CADiSP - A GRAPHICAL COMPILER FOR THE PROGRAMMING OF DSP IN A COMPLETELY SYMBOLIC WAY

Alois Knoll and Rupert Nieberle

Technische Universitaet Berlin
Sekr. FR 2-2
D-1000 Berlin 12
West Germany

ABSTRACT

The CADiSP system is an experimental software development environment for digital signal processing. It has been designed in an attempt to provide an engineering oriented tool for the development of algorithms and efficient code generation for DSP applications. Following the rules of modern software engineering it supports object oriented techniques and symbolic programming. This paper discusses the approach of using an interactive graphical editor for specifying systems by interconnecting black boxes hierarchically on different levels of abstraction and then automatically translate the block diagram into executable code. Basic building blocks for the most frequently used DSP functions are supplied in a library but the programmer may freely define new functions using a specially designed new language for expressing DSP algorithms. He may also use the assembler language of the target processor which is ameliorated by graphical elements to support structured programming even on this level. On all levels of abstraction the graphical specification is complete and thus may also serve as documentation. Possible extensions of the system are briefly discussed.

INTRODUCTION

The design goals for a flexible and powerful development system covering a broad range of applications are manifold. First of all, the system should be based on an extensible programming language suitable for the notation of DSP-algorithms. Secondly, it must be easy and intuitive to work with so as to make program development a rapid process. Therefore, compilers for the compilation of block diagrams as well as for the compilation of the underlying algorithmic language, a graphical interface for interactive entry of block diagrams, a simulator and a debugger are major components. The system must provide mechanisms to support the efficient reuse of software. Finally, it must be modular in order to be easily extensible, e.g. to support programming of multiprocessor systems.

The aim is to make program development painless even for a user unfamiliar with specific implementation details of the development system and only interested in his own application. We present an outline of our experimental system being designed following the guidelines of other systems that evolved in recent years [1...3] but stressing the issue of easy extension of functionality, integrating aspects of modern software engineering and in particular generating efficient code (the system performs code generation primarily for the DSP 56000).

LEVELS OF ABSTRACTION

The system is divided into three main levels of abstraction:

a) the problem-oriented level III on which applications are built using powerful modules
b) the level of primitives (level II) on which the sophisticated modules available on level III are defined using primitives like multiplication, addition, etc. and
c) the processor level I on which primitive functions are defined for use on level II.

By employing the same or similar graphical methods and rules on three levels of abstraction, the user may seamlessly integrate software specific to his application into predefined modules on each of these levels.

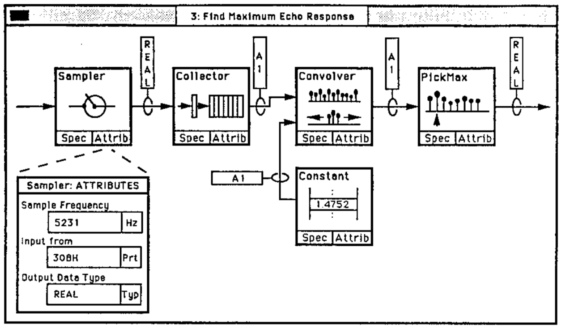
The Application-Oriented Level III

On the *problem-oriented* top level (level III), processing entities may be picked from a library of powerful building blocks, e.g. samplers, convolvers, signal-generators or filters. These entities can be "plugged together" by placing a labelled arc between any two of them (see fig. 1). The direction of the arc linking two processing entities indicates the direction of the data-token flow. The labels of the arcs denote the data type of each element of the stream of data-tokens. The data flow conforms to the synchronous data flow model [4] which essentially implies that the number of data tokens processed by each building block on its invocation is known and fixed. Therefore, finite buffers between entities will suffice. The data types of input/output data streams of connected entities must match and they must be defined at compile-time, nevertheless generic definitions of processing entities accepting different kinds of input/output streams are possible. In the case of a generic definition, the graphical compiler will decide at compile-time which instance of the processing element to choose. This mechanism simplifies the reuse of software modules in different processing contexts without sacrificing the benefits of strong typing.

If a user wishes, one or more of the building blocks of level III may form the basis for even more powerful building blocks, the latter inheriting the properties of the former and thus constituing an even higher level of abstraction. Building blocks on all levels may be copied and renamed to exclude them from changes made to the original prototype of the class.

An *Attrib*ute and a *Spec*ification field characterize the building block in full detail. If activated ("clicked on"), the attribute window displays parameters necessary for tailoring the function

CH2847-2/90/0000-1077 $1.00 © 1990 IEEE



Sampler: ATTRIBUTES
Sample Frequency 5231 Hz
Input from 308H Prt
Output Data Type REAL Typ

Fig. 1: Program Generation on Level III

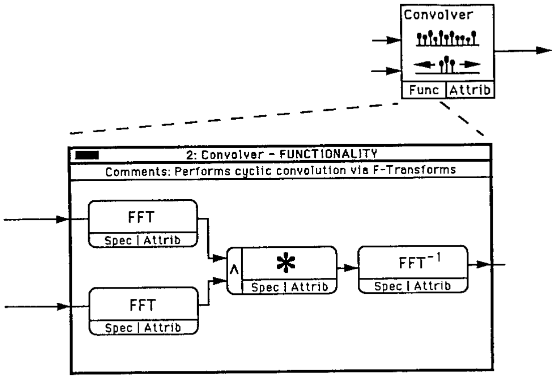


Fig. 2: Level II - DSP-Primitives

to the specific needs of the environment it is used in. All of these parameters may be changed upon activation of the attribute window. The activation of the specification window reveals the interconnection of level II blocks that realize the services of the level III block. This specification is depicted graphically in terms of the graphical data-flow language of level II (see fig. 2).

Level II: Building Blocks from DSP-Primitives

On level II, common *DSP-primitives* are predefined. As was the case on level III, these primitives are visualized graphically. Plugging together these primitives, the functionality of the blocks of level III is obtained (fig. 2). DSP-primitives available on this level are addition, multiplication, shifts, FFTs, etc. over predefined or user-specified data types. Obviously, the separation between levels II and III is not always clear cut: It might be argued, that an FFT is a top-level operation. However, if the user wishes to make this operation available on the top level, it may easily be "wrapped" into a building block of that level.

Down to and including level II, all building blocks are strictly processor-independent. Nothing has been said about the architecture of the processor nor the language the algorithm was written in. Therefore, a simulation of the final application is easy in principle (not yet implemented) and straightforward if the primitives are implemented to run on the host computer of the development system: Apart from temporal behaviour and possible numerical differences due to integer arithmetic, the application will run on a dedicated signal processor in exactly the same way as on the host computer. After the algorithm proves to work correctly, the primitives running on the host can be replaced with the primitives executable on the DSP.

Level I: Program Code

The realization of the primitives of level II reveals upon activation of the "Spec" field (fig. 2). On the lowest level I, the user has access to the *program code* realizing the functions of level II. Even on this processor-dependent level, all programming can be done in a way comparable to the higher levels but more textual entry is necessary. Block diagram and related graphical techniques are perfectly adequate for representing the interconnection of system building blocks and data flow, but for expressing algorithms they have not proven to be useful tools [5] and are certainly inferior to programming languages. It is possible to do all the programming graphically by introducing icons representing data, control flow, a register model of the processor and addressing modes. This does have advantages pertaining to readability and self documentation [6]. However, when organizing programs this way, it turns out that even moderately complex programs rapidly need much more space than can be handled comfortably. It is more promising to use only a few graphical elements to indicate control flow and possible parallel operations (e.g. data moves) or to resort to a powerful programming language permitting compact notation of algorithms. This is the main reason why programming on this lowest level can be done using ImDiSP, a high-level language specifically designed for digital signal processing and control systems applications. Alternatively, the functionality of blocks may be specified using the assembler language of the target processor. As will be shown below, the introduction of graphical elements on the assembler level may reduce textual entry to a minimum.

A) The programming language ImDiSP

The imperative language ImDiSP has been designed with two goals in mind:

- The special functionality of widespread digital signal processor architectures should be accessible to the high-level language programmer.
- The language should provide constructs that permit the specification of typical DSP algorithms without sacrificing potential inherent parallelism of calculations, i.e. without having to sequentialize them because of constraints imposed by the language (as is usually the case with programs written in C or Pascal).

Like every modern imperative language ImDiSP is strongly typed and offers all the control constructs necessary for structured programming (sequence, iteration, selection) as well as procedures and blocks as basic means for structuring program text. Basic data types are boolean, integer, real and complex. These types may be used to form arrays and records. The expressive power of the language is based on its set of operators manipulating array structures, the predominant data type in DSP applications.

All standard arithmetic operators are overloaded: They do not only operate on integers and reals but also on complex numbers, arrays, slices (sub-arrays) and matrices composed of the base types. A simple example is the multiplication of two vectors: Let a and b be vectors of the same dimension. Then, c := a*b multiplies these vectors element by element creating a new vector which is assigned to the variable c (also of type vector).

Like in APL, it is possible to combine "inner" and "outer" operations using a dot notation. For example, the dyadic operator +.* applied to two vectors (as in c := a +.* b) will first multiply the vectors element by element and subsequently sum up the products to yield a result of the base type of the vectors. Using this technique of combining operators, an expression like

$$y = \sum_{i=0}^{N} h_i x_{N-i}$$

is easily translated into the following code segment:

```
y := x[^ .. 0] +.* h
```

where "^" used as an array subscript denotes the upper bound of the array. The difference equation of a recursive filter

$$y_n = \sum_{\nu=0}^{N} a_\nu x_{n-\nu} - \sum_{\mu=1}^{M} b_\mu y_{n-\mu}$$

with coefficient vectors a (dimension N+1) and b (dimension M) can be realized in just a few lines of code:

```
loop
        x[^] := input;                                    -- get input value
        y[^] := a +.* x[^ .. 0]  -  b +.* y[^ - 1 .. 0];  -- do the filter
        output := y[^];                                   -- output result
        delay (x, 1);        -- shift contents of array x by one element
        delay (y, 1);        -- shift contents of array y by one element
end loop;
```

where input and output are I/O port addresses.

The standard procedure delay pushes the contents of an array down by an arbitrary number of elements leaving the top elements (high index) undefined. This procedure is normally applied to objects of a special one dimensional array type circular array that implements circular buffers of any base type. If certain conditions are met, the compiler makes use of the modulo-n addressing mode (a feature available on most signal processors) when delaying the array contents. This results in very low overhead code. Other standard procedures of the language are biquad, used for realizing cascade filters and the procedure butterfly that implements DIT/DIF butterflies and the necessary address calculations.

Processor features directly accessible to the ImDiSP programmer are saturation arithmetic which may be turned on and off before and after any instruction by means of a pragma (metacommand) and hardware do-loops. Bit-reversed addressing of array elements is possible. Operators are provided that permit the uniform application of a certain operation to all elements of an array, the extraction of matrix diagonals and the determination of the smallest or greatest element of an array.

To allow procedures to have state, all variables declared local to a procedure are static by default, i.e. their value does not change between calls. A pragma exists that can make them volatile which means that they become undefined upon exit of the procedure. Volatile variables can be held in registers which removes the need for memory fetches thus making execution much faster.

A future extension of the language will be representation clauses of data types that direct the compiler to internally represent data types according to the specification of the programmer. If it turns out to be necessary, pointer types and associated operators will be added to the language. It is not intended to introduce language constructs supporting programming in the large (module, import, export statements, etc.). As ImDiSP is primarily intended for defining algorithms "behind" level II building blocks, program size should always remain on a relatively small scale and procedures are sufficient for structuring the program. All aspects of modularization and information hiding are realized by the higher CADiSP levels.

Using ImDiSP, it is possible to specify algorithms involving structured data in a very compact form leaving it to the compiler to serialize the code as far as necessary. This way it is obviously much easier to generate efficient code than based on a notation that forces the programmer to write down sequential steps and then try to re-parallelize them.

B) Structured Assembler

The basic routines of the CADiSP library are based on standard subroutines as furnished by the manufacturer of the processor. Combining them on levels II and III and adding user-supplied routines written in ImDiSP where necessary should cover most potential applications. Nevertheless, when a user needs a certain function not available in the library and hard limits are imposed on the execution time, he may wish to implement his own routine in assembler. To facilitate this task, graphical elements resembling those of flow charts are used. The purpose of these elements is twofold: They are used to express control flow and force the programmer to declare which resources of the machine are used by his routine in a structured way.

A sample program composed of these graphical elements available on this level is shown in fig. 3: Each of the blocks contains an instruction; control flows along the lines. Lines may be drawn that are expanded to unconditional jumps automatically. Loops are denoted by arrows and a loop count. When programming, the user picks instructions from a menu of available instructions and then is prompted for the operands. Upon completion of programming the procedure he must enter all the information required by scheduler (registers used, entry/exit/main procedure names, amount of buffer memory, etc.) into a predefined table (not shown in fig. 3).

Our experiments exploring the benefits of this support of assembler programming are still of a very preliminary nature. However, the experiments indicate that it may be of some help for the novice programmer, certainly improves clarity of documentation and helps to prevent unstructured coding.

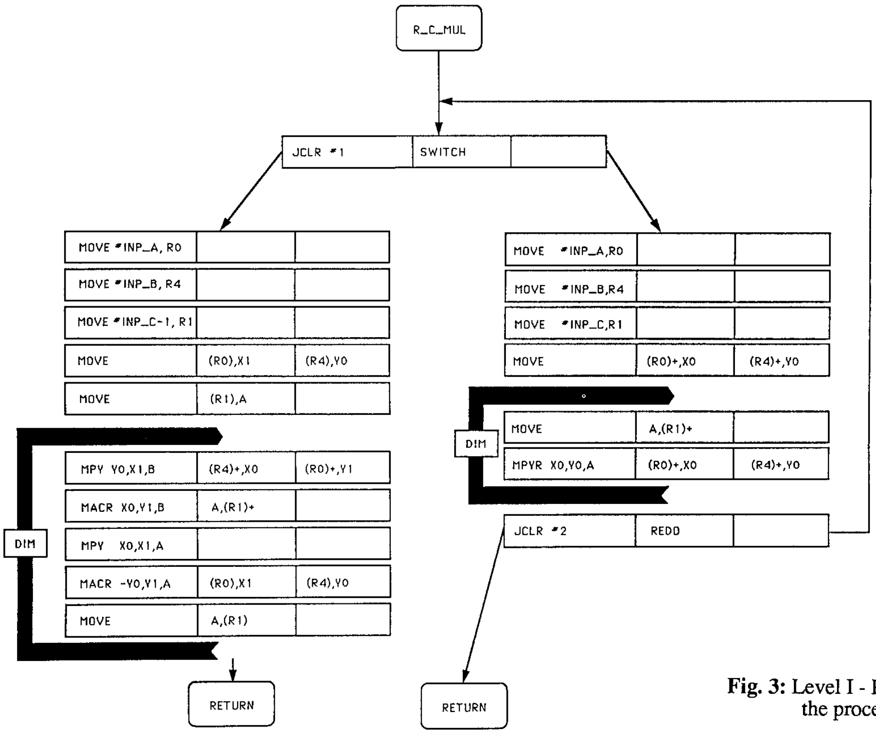


Fig. 3: Level I - Programming the processor

SCHEDULING

Scheduling, i.e. the determination of the order and frequency in which subroutines representing building blocks are called, is done completely at compile time. No dynamic scheduling, as necessary with asynchronous data flow models, takes place. The approach we have taken is simple: The scheduler examines the topography of the network and the number of data tokens that are produced/consumed on invocation of each block. Based on this input it creates a calling scheme for the subroutines and allocates memory space for circular buffers if necessary. Input/Output will be interrupt or event driven in most cases; no scheduling is necessary here.

CONCLUSION

We have outlined a system for programming DSP applications. With this system, most of the programming can be done graphically, the user may fully concentrate on the solution of his problem. He will not find himself fiddling around with parameter passing mechanisms, register allocation schemes and other issues not related to his signal processing problem. Moreover, this approach results in a natural way of modularization: All building blocks of level III and all primitives of level II are completely isolated from each other. Each of these instances has its own data-space, there are no common variables and consequently no undesired side-effects. Data and code of the instances are firmly encapsulated within the module with no access possible from the outside.

The graphic layout provides for complete documentation: It represents specification, program and comments, all rolled into a single compact document. Possible future extensions of the system are better scheduling algorithms, a postoptimizer to improve efficiency of the generated code, placement of processes on multiprocessor networks, addition of simulation facilities and the integration of a debugger.

REFERENCES

[1] Kopec, G.
The Integrated Signal Processing System ISP
Trans. IEEE ASSP-32, No. 4, 1984

[2] Karjalainen, M. et al.
QuickSig - An Object Oriented Signal Processing Environment
Proc. ICASSP-88, New York, 1988

[3] Lee, E.
Programmable DSP Architectures: Part II.
IEEE ASSP Magazine, Vol.6, No.1, January 1989.

[4] Lee, E. et al.
Synchronous Data Flow
Proc. IEEE, Vol. 75, No. 9, Sep. 1987

[5] Shu, N.
Visual Programming
Van Nostrand Reinhold, NewYork 1988

[6] Nieberle, R. and Knoll, A.
CADiSP - An Approach for the Integration of Interactive, Real-Time and Distributed Features in an Environment for the Development of DSP-Programs
89th Convention of the AES, New York, 1989