LOGIC DESIGN

The purpose of a design process is to develop a hardware system that realizes certain user-defined functionalities. A hardware system is one constructed from electronic components. Signals enter and leave the system. They are either analog or digital. Information carried by an analog signal is continuous, whereas information carried by a digital (binary) signal is discrete, represented as 1 and 0. Input signals are processed by the hardware system which produces the output signals. Signals are also generated internally, and can be either digital or analog. Digital subsystems can be combinational or sequential. There are two types of digital sequential systems; synchronous systems and asynchronous systems. A synchronous system is one whose elements change their values only at certain specified times determined clock changes. Inputs, states, and outputs of an asynchronous system can change at any time.

A design process develops a hardware system capable of performing some predefined functionality. The functionality of a digital hardware system can be realized by two processes: using logic circuits that are implemented with logic gates and/or using software to drive the system. The former process is referred to as hardware implementation, and the latter as software implementation. The use of the software is related to the use of microprocessors. If microprocessors are used in a design, the design is referred to as a microprocessor-based design.

In a microprocessor-based design, the functionalities are implemented partially by hardware and partially by software.

The designer needs to divide the tasks between hardware and software, which is sometimes referred to as software-hardware co-design. The hardware portion includes the microprocessor, memory, peripheral integrated circuits (IC), and glue logic (glue logic are circuits that "glue" digital components together). The software is a set of computer programs stored in the memory.

Logic circuits for digital hardware systems may be combinational or sequential. A combinational circuit is one whose output values depend only on its present input values. A sequential circuit is one whose outputs depend not only on its present inputs but also on its current (internal) state. In other words, the present outputs are functions of present and previous input values. The input history is remembered by memory elements (e.g., registers). A register is a set of flip-flops.

The control unit of a digital system is normally implemented by a sequential circuit. The data path can be implemented by either a sequential circuit or a combinational circuit (and usually, also some registers). The data path may consist of logic, arithmetic, and other combinational operators and registers, as well as counters, memories, small state machines, interface machines, and other sequential blocks. Logic design research develops procedures for efficient design of digital circuits. Various technologies and related design methodologies as well as computer tools are used to transform high level system characterizations to working devices.

Complex modern systems include subsystems that require both digital hardware design and microprocessor-based design. To design such systems, the designers need to be familiar with both the co-design methodologies and co-design tools.

MATHEMATICAL CHARACTERIZATION

The mathematical characterization is concerned with mathematical specification of the problem as some kind of transformation, equation solving, and so on. In the case of digital circuit/system applications, the mathematical characterizations include, for example, the following models: regular expressions, extended regular expressions, data flow graphs, Petri nets, finite state machines, Boolean functions, timed Boolean functions, and physical design models. The physical design models can only be realized in hardware. All of the other models mentioned above can be realized either in hardware or in software or in both.

The goal of mathematical characterizations is to provide the ability to investigate formally the problems of equivalence, optimization, correctness, and formal design correct from specification, by transformational methods.

Nowadays, most of the design is done automatically by electronic design automation (EDA) tools. The logic and system designers not only use the EDA tools, but also often design their own tools or adapt and personalize the existing tools. That is why the problems of logic representation and mathematical characterization are unseparable from the logic design, and will be devoted here due attention.

High-Level Behavioral Specifications

Regular expressions are an example of high-level behavioral specification of a sequential circuit. They describe the input sequences accepted by a machine, output sequences generated by a machine, or input-output sequences of a machine. Regular expressions are used in digital design to simplify description and improve optimization of such circuits as sequence generators or language acceptors. They use some finite alphabet of symbols (letters) and the set of operations. The operations are concatenation, union, and iteration. Concatenation $E_1 \cdot E_2$ means subsequent occurrence of events E_1 and E_2. Union $E_1 \cup E_2$ means logical-OR of the two events. Iteration E^* of event E means repetition of the event E an arbitrary finite number of times or no occurrence of this event. The simplest event is an occurrence of a single symbol. Extended regular expressions generalize Regular Expressions by adding the remaining Boolean operations. All the Boolean operators can be used in an extended regular expression. For instance, negation or Boolean product.

Petri nets are concurrent descriptions of sequential processes. They are usually converted to finite state machines or directly converted to sequential netlists. Because they are also used in concurrent system specification, verification, and software design, Petri nets are increasingly used in software-hardware codesign and to specify hardware (25).

Finite State Machines

Finite state machines (FSMs) are usually of Mealy or Moore types. Both Moore and Mealy machines have the following: the set of input symbols, the set of internal states (symbols), and the set of output symbols. They also have two functions: the transition function δ and the output function λ. The transition function δ specifies the next internal state as a function of the present internal state and the present input state. The output function λ describes the present output state. Moore machines have output states which are functions of only the present internal states. Mealy machines have output states which are functions of both present internal states and present input states. Thus state machines can be described and realized as composition of purely combinational blocks δ and λ with registers that hold their states.

Parallel state machines are less commonly used compared to Moore machines and Mealy machines. In a parallel state machine several states can be successors of the same internal state and input state. In other words, the parallel state machine is concurrently in several of its internal states. This is similar in principle to concurrently having many tokens in places of the Petri net graph description.

Nondeterministic state machines are another model. In a nondeterministic state machine, there are several transitions to next internal states from the same present input state and the same present internal state. From this aspect, nondeterministic state machines are syntactically similar to parallel state machines. However, the interpretation between these two machines is different. In a nondeterministic state machine, the several transitions to a next internal state is interpreted that any of these transitions is possible, but only one is actually selected for next stages of design. The selection may occur at the state minimization, the state assignment, the state machine decomposition, or the circuit realization of the excitation and output logic. The transition is selected in order to simplify the circuit at the next design stage, or to improve certain property of the circuit. The above selection is done either automatically by the EDA tools, or manually by a human. Nondeterminism expands the design space, and thus gives the designer more freedom to improve the design. However, this can also lead to a more complex or a longer design process.

There are several other generalizations of FSMs, such as Buechi or Glushkov machines, which in general assume more relaxed definitions of machine compatibility. For instance, machines can be defined as compatible even if their output sequences are different for the same starting internal states and the same input sequences given to them, but the global input–output relations of their behaviors are equivalent in some sense. All these machines can be described in tabular, graphical, functional, HDL language, or netlist forms, and realized in many listed below technologies.

Boolean Functions Characterizations

Boolean functions are characterized usually as truth tables, arrays of cubes, and decision diagrams. Representations can be canonical or noncanonical. Canonical means that the representation of a function is unique. If the order of the input variables is specified, then both truth tables and binary decision diagrams are canonical representations. Cube representations are not canonical, but can be made canonical under certain assumptions (for instance, all prime implicants of a completely specified function). In a canonical representation the comparison of two functions is simple. This is one of the advantages of canonical representations. This advantage has found applications in the verification and synthesis algorithms.

Good understanding of cube calculus and decision diagrams is necessary to create and program efficient algorithms for logic design, test generation and formal verification.

Truth Tables and Karnaugh Maps

A truth table for a logic function is a list of input combinations and their corresponding output values. Truth tables are suitable to present functions with small numbers of inputs (for instance, single cells of iterative circuits). Truth tables can be easily specified in hardware description languages such as VHSIC hardware description language (VHDL).

Table 1 shows the truth table of a full adder. A full adder is a logic circuit with two data inputs A and B, a carry-in input C_{in}, and two outputs Sum and carry-out C_{out}.

Karnaugh maps are two-dimensional visual representations of truth tables. In a Karnaugh map, input variables are partitioned into vertical and horizontal variables, and all value combinations of input variables are expressed in Gray codes. The Gray code expressions allow the geometrically adjacent cells to become combinable using the law $AB + A\overline{B} = A$. For instance, cells $a\overline{b}c\overline{d}$ and $abc\overline{d}$ are combined as a product $ac\overline{d}$. For functions with large numbers of inputs, the corresponding truth tables or Karnaugh maps are too large.

Table 1. Truth Table of Full Adder

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Cube Representation. An array of cubes is a list of cubes, which is usually interpreted as a sum of products of literals, where a cube corresponds to a product of literals. A (binary) literal is a variable or a negated variable. In binary logic, symbol 0 corresponds to a negated variable, symbol 1 to a positive (affirmative, nonnegated) variable, symbol X to the absence of a variable in the product, and symbol ϵ to a contradiction. A cube is a sequence of symbols 0, 1, X, and ϵ, corresponding to their respective ordered variables. For instance, assuming the order of variables: x_1, x_2, x_3, x_4, the cube 01X1 corresponds to the product of literals $\overline{x_1}x_2x_4$, and the cube $0\epsilon X0$ is an intermediate data generated to show contradiction or a nonexisting result cube of some cube operation. A minterm (a cell of a Karnaugh map and a row of a truth table) is thus a sequence of symbols 1 and 0. Arrays of cubes can also correspond to exclusive sums of products, products of sums, or others. For instance, the array of cubes {01X1, 11XX} describes the sum-of-products expression $\overline{x_1}x_2x_4 + x_1x_2$ called also the cover of the function with product implicants (usually, with prime implicants). Depending on the context, the same array of cubes can also describe the exclusive-sum-of-products expression $\overline{x_1}x_2x_4 \oplus x_1x_2$, or a product-of-sums expression $(\overline{x_1} + x_2 + x_4) \cdot (x_1 + x_2)$. The correct meaning of the array is taken care of by applying respective cube calculus operators to it.

An algebra of cube calculus has been created with cubes and arrays of cubes and operations on them. The most important operators (operations) are negation of a single cube or nondisjoint sharp, disjoint sharp, consensus, crosslink, intersection, and supercube of two cubes. The cube operators most often used in EDA programs are presented briefly below. The nondisjoint sharp, $A\#B$, creates a set of the largest cubes in function $A \cdot \overline{B}$. Disjoint sharp, $A\#_dB$, creates a set of disjoint cubes covering function $A \cdot \overline{B}$. Sharp operations perform graphical subtraction and can be used in algorithms to remove part of the function that has been already taken care of. Consensus of cubes A and B is the largest cube that includes part of cube A and part of cube B. Supercube of cubes A and B is the smallest cube that includes entirely both cubes A and B. Consensus and supercube are used to create new product groups. Intersection of cubes A and B is the largest common subcube of cubes A and B. It is perhaps the most commonly used cube calculus operation, used in all practically known algorithms. These operations are used mostly in the inclusive (AND–OR) logic. Crosslink is the chain of cubes between two cubes. The chain of cubes covers the same minterms as the two cubes, and does not cover the minterms not covered by the two cubes. Since $A \oplus A = 0$, an even number of coverings is treated as no covering, and an odd number of coverings is treated as a single covering. It is used mostly in the exclusive (AND–EXOR) logic, for instance, in exclusive-sum-of-products minimization (21). Positive cofactor f_a is function f with variable a substituted to 1. Negative cofactor $f_{\overline{a}}$ is function f with variable a substituted to 0.

Cube calculus is used mostly for optimization of designs with two or three levels of logic gates. It is also used in test generation and functional verifications. Multivalued cube calculus extends these representations and operations to multivalued variables. In multivalued logic, each variable can have several values from a set of values. For a n-valued variable, all its literals are represented by n-element binary vectors where value 0 in the position corresponds to the lack of this value in the literal, and value 1 to the presence of this value. For instance, in 4-valued logic, the literal $X^{0,1,2}$ is represented

as a binary string 1110, which means the following assignment of values: $X^0 = 1, X^1 = 1, X^2 = 1, X^3 = 0$. It means, the literal $X^{\{0,1,2\}}$ is a 4-valued-input binary-output function defined as follows: $X^{\{0,1,2\}} = 1$ when $X = 1$, or $X = 2$, or $X = 3$, $X^{\{0,1,2\}} = 0$ when $X = 4$. Such literals are realized in binary circuits by input decoders, literal generators circuits, or small PLAs. Thus, multivalued logic is used in logic design as an intermediate notation to design multilevel binary networks. For instance, in 4-valued model used in programmable logic array (PLA) minimization, a 4-valued set variable corresponds to a pair of binary variables. PLA with decoders allow to decrease the total circuit area in comparison with standard PLAs. This is also the reason of using multivalued logic in other types of circuits. Well known tools like MIS and SIS from the University of California at Berkeley (UC Berkeley) (23) use cube calculus format of input/output data.

A variant of cube calculus representation are the factored forms (for instance, used in MIS), which are multilevel compositions of cube arrays (each array specifies a two level logic block). Factored form is thus represented as a multi-DAG (directed acyclic graph with multiedges). It has blocks as its nodes and logic signals between them as multiedges. Each component block specifies its cube array and additionally its input and output signals. Input signals of the block are primary inputs of the multilevel circuit, or are the outputs from other blocks of this circuit. Output signals of the block are primary outputs of the multi-level circuit, or are the inputs to other blocks of this circuit. Initial two-level cube calculus description is factorized to such multi-level circuit described as a factored form. Also, a multilevel circuit can be flattened back to a two level cube representation.

Binary Decision Diagrams. Decision diagrams represent a function by a directed acyclic graph (DAG). In the case of the most often used binary decision diagrams (BDDs), the nodes of the graph correspond to Shannon expansions (realized by multiplexer gates), controlled by the variable a associated with this node: $F = a \cdot F_a + \overline{a} \cdot F_{\overline{a}}$. Shared BDDs are those in which equivalent nodes of several output functions are shared. Equivalent nodes g and h are those whose cofactor functions are mutually equal: $g_a = h_a$ and $g_{\overline{a}} = h_{\overline{a}}$. Ordered BDDs are those in which the order of nodes in every branch from the root is the same. A diagram can be obtained from arbitrary function specifications, such as arrays of cubes, factored forms, expressions, or netlists. The diagram is obtained by recursive application of Shannon expansion to the function, next its two cofactors, four cofactors of its two cofactors, and so on, and by combination of any isomorphic (logically equivalent) nodes. The function corresponds to the root of the diagram. There are two terminal nodes of a binary decision diagram, 0 and 1, corresponding to Boolean false and true. If two successor nodes of a node S_i point to the same node, then node S_i can be removed from the DAG. There are other similar reduction transformations in those diagrams which are more general than BDDs. Decision diagrams with such reductions are called reduced ordered decision diagrams.

In addition, negated (inverted) edges are introduced in BDDs. Such edges describe negation of its argument function. In Kronecker decision diagrams (KDDs) three types of expansion nodes exist: Shannon nodes (realizing function $f = a \cdot f_a + \overline{a} \cdot f_{\overline{a}}$), positive Davio nodes [realizing function $f = a \cdot (f_a \oplus f_{\overline{a}}) \oplus f_{\overline{a}}$], and negative Davio nodes [realizing function $f = \overline{a} \cdot (f_a \oplus f_{\overline{a}}) \oplus f_a$]. All of the three possible canonical expansions of Boolean functions are thus included in KDD. Other known decision diagrams include zero-suppressed binary decision diagrams (ZSBDDs) and moment diagrams. They are used primarily in verification or technology mapping. Multivalued decision diagrams have more than two terminal nodes and multivalued branchings with more than two successors of a node. These diagrams allow one to describe and verify some circuits (such as large multipliers) that are too large to be described by standard BDDs. Some diagrams may also be better for logic synthesis to certain technologies.

There are two types of decision diagrams: canonical diagrams and noncanonical diagrams. Canonical diagrams are used for function representation and tautology checking. ZSBDDs and KDDs are examples of canonical representations. An example of noncanonical decision diagrams is a free pseudo-Kronecker decision diagram. In this type of diagram, any types of Shannon and Davio expansions can be mixed in levels and all orders of variables are allowed in branches. Free pseudo-Kronecker decision diagrams are used in synthesis and technology mapping (21,22). Decision diagrams can be also adapted to represent state machines. By describing a state machine as a relation, the (logic) characteristic function of the machine can be described by a decision diagram.

Level of Design Abstraction

A design can be described in different levels of abstraction, as shown in Fig. 1.

- Architecture level (also called behavioral level). At this level, the designer has the freedom to choose different algorithms to implement a design (for instance, different digital filtering or edge detection algorithms). The emphasis is on input–output relations. Different implementations for the same function can be considered. For instance, for a given function, one can chose between two logic implementations: sequential and parallel combinational (arithmetic adder, comparator or multiplier being good examples).
- Register transfer level (RTL). At this stage, the design is specified at the level of transfers among registers. Thus, the variables correspond to generalized registers, such as

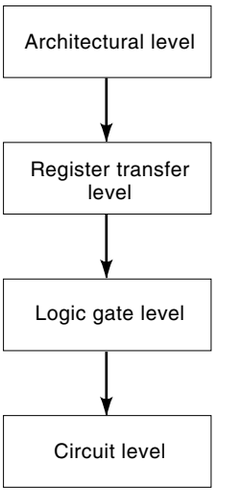

Figure 1. The abstraction levels of a logic design.

shifters, counters, registers, memories, and flip-flops. The operations correspond to transfers between registers and logical, arithmetical and other combinational operations on single or several registers. Examples of operations on a single register are shift left, shift right, shift cyclically, add one, subtract one, clear, set, negate. An example of more general register-transfer operations is $A \leftarrow B + C$, which adds the contents of registers B and C and transfers the result to register A. A register-transfer description specifies the structure and timing of operations in more detail but still allows for transformations of data path, control unit, or both. The transformations will allow improved timing, lower design cost, lower power consumption, or easier circuit test.

- Logic level (gate level). At this level every individual flip-flop and logic bit is specified. The timing is partially fixed to the accuracy of clock pulses. The (multioutput) Boolean functions with certain number of inputs, outputs, and certain fixed functionality are specified by the user or obtained by automatic transformations from a register-transfer level description. These functions are specified as logic equations, decision diagrams, arrays of cubes, netlists, or some hardware description language (HDL) descriptions. They can be optimized for area, speed, testability, number of components, cost of components, or power consumption, but the general macropulses of the algorithm's execution cannot be changed.
- Physical level. At this level a generic, technology-independent logic function is mapped to a specific technology—such as electronically programmable logic devices (EPLD), complex programmable logic devices (CPLD), field programmable gate arrays (FPGA), standard cells, custom designs, application specific integrated circuits (ASIC), read only memory (ROM), random access memory (RAM), microprocessor, microcontroller, standard small scale integration (SSI)/medium scale integration (MSI)/large scale integration (LSI) components, or any combinations of these. Specific logic gates, logic blocks, or larger design entities have been thus defined and are next placed in a two-dimensional area (on a chip or board) and routed (interconnected).

Logic Design Representations

A logic function can be represented in different ways. Both behavioral (also called functional) representations and structural representations are used in logic designs. The representations can be used at all different levels of abstraction: architecture level, register-transfer level, and logic level.

Waveforms. Waveforms are normally used for viewing simulation results and specifying stimulus (input) to the simulator. Recently they are also being used increasingly as one possible input data design specification, especially for designing asynchronous circuits and circuits that cooperate with buses. Figure 2 shows the waveforms of a full adder.

Logic Gate Networks. Standard design uses basic logic gates: AND, OR, NOT, NAND, and NOR. Recently EXOR and XNOR gates were incorporated into tools and designs. Several algorithms for logic design that take into account EXOR and XNOR gates have been created. For certain designs, such as arithmetic datapath operations, EXOR based logic can decrease area, improve speed and power consumption, and improve significantly the testability. Such circuits are thus used in design for test. Other gate models include designing with EPLDs, which realize AND–OR and OR–AND architectures, corresponding to sum-of-products and product-of-sums expressions, respectively. In standard cell technologies more powerful libraries of cells are used, such as AND–OR–INVERT, or OR–AND–INVERT gates. In FPGAs different combinations of multiplexers, cells that use positive Davio (AND–EXOR) and negative Davio (NOT–AND–EXOR) expansion gates, or similar cells with a small number of inputs and outputs are used. The lookup-table model assumes that the arbitrary function of some small number of variables (3, 4, or 5) and small number of outputs, usually 1 or 2, can be realized in a programmable cell. Several design optimization methodologies have been developed for each of these models.

Boolean Expressions. Boolean expressions use logic functors (operators) such as AND, OR, NOR, NOT, NAND, EXOR, and MAJORITY, as well as literals, to specify the (multioutput) function. In order to specify netlists that correspond to DAGs, intermediate variables need to be introduced to the expressions. Every netlist or decision diagram can be specified by a set of Boolean expressions with intermediate variables. Boolean expressions can use infix (or standard), prefix (or Polish), or postfix (or reverse Polish) notations. Most modern specification languages use infix notation for operators such as AND or OR. Operator AND can sometimes be omitted, as in standard algebraic notations. In conjunction with operators such as NAND, both infix and prefix notations are used. For instance, (NAND a b c) in prefix and (a NAND b NAND c) in infix. Care is recommended when reading and writing such expressions in hardware description languages and input formats to tools. It is always good to use parentheses in case of doubt about operators' precedence. In some languages, arbitrary operators can be defined by users and then can be used in expressions on equal terms with well-known operators. Expressions can be created for SOP (sum-of-products), POS (product-of-sums), factorized SOPs and POSs, and other representations as a result of logic synthesis and optimization algorithms. Some of these algorithms will be described in the section on combinational logic design.

Behavioral Descriptions. A logic system can be described by hardware description languages (HDL). The most popular ones are Verilog and VHDL. Both Verilog and VHDL can describe a logic design at different levels of abstraction, from gate-level to architectural-level representations. Both are now industrial standards, but VHDL seems to gain its popularity faster, especially outside the United States. In recent years

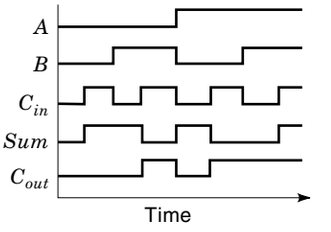

Figure 2. Waveforms for the full adder.

several languages at higher level than VHDL have been proposed, as well as preprocessors to VHDL language from these new representations, but so far none of them enjoyed wide acceptance (e.g., State Charts, SpecCharts, SDL, and VAL). State Charts and SpecCharts are graphical formalisms that introduce hierarchy to state machines. SDL stands for the Specification and Description Language. It is used mainly in telecommunication. VHDL Annotation Language (VAL) is a set of extensions to VHDL to increase its capabilities for abstract specification, timing specification, hierarchical design, and design validation. Other known notations and corresponding data languages include regular expressions, Petri nets, and path expressions.

Design Implementation

A design can be targeted to different technologies: full custom circuit design, semicustom circuit design (standard cell and gate array), FPGAs, EPLDs, CPLDs, and standard components.

In the full custom circuit designs, the design effort and cost are high. This design style is normally used when high-quality circuits are required. Semicustom designs use a limited number of circuit primitives, and therefore have lower design complexity and may be less efficient when compared to the full custom designs.

Design Verification

A design can be tested by logic simulation, functional testing, timing simulation, logic emulation, and formal verification. All these methods are called validation methods.

Logic Simulation. Logic simulation is a fast method of analyzing a logic design. Logic simulation models a logic design as interconnected logic gates but can also use any of the mathematical characterizations specified previously (for instance, binary decision diagrams). The simulator applies test vectors to the logic model and calculates logic values at the output of the logic gates. The result of the logic simulation can be either logic waveforms or truth tables.

Timing Simulation. Timing simulation is similar to logic simulation, but it also considers delays of electronic components. Its goal is to analyze the timing behavior of the circuit. The results from the timing simulation can be used to achieve target circuit timing characteristics (e.g., operational frequency).

Formal Verification. While simulation can demonstrate that a circuit is defective, it is never able to formally prove that a large circuit is totally correct, because of the excessive number of input and state combinations. Formal verification uses mathematical methods to verify exhaustively the functionality of a digital system. Formal verification can reduce the search space by using symbolic representation methods and by considering many input combinations at once. Currently, there are two methods that are widely used: model checking and equivalence checking. Model checking is used at the architectural level or register-transfer level to check if the design holds certain properties. Equivalence checking compares two designs at the gate level or register-transfer level. It is useful when the design is transformed from one level to another level, or when the design functionality has changed at the same level. Equivalence checking can verify if the original design and the modified design are functionally equivalent. For instance, two Boolean functions F_1 and F_2 are equivalent when they constitute a tautology $F_1 \Leftrightarrow F_2, = 1$, which means the function $G = F_1 \Leftrightarrow F_2$ is equal to 1 (or function $F_1 \oplus F_2$ is equal to zero) for any combination of its input variable values. A more restricted version of tautology may involve equality only on combinations of input values that actually may happen in actual operation of the circuit (thus "don't care" combinations are not verified). Verification of state machines in the most narrow sense assumes that the two machines generate exactly the same output signals in every pulse and for every possible internal state. This is equivalent to creating, for machines M_1 and M_2 with outputs z_1 and z_2, respectively, a new combined machine with output $z_{com} = z_1 \oplus z_2$ and shared inputs, and proving that output $z_{com} = 0$ for all combinations of state and input symbols (9). A more restricted equivalence may require the identity of output signals for only some transitions. Finally, for more advanced state machine models, only input–output relations may be required to be equivalent in some sense. Methods based on automatic theorem proving in predicate calculus and higher order logic have been also developed for verification and formal design correct from specification, but are not yet much used in commercial EDA tools. Computer tools for formal verification are available from EDA companies and from universities (e.g., VIS from UC Berkeley (5), and HOL (10) available from the University of Utah).

Design Transformation

High-level design descriptions make it convenient for designers to specify what they want to achieve. Low-level design descriptions are necessary for design implementation. Design transformations are therefore required to convert a design from a higher level of abstraction to lower levels of abstraction. Examples of design transformations include removal of dead code from the microcode, removal of dead register variables, minimization of the number of generalized registers, cost minimization of combined operations units (SUM/SUBTRACT, MULTIPLY, etc.), Mealy-to-Moore and Moore-to-Mealy transformations of state machines (which can change the system's timing by one pulse), transformation of a nondeterministic state machine to an equivalent deterministic machine, transformation of a parallel state machine to an equivalent sequential machine, and mapping of a BDD to a netlist of multiplexers.

Logic Design Process

A logic design is a complex process. It starts from the design specification, where the functionality of the system is specified. Design is an iterative process involving design description, design transformation, and design verification. Through each iteration, the design is transformed from a higher level of abstraction to a lower level. To ensure the correctness of the design, verification is needed when the design is transformed from one level to another level. Each level may involve some kind of optimization (for instance, the reduction of the description size). The logic design process is shown in Fig. 3.

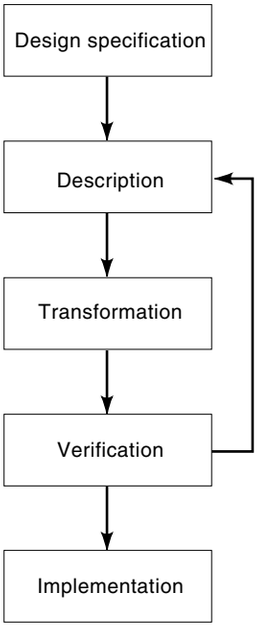

Figure 3. The logic design process.

COMBINATIONAL LOGIC DESIGN

A combinational logic design involves a design of a combinational circuit. For instance, the design may assume two levels of logic. A two-level combinational logic circuit consists of two levels of logic gates. In the sum-of-products two-level form, the first (from the inputs) level of gates are AND gates and the second level of gates are OR gates. In the product-of-sums two-level form, the first level of gates are OR gates and the second level of gates are AND gates.

The reason of logic minimization is to improve the performance and decrease the cost by decreasing the area of the silicon, decreasing the number of components, increasing the speed of the circuit, making the circuit more testable, making it use less power, or achieving any combination of the above design criteria. The optimization problem can be also specified to minimize certain weighted cost functions under certain constraints (for instance, to decrease the delay under the constraint of not exceeding certain prespecified silicon area).

There are usually two logic minimization processes; the first one is generic and technology-independent minimization, the next one is technology-dependent minimization, called also technology mapping. This second stage may also take into account some topological or geometrical constraints of the device.

Two-Level Combinational Logic

There are two types of programs for two-level logic minimization. Exact programs minimize the number of product implicants, the number of literals, or some weighted functions of the two. Heuristic or approximate programs attempt to minimize these cost functions but do not give assurance of their minimum values. Usually, the exact programs generate all prime implicants or a subset of them, which can be proven to include at least one minimal solution. The prime implicant is a product of literals from which no literal can be removed. These implicants are then the largest products-of-literals groups of true minterms in a Karnaugh map that imply the function. Next, an exact program creates a covering table and find its best covering with prime implicants. Such a table has true minterms as columns and prime implicants (or their subset) as rows (or vice versa). If an implicant covers (includes) a minterm, it is denoted by an entry 1 at the intersection of the row corresponding to the implicant and the column corresponding to the minterm. The exact solution (minimum, product-wise) is the subset of rows that cover (have minterms in) all columns and that has the minimum number of rows. The minimum solution (literal-wise) is the subset of rows that cover (have minterms in) all columns and has the minimum total number of literals in product terms (or that minimizes another similar cost function). Some algorithms use the concept of essential prime implicants. An essential prime is a prime that includes a certain minterm that is covered only by this prime. A redundant prime is an implicant that covers only minterms covered by essential primes. Redundant primes can thus be disregarded. A secondary essential prime is a prime that becomes essential only after previous removal of some redundant primes.

The essential primes found are taken to be the solution and the minterms covered by them are removed from the function (using for instance sharp operator of cube calculus). This causes some primes to become redundant and results in origination of the secondary essential primes. This process of finding essential and secondary essential primes is iterated until no further minterms remain in the function—thus the exact solution is found without creating and solving the covering table. Functions with such a property are called noncyclic functions. Most of real-life functions are either noncyclic or have large noncyclic cores, which is the reason for the relative efficiency of such algorithms.

Approximate algorithms try to generate primes and find primes cover at the same time; thus they reshape the current cover by replacing some groups of primes with other primes, applying cube operations such as consensus.

Program Espresso from UC Berkeley (4) is a standard for two-level logic functions. The original program was next extended to handle multivalued logic to allow for PLAs with decoders, function generators, and preprocessing PLAs. Espresso–MV is useful for sequential machine design, especially state assignment and input/output encodings. Its ideas help also to develop new programs for these applications. Although heuristic version of Espresso does not guarantee the exact solution, it is close to minimum on several families of practical functions. Its variant, Espresso–Exact, finds the minimum solution, and program Espresso-Signature can find exact solutions even for functions with extremely large number of prime implicants. This is due to a smart algorithm that can find exact solutions without enumerating all the prime implicants. There are some families of practical functions for which Espresso does not give good results and which are too large for Espresso–Exact or Espresso-Signature. Programs such as McBoole or other internally designed programs are combined with Espresso as user-called options in some commercial tools. Two-level minimization is used in many programs for multilevel synthesis, EPLD-based synthesis, and PLD/CPLD device fitting. These algorithms constitute the most advanced part of today's EDA theory and practice.

Topics close to sum-of-products minimization are product-of-sums design, three-level design (AND–OR–AND or OR–AND–OR), four-level design (AND–OR–AND–OR), and other designs with a few levels of AND–OR logic. Algorithms for their solution usually generate some kind of more complex implicants and solve the set-covering or graph-coloring problems, either exactly or approximately. Such approaches are used for PLD and CPLD minimization.

Another two-level minimization problem is to find, for a given function, the exclusive-sum-of-products expression with the minimum gate or literal cost. Several algorithms have been created for this task (21,22). Tools for circuits that have few, usually three, levels and have levels of gates AND, EXOR and OR, in various orders, have been also recently designed (21,22).

Many concepts and techniques used in two-level minimization (for instance, essential implicants or covering tables) are also used in multilevel synthesis. Similar techniques are used in sequential synthesis (for example, a binate covering problem is used in both three-level design and state minimization, and clique covering is used in creating primes and in several problems of sequential design).

An important stage of the logic design involves finding the minimum support of a function, which means the minimum set of input variables on which the given function depends. This can be used for logic realization with EPLDs (because of the limited number of inputs) or in ROM-based function realization.

Many efficient generic combinatorial algorithms have been created for logic synthesis programs. They include: unate covering (used in SOP minimization, decomposition and minimum support problems), binate covering (used in state machine minimization and three-level design), satisfiability (is F = 0? if yes, when?), tautology (is F = G?), and graph coloring (used in SOP minimization and functional decomposition). All these algorithms can be used for new applications by EDA tool designers.

Multilevel Combinational Logic

Factorization. A multilevel combinational logic circuit consists of more than two levels of logic gates. There are several design styles that are used to obtain such circuits. The first method is called factorization. It recursively applies factoring operations such as $ab + ac = a(b + c)$ and $(a + b) \cdot (a + c) = a + (bc)$. Some factoring algorithms also use other transformations as well, such as $a\bar{b} + a\bar{c} = a\overline{bc}$, and $abc\overline{cd} = ab\overline{abcd} = ab\overline{acd}$. Efficient factoring algorithms based on kernels and rectangle covering have been created (11). They are used in many commercial tools, and are still being refined and improved to add special functionalities (for instance, improved testability). They are also being adapted for state assignment or reduced power consumption of the circuit. Another advantage of factorization is that it allows representation of large circuits. A high-quality multi-level program, SIS, based mostly on factorization, can be obtained from UC Berkeley (23).

Functional Decomposition

The second group of multilevel synthesis methods are those based on functional decomposition. Such methods have originated from early research of Ashenhurst (2), Curtis (7), and Roth/Karp (20). Functional decomposition methods do not assume any particular type of gates: rather they just decompose a larger function to several smaller functions. Both functions can be specified as tables, arrays of cubes, BDDs, netlists, or any other aforementioned logic representations, both binary and multivalued. Functional decompositions can be separated into parallel and serial decompositions. Parallel decomposition decomposes multioutput function $[F_1(a, b, c, \ldots, z), F_2(a, b, c, \ldots, z), \ldots, F_{n-1}(a, b, c, \ldots, z), F_n(a, b, c, \ldots, z)]$ to several, usually two, multioutput functions, called blocks. For instance, $[F_1(a, b, c, \ldots, z), F_2(a, b, c, \ldots, z), \ldots, F_{n-1}(a, b, c, \ldots, z), F_n(a, b, c, \ldots, z)]$ is decomposed into $[F_{i_1}(a, \ldots, x), \ldots, F_{i_r}(c, \ldots, z)]$ and $[F_{i_{r+1}}(a, b, \ldots, y), \ldots, F_{i_n}(c, \ldots, x)]$, such that each component function depends now on fewer variables (thus the support set of each is decreased, often dramatically). This problem is similar to the partitioning of a PLA into smaller PLAs.

Serial decomposition is described by a general formula:

$$F(A, B, C) = H(A \cup C, G(B \cup C)) \tag{1}$$

where the set of variables $A \cup C$ is called the set of free variables (free set), the set of variables $B \cup C$ is called the set of bound variables, and the set of variables C is called the shared set of variables. If $C = \emptyset$, the decomposition is called disjoint, otherwise it is called nondisjoint. Function G is multioutput (or multivalued) in Curtis decomposition, and single-output in classical Ashenhurst decomposition. Function G can be also multivalued. Every function is nondisjoint decomposable, and many practical functions are also disjoint decomposable. The more a function is unspecified (the more "don't cares" it has), the better are the decompositions and higher the chances of finding disjoint decompositions with small bound sets.

It was shown that practical benchmark functions are well decomposable with small or empty shared sets, in contrast to randomly generated completely specified functions, for which such decompositions do not exist. Most of the decomposition methods decompose recursively every block G and H, until they become non-decomposable. What is defined as non-decomposable depends on any particular realization technology (for instance, any function with not more than four variables is treated as non-decomposable, assuming the lookup-table model with four input variables in a block). In a different technology, the decomposition is conducted until every block becomes a simple gate realized in this technology (for instance, a two-input AND gate, a MUX, or a three-input majority gate). Important problems in decomposition are finding good bound sets and shared sets and the optimal encoding of multivalued functions G to binary vectors, in order to simplify concurrently both functions G and H.

Designing combinational logic using ROMs or RAMs is another important area. Because of the limited width of industrial chips, one has to create additional combinational address generator circuits, or other circuits that collaborate with the memory chip. Some of these techniques are quite similar to decomposition.

Decomposition methods are not yet used in many industrial tools, but their importance is increasing. A universal functional decomposer program can be obtained from Portland State University (19).

Decision Diagrams. Finally, the last group of multilevel synthesis methods is based on various kinds of decision dia-

grams. In the first phase the decision diagram such as a BDD, KFDD, or ZSBDD, is created, and its cost (for instance, the number of nodes), is minimized. An important design problem is to find a good order of input variables (i.e., one that minimizes the number of nodes). For synthesis applications these diagrams are not necessarily ordered and canonical, because the more general diagrams (with less constraints imposed on them) can correspond to smaller or faster circuits. A universal Decision Diagram package PUMA that includes BDDs, KFDDs, and ZSBDDs, is available from Freiburg University (8). Free BDDs, with various orders of variables in branches, or noncanonical Pseudo–Kronecker decision diagrams with mixed types of expansions in levels, are used in technology mapping (22). In the next phase, certain rules are used to simplify the nodes. Among these rules, the propagation of constants is commonly used. For instance, a Shannon node (a MUX) realizing a function $a \cdot 0 + \overline{a} \cdot b$ is simplified to AND gate $\overline{a} \cdot b$. MUX realizing $ab \oplus \overline{a}$ is simplified to OR gate $\overline{a} + b$. All rules are based on simple Boolean tautologies. For instance, a positive Davio node realizing a function $a \cdot b \oplus a$ is simplified to an AND gate $a \cdot \overline{b}$. The heuristic algorithms that apply these transformations are iterative, recursive, or rule-based. They are usually very fast.

For some technologies, the stage of generic multilevel optimization is followed by the technology-related phase (called technology mapping), in which techniques such as DAG covering, or tree covering by dynamic programming are applied (11). At this stage, the particular technological constraints of a given target technology are taken into account by redesigning for particular cell libraries, fitting to certain fan-in or fan-out constraints, minimizing the number of cells to fit the IC device, or decreasing the number of connections to fit the routing channel width.

SEQUENTIAL LOGIC DESIGN

Sequential logic design involves designing sequential circuits. A sequential circuit contains flip-flops or registers. A good understanding of flip-flop's behavior is necessary to design sequential circuits. A flip-flop is an elementary register with a single bit. Flip-flops are synchronous and asynchronous. An example of an asynchronous flip-flop is a simple latch that changes its state instantly with the change of one of its inputs; it is set to state ON with logic value 1 on input S (set) and reset to state OFF with logic value 1 on input R (reset). The disadvantage of such a latch is a nonspecified behavior when both set and reset inputs are active at the same time. Therefore, synchronization signals are added to latches and more powerful edge-triggered or master-slave circuits are built, called the synchronized flip-flops. The most popular flip-flop is a D-type flip-flop. It has a clock input C, a data input D, and two outputs, Q and $\overline{Q}$. Output $\overline{Q}$ is always a negation of Q. The present state of input D (i.e., the excitation function D of the flip-flip) becomes the next state of output Q upon a change of the flip-flop's clock. Therefore, we can write an equation, $Q' = D$, where Q' is the new state of the flip-flop. The state changes may occur at the raising slope or the falling slope of the clock. The moment of change is irrelevant from the point of view of design methodologies, and modern methods assume that all change moments are of the same type. It is not recommended to design a sequential circuit that changes its states with both leading and falling slopes. A T-type flip-flop triggers its state from 0 to 1 and from 1 to 0 whenever its input T is 1 during the change of clock. It remains in its current state if the input T is 0 during the change. A JK flip-flop has two inputs; J is the setting input, K is the resetting input. Thus, with J = 1 and K = 0 the flip-flop changes to state 1 (with the clock's change). If both inputs are equal to 1, the flip-flop toggles, thus working as a T flip-flop. If they are both 0, it remains in its present state. Various procedures have been created to design special machines (such as counters or registers), general Finite State Machines, or other sequential mathematical models, with these flip-flops.

Standard FSM Design Methodology

Sequential logic design typically includes three phases. In the first phase a high-level description is converted into a state machine or equivalent abstract description. For instance, a regular expression is converted into a regular nondeterministic graph. The graph is converted into an equivalent deterministic graph, and finally into a Moore or Mealy machine state table. In some systems this table is then minimized. State minimization always attempts to reduce the number of internal states, and sometimes, also the number of input symbols. In some algorithms the numbers of output bits are also minimized. After minimization, the machine has a smaller table size but is completely equivalent, with accuracy of the clock pulses, to the initial table. The next design stage is the state assignment, in which every present and next state symbol in the table is replaced with its binary codes. In this way the encoded transition table is created, which functionally links the encoded present internal states, present input states, present output states, and next internal states. Now combinational functions δ and λ have been uniquely determined. They are usually incompletely specified. In some systems, the encoding (state assignment) is also done for input and/or output symbols. Assignment of states and symbols is done either automatically or manually. Good assignment leads to a better logic realization in terms of the number of product terms, literals, speed, etc. Several good assignment programs, KISS, MUSTANG, and NOVA, are available in the SIS system from UC Berkeley. DIET encoding program is available from University of Massachusetts, Amherst (6). State minimizer and encoder STAMINA is available from the University of Colorado. In general, much information about public domain or inexpensive programs for state machine design is available on the World Wide Web (WWW).

Because the minimized table is not necessarily the best candidate for state assignment, in some systems the phases of state minimization and state assignment are combined into a single phase and only partial state minimization results as a byproduct of state assignment of a nonminimized table (15). This means that some compatible states may be assigned the same code. This is done to minimize some complex cost functions, which may take into account all kinds of optimizations of logic realizing this machine: area, speed, power consumption, testability, and so on. State assignment is a very important design phase that links the stages of abstract and structural synthesis of state machines. It can influence greatly the cost of the solution [for instance, in FPGA or programmable array logic (PAL) technologies]. It can improve dramatically the testability of designs, and n out of k codes are used for

this task. For FPGAs good results are usually achieved by encoding machines in 1 out of k (or, one-hot) codes.

State Machine Decomposition

Another methodology of designing sequential circuits is decomposition. There are basically two methods of decomposition. Formal decomposition of state machines is a generalization of functional decomposition of Boolean and multivalued functions. Most of the decomposition methods are based on the partition theory (12), a mathematical theory also used in state assignment. This theory operates on groups of states that have some similar properties and groups of states to which these states transit under a given input. Other decomposition methods operate on state graphs of machines, as on labeled graphs in the sense of graph theory. They use graph-theoretical methods to partition graphs into smaller subgraphs that are relatively disconnected. Yet another decomposition method decomposes the given state table into two tables: one describes a state machine, such as a counter or shift register, and the other describes a remainder machine. For instance, methods have been developed to decompose a state machine to an arbitrary counter and the remainder machine. Another method decomposes a FSM into a special linear counter (that uses only D flip-flops and EXOR gates) and a remainder machine. Yet another decomposition type uses two counters, shift registers, fixed preprocessor or postprocessor machines, and many other variants as one of the blocks of the decomposition. Each of these methods assumes that there is some kind of elementary component machine that can be realized more inexpensively, can be realized in a regular structure, or has a small delay. Finally, methods have been developed to decompose a machine into several small machines, each of them realizable in some technology (for instance, as a logic block of a CPLD or a FPGA, or in a single PAL integrated circuit).

In addition to formal decomposition methods that operate on state tables, there are many informal and specialized decomposition methods that are either done by hand by the designer or are built into VHDL compilers or other high-level synthesis tools. For instance, many tools can recognize registers, counters, adders, or shifters in a high-level specification and synthesize them separately using special methods. Most of the existing approaches for sequential logic design assume the use of specific types of flip-flops, typically D flip-flops. There are, however, methods and algorithms, especially used in EPLD environments, that take into account other types of flip-flops, such as JK, RS, and T.

To directly convert and decompose high-level descriptions such as Petri nets, SpecCharts, or regular expressions to netlists of flip-flops and logic gates, or registers/shifters and logic equations, special methods have been also created, but are not yet very popular in commercial systems.

If a sequential circuit does not meet the requirements, there are postprocessing steps such as retiming, re-encoding, re-synthesis, speed-up, etc. (16,17), which can improve the design to achieve a higher performance. For instance, retiming allows shifting of registers through logic without changing its functionality but affecting the timing. This method is applied at many description levels of synchronous circuits: behavioral, register-transfer, architectural, logic, etc. It is most often used as a structural transformation on the gate-level, where it can be used for cycle-time minimization or for register minimization under cycle-time constraints (18). It has also been used for low power design and as a general optimization technique in architectural and logic synthesis.

Finally, approaches have been created that combine these different synthesis and decomposition methods, with the technology for which they are applied, such as EPLD, FPGA, PLA, ROM, or custom design. Design methods have also been created to realize finite state machines with RAMs, ROMs, and content addressable memories (CAM).

MICROPROCESSOR-BASED DESIGN

A microprocessor-based design involves a design of a digital system which contains one or more microprocessors.

What Is a Microprocessor?

A microprocessor is a general-purpose digital circuit. A typical microprocessor consists of a data path and a control unit, as shown in Fig. 4. The data coming from the system's input is manipulated in the data path and goes to the system's output. The data path contains registers to hold data and functional units to perform data-processing operations. These operations include arithmetic operations, logic operations, and data shifting. The control unit contains a program counter, an instruction register, and the control logic. The control unit controls the data path with regard to how to manipulate the data.

The microprocessor is operated under the control of software. The software is stored in standard memory devices. The software enables the same microprocessor to perform different functions.

A computer system is a combination of hardware and software designed for general-purpose applications. The microprocessor is the major component of a computer. Besides the microprocessor, a computer system hardware includes memory (RAM and ROM) and input/output devices. Memory can be a main memory and a microcode memory. Additional circuits, contained in FPGAs or special ASICs, and designed using the previously outlined techniques, can be also incorporated.

What Is a Microprocessor-Based System?

The fundamental microprocessor-based system structure contains microprocessor, memory, and input/output devices with address, data, and control buses (24). A microprocessor-based system is shown in Fig. 5.

The functionalities performed by a microprocessor are determined by the programs. These programs, commonly called software, are stored in the memory or off-line storage devices (e.g., hard drives). There are different technologies used in memory: static random-access memory (SRAM), dynamic random-access memory (DRAM), read-only memory (ROM), elec-

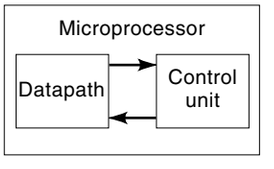

Figure 4. A simple microprocessor consisting of a datapath and a control unit.

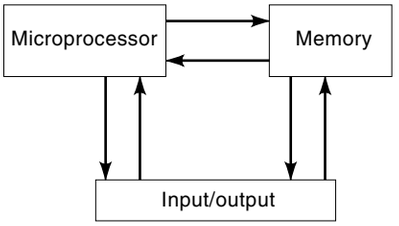

Figure 5. A microprocessor-based system.

trically programmable read-only memory (EPROM), electrically erasable programmable read-only memory (EEPROM), and flash memory. A program stored in a read-only memory is called a firmware.

The basic operation of all microprocessor-based systems is the same. The program in the memory is a list of instructions. The microprocessor reads an instruction from the memory, executes that instruction, and then reads the next instruction.

Logic Design Using a Microcontroller

A microcontroller is a complete computer system on a chip. Like the microprocessor, a microcontroller is designed to fetch and manipulate the incoming data, and generate control signals. A microcontroller contains a microprocessor, input/output (I/O) ports, timer/counter, and interrupt-handling circuit. A typical microcontroller (e.g., 8051) contains both serial and parallel I/O ports (3). Microcontrollers are widely used in applications like motor control, remote access, telephones and so on. The same microcontroller (e.g., 8051), running different software, can be used for different applications. Consequently, the overall product cost of a microcontroller-based design is much lower than an ASIC-based design.

While microcontrollers are commonly used in control applications, microprocessors are often used for applications requiring large amounts of I/O, memory, or high-speed processing. Such applications include data processing and complex control applications. For instance, personal computers mainly perform data processing.

Hardware Software Tradeoffs

A logic function can be realized in hardware, as discussed in the previous sections, or in software. In most cases, however, the required functionalities are realized partially by specially designed hardware and partially by software.

If the hardware is used more than the software, or vice versa, the designs are referred to as a hardware-intensive design or a software-intensive design, respectively. A designer can make a tradeoff between the hardware- and software-intensive realizations. Performance is usually better with hardware implementations than with software implementations for a variety of reasons. The microprocessor is a general-purpose device, and some speed is sacrificed for generality. Microprocessors perform tasks in sequential fashion. Logic circuits can be designed to perform operations in parallel. Most logic functions occur in tens of nanoseconds. A microprocessor instruction execution time ranges from several hundred nanoseconds to tens of microseconds.

A hardware-intensive design requires more hardware in the system and therefore increases the production cost. The performance is usually higher, and the cost for software development is reduced. Software-intensive approaches, on the other hand, require more software development and are slower. In return, the flexibility may be enhanced and the production cost is reduced.

Microprocessor Selection

There are many different microprocessor product families on the market. The number of bits processed in parallel inside the microprocessor is a primary criterion with which to evaluate the performance of a microprocessor. The low-end products are 4-bit or 8-bit ones. Typical microprocessors are 16 or 32 bits wide. The 64-bit products are currently entering the market. There are two factors that should be considered in microprocessor selection: architecture and development tools.

There are two types of microprocessor architectures: complex instruction set computer (CISC) and reduced instruction set computer (RISC). A CISC architecture has a larger instruction set than a RISC architecture. Besides the instruction sets, other considerations regarding architecture include on-chip and off-chip peripherals, operating frequency, and prices.

Development tools for microprocessor-based systems include in-circuit emulators, logic analyzers, and on-chip debuggers. In-circuit emulators are specially designed hardware that emulate the microprocessor operations in the target system. An in-circuit emulator has its own microprocessor and its own memory. During debugging, the tasks are run on an emulator's microprocessor and memory. The breakpoint can be set by the user through software to trace the system's operations. The emulator is connected to a workstation or a PC. Thus the user can monitor the system's performance in real time. Logic analyzers are devices that can monitor the logic values of a target system. They can be used to monitor any bus, control line, or node in the system, and they monitor the microprocessor passively. On-chip debuggers are software programs that can monitor a microprocessor's on-chip debugging registers.

Design Process

A microprocessor-based system can be as simple as a liquid crystal device (LCD) controller and can be as complex as a modern network management system. In spite of the diversity in the system complexity, the design of a microprocessor-based system always starts with a system-level specification. After the system-level functions are clearly defined in the system specification, the overall function is divided into different functional blocks. The hardware/software implementation and the selection of the components for each functional block will be determined at this stage. At this point, the tradeoff between hardware and software implementation and the advantage/disadvantage of each component need to be evaluated.

The system design activity is typically divided into hardware design and software design. Depending on the complexity of interaction between hardware and software, the two designs may need to be tested together in an integrated environment. The most commonly used tool for the integrated debugging is the in-circuit emulator, often referred to as ICE. An in-circuit emulator can run the software in the target microprocessor and provide the capability of monitoring the internal registers. As a result, an in-circuit emulator is an effi-

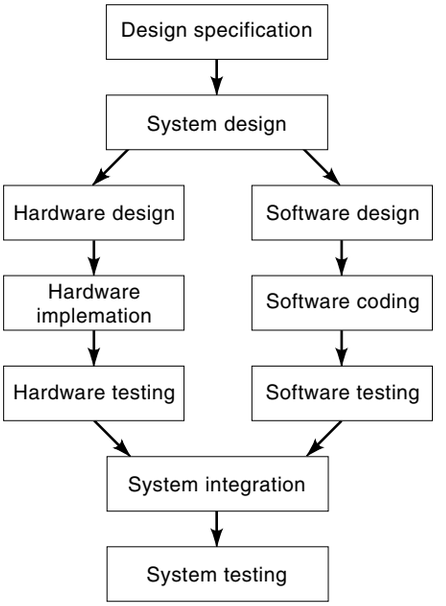

Figure 6. The process of a microprocessor-based design.

cient tool for debugging the software in real time. It can also interface with the hardware to provide the real application environment. This hardware-software co-verification is increasingly important in complex system design. A logic analyzer is also used to trace the signals in a real-time operation. This signal tracing involves the data storage and manipulation of the signal waveforms. Many in-circuit emulators have this capability built into the system.

A real-time design involves processing of the events at the speed at which the events occur. A popular example is the display of image data coming from the network. The image processing includes image data decompression and displaying onto the monitor window with a specified location and dimension. A real-time design is often performance demanding and needs to coordinate different event flows. Interrupt handling can also be complicated. In most cases, the design contains hardware circuit design and one or more processors. The hardware-software co-design and/or co-verification become imperative in a complex real-time design.

In summary, a microprocessor-based system design includes the following design activities: design specification, system design, hardware/software tradeoffs, microprocessor selection, other IC selection, software design and implementation, hardware design and implementation, hardware testing, hardware/software integration, hardware and software co-verification, and system testing. Figure 6 shows the stages of the microprocessor-based system design process.

Comparing Microprocessor-Based Design and Hardware Design

Microprocessor-based designs have several benefits. Software control allows easier modification and allows complex control functions to be implemented far more simply than with other implementations. A hardware design implementation is forwarded to the manufacturer and needs to be fully tested. A software implementation is more flexible than a hardware implementation. It has the ability to revise the design quickly and easily. Since the standards, specifications, and customer requirements are evolving constantly, the flexibility is an important design consideration.

On the other hand, while hardware designs are less flexible, they usually have better performance. Furthermore, microprocessor-based designs normally require additional devices, including program memory, data memory, and glue logic. Consequently, the requirements for board space and power consumption may be increased.

BIBLIOGRAPHY

1. P. Ashar, S. Devadas, and A. R. Newton, *Sequential Logic Synthesis,* Boston: Kluwer, 1992.
2. R. L. Ashenhurst, The Decomposition of Switching Functions, *Proc. of the Symposium on the Theory of Switching,* April 2–5, 1957, Ann. Computation Lab., Harvard University, 29, pp. 74–116, 1959.
3. K. J. Ayala, The 8051 Microcontroller, Architecture, Programming, and Applications, West Publishing Company, 1991.
4. R. K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis,* Boston: Kluwer, 1984.
5. R. K. Brayton et al., VIS: A System for Verification and Synthesis, in *Computer-Aided Verification,* July 1996.
6. M. Ciesielski and J. Shen, A Unified Approach to Input-Output Encoding for FSM State Assignment, *Proc. Design Automation Conf.,* 176–181, 1991.
7. H. A. Curtis, *A New Approach to the Design of Switching Circuits,* Princeton, 1962.
8. R. Drechsler et al., Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams, *Proc. of the Design Automation Conference,* San Diego, CA, June 1994, 415–419.
9. A. Ghosh, S. Devadas, and A. R. Newton, *Sequential Logic Testing and Verification,* Boston: Kluwer, 1992.
10. M. J. C. Gordon and T. F. Melham (eds.), *Introduction to HOL: A theorem proving environment for higher order logic,* Cambridge, UK: Cambridge University Press, 1993.
11. G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms,* Boston: Kluwer, 1996.
12. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines,* Upper Saddle River, NJ: Prentice-Hall, 1996.
13. R. H. Katz, *Contemporary Logic Design,* Menlo Park, CA: Benjamin/Cummings Publishing Company, 1994.
14. Z. Kohavi, *Switching and Finite Automata Theory,* New York: McGraw-Hill, 1970.
15. E. B. Lee and M. Perkowski, Concurrent Minimization and State Assignment of Finite State Machines, *Proc. IEEE Conference on Systems, Man and Cybernetics,* Halifax, Canada, Oct. 1984, pp. 248–260.
16. C. Leiserson, F. Rose, and J. Saxe, *Optimizing Synchronous Circuitry by Retiming, Third Caltech Conference on VLSI,* 1983, pp. 87–116.
17. G. De Micheli, *Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization, IEEE Trans. on CAD,* **10**: (1), Jan. 1991, pp. 63–73.
18. G. De Micheli, *Synthesis and Optimization of Digital Circuits,* New York: McGraw-Hill, 1994.
19. M. Perkowski et al., Decomposition of Multiple-Valued Relations, *Proc. ISMVL '97,* Halifax, Nova Scotia, Canada, May 1997, pp. 13–18.
20. J. P. Roth and R. M. Karp, Minimization over Boolean Graphs, *IBM Journal Res. and Develop.,* No. 4, pp. 227–238, April 1962.
21. T. Sasao (ed.), *Logic Synthesis and Optimization,* Boston: Kluwer, 1993.

22. T. Sasao and M. Fujita, *Representations of Discrete Functions, Boston: Kluwer, 1993.*
23. E. M. Sentovich et al., SIS: A System for Sequential Circuit Synthesis, *Tech. Rep. UCB / ERL M92 / 41,* Electronics Research Lab., Univ. of California, Berkeley, CA 94720, May 1992.
24. M. Slater, *Microprocessor-Based Design, A Comprehensive Guide to Effective Hardware Design,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
25. M. C. Zhou, *Petri Nets in Flexible and Agile Automation,* Boston: Kluwer, 1995.

NING SONG
Lattice Semiconductor Corp.
MAREK A. PERKOWSKI
Portland State University
STANLEY CHEN
Lattice Semiconductor Corp.

LOGIC DESIGN. See also NAND CIRCUITS; NOR CIRCUITS.
LOGIC DEVICES, PROGRAMMABLE. See PROGRAMMABLE LOGIC DEVICES.
LOGIC, DIODE-TRANSISTOR. See DIODE-TRANSISTOR LOGIC.
LOGIC, EMITTER-COUPLED. See EMITTER-COUPLED LOGIC.
LOGIC EMULATORS. See EMULATORS.
LOGIC, FORMAL. See FORMAL LOGIC.
LOGIC, FUZZY. See FUZZY LOGIC.
LOGIC GATES. See INTEGRATED INJECTION LOGIC.
LOGIC, HORN CLAUSE. See HORN CLAUSES.
LOGIC, MAGNETIC. See MAGNETIC LOGIC.
LOGIC, MAJORITY. See MAJORITY LOGIC.
LOGIC NETWORKS. See COMBINATIONAL CIRCUITS.
LOGIC OPTIMIZATION. See LOGIC SYNTHESIS.
LOGIC PROBABILISTIC. See PROBABILISTIC LOGIC.