# ASPECTS OF BOUNDARY LOGIC

William Bricken
December 2001

I've enclosed some supplementary notes and examples to help to describe the BTC intellectual property and the Boundary Logic (BL) foundations for these tools. The small examples emphasize comparisons between iconic and traditional approaches to problems presented in *Contemporary Logic Design* (CLD).

## The Nature of Boundary Logic

Boundary Logic is a calculus of containers (mathematically, partially ordered lattices). A container can be interpreted as a conventional NOR gate with an arbitrary number of inputs. Although BL therefore can be *read* as conventional circuitry, to do so obscures the powerful transformations which arise only from a container-based interpretation. Formally, BL is more succinct than logic, the mapping from logical forms to containers is many-to-one.

BL is utterly simple, once a few perspective changes have been accepted. These are simply stated:

- Rather than 1 and 0, BL uses *presence* and *absence*. Absence permits half of a binary system to simply not exist. Note the similarity to switching networks where absence of a connection has meaning.

- To manipulate presence and absence, basic objects require an *inside* and an *outside*, thus containers. Conventional variables lack an inside while conventional pointers lack an outside.

- Textually, a parenthesis (called ***parens***) is a container. The following parens contains the objects A and B: (A B). This parens contains nothing: ( ). All combinational circuit structures are simply well-balanced parens. Sequential registers with reentry undermine specific inside/outside distinctions in a principled manner.

- Containers are spatial objects which *do not possess* ordering, grouping, or cardinality properties (commutativity, associativity, arity).

- Parens forms are transformed and simplified by *deletion* (erasure) operations rather than by rearrangement.

- Transformational rules rely on the *transparency* property of parens: everything inside a nest of parens containers is transparent to the outside. This permits multilevel operations independent of path length.

Note that nesting of parens is equivalent to multiple levels of logic. In general, well-formed parens are trees which are graphs, so that the parens form is also a succinct representation of a logic graph or network.

## The Map from Parens to Gates

NOR is containment:

| | |
|---|---|
| NOR[a, b, c] | (a b c) |

From the single mapping of NOR to parens, the parens form of other simple gates is easily deduced. The NOR of a single argument is NOT:

| | |
|---|---|
| NOT[a] | (a) |

The NOR of no arguments is TRUE:

| | |
|---|---|
| TRUE | ( ) |

Since a parens negates its inside, the TRUE empty parens is the negation of nothing. That is, FALSE has no representation, it is absence:

FALSE

OR is sharing space:

| | |
|---|---|
| OR[a, b, c] | a b c |

In BL, forms are *disjoint* in that space provides no connective or relational structure. Sharing space is defined solely as being the contents of the same container. Forms sharing the same space can be processed in parallel. Finally, AND is formed by containing a space and every object within that space:

| | |
|---|---|
| AND[a, b, c] | ((a)(b)(c)) |

Therefore any combinational form can be expressed as a well-formed parens form. It is illustrative to transform DeMorgan into parens notation:

AND[a, b] = NOT[ OR[ NOT[a],NOT[b] ]]

((a)(b)) = ((a)(b))

Other examples:

| | |
|---|---|
| XNOR[a,b] | (a b) ((a)(b)) |
| 2/3MAJORITY[a,b,c] | ((a b)(a c)(b c)) |
| HALF-ADDER[a,b] | Sum = ((a b) ((a)(b)))<br>Carry = ((a)(b)) |
| FULL-ADDER[a,b,ci] | Sum = ((hsum ci) ((hsum)(ci)))<br>Carry = ((a)(b)) ((hsum)(ci))<br>hsum = ((a b) ((a)(b))) |

BL is an algebraic system, function composition is achieved by substitution. Concepts such as "don't cares", incomplete specification, and Boolean cubes are irrelevant to the algebraic process. I'll return to the full-adder after introducing the transformation rules.

*All* of the mechanism for two-level and multilevel Boolean simplification reduces to three deletion equations:

| | | | |
|---|---|---|---|
| (A ( )) | = | <void> | Occlusion |
| ((A)) | = | A | Involution |
| A {A B} | = | A {B} | Pervasion |

Capital letters refer to any form regardless of complexity. Each equation proceeds from left to right via deletion of structure. The curly braces in Pervasion convey the transparency rule, they stand in place of any intervening content at the same or a deeper level of nesting. Pervasion is thus a multilevel simplification rule, some examples:

a (a b) = a (b)

a (b (c (a d))) = a (b (c (d)))

(a b) (c (d (e (a b))) = (a b) (c (d (e)))

For illustration, here is the BL approach to a Boolean simplification problem in CLD (2.13e, p.105). Spacing is used to emphasize which structures have been deleted:

```
f[w,x,y,z] = x + xyz + x'yz + x'y + wx + w'x

x ((x)(y)(z)) (((x))(y)(z)) (((x))(y)) ((w)(x)) (((w))(x))  Transcribe
x ((x)(y)(z)) (  x  (y)(z)) (  x  (y)) ((w)(x)) (  w  (x))  Involution
x (( )(y)(z)) (     (y)(z)) (     (y)) ((w)( )) (  w  ( ))  Pervasion
x             (     (y)(z)) (     (y))                      Occlusion
x             (     (y)(z))        y                        Involution
x             (     ( )(z))        y                        Pervasion
x                                  y                        Occlusion

f = x + y                                                   Transcribe
```

**Figure I** and **Figure II** provide circuit schematic views of two BL multilevel simplification steps. Each schematic identifies a generic pattern and a deletion of wires and/or gates which maintains functional equivalence. Note that a schematic which maps to a parens representation consists only of NOR gates. Unused pins in each schematic represent arbitrary inputs.

Returning to the full-adder,

```
Sum   = ((hsum ci) ((hsum)(ci)))
Carry = ((a)(b)) ((hsum)(ci))
hsum  = ((a b) ((a)(b)))
```

I'll first substitute the form of hsum for the label "hsum", then reduce the resulting form:

```
Sum = ((((a b) ((a)(b))) ci) ((((a b) ((a)(b))))(ci)))
      ((((a b) ((a)(b))) ci) (  (a b) ((a)(b))  (ci)))      Involution

Carry = ((a)(b)) ((((a b) ((a)(b))))(ci))
        ((a)(b)) (  (a b) ((a)(b))  (ci))                   Involution
        ((a)(b)) (  (a b)           (ci))                   Pervasion
```

## Graph Form

To achieve network structure sharing and technology mapping, a slightly more elaborate notation allows labels to be associated with any parens form. Lists keep track of i/o and internal components. The list delimiters are square brackets to avoid confusion with BL parens. The graph form permits decomposition and rearrangement of a logic network into nodes identified solely by parens shapes. As an example, the above form of the full-adder is shown first, then successive decompositions are illustrated.

```
[full-adder1 [[in1 a][in2 b][in3 ci]] [[sum 1][carry 2]]
 [[1   ((((a b) ((a)(b))) ci) ((a b) ((a)(b)) (ci)))   ]
  [2   ((a)(b)) ((a b)(ci))                            ]   ]]
```

Identifying and separately labeling repetitive parens patterns achieves structure sharing:

```
[full-adder2 [[in1 a][in2 b][in3 ci]] [[sum 1][carry 2]]
 [[1   (((3 4) ci) (3 4 (ci)))   ]
  [2   4 (3 (ci))                ]
  [3   (a b)                     ]
  [4   ((a)(b))                  ]   ]]
```

Separating inverters reduces the complexity of each node, and is equivalent to conventional bubble notation for gates:

```
[full-adder3 [[in1 a][in2 b][in3 ci]] [[sum 1][carry 2]]
 [[1   ((7 ci) (5 6))   ]
  [2   4 (3 6)          ]
  [3   (a b)            ]
  [4   ((a)(b))         ]
  [5   3 4              ]
  [6   (ci)             ]
  [7   (5)              ]   ]]
```

The circuit is next technology mapped to simple 2-input gates by decomposing parens patterns:

```
[full-adder4 [[in1 a][in2 b][in3 ci]] [[sum 1][carry 2]]
 [[1   (8 9)      ]                                      NOR
  [2   4 10       ]                                      OR
  [3   (a b)      ]                                      NOR
  [4   ((a)(b))   ]                                      AND
  [5   3 4        ]                                      OR
  [6   (ci)       ]                                      NOT
  [7   (5)        ]                                      NOT
  [8   (7 ci)     ]                                      NOR
  [9   (5 6)      ]                                      NOR
  [10  (3 6)      ]   ]]                                 NOR
```

Alternative mappings are visually apparent and are achieved by simple algebraic matching and substitution.  The misII full-adder in Figure 3.22 of CLD has the following parens form in which the parens pattern of each MSU gate primitive is easily recognized:

```
[full-adder5 [[in1 a][in2 b][in3 ci]] [[sum 1][carry 2]]
 [[1   ((a 3) ((a)(3)))     ]                                2310
  [2   ((4)) ((b)(ci))      ]                                1890
  [3   ((b ci) ((b)(ci)))   ]                                2310
  [4   (a) (b ci)           ]                                1890
  [5   (b)                  ]                                1310
  [6   (ci)                 ]   ]]                           1310
```

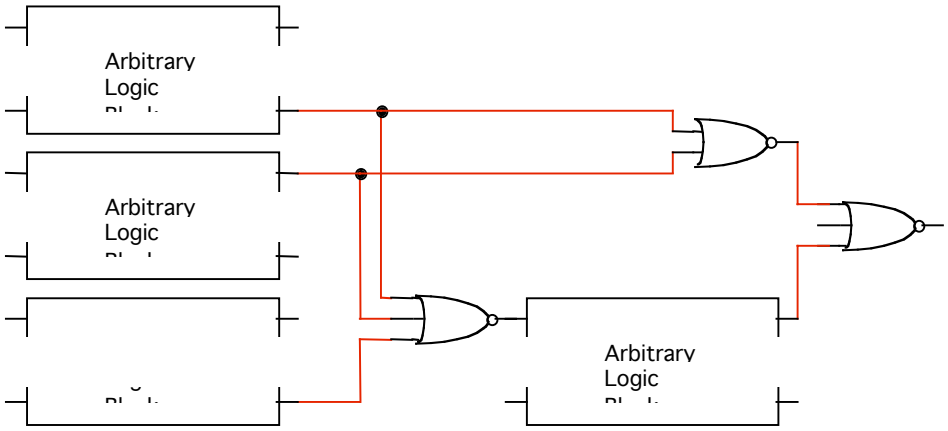
Surprisingly, all BL algorithms are low degree polynomial, basically sorting and pattern-matching. Yes, all interesting Boolean transformations are still intractable, however BL removes sufficient complexity that only rare pathological forms require the extra effort. The BTC algorithms produce world-class minimization without exponential processes. BL is a tractable multilevel calculus for Boolean minimization.

## Developed Hardware

We have developed several architectures which take advantage of BL forms. The *computational mesh* is essentially a ROM which can be wired for multilevel logic. The *BILD architecture* is dynamically reconfigurable, using FlashRAM cells for function specification and a small engine which evaluates the spatial configuration in memory. This architecture has fully predictable wiring and process delays. Parens functions expressed as bit-streams permit an extremely efficient single-pass simulator using standard CPUs. All architectures are blind to circuit structure and complexity, accommodating sequential as well as combinational forms.

Figure I: Subsumption

BEFORE




AFTER

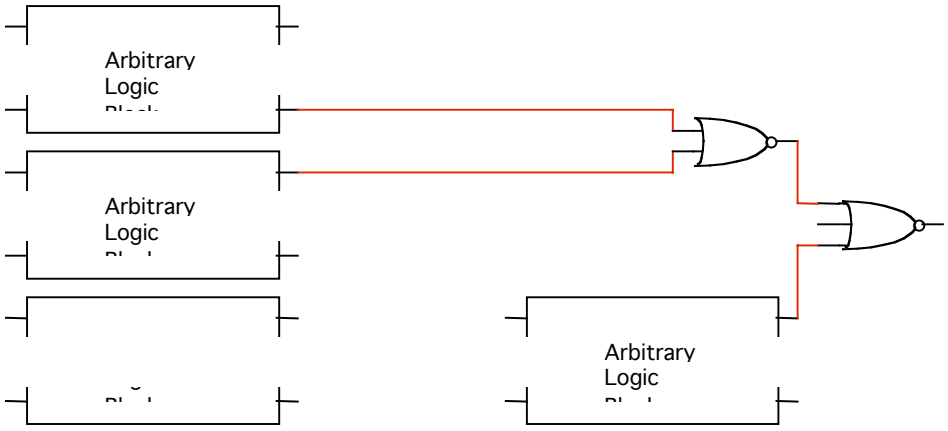
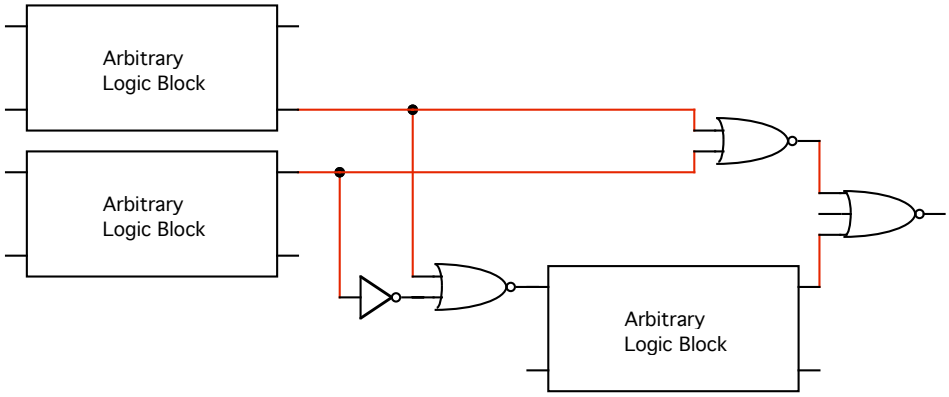


Figure II: Cancellation

BEFORE




AFTER