TECHNICAL REPORT NO. 283

On the Interplay of Synthesis and Verification Experiments with the FM8501 Processor Description

by

Steven D. Johnson, Robert M. Wehrmeister and Bhaskar Bose

Revised: November, 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

On the Interplay of Synthesis and Verification*
Experiments with the FM8501 Processor Description

Steven D. Johnson, Robert M. Wehrmeister and Bhaskar Bose

Indiana University Computer Science Department
Bloomington, Indiana, U.S.A

Verification and *synthesis* are interdependent aspects of engineering which reflect alternative ways of reasoning between specifications and implementations. They must be integrated if either is to reach its potential in practice. The experimentation reported here explores the interplay between *design derivation,* or synthesis by algebraic refinement, and verification by direct proof. A transformation system for digital design derivation (DDD), which is based on functional algebra, is applied to Hunt's *FM8501 processor description,* which is verified in the functional Boyer-Moore logic. A comparison of synthesized and verified architectures isolates those qualities of the implementation that require a proof of correctness. DDD is also used to reduce the FM8501 implementation to a gate level hardware description. The mechanized algebra sustains correctness as physical organization is imposed on the implementation. Thus, integrated synthesis obviates tedious proofs of behavioral equivalence at lower levels of description. The key requirement for integrating verification and synthesis is a unified treatment of abstraction. Since they were composed for the purpose of conducting a proof, the FM8501 descriptions present a good test for a synthesis system.

1. Introduction

Design verification employs theorem proving to help establish that an implementation satisfies its specification. Mechanical verification is attractive because, in principle, it allows absolute freedom in engineering. Insight and experience produce a good design which is then certified correct by a computer. However, the brief history of experience suggests that, for sequential systems, provability must be considered throughout the design process. It is so hard to engineer proofs, so one must design for verification.

Design synthesis can be characterized as the use of algebra to help correctly translate a specification into an implementation. Mechanical synthesis is attractive because, in principle, it codifies valid engineering. By constraining design to a lexicon of behavior preserving

*This research was supported, in part, by the National Science Foundation under grants numbered MIP 87-07067 and DCR 85-21497

To appear in the proceedings of the IFIP WG10.2/WG10.5 Workshop on Applied Formal Methods for VLSI, Leuven, Belgium, November 1989.

transformations, a synthesis system assures the equivalence of source and target descriptions. The history of practice for this approach is equally brief, but its limitations are also evident. An overwhelming number of design tactics must be accounted for; hence, existing synthesis systems are practical only for narrow problem classes [1].

Since human engineering practice employs both proof and algebraic refinement, automation is needed for both forms of reasoning. It is a thesis of this research that verification and synthesis must be integrated if either is to reach its potential in practice. This paper reports on a preliminary exploration of their interplay in a formal framework. A transformation system called *DDD*, under development at Indiana University, is applied to Hunt's FM8501 processor description, which was verified in the Boyer-Moore system [2]. These two systems are compatible in the sense that they operate in the same theory of first-order functional expressions. In fact, they both manipulate the same concrete syntax of Lisp s-expressions, elements of which are sketched in Section 2.

This experimentation is primarily concerned with issues of sustaining mechanically assured correctness when two (or more) systems of reasoning tools are applied to one design. Hence, the existence of a mechanized proof made the FM8501 an ideal subject for the synthesis exercise. Independent of mechanized support, we are also interested in the philosophical questions of how an intelligent designer organizes analytic and generative thinking in design. Being of a moderate size, the FM8501 also proves to be a good initial exercise for exposing issues of scale. However, because of its size, it is impossible to present the specific tactics used in the derivations. More details and several smaller examples can be found in [3, 4, 5].

Section 3 is a brief description of the DDD system. It implements specialized functional algebra for the interactive derivation of digital implementations from a certain class of applicative control specifications. DDD reduces abstract specifications to boolean network descriptions, which are fed to existing logic synthesis tools to produce circuit realizations. DDD is more like a theorem prover than a hardware synthesizer; it manipulates arbitrary functional expressions which happen to model the behavior and structure of digital networks. Ultimately, it is intended to serve as a "back end" for applicative-program transformation systems, providing a path to hardware realization.

Section 4 reviews Hunt's description and proof of the FM8501, a simple general-purpose processor. There are two main parts to Hunt's proof exercise. First, a binary representation of arithmetic is proved correct with respect to an abstract theory of integers. Second, a microcoded architecture is proved to implement an instruction-level description of machine behavior. Hunt also shows that it is possible to automatically reduce the implementation expression to a network of gates and clocked flipflops. He obtains this network by a bottom-up expansion of auxiliary function definitions, followed by an exhaustive elimination of common terms. Details of the Boyer-Moore logic, the FM8501 architecture, and the proof itself, are given in [2].

Because it operates on a compatible syntax, we were able to apply DDD directly to the published descriptions of the FM8501. In separate exercises DDD was used to manipulate the specification and implementation expressions. In the first exercise, the object was to synthesize an architecture from the specification and compare it with Hunt's verified architecture. As discussed in Section 5, there were significant differences between these descriptions, due to Hunt's use of a different memory model in his implementation. In the second exercise, the goal was to manage the translation the implementation expression into a physical description. This derivation is summarized in Section 6.

The DDD system is in early stages of development, and it was expected that these exercises would expose gaps in its synthesis capability. Particularly in the first derivation, a number of steps required manual intervention or the implementation of additional algebra. Thus, we do not claim (except, perhaps, in principle) the degree of 'mechanical correctness' that is attained in the FM8501 proofs and in similar verification exercises [6, 7, 8]. On the other hand, the new algebra dealt primarily with an unfamiliar description style, and was of a kind that one would expect in a mature transformation system.

Observations about the experimentation—it is premature to draw conclusions—are summarized in Section 7. Our perspective is that synthesis automates the mundane aspects of design, leaving it for verification to validate the inventive aspects. In contrast to [9], [10], and others, we do not see all reasoning as transformational in nature. The central issue for integrating algebra and proof is maintaining the global correctness of a design description as various reasoning tools are applied locally. Our synthesis of an FM8501 implementation came quite close to the verified description, but it could not predict the memory model chosen for the machine, nor could it produce identical microcode. The discrepancies affirm a need for verification, but their small number suggests that less *needed* to be proved about the FM8501 than was actually proved. Though much of the design is readily synthesizable, little is known about the problems for maintaining correctness in the context of synthesis. The purpose of this experimentation is to learn more. Although there is ample evidence of the need for synthesis, especially at lower levels of implementation, the manipulations of FM8501's boolean description (Section 6) illustrate that a synthesis algebra must be flexible enough to navigate conceptual hierarchies of description.

2. Notation and Terminology

The notion of correctness relates two levels of description. Following current terminology [11], the more abstract description is called a *specification* and the more concrete description is called an *implementation*. There may be many stages of implementation between an initial specification and its final *realization* in hardware. In this paper, correctness refers to a behavioral interpretation of the design notation. In the functional modeling theory used here, behavior is given by a mapping from an initial state and a sequence of inputs, to a final state and a sequence of outputs. The determination of correctness is a matter of showing that two functions are equal, or congruent under a preestablished correspondence between their domains.

In verification, the specification and implementation are given, and the task is to produce a proof of equivalence between the two. In an algebraic characterization of synthesis, an initial description is given, and the task is to develop a sequence of transformations leading to an implementation (It is somewhat misleading to use the word 'specification' for the source expression, since it is the derivation that determines the implementation.). Superficially, synthesis is a form of proof by reasoning from specifications toward implementations, while verification works in the other direction.

Both the Boyer-Moore theorem prover and the DDD transformation system manipulate Lisp *s-expressions*. Both systems deal with purely functional forms of Lisp expression, with certain syntax extensions for the sequential systems described in Section 3. That is, the manipulated forms execute directly as programs. The Boyer-Moore logic is first order. DDD is intended to be a higher order algebra, but it is essentially a first order system in its present form.

We shall sketch enough of the syntax to cover the examples in this paper. Some knowledge of Lisp expression is assumed. The definitive language references are [12] and [13]. An s-expression is either an atomic symbol, that is, a numeral or a literal, or a list, `(`$S_1 \cdots S_n$`)` of s-expressions. Square brackets, `[` $\cdots$ `]`, are sometimes used instead of parentheses. Teletype font is used for concrete syntax. In the language sketch below, upper case variables refer to expressions and corresponding lower case variables refer to an expression's value; for instance, E has value e. The language has the usual semantics of a lexically scoped, applicative-order functional dialect.

- the expression `(`$F\ E_1 \cdots E_n$`)` denotes the application of the function value f to arguments $e_1, \ldots, e_n$. Any literal other than `if`, `defn`, `lambda`, `let`, `letrec`, and `seqs` (Section 3), appearing in an applied position, refers to a defined function or a primitive operation.
- The form `(if` $P\ E_0\ E_1$`)` is an *if-then-else* expression, returning e_0 if P has a true value and e_1 otherwise.
- The form `(list` $E_1 \cdots E_n$`)` produces a list of values, `(`$e_1 \cdots e_n$`)`.
- The expression `(let ([`$X_1\ E_1$`]` $\cdots$ `[`$X_n\ E_n$`])` E`)` binds each identifier X_i to the lexical value of e_i for the evaluation of E. The keyword `letrec`, in place of `let`, forms recursive bindings.
- The special command `(defn` N `(`$X_1 \cdots X_n$`)` E`)` defines a function named N, with parameters $X_1, \ldots, X_n$, and body E. Function expressions see limited use in DDD: the form `(lambda (`$X_1 \cdots X_n$`)` E`)` denotes a function like N.

3. The DDD Transformation System

DDD (for Digital Design Derivation) is an experimental transformation system under development at Indiana University. It is essentially an s-expression editor, implementing a general design algebra on purely functional forms. DDD is implemented in the Lisp dialect Scheme.

The system is specialized toward digital design in several ways. It is integrated with existing tools for hardware generation, including boolean minimizers and synthesizers for programmable technologies. The mathematical foundations for the DDD system are established in [3], and further details about design derivation are given in [4, 14, 15].

A behavioral description in DDD is an *iterative system* of mutual function definitions:

```
(letrec (
  [ $F_1$ (lambda ($X_1$ ··· $X_n$) $E_1$) ]
    ⋮
  [ $F_m$ (lambda ($X_1$ ··· $X_n$) $E_m$) ]
) $E$ )
```

The defining expressions, $E_1, \ldots, E_m$, are tail-recursive conditional expressions

This class of defining schemata characterizes finite-state control. When iterative systems are interpreted as hardware specifications, the parameters $X_1, \ldots, X_n$ denote storage elements and I/O ports. The tail-recursive call (F_j T_1 $\cdots$ T_n) is interpreted as a transfer of control to the state F_j as the X_i are simultaneously updated by values T_i. DDD specifications are semantically equivalent to the imperative specifications of most high-level synthesis systems. In contrast to behavioral synthesis systems, DDD specifications may be expressed over any vocabulary of base constants, operations, and tests. DDD implements a 'free' algebra, independent of the underlying level of description. Implementation specific knowledge, of numeric representations for example, must be programmed into the system.

DDD translates iterative specifications into *sequential system* descriptions, which are regarded as networks of simultaneous *signal definitions.* The form of a sequential system is:

```
(seqs ( [$X_1$ $S_1$]
           ⋮
        [$X_n$ $S_n$] )
   $E$ )
```

Signal expressions, $S_1 \ldots, S_n$ are composed of variables, constants, applications of base operations, and *delay expressions.* A delay expression has the form `((! ` V`) `S`)`, where V is a base term—usually a constant or a variable—and S is another signal expression.

In the *behavioral interpretation*, the variables $X_1, \ldots, X_n$ range over infinite sequences and model synchronous activity over time. If S denotes the sequence $\langle s^0, s^1, \ldots\rangle$, then ((! V) S) denotes $\langle v,\ s^0,\ s^1, \ldots\rangle$. Applications of base operations extend to mappings: the term (F S) denotes the sequence $\langle f(s^0),\ f(s^1), \ldots\rangle$.

In the *schematic interpretation,* each expression denotes a graphical figure: delay expressions correspond to register symbols and simple terms correspond to feedback-free combinational subsystems.

Since descriptions manipulated by DDD are valid Lisp expressions, designs can be animated by running them as Lisp programs. However, the `seqs` construct is a syntax extension. For the behavioral interpretation, signals are typically modeled by stream-like representations; `seqs` is essentially `letrec` with primitive operations extended to map over these streams.

A simple example, using the *factorial* function, is shown below. An iterative system defines a factorial algorithm.

```
(letrec ([ FAC (lambda (N M)
                  (if (zero? N)
                      M
                      (FAC (- N 1) (* M N)))) ])
     (FAC X 1))
```

Considered as a control specification, FAC is a single-state machine (as are the specifications in Figures 1 and 2). FAC translates immediately to a sequential system:

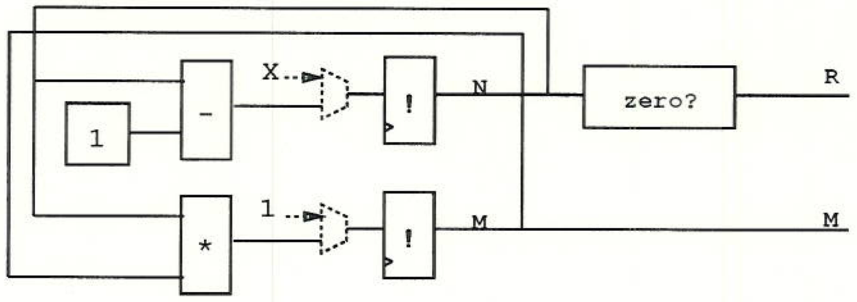
```
(seqs
  ([ N ((! X) (- N 1)) ]
   [ M ((! 1) (* N M)) ]
   [ R (zero? N) ])

  (list N M R))
```

with the behavioral interpretation,

$$N = \langle x, (x-1), (x-2),\ \ldots, 0, -1, \ldots\ \rangle$$
$$M = \langle 1, 1\cdot x, 1\cdot x\cdot(x-1),\ \ldots, x!, 0\ldots\ \rangle$$
$$R = \langle \texttt{false}, \texttt{false}, \texttt{false}, \ldots, \texttt{true}, \texttt{false}, \ldots\ \rangle$$

Signal M holds (FAC X 1) as soon as signal R holds true. The schematic interpretation of the FAC system is the network:



In this paper sequential systems are depicted by drawings like the one above (e.g. Figure 4). However, DDD manipulates expressions; it has no provisions for graphics.

Sequential systems may only abstractly describe circuits. For example, the FAC system, above, does not explain how the registers are initialized or detail synchronization with the multiplier, assuming it is sequential. Much of the DDD algebra is for manipulating structural descriptions in order to compose the actual architecture of a design. The algebra preserves the sequential interpretation of behavior while improving some aspect of the schematic interpretation.

In algebraic synthesis, structural manipulations are characterized by *factorizations,* which imposes modularity on a description. The two primary uses of factorization are to assign operations in units and to hide the representations of complex type structures. A typical derivation step in the design of a processor is to collect arithmetic operations into some number of ALU components. Similarly, complex-valued registers are encapsulated as communicating processes which hide representation details; for instance, a signal of "type memory" is factored as a component which communicating addresses and contents. The algebra of factorization is detailed in [5].

A third task of derivation algebra is to introduce representations. Once structural manipulations have isolated a realizable subsystem, a translation is made to a more concrete level of description. The final stages of synthesis reduce the design to a collection `seqs` expressions in which the base operations are boolean primitives. These gate networks are easily translated to a form accepted by gate-level minimizers and logic synthesis tools, which generate the circuit. DDD is thus integrated with Berkeley [16] (*espresso*, *mpla*, *mquilt*, *magic*) and Altera [17] CAD facilities to generate physical circuitry.

Physical organization generally follows a hierarchy that is distinct from that of a structural description. For example, a bit-slice decomposition projects much of the whole data path into every physical module. The algebra of this phase performs massive restructuring transformations to establish an appropriate physical organization. More details about this aspect of derivation are given in [4].

In summary, the DDD system addresses the three classical aspects of synthesis—control synthesis, structural manipulation, and physical organization—in an algebraic framework, translating among dialects of a single functional modeling language. Iterative function definitions are used for control specification and systems of 'stream equations' describe networks and model their behavior. When using DDD, the engineer composes a script of transformation commands which is applied to the initial design description. Derivation is guided by inspecting the intermediate expressions. DDD provides a secure algebra but it performs no automatic analysis or optimization; these must be supplied by an intelligent designer. The exercises discussed in this paper involved three to five weeks of part-time activity. Execution of a complete derivation script takes about twenty minutes to produce the design files needed for logic synthesis; however, the typical partial derivations used in design take just seconds to run. Several designs have been built in this fashion using programmable technologies (PLA and PLD).

4. The FM8501 Microprocessor Description

Hunt's FM8501 processor is a general register CPU, designed as a formal verification exercise [2]. Figure 1 is the top level of Hunt's specification, a function called `SOFT`. The full specification is a hierarchical collection of auxiliary function definitions, each of which is a closed combination. `SOFT` is an instruction level specification, analogous to a programmer's model of the machine. Six of it's seven parameters account for a file of eight general registers (`reg-file`, including a program pointer), an external memory array (`real-mem`), and four condition flags. The seventh parameter, `lst`, models the passage of time measured by instruction cycles.

Figure 2 is the top level of an implementation description called `BIG-MACHINE`. With its hierarchy of auxiliaries, `BIG-MACHINE` describes the behavior of a micro-coded instruction interpreter for FM8501. Its twenty parameters account for additional state, including an instruction register (`i-reg`), temporary accumulators (`a-reg`, `b-reg`), buffers, and so on. A global array constant, `micro-store`, issues microinstructions addressed by the `mar` register. The last two parameters model memory behavior and time, now measured in micro cycles.

Both SOFT and BIG-MACHINE are expressed over a ground type of bit-vectors and boolean logic; and both involve the description of an ALU and other function units. Coercions and interpretations are defined, which relate these binary representations to integer quantities. The abstractions support a higher level view of programming the FM8501. Equally important to the proof, they are needed for integration with the symbolic data space used to model the machines.

In the Boyer-Moore system, Hunt proved the binary representation of arithmetic and then a behavioral correspondence between SOFT and BIG-MACHINE. Thus, he gives a machine validated proof of BIG-MACHINE's correctness with respect to SOFT. Hunt goes on to reduce BIG-MACHINE to a network of gates and registers. This is done by first expanding all auxiliary function invocations and then identifying all common subexpressions.

5. Transformations on the FM8501 Source Description

In this exercise the primary goal was to derive a reasonable architecture from the instruction level specification. By design, DDD system a vehicle for interactive engineering. It provides a secure algebra, but a human designer must develop the derivation strategy. We are able to derive an architecture quite close the one Hunt proposes; judging from his sketch of an FM8501 architecture.

The function SOFT in Figure 1 is just the top level of the machine specification. It was necessary to expand auxiliary definitions in order to exposing entities of interest in an implementation. Prior to the expansion, some of Hunt's auxiliary definitions were manually altered, in order to aid visualization. The redefinitions simply permuted the formal parameter lists—the defining expressions were not changed. Of course, the corresponding actual arguments in applied occurrences were also rearranged. The expansions exposed simple gates, adders, the abstraction of a memory module, and so forth. For instance, the term in SOFT defining the next-state value for the single-bit condition flag, c-flag, expands from

```
(update-v
    (b-cc-set (current-instruction reg-file real-mem))
    c-flag
    (c (bv-alu-cv-results reg-file real-mem c-flag)))
```

to the 2,000-character term shown in Figure 3. The initial SOFT definition, together with its help functions, consists of about 6,000 characters. The expansion resulted in a specification of about 105,000 characters, but this was not a complete unfolding of the description.

The expanded version of SOFT was then transformed to reduce the implied parallelism. This major derivation step was done by hand, but the algebra has since been automated and successfully applied to comparable designs [14].

SOFT is of the same general form as the FAC example of Section 2. Considered as a single-state machine, it translates directly to a sequential system description. However, the resulting system would not map onto the target architecture because too much data is transferred in the single control state. To reduce parallelism, a serializing transformation is repeatedly applied to the specification. The process is analogous to *scheduling* in high level synthesis [18]. A call of the form $(F\ T_1\ \cdots\ T_n)$ is replaced a sequence of function calls, whose composition is equivalent to the original term. New states are introduced leading to the control point, F :

$$\begin{array}{lll}
 & \vdots & \\
[& G^1 & \texttt{(lambda } (x_1\ \cdots\ x_n)\ (G^2\ T_1^2\ \cdots\ T_n^2)\texttt{)}\ ] \\
[& G^2 & \texttt{(lambda } (x_1\ \cdots\ x_n)\ (G^3\ T_1^3\ \cdots\ T_n^3)\texttt{)}\ ] \\
 & \vdots & \\
[& G^{k-1} & \texttt{(lambda } (x_1\ \cdots\ x_n)\ (F\ T_1^k\ \cdots\ T_n^k)\texttt{)}\ ] \\
 & \vdots &
\end{array}$$

The original call is replaced by $(G^1\ T_1^1\ \ldots\ T_n^1)$, and the terms T_j^i are chosen so that this expression is reducible to $(F\ T_1\ \cdots\ T_n)$. The construction of intermediate terms is guided by architectural constraints. For example, if only one application of the `ALU` function is permitted in a clock cycle—if there is just one ALU device—there can be no more than one occurrence of the symbol `ALU` in $\{T_1^i,\ \ldots,\ T_n^i\}$ for any i.

This phase of derivation also adds new registers to hold intermediate results. For instance, a parameter `i-reg` was added to hold the current instruction. About eighteen serialization steps produce a system with thirteen distinct control states. The `BIG-MACHINE` microcode holds fourteen instructions. The discrepancy is explained in a moment.

Subsequent algebra is performed entirely by the DDD system. A sequential system is built from the expanded and serialized version of `SOFT`. It is initially composed of a subsystem for control and an abstract description of architecture. Several factorizations are applied to identify common terms, combine operations, and encapsulate modules such as the register file and the external memory. The resulting sequential system, shown in Figure 4(b), closely resembles Hunt's block diagram of FM8501, shown in Figure 4(a) [2, Fig. 1-1].

Not all of the differences between these systems are significant. For example, feeding `a-reg` and `b-reg` through the ALU—anticipating a conventional bus organization—is not entailed by the `SOFT` specification. Also, the micro-control of FM8501 could not be synthesized by DDD, which instead builds a hard-wired controller.

The significant difference between `BIG-MACHINE` and the derived architecture is the absence in the latter of the `reset`, `dtack`, `read`, `write`, and `no-store` registers. Since there is no mention of these entities in `SOFT`, one would not expect them to arise in a derivation. Except for `reset`, which might have appeared in the specification, these registers implement a synchronization protocol with external memory. This protocol also accounts for surplus microinstruction mentioned earlier. Registers `dtack` and `read/write` reflect a change in the abstraction of memory behavior.

In `SOFT`, Hunt models memory as a functional abstraction. At each step of execution `SOFT` constructs a new memory value reflecting the effect of the current instruction. This is the conventional treatment in functional modeling, and it is consistent with way things are done in DDD. The `SOFT` derivation produced a description in which memory, like the multiplier in `FAC` (Section 2), it behaves as a combinational process.

In `BIG-MACHINE`, memory is a process abstraction, whose introduction raises valid tasks for verification. While 'standard' synchronization protocols—`reset`, for example—might be anticipated by a synthesis system, more complex forms of coordination could not. Thus, there is really something to prove about BIG-MACHINE, namely, that it interacts correctly with the chosen model of memory.

In integrating synthesis with verification, one would like to be able to isolate those details that require direct proof. DDD did not invent the memory protocol but it correctly synthesized the rest of the circuit. Under the same hypotheses of communication, it might be easier to prove equivalence between DDD's version architecture and BIG-MACHINE's, but such a proof has not been attempted.

A more finely grained interplay between algebraic derivation and verification is illustrated in Figure 5. The FM8501 block diagram contains a subsystem selecting the outputs of two operations, inc and dec (Figure 5(a)). A DDD factorization combines these into a single component (Figure 5(b)). This is not a mere rewriting. It is correct only because just one output is used at any time. In addition to determining this, DDD had to introduce an instruction signal to control the incrementer/decrementer. It may be non-trivial to reverify the entire implementation after a local refinement such as this, so it is better to employ a secure algebra. On the other hand, DDD has introduced a new component, inc/dec, and it now becomes necessary to verify the subcircuit used to implement it.

6. Transformations on the FM8501 Target Description

The second derivation exercise carries the BIG-MACHINE description to the point of hardware realization. The exercise shows how transformations are used to manage details as descriptions are reorganized toward geometric goals. This stage of derivation often involves massive revisions to the structure of a description because the physical organization of the implementation has little in common with its functional hierarchy.

Figure 6 gives a sense of the derivation goal, although it does not portray the actual organization. The task is to reduce BIG-MACHINE to a physically meaningful collection of boolean subsystem descriptions, for input to logic synthesis tools. Registers and certain combinational functions are partitioned into bit slices, shown as heavily outlined boxes. DDD's role is to maintain correctness both within and among the subsystems.

Hunt performed this compilation *en masse* by doing a bottom-up expansion of BIG-MACHINE, followed by an exhaustive elimination of common subterms [2, Ch. 6]. The fully expanded definition is composed of roughly eleven million gates, but it reduces to a network of about 350 signals with roughly 1,800 gates. Since the growth of the intermediate expression is exponential, a top-down expansion would be required for a larger design [19]. By using DDD to manage the expansion, the explosion of intermediate terms was contained. First, an architecture, described in terms of vectors, numbers, and symbolic quantities, was derived as in the previous section. Next, the description was expanded to the binary representation level and restructured toward a physical organization. The intermediate terms were much smaller—on the order of 100,000 characters with the largest structure actually inspected being about 10,000 characters. The penalty was a larger implementation (estimated to be twice the size of one automatically generated) with better routing complexity.

The 1,800-gate BIG-MACHINE could almost certainly be automatically assembled to VLSI using current logic synthesis tools. Hence, using DDD to reorganize this design is hardly a telling demonstration for interactive synthesis at the gate level. However, larger designs would require management, and even for FM8501, mechanized algebra might play a role in meeting performance goals. What the exercise *does* demonstrate is the need to deal with the descriptive abstractions used in verification, in order sustain mechanically assured correctness in later stages of implementation.

Like SOFT, BIG-MACHINE is an iterative algorithm from which an initial system description was mechanically constructed. Unlike SOFT, BIG-MACHINE is truly a one-state machine: control is represented by a microprogram store (mar in Figure 2, mar/ROM in Figure 4(a)). Hence, the initial system factors into the architecture of Figure 4(a) without the introduction of additional control states.

By itself, Figure 2 is not a very illuminating specification. As in the SOFT derivation, the first step was to expand BIG-MACHINE far enough to expose relevant operations. A separate auxiliary function is used to compute the next value of each register. For instance, updating the condition flag depends on the current c-flag, the instruction, the microinstruction, and the ALU outputs. The c-flag register is updated by the c-flag combination:

```
(defn c-flag (c-flag a-reg b-reg i-reg mar)
  (update-v (b-and (b-we-alu-result (micro-rom mar))
                   (b-cc-set i-reg))
            c-flag
            (c (bv-alu-cv a-reg b-reg c-flag
                          (bv-alu-op-code i-reg)))))
```

The architecture derived in DDD from BIG-MACHINE was identical to Hunt's block diagram (Figure 4(a)). This system description was restructured into a bit-slice organization. First, signals participating in bit slices were isolated as a subsystem of the design. Most of the registers, selectors, and constants were included, as well as certain bit-parallel operations. Also incorporated were operations for field extraction and type coercion. These reflect the logical structure of the design, but in the bit-slice decomposition they either disappear or project to identity functions.

A mapping was defined showing how the registers are partitioned into bit slices. The DDD system decomposed the subsystem into sixteen parallel systems of boolean equations. These subsystems were assembled to programmable targets (PLA and PLD) using available logic synthesis tools. The FM8501 was not actually built, but a several designs of similar size and architecture have been [4, 14, 15].

7. Observations and Directions for Further Work

If mechanically assured correctness is to be carried into practice, a broad view of formal methods must be taken. Proving correctness—however defined—reflects just one facet of reasoning in engineering. Synthesizing correct implementations reflects another. Both forms of thought are used in any significant design effort. The benefits of automated reasoning cannot be accurately judged, nor can methodology be fully developed, until there is integrated support for both approaches.

A key issue is the management of abstraction. The FM8501 descriptions are composed primarily for the purpose of conducting a proof. They reflect conceptual structures, saying little about physical organization. It is necessary to impose other hierarchies—of architecture and geometry, for example—in order to interface with realization tools. The entailed reverification descriptions can be taxing, as witnessed by Cohn [7], Joyce [8], and others.

As these exercises demonstrate, a mechanized algebra can maintain correctness in the passage from one realm of description to another, and as well, from one level of representation to another. In order to integrate with verification however, there must be flexible support of formal abstraction methods. DDD is a formal system by virtue of its grounding in an abstract functional theory. It implements enough algebra over first order applicative expressions to work directly with Boyer-Moore notation. With some development, DDD was able to navigate the abstractions employed by Hunt to prove the FM8501.

Using DDD, we could engineer the reduction of `BIG-MACHINE` to a realizable description. The outcome suggests that correctness at the register transfer level is a reasonable departure point for sequential system verification: it appears practical to synthesize from this level into silicon. It should not be necessary, for example, to prove that a particular physical organization is correct with respect to a low-level structural description; and indeed, Hunt attempts to synthesize a gate-level network directly from `BIG-MACHINE`.

Both `SOFT` and `BIG-MACHINE` are expressed in terms of binary representations. They might have been described over a more abstract basis, involving entities such as *integer* and *address*. DDD is capable of incorporating binary representations of these values. Thus, the machine correctness proof might have been conducted at a higher level of description. However, correct algebraic synthesis depends on *correct* representations; thus, verification still plays the pivotal role in establishing the validity of a design.

The attempted derivation of `BIG-MACHINE` from `SOFT` exposed the inventive aspects of Hunt's proof exercise. DDD was *nearly* able to produce from `SOFT` the machine that Hunt intended. Whether it might be easier to prove `BIG-MACHINE` correct with respect to the derived architecture for `SOFT`, rather than `SOFT` itself, is a question for future research. For example, DDD did not generate the microcode representation of control designed by Hunt, but it would be fairly easy to prove the microcode sequencer correct with respect to the controller.

In `SOFT`, memory is essentially a functional abstraction, while in `BIG-MACHINE` it is a sequential process. Hunt's proof entails a congruence between time models, as well as the introduction of a synchronization protocol. DDD does not currently support any form of sequential-process decomposition. Although this will be remedied in the future, a dependence on verification will remain: translating the memory-function into a memory-process induces new verification conditions for the system and its peripheral environment. One would expect an integrated synthesis system to generate these verification conditions.

DDD is in early stages of development and these exercises must be regarded as an illumination of issues rather than a practical demonstration. At the front end, it was necessary to develop means to introduce of control states in `SOFT`. This is a fairly deep analysis problem, receiving much attention in high-level synthesis research. However, the algebra itself is fairly simple. We also added basic editing functions, such as the expansion of combinators, and some *ad hoc* transformations, such as those for reordering parameter lists. More experience is needed to determine the impact of these low-level facilities on design management.

Broader targeting capabilities are also planned. The goal in DDD development is to bring high-level descriptions to the point that logic synthesis tools are applicable. However, we lack a sufficient formal characterization of logic-synthesis tools. The narrow path to realization through programmable technologies will expand as new tools are understood.

Although Hunt's descriptions exposed no unexpected problems for the algebra, minor changes in the style of expression would have simplified the derivation task. Since the initial step is always to expand definitions it is better if relevant features appear higher in the definition hierarchy. Here is a typical example. The auxiliary function `update-v-nth`, below, conditionally alters one element a vector of bits. After coercing the binary representation, `v-n`, to an index, `update-nth` performs the replacement on the list object representing the vector.

```
(defn update-v-nth (c v-n lst value)
  (update-nth c (bv-to-nat v-n) lst value))

(defn update-nth (c n lst value)
  (if (and (truep c)
           (listp lst))
      (if (zerop n)
          (cons value (cdr lst))
          (cons (car lst) (update-nth c (sub1 n) (cdr lst) value)))
      lst))
```

`Update-nth` is providing with a metalogical model of vectors, for which `update-v-nth` is the logical interface. Now, the update is predicated on the boolean value `c`. The `(truep c)` test is of interest to the hardware implementation, but the list representation of vectors is not. Thus, moving this predicate into `update-v-nth` eliminates a level of expansion. Such a change is unlikely to impair a correctness proof, but could have a substantial impact on the size of the manipulated expressions. In other words, the interplay of derivation and verification is finely grained indeed.

References

[1] Borriello, Gaetano and Detjens, Ewald, High-level synthesis: current status and future directions, *Proceedings of the 25th ACM/IEEE Design Automation Conference,* 1988.

[2] Hunt, Warren A., Jr., *FM8501: A Verified Microprocessor*. Ph.D. dissertation, The University of Texas at Austin. Also published as Technical Report 47 (Institute of Computing Science, The University of Texas at Austin, Austin, 1985).

[3] Johnson, Steven D., *Synthesis of Digital Designs from Recursion Equations* (The MIT Press, Cambridge, 1984).

[4] Johnson, Steven D., Bhaskar Bose, and C. David Boyer, A tactical framework for digital design," in: Birtwistle, G. and Subrahmanyam, P.A. (eds.), *VLSI Specification, Verification and Synthesis,* (Kluwer, Boston, 1987) 349–384.

[5] JOHNSON, STEVEN D. Manipulating Logical Organization with System Factorizations, in: Leeser, M. and Brown, G. (eds.), *Hardware Specification, Verification and Synthesis: Mathematical Aspects,* Proceedings of the Cornell Mathematical Sciences Institute work shop, July 1989 (Springer, New York, in preparation).

[6] GORDON, M. J. C., Proving a Computer Correct, University of Cambridge, Computer Laboratory, Technical Report No. 42, 1983.

[7] COHN, AVRA, Correctness Properties of the Viper Block Model: The Second Level, preliminary papers for the Banff Hardware Verification Workshop, University of Calgary, 1988.

[8] JOYCE, JEFFREY J., Formal Specification and Verification of Microprocessor Systems, in: Winter, S. and Schummy, H. (eds.)i*Euromicro '88,* Proceedings of the 14th Symposium on Microprocessing and Microprogramming, Zurich, 1988 (North-Holland, Amsterdam, 1988).

[9] VERKEST, D., P. JOHANNES, L. CLAESEN, AND H. DE MAN Formal Techniques for Proving Correctness of Parameterized Hardware using Correctness Preserving Transformations, in Milne, George J. (ed.), *The Fusion of Hardware Design and Verification* (North-Holland, Amsterdam, 1988) 77–98

[10] BOUTE, RAYMOND T., System semantics and formal circuit description, *IEEE Transactions on Circuits and Systems* **CAS-33**(12):1219-1231 (1986).

[11] CAMUARATI, PAOLO AND PAOLO PRINETTO, Formal Verification of Hardware Correctness: Introduction and Survey of Current Research, *Computer* **21**(7):8–19 (1988).

[12] BOYER, R.S AND J.S. MOORE, *A Computational Logic* (Academic Press, New York, 1979).

[13] REES, JOHATHAN AND CLINGER, WILLIAM, The Revised^3 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices* **21**:12 (1986).

[14] BOYER, C. DAVID AND STEVEN D. JOHNSON, Using the Digital Design Derivation System: case study of a VLSI garbage collector implementation, in: Darringer, J. A., and Ramming, F. J. (eds.), *Computer Hardware Description Languages and their Applications,* (Elsevier, Amsterdam, in preparation).

[15] WEHRMEISTER, ROBERT M., Derivation of an SECD Machine: Experience with a Transformational Approach to Synthesis, (Indiana University Computer Science Department Technical Report No. 290, Bloomington, September 1989).

[16] SCOT, WALTER S., ROBERT N MAYO, GORDON HAMACHI, AND JOHN K. OUSTERHOUT, 1986 VLSI Tools, Report No. UCB/CSD 86/272, (Computer Science Division—EECS, University of California at Berkeley, 1985).

[17] ALTERA CORPORATION, *Altera Programmable Logic User System User Guide (Version 4.0)* (Altera Corporation, Santa Clara, 1985).

[18] DANIEL D. GAJSKI, DANIEL D.(ED.), *Silicon Compilation* (Addison Wesely, Reading, 1987).

[19] HUNT, WARREN A., JR., personal communication.

```
(defn SOFT (reg-file real-mem c-flag v-flag z-flag n-flag lst)
  (if (nlistp lst)
      (list reg-file real-mem c-flag v-flag z-flag n-flag)
      (SOFT (reg-file-after-oprd-b-post-increment
               reg-file real-mem c-flag v-flag z-flag n-flag)
            (real-mem-after-alu-write
               reg-file real-mem c-flag v-flag z-flag n-flag)
            (update-v
               (b-cc-set (current-instruction reg-file real-mem))
               c-flag
               (c (bv-alu-cv-results reg-file real-mem c-flag)))
            (update-v
               (b-cc-set (current-instruction reg-file real-mem))
               v-flag
               (v (bv-alu-cv-results reg-file real-mem c-flag)))
            (update-v
               (b-cc-set (current-instruction reg-file real-mem))
               z-flag
               (zerop (bv-to-nat (bv (bv-alu-cv-results
                    reg-file real-mem c-flag)))))
            (update-v
               (b-cc-set (current-instruction reg-file real-mem))
               n-flag
               (negativep (bv-to-tc (bv (bv-alu-cv-results
                    reg-file real-mem c-flag)))))
            (cdr lst))))
```

FIGURE 1

The SOFT specification of FM8501 [2]. (Copyright ©1985 Warren A. Hunt, Jr. Reprinted with permission of the author).

```
(defn BIG-MACHINE (mar read write dtack reset no-store data-out
                   reg-file addr-out c-flag v-flag z-flag n-flag
                   a-reg b-reg i-reg visual-mem real-mem
                   memory-watch-dog-history oracle)

  (if (nlistp oracle)

      (list mar read write dtack reset no-store data-out reg-file
            addr-out c-flag v-flag z-flag n-flag a-reg b-reg i-reg
            visual-mem real-mem memory-watch-dog-history)

      (BIG-MACHINE (mar mar i-reg dtack reset no-store)
                   (read mar i-reg)
                   (write mar i-reg no-store)
                   (dtack (car oracle))
                   (reset (car oracle))
                   (no-store no-store c-flag v-flag z-flag n-flag
                             i-reg mar)
                   (data-out data-out a-reg b-reg c-flag i-reg mar)
                   (reg-file reg-file data-out i-reg mar no-store
                             reset)
                   (addr-out addr-out reg-file i-reg mar reset)
                   (c-flag c-flag a-reg b-reg i-reg mar)
                   (v-flag v-flag a-reg b-reg c-flag i-reg mar)
                   (z-flag z-flag a-reg b-reg c-flag i-reg mar)
                   (n-flag n-flag a-reg b-reg c-flag i-reg mar)
                   (a-reg a-reg visual-mem reg-file i-reg mar reset)
                   (b-reg b-reg visual-mem reg-file i-reg mar reset)
                   (i-reg i-reg visual-mem mar)
                   (visual-mem real-mem read write addr-out
                               memory-watch-dog-history
                               (dtack (car oracle))
                               (reset (car oracle)))
                   (real-mem real-mem read write addr-out data-out
                             memory-watch-dog-history
                             (dtack (car oracle))
                             (reset (car oracle)))
                   (watch-dog read write (dtack (car oracle))
                              data-out addr-out)
                   (cdr oracle)
                   )))
```

FIGURE 2

The BIG-MACHINE description of FM8501 [2]. (Copyright ©1985 Warren A. Hunt, Jr. Reprinted with permission of the author).

```
(update-v
    (b-cc-set (current-instruction reg-file real-mem))
    c-flag
    (c (bv-alu-cv-results reg-file real-mem c-flag)))
```

before expansion

```
(if (b-cc-set (v-nth real-mem (nth reg-file 0))) (c ((bv-alu-op-code
(v-nth real-mem (nth reg-file 0))) bv-alu-cv (if (b-direct-reg-a
(v-nth real-mem (nth reg-file 0))) (v-nth (update-nth reg-file t
0 (v-nat-inc (nth reg-file 0))) (bv-oprd-a (v-nth real-mem (nth
reg-file 0)))) (if (b-indirect-reg-a-dec (v-nth real-mem (nth
reg-file 0))) (v-nth real-mem (v-nat-dec (v-nth (update-nth reg-file
t 0 (v-nat-inc (nth reg-file 0))) (bv-oprd-a (v-nth real-mem (nth
reg-file 0)))))) (v-nth real-mem (v-nth (update-nth reg-file t 0
(v-nat-inc (nth reg-file 0))) (bv-oprd-a (v-nth real-mem (nth
reg-file 0))))))) (if (b-direct-reg-b (v-nth real-mem (nth reg-file
0))) (v-nth (update-v-nth (update-nth reg-file t 0 (v-nat-inc (nth
reg-file 0))) (b-indirect-reg-a-dec (v-nth real-mem (nth reg-file 0)))
(bv-oprd-a (v-nth real-mem (nth reg-file 0))) (v-nat-dec (v-nth
(update-nth reg-file t 0 (v-nat-inc (nth reg-file 0))) (bv-oprd-a
(v-nth real-mem (nth reg-file 0)))))) (bv-oprd-b (v-nth real-mem
(nth reg-file 0)))) (if (b-indirect-reg-b-dec (v-nth real-mem
(nth reg-file 0))) (v-nth real-mem (v-nat-dec (v-nth (update-v-nth
(update-nth reg-file t 0 (v-nat-inc (nth reg-file 0)))
(b-indirect-reg-a-dec (v-nth real-mem (nth reg-file 0))) (bv-oprd-a
(v-nth real-mem (nth reg-file 0))) (v-nat-dec (v-nth (update-nth
reg-file t 0 (v-nat-inc (nth reg-file 0))) (bv-oprd-a (v-nth
real-mem (nth reg-file 0)))))) (bv-oprd-b (v-nth real-mem (nth
reg-file 0)))))) (v-nth real-mem (v-nth (update-v-nth (update-nth
reg-file t 0 (v-nat-inc (nth reg-file 0))) (b-indirect-reg-a-dec
(v-nth real-mem (nth reg-file 0))) (bv-oprd-a (v-nth real-mem
(nth reg-file 0))) (v-nat-dec (v-nth (update-nth reg-file t 0
(v-nat-inc (nth reg-file 0))) (bv-oprd-a (v-nth real-mem (nth
reg-file 0)))))) (bv-oprd-b (v-nth real-mem (nth reg-file 0)))))))
c-flag)) c-flag)
```

after expansion

FIGURE 3
SOFT's next-state value for `c-flag`

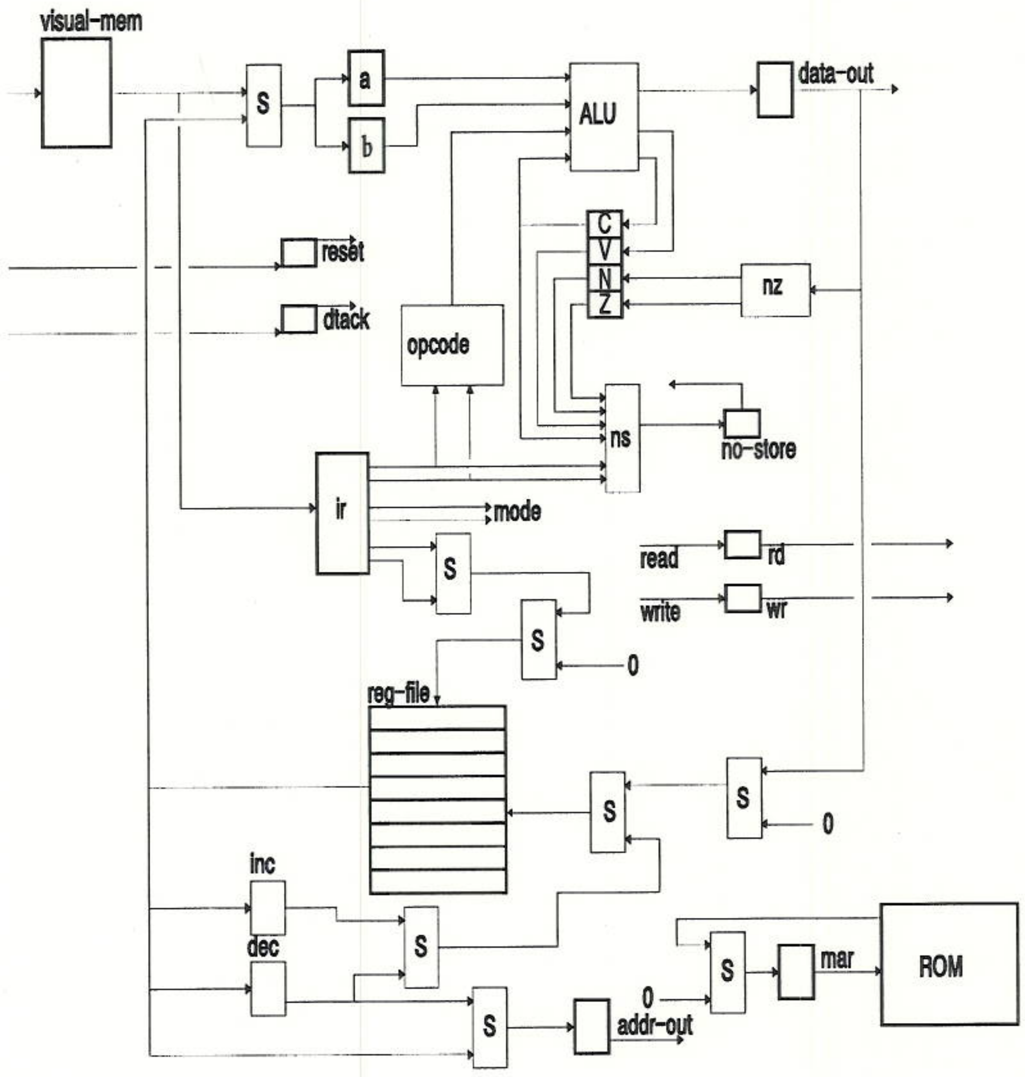


FIGURE 4(A)
Hunt's architecture for BIG-MACHINE [2, Fig. 1–1]

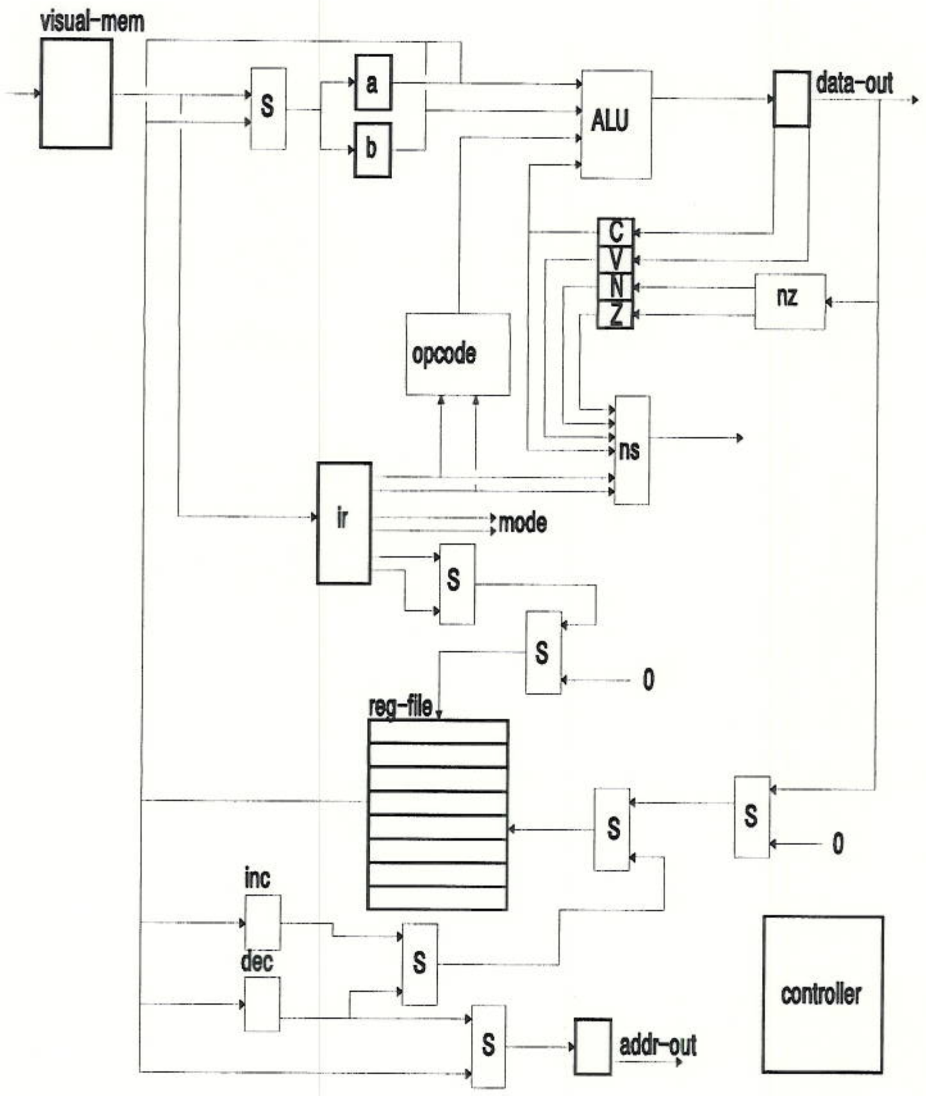


FIGURE 4(B)
Architecture derived from SOFT by DDD

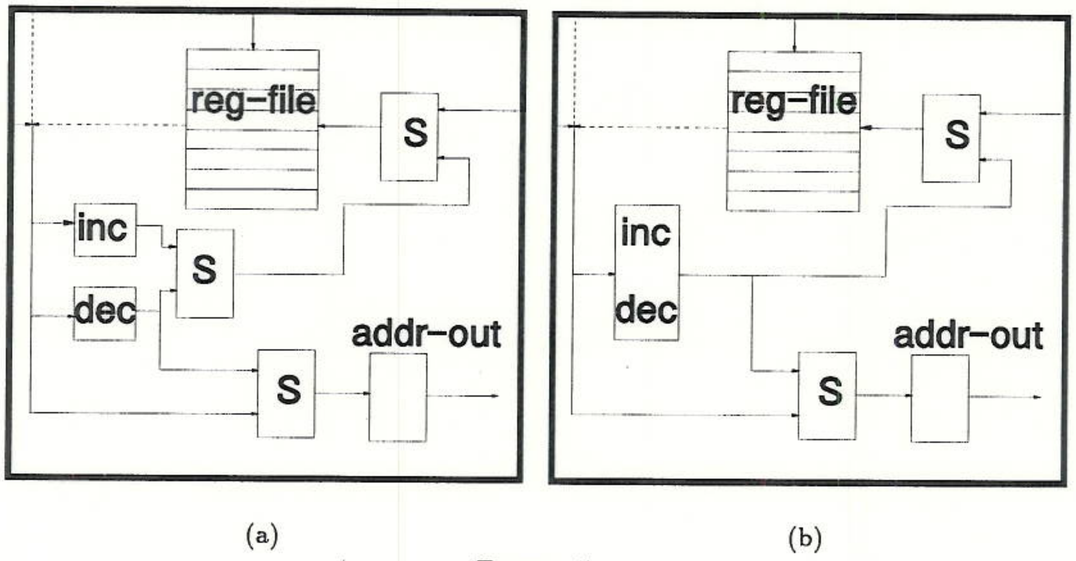


FIGURE 5
Detail of a Local Factorization.

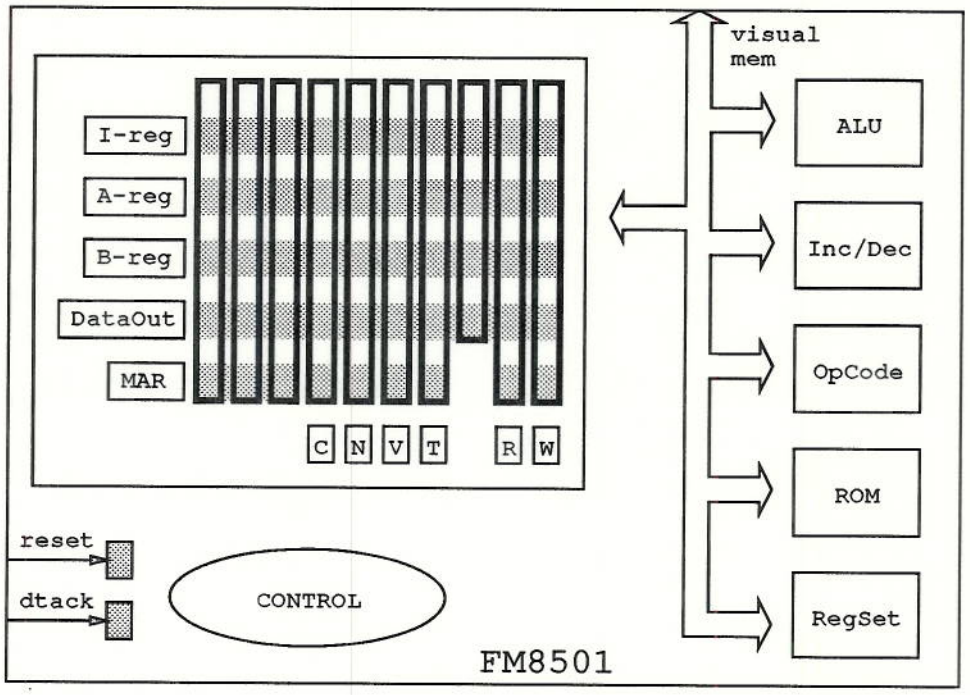


FIGURE 6
Physical Organization of `BIG-MACHINE`