Digital Signal Processors for real-time audio processing

Francisco J. Casajús-Quirós
(1) ETSI Telecomunicación-UPM, Ciudad Universitaria, 28040 Madrid, Spain
javier@gaps.ssr.upm.es

Abstract

We analyze the requirements for digital signal processors used in audio applications. Methods for assessing the real computing power expected from them are presented. Proper requirements for a real-time development platform are also described. Code development tools are also included. Finally a review of common processors is done according to those criteria.

1 Introduction

Generation of digital audio effects by high-level software, using off-line processing is fairly common. This is necessarily so in those cases that require strong interaction between the musician/composer/technician and the programs that generate the sound. Many complex effects need the manual adjustment of several parameters, for which purpose a clever operator with good knowledge of the "meaning" of the sound is the only solution. The result of such interaction is usually unique and deeply ingrained in the "art" of sound.

On the other hand there is a growing body of techniques for audio processing that require no interaction whatsoever with the composer, whereas they do allow a very interesting dialogue with the performer. From filtering, through reverberation to real-time harmonisation; a lot of those techniques can be used in real time. As available computing power (per unit cost) increases, more automatic intelligence can be put into those algorithms, thus enlarging the range of what can be done in real-time.

Afterwards we will have gone round the circle. Real-time implementations allow an enormous degree of interaction. The result of changing parameter settings can be heard immediately, in a time-varying fashion if you want. Testing on a wide variety of sounds is easy and convenient. In this way real-time processing becomes a composer tool again.

This paper is addressed to those first considering the possibility (or the need) of implementing sound processing algorithms in real-time. The process might not be easy in certain cases, but after 15 years of general-purpose digital signal processors, a lot can be done by most people with a little effort.

In the following sections a few basic questions are given some kind of answer. Section 2 analyses what kind of DSPs are adequate for audio processing. In section 3 some guidelines concerning the processing speed of processors are be used to show what be be expected in from them in terms of performance. Section 4 describes the external interfaces that are most likely to be used in audio applications. Software tools that can be used in order to generate and debug code are found in section 5. In order to save effort, it is advisable to use a commercial board that incorporates a DSP plus memory and useful peripherals, so that no additional hardware is needed. The elements of boards typically used for audio processing are described in section 6.

2 Wordlength

Of the plethora of DSPs that can be found in the market, only a few can be actually used for audio processing. For many audio applications 16 bit quantization of the input sound is used. This is also the minimal wordlength used nowadays, 8-bit and 12-bit wordlengths being intended for voice systems. However DSPs with 16-bit wordlength should not be used for high-quality audio processing. The reason is well known: the product of two 16-bit numbers has 32 bits. Since this length cannot be handled by the processor, usually the 16 most significant bits are kept, after rounding the least significant of them. Of course this means that we have introduced some error (or rounding noise) in the resulting number. The error is certainly very small (for large numbers), but it accumulates along the processing chain. For a 16-bit input sample, in a 16-bit processor, every multiplication with rounding introduces an error, the power of which equals the quantization noise of the analog-to-digital converter.Thus rounding noise can easily become audible.

The problem can be of course alleviated if we keep 24-bit out of the 32-bit product. In this case the rounding error will affect digits that are 8 places to the right of the least significant bit of the input sample. Rounding noise power resulting from one multiplication is 256 weaker than quantization noise. This means that 256 multiplicative processing stages can be performed before rounding noise has a chance of being heard.

That is the reason why processors like NEC's 77229 or Motorola's DSP56002/DSP56300/DSP56600 feature 24-bit wordlengths.

Figure 1 illustrates this in a particular case. We have taken 10 bits as the wordlength of the analog-to-digital converter for simplicity. The example shows that 5 extra bits for calculations are welcome.

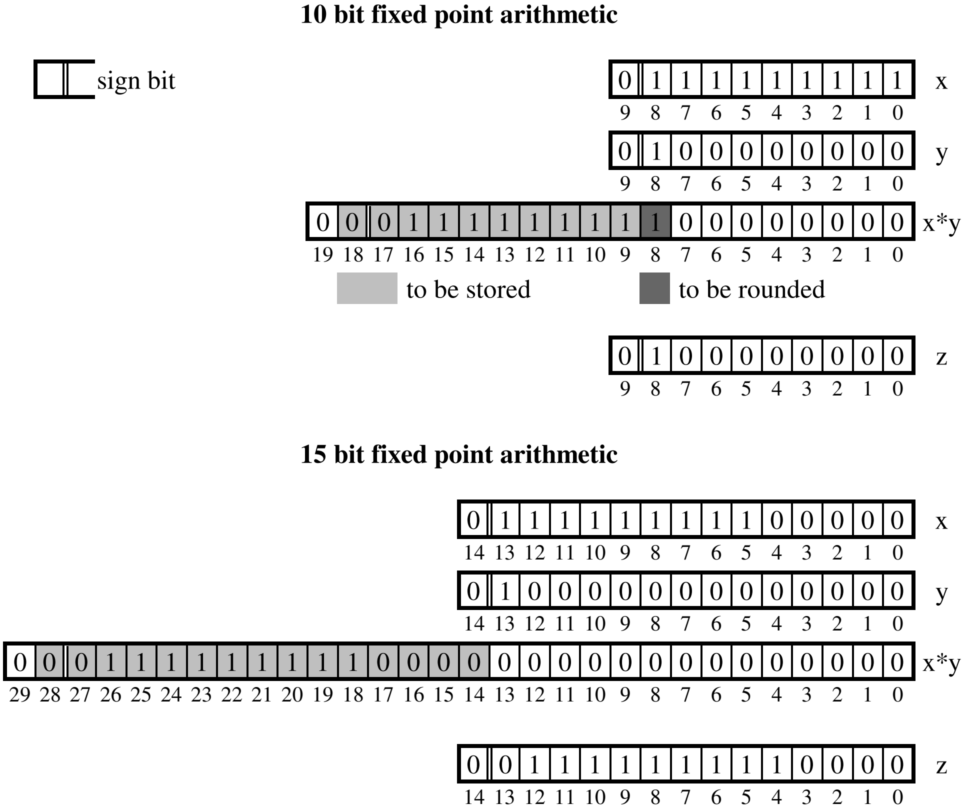

Figure 1: Rounding error for 10-bit samples. If binary point is assumed to be after the sign bit then $x=1-2^{-9}= 0.998046875$, $y=1/2$ and $z=0.4990234375$ (with infinite precision).
For 10 bit arithmetic the number that gets stored is
$z=0.498046875$ (without rounding)
$z=0.5$ (with rounding)
For 15 bit arithmetic the exact result gets stored.

So far we have been only considering fixed-point processors. Another possibility is the use of floating point processors. Practically all of then have 32-bit wordlength with 24-bit mantissas and 8-bit exponents. Since the length of the mantissa is the number of digits that are retained after rounding (or truncation), it is the 24 bits of the mantissa that set a lower bound to rounding noise, equivalent then to 24-bit fixed point arithmetic. The exponent only keeps track on the binary point, it has two beneficial effects: it can handle samples with 24-bit resolution within much wider set of values than fixed point arithmetic; moreover: whereas rounding noise power is fixed for fixed-point arithmetic (no matter the size of signal samples), it is also fixed but relative to the size of the samples for floating point arithmetic. Floating-point arithmetic is also much easier for the programmer, who usually forgets about problems of normalization.
Therefore the use of 32-bit floating point devices like Analog Devices' ADSP-2106x (SHARC), Texas Instruments' TMS320C3x/TMS320C67x or Motorola's DSP96002 is almost always advantageous. However we will show in the next section that, being in general slower than their fixed-point counterparts, they are by no means the selection of choice for audio processing.
In the foregoing paragraphs 16-bit signals have been assumed. With 20-bit and 24-bit signals becoming more popular, 32-bit mantissas (or fixed-point wordlengths) will be necessary. Indeed 32-bit arithmetic can be handled by the above processors, in the same way that 16-bit fixed-point processors can deal with 32-bit floating point arithmetic: through emulation. Unfortunately this means that multiplications are not carried out by single multiply instructions but as sequences of elementary instructions. Therefore the equivalent computational capacity of the processor drops rapidly.

3 Speed

In this section we describe how the computing power of a particular processor can be estimated, either from data provided by the manufacturer, or conducting some simple tests. This must be done in order to assess what is in principle possible for a given processor, and what is definitely out of its capabilities.

It is also important not to overestimate the computing power required by an audio processing application. Although one should in principle use the best processor available, it might be unnecessarily expensive. One should bear in mind that the prices of DSP boards one single user is likely to need, range two orders of magnitude.

Not less important is the fact that selection of the very newest processor might be a source of trouble until its development environment becomes mature (and free of most bugs).

3.1 MIPS, MOPS, MFLOPS

Table I shows as an example the performance of a few processors as indicated by the manufacturer, taken from readily available datasheets. The instruction cycle has also been included in the table. Those processors will be used as an example throughout the article because they are most likely to be used in the future for audio applications [1, 2, 3].

Processor	Performance	Instruction Cycle (nsec)
Analog Devices ADSP-21065	60 MIPS, 180 MFLOPS, 180 MOPS	17
Motorola DSP56302	100 MIPS	10
Texas Instruments TMS320C67	1336 MIPS, 334 MMACS	6

These figures are somewhat ambiguous. For instance the ADSP-21065 is said to perform 60 million instructions per second (MIPS), which is consistent with an instruction cycle of 17 nsec. However since some instructions perform three floating point operations simultaneously (e.g. multiply and add), the processor has a peak computing power of 180 million floating point operations per second (MFLOPS), but in practice this means 60 million multiply-accumulate per second (MMACS). The DSP56302 carries out 100 MIPS that also can be interpreted as MMACS, so no ambiguity results, and the DSP (which uses fixed point arithmetic) is consistently faster than the ADSP-21065, which is floating-point and more or less of the same generation.

The TMS320C67 features a novel architecture in what DSPs are concerned. It has 8 processing units capable of working in parallel. This includes the capability of performing two multiplications and two additions in the same instruction cycle. But the figure of 1336 MIPS must be read carefully. There are 167 million instruction cycles per second. If all 8 units work in those cycles, one obtains 1336 MIPS, but with only 2 multiply-adds per instruction cycle which gives a peak performance of 334 MMACS.

It seems that when one must compare performance, one should use the instruction cycle as a gross indicator, or better still the MMACS figure, although this one might not be obvious from the manufacturer's documentation. Since all processors seem able to carry out at least a multiply-add in one instruction cycle, the MMACS figure can be obtained by taking the reciprocal of the instruction cycle, and multiplying by the number of multipliers-accumulators in the CPU, should there be more than one (this only happens in the case of the TMS320C67, that has 2 arithmetic units).

3.2 Parallelism

It has been mentioned that the TMS320C67 has 8 processing units working in parallel. They are not all equivalent: there are two multipliers, two shifters, two addressing units, etc. It is clear that there will be one instruction being executed at every instruction cycle. Not every program however will be capable of sequencing its operations so as to use all 8 units for a useful purpose at each and all instruction cycles. This means that the full figure of 1336 MIPS will be attained only by certain programs.

Of course this is a fact (not a weakness) inherent in all systems with parallel processing. When computing power from one processor is not enough, one can always try to use multiple processors working in parallel. Analog Devices' ADSP-21065 offers glueless parallel processing with two processors and can be enlarged to many more. TI's TMS320C67 is already a parallel one. It is not our purpose to go into the depths of parallel systems, which require very special operating systems and compilers, but remember that not all algorithms are amenable to processing in parallel and that final performance will in general be far from the performance of one processor times the number of processors.

3.3 Test code

The performance of a processor in MMACS is just a rough indicator of its suitability for a specific application, because it only takes into account multiply-add operations. However computation intensive an algorithm might be, addressing is often as big issue with DSPs as crude computation. After all DSPs have extremely good hardware multipliers accumulators, for them multiplying two numbers is as complex as loading them into registers. That is one of the reasons why more or less utilized benchmarks have been developed. The following are fairly common and well documented for almost all processors:

FIR filter: all processors can one way or another process 1 filter tap per instruction cycle.

IIR filter: addressing is somewhat more involved, usually 4 instruction cycles per biquad section

1024-point complex FFT: addressing very complicated, implementations vary considerably, some processors have addressing units that generate FFT addresses by hardware. As a rule of thumb, addressing can increase execution times by a factor 3 to 6 times the time required by arithmetic operations alone.

Divide: for floating point processors does not have special difficulties. Addressing is minimal, computations are merely multiplies in a Newton iteration with an initial value easily generated. 6 instruction cycles are usually enough.

For fixed point processors division is difficult in general due to the possibility of unbounded results. A lot of effort is spent in controlling arithmetic. Most reported implementations are special cases that can be solved in 24 instruction cycles, for 24-bit wordlength.

These benchmarks have been chosen among other reasons because they are by far the most common operations in digital signal processing. However implementation difficulties resulting from underestimation of the computing power required, are all too common. There are as many cases as implementations. However we will try in the following to present a few examples that illustrate some particular problems that arise frequently.

The multiply-add-shift loop

This loop is by far the most common operation in DSP, because it implements a Finite Impulse Response (FIR) filter, the output of which is calculated as

$$y(n) = \sum_{k=0}^{N-1} b_k x(n-k)$$

Normally output samples are calculated one at a time, so there is no need of storage for more than the latest. However some memory buffer must be provided in order to store past samples of the input. This buffer must be updated whenever a new input sample arrives. This can be done by means of circular buffering and pointers, but the most common solution is to shift values to the next position in the buffer.

C-like pseudocode for this operation should be something like:

```
/* FIR filter with N taps */

real  b[N]  // Filter coefficients
      z[N]  // Filter memory
      aa    // Accumulator
      x     // Input sample
      y     // Output sample

while(1)
{
   x = receive_input_sample();
   aa = b[N-1]*z[N-1];
   for ( n=N-2; n>=0; n-- )
   {
      aa = aa + b[n]*z[n];
   // Multiply-accumulate
      z[n+1] = z[n];
   // Update memory
   }
   z[0] = x;
   y = aa;
   send_output_sample(y);
}
```

Practically any DSP should be able to carry out the `for` loop in one cycle per iteration, with some restrictions. This means a multiply-add operation plus a data move in memory. Therefore the overhead that the loop control instructions create can be very significant. It logically implies that DSP manufacturers include special hardware features so as to be able to perform zero-overhead loops. They might be required to be executed in an internal cache memory, or using exotic forms of addressing. Processor architectures are optimized towards this purpose, so that the programming of multiply-add-shift loops that are maximally efficient should be almost inmediate.

The maximum search routine

The need to find the maximum value of a data table does not appear particularly often in DSP algorithms. It does appear however and also illustrates a typical weakness of DSP processor.

Pseudocode for this operation would look like:

```
/* Find maximum value of L words */

real  x[L]  // Input vector
      xmax  // Maximum value
int   n     // Loop counter
      nmax  // Index of max. value

   xmax = x[0];
   nmax = 0;
   for ( n=1; n<L; n++ )
      if( x[n] > xmax )
      {
            xmax=x[n];
            // Update maximum value
            nmax=n;
            // Update index
      }
```

Most processors can handle zero-overhead loops, they are not restricted to multiply-add loops. However when the group of instructions within the loop includes a condition, regularity is often broken. Although the foregoing algorithm is conceptually simple and does not require any arithmetic operations, DSP processors can be very inefficient when implementing it. Or, at least, efficiency is far worse than in the case of multiply-add loops.

FFT butterfly

The possibility of signal processing in the frequency domain is a key fact of DSP. Nowadays it is practical only because fast algorithms exist for the computation of Discrete Fourier Transforms. These algorithms are known generically as FFT algorithms. They are used very frequently and, as mentioned above, are a typical benchmark for DSP performance. Here we will use them to illustrate another fact of DSP programming. So far our examples have used sequential addressing of vectors, which any processor can handle efficiently by means of autoincrement registers or something similar. But for FFT calculations (and many other DSP algorithms) addressing is quite an issue. Let us examine the basic equation of an FFT butterfly [4]:

$$x_m(p) = x_{m-1}(p) + w(r)x_{m-1}(q)$$
$$x_m(q) = x_{m-1}(p) - w(r)x_{m-1}(q)$$

In general the multiplications and the additions are complex. For our purposes we will ignore this fact in order to concentrate on the addressing problem. Index m indicates the FFT stage, for a given value of it, pseudocode implementing the above computation would be as follows:

```
for(n=0;n<Number_of_butterflies;n++)

{
   p = get_next_p(m);
   q = get_next_q(m,p);
   r = get_next_r(m,p);
   aa0 = x[p]+ w[r]*x[q];
   aa1 = x[p]-w[r]*x[q];
   x[p] = aa0;
   x[q] = aa1;
}
```

Addressing vector x is done in a non linear fashion known as bit reversal addressing. It can be solved in a variety of ways, but it can be as important as arithmetic operations. Remark also that we use two values of the vector in order to calculate those very same values, which requires intermediate variables for temporary storage. Lastly observe than the same product is used in two different additions. Some processors provide means in order to avoid two calculations of the same product or intermediate storage of its value, which would result in additional addressing overloading of the loop.

Block floating point operation

For fixed point processors the dynamic range of internal variables can be a serious concern. The above example of the FFT butterfly could be modified in order to show how such problems are dealt with in practice. For that purpose assume that all variables in the FFT butterfly are fixed-point. They also share a variable exponent for all of them. This notation is known as block floating point. To use it efficiently the processor must provide some means to normalize variables so that the exponent for a block of data can be found easily. If this is done somehow large dynamic ranges, that would result in overflow for fixed point arithmetic, can be coped with as the following pseudocode for the FFT butterfly shows:

```
for(n=0;n<Number_of_butterflies;n++)

{
   p = get_next_p(m);
   q = get_next_q(m,p);
   r = get_next_r(m,p);
   aa0 = x[p]+ w[r]*x[q];
   aa1 = x[p]-w[r]*x[q];
   if ( aa0 or aa1 overflow )
   {
      divide all values of x by 2;
      recalculate aa0 and aa1;
      increase block exponent by 1;
   }
   x[p] = aa0;
   x[q] = aa1;
}
```

The additional code breaks the regularity of the loop by introducing a complex condition. If the condition is met the amount of additional computation and addressing is not negligible. It should then be obvious that block floating point and other fancy notations can be extremely inefficient, unless special hardware resources are available.

4 External interface

The external interface of a DSP processor can be fairly complex. In what follows we will highlight only those elements that are likely to interest the effects developer. For instance many processors allow the use of different kinds of memory: fast, slow, SRAM, DRAM, ROM, EPROM, etc. Interfaces for test and emulation like JTAG are also available. They are very important when the time for a selfcontained implementation arrives, but not at the algorithmic implementation stage. Implementation is best done in development boards that already contain all the elements that are likely to be used by the developer. Of interest to the latter are some forms of interface which can be specially useful.

DMA

Direct Memory Access allows external devices to access the memory without intervention of the control unit. In practice this means that the programmer needs not concern him/herself with the control of the external device. Instead a simple check of the presence of the expected data is enough. For this purpose the processor uses a special-purpose controller with some external pins that must be used by the external device. But once the hardware is set up, interaction with the main program is minimal.

Parallel and serial ports

By parallel ports we understand that the processor has some dedicated pins that allow the transfer of data to or from external devices, more than one bit at a time. Data transfers can be fast, but the programmer must hold control over every single one. For some applications this is exactly what is needed, for instance, when commands or parameters settings are sent to the processor.
Serial ports have more or less the same functionality but minimise the number of external pins, because transfer are done through a single serial pin, plus possibly a couple of control pins.

Multiprocessor support

Analysis of the computational needs of some effect leads sometimes to the conclusion that only several processors working in parallel can provide the necessary computing power. The implementation is almost always far from trivial. Under the circumstances it is very useful to utilize processors that provide support for parallel processing.
Parallel processing among several processors usually implies the transfer of large amounts of data or, at least, control information. This is usually done by means of ports (serial or parallel) because the programmer must often hold complete control over the amount and timing of data transfers.

5 DSP boards

Most audio effects implementation for real time can be done on general purpose development boards. All manufacturers also provide these boards. Besides that third parties also produce those kind of boards usually with more sophisticated hardware, that is to say with more peripherals and capabilities of interaction with the external world. In what follows we will list the very few requirements that the effects developer is likely to appreciate. See figure 2.

ADC and DAC

Obviously no audio processing is possible without analog audio input/output capabilities. Most boards, even the cheapest ones, have a stereo 16-bit audio codec, preferably using DMA channels or serial ports. The manufacturer usually provides examples of input/output routines in order to ease the use of the analog interface.
Some boards allow only a limited number of sampling frequencies. In others an oscillator must be physically changed in order to alter the sampling frequency.

Serial interfaces: MIDI, AES/EBU, MADI

For musical applications the developer often needs to receive, transmit or manipulate musical data in MIDI format. All processors can in principle handle the format via a serial or parallel port. Electrical compatibility will usually require some electronics external to the processor chip. Some third party manufacturers provide this.
As a limited example considerer that for a PC board it is difficult to keep the analog noise level below the level of the least significant bit of a 16-bit ADC. The problem is much worse for higher resolutions. Therefore external, high quality, professional conversion systems can be much better that onboard converters. But some form of digital interface must be provided in order to access the data. Serial ports can be enhanced with drivers, TAXI chips, etc. in order to communicate with external systems via standard interfaces like AES/EBU, SPDIF, MADI, etc. Propietary interfaces for some systems already exist. However they usually communicate with a board of the same company where the DSP processor cannot be chosen, if it can programmed at all by the user.

Host interface

Development boards are hosted usually by a personal computer using a PCI slot (more expensive solutions for other buses like VME also exist). The host is used of course to download programs to the DSP. Control of the execution, debugging and so on is also carried via the host. In principle such tasks can be accomplished by using the serial port of the PC. For audio effects applications the developer will usually like to interact with the processing in real time. This means that some parameters are changed. The result is listened to even as those changes occur. Changes are very often done by means of the PC user interface. A fast interface with the PC plus some support (in the form of routines) from the board manufacturer are essential.
In this sense the board shoud be apt to accessed by a user program running on the PC. This opens the possibility of communicating other PC boards via user programs. For instance, data from/to MIDI input/output boards can be passed to/from the DSP board via the PC and so on.

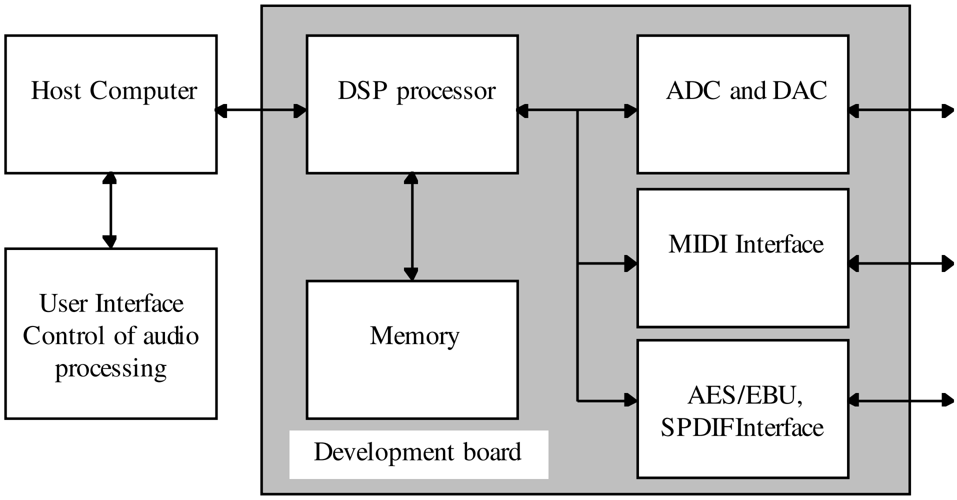

Figure 2: Typical structure of a DSP system for audio processing

6 Code development

A final word about the tools that are used to create and test the program that generates the effect on the DSP processor. All manufacturers provide the following tools for their processor. They are software tools that run in principle with no processor board at all.

Compiler

C compilers can be very useful in order to create a first working version of the processor code. They are practical for floating point processors. Less so for fixed point processor because fixed-point arithmetic is somewhat involved, many times full control of arithmetic problems is only possible with low level programming.
On the whole the main drawback of C compilers is they produce code that is not very efficient. If the computing power available is big, this is no problem at all. When the programmer is using the processor at the utmost speed, low-level programming can be advisable. In any case the C code must be written after the style guidelines of the manufacturer in order to produce compilable, efficient code.

Optimizer

Some manufacturers include an optimizer as a separate program. It can sometimes greatly improve the performance of an assembly or C program. Sometimes optimizers crash hopelessly

Assembler/Linker

Assembler/linkers are essential in order to produce the file that is finally loaded in the board memory. They take their input in assembly language. Some programmers never use other thing. For the nonexpert its direct use is necessary for development of computation intensive applications.

Simulator/Debugger

Once the program has been produced it is necessary to check its correct performance. Simulator/debugger programs allow the execution of the program as though it were running on a real processor. Simulated registers, memory, etc. can be examined. They do not require a real processor to be present and can provide useful statistics such as execution times. Interfacing to test signals contained on disk files is also greatly simplified.
Some of them also allow the possibility of using a real processor on an evaluation board in order to execute the program. The debug interface remains the same. There is also the possibility of accessing real peripherals such as MIDI or ADCs. But bear in mind that, as soon as they use a board, they are hardware dependent and must be provided by the board manufacturer.

7 Conclusions

A comparison of the three processors mentioned in section 3.1 was intended at this point. Due to difficulties in the availability of some of them it has been impossible to carry out the test in time. The results will be presented at the conference.

References

[1] Analog Devices. 1998. "Datasheet of ADSP-21065".

[2] Motorola. 1998. . "Datasheet of DSP56302".

[3] Texas Instruments. 1998. . "Datasheet of TMS320C67"..

[4] Oppenheim, A, and Schafer, R. 1989. *Discrete-time signal processing*, Prentice-Hall.