Bricken™ Technologies

# Bricken Technologies FAQ

**DRAFT**

Version 5.4

# Table of Contents

# 0. Introduction

This FAQ is being developed to provide in a single document answers to questions that arise from diverse audiences: investors and potential investors, consultants, contractors, advisors, recruiters, potential employees, and potential strategic partners to mention just a few. Since this FAQ addresses the interests of a diverse audience, a given reader may only be interested in certain topics. Thus the detailed Table of Contents is provided as a navigation tool.

Also, in a document such as this, it is not feasible to provide in-depth discussions of certain topics, especially those relating to Boundary Mathematics and to the details of BTC's software and hardware technologies. We expect and look forward to providing more detailed information appropriate to particular conversations.

The questions addressed in this draft are those that have arisen most frequently, or whose answers are necessary as background to other related topics.

# 1. Bricken Technologies Corporation

## *1.1. What is BTC?*

BTC was founded in 2001 to commercialize semiconductor and related software products created by Dr. William Bricken and acquired by the Company at its formation. These innovations are based on Boundary Mathematics, which is described elsewhere in this document.

## *1.2. Who is William Bricken?*

William Bricken received his Ph.D. from Stanford University. Among other accomplishments he was an early pioneer in virtual reality systems. He has been evolving boundary mathematics and applying this approach to difficult logic and computational problems, including the design and optimization of digital circuits and semiconductors, for most of his career.

## *1.3. What is Iconic™ Logic?*

Iconic Logic is BTC's trade name for "boundary logic," or "void-based logic," a system of logic which is simpler than conventional approaches to logic. An *icon* resembles what it refers to; therefore Iconic Logic is a visual presentation of the meaning of logic. The data structures and algorithms of Iconic Logic are much simpler than those of conventional logic because Iconic Logic has no explicit representation of the logical concepts of `FALSE` and `OR`.

BTC's hardware and software products are based on Iconic Logic. For the sake of accuracy and consistency, in most of this document we refer to "void-based logic" instead of Iconic Logic.

## *1.4. What are BTC's products?*

BTC is bringing to market a new kind of programmable/reconfigurable logic device and the software application required to program the chip.

### *1.5. What problems do BTC products solve for customers?*

The Company's reconfigurable products will not require the often extensive retiming of designs that often arises in using Field Programmable Gate Arrays (FPGAs) nor concern with floorplanning, placement, and routing. Other benefits are discussed in conjunction with specific product architectures or capabilities.

The optimizing compiler will optimize the circuit, that is, find an equivalent circuit that requires fewer resources (gates, wires, delay) and then compile the circuit specification into the form that programs the functionality of BTC's reconfigurable chip.

### *1.6. Markets*

The principal early markets for BTC hardware products are communications, computing, and consumer electronics.

### *1.7. Business Model*

BTC will operate as a fabless semiconductor company whose principal source of revenue will be from sale of hardware products. The Company may also choose to generate licensing revenue from software, semiconductor designs, and from other intellectual property assets.

BTC expects to rely primarily on indirect sales channels, for example, distributors and manufactures representatives in North American and in other major markets such as Europe and Asia.

### *1.8. Product Status*

A first version of BTC's optimizing compiler has been completed. The Company has been running the innovative cores of its reconfigurable hardware products as software simulations for more than one year. A first software simulation of one core design has been implemented as a simulation in an FPGA.

### *1.9. Competitors*

#### 1.9.1. Hardware Competitors

BTC hardware products will compete with those of Altera, Xilinx, Lattice, Cypress, and Atmel.

#### 1.9.2. Software Competitors

BTC's software tools may provide certain functions that are currently provided by Electronic Design Automation (EDA) tools from Synopsis, Cadence, and Synplicity, among others.

### *1.10. Intellectual Property*

#### 1.10.1. Who owns Bricken's innovations?

William Bricken has secured written releases or assignments of ownership from all those who might reasonably have a claim on his innovations related to boundary mathematics, including software and designs for semiconductors. In turn, all of these assets were acquired by the company at formation or shortly thereafter.

### 1.10.2. How many patent applications has BTC filed?

BTC has filed four patent applications in the United States and intends to file additional applications in the US and commercially important foreign jurisdictions.

### 1.10.3. What are BTC's Trademarks?

BTC's trademarks include BILD, Iconic, Iconic Logic, ILOC, and Comesh.

## 2. BTC Optimizing and Compiling Software

### *2.1. What are the benefits of the BTC software?*

The benefits of BTC's logic reduction and circuit optimization software tools arise directly from the void-based logic algorithms that they implement. These benefits include:

- Improves the cost/performance of most conventional circuits by at least 10%
- Provides automated design optimization using formal methods
- Provides formal verification of design modifications
- Provides powerful mathematical transformations of circuit structure that are not currently available in commercial tools. These include
    - Boolean minimization
    - more efficient technology mapping
    - formal theorem prover

### *2.2. Circuit Optimization*

### 2.2.1. What is circuit optimization?

A given functionality can be expressed using many different but equivalent circuit structures. Optimization is finding the structure for a given functionality that best suits the architecture and resources of the execution hardware. Circuits may be optimized with respect to several factors, parameters, or characteristics, including the following:

- die area and number of transistors
- wiring and interconnect
- timing and propagation delays
- power consumption
- noise and interference

### 2.2.2. What is the Boolean Minimization problem in the context of circuit design?

Pragmatically stated: how can we eliminate unnecessary and redundant logic from a specification, and thus improve the performance of a circuit implementation.

### 2.2.3. How does void-based logic address this minimization problem?

By being computationally simpler than the conventional techniques of logic, void-based logic is able to better recognize and delete redundant logical structures. By having a more homogeneous representation, void-based logic forms are easier to rearrange to meet resource restrictions and other optimization constraints. Void-based logic is simple enough to address many difficult optimization problems that other tools cannot.

### 2.2.4. Does void-based logic solve the intractability problem?

Almost every Boolean problem is complex in that it gets exponentially more difficult as the problem size grows. These problems are called *intractable* because no computational algorithm, including void-based logic, can address them efficiently. However, in contrast to other logic techniques, void-based logic is sufficiently simple to be able to address all but the most tangled and pathological problems. Fortunately, such difficult problems do not occur in most logic problems or circuit designs.

## *2.3. Software EDA Tools*

### 2.3.1. What is the Iconic Logic™ Optimizing Compiler?

The Iconic Logic™ Optimizing Compiler (ILOC™) is software that converts a functional specification into a configuration of containment relations. This configuration is stored in BTC's reconfigurable chip, more specifically, in a memory-like array we call the Circuit Configuration Array (CCA).

### 2.3.2. What does it optimize?

Expressing functionality in void-based logic permits removal of logical redundancy and minimization of connectivity between logic components in a circuit. The Optimizer modifies the representation of logic functionality to best fit into the CCA resources of BTC hardware architectures.

### 2.3.3. What does it compile?

Compilation means converting from a specification language to a machine language. The Compiler thus converts descriptions of circuit functionality into the representation expected by the BTC hardware architectures.

### 2.3.4. In what format is the input?

Generally, any formal specification of a desired functionality is suitable for input. More specifically, input can be Hardware Definition Languages (HDLs) such as Verilog, netlists, Finite State Machines, look-up tables, or logic equations.

### 2.3.5. What format is the output?

Although output can match any form of input specification, the two formats supported initially are netlists that can be fed back into traditional EDA tools and Circuit Configuration Arrays that effectively program BTC's reconfigurable chips.

### 2.3.6. What can the user control?

Since users will not be familiar with the techniques of void-based logic optimization, the initial commercial version of the Optimizing Compiler is completely automatic. It provides an estimate of processing time for a submitted circuit that the user may decrease by setting a lower Optimization effort level.

### 2.3.7. How do I integrate the output of the Optimizing Compiler into a traditional design flow?

At any time during processing, the circuit configuration being processed by the Optimizer can be exported as a conventional netlist (EDIF format, for example). Licensing from BTC will be required for any use of optimization output other than in conjunction with BTC's reconfigurable chip products.

## 3. Hardware Architectures

BTC currently has defined a number of novel semiconductor architectures. Presently the company is evaluating two architectures for the core of its reconfigurable logic device products. These are the Bricken Iconic Logic Device (BILD™) architecture and the Computational Mesh (Comesh™) architecture. These architectures define the core of BTC's reconfigurable products, which in practice will include additional features and support for market-required standards and capabilities.

### *3.1. What are the benefits of BTC hardware?*

The benefits of the BTC hardware architectures arise directly from the void-based algorithms that they implement. These benefits include:

- Greater ease of use leads to greatly improved time-to-market.
- Reconfigurable at memory-load speeds.
- Accommodates any type of circuitry.
- Best case performance for each input vector.
- Reduced power consumption due to fewer logic transitions.
- More inexpensive manufacturing and higher fabrication yield due to homogeneous array technology.
- Backward compatibility with design methods, tool chains, and fabrication processes.
- Fully automated layout with no physical constraints due to wiring.
- Predictable timing, even with design changes.
- Flexible pin assignment that can be changed dynamically.

## *3.2. BILD™ Reconfigurable Core Architecture*

### 3.2.1. What is the BILD™ core architecture?

The BILD™ core architecture is one of several designs for a family of semiconductors. Most members of this BILD™ family are reconfigurable; that is, their functionality may be easily changed or altered.

The BILD™ architecture is founded on void-based Iconic Logic. The architecture consists of a Circuit Configuration Array (CCA), the BILD™ engine, and standard registers for masking, input, and output.

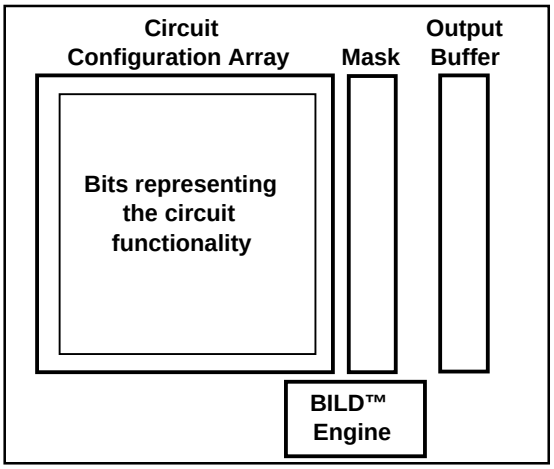


**Figure 1: BILD™ Core Architecture (not to scale)**

### 3.2.2. What is a Circuit Configuration Array?

An array of memory-cells (a homogeneous transistor array) that stores a spatial configuration of bits that represents the functionality of a circuit.

### 3.2.3. What kind of memory-cell can it use?

Standard SRAM may be desirable for low-density logic applications with embedded memory. The Company is also exploring various 1-transistor SRAM cell architectures. ROM is desirable for logic intensive applications. MaskROM is best when reconfigurability is not required. FlashROM or some other inexpensive and dense memory-cell is appropriate for most other applications. PROM, EPROM and EEPROM memory-cells are possible for specific limited applications.

### 3.2.4. Is the BILD™ memory-cell like CAM memory?

CAM stands for Content Addressable Memory. CAMs have circuitry that allows parallel matching of words in memory. BILD™ memory-cells resemble standard memory-cells with wiring for parallel access. The CCA is similar to a CAM only in that it is accessed in parallel.

### 3.2.5. What is the BILD™ engine?

The engine is a small unit of conventional logic that controls the evaluation of a CCA. Processing steps include querying the CCA configuration to determine which rows to mask, examining if the mask is full, and posting output. A masked row means that the information in the CCA for that row is no longer needed to determine the output.

### 3.2.6. How large is the BILD™ engine?

The BILD™ Engine is integrated with the CCA and the register files. It consists of a few gates associated with each CCA row, and less than 100 logic gates associated with the Engine functionality.

### 3.2.7. How does a BILD-architecture chip work?

Progress in evaluating a particular circuit occurs by masking, or eliminating, array rows from further consideration. When all rows are masked, computation is over. As specific rows are masked, appropriate output results may be posted to the output vector. The BILD™ Engine controls masking and output.

### 3.2.8. How does the BILD architecture deal with multiple clocking regimes?

Among other techniques, initial BILD™ products are expected to handle multiple clocking regimes though the partitioning of the CCA into logical blocks, each of which can be driven by a separate BILD™ Engine and each of which would have it its own associated register banks.

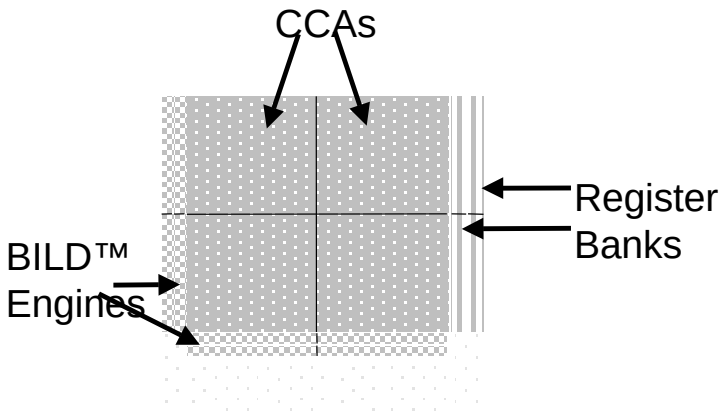


**Figure 2. Functional Block Architecture**

The partitioning of the user's circuit will be handled by the optimizing compiler, whose output will include the required configuration array information and partitioning and clock related information.

### 3.2.9. How does the BILD™ architecture chip support resources commonly found on CPLDs? On FPGAs?

BTC expects its initial BILD architecture products to include various fixed blocks that provide additional resources and capabilities, including, for example, embedded DRAM or SRAM blocks and logic blocks supporting market required functions and/or IP cores.

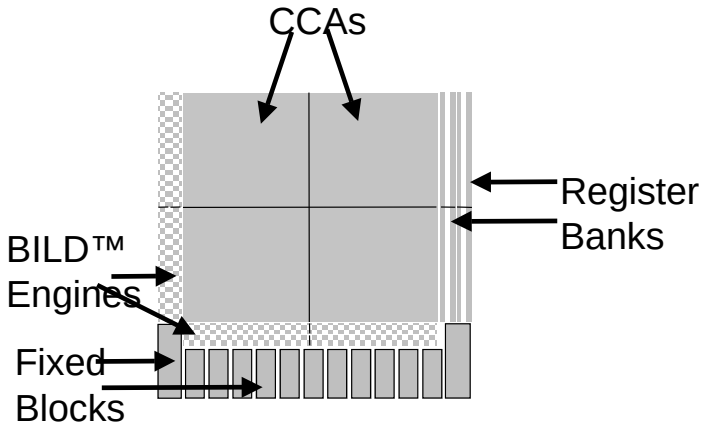


**Figure 3: Functional Block Architecture with Fixed Blocks**

Going forward, the Company will evaluate the relative merits of providing certain capabilities hardwired or in software that can be combined with the user's programming. In practice, BTC expects some of each.

### 3.2.10. What is the logic gate density of the BILD™ CCA?

Density depends of the fabrication feature-size, which we will call S. Feature-size determines the finest-grain component in a circuit. Currently, a .18 micron feature-size is common ; in a few years this is expected to decrease to .10 microns or less. Silicon area can be expressed in square millimeters ($mm^2$) to aid comprehension of micron-level sizes. There are 1000 microns in a millimeter. A .25 micron feature-size, for example, fits 4000 feature units per millimeter. A .10 micron feature size fits 10000 feature units per millimeter.

The relative *area* of circuit design components can be expressed in square feature-size units, $S^2$. For instance, the area taken up by one bit of FlashROM is about 10 $S^2$, which is a square of about 3 feature-size units on a side. Memory-cells require a storage mechanism, read and write lines, power, and a sometimes a clock signal. This silicon mechanism takes up around 3 feature-size units square for FlashROM. For comparison, an ASIC gate requires about 700 $S^2$, a square of about 26 feature-size units on a side, and 70 times larger in area than a FlashROM cell. An ASIC-equivalent gate in an FPGA logic block plus interconnect is about 30000 $S^2$, which is about 40 times larger in area than an ASIC gate, 3000 times larger than a FlashROM cell.

A table of relative logic density per $mm^2$ for different design components follows. We assign the area of an ASIC gate a relative density of 1, for comparison of logic densities. (All numbers have been rounded.)

| *Component* | *Area in $S^2$* | *.25u Components/$mm^2$* | *Relative-Density* |
|---|---|---|---|
| 1 bit FlashROM | 10 | 1600K bits | |
| 1 bit SRAM | 50 | 320K bits | |
| 1 ASIC gate | 700 | 23000 gates | 1 |
| 1 FPGA gate | 30000 | 530 gates | .02 |
| 1 BILD™ gate | 2300 | 7000 gates | .3 |
| 1 COMESH™ gate | 4300 | 4300 gates | .2 |

The following table presents two row densities, one for uncompressed square CCAs, and one for compressed CCAs that optimizes array usage. The table also includes projected gate densities per $mm^2$ and the feature size area per gate. Conservatively, we assume three logic gates per CCA row. For one square millimeter of silicon:

| | *Not-compressed* | | | *Compressed* | | |
|---|---|---|---|---|---|---|
| ***Feature*** | ***Rows*** | ***Gates*** | ***Compression*** | ***Rows*** | ***Gates*** | ***$S^2$/gate*** |
| .25 | 570 | 1700 | 4x | 2300 | 6900 | 2300 |
| .18 | 800 | 2400 | 6x | 4800 | 14400 | 2200 |
| .13 | 1100 | 3300 | 8x | 8800 | 26000 | 2300 |
| .10 | 1400 | 4200 | 10x | 14000 | 42000 | 2400 |

The relative areas of different CCA cell-level components remain somewhat constant across different feature-sizes, since all components benefit from smaller fabrication sizes. BILD™ chip density is largely determined by the memory-cell architecture of the configuration array. Our estimate of 50 $S^2$ is based on preliminary cell-level functional designs. BTC expects that over time, with design refinement, these cell areas can be significantly reduced, to around twice that of a FlashROM cell, rather than five times. This would increase the gate density of the BILD™ product by approximately 50%.

For comparison, independent of feature-size, the compressed BILD™ architecture gate density is about one-third that of a sea-of-gates ASIC, and about thirteen times more dense than an FPGA.

## *3.3. Comesh™ Reconfigurable*

### 3.3.1. What is the “computational mesh” core architecture?

Another architecture that defines a potential family of reconfigurable products is the *Computational mesh (Comesh)* architecture, which like the BILD™ architecture, is an array-based technique for the evaluation of circuit functionality. It is quite similar to the BILD™ architecture, except that rather than controlling function evaluation with an Engine, the Comesh™ architecture chip propagates signals across a memory array directly.

### 3.3.2. How does it work?

Interconnect points in the mesh store the circuit functionality and connectivity. Signals propagate from input through interconnect to output similar to an ASIC. The wiring of the Comesh™ chip is the mesh itself, while the gates of a Comesh chip are the particular active array cells.

### 3.3.3. How does it handle registers?

Each register is one Comesh™ row. Register state is posted as would be any output. On the following clock cycle, register state is fed back into the Comesh™ as input.

### 3.3.4. How does it handle multiple clocking regimes?

We expect the same techniques used to handle clocking regimes in the BILD™ architecture will be applied to Comesh-based products as required by the market.

### 3.3.5. What is the density of a Comesh™ array?

Comesh™ densities can be expressed in rows per millimeter. Comesh™ architecture accommodates about one gate per row. We assume a Comesh™ memory-cell size of two times FlashROM, around 20 $S^2$. The following table presents two row densities, one for uncompressed square arrays, and one for compressed arrays that optimizes array usage. For one square millimeter of silicon:

| | *Not-compressed* | | | *Compressed* | | |
|---|---|---|---|---|---|---|
| ***Feature*** | ***Rows*** | ***Gates*** | ***Compression*** | ***Rows*** | ***Gates*** | ***$S^2$/gate*** |
| .25 | 900 | 900 | 4x | 3600 | 3600 | 4400 |
| .18 | 1250 | 1250 | 6x | 7500 | 7500 | 4100 |
| .13 | 1700 | 1700 | 8x | 14000 | 14000 | 4200 |
| .10 | 2200 | 2200 | 10x | 22000 | 22000 | 4500 |

The BILD™ CCA achieves around twice the gate density of a Comesh™ CCA. This is offset by the Comesh™ architecture function evaluation time, which is over five times faster than the BILD™ approach. We see here a quite common design trade-off between processing speed and logic elements.

Comparatively, the Comesh™ architecture gate density is about one-sixth that of an ASIC, and about eight times more dense than an FPGA.

### 3.3.6. What kind of memory-cell arrays can be used?

Comesh™ architecture is designed for ROM cells, either hardwired, program-once, or reprogrammable.

### 3.3.7. How does decreasing feature-size effect Comesh densities?

Since memory-like hardware benefits disproportionately more than logic as feature-sizes decrease, we anticipate ever improving densities for BTC products compared to FPGA and CPLD products. Over the last decade, logic density has increased at a rate of 135% per year, while memory has increased at a rate of 160% per year. This improvement can be seen in the Semiconductor Industry Association projections for memory densities:

| *S* | *year* | *MS²/mm²* | *Mbits* | *chip area* | *Mbit/mm²* | *S²/bit* |
|---|---|---|---|---|---|---|
| .35 | 95 | 8 | 64 | 190 | .34 | 24 |
| .25 | 98 | 16 | 256 | 280 | .9 | 18 |
| .18 | 01 | 31 | 1000 | 420 | 2.4 | 13 |
| .13 | 04 | 59 | 4000 | 640 | 6. | 10 |
| .10 | 07 | 100 | 16000 | 960 | 17. | 6 |
| .07 | 10 | 204 | 64000 | 1400 | 58. | 4 |

Toshiba's recent .13u process 1 Gbit NAND FlashROM has a cell density of 5 $S^2$. With reasonable cell designs, we would expect each Comesh™ cell to require an area of about 9 $S^2$ using this technology. This would increase the expected gate density for a .13 micron Comesh™ CCA to around *20000 gates per* $mm^2$.

#### 3.3.8. Is Comesh™ like CAM memory?

Comesh™ architecture has no resemblance to CAM.

## 4. Formal Methods and Boundary Mathematics

### *4.1. What are the benefits of boundary mathematics?*

The primary benefit of using void-based mathematical techniques is that they are inherently simpler. The void replaces half of each logical dual, so that `TRUE` is represented but `FALSE` is not, `AND` is represented but `OR` is not, `EXISTS` is represented but `FORALL` is not. The void has excellent computational properties: it takes up no memory space, requires no processing time, and cannot in any way increase the complexity of a computation. By using only one ground token rather than two, void-based representations are homogeneous, requiring no parsing, no identification of different operators, no precedence rules, and no notions of ordering, grouping, or arity. Void-based algorithms rely on simple sorting and pattern-matching.

### *4.2. Formal Methods and Systems*

#### 4.2.1. What is a formal system?

Formal methods apply mathematical tools to pragmatic domains. A formal system is a way of representing and transforming a problem domain (such as software and hardware design) using provably correct mathematical tools. Formal systems consist of definitions, axioms (or rules), representations, and methods of transforming representations. Formality is maintained by never violating the rules or the transformation methods.

#### 4.2.2. Why is formality important?

Many problem domains are too complex for people to understand without computational assistance. Formal software tools are the only way to be sure a complex transformation is correct.

### 4.2.3. Is void-based logic a formal system?

Void-based logic is a formal system with very simple transformation rules that, when followed, make the solution of logic problems relatively straight-forward.

## *4.3. Boundary Mathematics and Logic*

### 4.3.1. What is Boundary Mathematics?

Boundary mathematics is the formal expression of common mathematical ideas (numbers, logic, and sets) in two and more spatial dimensions rather than in one-dimensional symbolic strings. The field is also known as Diagrammatic Formal Systems. Topology and knot theory are two formal domains of mathematics that use spatial forms. Some diagrammatic formal systems include Conway's Game of Life and other cellular automata, the software program Geometer's Apprentice that facilitates geometry proof through manipulation of diagrams, Barwise's software program Hyperproof that converts elementary logic into object manipulation, manipulable Cuisennaire rods for kindergarten arithmetic, and Venn diagrams for sets.

### 4.3.2. What is Boundary Logic

The term boundary logic refers to the use of formal spatial/diagrammatic systems to express the concepts of logic. Boundary logic uses boundaries, maps, icons, graphs, diagrams and other spatial forms to represent logic and transformations of logical form, such as tautology detection, inference, deduction, and minimization. Some types of boundary logic are Venn diagrams, Existential Graphs, Karnaugh maps, Laws of Form, and void-based logic.

### 4.3.3. What is void-based logic?

Void-based logic uses non-representation, the void, as an active computational element. A binary system has two ground concepts; for logic, these are `TRUE` and `FALSE`. Void-based logic replaces one of these concepts, `FALSE`, with the void, that is, with an *absence* of a token. The concept `TRUE` is represented by the presence of a token. Void-based logic therefore combines truth with existence and falsity with non-existence or absence. This makes void-based logic a *unary* rather than a binary logic.

Void-based logic uses deletion or erasure as the primary computational operation. Technically this is known as *void-substitution*: the void is substituted for configurations that, by rule, have no impact on the meaning of the form as a whole. Similarly, configurations that are void-equivalent cannot impact the meaning of a problem, since by definition the void does not interact with any tokens. Conventional logics use rearrangement and collection of facts as the primary computational techniques, as a consequence, conventional formal software tools tend to bog down with accumulated details.

### 4.3.4. What does void-based mean?

Void-based means that the primary tool of representation and computation is absence, or non-existence. Conventional symbolic logic uses symbols such as `TRUE`, `FALSE`, `AND`, `OR`, `NOT`, and `IMPLIES` to represent concepts, it is symbol-based. In contrast, void-based logic uses containers rather than symbols to represent the concepts of logic.

*Containers* support an inside and an outside. This extra expressability permits the simplification of the rules and representations of symbolic logic. When the inside of a container is *empty*, it contains nothing. The inside is void, but a container lets us indicate that void. Empty containers allow the concept of

FALSE to be associated with emptiness. Symbols do not have a way to access or use absence because they do not have an inside.

Typographically, delimiting tokens such as parentheses, brackets, braces, and the like are containers in that they delineate an inside and an outside. For example, in the form (A B), the object A and the object B are on the inside of the parenthesis. In the form A(B), the object A is on the outside while the object B is on the inside.

### 4.3.5. How do you convert standard logic into containers?

Textually, a parenthesis (called ***parens***) is a container. All elementary logic problems and combinational circuit structures are simply well-balanced parens. The map between parens containers and logic 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)) |

By examining the forms on the right, we see that all the tokens of symbolic logic are converted into parens containers. Some examples of converting conventional symbolic logic to void-based logic follow:

| | |
|---|---|
| FALSE OR TRUE | ( ) |
| FALSE AND TRUE | (( )(( ))) |
| a IMPLIES a | (a) a |

Formally, void-based logic is more succinct than symbolic logic; the mapping from logical forms to containers is many-to-one. This can be seen in the example of transcribing DeMorgan's Law;

| | |
|---|---|
| a AND b = NOT (NOT a OR NOT b) | DeMorgan |
| ((a)(b)) = ( (a) (b) ) | transcribe |

### 4.3.6. What are the essential concepts and features of void-based logic?

Void-based logic is a *calculus of containers* (mathematically, partially ordered lattices). Containers are spatial objects that *do not possess* ordering, grouping, or cardinality properties (commutativity, associativity, arity). Thus, a parens form is a spatial object without any notion of sequencing or connectivity.

Since the void is indistinguishable from the white-space of a page, expressions equivalent to the void are everywhere throughout a form, in all the places where white-space exists. Thus any `FALSE` form can be placed anywhere throughout a parens form.

Parens forms are simplified by following transformation rules that identify which structures can be deleted as being void-equivalent.

Transformational rules rely on the *transparency* property of containers: the inside of a container is transparent to the outside. Transparency allows transformations at a distance; that is, objects do not need to be in the same space to effect each other. In multilevel circuits, for example, transparency allows gates near the output to effect changes to gates near the input, without propagating information through the gates along the path between input and output.

### 4.3.7. What are the transformation rules of void-based logic?

Like the elementary arithmetic of numbers, void-based logic has rules for containers that do not include variables. These are:

```
( ) ( ) =  ( )                    CALLING
 (( ))  = <void>                  CROSSING
```

Like the elementary algebra of numbers, void-based logic is an equational system, the primary method for expressing equivalence is with an equals sign, =. The axioms, or rules, of void-based logic are:

```
(A ( ))  =  <void>                OCCLUSION
((A))    =  A                     INVOLUTION
A {B A}  =  A {B}                 PERVASION
```

Capital letters refer to any parens form regardless of complexity, including spatial collections of separate forms and the absence of any forms. 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 a multilevel simplification rule: what is outside can be deleted or inserted inside, independent of the depth of nesting within a parens form.

### 4.3.8. What does void-based computation look like?

The void-based approach to an example Boolean minimization problem follows. The problem is expressed in the symbolism both of logical connectives and of Boolean algebra. Spacing in the void-based reduction is used to emphasize deletions. The particular deletion rule is listed on the right.

```
b OR ((NOT a) AND b) OR ((NOT b) AND c) OR (c AND d)   problem
f(a,b,c,d) = b + a'b + b'c + cd
                                                       Rule
      b (((a))(b)) (((b))(c)) ((c)(d))                 transcribe
      b (  a  (b)) (  b  (c)) ((c)(d))                 involution
      b (  a  ( )) (     (c)) ((c)(d))                 pervasion b
      b            (     (c)) ((c)(d))                 occlusion
      b                   c   ((c)(d))                 involution
      b                   c   (( )(d))                 pervasion c
      b                   c                            occlusion

f = b + c                                              transcribe
b OR c
```

The first application of the Pervasion rule on line 3 uses transparency: there is a `b` on the outside, and two other `b`s on the inside, in different containers, at different depths, and with different intervening structure. Transparency gives permission to ignore these differences.

Here is the void-based proof of the conventional law of logic called *modus ponens*:

```
(a AND (a IMPLIES b)) IMPLIES b                    modus ponens
```

The logic expression is transcribed into parens in parts:

```
       (a IMPLIES b)                          (a) b
 a AND (a IMPLIES b)                     ((a) ((a) b))
(a AND (a IMPLIES b)) IMPLIES b         (((a) ((a) b))) b
```

The proof will reduce the parens form of *modus ponens* to the form of `TRUE`, which is an empty parens.

```
(((a) ((a) b))) b                                  transcribe
  (a) ((a) b)   b                                  involution
  (a) (     )   b                                  pervasion (a) b
      (     )                                      dominion, QED.
```

The last step in the proof uses a void-based theorem called Dominion. The Dominion theorem is proved below. This is an example of an equational proof; the left-hand-side of the equation to be proved is converted via substitutions into the right-hand-side.

```
  A ( )  =?=  ( )                    dominion
((A ( )))                            involution
(       )                            occlusion, QED.
```

### 4.3.9. What is the history of void-based logic

The American logician and philosopher Charles S. Peirce first conceived of and published results in void-based logic in 1898. Peirce called his work *Existential Graphs*.

In 1854, Boole presented the first formal system of logic. In the late nineteenth century many iconic and symbolic forms of logic were explored. Only in the early 1900s was logic formalized into a complete and consistent symbolic system. Symbolic logic was adopted throughout the logic community; the iconic forms were considered to be too limited and clumsy.

The next major contribution to void-based logic was George Spencer-Brown's 1969 book *Laws of Form*, a mathematics text that introduced void-based logic within an algebraic framework. Spencer-Brown clearly stated the containment rules that express logic as enclosures.

### 4.3.10. What are the practical benefits of using void-based logic?

Void-based logic reduces the complexity of representing both logical ideas and logical computation, leading to efficient new logical operations, deductive processes, and software and hardware architectures for digital computation.

As well, boundary logic provides a new economy of thought. Perhaps the most penetrating idea introduced by the iconic approach to deduction is that what we know as rationality is not the formal symbolic manipulation of complex and challenging logical formula. Rather it is a simple and inevitable consequence of enclosing empty space. Understanding that logic can be seen and experienced as a spatial idea requires study of boundary mathematics.

### 4.3.11. Why wasn’t this approach adopted earlier?

Peirce's work on Existential Graphs was so foreign an idea that it was generally ignored by the logic community during the formalization of logic in the early twentieth century. Once symbolic logic was accepted as the standard for logical representation, boundary logic approaches were not studied. George Spencer-Brown's 1969 book *Laws of Form* also contained such unusual ideas that logicians concluded that it was "just another representation of Boolean algebra", one without significantly interesting properties. Making use of void-based logic requires a willingness to study an area that is very much “outside the (proverbial) box.” As yet, only a few people have done this.

### 4.3.12. What are the disadvantages of the boundary logic approach?

Logic has evolved over thousands of years within language, and has been learned by everyone as a symbolic skill. Since void-based logic combines several new tools and perspectives, particularly casting conventional logic into a spatial and algebraic framework, it is very unfamiliar to almost everyone. This unfamiliarity is its principal disadvantage.

### 4.3.13. Who else is working in this area today?

George Spencer-Brown, author of boundary logic text *Laws of Form*.

Professor Louis Kauffman, widely published on boundary mathematics. His principal expertise is in knot theory. University of Illinois at Chicago Department of Mathematics.

Dr. Richard Shoup, President of the Boundary Institute, applying boundary mathematics to certain fundamental questions of physics.

Don Roberts, 1973 thesis on Peirce's Existential Graphs.

John Sowa, IBM researcher, initiated the field of Conceptual Structures, which evolved from Peirce's Existential Graphs.

Dirk Baecker, applying boundary mathematics to sociology.

Donald Kunze, an architecture professor at Penn State. Applying boundary mathematics to literature and to architecture.

There are several researchers in diagrammatic logic affiliated with the Visual Inference Lab at Indiana University, and with the Center for Study of Language and Information at Stanford. These include Jon Barwise, Gerard Allwein, Eric Hammer, and Sun-Joo Shin.

### 4.3.14. Where can I find more information?

| | |
|---|---|
| http://www.lawsofform.org | Laws of Form (has an extensive bibliography) |
| http://www.peirce.org/ | Charles Peirce |
| http://users.bestweb.net/~sowa/cg/ | Conceptual Graphs |
| http://www-csli.stanford.edu/ | Center for the Study of Language and Information |
| http://www-vil.cs.indiana.edu/ | Visual Inference Lab, Indiana University |

Those seeking information about Iconic Logic and BTC's approaches to void-based logic should contact BTC directly.