NOTES ON CIRCUITS AND BOUNDARY LOGIC

William Bricken

December, 1994

N-BIT BINARY MULTIPLIER

I selected binary multiplication as a case study of boundary techniques for circuit improvement. The essential issue is *level of abstraction*.

With current techniques, binary multiplication requires one shift operation for every bit in the multiplicand, and one additional n-bit addition for every 1 in the multiplicand.

Current technique assumes bits bound as instances of numbers, restraining a multiply circuit to do only finite arithmetic. Circuits operate only on binary logic, further restraining a circuit to express numerical computation as logic.

The alternative is to place computation at higher levels of computational abstraction. Arbitrary FPGA circuits abstract hardwired circuits, as look-up tables, but still require the specification to be logical. Dynamic assembly-level programming allows an abstract machine to be configured for specialized computation. Vuillemin's arithmetic circuits might contribute here.

The strength of circuitry, dynamic or customized, is that every gate processes bit information every clock tick (at100 MHz). The weakness is that a tremendous amount of circuitry is required to make the ground logic express abstract ideas, such as multiplication.

Two approaches are suggested to improve computation:

Dynamic hardware: configures itself on the fly to the specification of the problem, thereby avoiding unnecessary machine instructions and cycles. The idea is to create a reducible RISC instruction set for an abstract engine that dynamically reconfigures circuits for current problems.

Abstract engine: processes software logic specifications as if they were circuits using generalized algebraic pattern-matching implemented on a fine-grain parallel processor. The idea is to find sufficient efficiency in processing higher-level symbolic descriptions that the speed of circuits (and their complexity) is obviated.

For example, a one-bit multiply maps to logical and, but a two-bit multiply has to contend with a possible carry (in the case of 11x11=01). But two-bit (mod 4) multiply can avoid carrying altogether by using a mod4 code, so that 3 * 3 = 1 via direct lookup.

Complexity in multiply circuits comes from the carry operation when adding, but in binary place notation, the shift adds additional circuit complexity. The abstract substitution engine can operate on any symbolic structure. Two problems remain:

1. Implementing a symbolic engine in silicon that is very fast.

2. Finding a symbolic representation that is easily computable.

Symbolic representations with good computability structure require both representations that are not space consuming and transformations that are not time consuming.

CIRCUITS

In exploring the feasibility of implementing circuits as distinctions, I tested the approach on several elementary circuits, without difficulty. These include:

the sixteen two-variable logical gates
half-adder
full-adder
four-bit magnitude comparator
4-to-1 line mux
RS flip-flop
clocked D flip-flop
four-bit sequential pattern recognizer
three-bit up-down counter

Symbolic match-and-substitute (m&s) boundary evaluation of standard circuits has the following characteristics:

Combinatorial circuits require two symbolic steps: binding of variables and arithmetic evaluation. Assume each of these functionalities can be implemented in a fine grain parallel pattern-matcher. Each m&s cycle takes about four clock ticks (assuming 80 MHz, this yields 2x10^7 m&s per second).

Binding takes one m&s cycle: since all variables are unique, they do not conflict. Wiring variables to be bound to values is the same as naming pins and providing a pointer to the pin pulse value.

Evaluation takes n m&s cycles, where n is the depth of nesting of boundaries in the expression. When expressions are in conjunctive normal form (CNF), n = 2.

Due to parallelism, these speeds are completely scalable, modulo the number of physical processing cells. That is, a micro-processor with 3000 gates and forty input variables would take the same time to evaluate as a single and gate.

The catch is the available physical cell structure. The input variables would have multiple binding sites in a symbolic expression, the number of sites is exponentially related to the shallowness of the boundary expression. Thus, optimization of boundary expressions for a particular parallel cell array would trade depth of nesting (and thus time of evaluation) for available routing connections (and thus number of variable references).

The relevant optimization theorems: CNF guarantees a maximum of two levels of nesting, at the cost of potentially exponential variable reference. INF (Implicate Normal Form) guarantees at most two references to any one variable (its literal and negated form), at the cost of linear growth in depth of nesting. It is an empirical question which form will be best, but in a practical implementation, the routing will be a more contested resource than cycles to completion.

TIME

Circuitry permits information to be expressed in space (as bandwidth) and in time (as clocked cycles). Logic design often selects between the two, using manufacturing costs as a modulator. N-bit addition, for example, can be achieved by one n-bit parallel adder (gaining time at the cost of circuitry) or by n repetitions of a single bit adder with appropriate memory (gaining circuit simplicity at the cost of processing time).

In a symbolic processor, all variables are bound as their values become available. If the input line is sequential, then binding is sequential, taking time. Input is parallel to the extent that input bandwidth supports parallel bits. Bindings can be achieved in a single cycle.

However, some binding regimes are strictly sequential, those that rely on previous calculated values. In logic for example, transitivity is strictly sequential. Recursive functions are also strictly sequential, although they can be implemented as linear sequential processes in a single processor or as log-linear sequential processes in a multiprocessor architecture. The essential point is that a symbolic approach permits dynamic choice between linear and log-linear computation, depending on processor availability. The gain is that registers are not needed to store intermediate results, since variable names do that job. By expanding time sequence into spatial variable reference, memory is eliminated.

The central question then is: can we construct a sufficiently efficient bit-level processor for parallel pattern-matching to provide optional parallel arithmetic and logic processing. An attendant question is: can we create a symbolic system with sufficiently efficient representation and transformation rules to justify the hardware development? The route is to implement the representation theory as a hardware emulation, and then evaluate performance.

GRAPHS AND BOUNDARIES

Boundary math provides a theory of efficient representation and transformation based on void substitutions. We will assume a hardware architecture that can accomplish void substitution in parallel.

Since we know the map from boundaries to logic, we can use boundary forms to represent functionally equivalent circuits. The idea is to convert standard circuit elements to distinction networks, and then to compute over distinction networks in order to emulate the circuit.

This provides several essential advantages:

-- Distinction networks can be reduced in parallel.

-- Distinction networks can be manipulated algebraically, providing provability, partial evaluation, and optimization.

-- Distinction networks can be displayed in a variety of visual forms.

-- Distinctions map one-to-many to logic, essentially simplifying logical expressions at parse time.

-- Distinction networks can be treated as abstractions, essentially eliminating the circuit level logic interpretation.

-- Boundary math integrates numeric and symbolic computation.

-- Boundaries can be interpreted both descriptively and functionally.

-- Imaginaries can be treated symbolically, avoiding traps in realizations.

The abstraction hierarchy for the representation:

interfaces (displays)
programs
functions
circuit components
and, or, not
distinctions

The rough architecture:

```
Domains
   |
Input
   |
Parsers
   |
Internal Representation  <-->  Engine
   |
Unparsers
   |
Display Manager
   |
Output
```

An essential question is: does the expressability of circuit specification languages map onto boundary representations? It does, rather easily. However, circuits, like mathematical expressions, have equivalent forms, and circuit layout introduces non-mathematical factors into design choices. The two primary considerations are

Area of chip (smallness of circuit), and
Speed of chip (minimal propagation delays, implying shallowness)

The primary observation is that circuits include a huge amount of mechanism in order to make logic manifest. This includes

combinatorial pathways with logical gates
flip-flops for time changing data
clocks for synchronization of multiple processes

Clock logic can be expressed within a logical specification as another variable. Time-based memory can be also constructed from new variables.

EMULATION ARCHITECTURE SUMMARY

Construct parsers from logical specifications of circuits to distinction networks. Evaluate and reduce d-networks, returning the computed output of the circuit. Display the reduction process using visual boundary techniques.

BINARY DECISION TREES

We reviewed models based on ordered evaluation of variables in a logical expression. The Losp representation provides a simple decision tree evaluation, in which the branches generated by a variable evaluation are made manifest by substitution (of void and mark) and simplification.

The issue of ordering variable evaluations is also simplified by boundary techniques, since heuristics and canonical forms are direct consequences of the boundary structure.

It is necessary to distinguish between evaluation techniques (arithmetic), which are physical manifest, and minimization techniques (algebra), which are accessible only symbolically.