

SoC Methods and Architecture for Realizing Fast Cryptographic Computing Engines

James P. Davis, *Member, IEEE*, Duncan A. Buell, *Senior Member, IEEE*, and Sreesa Akella, *Student Member, IEEE*

Abstract— We present an SoC-based design method and architecture pattern for implementing elliptic curve numerical algorithms directly into SOC “engines”, thus eliminating much overhead associated with algorithm processing in software, and allowing for on-chip parallelization. The solution formulations use a direct mapping to application-specific, SoC-based custom computing machines designed specifically for cryptographic data processing tasks. Such VLSI computing “engines” are realized as specialized systems that can either be integrated with standard computing platforms (e.g., with PCs as add-on cards) or used to construct specialized computing environments for numerical integer data processing for applications such as cryptography.

Index Terms— algorithmic state machines, cryptography, design-for-synthesis, elliptic curves, finite field arithmetic.

I. INTRODUCTION

Many of the major computational problems in modern data communications involve the issue of how to provide information over a channel securely. For a number of years, the National Institutes of Standards and Technology (NIST) has promulgated standards for cryptographic protocols, some of which involve public key algorithms [1], [2]. At the heart of public key computations such as these are basic arithmetic operations, either for multi-precise integers or for elements in finite fields of characteristic 2. What we will describe is a design methodology for creating implementations of numerical algorithms in SOC “engines” that can greatly speed up the basic arithmetic without sacrificing flexibility.

II. ARITHMETIC FOR PUBLIC KEY CRYPTOGRAPHY

A. The Canonical Computation

For pedagogical purposes we will use a canonical representation of an elliptic curve proposed by NIST for cryptographic applications. For a *prime field* $GF(p)$ defined modulo a large prime p , such a curve can be written

$$Y^2 = X^3 + AX + B \quad (1)$$

for constants A and B . For a *binary field* $GF(2^n)$ defined by an irreducible polynomial of degree n with coefficients in $GF(2)$, the curve must be written in its more general form

$$Y^2 + XY = X^3 + AX + B \quad (2)$$

Given two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on a curve defined over a prime field, we will need to compute the sum $P_3 = P_1 + P_2$ on the curve, which can be done in the following way. To avoid the need for a modular division, we first change the curve to homogeneous coordinates, of the form below.

$$Y^2Z = X^3 + AXZ^2 + BZ^3 \quad (3)$$

Now, given points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, we compute the sum $P_1 + P_2 = P_3 = (x_3, y_3, z_3)$ as follows.

$$u = y_1z_2 - y_2z_1 \quad (4)$$

$$v = x_1z_2 - x_2z_1 \quad (5)$$

$$t = y_1z_2 + y_2z_1 \quad (6)$$

$$w = x_1z_2 + x_2z_1 \quad (7)$$

$$x_3 = 2v(z_1z_2u^2 - wv^2) \quad (8)$$

$$y_3 = tv^3 + u(2z_1z_2u^2 - 3v^2w) \quad (9)$$

$$z_3 = 2z_1z_2v^3 \quad (10)$$

The formulae are slightly simpler if one wishes to add a point to itself on the curve (called *doubling* the point). The formulae for this part of the analysis can be found in [3], [4]. In the case of the binary fields, the algebraic formulations are different we are dealing with number *mod 2* such that $2 = 0$; however, the process is much the same.

Now, in point of fact, all these operations are carried out in the finite field over which the problem is defined. In the case of curves defined over a prime field, the underlying operations are multi-precise arithmetic modulo the prime p . In the case of curves in a binary field, the operations are arithmetic modulo the primitive polynomial $f(x)$ chosen to define the field.

For purposes of this paper, we formulate the computation of

Manuscript received April 12, 2002.

J. P. Davis, D. A. Buell and S. Akella are with the Department of Computer Science and Engineering, Swearingen Engineering Center, University of South Carolina, Columbia, SC 29208 USA. (J. P. Davis phone: 803-413-3484; fax: 803-777-3767; e-mail: jimdavis@cse.sc.edu).

P_3 by defining the following set of time steps over which the operations can be calculated. Note that this analysis is done solely from the standpoint of the software implementation of the algorithm in C. The actual transformation of this algorithm into a register-level architecture affords some flexibility to sequence and schedule the operations according to a number of criteria—including clocking style, use of registers versus latches in the data path, use of a state machine to gate the signal paths through MUX-based logic, and organization of logic blocks according to the device type for realizing the cryptographic system-on-a-chip.

1. Time Step 1:	
(1)	$= x_1 z_2$
(2)	$= x_2 z_1$
(3)	$= y_1 z_2$
(4)	$= y_2 z_1$
(5)	$= z_1 z_2$
2. Time Step 2:	
(6)	$= (1) + (2) = x_1 z_2 + x_2 z_1$
(7)	$= (1) - (2) = x_1 z_2 - x_2 z_1$
(8)	$= (3) + (4) = y_1 z_2 + y_2 z_1$
(9)	$= (3) - (4) = y_1 z_2 - y_2 z_1$
3. Time Step 3:	
(10)	$= u^2$
(11)	$= v^2$
4. Time Step 4:	
(12)	$= (5) * (10) = z_1 z_2 u^2$
(13)	$= v * (11) = v^3$
(14)	$= (6) + (11) = v^2(x_1 z_2 + x_2 z_1)$
5. Time Step 5:	
(15)	$= (12) - (14) = z_1 z_2 u^2 - v^2(x_1 z_2 + x_2 z_1)$
(16)	$= (8) + (13) = v^3(y_1 z_2 + y_2 z_1)$
(17)	$= (5) * (13) = z_1 z_2 v^3$
6. Time Step 6:	
(18)	$= (16) - (12) = v^3(y_1 z_2 + y_2 z_1) - z_1 z_2 u^2$
(19)	$= (15) + (15) = 2(z_1 z_2 u^2 - v^2(x_1 z_2 + x_2 z_1))$
(20)	$= (17) + (17) = 2z_1 z_2 v^3 = z_3$
7. Time Step 7:	
(21)	$= (19) - (14) = 2z_1 z_2 u^2 - 3v^2(x_1 z_2 + x_2 z_1)$
(22)	$= v * (19) = 2v(z_1 z_2 u^2 - v^2(x_1 z_2 + x_2 z_1)) = x_3$
8. Time Step 8:	
(23)	$= u * (21) = u(2z_1 z_2 u^2 - 3v^2(x_1 z_2 + x_2 z_1))$
9. Time Step 9:	
(24)	$= (23) - (16) = y_3$

B. Characteristics of Computation

The reason that this arithmetic becomes computationally intensive should be obvious. Both prime field and binary field

arithmetic are CPU-intensive in low level kernels for which standard processor architectures have not been designed. Our goal in designing special SoC architectures for these computations is to add both the power of specialized application-specific processing to the task as well as the inherent parallelism that custom array-based logic provides.

Taking a layered approach to hardware design, one can introduce special processor design on three different levels [3]. If the computational bottleneck is, say, the multi-precise multiply operation, then one can first design specific ALUs for the prime field or binary field arithmetic. At this point the bottleneck will become data movement--the arithmetic operations will be fast, but it will not be possible to provide the ALUs with sufficient data to keep them occupied. Expanding the scope of the implementation, an entire elliptic curve addition could be designed into a processing unit. There are several different and competing formulations of the elliptic curve and its arithmetic; in most, however, it is possible to make use of three parallel multiply units to speed up the addition of two points.

It may still happen that the implementation of elliptic-curve addition has a data bottleneck. In this case, one must look more generally at the use of the curve in the Cryptographic protocol. In most instances, it is not a single addition of points that is to be computed. Rather, one is computing the scalar multiple $M \cdot P$ of some point P for some large integer M . This operation requires on average $\lg M$ doublings of P and $1/2 \lg M$ additions. If the implementation of a single curve addition is insufficiently compute-intensive to justify an SoC architecture and implementation, it will almost certainly be the case that the implementation of an entire scalar multiplication will be sufficiently compute-intensive [9].

C. Realizing Algorithms in SOC Architectures

As shown in Fig. 1, we can consider the decision of how to realize an algorithm as being across a continuum of potential architectures. At one end, using standard microprocessor-based architectures (such as those that comprise PCs and Unix-based workstations and networks), we have technology that is designed for a high degree of programmability (evidenced by the programming languages and environments supported), generality and backwards compatibility.

Most algorithmic work in elliptic curve computation is done with architectures on the left side of Fig. 1. The principal task is thus to optimize the execution of the algorithm, written in a software language such as C, on the selected general-purpose architecture. Attempts are also made to parallelize execution by spanning the execution across multiple, networked systems of similar class [3]. Although there are many variations on this theme, the execution model underlying all of these architectures is based on the Von Neumann architecture and its variants, which has been in use for 50 years [5]. Furthermore, most classes of these architectures execute a layer of “virtual machines” consisting of various software components (operating system, drivers, application programs, etc.) designed to make the architecture of general use to a broad

class of algorithms and application programs.

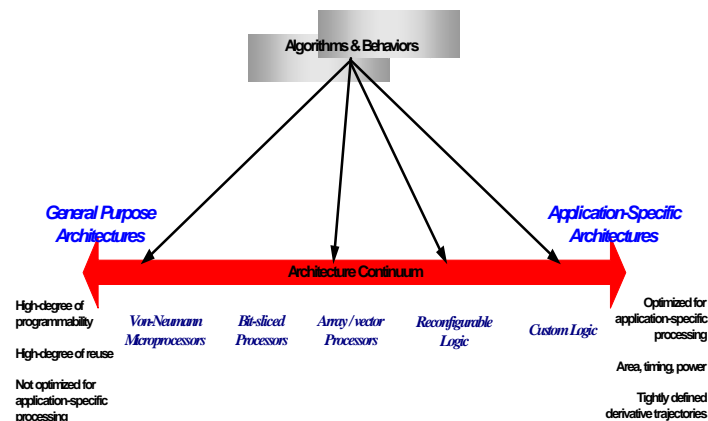


Fig. 1. The continuum of possible architectures for realizing computational algorithms consists of conventional computers based on the layered “virtual machine” convention, whereas we are examining the exploitation of application-specific custom logic to provide configurable blocks that can be used to construct systems-on-a-chip compute “engines” for specialized problems in cryptography.

At the other end of the continuum are architectures based on the principal of designing application-specific VLSI circuits to implement custom logic functions and algorithms. Solutions of this type tend to be highly specialized to the algorithms used in the application, and allow specialized packaging. What these architecture options on the right side of Fig. 1 sacrifice in terms of generality, they trade off in speed and efficiency for the computational problem at hand.

Thus, the basis for our work is the recognized efficiencies to be gained by realizing the specialized functionality of a class of elliptic curve algorithms, using the understood benefits in selecting SoC methods and architectures. Many of these architectural benefits of custom logic solutions have been discussed elsewhere [6], [7], [8], [9].

Therefore, given this premise, the problem treated in this paper becomes one of coming up with an effective and reasonably accessible means to move from specifying algorithms for elliptic curve analysis such that they can be quickly realized in systems-on-a-chip using engineering practices common to this endeavor. To do this, we present a methodology for transitioning from algorithm to custom logic architecture that can be built using SoC VLSI circuits.

III. METHODOLOGY EXPLAINED

A. Moving from Algorithm to Register-level

For creating algorithm specifications that can easily be realized in custom digital VLSI logic, we start with a representation of the algorithm. It is common in the scientific computing domains to jump directly into writing algorithms in a language such as C, once the algorithm has been appropriately mathematically formulated. In our approach, we take the algorithm description and create a graphical flowchart-like representation, based on the Algorithmic State

Machine (ASM) chart [10].

Next, we add to this flowchart-like set of structures a set of “bindings” to each of the algorithm “steps”. The steps are organized and ordered according to when they might be expected to execute, based on the specification of appropriate clocking. The basic principle of organization conforms to the definition of a finite state machine (FSM), as represented in a state diagram. The structure of the state diagram is captured as a refinement of the flowchart, turning it into a *flowdiagram* [11]. Through this incremental refinement, we capture the control flow of the algorithm, and can make partitioning decisions as to which parts of the algorithm might be performed concurrently in the hardware realization.

The next step is to add the actual data manipulation statements associated with the algorithm. These are individually specified as register-transfer (RTL) statements involving standard arithmetic, Boolean and steering logic functions that are well-understood abstractions in digital logic and computer architecture design [13].

B. Algorithm Representation

In considering algorithm design for on-chip systems, we make several assumptions about the “typical” organization of such systems. Researchers and practitioners have developed these assumptions over time, and they form the basis for designers to understand VLSI-based systems. First, system components can be partitioned and defined in terms of their control and data behaviors [12]. For on-chip systems, we can think of an arbitrary system block implementing an algorithm in terms of its control path and data path (in which we also include memory arrays). The partitioning of on-chip system components in our approach is illustrated in Fig. 2.

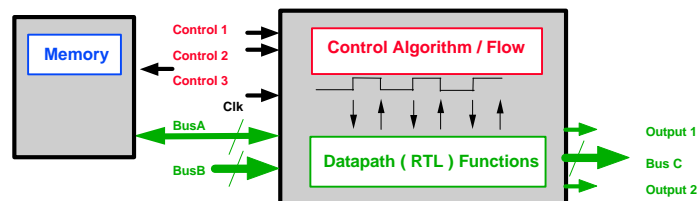


Fig. 2. Partitioning an algorithm block into control path, data path and memory. This most basic formulation allows the constituent aspects of the algorithm to be specified using appropriate abstractions, captured within a single unified model.

C. Notational Constructs of a Flowdiagram

Once we create a hierarchical structure of functionally partitioned blocks, we further refine each of the bottom-level blocks with a register-level notation. We use a graphical *flowdiagram* to create a partially ordered temporal sequence of data operations. Initially, this sequence represents the precedence of data operations in each system block of the algorithm. The designer would refine and extend the flowdiagram with explicit clocking information later in the specification process. The flowdiagram is an enhancement of the ASM notation [10], but has been augmented to include

explicit references to data operations, sequenced and scheduled according to control-step behavior in the design.

A flowdiagram has a graphical symbol set similar to the traditional flowchart. It is capable of representing the basic constructs of control flow common to algorithmic design: *sequence*, *selection*, and *iteration*. One extension to the ASM chart is the addition of a notation for describing data operations. This notation is based on the concept of register-transfer assignment, as defined in [6], [13]. In a set of concurrent design threads, flowdiagrams may model interactions of potentially many control and data path units.

An algorithm designer constructs a flowdiagram model to represent the control flow for each function block in the design. Multiple flowdiagram "threads" can be created to model concurrent behavior for a given block. The threads interact with one another using standard mechanisms for concurrent systems, such as synchronization and cooperation on shared tasks (using polling, handshaking and interrupts), and competition for shared resources (using arbitration). The primitives are used to build up higher-level architecture "patterns", used in the construction and evaluation of architectural solutions to numerical processing algorithms.

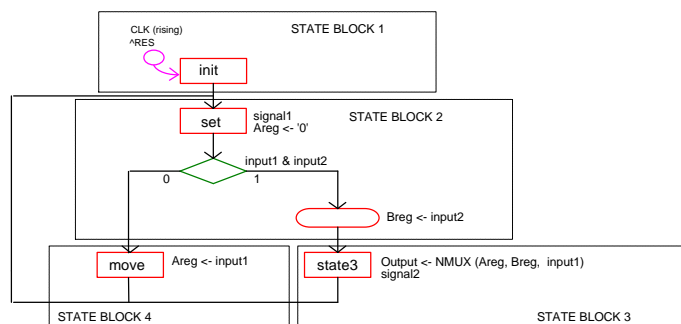


Fig. 3. The state block structure of a flowdiagram is shown. The architecting activity consists of defining the set of operations that are to be executed during each state (based on the time step of a defined clock). Most architecture representations tend to be data path centric; however, we have found that a control centric representation allows a more direct mapping from algorithm to architecture.

IV. THE METHODOLOGY APPLIED

We can discuss other aspects of the notation by looking at the specific elliptic curve formulation as presented earlier and its realization as a state machine with register-transfer data operations in a flowdiagram. We look at operation sequencing, operation scheduling and resource allocation—essential steps in transforming an abstract algorithm into a form that can be directly realized in custom logic [6], [8].

A. Specifying Operation Sequencing

An algorithm designer specifies data operations in the flowdiagram notation using RTL-level expressions. An expression can be an assignment of some value or signal to another signal. In addition, expressions can have explicit data operations, represented as functional transformations, specified as part of their right-hand side. Expressions,

represented as RTL notation, are attached to specific states in the flowdiagram, indicating that these operations are sequenced according to their attached state, and scheduled once execution reaches that state.

B. Specifying FSM Operation Scheduling

Scheduling information is annotated to the evolving flowdiagram representation using two mechanisms. First, once the algorithms designer expresses individual data operations in terms of defined *bus* structures, these buses can be "bound" to specific resource types. The designer will take each bus and declare it as being a *register*, *latch*, *wire* or other bus structure.

Second, the designer selects specific clocking regimes, based on requirements for moving data through the data path. Referring to our earlier notion of on-chip systems architecture, we have both a control path unit and data path unit (including memory arrays). In synchronous sequential designs, both units are sensitive to a clock event. A designer can define this clock event using a duly defined set of clocking signals, or another *aperiodically* triggered signal generated somewhere else in the design may be defined as the clock.

Once a designer annotates the flowdiagram with clocking information, the operations attached to individual states are scheduled. The behavior of each data operation, and the availability of each result on the output of the data path unit, now depends on two things: (1) the relationship between the selected clocking schemes of the control and data paths, and (2) the selected bus structures of the data paths involved in each operation (e.g., registered versus non-registered).

C. Specifying RTL Resource Allocation

As in many engineering design disciplines, VLSI design has its basic set of high-level primitive building blocks, from which larger units can be built. At each level of VLSI design abstraction, there is a well-defined set of such primitives. For example, designers use individual flip flops and logic gates at the gate level, and bus-oriented logical and arithmetic functions at the register-transfer level. The flowdiagram notation is built around an implicit understanding of these primitives.

The basic register unit is a *bus*, which is an abstract signal path through the design. Buses are read from--and written to--in expressions of data operations in the flowdiagram. This corresponds to a variable appearing on the left and right hand sides of assignment statements in conventional programming languages such as C.

A flowdiagram incorporates the notion of *macro-functions* directly into the notation, in that the flowHDL® tool environment [15] supporting flowdiagram creation includes a library of macro-function primitives. Such macros are "functions" in a mathematical sense, taking a set of input arguments, applying a transformation operation on the inputs and returning a result, without side effects. Macros are used in assignment expressions to specify the behavior of individual data path units, without specifying their specific bindings. This is left up to the downstream synthesis tools.

D. Specifying Lifetime of Data Values

One important aspect of algorithm realization in digital logic is determining the lifetime of specific data values [6]. In other words, we wish to know how long a specific data value will be valid on the signal path. From a circuit perspective, this lifetime indicates the number of clock cycles the signal is to be driven to its assigned value before the value is scheduled to change.

Using the flowdiagram notation, we assign data values to signals/buses, in the same manner as making assignments in a conventional programming language or HDL. The data value is held on the bus, across some number of explicit control cycles, until the bus is driven to a different value through a subsequent assignment. We can alternatively “assert” a signal, indicating that the signal is to go to its “high” value for the duration of the scheduled control cycle, at which time the signal returns to its default value for subsequent cycles, or for any cycles where not explicitly driven.

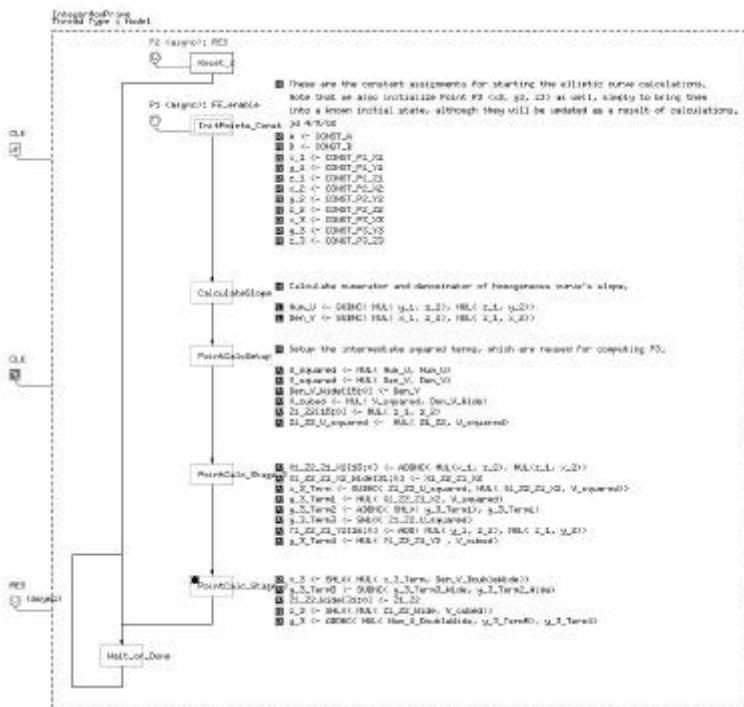


Fig. 4. The flowdiagram representation of the elliptic curve equations is shown, a state machine providing the sequencing and scheduling of operations according to state assignment. Many of the operations are allocated to wires, thus implying that, instead of registers holding intermediate results according to the algorithm’s time steps shown earlier, we chose to gate some of the intermediate operations using the state registers.

V. CONCLUSION

In this paper, we have outlined an approach to creating architectures for numerical algorithms using an SoC design style based on an ASM-based register-level design notation and methodology. The topology of the architecture can be inferred by the designer, based on a set of heuristics (not described in this paper). Such architectures can be evaluated using the well-understood design-for-synthesis style [11].

Although this methodology has been used in industry for

some time, we now use it to address scientific computation, such as those associated with numerical integer factoring, \ having direct application in cryptography. We are exploring a range of SoC architectures that are an appropriate match to the algorithms requiring such novel computation approaches. It has been pointed out elsewhere [3], [9] the importance of speeding up the processing of these algorithms by orders of magnitude over what is achievable using conventional computing architectures. We wish to do this without having to use HDLs as the specification medium for the architectures.

Thus, this SoC-based methodology and tools, including how the tools automatically generate synthesizable VHDL and Verilog, having been discussed in more detail in [11], [14], and the architectures that it is able to create, are a starting point for exploring the implementation of numerical factorization algorithms—of which the elliptic curve approach is but one example.

This research is being carried in a number of directions: (1) to create a set of high-level architecture patterns that can be matched to appropriate numerical algorithms, (2) scaling of the basic architecture patterns through component instantiation and multi-dimensional parallelism through reconfigurable computing architectures [9], and the extension of the methods and notation for more intuitive use by non-engineers.

REFERENCES

- [1] D. Johnson and A. Menezes, “The elliptic curve digital signature algorithm,” *Technical Report CORR 99-34*, University of Waterloo, Center for Applied Cryptographic Research, 1999.
- [2] National Institute for Standards and Technology, “Recommended elliptic curves for federal government use,” *Technical Report*, NIST, www.csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.ps, 1999.
- [3] D. A. Buell, “Factoring: algorithms, computers, and computations,” *The Journal of Supercomputing*, 1:191-216, 1987.
- [4] A. Menezes, *Elliptic Curve Public Key Cryptosystems*, Kluwer Academic Publishers, 1993.
- [5] A. S. Tanenbaum, *Structured Computer Organization*, 4th ed., Prentice-Hall Publishers, Inc., 1999.
- [6] R. Camposano, and W. Wolf (eds.), *High-level VLSI Synthesis*, Kluwer Academic Publishers, 1991.
- [7] D. Thomas, J. Adams, and H. Schmit, “A Model and Methodology for Hardware-Software Codesign”, *IEEE Design & Test of Computers*, Vol. 10, No. 3, September 1993, pp. 6-15.
- [8] D. Gajski, and L. Ramachandran, “Introduction to High-level Synthesis”, *IEEE Design & Test of Computers*, Vol. 11, No. 4, Winter 1994, pp. 44-54.
- [9] D. A. Buell, J. M. Arnold, W. J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press, Los Alamitos, Calif., 1996.
- [10] C. R. Clare, *Designing Logic Systems Using State Machines*, McGraw-Hill Publishing Co., 1973.
- [11] J. P. Davis, “High-level Design-for-Synthesis: the Next Step”, in *EDA&T 95: Conference on Electronic Design Automation and Test*, Asian Sources Media Group, 1995, pp. 378-391.
- [12] C. Mead, and L. Conway, *VLSI Systems Design*, Addison Wesley Publishing Co., 1978.
- [13] J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill Publishing Company, Inc., 1979.
- [14] J. P. Davis, S. Nagarkar and J. K. Matthewes, “High-level Design of On-chip Systems for Integrated Control and Datapath Applications”, in *Design Supercon-96: On-Chip System Design Conference*, Hewlett Packard Company, Inc., 1996.
- [15] *FlowHDL Users Manual*, © 1998, KBS Corporation.