COMPUTATIONAL ARCHITECTURES

William Bricken

August 2001

After a brief introduction to Boundary Logic computation, the computational architecture of several occlusion-based models is described.

CONTENTS

BOUNDARY LOGIC

Containment

Containment is constraint, it is the establishment of semantic boundaries. A container defines a limit which when crossed changes semantic value.

The relation of containment is one of distinction, one side of the container is distinguished from the other by its ability to hold or contain. That which is distinguished is space itself, a space is distinguished when it is in a container.

Containers can be impermeable (e.g. closed boxes and egg shells) or semipermeable (e.g. cups and vacuum cleaners), but not permeable (e.g. doors and glass). With reference to a particular side of a container, impermeability is *opaqueness*, permeability is *transparency*. Thus, for a semipermeable container, the side which can contain is opaque to its contents, while the other side is transparent. The semantics of containment most usually adopts the convention that the outside is transparent, that you can put things into containers without ressitence. The inside is opaque, you should not be able to take things out of containers without changing what is meant by containment.

Boundary mathematics provides conceptual structures that can potentially unite the various branches of foundational mathematics. As is demonstrated herein, logic is built upon semipermeable boundaries. Sets are constructed of impermeable boundaries. Integers are also made of impermeable boundaries, differing from sets in their treatment of the empty container.

The containment relation is antisymmetric; if two containers contain each other, they are the same container. Containment is transitive; the container of another container also contains the contents of the other container. That containers do not contain themselves makes containment irreflexive, and thus different from both equivalence and partial ordering relations.

The containment relation is one to many: a container can hold any number of individual objects, but an object has only one container. *Replications* of a unique object can occupy multiple containers.

Parens Form

Containers can be represented in linear text by well-formed parentheses (WFP). When interpreted as containers, parentheses are called *parens*. A parens expression makes containment visually obvious: each parens contains other parens forms, grounding in either an empty parens or a label. Paired delimiters in a linear space define the containment partitions of that space.

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 the following six containment pairs:

```
          reader [ 0
                   0 ( 1
                       1 ( 2
                           2 ( a
                       1 ( 3
                           3 ( b
```

The left-parens represents a containment relation between two labeled objects, locations, or spaces. When a containment boundary is crossed inwardly, the depth of nesting is incremented by one. Note that variable labels are implicit boundaries.

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>}
```

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.

Void-Based Computation

The representational space within which a distinction is marked is initially void; it is a space with no contents, no structure, and no explicit representation. Computation in spatial models is *void-based*, in that all transformations are with reference to the void, to the absence of marks. Colloquially, void-based computation proceeds through *erasure operations*. Deletions are obviously more desirable that rearrangements, since deletion removes structure, explicitly simplifying an expression. Technically, erasure is *void-substitution*, the void is substituted for a void-equivalent expression.

The assertion

X = <void>

states that X is a *void-equivalent*. Like the void, X can arbitrarily exist in any location in a spatial form, that is, anywhere within the boundary representation.

Void-equivalent forms cannot interact with the semantics of the representation; being essentially non-existent, they cannot effect the validity of a computation. However, void-equivalent forms can be used to generate a diversity of structural, or syntactic, variants. Structural variants for a given logical function are usually due to incorporation of forms which are void-equivalent. Tautological complexity is simply the tangling of void-equivalent forms.

Void-Based Ground

The essential idea is to use non-representation, the non-existence of an indicator, to eliminate reference to one of the two logical ground values. Thus

Circuitry	*Logic*	*Parens*
1	True	( )
0	False	<void>

The empty container (a mark) is a ground value. We associate the mark with the concept True. As is characteristic of void-based systems, the mark has a dual interpretation as an operator, crossing a mark changes its value.

An empty parens contains the void. Empty space, the absence of a container or of a mark, is the absence of True, i.e. it is False. The technical problems of reading and returning voids are easily addressed:

```
        0          False          (( ))
```

An empty container negates the void it contains, changing value from False to True. A container of an empty container negates the mark it contains, returning the value to False. The explicit representation of False is of use only at output, it is not used or needed for computation.

The two elementary forms of False (one existent and one not) define a void-equivalence rule:

(( )) = <void>   *Crossing*

Boundary Logic Arithmetic

In evaluating the parens form of a combinational logic circuit, all variables are converted into grounds, that is, into empty parens or into the void.

Consider this parens form:

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

```
Let the variable bindings be          {(a  0),(b  1)}
which expressed in boundaries is      {(a   ),(b ())}
```

Substitution yields

```
        ((a)(b))    ==>   (( )(( )))
```

In parens notation, when inputs/variables are bound, all logical forms (including combinational circuits in general) reduce to a configuration of well-formed parentheses (WFP). Identification of WFPs is the simplest problem which separates state-less representations (such as simple finite state automata) from those requiring state (automata with stacks).

Variables become values when inputs are bound. The value of a positive input is indicated by an empty parens, (). The value of a negative input is void (i.e. it is a no-op, it is simply not recorded, it is erased).

The axioms of boundary arithmetic interpreted as logic were defined in George Spencer-Brown's 1967 book *Laws of Form*.

( ) ( ) = ( )   *Calling*

(( )) = <void>   *Crossing*

Boundary Logic Algebra

Boundary logic is axiomatized by two void-based axioms:

Occlusion provides an end condition for evaluation.

(( ) A) = <void>   *Occlusion*

The Pervasion axiom permits algebraic reduction of boundary forms:

{A} {B {A}} = {A} {B}   *Pervasion*

The generalization of double-containment is the theorem of Involution:

((A)) = A   *Involution*

Pervasion asserts that the parens boundary is semi-permeable, it is transparent to outside forms, and, necessarily for the distinction of containment, opaque to inside forms.

The unique aspect of this axiomatization is that the left-hand-side of each equation is converted into the right-hand-side solely through a void-substitution. Occlusion erases all forms in a spatial context as well as that context; Pervasion erases replications of nested forms; and Involution erases double parens.

Occlusion

Algorithmic identification of a WFP (without variables or labels) is equivalent to logical evaluation (without algebraic abstraction), using the Occlusion theorem as the single reduction rule:

({A} ( )) = <void>   *Occlusion*

An empty container erases (deletes, voids) its container and all contents.

There is a single terminal variant: when an empty parens remains in the outermost space, a theorem based on Occlusion is needed.

( ) {A} = ( )   *Dominion*

The form on the left-hand-side of occlusion is a void-equivalent, it can exist arbitrarily anywhere within any expression (including itself) any number of times. Increasing the size of an expression by manifesting void-equivalents is rarely motivated. Acknowledging the potential presence of void-equivalents enables a quite powerful set of unique reduction tools.

Every evaluation algorithm herein describes the same mathematical operation, that of Occlusion. No other transformations are used during evaluation. When the representation (the data structure) varies, the evaluation algorithms take on substantively different operational characteristics.

Virtual Insertion

Pervasion can occur in either direction. When inserting, the inserted form can be considered to be virtual. Virtual forms can be placed arbitrarily within other forms, and can be used to reduce logical complexity, depending solely upon utility and objective. However, virtual forms are void-equivalent, they do not actually exist. Thus virtual forms can be deleted by choice when not useful. They are thus similar to queries, questions asked by the user which return answers but which do not impose a symbolic overhead.

Virtual insertion is based on the general principle of Pervasion, that *all forms in shallower spaces are arbitrarily in deeper spaces*, that boundaries are semipermeable. Virtual insertion treats all inserted forms as void-equivalents, as hypothetically, or virtually, inserted into the deeper spaces. Pervasion is an *option* rather than an operation:

A (... (N)) = A (... (^A^ N))   *Virtual Insertion*

The inserted form, highlighted by carets, is treated as a *query*. When it returns a positive answer, the subform which defines its current context is occluded. Virtual insertion is not search, it is simply percolation of a possibility down a graph form, with clear termination conditions at the leaf nodes. As the virtual form descends, it may itself be reduced due to Pervasion of the subforms it passes. That is, subforms in the form being inserted into can Extract subforms of the virtual form.

One of three things may happen to the *inserted* form:

1) it vanishes (becomes void), in which case it has no reducing effect

2) it reduces to ( ), in which case its immediate context is erased by Occlusion, or

3) it reaches the deepest space while still containing variables, in which case it has no reducing effect.

A logic minimization problem using virtual insertion:

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

      ((a b) (a d (b c)))                            Initial form
      ((a b) (a d (b c) ^(a b)^))                    Query (a b)
      ((a b) (a d (b c) ^(  b)^))                    Extract ^a^
      ((a b) (a d (b c ^(b)^)))                      Query (b) deeper
      ((a b) (a d (b c ^( )^)))                      Extract ^b^
      ((a b) (a d))                                  Occlude (b c)

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

DISTINCTION NETWORKS

A distinction is a minimal difference. In this paper, distinctions and containers are synonymous, both refer to a minimal difference. Containment is a more physical concept, in that physical objects can embody containment relations. Distinction is a more abstract concept, it refers to the pure notion of difference. Here, networks of distinctions (dnets) are interpreted as logic circuits.

Configurations of distinctions can be represented by well-formed parenthesis structures, by nested containment relations. These configurations can also be expressed as graphs, called *distinction networks*, in which nodes represent distinguished spaces and arcs represent the is-contained-by relation.

Distinction networks map onto logic networks (combinational circuits), with the distinction being interpreted as a variary NOR. However, distinction networks can be interpreted in a spatial mathematics which is simpler than the logical interpretation embodied in circuit configurations.

Spatial models are inherently more succinct than textual models. The functional morphism remains, but spatial languages require less representational mechanism. Computational effort is also effected, spatial representations require fewer transformations to arrive at the same result.

Using distinctions (or containers) to represent the syntax and semantics of logic is called Boundary Logic.

Distinction networks are directed acyclic graphs. When interpreted as logic, links represent logical implication and dnodes are distributed autonomous control centers. Each dnode represents a container which implements the Occlusion transformation. Asynchronous communication between dnodes of local decisions produces global computational results without global coordination.

Distinction networks differ from conventional circuitry in that the logical operation:

- is singular
- is associated with wires rather than with gates
- is independent and autonomous
- operates by disconnection rather than by signal propagation

Algorithm

Each dnode has a local memory which stores the identifier (location) of its upper connections (UPPERS) and its lower connections (LOWERS).

1. At initialization, each dnode polls its lowers. If there are none, the dnode sends an DELETE message to each of its uppers.

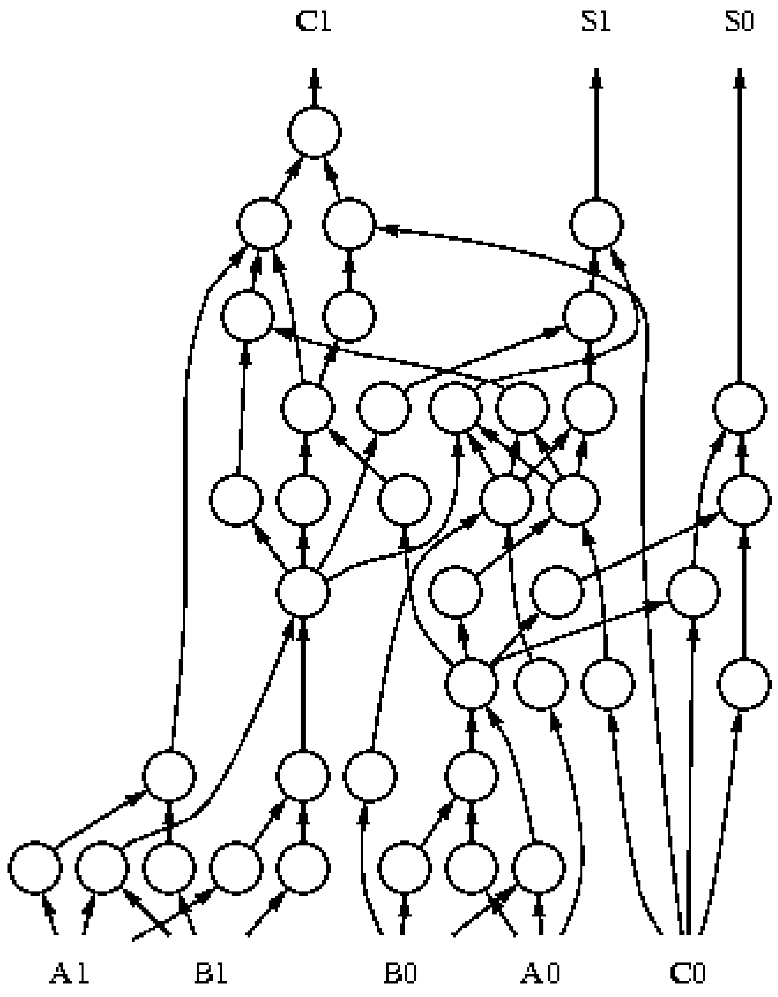
2. Receiving DELETE causes a dnode to disconnect itself from each of its uppers. This is achieved by sending a DISCONNECT message.

3. Each dnode which receives DISCONNECT deletes the source of the message, and then polls its lowers, returning to step 1.

Examples Of Distinction Networks

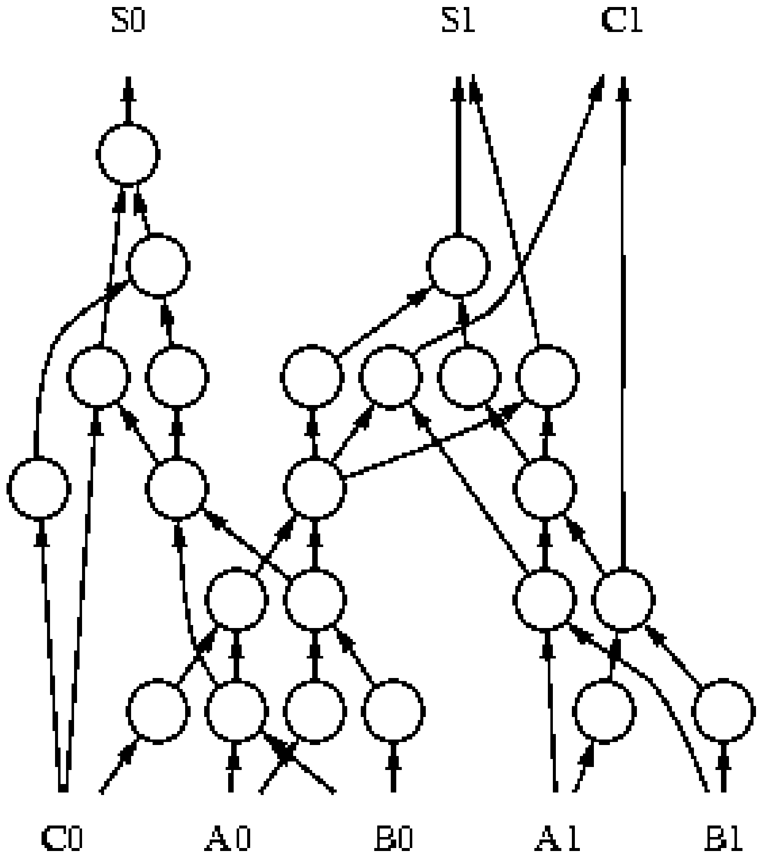
Three versions of the same simple circuit follow. The circuit's function is to add two two-bit binary numbers.

The architecture for this adder is called *carry-bypass*, because it optimizes computation of certain cases of the carry from a sum. Below, the binary numbers are (a1 a0) and (b1 b0); the carry from a previous addition is c0. The sum consists of two bits (s1 s0) and a carry c1. This adder has been converted from logic gates to distinction nodes (the circles).



A two-bit carry-bypass adder expressed as a distinction network

After Losp/Pun optimization, the distinction network looks like this:



Optimized two-bit carry-bypass adder

The original adder consists of 39 distinction gates and 65 wires, with a depth (path delay) of 9. The optimized adder has 23 gates and 40 wires, with a depth of 6.

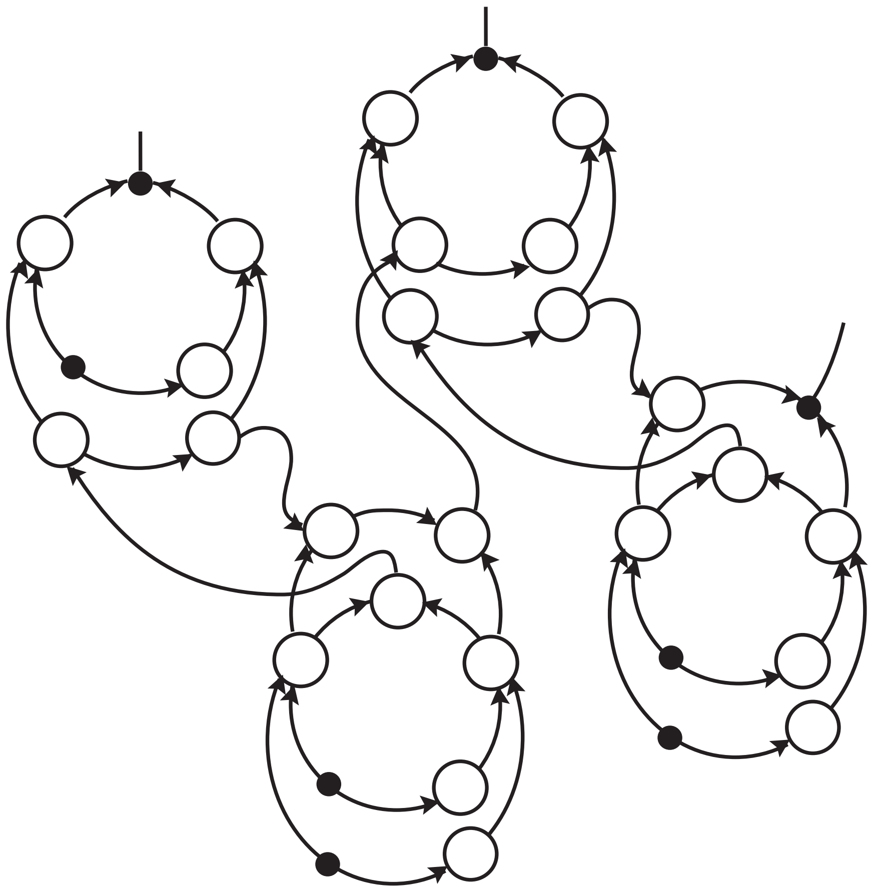
Note: in minimizing this adder, the carry-bypass feature is removed. Although this circuit is simpler, it may not be as fast as the original.

The final version of this circuit consists of a slight structural modification to the optimized circuit that highlights the structural patterns. This structure is not perceivable in the two other diagrams.

Each large circular loop is an if-then-else logic function. The labeled nodes A through E are the identical nodes labeled respectively -A through -E.

They have been visually separated again to highlight patterns and to unclutter the presentation.

This version has 19 nodes, 38 wires, and a depth of 6



The two-bit carry-bypass adder
configured to emphasize its logical structure

OCCLUSION ARRAYS

An Occlusion Array is a modified read-only memory attached to void-based reading hardware.

All types of circuits are accommodated by the Occlusion Array:

- multiple output circuits
- sequential circuits with registers
- pipelined circuits
- parallel array circuits
- limited fanin and fanout circuits
- programmable logic array AND/OR circuits

The hardware "reads" the *output* of the soft-hardware circuit in its memory array by successively masking (or deleting) the irrelevant parts of memory, those parts which are void-equivalent. What remains (the output) is written onto the output *screen*. The *memory* is never altered, except when changing the functionality of the Occlusion Array. All deletions are achieved by a *mask* vector, which together with the output screen are the only dynamic parts of the array. *Filters* identify input and output rows.

```
     MEMORY       SUPPORTING HARDWARE

     circuit      masks filters screen

----------------  -----  -----  ---
|       1 1    |  | | |  |1| |  | |
|       1   1  |  | | |  |1| |  | |
|         1 1  |  | | |  |1| |  | |
|             1|  | | |  | | |  | |
|             1|  | | |  | | |  | |
|             1|  | | |  | | |  | |
|              |  | | |  | |1|  | |
----------------  -----  -----  ---
```

This circuit computes the combinational function *2-out-of-3 majority*. The top three rows are three inputs; the bottom row is the output.

The array is square, having identical indices in each dimension. In the array, rows and columns identify specific variables (inputs) and containers (parens). A mark in the array identifies a relation between forms, the column form *contains* the row form. Rows identify contents; columns identify containers.

The Void-Based Mechanism

Void-equivalent parts of memory are identified using the Occlusion rule:

(A ( )) ==> <void>   *Occlusion*

The rule implemented by the supporting hardware locates empty parens by identifying empty columns, i.e. those containers without contents. It then masks the *context* of the empty container.

When an identified empty parens happens to be an output, a positive value (1) is written on the output screen.

Masking continues until the entire memory is masked. At that time the output screen contains all positive outputs from the circuit. Outputs not identified with a positive mark are negative.

The void-based mechanism differs from conventional computation in these ways:

- Computational progress is by *elimination* of irrelevant containers rather than by accumulation of logical transformations.

- Signals are not propagated through a network of logic gates, rather a single *question* is asked of memory: Is this column both non-masked and empty?

- Void-based computation is *unary* rather than binary. Only positive outputs are computed, negative outputs are inferred.

- Time (clocked structural loops which store transient results in register states) is not used as a mechanism for reuse of circuit structure.

Computational time for arrays is nearly independent of circuit size and complexity, similar to pipelining in conventional circuitry. The computational delay in an array scales logarithmically to circuit size, and is thus very close to constant,

- An implicit difference is embodied in the parens form itself: computation is over containers rather than values.

Generic Occlusion Algorithm

The reductive power of the occlusion theorem comes from its identification of a void-equivalent.

The generic occlusion algorithm reduces any parens form which contains an empty mark, ( ). Since forms are assembled in space, without associative or commutative structure, the appropriate implementation is strongly parallel. Parallelism can be simulated by looping over all items within a particular space, since forms sharing a space are independent (not coupled). In parens forms, spaces are nested. These are best addressed by recursion, or more succinctly, a parallel mapping across all forms in a space.

Recursive Algorithm

In this recursive mapping algorithm, "the form" is the input.

English description

```
if the form is empty or an atom
  then return the form
  else if the form contains a mark
    then delete the form
    else apply occlusion to each subform in the form.
```

Pseudo-English code

```
(if the-form-is (a-mark or an-atom)
  then return-the-form
  else (if (the-form-contains a-mark)
         then delete-the-form
         else (map occlusion over-each-subform)))
```

LISP code

```
(defun generic-occlusion (pf)
  (cond
    ((or (null pf) (atom pf)) pf)
    ((member mark pf) <no-op>)
    (T (map occlusion pf))))
```

BIT-STREAM SIMULATORS

Through a simple encoding, boundary logic can express Boolean functionality in bit-streams. This permits a stack based algebraic simulation of circuit functionality. When input variables are bound, the bit-stream simulates the running circuit. As well, the reduction behavior of the simulator provides measures of inherent parallelism and sequentially of the circuit logic.

Initial estimates suggest that a hardware bit-stream simulator should be able to simulate an ASIC within an order of magnitude of the speed of the actual (non-pipelined) ASIC.

Algorithm

Encode parens form without variables as

```
(=1        and        )=0.
```

The bit-pattern matching engine uses these rules for evaluation:

```
(()) =        is  1100 => ____

()() = ()     is  1010 => 1__0  or alternatively 10__ or __10
```

A slightly more sophisticated algorithm is required to apply Occlusion directly.

The algorithm is stack-based, using two accessor functions.

```
Accessor:   (get n) => get n values from bit-stream
Backtrack:  (pop n) => get top n values on processed-stack
```

Since all bits on the processed stack are 1s, the stack is implemented by a counter. When the first four bits do not match a reduction pattern, the top 1 is removed from the stream and the counter is incremented.

(pop n) decrements the counter by n while adding 1-bits to the front of the stream.

Three implementations of the scanning algorithm are available:

- in software on a conventional multiprocessor
- in customized assembly language on a multiprocessor
- in hardware

Since the simulator can freely and opportunistically use both the signal and the gate interpretation, it is both efficient and small. The current implementation of the simulator requires approximately 200 gates, or 40 4LUTs in a FPGA. The LUT delay path is maximally about 25 deep. The larger and deeper the circuit being simulated is, the more efficient the bit-engine simulation is, relative to the actual circuit.

The simulator is most efficient for circuits expressed in the deeply nested implicate form, since the length of the bit-stream circuit determines the number of processing cycles (without parallel bit-processors) and this metric decreases when depth increases. A typical circuit transcribes into

    2*(gates + inputs) bits

prior to Losp optimization, with at least a factor of two decrease during optimization. Thus the number of bits is roughly equal to the number of gates in a conventional notation.

Input can be made parallel at each parens boundary. Since the bit-engine uses a one-bit input stream, a 16bit input bus and a multiplexer-pair could feed 16 bit-engines in parallel at little cost in load-balancing logic. One Xilinx XC4025 chip could comfortably hold 16 simulators together with the parallelization logic to handle 16 bit-streams.

Here is a rough computation of the expected speed of this algorithm when implemented on an ultra-wide input bus: A 1000 bit-width input can encode about 500 distinction gates after variable binding. (This covers about 2/3 of the EDIF benchmarks.) It will take an average of (log 1000/2) = 9 ticks to evaluate. A tick can be done in 4 machine cycles. At 144MHz, this gives us a simulation capability of 144M/32 = 4M test vectors per second.