THE ENCODING PROGRAM FOR CONCURRENT FINITE STATE MACHINES REALIZED USING PLD DEVICES

Marek A. Perkowski,
Department of Electrical Engineering, Portland State University,
P.O. Box 751, Portland, OR 97207, tel. (503) 725-3806 x 23..

Loc Bao Nguyen, Intel Scientific Computers
5200 NE Elam Young Pkwy, Beaverton, OR 97006.

ABSTRACT

A Concurrent Finite State Machine (CFSM) is a network of Finite State Machines, each FSM from the network has an arbitrary number of symbolic input and output ports that communicate to other FSMs and to the outside world. Such machines include, in particular, FSMs with counters, stacks, subroutine registers, encoders and decoders. Program AE, Assignment Expert, is described, which finds the encodings of input ports, output ports and internal states of the FSMs. It finds the solution to the constrained problem of simultaneous input/output/internal-state assignment of CFSMs. It is a two-pass method: a very fast modified quadratic assignment algorithm for graph embedding is first iterated several times, using different quality functions and realization-related cost functions. Next, the knowledge-based, rule-implemented, optimizing transformations are executed. To improve the results, the technology-related cost of the logic mapping is used in the optimization loop. This technique is particularly useful to implement large concurrent state machines realized using several instances of new types of Programmable Logic Devices (PLDs).

1. THE CONCURRENT FINITE STATE MACHINES

A Concurrent Finite State Machine (CFSM) is a *network of Finite State Machines (FSMs)* that communicate through their input and output signals. Outputs of some machines are inputs to another ones, input signals can be also shared among the FSMS. The DIADES system [57,58] describes a digital circuit, usually a controller, as a CFSM. A CFSM is realized with one or more PLD devices. There is a need to create design methodologies for efficient realization of such machines.

The general abstract model for a single FSM M_i from the CFSM network is one having an arbitrary number of *symbolic input ports:* $IP_1, \ldots, IP_{n_i}$, and *symbolic output ports:* $OP_1, \ldots, OP_{m_i}$. The *component machine of a CFSM* can be a Mealy or (more often) Moore type. The component Mealy FSM is defined as follows:

$M_i = < IP_1, ..., IP_{n_i}, IS_1{}^i, ..., IS_{n_i}{}^i, DI_i, A_i, OP_1, ..., OP_{m_i}, OS_1{}^i, ..., OS_{m_i}{}^i, DO_i, \delta_i, \lambda_i >$,
where:

- $IP_1, ..., IP_{n_i}$ are symbolic input ports,
- $IS_j{}^i = \{ I^1{}_j, ..., I^s{}_j, ..., I^{k^j}{}_j \}$, s=1,...,k^j, is the set of the symbolic input values (symbols) of port IP_j, j=1,...,n.
- DI_i is the set of (direct) logic input signals (binary),
- A_i is the set of symbols of states,
- DO_i is the set of (direct) logic output signals,
- $OS_j{}^i = \{O^1{}_j, ..., O^s{}_j, ..., O^{g^j}{}_j\}$, s=1,...,g^j, is the sets of the symbolic output values of port OP_j, j=1,...,m.
- $\delta_i = IS_1 \times ... \times IS_{n_i} \times L(DI_i) \times A_i \dashrightarrow A_i$ is the transition function,
- $L(DI)$ is a logic function on set of variables DI_i,
- $\lambda_i = IS_1{}^i \times ... \times IS_{n_i}{}^i \times L(DI_i) \times A_i \dashrightarrow L(DO_i) \times ... \ OS_1{}^i \times ... \times OS_{m_i}{}^i \times DO_i$
  is the output function,
  $L(DO_i)$ is a logic function on set of variables DO^i.

The assignment problem for a component machine M_i is to find a code $code(S_j)$ (a binary sequence) for each of the symbols S_j from the sets: A_i, $IS^i{}_j$, j=1,...,n_i, and $OS^i{}_j$, j=1,...,m_i. (each symbol in each port needs to be encoded). Assigning codes to symbols from A_i is called the *state assignment.* Assigning codes to symbols from $IS^j{}_i$, j=1,...,n_i, is called the *input port assignment (input assignment).* Assigning codes to symbols from $OS^i{}_i$, i=1,...,n_i, is called the *output port assignment (output assignment).* In general, assigning a code *code* to any symbol S will be called *assignment* or *encoding* of this symbol. Sometimes some of the codes are given already in the initial specification of the machine (for instance, this happens in classical machines where inputs and outputs are initially encoded). Therefore, the direct input and output signals have been introduced to our model.

The CFSM is defined as a network of component FSMs:
$CFSM = <\{M_i, i=1,...,K\}, R>$,
where:

- $\{M_i\}$ is a set of component FSMs,
- $R = OP \times IP_1 \times ... \times IP_m$ (where $OP = \cup OP^i$, i = 1,...,m)) is the relation of connecting the output ports to input ports (one output port can be connected to any number of input ports).

Let us observe, that in this model a classical FSM M = <X, A, Y, δ, λ> with one input set X and one output set Y, realized using D type flip-flops, becomes a machine with two input ports: IP_1, IP_2, two output ports: OP_1, OP_2, and the following contraints:
$A = IS_1 = OS_1$ (loop from output to input), $Y = OS_2$, $X = IS_2$.
IP_1 and OP_1 are called the *feedback pair of ports.* Let us observe that the output port from this pair can be directly used as an input port by some component machine other than M (essentially, this happens quite often in concurrent state machines that realize, for instance, the Petri Nets).

Let us observe that since the component machines from the network are interrelated through the ports - the assignments of their ports are also related. For instance, if the port OP_2 of machine M_3 is connected to the port IP_4 of machine M_7 then finding a set of codes for symbols from $OS^3{}_2$ will mean using these codes also for $IS^7{}_4$. Let us also observe that if all input and output ports of a component machine M_i (other than the feedback ports) become encoded as the result of assignments in all the component machines that share ports with M_i, then the assignment problem of M_i becomes a classical state assignment problem of encoding only the internal states of M_i. Let us finally observe that each machine of CFSM can be in particular case a purely combinational logic. This makes the above model very general, it includes many practical realizations of controllers, discussed in literature and listed in the next section.

2. REALIZATION OF CFSMS IN PLD. THE ASSIGNMENT PROBLEM.

A Fan-In OR Constrained Logic Realization is one, like in a PAL or a EPLD, where there is a limited fan-in for the OR level of logic, but practically unlimited fan-in for the AND level.

The following constraints must be taken into account when designing state machines using PALs or EPLDs (from now, we will use a generic term PLD to all such devices).

1. Most of the commercial registered PLDs implement only the D-type flip-flops. This type is still the most popular among the high speed PLDs.
2. For the 20 and 24 pin PLDs, there are at most eight registered outputs. Hence, this will limit how large the FSM can be.
3. Each D-input of the above eight registered outputs has at most eight products in the sum term. This constraint severely restricts the design as well.
4. The number of inputs is limited to 21 and is found sufficient for most of the machines designed in practice at the board level.
5. Even if in the newest PLDs the numbers and types of flip-flops, inputs and outputs are larger, the basic design contraints and cost functions remain of the same nature.

Because of these restrictions, only a small and medium size FSM can be designed using a single instance of a PLD. This is, however, not a problem for two reasons:

1. From our personal experience, the component FSMs of less than 15 inputs and 8 states are frequently encountered in the board level design.
2. Larger machines can be decomposed to such machines using the methods recently developed [2,25,57,74,75].

In addition, each component machine has normally more than one output signal. It is then obvious that the output pins are scarce resources in the PLD-based FSM design. As the consequence, the outputs of the machine are normally encoded as the state variables to save the I/O pins for some extra functions (either input ot output). This design style is especially practical for realizing CFSMs.

With this design style, the designer knows often in advance the minimum number of the flip-flops that are to be used for each component FSM, and all that is necessary is a method to assign binary codes to the state variables, such that the excitation functions described by the Boolean equations will fit into the device.

At the moment, there are several programs available for the state asssignment of classical FSMs. The well-known Kiss [17,19,21,22], Mustang [24], and NOVA [70] are in the public domain. Stash (INTEL) [13], Capuccino (IBM) [20,23] and Mustard (AT&T) [72,73] are proprietary. However, often, the tools that we are familiar with (Kiss, Mustang, Nova, Stash), do not take sufficiently the fan-in constraints, that exist not only for the PLDs but also for other design styles, such as the multi-level logic.

It has been shown that there are several advantages of realizing the excitation logic of a PLD-realized FSM in the form in which the AND plane is done using the PLA-like regular layout plane but the OR plane is absent: OR gates are composed of single fan-in constrained cells, also gates other than OR (such as AND/NAND, EXOR, etc.) are used. This fact is increasingly taken into account in new PLD architectures. Also, some other exotic logic cells are becoming popular, such as the Conditional Decoder gate in the recently introduced CY7C361 device from Cypress Semiconductor, intended to realize concurent state machines, Regular Expressions, and Petri Nets.

When realizing the logic for the microprogrammed state machines (the hybrids of microprogrammed controllers and state machines, they inherit best properties of both their parent machines - see [2]) one has to assign, in some variants, not only the internal states, but also the input states and the output states of the machine. In such cases the state machine has the input encoder, the output decoder, or both of them. In [54] several microcoded machines are described (using both counters, and *separate registered subsystems for non-branching parts of FSMs).* The methods introduced here would also improve the efficiency of such machines. Similarly, when the FSM state minimization technology decribed in [60] is used, the input encoder is the result of the *input minimization procedure,* and the problem of code assignment for this encoder exists. The methodology presented in this paper can be used for all these design approaches as well. Finally, many FSM design methodologies use networks of FSMs created from direct CFSM description, high level description such as a Petri Net or parallel program graph [58], FSM decomposition [43,48,35,76] or partitioning. No program currently exist for the state assignment of CFSMs and FSMs for constrained logic realization, and programs like Nova, Kiss or Stash applied to component machines can sometimes give results worse than by hand. Also, those programs do not take into account the constraints resulting from the structure of the network of machines. No CAD tools exist which would minimize the excitation functions for FSMs and especially CFSMs, realized in PLDs.

CH2819-1/90/0000-0204$01.00©1991 IEEE

The program described here, AE for Assignment Expert, gives very good results for machines implemented in PLDs, it was applied with good results to large ASIC VLSI machines and large structurally decomposed CFSMs. Moreover, it finds the solution to the constrained problem of simultaneous input/output/internal-state assignment of concurrent FSMs, and not only the assignment of internal states, as most of the well known methods. We hope that the results of this research will be quite useful practically for the companies that develop the new generation synthesis tools for PLDs.

3. THE ASSIGNMENT OF CFSMS.

The AE program assigns automatically the internal states of all component machines, as well as the input and output symbolic signals in all ports of the component machines. The assignment process of a CFSM is a sequence of assignments of ports in its component machines. Such assignments are called *partial assignments*. All partial assignments are realized on very similar basic principles.

AE uses several types of rules: rules to select the order of partial assignments, rules to create input, state and output assignments. Below, the most important of those rules will be discussed. The Lisp-implemented rules collaborate with the heuristic algorithms (such as the Quadratic Assignment), and the algorithmic subroutines (such as the generation of all set partitions of certain type).

The order of the ports for partial assignments is decided heuristically. Based on the structure of the network of FSMs, and the complexity of the component machines and their ports, the complexities of all the partial assignment problems for all ports are evaluated. Next, the assignment problems evaluated as the most complex are solved in the first rate. Once the port is encoded in one machine, its code is propagated through the network to all machines that share this port. This improves assignment quality of all these machines, since the assignments for them are now performed in more realistic terms.

The user has also the option of manually assigning codes to any selected type(s) of states or symbolic signals. Moreover, the user has the option to restrict the set of assignment choices by declaring the set of codes to select codes from in the state assignment processes. For instance, he can enumerate a set of codes to choose from, he can also declare codes smaller than some selected number (codes, as the binary vectors can be represented internally as numbers in a computer). He can, as well, declare any applicable "n out of m" code. (A special circuit can be added that will detect in real time that the machine goes out of this code. Therefore, all single stack-at faults and most of multiple faults are detected in a run). Finally, the user has the option to control the assignment process by use of the *program parameters*. User creates the agenda of actions to execute, and this agenda is updated and modified by the programs of the system during the design process. The concept of *agenda* is a dynamic generalization of the concept of *script*.

Each partial assignment is found in two phases:

- Algorithmic Hypercube Embedding, realized by program FASS (Fsm ASSignment).
- Knowledge-based code improvement, realized by program RUBASS (Rule Based ASSignment).

4. THE FIRST PHASE OF THE PARTIAL ASSIGNMENT

Program FASS uses subroutine HyperCoder which solves the partial assignment for a hypercube of a given dimension as the modification of the *Quadratic Assignment Problem*. In the quadratic assignment approach to state assignment, like in many other combinatorial problems of logic design, one can distinguish two phases:

1. Creation of the *assignment graph AG*, and formulation of the cost function CF to be optimized.
2. Solution to the Quadratic Assignment Problem for graph AG that minimizes the function CF.

Different methods exist in AE to create the AG graphs; for input ports, output ports and internal states. Their principles are however the same: the more desired it is to have symbols Si and Sj to be of small Hamming distance after the assignment in order to minimize logic, the higher value of a weighted cost function $edge\_cost(S_i,S_j)$ should be assigned to the edge(Si, Sj) in the AG. Different approaches have been used for the creation of the assignment graph in different papers. Saucier [Sauc 72] creates a nonoriented weighted graph, whose edges correspond to the transitions between the states of the FSM. Moroz [Moro 70] creates an oriented graph, whose edges correspond to the oriented transitions between the states. Although he writes about the embedding, his work can be treated as an approximate solution to the quadratic assignment of particular type, in which the graph is oriented, the costs of edges are equal, and the cost function is defined as in the quadratic assignment problem. This approach attempts to form Grey code assignments, the method that has some additional advantages, especially for asynchronous machines. Armstrong [Arms 60, 60a] formulates a nonoriented graph, whose edges are created according to several principles of adjacency. We created a method that uses a generalized weighted cost function for each edge of AG: the value of this function is a weighted sum of *component evaluation functions*. By selecting appropriate values of coefficients we are able to emulate the well known methods and compare their applicability to various assignment problems. Some of the partial functions are new and correspond to the input and output ports assignments.

Also the solution to the second phase is different in our approach. In contrast to MUSTANG which uses the cost function:

$$CF = \sum_{i=1}^{i=NN-1} \sum_{j=i+1}^{j=NN} edge\_cost(S_i, S_j) * HAMMING[code(S_i), code(S_j)]$$

where: S is the set of symbols, NN = CARD(S), $S_i S_j$ are symbolic states, $code(S_i)$ is a code of state S_i, and HAMMING is a Hamming distance of codes, we minimize the function:

$$CF = \sum_{i=1}^{i=NN-1} \sum_{j=i+1}^{j=NN} edge\_cost(S_i,S_j)*EVAL[code(S_i), code(S_j)] + F2$$

where function EVAL is defined as follows:

$$EVAL(code_1, code_2) = \left\{ \begin{array}{l} 0 \quad \text{if } HAMMING(code_1,code_2) = 1, \\ \sigma(HAMMING(code_1, code_2)), \quad otherwise \end{array} \right\}$$

where σ is a sigmoidal function of one variable.

Additionally, we minimize the number of true minterms by selecting codes that have as few symbols "1" as possible. This is done by calculating the following F2 component of the cost function:

$$\alpha * \sum_{i=1}^{NN} COSTON(S_i) * Number\_Of\_Ones(code(S_i))$$

where:
α is a parameter, COSTON is an evaluation of difficulty of creating implicants in realization of state S_i (based on the number and location of symbols S_i).
Our function CF has advantages of approaches from [50,2,3,24], but can be made similar to any of them.

Different constructive algorithms have been used by the above authors to embedd the created by them graphs to hypercubes. They do not give any, other than heuristic, explanations of adequacy of the proposed by them assignment (embedding) techniques. Also, the program of Saucier is only for asynchronous machines.

FASS embedds the AG graph to a hypergraph in such a way that the pairs of nodes of AG that have high connection cost in their respective edges, are placed in adjacent nodes of a hypercube, or nodes of small Hamming distance. FASS does not assume to design a circuit with the minimum possible number of flip-flops, as do several of the classical approaches, neither it wants to satisfy all the groups from multi-valued minimization, as in one of the approaches of Micheli. The dimension of the hypercube is first selected as the minimum number of binary signals (flip-flops) necessary to realize the given set of symbols from the port. Then FASS looks for the optimum realizations by gradually increasing the number of binary signals. Only few numbers are investigated, not much larger than the minimum number of the signals. The costs of the embeddings to hypercubes of larger dimensions are compared. When essential worsening of the cost occurs, the growth of the hypercube is interrupted. As the practice shows such approach is sufficient. This approach is consistently used by FASS to all kinds of symbolic ports, when it calls HyperCoder with corresponding AG graphs and respective hypercubes of increasing dimensions.

HyperCoder is a constructive algorithm that successively selects nodes of the AG graph and assigns them to nodes of a hypergraph. In contrast to most quadratic assignment algorithms, HyperCoder does not use the explicit assignment graph adjacency matrix (which would be very inefficient) but constructs the codes for the node symbols step-by-step, while traversing the AG graph. HyperCoder uses several methods to select the next symbolic node of AG: most of these methods select the node SS that is most connected to the already assigned symbolic nodes. The nodes in AG that share edges with SS are called its *neighbors* and denoted by *neighbors(SS)*. A set of all codes assigned to nodes from neighbors(SS) is found. Next, for each of those codes C_r a set of all candidate codes in Hamming distance one to it is generated. We call it a $candidate(C_r)$ set of codes. A set CANDIDATES(SS) is created as the set sum of all such candidate sets. The already used codes are removed from CANDIDATES(SS). Next, a code is selected from CANDIDATES(SS) that locally minimizes the current weighted cost function CF, calculated only for the SS and its already assigned neighbors from neighbors(SS). This code is assigned to SS and the procedure iterates by selecting next SS node, until all nodes were assigned.

FASS realizes a *multi-variant synthesis* concept. This means, that different design approaches are created with use of different kinds of AG graphs and different assignment algorithm variants realized by HyperCoder. We have examples, that multi-variant synthesis gives better results for CFSMs than a single method, iterated many times. The principle is this: we can create different assignment graphs and next find the assignments for them, using various methods. If we have two methods of creating the assignment graphs, and two methods of solving them, this approach would create four solution methods. It is usually better to try these four methods, than to iterate one method four times. FASS iterates similar variants of HyperCoder, each of them using different graph AG created from the machine's list of transitions, slightly different cost function CF, and rules for selecting next node. It evaluates the variants using CF. The best of the selected according to CF variants are next evaluated again, after logic minimization, this time using a more complex cost function, based on the technology mapping. This function checks also input, output, state variable and fan-in constraints, essential for the PLD realization. Although this idea seems extremely obvious, surprisingly to our knowledge nobody has yet reported its usage.

The FASS program generates quickly first approximate solution and next generates other solutions with more sophisticated methods, using a family of evaluation functions used in weighted functions for AG edge costs and assignment rules. This permits to have at least one solution when the user's allotted time is exhausted. When sufficient computer time is allowed, the program can work long and there is a chance that better solution will be found (The question is this: is it worthy to work the whole weekend to find a 3% better realization of a 100 states' machine? It seems worthy when one wants to fit the design to a device, the chip resources are only slightly exceeded, and there is a hope that better assignment will allow to realize the machine without increasing the chips count).

5. THE SECOND PHASE OF THE ASSIGNMENT.

The second pass of AE uses the heuristic rule-based program RUBASS to locally improve the code, generated by FASS. The rules are applied while traversing the *List of Transitions*, which represents the symbolic transition graphs of all the component FSMs. The transitions are in the form:

<input logic function of a transition> <input port symbol> ... <input port symbol> <output port symbol> ...<output port symbol> <output logic function of a transition> ... <output logic function of a transition>

The logic function is not necessarily a binary vector, ternary vector(with don't cares) or a "cube calculus" cube, as in the known approaches. The ports may be encoded, partially encoded or not encoded at all. From such list of transitions several Transition Graphs can be created: one for each symbolic port being not in feedback, and one for each feedback pair of ports. The symbols from this port become the states (nodes) of the Transition Graph.

Firing a rule invokes other rules-candidates to the agenda. Those rules, the nodes and the state variables where to apply them, are selected heuristically, on global or local bases. The globally selected are the nodes that have maximum values of some measures:

in first order - TOTAL, next COSTON, next the number of neighbors, and finally - the number of adjacent arrows. The locally selected rules select new nodes of the transition graph that are the neighbors (in first order - the predecessors, next - the succesors) of the previously assigned/modified nodes. When there are several equally rated choices - a random selection of a rule number, a node and a variable is done.

The rules used in the program have been derived from the practical experiences of attempts to improve assignments. As mentioned earlier, the number of product terms for a registered output PLD (PAL) is very limited (only 8 terms). Hence, the excitation function realization exceeds often the limit imposed by the PLD architecture. The method to assign binary codes is, therefore, very important, because the complexity of the excitation functions and the number of the product terms in particular, are the direct results of the state assignment. We wanted, therefore, to create a method that produces the PLD-optimized solution as often as possible, for sizes of FSMs encountered in practical PLD realizations.

Basically, there are two classes of designs.

a) The output signals are other types of signals than the state variables.
   - the outputs are functions of the inputs and state variables (Mealy model).
   - the outputs are functions of the state variables only (Moore model).

b) The outputs are encoded as the state variables (Moore model).

In the class b), there is less freedom to perform the state assignment than in the class a), because the state assignment is dictated by the output signals polarity.

Example 5.1. The state variables for the state assignment from Fig. 5.1 are: {R1, R0}. Output: $y = R1\overline{R0} + \overline{R1}R0$. This design takes 3 output pins, whereas for the assignment of the same machine shown in Fig. 5.2 the output: y = R0.

This design takes only 2 output pins. In this scheme, however, the state variable R0 in states B and D is dictated by the polarity of the output y.

At this point we are ready to introduce the set of heuristic rules for state assignment which attempt to minimize the excitation functions.

Definition 5.1. COSTON and COSTOFF. Let set V be a set of encoding variables. Let $V_{i,n} = (0, 1)$ for all $V_{i,n} \in V$, where subscript i is for state variable i, and subscript n is for the current state. Let X be a set of branching conditions, i.e. X = {A, Y, Z}. Thus CARD{A} = 3.

COSTON:

If $V_{i,n} = 0$ then COSTON = 0.

If $V_{i,n} = 1$ then COSTON = number of product terms going to the state plus the number of product terms looping in that state.

The set D in Fig. 5.3 is the *set of looping product terms* for that state.

COSTOFF:
If $V_{in} = 0$ then COSTOFF = 0.
If $V_{in} = 1$ then COSTON = number of product terms going out of that state.

Example 5.2. COSTON = CARD{A,B,C,D} = 4, COSTOFF = CARD{E} = 1. TOTAL = COSTON + COSTOFF.

Note that E is a complement of D. Otherwise, the transition from state n to the next state n+1 would not be deterministic.

Let us observe that first, before state assignment, the costs COSTON, COSTOFF and TOTAL are calculated for the worst case. Next, when partial codes are already known, these costs can be recalculated more accurately and will be never higher.

Example 5.3. The transition function for state B is shown in Fig. 5.4 and Table 5.0 below. State variables: V = {V_2, V_1, V_0}. The variables: X, Y, K, and Z are the input variables and they constitute the branching conditions.

Implication of COSTON and COSTOFF. The COSTON and COSTOFF costs together determine the number of product terms that need to be created for the state variable under consideration, when the state machine transits from the current state to the next state.

Method for writing equations directly from the state graph. (analogous method is used for the state graph in the form of transitions).

For the D-type flip-flop, the transition table is as in Table 5.1.

The following rules apply:

1) If $V_{in} = 0$ and $V_{in+1} = 0$ then no equation is needed. It is a free transition.
2) If $V_{in} = 1$ and $V_{in+1} = 0$ and there is no looping back at V_{in} then no equation is needed. It is a free transition.
3) If $V_{in} = 1$ or 0 and $V_{in+1} = 1$ then the equation is needed. The number of product terms depends on the input set.

Example 5.4. Write the transition equation for next state J, assuming V = {V_3, V_2, V_1, V_0} (Fig 5.5). code(J) = 0101, code(J) = 0011. Branching condition = $XY + Z$, CARD{XY, Z} = 2.

Equation for state J:
For V_3 = none, cost = 0. For V_2 = none, cost = 0.
For V_1 = (0101) * ($XY + \overline{Z}$) => produces two terms, cost = 2.
For V_0 = (0101) * ($XY + \overline{Z}$) => produces two terms, cost = 2.

From the above example, the following RULE 1 can be then formulated.

RULE 1.
The higher the value of COSTON of state S_i, the more zero bits should be assigned to the state variables in S_i.

This rule is used to locally improve codes by changing $V_{i,n}$ variable values from 1 to 0 in state codes so that a proper encoding (one-to-one mapping among states and codes) is preserved.

NOTE. For any FSM, the reset signal is needed to reset the FSM to a known state on the power up or during the reset condition. Thus, the reset state normally has the highest COSTON and is assigned binary code 0.

There is a method which can bring the FSM to a known state without using the reset signal. This is achieved by making all the unused states in the state diagram to branch to some selected internal state.

Example 5.5. Consider the two-bit up-counter from Fig. 5.6. When the input X is high, the counter will count up. To be able to control the counter, we introduce the signal reset to bring it to the known state A (during reset = 1). Thus at every state, the counter will enter the state A and stay there until the reset signal is removed. The cost of state A in this example is, therefore, 5 and it is the highest cost. We respect to what was mentioned, to optimize the excitation function for this example, one assigns code 00 to state A. Normally, the reset is shown as in Fig. 5.7.

RULE 2.
If:
- there is a transition from state SA to state SB, and
- the state variable V_i in state SA is already assigned to be 1, and
- there is looping condition in state SA

then, if possible, assign V_i in state SB to value 1.

If in state SA V_i = 1 then assign 1 to V_i in state SB. Hence, COSTON of V_i in state SB = 1 since COSTON of V_i in state SB = SA * (D + $\overline{D}$) = SA * (1) = SA => CARD{SA} = one term. Rule 2 is illustrated in Fig. 5.8.

RULE 3.
If:
- there is a transition from state SA to state SB and
- there is no looping condition in SA,

then assign 0 to V_i in state SB, to achieve a free transition.

Rule 3 is illustrated in Fig. 5.9.

NOTE. Rule 3 will give better results than the Grey Code Assignment. However, one has to pay attention to the combinational outputs of the state machine, because since the assigned state codes are not in the Grey code, the output may glitch due to more than one variable changing, and variables' delays being not equal.

Let us observe that the number of times the symbol 0 or 1 can be assigned to any variable is limited by the number of flip-flops used in the design. For some machines, therefore, more state variables need to be introduced in order to fit the PLD device, when using the above rules, which must be verified by RUBASS whether the device resources are not violated.

Output Considerations.

The outputs of FSM can be the registered or combinational outputs. In the latter case, it can be in the Moore or Mealy machine form. This type of outputs required the Grey Code assignment (only one variable changes per any state transition) or the consensus prime implicants must be added in logic realization to avoid glitches (static hazards). In the first case, the outputs are clocked. The glitches will, therefore, not occur. In addition, the registered outputs are faster than the combinational outputs by a tpd (15 ns if B-PAL type is used); and 15 ns is a lot of time in a high speed design.

Observations.

1. The two schemes occupy the same number of pinouts.
2. Registered outputs are more reliable due to no glitching.
3. Registered outputs are faster.

The following is a complete example of a DRAM BUS INTERFACE design. The first part will illustrate the result of the Grey Code assignment. The second part will show the result of using the above rules.

Example 5.6. The state diagram shown in Fig. 5.10 was encoded using the Grey Code. The Boolean equation version (the output from LOGMIN) is given in Fig. 5.11. We observe that: variable R2 has two terms; variable R1 has four terms; variable R0 has six terms.

The state diagram shown in Fig. 5.12 is encoded by RUBASS using the above rules. The Boolean equation version is given in Fig. 5.13. The following facts can be observed as well: variable R2 has four terms; variable R1 has four terms; variable R0 has two terms.

This example has shown that by using the above rules one achieves a better result compared to that of the Grey Code assignment method. In fact, on most real-life cases the result was either equal or better when compared to the results from STASH (a CAD tool of INTEL which does heuristic state assignment - an improved version of KISS).

The design sequence of rules is as follows.

1. The COSTON, COSTOFF, and TOTAL costs are found as in Table 5.3. COSTON of state A = 5 due to the RESET signal. COSTON of state D = 4 due to the inversion of S2 S1 S0 and the transition of ACC PHIT into the node. COSTOFF of state B = 1 because RDY ($FP + \overline{FP}$) = RDY.
2. With respect to those worst case costs evaluations, by RULE 1, state A is selected as having the highest cost and is assigned a code with most zeros - 000.
3. Next node D is taken as having second highest TOTAL cost. RULE 1 is applied again and the code 010 is arbitrarily chosen.
4. Next node C is considered (B, and C have the same TOTAL, COSTON and COSTOFF costs and numbers of neighbors, but C has more adjacent arrows). RULE 2 is applied on variable R1. Thus the code 110 is chosen.
5. Next node B is considered. RULE 1 is applied and the code 001 is created.
6. Lastly, node E is taken. RULE 3 is applied and the code 100 is generated. This way, all codes from Table 5.3 have been generated.

The equations are listed below and it can be seen that the maximum number of sum terms for each variable is four, compared to six from the Grey Code assignment above. This will give a better chance of fitting the device. The listing of the equations after using these rules is presented in Fig. 5.12.

6. CONCLUSION.

AE was able to find very good quality solutions for both small and large machines. In many cases the results better than those from Grey Code, Moroz, Kiss, Mustang and Stash were generated. Several state assignments were found with AE for machines of more than 100 states. Since it is written in Lisp, AE is much slower than all those programs, but it does also more job. It is integrated with state minimization and other high-level tools. The quality of PLD fitting ("Can the assigned circuit be fitted or not?") in in our opinion more important than the speed of the program.

The processing times of AE range from **about a minute to several hours,** because the novel approach of this program is to give the user the *trade-off* between the speed of optimization and its quality. For instance, in one experiment the 34 state machine was assigned in 10 hours (real time). 100 iterations were executed. 3 solutions, each of them better than the previous one, were found in the first 6 iterations and next all solutions were worse. Because of such method, we think that we can claim that the best solution found is the optimal or quasi-optimal one (this is, however, of course not the proof). The solution that was only one term worse than the "optimal" solution was found after about three minutes. This proves the quality of our heuristics. The time of 10 hours is mentioned here only as an **example** of the "trade-off" abilities of the program and the proof of the quality of its heuristics. It does not imply in any way that on has to run the program 10 hours on a 34-state machine.

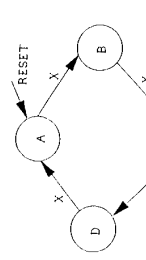

Fig. 5.7. *Reset signal.*

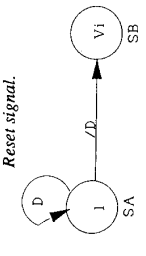

Fig. 5.8. *Rule 2.*

Fig. 5.9. *Rule 3.*

Table 5.1. *The D-type flip-flop the transition table.*

D\Q	0	1
0	0	0
1	1	1

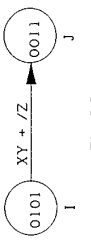

Fig. 5.5. *Transition equations.*

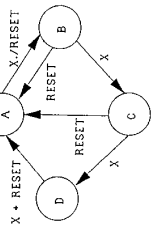

Fig. 5.6. *Rule 1.*

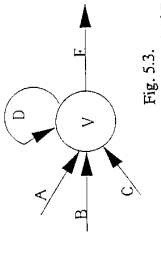

Fig. 5.3. *COSTON and COSTOFF.*

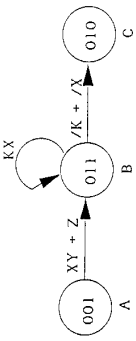

Fig. 5.4. *COSTON and COSTOFF calculation.*

State Vars	COSTON	COSTOFF
V_2	0	0
V_1	3	2
V_0	3	2

Table 5.0.

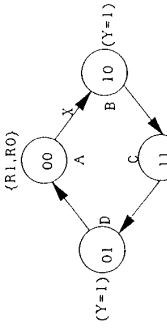

Fig. 5.1. *Combinational output.*

Fig. 5.2. *Registered output.*

State	COSTON	COSTOFF	TOTAL	CODE
A	5	1	6	000
B	2	1	3	001
C	2	1	3	110
D	4	1	5	010
E	1	1	2	100

Table 5.3. *The COSTON and COSTOFF calculations.*

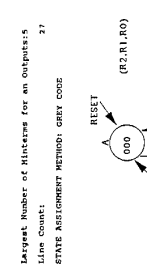

Fig. 5.12. *The listing of the equations.*

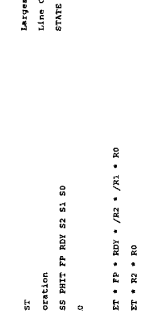

Fig. 5.10. *Grey Code state assignment.*

Fig. 5.11. *Rule based state asignment.*