D6.23

A LISP-BASED HIGH-LEVEL PROGRAMMING ENVIRONMENT FOR THE TMS320C30

Matti Karjalainen

Helsinki University of Technology, Acoustics Lab.
Otakaari 5A, 02150 Espoo, Finland

ABSTRACT

In search for more flexible and powerful tools for DSP programming we have developed an integrated programming environment called QuickSig [1]. This paper describes the subsystem for signal processor code generation. Our target processor in this study has been the TMS320C30. The entire software system is based on the use of the Common Lisp language to achieve an exceptional degree of system integration resulting in high flexibility, interactivity and programming productivity. All levels from algorithm development to object code generation, including the assembly language, use the same Lisp notation. The principles and implementation of the Lisp-based programming environment for the TMS320C30 are discussed along with some programming examples.

INTRODUCTION

New generation signal processor chips like the TMS320C30 have a high throughput and a flexible set of floating-point operations [2]. Yet currently, the optimized program code production is largely based on such low-level tools as an assembler. The C language is rapidly becoming available and standard for new DSP chips [3], [4], but it must be combined with routine libraries and assembly code to achieve the best optimizations. What is even more difficult is to flexibly integrate these programming tools with interactive high-level simulation and algorithmic development environments.

The need for integrating all of these programming levels was the starting point of this study. The high-level DSP programming environment QuickSig, developed by us and reported earlier [1] offered a good basis. QuickSig is implemented in the Common Lisp language and it is a highly interactive software tool supporting a wide variety of DSP functions and operations with good graphics facilities. It also includes an experimental block diagram editor and compiler. This approach is especially interesting for DSP processor code generation.

The QuickSig system which includes the TMS320C30 programming tools has been developed on the Symbolics Lisp machine. The TMS320C30 software is portable and runs also on Apple Macintosh computers with a minimum of 2 MBytes of RAM memory. Other parts of the QuickSig software are mostly based on the New Flavors object oriented programming framework and can be used on Symbolics machines only.

Some rethinking of DSP program development

Traditional and current DSP programming tools lack several features that are available in the best modern software environments. We feel that it is a common misconception to think that DSP processor programming is and should be a kind of low level activity. High level intelligent tools are needed to solve complex problems and DSP programming with all its performance demands and code optimization requirements belongs definitely in this domain. Some of the most desirable features that should be included are:

* High integration level of the programming system and its subparts. There should be a natural inteface between the different parts (e.g. common syntax). All tools should be interactive and available at the same time instead of switching between algorithmic development, compilers, assemblers, simulators, emulators, editors etc.
* Good integration of software and hardware tools is needed. There should be support for multiple processors, both real and simulated. (See the idea of generic processor below.)
* Advanced user interface including both graphical and textual means of interaction with windowing techniques and mouse & menu driven control.
* Fast programming by incremental compilation and reduction or elimination of the "edit-compile-link-load-debug-and-back-to-editor"-loop. The tools like compilers, assemblers, simulators etc. should also be fast (at least thousands of machine instructions per second).
* High reusability of all program constructs. One important approach to achieve this is the object-oriented programming paradigm.
* Extensive support of libraries and other program constructs.
* Various levels of data and procedural abstraction for programming should be available.
* Several forms and levels of both automatic and user controllable code optimization should be supported.
* The system and its parts should be extendable and programmable by the user to create new levels of abstraction.
* Easy manipulation of symbolic expressions and computational forms at all levels of the system.

Our experimental programming environment described below supports all these requirements or reveals a high potential for the implementation of these features. The weakest link is the lacking support from traditional Fortran and C libraries. This is no principal limitation; due to the exploratory nature of the study towards new approaches we had to exclude this interface.

Why Common Lisp as a system language for DSP

Common Lisp [5] is not a well known and widely used programming language in the DSP society. As a system language it has several advantages that make it attractive, e.g.:

* Excellent symbolic manipulation features. Lisp is the best known implementation language for artificial intelligence.
* Coherent and flexible syntax for all kinds of programming. Common Lisp is perhaps the most general purpose language available.
* Good numeric computation features including complex numbers, rationals and bit-level manipulation.
* It supports a wide variety of important data structures (e.g. several types of arrays).
* Excellent extendability and add-on features like object-oriented programming, good editors, debuggers, etc. It has an incremental compiler for flexible and fast program development.

There are also some drawbacks like the relatively high demand on memory space and slowness that have traditionally made Lisp somewhat unpractical. New implementations like Allegro Common Lisp [6] on the Apple Macintosh run small to medium size programs efficiently even with a small 1-2 Mbyte memory. So called Lisp machines give the highest performance and the best tools for program development.

CH2673-2/89/0000-1150 $1.00 © 1989 IEEE

SYSTEM DESCRIPTION OF THE PROGRAMMING ENVIRONMENT

Instead of looking at separate programming tools like compilers, assemblers, simulators, debuggers etc. we have approached the building of the programming environment from an integrated software point of view and from the users perspective. Fig. 1 shows the relations of the sub-units.

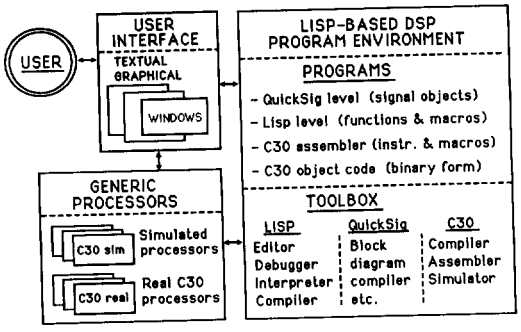

Fig. 1. System description of the Lisp-based DSP programming environment

The user interface allows for interactive operations on any parts of the system at any time. Textual interaction (by Lisp interpreter or more specialized dialogs) is the most general way of accessing all objects. Graphically oriented access by multiple windows, mouse control, menus etc. has more limited access but offers a much more natural and convenient way of communicating with the more object-like parts of the system.

The Lisp-based programming environment is composed of a toolbox and the programs of different abstraction levels. The toolbox contains general Lisp-level tools, QuickSig tools and the C30 tools especially for TMS320C30 programming. Lisp tools include the Lisp editor, interpreter, compiler and debugger. A specific feature is that the Lisp compiler is used to construct the whole system and all its components. The Lisp interpreter (Lisp listener) and editor are at the same time the most fundamental textual components of the user interface. The Lisp debugger can be used for debugging all levels of programs including the C30 assembly programs.

The tools for the TMS320C30 are the assembler, the source-code simulator and the compiler. These components are used primarily by the system itself; the user interacts normally only with the programs and processors so that these tools are needed explicitly only during some debugging. The QuickSig level tools are briefly discussed later.

Programs in the environment may represent different abstraction levels although they always follow the Lisp syntax. C30 object code is very seldom seen by the user. C30 assembly code is the symbolic machine code with Lisp compatible list notation. It can also be interpreted which makes the source-code level simulation of the C30 possible. The ordinary Lisp language level is the fundamental way of constructing programs, building new tools and extending the system. The C30-compiler converts Lisp programs to C30 assembly code (that still has a list syntax). The QuickSig level is a new abstraction mechanism that adds several DSP-specific features to the environment.

The most important object for the TMS320C30 programmer is the processor itself. In our system the user can observe a set of generic processors. These can be simulated ones or mappings of real processor hardware. The idea of generic processors means that the user as well as the programs view both kinds of processors (almost) identically. These processor objects can be monitored and called for execution in the same way and they can communicate with each other to allow for multiprocessor programming.

TMS320C30 ASSEMBLER WITH LISP SYNTAX

The main idea for the integration of high- and low-level programming in our system is to use a modified assembler syntax for the TMS320C30 that is entirely compatible with Lisp notation. This immediately makes it possible to call any assembler instruction from the Lisp interpreter (interactive simulation) and to include instructions in any high-level Lisp programs (source-code level simulation) including QuickSig-level functions. The modified list notation vs. normal assembly syntax is compared in the following examples:

Normal syntax:	*Lisp syntax*: (case insensitive)
`LDF -12.3,R0`	`(.LDF -12.3 '.R0)`
`ADDF3 *AR0++(IR0),R0,R1`	`(.ADDF3 '(.AR0 >++ .IR0) '.R0 '.R1)`

It is easy to convert standard C30 source code between these notations although we have not yet implemented this.

The C30 addressing modes with Lisp syntax are denoted in the following way. If the operand is a number it means immediate addressing. A register symbol is interpreted as register addressing. (Register names as well as many other assembler-level symbols are preceeded by a dot, e.g.: .R0. Since Lisp functions evaluate arguments the quote mark (') is needed above to prohibit the evaluation of register symbols.) Direct addressing is denoted by a number as an element of a list, e.g. (1024). Indirect addressing is given as a list where the first element is the auxiliary register followed by the pre/post and increment/decrement mode symbol. The third element is the offset or index register and the fourth is for special cases like circular and bit-reverse addressing. Some familiarity with the Lisp language is needed to recognize the full flexibility and representational power of this modified assembly syntax.

The assembly language contains also functions and forms that correspond to directives in conventional assemblers. These include memory allocation from various types of memory blocks, the use of labels, etc. Object code can be generated for the TMS320C30 from ordinary instruction forms by using the `c30-obcode` Lisp function:

```
(c30-obcode  '(.LDF  -12.3  .R0))
```

This is the integrated assembler function that is used normally only by the system itself. The entire assembly process is fast due to the fact that it runs by calling precompiled Lisp functions. (Notice that no parsing of source code text is needed at assembly time.) Typically several thousands of machine instructions per second are assembled on a Symbolics Lisp machine and 500 - 2000 on the Apple Macintosh running Allegro Common Lisp.

Writing macro instructions

Easy manipulation of symbols and expressions is an inherent feature of the Lisp language. Due to this the programming environment shows its flexibility when writing new macro instructions and subroutines on the basis of the existing ones. The following examples demonstrate how to use the `def-c30-instruction` form to build new macro instructions.

Constructed on top of the original instructions there is a widely useful layer of predefined macros with generalized addressing modes. The idea is that such generalized instructions expand to one or more ordinary instructions according to the addressing modes that are needed while trying to minimize the number of instructions. (TMS320C30 has many kinds of limitations how the addressing modes can be combined.) As a simple example the macro `.ldf*` is defined to expand to the ordinary `.ldf` instruction (loads source `srcf` to destination `dstf` register) unless the source is equal to the destination. In such a special case no code is generated by the assembler and no operation is carried out by the simulator.

```
(def-c30-instruction .ldf* (srcf dstf)
   (unless (eq srcf dstf) (.ldf srcf dstf)))
```

The generalized instruction `.stf*` stores the source `srcf` register to any destination `dstf` having three different cases:

```
(def-c30-instruction .stf* (srcf dstf)
  (cond ((eq dstf srcf) nil)
        ((c30-register-p dstf 7)
         (.ldf srcf dstf))
        (t (.stf srcf dstf))))
```

The first condition is selected when the source register and destination are equal: no code is generated. In the second case the destination is another register (.R0 - .R7) and a `.ldf` instruction is needed for the transfer. The last alternative covers all other cases where the store instruction `.stf` is used.

As a further example of writing assembly level macros the `.limitf*` instruction is defined to limit a floating-point number `srcf` between `-limitf` and `+limitf` with generalized addressing modes for `srcf`, `limitf` and destination `dstf`.

```
(def-c30-instruction .limitf*
    (srcf limitf dstf &key (regx '.r0) (regy '.r1))
  (.ldf* srcf regx)
  (.cmpf limitf regx)
  (.ldfcond '.gt limitf regx)
  (cond ((numberp limitf)
         (.seq (.cmpf (- limitf) regx)
               (.ldfcond '.lt (- limitf) regx)))
        (t (.seq (.negf limitf regy)
                 (.cmpf regy regx)
                 (.ldfcond '.lt regy regx))))
  (.stf* regx dstf))
```

The source is first loaded by `.ldf*` to register `regx` that is defined by an optional keyword argument. If no specification of `regx` is given register .R0 is used. This is then compared with `limitf` that may have any addressing mode. If the limit is exceeded the limit value is loaded by `.ldfcond` to `regx`. Next the negative limit is checked. If `limitf` is a number, a negative immediate mode comparison and conditional load are carried out. If `limitf` is not immediate then its negative value is loaded to `regy` (default .R1), compared with `regx` and a conditional load is done. Finally the result is saved by `.stf*` in all cases. (The special form `.seq` is used to return a sequence of instructions from a Lisp expression.)

To see how the assembler expands instances of macro instructions (by `c30-linearize`) the following case shows the result of a `.limitf*` form when the `srcf`, `limitf` and `dstf` are direct addressing references to global variables `.x1`, `.x2` and `.x3`, residing in absolute addresses 0, 1 and 2.

```
(c30-linearize
    (.limitf* (.direct '.x1)
              (.direct '.x2) (.direct '.x3)))
=>
((.LDF (0) .R0)
 (.CMPF (1) .R0)
 (.LDFCOND .GT (1) .R0)
 (.NEGF (1) .R1)
 (.CMPF .R1 .R0)
 (.LDFCOND .LT .R1 .R0)
 (.STF .R0 (2)))
```

SOURCE CODE SIMULATION

The Lisp-based assembly language for the TMS320C30 is very interactive. An instruction, e.g. `(.absf -12.3 '.r0)` can be typed in the Lisp listener and the resulting value of register .R0 may be asked for by the form `(.r0)` => 12.3. All registers have an access function named after the register. The value of a register may be assigned by the generalized variable assignment form of Lisp, e.g. by `(setf (.r5) (* 2.0 pi))` whereby register .R5 will have the value 2π. Functions `.read` and `.write` are available to examine the values of memory locations.

These examples are related to a feature that can be called an interactive source-code level simulator. Ordinary or macro instructions can be called for execution from the Lisp listener or from any Lisp program. This is an attractive feature for program development and debugging.

Source-code level simulation has, when compared to the more commonly used object-code level simulation of a processor, both advantages and drawbacks. The main advantage in our case is the high flexibility and efficiency of source-code simulation. Typically several thousands of machine instructions per second can be evaluated. Assembly language and Lisp code can also be mixed (see the QuickSig example below) so that only parts of the desired algorithm are simulated for C30 at a time.

The drawbacks of our approach are that some cases cannot be simulated on the source-code level and the simulation is not a full bit-by-bit equivalent to what the real processor does. Program flow instructions like branches and calls can be used only as part of a macro instruction. Interlocked operations, interrupts etc. cannot currently be simulated. Delayed branches are transformed to undelayed ones for simulation. The cache and most details of the pipeline are also excluded from the simulator and the floating-point computations follow the precision of Common Lisp rather than that of the TMS320C30.

Our decision to simulate only the source-code level was based on the observation that in most cases it is the functionality of the program in general that should be tested flexibly by simulation. If the details of computation are needed, e.g. the exact values of data, execution times, etc., a real processor with self-emulation (as a virtual processor) is a better alternative than an object-code level software simulator. In our system the idea of the generic processor makes this self-emulation compatible with the source-code simulation so that no object-code simulator is needed.

DSP LISP COMPILER FOR TMS320C30

All modern programming environments for digital signal processors should support one or more high-level source languages. Typically this is the C language [3], [4]. In our case the natural choice was Lisp. The implementation of a full Common Lisp compiler is, however, out of the scope of this study and the Lisp language with run-time type checking and dynamic memory management is even non-optimal for DSP purposes. Due to this our experimental Lisp compiler is limited and strongly numerically oriented.

The role of the Lisp compiler is to simplify the expressions given to it and to transform the Lisp definitions to assembly code with optimized addressing forms and ordering of execution. Speed is often the primary requirement in DSP programming and due to this fact our compiler prefers register variables instead of stack-based computation whenever possible. The compiler is user extendable both to define new Lisp-level functions or macros and to define new basic functions that consist of manually optimized assembly instructions.

QuickSig LEVEL COMPUTATION

QuickSig is an object-oriented high-level signal processing environment [1]. It is based on the notion of signals and signal processing systems as objects. QuickSig is written in Common Lisp with New Flavors object extension and runs on Symbolics Lisp machines.

Signals and related objects are represented in QuickSig so that the details of implementation are hidden in most cases from the user. This data abstraction mechanism allows for a systematic and high-level syntax for signal processing programs. The signals, for instance, are typically sample sequences where the sample index interval is an internal property of the object. The automatic management of these intervals is called span processing. The QuickSig environment contains several object classes, e.g. signal windowing objects and filters, that are based on the concept of signals as objects.

One of the main features in the QuickSig system is to retain the simple syntax of Lisp, like in scalar computation

`(+ 1 2) => 3,`

in the domain of signal processing. For signal objects we can define the function "add" to mean additive mixing of the signals, sample by sample,

`(add sig1 sig2) => sig3` (a new signal object),

whatever the internal representations of the signals `sig1` and `sig2` may be. The generic function `add` can be applied as well to combinations of scalars and signals.

QuickSig has been found to be a flexible DSP extension to Common Lisp. It is very useful for exploratory programming and in the study of new algorithms or signal processing principles. The testing of DSP programs by simulation can be carried out with real signal objects available in the QuickSig system and the results can be immediately displayed graphically. As an example of easy testing of the assembly-level macro instructions the effect of the `.limitf*` macro on a sine wave signal is shown in Fig. 2. First an interface function is defined to write to the C30, to call `.limitf*` and to read the result:

```
(defun limitf-call (x limit)
   (setf (.r0) x)
   (.limitf* '.r0 limit '.r0)
   (.r0))
```

The application of the limiting function to a sine-wave signal `sine-sig` and the graphical display of the original and the peak-clipped signals are invoked simply by the form:

```
(m-draw
  (list sine-sig
    (s-call #'limitf-call sine-sig 20.0))).
```

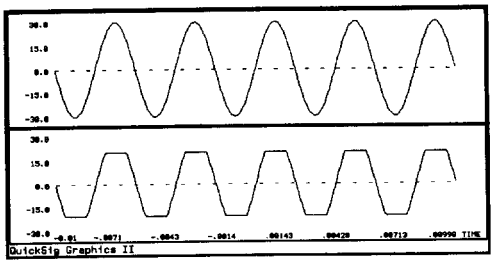

Fig. 2. Simulating and displaying the application of the macro instruction `.limitf*` to a sine wave signal.

In a typical C30 programming task a QuickSig simulation of the algorithm is first carried out. Then it is possible to develop the program for the processor incrementally by transforming high-level algorithms step by step to C30 Lisp or assembly code. The software simulation and testing with a real processor continues until the whole program runs on the DSP chip.

THE USE OF GENERIC PROCESSORS

The system can have two types of C30 processor objects, simulated and real ones. In both cases the processors have the same communication protocol and way of use (thus the word generic processor). Each real processor runs a virtual processor program that can be examined and called for execution just like the software simulated ones. Based on the fast and transparent assembly process and down-loading of object code the user can experiment with incremental and interactive program development on a real processor.

As a further step the C30 processor should have a more advanced real-time operating system supporting DSP multiprocessing, stream i/o, etc. The current system is a good starting point for this.

USER INTERFACE

The user interface consists of two main ways to communicate with the system. Textual interface based on the Lisp listener (interpreter) and editor & compiler is the most universal way of interaction and program development. This can be used on all levels of the system.

Another approach that is to be supported by modern software environments is a more graphically oriented one: the use of the mouse & menu control and graphics in multiple windows. We have experimented with this approach also in the domain of assembly-level programming, see Fig. 3. The processor object can be examined in a dialog window that shows the most important state parameters. Register values can be edited by clicking over the register position. The bits of the status register can also be manipulated separately.

The memory resources are also seen in the processor dialog. A memory examiner window can be opened to see the addresses, allocations and values of memory positions. (The allocation of e.g. variables from a memory block are registered in the memory block object that is a part of a memory. A memory object is further a part of a C30 processor instance in the sense of object-oriented programming). Such interaction with the processor instances makes it much more convenient to operate with complex objects than the textual way of interaction.

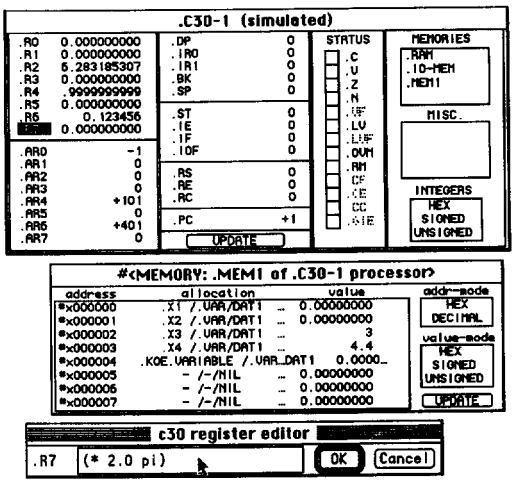

Fig. 3. An example of the user interface (Macintosh version) including a processor window, a memory examiner and a register editor.

The QuickSig environment also has graphical interface options. The signals, signal windowing objects, multidimensional signals including spectrograms etc. have their own presentations and layouts. In principle there could be any number of different presentations for each object in the system corresponding to various viewpoints and aspects to look at the objects.

An important subject of R&D in advanced programming environments is the methodology of graphical programming. We have experimented with a block diagram editor and compiler as a means of high-level specification of DSP programming [7]. Such a block diagram is at the same time a specification, a program and the documentation. The aim of the block diagram compiler is to transform the network-like description to executable instructions. Our programming environment is attractive for such experiments due to the advanced manipulation of symbolic expressions and support for several levels of program abstraction (assembly, Lisp and QuickSig levels).

ACKNOWLEDGEMENT

This study is a part of the research project "Symbolic Signal Processing" that is financed by the Academy of Finland.

REFERENCES

[1] *M. Karjalainen, T. Altosaar, P. Alku,* QuickSig - An Object-Oriented Signal Processing Environment. Proc. ICASSP-88 New York 1988.

[2] TMS320C30 Specifications. Texas Instruments Inc.

[3] *R. Simar, A. Davis,* The Application of High-Level Languages to Single-Chip Digital Signal Processors. Proc. of ICASSP-88, New York 1988.

[4] *J. Hartung, S. Gray, S. Haigh,* A Practical C Language Compiler/Optimizer for Real-Time Implementation on a Family of Floating Point DSPs. ICASSP-88, New York 1988.

[5] *G. Steele,* Common Lisp, The Language. Digital Press, 1984.

[6] *E.R. Tello,* Allegro CommonLISP. BYTE, Jan. 1988.

[7] *M. Karjalainen, S. Helle,* Block Diagram Compilation and Graphical Editing of DSP Algorithms in the QuickSig System. Proc. of IEEE ISCAS-88, Helsinki 1988.