BAD : A BASIC LANGUAGE COMPILER FOR THE TMS 320 C31 DSK

F. Auger

GE44-IUT de Saint Nazaire,
CRTT, Bd de l'Université, BP 406, F-44602 Saint Nazaire cedex, France
`auger@ge44.univ-nantes.fr`

ABSTRACT

This paper presents a new programming language for the TMS 320 C31, which provides a better access to the processor hardware than the C language, and an algebraic syntax of runtime expressions, unlike TI's assembly language. Several examples show the readability of signal processing algorithms implemented with this Basic-like language.

1. INTRODUCTION

The TMS 320 C31 floating-point DSP [11] is a versatile and powerful 32-bit signal processor. As the forthcoming C33, this component belongs to the third generation family of DSP's, which can bring low-cost industrial solutions for signal processing [7, 12], process control [2, 13] and instrumentation applications.

Of course, a necessary condition to meet this goal is the availability of accessible development systems, for an easy code generation. For the time being, only the assembly [14] and the C [15] programming languages are proposed by Texas Instruments. Both languages have well known advantages and drawbacks. Coding directly with the assembly language yields shorter and faster code, but requires a detailed knowledge of both the architecture and the instruction set of the DSP. Because of the non-orthogonality of the instruction set (all addressing modes are not allowed for all instructions), this may require several trial and error loops. Designing relevant applications in a reasonable amount of time is therefore possible for experienced users only, and is out of reach for many applications engineers and many undergraduate students. Choosing the C language is safer, requires less code rewriting when the algorithms are modified, and provides an easier readability of the code, at the expense of a larger and slower binary code. However, one may ask if the C language is a good choice when instructions such as [5]

```
volatile int *PBASE=(volatile int *) 0x808000 ;
```

are required just to make the `PBASE` variable point to the address `808000`h. More generally, the particular features that distinguish DSP's from conventional microprocessors do not seem to be easily used through the standard ANSI C [9]. Using interrupts in C is also possible, but certainly not simple [4].

This paper reports the design of a new compiler, which allows signal processing applications to be written more easily. This freely available compiler [3] should allow the use in realistic projects of the TMS320 C31 starter kit (DSK), in a digital signal processing laboratory [5], even at an undergraduate level. Some of its elements and ideas are borrowed from the Parallax Basic [8] and from the MATLAB programming language [16]. In section 2, we show that this compiler provides many possibilities of high level general purpose programming languages, and generates efficient assembly language source code. Section 3 shows how it can suit to the implementation of signal processing applications.

2. THE BAD FUNCTIONALITIES

The aim of this section is to show the simplicity of the proposed language, and its appropriateness to signal processing algorithms. The translation mechanisms of the compiler are also shown through several examples.

Every BAD source file must start by a program header which begins by the keyword `dspbasic`, followed by the target identifier `c31dsk`. For the time being, this is the only target identifier allowed, but future extensions of the compiler could be possible for other C31 boards and/or other DSP's. Comments (starting by a quote) are highly advisable in this program header.

After this prologue, constants and variables declarations may be found. As the internal registers of the C31, the BAD compiler provides only two scalar data types : 32 bits signed integers (`int`), and 32 bits floating-point real numbers (`float`). As in PBASIC [8], all variables must be declared before they are used, and are global (variables known only locally within subroutines do not exist). Constants and variables are simply defined by the keywords `const` and `var` respectively. Each variable is initialized to zero, unless the data type is followed by an optional initial value, as in the following source code example:

```
Const    length 10      ' constant length equals 10
Var      y  float 5      ' one float
Var      ~y1 float 17    ' another float
Var      j  int          ' an integer
```

The object code in assembly language produced by the compiler for these Basic instructions is:

```
length  .set    10      ; const length 10
y       .float  5       ; var  y   float 5
y1      .float  17      ; var ~y1 float 17
j       .word   0       ; var  j   int
_y      .word   y       ; address of y
_j      .word   j       ; address of j
```

The compiler is not case sensitive, and all upper case characters used in the source code are converted into lower case ones. It should be underlined that each Basic instruction is recalled in comments, so as to provide more intelligibility (these comments will be sometimes removed below, so as to respect the format of the proceedings). The assembly language variables `_y` and `_j` hold the addresses of the variables `y` and `j`, allowing their access by indirect addressing modes. These localization variables are not generated when the Basic variables are preceded by a tilde, as for `y1`.

Moreover, so as to use as much as possible the eight main registers of the DSP, the variables `x0`, `x1`, `x2`, `x3` and `i0`, `i1`, `i2`, `i3` are predefined real and integer variables. These variables are alias names of the internal registers `R4`, `R5`, `R6` and `R7`, which can be used to store either real or integer values.

binary operators allowed for both real and integer variables	
+	arithmetic addition
−	arithmetic subtraction
*	arithmetic multiplication
binary operators allowed for integer variables only	
&	logical and
|	logical nand
~	logical or
#	logical xor

Table 1: allowed binary operators.

assignment operators allowed for both real and integer variables		
ass. op.	example	
=	x0=2	$x0 = 2$
+=	x0+=2.5	$x0 = x0 + 2.5$
−=	x0-=2.1	$x0 = x0 - 2.1$
*=	x0*=5.0	$x0 = x0 * 5$
!=	x0!=y	$x0 = \mathrm{abs}(y)$
assignment operators allowed for integer variables only		
ass. op.	example	
&=	i &= 2	$i = i \text{ and } 2$
|=	i |= 2	$i = \overline{i \text{ and } 2}$
~=	i ~= 2	$i = i \text{ or } 2$
#=	i #= 5	$i = i \text{ xor } 2$

Table 2: allowed assignment operators.

The first four registers are used by the compiler as scratchpad registers. Using these variables is highly advisable for a shorter code length, since all variables defined by the user are placed in memory. But of course x0 and i0 can not be used at the same time, since they both refer to the same R4 register.

Defining arrays is also possible, either by writing the appropriate number of initial values, or by writing the size of the array between brackets, as in the following example:

```
var Primes        int        2,3,5,7,11
var SignalSegment float(75)
```

The first possibility may be preferred for small arrays, whereas the second one is suitable for large ones. The object code in assembly language produced by the compiler for these Basic instructions is:

```
primes  .word 2,3,5,7,11
signalsegment
        .loop 75
        .float 0
        .endloop

_primes .word primes
_signalsegment .word signalsegment
```

The first index value of an array is always 0. The 0th cell of an array can be referred by using just the array's name, without an index value. Arrays should be manipulated with care, because the compiler does not test the access to out-of-range locations, so as to give the programmer more freedom (and hence more misleading opportunities).

Both variables and constants can be used in runtime math or logical expressions, which can be of three kinds only :

$$var1 \text{ op= } value,$$
$$var1 \text{ op= } var2$$
$$var1 \text{ op= } var2 \text{ binop } var3$$

where *var1*, *var2* and *var3* are three variable names, binop is a binary operator, and op= is an assignment operator. The allowed binary operators and assignment operators for real and integer variables are listed in tables 1 and 2. The following basic instructions are examples of legal statements, in which the semicolons are statement separators, pulling two instructions written on the same line apart.

```
y=length ; y*=1.5 ; x0=6
j+=1; y(j)+=x0+y; x0!=y(1)
```

and the object code produced by the compiler for these instructions is:

```
        ldi        @_y, AR1      ; y=length
        ldf        length, R0
        stf        R0, *AR1

        ldi        @_y, AR1      ; y*=1.5
```

```
        ldf        1.5, R0
        mpyf3      R0,*AR1, R0
        stf        R0, *AR1

        ldf        6, R4         ; x0=6

        ldi        @_j, AR1      ; j+=1
        ldi        1, R0
        addi3      R0, *AR1, R0
        sti        R0, *AR1

        ldf        R4, R0        ; y(j)+=x0+y
        ldi        @_y, AR1
        ldf        *AR1, R1
        addf3      R1, R0, R0
        ldi        @_j, AR2
        ldi        *AR2, IR0
        addf3      R0, *+AR1(IR0), R0
        stf        R0, *+AR1(IR0)

        ldi        @_y, AR1      ; x0!=y(1)
        ldi        1, IR0
        ldf        *+AR1(IR0), R0
        absf       R0, R0
        ldf        R0, R4
```

The conciseness of the translation of x0=6 is evidenced when compared to the one for y=length. The compiler has optimized the translation of y(j)+=x0+y, detecting that both y(j) and y belong to the same array.

This reduced number of runtime expression structures may seem very restrictive, but this limitation can be justified by several arguments :

- The use of immediate values is limited to the first statement only, because the C31's architecture allows immediate addressing only with 16 bits integer and floating point numbers. Such instructions must therefore be used with care, and variables used as much as possible.

- Integer or float divisions are not allowed, because the C31 DSP does not have these operations in its instructions set. They are implemented however by special subroutines.

- The limitation of runtime expressions to simple binary operators forces the user to expand more complicated expressions into simple ones, which are nearer to the DSP instruction set. From an educational point of view, this can show more clearly what a DSP can do. This expansion can also give a better view on the complexity of an algorithm.

- Since there is only one binary operator per instruction, a precedence rule does not need to be defined, which reduces the potential for misinterpretation.

- As is, the legal statements allowed by the compiler are very close to the algebraic syntax of the instructions of the assembly language for the DSP's designed by Analog Devices, such as the AD21020 [1].

- Since the instructions of the source program are simple, the object code produced by the compiler can be easily understood, allowing a peephole optimization.

It should be also underlined that both the value *and the address* of user defined variables can be modified by runtime expressions, thanks to the @ operator, as in the example below :

```
@ScannedValue=@RealArray
ScannedValue += 5.6 ; @ScannedValue+=1
```

The object code produced by the compiler for these instructions can be found in [3]. This possibility should allow dynamic memory allocation mechanisms, and as much flexibility as available with pointers in the C language.

Flow control statements are possible thanks to labels, which must start by a colon. The existence of labels allows :

- unconditional jumps, with the `goto` instruction. As an example, the following Basic instructions,

```
:LoopStart
 x0 += x1*x2 ; goto LoopStart
```

will be translated into the following object code:

```
loopstart         ; :loopstart
        ldf       R5, R0    ; x0 += x1*x2
        ldf       R6, R1
        mpyf3     R1, R0, R0
        addf3     R0, R4, R0
        ldf       R0, R4

        br        loopstart ; goto loopstart
```

- conditional statements, which, as in PBASIC [8], are implemented by the following syntax:

```
        if condition thengoto label
```

For example, the instruction

```
if x0>y thengoto LoopStart
```

will be translated into

```
        ldf       R4, R0
        ldi       @_y, AR1
        ldf       *AR1, R1
        cmpf      R1, R0
        bgt       loopstart
```

- subroutine calls, with the `gosub` instruction. A subroutine must start by a label definition, and ends by a `return`. So as to take advantage of a possibility of the C31 instruction set, subroutines may be conditionally aborted thanks to the `returnif condition` structure. Arguments and returned data are gathered in an array, whose name *must* start by the subroutine name, followed by `srda` (for *subroutine data exchange*). It is advised to put the arguments first, followed by the outputs. Inside itself, a subroutine accesses to its `srda` by the predefined real and integer arrays `fsrda` and `isrda`, as in the following example,

```
Var MySubroutineSrda float 0, 0
...
gosub MySubroutine
...
:MySubroutine
 fsrda(1)=fsrda(0)+x0;
 return
```

which will be translated into:

```
mysubroutinesrda    .float 0, 0
_mysubroutinesrda   .word  mysubroutinesrda
...
        push      ar0 ; gosub mysubroutine
        ldi       @_mysubroutinesrda, ar0
        call      mysubroutine
        pop       ar0
...
mysubroutine        ; :mysubroutine
        ; fsrda(1)=fsrda(0)+x0
        ldf       *AR0, R0
        ldf       R4, R1
        addf3     R1, R0, R0
        ldi       1, IR0
        stf       R0, *+AR0(IR0)

        rets      ; return
```

- simple access to interrupts, thanks to predefined labels, whose names are taken from the manufacturer's literature (see [11], p 7-26). Interrupts start by a predefined label, and end by a `reti` instruction. Conditional ends of interrupt are also allowed by the `retiif condition` structure. As an example, the following instructions

```
:xint0
 x0+=2.5 ; reti
```

will be translated into

```
; this program uses the xint0 interruption
        .start "xint0sect", 0x809FC5
        .sect  "xint0sect"
        br xint0

        .text
xint0                       ; :xint0
        ldf    2.5, R0      ; x0+=2.5
        addf3 R0, R4, R4

        reti                ; reti
```

Moreover, data-driven flow control is also possible by the `goto var1` and `gosub var2` statements. In the first case, *var1* must be an integer holding the address of a label. In the second statement, *var2* must be an array of two integers. The first one holds the address of the subroutine, and the second one the address of the "srda". The labels held in *var1* and *var2* must be defined by two successive colons, as in the example below:

```
Var mySub2Srda float 0, 0
Var RightLabel int 0
Var RightSub   int 0, 0
...
::GoodPlace
 RightLabel = @GoodPlace
 RightSub(0) = @MySub2 ; RightSub(1) = @MySub2Srda
 gosub RightSub
...
 goto RightLabel
...
::MySub2
 fsrda(1)=fsrda(0)*x0 ; return
```

The object code produced by the compiler for these instructions can be found in [3]. Such possibilities, which do not exist in the C language, can efficiently replace multiple choice conditional statements such as the `switch` structure in MATLAB 5 [16].

Finally, assembly language instructions may be included, preceded by the `asm` keyword. This can make the access to the hardware features of the DSP easier. Some parts of code where the DSP spends most of its time may also be advantageously written in assembly language.

3. DESIGNING SIGNAL PROCESSING APPLICATIONS WITH BAD

For a fast and easy design of signal processing applications, an engineer needs not only an easy-to-use programming language, but also a library of the most usual transcendant functions (sin, cos, log, exp ...) and of the most frequently used signal processing tools (FIR and IIR filters, FFT, DCT, ...). Most of the transcendant functions are currently available, whereas only some of the signal processing building blocks are already implemented. But we hope that more and more will be available thanks to cooperative work. Designing a signal processing application will therefore only require to code the specific features, integrating safe general purpose functions written by experts.

As an example of the use of the proposed language for the implementation of signal processing algorithms, the main core of a tracking Kalman filter [10] is presented below. The complete program, together with more explanations and the corresponding object code, can be found in [3], from where other examples can also be retrieved.

```
dspbasic c31dsk
' tracking kalman filter, F. Auger, nov 1999

' elements of the covariance matrices
var p11pred float ; var p11est float 10
var p12pred float ; var p12est float  0
var p22pred float ; var p22est float 10
' Kalman gains
var k1       float ; var k2       float

var quarterqr float ' 0.25* q/r,
var error  float    ' error=y-H*x

var FinvSrda  float 0,0        ' srda of finv
var FinvMask  int   0xFF7FFFFF ' mask used by finv

var SETSP     int   0E970300h ' serial line config
var AICSETSRDA int  1,5,0x1f,0x99 ' srda of aicset
var twaitsrda int   0         ' useless srda
var AICIOSRDA int   0,0       ' DAC ADC

:InitInterrupt
 ' scf=625 kHz, Fs=20161 Hz, full scale
 gosub aicset ' init of the AIC in interrupt mode
:MainLoop
 ' prediction step :
 ' x1[k+1|k]=x1[k|k]+x2[k|k] and x2[k+1|k]=x2[k|k]
 ' x1[k|k] and x1[k+1|k] are stored in x1
x1+=x2 ' x2[k|k] is stored in x2 and not modified

' computing the covariance matrix of the
' predicted state ; x0 is used as scratchpad
' p11[k+1|k]=p11[k|k]+2*p12[k|k]+p22[k|k]+0.25*q
p11pred=p11est+p22est; p11pred+=p12est+p12est
x0=quarterqr ; p11pred+=x0

' p12[k+1|k]=p12[k|k]+p22[k|k]+0.5*q
x0*=2; p12pred=p12est+p22est; p12pred+=x0

x0*=2; p22pred=p22est+x0 ' p22[k+1|k]=p22[k|k]+q

' K[k+1]=P[k+1|k]*H'/(H*P[k+1|k]*H'+r)
' first compute x0=1/(p11pred+r)
FinvSrda(0)=1; FinvSrda(0)+=p11pred; gosub Finv
x0=FinvSrda(1); k1=p11pred*x0; k2=p12pred*x0

aiciosrda(0)=x1   ' predicted position as output
idle                   ' wait for an interrupt
error=aiciosrda(1)-x0;   ' correction step
```

```
' xi[k+1|k+1]=xi[k+1|k]+ki[k+1]*e[k+1], i=1,2
x1+=k1*error ; x2+=k2*error

' covariance matrix of the estimated state
x3=p12pred; x3*=x3*x0; p22est=p22pred-x3
p12est=p12pred*x0 ; p11est=p11pred*x0

goto MainLoop

:xint0                 ' interrupt from the AIC
' call aicio, and return from interrupt
gosub aicio ; reti
```

4. CONCLUSION

The proposed Basic language compiler should avoid many students to be scared by the implementation of signal processing algorithms on DSP's [17]. When compared to the assembly language, it should also make generation, testing and maintenance of large programs cheaper and less tedious.

5. REFERENCES

[1] Analog Devices Inc, *"AD21020/10 User's Manual,"* 1995.

[2] I. Ahmed ed., *"Digital control applications with the TMS320 family,"* Texas Instruments Inc., 1995.

[3] http://crttsn.univ-nantes.fr/~auger/bad

[4] Leor Brenman, *"Setting up TMS320 DSP interrupts in C,"* Literature Number SPRA 036, 1995.

[5] R. Chassaing, *"Digital signal processing laboratory experiments using C and the TMS320 C31 DSK,"* Wiley, 1998.

[6] B.W. Kernighan, D.M. Ritchie, *"The C programming language, second edition,"* Prentice Hall, 1988.

[7] P. Papamichalis ed., *"Digital signal processing applications with the TMS320 family: theory, algorithms and implementations (Vol 3),"* Texas Instruments Inc., 1990.

[8] Parallax Inc., *"Basic Stamp programming manual V 1.9,"* available from http://www.parallaxinc.com.

[9] A. Schwarte, H. Hanselmann, *"The programming language DSPL,"* PCIM, june 25-28, 1990. Also published in [2], pp 171–182.

[10] J. Tan, N. Kyriakopoulos, *"Implementation of a tracking Kalman filter on a digital signal processor,"* IEEE Trans. on Ind. Electronics, Vol 35, No 1, feb 1988. Also published in [2], pp 399–407.

[11] Texas Instruments Inc., *"The TMS320 C3x User's guide,"* Literature Number SPRU 031E, july 1997.

[12] Texas Instruments Inc., *"The TMS320 C3x general purpose applications user's guide,"* Literature Number SPRU 194, jan 1998.

[13] Texas Instruments Inc., *"The TMS320 C31x embedded control technical brief,"* Literature Number SPRU 083, feb 1998.

[14] Texas Instruments Inc., *"The TMS320 floating-point DSP assembly language tools user's guide,"* Literature Number SPRU 035C, feb 1998.

[15] Texas Instruments Inc., *"The TMS320 floating-point DSP optimizing C compiler user's guide,"* Literature Number SPRU 034G, march 1997.

[16] The MathWorks Inc., *"MATLAB functions reference guide (version 5),"* jan 1998.

[17] C.H.G. Wright, T.B. Welch, W.J. Gomes III, *"Teaching DSP concepts using Matlab and the TMS320C31 DSK"* Proc. IEEE ICASSP, 1999.