

2. EXISTING DSP SPECIFICATION ENVIRONMENTS

The C language is casually used by DSP designers because it is a world wide accepted standard for all kind of application domains and because compilers are available, but not because it really helps them in the specification phase. When studying the example of fig.1, one can easily derive constraints induced by C both on the specification and on the implementation. Since C is a procedural language, the sequential ordering of the statements together with the multiple assignment capability make the implementation on parallel DSP processors, very inefficient. The delay operator must be made implicit through difficult scheduling which make it impossible for the C compiler to use special delay instructions. C does not provide fixed-point types (most DSP processors work in fixed-point) nor means to specify truncation and overflow characteristics that are used on most DSP implementations. This means that the designer must write and try to optimize his specification according to a specific implementation and by doing so he wastes time and loses the advantages of a general programming language. Finally, code optimization is very hard.

A variety of DSP specification systems [3], [4], [6] have been proposed in the past and some of them are commercially available. Most of those systems also use the data-flow concept and rely heavily on schematics and predefined macro-cells. Modern day signal processing is however far more elaborated and requires non-linear operations, conditionals, iteration and recursion and a precise manipulation of the data representation. Most of those features (inconsistent with the overall data flow semantics) are only poorly handled in the existing systems. They are organized around two non unified levels of abstractions: the interconnection level and the macro block level. The introduction of new functions requires the user to modify or create macro blocks (in Lisp, C or assembler) which can be difficult and time consuming.

3. SILAGE

The language SILAGE has been developed in Berkeley by P. Hilfinger in 1984 [2] to specifically meet the expectations of the DSP designers. SILAGE is an applicative language, designed to capture the inherent parallelism of DSP algorithms in a linear textual form. The language has proven its suitability as a signal processing specification language and has been used as the input language for a number of design synthesis and code generation systems at different universities and within the industry [1], [5].

However, experiences with the language revealed some of its deficiencies when describing complex operations such as matrix manipulations and multiple rate asynchronous systems. A number of extensions to the language have been made, which will, in our opinion, cope with the mentioned problems.

A one to one mapping between the signal flow graph (fig.1-2) and the Silage description can be observed. This example also demonstrates some other important features of the Silage language: applicative language, timing information and time domain operations, data typing, hierarchical description, pragmatic directives.

Silage Description

```
#define word fix<20,8>
#define coef fix<10,4>

func main (in: word) out: word =
begin
  b@1 = 0;
  (a,b) = Adaptor (In, c, 0.1);
  (c,d) = Adaptor (b@1, d@1, 0.001);
  (e,f) = Adaptor (In, f@1, 0.01);
  out = word ((e - a) * 0.5);
end;

func Adaptor (In1, In2: fix; gamma: coef) Out1, Out2: fix =
begin
  Out1 = State + In2;
  State = (In2 - In1) * gamma;
  Out2 = State + In1;
end;

pragma (1 Alu, 1 Mult)
```

figure 2: Silage code corresponding to fig.1.

3.1 Applicative Language

A natural textual representation for data-flow semantics is an applicative language: a language whose fundamental operation is function application, and that has no variables or assignment operator. A program consists of an unordered set of definition of functions and signals. The ordering or concurrency of operations can be determined by the compiler and the amount of parallelism is only limited by data dependencies. This allows to derive several different implementations from a single specification of an algorithm.

Very simple procedural programs can be transformed into single assignment form by renaming. This is not the case anymore for algorithms containing nested iterations, which often occurs in DSP algorithms.

3.2 Timing Information and Time Domain Operations

Instead of deriving the timing of the operations from the ordering of the input description, it can be deduced from the signal dependencies (single assignment) and the presence of the delay operators (@), see fig. 2. A delayed signal acts as a leaf node to the scheduling process. It is interesting to note that replacing the "@" symbol by "z⁻¹" results directly in the Z-domain description of the filter (which is the favored representation of DSP-system engineers).

Multi-rate functions are also available in Silage. The *interpolate* function takes as input *N* signals with equal sample rates, and merges them into a new signal with a sample rate which is *N* times higher than the original. The *decimate* function does the opposite. The *switch* function has been introduced as a generalization of the previous two, it transforms *N* input signals into *M* output signals. The sampling rate of the output is *N/M* times the sampling rate of the input signals.

3.3 Data typing

All signals in the examples are of the fixed point type: `fix<20,8>` means word-length of 20 bits, position of decimal point is 8. Integer, floating point and array types are also available in Silage.

Operations have default types. Data types are determined by deduction (and induction) from the input and output data types.

```
fix<w1,d1> * fix<w2,d2> -> fix<w1+w2, d1+d2> (*)
```

No declaration is thus needed for an intermediate signal. Nevertheless coercion can be used to enforce a data type on a certain signal (as is done for the "Out" signal in the main function):

```
c = fix<16,15> (a * b)
```

In the new version of the language the user is able to select or write different rules that derive default types for standard operations (same syntax as (*)).

New deduction rules for deriving default types are not always sufficient. Generic types have therefore been introduced, which allow for the definition of format free functions (the input and output signals for the Adaptor function for instance have no defined type). "Compile-time functions" can be used to combine and define new types:

```
a = fix<width(c/d),12> (x*y);  width of a is the one
of c/d, position of decimal point is 12, value is x*y
```

This makes it possible to provide functional libraries without committing to a defined data representation. Reusability is an important aspect of the language.

The rigorous data type definitions make it possible to model exactly the effects of truncation, rounding and saturation. Those effects are of crucial importance in signal processing, where the quality of an algorithm is measured in terms of the signal to noise distortion ratios. Because these types and characteristics are built-in, it is easy for a tool to identify them and to use specialized DSP knowledge in order to handle them efficiently. Basic operators can be redefined by system or user-defined functions within well-defined scopes. Redefinition makes algorithms easy to read, to write and to port:

```
oper + (x,y) = add(x, y, zero_saturate+rounded);
```

3.4 Functional Language

Functions are used to hierarchically describe an algorithm (fig 2.). Each function definition consists of a number of input parameters, output parameters and a body with local signal definition. A function call may be replaced by in-line code, in which case special care must be taken with delays on input/output parameters. Silage also provides the user with some predefined functions: absolute value, bit extraction, bit merging, integration, differentiation,.. vector operations like component-sum, sum, maximum, dot- and cross-product, vector delay, tapped-delay line vector,....Here follows the definition of an FIR filter:

```
out = ComponentSum ( DotProduct (TappedDelayLine(x) , coef [] ) )
```

3.5 Pragmatic Directives

A `pragma` is a compiler directive that supplies non-algorithmic information about a program. This information is useful for some compilers. In the above example, the `pragma` is used to direct the silicon compiler to map the algorithm on an architecture consisting of one ALU and one multiplier.

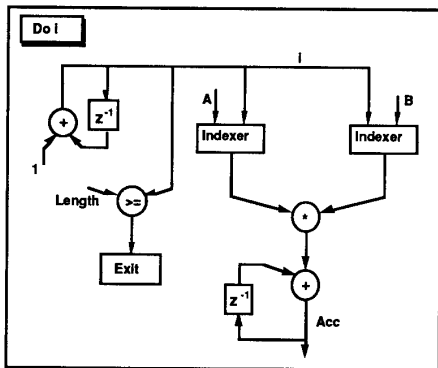
3.6 Loops and Conditionals

The above example only shows a few characteristics of Silage. Silage also provides powerful iterator constructs like definite and indefinite iterations, conditional expressions, logical

expressions,...

A loop in Silage is applicative, which means that any statement within the loop is still considered as a definition (not an assignment). Since the redefinition of signal is forbidden, each signal appearing in a loop must be indexed. This allows the compiler to implement a loop in any order, to expand it, to perform loop optimization such as loop migration. This has the disadvantage of forcing the user to introduce vectors which do not correspond to real vectors. This constraint has been removed by extending the use of the delay operator to loops. In that case the delay operator refers to the previous value of a signal or a variable (fig. 3). This allows the user to elegantly specify loops in an applicative way. A definite loop must always be expandable (in software or in hardware), therefore the loop index must be manifest (known at compiled time).

New "control expressions" have been introduced to superimpose a macro control flow on top of the data flow semantics. Examples of such expressions are the indefinite loop (fig. 3) and the if-then-else control construct. The initial delay concept has been extended in order to define multiple "time levels". In this case the delay operator refers to the previous value of a signal or a variable within its time level.



```

do    i@@1 = 0;           initialize
      Acc@@1 = 0;
      i = i@1 + 1;       increment index
      Acc = Acc@1 + A[i]*B[i];
      out = exit (i >= Length) -> Acc;
od;

```

fig. 3 Indefinite loop and corresponding schematics

When performing non-manifest array addressing (which corresponds to pointer arithmetic), the designer can explicitly declare the bounds on the array sizes and the queues with the aid of manifestly bounded expressions. This results in far more efficient solutions than currently possible within applicative frameworks.

4. SILAGE ENVIRONMENT

A number of extremely efficient tools have already been built around SILAGE [1], [5]: compilers, translators to C and VHDL, system and bit-true simulators, bit-serial and bit-parallel silicon compilers, optimized code generators for commercially available DSP processors and code generator for multi-processor systems.

5. CONCLUSIONS

The language Silage has been presented together with a number of extensions to the original design which result in a powerful specification language and design environment of complex DSP systems. This has been very clearly demonstrated by the coding of many DSP specifications including a complete compact disk with error correction, speech and image recognition and coding (edge detection, ray tracing), adaptive filters for telecommunication, complex interpolation algorithms for digital audio, autocorrelation matrices and inversion,...

REFERENCES

- [1] D.R. Genin, J. De Moortel, D. Desmet, E. Van de Velde, "System Design, Optimization and Intelligent Code Generation for Standard DSP", ISCAS 89, Portland.
- [2] P. Hilfinger, "A High-Level Language and Silicon Compiler for DSP", Proceedings of the Custom Integrated Circuits Conference, May 1985.
- [3] G.E. Kopec, "The Signal Representation Language", IEEE Trans. on ASSP, ASSP-33(4), August 1985.
- [4] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow", IEEE Proceedings, September 1987.
- [5] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, F. Catthoor: "Cathedral II : A Synthesis System for Multiprocessor DSP Systems", in "Silicon Compilation", Ed. Gajski, Addison Wesley 1988.
- [6] K.S. Shanmugan, G.J. Minden, E. Komp, T.C. Manning and E.R. Wiswel, "Block Oriented System Simulator (BOSS)", Telecom. Lab, Univ. of Kansas, Inter. Memorandum, 1987.