SYNTHESIS OF CONTROL SYSTEMS

GIOVANNI DE MICHELI

IBM T.J.WATSON RESEARCH CENTER
YORKTOWN HEIGHTS, NEW YORK
USA

1. INTRODUCTION

Computer-Aided Design (CAD) tools have made the design of Very Large Scale Integration (VLSI) circuits possible. Research emphasis is now directed towards design methods for automated synthesis and verification of high-performance circuits from behavioral-level specifications. In particular, the capability of designing rapidly high-performance processors is a key to commercial competitiveness. Fast turn-around designs allow to achieve system-level trade-off comparisons, by trying several architectures in the search for the best match between system structure and implementation technology. In this perspective, integrated synthesis systems are necessary for the research and development of integrated circuits and systems in the future years.

Several approaches to automate the synthesis process have been proposed and are referenced in this book. In the beginning it was expected that automated synthesis systems could solve most of the designers' needs: in particular that the "push-button" conversion of high level specifications into the mask geometries could yield circuits acceptable for industrial standards. Unfortunately the synthesis system of the first generation, called silicon compilers, did not meet the expectations: the "compiled" circuits did not have an efficient utilization of the silicon area and the timing performances were not comparable with those obtained by designing the circuit with other methodologies. For this reason the synthesis systems of the subsequent generation were designed with circuit performance optimization in mind. Because of the numerous possibilities in exploiting trade-offs at the design level, the silicon compilers evolved into design environments, i.e. complex CAD systems supporting synthesis, verification, testing, ...

We present here structures, methods and algorithms for circuit synthesis and optimization of control units for VLSI processors. We consider first the design methodologies used for control units and how they evolved with the advent of VLSI circuit technology and computer-aided design synthesis tools. Then we review the steps in the synthesis of control structures and how they are related to optimization techniques. We show how control functions can be described by tables of symbols and how tables can be processed. In particular we present the symbolic design methodology that can be applied to to the tabular description of interconnected combinational and sequential circuits. The goal of this methodology is to achieve an optimal synthesis of the circuits implementing control units in terms of silicon area. The computer implementation of the algorithms and experimental results are then presented.

2. CONTROL STRUCTURES

With the design of single chip micro-processors, the design of data-path and control-units has evolved in the direction of a unified implementation style. In particular, high performance processor designs benefit from the physical proximity of data-path and control and the possibility of tuning the control-unit to the data-path design.

Different design styles have been used for control-unit design. In the past, there was a clear distinction between custom and structured control-unit implementations. Custom implementations tend to exploit particular choices to enhance performance. For example, the control part of the Z8000 micro-processor was implemented by a custom interconnection of (random logic) gates. Structured implementations, as in the case of the M68000, take advantage of circuit design regularity to support complex function designs, as well as engineering changes. In a structured implementation of the control part the sequencing and control functions are implemented by a structured array that is referred to as sequencing and control-store or more simply control-store. The control-store is implemented by a Read Only Memory (ROM), a Programmable Logic Array (PLA) or a more elaborate structure [ANDR80].

Control units with structured implementations have been called micro-programmed, when the control mechanism can be programmed by the personality of the control store, which is a binary representation of a sequence of operations defined by the micro-program. In the seventies, with the advent of VLSI circuits, micro-programming became a way to increase the regularity of the chip structure [LEWI81]. Davis [DAVI72] recognized this new view of micro-programming and gave the following definition: "One particular class of control mechanism uses regularly organized storage arrays to contain a large part of the control information. Machines employing such control mechanism are said to be micro-programmed."

There are several advantages in using micro-programming. The design of the control unit is flexible, can be defined at a later stage of the processor design and can be easily modified. Moreover micro-programming allows a processor to emulate the instruction set of other machines. Different implementation strategies of micro-programmed processors have been used: the control-store can be located on-chip or off-chip and can be writable or not. There is a trade-off in these choices. High performance processors take advantage of the proximity of on-chip control store. However, only processors with writable or off-chip control store can take full advantage of the micro-programming feature of altering the micro-program when the processor design is complete. Only in this case it is correct to talk about micro-programmable processors. For practical reasons, processor have mainly read-only on-chip control-store and therefore should be called micro-programmed. We will consider this class of processors in the sequel.

The advent of automated synthesis systems, or silicon compilers, has changed the perspective of micro-programming. In reality, the entire processor is described as a program to the synthesis system which generates the layout of the masks needed to fabricate the chip. This process is automatic and takes a time which is negligible with regard to that needed to write the program describing the processor. The program can be altered quickly to perform a change in the processor design. Different approaches to synthesis exist as well as different families of high-level language representations [GOLD85]. Some synthesis systems are based on structural language descriptions, which specify the

circuit structure and in particular that of the control unit. In this case the micro-program can be seen as the routine defining the operation of the control unit. Therefore, the concept of micro-programmable processor is now related to the existence of a representation of the control structure as a structured sequence of elementary steps. Moreover, the distinction between custom and structured implementation of the control-store is fading as powerful synthesis systems [DARR84] [BRAY85a] have been able to generate implementations of multiple-level logic functions as interconnection of gates in a quasi-structured fashion. (For example as stacks of arrays of custom cells and channels providing the interconnection [OTTE84].) Some automated synthesis systems [SISK82] [BLACK85] [CAMP85] use behavioral-level circuit descriptions. In this case, the control structure is not even specified as an input datum; the structure is derived automatically from the system behavioral specifications and the resources (e.g. ALU, shifter, ...) used to synthesize the data-path.

The computer-aided synthesis of control units is based on the assumption of a specific model for the control structure. In principle, a general-purpose micro-sequencer structure could be used. However, such a choice would not exploit the particular features of each architecture and would therefore be inefficient. For this reason, we look for the common features in different control structures and we propose a general methodology for the optimal synthesis of the corresponding circuits. Before describing the circuit model, it is interesting to show how control structures have been implemented in two micro-processor designs.

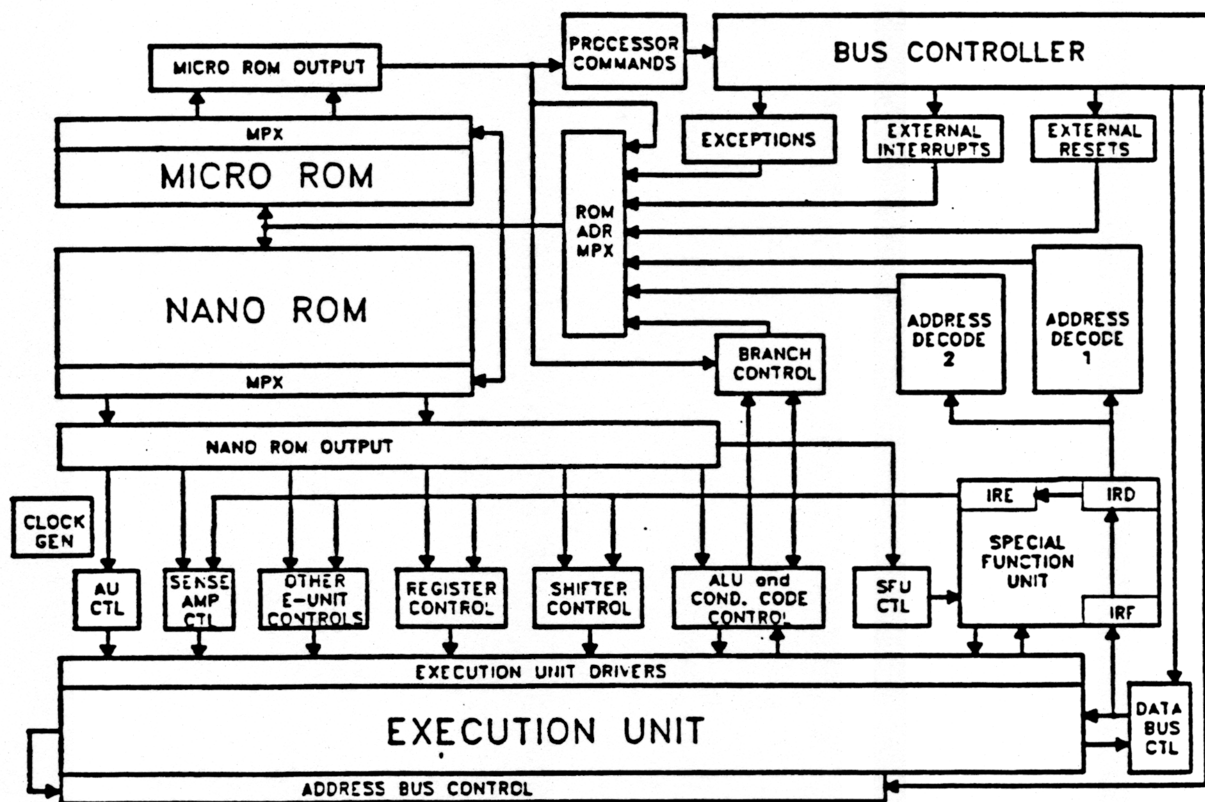

Fig 2.1 The u370 processor.

We consider first the design of a VLSI implementation of a Complex Instruction Set Computer (CISC): the micro-370 [HADS85]. The micro-370 is a 32-bit, single-chip, System/370 micro-processor. It executes 102 System/370 instructions. The control-unit is sketched in Fig. 2.1. The sequencing and control stores are kept in two ROMs which are called micro and nano ROM respectively. The sequencing may be altered by commands from the bus controller, the branch control and the special function unit. In essence, the control-unit consists of a Moore-type finite automaton (micro ROM) and a control decoding unit (nano ROM).

Let us consider now the design of the control part of a 801 processor, which has a streamlined architecture. This architecture has been the archetype of a family of processors, called Reduced Instruction Set Computers (RISCs), because the instruction set was simplified by deleting those instruction which are seldom executed and penalize the processor performance. We consider a particular implementation of the 801 micro-processor, done using the Yorktown Silicon Compiler [BRAY85c]. The 801 processor was architected and designed so that each instruction could be executed in a single cycle. The processor was implemented as a 4-stage pipeline. The execution control and interrupt circuit has the structure of a decoding network, that maps the representation of the instruction corresponding to each stage of the pipe to the controls of the related resources (Fig. 2.2). In essence, the control structure is a combinational decoding network.

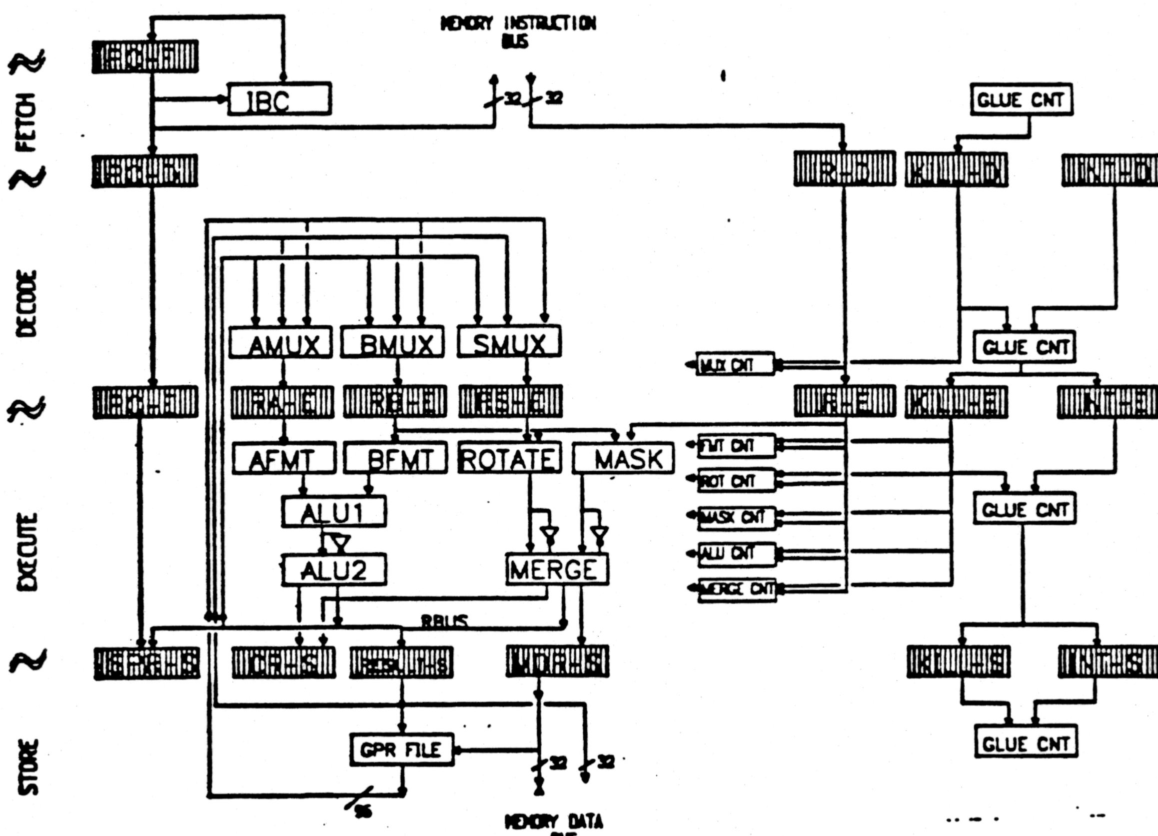

Fig 2.2 The 801 processor, as designed by the YSC.

Though the two example use radically different choices for the control implementation, there is a common feature in the representation of the two control structures. The two control units are personalized by one (or more) blocks of combinational logic. In the first case, the combinational function was implemented by a ROM, in the second by an interconnection of gates. The programmability feature of the control unit is retained in both cases by the personality of the combinational logic. In the first case, this personality is represented by the 0-1 patterns of the ROMs. In the second, the personality is represented by an interconnection of gates, which is a direct translation of the high-level tabular representation of the control operation. In the chip design described in [BRAY85c], the translation was performed automatically by the Yorktown Silicon Compiler.

In general, a control unit can be modelled as a **deterministic automaton** or **deterministic Finite State Machine** (FSM). Since sequential functions can be represented, using the Huffman model [HILL81], as the combination of a combinational circuit and a memory, then a generic control unit can be represented as an interconnection of a combinational circuit and memory. In particular, if the control function is combinational in nature, as in the case of the 801 design [BRAY85c], the memory component is not implemented and the automaton has only one state. If the control function is not combinational, we assume that memory is synchronized to the system clock (synchronous FSM) to avoid race conditions, as done in most designs. Memory is then implemented by clocked registers, which store the control state of the machine. We assume here the use of Delay-type registers.

Though this model is fairly general, a control unit implementation using an interconnection of one combinational block and a set of registers can be effective only for control functions of small to medium-sized complexity. When the control function is very elaborate, as in the case of some microprocessor designs, it is convenient to break the combinational block into an interconnection of circuits. For example, multiplexers may be useful to implement the next control-state function following a conditional branch. Auxiliary circuits, such as counters and stacks may be used. The structure of the control unit can be therefore generalized to an interconnection of combinational circuits, whose personality can be programmed, and fixed-structure circuits, such as registers and auxiliary circuits. We consider this generalization as a canonical structure for control units and we concentrate on the design of the combinational blocks that personalize the control unit and their optimal implementation.

3. COMPUTER-AIDED DESIGN OF CONTROL STRUCTURES

The synthesis of a control unit can be partitioned in the following tasks: **functional design, logic design** and **physical design**. Functional design consists of mapping the high-level specification into a control structure. Logic design consists of transforming this structure into Boolean equations (or an equivalent representation) and to optimize this representation. Physical design addresses the generation of the mask geometries from the logic specifications. Several computer-aided tools have been developed for the last two tasks, because the underlying problems are well understood. The automatic synthesis from behavioral-level specifications is still an exploratory area for design automation tools.

Functional design consists of defining the structure of a control unit from the high-level specification of the entire circuit. Its starting point is the description of the processor in a behavioral language. Its goal is to define the sequencing of the micro-operations and the timing of the control signals to the data-path as well as the partitioning of the control unit into functional blocks. In other words, functional design provides a control structure as an interconnection of logic blocks and timing blocks (reg-

isters). The information needed to perform the functional design is embedded in the behavioral language description. In particular, this description contains the sequencing of the operations independently of the resources used as operators.

Because of its complexity, in the past this task has been performed directly by the system designer, who transformed the description of the architecture in a natural language into a flow-chart or equivalent representation. For example, Tredennick reported in [TRED81] on the functional design of the M68000 control unit, which was achieved by the designer himself without the use of computer aids. Similarly, Brayton *et al.* reported in [BRAY85c] on the process of transforming the description of a 801 processor in a natural language into the YLL structural language [BREN84]: the control of the processor was specified using YLL routines and tables. In other cases, the specifications of the control unit are given directly by means of a micro-program. Micro-compilers are used to to generate the tables that represent the sequencing and the control signals. This tables are then translated into the 0-1 patterns of a ROM (if ROM are used for the control store) or into the specifications of a PLA [PAPA79] or other hardware circuit. In the case of ROM implementation, it is important to minimize the number of the words representing the micro-program. This kind of optimization at the functional level can be done by using compiler optimization techniques [AGER76].

Computer-aided tools for functional design, that make automated synthesis possible starting from the processor behavioral specifications, have been developed only recently [SISK82] [BLACK85] [CAMP85]. The most straight-forward approach to behavioral synthesis is to determine the structure of the data-path first and then to infer the timing and sequencing of its control signals. In this case, the states of the processor and the timing of the control signals to the data-path are derived from the sequencing information in the behavioral description and the timing specifications of the resources used in the data-path as operators. The specification of the control unit can then be generated in a form equivalent to a table that specifies the control states and the micro-operations. Note that the synthesis of the control unit does not necessarily have to follow the data-path synthesis. Indeed, they could be synthesized at the same time. Optimization techniques can then be used to exploit the trade-off between data-path cycle time and resource allocation by introducing/reducing cycles to perform each instruction. As a result, the number of machine states and/or state transitions is modified accordingly. There are some difficult problems associated with optimal functional design which are still under investigation. One is the determination of the optimal pipe-lining of a processor. Another one is to determine a partition of the control unit into resources (e.g. a hierarchy of finite automata) which fits the architecture. Knowledge-based systems, such as the Design Automation Assistant [KOWA85], could be used in this perspective.

The output of the functional design phase is the interconnection of the control structure and the specification of its blocks. Since combinational and sequential functions can be expressed in tabular format, we use here tables of symbols (mnemonics) to represent them. In general, control functions can be represented by interconnection of sequential and combinational functions as a set of linked tables. The formalism of the tabular representation will be presented in the next section.

Logic design consists of mapping the structural description into a representation in terms of logic variables. The straightforward mapping of functional tables into Boolean equation is a trivial procedure but yields inefficient implementations in terms of silicon area and switching-time performances. Logic design includes a set of optimization procedures aiming at simplifying the circuit implementing

the function. This simplification correlates with silicon area reduction and switching-time improvement. Logic optimization includes a vast set of techniques, that apply to sequential and combinational functions with different flavors according to the target implementation technology. We consider in this section methods that apply to ROM and two-level logic implementations of combinational functions (including the combinational component of sequential functions). If multiple-level logic implementations are used, the techniques for optimal two-level logic design described here are used to obtain a "good" starting point for multiple-level synthesis. Circuit optimality is related to an implementation choice. For ROM implementations, the optimality is measured by the the number of words and the number of bits. For two-level logic implementations, the optimality is usually measured by the cardinality of the corresponding Boolean cover. For multiple-level logic implementations, the optimality may be measured as a function of the total number of gates and literals. These last two measures correlate to the silicon area penalty of the implementation.

An important task in optimal design of sequential circuits is **state minimization**, which is the search for a representation of the control function using a minimal number of states. Reducing the number of states corresponds to reducing the number of transitions in the sequencing function and eventually to the reduction of the number of words in a ROM implementation and of the number of implicants in a two-level logic implementation. Moreover, a reduction of the number of states corresponds to a reduction of the number of bits needed to represent the states and therefore the number of bits in a ROM and to a simplified transition function in a two-level logic implementation. State minimization has been the object of extensive research. We refer the reader to [HART66] for the basic formalism and to [REUS86] for some recent results and an updated set of references. It was shown in [HOPC71] that the reduction of completely specified finite automata can be achieved in $O(n \log n)$ steps. The minimization of incompletely specified automata is a NP-complete problem [PLEE73]. Unfortunately, most automata derived from control structures are uncompletely specified. Therefore heuristic techniques are used.

The following steps in optimal logic design described in this section depend on the nature of the circuit implementation. If ROMs are used to implement the sequencing and control store, then logic optimization aims at reducing the number of words and/or the number of bits. The number of words can be reduced by several techniques. In particular the detection of possible concurrency of micro-operations as well as data and resource dependencies are used in this perspective. We refer the interested reader to [AGER76] for a set of references. The number of bits in the ROM can be reduced by encoding techniques. For example, consider a ROM implementing the control store only. (Assume the sequencing function is stored elsewhere.) If n resources have to be controlled, then n bits are needed by a fully horizontal microcode implementation. Note however that only as many bits as the ceiling of $\log_2 n$ are necessary in the case of a vertical microcode organization. While vertical microcode allows to reduce maximally the bit dimension of the ROM, it requires a decoding network at its outputs (Fig 3.1). A complete decoding network may be expensive in terms of silicon area and offset the advantage of the ROM bit reduction. Intermediate solution are often used. The basic idea is to group the bits controlling the resources into fields and use one decoder for each field (Fig 3.2). In this case, no word in the ROM can activate two resources whose control is encoded in the same field. This restriction is rather mild, because the possible combinations of concurrent resource controls is often much smaller than 2^n. Schwartz [SCHW68] proposed a an encoding method that minimizes the number of decoders. However Schwartz's method does not minimize the number of bits under the assumption that resources controlled simultaneously by at least one word are assigned to different decoders.

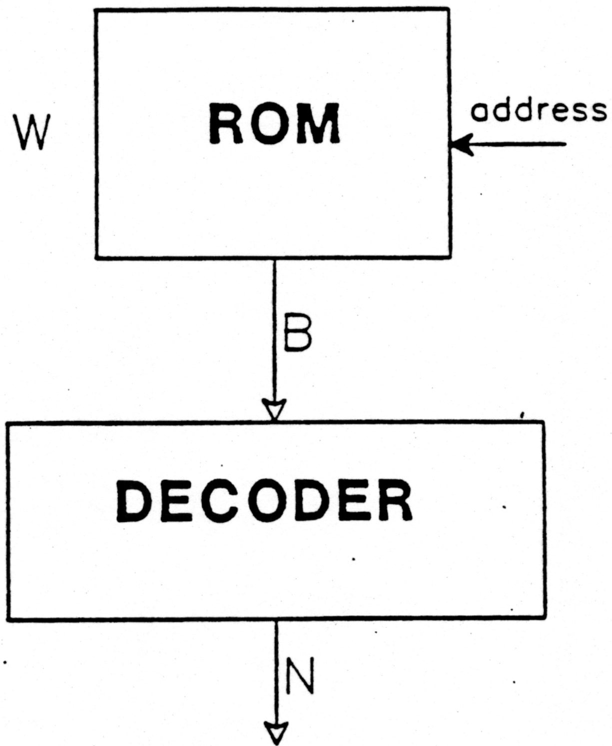

Fig 3.1 ROM and decoder.

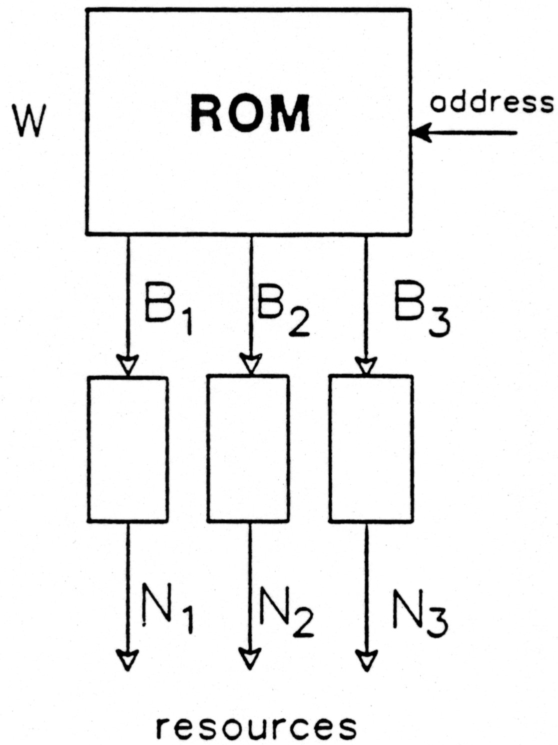

Fig 3.2 ROM with multiple decoders.

Grasselli *et al.* [GRAS70] reformulated the method in terms of switching theory and presented an algorithm for computing an encoding that minimizes the number of bits in the ROM. When a multiple output two-level logic circuit is used to implement the sequencing and control store, then the logic optimization problem corresponds to minimizing the number of implicants and the number of bits used to represent the control state. The control store may be implemented by one logic function or by the interconnection of two (or more) functions. We consider in this section the case in which the sequencing and control store in implemented by one function, i.e. a two-level function implements the combinational component of the control FSM. The extension of logic optimization to the interconnection of logic circuits will be shown in the next section.

The problem of assigning a binary representation to the control states is a classical problem of switching theory and is referred to as **state assignment** or **state encoding** [HART66]. State assignment affects the number of implicants in a minimal Boolean cover of a two-level implementation of the control store. If a PLA is used to implement the Boolean function,[1] we may assume that each PLA row implements an implicant and each column is related to an I/O signal (primary input/outputs and the encoding of the present/next states.) The PLA area is proportional to the product of the number of rows times the number of columns. Both row and column cardinality depend on state encoding. The (minimum) number of rows is the cardinality of the (minimum) cover of the FSM combinational component according to a given assignment. The encoding-length is related to the number of PLA columns. Therefore the PLA area has a complex functional dependence on state assignment. For this reason two simpler optimal state assignment problems are considered: i) find the assignment of minimum code length among the assignments that minimize the number of implicants; ii) Find the assignment that minimizes the number of implicants among the assignments of given code length. The optimum solution to the state assignment problem which minimizes the PLA area can be seen as a trade-off between the solutions to problem i) and ii). Note that the above problems are still computationally difficult and to date no method (other than exhaustive search) is known that solves them exactly. Therefore heuristic strategies are used to approximate their solution. We refer the reader to [DEMI85a] for an extended set of references and a critical survey of most of the previous techniques for the state assignment problem. The selection of the type of registers used to store the machine state affects also the size of the implementation. This problem was addressed by Curtis [CURT69] [CURT70] in connection with the state assignment problem.

If the control function is combinational, then logic design consists of encoding the controls into Boolean variables and determining an optimal combinational function. For two level-logic implementation, this is the classic **logic minimization** problem [BRAY84b].

Physical design is the synthesis of the layout of the circuit implementing the control unit layout from the logic specification. It depends on the layout style and the processing technology. If PLA based implementations are chosen, the circuit can be further optimized by using folding or partitioning techniques [DEMI84a] [RUDE85b]. If custom cells implementations are used, as in the case of the Yorktown Silicon Compiler, then the methods described in [BRAY85a] apply.

[1] It is assumed that the PLA is not folded nor partitioned for the sake of simplicity.

4. SYMBOLIC DESIGN

We consider here an innovative approach to the state encoding problem and the minimization of the combinational part of a control unit. This approach is called **symbolic design methodology**.

In the standard approach to synthesis [HILL81], Boolean representations of switching functions are obtained from the structural description by representing each each mnemonic entry in a table by Boolean variables. The optimization of logic functions, and in particular two-level logic minimization, is performed on the Boolean representation. The result of logic optimization is heavily dependent on the representation of the variables. As an example, the complexity (in particular the minimal cardinality of a two-level implementation) of the combinational component of a finite-state machine depends on the assignment of Boolean variables to the internal states [HART66].

The **symbolic design** methodology presented here avoids the dependence on the variable representation in the optimization process and consists of two steps: i) determine an optimal representation of a switching function independently of the encoding of its inputs and outputs; ii) encode the inputs and outputs so that they are compatible with the optimal representation. This technique can be applied to solve the following problems of logic design:

P1) Find an encoding of the inputs (or some inputs) of a combinational circuit that minimizes its cost.
P2) Find an encoding of the outputs (or some outputs) of a combinational circuit that minimizes its cost.
P3) Find an encoding of both the inputs and the outputs (or some inputs and some outputs) of a combinational circuit that minimizes its cost.
P4) Find an encoding of both the inputs and the outputs (or some inputs and some outputs) of a combinational circuit that minimizes its cost and such that the encoding of the inputs is the same as the encoding of the outputs (or the encoding of some inputs is the same as the encoding of some outputs).

Finding an optimal state assignment of a sequential circuit is equivalent to solving problem P4, when the sequential circuit is implemented by feeding back (possibly through registers) some outputs of a combinational circuit to its inputs. Similarly, finding the encodings of the signals connecting two (or more) combinational circuits, that minimize the total cost, can be reduced to problem P4. The author presented in [DEMI84c] [DEMI85a] an approximation to the solution of the state assignment problem, in which the cost was minimized with regard only to the encoding of the inputs. In particular, the technique presented in [DEMI84c][DEMI85a] solved only problem P1. Problem P2 was attacked by Nichols [NICH65], but the algorithm he presented could deal only with small-scale circuits. A solution to problems P1-P4 was presented for the first time in [DEMI86a].

Though the symbolic design methodology is fairly general, we restrict our attention here to two-level *sum of products* implementations. Since the area of the physical implementation has a complex functional dependence on the function representation (even by using PLA implementations [DEMI85a]), we consider a simplified optimization technique that leads to quasi-minimal areas. In particular we attempt to find first a *sum of products* representation that is minimal in the number of

products, and then a representation of the input/outputs that is minimal in the number of Boolean variables.

The difficulty in solving problems P2-P4 is related to finding a minimal two-level representation of a switching function independently of the encoding of both inputs and outputs. We use here a technique called **symbolic minimization.** Symbolic minimization consists of determining a minimal encoding-independent two-level *sum-of-products* representation of a switching function. It is minimal in number of product-terms and independent of the encoding of all (or part of) the inputs and outputs [DEMI85c]. The minimal symbolic representation is an intermediate step towards the determination of a corresponding Boolean representation. For this reason, three encoding problems are introduced in Section 6 to transform the minimal symbolic cover into an equivalent Boolean representation.

We present first an informal overview of symbolic design. The methodology is introduced by elaborating on an example. We consider first a combinational circuit (Fig. 4.1) and we use symbolic design to find a representation of its inputs and outputs and a corresponding two-level implementation that minimize its cost. This is equivalent to solving problem P3. Note that problem P1 or P2 can be derived from problem P3 by considering only the circuit inputs or outputs.

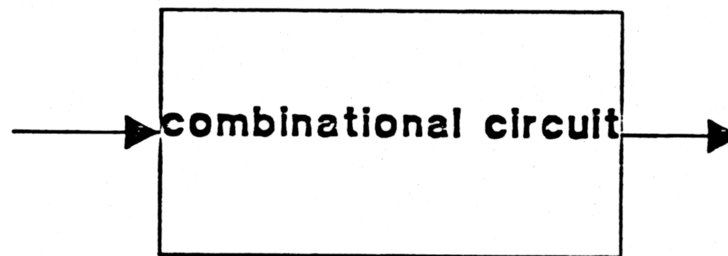

Fig 4.1 Combinational circuit.

Example 4.1: The following truth table specifies a combinational circuit; in particular an instruction decoder. There are three fields: the first is related to the addressing-mode, the second to the operation-code and the third one to the corresponding control signal. The circuit has two 2 inputs and 1 output. Each row specifies a symbolic output for any given combination of symbolic inputs.

INDEX	*AND*	*CNTA*
INDEX	*OR*	*CNTA*
INDEX	*JMP*	*CNTA*
INDEX	*ADD*	*CNTA*
DIR	*AND*	*CNTB*
DIR	*OR*	*CNTB*
DIR	*JMP*	*CNTC*
DIR	*ADD*	*CNTC*
IND	*AND*	*CNTB*
IND	*OR*	*CNTD*
IND	*JMP*	*CNTD*
IND	*ADD*	*CNTC*

In the standard approach to synthesis, each word (mnemonic string) in the table would be encoded by a string of binary symbols (i.e. 1's and 0's). Then, the encoded table would be minimized by a logic minimizer. In symbolic logic design, the table is minimized first (i.e. a table consisting of a minimal number of rows is computed) and then the table is encoded.

A first approach to symbolic minimization can be achieved by grouping the set of inputs that correspond to each output symbol. This process of reducing the size of the table is called here **disjoint minimization**, because the table is considered as set of independent sub-tables corresponding to each output symbol. Remarkably, disjoint minimization can be achieved by using techniques of multiple-valued logic minimization [BRAY84b] [DEMI85a], as shown is Section 5.

Example 4.2: From the table of Example 4.1, we can see that the addressing mode *INDEX* and any operation-codes *AND OR ADD JMP* correspond to the control *CNTA*. Similarly either one of the following conditions:

addressing mode *DIR* and operation-codes *AND* or *OR*

addressing mode *IND* and operation-code *AND*

correspond to control *CNTB*. The entire table can be expressed as a set of conditions:

INDEX	*AND OR ADD JMP*	*CNTA*
DIR	*AND OR*	*CNTB*
IND	*AND*	*CNTB*
IND	*OR JMP*	*CNTD*
DIR IND	*ADD*	*CNTC*
DIR	*JMP*	*CNTC*

Note that this table is more compact than the previous one, because it requires only 6 rows instead of 12.

The problem now is to find a Boolean representation of the symbols, corresponding to a Boolean cover representation of the function, with as many rows as the compacted table. While the encoding problem will be presented in detail in Section 6, we show here by an example the consequence of the choice of a particular encoding.

Example 4.3: Consider this particular Boolean encoding of the words.

INDEX = 00	*AND* = 00	*CNTA* = 11
DIR = 01	*OR* = 01	*CNTB* = 01
IND = 11	*ADD* = 10	*CNTC* = 10
	JMP = 11	*CNTD* = 00

Then the function can be represented by a Boolean cover as:

```
00  **  11
01  0*  01
11  00  01
11  *1  00
*1  10  10
01  11  10
```

where a *don't care* condition on a binary input variable is represented by *. Note that the fourth Boolean implicant can be deleted, because its output part is 00. Moreover note that by deleting this implicant this cover is minimum, i.e. there exist no Boolean covers corresponding to this encoding with fewer than five implicants. (This can be proven experimentally by running an exact minimization algorithm [MCCL56], like that implemented by program ESPRESSO-II with the"exact" flag [RUDE85a].)

We question now to what extent symbolic design guarantees the minimality[2] of the Boolean cover, obtained by replacing the words by their corresponding binary encoding.

Example 4.4: Consider this other Boolean encoding of the words, in which we changed only the encoding of the output symbols:

INDEX = 00	*AND* = 00	*CNTA* = 00
DIR = 01	*OR* = 01	*CNTB* = 01
IND = 11	*ADD* = 10	*CNTC* = 10
	JMP = 11	*CNTD* = 11

Then the function can be represented by a Boolean cover as:

```
00  **  00
01  0*  01
11  00  01
11  *1  11
*1  10  10
01  11  10
```

Note that the first Boolean implicant can be deleted, because its output part is 00. However note that this cover is not minimum: there exist now a minimum Boolean cover corresponding to this encoding with three implicants and that can be computed from the above one by a standard minimization technique. Namely:

```
*1  0*  01
*1  1*  10
11  *1  11
```

Note that the first cover has two pairs of Boolean implicants with 01 and 10 in the third field and that are merged into two single implicants in the minimum cover. This is possible

[2] The optimality of a Boolean cover is measured by its cardinality, i.e. by the number of its implicants. A Boolean cover of a function is **minimum**, if there exist no cover of that function having a smaller cardinality. A Boolean cover of a function is **minimal**, if its cardinality is minimum with regard to some local criterion. Usually a Boolean cover of a function is said to be minimal, if no proper subset is a cover of the same function [BRAY84b].

because the third implicant of the minimum cover has 11 in the third field, and 11 covers 01 and 10.

The reason for this additional reduction is in the covering relations among the encoded output symbols. Note that when we optimized the symbolic table by disjoint minimization, our goal was only to group the input symbols corresponding to each output symbol independently. The relations among the output symbols were neglected. For this reason it is important to exploit the relations among the output symbols at the symbolic level. **Symbolic minimization**, formally defined in Section 5, is a technique that determines an optimal ordering of the output symbols. This ordering is related to the covering relations among the binary encodings of the output symbols, and is responsible for the additional reduction of the table size, as shown by Example 4.4.

Example 4.5: Consider the following table:

INDEX	*AND OR ADD JMP*	*CNTA*
DIR IND	*AND OR*	*CNTB*
DIR IND	*ADD JMP*	*CNTC*
IND	*JMP OR*	*CNTD*

Here, an ordering relation is assumed that allows control *CNTD* to override control *CNTB* and *CNTC* when both are specified. The table, together with this ordering relation, is an equivalent representation of the function specified by Example 4.1. It is an example of the result of a symbolic minimization. Note that this table can be transformed into the Boolean cover of Example 4.4 by replacing each symbol by its encoding. Moreover, the encoding of *CNTD* covers bit-wise the encoding of *CNTB* and *CNTC* and allows *CNTD* to override *CNTB* and *CNTC*. Note that the first implicant can be deleted, by assuming that *CNTA* is the default output and that the encoding of *CNTA* is 00, which is covered bit-wise by the encoding of all other outputs.

Once a minimal table has been found, the encoding of the words into Boolean variables is driven by the grouping of the input symbols (group of symbols appear on the same row of a minimal table) and the ordering of the output symbols generated by symbolic minimization. Note that disjoint minimization deals with each output symbol independently, and therefore does not provide information for an encoding of the output symbols that optimize the table size. Therefore disjoint minimization can be used only to solve problem P1 of symbolic design.

We describe in detail in Section 6 how to compute an encoding of the input and output symbols that is compatible with a minimal table. Such an encoding allows to transform the minimal symbolic cover into a Boolean representation with as many product-terms as the minimal symbolic cover. Even though this mapping is not sufficient to imply the minimality of the Boolean cover, the encoded cover can be considered a good solution to the problem. It is important to remark that the length of the encoding (i.e. the number of binary variables) needed to encode each symbol may have to be larger than the minimum length required to distinguish all the symbols in each field (i.e. the ceiling of the logarithm in base 2 of the number of elements in each field). Therefore it is interesting to compute minimal length encodings compatible with a minimal symbolic representation and to trade off possibly the minimality of the number of rows in a table for the number of bits required to encode each field.

Let us now consider the design problem P4. A finite state machine, implementing a sequential circuit, can be implemented by feeding back the (some of the) outputs of a combinational circuit to its inputs, possibly through a register (Fig. 4.2). Finite state machines are generally represented by state tables. State tables consist of four fields related to the primary inputs/outputs and to the present/next state representation. Each field may be partitioned into sub-fields. Optimal state assignment can be solved by symbolic design by minimizing the state table using symbolic minimization and by computing a state encoding compatible with the minimal table [DEMI86a]. The feedback path makes this problem different from designing combinational units. In particular, the state symbols appear in both an input and an output column of the state table and must be encoded consistently: the set of state symbols must be encoded while satisfying the group and the ordering constraints simultaneously. The limitations and the encoding procedure is described in Section 6.

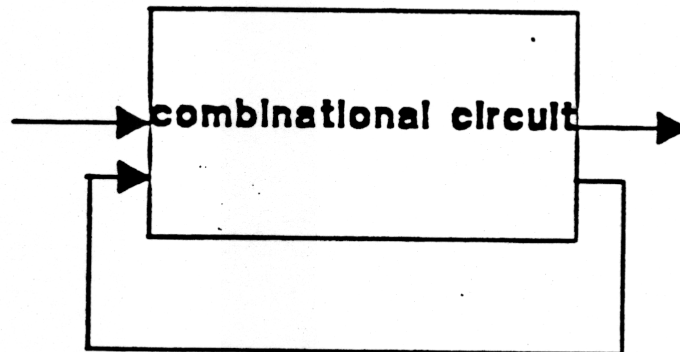

Fig 4.2 Sequential circuit.

STATE	OP-CODE	MODE	NEXT-STATE	CONTROL-SIGNALS
I1			I2	00000000000000111
I2			A1	00000000111011000
A1	JMP	DIRECT	I1	00000100000100100
A1	SRJ	DIRECT	A3	00000000001000001
A1	SAC	DIRECT	A4	00000010000000000
A1	ISZ	DIRECT	A4	00000000000000000
A1	LAC	DIRECT	A4	00000000000000000
A1	AND	DIRECT	A4	00000000000000000
A1	ADD	DIRECT	A4	00000000000000000
A1	JMP	INDIRECT	A2	00000000000000000
A1	SRJ	INDIRECT	A2	00000000001000001
A1	SAC	INDIRECT	A2	00000000000000000
A1	ISZ	INDIRECT	A2	00000000000000000
A1	LAC	INDIRECT	A2	00000000000000000
A1	AND	INDIRECT	A2	00000000000000000
A1	ADD	INDIRECT	A2	00000000000000000
A1	JMP	INDEXED	I1	00001101000100100
A1	SRJ	INDEXED	A2	00000000001000001
A1	SAC	INDEXED	A3	00001101000100000
A1	ISZ	INDEXED	A3	00001101000100000
A1	LAC	INDEXED	A3	00001101000100000
A1	AND	INDEXED	A3	00001101000100000
A1	ADD	INDEXED	A3	00001101000100000
A2		DIRECT	A3	00000001010001000
A2		INDIRECT	A3	00000001010001000
A2		INDEXED	A3	00001101000100100
A3	JMP		I1	00010101000000100
A3	SRJ	DIRECT	A4	00010110000000110
A3	SRJ	INDIRECT	A4	00010110000000110
A3	SRJ	INDEXED	A4	00000010000000111
A3	SAC		A4	00000010000000000
A3	ISZ		A4	00000000000000000
A3	LAC		A4	00000000000000000
A3	AND		A4	00000000000000000
A3	ADD		A4	00000000000000000
A4	JMP		E1	00000001010000001
A4	SRJ		I1	00000001000000001
A4	SAC		I1	00000001000000001
A4	ISZ		E1	00000000010000000
A4	LAC		E1	00000001010000001
A4	AND		E1	00000001010000001
A4	ADD		E1	00000001010000001
E1	LAC		I1	00110100000000000
E1	AND		I1	01000100000000000
E1	ADD		I1	00100100000000000
E1	ISZ		E2	00010100001000010
E2	ISZ		E3	10010110000000010
E3			I1	00000001000000101

Fig 4.3 State table of control unit.

Example 4.6: We consider here the finite state machine implementing the control unit of the microprocessor design described by Langdon in Chapter 5 of [LANG82]. The state table of Fig. 4.3 describes the memory-reference instructions only. The control-unit has nine states, corresponding to instruction-fetch, operand-address evaluation and instruction execution. The states are labeled by mnemonic strings, namely: I1, I2, A1, A2, A3, A4, E1, E2, E3. Seven operations are considered, namely: JMP (jump), SRJ (subroutine jump), SAC (store accumulator), ISZ (increase and skip on 0), LAC (load accumulator), AND (and), ADD (add). Three modes of memory addressing are considered: DIRECT, INDIRECT and INDEXED. The operation and addressing mode are specified by two instruction fields. Each row of the table shows an action as a consequence of particular conditions. There are five fields in each row. Two fields correspond to the present and next control states. Two fields correspond to the primary inputs, i.e. the operation code and addressing mode. The last field corresponds to the control signals. For the sake of clarity in Fig. 4.3, an unspecified field in the table corresponds to a *don't care* condition, i.e. to the specification of all possible words in the field. For example, the first row shows that when the control-unit is in state I1, the state of the control-unit at the next cycle is I2 and control signal 0000000000000111 (incrementing the program-counter) is issued. The third row shows that when the control unit is in state A1, the op-code is JMP and the addressing mode is DIRECT, then the next control-state is I1 and the control signal is 00000100000100100.

Eventually symbolic design can be applied to interconnected logic circuits. Consider two units, to be implemented by two-level logic macros, that communicate through a bus (Fig.4.4). If the representation of the information is transmitted across the bus is irrelevant to the design, symbolic optimization can be used as follows. The transmitting unit can be represented by a table with a symbolic output field and the receiving unit by another table with a symbolic input field. The tables corresponding to both units are optimized by symbolic minimization and the set of symbols, representing the communication signals, can be encoded as in the previous cases. Needless to say, this method can be extended to the interconnection among any number of modules, implementing combinational or sequential functions.

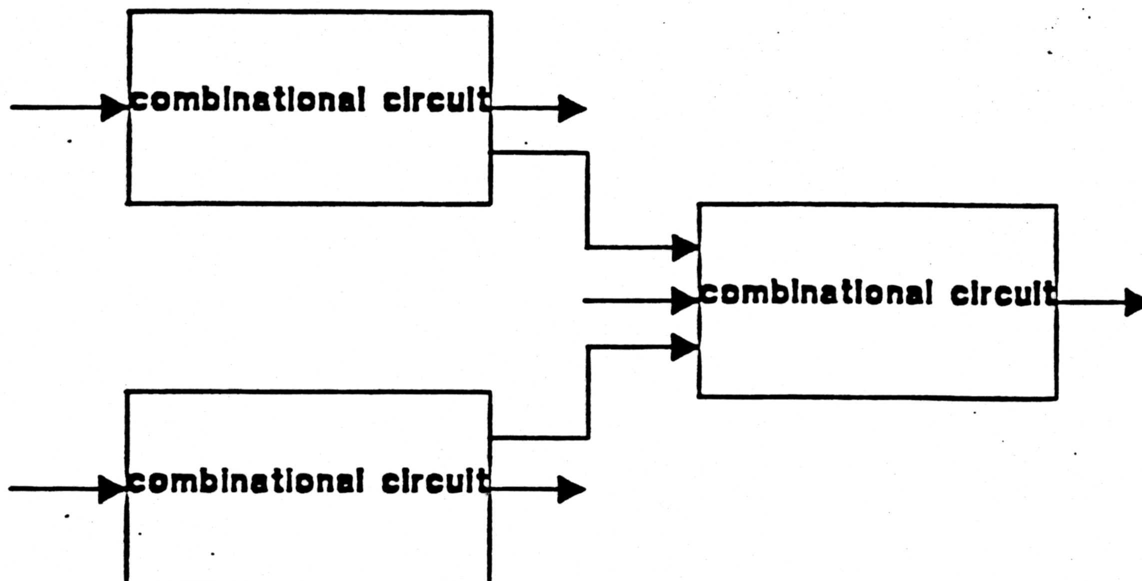

Fig 4.4 Interconnected logic circuits.

5. SYMBOLIC MINIMIZATION

5.1 *Definitions*

Symbolic functions are switching functions whose variables can take a finite set of values. Each value is represented by a word (or mnemonic), i.e. by a string of characters. A symbolic variable s has a set of admissible values S. The symbol ϕ is reserved to denote that variable s does not take any value of S. For functions of n input variables and m output variables, let $S_i^I, i = 1, 2, \ldots, n$ and $S_i^O, i = 1, 2, \ldots, m$ be the set of admissible values for the corresponding variables s_i^I and s_i^O. Then the domain of the symbolic function is the Cartesian product $S^I \equiv S_1^I \times S_2^I \times \cdots \times S_n^I$ and the range is the Cartesian product $S^O \equiv S_1^O \times S_2^O \times \cdots \times S_m^O$. A generic element of the domain is denoted by s^I and one of the range by s^O.

A completely specified symbolic function of n input variables and m output variables is a function $f: S^I \rightarrow S^O$, that maps each element of the domain to an element of the range. An incompletely specified symbolic function is a function having the property that, for some inputs, some output variables can take any value in the corresponding range. The collection of these points of the domain is called the *don't care* set of that particular output variable.

> **Example 5.1:** The truth table of Example 4.1 is a representation of a completely specified symbolic function with $n = 2$ inputs and $m = 1$ outputs. Here, $S_1^I = \{DIR, IND, INDEX\}$; $S_2^I = \{AND, OR, ADD, JMP\}$; $S^O = \{CNTA, CNTB, CNTC, CNTD\}$.

Boolean or binary-valued functions are symbolic functions, whose variables can take the values $S = \{0, 1\}$. The domain is $\{0, 1\}^n$. The range of a completely specified function is $\{0, 1\}^m$. For incompletely specified functions, let the symbol $*$ represent the *don't care* condition. Then the range is $\{0, 1, *\}^m$. Similarly, the variables of multiple-valued functions can take the values $S = \{0, 1, \ldots, p-1\}$, where p is the radix of the representation [RINE77] [HURS84]. Algebras have been developed for both the Boolean and the multiple-valued [POST21] representations. The representation of the result of Boolean operations are based on a linear order of S. Let $r: S \rightarrow N$ be an enumeration consistent with the linear order [PREP73], where N is the set of natural numbers. Then:

i) Product (AND) : $s \wedge s' \equiv r^{-1} \min(r(s), r(s'))$
ii) Sum (OR) : $s \vee s' \equiv r^{-1} \max(r(s), r(s'))$
iii) Complement (NOT) : $\bar{s} \equiv r^{-1} (p - 1 - r(s))$

Note that the order does not affect the semantics of the representation of a switching function; however the order may strongly affect the size of the representation. For example, canonical representations, such as *sum-of-products* or *product-of-sums* depend on the linear order of S : in particular the minimal representations of a Boolean function and of its complement as *sum of products* have different sizes, and algorithms have been developed to exploit this fact [SASA83].

Representations of symbolic functions depend on the definitions of the operations among words. Unfortunately, no order relation is meaningful ***a priori*** among the elements of a symbolic description. For this reason, operations on symbolic representations are related to order relations among words and appropriate order relations are introduced to obtain convenient representations of symbolic functions.

In the following presentation, single-output functions are considered first, i.e. $m = 1$, to simplify the notations. The extension to multiple-output functions is then shown.

Symbolic functions are represented here in a particular canonical notation: *sum of products* or more exactly *sum of products of symbolic literal functions*. Let S be the set of admissible values for a variable s. A symbolic literal is a non-empty subset $\sigma \subseteq S$. For any variable $s \in S$, the symbolic literal function is defined as follows:

$$l(s, \sigma) \equiv \begin{cases} \text{TRUE if } s \in \sigma \\ \text{FALSE else} \end{cases}$$

Example 5.2: Consider the set $S_1^I = \{DIR, IND, INDEX\}$ corresponding to the set of admissible values of the first input variable in Example 5.1. An example of a symbolic literals is *DIR IND*. The corresponding literal function is TRUE for either $s = DIR$ or $s = IND$.

By using a *sum of product of symbolic literal functions* representation, only the order in S^O affects the representation, because the literal function maps words into the pair of values (TRUE,FALSE) independently of the order in S^I. Note that if a linear order relation is applied on the range, symbolic function representations in this canonical form are equivalent to multiple-valued logic function representations [RINE77] in the same form. In particular, a multiple-valued representation can be obtained from a symbolic representation by interchanging each symbol s with $r(s)$, where $r(\bullet)$ is the appropriate enumeration.

Since an order in S^O is not necessarily given, the definitions of the representation of a symbolic function are compatible with a set, possibly empty, of partial order relations among the elements of the range. Let $R = \{(s, s'); s, s' \in S^O\}$ be a partial order on S^O. We say that s covers s' if either $s = s'$ or $s' = \phi$ or $(s, s') \in TR$, where TR is the transitive closure of R [AHO74]. The symbolic sum of two words s and s' is well-defined only if a covering relation exists among the elements involved. In particular:

$$s \vee s' = \begin{cases} s & \text{if } s \text{ covers } s' \\ s' & \text{if } s' \text{ covers } s \end{cases}$$

Else the symbolic sum is ambiguous or ill-defined.

A **symbolic product-term** (or symbolic product) of literals is the $n + 1$-tuple $(\sigma_1, \ldots, \sigma_n, \tau)$, where $\sigma_i \subseteq S_i^I$, $i = 1, 2, \ldots, n$; $\tau \in S^O$. The word τ is called the **output-part** of the literal. A **symbolic product** $p(s^I, \tau)$ of **literal functions** $l_i(s_i^I, \sigma_i)$, $i = 1, 2, \ldots, n$ is a function:

$$p(s^I, \tau) = \begin{cases} \tau \text{ if } l_i(s_i^I, \sigma_i) = \text{TRUE}, \ \forall i = 1, 2, \ldots, n \\ \phi \text{ else} \end{cases}$$

Example 5.3: A symbolic product of the function in Example 2.1 is:

DIR AND CNTB

The symbolic product function takes the value *CNTB* when the two inputs take the values *DIR* and *AND* respectively.

Two products p_1, p_2 intersect ($p_1 \cap p_2 \neq \phi$) if $\exists s^I \in S^I$ such that $p_1(s^I, \tau_1) \neq \phi$ and $p_2(s^I, \tau_2) \neq \phi$. Two sets of products, P_1 and P_2 intersect ($P_1 \cap P_2 \neq \phi$) if $\exists p_1 \in P_1$ and $\exists p_2 \in P_2$ such that p_1 and p_2 intersect. Two products are **output-disjoint** if either they do not intersect or they have the same output-part, i.e. $p_1(s^I, \tau_1)$ intersects $p_2(s^I, \tau_2)$ implies $\tau_1 = \tau_2$. A set of products is output-disjoint if the products are pair-wise output-disjoint.

Example 5.4: Consider the two symbolic products:

DIR IND	*ADD*		*CNTC*
DIR	*ADD JMP*		*CNTC*

The two symbolic product intersect because, for the symbolic input $s^I = DIR\ ADD$, the corresponding symbolic product functions specify a symbolic value. Since both symbolic product functions take value *CNTC*, the product terms are output-disjoint.

A symbolic function can be represented in a *sum of product* form, if $\forall s^I \in S^I$ for which the function is specified, the operation of symbolic sum among products is well-defined. In particular, such a representation always exists in the following two cases: i) for any linear order on S^O; ii) if the representation is a sum of pair-wise output-disjoint products. In the former case symbolic sum is always well-defined because a covering relation is defined between each pair of symbols in S^O. In the latter, only symbolic sum of identical values is required.

Sum of product representations are conveniently represented in tabular forms, as a stack of product-terms. A **symbolic implicant** is a symbolic product $p(s^I, \tau)$ such that $\forall s^I \in S^I$ for which the symbolic function is specified, $f(s^I)$ covers $p(s^I, \tau)$. A **symbolic cover** of a symbolic function is a set of implicants $P = \{p_1, p_2, \ldots, p_{|P|}\}$ whose sum is $f(s^I)$, $\forall s^I \in S^I$ for which the symbolic function is specified. Since symbolic sum depends on the order R on S^O, we denote a symbolic cover by the pair $C(P, R)$. The **cardinality** of a symbolic cover is $|P|$ and depends on R. A **minimum** symbolic cover of a symbolic function is a cover of minimum cardinality. A **minimal** (local minimum) symbolic cover of a symbolic function is a cover such that no proper subset is a cover of the same function.

Example 5.5: The following table is a symbolic cover of the function specified in Example 5.1.

INDEX	*AND OR ADD JMP*	*CNTA*
DIR	*AND OR*	*CNTB*
IND	*AND*	*CNTB*
IND	*OR JMP*	*CNTD*
DIR IND	*ADD*	*CNTC*
DIR	*JMP*	*CNTC*

Note that the product-terms are pair-wise output-disjoint. Therefore this representation is output-disjoint and is compatible with any set R of partial order relations on S^O, and in particular the empty set. The following table is another symbolic cover of the function specified in Example 5.1.

INDEX	*AND OR ADD JMP*	*CNTA*
DIR IND	*AND OR*	*CNTB*
DIR IND	*ADD JMP*	*CNTC*
IND	*JMP OR*	*CNTD*

Here, $R = \{(CNTD, CNTB);(CNTD, CNTC)\}$. Note that the fourth product-term has an intersection with the second and third one and these products are not output-disjoint. By choosing this particular partial order, the cover cardinality is reduced by two. Moreover, note that the first implicant can be removed by assuming that *CNTA* is the default output, as pointed out in Example 5.5.

5.2 Symbolic minimization

Symbolic minimization is a procedure that attempts to determine a symbolic cover of a symbolic function in a minimum number of product-terms. Finding a minimum symbolic cover is a difficult task. An analysis of the computational complexity of the problem has not been done yet. However we conjecture that any method to find a minimum cover should involve the solution of a covering problem, which is a NP-complete problem [GARE78]. Therefore heuristic algorithms are used to determine a minimal (local minimum) solution. It is important to remark that recent progress in heuristic logic minimization has led to techniques which very often yield minimum solutions in the binary [HONG74] [BRAY84b] and multiple-valued [RUDE85a] case. We assume that the reader is familiar with heuristic logic minimization [HONG74], [BRAY84b]. Since most of the routines in the symbolic minimization algorithm are based on logic minimizer ESPRESSO-II, we refer the reader to [BRAY84b] for details.

Symbolic minimization is achieved by an iterative improvement of the initial cover $C^0(P^0, R^0)$. The symbolic function is described as input by the set of products P^0, while R^0 is an empty set of ordering relations, because no order relation is meaningful ***a priori*** among the elements of a symbolic description. Therefore P^0 is always a set of output-disjoint products. The main idea of symbolic minimization is to generate the order R during the minimization process. The symbolic minimizer detects partial order relations that are necessary to define sums of product-terms which would decrease the symbolic cover cardinality. As a result the order relations are determined ***a posteriori*** by the minimizer. The output of the minimizer is a minimal cover $C(P, R)$.

Symbolic minimization is a very complex technique and is fully documented in [DEMI86a]. To give a flavor of this technique, we present here a simplified version of the main loop of the symbolic minimization algorithm, which applies to symbolic functions with one symbolic output only. In this perspective, symbolic minimization is achieved by an iterative loop, that uses a multiple-valued-input, binary-valued output minimization procedure. This procedure can be regarded as a black box, that takes as input the representation of a multiple-valued-input function and its *don't care* set and returns a minimal representation. Computer programs ESPRESSO-II [BRAY84b], ESPRESSO-MV [RUDE85a] and MINI [HONG74] can be used in this regard.

It is convenient to represent the partial order R by a directed acyclic graph $G(V, A)$, where the vertex set V is in one-to-one correspondence with S^o. The edge set A is initialized empty and is constructed during the minimization process. An edge between two vertices defines a order relation between the corresponding elements of S^o. Therefore the sum of two distinct elements of S^o is well-defined if there is a directed path between the corresponding vertices.

Let us arbitrarily label the elements of the range: $S^o = \{s_i^o; i = 1, 2, \dots, q\}$. Let $ON_i, i = 1, 2, \dots, q$ be the subset of the initial set of product-terms P^0 consisting of the product-terms

whose output part $\tau = s_i^O$. Note that the set ON_i does not intersect the set ON_j if $i \neq j$, because P^O is output-disjoint. Each set ON_i specifies a symbolic single-output single-valued function. The original symbolic function can be seen as a collection of q multiple-valued-input, binary-valued-output functions whose *on* set corresponds to the points of the domain mapped into s_i^O, whose *off* set corresponds to those points mapped into s_j^O, $j \neq i$, and whose *don't care* set corresponds to the unspecified points [BRAY84b]. A representation of each set ON_i by a minimal number of product-terms, denoted here by M_i, can be obtained $\forall i = 1, 2, \dots, q$ by using a multiple-valued-input, binary-valued-output minimization technique. In principle, by performing q minimizations in this way, a minimal cover of the original function can be computed as $P = \cup_{i=1}^{q} M_i$, i.e. as a collection of the q minimal covers M_i. It is shown in [BRAY84b] how to perform the q minimizations simultaneously. This procedure is called here **disjoint minimization**, because the minimal cover P of any completely specified function is output-disjoint.[3] Disjoint minimization does not exploit the benefit of choosing an order R to minimize the cover, and therefore is a weak optimization technique.

The main idea of symbolic minimization is that the cover P is not constrained to be output-disjoint, by introducing appropriate order relations among the elements of S^O. For example, suppose that $(s_j^O, s_i^O) \in R$. Then any point of the domain represented by ON_j can be used to reduce the cardinality of ON_i in the minimization process. In the minimal symbolic representation, such point is still mapped into s_j^O because $(s_j^O, s_i^O) \in R$. In other words, the subset of the domain represented by ON_j is a part of the *don't care* set while minimizing ON_i. We represent the *don't care* set by the set of product-terms DC_i In this case, $ON_j \subseteq DC_i$.

Example 5.6: Consider the first cover of Example 5.5. Let $s_i^O = CNTB$ and $s_j^O = CNTD$. Then, ON_i is:

DIR	*AND OR*	*CNTB*
IND	*AND*	*CNTB*

Suppose $(CNTD, CNTB) \in R$. Then DC_i includes:

IND	*OR JMP*	*CNTD*

Therefore the point of the domain $s^I = IND \;\; OR$ can be used to reduce the cardinality of ON_i, that can be represented as:

DIR IND	*AND OR*	*CNTB*

To minimize ON_i, an explicit representation of the corresponding *don't care* set is needed. Equivalently, the *off* set can be specified and the *don't care* set obtained by complementation of the *on* and *off* sets. To take advantage of the order relations, we use a definition of the *off* set different from that used in [BRAY84b] and mentioned before. For our purposes, the *off* set corresponding to ON_i is the subset of S^I that is mapped by the function f to a value different than s_i^O and covered by s_i^O, because $\forall s^I \in S^I$ s.t. $f(s^I) \neq s_i^O$ and $f(s^I)$ is covered by s_i^O, $p(s^I, s_i^O)$ is not an implicant of the function. If $G(V, A)$ represents the partial order, then the *off* sets can be defined as a set of product-terms: $OFF_i = \cup_J ON_j$; $J = \{ j \text{ s.t.} \exists \text{ a path from } v_i \text{ to } v_j \text{ in } G(V, A) \}$.

[3] If the original function is incompletely specified, the minimal cover P is still output-disjoint if we restrict the definition of intersection among implicants by considering S^I as the *care* set of the function.

At each iteration of the symbolic minimization loop, M_i is obtained by minimizing ON_i, using a routine that performs multiple-valued-input binary-valued output minimization. We invoke the minimization routine with the pair (ON_i, OFF_i), so that the corresponding *don't care* set DC_i, computed by the minimizer by complementation, includes by construction all the sets ON_j for which no path exists in $G(V,A)$ from v_i to v_j. As a result, minimization may be very efficient in reducing the cardinality of ON_i because of the particularly advantageous *don't care* set. If M_i intersects ON_j, the relation (s_j^o, s_i^o) is recorded, by adding (v_j, v_i) to the edge set of the graph.

SYMBOLIC MINIMIZATION LOOP

Data ON_i, $i = 1, 2, \dots, q$;
Data $G(V,A)$;
$A = \phi$; $P = \phi$;
for (k = 1 to q) {
    i = select (k) ;
    $OFF_i = \cup_j ON_j$; $J = \{j \mid \exists$ a path from v_i to v_j in $G(V,A)\}$;
    M_i = minimize (ON_i, OFF_i) ;
    $A = A \cup \{(v_j, v_i)$ s.t. $M_i \cap ON_j \neq \phi\}$;
    $P = P \cup M_i$;
};

Procedure select sorts the sets ON_i according to a heuristic criterion. Procedure **minimize** is a call to a multiple-valued-input binary-valued-output minimizer. The algorithm generates a set of symbolic products $P = \cup_{i=1}^{q} M_i$ and the directed graph $G(V,A)$. It is proven in [DEMI86a] that the graph $G(V,A)$ generated by the symbolic minimization loop is acyclic (i.e. R represents a partial order relation on S^o) and that $C(P,R)$ is a minimal cover of the original symbolic function represented by $C^o(P^o, R^o)$.

The order R depends on the heuristic sorting of the sets ON_i, done by procedure select. As a result, the cardinality of the cover $C(P,R)$ generated by the algorithm strongly depends on this routine.

The simplified algorithm described above invokes q times the minimization procedure. Its computational complexity grows linearly with the number of elements in the range of the function. The minimization procedure is a heuristic procedure: no theoretical bounds on the computational complexity have been proven for heuristic minimization; however experimental results have shown that it is practical to minimize a wide range of logic functions [HONG74] [BRAY85] and therefore the algorithm can be used in this perspective.

Several heuristics have been tried. Note that as more edges are added to the graph, it is more likely that the *off* sets become large and the *don't care* sets small. An effective heuristic is to sort the sets ON_i in descending order of cardinality, so that the largest sets will benefit from large *don't care* sets. A key ingredient for an effective reduction of the cover minimality is that the graph should be kept as sparse as possible by introducing only the ordering relations needed to reduce the cover cardinality. Keeping the graph sparse corresponds to keep many degrees of freedom to order S^o in the later iterations of the algorithm. Note that if **minimize** is a "standard" minimization algorithm, it aims at reducing both the product and literal cardinality in each ON_i. Therefore, a local optimization of the number of literals in the **minimize** procedure may introduce new edges in the graph and reduce the

likelihood of reducing the symbolic cover cardinality at a later step. For these reason, the symbolic minimization loop has to be modified for efficiency, by tuning the **minimize** routine to the symbolic minimization problem. To obtain an efficient algorithm for symbolic minimization it is necessary to "open the black box" and examine more carefully the operations that procedure **minimize** performs. We refer the interested reader to [DEMI86a] for details.

6. ENCODING PROBLEMS AND ALGORITHMS

6.1. Encoding problems

Symbolic minimization is used as an intermediate step in solving problems P1-P4 of Section 1. Since the final result must be a binary-valued logic circuit implementation, the symbolic representation has to be translated into a binary-valued (Boolean) one. If a multiple-valued circuit implementation technology were available (including logic gates implementing the literal function [RINE77]), then the (minimal) symbolic representation could be mapped into a multiple-valued representation with the same cardinality by interchanging the words s with $r(s)$, where $r(\cdot)$ is an appropriate enumeration. If a partial order relation exists on a set of words, then the enumeration must be consistent with it.

Let us consider first problems P1-P3. The goal of the following encoding technique is to find a binary-valued *sum of products* representation of the switching function with as many product-terms as the (minimal) symbolic representation. To construct such a Boolean cover, it is sufficient to determine: i) an encoding of the words related to each symbolic input variable such that each symbolic implicant can be represented by one Boolean implicant; ii) an encoding of the words related to each symbolic output variable that preserves the covering relations, i.e. such that the encoding of the sum of any subset of symbolic products is the sum of the corresponding Boolean products. For this reason we consider two encoding problems: the former is related to the encoding of the symbols representing the input variables; the latter to the encoding of the output variables.

Let us consider first the encoding of the input variables. If a variable can take at most two symbolic values, it has a trivial Boolean encoding. In the general case, a variable that can take more than two symbolic values is represented by more than one binary-valued variables. Then, to achieve the goal of encoding each symbolic implicant by one Boolean implicant, we must represent each symbolic literal by one product of Boolean literals. A product of Boolean literals is called **cube** or **face**, because it is a subspace of the Boolean hyperspace that can be represented by a hypercube. Therefore the encoding of the words must be such that each symbolic literal can be represented by a face (Boolean cube) that is a subspace of the Boolean space that contains the encoding of all and only the symbols in the literal [DEMI85a]. If *don't care* words are specified in that literal, than it is indifferent whether the face representing the literal contains the encoding of the *don't care* words or not.

The problem of encoding the words related to the output variables is different, because the output-part of the symbolic implicants corresponding to an output variable consists of one word only (and not of a symbolic literal with possibly more than one word, as in the case of the input variables). However, the encoding of the words related to the output variables must be such that the covering relations are preserved while transforming the symbolic cover into a Boolean cover. Therefore the encoding must be such that, for any two words joined by an order relation, the corresponding encoding are linked by a covering relation, i.e. the first word covers bit-wise the second word.

The encoding problem derived from problem P4 has an additional constraint. In this case, there is one set (ore more sets) of words corresponding to both input and output variables. The encoding of this set of words must satisfy the requirements for the encoding of the input variables and the output variables simultaneously.

We present now formally the encoding problems. Let S be a set of words to be encoded and let $n_s = |S|$. Let n_p be the cardinality of the (minimal) symbolic cover. Let n_b the encoding length, i.e. the number of Boolean variables used to represent S. The encoding problem is studied using matrix notation. Some matrices we consider have pseudo-Boolean entries from the set: $\{0, 1, *, \phi\}$ where * represents the *don't care* condition (i.e. either 1 or 0) and ϕ represents the empty value (i.e. neither 1 nor 0). Logical product and sum on pseudo-Boolean variables is defined as follows:

$\wedge$	0	1	*	ϕ
0	0	ϕ	0	ϕ
1	ϕ	1	1	ϕ
*	0	1	*	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ

$\vee$	0	1	*	ϕ
0	0	*	*	0
1	*	1	*	1
*	*	*	*	*
ϕ	0	1	*	ϕ

Let S be the set of values taken by a symbolic input variable. Let us consider the set of literals in the (minimal) symbolic cover related to that variable. The **word-literal incidence matrix** A (or incidence matrix in short) is a matrix: $A \in \{0, 1, *\}^{n_p \times n_s}$

$$A = \begin{bmatrix} a_{1\bullet} \\ a_{2\bullet} \\ \cdots \\ a_{n_p\bullet} \end{bmatrix} = \left[a_{\bullet 1} \,|\, a_{\bullet 2} \,|\, \cdots \,|\, a_{\bullet n_s} \right] = \{a_{ij}\} \text{ where: } a_{ij} = \begin{cases} 1 & \text{if word } j \text{ belongs to literal } i \\ * & \text{if word } j \text{ is a } \textit{don t care} \text{ word in literal } i \\ 0 & \text{else} \end{cases}$$

where a *don't care* word in a literal is a word that may appear or not in a literal without affecting the function representation [DEMI86a].

Example 6.1: Let S be the set of operation-codes in the symbolic function specified in Examples 4.1 and 5.5, i.e. $S = \{AND, OR, ADD, JMP\}$. Consider the minimal symbolic cover of Example 5.5. Then:

$$A = \begin{bmatrix} 1111 \\ 1100 \\ 0011 \\ 0101 \end{bmatrix}$$

Let now S be the set of values taken by a symbolic output variable. The **partial order adjacency matrix** $B \in \{0, 1\}^{n_s \times n_s}$ (or adjacency matrix in short) is the adjacency matrix of the graph representing the transitive closure of the partial order R. If word i covers word j, then $b_{ij} = 1$. If word i covers word j and word j covers word k, then $b_{ij} = 1$, $b_{jk} = 1$ and $b_{ik} = 1$. Since covering is a transitive relation, we represent directly all the implied covering relations by matrix B. Moreover, since it is trivial that each word covers itself,we choose not to represent it by convention, i.e. $b_{ii} = 0$, $i = 1, 2, \ldots, n_s$.

Example 6.2: Let S be the set of controls in the symbolic function specified in Examples 4.1 and 5.5. Consider the minimal symbolic cover of Example 5.5. Then:

$$B = \begin{bmatrix} 0000 \\ 0000 \\ 0000 \\ 0110 \end{bmatrix}$$

The encoding matrix E is a matrix $E \in \{0, 1\}^{n_s \times n_b}$

$$E = \begin{bmatrix} e_{1\bullet} \\ e_{2\bullet} \\ \dots \\ e_{n_s \bullet} \end{bmatrix} = \left[e_{\bullet 1} \,|\, e_{\bullet 2} \,|\, \dots \,|\, e_{\bullet n_b} \right]$$

whose rows are the encodings of the words.

Definition 6.1: Let $a \in \{0, 1, *, \phi\}$ and $x \in \{0, 1, *, \phi\}$. The selection of x according to a is:

$$a \bullet x = \begin{matrix} x & \text{if } a = 1 \\ \phi & \text{else} \end{matrix}$$

Selection can be extended to two dimensional arrays and is similar to matrix multiplication.

Definition 6.2: Let $A \in \{0, 1, *, \phi\}^{p \times q}$ and $X \in \{0, 1, *, \phi\}^{q \times r}$. The matrix pseudo-Boolean selection is:

$$A \bullet X = C = \{c_{ij}\}^{p \times r}$$

where: $c_{ij} \equiv \bigvee_{k=1}^{q} a_{ik} \bullet x_{kj}$ or equivalently $c_{ij} \equiv a_{i1} \bullet x_{1j} \vee a_{i2} \bullet x_{2j} \vee \dots \vee a_{iq} \bullet x_{qj}$.

Let us consider the problem of encoding the symbolic input variables first. This problem was presented in [DEMI85a] for the first time. We report here the most relevant results in a more general formulation. We represent the encoding of the symbolic literals by the face matrix $F \in \{0, 1, *, \phi\}^{n_p \times n_b}$

$$F = \begin{bmatrix} f_{1\bullet} \\ f_{2\bullet} \\ \dots \\ f_{n_l \bullet} \end{bmatrix}$$

Each row of F is a face of the n_b-dimensional Boolean hypercube and corresponds to the face that encodes the symbolic literal. The face matrix can be obtained by performing the matrix pseudo-Boolean selection of E according to an incidence matrix A:

$$F = A \bullet E$$

Example 6.3: Consider the incidence matrix of Example 6.1 and the encoding of Example 4.4. Then:

$$E = \begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix} \qquad F = A \bullet E = \begin{bmatrix} ** \\ 0* \\ 1* \\ *1 \end{bmatrix}$$

Let now $\bar{A} = \{\bar{a}_{ij}\}$ where $\bar{a}_{ij} = 1$ if $a_{ij} = 0$; else $\bar{a}_{ij} = 0$. Then $\bar{F}^i \equiv \bar{a}_{\cdot i} \bullet e_{i\cdot}$ is a matrix whose rows are: i) the encoding of word i, if word i neither belongs to the symbolic literal j nor is a *don't care* word; ii) empty values. An encoding matrix E is said to satisfy the **input constraint relation** for a given incidence matrix A if:

$$\bar{F}^i \wedge F \equiv \begin{bmatrix} \bar{f}^i_{1\cdot} \wedge f_{1\cdot} \\ \bar{f}^i_{2\cdot} \wedge f_{2\cdot} \\ \cdots \\ \bar{f}^i_{n_l\cdot} \wedge f_{n_l\cdot} \end{bmatrix} = \Phi \quad \forall i = 1, 2, \dots, n_s$$

where Φ is the empty matrix, i.e. a matrix whose rows have at least one ϕ entry and therefore representing no point in the Boolean space.

Example 6.4: The encoding matrix of Example 6.3 satisfies the input constraint relation. However if we swap the first two rows of E, the input constraint relation is no longer satisfied, because the encoding of the third word, ADD, intersects the fourth face, or equivalently:

$$\bar{F}^3 \wedge F = (\bar{a}_{\cdot 3} \bullet e_{3\cdot}) \wedge (A \bullet E) = \left(\begin{bmatrix}0\\1\\0\\1\end{bmatrix} \bullet [10]\right) \wedge \left(\begin{bmatrix}1111\\1100\\0011\\0101\end{bmatrix} \bullet \begin{bmatrix}01\\00\\10\\11\end{bmatrix}\right) =$$

$$= \begin{bmatrix}\phi\phi\\10\\\phi\phi\\10\end{bmatrix} \wedge \begin{bmatrix}**\\0*\\1***\end{bmatrix} = \begin{bmatrix}\phi\phi\\\phi 0\\\phi\phi\\10\end{bmatrix} \neq \Phi$$

The problem of encoding the values taken by a symbolic input variable is equivalent to finding an encoding matrix satisfying the input constraint relation. An optimal solution is one of minimal encoding length. Therefore we can state:

Encoding problem E1: Given an incidence matrix A, find an encoding matrix E with minimal number of columns that satisfies the input constraint relation.

Let us consider now the problem of encoding the output variables.

Definition 6.3: Let $A \in \{0, 1\}^{p \times q}$ and $X \in \{0, 1\}^{q \times r}$. The **matrix Boolean selection** is:

$$A \circ X = C = \{c_{ij}\}^{p \times r}$$

where: $c_{ij} \equiv \bigvee_{k=1}^{q} a_{ik} \wedge x_{kj}$ and the sum ($\vee$) and product ($\wedge$) operators on Boolean variables have the usual meaning:

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

Let now $G = B \circ E$. Row $i, i = 1, 2, \dots, n_s$ of matrix G is the logical sum of the encoding of the words that must be covered by the encoding of word i. Therefore we say that a matrix E satisfies the output constraint relation for a given adjacency matrix B, if E covers G or equivalently:

$$\bar{E} \wedge G = O$$

where $\bar{E}$ is the Boolean complement of E and O is the matrix of 0 entries.

In this case, the problem of encoding the values taken by a symbolic output variable is equivalent to finding an encoding matrix satisfying the output constraint relation. An optimal solution is one of minimal encoding length. Therefore we can state:

> Encoding problem E2: Given an adjacency matrix B representing to a partial order, find an encoding matrix E with minimal number of columns that satisfies the output constraint relation.

The solution of problem P1 (P2 or P3) requires the solution of encoding problem E1 (E2 or both) for each symbolic variable, after symbolic minimization. The solution of problem P4 requires the encoding of one set (or more sets) of words corresponding to both input and output variables, after symbolic minimization.

> Encoding problem E3: Given an incidence matrix A and an adjacency matrix B representing a partial order, find an encoding matrix E with minimal number of columns that satisfies both the input and the output constraint relation.

The existence of solutions to the encoding problems E1, E2 and E3 was shown in [DEMI85a] and [DEMI86a]. In particular, there are always encoding that satisfy either the input or the output constraint relation and that can be derived from matrices A and B. However, in general, these encodings are not solutions to problems E1 and E2, because E does not have a minimal number of colums n_b. Problem E3 admits a solution under a restrictive assumption on matrices A and B [DEMI86a]. Therefore, there exist cases in which some constraints have to be relaxed to be able to find a solution [DEMI86A].

6.2. Encoding algorithms

A solution to the encoding problems E1, E2 or E3 is an encoding of minimal length the satisfies the input, output or both constraint relations. Unfortunately, these are computationally complex problems of combinatorial optimization and it is not known whether an optimal solution can be computed by non-enumerative procedures. Since the growth of computation-time as the size of the problem increases is a practical limitation to computer-aided design systems, we consider here heuristic algorithms for the solution of the above problems. Experimental results show that the encodings constructed by these algorithms have reasonably short length, and often equal to the minimum length solution when this is known.

The heuristic techniques presented below solves the encoding problems using greedy strategies. An encoding matrix E is grown from an initial seed matrix by concatenating rows and/or columns. At each step, the best local concatenation is computed, while keeping the previously computed encod-

ing matrix as such. As a result the encoding matrix grows in size, until all the rows are encodings for the words in S that satisfy the constraint relations. The heuristic selections that drive the algorithm attempt to minimize the steps that increase the number of columns to obtain a solution, and therefore guarantee a weak optimality.

There are two major approaches to constructing matrix E: a row-based method and a column-based one. In the former method, the encoding matrix is constructed row by row, i.e. by computing the encoding of the words one at a time [DEMI85a]. In the latter approach, the encoding matrix is constructed column by column, i.e. by computing one bit of the encoding of all the words at each pass. This idea was introduced first by Dolotta [DOLO64], to solve the optimal state assignment problem, though the encoding problem was stated differently.

A row-based encoding method was presented in [DEMI85a], to solve problem E1. This technique is also effective for solving problem E2. The algorithm can be sketched as follows:

STEP 1: Select a word not yet encoded.
STEP 2: Determine the encodings for that word satisfying the constraint relation restricted to that word and to the previously encoded words.
STEP 3: If no encoding is found, increase the code dimension by adding a column to E and go to STEP 2.
STEP 4: Assign an encoding to the selected word by adding a row to E.
STEP 5: If all words have been encoded, stop. Else go to STEP 1.

We refer the reader to [DEMI85a] for the details of the algorithm when applied to problem E1 and to [DEMI86a] for problem E2. The row-based encoding algorithms generate short encodings (see [DEMI85a] for experimental results) but fail to be effective for large examples. In particular, the candidate encodings generated at STEP 2 may increase exponentially with the encoding length n_b. Therefore, when the encoding length increases, the computer implementation of the algorithm slows down considerably. Moreover, it is complex to handle problem E3 with this approach, because the effectiveness of the algorithm depends heavily on the heuristic ordering of words at STEP 1, and for both input and output constraints there may exist conflicting orderings.

The need to handle both constraints simultaneously, as well as a desired computational-time complexity linear in n_b, has lead to the development of a column-based algorithm. In a column-based algorithm a single-bit encoding of all the words is introduced at each step. While there always exist single-bit encodings of all the words that satisfy the output constraint relation, it is unlikely that such an encoding satisfies the input constraint relation. Therefore we say that, given an incidence matrix A, an encoding matrix E partially satisfies the input constraint relation, if E satisfies the input constraint relation for A', where A' is a subset of the rows of A. The satisfaction ratio is $\frac{n_{p'}}{n_p}$, where $n_{p'}$ is the row cardinality of A'. Then, the column-based encoding algorithm can be sketched as follows:

STEP 1: Select a column vector in $\{0, 1\}^{n_s}$ that satisfies the output constraint relation and corresponding to a maximal satisfaction ratio.
STEP 2: If at the first pass, let E be the selected vector. Else, append the selected column vector to E.
STEP 3: If E satisfies the input constraint relation, stop. Else go to STEP 1.

The algorithm is described in detail in [DEMI86a]. We mention some properties of the encoding problems that are relevant to understand the strategy of this method. If each column of E satisfies the output constraint relation, so does the entire matrix E and *vice versa*. This is not true for the input constraint relation. In particular, if E satisfies the input constraint relation, then a subset of columns of E may not satisfy it. However if E satisfies the input constraint relation for A', then by appending a column to E, the augmented matrix satisfies the input constraint relation for A' as well. In other words, the satisfaction ratio cannot decrease.

Therefore the strategy of the algorithm is to select columns that increase the satisfaction ratio until it reaches unity and the algorithm terminates. The selection of the column to be appended to E is the crucial part of the algorithm, because the goal is to construct a matrix E with minimal number of columns. It is shown in [DEMI86a] that when the algorithm is used to solve problem E1, by selecting as columns the transpose of the rows of A, we construct as a solution $E = A^T$, which is a valid solution if we replace the * entries by 0's or 1's. In other words, there exist a trivial column selection that increases the satisfaction ratio by at least $\frac{1}{n_p}$, i.e. one column is needed to satisfy the input constraint set by each row of A. On the other hand, if the algorithm is used to solve problem E3, there exists also a column selection that increases the satisfaction ratio by at least $\frac{1}{2 \times n_p}$, i.e. at most two columns will be needed to satisfy the input constraint set by each row of A, while satisfying any admissible output constraint. However, since the optimality of the solution is measured by the number of columns of E, we need column selections that maximize the increase of the satisfaction ratio. For this reason, the column selection routines looks for maximal subsets of input constraints (rows of A) that can be satisfied by one column selection. This corresponds to finding a column that maximizes the satisfaction ratio. If an upper bound n_{b_max} on the encoding length is specified, then it is imperative that the rows of E are different from each other after n_{b_max} column assignments. Suppose there are groups of identical rows in E with at most cardinality $2^{(n_{b_max} - n_b)}$, after having assigned n_b columns. The column selection is done such that, if there are groups of identical rows in $[E \mid e]$, their cardinality is most $2^{(n_{b_max} - n_b - 1)}$, where e is the column being appended. This requirement restricts the selection of the column vector e. We refer the interested reader to [DEMI86a] for the details.

The column-based encoding algorithm can be summarized as follows in a qualitative way. The algorithm constructs an encoding matrix E. If problem E2 or E3 have to be solved, each column is chosen as to satisfy the output constraint relation. Therefore so does the entire matrix E. If problem E1 or E3 have to be solved, each column is chosen as to satisfy partially the input constraint relation, or equivalently to satisfy the input constraint relation for a subset of the rows of A. These rows are then discarded from further consideration. After a finite number of steps, matrix E satisfies the input constraint relation. The heuristic selection of the column attempts to increase maximally the satisfaction ratio at each step, and therefore guarantees a weak minimality of the column cardinality of E. If an upper bound on the encoding length is specified, then the encoding may satisfy only partially the constraint relations, but attempts to satisfy most of the constraints.

The construction of matrix E with bounded column cardinality allows to trade off the minimality of the cover cardinality for that of the encoding length. In particular, given an encoding of length n_b, it is possible to determine the cover cardinality (or a bound on the cover cardinality n_p) on the basis of the satisfied constraints. Therefore, for a particular switching function, it is possible to determine a set of pairs of parameters (n_b, n_p) that relate to the size of the implementation.

As a final remark, we would like to compare the column-based encoding algorithm with the one proposed by Dolotta [DOLO64] and later perfected by Weiner [WEIN67b], Torng [TORN68] and Story [STOR72]. These algorithms addressed the state assignment problem for finite state machines and the mechanism of the encoding algorithm is similar to the one presented here. However the selection of the columns was based on a heuristic criterion: in particular a scoring function was used to select a column on the basis of the likelihood that logic minimization, which would have followed the encoding, could reduce the two-level cover cardinality. (Story's method optimized the number of and-or inputs for each column choice.) In our approach we relate each column assignment to the satisfaction of some input constraints and therefore to a known reduction of the cover cardinality. Therefore column selection is related to cover minimality in a deterministic way. However our encoding technique is still a heuristic one, because the greedy strategy considers and assigns only one column at a time.

6.3. Implementation of the symbolic method and experimental results

The symbolic minimization algorithm has been implemented in a computer program called CAPPUCCINO, because it is based on the logic minimizer ESPRESSO-II. CAPPUCCINO is written in APL and incorporates a modified version of the ESPRESSO-II original program [BRAY84b]. CAPPUCCINO implements the symbolic minimization algorithm described in [DEMI86a] and sketched in Section 5.2. Table 6.1 summarizes the results:

RESULTS OF SYMBOLIC MINIMIZATION			
Example	Original cover cardinality	Minimal cover cardinality	Minimal disjoint-cover cardinality
EX1	24	9	16
EX2	91	49	57
EX3	170	78	78
EX4	11	5	8
EX5	25	10	11
EX6	24	17	24
EX7	115	89	94
EX8	107	57	92
EX9	184	106	115
EX10	16	14	15
EX11	166	102	111
EX12	49	11	12
EX13	25	10	11
EX14	20	8	13
EX15	56	23	24
EX16	32	15	16
EX17	108	46	55
EX18	32	17	18
EX19	14	9	10
EX20	30	22	23

Table 6.1

The first two numeric columns show the original and final cover cardinality. The last column shows the final cardinality obtained by disjoint-minimization, by using program ESPRESSO-II. In some cases (EX1, EX2, EX8, EX17, ...) CAPPUCCINO does significantly better than ESPRESSO-II in reducing the cover cardinality. In some others, the advantage of introducing covering relations among the output symbols is not a major factor in reducing the cover cardinality. These comparisons have to be understood with caution, because we are comparing different minimization techniques and not program performances.

Computing time ranges from few seconds to about 20 minutes for the largest example on an IBM 3081 computer. Note that APL is interpreted and therefore the execution is much slower with comparison to compiled code programs. Today, CAPPUCCINO is limited to covers of about 2000 symbolic product-terms, due to memory limitations of the APL workspace and computing time. A compiled code implementation of the algorithm based on the data structure and the routines of program ESPRESSO-MV [RUDE85a] (in place of the APL version of ESPRESSO-II) would definitely increase the capability and the performance of the program.

The column-based encoding algorithm has been implemented in program CREAM. CREAM is written in APL and consists of about 25 functions. CREAM is designed to be used with CAPPUCCINO: it takes the representation of a minimal symbolic cover and generates an encoding that can replace the symbolic entries. The final result is a Boolean cover that can be implemented as a PLA (after having been folded or partitioned, if desired), or used as a starting point for multiple-level synthesis, as done by the YLE program in the Yorktown Silicon Compiler [BRAY85a].

CREAM solves the encoding problem E1, E2 or E3 at user's request. It accepts an upper bound on the number of columns to be used in the encoding. For a given bound and the corresponding encoding of all the symbolic fields, it is possible to estimate the area taken by a PLA implementation. Therefore, it is possible to estimate the area as well as the aspect-ratio. This computation can be done for different bounds on the encoding length and therefore a designer can choose among several implementations with different areas and aspect ratios.

CAPPUCCINO and CREAM have been tested on several examples. Some results are reported in the Table 6 .2.

OVERALL RESULTS						
Example	n_i	n_s	n_o	n_p	$n_{p'}$	n_b
EX1	2	6	2	24	11	4
EX2	7	16	7	91	49	8
EX3	1	9	2	170	78	12
EX4	2	4	1	11	6	2
EX5	2	9	1	25	10	5
EX6	1	12	1	24	17	7
EX7	7	48	19	115	89	10
EX8	8	20	6	107	68	7
EX9	11	32	9	184	107	9
EX10	1	8	1	16	14	4
EX11	9	30	10	166	103	12
EX12	4	4	4	49	11	3
EX13	2	11	1	25	10	6
EX14	4	5	1	20	8	4
EX15	8	7	5	56	23	5
EX16	8	4	5	32	15	4
EX17	4	27	3	108	49	11
EX18	4	8	3	32	17	4
EX19	2	7	2	14	9	3
EX20	4	15	3	30	22	7

Table 6.2

The examples are sequential circuits, i.e. we are solving problem P4 by symbolic minimization first and by solving encoding problem E3 after. The symbolic representations have four fields corresponding to the primary inputs/outputs (which are represented by Boolean variables) and the present/next states (which are represented by symbolic variables). The first four numeric columns represent the parameters of the function: n_i is the number of primary inputs, n_s is the number of states, n_o is the number of primary outputs and n_p is the cardinality of the initial symbolic cover. The last two columns show the final cardinality of the Boolean cover and the state encoding length. Note that for a few examples, (as EX1 , EX4 , ...) the Boolean cover cardinality is larger than the minimal symbolic cardinality (reported in the third column of Table 6.1), because it was not possible to satisfy all constraints in the encoding. Note also that a further reduction in cardinality may be obtained by minimizing again the Boolean covers that are not minimal. The computing time is in the order of few seconds for all these examples, on an IBM 3081 computer.

In Table 6.3 we try to evaluate the optimality of the encoding, in terms of encoding length.

ENCODING–LENGTH COMPARISONS					
Example	n_s	$\log_2 n_s$	n_b	$n_{b'}$	$n_{b''}$
EX1	6	3	4	3	3
EX2	16	4	8	7	7
EX3	9	4	12	12	12
EX4	4	2	2	2	2
EX5	9	4	5	4	4
EX6	12	4	7	4	6
EX7	48	6	10	8	8
EX8	20	5	7	6	7
EX9	32	5	9	7	7
EX10	8	3	4	4	3
EX11	30	5	12	9	9
EX12	4	2	3	3	3
EX13	11	4	6	4	4
EX14	5	3	4	3	3
EX15	7	3	5	5	5
EX16	4	2	4	4	4
EX17	27	5	11	9	7
EX18	8	3	4	4	4
EX19	7	3	3	3	3
EX20	15	4	7	7	5

Table 6.3

The second column represents the number of words to be encoded n_s ; the third the minimal number of bits needed to encode the words regardless of any constraint on their encoding (i.e. *ceiling*($\log_2 n_s$)); the fourth column the encoding length as constructed by CREAM as a solution to problem E3. These three columns show that the encoding length computed by CREAM is close enough to the minimum length: for most of the examples *ceiling*($\log_2 n_s) \leq n_b \leq 2 \times$ *ceiling* ($\log_2 n_s$) . For some examples (as EX14, EX16, EX18, EX19, ...) it is possible to prove that no encoding of length inferior to n_b can be a solution of the encoding problem. The results of CREAM can be compared with those obtained by program KISS [DEMI85a], which uses a row-based encoding algorithm. The length of a solution to problem E1 computed by CREAM is given in the fifth column of the table and the length of an encoding constructed by KISS in the last column. (The algorithm of KISS can solve only problem E1.) Note that for some examples, the row-based algorithm gives a shorter encoding. However, the corresponding implementation requires more product-terms, as shown by the fourth column of Table 6.1.

7. CONCLUSIONS

We have reviewed some relevant problems in the area of computer-aided synthesis of circuits for control units of VLSI processors. Several algorithms and CAD programs have been developed for the logic and physical design task. However, circuit synthesis from behavioral-level specifications is still an exploratory area for CAD algorithms. The problems arising from optimizing the logic representation are mostly computationally intractable. Heuristic methods have been developed, but they still lack the capability of handling efficiently large designs.

The symbolic design methodology has been presented. It may be applied to the synthesis of interconnected combinational and/or sequential circuits. In particular, it can be used to find good solutions to the optimal state assignment problem. Symbolic minimization allows encoding-independent optimization of switching functions represented by tables of symbols and the encoding algorithms construct a binary representation of the symbols that translate the minimal symbolic cover into a compatible Boolean cover.

Exploratory work in the field of logic design of control circuits needs to address many unresolved problems. First, today's optimization methods must be extended to multiple-level logic representations and a theoretical framework for multiple-level synthesis must be defined. In particular, logic and symbolic minimization must be extended to multiple-level circuits. Then, the relations among different tasks of logic design must be understood and exploited, with particular reference to: i) the partitioning problem of a control unit into functional blocks; ii) the state minimization problem; iii) the selection of the type of registers to store the control-state information; iv) the optimal encoding and minimization of two-level and multiple-level switching functions. Eventually, an efficient CAD system must allow the designer to explore the trade-off among circuit structures, to find the best match between circuit architecture and implementation technology.

The task that CAD scientists are facing for improving computer design methods is very challenging: up to now we have only scratched the surface of a massive body of difficult problems.

8. REFERENCES

[AGER76] T. Agerwala "Microprogram Optimization: a Survey" IEEE Trans on Comput., vol. C-25, pp. 962-973, oct 1976.

[AHO74] A.V.Aho J.E.Hopcroft and J.D.Ullman , "The Design and Analysis of Computer Algorithms", Addison Wesley, 1974.

[ANDR80] M. Andrews, Principle of Firmware Engineering in Microprogram Control, Computer Science Press, 1980.

[BLAC85] R. Blackburn and D.Thomas, "Linking the Behavioral and Structural Domains of Representations in a Synthesis System", Proc. Des. Autom. Conf., Las Vegas, pp. 374-380, Jun 1985.

[BREN84] N.Brenner "The Yorktown Logic Language: an APL-like Design Language for VLSI Specifications" Int. Conf. on Circ. and Comp. Des., Rye, NY, pp.11-15, Sep 1984.

[BRAY84b] R.Brayton, G.D.Hachtel, C.McMullen and A.L.Sangiovanni- Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[BRAY84d] R.Brayton, C.L.Chen, C.McMullen R.Otten and Y.Yamour "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", Proc. Cust. Integr. Circ. Conf., Rochester, NY, pp. 346-350, may 1984.

[BRAY85a] R.Brayton, N.Brenner, C.Chen, G.De Micheli, C.McMullen and R. Otten "The Yorktown Silicon Compiler" Proc. Int. Symp. on Circuit and Systems, Kioto, Japan, pp. 391-394, Jun 1985.

[BRAY85c] R.Brayton, C.Chen, G.De Micheli, J.Katzenelson C.McMullen R. Otten and R.Rudell "A Microprocessor Design Using the Yorktown Silicon Compiler" Proc. Int. Conf. on Circuit and Comput. Des., Rye, N.Y., pp. 225-230, Oct 1985.

[CAMP85] R.Camposano, "Synthesis Techniques for Digital System Design", Proc. Des. Autom. Conf., Las Vegas, pp. 475-480, Jun 1985.

[CURT69] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 1", IEEE Trans on Comput., vol. C-18, pp. 1121-1127, dec. 1969.

[CURT70] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 2", IEEE Trans on Comput., vol. C-19, pp. 66-73, jan. 1970.

[DARR84] J.Darringer, D. Brand, J.Gerbi, W. Joyner and L.Trevillan, "LSS: A System for Production Logic Synthesis", IBM Journal of Research and Development, Vol. 28, No.5 pp.537-545, Sept. 1984.

[DAVI72] P. Davies "Readings in Microprogramming", IBM Jour. of Res. and Dev., vol. 11, No. 1, pp. 16-40, jan 1972.

[DEMI84a] G.De Micheli, M.Hofmann, A.Newton and A.L.Sangiovanni Vincentelli, "A Design System for PLA-based Digital Circuits" in "Advances in Computer-Aided Engineering Design", Jay Press, 1985.

[DEMI84c] G.De Micheli, R.Brayton and A.L.Sangiovanni Vincentelli, "KISS: a Program for Optimal State Assignment of Finite State Machines", Proc. ICCAD Santa Clara, Nov 1984.

[DEMI85a] G.De Micheli, R.Brayton and A.L.Sangiovanni Vincentelli, "Optimal State Assignment for Finite State Machines", IEEE Transactions on CAD/ICAS, Vol CAD-4, No. 3, pp.269-284, July 1985.

[DEMI85c] G. De Micheli, "Symbolic Minimization of Logic Functions", Int. Conf. on Comp. Aid. Des., Santa Clara, pp. 293-295, Nov. 1985.

[DEMI86a] G.De Micheli, "Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-level Logic Macros", IEEE Transactions on CAD/ICAS, Oct 1986 (in print) and IBM Research Report RC 11672.

[DOLO64] T.A.Dolotta and E.J.McCluskey, " The coding of internal states of sequential machines", IEEE Trans. Elect. Comp., vol EC-13, pp. 549-562, Oct. 1964.

[GARE78] M.R.Garey and D.S.Johnson, "Computers and Intractability", W.H.Freeman and Company, San Francisco, 1978.

[GOLD85] A.Goldberg, S.Hirshhorn and K.Lieberherr, "Approaches toward Silicon Compilation" IEEE Circuits and Devices, Vol.1, No.3 , pp. 29-39, May 1985.

[GRAS70] A.Grasselli and U.Montanari, "On the minimization of read-only memories in microprogrammed digital computers" IEEE Trans. on Comput. pp. 1111-1114 Nov. 1970.

[HADS85] R.Hadsell, "Micro/370" in "Microarchitectures of VLSI Computers", Proceedings NATO ASI Series E, No. 96, Martinus Nijhoff, The Netherlands, 1985.

[HART66] J.Hartmanis and R.E.Stearns, "Algebraic Structure Theory of Sequential Machines", Prentice Hall, 1966.

[HILL81] F.Hill and G.Peterson, "Introduction to Switching Theory and Logical Design", Wiley, 1981.

[HONG74] S.J.Hong,R.G.Cain and D.L.Ostapko, "MINI: a Heuristic Approach for Logic Minimization", IBM Jour. of Res. and Dev., vol. 18, pp. 443-458, Sep. 1974.

[HOPC71] J.Hopcroft, "An n log n Algorithm for Minimizing States in a Finite Automaton", in "Theory of Machines and Computation", (Kohavi editor), pp.189-196, Academic Press, 1971.

[HURS84] S.K.Hurst, "Multiple-Valed Logic- Its Status and its Future" IEEE Trans on Comput., vol. C-33, No 12, pp. 1160-1179, Dec 1984.

[KOWA85] T. Kowalski and D.Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base" Proc. Des. Autom. Conf., Las Vegas, pp. 252-258, Jun 1985.

[LEWI81] T. Lewis and B. Shriver, "Introduction to Special Issue on Microprogramming", IEEE Trans on Comput., vol. C-30, pp. 457-459, jul 1981.

[LANG82] G.Langdon, "Computer Design", Computech Press, 1982.

[MCCL56] E.J.McCluskey, "Minimization of Boolean Functions", Bell Laborat. Techn. Jour., vol. 35, pp. 1417-1444, Apr. 1956.

[NICH65] A.Nichols and A.Bernstein, "State Assignemnt in Combinational Networks", IEEE Trans on Elect. Comp., vol. EC-14, pp. 343-349, Jun. 1965.

[PAPA79] C. Papachristou, "A Scheme for Implementing Microprogram Addressing with Programmable Logic Arrays", Digital Processes No. 5 pp. 235-256 may 1979.

[PLEE73] C.Pleeger, "State Reduction of Incompletely Specified Finite State Machines", IEEE Trans. on Comput. pp. 1099-1102, 1973.

[POST21] E.L.Post, "Introduction to a General Theory of Elementary Propositions" Amer. J. Math., vol. 43, pp. 163-185, 1921.

[PREP73] F.Preparata and R.Yeh, "Introduction to Discrete Structures", Addison Weseley, 1973.

[REUS86] B.Reusch and W.Merzenich, "Minimal Coverings for Incompletely Specified Sequential Machines", Acta Informatica, No.22, pp. 663-678, 1986.

[RINE77] D.Rine, "Computer Science and Multiple-Valued Logic", North Holland, 1977.

[RUDE85a] R. Rudell and A. Sangiovanni-Vincentelli, "ESPRESSO-MV: Algorithms for Multivlued Logic Minimization" Proc. Custom Int. Circ. Conf., Portland, Oregon, May 85

[RUDE85b] R. Rudell, A. Sangiovanni-Vincentelli and G.De Micheli, "A FSM Synthesis System", Proc. Int. Symp. on Circ. and Sys., pp.647-650, Kyoto, Japan, Jun 85.

[SASA83] T.Sasao "Input Variable Assignment and Output Phase Assignment of PLAs", IBM Research Report, No. 1003, June 1983.

[SCHW68] S.Schwartz, "An algorithm for minimizing read-only memories for machine control" Proc. IEEE Symp. on Switch. and Autom. Theory, pp. 28-33, 1968.

[STOR72] J.R.Story H.J.Harrison and E.A.Reinhard, "Optimum State Assignment for Synchronous Sequential Circuits", IEEE Trans. on Comp., vol. C-21 pp. 1365-1373, Dec. 1972.

[SISK82] J.Siskind, J.Southard and K.Crouch, "Generating High Performance VLSI Designs from Succint Algorithmic Descritions", Proc Conf. on Adv. Res. on VLSI, pp. 28-40, 1982.v. 1970.

[SU72] S.Y.H.Su and P.T.Cheung, "Computer Minimization of Multi-Valued Switching Functions", IEEE Trans on Comput., vol 21, pp. 995-1003, 1972.

[TORN68] H.C.Torng, "An Algorithm for Finding Secondary Assignments of Synchronous Sequential Circuits", IEEE Trans on Comput., vol. C-17 pp. 416-469, May 1968.

[TRED81] N.Tredennick, "How to Flowchart for Hardware", IEEE Computer, pp. 87-102, Dec 1981.

[WEIN67b] P.Weiner and E.J.Smith, "Optimization of Reduced Dependencies for Synchronous Sequential Machines", IEEE Trans on Elect. Comp., vol. EC-16, pp. 835-847, Dec. 1967.