MODELING FOR HARDWARE AND SOFTWARE INTEGRATION

William Bricken
December 1995

Sections

Digital Modeling
VonNeumann Abstraction
    The vonNeumann Trade-off
The Virtual/Physical Interface
Reconfigurable Computing
Modeling Circuitry
Modeling Computation
    Simple
    Model
    Computation
Digital Circuits as Computational Models
    What Is Digital Computation?
    Representing Computation
Outline of a Deep Model of Computation
    Some design notes
Outline of a Deep Programming Language
    Descriptive aspects of a generic DPL
    Algorithmic process aspects of a generic DPL
    Physical aspects of a generic DPL
Hierarchical Complexity
    A data structure hierarchy

Digital Modeling

Whenever we represent physical reality, we represent it with a model. The model can be relatively accurate or inaccurate; it can capture some or many features; it can be either formal or informal. But the intent of all models is to provide a symbolic structure which can be used to aid conceptualization of reality.

The construction of a model is the crossing of a boundary between physical and virtual, between realization and intention, between the thing and the indication of the thing. The crossing from concrete to concept is a negotiation between what is actually happening and our ease of thought about what is happening. Since humans have naturally limited perceptual capabilities and are finite in both time and space, models are mandatory to express those physical events which lie outside the gross phenomena that impinge upon our bodies and our senses.

However, boundary crossing is symmetrical, what is *from* and what is *to* is solely a matter of perspective. It is our choice of a particular point-of-view to treat symbolic models as a representation (and thus a mere shadow) of physical reality. In this paper I will exercise the option to choose an alternative location for our point-of-view (POV), to see the physical reality of a computer as a model of abstract computation.

Digital computation is inherently virtual rather than physical. Computation exists in our minds, not in physical reality. Yes, we use computation and mathematics in general to model physical things, but increasingly the goal of computation is constrained solely to virtual events. For example, the word processor I am using this very moment is doing digital computation with the express goal of constructing textual representation of ideas. The entire process anchors only in the most tangential way to physical reality. Physical devices, such as the keyboard and the silicon gate array, are not used during text editing to model physical reality, they are used to subserviate physical reality to my actual goal of constructing a collection of quite virtual ideas. That is, the physical computer serves as *a model of* abstractions. In that sense, the inadequacies of the physical processes to exactly replicate the concepts being constructed are inadequacies of a model to reflect its grounding.

This presentation of digital modeling is bias toward the conceptual/virtual side of the boundary between physical and virtual, placing computation within the realm of idea rather than natural behavior. The alternative POV that physical devices are a model of conceptual abstraction extends well beyond the use of computers. The printed words that writer and reader both share are themselves modeling a set of concepts. English language is often a very poor model for some abstractions, and words can hardly do justice to some thoughts and feelings. Often, mathematical language is used to anchor our ideas more precisely. When dealing with ideas that do have the opportunity to strongly interact with the senses, such as the momentum of a large rock hurdling toward one's head, we use mathematical abstractions which have strong ties with physical reality as we know it. In contrast, the pencil and paper jottings that a third grade student generates while trying to multiply two three-digit numbers are physical models of purely cognitive computational processes. These jottings have no expected anchor to any physical reality. In the domains of quantum mechanics and Zen mysticism and digital computation, the phenomena being modeled are essentially conceptual and not physical. In these cases, languages and mathematics are the reality principle. Physical representations such as the linear accelerator, the printed page, and the silicon wafer are all themselves models, representations of the abstraction being experienced.

Mediating between abstraction and physical model, between intention-as-goal and computer-as-model is both subtle and difficult. Our current notion of computer science is radically misguided, focusing on the shadow rather than the object. The fundamental principle is that tools and interfaces should address intention, not the processes and weaknesses of the model which

represent the intention. I am suggesting interface menus which permit choices about reaching goals rather than choices about how the model of these goals works. I am suggesting programming languages which permit specification of intention rather than specific control over algorithms which steer the modeling of intention. I am suggesting machine instructions which configure the computational model rather than the computation. And I am suggesting silicon architectures which dynamically adapt to specification rather than rigidly reflecting a particular algorithmic process.

VonNeumann Abstraction

VonNeumann abstraction is the great innovation that made computing on the first digital machines practical. It mediated between the stark literalism of signals on wires and the abstract intentions of generalized modeling (such as computing ballistic trajectories). The hierarchy of encodements that grew from digital machine codes through instruction sets, assembly languages, programming languages, programming metaphors, and user interfaces (aka good programming languages) has lifted our model of computation from the wires of the machine up to the touch-and-feel of the machine user. These encodements all have the same objective: to permit the human user to compute anything that can be specified as computable. The responsibility of the human user is to *clearly specify intentions* that can be encoded down through the hierarchy of abstractions to the digital oscillations that, over the last fifty years, we have learned to manipulate through programming as more and more complex computation.

Essentially, the abstract encodement that maps, for example, digital oscillations at 100 MHz onto virtual reality interfaces, can be designed to be a *single* encodement. The historical evolution of computers has been so successful that the techniques which motivated ENIAC in 1946 have never been deconstructed pragmatically. Layers of encodement with faded utility still permeate our software practices, carrying with them layers of irrelevant concept structure. Hard problems strongly associated with immature architectures were encountered and solved, these solutions were then written into the structure of our abstract tools. Historical inertia has led to confused abstraction boundaries within software tools: system resource control is confounded with programming languages (compare C to LISP), interface is confused with programming (lack of API abstraction barriers), bit-level processes are separated from word-level processes (combinatorial and sequential circuits having different models yet both implicate time and space). Upon re-examination, we have found that a single encodement, with suitable provisions for handing complexity, is sufficient to map any mathematical specification directly onto digital signals.

Similarly, the physicality of "putting-plugs-in-boards" generated a hierarchy that grew from digital logic codes through geometric manufacturing specifications, structural description languages, register transfer languages, schematic editors, automated synthesis, and reprogrammable hardware (aka

reconfigurable computing). These various realization tools also have a single objective: to permit the human designer to efficiently manufacture transistor arrays that modulate digital signals to meet computational intention. The responsibility of the designer is to *produce efficient computing machines* that respond with mathematical precision to encoded intentions.

Essentially, the concrete manufacturing process that realizes computational machines can be designed as a single process which constructs hardware for a specific set of computational capabilities (as defined by the hardware architecture and by the software assembly language). However, transistor density has become so great that design techniques must be highly automated and designs themselves highly structured. The languages used to specify the construction of physical networks of transistors (or more generally, *interactive distinction nets*), moreover, became disconnected from literal intention when the vonNeumann abstraction of machine instructions was incorporated into hardware design.

The vonNeumann Trade-off

Circuitry which is dedicated for a particular purpose (ASICs) treats all incoming signals as computational data. The vonNeumann abstraction is to treat some "data" as instructions, in essence to construct *meta-data*, data about how to handle the data of computation. Meta-data is commonly known as machine instructions, control language, or opcodes. A vonNeumann machine differentiates data from control information, but treats both equally in memory. This provides several advantages, such as

ease of programming iteration and branching
the ability to modify instructions during computation
generalized reuse of special purpose circuitry (such as the ALU)

By abandoning symbolic literalism in computation (the idea that the specification and the circuitry are closely analogous) in favor of symbolic abstraction (the idea that specification merges two languages: representation of data to be manipulated and representation of control describing how to conduct the manipulation), vonNeumann engineered a classic trade-off between physical and virtual. Hardware became general purpose in the symbolic domain, at the cost of becoming constrained in the physical domain. The physical constraint, called the vonNeumann bottleneck, is that computation was forced into a sequential model, which required tracking of the location of information (program counters) and synchronization of the timing of transformations (global clocks).

The vonNeumann trade-off was so appealing during the era of weak computing machines that it dominated the foundations of practical computation to such an extent that today all programming languages and nearly all computers still maintain the distinction between process and process control. As a result, software programming includes both function specification and algorithm

specification. Hardware design includes both functionality and control. But most seriously, software design and hardware design became incommensurable, despite the fact that both are intended to realize the same mathematical intentions.

At the circuit component level, two types of signals are transacted, logic and control. The control signal (which is also expressed in logical operations) addresses the behavior of the circuit but not its functionality, while the logic signal addresses the functionality of the circuit but not its structural behavior. The purely sequential model of computation imposes a severe constraint on the natural parallelism of physical circuitry. Separate process and control regimes make circuitry far more complex both to design and to manufacture.

At the software language level, procedural languages which map onto instructional sequences have recently given way to declarative and functional languages, but only on the surface, since each is still reduced to sequential machine instructions (as the cost of inherent performance efficiency). The purely procedural model of computation imposes severe constraints on the programmer by requiring a cognitive model not only of functional intention but also of the manner in which the hardware enacts that intention.

The vonNeumann trade-off erodes the once intimate connection between the physical world of hardware and the virtual world of mathematics. The abstraction of program control has disassociated both the design of hardware from the specification of our intentions and the design of software from the physical grounding of mechanical consequence. The path to re-integration of the computing enterprise is to reconnect specification directly to circuitry, to abandon the vonNeumann architecture. Upon re-examination, we have found that a direct mapping from mathematical specification to transistor network design, with suitable provisions for handling physical resources (generalized costs), is sufficient for efficient construction of efficient reconfigurable computing machines.

The Virtual/Physical Interface

There is, of course, always a barrier to be crossed, the same one crossed by vonNeumann: the virtual/physical (aka mind/body, ideal/real, symbolic/concrete) boundary. The virtual/physical boundary connects words and actions, ideas and behaviors, concept and implementation. All experienced programmers know the immense difficulty of specifying intentions clearly enough to enact them on computational hardware. The experience of AI, for example, credits 90% of the effort in constructing an expert system to knowledge engineering, to specifying intentions in computational terms.

We can elect to partition the world in any manner. We place value judgments on some structural properties of partitions and thereby create preferable perspectives. For instance, orthogonal concepts (as in +/- or

horizontal/vertical or inside/outside or data/control) permit the construction of reductionalistic perspectives, make computational tools and algorithms easy and tractable, and support the illusion of objects distinct from processes. Other examples of valued partitions: integer/real, P/NP, stable/transient, abstract/concrete.

It is always possible to construct a partition between concept and implementation. The entire enterprise of scientific description is one of negotiating the reality of physical behavior with the idealism of mathematical simplification. The key shift in computation occurred in the 15th century, when variables were abstracted from applied integers, and negative numbers were invented. The Pythagorean discovery of irrationals would also qualify as a fundamental shift in abstraction, had the Greeks not labeled irrationals as monsters.

The computational/cognitive qualities of crossing the virtual/physical distinction are implementation independent by definition, since the implementation sits squarely on the physical side of the distinction, while mathematics is squarely on the virtual side. Generally any *mathematics* is implementation independent. (Some mathematics, like infinite precision and eternal processes, is implementation impossible.)

The transition from concept to concrete involves a successive compromise of ease of thought (cognitive load) in favor of ease of digital manipulation, which in turn is defined by the economics of semi-conductor manufacturing. The straightforward way to integrate hardware and software is to carefully select and design the most inconspicuous place to put the crossing between intention and realization.

Hardware/software partitioning is a deep design choice. What is physical and what is symbolic can be cut in many ways, and suitable cuts are strongly task specific. A unified model would allow us to parameterize resources and functionalities (i.e. hardware and software; chip-area and timing; hardware architecture and programs) freely, to place the abstraction/realization barrier where we will not trip over it.

The vonNeumann choice places the concept/implementation distinction in between two fundamentally digital (aka abstract, software) processes, between the design of specification and the design of computing hardware:

```
hierarchy of abstraction specification languages
      machine language specification
            vonNeumann tradeoff
      circuit behavioral specification
hierarchy of realization specification languages
```

By placing the virtual/physical boundary much closer to the hardware, many artifactual levels of software encodement can be unified. The apparent gap between hardware and software can be reduced to inconsequentiality. Circuit

design would remain an abstract software process, relegating hardware design to the construction of homogeneous reconfigurable arrays. Real-time hardware reconfiguration would completely replace machine language abstraction as the route to general purpose computing. All hardware resources would be available for all computing tasks, while the hardware structure (architecture, connectivity) which achieves particular computational results would be configured dynamically as needed. Software specification of intentions would directly program the hardware configuration ("make-by-need circuitry") rather than the vonNeumann machine codes. The resulting virtual/physical interface could eliminate both the abstraction and the realization hierarchies:

```
single abstraction specification language
      machine distinction networks with real-time reconfiguration
single realization design
```

We will call the direct programming of reconfigurable hardware *a deep model* of computation.

Reconfigurable Computing

The notion of abandoning vonNeumann computation in favor of a deep model is reasonable only in context of recent developments in computer hardware and software. Similar approaches are currently occurring at all fields of Computer Science (such as real-time code generation through programs that write programs, dynamic opcode configuration, partial function evaluation, and dynamic FPGA layout). In order to push a specification down into circuitry, we require practical reconfigurable hardware (exemplified by FPGA approaches). In order to reconfigure in real-time, we require a model of computation which can fully express the intentions of programs and is efficient to optimize and route. The accrued advantages include:

```
general purpose circuitry without intermediate abstractions
arbitrary algorithmic approaches, particularly parallelism
minimal programming and design effort
dynamically customized efficient architectures
homogeneous, easy-to-manufacture hardware arrays
```

In order to meet performance demands, modern multimedia computing requires specialized circuitry for many tasks, such as modem, audio, 3D graphics, videoconferencing and compression. These diverse functionalities have very little circuitry or program code in common. The broad performance needs currently require a general purpose cpu to be augmented with several special purpose accelerators. The idea of using a diverse set of machine instructions to cover all the functionalities results in poor cpu performance, with many opcodes unused and redundant. That is, the conventional vonNeumann processor is no longer suitable for general purpose computation. Special purpose circuitry, such as DSPs, are also not suitable, since they are hard to

program, hard to compile, and not general purpose. The solution for modern computing is reconfigurable hardware.

Reconfigurable computation (and deep models) shift the burden of program control to a software compiler, minimizing the costs incurred by the use of hardware control circuitry and clocks. Since a compiler has global program information for optimization, and can include models of machine resources and constraints, this approach places the minimum burden on hardware design and configuration. In particular, hardware resources can be structured to the needs of the task, so performance itself can be parameterized across both available time and available space. Specification can be seen as generating an abstract circuit structure, outside of time and space, while implementation can be seen as the construction of circuitry resources to meet performance needs.

Modeling Circuitry

The *behavior* of a circuit is a mapping from the input of a black-box (the circuit) to its output. Behavior consists of a *functional model* expressed in propositional logic and a *temporal model* expressed in delay and latency. The *structure* of a circuit is defined by the hierarchical decomposition of the black-box into components (themselves either primitive or complex) and into connections between components. Finally, the *physical view* of a circuit refers to actual physical components (transistors and wires) and their physical properties.

(This literal, object-oriented description centers on gates (or transistors) as objects and voltage transitions through gates as process. Alternative conceptualizations could focus on transitional activities as primary, or on temporal harmonics.)

The behavioral, structural, and physical views of a circuit can each be modeled by architectural, logical, and geometric abstractions, forming nine different ways to describe circuitry.

behavioral architecture	machine code (operation, dependency)
structural architecture	block and bus connectivity
physical architecture	semi-conductor board
behavioral logic	function, state-transition
structural logic	network of Boolean nodes
physical logic	transistors, gates
behavioral geometry	signal propagation routes
structural geometry	placement and routing
physical geometry	time and space constraints

The behavior of a correct circuit is its functional specification. A circuit is given a structure by assigning its logical specifications to a gate level netlist. A structure is then given a physical interpretation by assigning operations to resources through binding and scheduling.

Abstract models are used throughout all aspects of the design process. Most of these models are based on graphs, and include netlists, Boolean networks, state diagrams, transition tables, and dataflow and sequencing graphs.

In converging behavioral and structural models of computation, several design choices are available, including:

mapping the behavioral directly onto the structural (dnets)
minimizing the behavioral/structural map by some structural criteria
(egs: depth, number of nodes, number of links)
optimizing the structural
constraining the structural to meet specified limits

Similarly, in converging the structural and physical models, the design choices include:

mapping the structural directly onto the physical (eg ASICs)
processing an abstraction of structure (eg graph-machine)
processing an abstraction of process, (eg vonNeumann-machine)
mapping the structural onto a configurable tableau (eg FPGA, PLA)

The deep modeling approach is one of direct mapping from behavior to structure to physicality. Boundary mathematics makes the first transition feasible, while reconfigurable hardware makes the second mapping feasible.

Physical models are used to characterize the efficiency of physical behavior. Area and connectivity are structural metrics which are relatively independent of design transformations (they are linear and additive). In contrast, performance metrics (delay, cycle-time, latency, throughput) are not additive but depend upon structural analysis and can change non-linearly with design changes. Modeling temporal behavior is further complicated by loops and branches in programming logic which make performance estimation context-dependent. Additional control logic in the form of completion signals is then necessary to structure timing bounds.

Optimizing a design implicates all circuit models and views. Behavioral optimization is the minimization of specification functionality. Structural optimization focuses on the flow of data and control through the circuit network. Physical optimization can involve the selection of transistor technologies and physical manufacturing processes. Due the complexity of the optimization task, often particular parameters are bounded or held constant. Thus, typical optimization trade-offs include minimization of area for a given latency, and minimization of cycle-time for a given area or latency. Finally, a circuit design is constrained by available resources (such as ALU

functionality, memory size, and available connectivity and interface wiring), by i/o format and timing, by performance requirements, and most importantly, by manufacturing costs.

In the behavioral domain, the internals of a circuit as a black-box are abstracted away, although there is a necessity to be able to compose logical black-boxes into broader functionalities (traditionally called computer architecture). Hardware composition, in contrast, is difficult since it is the convergence point not only of circuit behavior, but also of the algorithmic programming model and of the software specification. The current complexity of machine architecture is a direct result of a vonNeumann computing strategy, which requires hardware resources to be bound, shared, scheduled, and synchronized.

Composition of functionality is a dominant difficulty in both hardware and software design. Hardware does not support algebraic variables, abstract looping and branching, and symbolic optimization. Conversely, software function composition does not support the spatial parallelism of machines or temporal synchronization of physical processes. Software languages provide design options for variable binding, temporal ordering of evaluation, and function composition. Conversely, hardware architectures provide design options for instruction-stream parallelism, message passing, and shared memory.

The primary intention of a compiler is to convert specification into machine language. Internal digital formats (machine languages) are designed to meet different criteria than user specification languages, so a compiler essentially mediates between cognition-friendly abstraction and circuit-friendly abstraction. Both compiler input and output is symbolic, but the purpose of encodement shifts from functionality to efficiency. That is, specification languages are designed to be closest to mental models of computation while machine languages are designed be closest to performance models of circuits.

The proposed deep model of computation attempts to unify these differences by combining a homogeneous reprogrammable hardware array which is easy to manufacture with a software circuitry design model which is easy to compile. Furthermore, specification of intentions is expressed in a language which compiles directly to the circuitry design model, eliminating the intermediate design levels of machine architecture.

Modeling Computation

Integrated circuit designs, programming languages, and math models all *describe computation*. How does one organize all the different descriptions of computation into a simple structure? Perhaps it is best done by starting from the simplest (most abstract) components and structures. This would represent a top-down approach, as opposed to the bottom-up evolution that characterizes

our current model of computation. The objective then, is to identify *a simple model of computation*, one which generates the smallest gap between ideal and real.

Simple

A no-principles, void-based formal model is the simplest ground. Non-formal models cannot be simple since they fail to offer structures such as invariance. symmetry, and enumerable domains. Non-void-based models obviously introduce initial assumptions.

Model

Models are abstractions of real processes, a mapping from symbolic to physical. Modeling is the enterprise of mathematical description of reality. Models can also map from symbolic to symbolic. In order to maintain simplicity, the objective is to collapse all symbolic modeling hierarchies. Models assume representability, and include data structure, accessors, constructors, and preservative (invariant) transformations.

Computation

Computation is a realized behavior which is defined by a conceptual substrate (Mind/Mathematics) and supported by a physical substrate (Circuit/Signals). Computing algorithms unite a mathematical objective with a processing device, directly orchestrating a virtual/physical boundary crossing.

How then do we go from <void> to <computation> to <understanding/interface> minimalistically? And how do we then link this minimal construction to the physical world of hardware manufacturing? We start with boundary mathematics as the core. Mapping from boundary techniques to more abstract levels of mathematics is an on-going enterprise, with existing results in the fundamental mathematical structures of logic, algebra, numbers, and sets. Mapping boundary techniques to more concrete levels of implementation and manufacturing has been the focus of this year's research.

The essential idea is to begin with nothing and build both toward abstraction and toward realization. Distinction as abstract logical existence melds with distinction as concrete semi-conductor (transistor) switching. Mathematics is mapped onto circuitry directly, so that the form of the circuit reflects the form of the intention. Mathematically, crossing serves as the hierarchy archetype while calling serves as the cardinality archetype. Physically, crossing serves as the transition archetype while calling serves as the wiring connectivity archetype.

The deep model approach uses the underlying principles of both physical and virtual to unite the two. Comparable enterprises include Chomsky's development of deep models of language which unified the various mother

tongues of the world, Jung's systemization of psychological archetypes, and Campbell's identification of the communality of myth and religion across diverse cultures.

Our goal is a deep model of computation which will unify the construction of computational machines with the construction of computational intentions. The project's development work (PARTS board, BobTools, etc.) provides test-cases for determining the sanity of our deep model, since a deep model must engender a unified perspective over the myriads of technical issues. Problems that do not fall out of deep structure point to myopia in that model.

Digital Circuits as Computational Models

The tools we use professionally to bridge the gap between the abstract process of computation and the models of computation which are expressed as the physical behavior of digital circuits are fairly limited, encompassing only a few methodological principles, basically because the agreed upon goal is to constrain the physical behavior of circuits to the domain of useful computation. The engineering literature universally adopts the traditional POV that the model of computation approximates the behavior of circuits, that behavior is paramount and that computation is the artifact. I have reinterpreted this viewpoint so that the behavior of circuits is seen as a model of the intention of computation. This perspective is quite natural, given that circuits are constructed solely for the purpose of computation. The vagaries of circuit behavior are artifactual; computation is not the model but the actuality.

Circuits model computation. Weaknesses in the model of computation expressed by circuits include

- physical timing glitches
- difficulties with manufacturing
- non-optimal circuit logic
- ...

Weaknesses in the conceptualization of computation include

- inadequate and misguided specifications of intention
- poor correspondence between computational intention and physical models
- non-formal representation of computation
- overly elaborate and redundant conceptual structures
- ...

Computation is a mathematical structure with includes both traditional algebraic structures and the evolution of these structures over time. Traditional programming languages provide a poor representation of mathematical intention, primarily because they are historically associated with the behavior of physical circuits rather than with mathematical intention. The POV that computation is modeled by silicon behavior requires a fundamental realignment of software and hardware specification. Here,

hardware structure is seen as dependent on software specification. Software languages begin at very-high-level specification languages, and then compromise their integrity in order to match the limitations of the physical model.

The essential difference between the goals of software and hardware is between abstraction and efficiency. The goal of programming is expression of abstraction. The goal of model building is the construction of efficient physical simulations of abstraction.

Very specifically, the tools provided in a programming language such as C, which allow the programmer to conform specification to the efficiency of a particular hardware architecture, are not computational. C is an engineering tool, no different than a silicon-waver mask or a drill press. The skills of using C are not those of programming, they are those of simulating programming on an existent physical model. A simple way to say this is that programming languages should reflect our cognitive model of computation rather than the computers' implementation model of computation.

There are very few programming languages which directly address the abstractions they intend to address. Pure LISP comes close, but is still deeply flawed. The control structures of LISP (cond and recursion) and the test predicates (eq and null) are mathematically appropriate. The constructor (cons) and destructors (car and cdr) are also appropriate when abstracted from their referent data structures to mean *put* and *get*. Even the reflection operators (eval and quote) are appropriate for meta-mathematical modeling. However, the list data structure is completely an implementation artifact. Worse, the structure of most LISP algorithms reflects list processing, subserviating the abstraction of process itself to the implementation.

Clean approximations of mathematical programming languages include Prolog III, a pure declarative constraint language, and the mathematical style of programming encouraged by Mathematica.

What is Digital Computation?

Digital computation is a realized behavior which is defined by a conceptual substrate (Mind/Mathematics) and supported by a physical substrate (Circuit/Signals). Computing algorithms unite a mathematical objective with a processing device, directly orchestrating a virtual/physical boundary crossing.

Representing Computation

As is conventional in mathematics, I will assume that the representation of the elements and operations of a domain is an arbitrary choice (given that a functional mapping is maintained between alternative representations). Thus,

within the domain of finite integers, we are free to choose decimal, binary, or other representational forms. Operations within the arithmetic of integers (addition, multiplication, etc) mean the same whether or not they are applied to binary or decimal notations, although the algorithms can be quite different. Further we are free to map across domains (given that isomorphism is maintained). For example, we can not only represent decimal addition as binary addition but we can also represent binary addition as Boolean operations. The selection of representation is one of convenience. In particular, Boolean transformations on finite binary strings is very convenient for digital implementations, while it does not undermine the essential mathematical acts of adding and multiplying integers. [This point is far more subtle than commonly acknowledged, and such mappings require care.]

For the purposes of this paper, I will restrict the notion of computation to the domain of Boolean transformations, and those domains which can be mapped onto Boolean transformations. This includes all of finite arithmetic and all of pattern matching. In general, almost all of finite mathematics is available within this restriction. Another way of looking at this is that the theory of computability (Turing machine equivalence) is defined on the domain of Boolean transformations, since at the base this is all that computers do.

A computational step is therefore a Boolean transformation. Multiple steps can occur at the same time, but in order to relate them, the results of multiple steps must be combined at some point sequentially.

Computation is therefore a partial ordering of Boolean transformations.

Using the representational tools of boundary mathematics, we can reduce the domain of Boolean transformations to one essential transformation, that of crossing distinction, which for logic can be interpreted as negation. Composition of operations (and thus partial ordering) can be reduced to two essential transformations, that of combining in space, which for logic can be interpreted as disjunction, and that of combining in (non-directional) sequence, which for logic can be interpreted as a deductive step.

Thus we can reduce the entire domain of finite computation to two abstract structures, composition and distinction. Combining in space and distinction can be expressed as the vertices of a net (a bipartite undirected graph). Vertices represent mathematical objects, either simple or compound. Edges in the net then represent sequential composition, proof steps, connection pins.

Combining in space can be implemented by joining physical wires, while crossing distinction can be implemented by inverting a signal. Complex Boolean structures are formed by constructing edges between space nodes and distinction nodes. Boolean signal propagation over the partial ordering then represents sequential/parallel evaluation.

Note that the net structure isolates composition and structure. The only operation/action applied to distinction is non-directional crossing. The only operation/action applied to space is non-directional sharing or joining. The only composition operation is non-directional connection of space to distinction. The domain of these sd-nets expresses all possible crossings and all possible joinings, while the topology of sd-nets expresses all possible partial orderings of crossings and joinings. Thus, sd-nets express all possible sequential and parallel computations.

The additional advantage of sd-nets is that they also provide an intuitive model of circuits, with distinctions being transistor-pairs which invert a signal, spaces being wire-OR, and net topology being the circuit topology while enacts a particular Boolean computation. Thus sd-nets unite software programming (in the form of Boolean specification) with hardware circuitry (in the form of net layout).

Outline of a Deep Model of Computation

The unifying purpose of a deep model is to reduce both cognitive load (increase ease of programming) and computational load (increase hardware efficiency).

The three skeleton components of a deep model of computation are:

1) human intention (expressed as mathematics)
2) digital algorithm (expressed as distinction networks)
3) physical circuit (expressed as transistor networks)

Intention introduces a set of mathematical constraints by defining a functionality and possibly input and output conditions. Algorithms introduce a set of processing constraints by identifying the complexity of and the transformational choices for an intention. Circuitry introduces a set of physical constraints by virtue of being made of materials configured in space and in time.

These constraint sets are nested, with intention, algorithm, and circuitry each restricting the range of possibility of the next by enforcing an invariance relationship. At the same time, each offers a range of choice in the realization of that invariance. For a given intention, there are many algorithms; for a given algorithm, there are many physical circuits. A given intention thus provides the most flexibility of design. A given algorithm is constrained by intentional invariance, but permits a wide choice of circuitry realization. A given circuit is required to be intentionally and algorithmically invariant, it also must conform to the demands of physical reality and is thus limited by time/space practicality.

Design choices broaden as we move from intention to algorithm to circuitry, while the behavioral space of the design narrows. The selection of a

particular design which achieves a particular intention is driven only by *efficiency (cost effectiveness) of performance.*

In crossing from virtuality to physicality, we encounter an asymmetry: the goal, structured as abstract mathematics, dominates the implementation, structured as a physical device. (We rarely hear: "See what you can compute with this device.") That is, physical implementation depends upon (is defined by) the abstract intention, while abstract intentions do not depend on their implementation.

Boundary mathematics characterizes these relationships through *pervasion*. The physical pervades the digital, which in turn pervades the intentional. From the perspective of a pervading space, all interior spaces are constrained. From the perspective of an interior space, outer spaces are unperceivable, and thus do not constrain. Design begins from the inner most space and crosses outward, accumulating additional performance constraints characteristic of each outer space. Physical implementation occurs in the outermost space and must hold inner spaces invariant (or lose pervasion).

In developing a model of intention/algorithm/circuitry, we must characterize each crossing. Let us denote the intention/algorithm crossing as the *specification boundary*, and the algorithm/circuitry crossing as the *implementation boundary*. Since the focus is on computation, both the formulation of proper intentions and the material physical behavior of silicon transistors are excluded from the model. We assume that the intention of the computation is adequately modeled by a mathematical structure. We also assume that the behavior of transistors is adequately modeled by the physics of their behavior, expressed in terms of a mathematical description.

The simplest modeling strategy is to develop a common modeling language for both the specification and the implementation boundaries. This potential common model is exactly what the vonNeumann trade-off makes impossible, since it subjugates both intention and circuitry to a particular algorithmic approach. A deep model would unite specification with algorithm symbolically, while postponing circuit structuring until all symbolics (including circuit design) are unified.

The alternative of uniting algorithm with circuitry (rather than uniting algorithm with specification) is also possible but not appealing due to the asymmetry (pervasion) of physical and virtual. Such a model would have a single algorithm fixed in hardware, thus constricting freedom of design prematurely.

Thus, the deep model strategy assumes that:

1) intentions are stated mathematically and properly,

2) algorithms are modeled as mathematical transformations and introduce computability constraints,

3) circuitry directly reflects the mathematical algorithms and introduces physical constraints, and

4) all symbolic representations are collapsed in a single compilation from intention to circuitry.

This approach shares much in common with the goals of the functional and logical programming communities, as well as with models of dynamic code generation, dynamic machine languages, and reconfigurable hardware.

The ingredients for the deep model approach to computation:

1) mathematical specification language
   implementation and algorithm independent
   declarative algebraic logic

2) boundary mathematics compiler
   void-based and real-time
   input is math specification/program
   output is optimized dnet
   includes algorithmic performance model

3) compiled dnet technology mapper
   input from dnet compiler
   output is reconfiguration bit-stream to target architecture
   includes physical technology constraints

4) boundary math reconfigurable hardware array
   cells specialized for dnet representation of computing.

Currently, hardware description languages (in contrast to software programming languages) include concurrent execution models, behavioral and structural views of the circuit, and timing. A deep model would move concurrency up into algorithm optimization, confound structure with function (using boundary logic), and localize timing through both asynchronous models and real-time resource creation.

Some design notes

The significant problems for the design of computation, in order, are

0) cost
1) functional invariance
2) testability
3) maintainability
4) performance maximization

Design Principles

1) a deep model of computation which collapses all digital maps
2) principled abstraction hierarchy
3) automated performance optimization and constraint maintenance

This is intended to (respectively)

1) make some design levels transparent to the user,
2) eliminate some concepts, and
3) provide bigger design steps.

Symbolic environment tools:

1) enforce hard constraints, parameterize soft constraints
2) model all implicit design concepts
3) permit meta operations (hierarchically)
4) model and parameterize resources and performance.

Outline of a Deep Programming Language

A deep programming language (DPL) must be able to express the functional, temporal, and physical characteristics of computation. The following outline specifies the abstractions required by such a language, but does not specify language details such as particular data structures, sequencing regimes, and hardware constraints.

Descriptive aspects of a generic DPL

Vocabulary

a finite set of typed names

Types

the domains over which particular names can range. Can be:
data types (bit, word, Boolean, string, list, network)
constants/grounds

operators (functions, predicates)
program execution types (memory location, signal transitions)
resources (memory, operator circuits, i/o devices)
constraints (circuit area, wiring, delay, cycle-time)

Expressions

constructions of names

Assertions

equivalence relations between expressions and constants
always Boolean
quantifiers specify kind of anchoring of name to domain
all control structures are assertions

Interpretations

assignment of names to objects in their domain of reference
listable over expressions
bound variables are independent/irrelevant

There are many structural decisions which define the character of a descriptive language. A partial set of (suggested) language architecture decisions includes:

1. A program is a set of expressions. Categorical composition permits a *set* to also be an expression, and operators to transform sets, with no implication of sequentiality.

2. Types are hierarchically arranged into abstract, algorithmic, and physical supertypes.

3. All expressions are either sets or assertions (declarative model). Sequencing is algorithmic rather than declarative.

4. The primitive data type is bit, all others are expressions of bits.

5. Interpretations convert software behavior to hardware behavior.

Algorithmic process aspects of a generic DPL

States

an interpretation of the set of program execution names

Transitions

the change of at least one name's interpretation (change of state)
the identity transition is *Idle*
consists of
enabling condition (guard)
modifications (names-next = expression-of-names-now)
transition assertion:
Interpretation(names-now) => Interpretation(names-next) = T
can be enabled or disabled, depending on enable-value
parallelism is more than one concurrent transition

Initialization

an assertion which permits transitions to begin to occur

Computation

an cyclic network (i.e. infinite sequence) of states with
initialization condition true
consecutive states (name-now and name-next) explicitly
permitted by a transition assertion
either infinitely changing or has terminal state
followed by infinite idles

Some (suggested) process architecture decisions:

1. Transitions can occur in parallel, many states can change at the same time.

2. Initialization is global.

3. Observation of computation requires a break in the computational cycle.

4. Control of transitions is physical rather than algorithmic.

5. Time is modeled solely by the concept of *next*, as introduced by a transition assertion

Physical aspects of a generic DPL

Properties

distinction or composition of distinctions
area, propagation delay, transfer curve, etc.
fan-in, fan-out

Connectivity

available wiring and bussing
transfer delays

Boundaries

transitions where costs change

Some (suggested) physical architecture decisions:

1. The target reconfigurable array is modeled as a set of physical resources which are constrained by the behavioral specifications.

2. All hardware components have an interval-based space/time model.

3. Boundaries are high priority constraints.

Some of the ways that boundary logic condenses a generic DPL include:

expressions are formed by wire connection
assertions are grounded to T and erase when F
interpretations are Boolean only
only permitted transitions are connect/disconnect from dnode
dnodes idle unless connectivity changes
all connectivity has logical semantics

Hierarchical Complexity

One early elaboration to a simple model is the introduction of structures of simple components, that is component hierarchies or systems. The purpose of hierarchy is to hide levels. Other hierarchical elaboration techniques include abstraction barriers, extended models, lattices, and recursion.

The general strategy for reducing complexity through hierarchical abstraction is to:

1) provide structured hierarchical partitions
2) hide as much as detail as possible at each level
3) provide as much programmability within levels as possible.

There are essentially two (and only two) orthogonal types of hierarchy: abstraction and meta. The meta hierarchy (aka reflection, self-reference) involves a change of the logical type of components, in particular, it introduces hierarchy of reference. Examples include data/control, arithmetic/algebra, virtual/physical, behavior/structure. Meta-hierarchy introduces only two levels, characterized by use/mention.

The second type of hierarchy is the abstraction hierarchy, which may have as many levels as are useful. Abstraction hierarchies group components into single units without essentially changing their logical type. Examples include opcode/programming language, OO inheritance, and circuit/component/chip.

Our current conceptualization of computation is a tangle of confounded hierarchies and goals. Several hierarchical decompositions are described below, in order to characterize current modeling ideas.

First, from the perspective of boundary mathematics, a rough abstraction hierarchy:

void	
distinction	indication/sentience
crossing and calling	space and time
structural abstraction	expressions
process abstraction	enactment
physical abstraction	modeling

From the perspective of physical circuit design, a rough abstraction hierarchy:

design model	abstract behavior
architecture model	abstract structure
performance model	abstract efficiency
correctness of behavior	functionality
efficiency of behavior	performance
actual behavior of physical circuit	reality

From DeMicheli, on digital circuit optimization:

Meta-formalism	distinctions	[not in DeMicheli]
Formalisms	graph theory, Boolean algebra	
Fundamental problems	coloring, covering, satisfiability	
Tasks		
architectural	scheduling, resource sharing	
logical	state minimization, testability	
	two-level combinatorial	
	multilevel combinatorial	
	sequential	
geometrical	binding, routing	
Pragmatics		
area		
performance (delay, cycle-time, latency, throughput)		
bindability		
testability		

A data structure hierarchy

bit
      combinatorial circuits generate bit transformations,
      f(in) = out, Domain = {0,1}

word
      bits are combined into words which support abstract interpretation.
      f(in) = out indexed by time, D = {8-bit words}
      this introduces grouping in time (registers) and space (busses)

instruction
      words are interpreted as opcodes, computational architecture
      f(in) defines a process on words which is embedded in the IC,
      with out in a orthogonal abstract domain (i.e. machine instructions)
      D = {opcodes}

program
      instructions are assembled into sequences in the instruction domain
      f(in) defines a sequence of processes which are manipulated
      by a hierarchy of abstract programming languages
          (machine code, assembly, programming language, high level language)
      D = {computational processes}
      note that D is defined by the specific machine architecture

message
      programs are interleaved across devices by the operating system
      f(in) extends beyond algorithm to resource coordination
      such as memory management, file i/o, interactivity interrupts.
      D = {resource management messages}

application
      resources and computation are coordinated for a particular task,
      introducing the idea of an end-user.
      f(in) are application specific codes
      D = {application abstraction and functions}

user interface
      application tools are presented to the user in a model
      which is close to the user's mental model of the task

Note that in the above, IC design is an application, and should hide system, program, instruction, etc. levels. Also note that these lower levels are usually given (by the machine you are working on). Thus language design is tightly connected with implementation architecture whereas interface design is not.

The above hierarchy embodies a severe restriction: it is purely data structure oriented. A process hierarchy (eg transition logic) would be very different and very useful (recall that data structure/domain focus is an artifact of object perception in a material culture).

The most natural approach is a *constraint hierarchy*, starting at the task specification (a set of behavioral and functional constraints) and narrowing through constraints imposed at each lower level of implementation. Rather than expressing specifics in a design space (this object and that process), express objectives of the design process and let the automated tools keep you within the constrained design space.

The constraint model is easily expressed through assertion of invariant equations. This is exactly the model of physics, where natural laws are expressed mathematically as equations. The invariance relations asserted by physical laws constrain physical behaviors. Computational invariance takes the form of a behavioral/functional specification, expressed as a formal program. Thus, a route for bridging the specification and implementation boundaries is to associate specification with a set of invariant equations which constrain behavior, and to associate implementation with a set of invariant equations which constrain the structure of space-time (i.e. performance) in which behavior takes place.