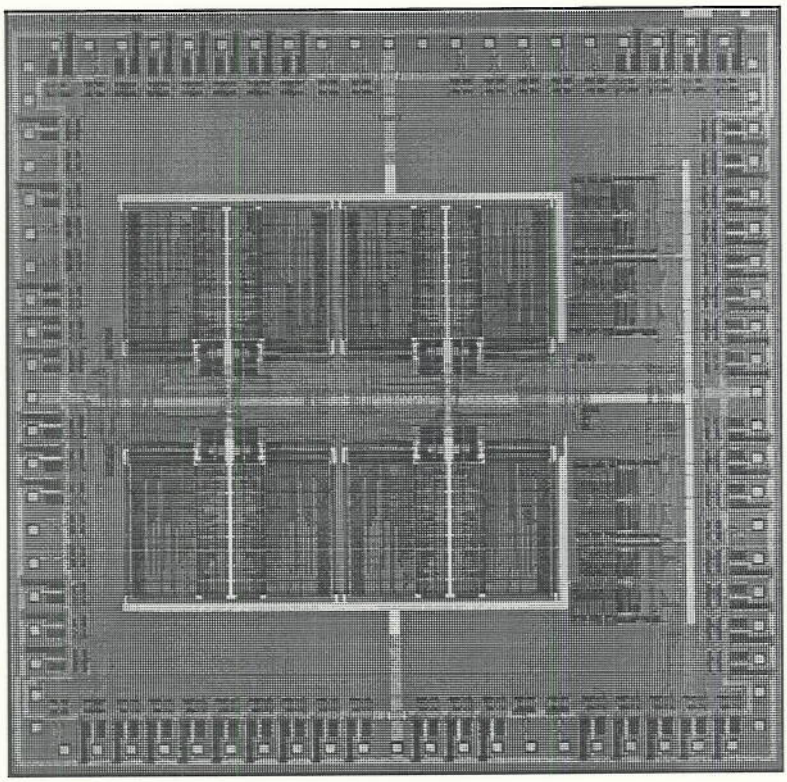
TECHNICAL REPORT NO. 274

Using the Digital Design Derivation System: Case study of a VLSI garbage collector implementation

by

C. David Boyer and Steven D. Johnson

April, 1989



COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

TECHNICAL REPORT NO. 274

Using the Digital Design Derivation System: Case study of a VLSI garbage collector implementation

by

C. David Boyer and Steven D. Johnson

April, 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

Using the Digital Design Derivation System:
Case study of a VLSI garbage collector implementation*

C. David Boyer and Steven D. Johnson
Computer Science Department
Indiana University
Bloomington, Indiana

The *DDD* transformation system, under development at Indiana University, reflects an approach to digital design synthesis based on a purely functional algebra. The system has been used to derive several working designs realized in PLD (programmable logic device) technologies. This paper reports on the reimplementation of one of these designs, a garbage collector, in VLSI. The design example provides a context for discussing the method of synthesis and its mechanization. A generic system of control transformations was developed to aid in meeting the architectural constraints of the VLSI target. The exercise also illustrates the role of the algebra in managing the translation from conceptual architecture to physical organization.

1. Introduction

A stop-and-copy garbage collector is implemented in VLSI using a Digital Design Derivation (DDD) system currently under development at Indiana University. DDD reflects a formal approach to digital design synthesis based on the algebraic manipulation of purely functional expressions. It is the mechanized part of a design method whose benefits include the support of abstraction, provisions for animation, and a unified foundation for design automation. The project described here is part of a continuing effort to exercise the approach at practical levels of engineering.

DDD implements basic algebra for the interactive synthesis of synchronous systems. At this stage of its development, it is essentially an editor for making correctness preserving transformations on certain applicative specifications. These transformations may be applied at various levels to synthesize hardware system descriptions. A *design derivation* is that sequence of transformations used to produce an implementation. DDD is integrated with existing logic synthesis tools, currently PLD programmers and PLA layout assemblers. Thus, the derivation goal in DDD is to produce a realizable boolean system description. Section 2

---

*Research reported herein was supported, in part, by the National Science Foundation under grants numbered MIP 87-07067 and DCR 85-21497.
Paper to appear in: *IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages*, June, 1989.

is a brief review of formal research leading to the implementation of DDD and previous experimentation with the system.

Earlier, a PLD prototype garbage collector was derived with these tools [6]. This paper describes adaptations of DDD to reimplement the same design in VLSI. One of the requirements of the project was to use the same initial description for both implementations. Section 3 is a discussion of this behavioral specification which is shown in Appendix A. The original specification held implications for architecture that could not be attained in the VLSI target. For instance, the PLD implementation drove two parallel memories and included three adder circuits. Due to pin and area limitations, memory accesses and arithmetic had to be serialized. New transformations were developed for this kind of refinement. These are discussed in Section 4.

The underlying purpose of this project was to expose issues for formal and practical development. An emerging thesis of this research is that a flexible low level algebra is needed to manage the physical reorganization of digital descriptions. The data path of the VLSI collector is realized in three chips. The latter stages of derivation are concerned with sustaining correctness through the massive restructurings involved. Section 5 is a discussion of this aspect.

DDD transforms executable expressions. Section 6 explains how animation of the design notation was used in testing and simulation. The conclusion surveys directions for future research and experimentation.

2. The Approach to Synthesis

The DDD system embodies a general approach to system synthesis as applied to digital design. A functional algebra is employed to manipulate formulas modeling the structure and behavior of hardware systems. An implementation is derived by applying a sequence of correctness preserving transformations to a higher level description. The syntactic medium is a dialect of lambda calculus augmented with forms for recursive definition. This language enjoys a simple algebra. A goal of formal research is to study the algebra of synthesis by distilling it to a collection of basic laws. The same strategy is followed by Sheeran, who applies her work to a different class of design problems [15].

Johnson established a formal connection between the use of recursive function definitions to model control and the use of recursive sequence (or 'stream') definitions to model synchronous systems [9]. The theoretical foundation provides a framework for dealing with abstraction in design. DDD implements algebra that is independent of the *basis*—or ground vocabulary of constants, operations, and tests—to which it is applied. Although there is some back-end specialization for boolean terms, knowledge of particular bases (e.g. binary arithmetic) must be developed orthogonally.

In segregating the algebra of the metalanguage, the aim is to assure that the methodology and its design tools are adaptable to higher levels of description and disparate implementation targets. This quality distinguishes DDD from silicon compilers (as found in [5]), which

employ a substantial built-in expertise about representations and problem classes. DDD is more like a mechanized logic, oriented to synthesis rather than proof. It is conceived as an interactive editing vehicle, providing basic algebra for securing correct implementations relative to the level of description. It also provides programming tactics for moving between levels.

Techniques for manipulating complex bases are developed in [7, 8], where higher order constructs are used to gain representation independence and user-defined data types are used for modularity. The conceptual organization of a design—it's structural description—is manipulated by *system factorizations* [6], which decompose designs into communicating subsystems. These transformations encapsulate external processes, such as memories and I/O ports, and develop the architecture of the data path (See Section 5).

DDD is implemented in the Lisp dialect Scheme [13]. It operates on purely functional forms represented by Lisp s-expressions. In this sense it is a program transformation system. Given definitions of the base operations, a description at any stage of derivation executes directly as a Scheme program (with syntax extensions for streams). One major benefit of this automation strategy is that the symbolic processing capabilities of Scheme are available to model the design. Good animation techniques reduce the need to develop interpreters and extractors around the design language. Two examples are presented in Section 6. The same text that was input to the DDD system for digital synthesis was run in a production environment to test the design, and the resulting execution traces were used as test vectors for a switch level simulation of the derived circuit. All this was done directly in Scheme.

By convention, we refer to the source design description as a *specification,* the derived target description as an *implementation,* and the physical product as a *realization.* The terminology is misleading in the sense that a source description is only one component of the complete specification. The complete specification also includes the sequence of DDD commands that, when applied to the initial description, composes the implementation. These transformations impose a conceptual architecture for the design, incorporate boolean representations for the basis, and manipulate the physical organization of the realization. The last of these aspects is the least developed, but perhaps the most important for practical applications. Exploration of the algebra for physical decomposition was a primary motive for this exercise.

3. Stop-and-copy: Specification and Algorithm

Heap processing languages make memory available to the processor through allocation operations. These instructions invoke a storage manager which returns fresh cells of memory. There are no deallocation instructions. A cell becomes free when it is inaccessible through the processor's registers. When memory is exhausted, the allocator calls a garbage collector to reclaim any free cells.

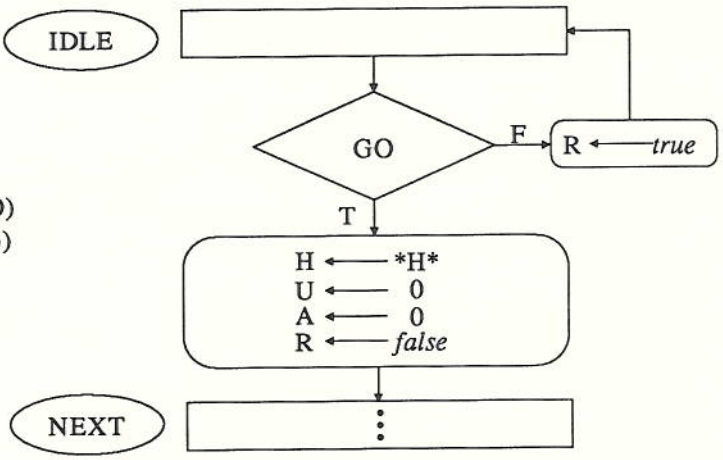
A stop-and-copy collector [3, 4] divides memory into two semispaces, only one of which is used at any time. The collector copies and compresses the heap image to the inactive

(new) space. The roles of the semispaces are then exchanged and the allocator resumes control. This collector is considerably faster than mark-sweep varieties because it visits each cell fewer times and never visits free cells. The disadvantage of stop-and-copy is the space overhead; only half of memory is usable. Originally, this algorithm was chosen for hardware implementation for the speed improvements possible with parallel semispaces. A PLD prototype was implemented which drove two memory units. The implementation ran at sixty times the estimated rate of an M68000 based software benchmark.

The specification used for stop-and-copy is shown in Appendix A. The twelve function definitions form an *iterative system* called GC. The function definitions are parameterized conditionals, composed of if expressions and case expressions. Each of the conditional branches is tail recursive. The translation of more general recursion schemes to this form has been widely studied; basic results are outlined in [9]. DDD synthesis currently begins with iterative specifications.

GC can be viewed as an Algorithmic State Machine (ASM) description [16]. Each function represents a point of control. The parameter lists declare the 'registers' at this level of description. Each function invocation can be thought of as a parallel assignment to the registers and a transfer of control. The function IDLE from GC and an equivalent ASM flow diagram are shown below. An oval to the left of the box contains the name of the state. Statements within boxes with square corners denote unconditional actions for that state. A diamond contains a control choice, leading to boxes with rounded corners containing conditional actions for that state. Actions in a given state are performed in parallel. In state IDLE, if GO is true, then registers M, D, C, and GO remain unchanged and registers H,

```
(IDLE (lambda (M H D C U A R GO)
  (if GO (NEXT M *H* D C 0 0 false GO)
         (IDLE M H D C U A true GO))))
```



U, A, and R are updated with values *H*, 0, 0, and *false*, respectively, and NEXT becomes the new control-point. If GO is false, then R is loaded with *true* and the control-point does not change. ASM diagrams are used to illustrate the effect of control transformations discussed later.

The basis for the GC specification is an aggregate of several complex types. Parameter M has type *memory*, which in turn is parameterized by *address* and *content* entities. M is

subject to 'read' operations, R_O and R_N, and 'store' operations, W_O, and W_N, for the old and new semispaces. In addition there is an operation M_{FLIP} which toggles the status of the semispaces. Registers H and D each hold a *content*, subject to field manipulation operations *ptr*, *tag*, and *cell*. C, U, and A are of type *address* with arithmetic operations *inc, dcr, add, addinc*, and *btow* (the last of these rounds string lengths to word boundaries). Parameters GO and R represent boolean communication ports, used for hand shaking between the collector and the allocator.

The basis is not interpreted by the DDD system; the names used for operations and tests are arbitrary. There is no built in understanding of arithmetic and its laws, or any other primitive type structures. DDD currently operates on an untyped notation, which means that it lacks facilities to check for consistent use of the basis. The system relies on the correct use of operations; one reason for executing the behavioral description is to determine if there are type errors.

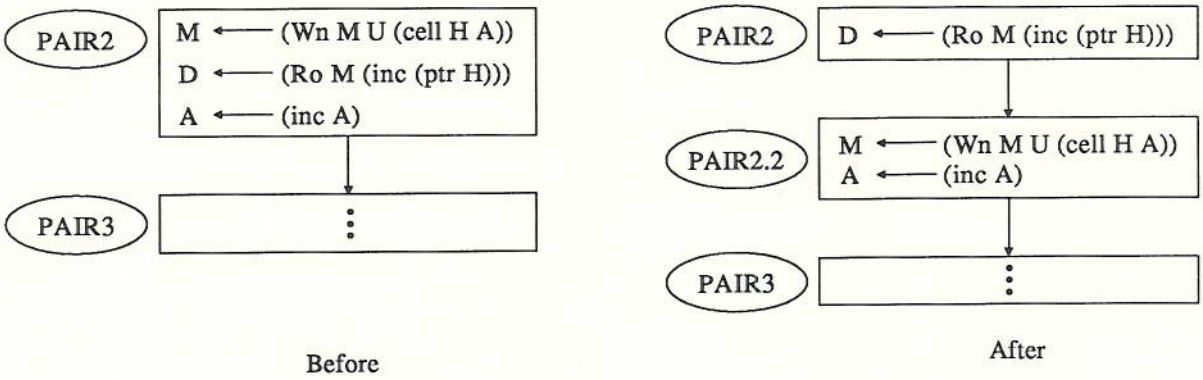
GC is the designer's initial estimate of architecture. The register transfers define the inner connectivity of the machine. Operations such as *add* abstractly refer to physical circuitry. However, distinct operations are not necessarily implemented by distinct subcircuits; one ALU may implement several arithmetic operations. Ideally, the specification is a description of the algorithm, in a compact, natural style, free of presumptions about devices and architecture. It is the job of the transformation system to compose an architecture.

4. Serialization of the Specification

Each function call in the GC description represents a set of actions to be accomplished in a clock cycle. However, in some cases the actions cannot be parallel because of implementation constraints. For example, in the function PAIR2, shown to the left below, there are four parallel operations: W_N, R_O, and two occurrences of *inc*. Implementing the collector in PLDs gave enough connectivity to build a two-memory, multiple ALU system. This was a design decision that traded printed circuit area for speed. The VLSI implementation is more restricted with respect to silicon area and pins; hence, additional restrictions on architecture are necessary.

A serialization algebra was added to DDD in order to automate the imposition of such restrictions. In the DDD framework, transformations are applied to the iterative system, generating a new control specification according to the designer's intent. This process is highly interactive. An estimate of the physical characteristics of the design is based on feedback from the transformation system. If certain operations cannot be supported in parallel, the transformations serialize them.

Architectural modifications are reflected in control. In PAIR2 there are two memory operations, a write to *new* memory and a read from *old* memory. PAIR2 is expanded by the serialization program, as shown on the right, by specifying that one memory operation is allowed per state. An extra state, PAIR2.2, is introduced in which the second memory operation is performed. A similar serialization is applied to the *inc*, *add*, and *addinc* operations, although in PAIR2 the *inc* operations happen to be ordered as a byproduct of the



memory serialization.

The serialization transformation takes a list of the operations that cannot occur in parallel. The specification is searched for states which violate the condition. If found, permutations of the set of actions are searched until an ordering is found that allows the actions to be correctly executed in two states. In some instances, no ordering can be found. For example, a state containing the actions

$$D \leftarrow (\text{inc } H)$$
$$H \leftarrow (\text{add } D\ H)$$

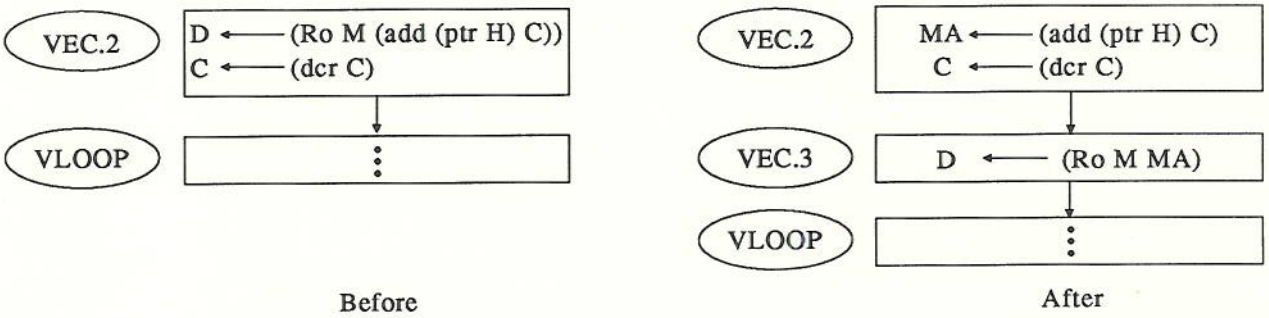
prohibits a simple expansion. In cases such as this, a register must be added or a dead variable analysis performed to find an available register of the correct type. The analysis is not currently automatic.

In addition to the limitation of one memory operation per state, the specification is transformed so that memory operations execute in two states instead of one. This simplifies timing of memory operations. In a term like

$$(R_x\ M\ \textit{Address})$$

Address may be a combinational expression, as it is in VEC.2 (below left), where the operand for R_o is *(add (ptr* H) C). From the designer's perspective this is a good abstraction for memory; compact and easy to understand. Building hardware to conform to this abstraction is difficult. Implementing memory access in this way imposes delays to allow for setup times. A conventional solution is to latch the memory address (MA) and data (MD). These values are computed in one clock cycle and used in the next. Serialization is used to expand memory operations. One difference is that the term itself is split.

Registers MA and MD, are added to the specification. During the first state of a memory operation, MA and MD are loaded with the corresponding argument from the original memory operation. In the second state, memory operations are modified to use MA and MD as parameters in place of the combinational signals. These transformations are automatic.



Before After

As a result of serialization the architecture is implementable in VLSI. The expanded GC is 37 states, three times longer than the original. From the designer's perspective, Appendix A is a better description of the collector because it is shorter and more abstract. From the implementation standpoint, the new description more concretely describes the architecture. Two kinds of serialization were used to transform the iterative specification into a suitable form. No attempt was made to generalize or characterize the serialization process. An *ad-hoc* approach was taken, the tools necessary to implement this design were developed within the framework of DDD. Although the transformations have proved useful in several other designs, there are cases where they are inadequate.

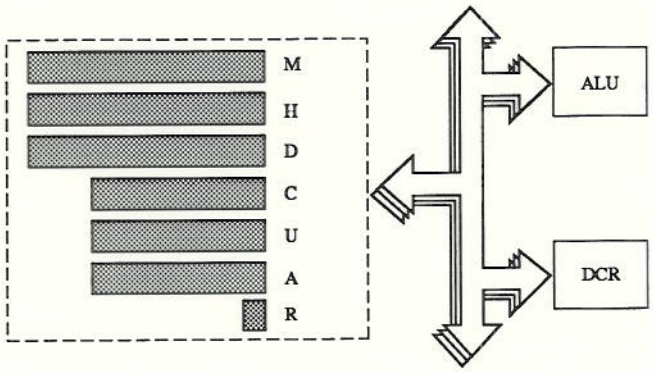
5. Synthesizing Control and Manipulating Architecture

The serializations in the previous section are presented as local transformations on flow diagrams, but they are implemented by symbolic transformations on applicative expressions. In essence, they are source-to-source program transformations. The next stage of derivation decomposes the algorithmic description into an abstract structural description governed by a synthesized control device. Details of control synthesis are given in [6] with formal justifications in [9]. Briefly, control is synthesized by building a finite state machine with one state for each defined function. The conditional expressions within the function definitions are abstracted to a *selection combination* for the design. Selection is distributed to each of the design's registers.

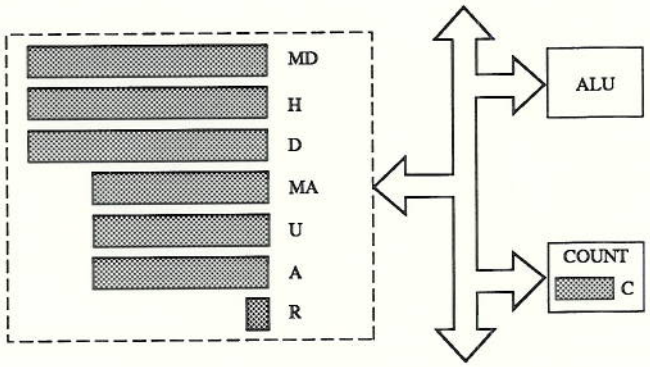
The byproduct of control synthesis is a structural description composed of register entities and functional components. This conceptual architecture is then interactively manipulated toward a physical organization. The automation provided by DDD sustains correctness through the sometimes pervasive transformations involved at this stage of engineering. Equally important, the system supports hierarchic decomposition associated with physical reorganization.

Though this particular design is probably subject to direct silicon compilation, its derivation illustrates the kinds of manipulations needed for the realization of abstract specifications. The DDD algebra 'scales' to higher levels of system description because there is no built-in expertise about representation techniques or targeting tactics. Such knowledge must be programmed in the DDD algebra.

The diagram below sketches the logical organization that results when control is extracted from the original GC specification. A number of register-like objects communicate with various combinational components. The key detail is the number of communication paths. Each path in the diagram stands for the selection combination for an individual register. The derivation reported in [6] directly implements the multiple paths, resulting in a highly parallel design.



The serialization transformations discussed in the previous section have the effect of reducing the number of physical paths needed. The degree of serialization is a matter of choice. In this instance, it is carried to the point of a conventional organization, centered about a single memory channel.



The description of the architecture is still abstract, still expressed in terms of the complex and symbolic entities of the original basis. Also, it remains executable as a simultaneous system of streams. While the description is still at a high level, it is useful to model its behavior in this way. Techniques for modeling with streams are discussed by O'Donnell [11] and Johnson [8].

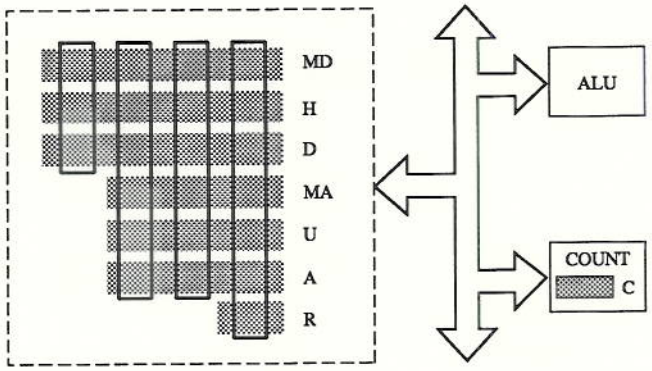
A collection of transformations called *system factorizations* refine the view of the organization by combining occurrences of similar terms into modules [6]. For example, the C register, which is only loaded and decremented, is encapsulated as a counter (compare the two figures above). Similarly, various arithmetic combinations are factored into an ALU component. Thus, the factorization commands entered in DDD comprise the structural description

of the design. Declaring the module decomposition at this stage of derivation follows the principle of deferring decisions on architecture until after control is developed [16, page 166]. The factorization algebra is sufficiently general to allow the engineer to experiment with the organization, assigning components to specific operations.

The derivation is now directed toward a physical realization, the goal being to reduce the description to a boolean system for logic synthesis. The first step is to define binary representations for the abstract registers and provide boolean implementations for the combinational functions. With this information, DDD expands the design description to a system of digital signal definitions.

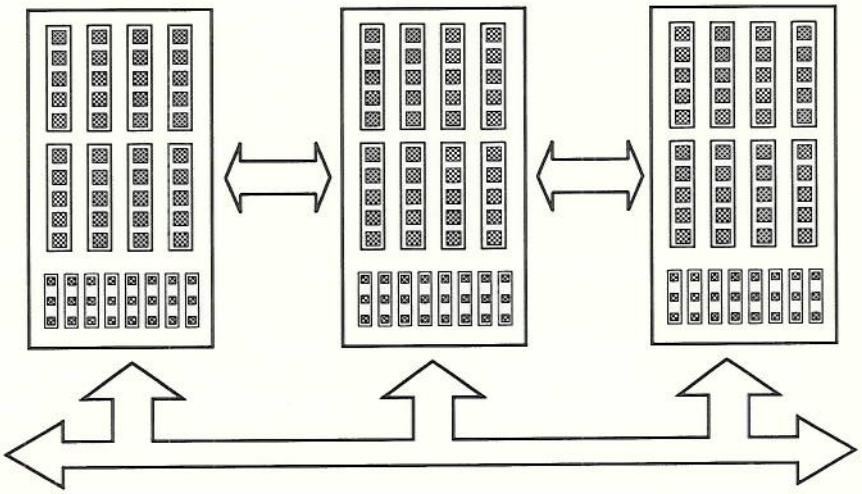
In general, the development of a physical realization entails a reorganization of the design into physical partitions. The partitioning occurs at several levels, and the resulting hierarchy can be quite distinct from that of the conceptual architecture. An MSI implementation might associate a device with each register. On the other hand, the high degree of connectivity in PLD and PLA targets can be exploited by a bit-slice partitioning. Busing the data path is also a method of bit slice reorganization.

The next diagram indicates the bit slice reorganization used for the garbage collector. The registers—as well as their selection combinations and certain arithmetic operations, not shown—are assembled in one-bit projections. It is a task of the derivation system to correctly sustain the connectivity of this new decomposition. The algebraic character of physical reorganization is a matter of current research and further experimentation. However, there is support in DDD along this particular implementation path.



The DDD system formats boolean system descriptions for available optimizers and device programmers. Except for logical simulation, discussed in the next section, facilities of the UCB/CSD VLSI tools (namely, *Espresso*, *Mpla*, *Mquilt*, and *Magic*) [14] were used for logic synthesis. In the earlier PLD wire-wrap prototype, each package contains one bit slice. In other PLD synthesis exercises, different partitionings have been used. For this VLSI prototype, the same boolean systems are retargeted to PLAs. Reorganization strategies for alternative technologies are now being pursued.

The VLSI realization packages sixteen bit slices on a small chip frame. A single instance of the chip contains a full tag field and eight bits of addressing. The next diagram gives an idea of the partitioning.



A collector for a 24-bit address space requires three identical chips, only one of whose tag fields is used, and an external PLD controller. The three-chip decomposition was done to save prototyping costs by using the smallest appropriate chip frame. It does not result in a viable realization. A distribution of arithmetic propagates carries across chip boundaries; hence, the speed of the circuit is below what is needed to drive a conventional memory at reasonable rates.

It would be difficult to characterize, much less automate, a partitioning task such as this. Finding a reasonable physical organization involved disparate design considerations. Some of these, such as reducing prototyping expenses, are decidedly mundane and bear little relationship to the design problem. The view reflected in DDD is that such considerations are explained by the derivation, that is, the sequence of transformations applied to the initial description. The goal is to provide a highly flexible algebra for tactical design management.

6. Simulation

The garbage collector's bit slices and other combinational functions were synthesized by a PLA generator. Routing between the PLAs was done by hand using the *Magic* drafting aids [14]. Since there was manual intervention in the design process, switch level simulation was employed to confirm the realization. The COSMOS simulator [1] was used for this purpose. Generation of good test vectors for simulation is crucial. By using the serialized specification to generate test vectors, we were able to develop a good test and, at the same time, an exact correspondence between source and target behavior.

Given definitions for the base operations, GC is a working Scheme program. With output statements added, the serialized version issues an execution trace; it is unnecessary to develop an interpreter to extract such a trace. The collector was executed against sample heaps, generated by a production list processing system which was modified to allow off-line garbage collection. A variety of programs were run until storage was exhausted, at

which time the heap and registers are written to a file. With appropriate base definitions installed, the serialized GC collected the heaps, which were then returned to the running system. In this way, it was established that the collector algorithm worked in a production setting.

With tracing enabled, test vectors were accumulated against actual heap images. A pair of vectors was issued at each function-call (clock cycle). The first contained inputs to the chip and the second contained expected outputs. COSMOS was configured to compare the simulation results with the expected values. These simulations exposed several errors in the manual routing. No errors were found in the derived circuitry. After the errors were corrected, the switch level simulation conformed exactly to the observed behavior of GC.

7. Conclusions

In this design exercise, the DDD system is used as a vehicle for exploring digital design in a functional calculus. Serialization is explored using source-to-source transformations on behavioral specifications to impose architectural restrictions on the design. Parallel arithmetic operations are serialized to conserve layout area and I/O pins. The dual-memory abstraction of the PLD prototype was transformed to a more standard model. Thus, the behavioral source description, GC, could be retargeted to significantly different logical architectures, exhibiting various degrees of parallelism. The method of serialization is *ad-hoc* and exploratory. The goal is to characterize these kinds of design tactics formally. Of course, the analysis for serialization is also of interest, but there is also a need for effective human interaction. The character of the interaction is a central topic in this experimentation.

One of the advantages of the DDD approach is the ability to manage both logical and physical hierarchies throughout the design process. The ability to manipulate the physical composition was essential for implementing the design on a three-chip set. Though this particular implementation is not realistic—the design could probably be compiled to a single chip from its derived boolean description using modern logic synthesis tools—the exercise reveals the types of manipulations needed for implementing high level systems. The DDD algebra 'scales' to higher levels of system description because there is no built-in expertise about representation techniques or targeting tactics. Such knowledge must be programmed in the DDD algebra.

The executable nature of specifications to the DDD system allowed the design to be thoroughly tested. GC was run in a production heap processing environment. Executability of intermediate forms of the transformations allowed us to establish a high level testing environment for the layout. The same test data used to establish the correctness of GC was used to generate test vectors for a switch level simulation. An exact correspondence between the source and target descriptions was maintained.

Acknowledgments

Robert Wehrmeister provided essential support in the integration of design and fabrication tools, and helped establish a test environment for software testing of the garbage collector. Bhaskar Bose is the main implementer of the DDD system [2] and also assisted with the derivation process.

References

[1] Beatty, Derek, Brace, Karl, Bryant, Randal E., Cho, Kyeongsoon and Huang, Lawrence, User's Guide to COSMOS a COmpiled Simulator for MOS Circuits, Computer Science Department, Carnegie Melon University (November, 1987)

[2] Bose, Bhaskar, DDD - A Transformation System for Deriving Digital Design, Indiana University Department of Computer Science Technical Report (in progress)

[3] Cheney, C. J., A Nonrecursive List Compacting Algorithm, *Comm. ACM* 13, 11 (November 1970) pp. 677-678.

[4] Fenichel, Robert R., and Jerome C. Yochelson, A LISP Garbage-Collector for Virtual-Memory Computer Systems, *Comm. ACM* 12, 11 (Nov. 1969) 611-612.

[5] Gajski, Daniel D., (ed.) *Silicon Compilation* (Addison-Wesley, Reading, 1988)

[6] Johnson, Steven D., Bose, Bhaskar and Boyer, C. David, A Tactical Framework for Digital Design, in: Birtwistle, G. and Subrahmanyam, P.A., (eds.), *VLSI Specification, Verification and Synthesis* (Kluwer Academic Publishers, Boston, 1988) pp. 349-383.

[7] Johnson, Steven D., Digital Design in a Functional Calculus, in: Milne, G. and Subrahmanyam, P.A. (eds.) *Formal Aspects of VLSI Design* (North-Holland, Amsterdam, 1986) pp. 153-178.

[8] Johnson, Steven D., Applicative Programming and Digital Design, *Proc. Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1984), pp. 218-227.

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

[10] Keutzer, Kurt, and Wayne Wolf, Anatomy of a Hardware Compiler, *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988) pp. 95-104.

[11] O'Donnell, John T. *Hardware description with recursion equations*, in: Barbacci, M.R. and Koomen. C.J., (eds.), *Computer Hardware Description Languages and their Applications* (North-Holland, Amsterdam, 1987) pp. 363-382.

[12] Peyton-Jones, Simon L., *The Implementation of Functional Programming Languages* (Prentice Hall, Englewood Cliffs, 1987)

[13] Rees, Jonathan and Clinger, William, The Revised[3] Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices* 21(12), (December 1986)

[14] Scott, Walter S., Mayo, Robert N., Hamachi, Gordon and Ousterhout, John K., (eds.), 1986 VLSI Tools, Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California at Berkeley, (1985)

[15] Sheeran, Mary, Retiming and Slowdown in Ruby, in G.J. Milne(ed.) *The Fusion of Hardware Design and Verification* (North-Holland, Amsterdam, 1988) pp. 289-308.

[16] Prosser, Franklin P. and Winkel, David, *The Art of Digital Design,* second ed. (Prentice-Hall, Englewood Cliffs, 1987)

Appendix A. Garbage Collector Specification, GC

```
(letrec
 (Idle (λ (M H D C U A R GO)
  (if GO (Next M *H* D C 0 0 false GO)
         (Idle  M H D C U A true GO))))

 (Driver (λ (M H D C U A R GO)
    (if (= U A) (Show-avl (M_FLIP M) H D C U A true GO)
                (Next M (R_N M U) D C U A R GO))))

 (Show-avl (λ (M H D C U A R GO)
  (if GO (Show-avl M H D C U A R GO)
         (Idle M H D C U A R GO))))

 (Next (λ (M H D C U A R GO)
  (if (pointer? H)
     (Type M H (R_O M H) C U A R GO)
     (if (bvec-head? H)
        (Driver M H D C (addinc U (btow (ptr H))) A R GO)
        (Driver M H D C (inc U) A R GO)))))

 (Type (λ (M H D C U A R GO)
  (if (eq? fwd (tag D))
     (Driver (W_N M U (cell H D)) H D C (inc U) A R GO)
     (case (tag H)
       (pair   (Pair1 (W_O M H (cell fwd A)) H D C U A R GO))
       (vec    (Vec (W_N M U (cell H A)) H D (ptr D) (inc U) A R GO))
       (bvec   (Bvec (W_N M A D) H (cell D (btow (ptr D))) (btow (ptr D)) U A R GO))
       (fbvec  (Driver M H D C (inc U) A R GO)))))))

 (Pair1 (λ (M H D C U A R GO)
  (Pair2 (W_N M A D) H D C U A R GO)))

 (Pair2 (λ (M H D C U A R GO)
  (Pair3 (W_N M U (cell H A)) H (R_O M (inc (ptr H))) C U (inc A) R GO)))

 (Pair3 (λ (M H D C U A R GO)
  (Driver (W_N M A D) H D C (inc U) (inc A) R GO)))

 (Vec (λ (M H D C U A R GO)
  (Vloop (W_N M A D) H (R_O M (add (ptr H) C)) (dcr C) U A R GO)))

 (Vloop (λ (M H D C U A R GO)
  (if (= C -1)
     (Driver (W_O M H (cell fwd A)) H D C U (addinc A (ptr D)) R GO)
     (Vloop (W_N M (addinc C A) D) H (R_O M (add (ptr H) C)) (dcr C) U A R GO))))

 (Bvec (λ (M H D C U A R GO)
  (Bloop (W_N M U (cell H A)) H (R_O M (add (ptr H) (ptr D))) (dcr C) (inc U) A R GO)))
```