High-Performance Architecture for Binary-Tree-Based Finite State Machines

Raouf Senhadji-Navarro and Ignacio Garcia-Vargas

***Abstract*—A binary-tree-based finite state machine (BT-FSM) is a state machine with a 1-bit input signal whose state transition graph is a binary tree. BT-FSMs are useful in those application areas where searching in a binary tree is required, such as computer networks, compression, automatic control, or cryptography. This paper presents a new architecture for implementing BT-FSMs which is based on the model finite virtual state machine (FVSM). The proposed architecture has been compared with the general FVSM and conventional approaches by using both synthetic test benches and very large BT-FSMs obtained from a real application. In synthetic test benches, the average speed improvement of the proposed architecture respect to the best results of the other approaches achieves 41% (there are some cases in which the speed is more than double). In the case of the real application, the average speed improvement achieves 155%.**

***Index Terms*—Binary tree, field programmable gate array (FPGA), finite state machine, finite virtual state machine (FVSM).**

I. Introduction

TODAY, we can observe a growing interest in the devel-opment of new hardware architectures for implementing algorithms whose software solutions do not achieve the high performance required by current applications [1]. In many cases, these algorithms are based on binary trees; so, the efficient implementation of binary trees has a decisive influence on the overall performance of the system. In computer science, a binary tree is a data structure widely used in searching algorithms. Computer networks [2], [3], compression [4], automatic control [5], [6], or cryptography [7] are some application areas of binary trees. Many examples of hardware implementations that perform searches on binary trees can be found in the literature. Internet protocol (IP) address look-up engines for high-speed hardware routers are usually implemented by using binary trees [2]. Huffman decoders, which require to process binary trees, are implemented on hardware in order to speed up decompression algorithms [4]. In [3], a packet classifier based on a binary tree for network intrusion detection systems is proposed. Efficient hardware implementations of the binary trees used by the Knuth–Yao algorithm (called discrete distribution generating trees) are required in many cryptography applications [7]. Finally, general piecewise-affine functions, which are used in embedded control systems, can be implemented in hardware using binary search trees in such way that the function is evaluated at each node to determine the next node to visit [5], [6].

The flexibility and performance of field programmable gate arrays (FPGAs) make these devices an ideal platform for building custom hardware accelerators and embedded systems. There are many examples in the literature of FPGA-based implementations that use binary trees [3]–[5], [7]. In these applications, the circuit that performs searches on the binary tree can be modeled as a finite state machine (FSM) [2], [5]. We will refer to this kind of FSM as binary-tree-based FSM (BT-FSM). Many authors highlight that software FSM implementations on FPGA-based general processors cannot compete with specific hardware implementations in terms of speed [8], [9]. In software solutions, several instructions are required to determine the next node and output of the binary tree (or the FSM); so, each transition spends more than one clock cycle unlike hardware solutions. The efficient hardware implementation of BT-FSMs is a challenge, particularly when the binary tree has a large number of nodes [2].

In [10], a model of finite state machines based on a two-level memory hierarchy, called finite virtual state machines (FVSMs), was proposed. The aim of this model is to improve the performance of the FPGA-based implementations of state machines with a large number of states. The architecture presented in [10] (which will be referred as general FVSM architecture) allows the implementation of any FSM; however, the performance could be increased if the architecture is adapted to exploit the particular properties of BT-FSMs. In this paper, we propose a new architecture based on the FVSM model specifically designed for BT-FSMs, which is called binary-tree-based FVSM (BT-FVSM).

The next sections are organized as follows. In Section II, conventional implementations of BT-FSMs are outlined. Section III shows the general FVSM architecture and explains the features of BT-FSM implementations when this architecture is used. In Section IV, the proposed architecture is described and compared with the general FVSM architecture. Section V presents the experimental results. Finally, the main conclusions are summarized in Section VI.

II. Conventional Implementations of BT-FSMs

A BT-FSM is an FSM with a 1-bit input signal whose state transition graph (STG) is a binary tree. Like any arbitrary FSM, a BT-FSM can be implemented in an FPGA using

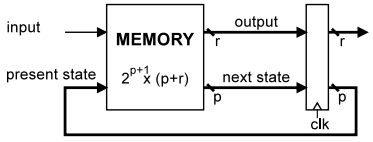


Fig. 1. Memory-based architecture of a BT-FSM.

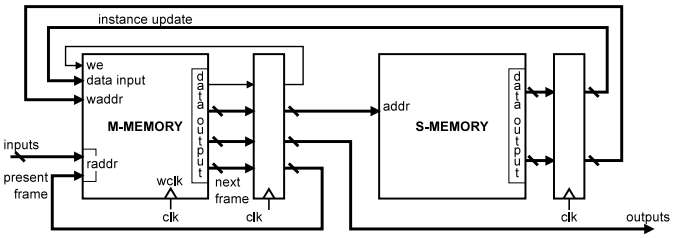


Fig. 2. General FVSM architecture.

two different approaches. On the one hand, BT-FSMs can be implemented by using registers and combinational logic that is mapped into look-up tables (LUTs). We will refer to this approach as *cell-based implementation*. On the other hand, BT-FSMs can be implemented by using memory (which will be referred as *memory-based implementation*).

Fig. 1 shows the architecture of a memory-based implementation of a BT-FSM. Each memory word contains a transition of the FSM, i.e., the value of the *output* signal and the next state encoding bits [11]. The memory address is composed of the *input* signal and the present state encoding bits. The next state is stored in the register and is fed back to the memory address signal as the present state. This architecture allows the implementation of Mealy FSMs with registered outputs. Therefore, a Moore FSM must be converted to an equivalent Mealy FSM in which the output of each transition takes the value of the output corresponding to the next state. Current FPGA devices include a large number of synchronous embedded memory blocks (EMBs) that can be used for storing the transitions of the FSM.

In a cell-based implementation, the maximum operating frequency is determined by the number of levels of LUTs, which depends on the complexity of the transition and output functions. In BT-FSMs, the number of states is a significant factor of the complexity of these functions. FPGA manufacturers recommend to map large FSMs into EMBs in order to improve the performance. For example, Xilinx asserts that large FSMs can be made more compact and faster by implementing them in EMBs [12]. In fact, cell-based implementations of large FSMs do not provide the best performance [13].

EMBs have a fixed access time independently of its content. Therefore, the performance is high if the BT-FSM can be stored on a unique EMB. However, if the memory depth (i.e., the number of words) is greater than the maximum depth available in EMBs [14], then some LUTs and embedded multiplexers are used for multiplexing the outputs of the EMBs. The delay of these extra logic elements and the routing overhead can reduce significantly the performance of the memory [11]. As a drawback, EMBs are limited resources in comparison with LUTs and they are essential in the design of memories in system-on-programmable-chips.

Despite memory-based implementations usually uses EMBs [15], it is also possible to use distributed memory (i.e., LUTs configured as little memories [16], [17]). These implementations can obtain high performance when the memory depth is small [11], [17]

Memory-based implementations can use either ROM (*ROM-based implementations*) or RAM (*RAM-based implementations*). RAM-based implementations have an important advantage respect to cell-based implementations: the behavior of the FSM can be modified in run-time. In some applications (e.g., IP routing [2]), run-time update is a requirement.

III. General FVSM Implementation of BT-FSMs

The FVSM model is based on a two-level memory hierarchy in which the states required by the behavior of the state machine are dynamically transferred from the secondary memory (called S-memory) to the main memory (called M-memory). The general FVSM architecture proposed in [10] (see Fig. 2) allows to implement state machines using this model. M-memory implements a RAM-based FSM with generic states (called *frames*) whose behavior is determined by the content of the memory. In order to be implemented in this architecture, an FSM is decomposed into a set of nondisjoint sub-FSMs called *instances*. At each moment, the instance stored on M-memory (called *present instance*) is the only active instance, which determines the FSM operation. So, in M-memory, frames play the role of the states of the FSM.

The FVSM architecture allows to implement any arbitrary FSM, including BT-FSMs. Fig. 3(a) shows a possible decomposition of a BT-FSM example into instances. This BT-FSM models a pattern recognizer, i.e., a circuit that recognizes a set of patterns from a sequence of bits [2]. For example, for the input value 01010001, the BT-FSM matches the patterns 010, 0101, and 010100 and generates the output values 0010, 0101, and 1100 at the states s_5, s_8, and s_{18}, respectively (these values represent the position of the patterns in the list shown in the figure caption). The output value 0000 indicates that no pattern has been matched. Note that s_{19} (called *final state*) is the only state that has more than one incoming transition. If this state is removed, the obtained STG is a binary tree.

In the example, the BT-FSM is decomposed into the instances $I_0, \ldots, I_5$. The *initial instance* (i.e., the instance that is loaded into M-memory when the state machine is reset) is I_0. The present instance can be changed by reading a set of states (called *instance update*) from S-memory and by writing them on M-memory. This operation, which requires two clock cycles, is carried out by the present instance itself without interrupting the proper FSM operation. For example, Fig. 3(b) shows the transitions that involve the change of the present instance from I_2 to I_5. At the first cycle, the instance update of I_5 is read from S-memory by setting the *addr* signal [denoted by addr (I_5) in the transition to s_2]. This instance update do not need to include s_{19} because this state belong to both I_2

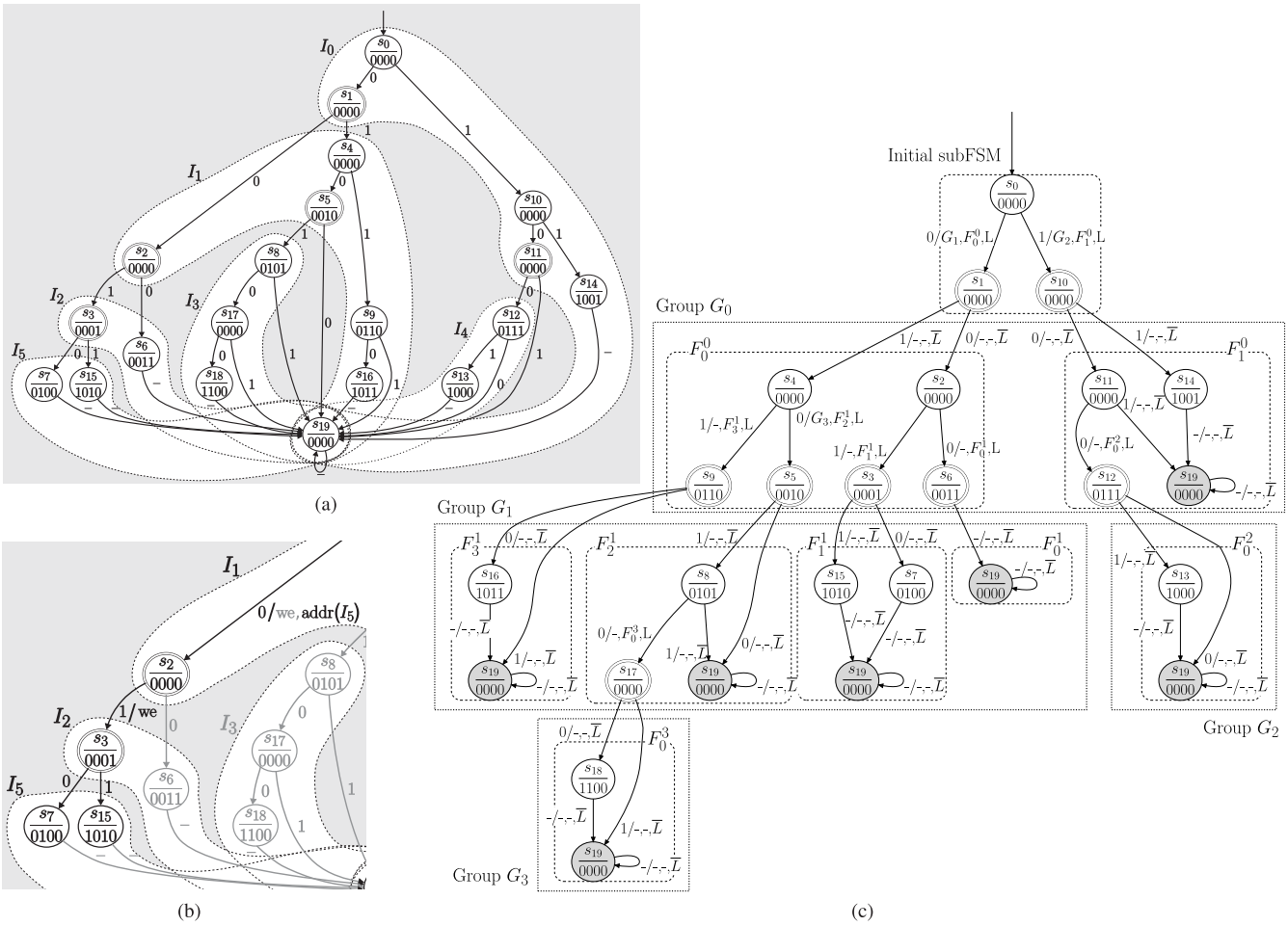


Fig. 3. Example of general FVSM and BT-FVSM implementations generated from a given BT-FSM which matches the patterns 001, 010, 000, 0010, 0101, 011, 100, 1001, 11, 0011, 0110, and 010100. (a) Decomposition into instances of the general FVSM implementation. (b) Detail of the signals involved in the change of the present instance from I_2 to I_5. (c) Decomposition into sub-FSMs of the BT-FVSM implementation.

and I_5. At the second cycle, the instance update is written on M-memory by setting the *we* signal (see the transition to s_3).

Instance changes occur at the output transitions of special states called *update states*, which are marked with a double circle in the BT-FSM example (e.g., s_3 is the update state of I_5). Therefore, the number of instances and their content are determined by the update states: each update state determines an instance that is composed of all states reachable from that update state without traversing any other update state. The initial instance is determined by the reset state in a similar way. As the instance update is read one clock cycle before the update state is reached, two states with the same father (e.g., s_3 and s_6) cannot be update states of different instances in this architecture [10]. The read and write operations are pipelined by using registers (see Fig. 2), which allow a throughput of one update per cycle. In the transition to s_2 of the example, the update of I_5 begins by setting *addr* and the update of I_2 finishes by setting *we*.

The performance of the FVSM architecture is determined by the slowest memory [10]. As the performance of a memory is mainly given by its depth [11], the memory with greater depth determines the overall performance. On the one hand, the depth of M-Memory is equal to the number of transitions of the greatest instance (i.e., the double of the number of states of that instance). On the other hand, the depth of S-memory is given by the number of instances updates (i.e., the number of instances different to the initial one). In the example, the depth of M-memory and S-memory are 12 and 5, respectively (so the performance is determined by M-memory).

An optimization process is used to find a decomposition into instances that balances the depth of both memories [10]. For n states and k instances, the minimum depth of M-memory is $2\lceil(n/k)\rceil$ whereas the depth of S-memory is $k-1$. Ideally, the highest performance is obtained when both values are equals; however, the best approximation is obtained when S-memory depth is

$$\left\lceil \frac{\sqrt{1+8n}-1}{2} \right\rceil. \tag{1}$$

This value, which will be referred as *optimal depth*, determines an upper bound for the performance of a general FVSM implementation with n states. In the example, the optimal depth is 6 and could be obtained with instances of 3 states (so both memories would have a depth of 6). Note that the optimal depth cannot be reached in the BT-FSM example due to the

topology of the STG and the constraint imposed on update states by the architecture.

M-memory is a dual-port RAM with asymmetric port width [18], which allows to use a word size in read operations different than in write operations. However, the ratio between the ports is limited both in memories based on EMBs and in those based on LUTs (i.e., distributed memories) [14]. For example, a 6-LUT can be configured as a RAM of 64×1, so a distributed RAM of 256×1 is composed of 4 RAMs of 64×1; therefore, the ratio is 1/4, i.e., it is possible to read 1 word and to write 4 words (1 word per each RAM). The decomposition shown in Fig. 3(a) has been done by supposing no constraint in the read/write ratio of M-memory (i.e., all states of the present instance can be updated in one write operation). In a real implementation, the optimization process increases the size of instances in order to guarantee the ratio imposed by the implementation of M-memory [19]. For example, if the ratio is 1/2, M-memory can only write 2 transitions (i.e., one state) in each write operation; therefore, the algorithm must extend the adjacent instances in order to guarantee that they only differ in one state. For example, $I_1 = \{s_2, s_4, s_5, s_9, s_{16}, s_{19}\}$ and $I_2 = \{s_3, s_6, s_{19}\}$ could be extended with $\{s_3\}$ and $\{s_4, s_5, s_9, s_{16}\}$, respectively (so the update from I_1 to I_2 consists on changing s_2 to s_6). Therefore, M-memory depth is increased from 12 to 14).

The balance between memories is done by a complex optimization algorithm [10]. In practice, the algorithm obtains solutions in which the depth of at least one of the memories is much greater than the optimal depth; this results in a significant degradation of the performance of general FVSM implementations. The reasons are summarized below. First, the optimal depth has been calculated without considering the topology of BT-FSMs, and so the constraint imposed on update states can prevent to reach that value. Second, the limited read/write ratio of M-memory can increase its depth. Finally, the algorithm is based on branch-and-bound techniques, and so an optimal solution is not guaranteed.

IV. BT-FVSM Architecture

The performance of FVSM implementations of BT-FSMs can be improved by exploiting the regular structure of binary trees. This structure allows a systematic decomposition of a BT-FSM into sub-FSMs that can overcome the problems resulting from the balance between the memories. Fig. 3(c) shows the decomposition of the BT-FSM example of Fig. 3(a). The sub-FSMs are marked by rounded rectangles. Each sub-FSM is composed of states of two consecutive levels of the tree. The states of the levels 0 and 1 compose an special sub-FSM called *initial sub-FSM* (in the example, it is composed by s_0, s_1, and s_{10}). The rest of sub-FSMs are obtained from the states of the odd levels of the tree, which are update states. Each update state determines a different sub-FSM that is composed of its child and grandchild nodes. So, each sub-FSM has at most 6 states (i.e., 12 transitions) and, thus, M-memory only requires 12 words. However, all states of the odd levels of the tree have to be update states of different instances, so this decomposition cannot be applied in the general FVSM architecture due to the constraint on update states that imposes this architecture.

The set of sub-FSMs obtained from all update states included in a given sub-FSM is called *sub-FSM group* (or simply, *group*). As each sub-FSM have a maximum of 4 updates states, the number of sub-FSMs of a group is at most 4. In Fig. 3(c), the groups are marked by rectangles, and each sub-FSM of a group is labeled with F_j^k, where k identifies the group and $j = 0, 1, 2, 3$ identifies the sub-FSM. For example, the group G_1 is composed of the sub-FSMs F_0^1, F_1^1, F_2^1, and F_3^1. All these sub-FSMs are determined by the update states included in F_0^0, which in turn are determined by s_1; therefore, regardless of the input sequence, when the state s_1 is reached, one of the sub-FSMs of the group G_1 will be required after two clock cycles. Due to this property, the read of a group from S-memory can start two clock cycles before the sub-FSM is written on M-memory. Therefore, S-memory could be controlled by a clock that operates at the half of the frequency of the clock of M-memory. However, S-memory would require to read four sub-FSMs in parallel (i.e., a group) and to select one of them after the read is finished. This is not possible in the general FVSM architecture, because only one instance update (i.e., the equivalent to one sub-FSM) can be read from S-memory. So, the instance update that will be loaded into M-memory must be known before the read operation starts. The final state of the BT-FSM (s_{19} in the example) is the unique state that belongs to more than one sub-FSM (this state is shown as a filled circle).

The BT-FVSM architecture is shown in Fig. 4. Although this architecture is also based on the FVSM model [10], it has been specifically designed to allow the implementation of the systematic decomposition of BT-FSMs presented above. In this architecture, M-memory is a RAM-based implementation of a generic FSM of a 1-bit input and 6 generic states (i.e., six frames). Therefore, any sub-FSM can be stored on it. S-memory contains the sub-FSMs in which the BT-FSM is decomposed. During the BT-FVSM operation, the sub-FSMs are dynamically transferred from S-memory to M-memory. At each instant of time, the sub-FSM stored on M-memory is the unique sub-FSM active, which is called *present sub-FSM*. Similarly, the sub-FSM that will be loaded into M-memory during the next write operation is called *next sub-FSM*.

Each state of a sub-FSM is statically assigned to one different frame. So, each frame contains one different state of the present sub-FSM according to this assignment. Table I shows a possible assignment of states to frames for the BT-FVSM shown in Fig. 3(c). For example, the frame f_3 contains s_5 (i.e., contains the transitions of s_5) when the present sub-FSM is F_0^0, and s_{19} when the present sub-FSM is F_1^0. Therefore, the present state of a BT-FVSM is determined by the present sub-FSM and the present frame. As the depth of M-memory is very reduced, it can be implemented using registers without loss of performance (in general terms, this is not feasible in the general FVSM architecture due to its larger depth). For this purpose, M-memory includes the registers called $T_0, T_1, \ldots, T_{11}$. Each one of these registers contains one of

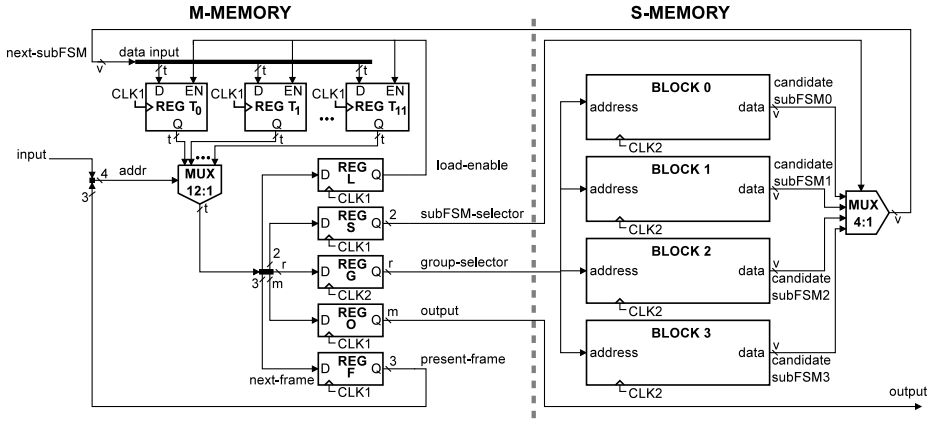


Fig. 4. BT-FVSM architecture.

TABLE I
ASSIGNMENT OF STATES TO FRAMES FOR THE BT-FVSM EXAMPLE

subFSM	f_0	f_1	f_2	f_3	f_4	f_5
Initial subFSM	s_0	s_1	s_{10}	–	–	–
F_0^0	s_2	s_3	s_4	s_5	s_6	s_9
F_1^0	s_{11}	s_{12}	s_{14}	s_{19}	–	–
F_0^1	s_{19}	–	–	–	–	–
F_1^1	s_7	s_{15}	s_{19}	–	–	–
F_2^1	s_8	s_{17}	s_{19}	–	–	–
F_3^1	s_{16}	s_{19}	–	–	–	–
F_0^2	s_{13}	s_{19}	–	–	–	–
F_0^3	s_{18}	s_{19}	–	–	–	–

the 12 transitions of the present sub-FSM (2 transitions per each one of the six frames). The whole content of M-memory can be modified in one clock cycle because the registers can be loaded in parallel. This allows to load a complete sub-FSM into M-memory in one clock cycle. This is an important difference from the general FVSM architecture that allows to overcome the problem resulting from the read/write ratio of M-memory.

M-memory also includes a set of registers (called F, O, G, S, and L) that store at each clock cycle the last transition of the FSM. The register F contains the present frame encoding bits, which corresponds to the next frame encoding bits of the transition. The register O contains the current value of the *output* signal. The registers G, S, and L store the value of the *group-selector*, *sub-FSM-selector*, and *load-enable* signals, respectively. These signals are used for transferring sub-FSMs from S-memory to M-memory, as will be explained later. The 12:1 multiplexer allows to select the transition via a word composed of the *present-frame* and *input* signals.

S-memory contains all sub-FSMs in which the BT-FSM is decomposed except the initial one. It is implemented with four memory blocks (called *S-memory blocks*) that are read in parallel. Each word of the S-memory blocks contains a complete sub-FSM (i.e., its 12 transitions). The sub-FSMs of a same group are stored on different S-memory blocks at the same address. So, it is possible to read a sub-FSM group from S-memory and to write one of its sub-FSMs on M-memory (the sub-FSM is selected by the 4:1 multiplexer).

Each S-memory block is a ROM of $n \times v$ bits with registered output, where n is the number of groups and v is the maximum number of bits of a sub-FSM. Each transition includes the values of the following signals: *group-selector* ($r = \lceil \log_2 n \rceil$ bits), *output* (m bits), *present-frame* (3 bits), sub-FSM selector (2 bits), and *load-enable* (1 bit); therefore, $v = 12(r + m + 6)$ bits. If run-time reconfiguration is required, RAM can be used instead of ROM. In this case, memories with a time-multiplexed port or dual-port memories must be used.

Like in the general FVSM architecture, the performance of the BT-FVSM architecture depends on the performance of M-memory and S-memory. However, the BT-FVSM decomposition allows S-memory to be controlled by a clock (called CLK2) that operates at the half of the frequency of the clock of M-memory (called CLK1). The depth of S-memory is practically always much greater than that of M-memory (i.e., 12 words). In addition, S-memory presents a more routing overhead due to its greater size. As a consequence, S-memory is the slowest memory. Therefore, the use of different clocks allows to reduce the impact of S-memory on the overall performance. On the other hand, according to (1), the depth of M-memory of BT-FVSM (i.e., 12 words) is less than the optimal depth of the general FVSM architecture for BT-FSMs with more than 78 states. In these cases, M-memory of BT-FVSM architecture presents better performance than the general FVSM architecture.

A. BT-FVSM Operation

When a BT-FVSM is reset, T_i registers (with $i = 0, \ldots, 5$) are initialized with the transitions of the initial sub-FSM; the L register, with 0; the F register, with the encoding bits of the initial frame; and the output registers of two of the S-memory blocks, with the sub-FSMs determined by the update states at level 1 of the tree (in the example, F_0^0 and F_1^0). At each CLK1 cycle, the present sub-FSM (i.e., the sub-FSM stored on M-memory) generates the output value of the BT-FSM. In addition, the present sub-FSM controls the process for transferring the next sub-FSM from S-memory to M-memory. This process begins by setting the *group-selector* signal, which contains the address where the corresponding sub-FSM group is stored on S-memory. Each S-memory block provides one

of the four sub-FSMs of the group, which are called *candidate sub-FSMs*. One of these candidate sub-FSMs will be finally stored on M-memory after two CLK1 cycles (note that an access to S-memory requires two cycles of CLK1). Finally, after these two cycles, the next sub-FSM is written on M-memory by setting the *sub-FSM-selector* and *load-enable* signals. The first one allows to select the next sub-FSM from the candidate sub-FSMs via the 4:1 multiplexer. The *load-enable* signal allows to load the transitions of the selected sub-FSM into *Ti* registers. The values of the *group-selector*, *sub-FSM-selector*, and *load-enable* signals are stored on the registers: G, S, and L, respectively. While the rest of registers of M-memory are controlled by CLK1, the register G is controlled by CLK2 because its value is used to address S-memory blocks. In Fig. 3(c), the transitions of the BT-FVSM are labeled with "$i/G_p, F_j^k, v$," where i represents the value of the *input* signal; G_p, the group addressed by *group-selector*; F_j^k, the sub-FSM selected by *sub-FSM-selector* (note that F_j^k does not belong to the group G_p; i.e., $k \neq p$); and v, the value of *load-enable* (L when it is enabled, and $\bar{L}$ otherwise). The symbol "-" represents a do not care value. The reading of a group and the loading of a sub-FSM into M-memory are controlled by the transitions to update states. In these transitions, the value of *sub-FSM-selector* selects one of the sub-FSMs of the last group read, and the value of *group-selector* addresses the new group to read. For example, in the transition from s_0 to s_1, the read operation of G_1 is started, however, the next sub-FSM selected is F_0^0, which belongs to G_0. Once s_1 is reached, the read operation of G_1 must begin because, regardless of the input sequence, one of the sub-FSM of this group (i.e., F_0^1, F_1^1, F_2^1, or F_3^1) will be required after two CLK1 cycles. In the transitions to nonupdate states, *load-enable* is disabled, and the value of *group-selector* does not affect to the current read operation from S-memory because the G register is controlled by CLK2.

In order to illustrate the operation of the BT-FVSM architecture, Fig. 5 shows a timing diagram of the BT-FVSM example for the input value 01010001. In Fig. 5, the cycles are numbered according to CLK1 clock. At each cycle, the present state is the state stored on M-memory at the present frame. So, the present state is given by the present sub-FSM and the present frame. Although the *present-frame* signal contains the frame encoding bits, for clarity, the timing diagram shows the identifier of the present frame (i.e., f_k). In addition, the present state is shown in brackets. States are stored on the frames of M-memory according to Table I. When the BT-FVSM is reset, the initial sub-FSM is loaded into M-memory (see *present-sub-FSM*), the *present-frame* signal is set to f_0, and the group G_0 is loaded into the output registers of S-memory blocks (see the *candidate-sub-FSMi* signals). As the first bit of the input sequence is 0, the present frame changes from f_0 to f_1 at the 2nd cycle (according to Table I, these frames contain the states s_0 and s_1, respectively). In the transition from s_0 to s_1 [see Fig. 3(c)], *group-selector* sets the address of G_1; so, the read operation of G_1 from S-memory begins at the second cycle (note that the rising edge of CLK1 and CLK2 match at the even cycles). In addition, in this transition, *load-enable* is high and *sub-FSM-selector* selects F_0^0 from the output registers of S-memory blocks (which have been set to G_0 by the *reset signal*); so, *next-sub-FSM* takes the value of F_0^0 at the second cycle and it is stored on M-memory at the third cycle (see *present-sub-FSM* at this cycle). Note that, unlike the rest of groups, G_0 has not been read from S-memory blocks due to the initialization done by the *reset* signal. At the fourth cycle, according to the transition from s_4 to s_5, the read of the group G_3 begins (see *group-selector*), and F_2^1 is selected as the next sub-FSM from the group G_1, which was read at the second cycle (note that an access to S-memory requires two cycles of CLK1). At this fourth cycle, the transition also sets the output value to 0010 and changes the present frame from f_2 to f_3 (which contains s_5 due to the present sub-FSM is F_0^0). At the fifth cycle, F_2^1 is stored on M-memory as the present sub-FSM. In a similar way, the last change of the present sub-FSM is done at seventh cycle. After s_{18} is reached, regardless the value of the input sequence, the present state will remain in the final state s_{19} until the state machine is reset.

TABLE II
SUB-FSM GROUPS BEFORE THE OPTIMIZATION PROCESS

G_k	F_0^k	F_1^k	F_2^k	F_3^k
$k=0$	$s_2\ s_3\ s_4\ s_5\ s_6\ s_9$	$s_{11}\ s_{12}\ s_{14}\ s_{19}$		
$k=1$	s_{19}	$s_7\ s_{15}\ s_{19}$	$s_8\ s_{17}\ s_{19}$	$s_{16}\ s_{19}$
$k=2$	$s_{13}\ s_{19}$			
$k=3$	$s_{18}\ s_{19}$			

TABLE III
SUB-FSM GROUPS AFTER THE OPTIMIZATION PROCESS

G_k	F_0^k	F_1^k	F_2^k	F_3^k
$k=0$	$s_2\ s_3\ s_4\ s_5\ s_6\ s_9$	$s_{11}\ s_{12}\ s_{13}\ s_{14}\ s_{18}\ s_{19}$		
$k=1$	s_{19}	$s_7\ s_{15}\ s_{19}$	$s_8\ s_{17}\ s_{19}$	$s_{16}\ s_{19}$

B. Optimization of BT-FVSM Implementations

The efficiency of BT-FVSM implementations can be improved by applying an optimization process to reduce the number of groups obtained from a given BT-FSM. This allows to reduce the depth and width of S-memory blocks, and the width of T_i and G registers. So, area and speed results can be improved. As the binary tree is usually incomplete, there are many sub-FSMs (different to the initial one) that have less than six states (these sub-FSMs are called incomplete sub-FSMs). A greedy algorithm is used to reduce the number of groups by joining incomplete sub-FSMs. This allows to reduce significantly the amount of memory used for implementing S-memory and, therefore, the overall performance can be improved. This algorithm is based on a heuristic similar to first fit decreasing, which is used in greedy bin-packing algorithms [20]. The algorithm guarantees that two states that belong to a same sub-FSM will belong to a same sub-FSM after the procedure is applied. For the BT-FVSM example, Table II shows the groups obtained directly from the decomposition shown in Fig. 3(c). Table III shows the groups after the joining procedure. The sub-FSMs F_1^0, F_0^2, and F_0^3 are joined in F_1^0. So, the number of groups is reduced to half.

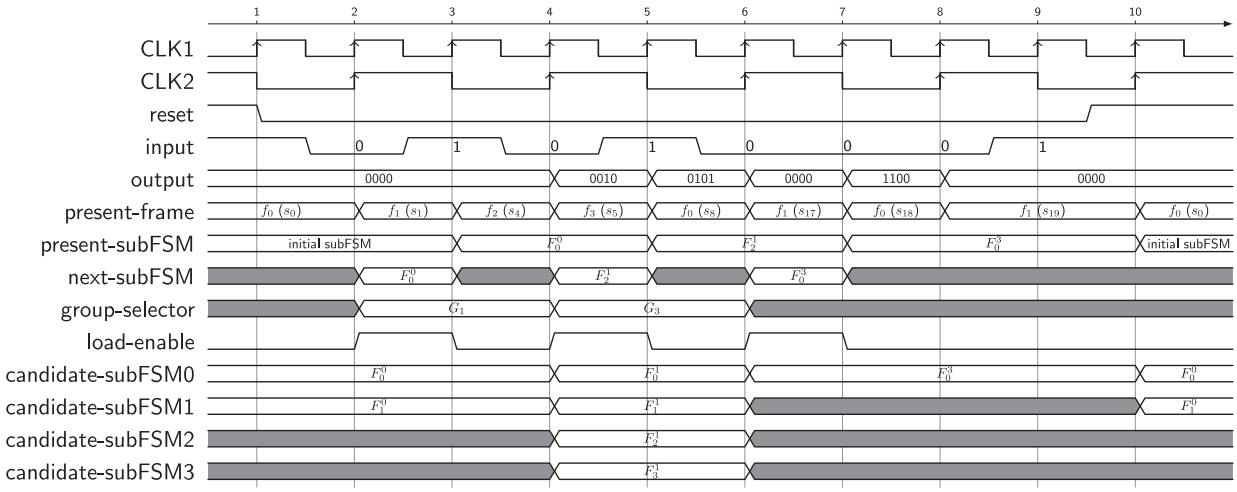


Fig. 5. Timing diagram of the BT-FVSM example for the input value 01010001.

V. Experimental Results

A comparison analysis between the techniques described in this paper is presented with the aim of evaluating the performance of the BT-FVSM architecture. The memory of the ROM-based implementation and S-memory of both FVSM architectures have been implemented using either LUTs (i.e., as distributed memory) or EMBs. So, this paper includes the following techniques: BT-FVSM with S-memory implemented using EMBs (BTFVSM-EMB) or LUTs (BTFVSM-LUT), the general FVSM architecture with S-memory implemented using EMBs (GFVSM-EMB) or LUTs (GFVSM-LUT), ROM-based implementation using EMBs (ROM-EMB) or LUTs (ROM-LUT), and cell-based implementation (FSM-LUT).

Two different sets of test benches have been used in this paper. The first set is composed of 16 synthetic BT-FSMs whose number of states ranges from 146 to 28 267, each of one generated from a different set of random patterns. These test benches have been used to compare all techniques using a Spartan-6 xc6slx75-2 FPGA device. This device includes 172 EMBs of 18 KB which can be configured as two independent EMBs of 9 KB (in the results, each EMB of 9 KB is computed as 0.5). The second set of test benches is composed of four large BT-FSMs obtained from the well-known Mae-East IP routing database [2]. In this case, the largest device supported by ISE WebPACK 14.6 (i.e., Virtex-6 xc6vlx75t-2) has been used. However, due to a BT-FSM that recognizes the whole database does not fit into this device, 4 BT-FSMs have been generated using only the first k IP addresses, where k is equal to 5000, 6000, 7000, and 8000. The number of states of these BT-FSMs ranges from 36 080 to 52 589. For these test benches, the BT-FVSM architecture is compared with the most competitive technique for large BT-FSMs.

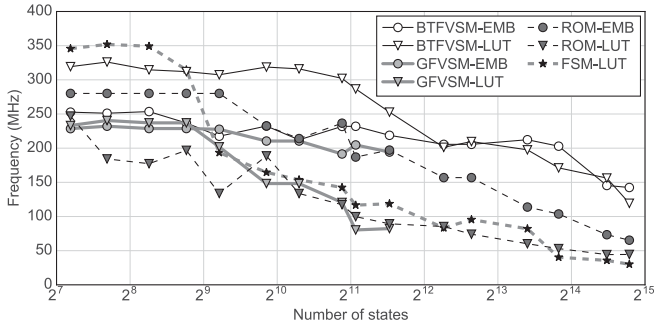
All designs have been described using VHDL. FSM-LUT implementations have been generated from a standard description of an FSM. In the rest of implementations, the standard components (like ROMs, multiplexors, etc.) are described using the VHDL template provided by the synthesis tool. In the case of FVSM implementations, M-memory is implemented using ASYMRAM description [18]. The clock signals for the BT-FVSM architecture are generated using the Xilinx Clock Generator IP core.

The VHDL descriptions of the BT-FVSM and the general FVSM implementations have been automatically generated from the BT-FSMs. In the case of the BT-FVSM implementations, the execution time ranges from less than 1 s to 1.8 h, with an average value of 14.3 min. In the case of the general FVSM implementations, it has only been possible to generate the ten smallest cases because in the rest of cases the generation was aborted after a time limit of 25 days. The execution time grows enormously with the number of states (e.g., the largest generated FVSM spent 23.6 days whereas the corresponding BT-FVSM was generated in only 62 s).

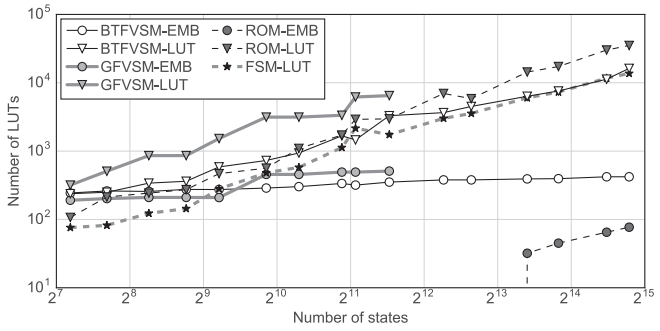
All designs have been synthesized and implemented using high-effort speed optimization because the goal of the proposed technique is to obtain high-performance implementations. In FSM-LUT implementations, *fsm_extract*, *fsm_encoding*, and *fsm_style* options were set to "yes," "auto," and "lut," respectively. In the rest of implementations, *rom_extract* was set to yes; *rom_style* was set to lut in LUT-based implementations and to "block" in EMB-based implementations. In the place-and-route stage, a timing constraint for the clock signal was set to the value obtained in the synthesis stage. The post place-and-route results are shown in Fig. 6.

A. Speed Results for Synthetic Test Benches

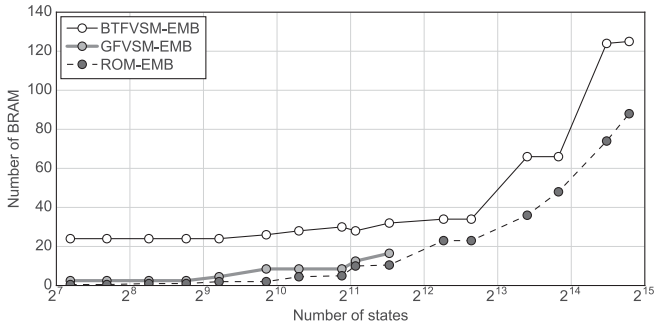
Fig. 6(a) shows the maximum operating frequency versus the number of BT-FSM states. The results show that either or both BT-FVSM implementations are the best options in all cases except the four smallest BT-FSMs (in which FSM-LUT is on average only 7% faster than BTFVSM-LUT). By considering all cases, the average improvement of BTFVSM-LUT and BTFVSM-EMB with respect to the best results of the other implementations are 36% and 23%, respectively (this measure achieves 41% when the best result of both BT-FVSM implementations is considered). In 25% of cases, the improvement is greater than 56% and 44% for BTFVSM-LUT and



(a)



(b)



(c)

Fig. 6. Experimental results for the synthetic test benches (Spartan-6). (a) Maximum operating frequency (*x*-axis is in logarithmic scale). (b) Number of LUTs (both axes are in logarithmic scale). (c) Number of EMBs (*x*-axis is in logarithmic scale).

BTFVSM-EMB, respectively (there are some cases in which the speed is more than double).

BTFVSM-LUT is faster than BTFVSM-EMB in BT-FSMs with a number of states less than 2^{12}. In these cases, the implementation of S-memory is more efficient when distributed memory is used because the depth is small [11], [17] (e.g., S-memory has only 166 words for the greatest BT-FSM with less than 2^{12}). For larger BT-FSMs, the results show that BTFVSM-LUT and BTFVSM-EMB have similar speed. The trend of the results suggest that BT-FSMs with very many states can be implemented more efficiently by using EMBs (this fact is confirmed in Section V-D).

As regards the general FVSM architecture, GFVSM-EMB and GFVSM-LUT never are the fastest option. In the one hand, BTFVSM-LUT is always faster than GFVSM-LUT (with an average speed increment of 81%). On the other hand, BTFVSM-EMB is faster than GFVSM-EMB in all cases except one.

The fastest conventional technique for more than 2^9 states is ROM-EMB. However, BTFVSM-LUT is faster than ROM-EMB in all cases, while BTFVSM-EMB is in the BT-FSMs with more than 2^{11} states. This is mainly due to two factors. First, as S-memory is controlled by a half-frequency clock, the speed of a BT-FVSM implementation doubles the speed of its S-memory. Second, the depth of S-memory is small (e.g., the depth for the largest test bench is 1728 in BT-FVSM and 56 533 in ROM-EMB). This allows S-memory of BTFVSM-LUT to be implemented efficiently using distributed memory, notably in the cases with less than 2^{11} states. In contrast, the number of EMBs used by BTFVSM-EMB is large compared to ROM-EMB [see Fig. 6(c)]; due to the routing overhead, BTFVSM-EMB is not competitive with ROM-EMB in the first five cases, in which ROM-EMB uses no more than 2 EMBs.

B. LUT Results for Synthetic Test Benches

Fig. 6(b) shows the number of LUTs versus the number of BT-FSM states. As regards implementations that use EMBs, ROM-EMB is the one that spends the least number of LUTs. On the other hand, GFVSM-EMB uses less LUTs than BTFVSM-EMB in the smallest five cases; however, this trend is reversed in the largest cases. LUTs are used for a different purpose in each kind of implementation. ROM-EMB only spends LUTs in large BT-FSMs in order to join EMBs. However, BTFVSM-EMB and GFVSM-EMB do not use LUTs for joining EMBs because the depth of S-memory is small. LUTs are used to implement the two multiplexers of the architecture in BTFVSM-EMB and to implement M-memory in GFVSM-EMB. The results suggest that in GFVSM-EMB the number of used LUTs grows faster with the number of states than in BTFVSM-EMB.

Regarding implementations that do not use EMBs, FSM-LUT spends the least number of LUTs in all cases except two (in which the best option is BTFVSM-LUT). GFVSM-LUT always gets the worst results. The average reduction of LUTs obtained by FSM-LUT respect to BTFVSM-LUT is 30%. The results show that this reduction decreases with the number of states; e.g., the average reduction is 52% in the cases of the first half of the graph whereas this value is 7% in the cases of the second half. On the other hand, BTFVSM-LUT obtains worse results than ROM-LUT when the number of states is less than 2^{10}. However, for BT-FSMs with more than 2^{12} states, BTFVSM-LUT is a better option. On average, ROM-LUT spends a 38% more LUTs than BTFVSM-LUT.

C. EMB Results for Synthetic Test Benches

Fig. 6(c) shows the number of EMBs versus the number of BT-FSM states. The average reduction of EMBs obtained by ROM-EMB respect to BTFVSM-EMB is 67%. On the other hand, for the ten smallest cases, GFVSM-EMB uses on average 75% less EMBs than BTFVSM-EMB. BTFVSM-EMB uses the largest number of EMBs due to the following reasons. First, even after applying the procedure for joining incomplete

TABLE IV
EXPERIMENTAL RESULTS FOR THE IP ROUTING DATABASE (VIRTEX-6)

Implementation	Avg. Freq. (MHz)	Avg. # of EMBs	Avg. # of LUTs
BTFVSM-EMB	286	131	446
BTFVSM-LUT	170	0	31950
ROM-EMB	112	101	19

sub-FSMs, the number of sub-FSM with less than six states may be significant, resulting in a wastage of memory. Second, the configuration of the depth of EMBs is limited to powers of two with a minimum value of 512 [14]; therefore, part of the address space of some EMBs is wasted when the depth of the ROM to be implemented does not match the available configurations of EMBs (we will refer to this effect as *memory fragmentation* [11]). This fragmentation particularly affects BTFVSM-EMB because each one of the four blocks of S-memory has less depth than the ROM of ROM-EMB and than the S-memory of GFVSM-EMB. In addition, S-memory blocks of BTFVSM-EMB usually have a very large width, so the number of EMBs that suffer memory fragmentation is high. For example, in the BT-FSM with the least number of states, ROM-EMB requires a ROM of 292×15 bits; so, only one half EMB configured as 512×16 is used. The S-memory of GFVSM-EMB is a ROM of 64×90 bits, which requires 5 half EMBs configured as 512×36 (i.e., a total of 2,5 EMBs). Finally, each one of the four S-memory blocks of BTFVSM-EMB is a ROM of 8×192 bits, which requires six EMBs configured as 512×36 (making a total of 24 EMBs).

The relative influence of the memory fragmentation in the EMB usage decreases with depth; this explains that the reduction reached by ROM-EMB respect to BTFVSM-EMB decreases with the number of states (the average reduction is 92% for the cases in the first half of the graph whereas this value is 42% in the cases of the second half). On the other hand, the size of S-memory of GFVSM-EMB is on average a 17% greater than that of BTFVSM-EMB, so it is expected that BTFVSM-EMB uses a less number of EMBs if the number of states is large enough.

With the aim of reducing the effect of the memory fragmentation in BTFSVM-EMB implementations, two S-memory blocks with depth less than 256 words could be mapped into the same EMBs by exploiting the dual-port feature available on EMBs [14]. In this case, the average EMB reduction respect to BTFVSM-EMB could be decreased to 59% in the case of ROM-EMB and to 55% in the case of GFVSM-EMB.

D. Results for the BT-FSMs Obtained from the IP Routing Database

The average values of the post place-and-route results are summarized in Table IV. BTFVSM-EMB is faster than BTFVSM-LUT in all cases. This confirms the trend observed in the results obtained for synthetic test benches. The average speed improvement of BTFVSM-EMB respect to BTFVSM-LUT is 72%. On the other hand, both implementations of the BT-FVSM architecture are faster than ROM-EMB in all cases. The average speed improvement of BTFVSM-EMB and BTFVSM-LUT respect to ROM-EMB are 155% and 53%, respectively.

VI. CONCLUSION

In this paper, the BT-FVSM architecture has been proposed. The goal of this architecture is to achieve high-speed implementations of BT-FSMs. The differences between the proposed architecture and the general FVSM architecture have been described in detail. An experimental study that include BT-FVSM, general FVSM, and conventional implementations has been presented. Both synthetic BT-FSMs and BT-FSMs obtained from an IP routing database have been used.

In synthetic test benches, the average speed improvement of the proposed architecture with respect to the best results of the other approaches (including the general FVSM) is 41% (there are some cases in which the speed is more than double). In the case of IP routing database, the average speed improvement achieves 155%. Therefore, considering all test benches, BTFVSM-LUT is the best option if EMBs must be preserved (FSM-LUT is slightly better only for BT-FSMs with less than 512 states), otherwise BTFVSM-EMB is the best option for BT-FSMs with a large number of states.

As future work, we plan to improve the area and speed results by using a more complex algorithm for joining incomplete sub-FSMs. In addition, we are going to generate BT-FVSM implementations that use the dual-port available on EMBs in order to reduce memory fragmentation.

REFERENCES

[1] M. P. Desai, H. Narayanan, and S. B. Patkar, "The realization of finite state machines by decomposition and the principal lattice of partitions of a submodular function," *Discrete Appl. Math.*, vol. 131, no. 2, pp. 299–310, Sep. 2003.

[2] M. Desai, R. Gupta, A. Karandikar, K. Saxena, and V. Samant, "Reconfigurable finite-state machine based IP lookup engine for high-speed router," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 501–512, May 2003.

[3] J. Li, Y. Chen, C. Ho, and Z. Lu, "Binary-tree-based high speed packet classification system on FPGA," in *Proc. Int. Conf. Inf. Netw.*, Jan. 2013, pp. 517–522.

[4] S. Beak *et al.*, "Novel binary tree Huffman decoding algorithm and field programmable gate array implementation for terrestrial-digital multimedia broadcasting mobile handheld," *IET Sci. Meas. Technol.*, vol. 6, no. 6, pp. 527–532, Nov. 2012.

[5] A. Oliveri, A. Oliveri, T. Poggi, and M. Storace, "Circuit implementation of piecewise-affine functions based on a binary search tree," in *Proc. Eur. Conf. Circuit Theory Design*, Antalya, Turkey, Aug. 2009, pp. 145–148.

[6] P. Brox *et al.*, "A programmable and configurable ASIC to generate piecewise-affine functions defined over general partitions," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 12, pp. 3182–3194, Dec. 2013.

[7] S. S. Roy, F. Vercauteren, and I. Verbauwhede, *High Precision Discrete Gaussian Sampling on FPGAs*. Heidelberg, Germany: Springer, 2014, pp. 383–401.

[8] S. L. Aritz, "Analysis of the FSMs implementation with mini-microprocessors in FPGAs," in *Proc. IEEE Int. Symp. Ind. Electron.*, Vigo, Spain, Jun. 2007, pp. 2290–2294.

[9] X. Zhang *et al.*, "Ztcore: Zero-waste tiny core for real-time control within FPGA," in *Proc. IEEE Int. Symp. Ind. Electron.*, Seoul, South Korea, Jul. 2009, pp. 359–363.

[10] R. Senhadji-Navarro and I. Garcia-Vargas, "Finite virtual state machines," *IEICE Trans.*, vol. 95-D, no. 10, pp. 2544–2547, 2012.

[11] R. Senhadji-Navarro, I. Garcia-Vargas, and J. L. Guisado, "Performance evaluation of RAM-based implementation of finite state machines in FPGAs," in *Proc. IEEE Int. Conf. Electron. Circuits Syst.*, Seville, Spain, 2012, pp. 225–228.

[12] *XST User Guide 13.1*, Xilinx, San Jose, CA, USA, 2011.

[13] I. Garcia-Vargas and R. Senhadji-Navarro, "Finite state machines with input multiplexing: A performance study," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 5, pp. 867–871, May 2015.
[14] *Spartan-6 FPGA Block RAM Resources User Guide*, Xilinx, San Jose, CA, USA, 2011.
[15] A. Barkalov, L. Titarenko, M. Kolopienczyk, K. Mielcarek, and G. Bazydlo, *Design of EMB-Based Mealy FSMs*. Cham, Switzerland: Springer, 2016, pp. 193–237.
[16] *Cyclone V Device Overview*, Altera, San Jose, CA, USA, 2016.
[17] *Spartan-6 FPGA Configurable Logic Block User Guide*, Xilinx, San Jose, CA, USA, 2010.
[18] R. Senhadji-Navarro, I. Garcia-Vargas, G. Jimenez-Moreno, and A. Civit-Balcells, "FPGA-based implementation of RAM with asymmetric port widths for run-time reconfiguration," in *Proc. IEEE Int. Conf. Electron. Circuits Syst.*, Marrakesh, Morocco, Dec. 2007, pp. 178–181.
[19] R. Senhadji-Navarro and I. Garcia-Vargas, "Minimum maximum reconfiguration cost problem," *Optim. Lett.*, vol. 10, no. 3, pp. 605–617, 2016.
[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Series of Books in the Mathematical Sciences). New York, NY, USA: W. H. Freeman, Jan. 1979.