REPRESENTATIONS OF BOUNDARY LOGIC

William Bricken
August 2001

CONTENTS

LOSP/PUN

Losp/Pun 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* (Pun) manipulates circuitry-specific transformations on structure, such as 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.

Pun 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. Pun identifies common gate patterns (XOR, MUX), common structures which can be shared by increasing fan-out (structure sharing), and common patterns found in technology libraries (technology mapping).

The *Circuit Design Engine* (CDE) aka *Circuit Design Explorer*, is a stand-alone application of Losp/Pun 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. Several are briefly described:

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

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

Pun notation is native to Pun, 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.

LISTING OF REPRESENTATIONAL FORMS

Each of these representations is mentioned within the sequel:

Conventional

• logic networks	(combinational circuits)
• timed logic networks	(sequential circuits)
• logic look-up tables	(FPGA circuits)
• clausal form	(2level circuits)
• adjacency list	
• adjacency matrix	
• EDIF	(circuit netlists)

Linear Boundary (list-based)

- clausal parens form
- deep parens form (Implicate Normal Form, INF)
- Losp-reduced parens form

  - parens bit-stream
  - parens form
  - containment pairs

Spatial Boundary (graph-based)

- extruded parens
- pun form (partitioned graph)
- distinction network (dnet)
- homogeneous dnet
- parens-pun form (parens network)
- cnf-pun form
- occlusion array

Throughput this paper, the 2-out-of-3 MAJORITY circuit is used as an example.

SIX REPRESENTATIONS OF A CIRCUIT

Below are six different representations of the same simple circuit, a one-bit full adder. Note that the carry output is the 2/3 MAJORITY function. The standard logic and standard circuit representations convey little information visually. The four boundary representations (parens, dnet, boundary, and circuit) illustrate the visual information inherent in spatial languages.

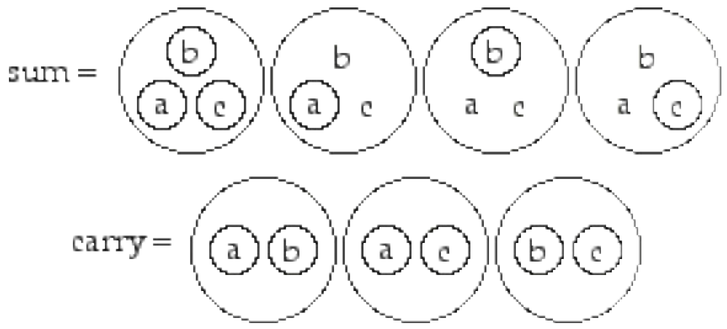
Standard Logic

$$\text{sum} = abc + a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c$$
$$\text{carry} = ab + ac + bc$$

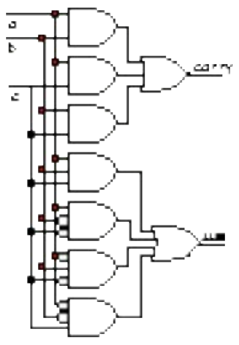
Parens Form

sum = ((a)(b)(c)) ((a)bc) (a(b)c) (ab(c))
carry = ((a)(b)) ((a)(c)) ((b)(c))

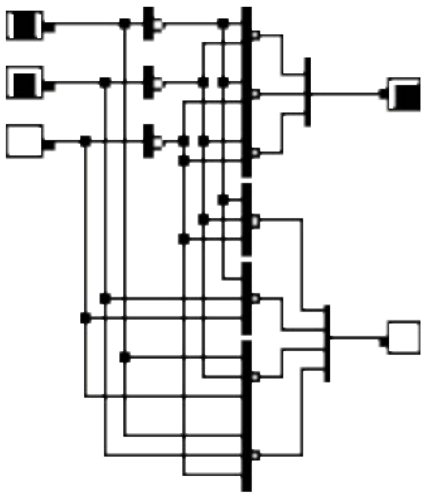
Encircling Boundaries



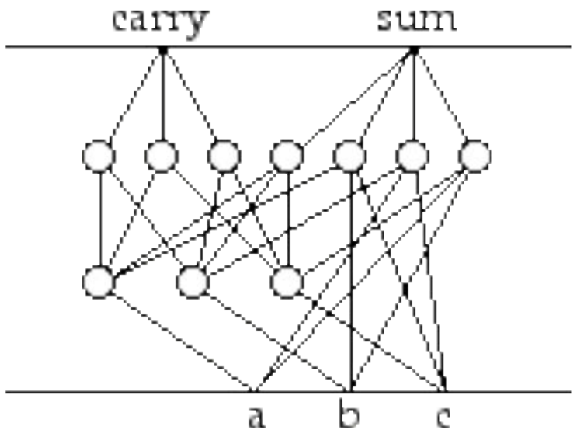
Circuit Schematic



Distinction Circuit



Distinction Network



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:
                                   |
                                 ( )
                                /   \
                              ( )   ( )
                               |   /   \
                               a  a     b
```

For notational convenience, we can use labels for the internal tree nodes, placing the actual distinction in back of the label. 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 tree clockwise by 90 degrees yields a circuit schematic-like representation:

```
      DNODE TREE:

                              a----2|---
                                        \
                              a---       1|---
                                  \     /
                                   3|---
                                  /
                              b---
```

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

```
      DISTINCTION NETWORK:

                                  2|---
                                 /     \
                              a--       1|---
                                 \     /
                                  3|---
                                 /
                              b--
```

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 often 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:          {<a 2>,<b 3>,<2 1>,<3 1>,<1 0>}
```

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 represent fan-out of nodes. Pun notation maps one-to-one onto simple netlist specifications using solely the variary NOR gate.

By convention, in pun notation all parens forms 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).

In this example, the DNF for the MAJORITY 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-form>)
```

Cells are labeled with integers; variables with letters. Labels within parens forms identify links in the network. Each cell can contain an arbitrary amount of logical structure, as determined by the accompanying parens form.

Distinction Networks

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 parens:

```
((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 fan-out.

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 Pun 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 Losp 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 *pun-parens 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 earlier.

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

This can be read for logic as

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

This form perhaps has a more natural semantic reading. 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)  ))
```

ADJACENCY LIST MODEL

When represented in pun notation, a distinction network has the same form as an adjacency list. The adjacency list representation evaluates by propagating grounds via substitution. The majority pun form:

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

Evaluation

For the input binding {(a 1),(b 0),(c 0)}, which is {(a ()),(b),(c)} in parens notation, we begin by substitution

```
((oa ~4))
((~1  (()  )  )
 (~2  (()  )  )
 (~3  (    )  )
 (~4  (~1 ~2 ~3)  ))
```

Cells with (()) are immediately deleted. Note that deleted cell labels need to be cleaned up using void-substitution.

```
((oa ~4))
((~3  (  )  )
 (~4  (~3)  ))
```

Cells with () as substituted again, and then deleted.

```
((oa ~4))
((~4  (())  )
```

The remaining cell is substituted, leaving the explicit output form:

```
((oa (())  ))
```

Another reduction generates the implicit output form:

```
((oa <void>))        which is logical False
```

RELATIONAL MODEL

Containment relations can be expressed as ordered pairs. The specific ordering indicates the inside/outside property of containers. A distinction graph is a set of adjacency relations, which can be expressed as a set of ordered pairs. Since it is conventional to evaluate a circuit from input to output, the containment pairs are expressed by the is-contained-by relation:

<contents container>	container model
<inside outside>	boundary model
<input output>	circuit model
<antecedent consequent>	logic model

The relational representation of the parens data structure ((a)(b)) is:

```
{<b 3>,<a 2>,<3 1>,<2 1>,<1 0>}
```

The MAJORITY example as a set of ordered pairs follows:

```
        ( ( a b) ( a c) ( b c))
      0   1 2      3      4

{<a 2>,<a 3>,<b 2>,<b 4>,<c 3>,<c 4>,<2 1>,<3 1>,<4 1>,<1 0>}
```

This too is a data structure which supports general computation via transversal rules. It represents a Boolean form, but with all of the logic apparently stripped away.

Evaluation

The functional reading of the relational model provides an evaluation technique. To evaluate a relational structure, variable bindings are first substituted into the set of relational pairs which defines a given logical functionality. Then the following transformation rules apply:

```
RULE 1:  delete all singletons <x> at initialization.

RULE 2:  for each <x y> with no z such that <z x>,
            delete <x y> and
            delete <y w> for all y bound by <x y>.
```

The termination conditions are:

{ } = F	all pairs deleted
{<x 0>} = T	one output pair remains

Here is the MAJORITY example:

```
{<a 2>,<a 3>,<b 2>,<b 4>,<c 3>,<c 4>,<2 1>,<3 1>,<4 1>,<1 0>}
```

Substituting bindings {(a 1),(b 0),(c 0)}, which is {(a ()),(b),(c)} in parens notation, the relational representation of MAJORITY becomes

```
{<() 2>,<() 3>,<2>,<4>,<3>,<4>,<2 1>,<3 1>,<4 1>,<1 0>}
```

Rule 1 applies only once at initialization. It deletes the singletons {<2>,<3>,<4>}, leaving:

```
{<() 2>,<() 3>,<2 1>,<3 1>,<4 1>,<1 0>}
```

Rule 2 applies repetitively until termination.

The first application of Rule 2 identifies {(),4} as values of x which meet the constraint that no pair <z x> exists. The corresponding values of y are {1,2,3}. Rule 2 deletes the following pairs directly, via delete <x y>:

```
{<() 2>,<() 3>,<4 1>}
```

The following pairs are deleted indirectly, via delete <y w>:

```
{<2 1>,<3 1>,<1 0>}
```

This deletes all pairs. What remains is the termination condition:

```
{ } = F
```

Consider an alternative binding {(a ()),(b ()),(c)}. Substituting this yields

```
{<() 2>,<() 3>,<() 2>,<() 4>,<3>,<4>,<2 1>,<3 1>,<4 1>,<1 0>}
```

Rule 1 deletes singletons {<3>,<4>}, which leaves

```
{<() 2>,<() 3>,<() 2>,<() 4>,<2 1>,<3 1>,<4 1>,<1 0>}
```

Rule 2 identifies x = {()} and y = {2,3,4} resulting in two sets of deletions

```
{<() 2>,<() 3>,<() 2>,<() 4>}             direct

{<2 1>,<3 1>,<4 1>}                       indirect
```

Removing these pairs leaves the termination condition

```
{<1 0>}
```

which is interpreted as True.

OBJECT-ORIENTED MODEL

Transformation of graph and tree based representations can be achieved through object-oriented message-passing. Each dnode is an autonomous object, potentially executing on a different resource. Dnode objects have two properties: the list of upper connections (uppers), and a list of lower connections (lowers). Naturally, all dnodes are identical but for the bindings of these properties.

Messages

Dnodes respond to three messages: INITIALIZE, DELETE and DISCONNECT

At initialization, each dnode is sent the INITIALIZE message, which is an instruction to the dnode to count the number of lowers it has.

When (lowers = none), the dnode sends a DELETE message to all uppers, and then deletes itself by sending itself a DELETE message.

When a dnode receives a DELETE message, it sends a DISCONNECT message to all its uppers, (optionally) a DELETE to all of its other lowers, and then a DELETE to itself. The optional DELETE sent to all other lowers is a version of garbage-collection.

When a dnode receives a DISCONNECT message, it removes that lower from its list of lowers and then checks for presence of lowers. If there are none, it sends DELETE to its uppers, and the messaging cycle repeats.

```
                                  DISCONNECT
       |                              |
      (1)         ==>              DELETE            ==>       <void>
     /   \                          /    \
    A    (2)                    DELETE   DELETE
```

Node 2 initiates a DELETE message to node 1. Node 1 responds by sending a DISCONNECT upwards and DELETEs downward and to itself.

DELETE can be achieved without reconfiguration of the structural model by permitting messages to pass through "deleted" nodes, as if invisible.

This procedure has been demonstrated to be strongly parallel (using a course grained 16-node hypercube), without contention, deadlock or thrashing. The paper Distinction Networks also addresses this technique.

BIT-STREAM MODEL

Consider the majority circuit with the bindings

```
      {(a 1),(b 0),(c 0)}                {(a ()),(b  ),(c  )}
```

Substituting into the parens form:

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

                  ((()  ) (()  ) (    ))
```

If the parens were a bit stream, with the encoding)=0 and (=1, we would have

```
            ((())(())())  =  111001100100
```

Naturally the parity of WFP is conserved in a 0/1 parity of the bit string.

The bit-stream reduction pattern for occlusion is

```
            1x10y0  =  <void>
```

where x and y are arbitrary collections of WFPs, including none.

Should this reduction pattern fail to occur at the top level, then the reduction algorithm is applied recursively to each bounded form, i.e. to each

```
            1z0.
```

Below, the extruded representations indicate containment relations and the processing sequence for bit-stream parens:

```
                  ((()())((())))   =  11101001110000

                  (            )      1            0
                   (    )(    )        1    01    0
                    ()()  (  )          1010  1  0
                           ()                  10
```

Evaluation of a bit-stream WFP can be achieved with a single-pass algorithm and a stack. The algorithm counts the depth of recursive descent. When an excess 0 is read in, the count is decremented by 1.

A paper Bit-Stream Evaluation also addresses this content.

HARDWIRED LOGIC MODEL

Distinction networks can be actualized as circuits with only inverters, using the technique of wiring connections together to achieve the semantics of logical OR. Rather than generalized NOR gates, incoming wires are simply joined, forming a *wire-or*. The dnode itself is then a simple inverter.

```
    -----=|1 --
   /    /      \
  /    a        \
 /      \        \
b        =|2------=|0----out
 \      /        /
  \    c        /
   \    \      /
    -----=|3 --
```

By rearranging the inverters to lay along a line, the wiring of the circuit becomes a path which crosses an *inverter bar* many times.

Although it is possible to design circuitry which would implement void-based logic rather than traditional logic, that would probably require a significant redesign of hardware components and manufacturing methodologies, and would also possibly surrender backward-compatibility.

The occlusion array algorithm which follows uses conventional technology to implement a non-conventional algorithm. This algorithm could also run with reduced efficiency in traditional memory arrays manipulated by machine instructions, rather than in specialized hardware.

OCCLUSION ARRAY MODEL

The occlusion algorithm can be implemented using hardware as a generic substrate. The hardware substrate itself would be implemented using conventional signal propagation circuitry. The substrate wiring would be specific to the occlusion algorithm, and plays a role equivalent to the vonNeumann architecture for software.

An *occlusion array* for a given circuit functionality is a spatial display of distinctions, each distinction corresponding to one dnode in a distinction network. This array is a binary-valued rank 2 square matrix. Formally, an occlusion array is the relation matrix for a set of containment pairs, and the adjacency matrix for a distinction network.

The significant difference between an occlusion array and a matrix is that the array has logical semantics, it can be used to evaluate logical expressions and circuits. The configuration of distinctions in an occlusion array mirrors the topology of a distinction network. This configuration acts to steer the propagation of values through the array, analogous to the propagation of values through a logical circuit. In contrast, matrices are usually formulated for numerical computation. The conventional matrix addition and multiplication operations do not apply to occlusion arrays.

Occlusion arrays are void-based, thus their behavior is unary rather than binary. Only positive output values are asserted as output, negative output values occur by default, in the absence of an asserted positive value. The termination condition is independent of value assertion, so a propagated value (either negative or positive) is not necessary to terminate evaluation.

Progress occurs by masking, or eliminating, dnodes (array rows). When all rows are masked, computation is over. As specific dnodes are masked, the transparency of the output vector might change. At termination, any particular output either remains transparent (i.e. has a 0 value) or has become opaque (i.e. has a 1 value).

Combinational   Components

```
          input           circuit          masks   filters    screen
                      -1-2-3-4-5-6-7
           ---       ----------------      -----    -----      ---
      1    | |       |              |      | | |    | | |      | |
      2    | |       |              |      | | |    | | |      | |
      3    | |       |              |      | | |    | | |      | |
      4    | |       |              |      | | |    | | |      | |
      5    | |       |              |      | | |    | | |      | |
      6    | |       |              |      | | |    | | |      | |
      7    | |       |              |      | | |    | | |      | |
           ---       ----------------      -----    -----      ---

            I                 C             M A      N U        O
```

• Input	I
• Occlusion array	C
• Mask	M
• Active	A
• Input filter	N
• Output filter	U
• Output screen	O

MAJORITY Example

Here the majority circuit is expressed as a pun body.

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

The occlusion array is

```
                 OUT
              -a-b-c-1-2-3-4
             ----------------
          a  |       1 1    |  in
          b  |       1   1  |  in
      I   c  |         1 1  |  in
      N   1  |             1|
          2  |             1|
          3  |             1|
          4  |              |  out
             ----------------
```

The configuration of marks within the array can represent any of the forms of MAJORITY in this paper. Here are some of them:

```
    -a-b-c-1-2-3-4-5-6-7-8-9
   --------------------------
a  |       1     1          |  in
b  |         1     1        |  in
c  |           1     1      |  in
1  |               1 1 1    |
2  |             1   1 1    |
3  |             1 1   1    |
4  |                     1  |
5  |                     1  |
6  |                     1  |
7  |                     1  |
8  |                       1|
9  |                        |  out
   --------------------------
```

```
Two-level version:

((~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)  ))
```

```
    -a-b-c-1-2-3-4-5-6-7-8
   ------------------------
a  |           1 1        |  in
b  |       1   1          |  in
c  |         1   1        |  in
1  |               1      |
2  |                 1    |
3  |                 1    |
4  |               1      |
5  |                   1  |
6  |                   1  |
7  |                     1|
8  |                      |  out
   ------------------------
```

```
Asymmetric version:

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

```
    -a-b-c-1-2-3-4-5-6-7-8
   ------------------------
a  |       1              |  in
b  |         1            |  in
c  |           1          |  in
1  |             1 1      |
2  |             1   1    |
3  |               1 1    |
4  |                   1  |
5  |                   1  |
6  |                   1  |
7  |                     1|
8  |                      |  out
   ------------------------
```

```
One-bit adder version:

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