USING OCCLUSION TO EVALUATE CIRCUITS

William Bricken
February 2001

CONTENTS

ABSTRACT

Evaluation of combinational and sequential circuits can be accomplished with a single, rapidly converging operation, called *occlusion*. The efficiency is achieved through the unary notation of Boundary Logic which confounds nested containment with both graph connectivity and logical implication.

This paper defines the essential semantic concepts, describes the generic occlusion algorithm, and presents the occlusion algorithm for several void-based models of circuit functionality. Examples trace the dynamics of occlusion during circuit evaluation. The concentration is on models which may be realizable in hardware, especially the occlusion array model (OAM). All techniques apply to sequential circuits with registers, as well as to the simpler combinational circuits.

The objective is to determine the cost/performance of boundary circuits compared to the conventional signals-and-gates actualization used in transistor-based silicon logic.

OCCLUSION

Reducing a well-formed parens (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, the precursor axiom to Occlusion is needed.

( ) A = ( ) DOMINION

Placing each side of this axiom within a container converts Dominion into Occlusion, via Involution

(( ) A) = (( )) = <void> Proof of occlusion

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.

GENERIC OCCLUSION ALGORITHM

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

(A ( )) = <void> OCCLUSION

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))))
```

Recursive Algorithm with Look-ahead

A more complex look-ahead algorithm keeps better track of the context of evaluation. The selector for look-ahead forms is strongly parallel.

English description

```
STEP 1    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 step two to the form.

STEP 2    select a subform from the contents of the form
            if the subform is an atom
              then return the subform
              else if the subform contains a mark
                then delete the subform
                else add the application of step 2 to the subform
                     to the application of step 2 to the rest of the form.
```

Pseudo-English code

```
(if (the-form-is (empty or an-atom))
  then (return the-form)
  else (if (the-form-is a-container-of-a-mark)
         then (delete the-form)
         else (if (the-content-selection is-an-atom)
                then (add the-content-selection
                       to (recur-on the-rest-of-the-form))
                else (if (the-content-selection has-a-shallow-mark)
                       then (recur-on the-rest-of-the-form)
                       else (add (recur-on the-content-selection)
                              to (recur-on the-rest-of-the-form)))))
```

LISP code

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

(define generic-occlusion-aux (pf)
  (cond
    ((null pf) nil)
    ((atom (first pf))
     (cons (first pf) (generic-occlusion (rest pf))))
    ((member mark (first pf))
     (generic-occlusion (rest pf)))
    (T (cons (generic-occlusion-aux (first pf))
             (generic-occlusion-aux (rest pf))))))
```

OCCLUSION ARRAY MODEL (OAM)

The algorithms which follow are applicable to both combinational and sequential circuits. Sequential processing includes additional steps for managing the state of registers.

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.

Transformations for void-based arrays are based on identifying the existence of a mark, its relationship to the void. Conventionally, matrix transformation is based on specific numerical combination of matrix cells.

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).

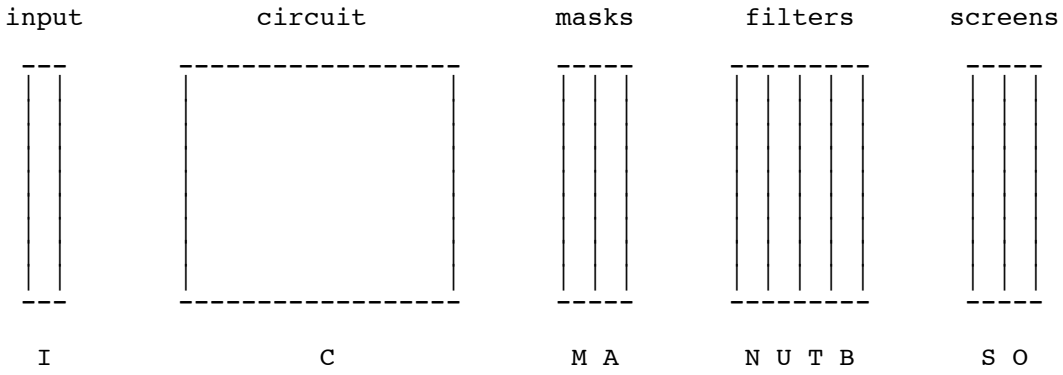
In the following, the components of the occlusion array model are described in detail, and the generic occlusion algorithm is transcribed into array behavior.

MODEL COMPONENTS (SEQUENTIAL and COMBINATIONAL)

Components

- Input I
- Occlusion array C
- Mask M
- Active A
- Input filter N
- Output filter U
- Register-top filter T
- Register-bot filter B
- Register state screen S
- Output screen O

The sequential circuit model consists of 10 components: one array and nine vectors, all with the same dimensions. Two pairs of these vectors do not intersect (S&O, T&B) and can possibly be combined.



Input vector

I = system Input

This dynamic element is activated with system input, at the beginning of computation. The maximal dimension of this vector could be substantively less than the array dimension (which includes all dnodes as well). For hardware parallelism, all dimensions are the same in this model.

Technically, the input vector is not necessary, since positive and negative inputs can be recorded directly into the masks.

Array

C = Occlusion array

This static element defines the functionality of the circuit. Once loaded, it is the static control element which guides computation but does not enter into it through any transformations. The array is analogous to the place and routing of a conventional FPGA circuit, and to the machine instructions of a conventional CPU. It abstracts wiring so that circuit functionality does not need to be expressed by physical wires. As an abstraction, it is as reconfigurable as machine instructions.

Rows and columns with the same occlusion array index identify the same dnode from different perspectives. Rows are the input perspective, columns are the output perspective. Marks in a row identify the column nodes for which the row node is an input. Marks in a column identify the row nodes which are input to that column node.

Prior to loading, the computational array must be sized for the particular circuit. The Losp optimizing compiler on parens forms can convert the circuit specification to a specified time/space configuration.

Masks

M = Mask
A = Active

Masks are the computational elements, the only structures (other than the input vector and the output screens) which change dynamically.

The mask vector permanently deletes/voids the corresponding dnode row and column until evaluation is completed. When all elements are masked, evaluation ends, and input/output is refreshed.

The active vector identifies which dnodes are currently being transformed, it tracks the evaluation frontier of the graph. This vector also tracks when specific registers receive input from the circuit computation, thus setting their state.

Filters

N = iNput filter
U = oUtput filter
T = register Top (output side) filter
B = register Bottom (input side) filter

Filters identify the type of particular row entries; specifically, the system inputs and outputs (not their i/o values), and the two parts of each register. The top of a register is the dnode which enters the register state into the computation as input, as choreographed by the clock. The bottom of a register is the dnode which receives new state from the computation and marks it on the register output screen. These two components are called the register-top and the register-bot.

Screens

S = register State
O = Output

Screens collect and display computational results, the system output and the saved state of the registers.

Only positive marks are collected by screens, which are initialized each input cycle to zero. When all rows are masked, outputs which have not been marked retain the logically correct zero/void value which was set at initialization time.

Dnodes which provide system output and are also further used within the circuit will not be scheduled for deletion until they have satisfied all of their containment relationships to the rest of the array.

REGISTERS

Clocked registers are indicated by marks in the top, or 0, row of a sequential circuit occlusion array. Row 0 contains the clock input, which is an input to every register in the single-clock system. Thus, marks in row 0 indicate a column which is a register.

The dnode representation of a register is the same as any other distinction. Since the logical semantics of a register is to pass a value unchanged, the register label is inside a double container. Involution assures consistent semantics.

The top and bottom nodes of occlusion array registers manage register behavior. The register-top holds the state and provides input to the evaluation process in the form of register state feedback. The register-bot receives the register state from the evaluation of the subcircuit leading into it.

The state of each register is stored in the register state vector. When the state is reentered into the computation, the behavior of register state vector is identical to that of the input vector.

Unlike conventional registers, all occlusion array register states are set to zero each full clock tick. During the next clock cycle (0->1->0) two opportunities occur for state to be set to positive. Only active registers (as defined by the active vector) can accumulate a mark indicating positive state. On the following clock cycle, the state values which have accumulated are fed back as input, the state vector is cleared, and the cycle repeats.

Negative state values remain zero by the following mechanism: when the state should be zero, the register holding that state will become masked during computation, thus eliminating the opportunity to be marked as positive.

The structure of a simple pun register is : ((@clok id)). This is separated into two dnodes when a distinction network is constructed:

```
(3 (@clok id))          register-top
(14 (3))                register-bot
```

The value of @clok is used by the evaluation engine to coordinate when register state values are collected and asserted.

Structurally, registers indicate cyclic patterns in a distinction network. They require a two-step mechanism which is implemented in the accompanying control logic rather than in the structure of the circuit. The gain is that registers become functionally equivalent to dnodes for the purposes of evaluating the array.

STATIC/DYNAMIC PARTITION

Static

- Occlusion array
- Input filter
- Output filter
- Register-top filter
- Register-bottom filter

Dynamic

- Input
- Mask
- Active
- Register State screen
- Output screen

Several components (array and filters) are specific to a specific circuit, but invariant to different inputs. These describe the static abstract circuit. Static components are loaded only once for each specific functionality.

Other components (interfaces and screens) are reset for each input vector, and change dynamically during the course of evaluation. (An exception is that the register state vector is reset every other clock transition, when clock ==> 0.)

COMBINATIONAL SUBSET

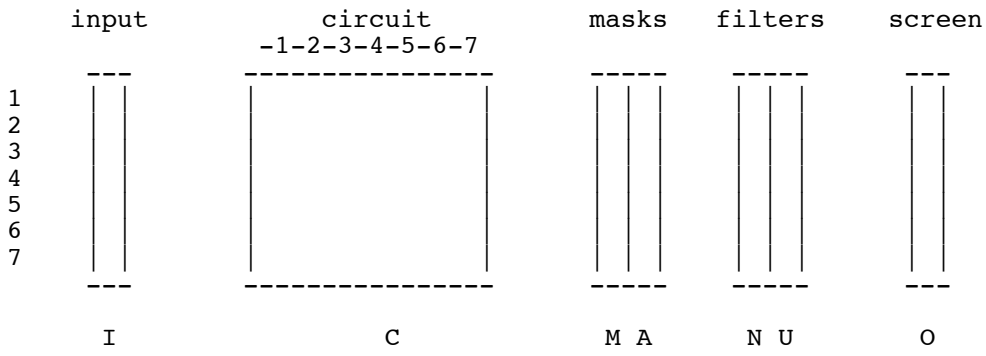
Combinational Components

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

Sequential Components

- Register-top filter T
- Register-bot filter B
- Register state screen S

The model for a combinational circuit is simpler in that registers are not needed. Registers essentially double the mechanism, they are a second type of i/o. The components necessary for combinational circuit evaluation form a simpler subset: one array and six vectors.



An alternative combinational algorithm which does not use an active vector is described later.

OCCLUSION FOR SEQUENTIAL AND COMBINATIONAL ARRAYS

This algorithm is the same for both types of circuits, with additional steps for register processing added in the case of sequential circuits.

A double bar, ||, on the right indicates independent steps which can be taken in parallel.

An asterisk, *, marked in the right column indicates a part of the sequential algorithm which can be skipped for combinational evaluation.

```
Initialize

      mark positive inputs into input-vector

      for every input
        if the input is positive, mark the active-vector
          else mark the mask
                                                                    ||
      for every register-top                                        *
        if the state is positive, mark the active-vector            *
          else mark the mask                                        *

Repeat

      for every active row
        when the row is-an-output, mark the output-vector
                                                                    ||
      when clock=0                                                  *
        for every active row                                        *
          for every register-bot marked in that row                 *
            mark the register-top state                             *
                                                                    ||
      for every active row
          mark the mask
                                                                    ||
      for every mark in every active row
          mark the mask

      when any row is unmasked
        for every unmasked row
          when that column is empty
            mark active
        repeat
```

SIMULATIONS

The functioning of an occlusion array is first illustrated with a simple combinational circuit, MAJORITY.

A sequential example follows, the ISCAS'89 LION circuit. Several structural versions of LION are included:

- raw distinction network
- relabeled homogeneous distinction network
- raw occlusion array

- reduced circuit
- reduced occlusion array

The functioning of the LION occlusion array is demonstrated by a symbolic circuit simulation. The simulation is shown for a Losp reduced array; the CNF array simulation is in a following section.

COMBINATIONAL EXAMPLE, THE MAJORITY CIRCUIT

Here the majority circuit is expressed as a pun body. Dnode numbering is offset since each variable is given a numerical label also. (Below letters are maintained for clarity.)

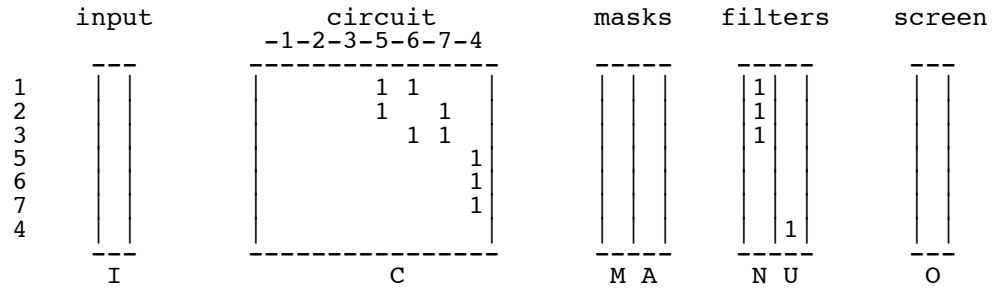
```
((   4  (5 6 7) )
 (   5  (a b)   )
 (   6  (a c)   )
 (   7  (b c)   ))
```

The occlusion array is

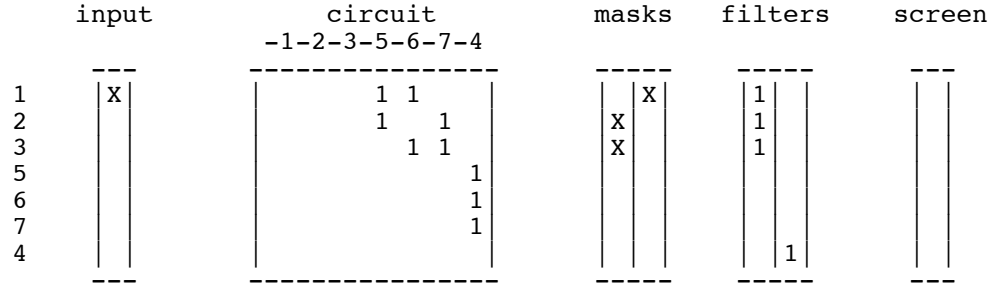
```
              OUT
        1 2 3 5 6 7 4

     1       1 1               a
     2       1   1             b
 I   3         1 1             c
 N   5             1
     6             1
     7             1
     4                         out
```

When the occlusion array for this circuit is loaded into the array processor, it will look like this:



We now provide a binding for inputs: {(a 1),(b 0),(c 0)}. New changes are marked as X.

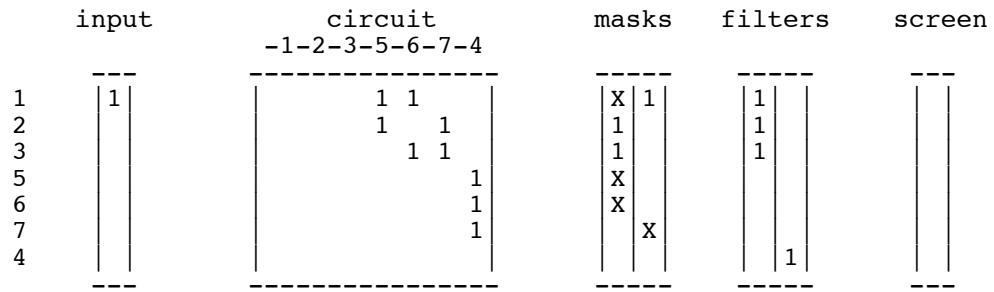


Two steps are taken during input initialization (illustrated above):

INITIALIZATION

1. Positive inputs are marked as active. {1}
2. Negative inputs are masked. {2,3}

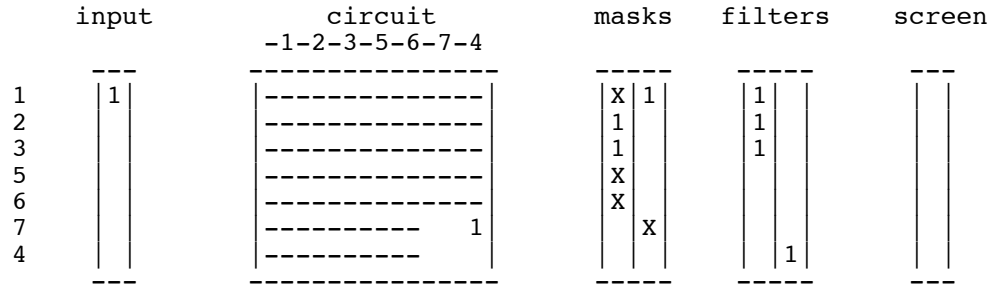
At this point the computational cycle begins. The steps for the first cycle leave the occlusion array in this state:



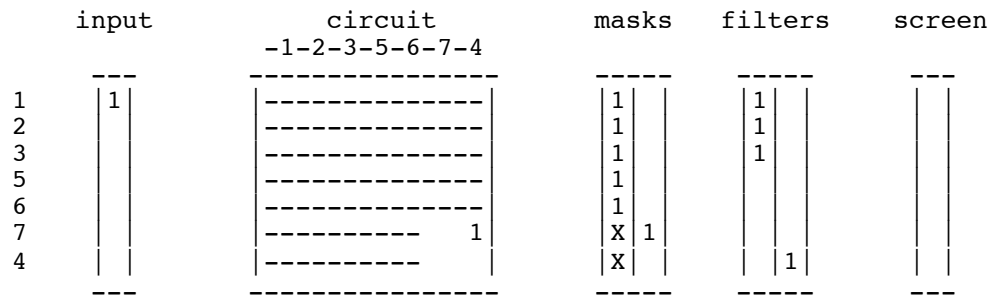
CYCLE 1

1-3.	Mark output-vector.	(none)
1-4.	Mask active.	{1}
1-5.	Mask marks in active.	{5,6}
1-6.	Test for opaque mask.	(fails)
1-7.	Mark as active any empty columns.	{7}

To help with the visualization of activating empty columns in step 1-7, below the masked inactive portions of the array have been obscured:



Next a second evaluation cycle (steps 3-7) begins.



CYCLE 2

2-3.	Mark output-vector	(none)
2-4.	Mask active	{7}
2-5.	Mask marks in active	{4}
2-6.	Test for opaque mask (succeeds)	
	Terminate	

When the output row 4 became masked, it was as a consequence of row 7 being active. In such cases the output-vector does not have an opportunity to become marked. It is in effect set to zero by the absence of an action.

Since the mask is completely marked, the evaluation is over. The output has nothing marked in row 4, so its value is False.

SEQUENTIAL EXAMPLE, THE LION CIRCUIT

The sequential process is illustrated by the ISCAS'89 circuit LION, after Losp reduction.

The evaluation algorithm is followed from EDIF to occlusion array simulation, with these illustrations of the pun form:

- Raw Distinction Network
  EDIF to raw-pun to algebraic raw-pun via the logic-to-parens map.

- Homogeneous Distinction Network
  Algebraic raw-pun to homogeneous raw-pun via relabeling.

- Raw Occlusion Array
  Homogeneous raw-pun to occlusion array
  via graph adjacency matrix.

- Optimized Lion Circuit
  Losp optimized pun to occlusion array.

- Simulation of Lion Circuit

RAW DISTINCTION NETWORK

When an EDIF netlist is transcribed into pun form, the "raw" translation is literal, using the Logic to Parens mapping while keeping all conceptual structures (such as ground, vcc, wires connecting input to output, etc.) as separate tokenized concepts. This is the initial raw-pun version.

Below, any ground states in the initial raw-pun have been evaluated. That is, all conceptual structures with logical ground semantics {T,F} have been removed through propagation of their values. These conceptual grounds exist to support physical entities (such as ground and vcc wires) which are no longer relevant for a void-based processor. Also below, the raw-pun version has been rendered as a distinction network, without any form of optimization.

The pun engine uses these typed labels for dnodes:

{(~n node), (!n register), (@ clock), ({a,b...} input), ({oa,ob...} output)}

```
((lion raw-pun (2000 11 1 11 38 49))
 ((main)
  ((a unk) (b unk) (@clok unk))
  ((oa 0))
  ((!1 (@clok ~5))
   (!2 (@clok ~6))
   (~3 (a))
   (~4 (b))
   (~13 (!1))
   (~12 (!2))
   (~7 (~3))
   (~8 (~4))
   (~19 (~9))
   (~14 (~10))
   (~18 (~11))
   (~6 (~14))
   (~9 (~15))
   (~10 (~16))
   (~11 (~17))
   (~5 (~19))
   (~16 (~27))
   (~17 (~28))
   (~15 (~30))
   (~20 (~3 ~13))
   (~22 (~4 ~7))
   (~21 (~4 ~13))
   (~25 (~7 ~12))
   (~23 (~7 ~13))
   (~26 (~8 ~12))
   (~24 (~12 ~13))
   (~29 (~3 ~8 ~12))
   (~27 (~22 ~23 ~25))
   (~28 (~25 ~26 !1))
   (~30 (~20 ~21 ~24 ~29)))))
```

HOMOGENEOUS DISTINCTION NETWORK

The typed labels and variable labels are now renumbered prior to the construction of the occlusion array. The dnet which follows is completely prepared for conversion to a occlusion array. It is in the *homogeneous distinction network form*, with node typing eliminated. This is a pure containment graph, the varieties of structures which support conventional physical circuits (gates, registers, wires, grounds, etc.) are not needed by the occlusion array technology.

```
((lion (2000 11 1 11 38 49))
 ((main)
  ((0 unk)(1 unk) (2 unk))
  ((oa 20))
  ((3 (0 7))
   (4 (0 8))
   (5 (1))
   (6 (2))
   (15 (3))
   (14 (4))
   (9 (5))
   (10 (6))
   (21 (11))
   (16 (12))
   (20 (13))
   (8 (16))
   (11 (17))
   (12 (18))
   (13 (19))
   (7 (21))
   (18 (29))
   (19 (30))
   (17 (32))
   (22 (5 15))
   (24 (6 9))
   (23 (6 15))
   (27 (9 14))
   (25 (9 15))
   (28 (10 14))
   (26 (14 15))
   (31 (5 10 14))
   (29 (24 25 27))
   (30 (27 28 3))
   (32 (22 23 26 31)))
```

RAW OCCLUSION ARRAY

Below, the non-optimized raw-pun circuit has been converted directly into a occlusion array with 33 rows and 48 marks. Without counting inverters, this reduces to 13 rows.

```
 -0-1-2-3-4-5-6-7-8-9101112131415161718192021222324252627282930313 2

 (      1 1                                                          )   0
 (          1                                                        )   1
 (            1                                                      )   2
 (                              1                             1     )   3
 (                            1                                    )   4
 (                    1                       1                 1  )   5
 (                      1                       1 1               )   6
 (      1                                                          )   7
 (        1                                                        )   8
 (                                                1 1   1          )   9
 (                                                        1     1  )   10
 (                                            1                   )   11
 (                                      1                         )   12
 (                                                1 1 1     1     )   13
 (                                          1 1   1 1             )   14
 (                  1                                             )   15
 (                        1                                       )   16
 (                          1                                     )   17
 (                            1                                   )   18
 (                                                                )   19
 (               1                                                )   20
 (                                                              1)   21
 (                                                              1)   22
 (                                                      1         )   23
 (                                                      1         )   24
 (                                                              1)   25
 (                                                      1 1       )   26
 (                                                        1       )   27
 (                                    1                           )   28
 (                                      1                         )   29
 (                                                              1)   30
 (                                  1                             )   31
```

Rows and columns refer to the same dnode from different perspectives (input and output). The marks in each row are the fanout of that row's dnode. The marks in each column are the inputs to the column dnode. The spatial arrangement of marks represents the functionality of the circuit.

REDUCED LION CIRCUIT

The same circuit functionality can be represented by different dnets and by different occlusion arrays. The LION homogeneous dnet is presented below, after Losp reduction. The accompanying occlusion array for the reduced dnet follows (this array will be used in further examples).

The array has 20 rows and 31 marks. With free inverters, this reduces to 12 rows. The mark at <13,8> is redundant; with respect to the occlusion array, this reduced circuit is not minimal.

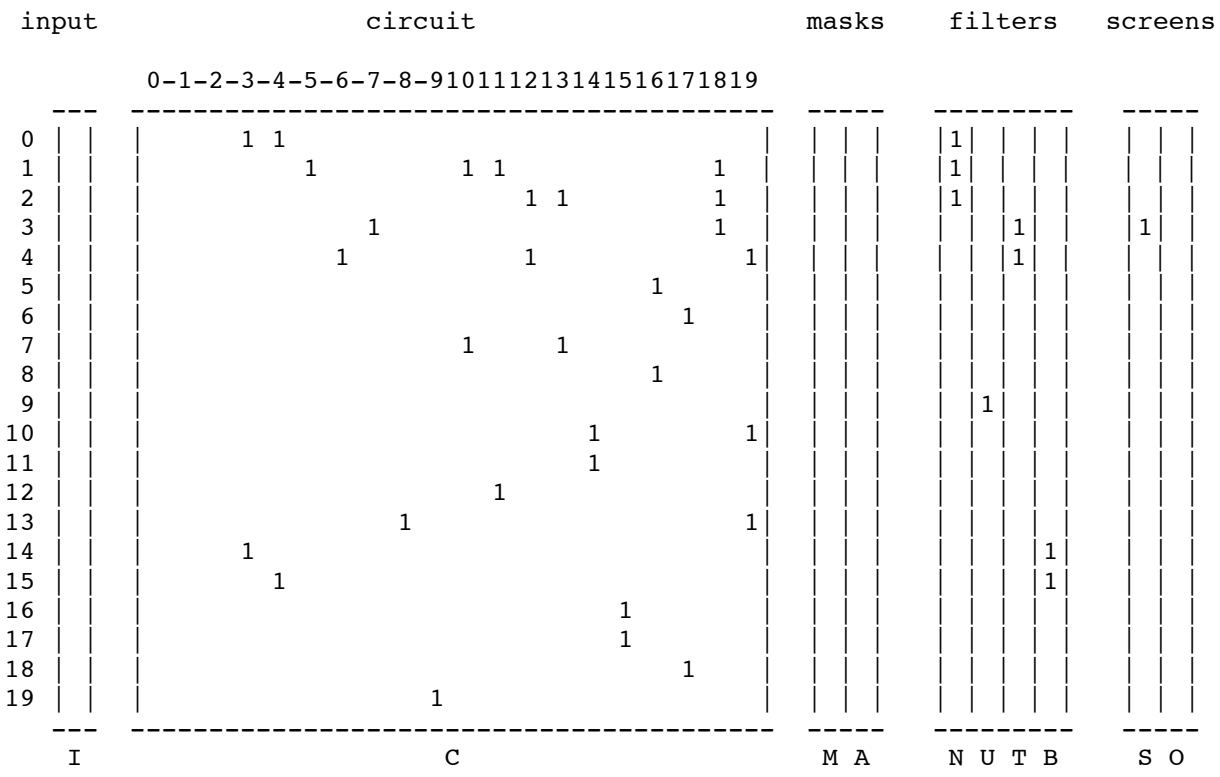
```
((lion (2000 11 1 11 32 0))
 ((main)
 ((0 unk)(1 unk) (2 unk) )
 ((oa 9))
 (( 3 (0 14)  )
  ( 4 (0 15)  )
  ( 5 (1)  )
  ( 6 (4)  )
  ( 7 (3)  )
  ( 8 (13)  )
  ( 9 (19)  )
  (10 (1 7)  )
  (11 (1 12)  )
  (12 (2 4)  )
  (13 (2 7)  )
  (14 (10 11)  )
  (15 (16 17)  )
  (16 (5 8)  )
  (17 (6 18)  )
  (18 (1 2 3)  )
  (19 (10 13 4)  ) ))
```

```
 0-1-2-3-4-5-6-7-8-910111213141516171819

(       1 1                                  )   0
(           1          1 1              1  )   1
(                          1 1          1  )   2
(               1                       1  )   3
(             1            1              1)   4
(                                    1      )   5
(                                      1    )   6
(                       1      1             )   7
(                                    1      )   8
(                                           )   9
(                                 1         1)   10
(                                 1          )   11
(                           1                )   12
(                   1                       1)   13
(       1                                    )   14
(         1                                  )   15
(                                   1       )   16
(                                   1       )   17
(                                       1   )   18
(                     1                     )   19
```

SIMULATION OF LION CIRCUIT

Prior to evaluation, the loaded occlusion array for the LION circuit looks like this:



For demonstration, the register in row 3 is assumed to be in a marked state from the previous evaluation cycle.

The operations to load a circuit functionality follow.

LOAD FUNCTIONALITY

I. Size and build array
II. Build 9 vectors
III. Load circuit into array
IV. Load filters:
input, output, register-top, register-bot

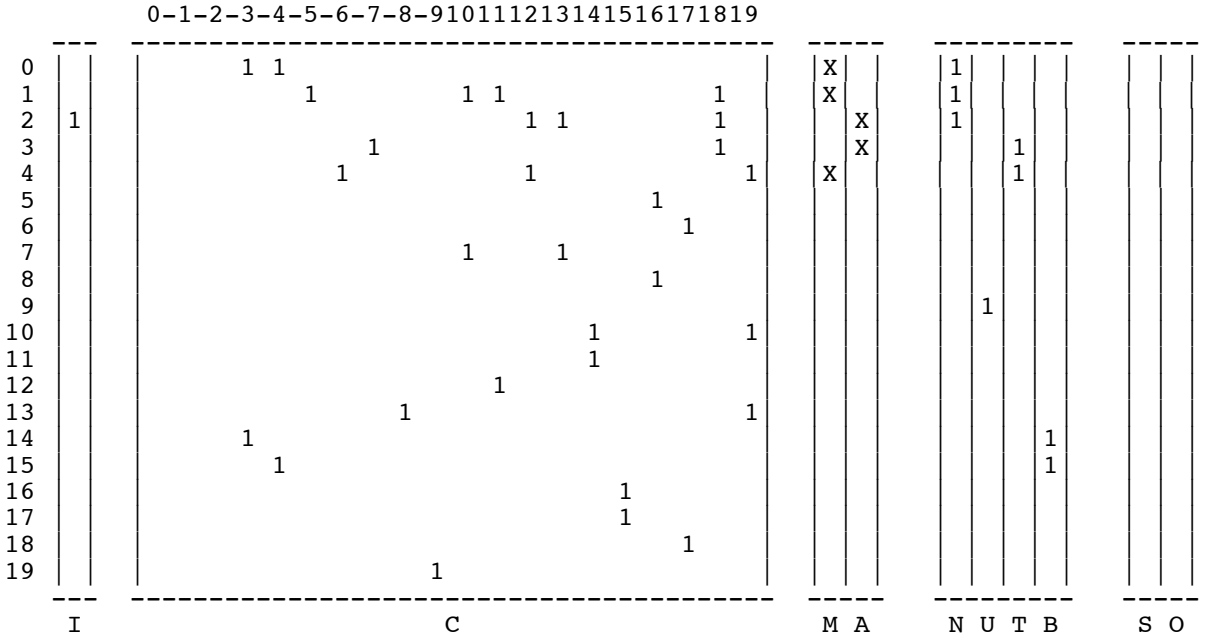
Row 0 is the clock, it equates with variable "a" in the input binding list. The clock logic is meta to the table logic; the zero row is not used during evaluation.

Since initialization is invariant over evaluation, all filters are pre-computed. Although inputs and registers can be computed from the array

(inputs = all empty columns, registers = row 0), the same is not true for outputs. The main output has an empty row, but intermediate outputs may not.

The array is initialized for evaluation. Assume a binding of

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



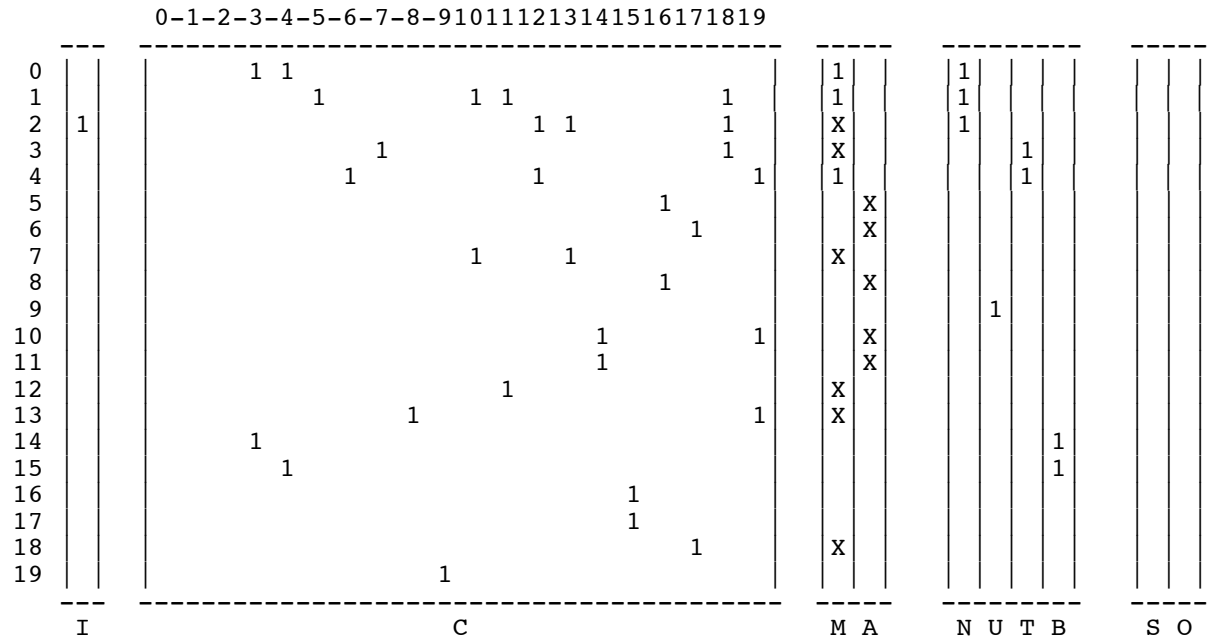
The operations which constitute initialization follow. All are parallel:

INITIALIZATION

1. Mark the input-vector with positive inputs {2}
2. Mark the active-vector with positive inputs {2}
3. Mark the mask with negative inputs {0,1}
4. Mark the active-vector with positive states {3}
5. Mark the mask with negative states {4}
6. When clock=0, reset the state-vector to 0.

The input vector meant to represent the clocked and bussed bit stream entering the occlusion array as input. It is not used during computation.

The evaluation loop now begins.



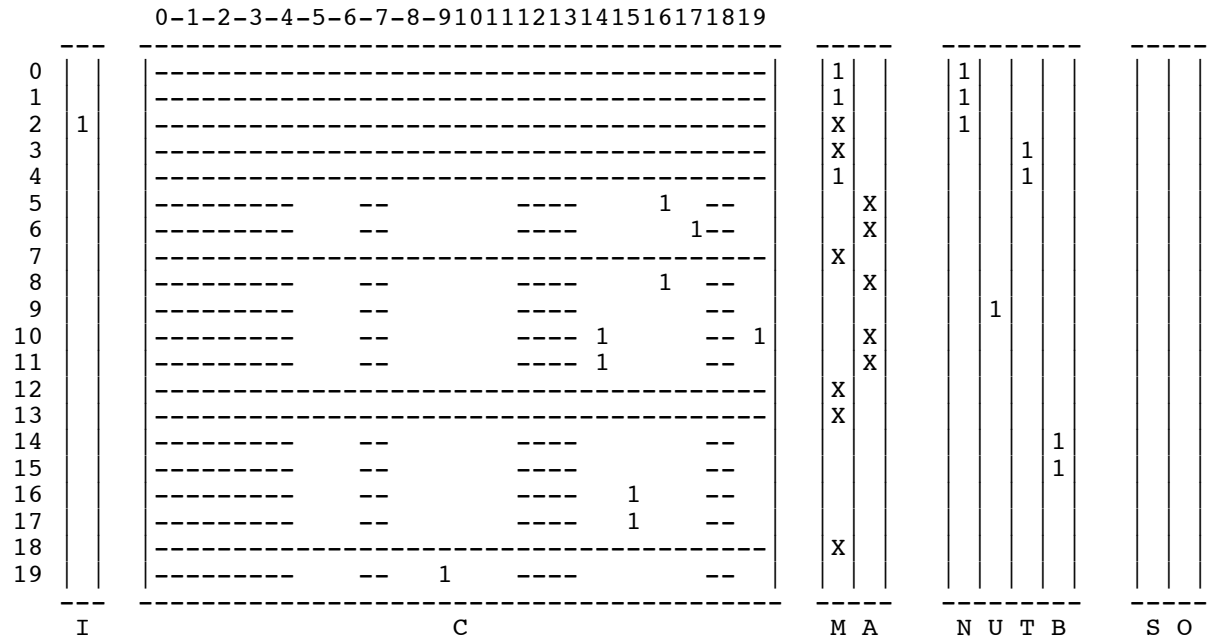
The operations for each evaluation cycle follow:

CYCLE 1

1-7.	Mark the output-vector with active outputs	(none)
1-8.	When clock=0, mark the state-vector with register-bots marked in active rows	(none)
1-9.	Mask active rows	{2,3}
1-10.	Mask rows marked in active rows	{7,12,13,18}
1-11.	Check for all masked	(fails)
1-12.	Clear active (optional)	
1-13.	Mark as active all unmasked empty columns	{5,6,8,10,11}
1-14.	Repeat	

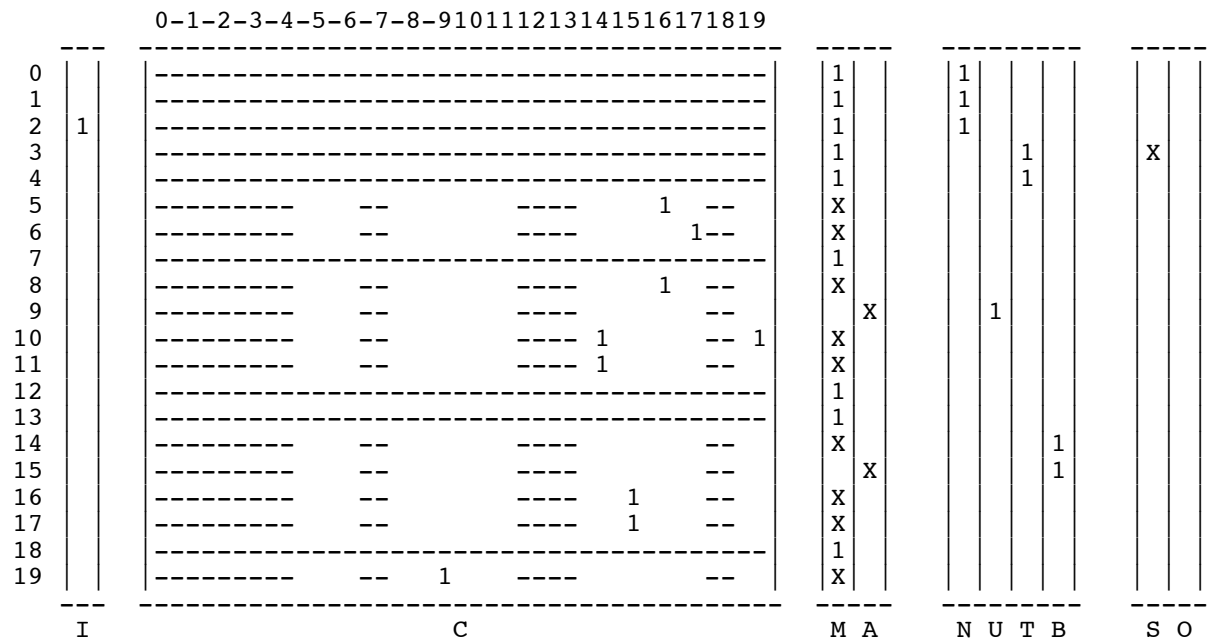
The clock value in Step 1-8 (clock=0) is necessary to keep the occlusion array output synchronized with test vectors designed for a conventional two-state clock. Further refinement of register placement and necessity should remove this two-state clock model from the occlusion array algorithm. Register minimization and retiming are currently open questions.

As a visual aid for step 1-13 (activating empty columns), all prior masked rows and columns have been shaded out in the following:



By inspection, columns {5,6,8,10,11} are empty.

The second cycle leaves the array in the following state:



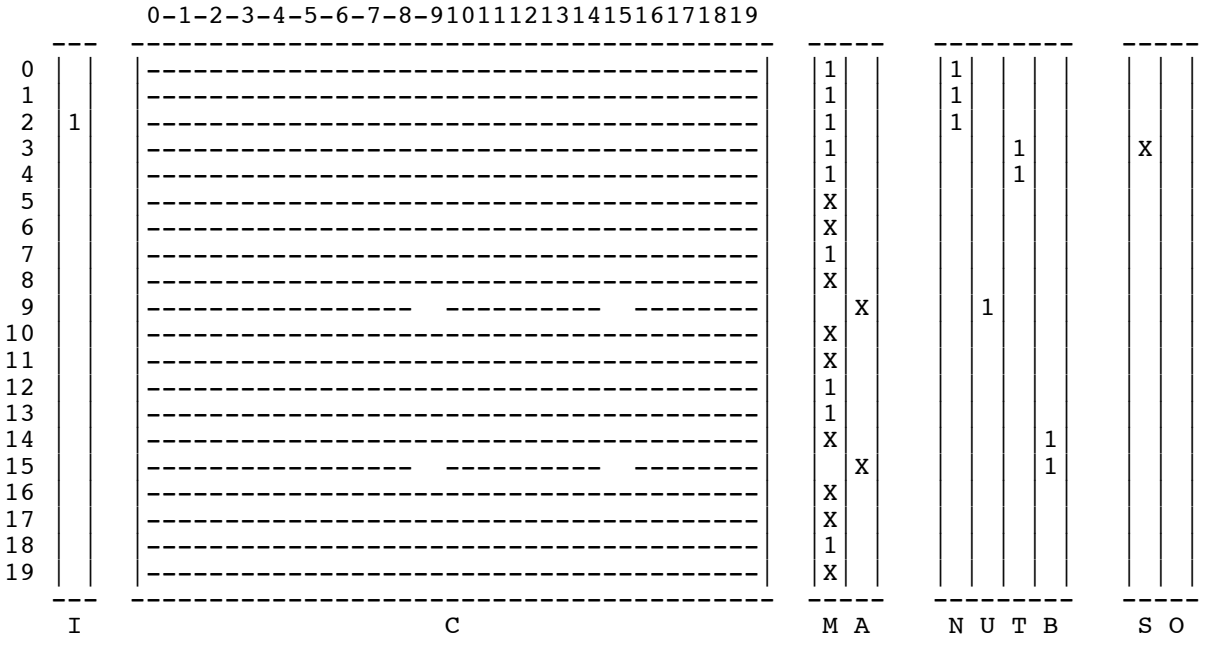
CYCLE 2

2-7.	Mark the output-vector with active outputs	(none)
2-8.	When clock=0, mark the state-vector with register-bots marked in active rows	{3}
2-9.	Mask active rows	{5,6,8,10,11}
2-10.	Mask rows marked in active rows	{14,16,17,19}
2-11.	Check for all masked	(fails)
2-12.	Clear active	
2-13.	Mark as active all unmasked empty columns	{9,15}
2-14.	Repeat	

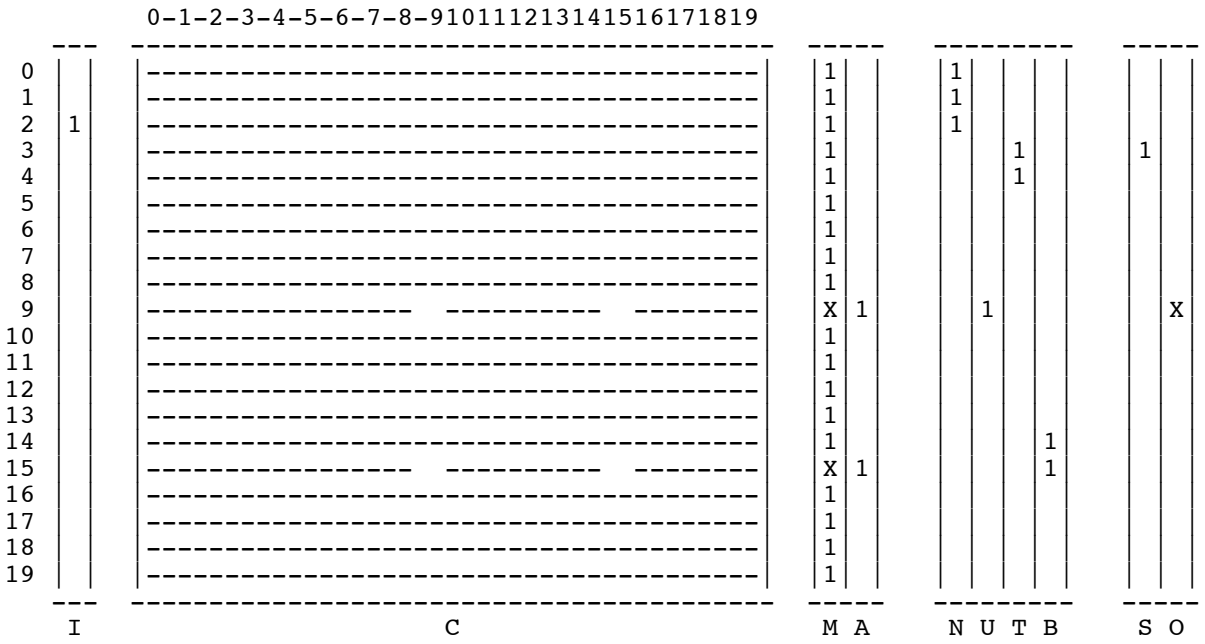
The register-bots are rows 14 and 15. The column for register-bot 14 is marked in the active rows 10 and 11, so the state of the corresponding top (row 3) is marked in the state vector.

Again the all-masked check fails, and empty rows are activated.

A visualization of what remains as empty rows are collected in Cycle 2. Four empty cells remain. Since both rows are empty, both rows have been activated.



Cycle three begins:



CYCLE 3

3-7.	Mark the output-vector with active outputs	{9}
3-8.	When clock=0, mark the state-vector with register-bots marked in active rows	(none)
3-9.	Mask active rows	{9,15}
3-10.	Mask rows marked in active rows	(none)
3-11.	Check for all masked. Terminate.	(succeeds)

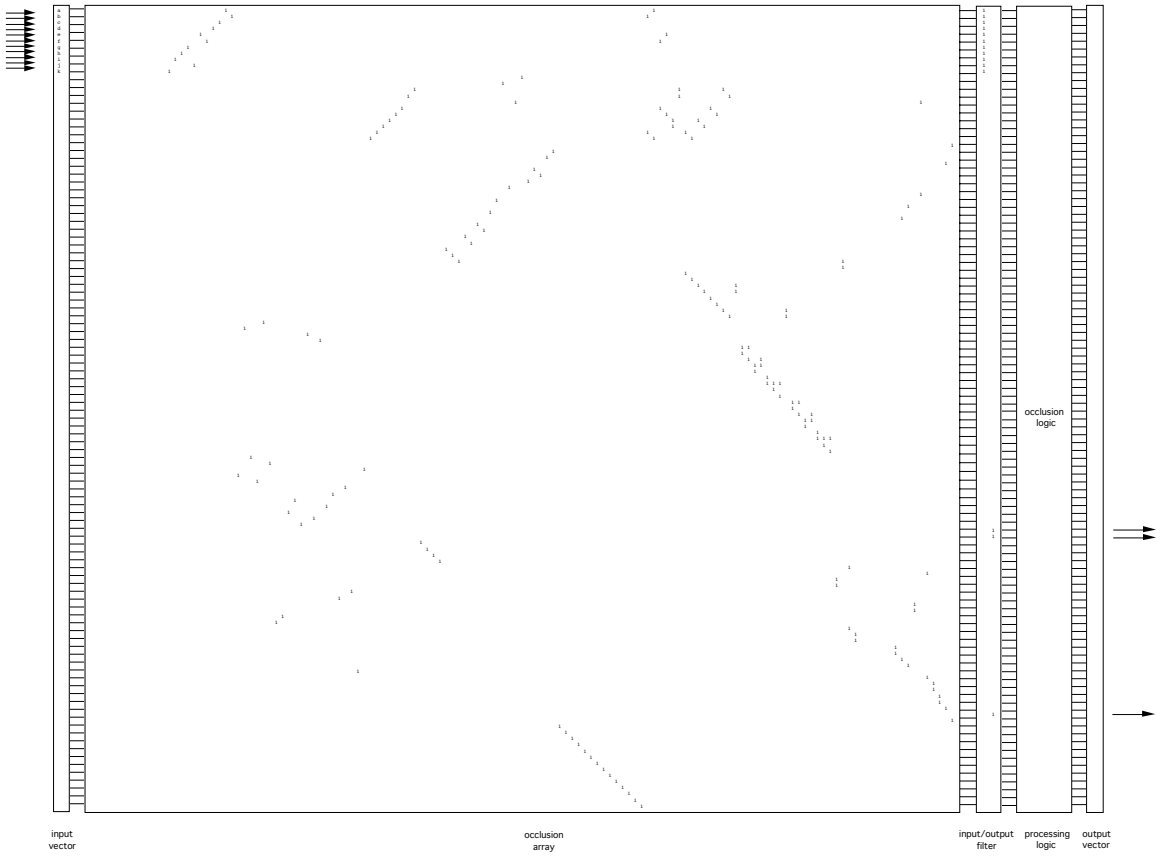
The all-masked test succeeds, and the output result can be read from the output vector O, while the register states are stored in the state vector S.

SCHEMATICS

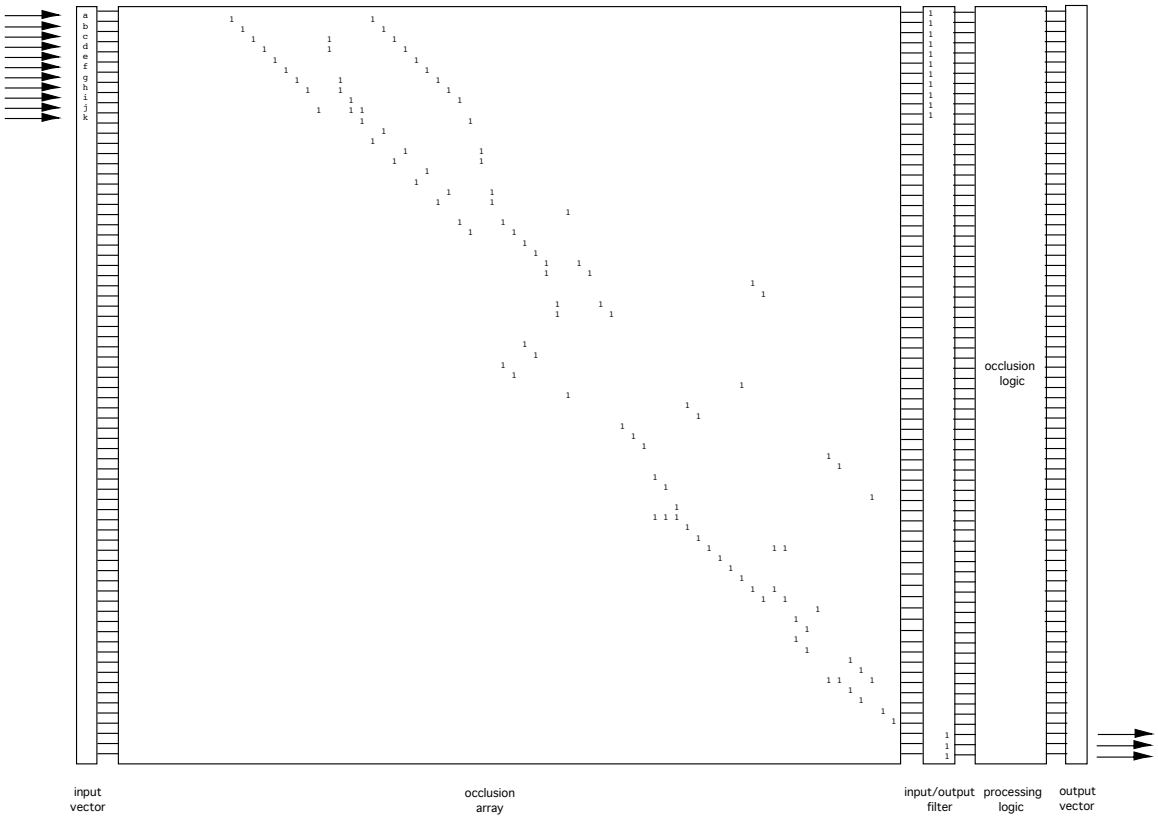
Five schematics of the Occlusion Array routing for the same circuit, cm85a, a 4-bit magnitude comparator with enables. The schematics show how parens reduction effects the size and layout of occlusion arrays.

Schematic I:	the unoptimized benchmark circuit
Schematic II:	diagonalized with trivial redundancy removed
Schematic III:	optimized two-level PLA form
Schematic IV:	optimized for conventional ASIC gates
Schematic V:	optimized for occlusion array

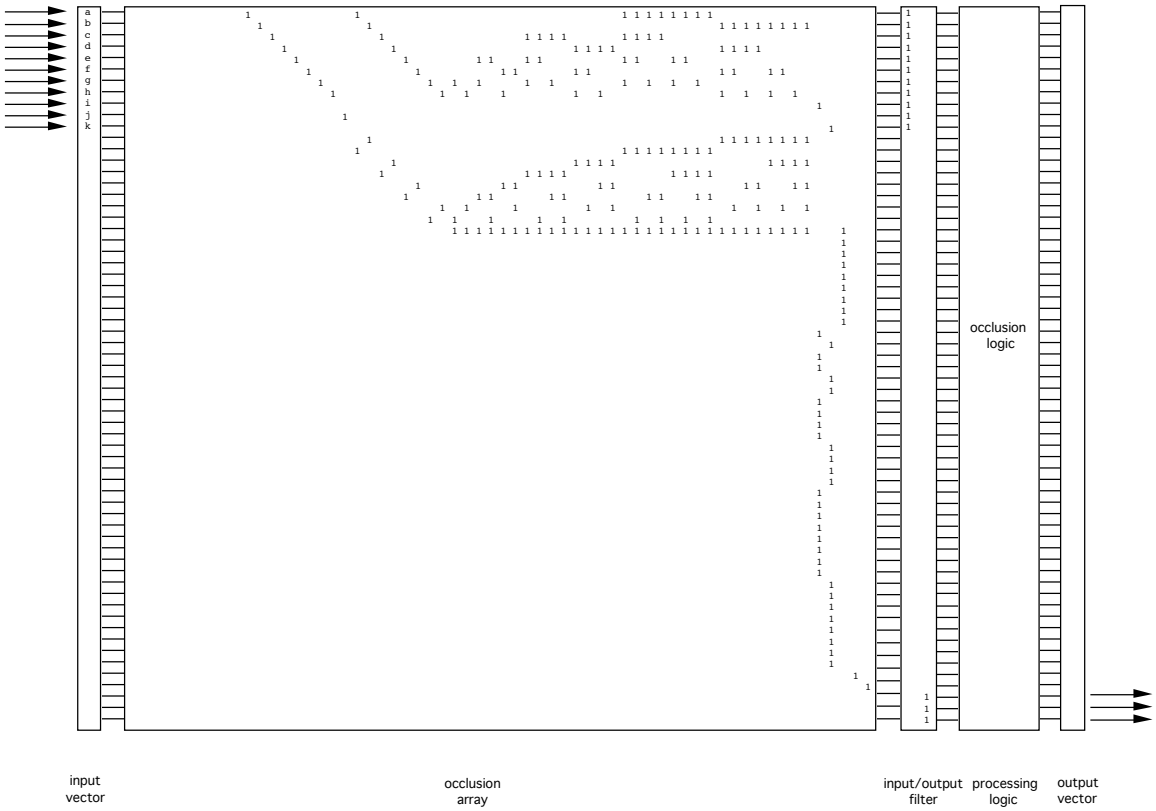
SCHEMATIC I: Occlusion Array (raw multilevel benchmark)



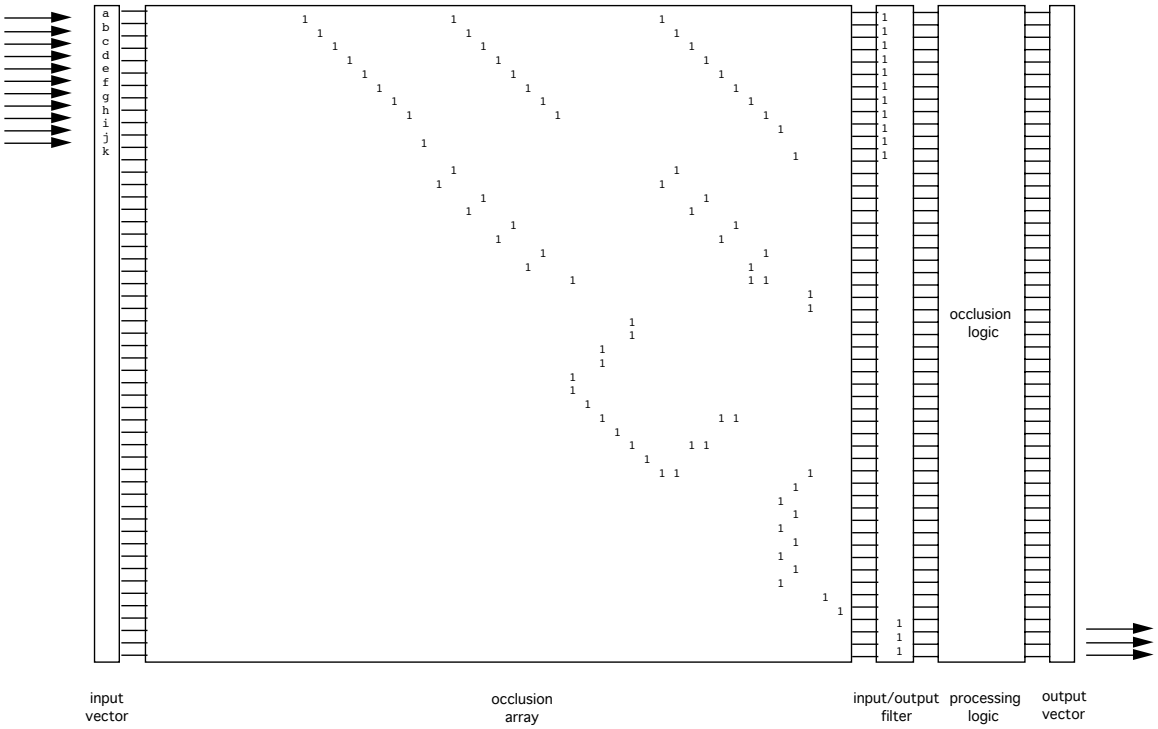
SCHEMATIC II: Occlusion Array (clean multilevel benchmark)



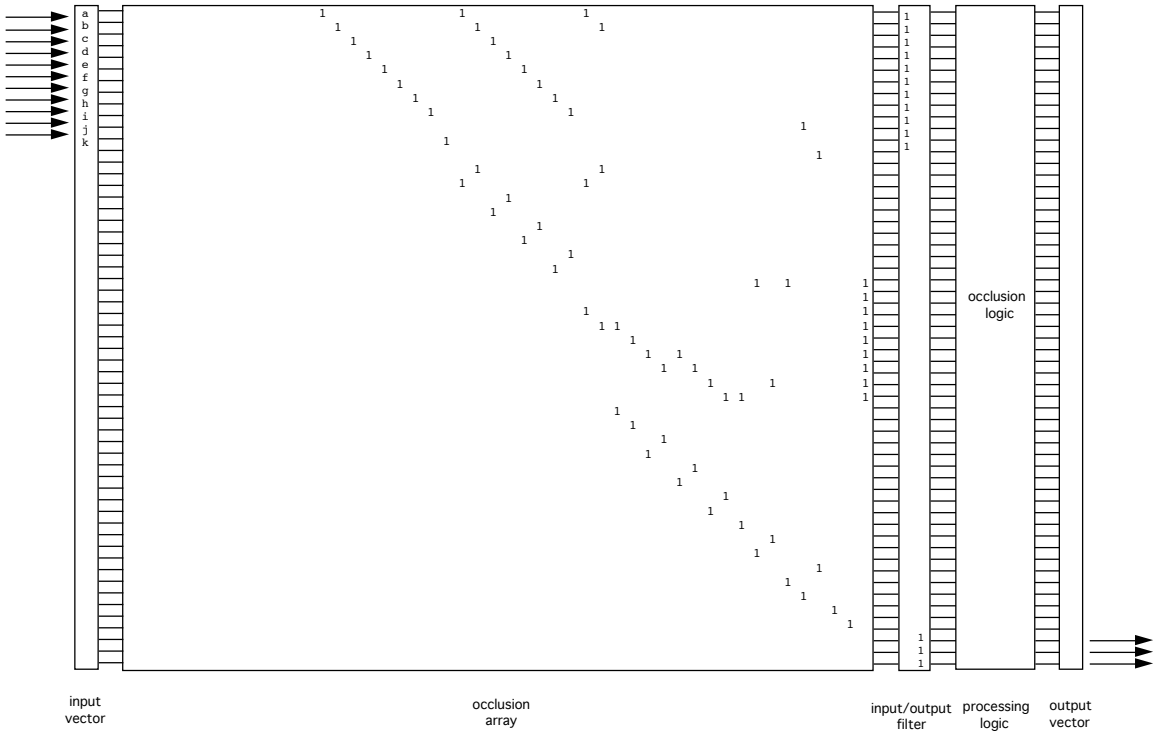
SCHEMATIC III: Occlusion Array (2-level distinction network)



SCHEMATIC IV: Occlusion Array (optimized for conventional ASIC)



SCHEMATIC V: Occlusion Array (optimized for occlusion array)



ENCODING

Generally, the same organization can be realized by many different structures. There are many different pun circuits for any given logical functionality. This variety was addressed in the previous section by contrasting raw arrays with reduced arrays.

Several other strategies for optimizing occlusion arrays follow.

- Parens-Pun Form
  A parens network with logic clustered into one parens expression.

- CNF-Pun Form
  Parens forms with a minimal depth of nesting..

- Deep-coding
  Adding all the possible wires which do not undermine the intended functionality.

- Relation Matrices
  Matrix transformations on occlusion arrays.

- Array Subdivision
  NxN array components which can be ganged together to form a larger array.

Deep-coding, relation matrices, and array subdivision are not discussed in this paper.

PARENS-PUN FORM

The occlusion array used above has one unnecessary mark (at <13,8>). That is, it is not fully optimized. Optimization in the space of circuit functionality is multi-dimensional, so at best a satisficing solution to a set of constraints is sought. Satisficing solutions are those within an acceptable tolerance with regard to physical resources.

In this section, another, slightly different, optimized form is presented, this one derived from the parens-pun form of a circuit. In a parens-pun network, nodes are permitted to contain arbitrary parens expressions, rather than single distinctions.

The parens-pun form of a combinational circuit is one single (sometimes large) parens expression. The parens-pun form of a sequential circuit requires several parens forms, thus the pun format is necessary for this representation. Each register requires a separate parens form to avoid

reentrant expressions. The minimal parens-pun form for sequential circuits consists of a parens expression for each register and for each output. This is equivalent to clustering the combinational logic between registers into a single node. For the LION body this is:

```
(( 3 (0 14)  )
 ( 4 (0 15)  )
 ( 9 (( 4 ((3) ((1)(2))) ))  )
 (14 (( 1 (2 3 4) ))  )
 (15 ((2 (1) (3)) ((4) (1 2 3)))  ))
```

Node 9 is the output.

Note that each register is represented by a register-top {3,4}, which has the clock input, and a register-bot {14,15}, which accumulates state. The two-dnode register constraint essentially blocks the occurrence of self-referential dnodes.

Converting the above parens-pun form back to a distinction network results in a slightly different homogeneous pun and a slightly different array.

(Below, some distinctions have been renumbered relative to the parens-pun LION body above.)

```
(( 3 (0 14)  )
 ( 4 (0 15)  )
 ( 5 (1)  )
 ( 6 (2)  )
 ( 7 (3)  )
 ( 8 (4)  )
 ( 9 (10)  )
 (14 (13)  )
 (10 (4 11)  )
 (11 (7 12)  )
 (12 (5 6)  )
 (13 (1 16)  )
 (15 (17 18)  )
 (16 (2 3 4)  )
 (17 (2 5 7)  )
 (18 (8 19)  )
 (19 (1 2 3)  ))
```

For comparison, the occlusion array for this alternative circuit follows. It too has 20 rows and 31 marks, however with free inverters, only 12 rows are needed. Alternative optimizations do not necessarily result in an array of the same size or with the same number of marks. It is very difficult to perceive visual similarities in different functionally equivalent arrays (that is, they are difficult to read). This is partially due to the fact that rows and columns can be in any location without changing the functionality of the array.

```
 0-1-2-3-4-5-6-7-8-910111213141516171819

(       1 1                                     )   0
(           1                   1             1)   1
(             1                       1 1     1)   2
(               1                     1       1)   3
(                 1   1               1        )   4
(                         1           1       )   5
(                         1                   )   6
(                       1               1     )   7
(                                         1   )   8
(                                             )   9
(                     1                       )   10
(                       1                     )   11
(                         1                   )   12
(                                 1           )   13
(       1                                     )   14
(         1                                   )   15
(                             1               )   16
(                                 1           )   17
(                                 1           )   18
(                                       1     )   19
```

CNF FORM

Structurally rearranging a circuit so that all logic between gates is in CNF (Conjunctive Normal Form) form permits pipelining of array computations. CNF form has only two layers of logic, it is the conjunction of the disjunction of input values (positive and negative). By limiting the layers of logic, the maximum number of cycles required for complete evaluation is standardized at three (2 layers of logic and 1 register bottom).

CNF is also known as Product of Sums. Another standard representation is Sum of Products (SOP), which would correspond to the logical Disjunctive Normal Form. Very conveniently, the two varieties of representational focus (AND-centered vs. OR-centered) are a single form in parens notation.

Logic	*Parens*
a AND b	((a)(b))
NOT a OR NOT b	(a)(b)

Here, the conversion from CNF to DNF is by removing the outer container.

DeMorgan transformations are notationally indistinguishable in parens form. One might say that the transformation is "built-in". However, the boundary math perspective is that such ransformations do not exist.

a AND b = NOT((NOT a) OR (NOT b))	DEMORGAN
((a)(b)) = ((a)(b))	Parens

The starting point for generating a CNF form is again the parens-pun form of the LION circuit:

```
(( 3 (0 14)  )
 ( 4 (0 15)  )
 ( 9 (( 4 ((3) ((1)(2))) ))  )
 (14 (( 1 (2 3 4) ))  )
 (15 ((2 (1) (3)) ((4) (1 2 3)))  ))
```

Each row in this body can be converted to CNF form, yielding:

```
(( 3 (0 14)  )
 ( 4 (0 15)  )
 ( 9 ((3 4) (4 (1)(2)))  )
 (14 (( 1 (2 3 4) ))  )
 (15 (( (4 (2)) (1 4) (3 4) (1 2 3) ))  )
```

Converting this CNF-pun form into a distinction network and homogenizing via renumbering yields a third variation:

```
(( 3 (0 8)  )
 ( 4 (0 9)  )
 ( 5 (10 11)  )
 ( 6 (1)  )
 ( 7 (2)  )
 ( 8 (14)  )
 ( 9 (17)
 (10 (3 4)  )
 (11 (4 6 7)  )
 (12 (2 3 4)  )
 (13 (4 7)  )
 (14 (1 12)  )
 (15 (1 4)  )
 (16 (1 2 3)  )
 (17 (13 15 10 16)  ))
```

The input variables are ((0 unk)(1 unk)(2 unk)). The output is ((oa 9)).

The accompanying occlusion array has 18 rows and 31 marks; without inverters, it has 12 rows.

```
 0-1-2-3-4-5-6-7-8-91011121314151617

(       1 1                                 )   0
(             1                 1 1 1       )   1
(               1           1       1       )   2
(                     1     1       1       )   3
(                     1 1 1 1   1           )   4
(                                           )   5
(                       1                   )   6
(                       1   1               )   7
(       1                                   )   8
(         1                                 )   9
(           1                             1 )   10
(           1                               )   11
(                               1           )   12
(                                         1 )   13
(                   1                       )   14
(                                         1 )   15
(                                         1 )   16
(                     1                     )   17
```

For small and moderate circuits (<~1000 rows) the combinational explosion of CNF forms is largely avoided by CNF optimization using the Losp engine.

Technically a CNF form has a mention of every input in every clause. This results in a homogeneous data structure which is well suited for the PLA logic circuitry of decades ago. The research community currently believes that deeply nested forms, INF forms, are necessary for today's large circuits.

The number of dnodes in a CNF form is the number of clauses. This may be less than the number of dnodes in a functionally identical INF form, as is the case here.

Optimizing CNF forms can reduce the number of clauses. Reducing the number of literal mentions is irrelevant for occlusion arrays, since more mentions add more marks but not more rows.

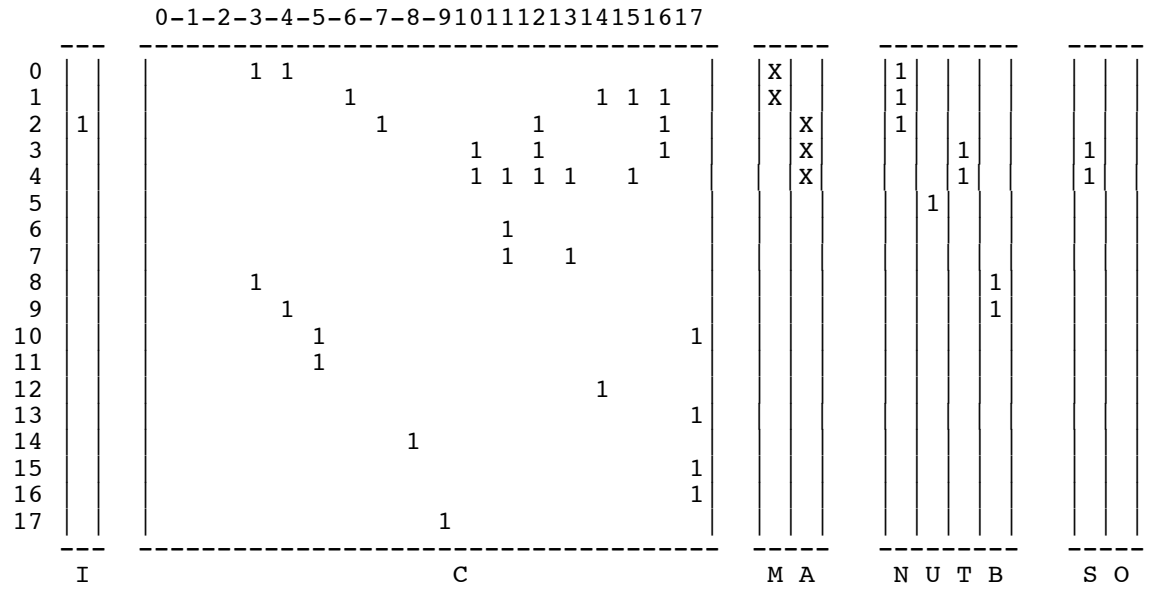
SIMULATION OF CNF LION CIRCUIT

This example illustrates the primary characteristics of CNF coding:

1. Positive outputs and register states are asserted in the final cycle.

2. The maximal number of cycles is 3.

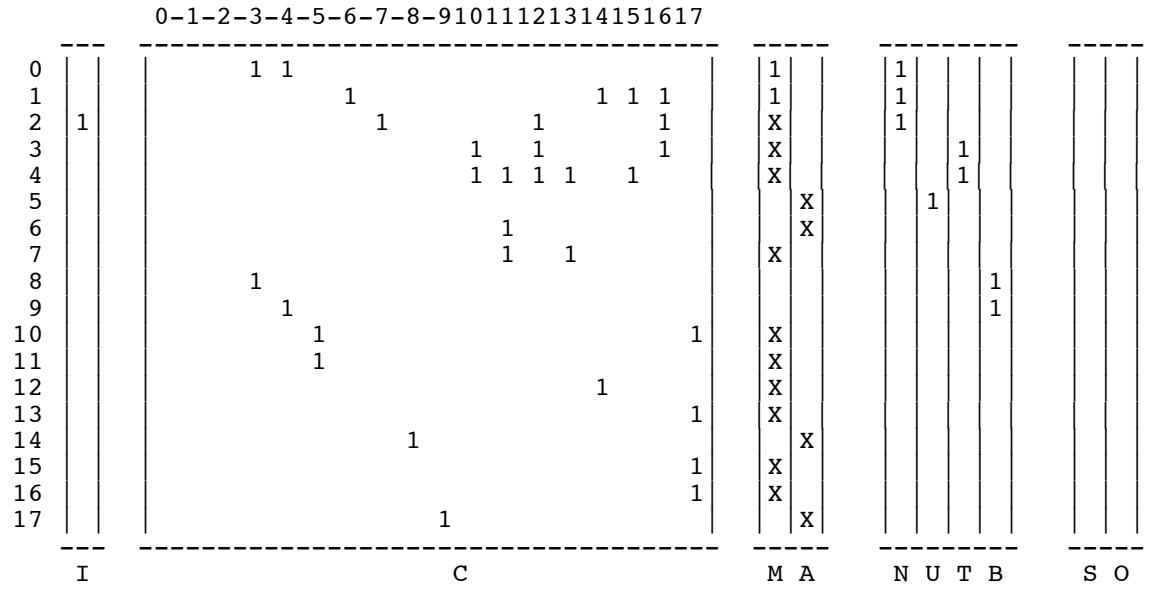
The loaded occlusion array for the CNF form of LION follows. The input values {(a 0),(b 0),(c 1)} have been asserted as initialization, as well as the register states {(3 1),(4 1)}. These input values differ from those used for the previous simulation of LION, and illustrate a positive output.



INITIALIZATION

1. Mark the input-vector with positive inputs {2}
2. Mark the active-vector with positive inputs {2}
3. Mark the mask with negative inputs {0,1}
4. Mark the active-vector with positive states {3,4}
5. Mark the mask with negative states (none)
6. When clock=0, reset the state-vector to 0.

The evaluation cycle begins, new changes are indicated by X.

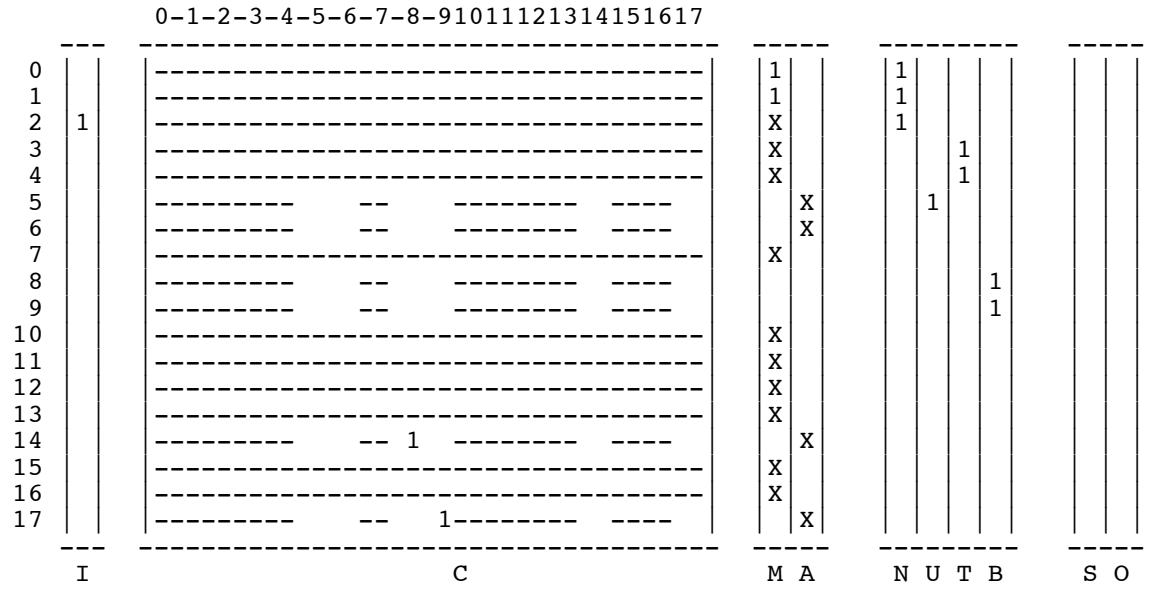


CYCLE 1

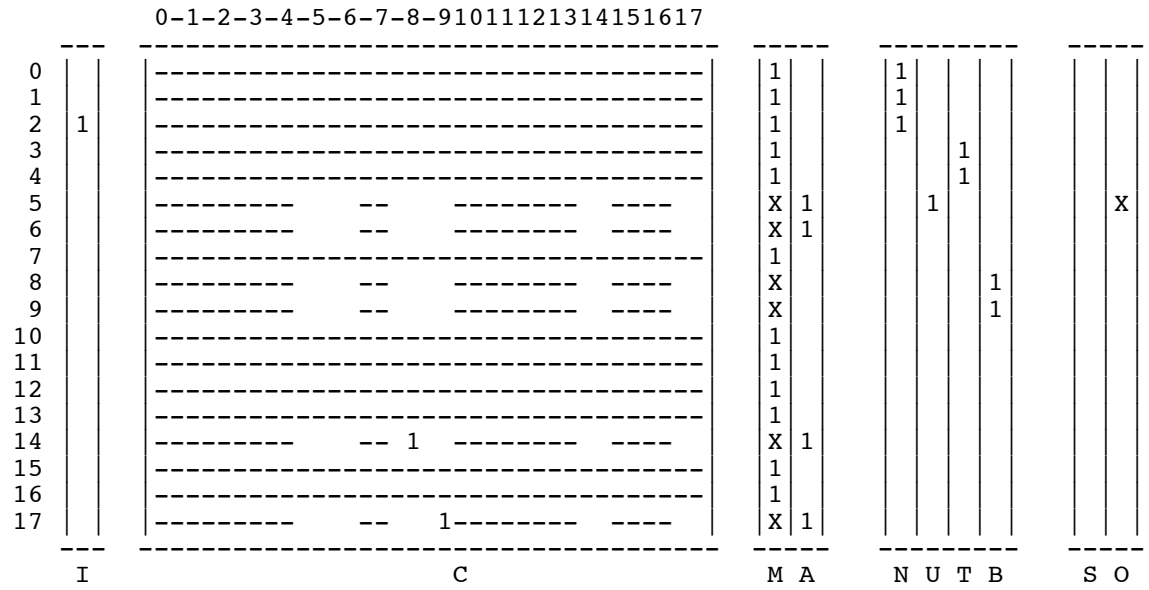
1-7,8.	No outputs are asserted.	
1-9.	Active rows are masked.	{2,3,4}
1-10.	Active row marks are masked.	{7,10,11,12,13,15,16}
1-11.	Termination fails.	
1-13.	Empty columns are marked active.	{5,6,14,17}

The results of the first evaluation cycle are displayed above in the state of the mask and active computational vectors (M and A), and in the output vectors (S and O). All output values are 0.

Below, masked rows are shaded for ease of reading the results for step 1-13.



The second cycle begins:



CYCLE 2

2-7,8.	Output is asserted.	{5}
2-9.	Active rows are masked.	{5,6,14,17}
2-10.	Active row marks are masked.	{8,9}
2-11.	Termination succeeds.	

The resulting output is positive, while both register states are negative.

COMPLEXITY MEASURES

The size and complexity of each occlusion array examined in the previous examples are compared in the following table:

Reduction type	*rows*	*w/o inv*	*marks*	*w/o inv*
Raw circuit	33	14	48	29
Losp reduced	20	13	31	24
Parens-pun reduced	20	12	31	23
CNF	18	12	31	25

The various forms of reduction result in varying array sizes, an indication that minimization is not trivial. The number of marks in an occlusion array is not relevant to the functioning of that array.

In general, optimization of occlusion arrays depends on several factors. The dominant design issue is a classical time-space trade-off. A smaller dnet with fewer distinctions uses less computational area. A deeply nested dnet

usually contains fewer dnodes than a functionally equivalent dnet which is shallower. Deep nesting takes more computational cycles for the worst case input, trading computational time for area.

Thus the dominant measures of complexity for occlusion arrays are size and cycles. Cycles can be reduced by making the circuit shallower; area can be reduced by decreasing the number of dnodes. Often a mean between these two can be measured by a composite measure, the simplest of which is (cycles * size).

CNF form provides time-standardization, at the cost of array size for larger circuits. Array subdivision can provide area-standardization.

OCCLUSION ARRAY ALGORITHM DESCRIPTIONS

Several other ways of describing the occlusion array algorithm follow. All algorithms apply equally to combinational and to sequential circuits. Some of the algorithms have been simplified to illustrate pure combinational code. The algorithms are:

- Five-step
  The algorithm is partitioned into five coherent steps. Description is pseudo-English combined with vector notation.

- Alternative termination branch for five-step
  Termination keys on the presence of active rows rather than an opaque mask. This approach is good for partial evaluation.

- Fine-grain steps
  The algorithm is partitioned into atomic operations. Description is in pseudo-English.

- Vector logic and assignment
  The fine-grain atomic operations are expressed as logical operations on vectors. The results of each operation are assigned to a vector.

- Symbolic vector logic
  Vector logic operations are grouped for each vector assignment, and expressed symbolically.

- LISP source code
  Verified LISP source code which implements the entire occlusion algorithm. The core control code models the five step algorithm.

A double bar, ||, indicates independent steps. An asterisk, *, indicates part of the sequential algorithm.

FIVE-STEP DESCRIPTION

A = active vector
M = mask
O = output screen
S = register state screen
U = output filter
B = register-bot filter

0. INITIALIZATION

Mark positive inputs into A.
Mask negative inputs into M.

1. PROJECT RESULTS

For all rows, when (U=1 and A=1), mark O.
For all rows, when (B=1 and A=1), mark S. *

2. MASK

For all rows, when (A=1),
mark M
and for all marks in each row,
mark M.

3. TERMININATION BRANCH

If M is opaque
then terminate
else

4. COLLECT NEW ACTIVE

For all columns, when (M=1 and column-is-empty),
mark A.

5. RETURN TO 1

ALTERNATIVE TERMINATION BRANCH

The operational termination condition is when rows are no longer scheduled for deletion. This can be expressed either by an opaque (full) mask, or by a transparent (empty) active-vector.

Alternative loop termination:

1. PROJECT RESULTS

2. MASK

3. COLLECT NEW ACTIVE

   For all columns, when (M=1)
   mark A.

4. TERMININATION BRANCH

   If A is empty
   then terminate
   else

5. RETURN TO 1

This approach is necessary whenever a circuit is partially evaluated.

FINE-GRAIN STEPS

```
1.  For every positive input to the system
      mark the input-vector

2.  For every input that is marked in the input-vector
      mark the active-vector
                                                                    ||
3.  For every input that is not marked in the input-vector
      mark the mask
                                                                    ||
4.  For every register-top with marked state                        *
      mark the active-vector                                        *
                                                                    ||
5.  For every register-top with unmarked state                      *
      mark the mask                                                 *

6.  Repeat

7.  For every active row that is an output
      mark the output-vector
                                                                    ||
8.  When clock=0                                                    *
      for every active row                                          *
        for every register-bot marked in that row                   *
          mark the register-top state                               *
                                                                    ||
9.  For every active row
      mark the mask
                                                                    ||
10. For every mark in each active row
      mark the mask

11. When any row is not masked
      for every unmasked row with an empty column
        mark the active-vector

12. When any row is not masked
      goto repeat
```

COMMENTS:

Step 3 could be mask := negative inputs.
Steps {4,5,8} are for sequential circuits only.
The repeat in step 12 is nested under step 11.

VECTOR LOGIC AND ASSIGNMENT

```
1.  input-vector :=
        bindings OR input-vector

2.  active :=
        active OR (input-filter AND input-vector)
                                                                        ||
3.  mask :=
        mask OR (input-filter AND NOT input-vector)
                                                                        ||
4.  active :=                                                           *
        active OR (register-top AND state)                              *
                                                                        ||
5.  mask :=                                                             *
        mask OR (register-top AND NOT state)                            *

6.  repeat

7.  output-vector :=
        output-vector OR (active AND output-filter)
                                                                        ||
8.  when clock=0,  register-top-state :=                                *
        state OR (active AND (row-of-active AND register-bot))          *
                                                                        ||
9.  mask :=
        mask OR active
                                                                        ||
10. mask :=
        mask OR (active AND row-of-active)

11. branch:  (APPLY OR mask)
        when F  exit

12.  T  active :=
          active OR (NOT mask AND NOT (APPLY OR column))

13   T  goto repeat
```

SYMBOLIC VECTOR LOGIC

Vector assignment operations have been combined into single assertions where possible. The distinction between combinational and sequential steps is lost. This description is intended to be closest to the physical logic wiring in the occlusion array substrate.

Ci refers to the row vector indexed by A.

Cj refers to the column vector indexed by not M.

Operations

```
      I := BINDINGS or I

      A := A or (N and I) or (T and S)
                                                                  ||
      M := M or (N and not I) or (T and not S)

      loop

            O := O or (A and U)
                                                                  ||
            when clock=0,
              S := S or (A and Ci and B)
                                                                  ||
            M := M or A or Ci

            if not (apply or M)
              exit
              A := A or (not M and not (apply or Cj))
```

LISP SOURCE CODE

```
(defun evaluate-circuit ()
  (initialize-vectors)
  (evaluate-circuit-loop))

(defun initialize-vectors ()
  (build-mask)
  (build-active)
  (build-in)
  (build-out)
  (initialize-in)
  (initialize-mask-and-active)
  (when (and *cam-clock* (= 0 *cam-clock*))
    (build-ffout)) )


(defun initialize-in ()
  (do ((i 0 (incf i))
       (binds *cam-input* (cdr binds)))
      ((= i *array-size*))
    (when (and binds (= 1 (cadar binds)))
      (setf (sbit *in* i) 1))))


(defun initialize-mask-and-active ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (= 1 (sbit *inmask* i))
      (if (= 1 (sbit *in* i))
        (setf (sbit *active* i) 1)
        (setf (sbit *mask* i) 1)))
    (when (and *cam-clock* (= 1 (sbit *ffmask* i)))
      (if (= 1 (sbit *ffout* i))
        (setf (sbit *active* i) 1)
        (setf (sbit *mask* i) 1)))))


(defun evaluate-circuit-loop ()
  (set-out)
  (set-ffout)
  (set-mask)
  (when (some-unmasked)
    (set-active)
    (evaluate-circuit-loop)))


(defun set-out ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 1 (sbit *outmask* i))
               (= 1 (sbit *active* i)))
      (setf (sbit *out* i) 1))))
```

```
(defun set-ffout ()
  (when (and *cam-clock* (= 0 *cam-clock*))
    (do ((i 0 (incf i)))
        ((= i *array-size*))
      (when (= 1 (sbit *active* i))
        (do ((j 0 (incf j)))
            ((= j *array-size*))
          (when (and (= 1 (sbit *ffinmask* j))
                     (= 1 (aref *array* i j)))
            (setf (sbit *ffout* (ff-dereference j)) 1)))))))


(defun ff-dereference (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*))
    (when (and (= 1 (sbit *ffmask* j))
               (= 1 (aref *array* i j)))
      (return j))) )


(defun set-mask ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (= 1 (sbit *active* i))
      (setf (sbit *mask* i) 1)
      (do ((j 0 (incf j)))
          ((= j *array-size*))
        (when (= 1 (aref *array* i j))
          (setf (sbit *mask* j) 1))))))


(defun set-active ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 0 (sbit *mask* i))
               (empty-col i))
      (setf (sbit *active* i) 1))))


(defun some-unmasked ()
 (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (not (= 1 (sbit *mask* i)))
      (return T))))


(defun empty-col (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*) T)
    (when (and (not (= 1 (sbit *mask* j)))
               (= 1 (aref *array* j i)))
      (return nil))))
```

ALTERNATIVE OCCLUSION ARRAY ALGORITHM (NO ACTIVE VECTOR)

The algorithm below uses look-ahead in order to stay O[N^2]. It assumes that register-tops are found by searching the rows of register-bots. Computational complexity can be exchanged for a better data structure. If association lists can be built in hardware, then this alternative algorithm is unnecessary. It differs only slightly from the previous algorithm, in implementation of erasure, and in dereferencing register components (register-top and register-bot).

These versions follow:

- Fine-grain steps
- Vector logic and assignment
- Vector logic and assignment for combinational circuits
- Pseudo-English code
- LISP source code
- LISP source code for combinational circuits

The main versions apply to both sequential and combinational circuits. Condensed algorithms which address only combinational circuits are included also.

The alternative algorithm is illustrated using the MAJORITY circuit.

ALTERNATIVE FINE-GRAIN STEPS

1. Initialize the input-vector with positive input values.

2. Initialize the mask with negative input values.

3. Mask all register-tops.

4. When a register state is positive,
   mask all rows marked in the register's row.

5. When a register state is positive and the register-top is an output,
   mark the output vector.

6. When a row is not masked and the column is empty,
   when the row is an output, mark the output vector

7. When a row is not masked and the column is empty,
   when the row is a register-bottom and the clock=0,
   mask the register-top.

8. When a row is not masked and the column is empty,
   mask the row.

9. When a row is not masked and the column is empty,
   mask all rows marked in the particular row.

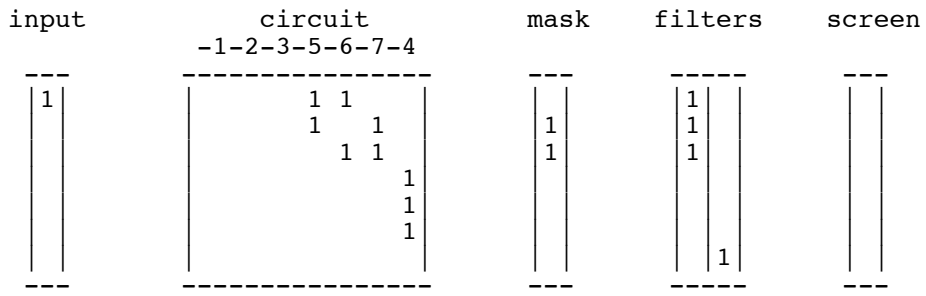
10. When the clock=0 and a row is a register-bot and that row is masked,
    when the register-top is not masked, mark the register state

11. When the clock=0 and a row is a register-bot,
    register-mask the register-top.

12. Go to 6.

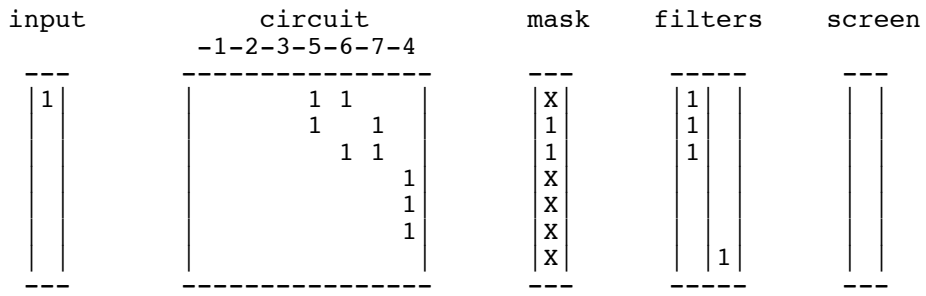
MAJORITY EXAMPLE, ALTERNATIVE OCCLUSION ARRAY MODEL

The loaded occlusion array for the MAJORITY circuit, with the input bindings: {(a 1),(b 0),(c 0)}



The difference in the two algorithms for initiation is that without an active vector, only the negative input is masked.

At this point the computational cycle begins.



CYCLE 1

1. Mark output (none)
2. Mask empty columns {1,7}
3. Mask marks in each empty column {4,5,6}
4. Opaque mask? Terminate.

Here, rows 1 and 7 are masked because their columns are empty (after ignoring all masked rows). Row 4 is masked because it is marked in row 7; rows 5 and 6 are masked because they are marked in row 1.

When the output row 4 became masked, it was as a consequence of row 7 being active. In such cases the output-vector does not have an opportunity to become marked. It is in effect set to zero by the absence of an action.

Since the mask is fully marked, the evaluation is over. The output has nothing in row 4, so its value is False.

ALTERNATIVE VECTOR LOGIC AND ASSIGNMENT

Below, read ":" as "which meet the following test(s)"

Initialize (all in parallel)

```
      input-vector :=  all input values: positive
                                                                          ||
      mask := all input values: negative
                                                                          ||
      mask := all rows: register-tops                                     *
                                                                          ||
      mask := all rows:                                                   *
                  marked in any other row:                                *
                        register and positive-state.                      *
                                                                          ||
      output := all rows:                                                 *
                 register-top and positive-state and is-an-output         *
```

Repeat when any row is not masked

```
      output := all rows: not-masked and empty-column and is-an-output

      when clock=0                                                        *
        mask := all register-tops:                                        *
                  register-bot is (not-masked and empty-column)           *

      mask := all rows: not-masked and empty-column.
                                                                          ||
      mask := all rows:
                  marked in any other row:
                        not-masked and empty-column.

      when clock=0                                                        *
        register-state := all rows:
            register-bot and masked and register-top not masked           *
                                                                          ||
      when clock=0                                                        *
        register-mask := all rows:  if register-bot, register-top         *
```

ALTERNATIVE VECTOR LOGIC AND ASSIGNMENT (COMBINATIONAL)

```
Initialize  (all in parallel)

      input-vector :=  all input values: positive
                                                                              ||
      mask := all input values: negative


Repeat when any row is not masked

      output := all rows: not-masked and empty-column and is-an-output

      mask := all rows: not-masked and empty-column.
                                                                              ||
      mask := all rows:
                  marked in any other row: not-masked and empty-column.
```

ALTERNATIVE PSEUDO-ENGLISH CODE

In this pseudo-English version, the long-names hide particular variables, loops, and functional complexities. Since the map from long-name to source code structure is 1-to-1, this code can be transcribed directly into the original source code, and is therefore executable when compiled.

```
(define initialize-in
  (for-every-row
    (when (the-row-is-an-input and the-input-is-positive)
      mark-the-input-vector)))


(define initialize-mask
  (for-every-row
    (when (the-row-is-an-input and the-input-is-negative)
      mask-the-row)
    (when the-row-is-a-register-top
      mask-the-row
      (when the-register-state-is-positive
        (when the-register-is-an-output
          mark-the-output-vector)
        mask-the-containers of-the-register-top))))


(define mask-the-containers (row)
  (for every column
    (when the-row-is-contained-by-the-column
      mask-the-container)))


(define repeat-evaluation
  (when (there-are-unmasked-rows)
    (mask-more-rows)
    (mark-register-states)
    (repeat-evaluation)))


(define mask-more-rows
  (for-every-row
    (when (the-row-is-not-masked and the column-is-empty)
      (when the-row-is-an-output mark-the-output-vector)
      (mark-the-register-mask-for-register-bots of-the-row)
      mask-the-row
      mask-the-containers of-the-row) ))
```

```
(define mark-register-states
  (when the-clock-ticks
    (for-every-row
      (when (the-row-is-a-register-bot and the-row-is-masked)
        (for-the-register-top
          (when the-register-is-not-masked mark-the-register-state)
          mark-the-register-top-mask))))


(define mark-the-register-mask-for-register-bots (row)
  (when (the-clock-ticks and the-row-is-a-register-bot)
    (for-the-register-top from-the-row
      mark-the-register-mask))


(define for-the-register-receiving-input (row)
  (locate-the-register-receiving-input-by-searching-the-column))


(define there-is-more-to-do
  (for-every-row
    (when the-row-is-not-masked there-is-more-to-evaluate)))


(define the-column-is-empty (row)
  (for-every-row
    (when (the-column-is-not-masked and the-column-is-marked-in-the-row)
      the-column-is-not-empty)))
```

ALTERNATIVE SOURCE CODE

```
(defun initialize-in ()
  (do ((i 0 (incf i))
       (binds *cam-input* (cdr binds)))
      ((= i *array-size*))
    (when (and binds (= 1 (cadar binds)))
      (setf (sbit *in* i) 1))))


(defun initialize-mask ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 1 (sbit *inmask* i)) (= 0 (sbit *in* i)))
      (setf (sbit *mask* i) 1))
    (when (and *cam-clock* (= 1 (sbit *ffmask* i)))
      (setf (sbit *mask* i) 1)
      (when (= 1 (sbit *ffout* i))
        (when (= 1 (sbit *outmask* i)) (setf (sbit *out* i) 1))
        (mask-containers i)))))


(defun mask-containers (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*))
    (when (= 1 (aref *array* i j))
      (setf (sbit *mask* j) 1))))


(defun evaluate-circuit-loop ()
  (when (do-some-more)
    (set-out-and-mask)
    (mark-register-state)
    (evaluate-circuit-loop)))


(defun set-out-and-mask ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 0 (sbit *mask* i)) (col-is-empty i))
      (when (= 1 (sbit *outmask* i)) (setf (sbit *out* i) 1))
      (mark-register-mask i)
      (setf (sbit *mask* i) 1)
      (mask-containers i)) ))
```

```
(defun mark-register-state ()
  (when (and *cam-clock* (= 0 *cam-clock*))
    (do ((i 0 (incf i)))
        ((= i *array-size*))
      (when (and (= 1 (sbit *ffinmask* i)) (= 1 (sbit *mask* i)))
        (let ((ff (dereference-to-ff i)))
          (when (= 0 (sbit *ffmasked* ff))
            (setf (sbit *ffout* ff) 1))
          (setf (sbit *ffmasked* ff) 1) )))))   ;order important


(defun mark-register-mask (i)
  (when (and *cam-clock* (= 0 *cam-clock*) (= 1 (sbit *ffinmask* i)))
    (let ((ff (dereference-to-ff i)))
        (setf (sbit *ffmasked* ff) 1)) ))


(defun dereference-to-ff (i)
  (do ((j 1 (incf j)))
      ((= j *array-size*))
    (when (= 1 (aref *array* i j))
      (return j))))


(defun do-some-more ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (= 0 (sbit *mask* i))
      (return T))))


(defun col-is-empty (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*) T)
    (when (and (= 0 (sbit *mask* j)) (= 1 (aref *array* j i)))
      (return nil))))
```

ALTERNATIVE SOURCE CODE (COMBINATIONAL)

```
(defun initialize-in ()
  (do ((i 0 (incf i))
       (binds *cam-input* (cdr binds)))
      ((= i *array-size*))
    (when (and binds (= 1 (cadar binds)))
      (setf (sbit *in* i) 1))))

(defun initialize-mask ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 1 (sbit *inmask* i)) (= 0 (sbit *in* i)))
      (setf (sbit *mask* i) 1)) ))

(defun mask-containers (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*))
    (when (= 1 (aref *array* i j))
      (setf (sbit *mask* j) 1))))

(defun evaluate-circuit-loop ()
  (when (do-some-more)
    (set-out-and-mask)
    (evaluate-circuit-loop)))

(defun set-out-and-mask ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (and (= 0 (sbit *mask* i)) (col-is-empty i))
      (when (= 1 (sbit *outmask* i))
        (setf (sbit *out* i) 1))
      (setf (sbit *mask* i) 1)
      (mask-containers i)) ))

(defun do-some-more ()
  (do ((i 0 (incf i)))
      ((= i *array-size*))
    (when (= 0 (sbit *mask* i))
      (return T))))

(defun col-is-empty (i)
  (do ((j 0 (incf j)))
      ((= j *array-size*) T)
    (when (and (= 0 (sbit *mask* j)) (= 1 (aref *array* j i)))
      (return nil))))
```