 UNLIMITED

(2)

RSRE
MEMORANDUM No. 4385

ROYAL SIGNALS & RADAR ESTABLISHMENT

SILAGE TO ELLA TRANSLATION

Authors: M G Hill & E V Whiting

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

RSRE MEMORANDUM No. 4385

UNLIMITED

CONDITIONS OF RELEASE

0074741 BR 114304

..... DRIC U

..... DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Royal Signals and Radar Establishment

Memorandum 4385

Title: SILAGE TO ELLA TRANSLATION

Authors: M G Hill and E V Whiting

Date: April 1990

Summary

A translation of Silage circuits into functionally equivalent ELLA circuits is considered. A series of ELLA macros which could form a library of translator functions is outlined.

Contents

1 Introduction

Silage is an applicative behavioural specification language especially suited for DSP applications. It originated at Berkeley [1], but the versions being considered in this document come from Philips (Eindhoven) [3] and IMEC [2]. There are significant differences between the Philips and IMEC versions. Both versions of Silage are used as input languages to suites of silicon compilers.

A study of the translation of Silage circuits to ELLA^{TM} is carried out. Sections 3 - 10 consider constructs of Silage common to both Philips and IMEC's Silage whilst section 11 considers those constructs only available in the IMEC version. The document outlines a series of ELLA macro functions which could be combined into a library for use in the translation of Silage circuits. It should be noted that although a Silage to ELLA translation is possible the resulting ELLA circuit is not necessarily the one that would have been chosen if the circuit had been originally described in ELLA. This work has been supported by the ESPRIT project 'SPRITE', which is developing a European Synthesis System with ELLA as one of its front-end hardware design languages.

2 ELLA

The ELLA system is an integrated hardware design tool-set, which comprises the ELLA language compiler, the ELLA Applications Support Environment (EASE), the ELLA simulator, and the ELLANET procedural interface [4-7]. The ELLA language can be used to describe hardware at all stages in the VLSI design cycle, from the earliest architectural concepts to the full implementation at gate level. The ELLA system was originally developed at the Royal Signals and Radar Establishment in Malvern, UK. and is now being enhanced in collaboration with Praxis Electronic Design of Bath. ELLA is now the de-facto standard high-level VLSI design language in the UK. The language constructs described in this document are available in ELLA Version 4.

3 Silage Signals

Every Silage program has associated with it a functional behaviour where signals denote infinite streams of samples. Each signal has a basic type in the same way that ELLA gives types to values, and this type must be explicitly known. Silage programs relate signals through a series of relations with other signals. The semantics of Silage can be thought of as data-flow [1], where a program represents a set of paths through which data flows, rather than a sequence of operations performed on memory locations i.e. control-flow.

4 Types

In Silage there are two basic types:

```
NUM <w,d>
BOOL <w>
```


where w and d are integers. In IMEC Silage these are called FIX<w,d> and INT<w> respectively, and INT< 1 > is given the special name 'BOOL'. If a signal is of type NUM<w,d> then it represents a fixed point number in 2's complement notation of length w, with the last d bits taken to represent those values after the decimal point and the most significant bit being the sign bit. In ELLA there are several ways in which such signal types could be described, only four possible formats will be considered here. Two formats involve the use of ELLA integers. first

```
TYPE eight = e/(2r00000000 .. 2r11111111),
     four  = f/(2r0000 .. 2r1111),
     inum_12_4 = (eight, four).
```

with a constant of type 'inum_12_4' defined by

```
CONST c_1 = (e/2r00000000, f/2r1100).
```

and second

```
TYPE first = f/(-128 .. 128),
     second = s/(0 .. 31),
     inum = (first, second).
```

with a constant of type 'inum' defined by

```
CONST c_2 = (f/0, s/8).
```

In both cases the constant represents the decimal value 0.8 expressed as an ELLA type which is a structure of two ELLA integers. In the case of constant 'c_1' the expression f/2r1100 is to be taken as representing a fraction and not the integer value '12'. For c_2 it appears that 0.8 is represented exactly, however 0.8 cannot be expressed exactly in NUM< 12, 4 > format. Thus with either approach great care is needed when operating on such types. Hence it would be necessary to convert such constants to bit representations before any calculations could be performed in order for the translated Silage program to give identical results to the original Silage program. This observation leads to the consideration of using ELLA characters since automatic translation to bit strings is provided by a series of built-in operators.

Consider

```
INT w = ....
TYPE bit = NEW b('0 | '1),
     num_w = STRING [w] bit.
```

where 'bit' is a user defined ELLA type which can take the value b'0 or b'1, and 'num_w' is a row of 'w' bits. In order for 'num_w' to represent a type such as NUM< w,d > it is necessary to adopt the convention that the last d 'bits' represent the fractional part. For example

```
NUM<12,4>
```

could be represented by

```
num_12 = STRING [12] bit
```

and the constant 'one' would be defined as

```
CONST one = b"000000010000".
```

However if the convention NUM< 12, 3 > were to be used then 'one' would still represent a valid quantity, but its value would be 2.0.

To avoid misinterpretation of such constants leads to the fourth approach, which is to define signal type NUM< 12, 4 > in the form

```
TYPE eight_bit = STRING [8] bit,
     four_bit  = STRING [4] bit,
     num_12_4  = (eight_bit, four_bit).
```

In which case 'one' would be defined as

```
CONST one = (b"00000001", b"0000").
```

The advantage of this approach is that strong type checking would always ensure that a constant of type 'num_12_4' could not have another representation. The advantage with the other character approach is that it provides a more compact denotation. In this document the last type definition will be used when converting Silage code into ELLA.

5 Arrays

In Silage arrays of signals can have any number of dimensions, however multiple 'makes' of functions are not allowed. In ELLA multiple 'makes' are possible plus multi-dimensional arrays of signals.

Note that Silage arrays start from zero but ELLA arrays start from one. Also the upper bound of a Silage array is one less than the size e.g. NUM< 12, 4 >[6]:c is an array of six elements indexed from 0 to 5.

Silage has a shorthand notation for combining array elements, for example if a,b,c are arrays of three elements then

```
a[] = b[] + c[]
```

is the same as

```
a[0] = b[0] + c[0]; a[1] = b[1] + c[1]; a[2] = b[2] + c[2];
```

or

```
a = b +c
```

or

```
(i: 0..2):: a[i] = b[i] + c[i]
```

whereas in ELLA the following is needed

```
[INT i = 1..3] a[i] := b[i] + c[i]
```

unless a new macro function ++, say, is defined which adds two vectors of arbitrary length, then it is possible to say

```
a := b ++ c
```

With the inclusion of function macro parameters in a future release of ELLA it will be possible to define an even more general macro, for example

```
MAC OP {FN GENERAL (TYPE ty,ty)->ty} = ([INT n]ty:in1 in2) -> [n]ty:
                                       [INT i = 1 .. n] in1[i] GENERAL in2[i].
```

When this macro is instantiated it is supplied with a function as a macro parameter, the input type of the function macro parameter defining the type of the inputs to the macro. The length of the vector of inputs is defined implicitly by the input variables. The advantage of such a general macro is that it can be used in a variety of situations, for example

```
a := b OP{+} c

res := gates1 OP{AND} gates2

bitres := bits1 OP{BITSMULT} bits2
```

6 Iterations

Iterations in Silage are of the form

```
(i : lwb .. upb ) :: a = b
```

where lwb, upb are integers and a, b expressions. Since Silage is an applicative language it has **single assignments** thus definitions of the form

```
(i : 1 ..10 ) :: a = a + 1;
```

are illegal. In such cases arrays should be used i.e.

```
(i : 1 ..10) :: a[i] = a[i-1] + 1;
```

In ELLA such expressions are of the form

```
[INT i = 2 .. 11] a[i] := a[i-1] + one;
```

for iterations in an ELLA sequence clause, and

```
FOR INT i = 2 .. 11 JOIN a[i-1] + one -> a[i].
```

for the functional form, where 'one' is an appropriate ELLA integer.

7 Constants

Constants in Silage are expressed as

```
(0.8 : NUM<7,6>)
```

This is a constant 0.8 written in a seven bit format with the last six bits representing the number and the remaining bit being the sign bit. To describe such a constant in ELLA would require a format like

```
NUM{7,6}r"0.8"
```

where NUM is an ELLA macro whose specification is

```
MAC NUM {INT m n} = (STRING [INT size] real:in)
                        -> (STRING [m-n] bit, STRING [n] bit):
```

where

```
TYPE real = NEW r( '0..'9 | '. | '- ).
```

With this macro the value of 'size' is determined from the input value when the macro is instantiated, so, for example, a string such as r"0.8" would have a 'size' of 3. The macro is written so that strings of characters of arbitrary length are read in and a bit representation is output in NUM<m,n> format. ELLA has a set of built-in operators which would assist in writing the body of the macro, for example the conversion of r"0.8" to a floating point number. It should be noted that Silage has a default casting rule together with a library for rounding, saturation etc. [2], and these must be taken into account when translating.

In ELLA expressions of the form NUM{7,6}r"0.8" are value delivering and therefore illegal in certain parts of ELLA i.e. as a test condition in a CASE statement. In such places the constant will need to be expressed explicitly i.e.

```
(b"0", b"110011")
```

8 Delay Signals

Silage uses built-in delays of the form

```
name @ int
```

where the signal 'name' is delayed by 'int' time units with 'int' being a **manifest** integer expression, that is it must be known at compile time. This delay returns the value of 'name' at 'int' time units before the present value. In ELLA there exist multiple delays, but they deliberately lose information for signals shorter than the delay time. The way to do pure delays is to have

```
DEL{n} name
```

where use is made of the generic delay macro

```
MAC DEL {INT n} = (TYPE t:in) -> t:
IF n = 0
   THEN in
   ELSE ( FN D = ([n]t)->[n]t:DELAY([n]?t,1).
          MAKE D: delay.
          JOIN IF n = 1
                  THEN in
                  ELSE (in CONC delay[1..n-1])
               FI -> delay.
          OUTPUT delay[n]
        )
FI.
```

It should be noted that the macro has been written in the above form for efficiency and hence only requires one delay per instantiation. With this macro the value of the type 't' is deduced from the instantiation of the macro. The macro returns the input value unaltered if n=0 and returns a delayed signal otherwise. When n > 0 an ELLA delay is defined and instantiated, this delay has 'n' inputs and the JOIN statement joins 'in' to the first delay input and each output of the delay to the successive input. The last delay output being delivered as the result. The 'CONC' command is an ELLA command that takes two arguments and concatenates them. In this macro it takes signals of type 't' and '[n-1]t' and returns a signal of type '[n]t', which is then joined to 'delay'.

In certain cases the above delay macro can cause more unit delays to be created than necessary, for example consider the expression

```
in@1 + in@2 + in@3
```

then using the above macro would give the ELLA statement

```
DEL{1}in + DEL{2}in + DEL{3}in
```

Whilst this expression would give the correct result it does use three separate delays. In such situations an optimised delay macro could be used which takes the form

```
MAC OPDEL {INT n, TYPE t}  = (t:in) -> [n+1]t:
IF n=0 THEN [1]in
       ELSE (  FN DEL = ([n]t)->[n]t: DELAY([n]?t,1).
               MAKE DEL:del.
               JOIN IF n=1
                      THEN in
                      ELSE (in CONC del[1..n-1])
                    FI -> del.
               OUTPUT in CONC del
             )
FI.
```

then the above expression would become

```
MAKE OPDEL{3,...}: del.
JOIN in -> del.

del[2] + del[3] + del[4]
```

which would only use one delay. All the unit delays used in this section have been initialised to represent an unknown but legal value. Initialisation of delays with other values will be considered in section 11.1.

9 Coercion

Silage only operates on bitstrings and this often requires coercion in order for compatibility with the built-in Silage operators. An example of a constant, 0.8, written in NUM< 7,6 > format which is then coerced to NUM< 12,4 > format is

```
NUM<12,4> ( 0.8 : NUM<7,6> )
```

In ELLA the same procedure can be achieved by

```
COER{12,4} ( NUM{7,6} r"0.8" )
```

where

```
MAC COER {INT m n} = ((STRING [INT m1] bit, STRING [INT n1] bit):in)
                              -> (STRING [m-n] bit, STRING [n] bit):
( IF m1 >= (m-n)
    THEN in[1][m1+1-(m-n)..m1]
    ELSE STRING [m-n-m1]b'0 CONC in[1]
  FI,
  IF n1 >= n
    THEN in[2][1..n]
    ELSE in[2] CONC STRING [n-n1]b'0
  FI
).
```

Note that COER and NUM have different input types and are therefore not interchangeable.

An important difference between ELLA and Silage is that the coercions necessary with Silage are not needed in ELLA. This is due to the built-in operators of ELLA having implicit type parameters (i.e. 'TYPE t' in macro OPDEL of the previous section) and hence they can operate on different length strings. However the rounding truncating rules would be different to Silage and thus for exact correspondence between ELLA and Silage COER would be needed. For all other cases no such coercion functions would be required.

10 Functions

A function in Silage is composed of three parts namely

```
heading   declarations   definitions
```

where the heading corresponds to an ELLA function spec and definitions to the ELLA body. The declarations part corresponds to definitions of LET's, VAR's, MAKE's etc. which in ELLA can occur anywhere within the function body. In Silage there can be no nested functions and no recursion. ELLA allows nested functions but has recursion only through macros.

A typical Silage function declaration could be

```
FUNC main ( in: NUM<12,4>) : NUM<12,4> =
```

In ELLA this would be written as

```
FN MAIN = (num_12_4: in) -> num_12_4:
```

An example of a complete Silage function is

```
FUNC main (in: NUM<12,4>) out: NUM<12,4> =                    |# heading #
   NUM <12,4>[6]: c;                                          |
   NUM <12,4>[7]: sum;                                        |# declarations #
BEGIN                                                         |
     c[0] = ...; c[1] = ... ; ... ;                           |
     out = sum[0];                                            |# definitions #
     sum[6] = (0.0 : NUM<12,4>);                              |
     (k : 0.. 5) :: sum[k] = sum[k+1]                         |
                              +  NUM<12,4>(c[k] * in@k);      |
END;                                                          |
```

Using the type 'num_12_4' defined in section 4 such a function can be written in ELLA in several styles: all of which have the same hardware representation as the Silage

```
FN MAIN = (num_12_4: in) -> num_12_4:
( SEQ
    LET c = ..... ;
    VAR sum := [7]( ... );
    [INT k = 1 ..6]sum[7-k] := sum[8-k] + COER{12,4}(c[7-k] * DEL{6-k}in);
    OUTPUT sum[1]
).
```

where "-", "*" are ELLA functions. It can be noted that an extension to ELLA will allow iterations to have negative step lengths, thus the above iteration could then be written in a form syntactically closer to Silage. The function MAIN can also be written without the use of sequences, but with recursive macros i.e.

```
FN MAIN = (num_12_4:in) -> num_12_4:
( LET c = .... .
  MAC SUM {INT n} = (num_12_4:ip) -> num_12_4:
  IF n=1
     THEN COER{12,4} (c[1] * DEL{0}ip)
     ELSE SUM{n-1}ip + COER{12,4}(c[n]*DEL{n-1}ip)
  FI.
  OUTPUT SUM{6}in
).
```

To write the function without the use of recursive macros or sequences requires the definition of a new type of general macro as follows

```
MAC WIRE {INT m n} = ((STRING [m-n] bit, STRING [n] bit):in)
                          -> (STRING [m-n] bit, STRING [n] bit): in.
```

This then allows

```
FN MAIN = (num_12_4:in) -> num_12_4:
( MAKE [6]WIRE{12,4}: c,
       [7]WIRE{12,4}: sum.
  JOIN ... -> c[1], ... , ... ... , ... -> c[6].
  JOIN zero -> sum[7].
  FOR INT k = 1..6 JOIN sum[k+1] + COER{12,4}(c[k] * DEL{k-1}in) -> sum[k].
  OUTPUT sum[1]
).
```

By using the macro 'WIRE' the format of ELLA resembles a Silage program with the MAKE statements looking like Silage declarations.

Silage also has an "IF" construct which more easily maps onto an ELLA "CASE" statement, since an ELLA "IF" statement needs to know whether the boolean test is true or false at compile assemble time in order that the correct hardware can be selected, irrespective of what data passes through the circuit. Thus consider the following Silage function

```
FUNC carry (in : NUM<12,4>) : BOOL<1> =
   NUM<13,4> : sum;
BEGIN
     sum = IF return@1 -> (0: NUM<13,4>)
             || sum@1 + NUM<13,4>(in)
           FI;
     return = sum != NUM<13,4>(NUM<12,4>(sum));
END
```

The 'IF' statement reads : IF return@1 delivers true THEN zero ELSE sum@1 + in FI. In this example the output is not named and hence the default name 'return' is taken for the output in the same way that ELLA uses the word OUTPUT to define the function result. When converted into an ELLA description using sequences this example becomes

```
FN CARRY= (num_12_4: in) -> bit:
( SEQ
     PVAR sum ::= (STRING[9]b'0, STRING[4]b'0),
          return ::= b'0;
     sum := CASE return OF
               b'0 : NUM{13,4}r"0.0"
              ELSE   sum + COER{13,4} in
            ESAC;
     return := sum NEQ COER{13,4}(COER{12,4}sum);
```

```
   OUTPUT return
).
```

Where "+" and "NEQ" could be user defined or taken from the library of functions and built-in operators that are supplied with ELLA. Both 'sum' and 'return' are defined to be persistent variables, that is they retain their value from one time step to the next. There is an implicit delay associated with each variable and thus an explicit delay is not required on either variable. This example can also be written using the 'WIRE' macro in the following way

```
FN CARRY = (num_12_4:in) -> bit:                          | # heading #
( FN BOOL = (bit:in)->bit:in.                             |
  MAKE WIRE{13,4}: sum,                                   |  # declarations #
       BOOL: return.                                      |
  JOIN CASE DEL{1}return OF                               |
         b'0 : NUM{13,4}r"0.0"                            |
        ELSE   DEL{1}sum + COER{13,4} in                  | # definitions #
       ESAC -> sum.                                       |
  JOIN sum NEQ COER{13,4}(COER{12,4}sum) -> return.       |
  OUTPUT return                                           |
).
```

and the comparison with the Silage version can be clearly seen. In general the function 'BOOL' would be declared externally in a predefined library.

Note that the constant 0.0 in the second arm of the case clause could also have been written as

```
                    b'0 : (b"000000000", b"0000")
```

or

```
                    b'0 : (STRING [9] b'0, STRING [4] b'0)
```

which would not have required use of the macro "NUM".

11 IMEC Silage

The IMEC version of Silage has several differences from the Philips version. The different parts will be considered and their translation to ELLA shown when such a route is possible.

In the IMEC version signals can be declared in the body of the function e.g

```
FUNC adap (a,b : NUM<12,4>) c: NUM<12,4>; d: NUM<12,4> =
```

```
BEGIN
      c = z@1 + z@2;
      d = c - z@3;
      z = a -b;
END;
```

Note that IMEC use the notation FIX< 12,4 > rather than NUM< 12,4 > but for consistency of notation in this document NUM will be used. Also this version of Silage allows variables to be used before they are declared (i.e. 'z' in the above example). Re-writing this function in ELLA would give

```
FN ADAP = (num_12_4: a b) -> [2]num_12_4:
BEGIN
      MAKE WIRE{12,4}: c, d, z.
      JOIN DEL{1}z + DEL{2}z -> c,
           c - DEL{3}z -> d,
           a - b -> z.
      OUTPUT (c,d)
END.
```

where "-" and "," are ELLA functions.

With the SPRITE extension of ELLA output names will be allowed, and hence 'c' and 'd' would not need to be 'made'.

11.1 Initialisation of Delays

In this version of Silage Delays can be initialised by use of the "@@" operator. Thus the statement

```
name @@ num = constant
```

defines the value of 'name' at 'num' time units before the start. A Silage program written with such initialisation functions is

```
FUNC f (in: NUM<12,4>) : NUM<12,4> =
BEGIN
     state @@1 = 0;
     state @@2 = 1;
     state = in + state@1 - state@2;
     return = state + state@1;
END;
```

Since 'state' is stored as an infinite vector the initialisation states that if the start time is zero then at t = -1, state = 0 and at t = -2, state = 1. ELLA has no direct way of imposing two conditions on a pure delay. The only way that ELLA can get round this is to combine two delays in series, the first delay being initialised to 1 and the second to 0 with 'state' being input to the second delay. Then 'state@1' will be the output of the second delay and 'state@2' will be the output of the first delay. Thus the translated version of this function becomes

```
FN F = (num_12_4:in) -> num_12_4:
( FN MULTDELAY = (num_12_4:in) -> [3]num_12_4:
  (   FN DEL1 = (num_12_4) -> num_12_4: DELAY( (b"00000000",b"0000"), 1).
      FN DEL2 = (num_12_4) -> num_12_4: DELAY( (b"00000001",b"0000"), 1).
      MAKE DEL1: del1,
           DEL2: del2.
      JOIN in -> del1,
           del1 -> del2.
      OUTPUT (in, del1, del2)
  ).
  MAKE MULTDELAY: state.
  JOIN ((in + state[2]) - state[3]) -> state.
  OUTPUT state[1] + state[2]
).
```

For more complicated initialisations the function 'MULTDELAY' will become progressively more complex. In an automatic translation two passes of a Silage program would be needed in order to define 'MULTDELAY' since the number and position of all initialisation values would be required before the function could be constructed.

NOTE : With the inclusion of CONSTant macro parameters in the ELLA language it is possible to write

```
MAC MULTDELAY {CONST ([INT n]TYPE t): c}  = (t:in) -> [n+1]t:
IF n = 0
   THEN in
   ELSE (  FN DEL = ([n]t)->[n]t: DELAY(c,1).
           MAKE DEL:del.
           JOIN (in CONC del[1..n-1]) -> del.
           OUTPUT in CONC del
        )
FI.

FN F = (num_12_4:in) -> num_12_4:
(  CONST c1 = (STRING[8]b'0, STRING[4]b'0),
         c2 = (b"00000001", STRING[4]b'0).
   MAKE MULTDELAY{ (c1,c2) }: state.
```

```
   JOIN (in + state[2]) - state[3] -> state.
   OUTPUT state[1] + state[2]
).
```

The macro MULTDELAY can be compared with the macro OPDEL defined in section 8 for the case when the delay initialisation is set to 'unknown'.

It can also be noted that an alternative form for an initialised delay may be used if all occurrences of the delay can be written within an ELLA sequence clause i.e

```
PVAR history ::= (c1, c2, c3, c4, ?type);
[INT i = 2 ..5] history[7-i] := history[6-i];
history[1] := in;
```

where c1,c2,c3,c4 are the initialisation constants.

These last two forms can be used in such cases where, say, the first 100 values require initialisation, however the notation is far from concise.

11.2 While Loops

This allows for data dependent termination of a loop i.e.

```
a = 0; sum = 0;
WHILE (in>0) DO
BEGIN
      a = a + 1;
      sum = sum + in;
END
```

It can be noted that this facility is not available on the system which is used as input to a silicon compiler. If this 'While loop' is considered to be a loop in time then the corresponding ELLA can be written as

```
PVAR a ::= zero, sum ::= zero;
CASE TEST(in) OF
  b'0 : (a := a + one;
         sum := sum + in
        ),
  b'1 :
ESAC;
```

where TEST delivers b'0 whilst "in > 0" but when the test fails delivers b'1 for that time and all subsequent time. Writing such a loop in un-sequenced ELLA gives

```
MAKE [2]WIRE{ ... }: a_sum.
LET c = (one, in).
FOR INT i = 1 .. 2 JOIN CASE TEST(in) OF
                            b'0 : DEL{1} a_sum[i] + c[i],
                            b'1 : DEL{1} a_sum[i]
                         ESAC -> a_sum[i].
```

where 'a_sum' is a vector whose first element corresponds with 'a' and second element with 'sum'.

Silage also uses 'While loops' as an open ended iteration loop, for example

```
sum, a : NUM<12,4>[max];
...
WHILE ( i < in ) DO
BEGIN
      sum[i] = sum[i-1] + a[i];
      i = i + 1;
END;
```

In ELLA this would need to be handled by the following approach for sequences

```
INT lwb = .. , upb = .. , max = ...

TYPE integer = int/(lwb..upb).

VAR sum := [max]( ... ), a := [max]( ... );
...
[INT i = 2..max]
   CASE it/i LT in OF
       b'1 : sum[i] := sum[i-1] + a[i]
   ESAC;
```

and by the following for non-sequence ELLA

```
INT lwb = .. , upb = .. , max = ...

TYPE integer = int/(lwb..upb).
```

```
MAKE [max] WIRE {12,4} : sum, a.
...
FOR INT i = 2 .. max
   JOIN CASE it/i LT in OF
           b'1 : sum[i-1] + a[i]
         ELSE sum[i]
        ESAC -> sum[i].
```

There is no construct in ELLA which allows for infinite loops. In ELLA the user must define bounded loops which terminate with a choosen value 'max'. This approach should be used with caution since 'max - 1' instantiations of 'LT' and '+' will be made irrespective of the size of 'in'.

The functions 'LT' and '+' are built-in operators that compare ELLA integers and add two bit strings, repectively. The CASE clause is used to decide whether to update 'sum' or not.

11.3 Interpolate, Decimate, Cut and Switch

These procedures allow functions to be partitioned into different processes with possibly different sample rates. Three different approaches can be used. First Interpolate, this takes N signals at a sample rate U ,say, and puts them together to form a single signal with a sample rate of U*N, decimate reverses this process. The final approach "cut" leaves the signals alone but divides up the program onto different processes which will run concurrently.

The 'switch' feature provides a practical shorthand for combining interpolate and decimate, for example

```
temp  = interpolate (a,b,c);
(x,y) = decimate (temp);
```

can be re-written as

```
(x,y) = switch (a,b,c)
```

such a construct being of use in the description of a transmultiplexer filter. Thus if (a,b,c) is a tuple signal sampled every 6 units, say, then 'temp' will be a signal with a sample rate of 2 units, which has values 'a' for units 1 & 2, 'b' for units 3 & 4, and 'c' for units 5 & 6. If (x,y) is sampled every 4 units then the first set of values of (x,y) will be (a,b), followed by (c,a) 4 units later and (b,c) 4 units after that. In general 'switch' has N inputs and M outputs.

The enhanced ELLA timing model described in [9] will allow transformation of interpolate and decimate onto ELLA macro functions of the form

```
MAC INTERPOLATE = ([INT n]word:input) -> word:
( SEQ
     TYPE intcount = NEW ic/(1..n);
     FN INC = (intcount:in) -> intcount:
               ARITH IF in = n THEN 1 ELSE in+1 FI;
     PVAR count ::= ic/1;
     LET out = input[[count]];
     count :=  INC count;
     OUTPUT out
).


MAC DECIMATE {INT n} = (word:newvalue) -> [n]word:
( SEQ
     TYPE intcount = NEW ic/(1..n);
     FN INC = (intcount:in) -> intcount:
                ARITH IF in = n THEN 1 ELSE in+1 FI;
     PVAR count ::= ic/1;
     PVAR out ::= [n]zeroword;
     LET pastout = out;
     out[[count]] := newvalue;
     count :=  INC count;
     OUTPUT pastout
).
```

and the use of the new ELLA timing primitives of SAMPLE, FASTER, SLOWER (see [9]) will then be used to simulate regions of different speed. However due to the different approaches of modelling time ELLA circuits will need to be operated at the outermost level with equal input/output sample rates.

11.4 Other Differences

Expressions in Silage can have multiple assignments such as

```
(a,b,c) = (x,y,z)
```

ELLA V4 does not have any multiple lets or assignments, however extensions to ELLA have been carried out for SPRITE to allow these features and consequently they will be available in the SPRITE extension of ELLA.

Silage has a library of operators which can be used by programs, however the arguments supplied to such operators require arguments of the same type, thus resulting in the necessity for coercion operators. ELLA also comes with a library of predefined functions and built-in operators which provides a wide range of operators for all ELLA types, including bit strings, and can be used directly or modified by users to suit their own personal requirements. This library provides more flexibility than operators in Silage since arguments of bit strings with different lengths can be supplied. The advantage of this is that ELLA

programs do not need to have coercion functions in order for the operators/functions to be used.

Silage also carries out optimisations on circuits when compiling, for example consider

```
state = in + state@1 - state@2;
return = state + state@1;
```

The two instances of delay @1 would not be made since state@1 = state at t-1 and state is stored as a vector in time. Thus "@" is a pointer to a value in the vector rather than being the actual value. This means that for large array structures duplication of store is avoided. With ELLA the case is slightly different, for example

```
state := in + DEL{1}state - DEL{2}state;
return := state + DEL{1}state;
```

would create two instances of DEL{1}, however when these expressions were being assembled the macro optimisation built into ELLA would ensure that only one instance of code would be produced.

Pragmas are also used in Silage programs, these are commands which instruct the host machine to perform some action, for example split the problem between processors or define a space in local memory. ELLA has the possibility of using attribute data for such tasks.

Silage also has generic functions which are similar to ELLA macros which have INTeger parameters. Silage generic function specifications are of the form

```
FUNC add (a,b : NUM<win, wd>): NUM<win, wd> =
```

where 'win' and 'wd' are user defined integers. Thus suppose this function is in a file 'add.sil', then to create two instantiations with different parameter values requires the following

```
#define add add_12
#define win 12
#define wd 4
#include "add.sil"

#define add add_18
#define win 18
#define wd 6
#include "add.sil"
```

The Silage generic function specification is analogous to the following ELLA macro specification

```
MAC ADD {INT win wd} = ((STRING [win-wd]bit, STRING [wd]bit):a b)
                        -> (STRING [win-wd]bit, STRING [wd]bit):
```

where 'win' and 'wd' are integer parameters supplied when the macro is instantiated. Thus to obtain two explicit instantiations of the macro within an ELLA description, analogous to the two Silage functions given above, requires the following statements

```
...
MAKE ADD{12,4}: add_12,
     ADD{18,6}: add_18.
...
```

It can be noted that ELLA macros can also be instantiated implicitly within an ELLA description.

12 Conclusions

This document considered translation of Silage circuits into functionally equivalent ELLA circuits. A series of ELLA macros which could form a library of translator functions has been outlined. In general there is a mapping from Silage onto ELLA. However translation of unbounded while loops would need inordinate amounts of hardware if translation was restricted to the space domain.

Future extensions to ELLA, which are being carried out under the ESPRIT project SPRITE, will assist designers by allowing them to write ELLA circuits in a Silage like notation, and hence translation will also become easier. In particular, extensions to the ELLA timing model allows interpolate and decimate functions, and extensions to the language allows multiple lets and makes, partial joins and named outputs.

Due to the different approaches of ELLA and Silage there is no simple translation of all ELLA programs into Silage. If a route from ELLA to Silage was required then either a subset of ELLA specifically chosen for its mapping onto Silage, or an expansion of the Silage language would be needed. However ELLA possess a series of transformations for converting high level ELLA constructs into lower level constructs. It is possible that these plus future transformations could transform ELLA circuits into a form suitable for translation into Silage. Being able to transform ELLA has important implications for the synthesis of circuits. It has already been demonstrated [8] that two circuits can be proved mathematically equivalent if written in a very primitive form of ELLA.

13 Acknowledgements

The authors would like to thank John Morison of RSRE Malvern, Wim Ploegaerts, Luc Claesen of IMEC, Leuven and Wim Smits, Thijs Krol of Philips Research Laboratories, Eindhoven for their comments in the preparation of this document. The authors also acknowledge the support of the ESPRIT 'SPRITE' project 2260.

14 References

1. P. Hilfinger, "Silage: A Language for Signal Processing", Internal report, UCB, 1985.
2. C. Scheers, "User Manual for the S2C Silage to C Compiler", Internal report, IMEC, 1988.
3. F. D. Schalij, "Silage: A User Manual", Internal report, Philips, Eindhoven, 1989.
4. Praxis Electronic Design, 20 Manvers Street, Bath, BA1 1PX, UK, "The ELLA V4 Reference Manual", 1990.
5. Praxis Electronic Design, Bath, UK, "The ELLA Tutorial", 1990.
6. Praxis Electronic Design, Bath, UK, "The ELLA System Overview", 1990.
7. Praxis Electronic Design, Bath, UK, "The ELLANET Reference Manual", 1990.
8. M. B. Davies "Mathematical Equivalence in a Primitive ELLA", RSRE memorandum number 4225.
9. M. G. Hill, E. V. Whiting "ELLA Timing", ESPRIT Project 2260 deliverable reference number D3.3 RSRE Y1-M12, 1989.

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....Unclassified.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4385	APRIL	1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
SILAGE TO ELLA TRANSLATION

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
HILL, M G WHITING, E V	21

Abstract

A translation of Silage circuits into functionally equivalent ELLA circuits is considered. A series of ELLA macros which could form a library of translator functions is outlined.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement: Enter any limitations on the distribution of the document
UNLIMITED

880.48