PUN FORMAT

William Bricken
August 2003

CONTENTS

ILOC

ILOC is a suite of three hierarchical software tools. At the core, the *Boundary Logic Engine* (Losp) provides Boolean minimization of logic function graphs. The *Circuit Graph Engine* (CDE) manipulates circuitry-specific transformations on structure, such as structure sharing and technology library mapping. The outer layer of functionality, the *Application Interface*, admits and generates EDIF specifications, provides batch-mode processing, test-vector verification, statistical analysis, and interfaces to CAD and other interactive systems.

ILOC calls Losp for minimization, then applies structural rearrangement algorithms to the minimized parens forms. Due to the homogeneity of parens notation, patterns in the circuit can be easily identified. ILOC identifies common gate patterns (XOR, MUX), common structures that can be combined by increasing fanout (structure sharing), and common patterns found in technology libraries (technology mapping).

The *Circuit Design Engine* (aka *Circuit Design Explorer)*, is a stand-alone application of Losp/ILOC as an EDA product prototype. The CDE generates candidate circuits within a parameterized design space. Candidates are structural variations of the same functionality. The designer specifies a configuration of behavioral, topological, and technology mapping parameters and provides a description of functionality (reference netlist, set of logic equations, HDL behavioral specification, FSM, etc.). When feasible, the CDE produces a circuit which satisfies the specified structural constraints.

MODELS

Boundary logic algorithms can be expressed using many different data structures, such as

- Parens Notation
- Pun Notation
- Adjacency lists
- Relations
- Object-oriented
- Bit-stream

Each of these may come with a specific mode of processing and set of algorithmic clichés (list, logic, set, relational, object, and bit-stream processing metaphors).

Pun notation is native to ILOC, while parens notation is native to Losp. Pun expresses the structure of circuits as a partitioned graph. It uses multiple data structures internally, as an artifact of experimental development.

Modeling Circuits

The relationship of containment is sufficient for the expression and transformation of Boolean logic and for the gate-and-wire components of a circuit. Formally, there is a morphism between the inside/outside distinction of containers, the antecedent/consequent distinction of logical implication, the input/output relation of variary NOR gates, and the parent/child distinction of directed graphs.

Registers and feedback in circuits are simply principled and choreographed breaking of containment; the state contained inside a register is transferred to the outside of the register (into the circuit as a whole) at specified locations and times.

The Advantage of Boundary Logic

In all representations, boundaries serve both to define objects and patterns, and as the connection between objects. (Boundaries both separate and connect.) In a circuit, each boundary is both a variary NOR gate and the wire exiting that gate. Connectivity and logical functionality are confounded in the Pun representation. The motivation for changing structure can be either behavioral (grouping logic) or topological (eliminating wires). The identicality of wires and gates in a distinction network circuit permits both wiring and logic to be minimized concurrently.

REPRESENTATIONS

Examples of several varieties of representation of boundary logic follow. All are for the same logical form.

LOGIC: (a AND (a OR b))

PARENS with NUMBERED SPACES:

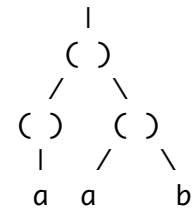
```
( ( a) ( a b) )
0 1 2    3
```

EXTRUDED PARENS:

```
(             )       1
  (  ) (    )       2   3
     a    a b
```

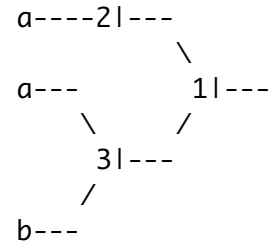
IS-CONTAINED-BY PAIRS: {<a 2>,<b 3>,<2 1>,<3 1>,<1 0>}

GENERALIZED NOR TREE:



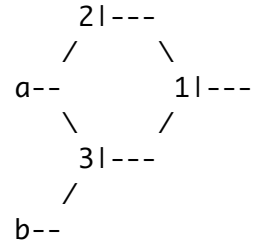
For notational convenience, we can use labels for the internal tree nodes. These nodes are called distinction nodes, or *dnodes* for short. Dnode labels identify the space contained by the dnode, the subgraph for which they are root. Rotating the NOR tree clockwise by 90 degrees yields a circuit schematic-like representation:

DNODE TREE:



When a node is inside more than one container, the tree becomes a graph, and nodes fanout to multiple containers. The containment relation is now many to many.

DISTINCTION NETWORK:



PARENS NOTATION

Containers can be represented in linear text by well-formed parentheses (WFP), called *parens*. A parens expression makes containment visually obvious: each parens contains other parens forms, grounding in either an empty parens or a label. The varieties of WFPs are countable with the Catalan numbers, and are studied under the technical name of Dyke languages.

A parens form represents a set of containment relations. This set also defines a graph, with spaces as nodes and distinctions as links. In the parens form below, the inside space of each delineated boundary is labeled with a unique integer; variables are their own labels and by convention their own container boundaries.

```
        ( ( a) ( b) )
  0    1 2    3
```

The outermost boundary, labeled 0, is the edge of the typographical space. This boundary is conventionally not represented for text, and commonly represented as a line frame for diagrams. In the above example, the outermost boundary is implicit in the whitespace which defines the page formatting.

The example has six containment pairs. These containment relations can be more conventionally represented as a set of ordered pairs:

x CONTAINS y: {<0 1>,<1 2>,<1 3>,<2 a>,<3 b>}

Similarly, these pairs could be ordered by the is-contained-by relation:

x IS-CONTAINED-BY y: {<1 0>,<2 1>,<3 1>,<a 2>,<b 3>}

PARENS SEMANTICS

The table of correspondence between logic and parens forms follows:

Logical Operators	*Parens Configurations*
FALSE	<void>
TRUE	( )
NOT *a*	(*a*)
a OR *b*	*a b*
a AND *b*	((*a*)(*b*))
a IMPLIES *b*	(*a*) *b*
IF *a* THEN *b* ELSE *c*	(((*a*) *b*)(*a c*))
a XOR *b*	((*a*) *b*)(*a* (*b*))
a IFF *b*	(((*a*) *b*)(*a* (*b*)))

The transformation rules for Boundary logic are

(( ) A) =	***Occlusion***
((A)) = A	***Involution***
{A} {B {A}} = {A} {B}	***Pervasion***

Occlusion indicates the ending condition of a reduction, when a forms is completely erased. Involution removes redundant forms. Pervasion is the workhorse of the system.

Capital letters stand in place of any parens forms. The curly braces are *not* normal parens; they indicate the that Pervasion rule applies at any depth of nesting, as if intervening parens were transparent.

PUN NOTATION

EDIF netlists specify timed logic networks. These netlists can be transcribed without reconfiguration into a raw *Pun format*. The Pun format can then be converted directly into a network of distinctions, or it can be optimized using the Losp reduction engine.

Pun notation is the graph version of parens notation, and is necessary to express general graphs as opposed to trees. Pun notation maps one-to-one onto simple netlist specifications using solely the variary NOR gate.

By convention, in Pun notation all graph nodes have an outer container. Multiple forms sharing a space are grouped by double containment:

```
a (b) (c d)  ==>  (( a (b) (c d) ))
```

Double containment maintains the logical semantics of each form.

Majority Example

Pun forms are illustrated by a simple, one-output combinational circuit, 2/3 MAJORITY. Semantically MAJORITY returns True whenever two or three inputs out of three are positive.

The functionality and representation of the circuit can be captured using a look-up table:

a b c	*fn*	*conventional clauses*	*parens clauses*
0 0 0	0		
0 0 1	0		
0 1 0	0		
0 1 1	1	(and (not a) b c)	(a (b)(c))
1 0 0	0		
1 0 1	1	(and a (not b) c)	((a) b (c))
1 1 0	1	(and a b (not c))	((a)(b) c)
1 1 1	1	(and a b c)	((a)(b)(c))

The set of inputs is {a, b, c}. The table represents all possible combinations of values for these inputs. The function column (fn) is marked whenever the input values meet the semantic specifications, in this case, two-out-of-three inputs are positive.

A clause is a conjunction of literals. A literal is a positive or negative atom, where atoms identify inputs. Each line of the look-up table can be converted into a clause by forming the conjunction of the three variables, with zero-valued variables negated. The total function specified by ones in the function column is represented by the disjunction of all indicated clauses. This is disjunctive normal form (DNF), also know as the sum of products form (SOP).

The *complete* DNF form, often used in logic, consists of all clauses (lines in the look-up table) joined by disjunction. The reduced DNF consists of only those lines marked in the function column. The complete DNF for XOR is:

```
XOR:        a b    fn       conventional   clauses          parens  clauses

      |     0 0    0     (and (not a)(not b)) = false      ( a  b ) =
     OR     0 1    1     (and (not a)     b ) = true       ( a (b)) = ( )
      |     1 0    1     (and      a (not b)) = true       ((a) b ) = ( )
      |     1 1    0     (and      a      b ) = false      ((a)(b)) =

           -AND-
```

Complete DNF

```
      ((not (and (not a)(not b))) OR
       (and (not a) b) OR
       (and a (not b)) OR
       (not (and a b)) )
```

Parens form of complete DNF

```
      [ [((a  b ))] [( a (b))] [((a) b )] [(((a)(b)))] ]
```

The square brackets above are exactly the same as normal parens, but as *highlighted* by their shape. Here, the conjunction (AND form) of clauses is highlighted.

In the MAJORITY example, the reduced DNF for the function is:

```
    (or (and (not a) b c) (and a (not b) c) (and a b (not c)) (and a b c))
```

In parens notation:

```
    (( (a (b)(c)) ((a) b (c)) ((a)(b) c) ((a)(b)(c)) ))
```

Note that boundary disjunction is spatial collection, with the collection contained by the double boundary, ((...)).

The representation of the clausal parens form in Pun notation follows:

```
                ((majority)
                 ((main)
                  ((a unk) (b unk) (c unk))
                  ((oa ~5))
                  ((~1  ( a (b)(c))  )
                   (~2  ((a) b (c))  )
                   (~3  ((a)(b) c )  )
                   (~4  ((a)(b)(c))  )
                   (~5  ((~1 ~2 ~3 ~4))  )
                   )))
```

The initial fields identify the circuit (majority), the library module (main), the inputs ((a unk)(b unk)(c unk)), and the outputs ((oa 0)). The body of the Pun form is the functional circuit:

```
((~1  ( a (b)(c))  )
 (~2  ((a) b (c))  )
 (~3  ((a)(b) c )  )
 (~4  ((a)(b)(c))  )
 (~5  ((~1 ~2 ~3 ~4))  ))
```

Each row of the pun-body identifies a cell, a labeled logic subgraph.

```
(~<label> <parens>)
```

Cells are labeled with integers; variables with letters. Labels within parens identify links in the network. Each cell can contain an arbitrary amount of logical structure, as determined by the accompanying parens form. The cell label can also be red as the name of a node in the logic graph.

Dnets

The above pun-body can readily be converted into a distinction network, for which each labeled node identifies a set of atoms bounded by a single distinction:

```
((majority)
 ((main)
  ((a unk) (b unk) (c unk))
  ((oa ~9))
  ((~9  (~8)  )
   (~1  (a)  )
   (~2  (b)  )
   (~3  (c)  )
   (~4  (a ~2 ~3)  )
   (~5  (~1 b ~3)  )
   (~6  (~1 ~2 c)  )
   (~7  (~1 ~2 ~3)  )
   (~8  (~4 ~5 ~6 ~7)  )
   )))
```

Here each label represents a link in the distinction network. Although labels identify the dnode they are paired with, they can also be thought of as wires connecting the labeled dnode with its occurrences within other dnodes. That is, labels identify the output wires from each dnode. Multiple occurrences of a label denote fanout.

The Pun form of a distinction network is also an adjacency list for a directed acyclic graph (DAG) in the case of combinational circuits, and a directed cyclic graph in the case of sequential circuits.

Since the connectivity of a distinction network is confounded with its logical functionality, the "logic" can be seen to reside in the nodes themselves (as generalized nor gates which transform signals), or in the links (as connectivity transforms).

Extruded Parens Form

The dnodes in the Pun body of a distinction network are easily identified using the extruded parens form:

```
(( (a (b)(c)) ((a) b (c)) ((a)(b) c) ((a)(b)(c)) ))         parens


(                                                )          9
 (                                              )           8
   (        ) (         ) (        ) (         )      4    5    6    7
    a ( )( )   ( ) b ( )   ( )( ) c   ( )( )( )     2 3  1 3  1 2  1 2 3
       b  c     a     c     a  b       a  b  c
```

Numbers in the right-hand tree refer to cells in the above circuit. The redundancy of numbered containers with single variables {1,2,3} has been eliminated by structure sharing. Here, literals are unique.

Parens-Pun Form

The following distinction network, generated by Pun reduction, is a functionally equivalent representation of MAJORITY:

```
((oa ~4))
((~1  (a b)  )
 (~2  (a c)  )
 (~3  (b c)  )
 (~4  (~1 ~2 ~3)  ))
```

Below, the Pun-body is expressed as a single parens form, the *parens-pun form*.

```
((majority)
 ((main)
  ((a unk) (b unk) (c unk))
  ((oa ~1))
  ((~1  ((a b) (a c) (b c))  )  )))
```

This representation obviates the need for a circuitry frame, and can be expressed as a parens expression directly

```
((a b) (a c) (b c))
```

Interpreted for logic, this reads:

```
((a OR b) AND (a OR c) AND (b OR c))
```

An equivalent parens expression is more deeply nested:

```
(( ((b) (a c)) ((c) (a b)) ))
```

Interpreted for logic, this reads:

```
((b AND (a OR c)) OR (c AND (a OR b)))
```

For comparison, the parens-pun for this equivalent logical expression follows:

```
((majority)
 ((main)
  ((a unk) (b unk) (c unk))
  ((oa ~1))
  ((~1  ((((b) (a c)) ((c) (a b))))  ) )))
```

As a dnet, this Pun-body is:

```
((oa ~8))
((~8  (~7)  )
 (~1  (b)  )
 (~2  (c)  )
 (~3  (a b)  )
 (~4  (a c)  )
 (~5  (~1 ~4)  )
 (~6  (~2 ~3)  )
 (~7  (~5 ~6)  ))
```

MAJORITY in the One-Bit Adder

Yet another form of the majority circuit is presented in the chart of spatial representations defined by the function look-up table.

```
(( ((a)(b)) ((a)(c)) ((b)(c)) ))
```

This can be read for logic as

```
((a AND b) OR (a AND c) OR (b AND c))
```

The Pun circuit would be:

```
((majority)
 ((main)
  ((a unk) (b unk) (c unk))
  ((oa ~4))
  ((~1  ((a)(b))  )
   (~2  ((a)(c))  )
   (~3  ((b)(c))  )
   (~4  ((~1 ~2 ~3))  ) )))
```

The accompanying dnet body is:

```
  ((oa ~8))
  ((~1  (a)  )
   (~2  (b)  )
   (~3  (c)  )
   (~4  (~1 ~2)  )
   (~5  (~1 ~3)  )
   (~6  (~2 ~3)  )
   (~7  (~4 ~5 ~6)  )
   (~8  (~7)  ))
```

PATTERNS

One advantage of Pun notation is that repetitive logic (such as bit-width operations and cascaded operators) becomes clearly visible as aligned boundary forms. The standard canonical form of a parens expression can be constructed solely by sorting the parens expression by depth and breadth of nesting, while sorting node labels alphanumerically. Below some patterns from the DMC54 circuit are exhibited as examples of ILOC structural organization.

DMC54 Patterns

Example input structures:

NOR2:

```
(~0002  (ai cj)   )
(~0003  (ck cl)   )
(~0004  (cm cn)   )
(~0005  (co cp)   )
(~0006  (cq cr)   )
(~0007  (cs ct)   )
(~0008  (cu cv)   )
(~0009  (cw cx)   )
```

AND3B (not part of the TSMC library):

```
   (~3597  (cn (cm) (co))   )
   (~3598  (cp (co) (cq))   )
   (~3599  (cr (cq) (cs))   )
   (~3600  (ct (cs) (cu))   )
   (~3601  (cv (cu) (cw))   )
   (~3602  (cx (cw) (cy))   )
```

AOI2BB1 with one node fanout:

```
   (~0768  (~0564 (aw ~3815))   )
   (~0769  (~0565 (ax ~3815))   )
   (~0770  (~0566 (ay ~3815))   )
   (~0771  (~0567 (az ~3815))   )
   (~0772  (~0568 (ba ~3815))   )
   (~0773  (~0569 (bb ~3815))   )
   (~0774  (~0570 (bc ~3815))   )
   (~0775  (~0571 (bd ~3815))   )
   (~0776  (~0572 (be ~3815))   )
   (~0777  (~0573 (bf ~3815))   )
   (~0778  (~0574 (bg ~3815))   )
   (~0779  (~0575 (bh ~3815))   )
   (~0780  (~0576 (bi ~3815))   )
```

Sequential MUX structure with three single node fanouts (non-TSMC)

```
   (~3632  (~3815 ((af) (~3039)) ((ag) (~0217)))   )
   (~3633  (~3815 ((ag) (~3039)) ((ah) (~0217)))   )
   (~3634  (~3815 ((ah) (~3039)) ((ai) (~0217)))   )
   (~3635  (~3815 ((ai) (~3039)) ((aj) (~0217)))   )
   (~3636  (~3815 ((aj) (~3039)) ((ak) (~0217)))   )
   (~3637  (~3815 ((ak) (~3039)) ((al) (~0217)))   )
   (~3638  (~3815 ((an) (~3039)) ((ao) (~0217)))   )
   (~3639  (~3815 ((ao) (~3039)) ((ap) (~0217)))   )
   (~3640  (~3815 ((ap) (~3039)) ((aq) (~0217)))   )
   (~3641  (~3815 ((aq) (~3039)) ((ar) (~0217)))   )
   (~3642  (~3815 ((ar) (~3039)) ((as) (~0217)))   )
   (~3643  (~3815 ((as) (~3039)) ((at) (~0217)))   )
```

XOR of ORs (non-TSMC)

```
   (~2227  ((~1824 ~2770 ~3334 ~3802) ((~1824 ~3334) (~2770 ~3802)))  )
   (~2249  ((~2136 ~2926 ~3519 ~3555) ((~2136 ~3519) (~2926 ~3555)))  )
   (~2261  ((~2141 ~2915 ~3520 ~3524) ((~2141 ~3524) (~2915 ~3520)))  )
   (~2244  ((~2559 ~2565 ~2796 ~3037) ((~2559 ~2565) (~2796 ~3037)))  )
```