ABSTRACT

This paper intends to present the current state of the art in Logic Synthesis within IBM and some aspects developed in IBM Éssonnes Component Development Laboratory. The application domain of Logic Synthesis is defined. Some high-level design languages are presented, since their practical features are essential in this process. The current synthesis approaches are detailed, with an emphasis on the algebraic approaches developed in Yorktown Silicon Compiler and implemented in Éssonnes. Examples are given of IBM machines designed with logic synthesis, and of an application to the optimization of an ASIC library.

THE APPLICATION DOMAIN OF LOGIC SYNTHESIS

Logic Synthesis is a set of techniques presently in a reasonable state of maturity, and currently used to produce VLSI designs. The expression usually refers to automated design actions to turn an abstract model, independent of any technology constraints, into a detailed logic model. This is typical of a Top-down design methodology.

Another very important aspect is remapping of an existing machine into a newer technology. Performance and manufacturing costs can be considerably improved while retaining a well-proven functional architecture and preserving a large existing software investment. This is currently applied to downsize several cards into one or two VLSI chips in a very spectacular manner.

ABSTRACTION & HARDWARE MODELS

The machine to be synthesized is modeled in a hardware description language of a variable degree of abstraction. A set of commonly accepted concepts have emerged (Fig. 1):

- General Specification
- Functional Model
- Register-Transfer Level Model
- Low Level Logic Structural model
- Implementation

The general specification level is often represented by a human written document in natural language, completed with many implicit facts connected with technical culture, architect's habits, past history, etc.

Description of the IBM S-370 computer architecture is a typical example, documenting in detail the instruction set, registers and machine states, external condition processing, etc.

The functional level is a behavioral model, expressed in an algebraic notation involving variables and operators, with no explicit reference to any physical objects or resources, nor to timing constraints.

The Register-Transfer-Model (RTL) model explicitly involves resource entities: registers, memories, and describes the transfers and operations between these resources and their logic conditions. Busses and structural concepts are not necessarily present.

The "Low Level Logic Structural network" is a structure of blocks, connected together by networks, each block being easily mapped to the circuit library of a given technology. At this stage one can proceed directly to implementation.

These models, of course, can be described by text or by an equivalent graph or network representation; the synthesis process can be viewed as deriving, from a machine description of a given level of modeling, a model less abstract and nearer to implementation.

HARDWARE DESCRIPTION LANGUAGES

A Hardware Description Language (HDL) is to describe a hardware model. Practically speaking, HDL's don't necessarily belong to one and only abstraction level: the structural aspect can be found at various levels. Often these HDL's allow to partition a model in a "structure" of "submodels" separated by "borders", linked with "connections"; each submodel can be internally described at various levels of abstraction. This facility allows a machine model to remain consistent while having its components evolve and undergo all stages of logic design independently from one another.

One success factor for a logic synthesis system is the HDL it supports, for practical reasons; the HDL features actually place some constraints on the synthesis process downstream.

Several HDL's have been used so far concurrently within IBM, mostly for historical and practical reasons: one of them being the availability of an efficient simulator operating from that HDL.

EPL/S (Extended Programming Language for Systems)

EPL/S is an extension of PL/S, a procedural programming language for writing system programs. EPL/S is typical of using an existing programming language and extending it to represent a behavioral model by adding specific macros and functions for this purpose, to represent the obviously missing hardware concepts. When compiled, this description becomes directly executable code (via links directly to a simulator and provides efficient behavioral simulation).

EPL/S thus has all the basic PL/S facilities, plus procedure statements (BLOCK, MINIMOD), timing control statements (WAIT, POST, SCHED), external connections (INPUT, OUTPUT). There are no other specifically hardware concepts (in particular, no hierarchy) and there is a wide gap between EPL/S and the low level boolean primitive network ready for implementation, to be filled by RTL languages.

BDL/CS (Basic Design Language for Cycle Simulation)

Initially conceived in Poughkeepsie, BDL/CS was designed as a flowchart language enabling the systems engineer to describe the behavior of his machine in a flow of actions and IF-THEN-ELSE's, as a pseudo-procedure (the statements can be re-ordered without changing the model); the associated simulator is a cycle-per-cycle simulator, with static simulation within a cycle. It can be considered a true RTL. Multiple simultaneous assignments are treated as an OR condition of each one of them; it is possible to describe independent entities operating in parallel.

Fig 1 - The range of abstraction of models.

CH2704-5/89/00000510/001.00 © 1989 IEEE
Developed alongside with a synthesis system and a simulation system, BDL/CS is completely technology independent and has been successfully used to describe logic networks of a million elements, mostly for the large IBM high-end processors. The self-documenting qualities of BDL/CS are appreciated.

shortcomings are a lack of hierarchical facilities (although it is possible to insert behavioral PL/S descriptions), and limitations to support multiple clocks and complex sequential logic. The clocking network is ignored, and this will have to be added manually later on to achieve a complete and consistent synthesis.

BOEBLINGEN DESIGN LANGUAGE

This language has been developed in Boeblingen Laboratory (11). Its purpose is to provide a structural hardware model description for a VLSI chip. Language concepts are Input & Output “pins”, combinatorial logic and other hardware elements (latches, drivers, receivers, etc). The description is made of simple operators (AND, OR, XOR, IF-THEN-ELSE, CASES, GATE), complex operators (MACROS, which can be hierarchically nested, and have conditional expansions). The statements are non-procedural (since they are to map to primitives by describing this primitive in a corresponding MACRO). The whole design can be partitioned.

This is a true structural language. There are no provisions for directly describing sequential or timing aspects, clocks, etc. but it is not the intent of the language.

The Boeblingen Design Language is linked to a synthesis system and has an associated simulator - the Boeblingen mixed level design verification simulator (12). This simulator has provisions for describing behavioral models by using an associated procedural programming-like language (cf. EPL/S). A particularly interesting feature of this simulator is that it allows simulation of multiple independent partitions that can be structural, or behavioral, or mixed, at will.

This global facility (Boeblingen Design Language + behavioral language + Mixed mode simulator) has proven quite flexible for design complete processors in VLSI (13).

SDL (System Design Language)

SDL originated from Rochester Laboratory, to be used to describe hardware for VLSI chips. SDL is a true RTL technology-independent. It has both structural and behavioral features that can be combined. The concepts include hardware facilities (INPUT, OUTPUT, OPERATOR, REGISTER, STORAGE, MACRO, etc.) and interactions between these facilities (OPERATION, AUTOMATON). There are IF-ELSE, DECODE, DO-ENDs for decision making; TRANSFER statements (LOAD, SET, RESET, CONNECT), logic operators (AND, OR, XOR) for AUTOMATON can accurately describe a finite-state machine. SDL is embedded in a comprehensive design methodology. SDL proved to be very flexible in its ability to describe a complex model with concurrent processes, sequential features, and a large capacity (40,000 logic gates in one design).

THE NEED FOR UNIFICATION: VHDL

Internally, the need evolved to unify these systems for obvious reasons. All the best features and concepts of these HDLs were put together to specify an "ideal", All-purpose HDL. It also happened that U.S. Department Of Defense contracted INTERMIX, Inc., together with IBM Corporation and Texas Instruments, Inc, to define a standard hardware description language, VHDL, an analyze-compiler, and a mixed mode simulator. Some people of the SDL team did actually participate in the definition of VHDL. VHDL is available today for everybody. IBM recently stated its intentions to promote the use of VHDL both internally and externally, to replace previous high-level Design Languages.

THE CURRENT LOGIC SYNTHESIS PROCESS

Two basic general approaches are currently active within IBM: One, which has started with applying a sequence of local transformations on a graph of the network, also using later on more global compiling techniques (1), (2), (5); this approach is currently implemented in Logic Synthesis System (LSS); Another approach which is more generally mathematic and algebraic, attempts at globally obtaining an optimized implementation (6), (7), (8).

An important point is that there is no absolute duality between local transform and global optimization. All tools developed have to utilize both techniques to achieve good results.

THE LOGIC SYNTHESIS SYSTEM (LSS):

The basic idea is to start from a high-level specification and to convert it, by a series of small steps, into a detailed structural boolean network ready for implementation. The structure used to store data during the process is conceptually made of “boxes” with terminals, these boxes being connected together by “wires” or signals. A box may be a primitive or may be defined by a network of more elementary boxes, thus allowing for a hierarchical structure. This storing scheme can house the implementation at all stages of its progressive flow from high-level abstraction to final implementation.

A sequence of transformations is applied to this network - the transformations are local, replacing a small subgraph of the network with another one, functionally equivalent. The transformations can be re-iterated and propagated, box after box, to the whole network. Each of these transformations is to change the network, gradually introducing the shapes and patterns of the final stage. There are also transforms of a more global nature, to perform some general optimizations.

The problem here is not necessarily to find the Optimal implementation, but to find a feasible one, that satisfies a large number of constraints: technology requirements, efficient use of features, satisfactory cell count and critical path length.

![Fig 2: A scenario for LSS synthesis](image)

Also some more global transformations are used, which are capable of achieving optimization beyond what strictly local transformations can provide (5).

Maintaining testability level is vital for VLSI production. Testability is related with redundancy: naively speaking, if a given circuit is removed from a network without modifying the apparent behavior, then this circuit is untestable and redundant. The stringent testability coverages goals (98.5%) are a must if a synthesized VLSI chip is to be accepted by an IBM production line. Some action must be taken in local transformations (avoid "re-convergent fan-outs", minimize circuits), more general approaches have been developed (4).

THE TRANSFORM TECHNIQUES

The techniques used can be very different (direct transformation, local and global pattern recognition, scoring techniques, global flow analysis). A given transform can be applied once or iterated, or propagated to the whole structure. The overall success relies
very much upon the proper sequencing of transforms. Also LSS works basically on a network structure, this allows to synthesize boolean circuits and also other classes of circuits that usually need very specific processing: latches, macros, interfaces, etc. This can be simply addressed by transforms. This approach has proven, in practice, very flexible and efficient at synthesizing and remapping a very wide variety of circuits, and makes LSS a comprehensive synthesis system.

THE ALGEBRAIC APPROACH

Based on the manipulation of boolean functions, these techniques have been developed in Yorktown Laboratory to the point of producing a general silicon compiler system (8). Essones Laboratory has applied them and enhanced them in the area of timing optimization and ASIC library support, and has developed a practical synthesis tool which is in current production use.

Boolean network optimization

A boolean network is one representation of a piece of combinatorial logic. It is a set of connected boolean expressions in the same way as a logic network is a set of connected circuits. A boolean network may be represented by a single matrix where rows are product terms and columns refer to primary inputs or intermediate variables. Each individual boolean expression is represented as a sum of product terms i.e. by a number of rows of the boolean matrix. At any step of the synthesis process, the matrix is a correct model of the design. Synthesis consists of applying a sequence of operators to this matrix. We will distinguish 3 classes of operations: Operations which optimize the boolean network, operations which map the network into a target technology and operations which optimize the timing performances of the design.

One main objective of logic synthesis is to provide a simplified boolean network i.e. a minimized representation of the design whatever the target technology is. We have used the following powerful set of algorithms for multi-level logic optimization developed in IBM Yorktown Lab:

* Simplification : This operator replaces a boolean expression by an equivalent but simpler one.
* Complementation : It replaces a boolean expression by its complement if simpler.
* Compaction : It eliminates "low value" intermediates or substitutes an expression into another if the result is simpler.
* Extraction : It extracts a sub-expression common to several expressions.

A correct sequence of these operations with a correct tuning of the procedure leads generally to an optimized design in terms of boolean representation. However, in some situations, results is not exactly what the designer wishes. Major problems we encountered are:

* Intermediate elimination or extraction may disturb significantly the incoming network. When further operators cannot reshape it conveniently, final results may be poor. This is particularly true for highly structured logics. One reason is that the tuning criteria, must based on the count of literals, are not always pertinent measures. Another reason is that boolean operators cannot really take into account the particularities of a technology or a logic.
* Kernel extraction and substitutions are monitored by design complexity and not by timing considerations. Boolean simplification of the design may severely impact its performances. Experience lead us to lightly modify the Yorktown operators to incorporate timing constraints. However results have not been really encouraging. Reason is that it is generally difficult, at this step of the process, to get a reliable estimate of the timing data and to validate the trade-off between boolean optimization and timing optimization.
* Special structures (such as multiplex, parity generator, decoders, etc.) must be detected and frozen as soon as possible. Otherwise, network transformations may lead to poor results.
* Boolean networks do not permit to easily handle multiple-outputs functions. This may be a serious handicap for some classes of structured logic.
* Redundancy removal remains a problem for multi-level logic.

**Boolean expression decomposition**

Once optimized, some expressions of the boolean network can be implemented as a single circuit, others cannot. Expressions which are too large must be broken into smaller pieces matching existing circuits (decomposition). Efficiency of this operation has a critical impact on the final result. Algorithms must take into account as precisely as possible all the characteristics of the circuits (boolean function, area, speed, pin dropping, fan-out, powering, etc.). This usually leads to incorporate into generic algorithms some technology-dependent procedures.

The present version of the program uses four types of decomposition algorithms: factoring, pattern matching, screening and peeling. Each algorithms is monitored by designer's specifications (Minimum silicon area and/or timing constraints on primary inputs and outputs).

* **Factoring**

Factoring is a process allowing the extraction of divisors of a boolean expression. Basically it is the same task as the above extraction operation (see 5). For a given expression, a list of divisors is returned by the extraction operator. The most "valuable" are extracted.

* **Pattern matching**

Technology characteristics may require to have some types of circuits extracted at the very beginning of the decomposition step. This is done by trying to recognize a given pattern in a given expression. The way to do that is to have an adequate description of the pattern, to find a set of variables matching the pattern and to verify that the sub-expression is the most optimized form of the original expression. This technique is used to extract some special functions such as Exclusive Or's, multiplexors, special And/or circuits, selectors etc. These circuits are selected for their physical or electrical characteristics.

* **Screening**

Screening is an operation which pulls apart components of an expression which are on disjoint sets of variables. To moderate some negative aspects of this operation, technology rules are used to balance the "values" of the components. These values monitor their extraction.

* **Peeling**

This is the general decomposition algorithm. It uses a simple factoring method. Given a boolean expression E of variables V, the operator tries to find a set of variables V' which satisfy the following condition: E = V' + F where F is the remaining part of E.

Peeling can be defined by the following steps:
1. Search for an expression F in V which is simplifiable. If F is simplifiable, it is extracted and the new expression is E = V' + F where F is the remaining part of E.
2. Repeat the above process on the new expression E until no further simplification is possible.

This method is very flexible since it can be easily monitored. For silicon area optimization, criterion is the occurrence frequency of each variable. For timing optimization, criterion is the arrival times of the input signals. The major problem of the method is that a sub-expression is kept as soon as it is mappable. This provides the minimum number of circuits but not always the minimum area nor the fastest design.

5 – 12
Timing performance optimization

Designers generally ask synthesis tools to guarantee timing requirements, i.e. the worst-case propagation delays between two register boundaries. This implies that:

- The designer may input all the pertinent timing information required for the piece of combinational logic he wants to synthesize.
- The program can perform a reliable timing analysis.
- The program can re-design the network to improve the performances where needed.

Timing driven decomposition

The first step in the synthesis process where timing constraints are considered is the decomposition step (See 1.2). The problem is to decompose a boolean expression such that the delay through the resulting network is as short as possible. One solution is to use a characteristic of the peeling method: The later an intermediate is created the nearer the output it appears. In other words, when a variable is late, it is better to use it as the peeling variable.

Delay path reduction algorithm

Delay path reduction is the operation which tries to improve the performance of a circuit by local re-syntheses of the network. The algorithm is composed of two phases. The first one determines and improves the parts of the network which have the most important impact on the timing performances. It performs a global analysis of the timing data and measures the influence of a given intermediate on the primary outputs. In other words, it computes for each intermediate and each primary output the gain in performance achievable when assuming that the intermediate has no more influence on the output. The second phase processes each output and tries to shorten the critical paths.

Other timing optimization features

Fan-out reduction

Two reasons leads to control circuit fan-outs. First one is the electrical functionality of the design. Second one is the optimization of the performances. A simple way to reduce the fan-outs is to add buffers at the outputs of the overloaded circuits. This operation generally increases the delays. Duplication of the circuit is another way but it increases the fan-outs of the feeding circuits. The buffering algorithm must be clever enough to use buffers everywhere they do not impact the performance and to duplicate circuits everywhere this does not overload the predecessors. The problem may be correctly solved.

Circuit powering

Generally, the same boolean function has several physical implementations corresponding to several circuit performances. This feature is another way to improve designs. It must be handled by the synthesis tool. The problem is that faster circuits have bigger input capacitances. Powering a circuit modifies the fan-outs of the feeding circuits in the same way as circuit duplicating. Thus, powering and fan-out control must be merged into the same set of algorithms. The complexity of these algorithms must be proportional to the resulting design improvements. Experience showed that they are much smaller than those provided by local re-syntheses.

Design partitioning and hierarchical synthesis

Another experience we got in using synthesis is that results are better when boolean operators are applied to a homogenous partition of logic i.e. a piece of logic corresponding to a functional entity. Besides, since the performance of some operators varies as a power of the number of expressions, the synthesizer provides faster designs and allows re-runs and tries when it is applied to a reasonably small logic network (less than 2000 gates).

Discussion to determine the right way between partitioned design and flat designs is not closed. However, some thorough experiments we did with fully hierarchical designs (non-IBM environment) showed a significant leverage of the method. Design cycles are much more shorter. Synthesis results are generally better. Responses to changes are much faster. For instance, the single shot synthesis of 863 gates consumes 1574 seconds CPU time. The hierarchical synthesis (9 partitions) of the same logic produces 827 gates and requires 484 seconds CPU time. Although hierarchy induces some additional procedures for the control of delay paths and circuit loading (fan-outs), we are convinced that the leverage it provides is important and undeniable. However, this opinion is not shared by a part of the designer community which prefers a more direct methodology.

An overall methodology is essential.

The whole VLSI Design Automation process must be kept consistent to achieve best results: synthesis alone is no panacea. Functional Equivalence between synthesis input and output can be checked by functional simulation (assuming there is a simulation coverage criterion) or better by boolean comparison. Performance must be monitored constantly from high-level design until final implementation, using increasingly refined timing models and preferably the same timing analysis process, and synthesis should forward the timing objectives to drive the physical design placement/wiring.

PRACTICAL RESULTS

Logic synthesis has been extensively used in production mode. The LSS synthesis process was initially experimented for 3081 processor design, and later on extensively on the 3090 processor series. It has also been used to synthesize almost all the fixed point co-processors and Modem signal processor chips.
The efficiency of an ASIC technology (Gate array/standard cell) relies on the overall density factor (considering the number of cells actually wireable, this incorporating the design of the cells, the number and organization of the wiring planes, and the efficiency of placement/wiring programs); another important factor is how adequate the variety and complexity of books appear, with respect to the machine to be synthesized. After extensive usage of the ASIC library, some books may appear quite useless after all, while some other ones, ill-rated at first, can prove to be quite effective.

So far, this has been appreciated empirically by designing a given library, having logic designers use it, making statistics on utilization frequency of each book, modifying the library contents to remove the "less useful" books, iterating, etc.

Logic synthesis can interestingly speed up this process: one needs only to code into the synthesis system a logic model of the ASIC library, including a performance and an area factor for each element, then a set of reference machine models can be run through synthesis as a benchmark, with all desired variations, to appreciate the contribution of every book to the quality of the result.

This technique has been applied in Essonnes with good success to optimize ASIC libraries. Results showed wide variations on the ability to take advantage of complex books in terms of final density or performance, the human logic designer being more handicapped in this respect.

One objection can be made here: this approach actually optimizes the library with respect to the behavior of the logic synthesis process, rather than according to how a human being performs manual logic design. Considering that more and more logic design is being done by logic synthesis, this objection tends to become less pertinent as logic synthesis becomes more widely used.

The situation here is somewhat similar to having to design the optimal instruction set of a new computer for best overall performance (MIPS) and usability. A human being programming in Assembler language does not exploit the instruction set in the same manner as an optimizing compiler does: an important point is getting the best overall performance and cost averaged over a significant sample of code. The fact that nowadays little code is written in assembly language does necessarily influence some instruction set implementations.

**CONCLUSION**

Logical Synthesis is today a mature process that has greatly improved VLSI design productivity: an average System designer can take advantage of the most recent VLSI technologies with very little investment in the actual silicon specifics, and in general improved results as for density and performance. The synthesis approach is completely remodelling the whole VLSI process.

**Acknowledgements**

The work on Logic Synthesis in Essonnes Laboratory owes much to the contribution of Philippe Bonneau.

**References**


