

## Getting Started with DSPs

---

Once you decide that a Digital Signal Processor is right for your application, you need a way to get started. Many manufacturers will sell you a low cost evaluation kit, allowing you to experience their products first-hand. These are a great educational tool; it doesn't matter if you are a novice or a pro, they are the best way to become familiar with a particular DSP. For instance, Analog Devices provides the EZ-KIT<sup>®</sup> Lite to teach potential customers about its SHARC<sup>®</sup> family of Digital Signal Processors. For only \$179, you receive all the hardware and software you need to see the DSP in action. This includes "canned" programs provided with the kit, as well as applications you can write yourself in assembly or C. Suppose you buy one of these kits from Analog Devices and play with it for a few days. This chapter is an overview of what you can expect to find and learn.

---

### The ADSP-2106x family

In the last chapter we looked at the general operation of the ADSP-2106x "SHARC" family of Digital Signal Processors. Table 29-1 shows the various members of this family. All these devices use the same architecture, but have different amounts of on-chip memory, a key factor in deciding which one to use. Memory access is a common bottleneck in DSP systems. The SHARC DSPs address this by providing an ample supply of on-chip dual-ported SRAM. However, the last thing you want to do is pay for more memory than you need. DSPs often go into cost sensitive products, such as cellular telephones and CD players. In other words, the organization of this family is determined by *marketing* as well as *technology*.

The oldest member of this family is the ADSP-21020. This chip contains the core architecture, but does not include on-chip memory or I/O handling. This means it cannot function as a stand-alone computer; it requires external components to be a functional system. The other devices are complete

*SHARC, EZ-KIT, EZ-LAB, VisualDSP, EZ-ICE*, the SHARC logo, the Analog Devices logo, and the VisualDSP logo are registered trademarks of Analog Devices, Inc.

PRODUCT	Memory	Notes
AD1460	4 Mbit ×4	Quad-SHARC, Four ADSP-21060's in the same module; provides an incredible 480 MFLOPS in only 2.05"×2.05"×0.16".
ADSP-21160M	4 Mbit	New! Features Single Instruction Multiple Data (SIMD) core architecture; optimized for multiprocessing with link ports, 64 bit external bus, and 14 channels of DMA
ADSP-21060	4 Mbit	Power house of the family; most memory; link ports for high speed data transfer and multi-processing
ADSP-21062	2 Mbit	Same features as the ADSP-21060, but with less internal memory (SRAM), for lower cost
ADSP-21061	1 Mbit	Low cost version used in the EZ-KIT Lite; less memory & no link ports; additional features in DMA for the serial port
ADSP-21065L	544 kbit	A recent addition to the family; fast and very low cost (\$10). Will attract many fixed point applications to the SHARC family
ADSP-21020	-0-	Oldest member of the family. Contains the core processor, but no on-chip memory or I/O interface. Not quite a SHARC DSP.

TABLE 29-1  
Members of the SHARC family.

computers within a single chip. All they require to operate is a source of power, and some way to load a program into memory, such as an external PROM or data link.

Notice in Table 29-1 that even the low-end products have a very significant amount of memory. For instance, the ADSP-21065L has 544 kbits of internal SRAM. This is enough to hold 6-8 seconds of digitized speech (8k samples per second, 8 bits per sample). On the high-end of the family, the ADSP-21060 has a 4 Mbit memory. This is more than enough to store an entire digitized image (512×512 pixels, 8 bits per pixel). If you require even more memory, you easily add external SRAM (or slower memory) to any of these devices.

In addition to memory, there are also differences between these family members in their I/O sections. The ADSP-21060 and ADSP-21062 (the high-end) each have six *link ports*. These are 4 bit wide parallel connections for combining DSPs in multiprocessing systems, and other applications that require flexible high-speed I/O. The ADSP-21061 and ADSP-21065L (the low-end) do not have link ports, but feature more DMA channels to assist in their *serial port* operation. You will also see these part numbers with an "L" or "M" after them, such as "ADSP-21060L." This indicates that the device operates from a voltage lower than the traditional 5.0 volts. For

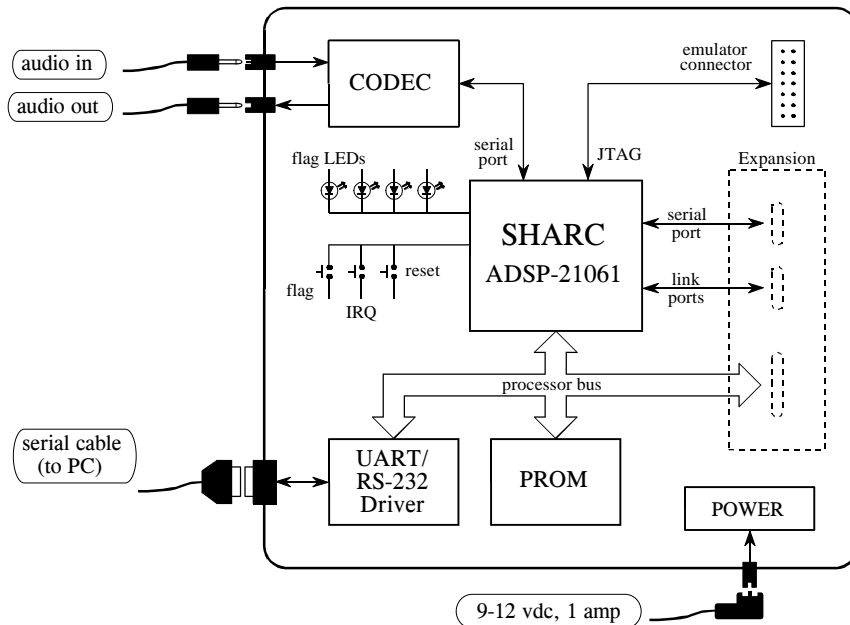


FIGURE 29-1

Block diagram of the EZ-KIT Lite board. Only four external connections are needed: audio in, audio out, a serial (RS-232) cable to your personal computer, and power. The serial cable and power supply are provided with the EZ-KIT Lite.

instance, the ADSP-21060L operates from 3.3 volts, while the ADSP-21160M uses only 2.5 volts.

In June 1998, Analog Devices unveiled the second generation of its SHARC architecture, with the announcement of the ADSP-21160. This features a Single Instruction Multiple Data (SIMD, or "sim-dee") core architecture operating at 100 MHz, an accelerated memory bus bandwidth of 1600 megabytes per second, two 64 bit data busses, and four 80-bit accumulators for fixed point calculations. All totaled, the new ADSP-21160M executes a 1024 point FFT in only 46 microseconds. The SIMD DSP contains a second set of computational units (arithmetic and logic unit, barrel shifter, data register file, and multiplier), allowing ADI to maintain backward code compatibility with the ADSP-2106x family, while providing a road-map to up to ten times higher performance.

## The SHARC EZ-KIT Lite

The EZ-kit Lite gives you everything you need to learn about the SHARC DSP, including: hardware, software, and reference manuals. Figure 29-1 shows a block diagram of the hardware provided in the EZ-KIT Lite, based around the ADSP-21061 Digital Signal Processor. This comes as a  $4\frac{1}{2} \times 6\frac{1}{2}$  inch printed circuit board, mounted on plastic standoffs to allow it to sit on

your desk. (There is also a version called the EZ-LAB, using the ADSP-21062, that plugs into a slot in your computer). There are only four connections you need to worry about: DC power, a serial connection to your personal computer, and the input and output signals. A DC power supply and serial cable are even provided in the kit. The input and output signals are at audio level, about 1 volt amplitude. Alternatively, a jumper on the board allows a microphone to be directly attached into the input. The idea is to plug a microphone into the input, and attach a set of amplified speakers (such as used with personal computers) to the output. This allows you to *hear* the effect of various DSP algorithms.

Analog-to-digital and digital-to-analog conversion is accomplished with an Analog Devices AD1847 codec (coder-decoder). This is a 16 bit sigma-delta converter, capable of digitizing two channels (stereo) at a rate of up to 48k samples/second, and simultaneously outputting two channels at the same rate. Since the primary use of this board is to process audio signals, the inputs and outputs are AC coupled with a cutoff of about 20 Hz.

Three push buttons on the board allow the user to generate an interrupt, reset the processor, and toggle a flag bit that can be read by the system. Four LEDs mounted on the board can be turned on and off by toggling bits. If you are ambitious, there are sections of the board that allow you to access the serial port, link ports (only on the EZ-LAB with its ADSP-21062), and processor bus. However, these are *unpopulated*, and you will need to attach the connectors and other components yourself.

Here's how it works. When the power is applied, the processor boots from an on-board EPROM (512 kbytes), loading a program that establishes serial communication with your personal computer. Next, you launch the *EZ-Lite Host* program on you PC, allowing you to download programs and upload data from the DSP. Several prewritten programs come with the EZ-KIT Lite; these can be run by simply clicking on icons. For instance, a band-pass program allows you to speak into the microphone, and hear the result after passing through a band-pass filter. These programs are useful for two reasons: (1) they allow you to quickly get the system doing something interesting, giving you confidence that it does work, and (2) they provide a template for creating programs of your own. Which brings us to our next topic, a design example using the EZ-KIT Lite.

## Design Example: An FIR Audio Filter

After you experiment with the prewritten programs for awhile, you will want to modify them to gain experience with the programming. Programs can be written in either assembly or C; the EZ-KIT Lite provides software tools to support both languages. Later in this chapter we will look at advanced methods of programming, such as *simulation*, *debugging*, and working in an *integrated development environment*. For now, we will focus on the easiest way to get a program to run. Little steps for little feet.

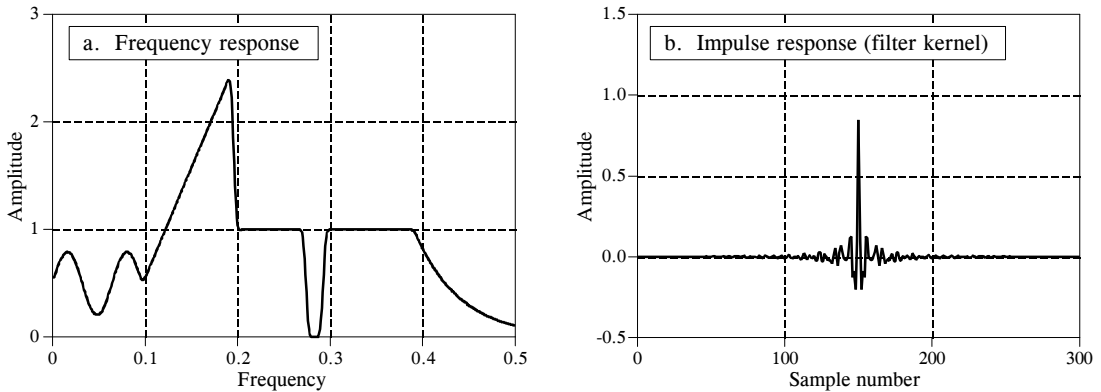


FIGURE 29-2

Example FIR filter. In (a) the frequency response of a highly custom filter is shown. The corresponding impulse response (filter kernel) is shown in (b). This filter was designed in Chapter 17 to show that virtually *any* frequency response can be achieved with FIR digital filters.

Since the source code is in ASCII, a standard text editor is all that is needed to make changes to existing files, or create entirely new programs. Table 29-2 shows an example of an FIR filter program written in assembly. While this is the only code you need to worry about for now, keep in mind that there are other files needed to make this a complete program. This includes an "architecture description file" (which defines the hardware configuration and memory allocation), setup of the interrupt vector table, and a codec initialization routine. Eventually you will need to understand what goes on in these sections, but for now you simply copy them from the prewritten programs.

As shown at the top of Table 29-2, there are three variables that need to be defined before jumping into the main section of code. These are the number of points in the filter kernel, **NR\_COEF**; a circular buffer that holds the past samples from the input signal, **dline[ ]**; and a circular buffer that holds the filter kernel, **coef[ ]**. We also need to give the program two other pieces of information: the sampling rate of the codec, and the name of the file containing the filter kernel, so that it can be read into **coef[ ]**. All these steps are easy; nothing more than a single line of code each. We don't show them in this example because they are contained in the sections of code that we are ignoring for simplicity.

Figure 29-2 shows the filter kernel we will test the program with, the same custom filter we designed in Chapter 17. As you recall, this filter was chosen to have a very irregular frequency response, reinforcing the notion that FIR digital filters can provide virtually any frequency response you desire. Figure (a) shows the frequency response of our test filter, while (b) shows the corresponding impulse response (i.e., the filter kernel). This 301 point filter kernel is stored in an ASCII file, and is combined with the other sections of code during linking to form a single executable program.

The main section of the program performs two functions. In lines 6 to 13, the data-address-generators (DAGs) are configured to manage the circular buffers: `dline[ ]`, and `coef[ ]`. As described in the last chapter, three parameters are needed for each buffer: the starting location of the buffer in memory (`b0` and `b8`), the length of the buffer (`l0` and `l8`), and the step size of the data being stored in the buffer (`m0` and `m8`). These parameters that control the circular buffers are stored in hardware registers in the DAGs, allowing them to access and manage the data very efficiently.

The second action of the main program is a "thumb-twiddling" loop, implemented in lines 15 to 19. This does nothing but wait for an interrupt indicating that an input sample has been acquired. All of the processing in this program occurs on a **sample-by-sample** basis. Each time a sample is read from the input, a sample in the output signal is calculated and routed to the codec. Most time-domain algorithms, such as FIR and IIR filters, fall into this category. The alternative is **frame-by-frame** processing, which is required for frequency-domain techniques. In the frame-by-frame method, a *group* of samples is read from the input, calculations are conducted, and a *group* of samples is written to the output.

The subroutine that services the sample-ready interrupt is broken into three sections. The first section (lines 27 to 33) fetches the sample from the codec as a fixed point number, and converts it to floating point. In SHARC assembly language, a data register holding a fixed point number is referred to by "r" (such as `r0`, `r8`, `r15`, etc.), and by "f" if it is holding a floating point number (i.e., `f0`, `f8`, or `f15`). For instance, in line 32, the fixed point number in data register 0 (i.e., `r0`) is converted into a floating point number and overwrites data register 0 (i.e., `f0`). This conversion is done according to a scaling specified by the fixed point number in data register 1 (i.e. `r1`). In the third section (lines 47 to 53), the opposite steps take place; the floating point number for the output sample is converted to fixed point and sent to the codec.

The FIR filter that converts the input samples into the output samples is contained in lines 35 to 45. All the calculations are carried out in floating point, avoiding the need to worry about scaling and overflow. As described in the last chapter, this section of code is optimized to take advantage of the SHARC DSP's ability to execute multiple instructions each clock cycle.

After we have the assembly program written and the filter kernel designed, we are ready to create a program that can be executed on the SHARC DSP. This is done by running the *compiler*, the *assembler*, and then the *linker*; three programs provided with the EZ-KIT Lite. The compiler converts a C program into the SHARC's assembly language. If you directly write the program in assembly, such as in this example, you bypass this step. The assembler and linker convert the program and external files (such as the architecture file, codec initialization routines, filter kernel, etc.) into the final executable file. All this takes about 30 seconds, with the final result being a SHARC program residing on the harddisk of your PC. The EZ-KIT Lite host is then used to run the program on the EZ-KIT Lite. Simply click

Before entering the main program, the following constant and variables must be defined:  
 NR\_COEF           The number of coefficients in the filter kernel (301 in this example)  
 dline[NR\_COEF]   A circular buffer holding the past input samples, in data memory  
 coef[NR\_COEF]    A circular buffer holding the filter coefficients, in program memory

```

001  /*****
002  *****/
003  *****/
004  main:
005
006  /* INITIALIZE THE DAGS TO CONTROL THE CIRCULAR BUFFERS */
007
008  b0 = dline;      /* set up dline[ ], the buffer holding the past input samples */
009  l0 = @dline;
010  m0 = 1;
011  b8 = coef;      /* set up coef[ ], the buffer holding the filter coefficients */
012  l8 = @coef;
013  m8 = 1;
014
015  /* ENTER A LOOP, WAITING FOR THE SAMPLE-READY INTERRUPT */
016
017  wait:
018  idle;
019  jump wait;
020
021
022  /*****
023  *****/
024  *****/
025  sample_ready:
026
027  /* ACQUIRE THE INPUT SAMPLE, CONVERT TO FLOATING POINT */
028
029  r0 = dm(rx_buf + 1);      /* move the input sample into r0 */
030  r0 = lshift r0 by 16;     /* shift to the highest 16 bits to preserve the sign */
031  r1 = -31;                 /* set the scaling for the conversion */
032  f0 = float r0 by r1;      /* convert from fixed to floating point */
033  dm(i0,m0) = f0;          /* store the new sample in dline[ ], and zero f12 */
034
035  /* CALCULATE THE OUTPUT SAMPLE FROM THE FIR FILTER */
036
037  f12 = 0;                  /* prime the registers */
038  f2 = dm(i0,m0), f4 = pm(i8,m8);
039  f8 = f2*f4, f2 = dm(i0,m0), f4 = pm(i8,m8);
040  /* efficient main loop */
041  lcntr = NR_COEF-2, do (pc,1) until lce;
042  f8 = f2*f4, f12 = f8+f12, f2 = dm(i0,m0), f4 = pm(i8,m8);
043
044  f8 = f2*f4, f12 = f8+f12;  /* complete the last loop */
045  f12 = f8+f12;
046
047  /* CONVERT THE OUTPUT SAMPLE TO FIXED POINT & OUTPUT */
048
049  r1 = 31;                  /* set the scaling for the conversion */
050  r8 = fix f12 by r1;        /* convert from floating to fixed point */
051  rti(db);                  /* return from interrupt, but execute next 2 lines */
052  r8 = lshift r8 by -16;     /* shift to the lowest 16 bits */
053  dm(tx_buf + 1) = r8;      /* move the sample to the output */

```

TABLE 29-2  
 FIR filter program in assembly.

on the file you want the DSP to run, and the EZ-KIT Lite host takes care of the rest, downloading the program and starting it running.

This brings us to two questions. First, how do we test our audio filter to make sure it is operating as we designed it; and second, what in the world is a company called *Analog Devices* doing making *Digital Signal Processors*?

## Analog measurements on a DSP system

For just a few moments, forget that you are studying *digital* techniques. Let's take a look at this from the standpoint of an engineer that specializes in *analog* electronics. He doesn't care what is inside of the EZ-KIT Lite, only that it has an analog input and an analog output. As shown in Fig. 29-3, he would invoke the traditional analog method of analyzing a "black box," attach a signal generator to the input, and look at the output on an oscilloscope.

What does our analog guru find? First, the system is *linear* (as least as far as this simple test can tell). If a sine wave is placed into the input, a sine wave is observed on the output. If the amplitude or frequency of the input is changed, a corresponding change is seen in the output. When the input frequency is slowly increased, there comes a point where the amplitude of the output sine wave decreases rapidly to zero. That occurs just below one-half the sampling rate, due to the action of the anti-alias filter on the ADC.

Now our engineer notices something unknown in the analog world: the system has a perfect *linear phase*. In other words, there is a constant delay between an event occurring in the input signal, and the result of that event in the output signal. For instance, consider our example filter kernel in Fig. 29-3. Since the center of symmetry is at sample 150, the output signal will be delayed by 150 samples relative to the input signal. If the system is sampling at 8 kHz, for example, this delay will be 18.75 milliseconds. In addition, the sigma-delta converter will also provide a small additional fixed delay.

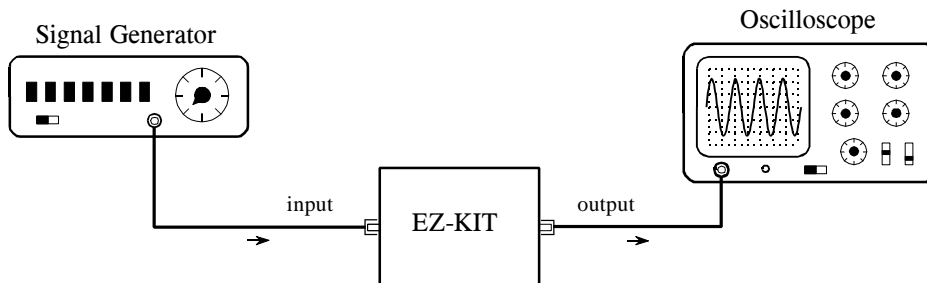
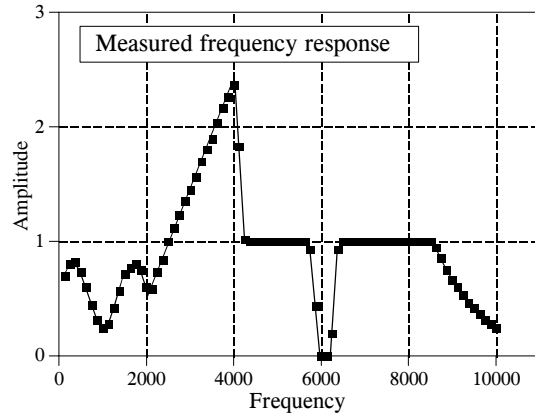


FIGURE 29-3

Testing the EZ-KIT Lite. Analog engineers test the performance of a system by connecting a signal generator to its input, and an oscilloscope to its output. When a DSP system (such as the EZ-KIT Lite) is tested in this way, it appears to be a virtually perfect analog system



FIGURE 29-4  
Measured frequency response. This graph shows measured points on the frequency response of the example FIR filter. These measured points have far less accuracy than the designed frequency response of Fig. 29-3a.



Our analog engineer will become very agitated when he sees this linear phase. The signals won't appear the way he thinks they should, and he will start twisting knobs at lightning speed. He will complain that the triggering isn't working right, and mumble such things as: "this doesn't make sense," "what's going on here?," and "who's been playing with my oscilloscope?" The performance of DSP systems is so good, it will take him a few minutes before he understands what he is seeing.

To make him even more impressed, we ask our engineer to manually measure the frequency response of the system. To do this, he will step the signal generator through all the frequencies between 125 Hz and 10 kHz in increments of 125 Hz. At each frequency he measures the amplitude of the output signal and divides it by the amplitude of the input signal. (Of course, the easiest way to do this is to keep the input signal at a constant amplitude). We set the sampling rate of the EZ-KIT Lite at 22 kHz for this test. In other words, the 0 to 0.5 digital frequency of Fig. 29-2a is mapped to DC to 11 kHz in our real world measurement.

Figure 29-4 shows actual measurements taken on the EZ-KIT Lite; it couldn't be better! The measured data points agree with the theoretical curve within the limit of measurement error. This is something our analog engineer has *never* seen with filters made from resistors, capacitors, and inductors.

However, even this doesn't give the DSP the credit it deserves. Analog measurements using oscilloscopes and digital-volt-meters have a typical accuracy and precision of about 0.1% to 1%. In comparison, this DSP system is limited only by the ~0.001% round-off error of the 16 bit codec, since the internal calculations use floating point. In other words, the device being evaluated is *one-hundred times* more precise than the measurement tool being used. A proper evaluation of the frequency response would require a specialized instrument, such as a computerized data acquisition system with a 20 bit ADC. Given these facts, it is not surprising that DSPs are often used in measurement instruments to achieve high precision.

Now we can answer the question: Why does *Analog Devices* sell *Digital Signal Processors*? Only a decade ago, state-of-the-art signal processing was carried out with precision op amps and similar transistor circuits. Today, the highest quality *analog* processing is accomplished with *digital* techniques. Analog Devices is a great role-model for individuals and other companies; hold on to your vision and goals, but don't be afraid to adapt with the changing technology!

## Another Look at Fixed versus Floating Point

In this last example, we took advantage of one of the SHARC DSP's key features, its ability to handle floating point calculations. Even though the samples are in a fixed point format when passed to and from the codec, we go to the trouble of converting them to floating point for the intermediate FIR filtering algorithm. As discussed in the last chapter, there are two reasons for wanting to process the data with floating point math: *ease of programming*, and *performance*. Does it really make a difference?

For the programmer, yes, it makes a large difference. Floating point code is far easier to write. Look back at the assembly program in Table 29-2. There are only two lines (41 and 42) in the main FIR filter. In contrast, the fixed point programmer must add code to manage the data at each math calculation. To avoid overflow and underflow, the values must be checked for size and, if needed, scaled accordingly. The intermediate results will also need to be stored in an extended precision accumulator to avoid the devastating effects of repeated round-off error.

The issue of performance is much more subtle. For example, Fig. 29-5a shows an FIR low-pass filter with a moderately sharp cutoff, as described in Chapter 16. This "large scale" curve would look the same whether fixed or floating point were used in the calculation. To see the difference between these two methods, we must zoom in on the amplitude by a factor of several hundred as shown in (b), (c), and (d). Here we can see a clear difference. The floating point execution, (b), has such low round-off noise that its performance is limited by the way we designed the filter kernel. The 0.02% overshoot near the transition is a characteristic of the Blackman window used in this filter. The point is, if we want to improve the performance, we need to work on the *algorithm*, not the *implementation*. The curves in (c) and (d) show the round-off noise introduced when each point in the filter kernel is represented by 16 and 14 bits, respectively. A better algorithm would do nothing to make these better curves; the shape of the actual frequency response is swamped by noise.

Figure 29-6 shows the difference between fixed and floating point in the *time domain*. Figure (a) shows a wiggly signal that exponentially decreases in amplitude. This might represent, for example, the sound wave from a plucked string, or the shaking of the ground from a distant explosion. As before, this "large scale" waveform would look the same whether fixed or floating point were used to represent the samples. To see the difference,

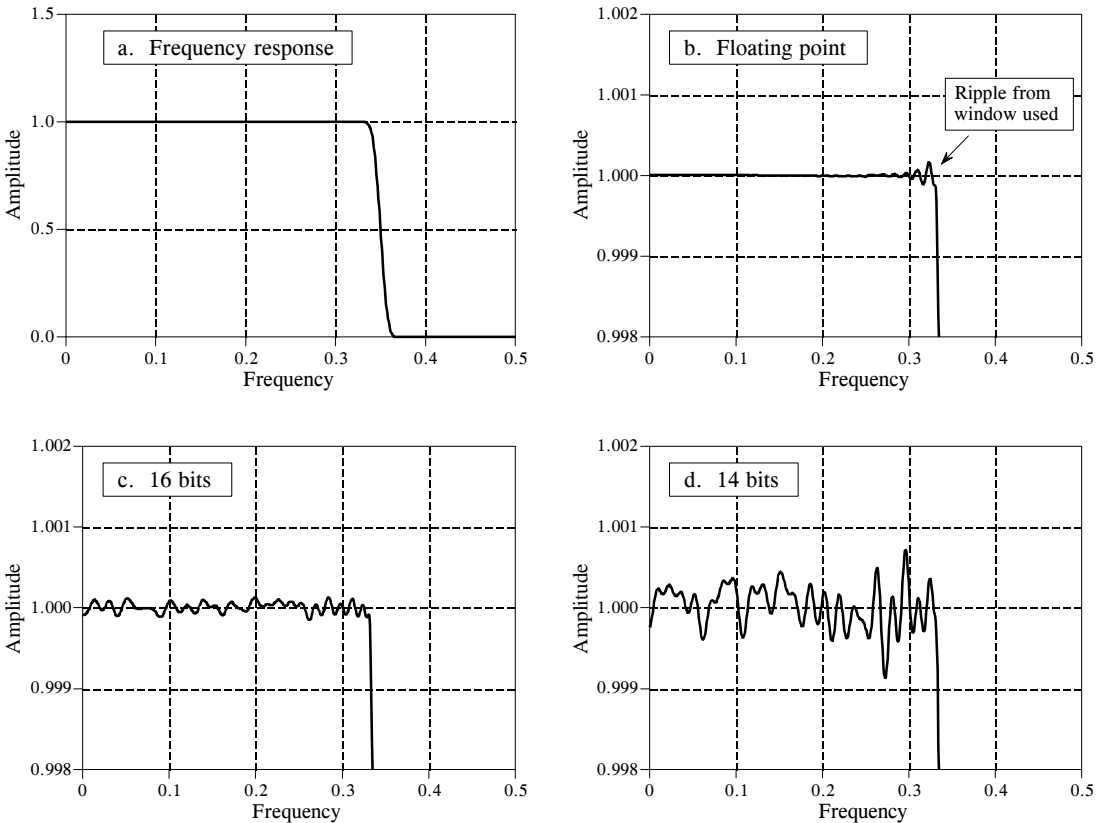


FIGURE 29-5

Round-off noise in the frequency response. Figure (a) shows the frequency response of a windowed-sinc low-pass filter, using a Blackman window and 150 points in the filter kernel. Figures (b), (c), and (d) show a more detailed view of this response by zooming in on the amplitude. When the filter kernel is represented in floating point, (b), the round-off noise is insignificant compared to the imperfections of the windowed-sinc design. As shown in (c) and (d), representing the filter kernel in fixed point makes round-off noise the dominate imperfection.

we must zoom in on the amplitude, as shown in (b), (c) and (d). As discussed in Chapter 3, this quantization appears much as additive random noise, limiting the detectability of small components in the signals.

These performance differences between fixed and floating point are often not important; for instance, they cannot even be seen in the "large scale" signals of Fig. 29-5a and Fig. 29-6a. However, there are some applications where the extra performance of floating point is helpful, and may even be critical. For instance, high-fidelity consumer audio system, such as CD players, represent the signals with 16 bit fixed point. In most cases, this exceeds the capability of human hearing. However, the best professional audio systems sample the signals with as high as 20 to 24 bits, leaving absolutely no room for artifacts that might contaminate the music. Floating point is nearly ideal for algorithms that process these high-precision digital signals.

Another case where the higher performance of floating point is needed is when the *algorithm* is especially sensitive to noise. For instance, FIR

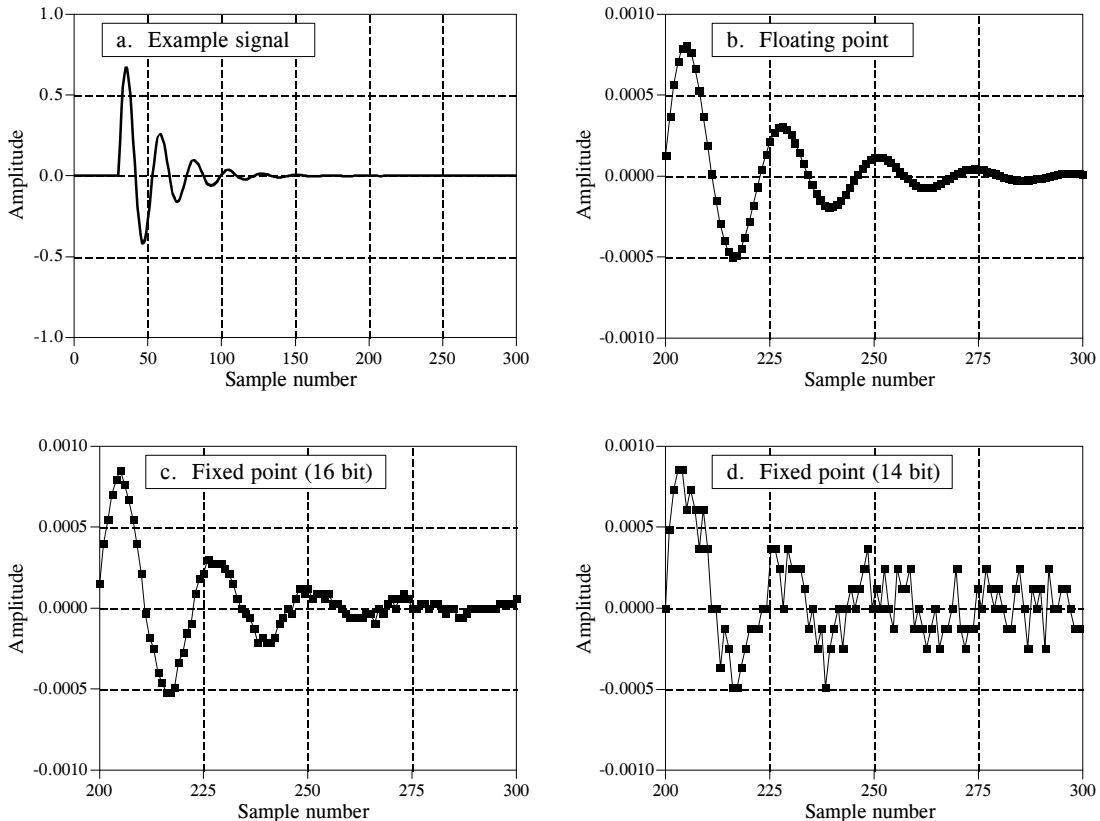


FIGURE 29-6

Round-off noise in the time domain. Figure (a) shows an example signal with an exponentially decaying amplitude. Figures (b), (c), and (d) show a more detailed view by zooming in on the amplitude. When the signal is represented in floating point, (b), the round-off noise is so low that it cannot be seen in this graph. As shown in (c) and (d), representing the signal in fixed point produces far higher levels of round-off noise.

filters are quite insensitive to round-off effects. As shown in Fig. 29-5, round-off noise doesn't change the overall shape of the frequency response; the entire curve just becomes noisier. IIR filters are a different story; round-off can cause all sorts of havoc, including making them unstable. Floating point allows these algorithms to achieve better performance in cutoff frequency sharpness, stopband attenuation, and step response overshoot.

## Advanced Software Tools

Our custom filter example shows the *easiest* way to get a program running on the SHARC DSP: editing, assembling, linking, and downloading, performed by individual programs. This method is fine for simple tasks, but there are better software tools available for the advanced programmer. Let's look at what is available for when you get really serious about DSPs.

The first tool we want to examine is the **C compiler**. As discussed in the last chapter, both assembly and C are commonly used to program

## MATH OPERATIONS

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent of quotient
cabsf	complex absolute value
cexpf	complex exponential
cos	cosine
cosh	hyperbolic cosine
cot	cotangent
div	division
exp	exponential
fmod	modulus
log	natural logarithm
log10	base 10 logarithm
matadd	matrix addition
matmul	matrix multiplication
pow	raise to a power
rand	random number generator
sin	sine
sinh	hyperbolic sine
sqrt	square root
srand	random number seed
tan	tangent
tanh	hyperbolic tangent

## CHARACTER &amp; STRING MANIPULATION

atoi	convert string to integer
bsearch	binary search of array
isalnum	detect alphanumeric character
isalpha	detect alphabetic character
isctrl	detect control character
isdigit	detect decimal digit
isgraph	detect printable character
islower	detect lowercase character
isprint	detect printable character
ispunct	detect punctuation character
isspace	detect whitespace character
isupper	detect uppercase character
isxdigit	detect hexadecimal digit
memchr	find first occurrence of char
memcpy	copy characters
strcat	concatenate strings
strcmp	compare strings
strerror	get error message
strlen	string length
strncmp	compare characters
strchr	find last occurrence of char
strstr	find string within string
strtok	convert string to tokens
system	sent string to operating system
tolower	change uppercase to lowercase
toupper	change lowercase to uppercase

## PROGRAM CONTROL

abort	abnormal program end
calloc	allocate / initialize memory
free	deallocate memory
idle	processor idle instruction
interrupt	define interrupt handling
poll_flag_in	test input flag
set_flag	sets the processor flags
timer_off	disable processor timer
timer_on	enable processor timer
timer_set	initialize processor timer

## SIGNAL PROCESSING

a_compress	A-law compressing
a_expand	A-law expansion
autocorr	autocorrelation
biquad	biquad filter section
cfftN	complex FFT
crosscorr	cross-correlation
fir	FIR filter
histogram	histogram
iffN	inverse complex FFT
iir	IIR filter
mean	mean of an array
mu_compress	mu law compression
mu_expand	mu law expansion
rfftN	real FFT
rms	rms value of an array

TABLE 29-3

C library functions. This is a partial list of the functions available when C is used to program the Analog Devices SHARC DSPs.

DSPs. A tremendous advantage of using C is the library of functions, both standard C operations, as well as DSP algorithms. Table 29-3 shows a partial list of the C library functions for the SHARC DSPs. The math group includes many functions common in DSP, such as the trig functions (sin, cos, tan, etc.), logarithm, and exponent. If you need these type of functions in your program, this is probably enough motivation in itself to use C instead of assembly. Pay special attention to the "signal processing" routines in Table 29-3. Here you will find key DSP algorithms, including: real and complex FFTs, FIR and IIR filters, and statistical functions such as the mean, rms value, and variance. Of course, all these routines are written in assembly, allowing them to be very efficient in both speed and memory usage.

```

/* CIRCBUF.C */
/* This is an echo program written in C for the ADSP-21061 EZ-KIT Lite. The */
/* echo program takes the talkthru program and adds a circular buffering scheme. */
/* The circular buffer is defined by the functions CIRCULAR_BUFFER, BASE, */
/* and LENGTH. The echo is performed by adding the current input to the oldest */
/* input. The delay in the echo can be modified by changing BUFF_LENGTH. */
/* */
#include <21020.h> /* for the idle() command */
#include <signal.h> /* for the interrupt command */
#include <macros.h> /* for the CIRCULAR_BUFFER and segment functions */

#define BUFF_LENGTH 4000

CIRCULAR_BUFFER (float,1,echo) /* define echo as 21k DAG1 reg i1 */
/* a DM pointer to a circular buffer */

volatile float in_port segment (hip_reg0); /* hip_reg0 and hip-reg2 are */
volatile float out_port segment (hip_reg2); /* used in the architecture file */

void process_input (int);
void main (void)

{ /* Make this a variable length array. If emulator is stopped at main */
/* and _BUFF_LENGTH in dm window is modified, the echo delay */
/* is modified. Do not make BUFF_LENGTH greater than stack size! */

float data_buff [BUFF_LENGTH];
interrupt (SIG_IRQ3, process_input);

BASE (echo) = data_buff; /* Loads b1 and i1 with buff start adr */
LENGTH (echo) = BUFF_LENGTH; /* Loads L1 with the length of the buffer */

/* as the array is filled, the nth location contains the newest value, while */
/* the nth + 1 location contains the oldest value. */

while (1)
{ /* the echo sends the sum of the most */
/* recent value and the oldest value */
float oldest, newest;
idle();

/* Echo is pointing to the nth location after the interrupt routine. */
/* Place the new value in variable 'newest'. After the access, update */
/* the pointer by one to point at location n+1. */
CIRC_READ (echo, 1 newest, dm);

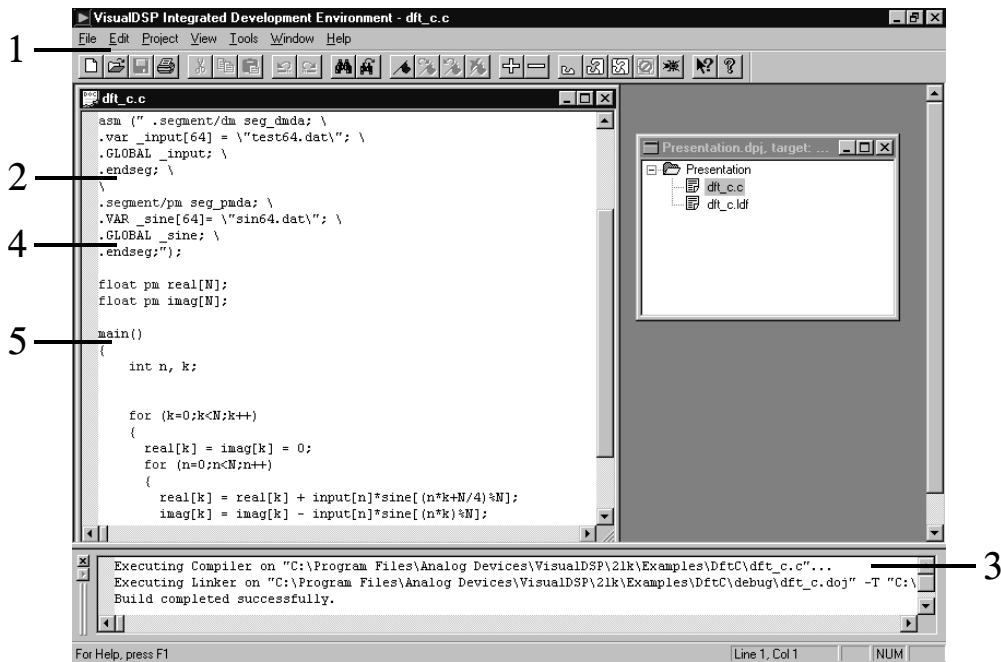
/* Now echo is pointing to n+1. Read the location and place value in */
/* variable 'oldest'. Do not update the pointer, since it is now */
/* pointing to the new location for the interrupt handler. */
CIR_READ (echo, 0, oldest, dm);

/* add the oldest and most recent and send out on port */
out_port=oldest+newest;
}
}

void process_input (int int_number)
{
/* The newest input value is written over the oldest value in the nth */
/* location and the pointer is not updated. */
CIRC_WRITE (echo, 0, in_port, dm);
}

```

TABLE 29-4



1. Move easily between Edit, Build, and Debug activities
2. Mix Assembly and C in a common source file
3. View "build" results
4. Powerful editor understands syntax
5. Easy access through bookmarks

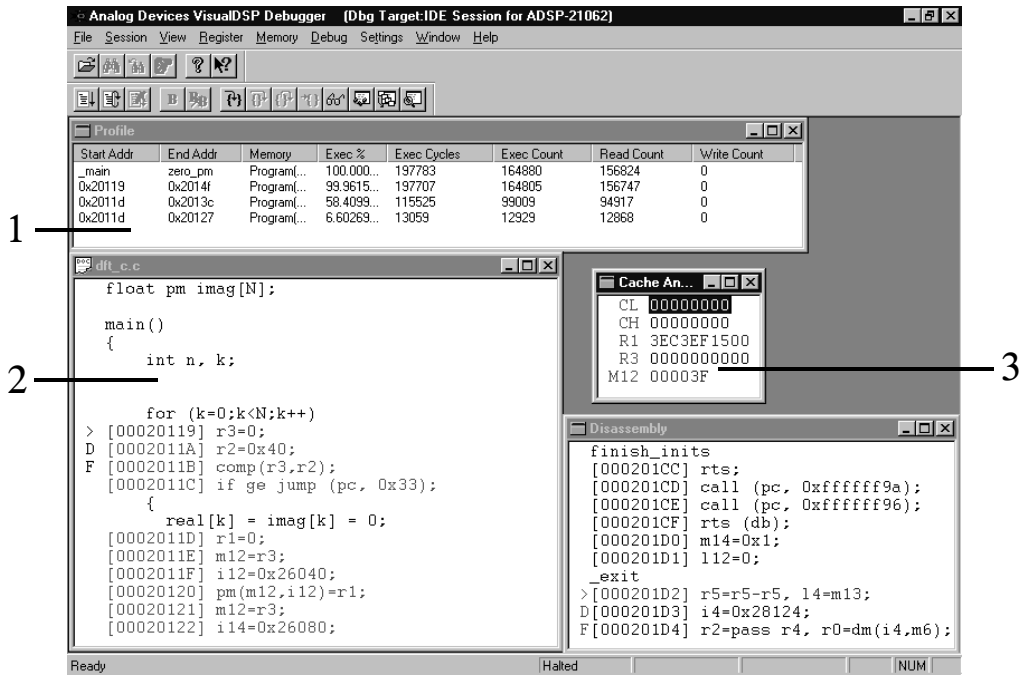
FIGURE 29-7

Example screen from VisualDSP. This provides an integrated development environment for creating programs on the SHARC. All of the following functions can be accessed from this single interface: editing, compiling, assembling, linking, simulating, debugging, downloading, and PROM creation.

Table 29-4 shows an example C program, taken from the Analog Devices' "C Compiler Guide and Reference Manual." This program generates an echo by adding a delayed version of the signal to itself. The most recent 4000 samples from the input signal are stored in a circular buffer. As each sample is acquired, the circular buffer is updated, the newest sample is added to a scaled version of the oldest sample, and the resulting sample directed to the output.

The next advanced software tool you should look for is an **integrated development environment**. This is a fancy term that means everything needed to program and test the DSP is combined into one smoothly functioning package. Analog Devices provides an integrated development environment in a product called **VisualDSP®**, running under Windows® 95 and Windows NT™. Figure 29-7 shows an example of the main screen, providing a seamless way to edit, build, and debug programs.

Here are some of the key features of VisualDSP, showing why an integrated development environment is so important for fast software development. The editor is specialized for creating programs in C, assembly, and a mixture of the two. For instance, it understands the *syntax* of the languages, allowing it to display different types of statements in different colors. You can also



1. Profile code to identify bottlenecks
2. View mixed C and Assembly listings
3. Create custom Register window

FIGURE 29-8

VisualDSP debugging screen. This is a common interface for both simulation and emulation. It can view a C program interspersed with the resulting assembly code, track execution of instructions; examine registers (hardware, software, and memory); trace bus activity; and many other tasks.

edit more than one file at one. This is very convenient, since the final program is created by linking several files together.

Figure 29-8 shows an example screen from the VisualDSP debugger. This is an interface to two different types of tools: simulators and emulators. **Simulators** test the code *within the personal computer*, without even needing a DSP to be present. This is generally the first debugging step after the program is written. The simulator mimics the architecture and operation of the hardware, including: input data streams, interrupts and other I/O. **Emulators** (such as the Analog Devices EZ-ICE<sup>®</sup>) examine the program operation *on the actual hardware*. This requires the emulator software (on your PC) to be able to monitor the electrical signals *inside* of the DSP. To support this, the SHARC DSPs feature an IEEE 1140.1 JTAG Test Access Port, allowing an external device to track the processor's internal functions.

After you have used an evaluation kit and given some thought to purchasing advanced software tools, you should also consider attending a training class. These are given by many DSP manufacturers. For instance, Analog Devices offers a 3 day class, taught several time a year, at several different locations. These are a great way of learning about DSPs from the experts. Look at the manufacturer's websites for details.