BIT-STREAM CIRCUIT SIMULATION

William Bricken

August 1997

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 sequentiality of the logic of the specific circuit.

The parens bit-stream capitalizes on the dual interpretation of parens as both logical values and logical operators. The BL reduction algorithm applies Crossing and Calling until the stream itself reduces to a representation of the output Boolean value. The number of passes of the breadth-first reduction algorithm counts the length of the critical path of the circuit for the given bindings.

The reduction algorithm is also implemented in a single pass, depth-first manner, using local backtracking in the form of boundary reference counting. This approach is ideal for an FPGA hardware simulator. Initial estimates suggest that a bit-stream hardware simulator should be able to perform within an order of magnitude of the speed of the actual ASIC.

The number of erasures occurring during evaluation is a measure of the computational complexity of the circuit, the increase in entropy seen as loss of knowledge of structure. The entropy interpretation of erasure-as-computation was inspired by Feynman:

"This realization, that it is the erasure of information, and not measurement, that is the source of entropy generation in the computational process, was the major breakthrough in the study of reversible computation."
-- Feynman on Computation (p150)

Bit-stream Evaluation of Logic

Encode parens form without variables as (=1 and)=0. The bit-pattern matching engine applies crossing and calling using these rules:

```
(()) =           is  1100 ==> ____
()() = ()        is  1010 ==> 1__0  or alternatively 10__ or __10
```

Underlines (blanks) are "skip over this space", i.e. pointer bypass. This achieves erasure of 4 (or 2) bits. When the stream begins with a series of 1s, the closing zeros require either another pass or (more efficiently) looking at the prior two bits on the process stack. A third approach (implemented) is to maintain a count of the excess opening 1s, similar to reference counting in a garbage collector.

For clarity of explanation, the S(tart) and E(nd) of the bit-stream can be marked with special tokens, S and E.

```
      halt:   SE           is halt-False
              S10...E      is halt-True
              S11...E      is continue
```

Should a 0 fail to erase through looking forward into the stream, we know that looking backward into the passed bit stack will yield either a 1 or the S token. Thus the single pass algorithm simply includes the rule:

```
      if current-bit=0
            then add the two prior 1 bits to the front of the stream.
```

In an actual bit-stream implementation, the additional codes for S and E are not readily available. Thus the implementation uses reference counts to manage the end-of-file. This is not a pure streaming approach, since it relies on knowing the initial length of the bit encoding of the circuit.

Example

The data structure for the AND graph with void values (AND 0 0), is

```
      (()()) = 110100
```

Consider this function to be stored in an array (a stream feeds this array dynamically):

i: index (both arrays)	0	1	2	3	4	5	6	7
B: bit-array (data)	S	1	1	0	1	0	0	E

The example reduces by calling

```
            (i2-i3-i4-i5 ===> i2-i3-i6-i7)
```

followed by crossing

```
            (i1-i2-i3-i6 ===> <void> ===> i0-i7)
```

Pseudo-code

The pseudo-code for applying rules in a single pass follows. The bit-stream is processed by accumulating four bits, examining patterns and either

1) passing the first bit to the processed stack if no match, or
2) consuming bits dictated by a match.

Let the stack of processed bits be A and the bit-stream be B

```
      Accessor:   (get n) ==> get n values from bit-stream B

      Backtrack:  (pop n) ==> get top n values on processed-stack A
```

Since all bits on the processed stack are 1s, the stack A is implemented by a counter. (pop n) decrements the counter by n while adding 1-bits to the front of the stream.

Initialize the bit stream with S and end it with E. The pseudo-code uses a look-ahead to see eof, thus S and E are never matched during reduction. Data fields include four valid bits plus end conditions marked with S and E.

```
            (data := (get 4))

            (when (data=Sxxxx)
                  (halt-if (data=SE--))
                  (halt-if (data=S10xx)))

            (halt-if (data=10E))

            (when (data=00E)
                  (data := (pop 2) + 00))

            (when (data=1010)
                  (data := 10 + (pop 2)))

            (when (data=1100)
                  (data := (get 4))

            (else (increment A)
                  (data := (rest data) + (get 1))
                  (recur) )
```

The trace for the AND example, S110100E:

```
      (get 4) -> S1101     A = 0       B = 00E
      (get 1) ->  1010     A = 1       B = 0E          1010 -> __10
      (pop 2) ->  100E     A = 1       B = <void>
      (get 1) ->  1100     A = 0       B = E           1100 -> ____
      (get 4) ->     E     A = 0       B = <void>      halt=false
```

Computational Complexity

Another example illustrates computational complexity:

```
            0 XOR 0   is  ((  ) (( )( )))  is  S1101101000E
```

with the following reduction

```
               form          time    entropy

            S1101101000E      0        0
                   --         1        1
                ---  -        2        3
             ---      -       3        5
```

Each backtrack is one parallel time step. Using a breadth-first multi-pass algorithm, one pass represents all rules which trigger in parallel. What's left after a pass represents the front of the value propagation wave through the circuit after one level of parallel "gate" evaluations.

The entropy S is proportional to the number of 1 bits erased in a step, so let

```
            1100 ==> ____  be S ==> S+2
            1010 ==> __10  be S ==> S+1
```

Thus the complexity of (AND 0 0) is (t=2 S=3) and (XOR 0 0) is (t=3 S=5), which provides a metric for Boolean function complexity, that incidentally summarizes all the variable symmetries in the particular function (see below).

Algebraic Bit-streams

The BL algebraic rules generalize the above scheme, adding bigger computational steps at the cost of recursive subroutines. Let X be any well-formed parens,

```
      ( ) A = ( )
                        10X  ==> 10_   with S ==> S+(1/2 length-of-X)
                        X10  ==> _10

      ((A)) = A
                        11x00 ==> __X__  with S ==> S+2

      (A (A)) = (A ( ))
                        1X1X00 ==> 1X1_00   with S ==> S+(1/2 length-of-X)
```

We increase the symbol set to 2 + (count of variables), use bit-encoded variables with (and) as special codes, and the simulator becomes an algebraic minimizer.

Algebraic extraction requires a pattern builder (to get a copy of X as a template), then handing that to the pattern mask to extract X. The current Losp code maps the intricacies of this recursive pattern matcher for arbitrary Boolean reduction.

This machine evaluates arbitrary Boolean expressions, and can even minimize Boolean algebra with one more pattern transformation rule, distribution:

```
            ((X Y)(X Z)) ==> X ((Y)(Z))  is  11XY01XZ00 ==> X11Y01Z00
```

This is an efficient universal Turing machine. It also looks promising as a four-bit wide FPGA engine, and as a 4n-bit wide CAM-type engine. The idea is to build the bit-XOR pattern matcher for rule application in one part of the FPGA, feed the circuit to be laid out through it, and it will optimize the layout using partial Boolean function evaluation, redundancy removal, etc.; the same functionality as the LISP list-based Losp engine, but now tractable at the silicon level.

There's also a nice calculus of Boolean complexity here. XOR for example has the general form:

```
            ((1 2)((1)(2))    with inputs 1 and 2.
```

What I've been looking for is a functional abstraction that is insensitive to which variables are which (like relativized Boolean cubes, group symmetries, etc). The trouble was that all standard abstractions (like variable permutation, commutativity) are too concrete. We need something like boundary nondirectionality in Dgraphs. So here it is, the XOR eg:

```
                    ((..)((.)(.)))
```

is input-independent

```
            XOR  =  11..011.01.000
```

where

```
             . = either possible value of a variable,

                   . = <void>  or   . = 10

            .. = the possibility space of two variables,

                  .. = <void>  or  .. = 10 (2 ways)  or  .. = 1010
```

Note that the space of

```
            .. = 1010
```

immediately and independently reduces by the rule:

```
            1010 ==> __10
```

So we can compute the minimum and maximum complexity of XOR, independent of variables:

```
            min:  11__011_01_000
```

by the XOR reduction above t=3, S=5 (I've "quoted" the substitution):

```
            max:  11`1__0'011`10'01`10'000

            is   111__0011100110000    t    S
                    --   --------      1    4 + 1
                  --  --               2    7
                 -      -        --    3    9   V=false
```

To see how nicely this handles the exponential increase of Boolean states, look at a tough three variable (non commutative) example:

```
      (a b c) ((a)(b) c) (a (b)(c))  is  1abc011a01b0c01a1b01c00

      ground-fn-abstraction  is          1...011.01.0.01.1.01.00

      min is                             1___011_01_0_01_1_01_00



      max is                             1m011m01m0m01m1m01m00   with m =10
```

(This has the familiar feel of operator as object.)

The idea is that we can bond the computational complexity in ticks and in negative information (entropy) without concern for the exponential set of test vectors. The BM minimization process gives us the range of possible circuit performances quickly.

A Transformation Example

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

so that

```
     1...011.01.0.01.1.01.00
   = 1...011.0111.0.01.1.0000
   = 1...011.01..011.01.000
   = 11.1..001.1.0.011.01.000
```

Is any particular implementation preferable? Max-min computation of each:

min	*t*	*S*
10110100110100	2	6
= 1011011100110000	3	7
= 10110101101000	3	6
= 1110011001101000	2	7

max	*t*	*S*
11^0011100110010011011001100 0	2	14+2
= 11^00111001111001001101100000	3	14+2
= 11^001110011^0011100110000	3	12+3
= 111011^0001101100100111001100000	4	15+1

^ is just a marker to get the time count right, keeps things parallel.
Of course the DNF form always takes two ticks, but it is not minimal entropy, and is maximal in variable references (substitution locales).

Some observations

Parens encodings have built-in 0/1 parity, so they are self-error-discovering.

The FSM (given well-formed input) is the minimal universal FSM (since all FSMs can be encoded as Boolean functions).

A parallel mask CAM architecture can simulate circuits in 2t cycles, which makes it as fast as an ASIC and still general purpose (this is modulo bit-width going into the CAM).

Diversion

Here are some interesting numbers:

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 time of 144M/32 = 4M test vectors per second. I.e., a general purpose simulator that runs at 25% speed of an ASIC itself, and is easy to manufacture.