Synthesis and Implementation of RAM-Based Finite State Machines in FPGAs

Valery Sklyarov

Department of Electronics and Telecommunications
University of Aveiro
Portugal
skl@inesca.pt

Abstract. This paper discusses the design and implementation of finite state machines (FSM) with combinational circuits that are built primarily from RAM blocks. It suggests a novel state assignment technique, based on fuzzy codes, that is combined with the replacement (encoding) of the FSM input vectors. It also shows how FSMs with dynamically modifiable functionality can be constructed and then implemented in commercially available FPGAs. The results of experiments have shown that FSMs with the proposed architecture can be implemented using less hardware resources, such as the number of FPGA configurable logic blocks (CLB), while at the same time extending their functional capabilities.

1 Introduction

The primary architectures (models) of FSMs usually have simple and regular structures at the top level [1]. They are considered to be a composite of a combinational scheme and memory. The former calculates new states in state transitions and forms outputs, and the latter is used to store states.

The top-level architecture can generally be reused for different applications. However, the combinational scheme is typically very irregular, and its implementation depends on the particular unique behavioral specification for a given FSM. Since such specifications vary from one implementation to another, we cannot construct a reusable FSM circuit. Note that there are some techniques (for example, microprogramming) and technology dependent design methodologies (for instance, those aimed at dynamically modifiable field programmable devices such as the Xilinx XC6200 family) that do allow reusable parameterizable templates to be built for FSM-based applications. However these approaches have many limitations, such as restricting the number of inputs that can affect state transitions in the FSM, requiring a considerable amount of RAM (ROM) for the combinational circuit, and necessitating rather complex circuits and time-consuming procedures for dynamic modifications to the logic and interconnections.

This paper presents an approach to the design of FSMs that is based on the use of RAM blocks, but of a modest size. It eliminates many of the restrictions that apply to traditional FSM architectures and current methods for FSM synthesis. Since very fast RAM-based CLBs are available within widely used and relatively cheap FPGAs such as the XC4000XL of Xilinx, we can apply the proposed technique directly to the design of FPGA-based circuits. Such an implementation is very efficient. Since each CLB can be configured to be 16x2 or 32x1 RAM (ROM), and its speed is comparable with ordinary logic, with the aid of the proposed technique we can implement very

regular and very fast dynamically modifiable circuits, something which is generally impossible using traditional approaches.

2 The Architecture of RAM-Based FSMs

The set of state transitions for any FSM can be presented in the form: $a_{to} = a_{from}X(a_{from},a_{to})$, where a_{from} and a_{to} are the initial and next state in the state transition, $X(a_{from},a_{to})$ is a product of the input variables from the set $X=\{x_1,...,x_L\}$ that cause the transition from a_{from} to a_{to}, L is the number of FSM inputs, $a_{from},a_{to} \in A=\{a_0,...,a_{M-1}\}$, A is the set of FSM states, and M is the number of states. The FSM generates outputs from the set $Y=\{y_1,...,y_N\}$ where N is the number of FSM outputs.

The structural model of a FSM assumes that all states are coded, and the size R of the code varies from the value $int\log_2 M$ (for binary state encoding) to M (for one-hot state encoding). Thus a RAM-based combinational circuit has R+L inputs and R+N outputs, and the size of RAM is equal to $2^{R+L}(R+N)$ bits. Even for binary state encoding and modest values of L and N, except for some trivial cases this size becomes very large.

Let's consider an example. Fig. 1,a shows a state transition graph for a FSM where M=5, R_{min}=3, and L=2. Fig. 1,b depicts a trivial RAM-based implementation of this FSM, and the RAM size is equal to $2^5(3+N)$. One way to decrease the memory required is to split the conditional state transitions using two additional intermediate states. However, any conditional state transition will then require twice the time, and the RAM required is still $2^4(3+N)$. We will also need additional hardware to multiplex the input variables x_1 and x_2 to the same RAM input.

The proposed approach is based on the so-called *fuzzy state encoding technique*, and it enables us to improve the implementation of FSMs, even for the trivial case we have just considered.

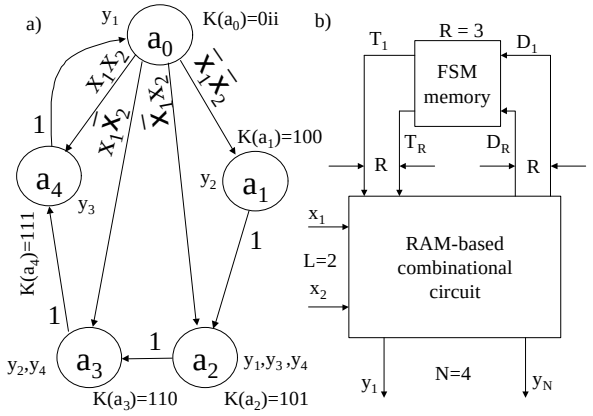

Fig. 1. State transition graph (a) and a feasible top-level structure of a RAM-based FSM (b)

Let us define $K(a_m)$ as the code for the state a_m. A code is *fuzzy* if its size is not fixed for all m. For example, the code of the state a_m might be 0ii, where the character "i" indicates that the bit must be ignored and is considered in a specific way. Note that "i" does not denote “don’t care”, which is designated as "-".

In order to distinguish the codes for different states, these codes must be orthogonal. This means that for any two codes, there is at least one bit position that is equal to 1 (0) in one code and 0 (1) in the other. For example, all the codes $K(a_0),...,K(a_4)$, shown in fig. 1,a are orthogonal. The size of the code $K(a_0)$ is equal to 1 and the size of all the other codes is equal to 3. Since bits 2 and 3 in the code $K(a_0)$ are not used for representing the code, we can employ them to identify transitions from the state a_0 in such a way that they are (see fig. 1): $a_0(\overline{T}_1\overline{x}_1\overline{x}_2)\Rightarrow a_1$, $a_0(\overline{T}_1\overline{x}_1x_2)\Rightarrow a_2$, $a_0(\overline{T}_1x_1\overline{x}_2)\Rightarrow a_3$, $a_0(\overline{T}_1x_1x_2)\Rightarrow a_4$. Now if the FSM is set in one of the states $\{a_1,a_2,a_3,a_4\}$, we can use the bits T_2 and T_3 from the FSM memory (see fig. 1,b) to identify the corresponding code. If the FSM is in the state a_0, the code $K(a_0)$ can be identified by just one bit T_1, and since the remaining bits are not needed in a_0 we can use the corresponding RAM inputs to distinguish conditional state transitions. Fig. 2 presents the complete implementation of the FSM scheme with the behavior given in fig. 1,a. Two extra blocks, $C(T_2,x_1)$ and $C(T_3,x_2)$, provide predefined selections of the signals between the parentheses, and send them to the RAM inputs T_2^* and T_3^*.

The idea that we considered above can be used to design RAM-based FSMs, and the synthesis of such FSMs involves the following two steps: 1) the replacement of input variables (encoding of input vectors) such that the number of inputs, L, for the RAM-based circuit is reduced; 2) state encoding that defines fuzzy codes for the FSM states that satisfy our requirements.

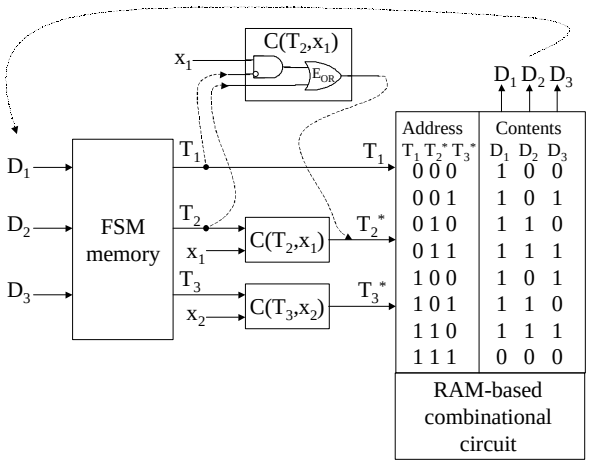

Fig. 2. Implementation of the FSM with fuzzy state codes

3 Replacement of Input Variables

The primary method for input variable replacement was considered in [2]. In this case, the combinational scheme of a FSM is composed of two sub-schemes. The first sub-scheme makes it possible to replace external input variables from the set X with new variables from the set $P = \{P_1,...,P_G\}$, and for many applications G << L. It performs the following function:

$$P(a_{from}, a_{to}) = \rho(a_{from}, X(a_{from}, a_{to})),$$

where $P(a_{from},a_{to})$ is the vector which can be used in the state a_{from} instead of the input vector $X(a_{from},a_{to})$, and ρ is a conversion function [2]. Since this is presented in detail in [2,3], we will just consider an example here that will be used later to illustrate our approach (see table 1, where for the sake of clarity we have omitted the output variables and they can be generated just as for the Moore FSM). The column $P_B(a_{from},a_{to})$ contains the variables P_1, P_2, P_3 that can be used instead of the variables $x_1,...,x_8$ after applying the method [2]. The conversions required are represented by the following Boolean functions: $P_1=a_1x_1 \vee a_2x_6 \vee a_5x_6$, $P_2=a_1x_2 \vee a_2x_7 \vee a_5x_4$, $P_3=a_1x_3 \vee a_2x_8 \vee a_5x_5$. Such a replacement enables very efficient implementations of functions such as P_1, P_2, and P_3, to be provided on multiplexers [3]. Another kind of replacement (which differs from [2,3]) is based on an encoding technique (see the column $P(a_{from},a_{to})$). However in this case the circuit that implements the Boolean functions $P_1,...,P_G$ would be less regular.

Table 1. Structural table of FSM

a_{from}	$K(a_{from})$	$a_{to} \Leftarrow AC$	$K(a_{to})$	$K(c_{from}^{to})$	$X(a_{from},a_{to})$	$P_B(a_{from},a_{to})$	$P(a_{from},a_{to})$
a_0	0000	a_1	0100	0000	1	not valid	not valid
a_1	01ii	$a_3 \Leftarrow c_1^3$ $a_3 \Leftarrow c_1^3$ $a_2 \Leftarrow c_1^2$ $a_4 \Leftarrow c_1^4$	0001 0001 1000 0011	0100 0100 0101 0110	$\overline{x}_1$ $x_1\overline{x}_2\overline{x}_3$ x_1x_2 $x_1\overline{x}_2x_3$	$\overline{P}_1$ $P_1\overline{P}_2\overline{P}_3$ P_1P_2 $P_1\overline{P}_2P_3$	$\overline{P}_3\overline{P}_4$ $\overline{P}_3\overline{P}_4$ $\overline{P}_3P_4$ $P_3\overline{P}_4$
a_2	10ii	$a_1 \Leftarrow c_2^1$ $a_5 \Leftarrow c_2^5$ $a_6 \Leftarrow c_2^6$ $a_9 \Leftarrow c_2^9$	0100 1001 0010 1110	1010 1001 1000 1011	$\overline{x}_7$ $\overline{x}_6x_7x_8$ $x_7\overline{x}_8$ $x_6x_7x_8$	$\overline{P}_2$ $\overline{P}_1P_2P_3$ $P_2\overline{P}_3$ $P_1P_2P_3$	$P_3\overline{P}_4$ $\overline{P}_3P_4$ $\overline{P}_3\overline{P}_4$ P_3P_4
a_3	0001	a_7	0111	0001	1	not valid	not valid
a_4	0011	a_5	1001	0011	1	not valid	not valid
a_5	1ii1	$a_5 \Leftarrow c_5^5$ $a_7 \Leftarrow c_5^7$ $a_8 \Leftarrow c_5^8$ $a_9 \Leftarrow c_5^9$	1001 0111 1100 1110	1001 1101 1111 1011	$x_4\overline{x}_5\overline{x}_6$ $\overline{x}_4$ x_4x_5 $x_4\overline{x}_5x_6$	$\overline{P}_1P_2\overline{P}_3$ $\overline{P}_2$ P_2P_3 $P_1P_2\overline{P}_3$	$\overline{P}_2\overline{P}_3$ $P_2\overline{P}_3$ P_2P_3 $\overline{P}_2P_3$
a_6	0010	a_0	0000	0010	1	not valid	not valid
a_7	0111	a_8	1100	0111	1	not valid	not valid
a_8	1100	a_0	0000	1100	1	not valid	not valid
a_9	1110	a_0	0000	1110	1	not valid	not valid

4 State Encoding

Consider the main idea of state encoding. Suppose that the code $K(a_1)$ of the state a_1 is 01ii. Since in table 1 there are three transitions from a_1, to different states a_3, a_2 and a_4, we can code these transitions as $[01(i=0)(i=0)] \Rightarrow a_3$, $[01(i=0)(i=1)] \Rightarrow a_2$, and $[01(i=1)(i=0)] \Rightarrow a_4$. If the results of replacement are presented in the column $P(a_{from},a_{to})$ of table 1, then the 3rd and the 4th outputs of the FSM memory will be mixed with the variables P_3P_4 by means of OR gates (see elements E_{OR} in fig. 2). In this case if the FSM is in the state a_1, the outputs T_3 and T_4 must be set to zero.

However, if we replace E_{OR} with E_{XOR}, which performs an XOR function, the outputs T_3 and T_4 could also be set to non zero values and we would obtain the same results by a trivial reprogramming of the RAM.

Let us designate $A(a_{from})$ as the subset of states that follow the state a_{from} and suppose that $k_{from} = |A(a_{from})|$. For our example in table 1 we have $A(a_0) = \{a_1\}$, $k_0=1$, $A(a_1) = \{a_3,a_2,a_4\}$, $k_1=3$, $A(a_2) = \{a_1,a_5,a_6,a_9\}$, $k_2=4$, etc. If $k_{max}=max(k_0,k_1,k_2,...)$, then $G_{min} = int\log_2 k_{max}$, where G_{min} is the minimum number of variables from the set P that affect individual transitions. For our example in table 1 $G_{min} = 2$ (see column $P(a_{from},a_{to})$).

Consider any subset $A(a_m) = \{a_{m1},a_{m2},...\}$, m=0,...,M-1, such that $k_m>1$. If $k_m=1$ then the code, $K(a_m)$, of a_m can be considered to be the RAM address and it can be directly used for the unconditional transition from a_m to $A(a_m)$. We want to generate codes for the states $a_{m1},a_{m2},...$ on the outputs of the FSM RAM so we can set the following correspondence: $c_m^{m1} \Rightarrow a_{m1}$, $c_m^{m2} \Rightarrow a_{m2},...$, where c_m^{m1}, $c_m^{m2},...$ are the symbols that correspond to the respective RAM address codes (AC) that will be designated as $K(c_m^{m1}),K(c_m^{m2}),...$. Let us also agree to call c_m^{m1}, $c_m^{m2},...$ ACs when there is no possible ambiguity. Finally we can build new subsets, which are $C(a_m) = \{c_m^{m1},c_m^{m2},...\}$, m=0,...,M-1. It is obvious that the following correspondence exists: $[A(a_m)=\{a_{m1},a_{m2},...\}] \Leftrightarrow [C(a_m)=\{ c_m^{m1},c_m^{m2},...\}]$, i.e. for each element a_{mi} there exists an address code c_m^{mi} and RAM provides the conversion $c_m^{mi} \Rightarrow a_{mi}$. In fact a_{mi} is the code written at address c_m^{mi} in RAM. Now we can formulate the target requirements for the encoding:

1. All ACs $c_m^{m1},c_m^{m2},...$ (m=0,...,M-1) with different superscripts must be unique (because they cause transitions to different states). As a result we can formulate the following requirement: $\forall A(a_m) \cap A(a_s)=\varnothing \ \exists \ \{K(c_m^{m1}),K(c_m^{m2}),...\}$ *ort* $\{K(c_s^{s1}),K(c_s^{s2}),...\}$, where *ort* is the relationship of orthogonality considered above and any element of the first set must be orthogonal to any element of the second set. Here $K(c_{from}^{to})$ is the binary code of c_{from}^{to} (see table 1). ;
2. ACs with the same superscript could be the same (because they cause transitions to the same states). Hence, if $A(a_m) \cap A(a_s) \neq \varnothing$, $\{K(c_m^{m1}),K(c_m^{m2}),...\}$ *ins* $\{K(c_s^{s1}),K(c_s^{s2}),...\}$ is allowed, where *ins* is the relationship of intersection (or non-orthogonality): $\{c_m^{m1},c_m^{m2},...\}$ *ins* $\{c_s^{s1},c_s^{s2},...\} \Leftrightarrow \{c_m^{m1},c_m^{m2},...\}$ ~~*ort*~~ $\{c_s^{s1},c_s^{s2},...\}$;
3. The predefined size, S, of the resulting codes is equal to R (S=R). In some cases S may be greater then R but we want to find the minimum value of S ($S \geq R$).
4. Variables $c_m^{m1},c_m^{m2},...$ in each individual subset $C(a_m)$, $k_m>1$, m=0,...,M-1, must be encoded in such a way that a maximum number of their bits with coincident indexes have constant values that are the same for all variables $c_m^{m1},c_m^{m2},...$ from $C(a_m)$.

The ACs that satisfy the requirements considered above can be obtained with the aid of a slightly modified encoding algorithm [3], which allows for a combinational circuit of a FSM to reduce the functional dependency of outputs on inputs. This is based on the iterative placement of the symbols that must be coded in special tables that look like Karnaugh maps. This method permits the encoding procedure to be carried out for quite complicated FSMs. The results of this step for our example (see table 1) are shown in fig. 3,a. Next we find the codes for the states a_m, such that $k_m>1$. They are: $K(a_1)=01\mathbf{ii}$ (see the squares c_1^3,c_1^2,c_1^4), $K(a_2)=10\mathbf{ii}$ (see the squares c_2^6,c_2^5,c_2^9,c_2^1), $K(a_5)=1\mathbf{ii}1$ (see the squares c_5^7,c_5^5,c_5^9,c_5^8). Here $A(a_1) \cap A(a_2)=\varnothing$ and

$A(a_1) \cap A(a_5) = \varnothing$. That is why $\{K(c_1^3), K(c_1^2), K(c_1^4)\}$ *ort* $\{K(c_2^6), K(c_2^5), K(c_2^9), K(c_2^1)\}$ and $\{K(c_1^3), K(c_1^2), K(c_1^4)\}$ *ort* $\{K(c_5^7), K(c_5^5), K(c_5^9), K(c_5^8)\}$ (see fig. 3,a). Since $A(a_2) \cap A(a_5) = \{a_5, a_9\} \neq \varnothing$, it is allowed $\{K(c_2^6), K(c_2^5), K(c_2^9), K(c_2^1)\}$ *ins* $\{K(c_5^7), K(c_5^5), K(c_5^9), K(c_5^8)\}$. That is why the segments $c_2^6, c_2^5, c_2^9, c_2^1$ and $c_5^7, c_5^5, c_5^9, c_5^8$ of the map in fig. 3,a have two shared squares marked with c_2^5/c_5^5 and c_2^9/c_5^9 and $\{K(c_2^6), K(c_2^5), K(c_2^9), K(c_2^1)\}$ *ins* $\{K(c_5^7), K(c_5^5), K(c_5^9), K(c_5^8)\}$ = {1101, 1111} (note that each shared square contains symbols c_m^s with equal superscripts s).

All non-encoded states just cause unconditional transitions (k_m=1), and they can be assigned any unused codes that are available in the second step. This is feasible because an address code for any unfilled square in fig. 3,a does not exist. It follows from the predefined requirements to be established for the encoding of input variables (see column $P(a_{from}, a_{to})$ of table 1). Let us consider, for example, the unused square 0111 of the map in fig. 3,a. It is easy to verify that this code cannot be generated on any transition from the state a_1 (see table 1) as well as on any other transition except the specified unconditional transition from the state a_7 (see fig. 3,b). If the map does not have sufficient room, the size, S, of the codes must be incremented and the map must be enlarged [3]. Thus the resulting value of S could be greater then R (which is, of course, undesirable). The final results of state encoding are shown in fig. 3, b.

Now the RAM has 4 inputs and 4+N outputs, and it must be programmed as follows (see columns $K(c_{from}^{to}) \Rightarrow K(a_{to})$): 0000⇒0100; 0100⇒0001; 0101⇒1000; 0110⇒0011; 1010⇒0100; 1001⇒1001; 1000⇒0010; 1011⇒1110; 0001⇒0111; 0011⇒1001; 1101⇒0111; 1111⇒1100; 0010⇒0000; 0111⇒1100; 1100⇒0000; 1110⇒0000. The Boolean functions $C(T_2, P_2)$, $C(T_3, P_3)$, $C(T_4, P_4)$ for additional blocks such as those shown in fig. 2, can be obtained directly from the columns a_{from}, $X(a_{from}, a_{to})$, $P(a_{from}, a_{to})$ of table 1. After trivial minimization they will be the following: $P_2 = \overline{T}_1 \overline{T}_2 (\overline{x}_4 \vee x_5)$, $P_3 = \overline{T}_1 \overline{T}_2 x_4 (x_5 \vee x_6) \vee \overline{T}_1 T_2 (\overline{x}_7 \vee x_6 x_8) \vee T_1 \overline{T}_2 x_1 \overline{x}_2 x_3$, $P_4 = T_1 \overline{T}_2 x_1 x_2 \vee \overline{T}_1 T_2 x_7 x_8$. Note that these functions are very simple and they are well suited to being implemented in widely used FPGAs, such as the Xilinx XC4000XL.

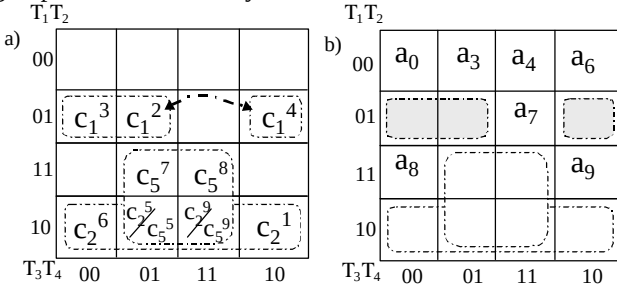

Figure 3. Maps that present the results of state encoding

The technique considered above includes many heuristic operations. Let us consider a method that enables us to solve this problem more formally. Consider the graph Λ, which reflects two following relationships α and β:

$$(c_m^s \; \alpha \; c_e^f) \Leftrightarrow (m = e); \quad (1)$$

$$(c_m^s \; \beta \; c_e^f) \Leftrightarrow (s = f). \quad (2)$$

Vertices of Λ correspond to symbols from the set $C(a_0), \ldots, C(a_{M-1})$ Two vertices c_m^s and c_e^f are connected with an edge if and only if $(c_m^s \; \alpha \; c_e^f)$ or $(c_m^s \; \beta \; c_e^f)$. We will

call the relationship (1) a hard relationship (because it strongly affects the results of state encoding) and the relationship (2) – a soft relationship (because it affects the quality of encoding but does not influence much to target requirements). That is why β edges will be shown in Λ with less thickness than α edges. Graph Λ for our example is shown in fig. 4,a. Let us designate $c_m=C(a_m)=\{c_m^{m1}, c_m^{m2},...\}$. All the vertices in each group c_m must be coded in such a way that permits our target requirements to be satisfied (see points 1-4 above). If $k_m = |\{c_m^{m1},c_m^{m2},...\}| = |A(a_m)|$ is the number of elements in c_m, then we have to use $h_m=\text{intlog}_2 k_m$ bits of code in order to distinguish all superscripts m1, m2,..., which cause transitions to different states. As a result we can consider h_m-cubes, which enables us to find out the address codes indicated by fuzzy positions of state codes that are designated by symbols i. For example, $c_1 = \{c_1^2, c_1^3, c_1^4\}$, $k_m = |\{ c_1^2, c_1^3, c_1^4\}| = 3$, $h_m=\text{intlog}_2 k_m= \text{intlog}_2 3 = 2$ and we have to consider 2-cubes, such as that can be represented by 4 squares of a Karnaugh map (or by 4 nodes of a 2-cube). If for a group c_m, $k_m<2^{hm}$, we will insert $2^{hm}-k_m$ dummy vertices and will connect them with the other vertices of c_m by dashed lines. For example, for the group c_1 in fig, 4,a ($k_m=3$)<($2^{hm}=2^2=4$). That is why we need to add $2^{hm}-k_m=4-3=1$ vertex, as shown in fig.4,b.

Now let us group all the vertices of Λ that correspond to symbols c_m^{m1}, c_m^{m2},... with the same subscript m, i.e. to the set c_m (see fig. 4,b, where the groups are encircled by dotted closed curves). Some groups may be connected (by edges) and other groups may be isolated. Fig. 4,c shows the new graph Γ that depicts the number of connections between the groups of Λ (see the number enclosed to the edge) and the number of vertices in each group of Λ (see the numbers enclosed to vertices).

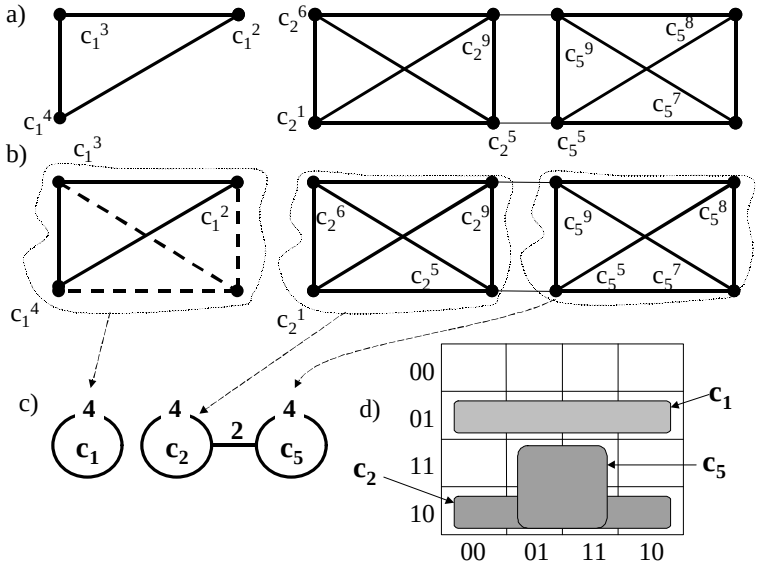

Figure 4. Graph Λ (a), extended graph Λ (b) , graph Γ (c) and encoding map (d)

Let us analyze the graphs Λ and Γ in more detail. Each vertex of Γ represents the respective group c_m of Λ, i.e. the group that has to be coded by the corresponding h_m-

cube. If two groups c_m and c_s (two vertices c_m and c_s of Γ) have been connected by an edge then they represent transitions to the same states and ACs for such transitions may be the same. The number $N(c_m,c_s)$ near the respective edge specifies the number of such transitions. So, if vertices c_m and c_s of Γ have been connected by an edge with $N(c_m,c_s)$ we need to accommodate c_m and c_s in such a way that: c_m is coded by h_m-cube; c_s is coded by h_s-cube; h_m-cube and h_s-cube have $N(c_m,c_s)$ intersecting (shared) squares. This is shown in fig.4,d, where c_1 is an isolated h_1-cube (i.e. 2-cube), c_2 and c_5 cubes that are intersecting by 2 squares, because $N(c_m,c_s)=2$ (see fig. 4,c). Finally we have to accommodate symbols c_i^j in the map (see fig. 4,d) in such a way that symbols with the same superscript will occupy the same square. This is trivial and we will obtain finally our previous table shown in fig. 3,a.

The method considered above is based on formal steps. A problem arises when we want to construct very complicated FSMs. In such cases the size of RAM becomes very large, and the circuit has a lot of redundancy. This problem can be overcome through a modular hierarchical specification of the FSM behavior such as that considered in [4]. This enables the description of the FSM to be decomposed into relatively autonomous fragments (sub-descriptions). Each fragment (module) can then be implemented in hardware using the approach we have discussed. The interaction mechanisms between the autonomous fragments were also considered in [4], and are based on the model of communicating FSMs with common stack memory.

4 Reusable Templates for RAM-Based FSM

Two blocks in fig. 2, the FSM memory and the RAM-based combinational circuit, are parameterizable and reusable. Since RAM is a dynamically modifiable unit, we can realize many run-time changes in the FSM behavior, even for our simple example in table 1 (for example, for given set $Y=\{y_1,...,y_N\}$ we can arbitrary redefine output values in different states for Moore FSM). In order to provide reusability for the entire FSM circuit, we must be able to program the functions of the additional blocks such as $C(T_2,x_1)$, $C(T_3,x_2)$ in fig. 2. However, for the functions $P_1,...,P_G$ (such as those shown in column $P_B(a_{from},a_{to})$ of table 1), the corresponding conversions can be provided by means of multiplexers controlled by the same RAM-based combinational circuit. This allows links between input variables from the set X and output variables from the set P to be programmed by dynamically modifying the RAM contents. This technique cannot be used directly for generating variables such as those shown in the column $P(a_{from},a_{to})$ of table 1. We can suggest two possible ways to solve this problem. First, it is possible to connect a RAM-based code converter (that transforms $P_B(a_{from},a_{to})$ to $P(a_{from},a_{to})$) to the multiplexer outputs. The second way assumes insignificant modifications to the method considered above. Suppose that all the states have been coded as shown in table 1, and K_f is the set of fuzzy codes. For our example we have $K_f=\{K(a_1), K(a_2), K(a_5)\}=\{01ii,10ii,1ii1\}$. Let us replace symbols i with symbols P_t having such t that shows the position of i in the code from left to right (starting from 1). For our example we can write the following sets of characters $01P_3P_4$, $10P_3P_4$ and $1P_2P_31$. Now we can change the indexes in the column $P_B(a_{from},a_{to})$ of table 1 in such a way that they will fit the respective bit positions in the subsets considered. For example, the symbols P_1,P_2,P_3 for the set $P_B(a_1,a_{to})=\{ P_B(a_1,a_2), P_B(a_1,a_3), P_B(a_1,a_4) \}$ could be re-indexed as follows $P_1 \Rightarrow P_3$, $P_2 \Rightarrow P_4$, $P_3 \Rightarrow P_5$. Thus new symbols P_3 and P_4 fit the third and fourth positions in the set $01P_3P_4$. The

new symbol P_5 can be used as a new bit for ACs. Unfortunately, in this case the size of the ACs has to be increased. This is the price of flexibility. Finally we will get: $P_B(a_1,a_2)=P_3P_4$, $P_B(a_1,a_3)=\overline{P}_3 \vee P_3\overline{P}_4\overline{P}_5$, $P_B(a_1,a_4)=P_3\overline{P}_4P_5$, $P_B(a_2,a_1)=P_4$, $P_B(a_2,a_5)=\overline{P}_3P_4P_5$, $P_B(a_2,a_6)=P_4\overline{P}_5$, $P_B(a_2,a_9)=P_3P_4P_5$, $P_B(a_5,a_5)=\overline{P}_2P_3\overline{P}_5$, $P_B(a_5,a_7)=\overline{P}_3$, $P_B(a_5,a_8)=P_3P_5$, $P_B(a_5,a_9)=P_2P_3\overline{P}_5$. Now we can implement our FSM based on RAM with the size of ACs equal to 5. The RAM becomes larger but we have been able to construct a dynamically modifiable circuit for the replacement of input variables. This circuit can be realized on programmable multiplexers [3] controlled by FSM RAM. Note that we still have two problems. Firstly, we might lose the flexibility of replacing non-encoded states, such as appeared in fig. 3,b. Since the size of the encoding map will be increased we could conceivably cope with this problem. Secondly, the outputs of a multiplexer-based circuit can be connected to the inputs of primary RAM in a different manner. This problem can be also solved in several ways. On the one hand we can build a RAM-based demultiplexer. On the other hand we can use pre-fixed connections by providing full set of multiplexers for the circuit that replaces variables. It is also possible to avoid flexible connections by setting up trivial constraints [3] for state encoding (for example, we can fix the number of bits with symbols i and their positions in all codes).

Thus the scheme of the FSM becomes fully dynamically reconfigurable, i.e. the functionality of the FSM can be changed after it has been implemented in hardware, even during run-time, by reloading the RAM blocks. Of course, each particular template has some constraints. However there are parameters for the template that can be altered. Thus the constraints can be changed by parameterization for the desired range of applications, much the same as changes of input/output numbers for RAM and ROM. Finally a library of templates can be created for FSMs having different characteristics, such as the values of L, N, M, $\max(k_m|\ m=0,...,M-1)$, etc.

5 Experimental Results

The proposed technique has been analyzed in several contexts. Firstly, we examined FSMs that are used in a variety of practical systems. This enabled us to estimate some parameters, such as the expectable range of G, the potential for behavioral specifications to be decomposed into smaller parts, and so on. Secondly, we compared the results of synthesis of the RAM-based FSM proposed in this paper, with the results obtained using known methods based on both binary and one-hot state encoding techniques.

All the experiments were performed with XS40 boards (XStend V1.2) of XESS that contain the Xilinx XC4010XL FPGA and the Xilinx Development System (series 1.5). Combinational circuits for FSMs were constructed from FPGA CLBs configured as RAM and ROM. For dynamic modifications we used dual port RAM library components, such as RAM 16X4D. Thus the first port took part in the FSM state transitions whilst the second port was used to reprogram RAM from the PC via the parallel port. Run-time modifiability is supported by software developed in Visual C++. Finally, FSMs with dynamically alterable behavior were implemented on statically configured FPGAs, such as the XC4010XL.

The reports of the experiments are summarized in table 2. The columns “Binary” and “One-hot” show the results obtained using the Xilinx Foundation Software. The average value of G from more than 50 practical FSMs is 2.7. Most of the FSMs were

decomposed into relatively autonomous modules to which we applied the proposed technique. In fact this granulation is quite natural, and arises from the modular nature of the specifications of the various operations controlled by the FSM [4]. We found that even in the case of partially hardwired circuits the majority of modifications that were needed in practice could be provided without redesigning the circuits. Note that in spite of the extended facilities, for all the examples the number of CLBs needed for the FSM circuits (see table 2) was less than that for the other methods of synthesis.

Table 2. The results of experiments

Name of example	Parameters of FSM (L/N/M/R/G)	Number of CLBs for FPGA XC4010XL		
		RAM-based FSM	Binary	One-hot
Proc.	15/10/47/6/3	60	96	166
Plot.	11/25/24/5/2	41	42	53
Teleav.	14/17/33/6/4	68	86	137
TechC	22/18/55/6/3	82	114	172
Conv	6/5/23/5/2	24	33	32
Vend.	7/4/18/5/2	22	30	28
TrafL.	3/7/11/4/2	9	15	12
LProc.	19/8/36/6/4	33	50	38
Abstr1.	8/0/10/4/2	8	14	10
Abstr2.	8/9/10/4/2	11	22	15

For each column “binary” and “one-hot” we synthesized the respective circuits using both available criteria (such as “optimized for area” and “optimized for speed”) and chose the best result. For some examples we could not obtain the final scheme using Xilinx synthesizers because the results of placement were negative. On the other hand in case of the proposed technique we were able to construct final circuits for all the examples.

6 Conclusion

We have described an approach to the design of RAM-based FSMs. It combines the state assignment technique proposed in the paper that allows the code for the states to be *fuzzy*, with the replacement (encoding) of input vectors. As a result, the structure of the FSM becomes well suited to implementation in commercially available FPGAs and the FSM acquires capabilities for dynamic modification. The latter can be achieved even when statically configured FPGAs such as the Xilinx XC4000XL are used.

References

1. Giovanni De Micheli: Synthesis and Optimization of Digital Circuits: McGraw-Hill, Inc., (1994)
2. Baranov, S.: Logic Synthesis for Control Automata. Kluwer Academic Publishers, (1994)
3. Sklyarov, V.: Synthesis of FSMs based on matrix LSI. Science and Technique, Minsk (1984)
4. Sklyarov, V.: Hierarchical Finite State Machines and Their Use for Digital Control. IEEE Transactions on VLSI Systems. Vol. 7, No 2 (1999) 222-228