Bitwise Encoding of Finite State Machines

José Monteiro, James Kukula, Srinivas Devadas
Department of EECS
MIT, Cambridge

Horácio Neto
Department of EECS
IST, Portugal

Abstract

We propose an innovative method of encoding the states of finite state machines. Our approach consists of iteratively defining the code word, one bit at a time. In each iteration the input state machine is decomposed into two submachines, with the first submachine having only two states. One bit is therefore sufficient to encode this submachine and it can be assigned arbitrarily as the particular value it assumes for each state is of minimal influence in terms of the machine implementation. The process is repeated again having as input the second submachine, until all the bits are encoded.

We provide experimental results which indicate that our method of iteratively defining one bit at a time can generally achieve superior results to existing sequential state assignment methods which try to solve large problems heuristically.

1 Introduction

The automatic synthesis of logical circuits from a higher level description has been subject of extensive investigation as it constitutes an effective way of dealing with the ever increasing complexity of integrated digital circuit projects. A fundamental part of these projects is made of sequential circuits (or finite state machines, FSM). The synthesis of this class of circuits involves some specific problems requiring special attention.

In the specification of a FSM the states are generally identified through mnemonics. To implement the machine as a logic circuit it is necessary to assign each of these mnemonics a unique binary code. This process is called *state encoding* and is the subject addressed by the present work.

Earlier encoding systems were based on algebraic approaches [6, 2, 1, 5]. These systems could only handle FSMs with a very limited number of states. More recent encoding systems (`Nova` [12], `Asyl` [11], `Mustang` [4]) use heuristic algorithms relying on cost functions to direct them to good solutions and are able to handle much larger FSMs. These systems attempt a global optimization by defining the whole code word for one state based on the codes of the already encoded states.

In this work we propose a different approach to the problem [9]. Instead of defining entire code words one state at a time, we define one bit of all the state codes at a time. Thus instead of aiming at a global optimization we try several consecutive local optimizations. We will show that a better final solution can be achieved, as each local optimization can be done more efficiently (because we are viewing a restricted space of solutions).

The definition of each bit corresponds to the decomposition of the FSM into two submachines. We restrict the number of states of one of these submachines to be two so that the encoding of this submachine is just the assignment of one bit of the original FSM's code word. The other submachine serves as input to the next iteration of this same process, until it too has only two states.

At each iteration, the machine decomposition is accomplished by defining pairs of states and this is the crucial operation in the system. State pairing is first attempted through an algebraic algorithm that tries to perform a serial decomposition. Serial decompositions generally lead to much better solutions, but they are not possible for most FSMs without increasing the length of the code word. Thus in the generic case, where no serial decomposition is possible, a general decomposition is performed and state pairs are defined by an heuristic algorithm that tries to minimize the complexity (*i.e.*, number of edges in the state graph) of the two submachines.

In section 2 we explain the algorithm of the proposed system in more detail. The two following sections, 3 and 4, describe respectively the algebraic and the heuristic approaches for the FSM decomposition. Section 5 presents some results obtained with this system and compares them with other well known encoding systems.

2 Description of the Algorithm

FSM decomposition has been proposed as a way to divide a complex problem into several simpler problems [3]. The set of interacting submachines has the same external behavior as the original lumped FSM. The state of the whole system is determined by the states of all the submachines. In terms of code words of the states, we can think of the decomposition as defining which part of the state code is generated by each machine.

The method described in this paper can be seen as a limit case of this approach which decomposes a FSM with n states into b two-state submachines, where $2^{b-1} < n \leq 2^b$ [9]. Each of these submachines is then responsible for the generation of one bit of the code word of the global FSM.

Assuming that the memory elements that store the state of the FSM provide both the value and its negation, assigning 0 or 1 to a state of one submachine (corresponds to inverting the same bit in all the code words of a FSM) does not affect the combinational logic of the FSM [8]. Thus the encoding of each submachine can be done arbitrarily and no generic encoding system is needed.

The process of determining the b submachines is iterative. The FSM decomposition is always done into two submachines, one with two states and the other with, generally, half the states of the original FSM. This submachine serves as input to the next iteration, until it too has only two states. This procedure is presented in algorithm 2.

Bitwise Encoding

```
headFSM = initialFSM ;
while( numberOfStates( headFSM ) > 2 ) {
    statePairing = serialDecomp( headFSM ) ;
    if( not ( statePairing ) ) {
        statePairing =
                generalDecomp( headFSM ) ;
    }
    headFSM = buildNextFSM( statePairing ) ;
    encode( tailFSM, $state_0$ ) = 0;
    encode( tailFSM, $state_1$ ) = 1;
}
encode( headFSM, $state_0$ ) = 0;
encode( headFSM, $state_1$ ) = 1;
```

In each iteration of this algorithm we create a *cover* [6] where each block has at most two states. Although we call these blocks state pairs, there can be blocks with only one state, but as we are targeting minimum length code words, the number of blocks must allow the reduction of one bit for the code word of the FSM for the next iteration (if the original FSM has n states, $2^{b-1} < n \leq 2^b$, then the maximum number of blocks is 2^{b-1}).

It can be observed that we first attempt a serial decomposition. This is not always possible, but when possible leads to a great improvement in the quality of the solution. Considering a maximum number of states per block of two simplifies the search for *preserved covers* [6] and an efficient algorithm has been developed that takes into account the specificity of this decomposition (section 3). Thus, even though for certain FSMs serial decompositions may not exist, it is worth to check for them given that the time overhead is negligible and that the possible gains can be very high.

When no useful serial decomposition is possible, a general decomposition is performed. The pairing of the states is based on the similarities of the transitions between them. Generating a pair of states that have one (or more) edge to/from the same state under the same inputs means that in the submachine these edges will become one. This heuristic algorithm is presented in section 4.

3 Algebraic Approach: Serial Decomposition

Finding a serial decomposition involves determining a preserved cover that defines the *head* submachine. Ambiguity trees are an efficient method to search for preserved covers [6].

In the current implementation of bitwise decomposition, the *tail* submachine corresponds to the two-state submachine (one interesting avenue for research is to make the two-state submachine the *head* submachine and see if it is easier to obtain serial decompositions). We use the restriction that we are only interested in covers formed by blocks with at most two states to improve the efficiency of the search for preserved covers for this particular case.

Instead of beginning the ambiguity tree with a block that includes all states of the FSM, we build an ambiguity tree for each possible state pair using each as the root block. The result is a preserved cover associated with each state pair and what this means is as follows: if we choose the state pair of the root block for our cover, then we must also include the remaining pairs of its associated cover.

If the number of blocks (state pairs) of the cover is such that the length of the code word cannot be met, then this cover is simply discarded. On the other hand, the number of state pairs generated in the ambiguity tree associated with one state pair may be too small, meaning that, because the states not contained in any of these pairs form blocks by themselves, the number of blocks is again too high for the code word length chosen.

For instance, consider a FSM with states {`A`, `B`, `C`, `D`} where the pair (`A`, `B`) generates no other pair, the same for (`C`, `D`) and that all other pairs imply too many pairs. We have two preserved covers, {(`A`, `B`); (`C`); (`D`)} and {(`A`); (`B`); (`C`, `D`)}, but neither is of any use, because we cannot encode both the *head* and *tail* submachines with one bit.

It is shown in [6] that the combination of preserved covers is still a preserved cover. So the covers with a number of state pairs that still permit the reduction of the code word of the submachine for the next iteration are kept and combined in the end. Returning to the example, the two preserved covers found can be combined, giving the cover {(`A`, `B`); (`C`, `D`)}. This cover can be used to serially decompose the machine and encode each of the submachines with one bit.

This method of searching for preserved covers is exhaustive, that is, if no cover is found, then the FSM does not have a two-way serial decomposition in which one of the submachines has two states. Still, it is typically much faster than the generic ambiguity tree approach. Although many ambiguity trees must be calculated in our approach, they are very simple as each starts with only two states, are closer to the termination conditions (that stop expanding the tree), and finally because we know the possible maximum number of blocks, it is possible to stop expanding the tree before reaching the end.

If more than one good preserved cover if found, the best one must be chosen. We are using the two-level minimizer `Espresso` [10] to estimate the complexity of the submachine that each cover implies. This is practical because the number of preserved covers found is generally small, and the minimization is fast.

4 Heuristic Approach: General Decomposition

In the general case where no serial decomposition of the FSM is found, we have to resort to using a general decomposition. Its format must have the same constraints, that is, a decomposition into two submachines, one of them with only two states. The aim is still to find a cover formed by pairs of states, but this time we use heuristics to define the cover.

The value of the cost function for the heuristic is based on *transition pairs* (initially defined in [7]). A list of pairs formed by combinations of transitions between states of the FSM is created and a heuristic value is associated with each. This value is a measure of the similarity between the input combinations that trigger each transition. The objective is to select the transition pairs that maximize the sum of the heuristic values.

Each transition pair defines two state pairs, one state pair for each transition, placing the present states as the first elements of the pairs and the next states as the second elements. For instance, the transition pair ($\mathtt{A}\rightarrow\mathtt{B}$, $\mathtt{C}\rightarrow\mathtt{D}$) defines the state pairs (A, C) and (B, D).

The motivation behind this approach is that this way we are simplifying the two-state submachine. Using the same transition pair as example, transitions $\mathtt{A}\rightarrow\mathtt{B}$ and $\mathtt{C}\rightarrow\mathtt{D}$ have similar input combinations (since it was chosen) so the corresponding transition in the *head* submachine (A, C)$\rightarrow$(B, D) can be merged, while in the two-state submachine these transitions just disappear. A slightly different approach could be used where the states in each transition of the transition pair form a state pair, i.e. ($\mathtt{A}\rightarrow\mathtt{B}$, $\mathtt{C}\rightarrow\mathtt{D}$) $\Rightarrow$ (A, B) and (C, D), which would lead to no transitions in the *head* submachine and a merge in the two-state submachine – the results are similar.

Not all transitions can be combined to form transition pairs. One state cannot be the present state in one transition of the pair and the next state in the other, as this would mean that the state would be in both positions of state pairs, corresponding to two different logic values for the encoding bit. Although there is no intrinsic constraint, transition pairs where a state appears twice to/from different states are also not allowed so as to reduce the number of possible combinations (we reduce computational complexity by elimination of solutions that are probably not good).

The same rules given above apply when determining which transition pairs can be merged into a given cover. The state pairs defined by the transition pair have to satisfy either of the following two properties: (1) Neither state in the pair is contained in any state pair that belongs to the cover or (2) the state pair is already in the cover.

The selection of the best set of transition pairs is a computationally complex problem and an exhaustive search cannot generally be accomplished within reasonable time. Thus we use an iterative approach to this problem.

After having generated and evaluated all possible transition pairs for the FSM, the pair with the highest heuristic value is picked. Then, all other transition pairs, sorted by descending heuristic value, are checked to see that the state pairs that they form do not conflict with the state pairs already in the present cover, in which case they are merged into the cover. When there are no more transition pairs left, states that are not present in the cover are randomly joined in pairs, and added to the cover.

The cover found is then evaluated, again using `Espresso` [10], and is kept if it is the best found so far.

This process is repeated a number of user defined times. At the end of each iteration, the heuristic values of the transition pairs are updated based on how good the minimized implementation obtained is. If a good implementation is obtained, then the transition pairs used to build the cover will have their heuristic values incremented, so the probability of being used again in future iterations is increased (and vice-versa in case of a bad implementation).

5 Experimental Results

To present experimental results, we have divided the FSMs into three different categories: pipelines, counters and general FSMs. Pipelines and counters with modulus power of two are particular cases of FSMs where serial decomposition can be performed in each iteration. They are included to show both the advantages of determining and performing serial decomposition instead of a general decomposition and also to give an idea of how far existing encoding tools are from obtaining close to optimum solutions.

Pipelines and counters were described in terms of state transition tables, just like any other FSM. The results obtained for these machines are given is table 1. The name `pipe1x`N corresponds to a pipeline with N memory elements in series, thus corresponding to a FSM with 2^N states and `c`N corresponds to a counter of module N (equals to the number of states of the FSM).

First we note that good preserved covers were found in all iterations. So only serial decompositions were performed. The solutions achieved by the **Bitwise** system are much better than those obtained by the remaining encoding systems. Further we can also observe that as the number of states of the FSM increases, the heuristic solutions keep getting worse.

Table 2 presents the results obtained for general FSMs from the MCNC benchmark set. Only two preserved covers were found among all the iterations of decomposition for the FSMs, one in the example `bbara` and another in `planet2`. In the case of `planet2`, the solution is significantly better with **Bitwise** than any other encoding system because a serial decomposition (algebraic calculation) was accomplished in the first iteration. The result did not improve much for `bbara` as this is a smaller FSM and the serial decomposition was found only on the second iteration.

For the remaining FSMs, only general decompositions were performed, but the results obtained with the **Bitwise** heuristic algorithm are still generally superior to the ones from the other encoding tools.

Circuit Name	2-level: # Terms in SOP			Multilevel: # of Literals			
	Mustang	Nova	**Bitwise**	Mustang	Nova	Industrial	**Bitwise**
pipe1x4	26	12	5	65	23	19	2
pipe1x5	47	23	14	124	61	57	20
pipe1x6	101	52	23	292	154	145	43
pipe1x7	212	103	39	635	308	328	71
c16	19	17	14	42	43	32	21
c32	31	32	21	82	94	62	23
c64	57	57	28	170	176	106	30
c128	107	114	36	331	361	217	44

Table 1: Results for Pipelines and Counters.

Circuit Name	2-level: # Terms in SOP			Multilevel: # of Literals			
	Mustang	Nova	**Bitwise**	Mustang	Nova	Industrial	**Bitwise**
arbseq	123	59	47	379	194	197	113
bbara	26	24	26	44	46	45	53
lion	10	9	8	20	29	22	22
pgm	46	38	29	198	178	157	117
planet	107	87	79	542	458	361	441
planet2	205	167	85	1087	885	731	448
sand	105	97	87	431	432	385	366
stack	32	13	8	95	23	21	19
styr	115	94	73	442	441	334	279

Table 2: Results for general FSMs.

References

[1] D. B. Armstrong. On the efficient assignment of internal codes to sequential machines. *IRE Transactions on Electronic Computers*, 11:611–622, October 1962.

[2] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. *IRE Transactions on Electronic Computers*, 11:466–472, August 1962.

[3] Pranav Ashar, Srinivas Devadas, and A. Richard Newton. *Sequential Logic Synthesis.* Kluwer Academic Publishers, 1992.

[4] S. Devadas, H. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: State assignment of finite state machines targeting multilevel logic implementations. *IEEE Transactions on Computer Aided Design*, 7(12):1290–1300, December 1988.

[5] T. A. Dolotta and E. J. McCluskey. The coding of internal states of sequential circuits. *IRE Transactions on Electronic Computers*, 13:549–562, October 1964.

[6] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Inc, 1966.

[7] James Kukula and Srinivas Devadas. Finite state machine decomposition by transition pairing. In *IEEE International Conference on Computer-Aided Design*, pages 414–417, November 1991.

[8] E. J. McCluskey and S. H. Unger. A note on the number of internal variable assignments for sequential switching circuits. *IRE Transactions on Electronic Computers*, 8:439–440, December 1959.

[9] José C. Monteiro. "Finite state machine encoding in automatic synthesis of logic circuits" (in portuguese). Master's thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 1992.

[10] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer Aided Design*, 6(5):727–750, September 1987.

[11] Gabrièle Saucier, Michel Paulet, and Pascal Sicard. Asyl: A rule-based system for controller synthesis. *IEEE Transactions on Computer Aided Design*, 6(6):1088–1097, November 1987.

[12] Tiziano Villa and A. Sangiovanni-Vincentelli. Nova: State assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer Aided Design*, 9(9):905–924, September 1990.