INTERNATIONAL JOURNAL OF CIRCUIT THEORY AND APPLICATIONS, VOL. 16, 347–369 (1988)

COMPUTER-AIDED DESIGN AND OPTIMIZATION OF CONTROL UNITS FOR VLSI PROCESSORS

GIOVANNI DE MICHELI
Computer Systems Laboratory, Stanford University, Stanford CA 94305, U.S.A.

SUMMARY

This review presents the models, methods and algorithms for synthesis and optimization of control units for VLSI processors. First, circuit structures used for control in the state-of-the-art processors are described. The control synthesis methods are then presented as well as the algorithms for optimizing the control unit representation. Among these techniques, the symbolic design methodology is described that can be applied to optimize the silicon area taken by some particular structures of control units.

1. INTRODUCTION

Very large scale integration (VLSI) processors are the heart of computers, signal and image processors and other systems that elaborate digital information. Their design is critical to the progress of the electronic industry: more powerful processors are needed every day and their design time has to shrink because of the competitiveness in the market-place. Computer-aided design (CAD) techniques are necessary ingredients for the fast design of processors which yield maximal performance with the state-of-the-art technology. In particular, circuit synthesis and optimization techniques have been effectively used to synthesize chips (or components) from high-level specifications without (or with limited) human intervention. The confidence in computer-aided circuit synthesis methods has grown tremendously in recent years.

VLSI processors are designed with different styles and technologies, according to the criticality of the performances and the projected production volume. To date, most high-performance general-purpose processors are still designed in a custom or structured custom design methodology, even though semi-custom implementations (e.g. standard cells, gate arrays or sea of gates) are being considered as viable solutions, especially in the case of application-specific integrated circuit (ASIC) processors. Structured array implementations are commonly used for control, and full custom control implementations also called hard-wired control structures, are rare. The control unit design requires careful attention. First, it must guarantee maximal performance of the data path. Secondly, it should not require excessive silicon area nor introduce critical paths. Processors may need a complex control structure that takes a significant fraction of the available chip area. The optimization of the area taken by the control section is important for several reasons, such as not to overflow the available silicon area, to increase the circuit yield, to decrease fault probability and to avoid the introduction of critical paths along the control structure due to extensive wiring.

Micro-programming has been extensively used in processor design. With the improvement in the scale of integration, the role and definition of micro-programming has also changed. We refer now to micro-programmed machines as those systems that use regularly organized storage arrays (control store) to keep a large part of the control information that is described in a high-level form by a micro-program.[2] There are several advantages in using micro-programming. The design of the control unit is flexible, it can be defined at a later stage of the processor design and it can be easily modified. Micro-programming allows a processor to emulate the instruction set of other machines. Different implementation strategies of micro-programmed processors are possible. For example the control store can be located on-chip or off-chip

0098–9886/88/040347–23$11.50

Received 30 June 1988

and/or it 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 is it correct to talk about micro-programmable processors. For practical reasons, most processors have read-only on-chip control store and therefore should be called micro-programmed. We will consider this class of processors in the sequel.

Micro-programming can be viewed as a first step towards the achievement of processors that can be automatically customized to a set of specifications described in a high-level programming language. A micro-programmed processor is explicitly partitioned into data path and control. The data path is fixed, whereas the control is programmable. A micro-compiler transforms a micro-program specified in a programming language into the personality of the control store, that is often implemented by a read only memory (ROM) or programmed logic array (PLA). Techniques for micro-code optimization[3] and simulation are used to develop efficient micro-programmed structures. Since the compilation time is negligible, it is then possible to customize quickly and automatically a control unit to a given specification. Unfortunately the data-path model cannot be programmed, or even modified, by a micro-compiler and this limits the range of applications of micro-programmable processors.

A step further in designing programmed processors is now achieved by means of *hardware compilers*, called also silicon compilers or automated synthesis systems.[1] In this case, the entire processor is described as a program which is compiled into an interconnection of circuits and eventually described by the geometries of the masks needed to fabricate the chip. Today fully automatic hardware compilation has been shown to be feasible[4] and has also proved to be a time-effective way of designing ASIC and general-purpose processors.[5] Among the main advantages of hardware compilation, we would like to stress the possibility of experimenting with architectural trade-offs to match the implementation technology. Indeed, architectural changes can be performed as quickly as the modification of a program. Hardware compilation takes several minutes of computing time for a medium-sized processor. Therefore, automatic hardware compilation opens new frontiers to processor design as well as provoking an evolution in the techniques for control synthesis. We refer the interested reader in the general aspects of hardware compilation to References 1 and 4 for further information.

In this paper we address the problem of control-unit design from high-level specifications. We describe the methods that are amenable for computer implementation and that are used in today's hardware compiler or components. In Section 2 we review some structures used for control units. Then we describe in Section 3 the two major control synthesis methodologies: control synthesis from behavioural system specifications. We review then in Section 4 the techniques used for control optimization, namely state minimization, state encoding and logic minimization. We show also how control structures can be designed using optimization techniques on symbolic representations.

2. CONTROL UNITS

The partition of a processor into data path and control is somehow artificial. In general it is assumed that a processor elaborates data by using some digital *hardware operators*, or *resources* (e.g. ALUs, shifters, etc.) according to specific instructions. The collection of the hardware operators is then called the *data path*, whereas the circuits that decode the instructions and control the hardware resources are called the *control unit*. The distinction is merely made for the designer's convenience of grouping circuits together and is not necessarily a sharp distinction. It may not correspond to a physical partition of the circuit. For example, an overflow/underflow detection circuit that generates an interrupt may be physically implemented as a part of the data path even though it performs a control function.

Control units of processors may have various degrees of complexity. A simple model of a control unit is a single *deterministic automaton* or *deterministic finite-state machine* (FSM), implementing a sequential function. Finite-state machines can be represented, using the Huffman model,[6] as an interconnection of a combinational circuit and memory (usually synchronous D-type registers), than holds the state information. The operation of the automaton can be described in terms of a state transition function and an output function which provides the signals to the control points of the data path.* In the simplest control realization, both functions may be implemented by a structured array (ROM or PLA) called control store. Otherwise the two functions may be implemented by two separate arrays, referred to as sequencing and control store respectively. Since the control store may be large, it is common to see two-stage control storage structures: a micro-ROM (or micro-PLA) that bears an encoded version of the output function which is decoded by a nano-ROM (or nano-PLA). When the control function is more elaborate, as in the case of some microprocessor designs, it is convenient to add additional structures. For example, when ROMs are used, the transition and output function are the contents of the memory at a given address. Next-state transitions to consecutive addresses need not be explicitly stored in the memory because the increment of the ROM address can be easily implemented by a counter. Therefore a control-unit model may include a loadable counter, which stores the state information and which provides an address to the sequencing and control store. When the next state has the following address, the counter is incremented. Otherwise it is loaded with the state information provided by the sequencing store. Sequencing and control store may be implemented by ROMs as well as by PLAs.[7] Other additional circuits may be useful to achieve an efficient implementation. Multiplexers may be used to implement the next control-state function following a conditional branch. The control store may be reduced by specifying common subsequences of control states, for example by using micro-subroutines in the micro-program description. An efficient implementation may take advantage of a hardware stack, to hold the return state information (micro-subroutine return address). 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 other circuits (e.g. counters, multiplexes, stacks, ...), which will be referred to as auxiliary circuits in the sequel.

The complexity of a control unit depends on the processor architecture. In the case of complex instruction set computers (CISCs), many instructions require several machine cycles and therefore the control unit embeds a micro-sequencer as well as appropriate auxiliary circuits. As an example, the IBM micro-370 processor has a complex control unit including a state machine control circuit that determines the next control store address, an 18-bit micro ROM and a 71-bit nano ROM that generate the control signals to the data path through the local PLAs. The sequencing may be altered by commands from the bus control, the branch control and the special function unit (Figure 1).[8] As another example, the control section of the DEC CVAX CPU chip is shown in Figure 2. The control store contains a ROM of 1600 words of 41 bits; 28 bits control the execution and 13 the micro-sequencer. The micro-sequencer selects the address for the control-store ROM from either the instruction box, current micro-instruction, test logic or from logic within the micro-sequencer. An eight-entry address stack is used to implement microcode subroutines and the exception retry mechanism.[9]

Much simpler control structures are used by reduced instruction set computers (RISCs), that execute most (or all) instructions in a single machine cycle. Therefore, the task of the control unit is mainly related to the decoding of the instructions and to the activation of the appropriate control signals to the required hardware operators in the data path. In a first approximation the control unit may be thought of as a combinational instruction decoder. Indeed, most control functions are done by combinational circuits (usually PLAs). However, complications arise when some instructions (such as memory references) require more than a single machine cycle or in the case that the processor is pipelined. Therefore global processor control must sometimes, incorporate sequential functions even in the case of RISC processors. For example, the MIPS-X processor,[10] a 20 MIPS peak processor designed at Stanford University, uses mainly combinational-logic control implemented by PLAs located above the part of the data-path being controlled. The choice of local control minimizes the wiring of control signals. Global control in MIPS-X is done with the aid of two sequential circuits that handle exceptions and cache misses. As another example, the original IBM 801 architecture was thought of in terms of single-cycle execution of all the

In the limiting case that the control function is combinational in nature, the transition function and the registers are not implemented.

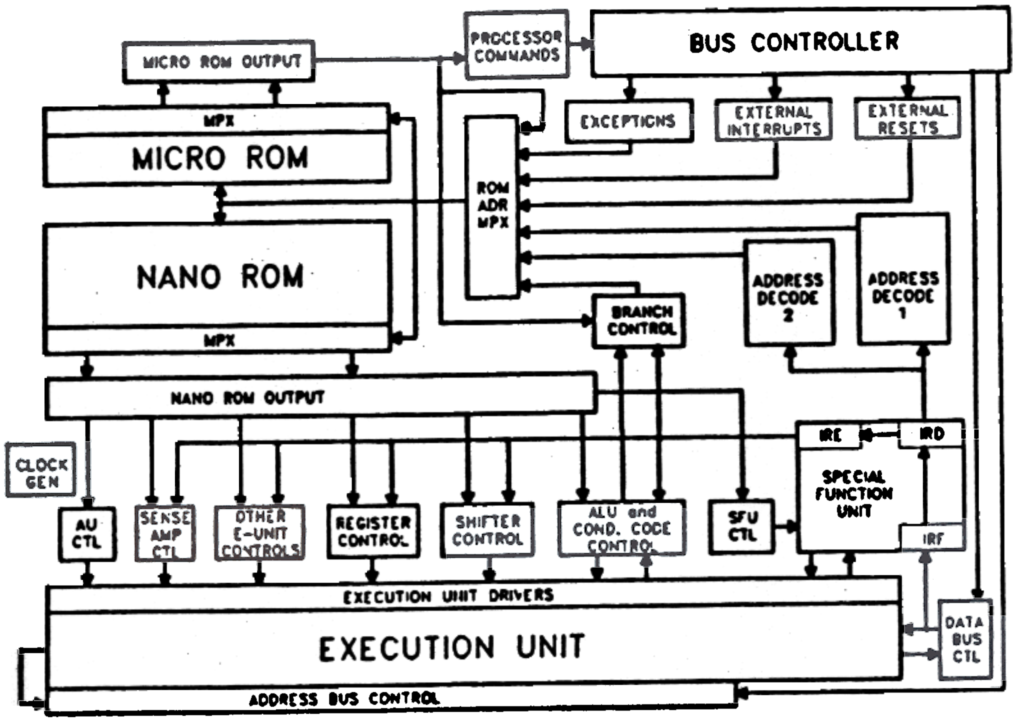


Figure 1. Block diagram of the μ-370 processor[8]

instructions and an asynchronous external memory reference mechanism. The control portion of the 801 implementation presented in Reference 5 consisted of combinational circuits, except for a small sequential circuit used to control the pending memory references. However, the IBM ROMP processor, which is the engineered version of the 801 architecture for work-station applications, is a RISC processor that supports some multiple-cycle instructions for the sake of improving the processor throughput. Its control unit is sequential.

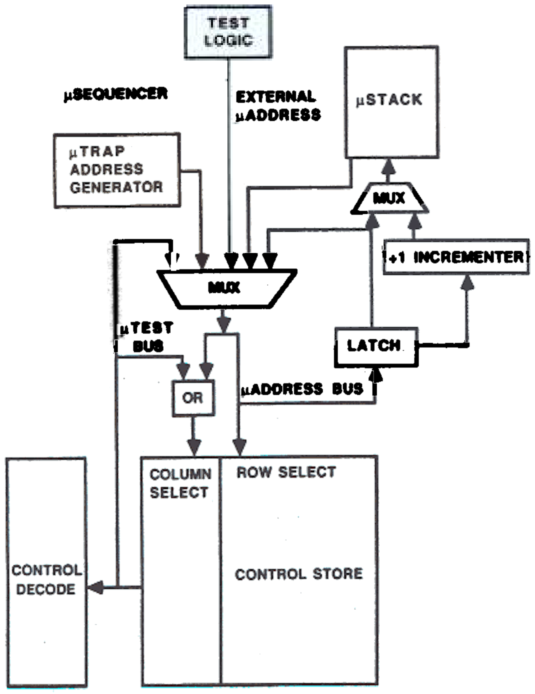


Figure 2. Block diagram of the DEC CVAX CPU[9]

3. DESIGN REPRESENTATION AND CONTROL SYNTHESIS

The high-level descriptions of digital systems may have different flavours.[11] They may be based on programming languages with imperative or declarative semantics, or use graphical inputs in terms of flow diagrams, forms or schematics.[4] It is possible to group the high-level descriptions into two main categories: *behavioural* and *structural* specifications. A behavioural specification describes a digital system in terms of its I/O response. A structural specification defines a system as an interconnection of *modules*. Each of these modules may be defined similarly in terms of its behaviour or its structure. Hierarchical specifications are often convenient ways of specifying digital systems. They may be purely behavioural or structural or a mixture of both.

A structural representation of a control unit defines explicitly the interconnections and the specifications of its components in terms of a tabular format or equivalent description (e.g. assembly-level micro-code). A behavioural representation of control, instead, describes the set of control signals to the data path for each instruction (e.g. a micro-program in a high-level micro-programming language). A behavioural representation of the entire hardware system may also imply the control information. We call *control synthesis* the set of transformations from a behavioural representation to a structural one. The advantages of automatic control synthesis are the simplification of the input description, the shortening of the design time and the higher quality of the generated structural description of control (object micro-code), because of the use of sophisticated optimization algorithms during synthesis.

There are two major cases of control synthesis techniques used for processor design: synthesis from a micro-program description and synthesis from behavioural specifications of the entire hardware system. In the first case it is assumed that the system being designed is partitioned into data path and control and that the micro-program describes the behaviour of the control unit. Several techniques have been used in the past for the design of commercial processors.[3,12,13] In the second case, the behavioural description of the hardware system is expressed in terms of its I/O response. Therefore the internal block specification of the control unit is invisible. The goal of control synthesis is then to extract the control function and map it into a structural specification.

3.1. Control synthesis of micro-programmable processors

The control unit behaviour is described by a micro-program and control synthesis is achieved by micro-compilers. The task of the micro-compiler is to map the micro-program onto the personality of the control store, usually implemented by a ROM, a PLA or an interconnection of ROMs and PLAs.[14,15] In particular, when the control store is a ROM or a PLA, the target of micro-compilation is a set of minterms or implicants. Optimization techniques may be used at various levels to reduce the storage requirement. Methods used in optimizing compilers, such as code motion, dead-code elimination and detection of possible concurrency of micro-operations are used to minimize the number of control steps, with few or no assumptions on the storage implementaton model. For this case, data-dependency graphs have been used as an underlying model for micro-code compaction.[16]

Other control optimization techniques rely on the implementation model. For a ROM-based implementation, the numbers of words and bits are the parameters that affect the storage size. Partitioning schemes are used when the number of words overflows the ROM capacity. Often, the sequencing and control store are implemented by separate arrays. The number of words may be further reduced by using counters and stacks, as shown in Section 2. The number of bits in the ROM can be reduced by encoding techniques. Consider the implementation of the control store (excluding sequencing) by a single ROM. There is usually a large number of output control signals.

There are two major approaches to the encoding of the control signals. In the former scheme, one bit is dedicated to each resource, i.e. n bits are used when n resources have to be controlled. In this case, a micro-word can activate any subset of the n resources and therefore arbitrary concurrency may be specified by a single word. Since using n bits results often in a wide-word ROM organization, such an encoding is called *horizontal*. In the latter scheme, a minimal number of bits is used to encode the resource controls, i.e. the ROM has as many bits as the ceiling of $\log_2 n$. In this case, a word may not specify simultaneous

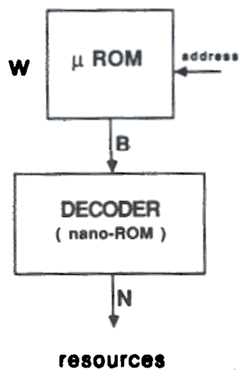


Figure 3. ROM with decoder

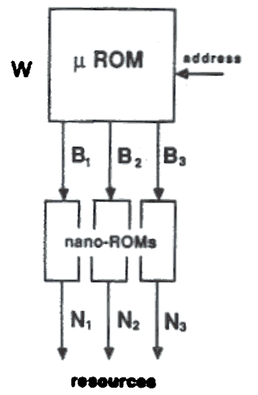


Figure 4. ROM with multiple decoders

controls to a subset of resources. Therefore more words may be needed than in a fully horizontal scheme and for this reason such an encoding is called *vertical.* Note that more micro-cycles may be needed to execute vertical micro-code, resulting in a potential loss of performance. In addition, a vertical encoding requires a decoding network which may either be distributed along the resources being controlled or centralized in another decoding array. In this case, the overall control store is said to be a two-stage control structure. When ROMs are used, they are called micro-ROM and nano-ROM respectively (Figure 3).

Fully horizontal and fully vertical encoding represent two extremes of a wide spectrum of intermediate solutions, encompassing trade-offs between storage size (in terms of area) and performance (in terms of micro-cycles). Intermediate encoding schemes can be used to optimize the storage of two-stage control structures, without affecting maximal concurrency. Therefore they optimize the performance and the number of words in the micro-ROM. The key is to partition the control signals into fields, such that no more than one control in each field is active at any given time. Each field is vertically encoded and a code is reserved to specify that no control is active. A separate decoder is required for each field (Figure 4). Schwartz[17] proposed a partitioning algorithm that minimizes the number of decoding arrays, under these assumptions. Unfortunately Schwartz's method does not necessarily minimize the number of bits required in the micro-ROM. This problem was addressed by Grasselli,[18] who reformulated the method under the same assumptions in terms of switching theory, and presented an algorithm for partitioning and encoding that minimizes the number of bits in the micro-ROM. Therefore a two-stage control structure, implemented by using this partitioning scheme, is comparable to a one-stage fully horizontal encoded array in terms of number of words in the micro-ROM and in the performance. The advantage is that the two-stage control structure uses fewer bits in the micro-ROM. The saving in size in the micro-ROM is compensated for by the additional decoders required (nano-ROMs). It is important to remark that multi-stage control schemes are also possible. In some control unit implementations, for example the IBM μ-370, the control store is implemented as a three-stage control structure, with a micro-ROM feeding a nano-ROM that drives several decoding arrays, implemented by PLAs (Figure 1).

For PLA-based implementations of the control store, similar techniques may be used for synthesizing the control structure, because there is always a trivial PLA implementation of any logic function stored in a ROM. Moreover, there are often PLA representations whose number of implicants is significantly smaller than the number of words in a corresponding ROM implementation. Therefore other techniques, such as logic minimization, can be used to reduce the storage requirement. In addition, the encoding of the control signals has an impact on the number of implicants in a minimized representation. Symbolic minimization techniques may be used to determine optimal encoding of the signals in a multi-stage control structure. Since these techniques (logic and symbolic minimization) are related to the control optimization, they are described in Section 4.

3.2. Control synthesis from behavioural system specifications

We consider now the control synthesis from behavioural specifications of the entire hardware system

Although behavioural representations differ widely in syntax and semantics, they bear similar underlying hardware models. For this reason, these representations may be abstracted by graphs, that represent data flow (e.g. Milner's data-flow algebra),[19] sequencing (e.g. SIF),[20] both sequencing and data flow (e.g. the value trace,[21,22] the YIF[23,24] and *dacon*[25]), or by Petri nets.[26] These graphical abstractions, called *intermediate forms*, capture in a concise form the data flow and/or the sequencing information. Therefore they have been used as starting point for hardware synthesis.

In general, a behavioural representation may be seen as a collection of concurrent processes. Each process is a sequence of *operations*, and it is described by a procedure with calls to subprocedures.* The control-flow statements in the behavioural representation and the data dependencies induce a precedence relation among the operations, that can be represented by a directed graph, called a *sequencing graph*. Each operation is identified by an *operator* and input/output *variables*. Data-path synthesis determines an appropriate interconnection of hardware resources that implement the operations according to the data-flow representation. Control synthesis identifies the function that controls the resources according to the original sequencing specifications along with a set of area/timing constraints. Control synthesis includes two major tasks: *sequencing control* definition, i.e. implementing the precedence relations represented by the sequencing graph, and *scheduling*, i.e. the assignment of the resource controls to the states of the control unit.

We make here some simplifying assumptions on the hardware model that are common to several hardware synthesis systems.[4] Synthesis is based on a synchronous control model. When the behaviour of the hardware system is described by a single procedure, the corresponding control is a synchronous finite-state machine. When procedure calls are used in the hardware specification, a hierarchy is induced by the procedure call mechanism. Often this hierachy is modified through the synthesis process to improve the circuit area and/or timing. We assume here that each node of the hierarchy is implemented by a data path and local control, as done by several hardware compilers, such as the YSC.[23] Therefore control is implemented by a hierarchical interconnection of synchronous finite-state machines. Eventually, coarse-grain hardware concurrency may be modelled at the behavioural level by multiple processes, as in the case of the HardwareC description.[20] In this case, the control for each process implementation is a hierarchy of finite-state machines. Process synchronization and control of interprocess communication may be achieved by hand-shaking mechanisms, as shown in Example 3.3a.

We show now the principles that are used to synthesize a synchronous controller from a sequencing graph that captures the hardware behaviour. To be specific, we refer here to the control synthesis model of the Hercules program.[20] Similar techniques have also been used by other synthesis approaches. We assume here that the sequencing graph is derived from well-structured code, supporting nested branching and looping constructs (e.g. no unrestricted *gotos*), for the sake of simplicity and elegance of representation. Each node of the graph, corresponding to an operation, can be in an active (executing) or in a waiting state. A node may be active as soon as all its predecessors have completed execution. This provides a simple synchronization mechanism for parallel operations. Alternative (sequences of) operations, arising from branching constructs in the specifications, are controlled according to the evaluation of the corresponding conditional. In the sequencing graph, alternative paths are encapsulated by *fork–join* node pairs, that do not correspond to actual operations, but that are used to ease control synthesis. In a conditional structure, the *join* node can be executed after any predecessor has completed execution, because only one of the alternative paths may be active for a given conditional.

Procedure calls define a hierarchy in the system specification. The control unit corresponding to each node in the hierarchy is a finite-state machine with an *enable* signal and a *done* signal. On a procedure call, the calling procedure enters a wait state and transfers control to the called procedure by the *enable* signal. The transitions from the wait state in the calling procedure are controlled by the *done* signal of the called procedure, which toggles at its termination.

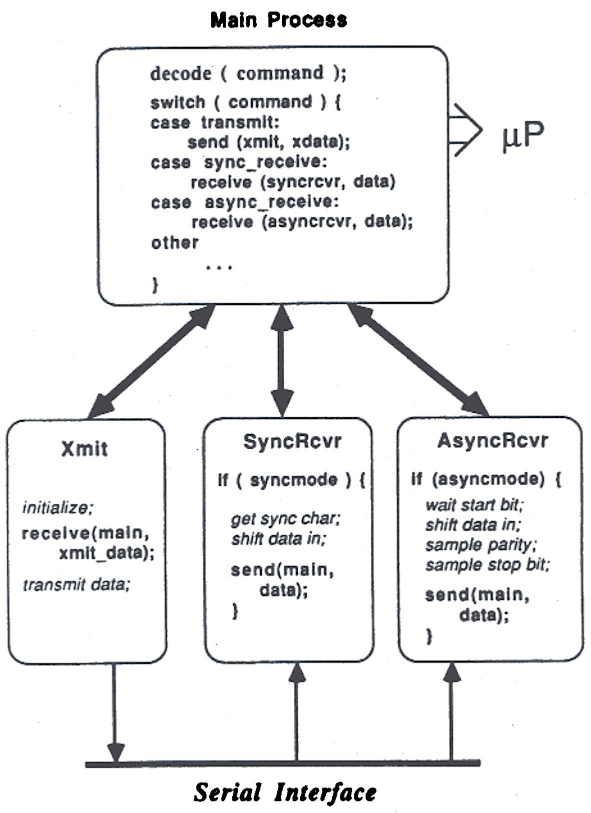
Looping constructs in the high-level description may cause cycles in the sequencing graph. In particular, they either have exit conditions that are known at hardware compile time or the exit condition may be

Some hardware synthesis systems support single process and/or single procedure descriptions only.

data dependent. For example, a looping construct may specify a fixed number of operations on some data or a repetition of an operation until some data-dependent condition is satisfied. Handling the first set of loops is simple, and loop unrolling techniques may be used to remove the cycle in the graph by repeating the operations in the loop body. Therefore such loops do not necessarily require a sequential control structure. The second set of loops cannot be unrolled, but their body can still be encapsulated by a *fork–join* pair. A control state is associated with the execution of the operations inside the loop body. The loop exit condition defines the transition from the state. Note that processing data-dependent loops can be seen as extracting (from the sequencing graph) a subgraph corresponding to the loop body and providing an appropriate linkage mechanism between the two graphs, so that the operations in the subgraph are iterated until the loop exit condition is met. Therefore, after having processed all the looping constructs, sequencing can be represented by a hierarchy of directed acylic graphs.

Inter-process communication can be described by different mechanisms. For example, HardwareC[20] supports a simple *send–reply* message-passing mechanism between any pair of processes. When executing a *send* instruction, the sending process enters a wait state and sends a *ready* signal to the receiving process. When executing a *receive* instruction, the receiving process enters a wait state if the *ready* signal is not asserted. Otherwise it reads the message and sends an acknowledgement signal to the sending process, which exits the wait state.

Example 1. Consider the segment of program in HardwareC[20] describing a portion of the I-8251 chip, shown in Figure 5(a). The chip is modelled as four concurrent processes: a main process, a transmitter and two receivers. The main process decodes a command. This is done by a call to the procedure *decode*. A conditional *switch* statement selects one out of three alternative paths. On an *xmit* command, the main process sends a message to the transmitter. Otherwise the main process waits for one of the receivers to send it a message. The corresponding sequencing graph for the main process is shown in Figure 5(b). The



```
decode ( command );
switch ( command ) {
case transmit:
    send (xmit, xdata);
case sync_receive:
    receive (syncrcvr, data)
case async_receive:
    receive (asyncrcvr, data);
other
    . . .
}
```

```
initialize;
receive(main,
    xmit_data);
transmit data;
```

```
if ( syncmode ) {
  get sync char;
  shift data in;
  send(main,
    data);
}
```

```
if (asyncmode) {
  wait start bit;
  shift data in;
  sample parity;
  sample stop bit;
  send(main,
    data);
}
```

Figure 5(a). Model of the I8251 in HardwareC

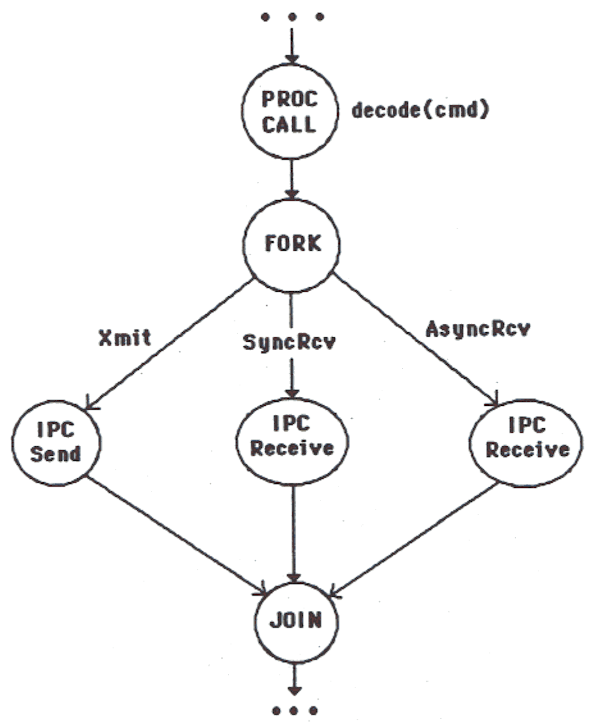


Figure 5(b). Sequencing graph of main process

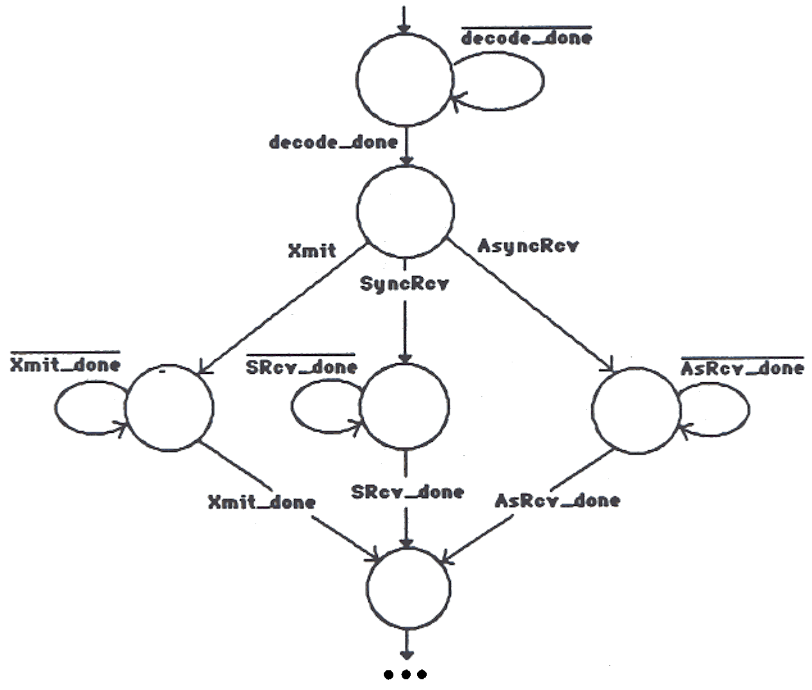

$\overline{\text{decode\_done}}$
decode_done
Xmit
SyncRcv
AsyncRcv
$\overline{\text{Xmit\_done}}$
$\overline{\text{SRcv\_done}}$
$\overline{\text{AsRcv\_done}}$
Xmit_done
SRcv_done
AsRcv_done

Figure 5(c). State diagram of main process topmost node represents the call to procedure *decode*, that can be described by another sequencing graph, not shown in the Figure. The *fork–join* pair encapsulates the alternative paths corresponding to the conditional statement. The control corresponding to the main process is shown in Figure 5(c) by a state-transition graph. Note that no state needs to be associated with the *fork–join* nodes. The topmost node issues an *enable* signal to the control unit corresponding to the procedure *decode*. The procedure *decode* returns the signal *decode_done*. The transition to the *send* and *receive* nodes depends on the conditional *switch* statement. In the state corresponding to *send*, the control of the main process sends a *ready* message to the control unit of the process *Xmit*. It stays in this state until it gets the reply *Xmit_done*, issued by the *Xmit* process, when the latter reaches the corresponding *receive* made. Similarly, in the states corresponding to *receive*, the control of the main process waits for a *done* signal from one of the two other processes.

Let us consider now the scheduling problems. Scheduling may be seen as a refinement of the sequencing control, which results in an assignment of the operations (and the corresponding resource controls) to the control states. Scheduling problems deal with: (i) the interaction between control and data path and in particular the possibility of sharing hardware resources; (ii) the delays of each operation and the timing requirements on the hardware I/O signals. Note that most scheduling algorithms consider only deterministic delays, i.e. each node of the sequencing graph must be labelled with a known delay. Therefore sequencing graphs with nodes corresponding to data-dependent conditions and inter-process communication primitives cannot be scheduled as a whole and require appropriate partitioning.

The design of the data path and control are strongly related, and in some hardware design systems the synthesis algorithms are interwoven. Consider, for example, a data path consisting of n identical operations on different variables. A maximally parallel implementation requires n identical hardware resources that can be scheduled in the same control state. Alternatively, a single resource can be used in n control states. Intermediate solutions are also possible. Since the silicon area grows with the number of resources being used, and the completion time of a sequence of operations depends on the number of control states as well as on the cycle time, area–timing trade-offs may be achieved by an appropriate resource allocation and scheduling scheme. The most straightforward approach is to decide on the resource allocation first and then on their scheduling. There is, however, a drawback in scheduling after allocating the resources: the system performance (related to scheduling) is bound to be a consequence of the decisions taken to optimize circuit area (in the allocation step). Therefore, the most appropriate formulation for this problem is to perform scheduling and resource allocation at the same time.

Most of the approaches used in recent synthesis systems are based on either scheduling under resource constraints or concurrent scheduling/allocation. Both are difficult problems, and no polynomial-time algorithm is known to solve them.[27] Scheduling under resource constraints was addressed by several researchers, because it is crucial not only for hardware synthesis but also for computer design and operation research.[28,29] Several heuristics have been proposed to solve this problem. We would like to mention here some recent approaches, relevant to hardware synthesis. A simple scheme is the *earliest possible* scheduling, where operations are scheduled as early as possible, under the constraint that the delay between two adjacent timing cuts is bounded from above by the required cycle time. When the number of resources is bounded, it may be necessary to delay the scheduling of some operations to a later state. The Emerald/Facet[30] system, the Design Automation Assistant,[31] MIMOLA[32] and Flamel[25] use this scheduling approach with some minor variations. Another scheduling method, called *list scheduling*, is used in References 33–35. It is based on a topological sort of operations by using the precedence relations implied by the data and control dependencies. The sorted operations are then iteratively assigned to control states in an order determined by a heuristic priority function.

Concurrent scheduling and allocation was used first by the MAHA program,[36] that computes instead an estimate of the critical path through the data path and then schedules the operations along the critical path by assigning them to control states. Then non-critical operations are scheduled. Paulin proposed and implemented in the HAL system[37] a force-directed method. Operations are assigned to states by an *urgency* scheduling method which blends *earliest possible* and *latest possible* scheduling, to determine the possible time-frames for each operation. Then an iterative force-directed scheduling algorithm attempts to balance the distribution of operations that make use of the same hardware resources. Concurrent allocation and scheduling were also proposed by Devadas,[38] who modelled these problems as a multi-dimensional placement problem and used an algorithm based on simulated annealing to solve it.

In control synthesis it is important to account for propagation delay through the hardware resources. Delays may vary according to the implementation style and determine the processor cycle time. Since performance is related to both the cycle-time and the number of cycles for instruction, it is then crucial to distribute the operations over a set of machine cycles on the basis of delay information. The scheduling algorithms can take operation delays into account by annotating the sequencing graph. In the limiting case that no resource bounds are specified (e.g. no area limitations are likely to be violated by a maximally parallel design), scheduling techniques simplify to the levelling of the sequencing graph that is made acyclic by loop removal and that is weighted by the delays at each node.

Hardware designers consider it very important to be able to specify I/O timing requirements along with the behavioural specification. Thus timing specifications on the hardware I/O signals may introduce bounds on the delay through a chain of operations.[39] These constraints must be taken into account for control unit design and resource scheduling. Similarly, defining the interface protocol for a processor in terms of timing constraints on the I/O signals is very important. Boriello[40] developed algorithms for synthesizing processor interfaces and their controls.

4. CONTROL OPTIMIZATION

Control synthesis constructs a structure consisting of an interconnection of sequential, combinational and auxiliary circuits. Without loss of generality, we may assume that the representation of the sequential and combinational circuits is given by tables, i.e. state tables for sequential circuits and logic covers, or truth tables for combinational ones. Although a general theory for control optimization does not exist yet, several techniques have been proposed in switching theory to optimize the representation of sequential and combinational circuits. Such techniques aim primarily at the reduction of the complexity of the logic, which correlates positively with the reduction of the gate count or silicon area. Note that the correlation has been observed in many instances, but only in a limited number of cases (such as the state encoding problem for PLA-based sequential circuits) it is possible to go beyond statistical correlation and prove relations between model minimality and area optimality.

Optimization techniques for sequential circuits include the minimization of control states and their optimal encoding. These are classical problems of switching theory[41] and they are often referred to as *state minimization* and *state encoding*. Optimization of combinational circuits, including the combinational component of sequential circuits, depends on the logic representation model. For two-level logic representation, the optimization problem is the classical two-level *logic minimization* problem. Topological optimization techniques, such as folding and partitioning,[42] can be applied to reduce further the silicon area of PLA-based two-level implementations after logic minimization. Recently, efficient multiple-level logic optimization techniques[43-46] have been introduced. Their appeal is related to the ease of mapping some logic function into multiple-level structures and to the correspondence between multiple-level models and semi-custom implementations such as standard cells or gate arrays. The multiple-level model has opened new problems: indeed the state encoding problem and the logic optimization problems had to be cast in a different formulation. The search for optimal algorithms with this model is still an area of active research.

4.1. State minimization

The state minimization problem is an important task for the optimization of any sequential circuit. It is independent of the hardware model used for the circuit implementation. State minimization 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 logic circuits required. 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 size of the state register.

State minimization has been the object of extensive research. We refer the reader to Reference 41 for the basic formalism and to Reference 47 for some recent results and an updated set of references. For state minimization, it is important whether the sequential circuit representation is *completely specified* or not. In the former case, the transition function and the output function have specified values for each pair of input condition and state. In the latter case, some *don't care* conditions are allowed.

In the case of completely specified sequential functions, it is possible to partition the state set into equivalence classes, such that for each state in a class and any input the next state belongs to the same equivalence class and the output is the same. Therefore the state minimization problem reduces to the problem of identifying the classes of equivalent states and merging together the states in each class. It was shown in Reference 48 that the reduction of completely specified finite automata with n states to a minimum representation can be achieved in $O(n \log n)$ steps.

Unfortunately, most sequential circuits representing control units are incompletely specified. The lack of a complete specification is due to the need for controlling only a fraction of the total resources in any given state. For example, if a shifter is not used in a control state (e.g. an ADD instruction) then the shifting length is a *don't care* condition. It is obvious that *don't care* conditions can be used fruitfully to minimize the states and the corresponding logic. However, this makes the problem intrinsically harder to solve. It has been shown that the minimization of incompletely specified automata is a NP-complete problem.[49] Therefore heuristic techniques are used. We refer the reader to Reference 47 for a set of references on the problem.

4.2. *State encoding*

The state encoding, or state assignment problem, applies to sequential functions and corresponds to choosing a Boolean representation for each control state. Control synthesis may use 1-hot encoding for the states, i.e. one bit is spent for each state. Such a representation is convenient for synthesis, owing to its simplicity, but it is not optimal for ROM or PLA implementations. The optimal state encoding problem corresponds to finding an encoding that optimizes the size of the logic representation with a given hardware model. In particular, when ROMs are used for the sequencing and control store, it is just convenient to reduce maximally the number of bits in the state representation, to minimize the bits required in the ROM and in the state register. In the case of a two-level logic implementation, the state encoding affects the number of implicants in a minimal Boolean cover. If a PLA is used to implement the Boolean function,* 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 and 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 encoding. The encoding length is related to the number of PLA columns. Therefore the PLA area has a complex functional dependence on state encoding. For this reason two simpler optimal state encoding problems are considered: (i) given the family of encodings that minimize the number of implicants, find an encoding of minimum code length; (ii) given the family of encodings of a given (possibly minimum) length, find an encoding that minimizes the number of implicants. In both problems, the search for an optimum encoding is broken down into two tasks to be performed sequentially: each task optimizes one parameter affecting the PLA area. In this perspective, they can be considered approximation strategies to the search for an optimum solution. 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.

Most of the classical state encoding techniques[50-52] attempted to solve problem (ii) with minimum code

* It is assumed that the PLA is not folded or partitioned for the sake of simplicity.

length. The relevance of problem (ii) was related to minimizing the number of feedback latches in discrete component implementations of finite state machines. Today, optimizing the total usage of silicon area (related only weakly to the number of storage elements) is the major goal in integrated circuit implementations of PLA-based finite state machines. For this reason some recent techniques for optimal state assignment do not require minimal encoding length. It is important to remember that most of the classical algorithms for state encoding have no practical implementation, since their complexity grows exponentially with the problem size and they can be applied effectively only to sequential circuits with few states. 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 Curts[53,54] in connection with the state encoding problem. We refer the reader to Reference 55 for an extended set of references and a critical survey of most of the previous techniques for the state encoding problem.

Recently, efficient heuristic methods for state encoding have been proposed for the PLA-based model.[7,55-57] An approach to concurrent state minimization and encoding was presented in Reference 58. The symbolic design method proposed in Reference 56 is dealt with in Section 4.4, because its scope goes beyond the state encoding problem. Amann[7] proposed a control structure based on an implementation of the sequencing and control store by two PLAs and on a loadable binary counter to hold the state information. A clever encoding scheme combines the approach used in Reference 56 with an assignment of consecutive binary codes to appropriate state chains. The transition among states in the chain is achieved by stepping the counter, whereas the other transitions are achieved by loading the next-state code into the counter.

Multiple-level logic circuits can be used for the control store. Automatic synthesis of multiple-level macro-cells have been successfully used.[5] Logic design of multiple-level sequential circuits was derived in the past from the two-level counterparts. Unfortunately, the optimality achieved for two-level circuits may no longer hold for multiple-level circuits. State encoding techniques for multiple-level circuits have not reached maturity yet. Heuristic approaches have been tried in References 59 and 60.

4.3. Logic minimization

Logic minimization techniques for control circuits are no different to those used for other classes of circuits. For the two-level model, heuristic minimizers such as ESPRESSO[61] and MINI,[62] and exact minimizers such as ESPRESSO-EXACT[63] and McBOOLE[64] have been shown to be effective and reliable tools.* Exact minimizers can be used on several medium-scale problems.

Logic optimization of multiple-level logic circuits is based on new Boolean techniques, as well as on simplified algebraic models.[43-46] Even though these techniques are rather new and the optimality properties are not yet guaranteed in most cases, they have been used successfully for processor design.

4.4. Symbolic design

We consider in this section a recent approach to the optimization of control structures consisting of interconnections of two-level logic circuits (which may be implemented by PLAs), registers and auxiliary units. In particular, this method may be applied to the state encoding problem. The approach is called *symbolic design methodology*. As a starting point, it is assumed that each logic circuit is described by a table. Each entry in the table is a *mnemonic* (i.e. a string of characters). Tables are called *symbolic covers* and may have multiple input and output fields. They can be seen as symbolic *sum of product* representations, where each row, called a *symbolic implicant*, specifies one or more output conditions in conjunction with the conditions expressed in each corresponding input field. Note that Boolean vectors

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 exists 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 if no proper subset is a cover of the same function.[61] Exact minimizers yield minimum covers whereas heuristic ones guarantee only minimality.

of 0s and 1s can be viewed as mnemonics and therefore Boolean covers, or truth tables, can be seen as symbolic covers as well.

Example 2. For the circuit structure shown in Figure 6(a), consider the table shown in Figure 6(b). The 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 inputs and one output. Each row specifies a symbolic output for any given combination of symbolic inputs.

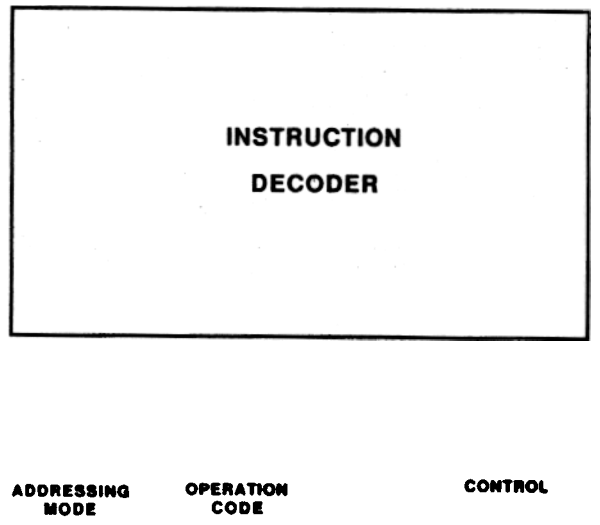


Figure 6(a). Combinational logic circuit: instruction decoder

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*

Figure 6(b). Symbolic representation of an instruction decoder. There are three addressing modes: *DIR* (direct), *IND* (indirect) and *INDEX* (indexed); four instructions: *ADD*, *OR*, *JMP* and *ADD*; four controls: *CNTA*, *CNTB*, *CNTC* and *CNTD*. Each row of the table shows a control signal to the data-path in conjunction with a given addressing mode and instruction

Example 3. Consider a simple sequential circuit implementation, as shown in Figure 7(a). The table of Figure 7(b) describes its function. In particular, it has four fields: primary inputs, present-states, next-states and primary outputs. The first and last fields are binary, the second and third are symbolic. Another example is given in Figure 7(c), showing a finite-state machine controller, that implements the control unit of the micro-processor described by Langdon.[65]

In the standard approach to synthesis, Boolean representations of switching functions are obtained by representing 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 mnemonics. As an example, the complexity (in particular the minimal cardinality of a two-level implementation) of the combinational

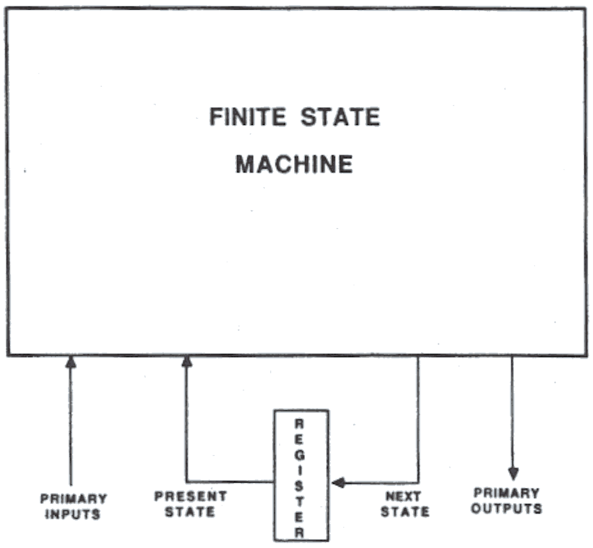


Figure 7(a). Sequential logic circuit: finite-state machine

0	*S1*	*S3*	0
0	*S2*	*S3*	0
0	*S3*	*S5*	0
0	*S4*	*S2*	1
0	*S5*	*S2*	1
	S1	*S3*	0
1	*S2*	*S1*	1
1	*S3*	*S4*	1
1	*S4*	*S3*	0
1	*S5*	*S5*	0

Figure 7(b). Symbolic representation of a sequential circuit implementing a finite-state machine. There is one primary Boolean input and one primary Boolean output. There are five symbolic states, namely: *S1*, *S2*, *S3*, *S4*, *S5*

component of a finite-state machine depends on the encoding of the states, that can be represented by mnemonics at the symbolic level.

The symbolic design method 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 find a Boolean representation of all (or some of) the inputs and/or outputs of a two-level combinational circuit that minimizes its complexity. In particular the method can be applied to combinational circuits with feedback (FSMs) to solve the state encoding problem, when this is formulated as problem (i) of Section 4.2. It can also be applied to arbitrary circuit interconnections with two-level combinational logic circuits (Figure 8).

We assume PLA-based implementations of the logic blocks. Since the area of the physical implementation has a complex functional dependence on the function representation, we consider a simplified optimization technique that leads to quasi-minimal areas. In particular we attempt to find first a representation that is minimal in the number of implicants by means of a technique called *symbolic minimization*, and then a representation of the input/outputs that is minimal in the number of Boolean variables by solving some *constrained encoding* problems. The method is presented by elaborating on the combinational circuit of Example 2. We refer the interested reader to Reference 56 for the details about the symbolic minimization and the encoding problems and algorithms.

Symbolic minimization reduces the size of the table by merging rows, or implicants, as in the case of Boolean minimization. Two mechanisms are used: (i) symbolic implicants with the same outputs are merged when their corresponding inputs can be expressed by a disjunctive relation, (ii) symbolic implicants

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	00000100000000000
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

Figure 7(c). Symbolic representation of a sequential circuit implementing a control unit. There are five fields: two correspond to the present and next control states, two fields correspond to the primary inputs (i.e. the operation code and the addressing mode) the last field to the controls. There are nine states, corresponding to instruction-fetch, operand-address evaluation and instruction execution. The states are labelled by mnemonic strings, namely: I1, I2, A1, A2, A3, A4, E1, E2, E3. Seven instructions 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 the next state and the Boolean control signals to the data path in conjunction with a given addressing mode, instruction and present state

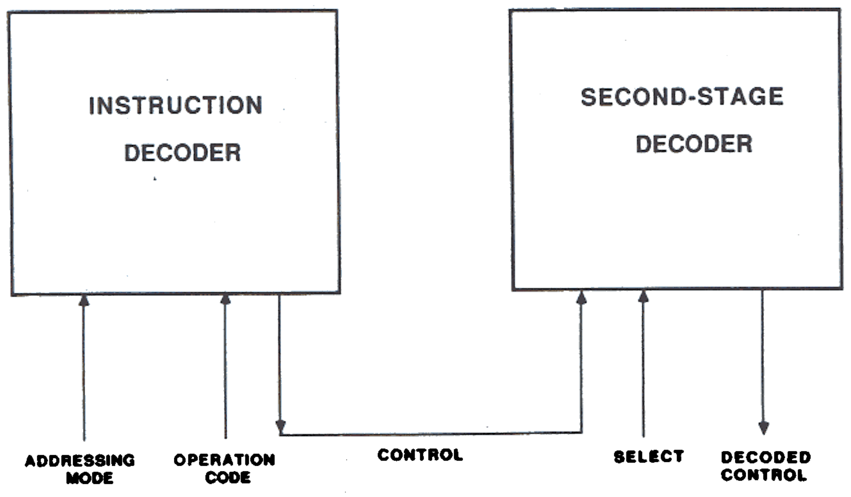


Figure 8(a). Interconnection of two combinational circuits: two-stage decoder

0	*CNTA*	100
0	*CNTB*	100
0	*CNTC*	010
0	*CNTD*	010
1	*CNTA*	100
1	*CNTB*	001
1	*CNTC*	001
1	*CNTD*	001

Figure 8(b). Symbolic representation of the second-stage combinational decoder

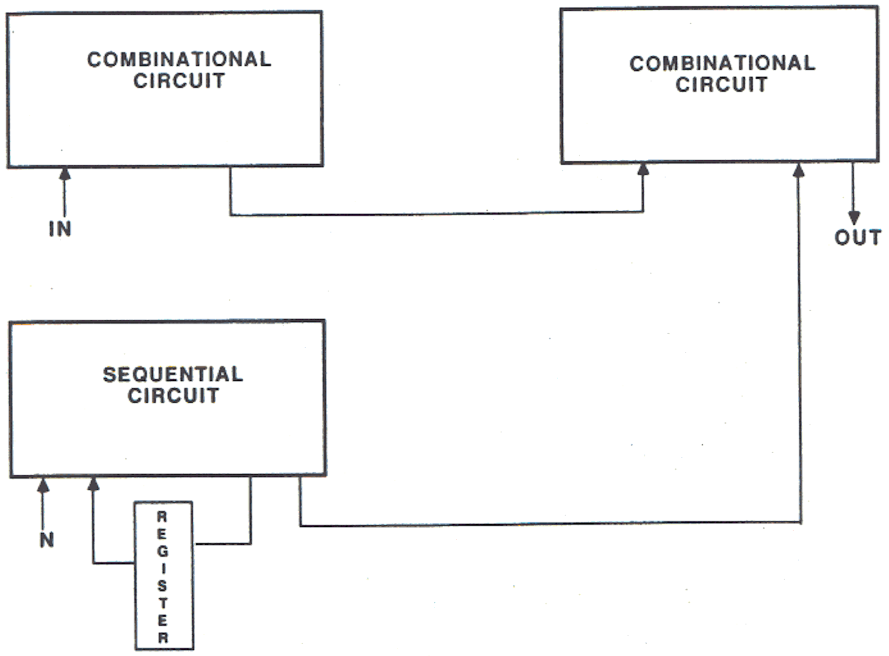


Figure 8(c). Interconnection of sequential and combinational circuits are merged if appropriate covering relations are maintained to preserve the meaning of the table. The first mechanism is a technique borrowed from multiple-valued logic minimization:[66] the second is unique to symbolic minimization

Example 4. From the table of Example 2 (Figure 6(b)), 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:

(i) addressing mode *DIR* and operation-codes *AND* or *OR*
(ii) addressing mode *IND* and operation-code *AND*

correspond to control *CNTB*. The entire table can be expressed by six implicants (instead of twelve) by using the first mechanism:

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 now one or more mnemonics may appear in a field of an implicant (e.g. *DIR IND* in the first field). Such grouping of mnemonics is important for the encoding, as shown later. By using the second mechanism, the table can be further reduced to

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

Here, a covering relation is assumed that allows control *CNTD* to override control *CNTB* and *CNTC* when both are specified. For example, the second and fourth implicant state apparently contradicting outputs for the input condition *IND OR*. Such conflicts must be resolved to preserve the integrity of the representation. Fortunately, no additional circuit is needed to implement the covering relations when conflicting outputs are specified. Indeed covering relations are implemented by the choice of an appropriate encoding, as shown below.

Symbolic tables are processed by a symbolic minimizer, such as CAPPUCCINO,[56] to minimize the number of symbolic implicants. Then mnemonics are encoded by considering each field one at a time. There are two sufficient conditions that guarantee that the encoded cover can be expressed with at most as many Boolean implicants as the symbolic cover cardinality. Namely, each literal or group of literals in a field must be encoded by a Boolean cube and the covering relations in the minimal symbolic cover must be preserved in the Boolean one. Therefore the encoding of the mnemonics is driven by two different mechanisms. For the input fields, we require that each group of mnemonics is encoded by one Boolean cube that contains the encoding of all and only the corresponding mnemonics. For the output field, each covering relation among mnemonics requires a bitwise Boolean covering relation among the corresponding encodings.

Example 5. Consider this Boolean encoding of the mnemonics:

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

Note that the mnemonic groups {*AND OR*}, {*ADD JMP*} and {*JMP OR*} can be expressed by onedimensional cubes, namely 0*, 1* and *1 (where * is a *don't care* condition on a variable). Note also that the encoding of *CNTD* covers bit-wise the encodings of *CNTB* and *CNTC*. Then the function can be represented by a Boolean cover as

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

Note that one Boolean implicant has been dropped, because its output field has only 0s.

The problem of finding Boolean codes that are compatible with a minimal symbolic cover always has a solution. However, 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. Therefore an *optimal constrained encoding problem* addresses the search for minimal-length encodings compatible with a (minimal) symbolic representation. An analysis of some encoding problems is reported in Reference 67. Heuristic algorithms have been reported in References 55 and 56, and have been successfully implemented.

Let us now consider the sequential circuit of Example 3. The finite state machine is implemented by feeding back some of the outputs of a combinational circuit to its inputs, possibly through a register. 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.[56] The feedback path makes this problem slightly more difficult, because 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 covering constraints simultaneously. For this case, necessary and sufficient conditions for the existence of a consistent encoding are given in Reference 56.

Example 6. Consider the finite-state machine of Example 3. The symbolic table can be minimized down to four symbolic implicants, namely

*	*S1 S2 S4*	*S3*	0
1	*S2*	*S1*	1
0	*S4 S5*	*S2*	1
1	*S3*	*S4*	1

under the constraints that *S1* and *S2* cover *S3* and that *S5* is the default state, i.e. *S5* is encoded by 000 and therefore covered by all other states. There are also two group constraints, namely: {*S1*, *S2*, *S4*} and {*S4*, *S5*}. An encoding that satisfies both sets of constraints for the same set of symbols is

S1	111
S2	101
S3	001
S4	100
S5	000

Note that the two groups are encoded by the cubes 1** and *00 respectively, and that the encodings of *S1*, 111, and of *S2*, 101, cover bit-wise the encoding of *S3*, namely 001. The corresponding Boolean cover is

*	**1	001	0
1	101	111	1
0	*00	101	1
1	001	100	1

Symbolic design can be applied to interconnected logic circuits. Consider first two cascaded combinational circuits, to be implemented by two-level logic macros, that communicate through a bus. For example, a micro-PLA and a nano-PLA can be used for instruction decoding, as explained in Section 2. Since the representation of the information transmitted from the micro-PLA to the nano-PLA is often irrelevant to the design, then symbolic optimization can be used as follows. The former circuit can be represented by a table with a symbolic output field and the latter by another table with a symbolic input field. The tables corresponding to both circuits are optimized independently by symbolic minimizaton and the set of symbols, representing the communication signals, are encoded as in the previous case, under the constraint that a consistent encoding is assigned to the symbols in both tables.

Example 7. Consider the cascade interconnection of two PLAs, as shown in Figure 8(a). Assume that the first circuit is the instruction decoder described in Example 2 and in Figure 6, and that the second circuit is described by the symbolic table of Figure 8(b). Assume that the first table can be minimized as in Example 3 and that the symbolic minimization of the second yields

0	*CNTA CNTB*	100
0	*CNTC CNTD*	010
*	*CNTA*	100
1	*CNTB CNTC CNTD*	001

We need now to find an encoding of *CNTA*, *CNTB*,*CNTC* and *CNTD*, subject to the constraints that *CNTD* covers *CNTB* and *CNTC*, and the group constraints: {*CNTA*, *CNTB*}, {*CNTC*, *CNTD*} and {*CNTB*, *CNTC*, *CNTD*}. The former constraints (covering) are due to the symbolic minimization of the first table (see Example 4) and the latter are related to the minimization of the second table. There is no two-bit encoding that satisfies all constraints, because of the last group constraint that requires a two-dimensional cube containing the encoding of *CNTB*, *CNTC* and *CNTD* and not containing the encoding of *CNTA*. The three-bit encoding

CNTA	000
CNTB	101
CNTC	110
CNTD	111

satisfies all covering and group constraints, and therefore allows one to implement both PLAs with the minimal number of implicants. Unfortunately, we need an extra bit in the communication bus between the circuits. Note that a two-bit encoding would be possible if we split the last product-term into two, namely:

1	*CNTB*	001
1	*CNTC CNTD*	001

In this case the group constraints can be easily satisfied by the two-bit encoding of Example 5, at the expense of one extra product-term.

Needless to say, this method can be generalized to the design of arbitrary interconnections of sequential and/or combinational units, as depicted for example in Figure 8(c). In particular it may be applied to any multi-stage control structure, where the stages, or some of them, are implemented by PLAs.

5. CONCLUSIONS AND PERSPECTIVES

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 structural, logic and physical synthesis and optimization. Control synthesis has evolved from the methods used for micro-programmable processors to new techniques that allow hardware compilation of a behavioural description of a processor. Although micro-programming allows the designer to customize a control unit using a high-level specification, now with hardware compilation the entire processor can be automatically synthesized. New design techniques may be explored. Since the partition of a processor into data-path and control fades when both are synthesized automatically and possibly using the same implementation style, methods for fine tuning the control unit to a given data-path become possible and they are key to achieving the ultimate performance with a state-of-the-art technology. Nevertheless control synthesis techniques (from behavioural specifications) still use simple structures for control and to date do not use (or use to a very limited extent) auxiliary structures, such as stacks, that have proved to be effective in control structures. For this reason, the structural synthesis of control units for processors still needs wide investigation.

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 large designs efficiently. Combinational-logic optimization techniques have reached some maturity, but only for the two-level logic model. In this case, representations with some thousands of implicants have been optimized efficiently. However, even with the two-level model, optimization of sequential logic circuits is still in its infancy. State minimization and state encoding techniques have shown to be effective for sequential structures with no more than one hundred states. The symbolic design method has been used successfully to optimize a control unit of a 32-bit processor with more than three thousand symbolic implicants and about one hundred states. However, this has been the largest example that has been optimized with such a method.

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. Even though some efficient programs have been recently developed for combinational multiple-level logic synthesis, a comprehensive theory of multiple-level logic optimization is still lacking. Moreover, techniques for sequential design based on multiple-level circuits, such as state assignment, are still in their infancy. In addition, the relations among different tasks of logic design must be understood and exploited, with particular emphasis on: (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 design system must allow the exploration of circuit structures and their trade-offs, to find the best match between circuit architecture and implementation technology.

ACKNOWLEDGEMENTS

This work has been supported under NSF grant no. MIP-8710748 and by a seed grant of the Center for Integrated Systems at Stanford.

REFERENCES

1. G. De Micheli, A. Sangiovanni-Vincentelli and P. Antognetti (eds), *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff, 1987.
2. P. Davies, 'Readings in microprogramming', *IBM J. Res. and Dev.*, **11**, (1), 16–40 (1972).
3. T. Lewis and B. Shriver, 'Special Issue on Microprogramming', *IEEE Trans. Comput.*, **C-30**, (July) (1981).
4. D. Gajski, *Silicon Compilation*, Addison Wesley, 1988.
5. 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.
6. F. Hill and G. Peterson, *Introduction to Switching Theory and Logical Design*, Wiley, 1981.
7. R. Amann and U. Baitinger, 'New state assignment algorithms for finite state machines using counters and multiple PLA/ROM structures', *Proc. Int. Conf. on Comp. Aid. Des.*, Santa Clara, November 1987, pp. 20–23.
8. R. Hadsell, 'Micro/370' in *Microarchitectures of VLSI Computers*, Proceedings NATO ASI Series E, No. 96, Martinus Nijhoff, The Netherlands, 1985.
9. P. Rubinfeld, D. Archer, D. Deverell, F. Fox, P. Gronowski, A. Jain, M. Leary, A. Olesin, S. Persels and B. Supnik, 'The CVAC CPU: a CMOS VAX microprocessor chip', *Proc. Int. Conf. on Computer Design*, Rye, N.Y., October 1987, pp. 148–152.
10. M. Horowitz, P. Chow, D. Stark, R. Simoni, A. Saltz, S. Przybylski, J. Hennessy, G. Gulak, A. Agarwal and J. Acken, 'MIPS-X: A 20-MIPS peak, 32-bit microprocessor with on-chip cache', *IEEE J. Solid State Circuits*, **SC-22**, (5), 790–799 (1987).
11. A. Goldberg, S. Hirshhorn and K. Lieberherr, 'Approaches toward silicon compilation', *IEEE Circuits and Devices*, **1**, (3), 29–39 (1985).
12. T. Agerwala, 'Microprogram optimization: a survey', *IEEE Trans Comput.*, **C-25**, 962–973 (1976).
13. T. Rauscher and P. Adams, 'Microprogramming: a tutorial and survey of recent developments', *IEEE Trans. Comput.*, **C-29**, (1), 2–19 (1980).

14. C. Papachristou, 'A scheme for implementing microprogram addressing with programmable logic arrays', *Digital processes*, Vol 5, (3–4), 235–256 (1979).
15. C. Papachristou, R. Rashid and S. Gambhir, 'VLSI design of PLA-based microcontrol scheme', *Proc. Int. Conf. on Comp. Des.*, Rye, N.Y., October 1984, pp. 771–775.
16. S. Isoda, Y. Kobayashi and T. Ishida, 'Global compaction of horizontal micro-programs based on the generalized data dependency graphs', *IEEE Trans. Computers*, C-2, (10), 922–932 (1983).
17. S. Schwartz, 'An algorithm for minimizing read-only memories for machine control', *Proc. IEEE Symp. on Switch. and Autom. Theory*, 1968, pp. 28–33.
18. A. Grasselli and U. Montanari, 'On the minimization of read-only memories in microprogrammed digital computers' *IEEE Trans. Comput.* 1111–1114 (1970).
19. R. Milner, 'Flow graphs and flow algebras', *J. ACM*, 26, (4), 794–818 (1979).
20. G. De Micheli and D. Ku, 'HERCULES: a system for high level synthesis', *Proc. Design Automation Conference*, Los Angeles, June 1988.
21. M. McFarland, 'The VT: a database for automated digital design', *Technical Reprot*, CMU, 1978.
22. D. Thomas, C. Hitchcock, T. Kowalski, S. Rajan and R. Walker, 'Automatic datapath synthesis', *Computer* 16, (12), 74–82 (1983).
23. R. Brayton, R. Camposano, G. De Micheli, R. Otten and J. van Eijndhoven, 'The Yorktown silicon compiler system' in D. Gaiskj (ed.), *Silicon Compilation*, Addison Wesley, 1988 and *IBM Report R.C. 12500*.
24. W. Rosenstiel and R. Camposano, 'Synthesizing circuits from behavioral level specifications', *Proc. 7th Int. Symp. on CHDL*, Tokyo, August 1985.
25. H. Trickey, 'Flamel: a high level hardware compiler', *IEEE Trans. CAD/ICAS*, CAD-6, (2), 259–269 (1987).
26. R. Brueck, B. Kleinjohann, T. Kathoefer and F. Rammig, 'Synthesis of concurrent modular controllers for algorithmic descriptions', *Proc. Design Automation Conference*, Las Vegas, July 1986, pp. 285–291.
27. M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, San Francisco, 1978.
28. T. C. Hu, 'Parallel sequencing and assembly line problems', *Oper. Res.*, (9), 841–848 (1961).
29. C. V. Ramamoorthy, K. M. Chandy and M. J. Gonzales, 'Optimal scheduling strategies in a multi-processor system', *IEEE Trans. Comput.*, C-21, (2), 137–146 (1972).
30. C. Tseng and D. Siewiorek, 'Automated synthesis of datapath in digital systems', *IEEE Trans. Comp. Aided Design*, CAD-5, (3), 379–395 (1986).
31. T. Kowalski and D. Thomas, 'The VLSI design automation assistant: what's in a knowledge base', *Proc. Des. Autom. Conf.*, Las Vegas, June 1985, pp. 252–258.
32. G. Zimmermann, 'MDS—the mimola design method', *Journal of Digital Systems*, 4, (3), 337–369 (1980).
33. J. Nestor and D. Thomas, 'Behavioral synthesis with interfaces', *Proc. Int. Conf. on Computer Aided Design*, Santa Clara, November 1986, pp. 112–115.
34. B. Pangrle and D. Gajski, 'State synthesis and connectivity binding for microarchitecture compilation', *Proc. Int. Conf. on Computer Aided Design*, Santa Clara, November 1986, pp. 210–213.
35. M. McFarland, 'BUD—bottom-up design of digital systems', *Proc. Design Automation Conference*, Las Vegas, July 1986, pp. 474–479.
36. A. Parker, J. Pizarro and M. Mlinar, 'MAHA: a program for data-path synthesis', *Proc. Design Automation Conference*, Las Vegas, July 1986, pp. 461–466.
37. P. Paulin and G. Knight, 'Force-directed scheduling in automatic data-path synthesis', *Proc. Design Automation Conference*, Miami Beach, July 1987, pp. 195–202.
38. S. Devadas and A. Newton, 'Algorithms for hardware allocation in data-path synthesis', *Proc. Int. Conf. on Comp. Des.*, Rye, N.Y., October 1987, pp. 526–531.
39. R. Camposano and A. Kunzmann 'Considering timing constraints in synthesis from a behavioral description', *Int. Conf. on Comp. Des.*, Rye, N.Y., October 1986, pp. 6–10.
40. G. Boriello and R. Katz, 'Synthesis and optimization of interface transducer logic', *Proc. Int. Conf. on Comp. Aid. Des.*, Santa Clara, November 1987, pp. 274–277.
41. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice Hall, 1966.
42. 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.
43. R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, 'MIS: a multiple-level logic optimization system', *IEEE Trans. CAD/ICAS*, CAD-6, (6), 1062–1081 (1987).
44. J. Darringer, D. Brand, J. Gerbi, W. Joyner and L. Trevillyan, 'LSS: a system for production logic synthesis', *IBM J. Res. and Dev.*, 28, (5), 537–545 (1984).
45. R. Brayton and K. McMullen, 'The decomposition and factorization of Boolean Expressions', *Int. Symp. on Circ. and Syst.*, Rome 1982, pp. 49–54.
46. K. Bartlett, W. Cohen, A. De Geus and G. Hachtel, 'Synthesis and optimization of multilevel logic under timing constraints', *IEEE Trans. CAD/ICAS*, CAD-5 (4), 582–596 (1986).
47. B. Reusch and W. Merzenich, 'Minimal coverings for incompletely specified sequential machines', *Acta Informatica*, (22), 663–678 (1986).
48. J. Hopcroft, 'An *n* log *n* algorithm for minimizing states in a finite automaton', in Z. Kohavi (ed.), *Theory of Machines and Computation*, Academic Press, 1971, pp. 189–196.
49. C. Pleeger, 'State reduction of incompletely specified finite state machines', *IEEE Trans. Comput.* 1099–1102 (1973).
50. J. Hartmanis, 'On the state assignment problem for sequential machines I', *IRE Trans. Elect. Comp.*, EC-10 157–165 (1961).
51. D. B. Armstrong, 'A programmed algorithm for assigning internal codes to sequential machines', *IRE Trans. Elect. Comp.*, EC-11, 466–472 (1962).

52. T. A. Dolotta and E. J. McCluskey, 'The coding of internal states of sequential machines', *IEEE Trans. Elect. Comp.*, EC-13, 549–562 (1964).
53. H. A. Curtis, 'Systematic procedures for realizing synchronous sequential machines using flip-flop memory: part 1', *IEEE Trans Comput.*, C-18, 1121–1127 (1969).
54. H. A. Curtis, 'Systematic procedures for realizing synchronous sequential machines using flip-flop memory: part 2' *IEEE Trans Comput.*, C-19, 66–73 (1970).
55. G. De Micheli, R. Brayton and A. L. Sangiovanni-Vincentelli, 'Optimal state assignment for finite state machines', *IEEE Trans. CAD/ICAS*, CAD-4, (3), 269–284 (1985).
56. G. De Micheli, 'Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros', *IEEE Trans. CAD/ICAS*, CAD-5, (4), 597–616 (1986) and *IBM Research Report RC 11672*.
57. L. Philipson, private communication.
58. M. Perkowski and B. Lee, 'Concurrent minimization and state assignment for finite state machines', *Proc. IEEE Conf. on Systems, Man and Cybernetics*, Nova Scotia, October 1984.
59. S. Devadas, H. T. Ma, A. Newton and A. Sangiovanni-Vincentelli, 'MUSTANG: state assignment algorithms of finite state machines for optimal multi-level logic implementation', *Proc. Int. Conf. on Comp. Aid. Des.*, Santa Clara, November 1987, pp. 16–19.
60. W. Wolf, K. Kreutzer and J. Akella, 'A kernel-finding state assignment algorithm for multi-level logic', *Proc. Design Automation Conference*, Annaheim, 1988.
61. R. Brayton, G. D. Hachtel, C. McMullen and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
62. S. J. Hong, R. G. Cain and D. L. Ostapko, 'MINI: a heuristic approach for logic minimization', *IBM J. Res. and Dev.*, **18**, 443–458 (1974).
63. R. Rudell and A. Sangiovanni-Vincentelli, 'Multiple-valued minimization for PLA optimization', *IEEE Trans. CAD/ICAS*, CAD-6, (5), 727–750 (1987).
64. M. Dagenais, V. Agarwal and N. Rumin, 'McBoole: a new procedure for exact logic minimization, *IEEE Trans. CAD/ICAS*, CAD-5, 229–238 (1986).
65. G. Langdon, *Computer Design*, Computech Press, 1982.
66. D. Rine, *Computer Science and Multiple-Valued Logic*, North Holland, 1977.
67. T. Villa 'Constrained encoding in hypercubes: algorithms and applications to logical synthesis', *UCB/ERL Memorandum M87/37*, May 1987.
68. D. Thomas, E. Dirkes, R. Walker and V. Rajan, 'The system architect's workbench', *Proc. Design Automation Conference*, Los Angeles, June 1988.