Another important compiler option is the Target Version option. When the application targets the TMS320C6711/6713 floating-point DSK target

DSP, go to the Target Version field and select 671x (-mV 6710) from the dropdown list. For the TMS320C6416 fixed-point DSK target DSP, select C64xx (-mv 6400). For the EVM target

TMS320C6701, select C670x.



(a)



(b)

Fig. 5.17:(a) Build options and (b) compiler options.

A common mistake when writing initmem.asm is to type the `.sect` and `.short` directives in the first column. Since only labels can start in the first column, this will cause an assembler error. When a compilation error message appears, click Stop Build and scroll up in the Build area to see the syntax error message. Double-click the red text that describes the location of the syntax error. Note that the initmem.asm file opens, and your cursor appears on the line that caused the error (see Figure 5.18). After correcting the syntax error, the file should be saved, and the project rebuilt.

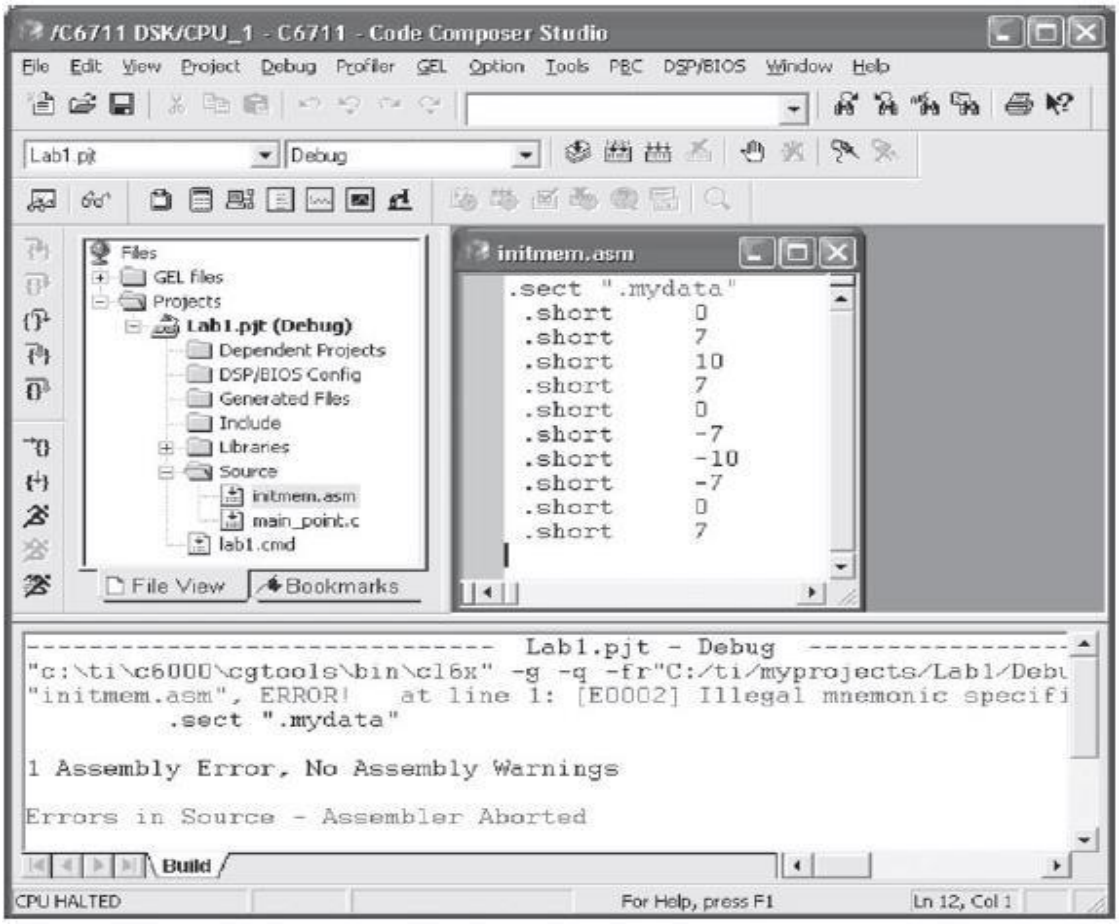


Fig. 5.18. Construction error

5.10 Debugging Tools (Lab 1.2)

Once the build process is complete without any errors, the program can be loaded and run on the target DSP. To load the program, choose File → Load Program, select the newly rebuilt lab1.out program, and click Open. To run the program, choose the Debug → Run menu item. You should see BEGIN and END appear in the Stdout window.

Now, let's check if the array of values is assembled in the specified memory location. CCS allows you to view the contents of memory at a specific location.

To display the contents of memory at 0x80000000, select View → Memory from the menu. The Memory Window Options dialog box will appear. This dialog box allows you to

specify various attributes of the Memory window. Go to the Address field and enter 0x80000000. Then, select 16-bit Signed Int from the Format field's drop-down list and click OK. A Memory window will appear as shown in Figure 5.19. The contents of the CPU, Device, DMA, and Serial Port registers can also be displayed by selecting View → Registers → Core Registers, for example.

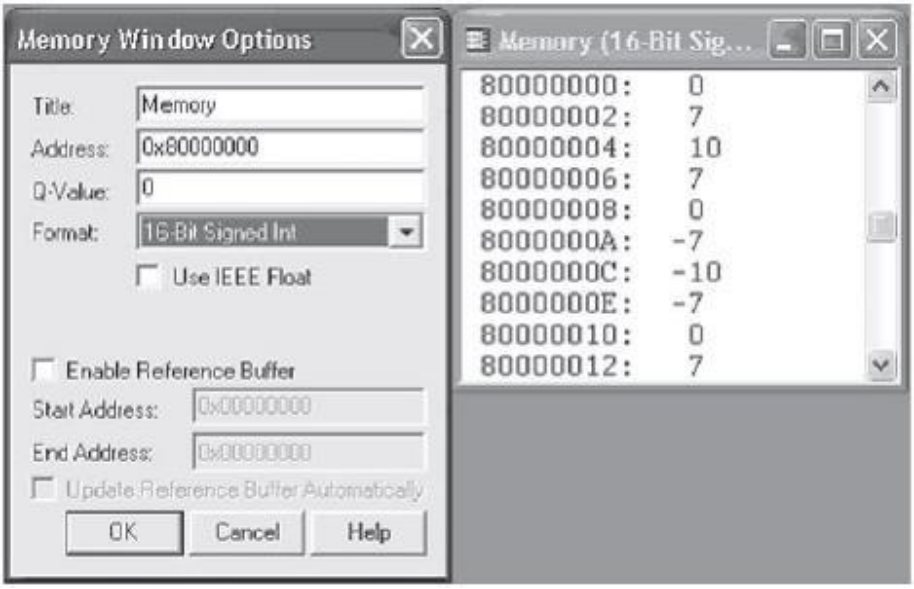


Fig. 5.19. Memory window options dialog box and Memory window.

There is another way to load data from a PC file into DSP memory. CCS provides a probe point capability, so a data stream can be moved from the PC's host file to the DSP or vice versa. To use this feature, a probe point must be defined in the program by placing the mouse cursor on the line where a data stream should be transferred and clicking the Probe Point button. Then, choose File → File I/O to bring up the File I/O dialog box. Click the Add File button and select the data file to load. Now, the file must be connected to the probe point by clicking the Add Probe Point button. In the Probe Point field, select the

probe point to make it active, and then connect the probe point to the PC file via File In:… in the Connect To field. Click the button

In the placeholder then click the OK button. Finally, enter the memory location in the Address field and the number of data points in the Length field. Note that a global variable name can be used in the Address field. The probe point capability is frequently used to simulate the execution of an application program in the absence of live signals. A valid PC file must have the correct file header and extension. The file header must conform to the following format:

MagicNumber Format StartingAddress PageNum Length

MagicNumber is set to 1651. The format indicates the format of the samples in the file : 1 for hexadecimal, 2 for integer, 3 for long, and 4 for float. StartingAddress and PageNum values are determined by CCS when a data stream is saved to a PC file. The length indicates the number of samples in memory. A valid data file must have the .dat extension.

A graphical display of data often provides better feedback on a program's behavior. CCS provides a signal analysis interface for monitoring a signal or data. Let's display the array of values at 0x80000000 as a signal or time graph. To do this, select View → Graph → Time/Frequency to display the Graph Property Dialog. The field names appear in the left column. Navigate to the Start Address field, click it, and type 0x80000000. Next, navigate to the Acquisition Buffer Size field, click it, and enter 10. Finally, click DSP Data Type, select a 16-bit signed integer from the drop-down list, and then click OK. A graph window appears with the selected properties. This is shown in Figure 5.20. You can modify any of these settings from the graph window by right-clicking, selecting Properties, and adjusting

the properties as needed. Properties can be updated at any time during the debugging process.

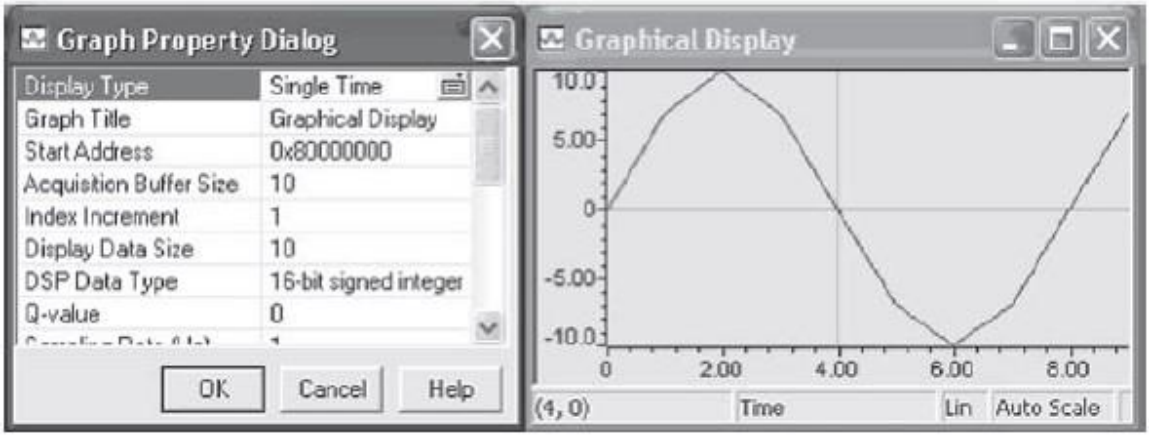


Fig. 5.20. Graph Property Dialog box and Graphical Display window.

To assign a pointer to the beginning of the assembled memory space, the memory address can be entered directly into a pointer. It is necessary to cast the pointer to a shorthand type because the values are of this type. The following code can be used to assign a pointer to the beginning of the values and iterate through them to print them to the Stdout window:

#include <stdio.h> void main() { int i; short *point;

```
   point = (short *) 0x80000000;
printf("BEGIN\n"); for(i=0;i<10;i++)
   { printf("[%d] %d\n",i, point[i]); }
printf("END\n");

}
```

Instead of creating a new source file, we can edit the existing main.c file by double-clicking it in the Project View panel, as shown in Figure 5.21. This will display the main.c source file in the right-hand pane of the CCS window. Then, enter the code and rebuild it. Before running the executable, be sure to reload the lab1.out file. Running this file should allow

you to see the values in the Stdout window. Double-clicking in the Project View panel provides an easy way to view any source or command file for review or modification.

During program development and testing, it's often necessary to check the value of a variable while the program is running. This can be done using breakpoints and watch windows. To view the pointer values in main.c before and after the pointer is assigned, choose File → Reload Program to reload the program. Then, double-click main.c in the Project View panel. You might want to maximize the window to see more files in one place. Next, place your cursor on the line that says point = (short *) 0x80000000 and press F9 to set a breakpoint. To open a watch window, choose View → Watch Window from the menu bar. This will open a Watch Window with local variables listed on the Watch Locals tab. To add a new expression to the watch window, select the Watch 1 tab, and then type the point (or any expression you want to examine) in the Name column. Next, choose Debug → Run or press F5. The program will stop at the breakpoint, and the Watch Window will display the pointer value. This is the value before the pointer was set to 0x80000000. By pressing F10 to skip ahead or using the shortcut button, you should be able to see the value 0x80000000 in the Watch Window.

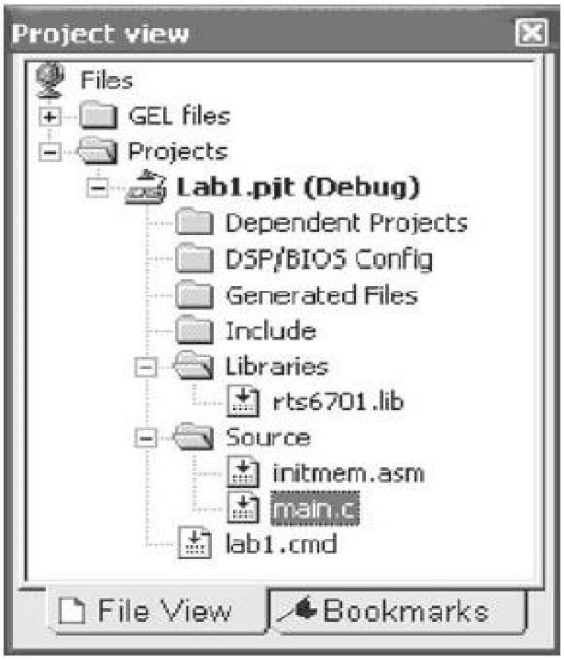


Fig. 5.21. Project View panel

To add a simple C function that adds the values, we can simply pass the pointer to the array and have an integer return type. For now, the concern isn't how the variables are passed, but rather how long it takes to perform the operation.

The following simple function can be used to add the values and return the result:

```
#include <stdio.h>
void main()
{ int i,ret; short
*point;
      point = (short *) 0x80000000;
printf("BEGIN\n"); for(i=0;i<10;i++)
      { printf("[%d] %d\n",i, point[i]); }
ret = ret_sum(point,10); printf("Sum =
%d\n",ret); printf("END\n");
```

```
}
int ret_sum(const short* array,int N)

{ int count, sum;
sum = 0;
   for(count=0; count < N; count++)
sum += array[count]; return(sum);

}
```

As part of the debugging process, it's usually necessary to compare or time the application program. In this lab, let's determine the execution time of the `ret_sum()` function. To perform this comparative analysis, reload the program and choose Profiler → Start New Session. This will display

Profile Session Name Type a session name, MySession by default, and then click OK. The Profile window, displaying code size and cycle statistics, will be docked at the bottom of CCS. Resize this window by dragging its edges or undocking it so you can see all the columns. Now, right-click the code inside the function you want to compare, and then choose Profile Function → in… Session. The function name will be added to the list in the Profile window. Finally, press F5 to run the program. Examine the number of cycles shown in Figure 5.22 for ret_sum(). It should be approximately 732 cycles (the exact number may vary slightly). This is the number of cycles required to execute the ret_sum() function.

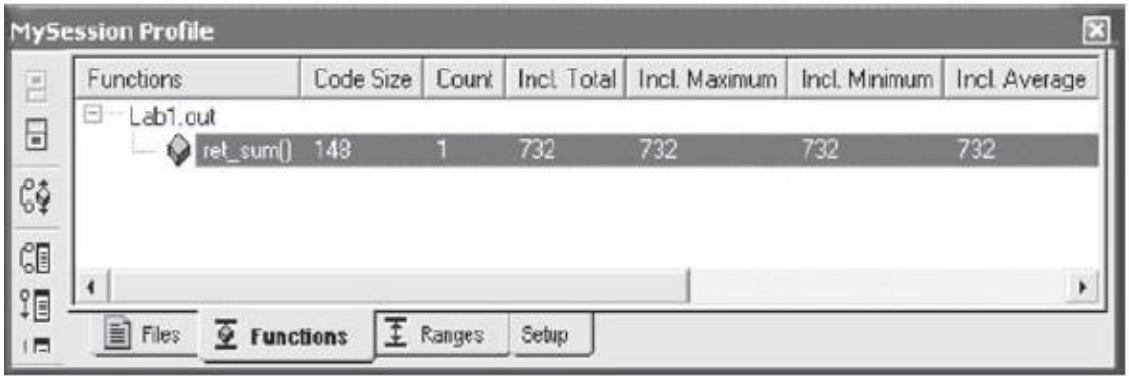


Fig. 5.22: Profile window.

There is another way to compare the codes using breakpoints. Double-click the main.c file in the Project View panel and choose View → Mixed Source /

ASMTo list the assembled instructions corresponding to the lines of C code, set a breakpoint on the calling line by placing the cursor on the line that reads `ret = ret_sum(point, 10)`, and then press F9 or double-click Selection Margin on the left side of the editor. Set another breakpoint on the next line, as shown in Figure 5.23. Once the breakpoints are set, choose Profiler → Enable Clock to activate the profile clock. Next, choose Profiler → View Clock to bring up a window displaying the Profile Clock. Now, press F5 to run the program. When the program stops at the first breakpoint, reset the clock by double-clicking in the inner area of the Profile Clock window. Finally, click Step Out or Run in the Debug menu to run and stop at the second breakpoint. Examine the clock count in the Profile Clock window. It should read 752. The gap between the breakpoint and profile approaches comes from the additional procedures for calling functions, e.g., passing arguments to the function, storing the return address, branching from the function, etc.

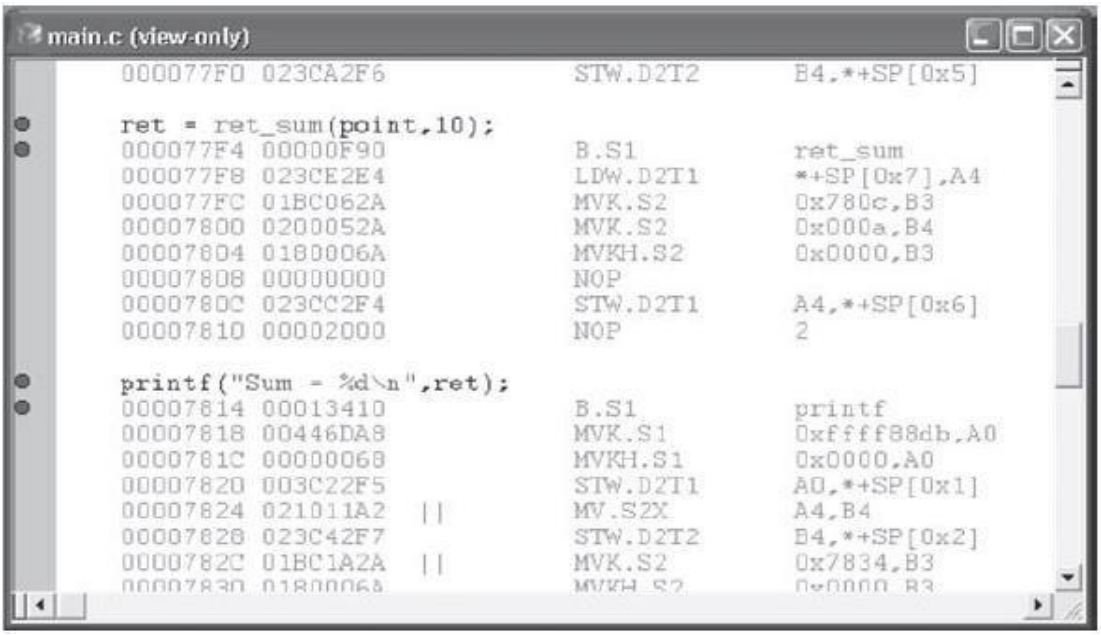


Fig. 5.23: Code execution time profiling with breakpoint.

A workspace containing breakpoints, probe points, charts, and watch windows can be saved for future reference. To do this, choose File → Workspace → Save Workspace As. This will display the Save Workspace window.

Enter the workspace name in the File Name field, and then click Save.
The file below is an assembly program for calculating the sum of values. Here, the two arguments of the sum function are passed into registers A4 and B4. The return value is stored in A4 and the return address in B3. The order in which the registers are chosen is governed by the argument-passing convention discussed later.

The function name must be preceded by an underscore, such as .global _sum.

Create a new source file sum.asm, as shown below, and add it to the project so that main() can call the sum() function.

```
        .global _sum
_sum:
```

```
        ZERO .L1 A9 ; sum register
        MV .L1 B4,A2 ;Initialize the counter with a passed argument: loop: LDH .D1 *A4++, A7 ; load value pointed to by A4 in reg. A7
        NOP 4
        ADD .L1 A7,A9,A9 ;A9 += A7
[A2] SUB .L1 A2,1,A2 ;decrement counter
[A2] B .S1 loop ;return to the loop
NOP 5
MV .L1 A9,A4 ;move the result to the return register
A4 B .S2 B3;reconnect to the address stored in B3 NOP 5
```

To save the file, go to the Save as type field and select Assembly Source Files (*.asm) from the dropdown list.

The main() program also needs to be modified by adding a function call to the sum() assembler function. This program is shown in Figure 5.24. Generate the program and run it. You should see the same return value.

```
#include <stdio.h>

void main()
{ int i, ret;
    short *point;

    point = (short *) 0x80000000;

    printf("BEGIN\n");

    for (i=0 ; i<10 ; i++)
    { printf ("[%d] %d\n",i, point[i]); }

    ret = ret_sum (point,10);
    print f ("C program Sum = %d\n", ret);

    ret = sum (point, 10);
```

```
    print f ("Assembly program Sum = %d\n", ret);
    print f ("END\n");
}

int ret_sum (const short* array, int N)
{ int count, sum;
        sum = 0;

    for (count=0; count < N; count++) sum
+= array[count];

    return(sum);

}
```

Fig. 5.24: Lab1's complete program.

Table 5.2 shows the number of cycles required to execute the sum function using several different versions. When a program is too large to fit in internal memory, it must be placed in external memory. Although the program in this lab is small enough to fit in internal memory, it is placed in external memory to study the change in the number of cycles. To move the program to external memory, open the file lab1.cmd and replace the line .text> IRAM with .text> CE0. As shown in Table 5.2, this generation slows execution to 2535 cycles. In the second version, the program resides in internal memory, and the number of cycles is therefore reduced to 732. By increasing the level of optimization, the number of cycles can be reduced further to 281. The assembly version of the program results in 472 cycles. This is slower than the fully optimized C program because it is not yet optimized. Optimization of assembly code will be discussed later. At this point, it is important to emphasize that, in all laboratories, the number of cycles indicated refers to C6711 DSK with CCS version 2.2. The number of cycles will vary slightly depending on the DSK target and the CCS version used.

Generation type	Code size	Number of cycles	
		Data in external memory	Data in internal memory
Program C in external memory	148	2535	2075
Program C in internal memory	148	732	382
–o0	64	464	113
–o1	64	463	111
–o2	100	404	57
–o3	84	281	33
Assembler program	64	472	150

Table 5.2: Number of cycles for different generations (on the DSK C6711 with CCS2.2).

5.11 EVM Target (Lab 1.3)

As described in Section 4.1 and illustrated in Figure 4.1, the EVM memory card has more internal/external memory space than the DSK memory card, thus allowing greater flexibility in loading code into the program/data section. Figure 5.25 shows an example using the EVM target where the data located at EXT3, say 0x03000000, is placed at the beginning of EXT3.

```
MEMORY
{

  PMEM: origin = 0x00000000, length = 0x00010000
  E XT2: origin = 0x02000000, length = 0x01000000
  E XT3: origin = 0x03000000, length = 0x01000000
  DMEM: origin = 0x80000000, length = 0x00010000 }

SECTIONS
{

      .vectors > PMEM
      .text > PMEM
      .bss  >  DMEM  .cinit  >
DMEM
      .const > DMEM
```

```
  .stack > DMEM
  .cio > DMEM
  .sysmem > DMEM
  .far > EXT2
  .mydata > EXT3
}
```

Fig. 5.25. Lab1 command file.

To place the program code in external memory, replace the line .text> PMEM via .text> EXT2.

The number of cycles on the EVM target is shown in Table 5.3. To generate the project, the rts6701.lib library for C6701 EVM or rts6201.lib for C6201 EVM must be added to the project. The Target Version field of the Build option must be selected as C670x (--mv 6700) for the TMS320C6701 floating-point target DSP or C620x (--mv 6200) for the TMS320C6201 floating-point target DSP.

fixed.

Finally, it's worth mentioning a point about resetting the EVM card. Sometimes you might find that your program can't be loaded into the DSP even if there's nothing wrong with it. In such cases, you need to reset the EVM card to resolve the issue. However, you must close CCS before resetting the card. Otherwise, the problem won't be solved. The exact timing may vary slightly depending on the DSK target and the CCS version used.

Generation type	Code size	Number of cycles	
		Data in external memory	Data in internal memory

Program C in external memory	148	1185	955
Program C in internal memory	148	541	355
–o0	64	282	97
–o1	64	281	95
–o2	100	213	36
–o3	84	129	21
Assembler program	64	139	133

Table 5.3. Number of cycles for different generations (on C6701 EVM with CCS2.2).

5.12 Simulator (Lab 1.4)

When no DSP card is available, the CCS simulator can be used to run lab programs. To configure CCS as a simulator, simply select "Simulator" in the Platform field during the CCS installation process, as shown in Figure 5.26. In the Import Configurations window, select the simulator option for one of the specified DSP cards. Clicking the Import button, followed by the Save and Quit button, configures the simulator and makes it ready for use. Note that while the simulator supports DMA and EMIF operations, McBSP, HPI, and Timer operations are not supported. The files for running labs via the simulator are provided in the simulator folder.

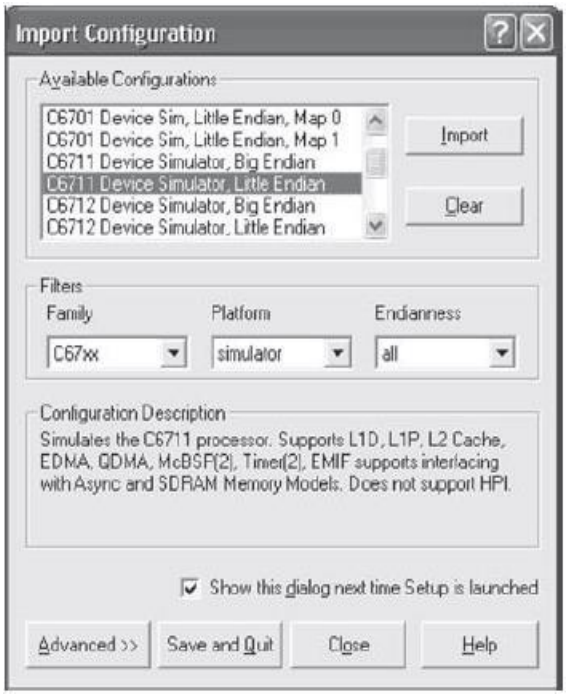


Fig. 5.26. Simulator installation

Chapter 6: Signal Processing Algorithms on DSP

6.1 Algorithm-architecture fit

The choice of a DSP processor for implementing a real-time algorithm depends on the application. Many factors influence this choice. These factors include cost, performance, power consumption, ease of use, time to market, and integration/interfacing capabilities.

Table 6.1 provides a list of currently available fixed-point and floating-point C6x processors. As can be seen in this table, instruction cycle time, speed, power consumption, memory, peripherals, package, and cost specifications vary among the products in this family.

Figure 6.1 illustrates the processing power of the C6x by showing a comparative speed analysis with other common DSP processors.

Device	RAM (Bytes) Data/Prog	McBSP	(E) DMA	COM	Timers	MHz	Cycles (ns)	MIPS	Standard activity - Total internal power (W) (full device speed)	Tension(V) Core, I/O	Housing
TMS320C6711-100	4K/4K/64K	2	16	HPI/16	2	100	10	600	1.1	1.8, 3.3	256 BGA, 27mm
TMS320C6711-150	4K/4K/64K	2	16	HPI/16	2	150	6.7	900	1.1	1.8, 3.3	256 BGA, 27mm
TMS320C6713-200	4K/4K/256K	2	16	HPI/16	2	200	5	1200	1.0	1.2, 3.3	208 TKFP, 28mm
TMS320C6713-225	4K/4K/256	2	16	HPI/16	2	225	4.4	1350	1.2	1.26, 3.3	272 BGA, 27mm
TMS320C6701-150	64K/64K	2	4	HPI/16	2	150	6.7	900	1.3	1.8, 3.3	352 BGA, 35mm
TMS320C6701-167	64k/64k	2	4	HPI/16	2	167	6	1000	1.4	1.9, 3.3	352 BGA, 35mm
TMS320C6416-500	16k/16k/1M	2+UTOPIA*	64	PCI /HPI 32/16	3	500	2	4000	0.64	1.2, 3.3	532 BGA, 23mm
TMS320C6416-600	16k/16k/1M	2+UTOPIA*	64	PCI /HPI 32/16	3	600	1.67	4800	1.06	1.4, 3.3	532 BGA, 23mm

* UTOPIA pins mixed with a third McBSP.

Table 6.1. Examples of C6x DSP product specifications

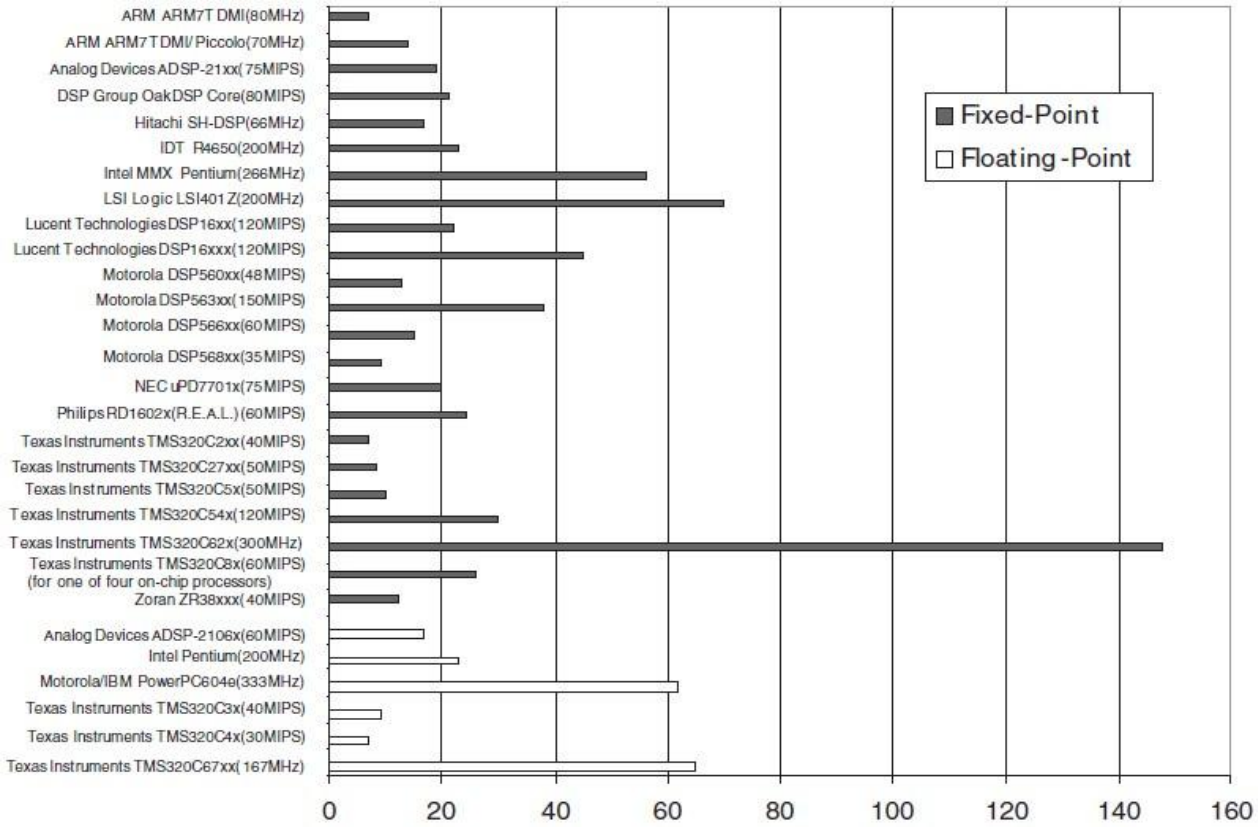


Fig. 6.1. BDTImark™ DSP Speed Metric by Berkeley Design Technology, Inc.1

*BDTImark is a summary measure of DSP speed, distilled from a series of DSP benchmarks independently developed and verified by Berkeley Design Technology, Inc. A higher BDTImark score indicates a faster processor. For a complete description of BDTImark and the underlying benchmarking methodology, as well as additional BDTImark scores, see http://www.bdti.com. © 2000 Berkeley Design Technology, Inc.

6.2 Real-time filtering (Lab 2)

The goal of this lab is to design and implement a finite impulse response (FIR) filter on the C6x. The filter design is done using MATLAB™. Once the design is complete, the filter code is inserted into the sampling shell program as an ISR to process live signals in real time.

Lab 2.1 Design of the FIR (Finite Impulse Response) filter

MATLAB or filter design packages can be used to obtain the coefficients of a desired FIR filter. For a more realistic simulation, a composite signal can be created and filtered in MATLAB. A composite signal composed of three sinusoids, as shown in Figure 6.2, can be created using the following MATLAB code:

```
Fs=8e3;
Ts=1/Fs;
Ns=512;
t=[0:Ts:Ts*(Ns-1)];
f1=750;
f2=2500;
f3=3000;
x1=sin(2*pi*f1*t);
x2=sin(2*pi*f2*t);
x3=sin(2*pi*f3*t); x=x1+x2+x3;
```

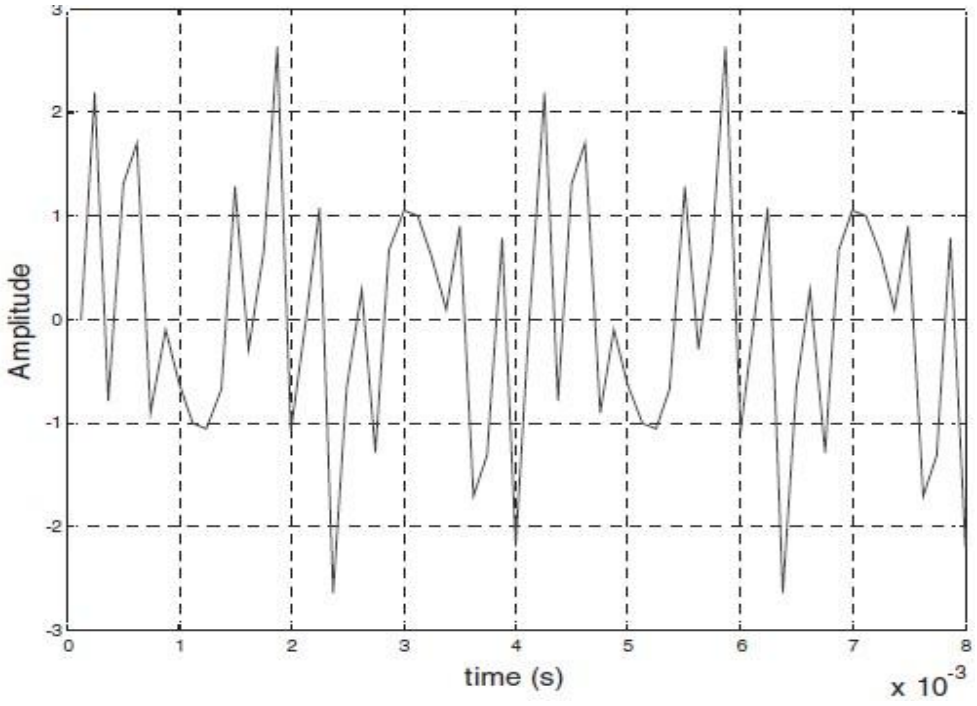


Fig. 6.2. Two cycles of composite signal.

The frequency content of the signal can be plotted using the MATLAB function 'fft'. Three peaks should be observed, at 750 Hz, 2500 Hz, and 3000 Hz. The frequency leakage observed on the plot is due to

windowing caused by the finite observation period. A low-pass filter is designed here to filter out frequencies above 750 Hz and retain the lower components. The sampling frequency is chosen to be 8 kHz, which is common in voice processing. The following code is used to obtain the frequency plot shown in Figure 6.3:

```
X=(abs(fft(x,Ns)));
y=X(1:length(X)/2);
f=[1:1:length(y)];
plot(f*Fs/Ns,y);
grid on;
```

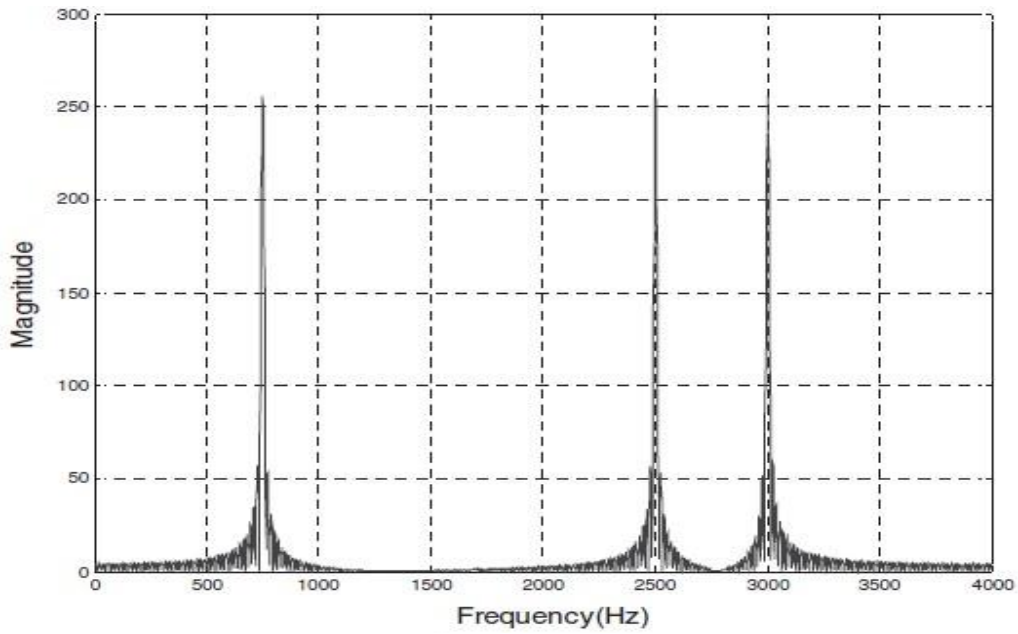


Fig. 6.3. Frequency components of the composite signal.

To design a FIR filter with a passband frequency of 1600 Hz, a stopband frequency of 2400 Hz, a passband gain of 0.1 dB, a passband attenuation of 20 dB, and a sampling rate of 8000 Hz, the Parks-McClellan method is used via the `remez` function in MATLAB [14]. The magnitude and phase response are shown in Figure 6.4, and the coefficients are given in Table 6.2. The MATLAB code is as follows:

```
rp = 0.1; %Passband ripple rs = 20; %
Stopband ripple fs = 8000; % Sampling
frequency f = [1600 2400]; % Cutoff
frequencies a = [1 0]; % Desired
amplitudes
% Compute deviations
dev = [(10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)];
[n,fo,ao,w] = remezord(f,a,dev,fs);
B = remez(n,fo,ao,w);
A=1; freqz(B,A);
```

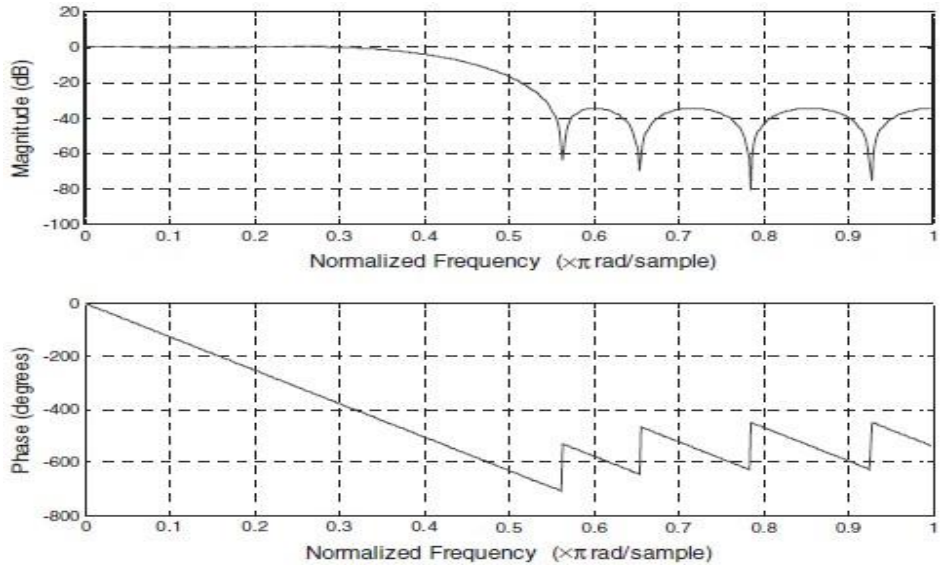


Fig. 6.4. Filter amplitude and phase response.

Coefficient Values Representation Q-15
B0 0.0537 0x06DF
B1 0.0000 0x0000
B2 –0.0916 0xF447
B3 –0.0001 0xFFFD
B4 0.3131 0x2813
B5 0.4999 0x3FFC
B6 0.3131 0x2813
B7 –0.0001 0xFFFD
B8 –0.0916 0xF447
B9 0.0000 0x0000
B10 0.0537 0x06DF

Table 6.2: FIR filter coefficients (RIF).

Using these coefficients, the MATLAB "filter" function is used to verify that the FIR filter is indeed capable of filtering 2.5 kHz and 3 kHz signals. The following MATLAB code allows for a visual inspection of the filtering operation:

```
% Figure 6-5 subplot(3,1,1);
va_fft(x,1024,8000);
subplot(3,1,2);
[h,w]=freqz(B,A,512);
plot(w/(2*pi),10*log(abs(h)));
grid on;

subplot(3,1,3); y
= filter(B,A,x);
va_fft(y,1024,8000);

function va_fft(x,N,Fs)
X=fft(x,N);
XX=(abs(X));
XXX=XX(1:length(XX)/2);
y=XXX;
f=[1:1:length(y)]; plot(f*Fs/N,y);
grid on;

% Figure 6-6 n=128
subplot(2,1,1);
plot(t(1:n),x(1:n)); grid
on; xlabel('Time(s)');
ylabel('Amplitude');
title('Original and Filtered Signals');
subplot(2,1,2); plot(t(1:n),y(1:n));
grid on; xlabel('Time(s)');
ylabel('Amplitude');
```

By observing the plots shown in Figures 6.5 and 6.6, we see that the filter is able to remove the desired frequency components from the composite signal. Note that the time response has an initial setup time, which makes the first few data samples inaccurate. Now that the filter design is complete, let's consider its implementation.

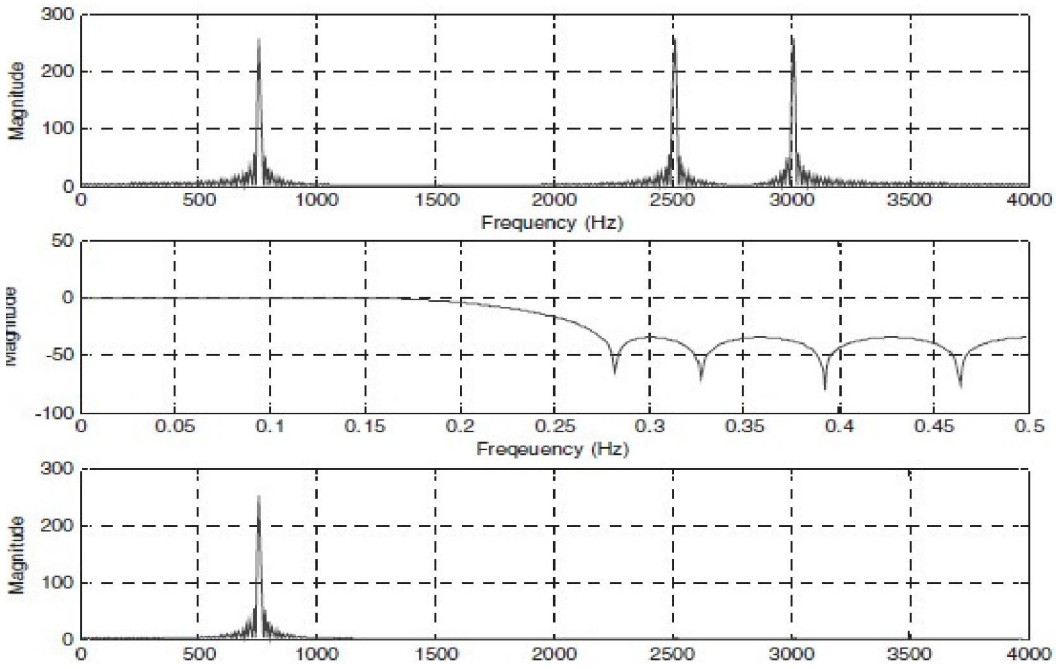


Fig. 6.5. Frequency representation of the filtering operation.

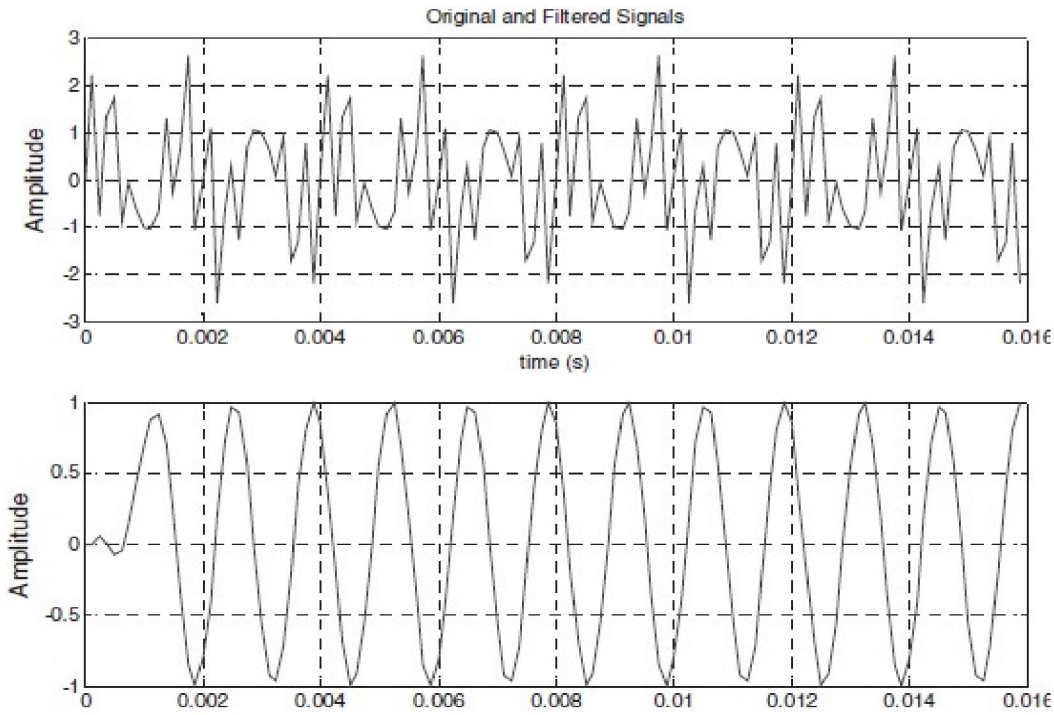


Fig. 6.6. Time domain representation of the filtering operation.

Lab 2.2 FIR Filter Implementation

A FIR filter can be implemented in C or assembly language. The goal of the implementation is to have a minimum cycle time algorithm. This means performing the filtering as quickly as possible to achieve the highest sampling frequency (the smallest sampling interval). Initially, the filter is implemented in C because it requires less coding effort. Once a working algorithm is obtained in C, compiler optimization levels (i.e., -o2, -o3) are enabled to reduce the number of cycles. An implementation of the filter is then performed in hand-coded assembly language, which can be software-piped for optimal performance. A final implementation of the filter is performed in linear assembly, and the timing results are compared.

The difference equation y [n] = $\sum_{k=0}^{N-1} B_k * x[n-k]$ is implemented to perform the filter. Since the filter is implemented on the DSK, the coding is done by modifying the sampling program in lab 2, which uses an ISR (interrupt routine) capable of receiving a sample from the serial port and sending it back without any modification.

When using the C6711 DSK with the audio daughterboard, the data received from McBSP1 has a width of 32 bits, with the 16 most significant bits coming from the right channel and the 16 least significant bits coming from the left channel.

Considering the Q-15 representation here, the MPY instruction is used to multiply the lower portion of a 32-bit sample (left channel) by a 16-bit coefficient. To store the product in 32 bits, it must be shifted left by one to remove the extended sign bit. Now, to export the product to the codec output, it must be shifted right by 16 to place it in the lower 16 bits. Alternatively, the product can be shifted right by 15 without removing the sign bit.

To implement the algorithm in C, the intrinsic _mpy() and shift operators "<<" and ">>" must be used as follows:

```
    result = (_mpy(sample,coefficient)) << 1;
result = result >> 16;
```
Or

```
result = (_mpy(sample,coefficient)) >> 15;
```

Here, both the result and the sample are 32 bits wide, while the coefficient is 16 bits. The intrinsic _mpy() function multiplies the lower 16 bits of the first argument by the lower 16 bits of the second argument. Therefore, the lower 16 sample bits are used in the multiplication.

For the FIR filter to function correctly, the current sample and N-1 previous samples must be processed simultaneously, where N is the number of coefficients. Therefore, the N most recent samples must be stored and updated with each incoming sample. This can be easily done using the following code:

```
void interrupt serialPortRcvISR()
{
      int i, temp, result= 0; temp
= MCBSP_read(hMcbsp);

      // Update array samples for(i=N-1;i>=0;i--)
           samples[i+1] = samples[i];

      samples[0] = temp;

      // Filtering
      for( i = 0 ; i <= N ; i++ )
            result += ( _mpy(samples[i], coefficients[i]) ) << 1;
result = result >> 16; MCBSP_write(hMcbsp, result);

   }
```

To complete the implementation of the FIR filter, we need to incorporate the filter coefficients:

```
#define N 10

// FIR filter coefficients
short coefficients[N+1] = { 0x6DF, 0x0, 0xF447, 0xFFFD, 0x2813, 0x3FFC,
0x2813, 0xFFFD, 0xF447, 0x0, 0x6DF};

int samples[N];
      .
      .
int main()
{

     .
```

```
    .
    for(i = 0; i <= N; i++ ) samples[i]=0;
    .
    .
}
```

6.3 Adaptive Filtering (Lab 3)

Adaptive filtering is used in many applications, ranging from noise cancellation to system identification. In most cases, the coefficients of an FIR filter are modified based on an error signal to match a desired signal. In this lab, a system identification example is implemented in which an adaptive FIR filter is used to match the output of a seventh-order IIR bandpass filter. The IIR (Infinite Impulse Response) filter is designed in MATLAB and implemented in C. The adaptive FIR is first implemented in C and then assembled using a circular buffer.

In system identification, the behavior of an unknown system is modeled by accessing its inputs and outputs. An adaptive FIR filter can be used to adapt to the system output based on the same input. The difference between the system output, d[n], and the adaptive filter output, y[n], constitutes the error term e[n], which is used to update the FIR filter coefficients. Figure
6.7 illustrates this process.

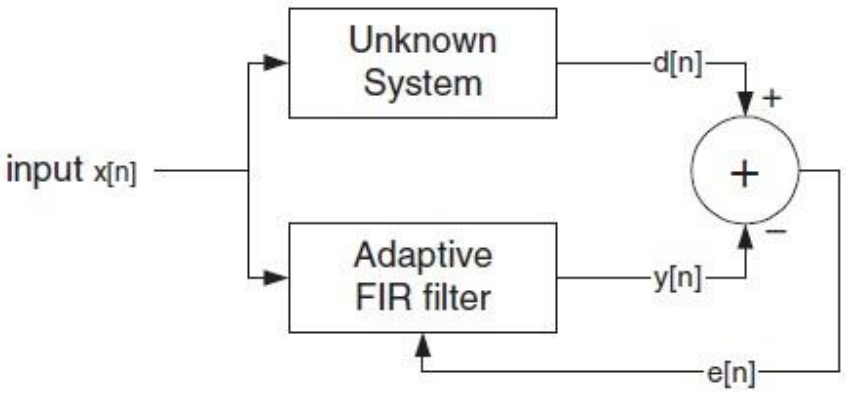


Fig. 6.7. Adaptive filtering

The error term calculated from the difference in outputs of the two systems is used to update each coefficient of the FIR filter according to the formula (least mean squares (LMS) algorithm [15]):

$$h_n[n] = h_{n-1}[k] + \delta * e[n] * x[n-k] \quad (6.1)$$

where h denotes the unit response of the sample or the coefficients of the FIR filter. The output y[n] is needed to approximate d[n]. The term δ indicates the step size. A small step size will ensure convergence but results in a slow adaptation rate. A large step size, while faster, may lead to the solution being ignored.

Lab 3.1 Design of the RII filter

A seventh-order bandpass IIR filter is used to act as the unknown system. The adaptive IIR is designed to match the response of the IIR system. Given a sampling frequency of 8 kHz, let the IIR filter have a bandwidth of π/3 to 2π/3 (radians), with a stopband attenuation of 20 dB. The filter design can be easily performed with the MATLAB function 'yulewalk' [14]. The following MATLAB code can be used to obtain the

filter coefficients:

```
Nc=7;
f=[0 0.32 0.33 0.66 0.67 1];
m=[0 0 1 1 0 0];
[B,A]=yulewalk(Nc,f,m); freqz(B,A);
%Create A sample signal
Fs=8000;
Ts=1/Fs; Ns=128;
t=[0:Ts:Ts*(Ns-1)];
f1=750;
f2=2000;%The one to keep
f3=3000;
x1=sin(2*pi*f1*t);
x2=sin(2*pi*f2*t);
x3=sin(2*pi*f3*t);
x=x1+x2+x3; %Filter it
y=filter(B,A,x);
```

The filter's functionality can be verified by applying a simple composite signal. Using the MATLAB "filter" function, check the design by observing that the frequency components of the composite signal falling within the stopband are suppressed. (See Table 6.3 and Figure 6.8.)

AB
1.0000 0.1191
0.0179 0.0123
0.9409 −0.1813
0.0104 −0.0251
0.6601 0.1815
0.0342 0.0307
0.1129 −0.1194
0.0058 −0.0178

Note: Do not confuse the A&B coefficients with the CPU A&B registers!

Table 6.3. IIR filter coefficients

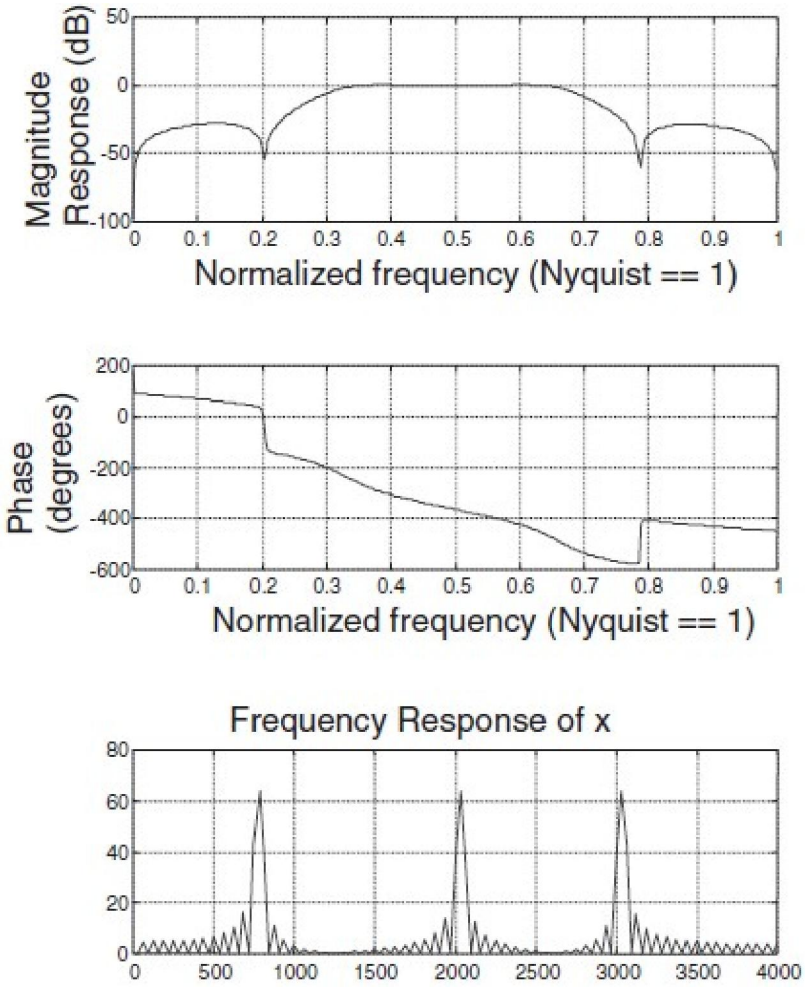


Fig. 6.8. Response from the RII filter.

Lab 3.2 Implementation of the IIR filter

The implementation of the IIR filter is first carried out in C, using the following difference equation

$$y[n] = -\sum_{k=1}^{N} a_k * y[n-k] + \sum_{k=0}^{N} b_k * x[n-k] \quad (6.2)$$

where ak and bk denote the coefficients of the IIR filter. Two arrays are needed, one for the input samples x[n] and the other for the output samples y[n]. Since the filter is of order 7, an input array of size 8 and an output array of size 7 are considered. The arrays are used to simulate a circular buffer, as this CPU property is not accessible in C. When a new sample arrives, all elements of the input array are shifted down by one. In this way, the last element is lost, and the last eight samples are always retained. The input array is used to compute the resulting output, and then the output is used to modify the output array. A simple implementation of this scheme is shown in the following code:

```
interrupt void serialPortRcvISR (void)
{

      int temp,n,ASUM,BSUM; short
input,IIR_OUT;

      temp = MCBSP_read(hMcbsp);
      input = temp >> S; // Scale factor

      for(n=7;n>0;n--) // Entry stamp
            IIRwindow[n] = IIRwindow[n-1];

IIRwindow[0] = input;

BSUM = 0;

for(n=0;n<=7;n++)
{ // Multiplication of Q-15 with Q-15
BSUM += ((BS[n]*IIRwindow[n]) << 1); // Results in Q-30. Shift by one for } // Remove the
sign extension bit

ASUM = 0; for(n=0;n<=6;n++)
{

      ASUM += ((AS[n] * y_prev[n]) << 1);

}

IIR_OUT = (BSUM - ASUM) >> 16;

for(n=6;n>0;n--) // Output buffer y_prev[n] = y_prev[n-1];
```

```
y_prev[0] = IIR_OUT;

MCBSP_write(hMcbsp, IIR_OUT << S); // Scale factor S

}
```

By running this program while connecting a function generator and an oscilloscope to the line input and output of the audio daughterboard, the functionality of the IIR filter can be verified. Each time the DRR (Matlab reference for an algorithm to estimate the forward energy ratio to be diffused) receives a new incoming sample from the function generator, the ISR is called. The new sample is then shifted to the right by the scaling factor S. This factor is included to correct for any possible overflow. In this lab, shifting is not necessary. Once the new sample is scaled, the last eight samples are retained by discarding the oldest sample and adding the new sample to the input buffer IIRwindow. This is done by shifting the data in the input array. Note that this array is global and is initialized to zero in the main function.

6.4 Circular Stamp

In many DSP algorithms, such as filtering, adaptive filtering, or spectral analysis, we need to move data or update samples (i.e., we need to deal with a moving window). The direct method of moving data is inefficient and uses many cycles. Circular buffering is an addressing method by which a moving window effect can be created without the overhead associated with shifting data. In a circular buffer, if a pointer pointing to the last element of the buffer is incremented, it is automatically wrapped and pointed to the first element of the buffer. This provides a simple mechanism for excluding the oldest sample while including the newest one, creating a moving window effect as illustrated in Figure 6.9.

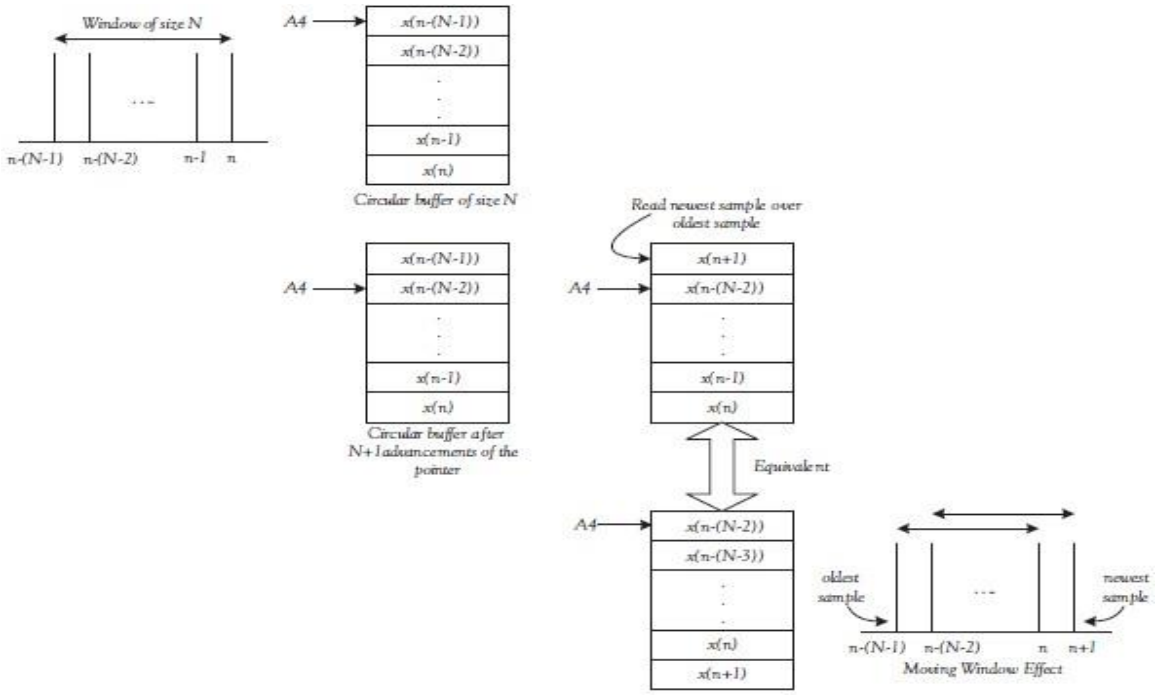


Fig. 6.9. Movable window effect.

Some DSPs have dedicated hardware for this type of addressing. On the C6x processor, the logic and arithmetic unit has built-in circular addressing capability. To use circular buffering, the circular buffer sizes must first be written to the BK0 and BK1 block size fields of the Address Mode Register (AMR), as shown in Figure 6.10. The C6x allows two independent circular buffers of powers of 2. The buffer size is specified as 2(N + 1) bytes, where N indicates the value written to the BK0 and BK1 block size fields.

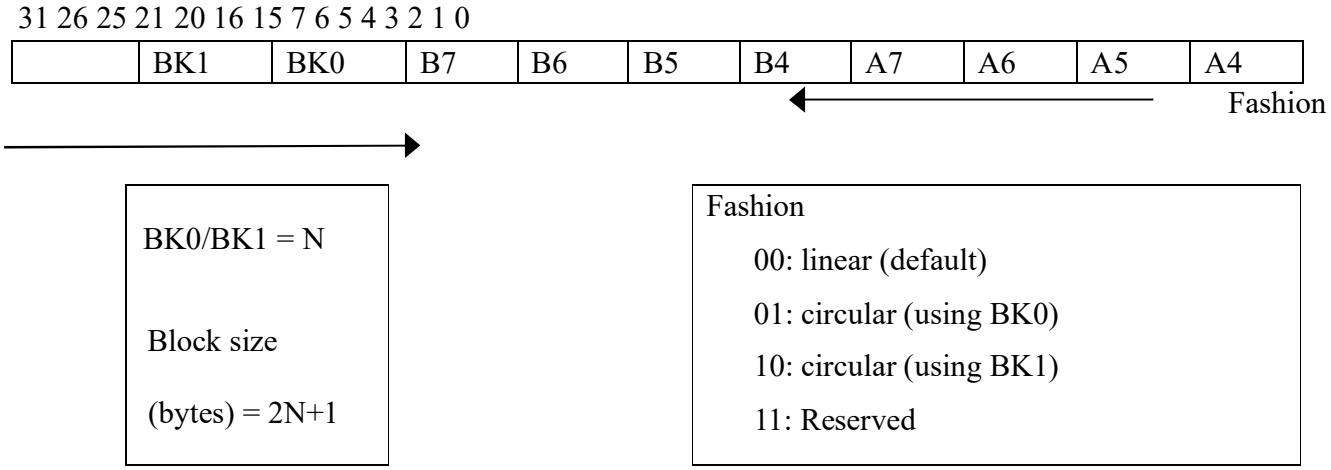


Fig. 6.10. AMR (Address Mode Register).

Next, the register to be used as the circular buffer pointer must be specified by setting the appropriate bits of AMR to 1. For example, as shown in Figure 8-2, to use A4 as the circular buffer pointer, bit 0 or 1 is set to 1. Of the 32 registers in the C6x, 8 can be used as circular buffer pointers: A4 to A7 and B4 to B7. Note that linear addressing is the default addressing mode for these registers.

Figure 6.11 shows the code for configuring the AMR register for a circular buffer of size 8, with a loading example. To configure such a circular buffer in C, you must use what is called an inline assembler as follows:

```
asm ("MVK.S2 0001h,B2");
asm ("MVKLH.S2 0002h,B2");
asm ("MVC.S2 B2,AMR");
```

```
; Blk size = 8, use  A4/BK0

    MVK.S2      0001H , B2
    MVKLH.S2    0002H , B2
    MVC.S2      B2, AMR

    LDH.D1      *A4++[2], A1 ; A1 = 0, A4=&s[2]
    LDH.D1      *A4++[3], A1 ; A1 = 2, A4=&s[1]
```

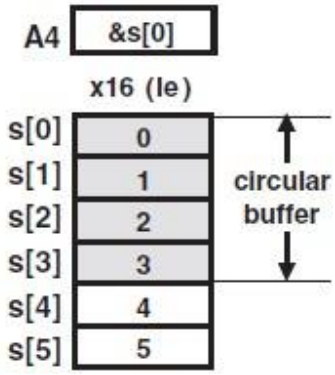


Fig. 6.11. Example of an AMR configuration.

When using circular buffers, care must be taken to align the data to the buffer size limit. In C, this can be achieved using pragma directives. Pragma directives specify which types of preprocessing are performed by the compiler. The DATA_ALIGN pragma can be used to align the symbol to a power-2 alignment limit constant (in bytes) as follows:

#pragma DATA_ALIGN (symbol, constant)

6.5 Implementation of the FFT (Fast Fourier Transform)

To enable real-time processing, the FFT is used, which leverages the symmetry properties of the DFT. The approach to calculating a 2N-point FFT is as described in the TI application report.

SPRA291 [16] is adopted here. This approach consists of forming two new N-point signals x1 [n] and x2[n] from the original signal at 2N points g[n] by dividing it into even and odd parts as follows:

$x1[n] = g[2n], 0 \leq n \leq N-1;$

$x2[n] = g[2n+1]$ (6.3)

From the two sequences x1 [n] and x2 [n], a new complex sequence is defined as

$x[n] = x1[n] + jx2[n], 0 \leq n \leq N-1$ (6.4)

To obtain G[k], DFT of g[n], the equation

$G[k] = X[k]A[k] + X^*[Nk]B[k]$, (6.5)

K = 0, 1, …, N-1, with X[N] = X[0] is used, where

$A[k] = ½(1-jW2kN)$ (6.6) and $B[k] = ½(1+jW2kN)$ (6.7)

Only N points of G[k] are calculated from equation (6.5). The remaining points are found using the complex conjugate property of G[k], G[2N - k] = G*[k]. Consequently, a 2N-point transformation is calculated based on an N-point transformation, leading to a reduction in the number of cycles. The code for the functions (split1, R4DigitRevIndexTableGen, digit_reverse, and radix4) implementing this approach is provided in the TI Application Report [16].

Figure 6-12 shows the result of the FFT where the signal was scaled down by a factor of 0, 2, 4, and 5, respectively. Scaling is performed to eliminate overflows, which are present at scaling factors of 0, 2, and 4. As these figures reveal, the input signal must be scaled down five times to eliminate overflows. When the signal is scaled down five times, the expected peaks appear. The total number of cycles for this FFT is 56,383. Since this is less than the capture time available for a 128-point data frame at a sampling rate of 8 kHz, this algorithm is expected to run in real time on the DSK.

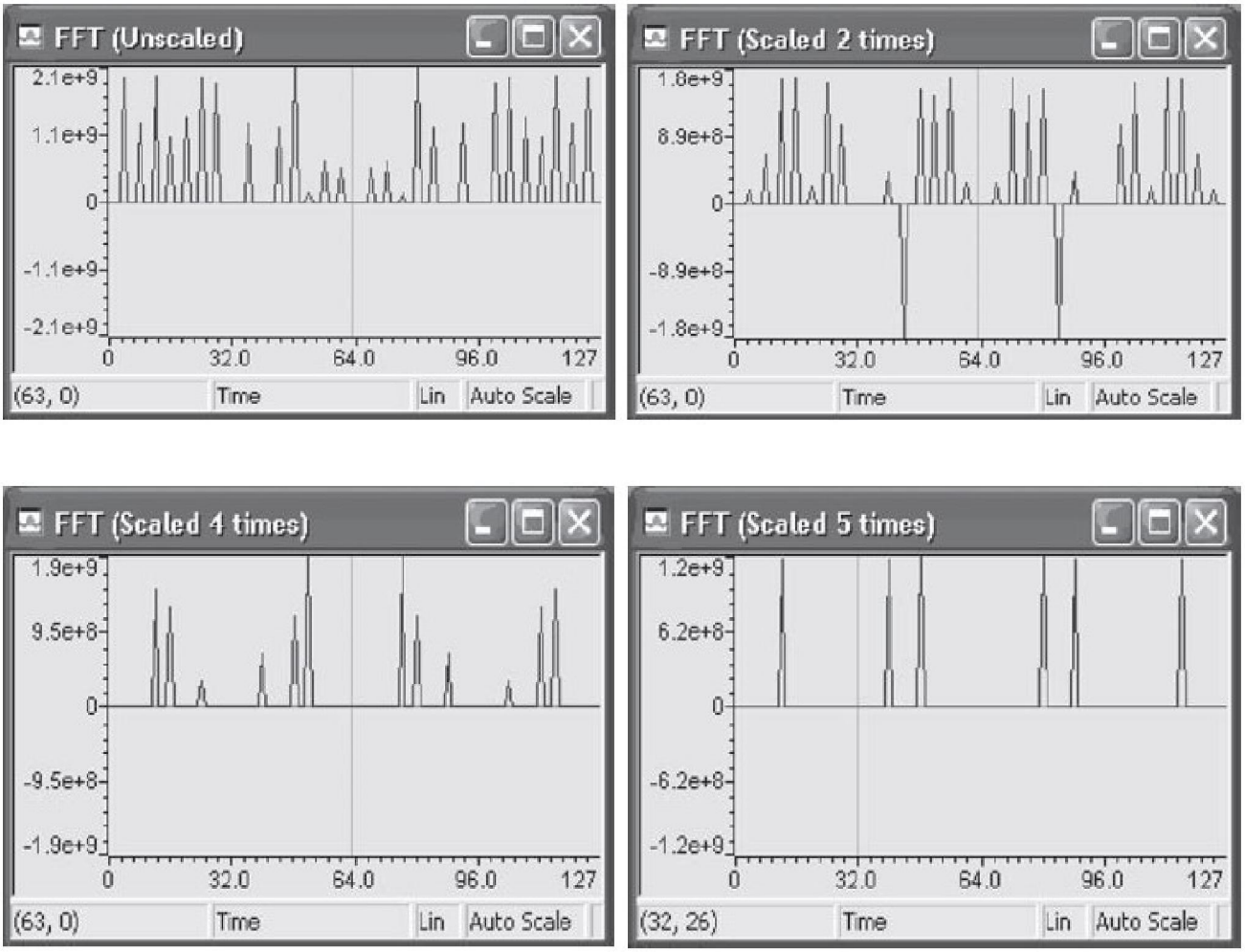


Fig. 6.12. Scaling to obtain a correct FFT amplitude response.

Bibliographical references

1. Nasser Kehtarnavaz, Real-Time Digital Signal Processing., 2005, Elsevier Inc. ISBN 0-7506-7830-5. 2. Texas Instruments, TMS320C6201/6701 Evaluation Module Reference Guide, Literature ID# SPRU 269F, 2002.

3. J. Proakis and D. Manolakis, Digital Signal Processing: Principles, Algorithms, and Applications, Prentice-Hall, 1996.
4. S. Mitra, Digital Signal Processing: A Computer-Based Approach, McGraw Hill, 1998.
5. B. Razavi, Principles of Data Conversion System Design, IEEE Press, 1995.
6. Texas Instruments, TMS320C6000 CPU and Instruction Set Reference Guide, Literature ID# SPRU 189F, 2000.
7. Texas Instruments, Technical Training Notes on TMS320C6x, TI DSP Fest, Houston, 1998.
8. Texas Instruments, TMS320C6000 Assembly Language Tools User's Guide, Literature ID# SPRU 186M, 2003.
9. Texas Instruments, TMS320C6000 Optimizing Compiler User's Guide, Literature ID# SPRU 187K, 2002.
10. Texas Instruments, TMS320C6000 CPU and Instruction Set Reference Guide, Literature ID# SPRU 189F, 2000.
11. S. Tretter, Communication System Design Using DSP Algorithms: With Laboratory Experiments for the *TMS320C6701 and TMS320C6711,*Kluwer Academic Publishers, 2003.
12. Sen M Kuo, Bob H Lee, Real-Time Digital Signal Processing, 2001 John Wiley. ISBN 0-470-84534-1. 13. BA Shenoi, Introduction to Digital Signal Processing, 2006 by John Wiley & Sons, Inc. ISBN-13 978-0-471-46482.
14. The Mathworks, MATLAB Reference Guide, 1999.
15. S. Haykin, Adaptive Filter Theory, Prentice-Hall, 1996.1
16. Texas Instruments, Application Report SPRA 291, 1997.