

Synthesis of FSMs on the Basis of Reusable Hardware Templates

VALERY SKLYAROV, IOULIIA SKLIAROVA, BRUNO PIMENTEL
Department of Electronics, Telecommunications and Informatics / IEETA
University of Aveiro
3810-193 Aveiro
PORTUGAL
skl@det.ua.pt, iouliia@det.ua.pt, pimentel@ieeta.pt, <http://www.det.ua.pt>

Abstract: - This paper suggests a reusable hardware template (HT) for finite state machines (FSM) and a method for the synthesis of FSMs based on such a template. The HT is a circuit with a predefined structure that has already been implemented in hardware (for example, in FPGA). By reprogramming its RAM-blocks we can implement a different functionality of the FSM. The proposed method permits the translation of a given FSM specification (that takes into account the parameters of a particular HT) into bitstreams for reloading the RAM-blocks. Run-time modifications are also permitted with the aid of dual-port memory. Note that the resulting FSM circuits are very fast and any state transition is performed within one clock cycle. The designed C++ program provides synthesis, verification and modeling of FSMs. The synthesized circuits were implemented and tested in Xilinx FPGAs. The synthesis methods considered permit various target requirements to be satisfied, such as minimizing the complexity of the circuit and the possibility for changes in the circuit functionality.

Key-Words: - Finite State Machine, Hardware Template, Logic Synthesis, FPGA

1 Introduction

Finite state machines (FSM) are probably the most widely used components in digital systems. Today almost all the available automatic design tools that are included in industrial CAD systems allow FSMs to be synthesized from their formal specifications [1]. For any FSM specification we can build hardware and software implementations. The former is usually very fast but inflexible, which means any modification implies repeating all the FSM synthesis steps. The latter can be easily modified but the resultant FSM is relatively slow and this can be a problem for a number of practical applications, such as [2,3]. It is known that a hardware implementation can be based on a software approach [4], making use of RAM-blocks, and the proposed approach to FSM synthesis relies on reusable hardware templates (HT). The HT is a memory-based circuit with a predefined structure that has already been implemented in hardware (for example, in FPGA). Its functionality can be changed by reprogramming its RAM-blocks. Available dual-port access to memory allows for run-time modifications. The considered technique provides two important advantages: an opportunity to accommodate the required functionality on a microchip with non sufficient hardware resources; possible changes to the behavior depending on external events that cannot be predicted in advance. Run-time modifications are considered to be an attractive way of allowing efficiency in circuit size [5]. A variety of techniques for run-time

reconfiguration were considered in [6]. Different practical applications of dynamically modifiable FSMs were discussed in [2,7,8].

Note that very fast RAM-based configurable logic blocks (CLB) as well as dedicated embedded RAM blocks are available within widely used and relatively cheap FPGAs. Thus it is possible to implement very efficient circuits based on the suggested method.

The remainder of this paper is organized in five sections. Section 2 discusses reusable circuits and suggests a novel architecture of HT for FSMs. Section 3 describes the synthesis flow for FSMs based on HTs and gives an example. Section 4 presents a C++ program that interacts with commercially-available design tools and can be used for synthesis, verification, modeling and implementation of FSMs. Section 5 discusses practical examples. The conclusion is in section 6.

2 Hardware Template

The hardware template (HT) is a circuit that contains elements with functions that can be changed and which are initially undefined. All the external connections of elements are fixed and they cannot be modified. The customizing of the HT is carried out by programming (reprogramming) its elements with changeable functions. In order to construct the HT, it is necessary to estimate all the likely constraints for future applications. In other words we should define a class of applications and the constraints for that class.

For a FSM, these constraints might be the maximum numbers of the input variables (L_{\max}), the output variables (N_{\max}), the states (M_{\max}) and the transitions from any state; also the maximum size of state codes (R_{\max}), etc. Fig. 1 shows an example of a HT for a FSM. It is composed of two RAM blocks, a FSM memory and a multiplexer. MRAM permits any input $x_i \in \{x_0, \dots, x_{L-1}\}$ of the multiplexer to be selected in such a way that $p = x_i$ and the value i can be specified by MRAM. For example if the FSM codes are binary values of state subscripts and if the input x_9 affects transitions from the state a_5 , then at the address 101 MRAM forms outputs 1001 which control the multiplexer 16:1. If for another application the transitions from a_5 are affected by x_3 then at the address 101, MRAM has to form outputs 0011. Clearly we can provide any correspondence between states a_0, \dots, a_{M-1} and inputs x_0, \dots, x_{L-1} . The RIV block (Replacement of Input Variables) shown in fig. 1 in the dashed rectangle permits any variable from the set $X = \{x_0, \dots, x_{L-1}\}$ to be replaced by a single variable p . State transition RAM (STRAM) enables us to generate codes for the next states and outputs. For example, if $R=4$ and we have the following state transitions $a_{10}x_2 \Rightarrow a_7$ and $a_{10}\bar{x}_2 \Rightarrow a_4$ then at the address 10101 STRAM contains the code $(D_1, \dots, D_4) = 0111$ and at the address 10100 - the code $(D_1, \dots, D_4) = 0100$. Obviously any subset of output signals y_1, \dots, y_N can be arbitrarily generated in any state transition.

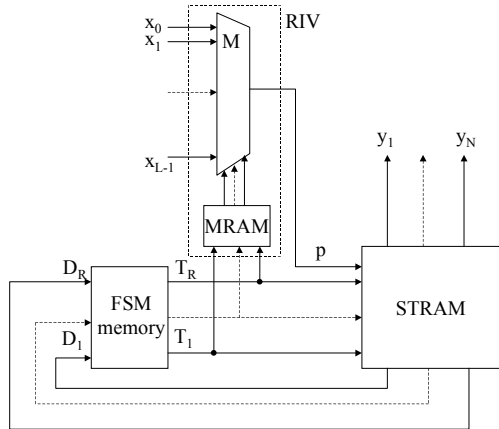


Figure 1. An example of a Hardware Template

By modifying the contents of MRAM and STRAM we can implement any desired FSM behavior within the scope of the constraints predefined for the HT. However the HT depicted in fig. 1 has a very significant constraint: any state transition can only be affected by a single input variable. An arbitrary state transition graph can be altered in such a way that allows this constraint to be satisfied by splitting state transitions and inserting dummy states as shown in

fig. 2. However in the general case this changes the behavior of the FSM, increases the number M of states, and reduces the speed of state transitions.

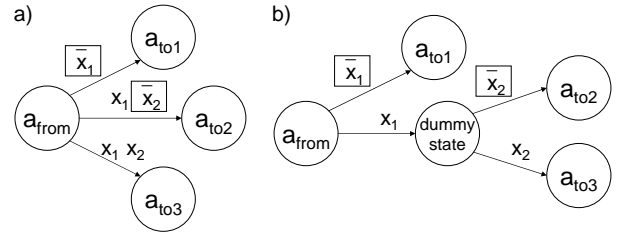


Figure 2. A fragment of a FSM state transition graph (a) and splitting transitions (b)

One possible way to resolve this problem is to increase the number of multiplexers, controlled by MRAM (i.e. the block RIV will have the same inputs and more than one output: p_1, \dots, p_G). However in this case the size of STRAM W will be equal to $W = 2^{R+G} \times (R+N)$ and for complex FSMs this requires lots of hardware resources. Fig. 3 depicts the proposed FSM HT, which allows this size to be significantly reduced.

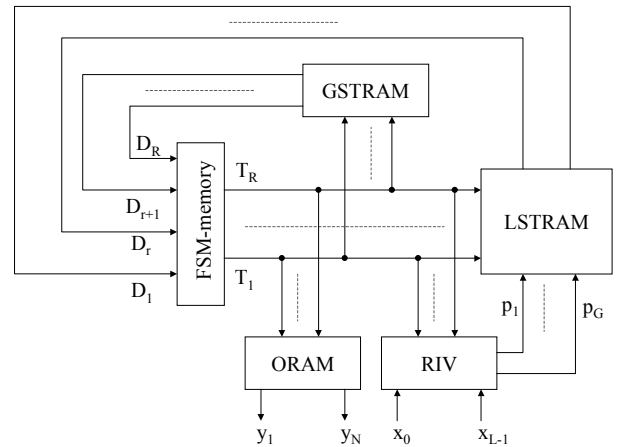


Figure 3. The proposed HT for a FSM

The main difference is in the decomposition of STRAM into two blocks that are GSTRAM (STRAM for Groups) and LSTRAM (STRAM for Local transitions within the selected group). Table 1 (which was taken from [9]) presents all the state transitions in a FSM, where a_{from} and a_{to} are the initial and next states in state transitions, $X(a_{\text{from}}, a_{\text{to}})$ is a product of the input variables from the set $X = \{x_0, \dots, x_{L-1}\}$ that cause the transition from a_{from} to a_{to} , L is the number of FSM inputs, $a_{\text{from}}, a_{\text{to}} \in A = \{a_0, \dots, a_{M-1}\}$, A is the set of FSM states, M is the number of states, $Y(a_{\text{from}})$ - is the set of active variables in the state a_{from} from the set $Y = \{y_1, \dots, y_N\}$, and N is the number of output variables. An active value of any output variable is 1. The meaning of all the remaining columns will be explained later.

Table 1. Structural table of FSM

a_{from} $Y(a_{\text{from}})$	K (a_{from})	X ($a_{\text{from}}, a_{\text{to}}$)	P ($a_{\text{from}}, a_{\text{to}}$)	a_{to}	K (a_{to})	D ($a_{\text{from}}, a_{\text{to}}$)
a0	0000	$\overline{x_1}$	$\overline{p_1}$	a1	0001	D4
		$x_1 x_2$	$\overline{p_1} p_2$	a2	0101	D2D4
		$\overline{x_1 x_2}$	$p_1 p_2$	a3	1101	D1D2D4
a1,y1,y3	0001 0010	$\overline{x_3}$	$\overline{p_1}$	a1	0001	D4
		$\overline{x_3 x_4}$	$\overline{p_1} p_2$	a3	1101	D1D2D4
		$x_3 x_4$	$p_1 p_2$	a4	1001	D1D4
a2,y2,y3,y4,y5	0100 0101	$\overline{x_5 x_6}$	$\overline{p_1} p_2$	a5	0011	D3D4
		$x_5 x_6$	$\overline{p_1} p_2$	a6	0111	D2D3D4
		$\overline{x_5 x_6}$	$p_1 p_2$	a8	1111	D1D2D3D4
		$x_5 x_6$	$p_1 p_2$	a7	1011	D1D3D4
a3,y2,y2,y6	1101	1	1	a4	1001	D1D4
a4,y6	1001 1010	1	1	a2	0100	D2
a5,y7,y8	011	$\overline{x_7}$	$\overline{p_1}$	a4	1010	D1D3
		x_7	p_1	a6	0110	D2D3
a6,y6,y8	0111 0110	$\overline{x_8}$	$\overline{p_1}$	a2	0100	D2
		x_8	p_1	a8	1000	D1
a7,y4	1111	$\overline{x_9}$	$\overline{p_1}$	a1	0010	D3
a8,y2	1000 1011	1	1	a9	1110	D1D2D3
a9,y7	1110	1	1	a0	0000	

Let us designate $A(a_{\text{from}})$ the subset of states that follow the state a_{from} and suppose that $k_{\text{from}} = |A(a_{\text{from}})|$ is the number of elements in the set $A(a_{\text{from}})$. For our example in table 1 we have $A(a_0) = \{a_1, a_2, a_3\}$, $k_0=3$, $A(a_1) = \{a_1, a_3, a_4\}$, $k_1=3$, etc. To calculate the next state in any state transition, we can: 1) identify the appropriate group $A(a_{\text{from}})$; and 2) identify the appropriate state a_{to} in the group $A(a_{\text{from}})$.

Let us assume that the appropriate group $A(a_{\text{from}})$ has already been chosen. Now in order to recognize the proper state we need just $\text{intlog}_2 k_{\text{from}}$ bits and this is significantly less than R. Note that only transitions within groups are caused by input variables from the set $X = \{x_0, \dots, x_{L-1}\}$. Any group $A(a_{\text{from}})$ can be selected knowing the state a_{from} . Thus for the point 1) above we need R bits and for the point 2) R+G bits. The block GSTRAM in fig. 3 has R inputs and permits the appropriate group $A(a_{\text{from}})$ to be selected. The block LSTRAM in fig. 3 has R+G inputs and allows the appropriate state transition within the group $A(a_{\text{from}})$ to be selected. Thus the code $K(a_{\text{to}})$ for the next state a_{to} is composed of two parts: $K_l(a_{\text{to}})$ that indicates a local transition within the group, and $K_g(a_{\text{to}})$ which specifies the group itself. The maximum number of groups can be equal to R and the maximum number of state transitions within a group can be equal to $\text{intlog}_2 k_{\text{max}}$, where $k_{\text{max}} = \max(k_0, \dots, k_{M-1})$. However we

can use a special encoding technique that permits state codes to be assigned in such a way that for majority of practical applications the size of the codes will be equal to R (i.e. this size will not exceed the size that is required for binary state assignment). Various groups $A(a_0), \dots, A(a_{M-1})$ can be combined into a smaller number of groups than M and this gives additional flexibility for the use of the HT in fig. 3. The last block in fig. 3 is an output RAM (ORAM) that permits values of output variables to be generated for any state. Since we are considering the Moore FSM model, this task is trivial because output values depend only on states.

The constraints that define the class of applications for the HT of fig. 3 are the following: the maximum number of input variables L_{max} , the maximum number of output variables N_{max} , the maximum size of the state codes R_{max} , and the maximum number of variables G_{max} from the set $P = \{p_1, \dots, p_G\}$. The latter can be determined on the basis of the value k_{max} . After the circuit is implemented, the behavior of any particular FSM within the specified class can be achieved by loading the RAM-blocks GSTRAM, LSTRAM, ORAM and MRAM (see fig 1 and fig. 3). For static modifications RAM-blocks can be replaced with ROM-blocks. Dynamic modifications can be provided with the aid of dual-port memory in such a way that the first port supports the primary functionality of the FSM and the second port permits reloading of the RAM-blocks to implement new functionality.

The HT we have considered requires RAM-blocks with the following sizes: $W_{\text{ORAM}} = 2^R \times N$, $W_{\text{GSTRAM}} = 2^R \times (R-r)$, $W_{\text{LSTRAM}} = 2^{R+G} \times T$ and $W_{\text{ORAM}} + W_{\text{GSTRAM}} + W_{\text{LSTRAM}} \ll W$.

Let us assume that our HT has the following parameters: $L=16$, $N=10$, $R=4$, $G=2$, $r=2$. It can be used to implement the FSM from table 1. The RAM blocks have the following complexity: $W_{\text{ORAM}} = 2^R \times N = 160$, $W_{\text{GSTRAM}} = 2^R \times (R-r) = 32$, $W_{\text{LSTRAM}} = 2^{R+G} \times T = 128$, $W = 2^{R+G} \times (R+N) = 896$ and $(W_{\text{ORAM}} + W_{\text{GSTRAM}} + W_{\text{LSTRAM}} = 320) \ll (W=896)$.

3 Synthesis of Template-Based FSMs

In order to synthesize a FSM circuit based on HT it is necessary to formulate the general requirements and target for individual synthesis steps. The functionality of a FSM is described by a traditional state transition graph [1], which can be presented in table form, such as that shown in columns a_{from} , $Y(a_{\text{from}})$, $X(a_{\text{from}}, a_{\text{to}})$, a_{to} of table 1. The design flow includes mainly the same steps that were proposed in [4] for RAM-based FSMs and they are the following:

1. State encoding allowing the decomposition of STRAM into GSTRAM and LSTRAM as considered above (see fig. 3).

2. Replacement of input variables from the set $X=\{x_0, \dots, x_{L-1}\}$ with variables from the set $P=\{p_1, \dots, p_G\}$ and building the RIV block.

3. Preparing bitstreams for the RAM-blocks GSTRAM, LSTRAM, ORAM and MRAM.

The first step can be done by reducing the functional dependency of outputs (which in our case are D_{r+1}, \dots, D_R) on inputs ($T_1, \dots, T_R, p_1, \dots, p_G$). In other words, the outputs D_{r+1}, \dots, D_R must not depend on the inputs p_1, \dots, p_G . For the second step we can use the method presented in [10] without any change. The results of replacement for our example in table 1 are shown in column $P(a_{from}, a_{to})$. The succeeding steps become trivial after the first step has been carried out (later we will demonstrate this on an example).

The first step is based on a special state assignment. Consider all codes $K(a_{from}, a_{to})$ for which $a_{to} \in A(a_{from})$. If for given a_{from} bit i ($(r-1) \leq i \leq R$) in all codes $K(a_{from}, a_{to})$ has values either 0 and don't care (-), or 1 and don't care, then bit i does not depend on input variables. Consider an example:

$$K(a_{from}, a_{to1}) = 0-100;$$

$$K(a_{from}, a_{to2}) = 01-11;$$

$$K(a_{from}, a_{to3}) = -1-10.$$

Here, three bits 1 (the left-most bit), 2 and 3 do not depend on input variables. In other words knowing just the state a_{from} we can calculate values of these bits. Note that don't care value (-) can be replaced with either 0 or 1. Any one of the codes so constructed is correct. Thus the states in each group $A(a_{from})$, $a_{from} \in \{a_0, \dots, a_{M-1}\}$ must be encoded in such a way that all their bits $r+1, \dots, R$ satisfy this requirement. This can be done with the aid of an encoding table (which looks like a Karnaugh map), which has 2^r rows and 2^{R-r} columns. All the rows are coded by different r -bit Boolean vectors and all the columns are coded by different $(R-r)$ -bit Boolean vectors. Fig. 4 presents such a table for our example.

		Bits 3 and 4 of FSM code			
		00	01	11	10
Bits 3 and 4 of FSM code	00	a_0	a_1	a_5	a_1
	01	a_2	a_2	a_6	a_6
	11		a_3	a_7	a_9
	10	a_8	a_4	a_8	a_4

Figure 4. State encoding for FSM in table 1

The rules for filling out the table are the following:

1. All the states from any sub-set $A(a_{from})$ must be accommodated within the same column of the table.

2. Any table box can be occupied by just one state.

3. If necessary, any state can be repeated in the table any number of times and this does not affect the complexity of future FSM circuits.

4. All FSM states must be accommodated in the table. Now the code of any FSM state will be composed of r bits of the row and $R-r$ bits of the column. As shown in table of fig. 4, $A(a_0) = \{a_1, a_2, a_3\}$, $A(a_1) = \{a_1, a_3, a_4\}$, $A(a_2) = \{a_5, a_6, a_7, a_8\}$, etc. The states for each individual subset $A(a_{from})$ were accommodated within one column of that table. The respective codes were written in columns $K(a_{from})$ and $K(a_{to})$ of table 1. These codes satisfy our requirement. Bits 3 (i.e. $r+1$) and 4 (i.e. R) that do not depend on input variables appear in **bold** font in column $K(a_{to})$. Some states, such as a_1 , a_2 , a_4 , a_6 and a_8 appeared twice in the table of fig. 4. All the respective codes must be recorded in column $K(a_{from})$ of table 1. If any state has more than one code all state transitions for this state must be repeated for all these codes. However this does not make the circuit more complicated. The bits identified above in **bold** font will be generated by GSTRAM and the remaining bits - by LSTRAM. Column $D(a_{from}, a_{to})$ contains active variables D_1, \dots, D_R that change the states of the FSM memory assuming that this memory is built from D flip-flops. Now information from table 1 can be used to program all the RAM-blocks depicted in fig. 3. Fig. 5 shows bitstreams for two MRAMs that control two multiplexers. The addresses are shown as decimal numbers and the output vectors are presented in binary form.

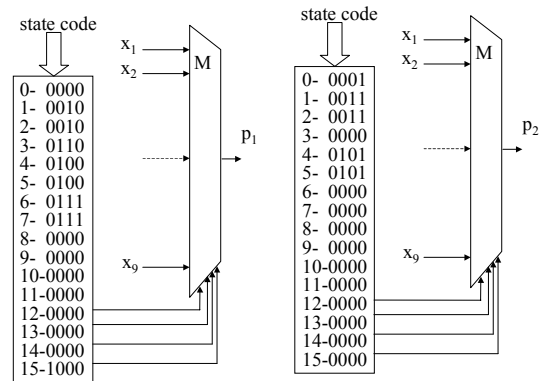


Figure 5. Bitstreams for programming MRAMs

Fig. 6 exhibits bitstreams for GSTRAM and LSTRAM. Some addresses (such as 12 for GSTRAM and 48-51 for LSTRAM) do not appear in fig. 6 because the code 1100 is not used for FSM states (see fig. 4). Data at these addresses can be of any value.

The bitstream for ORAM is the following: (0-00000000, 1-10100000, 2-10100000, 3-00000011, 4-01111000, 5-01111000, 6-00000101, 7-00000101, 8-01000000, 9-00000100, 10-00000100, 11-01000000,

13-01010100, 14-00000010, 15-00010000), where y_1 is presented by the leftmost bit and the number of bits is equal to $N=8$.

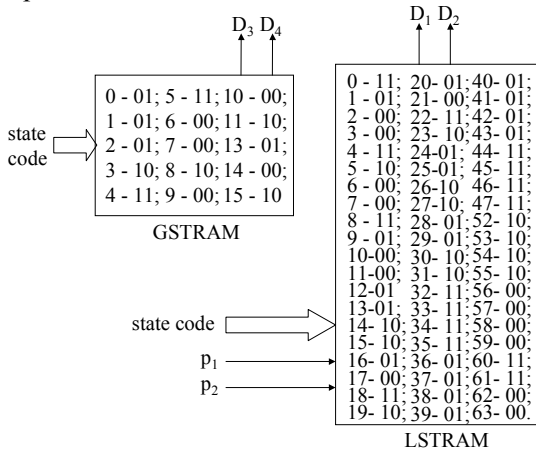


Figure 6. Bitstreams for programming GSTRAM and LSTRAM

Note that combining advantages of the proposed HT (see fig. 3) and architecture [4] we can additionally reduce the size of STRAM = GSTRAM + LSTRAM (see fig. 7). Now GSTRAM forms R-bit code. As before, the bits D_{r+1}, \dots, D_R enable us to identify group codes. The other bits are equal to 0 for conditional state transitions and they form the respective part of state codes for unconditional state transitions (this is exactly the same as it was proposed in [4]). Since variables p_1, \dots, p_G permit to identify an appropriate transition within any group they can be mixed with the respective part of state codes by means of OR gates. For any unconditional transition LSTRAM generates bits D_1, \dots, D_r of the code shown in the respective line of column $K(a_{i_0})$. For any conditional transition LSTRAM forms bits D_1, \dots, D_r by conversion of p_1, \dots, p_G to the appropriate values, shown in the respective line of column $K(a_{i_0})$. All the required details were considered in [4]. Thus the size of STRAM will be equal to $2^R \times (R+r)$ and this is less than in HT shown in fig. 3.

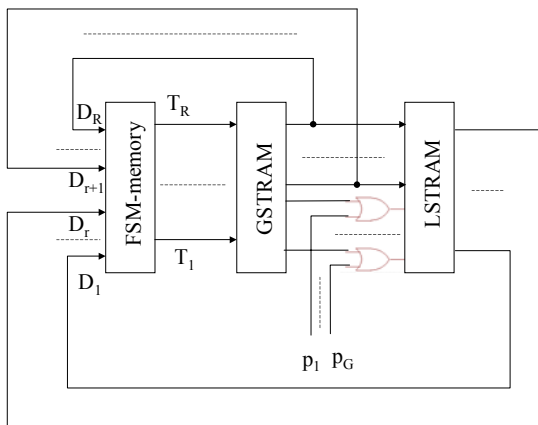


Figure 7. Possible modification of HT

4 Design Tools

All basic steps of FSM design, such as synthesis (see the previous section), modeling and implementation are supported by a C++ program, which includes four basic classes: *FSM_memory*, *RAM_block*, *Synthesizer* and *Hardware_template*. These classes provide all what is necessary for working with blocks of RAM, with HTs (for instance, translating behavioral specifications to bitstreams for the RAM-blocks) and with FSM models. A more complicated stack memory supports a hierarchical model of FSMs considered in [8] which can even be used to implement recursive behaviors, as demonstrated in [11] and employed in [12]. In order to carry out state encoding a slightly modified version of method [4] has been used.

The design tools considered can be integrated with commercially available CAD systems. Fig. 8 shows various possibilities for synthesis, verification and modeling of FSMs.

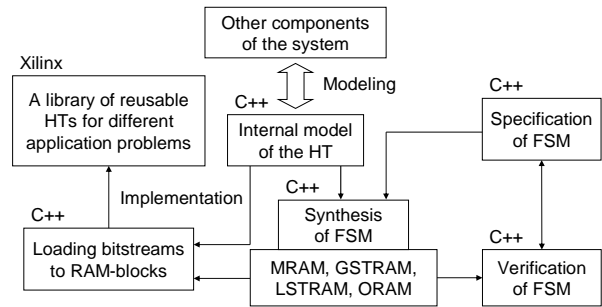


Figure 8T. The use of software tools for synthesis, verification, modeling and implementation of FSMs based on HTs

5 Practical Applications

To estimate how the technique can be used for the design of practical digital systems, we considered combinatorial problems. In particular, we focused on those which can be solved over Boolean and ternary matrices. Two distinctive characteristics of combinatorial computations are [13,14]: dealing with a large volume of data; and using a special search tree for processing this data. Thus, it is possible to evaluate the proposed technique by implementing elementary operations for handling the data and reprogramming the RAM blocks of a HT to apply the behavior of different combinatorial problem solvers. Two well known combinatorial problems have been considered: the covering problem [15,16] and the Boolean satisfiability problem [7,17-20]. The associated algorithms included the following elementary operations:

1. Checking for orthogonality between rows/columns.

2. Counting the number of ones in rows and columns.
3. Testing if one row/column covers another column/row.

The results of these experiments have shown that the proposed hardware template can be successfully employed for such devices, making it possible to use the same hardware for solving different problems.

6 Conclusion

The paper suggests a method of synthesis and implementation of FSMs based on a reusable hardware template (HT). The basic components of the HT are RAM blocks, which can be reprogrammed either before or during execution time, and this enables the FSM behavior to be changed by reloading the appropriate bitstreams. Synthesis, verification, modeling and implementation of the FSM can be done using the designed and implemented C++ program. The latter takes a specification of the concrete FSM (in the form of a state transition graph) and the parameters of the HT that is going to be used. Each particular HT can be employed for a set of FSMs that satisfy the predefined constraints described in the paper. Different HTs can be seen as library components designed with the aid of commercially available CAD software. The results of experiments have shown that the proposed approach can be directly used for practical applications.

References:

- [1] Giovanni de Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.
- [2] *Road vehicles – Interchange of digital information – Part 1: Controller area network (CAN) data link layer and medium access control*; ISO 11898. Part 2: *High-speed medium access unit*; ISO 11898.
- [3] J. Rabaey, *Silicon Platforms for the Next Generation Wireless Systems - What Role Does Reconfigurable Hardware Play?*, Proc. of FPL'2000, Villach, Austria, 2000, pp.277-285.
- [4] V. Sklyarov, *Synthesis and Implementation of RAM-based Finite State Machines in FPGAs*, Proc. of FPL'2000, Villach, Austria, August, 2000, pp. 718-728.
- [5] G. Brebner, *Field-Programmable Logic: Catalyst for New Computing Paradigms*, Field Programmable Logic and Applications, 8th International Workshop FPL'98, Springer, 1998, pp. 49-58.
- [6] N. Shirazi, W. Luk, P. Y. K. Cheung, *Run-Time Management of Dynamically Reconfigurable Designs*, Programmable Logic and Applications, 8th International Workshop FPL'98, Springer, 1998, pp. 59-68.
- [7] I. Skliarova, A. Ferrari, *Design and Implementation of Reconfigurable Processor for Problems of Combinatorial Computations*, Proc. of EUROMICRO Symposium on Digital Systems Design, Warsaw, September, 2001, pp. 112-119.
- [8] V. Sklyarov, *Hierarchical Finite-State Machines and Their Use for Digital Control*, IEEE Transactions on VLSI Systems, Vol. 7, No. 2, 1999, pp. 222-228.
- [9] V. Sklyarov, *Synthesis of Control Circuits with Dynamically Modifiable Behavior on the Basis of Statically Reconfigurable FPGAs*, 13th Symposium on Integrated Circuits and Systems Design: SBCCI2000, Manaus, Brazil, September, 2000, pp. 353-358.
- [10] S. Baranov, *Logic Synthesis for Control Automata*, Kluwer Academic Publishers, 1994.
- [11] V. Sklyarov, *FPGA-based implementation of recursive algorithms*, Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, 2004, pp. 197-211.
- [12] V. Sklyarov, I. Skliarova, B. Pimentel, *FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms*, Proc. of FPL'05, Tampere, Finland, 2005, pp. 235-240.
- [13] V. Sklyarov, I. Skliarova, A. Ferrari, *Hierarchical Specification and Implementation of Combinatorial Algorithms Based on RHS Model*, Proc. of DCIS'2001, Porto, 2001.
- [14] A. D. Zakrevski, *Combinatorial Theory of Logical Design*, Automatics and computer techniques, No.2, 1990, pp. 68-79.
- [15] A. Zakrevskij, *Combinatorial Problems over Logical Matrices in Logic Design and Artificial Intelligence*, Electrónica e Telecomunicações, vol.2, No.2, pp. 261-268.
- [16] I. Skliarova, A. Ferrari, *Exploiting FPGA-Based Architectures and Design Tools for Problems of Combinatorial Computations*, Proc. of SBCCI'2000, Manaus, Brazil, 2000, pp. 347-352.
- [17] P. Zhong, *et al.*, *Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability*, Proc. of the 34th Design Automation Conference, 1998.
- [18] M. Platzner, *Reconfigurable Accelerators for Combinatorial Problems*, IEEE Computer, April, 2000, pp. 58-60.
- [19] J. T. Sousa *et al.*, *A Configware/Software Approach to SAT Solving*, Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'2001.
- [20] M. Boyd, T. Larrabee, *ELVIS – A Scalable, Loadable Custom Programmable Logic Device for Solving Boolean Satisfiability Problems*, Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'2000.