Sticks & Stones: An Applicative VLSI Design Language

by
Luca Cardelli

Internal Report
James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.

CSR-85-81
July, 1981
Sticks & Stones:
An Applicative VLSI Design Language

by
Luca Cardelli

July 1981

University of Edinburgh
Department of Computer Science
Table of Contents

1. Introduction 1
2. Pictures 3
   2.1. Forms 3
   2.2. Restriction 5
   2.3. Renaming 6
   2.4. Composition 7
3. Bunching 8
4. Iteration 9
5. Paths and geometric renaming 13
6. Figures 17
7. Commands 19
8. Modules and externals 20
9. Efficiency 22
10. Acknowledgements 22
11. Syntax 23
12. Predefined Functions 27

List of Figures

Figure 2-1: A Blue Square 3
Figure 2-2: An n-MOS Inverter 4
Figure 2-3: Restriction 6
Figure 2-4: Renaming 6
Figure 2-5: Composition 7
Figure 3-1: Bunching 9
Figure 4-1: "times" Iteration 9
Figure 4-2: "for" Iteration 10
Figure 4-3: Double Iteration 11
Figure 4-4: A Selector 13
Figure 5-1: A Blue Cross 15
Figure 5-2: Geometric Renaming 16
Figure 5-3: Bunch Renaming 17
Abstract

A great deal of activity is currently being devoted to the development of design systems for VLSI, mainly because this seems to be the only way we can go about exploiting the amazing technologies that are becoming available [Mead 80]. We are concerned here about a design language for the hierarchical and topological description of stick diagrams and geometric layouts, with particular attention to syntactic clarity, expressiveness and flexibility.
1. Introduction

The most important attribute of a flexible design language for VLSI is perhaps its ability to parameterise a picture in any possible aspect, e.g. size, number and type of components and distance between them. This suggests that the language should be mainly text oriented (with graphic facilities), as in this case parameterisation can be easily achieved by procedure parameter passing. A display oriented language has instead severe problems in this respect: it is very easy to assemble figures on a screen by some pointing device, but it is difficult to express how these figures are actually meant to change as a function of some parameters.

Textual languages for graphics, however, suffer from severe drawbacks as the identification of text and image can be very difficult. Any such language should then be highly interactive with immediate visual feedback, and the syntax should recall as far as possible the structure of the picture, i.e. its topological properties. This contrasts sharply for example with graphic packages, in their use as extensions to existing host languages.

The kind of language we are interested in, should be able to express naturally VLSI circuits by their hierarchical structure and their topological properties [Buchanan 80, Rowson 80, Williams 77]. It should be mainly oriented towards describing stick diagrams, as this is the language MOS designers use to communicate ideas, and the structure of the circuits should appear through the text of the descriptions.

Here we present a language, called Sticks & Stones, based on these ideas, which admits a precise interpretation in terms of geometric layout. A purely topological version of the language can be used to specify and communicate stick diagrams in textual form. A more concrete and implementable version, obtained by adding the strictly necessary geometric details, can be used as an effective high-level design tool to prepare masks for VLSI processing.

In Sticks & Stones, pictures are handled just like an abstract data type within a general purpose programming language, so that every picture is
denoted by a program which builds it. The operations over pictures have been
inspired by Milner's Flow Algebra operators [Milner 79] because of their
syntactical clarity and expressiveness and of their algebraic properties.
These operations are topological in nature and give rise to programs which are
suggestive of the pictures they represent. Pictures are embedded in an
applicative higher-order language, which is based on a subset of Edinburgh ML
[Gordon 79]. The control structures of the language can be freely used to
define arbitrary parameterisations and conditional assemblies of pictures.

The language is applicative in two of the senses commonly attributed to
this word; it is expression oriented and free from side-effects. Expressions
seem to be more suited than statements to an interactive language. They
improve and enforce the structured description of complex pictures and help in
keeping information local. Every picture is taken to be an unmodifiable and
unbreakable object, which can only be used to make larger pictures, and which
can only be manipulated through its set of named ports. Picture composition
is then done by port names (and not by geometrical position or displacement)
with automatic translations and rotations.

Side effects might be needed to edit a picture, but we regard this problem
as completely distinct from that of picture construction. Editing a picture
is also very different from editing a text or a tree, as in the former case
there may be very troublesome context dependent effects, like those resulting
from increasing the size of a subcomponent. In this context, editing by
rebuilding can be much more convenient than editing by modifying, especially
if an adequate structure of program modules is provided.

If side effects are forbidden, a "correctness by construction" approach can
be applied. We might be able to show that a picture enjoys some property P
(e.g. absence of geometric rules violations) if its basic components have the
property P and if picture operations preserve the property P. Thus, the amount
of checking to be done when composing two pictures can be drastically reduced.

Sticks & Stones has been designed by Gordon Plotkin and me. This paper
describes the geometry-oriented implementation running on the ERCC DEC-10 at the University of Edinburgh; graphic output can be produced on Charles colour graphics terminals, HP-7221A plotters and Tektronix 4010 terminals. A more abstract discussion on VLSI design can be found in [Cardelli 81].

2. Pictures

A picture is either an elementary picture (called a form) or the composition of smaller pictures. Pictures form an abstract data type and are first-class objects in the language.

2.1. Forms

A form is made of a set of figures (boxes, polygons, etc.) with a sort. The sort of a picture is a list of ports, and ports are used to connect pictures together.

- let bluesquare = form(b.S : W port [0°0,0,1];
  b.E : W port [1°0,0,1];
  b.N : W port [1°1,1,1];
  b.W : W port [0°1,2,1])
with B box [0°0,1];

bluesquare = (b.S;W; b.E;W; b.N;W; b.W;W) : [1,1]

A phrase like "let bluesquare = ...;" is used to define the variable "bluesquare" at the top level (the string "--" preceding it, is the Sticks&Stones prompt). The answer from the system is "bluesquare = --", where "--" is the result of the evaluation of "...". In this case the result is a "( )" (i.e., a picture whose structural details have been omitted) of sort "( ... )" and of size 1,1 which is the size of the minimum enclosing rectangle.

The figure bluesquare (Figure 2-1) is a form (an elementary picture) made of a single B (blue) box with lower left corner at the point 0°0, and upper right corner at the point 1°1. It has four ports "b.S", "b.E", "b.N" and "b.W".

A port name can be any list of identifiers and numbers (starting with an identifier) separated by dots, like "a" or "aaa.bbb.1,c,3"; these identifiers
and numbers are called atomic parts of a compound port name. Port names have no semantic significance, but they will often suggest the function of their associated port (e.g. "b.E" will stand for "blue East").

The port "b.S" is a W (white) port starting at 0°0 in direction 0 degrees anticlockwise from the x axis, for a length of 1. The starting point of a port is its tail, and the other end is its tip. A port is looked at as a vector whose north is in the tail-to-tip direction.

A more complete example is this n-mos inverter (Figure 2-2):

```plaintext
let inverter =
   form (b.E:B port [5 5, 90, 4];
      b.W:B port [1 9, 270, 4];
      g.S:O port [2 0, 0, 2];
      r.E:N port [6 1, 90, 2];
      g.E:G port [6 4, 90, 2];
      r'.E:N port [6 7, 90, 2];
      g.N:O port [4 15, 180, 2];
      r.W:N port [0 3, 270, 2])
   with B box [1 4, 5 10]
   and G box [0 0, 6 8; 2 8, 4 15]
   and R box [0 7, 6 15; 0 1, 6 3]
   and Y box [0 5 5, 5 5 16 5]
   and C box [2 5 4 9];

inverter = (b.E:B; b.W:B; g.S:O; r.E:R; g.E:G;
   r'.E:N; g.N:O; r.W:N) : [6 16.5]
```
Figure 2-2: An n-MOS Inverter

Ports of type B (blue) G (green) and R (red) are drawn in the respective colour. Ports of any other type are also admitted, and are drawn in the foreground colour (depending on the graphic device).

Boxes can be of colour B (blue) G (green) R (red) Y (yellow) C (black) and W (white), and may overlap; other colours are syntactically admitted and are drawn in the foreground colour. Note that a list of rectangles can be specified after the keyword "box".

Ports should always be oriented anticlockwise around a picture. This is not mandatory, but picture composition is made connecting ports on their east sides (tail to tip and tip to tail), and the anticlockwise convention ensures that pictures are joined on their outer sides. A picture may have no ports and/or no figures. The empty picture is simply:

- form;
  <> : () : [0,0]

2.2. Restriction

Restriction is used to forget about some of the ports of a picture; the syntax is: expression, followed by ";", followed by a list of port names (see Figure 2-3):
- inverter \ b.W  !.E g.?;
<> : (r.W:R) : [6,16.5]

![Diagram](image)

Figure 2-3: Restriction

Question marks and exclamation marks are used to pattern match port names. Any variable beginning with an exclamation mark (like "!", "!!", "!?bo" or "!?bo") matches with a single atomic part of a compound port name, while any variable beginning with a question mark matches with an arbitrary number (zero included) of atomic parts.

In the example above we withdraw the port b.W, all the E(ast) ports and all the g(reen) ports from the inverter. The inverter itself is not affected by this operation and a truly new picture is generated.

2.3. Renaming

The renaming operation performs a simultaneous substitution over the ports of a picture; the syntax is: expression, followed by "(!", followed by a list of single renamings separated by ";", followed by ")". A single renaming "a\b" means "a becomes b" (see Figure 2-4).

- inverter [r'.E\inv.r'.E; l.W\inv.l.W];
<> : (b.E:B; inv.b.W:B; g.S:G; r.E:R; g.E:G; inv.r'.E:R;
g.W:G; inv.r.W:R) : [6,16.5]
Match variables instantiated in the left part of a substitution can be used in the right part to get group renamings like "I.W\inv.I.W" which is an abbreviation for "b.W\inv.b.W; r.W\inv.r.W". Note that "I.I" matches "a.a" but does not match "a.b", which is matched by "I.I", "I.?", "I.??" or "??", but not by "I" or "I.I.II". You can go as far as "I.I.II.\ Sub 1.I.I.??", which renames "a.a.b.3.5" into "b.a.3.5.3.5.b". A question mark in the left hand side can only appear as the last atomic part, otherwise the matching might be ambiguous. A matching variable in the right hand side which does not appear in the left hand side is illegal.

2. Composition

Having two pictures, we can compose them by port names; the syntax is: expression, "[;", list of single links separated by ";", "]", expression. A single link has the form: portname, "--", portname.

- redblue [r.E -- g.W] greenblue;

<> : (r.S;W; g.S;W; g;E;W; g.N;W; r.N;W; r.W;W) : [2,1]

where redblue and greenblue are defined similarly to bluesquare. This composition produces two adjacent squares (Figure 2-5), where the ports r.E of redblue and g.W of greenblue have been connected and forgotten.
Several links can be specified inside the composition brackets, separating them by semicolons. All the ports involved in a connection are forgotten in the result; whose sort is otherwise the union of the sorts of the composing pictures. Pattern matching is not allowed in composition; experience has shown that its use leads to unclear programs.

Composition is a symmetric operation (in the sense: \(P[i:p(1)\rightarrow q(1)];Q\cong Q[q(1)\rightarrow p(1)];P\)) and as an infix operator associates to the left. Every pair of ports which are being linked in a composition must have the same type and the same size. Composition with the empty picture by any pair of ports leaves a picture unchanged.

Connection of two ports is made tail to tip and tip to tail with no distance between them. In case of connection of several pairs of ports, the main link is connected first, and all the other pairs of ports must face each other, maybe with a gap in the middle. The main link is defined as the first link on the left, inside the composition brackets.

3. Bunching

Every port is actually a bunch, or collection of collinear vectors. Up to now we only considered single-vector ports, but a port can also be a list of vectors:

\[ R \text{ port } [0^\circ, 0, 1; 2^\circ, 0, 1; 5^\circ, 0, 1] \]

Every vector in a bunch must have the same type, orientation and size, and they must be collinear, but they can be differently spaced. Bunches may also
be interleaved. When two ports are composed, every vector in one port must match with a corresponding vector in the other port.

Bunches usually arise from composition: when two pictures are composed, the ports with equal names which are not being linked get bunched together:


\[\triangleright: (b.S:b; b.E:b; b.W:b; b.W:b) : [2,1]\]

Here b.S and b.N are two bunches of two, which are drawn as a single arrow in Figure 3-1. Again bunching only succeeds for collinear ports of the same size; otherwise an error is reported.

![Figure 3-1: Bunching](image)

**Figure 3-1: Bunching**

Bunches allow to compose regular arrays of pictures without having to index by renaming every picture in the array. They keep low the total number of ports in a picture making composition simpler and more efficient.

4. Iteration

Iteration is used to make regular arrays of cells, like in:

- 3 times bluesquare with [b.E:b.W];

\[\triangleright: (b.S:b; b.E:b; b.W:b; b.W:b) : [3,1]\]

![Figure 4-1: "times" Iteration](image)

**Figure 4-1: "times" Iteration**
which is equivalent to:

- bluesquare ![b.E—b.W]
  bluesquare ![b.E—b.W]
  bluesquare;

\[ \diamond : (b.S:w; b.S:w; b.N:w; b.W:w) : [3,1] \]

Iteration is totally equivalent to some obvious recursive program one might write in the language, but is more efficient and syntactically clearer.

Iteration often produces bunches.

Iteration variables are admitted in the "for" form of iteration:

- let blue = bluesquare ![b.?\?]
  and red = realsquare ![r.?\?]
  and green = greensquare ![g.?\?];

blue = \[ \diamond : (S:w; E:w; N:w; W:w) : [1,1] \]
red = \[ \diamond : (S:w; E:w; N:w; W:w) : [1,1] \]
green = \[ \diamond : (S:w; E:w; N:w; W:w) : [1,1] \]

- for square in ![blue; red; green]
  iter square
  with ![E—W];

\[ \diamond : (S:w; E:w; N:w; W:w) : [3,1] \]

![Figure 4-2: "for" Iteration](image)

which produces the picture in Figure 4-2. The iteration variable "square" takes in turn the values in the list.

Double iteration can be used to produce arrays:
- let squares array =
  for row in array
    iter for item in row
    iter item
    with [;S—W;]
    with [;S—W;]

squares = """

Figure 4-3: Double Iteration
(where """" means that 'squares' is a function). This is the definition of a function taking a list of lists (i.e. an array) of pictures and producing a parametric picture. It can be used as follow:

- squares [[blue; green; red ];
  [green; red; blue ]];
  [red; blue; green]];

<> : (S;W; E;W; N;W; W;W) : [3,3]

Sometimes it is useful to iterate concurrently through several lists; this feature is used in the following definition of "squares" which substitutes a green column every three input columns:
- let squares' array =
  for row in array
    iter for item in row and i in 1:length row
      iter (i mod 3)=0 => green | item
      with ([/E=x];
      with ([/S=N]);

squares' = ""

where the operation "n:m" produces the list of all numbers from n to m, and
"
"a => b | c" means "if a then b else c".

A selector is a realistic example of a parametric picture which can be
built by double iteration. We need first to define three basic building
blocks: 'pos' (an enhancement transistor), 'neg' (a depletion transistor) and
'out' (a piece of the common output):

- let pos =
  (form (r.S:R port [2"0,0,2];
    g.E:G port [6"2,90,2];
    r.N:R port [4"6,180,2];
    g.W:G port [0"4,270,2])
  with R box [2"0,4"6]
  and G box [0"2,6"4])

and neg =
  (form (r.S:R port [2"0,0,2];
    g.E:G port [6"2,90,2];
    r.N:R port [4"6,180,2];
    g.W:G port [0"4,270,2])
  with R box [2"0,4"6]
  and G box [0"2,6"4]
  and Y box [0.5"0,5,5.5"5])

and out =
  (form (g.S:G port [2"0,0,2];
    g.N:G port [4"6,180,2];
    g.W:G port [0"4,270,2])
  with G box [2"0,4"6; 0"2,6"4]);

We now need to put these pieces together; the following program takes a number
n and produces a selector with n control inputs (the n-bunch "r.N"), n
complemented control inputs (the interleaved n-bunch "r'.N"), 2 to the n input
lines (the 2*n-bunch "g.W"), one output line (the 1-bunch "g.N") and the
appropriate pattern of enhancement and depletion transisters (produced by the
auxiliary function "bit". 

- let sel n = 
  for i in 1..exp(2,n) 
  iter (for j in n::) 
    iter bit(i-1,j-1) s0 => 
      pos [[g.E—g.W]; (neg[r.7\r'.7])] | 
      neg [[g.E—g.W]; (pos[r.7\r'.7])] 
      with [[g.E—g.W]] 
    with [[r.S—r.N; r'.S—r'.N; g.S—g.N]] 
  where sel bit(i,j) = 
    j=0 => 1 mod 2 | bit(i//2,j-1); 

where 'exp' is exponentiation and '//' is integer division.

![Figure 4-8: A Selector](image)

5. Paths and geometric renaming

A path can be generated by taking a port and moving it around: the wake of 
the port forms the resulting path. The outcome of this operation is a list 
of polygons (one or more for every step the port has made) and a new port (i.e. 
the old port in the new position). Hence a path is the following data type:

path = (polygon list) x port

Given a path, the following operations extend it generating a new path:
stay: path \rightarrow \text{path} \\
move: \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
step: \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
rol: \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
rotr: \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
move': \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
step': \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
rol': \text{num} \rightarrow \text{path} \rightarrow \text{path} \\
rotr': \text{num} \rightarrow \text{path} \rightarrow \text{path}

The operation stay leaves a path unchanged;

The operation move takes a positive number \text{n}, a path \text{p} and moves the port of the path \text{n} units. The direction of movement is towards the east of the port (i.e., generally outwards with respect to the picture if anti-clockwise ports are used). The new path generated is made of the new port and the old polygon list with a new rectangular polygon having the old and new ports as edges.

The operation step is like move, but 'step n' means 'move \text{n} times the size of the port' for simple ports, and 'move \text{n} times the size of the vectors in the port' for bunches.

The operation rol (rotate left) takes a number \text{n} (in degrees), a path \text{p} and rotates the port of the path \text{n} degrees anti-clockwise describing a circular arc with center in the tip of the port. If the port is a bunch, the distances between the vectors are respected and the result is a set of concentric paths. The new path generated is made of the new port with the old polygon list plus the new polygon(s) generated by the rotation.

The operation rotr (rotate right) is the same as rol, but the rotation is clockwise and its center is in the tail of the port.

The operations move', step', rol' and rotr' are similar to their unprimed versions, but they move a port without producing any path between the old and new position. The operations move' and step' also accept negative arguments.

Functions from paths to paths are called path functionals; the following
are path functionals:

\begin{verbatim}
  stay
  move 2
  step 5
  rotl 90
  rotr 270
\end{verbatim}

Function composition is used to compose path functionals; in particular it is convenient to use the inverse function composition operator "&":

\[(f & g) x = g(f x)\]

Here is an example of a composite path functional:

\begin{verbatim}
  move 2 & rotl 90 & step 4 & rotr 90 & move 2
\end{verbatim}

note that "&" behaves like an append on paths, as function composition is associative.

How do we use path functionals? Ports are not available to the user as data objects separated from pictures, so that path objects can never be built, and there is nothing to apply path functionals to. The only place where is possible to use path functionals is in the geometric renaming feature of the renaming operation:

- `bluesquare [?\? move 2];`


The meaning of this is to rename every port in bluesquare by its own name, moving it 2 units outwards. The result is a blue cross of size [5,5] (Figure 5-1). The path functional "move 2" is applied in turn to the paths obtained pairing the ports of bluesquare with the empty list of polygons.

Here is a very flexible blue square which can be stretched symmetrically in four directions by applying a path to it:
- let bluewheel path = bluesquare [T\7 path];
  bluewheel = ""
- bluewheel (move 2 & rotr 45 & move 15 & rotr 135 &
  move 30 & rotr 45 & move 20 & rotr 270);
<> : (b.S; b.E; b.N; b.W; b.W; b.W; b.W; b.W; b.W; b.W) : [68.9, 68.9]

Figure 5-2: Geometric Renaming

A limited form of routing (called river-routing) can be obtained by using
geometric renaming on bunches, like in Figure 5-3:
- set 2 {g, w \ g, w rotr 60 & rotl 60 & move 6} \ ?;

\( \odot : (\cdot) : \{51,32,32\} \)

---

**Figure 5-3: Bunch R enamel**

6. Figures

There is a variety of elementary figures. Actually many of them have no application in VLSI and are intended mainly for graphics. All of the following options can appear syntactically after the keyword 'with' inside forms (in the place of boxes in the examples of the previous section).

- **dot [p1; ... ;pk]** draws dots at the specified points p1 ... pk.

- **line [l1; ... ;lk]** draws a set of lines l1 ... lk; every line is a list of points l1=[p1; ... ;pkl] which are joined by straight segments.

- **path [l1; ... ;lk]** draws a set of paths l1 ... lk; every path is a list of pairs of numbers and points l1=[n1,p1; ... ;nkl,pkl]. Adjacent points p(j),p(j+1) in a path are joined by a circular arc of aperture n(j+1) degrees (if n(j+1) is 0 or any multiple of 360, a straight segment is used) If n(j+1) is positive, the arc is convex on the east of the vector p(j)->p(j+1); if negative it is convex on the west. The first aperture n1 is not used.

- **spline [l1; ... ;lk]** draws a set of non-periodic cubic B-splines l1 ... lk; every spline is built from a list of control points l1=[p1; ... ;pkl]. The spline does not pass through the control points (except the first and the
last), but is tangent to every segment joining two adjacent control points.

loop [l1; ... ;lk] draws a set of periodic cubic B-splines l1 ... lk; every
spline is built from a list of control points lis=[p1; ... ;pk]. The spline
is tangent to every segment joining two adjacent control points (the last
point is adjacent to the first) and describes a closed curve.

box [p1,q1; ... ;pk,qk] draws a set of boxes with lower left corner at the
point p1 and upper right corner at the point q1.

poly [l1; ... ;lk] draws a set of polygons l1 ... lk; every polygon has a
line lis=[p1; ... ;pk] as perimeter. The last point pk is joined back to the
first.

area [l1; ... ;lk] draws a set of areas l1 ... lk; every area has a path
lis=nl,p1; ... ;nk, pk] as perimeter, where the first aperture nl is used to
join the last point back to the first.

blob [l1; ... ;lk] draws a set of blobs l1 ... lk; every blob has a loop
lis=[p1; ... ;pk] as perimeter.

text [p1,s1; ... ;pk,sk] draws a set of character strings s1 ... sk
starting respectively at the points p1 ... pk. Every string may contain
control information (following the escape character '%' according to this
code: '\r' change colour to red; '\g' change colour to green; '\b' change
colour to blue; '\y' change colour to yellow; '\b' change colour to background
(black for Charless, white for HP plotter etc.); '\f' change colour to
foreground (white for Charless, black for HP plotter etc.); '\d' ... '\y'
change text size (0=min, 9=max); '\s' halt plotting and wait for a carriage
return to continue (e.g. to change page on the HP plotter); '\x' for any other
character '\x' to actually display '\x' (e.g. '\s'). Note that the escape
character '%' is only interpreted by the plotting routines while the normal
escape character '/' should be used for any other purpose (e.g. to insert a
'\n').
7. Commands

The following commands are accepted at the top level.

mode: this command investigates the state of the environment, showing what options are active and what are not. Options are: print: when active, the result of every top-level evaluation is printed at the terminal. charles: when active, the result of every top-level evaluation is drawn on a Charles colour graphic terminal. tektronix: when active, the result of every top-level evaluation is shown on a Tektronix terminal. hpplot: when active, the result of every top-level evaluation is plotted on a HP-7221A plotter. drawnames: when a plotting device is active, draws the names of the ports at their location. drawports: when a plotting device is active, draws the ports at their location as little arrows. signature: when a plotting device is active, puts a signature 'Sticks&Stones' in the lower right corner. page: when a plotting device is active, plots in 'page' mode. Every picture shown will fit incrementally the available space from top to bottom (it will try to make pictures horizontally as large as possible). On the HP plotter, pictures will fit an A4 sheet of paper. logfile: produces a log file 'STICKS.LOG' containing a transcript of the terminal input. Use 'addmode logfile' to open a new logfile (destroying the old one) and start writing on it, and 'submode logfile' to save it and stop writing on it.

addmode m1, ..., mn: adds the modes m1 to the current mode.

submode m1, ..., mn: subtracts the modes m1 from the current mode.

print v: prints the object v; all the plotting motions are suppressed for the duration of this command.

draw v: draws the object v on the currently active device(s). Print is suppressed for the duration of this command. If v is a picture, it is plotted. If v is a list of n items, the screen is horizontally divided into n viewports, and every item in the list is drawn in a viewport; if an item in v is again a list, its viewport is divided vertically, and so on horizontally
and vertically to any depth. If \( w \) is not a picture, nothing is shown (this should be intended recursively).

**contents**: shows the names of the variables defined at the top level.

**undo**: the result of the last expression evaluated is always kept in the top level variable "it". The command "undo" can be used to reset "it" to its previous value (only once).

**use**: loads a module (described in section "Modules and externals").

**import**: imports an external picture (described in section "Modules and externals").

**export**: creates an external picture and generates a CIF file (described in section "Modules and externals").

8. **Modules and externals**

Some modules (called library modules) are predefined in the system, as for example "constants" (basic cells) and "pla" (pla generator). Modules can contain data (like "constants") or programs (like "pla"), and can be used by the command:

- use constants,pla;

which loads the definitions contained in constants and pla.

New modules can be generated by editing files with extension "*.STK", containing Sticks & Stones expressions and definitions. Every module can "use" other modules.

Externals arise when, at the end of a session, we want to save the pictures produced so far. If a very big and very time-consuming ALU has been produced, it can be saved as follows:

- export ALU;

ALU exported
This command generates: (i) a CIF file of the ALU, called "ALU.CIF", and (ii) a file containing boundary information about the ALU, called "ALU.STX". The ALU can be recalled by:

- import ALU;

  ALU = ☐ : ...

The advantage of externals is that it is possible to use the ALU in another session without having to build it again. To import something takes almost no time, as only boundary information (i.e., ports) is used (an imported picture is drawn as a white frame with ports). Moreover the ALU can be used as a component of a CPU, and when the CPU is exported, the system merges the already existing ALU.CIF file with the rest of the picture. CIF files generated by "export" can be used for plotting or for mask fabrication.

The import command is also used to interface already existing CIF files to Sticks & Stones. Given a CIF file REG.CIF, we only have to write a file REG.STX and then "import REG;". The STX file should contain a form describing the ports of the REG, and should declare it to have a figure (e.g., a box) of the right size:

```
let REG =
  form (VddIn:B port ...; VddOut:B port ...;
  GndIn:B port ...; GndOut:B port ...;
  BusIn:B port ...; BusOut:B port ...;
  ReadIn:N port ...; ReadOut:N port ...;
  WriteIn:B port ...; WriteOut:B port ...;
  ClockIn:N port ...; ClockOut:B port ...)
with W line [[0.0;36.0;36.0;36.0;36.0;0.0]];
```

"export" uses a "line" to generate a white frame, like in this example.

CIF files generated by Sticks & Stones are compact, as common subpictures are factorised into CIF symbols, and calls to these symbols are generated where necessary. Moreover they are commented: every CIF symbol is associated to the name(s) used in Sticks & Stones to denote it.
9. Efficiency

The composition algorithm is linear in the number of (bunch) connections and independent of the number of ports of the sorts involved. Every connection takes a constant time of about 1/20 sec. DEC/10 opu (not counting plotting time).

If possible, iteration should be used instead of recursion and the "times" form of iteration should be preferred. In the latter case the iteration body needs to be evaluated just once (because the language is applicative) instead of n times. But what is more important, the system can use a logarithmic algorithm instead of a linear one, producing at any step 1, 2, 4, 8, 16 etc. instantiations of the iteration body and then composing them up to get the desired number. The gain in efficiency is considerable: to produce a 16x16 array of four-port cells the "times" iteration takes 8 connections against the 255 of the "for" iteration.

Because of the absence of side-effects, it is possible to maximally share in memory all what is shareable; hence "let" should be used to factorise common subexpressions. An array of 16x16 cells can be produced by allocating just one call plus 8 connection records. If instead we put an expanded cell definition inside a double iteration with iteration variables we can cause the allocation of 256 identical cells plus 255 connection records.

10. Acknowledgements

Jeff Tansley and Irene Buchanan stimulated my interest in VLSI and VLSI design. Many of the good ideas contained in this paper originated from Gordon Plotkin, and other ones came up during discussions with Kevin Mitchell and Mark Snir. This work was carried out under a scholarship of the National Research Council of Italy and a scholarship of the University of Edinburgh.
I. Syntax

The following conventions are used: strings between quotes (""') are terminals; identifiers are non-terminals; \texttt{\textasciitilde} is disjunction; \texttt{[...]} means zero or one times "."; \texttt{( ... )n} means \texttt{n} or more times "." (default \texttt{n=0}); \texttt{[ ... / .... ]n} means \texttt{n} or more times "." separated by "/...." (default \texttt{n=0}); parenthesis 	exttt{( ... )} are used for precedence; juxtaposition is concatenation.

topterm ::= (command | toplet | topletree | term) ";"

command ::= mode | addmode | submode | print | draw | undo | use | begin | end |
          contents | import | export

mode ::= "mode"
addmode ::= "addmode" [ ide / "," ]
submode ::= "submode" [ ide / "," ]
print ::= "print" term
draw ::= "draw" term
undo ::= "undo"
use ::= "use" [ ide / "," ]
begin ::= "begin" port
end ::= "end" port
contents ::= "contents"
import ::= "import" ide
export ::= "export" ide

toplet ::= "let" declaration
topletree ::= "letree" declaration
term ::= variable | bool | string | number | point | pair |
      list | form | composition | restriction | rename | |
      conditional | abstraction | application | iteration |
      let | letree | where | whereree | parterm |
      and | or | not | minus | cons | append | sum | diff |
      times | divide | equal | great | less | greatereq |
      lesseq | range | mod | directcomp | reversecomp

variable ::= ide

bool ::= "true" | "false"

string ::= "" characters ""

number ::= unsgnedreal

point ::= term "" term

pair ::= term "," term

list ::= "[" (term / ";" ) "]"
form ::= "form" [sort] ["with" [figure / "and"]]
sort ::= "(" [port "=" id: ["port" term] / ";" ] ")"
figure ::= ide shape term
shape ::= "dot" | "line" | "path" | "spline" | "loop" |
        "box" | "poly" | "area" | "blob" | "text"
connection ::= "]:" [port "---" port / ";"] "]"
restriction ::= term "[" [match]!
rename ::= term "[" substitution / ";"] "]"
substitution ::= match "[" match [term] |
               match term
iteration ::= term "times" term "with" connection |
             "for" [struct "in" term / "and"] |
             "iter" term "with" connection
conditional ::= term "->" term "|" term
abstraction ::= "****" [struct]! "****" term
application ::= term term
let ::= "let" declaration "in" term
letrec ::= "letrec" declaration "in" term
where ::= term "where" declaration
wherequad ::= term "wherequad" declaration
declaration ::= [funstruct "=" term / "and"]!
funstruct ::= struct | ide [struct]!
struct ::= "(" [" : "] | ide | struct "=" struct |
          struct "," struct | [" [struct / ";"] "] |
          struct ";" struct | [" struct "]
parterm ::= "\" term ")\"
and ::= term "And" term
or ::= term "Or" term
not ::= term "Not" term
minus ::= ";-" term
cons ::= term ";-" term
append ::= term "&" term
sum ::= term "+" term
diff ::= term "-" term
times ::= term "*" term
divide ::= term "/" term
equal ::= term "=" term
greater ::= term ">" term
less ::= term "<" term
greateq ::= term "\geq" term
lesseq ::= term "\leq" term
range ::= term ":" term
mod ::= term "mod" term
directcomp ::= term "o" term
reversecomp ::= term "\&" term

letter ::= "a" | ... | "z" | "A" | ... | "Z" | "W"
digit ::= "0" | ... | "9"
ide ::= letter | ide letter | ide digit
matchide ::= "|" | "?" | ide "!" | ide "?" |
matchide !" | matchide ?" |
matchide letter | matchide digit

integer ::= digit | integer digit
unsignedreal ::= integer ["." integer]
port ::= ide | port ";" ide | port ";" integer
match ::= matchide | port ";" matchide |
match ";" matchide | match ";" ide |
match ";" integer

Precedence of operators. "a \* n" means that the infix operator \\"\*\" has left
precedence a and right precedence n. An expression "x \* y \* z" associates
like "((x \* y) \* z)" if n>m and like "x \* (y \* z)" if n<m. Hence m<n means
that \"\*\" is left associative and m>n that it is right associative.
100  Or  100
200  And  200
301  ,  300
401  ;  400
500  @  500
600  =  600
700  >  700
700  <  700
700  >=  700
700  <=  700
800  mod  800
900  -  900
1000  ::  1000
1100  +  1100
1100  -  1100
1200  #  1200
1200  /  1200
1200  //  1200
1300  o  1300
1300  &  1300
1400  1400  (application)
II. Predefined Functions

And (infix) boolean and.
Or (infix) boolean or.
Not (infix) boolean not.

= (infix) equality over booleans, numbers, points, pairs and lists only.
> (infix) greater than.
< (infix) less than.
>= (infix) greater then or equal to.
<= (infix) less than or equal to.
- (prefix) number complement.
+ (infix) number sum.
- (infix) number difference.
* (infix) number product.
/ (infix) number division.
// (infix) integer division.

mod (infix) number modulo: 'a mod b' is the remainder of 'a/b'.

lft point left: lft ("a"b) = a.
rht point right: rht ("a"b) = b.

fat pair first: fat (a,b) = a.
and pair second: and (a,b) = b.

hd list head: hd [a1; ... ;an] = a1 (n>0).

tl list tail: tl [a1; ... ;an] = [a2; ... ;an] (n>0).

null list null: null [] = true;
null [a1; ... ;an] = false (n>0).

| (infix) list cons: a [a1; ... ;an] = [a;a1; ... ;an] (n>0).

@ (infix) list append: [a1; ... ;an] @ [b1; ... ;bm]
= [a1; ... ;an;b1; ... ;bm] (n,m>0).
:: (infix) range: n::m = [n;n+1; ... ;m-1;m] (n<=m);
n::m = [n;n-1; ... ;m+1;m] (n>m).

length list length: length [a; ... ;an] = n (n>=0).

o (infix) function composition: (f o g) a = f (g a).

& (infix) reverse function composition: (f & g) a = g (f a).
REFERENCES

[Buchanan 80] I. Buchanan.
Modelling and Verification in Structured Integrated Circuit Design.

An Algebraic Approach to VLSI Design.


Introduction to VLSI Systems.
Addison-Wesley, 1980.

Flowgraphs and Flow Algebras.

[Rowson 80] J. A. Rowson.
Understanding Hierarchical Design.
PhD thesis, California Institute of Technology, Department of Computer Science, April, 1980.

[Williams 77] J. D. Williams.
Sticks -- A New Approach to LSI Design.