IEICE TRANS. INF. & SYST., VOL.E95–D, NO.10 OCTOBER 2012

LETTER

Finite Virtual State Machines

Raouf SENHADJI-NAVARRO[†a)], *Nonmember and* **Ignacio GARCIA-VARGAS**[†b)], *Member*

SUMMARY This letter proposes a new model of state machine called Finite Virtual State Machine (FVSM). A memory-based architecture and a procedure for generating FVSM implementations from Finite State Machines (FSMs) are presented. FVSM implementations provide advantages in speed over conventional RAM-based FSM implementations. The results of experiments prove the feasibility of this approach.

key words: *finite state machine, reconfiguration, RAM, FPGA*

1. Introduction

The inclusion of embedded memory blocks in FPGA devices has motivated a revival of interest in memory-based implementations. Manufacturers, like Xilinx or Altera, are beginning to provide tools to map logic into embedded memory blocks. Recently, many research works are focused on RAM-based implementations of Finite State Machines (FSMs) [1]–[7]. Different approaches have been proposed to improve the performance of these implementations (such as speed, area or power consumption) over conventional cell-based implementations [1]–[4]. In addition, as the functionality of a RAM-based FSM is defined by the data stored in the memory, these implementations offer a simple way to dynamically modify the functionality (adding or updating transitions and outputs by means of write operations). This run-time reconfiguration offers some advantages like being independent on the placement and routing and being applicable even on FPGAs without dynamic partial reconfiguration capabilities [6], [7].

Nowadays, there is a great interest in developing hardware implementations of problems traditionally solved by software in order to achieve the demanded high-performance. Two prominent examples of such problems are pattern matching and packet routing in networks [8], [9]. The transformation of these problems into equivalent state machines results in FSMs with a very large number of states (from a few thousand to millions). These large FSMs require novel approaches to obtain efficient implementations.

In this application context, the authors propose a new model of state machine called Finite Virtual State Machine (FVSM) to improve the performance of RAM-based FSM implementations. This model is inspired by the memory hierarchy of computer science. To the best knowledge of the authors, multiple memory levels and locality of memory references concepts have never been used in RAM-based FSM designs. The proposed model uses a memory hierarchy of two levels and exploits the locality of the FSM state references. However, unlike the cache accesses done by a processor, in which the data may be unavailable, an FSM must ensure the correct value of the output in any clock cycle. By using the cache analogy, the proposed approach can be viewed as a cache prefetching scheme where no cache miss is allowed.

The proposed architecture has two memory levels: main and secondary memory. The main memory implements a RAM-based FSM. Taking into account that only a subset of the states are reachable in a given period of time, the FSM is decomposed into different subFSMs, each of which is composed by the needed states for the FSM operation during a different period of time. At each moment, the subFSM stored in main memory is the active one. On the other hand, secondary memory stores information about all subFSMs.

The FVSM is designed in such a way that any particular subFSM needed by the evolution of the FSM is dynamically transferred from secondary to main memory before any of its states are reached. Unlike other approaches [7], [9], this transfer is not controlled by an external circuit, but by the active subFSM and it is done without interrupting the proper FSM operation. The aim of the proposed approach is to increase the operation speed by exploiting the fact that the RAM-based implementation of the subFSM on main memory has a better performance than the more complex full FSM. On the other hand, the model offers some advantages when it is used on reconfigurable applications. Any subFSM (even the active one) can be reconfigured in secondary memory by a host while the active subFSM continues in operation in main memory. So, the reconfiguration process can be done without interrupting the FSM operation.

2. Definition and Implementation of an FVSM

An FSM is a 6-tuple (X, Y, S, g, h, s_0) where X, Y, and S are a finite set of inputs, outputs, and states, respectively; $g : S \times X \rightarrow S$ is the transition function, $h : S \times X \rightarrow Y$ is the output function, and $s_0 \in S$ is the initial state. Let us define an FVSM as a 9-tuple $(X, Y, F, I, t, o, u, i_0, f_0)$ where X, Y, F and I are a finite set of inputs, outputs, state frames (hereinafter called frames), and machine instances (hereinafter

Manuscript received November 18, 2011.
Manuscript revised April 18, 2012.

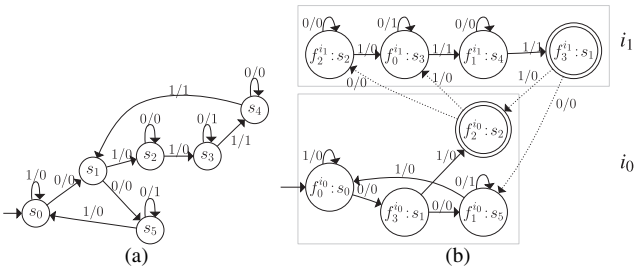
[†]The authors are with the Computer Architecture and Technology Department of the University of Seville, Spain.
a) E-mail: raouf@us.es
b) E-mail: iggv@us.es

DOI: 10.1587/transinf.E95.D.2544

called instances), respectively. The states of the FVSM (called virtual states) are given by the pairs $(i, f) \in I \times F$. The functions $t : I \times F \times X \rightarrow F$, $o : I \times F \times X \rightarrow Y$, and $u : I \times F \rightarrow I$ are the transition, output and update functions, respectively. The initial virtual state is (i_0, f_0) where i_0 and f_0 are the initial instance and frame, respectively. Given the present virtual state (i, f), the next frame and instance are determined by the transition function and the update function, respectively; so, the next virtual state is $(u(i, f), t(i, f, x))$ where $x \in X$. An instance change occurs when the next instance is different to the present one. The virtual state (i, f) is called an update state of i' if $u(i, f) = i'$ and $i' \neq i$. Our goal is to model a given FSM as an FVSM. Let us say that an FSM and an FVSM are equivalent if it exists a partial function $v : I \times F \rightarrow S$ such that $v(i_0, f_0) = s_0$ and $\forall\ s \in S$, $\forall\ x \in X$, $\exists\ (i, f) \in I \times F$ which satisfies the following condition: $v(i, f) = s$, $v(u(i, f), t(i, f, x)) = g(s, x)$, and $o(i, f, x) = h(s, x)$. Each instance determines a different subset of FSM states which are stored on the frames of main memory. These (usually non-disjoint) subsets define for each instance a different subFSM where the frames play the role of the FSM states. Figure 1 (a) and Fig. 1 (b) show an example of FSM and an equivalent FVSM. The subFSM defined by each instance is represented as a State Transition Graph (STG) where circles represent frames. In the STG of the instance i, the circle representing the frame f is labeled "$f^i : s$" if $v(i, f) = s$. The update states are denoted by double-line circles; and transitions involving instance changes, by slashed arcs.

Figure 2 shows the general architecture of an FVSM. It is composed of two memories (main and secondary memory) controlled by the same clock signal. Main memory is a simple dual-port memory with asymmetric port widths configured in read-after-write mode [6]. The write port allows the modification of the active subFSM while the FSM is operating by using the read port. The transitions of the active subFSM, composed by the next frame encoding bits and the outputs, are stored in main memory. An instance change consists in transferring from secondary to main memory the data needed to transform the present instance (i.e., the active subFSM) into the next instance. These data, called an *instance update*, are the transitions of the virtual states in which both instances differ. The address where an *instance update* must be stored in main memory is called *update address*. As the read port operates at transition-level, the write port width must be multiple of the read port width in order to transfer the *instance update* in a single write operation. The ratio between the width of write and read data port (called aspect ratio) is equal to the number of transitions of the largest *instance update*. FPGA resources are very suitable for implementing these asymmetric memories due to the high aspect ratio of distributed RAMs (unlimited) and block RAMs (up to 64 in each Xilinx memory blocks).

Each secondary memory word is composed by an *instance update* and its corresponding *update address*. It is addressed by the *instance update selection* signal. The operations to load an instance requite two clock cycles. In the



(a) (b)

clk	0	1	2	3	4	5	6	7	8	9	...
in	0	1	0	1	1	0	1	0	1	...	...
out	0	0	0	0	0	1	0	1	0	0	...
ps	s_0	s_1	s_2	s_2	s_3	s_4	s_4	s_1	s_5	s_0	...
pi	i_0	i_0	i_0	$\boldsymbol{i_1}$	i_1	i_1	i_1	i_1	$\boldsymbol{i_0}$	i_0	....
pf	f_0	f_3	f_2	$\boldsymbol{f_2}$	f_0	f_1	f_1	f_3	$\boldsymbol{f_1}$	f_0	....
f_0	T_{s_0}	T_{s_0}	T_{s_0}	$\boldsymbol{T_{s_3}}$	T_{s_3}	T_{s_3}	T_{s_3}	T_{s_3}	$\boldsymbol{T_{s_0}}$	T_{s_0}	...
f_1	T_{s_5}	T_{s_5}	T_{s_5}	$\boldsymbol{T_{s_4}}$	T_{s_4}	T_{s_4}	T_{s_4}	T_{s_4}	$\boldsymbol{T_{s_5}}$	T_{s_5}	...
f_2	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	T_{s_2}	...
f_3	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	T_{s_1}	...
ue	0	0	1	0	0	0	0	1	0	0	...
ius	-	1	-	-	-	-	0	-	-	-	...
ua	-	-	0	-	-	-	-	0	-	-	...
iu	-	-	T_{s_3}, T_{s_4}	-	-	-	-	T_{s_0}, T_{s_5}	-	-	...

ue: update enable signal; ius: instance update selection signal
ua: update address signal; iu: instance update signal

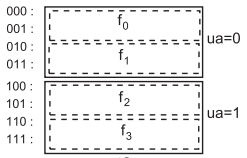
(c)

	Transitions for in = 0				Transitions for in = 1			
	nf	out	ue	ius	nf	out	ue	ius
T_{s_0}	f_3	0	0	1	f_0	0	0	-
T_{s_1}	f_1	0	0	-	f_2	0	1	-
T_{s_2}	f_2	0	0	-	f_0	0	0	-
T_{s_3}	f_0	1	0	-	f_1	1	0	0
T_{s_4}	f_1	0	0	0	f_3	1	1	-
T_{s_5}	f_1	1	0	-	f_0	0	0	-

(d)

Memory address	Memory data iu	ua
0	T_{s_0}, T_{s_5}	0
1	T_{s_3}, T_{s_4}	0

(e)



(f)

Fig. 1 (a) An FSM example, (b) an equivalent FVSM, (c) execution trace, (d) information related to each virtual state, (e) secondary memory content, and (f) main memory organization in read and write operations.

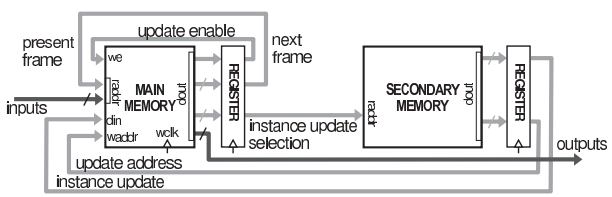


Fig. 2 General architecture of an FVSM.

first one, the *instance update* and its *update address* are read from secondary memory by setting the *instance update selection* signal. In the second one, the *instance update* is written on main memory at the *update address* by setting the *update enable* signal. The read and write operations are pipelined by using the registers allowing a throughput of one update per cycle. This update process is controlled by the active subFSM and it is carried out simultaneously with the FSM operation. So, the transitions of the FVSM must be extended with the *instance update selection* and *update enable*

signals.

Figure 1 (c) shows an execution trace of the FVSM example. Given any sequence of inputs (*in*), the sequence of outputs (*out*) generated by both the FSM and its equivalent FVSM are the same. In each clock cycle (*clk*), the trace shows the FSM present state (*ps*), the FVSM present frame (*pf*), the FVSM present instance (*pi*), the data contained by the frames in main memory (these data are detailed in Fig. 1 (d), and the control signals. The instances changes occur in cycles 3 and 8 (these changes are shown in bold type). For example, in cycle 8, the present instance changes from i_1 to i_0. This update process starts in cycle 6 by setting *ius* signal to zero in order to read the *instance update* of i_0 from secondary memory (Fig. 1 (e) shows the secondary memory content). In cycle 7, the *ue* signal enables the write operation of the *instance update* of i_0 on main memory. In cycle 8, the write operation is done. As Fig. 1 (f) shows, each write operation modifies two frames (i.e., four transitions). As main memory operates in read-after-write mode, the output takes the value 0 just in cycle 8, which corresponds to the new virtual state (s_5) stored in f_1 (the present frame in this cycle) by the update process. The described two-cycles update process is the same in any instance change of any FVSM.

In an FPGA, the speed of a distributed RAM decreases with depth because it is composed by smaller RAM components connected via multiplexors and decoders, whose complexity grows with depth. This speed degradation can also be important when using little memory blocks or memory blocks in power-aware design [10]. The critical path of the FVSM architecture is imposed by the memory with the higher depth in read operation (the depth of the main memory in write operation is always less than in read operation, so it has no influence on the critical path). The main memory depth in read operations increase with the number of inputs and the number of frames. The number of frames depends on the number of virtual states of the largest instance and the required aspect ratio. This ratio grows with the number of inputs and the number of virtual states of the largest *instance update*. On the other hand, the secondary memory depth grows with the number of *instance updates*. An optimization algorithm, called virtualization algorithm, is used to find the FVSM implementation with the highest speed. This algorithm minimizes the maximum depth of both memories (hereinafter called FVSM depth) taking into account the above mentioned parameters.

3. Virtualization Algorithm

The optimization problem of finding the best FVSM implementation is solved by a branch-and-bound algorithm whose candidate solutions are FVSMs. The algorithm dynamically constructs the tree of candidate solutions, each of which is defined by a subset of FSM states which will be the update states of the FVSM. The fact that the secondary memory only has one read port imposes restrictions over the implementability of the FVSM. Not all candidate solutions can be implemented in the proposed architecture. For example, in the FSM of Fig. 1 (a), the states s_2 and s_5 can not simultaneously be update states of an FVSM implementable in the proposed architecture because it requires to read two different instances updates from the secondary memory in the same clock cycle (transition from s_0 to s_1). So, only candidate solutions that are implementable are feasible solutions. The initial candidate solution (the root of the tree) is an FVSM where all FSM states are update states. Each child node of a candidate solution is created by selecting a different state to be deleted from the set of update states. So, the main memory depth of a node is less than or equal to that of their child nodes. This depth is used as a lower bound of the FVSM depth of any descendant node. Therefore, the subtree of a candidate solution can be discarded if its main memory depth is greater than or equal to that the best known solution.

Each update state determines a different instance which must be loaded at any output transition of this state. An instance is composed by all the virtual states that can be reached from its update state to any different update state. As the next instance must be loaded after its update state is reached, this state must also be included in the present instance. For example, in Fig. 1 (b), the FVSM is created from an arbitrary candidate solution with s_1 and s_2 as update states (see $f_3^{i_1} : s_1$ and $f_2^{i_0} : s_2$ in Fig. 1 (b)). When i_0 is the present instance, i_1 is loaded after s_2 is reached; so, i_0 must be composed by the virtual states s_0, s_1, s_5, and s_2.

4. Experimental Results

Table 1 compares the speed of implementations of RAM-based FSM and FVSM architectures. The FSM architecture is parametrized by the number of state encoding bits (*seb*) and the number of inputs (*in*). The parameters of the FVSM architecture are *in*, the number of frame encoding bits (*feb*), and the number of *instance update* encoding bits (*iueb*). Each case of Table 1 corresponds to the parameter

Table 1 Comparison between FSM and FVSM architecture.

in	seb	feb	iueb	imp (%)	in	seb	feb	iueb	imp (%)	in	seb	feb	iueb	imp (%)
1	7	4	6	24	1	11	8	8	39	3	9	4	8	79
1	7	5	5	23	1	12	7	8	96	3	9	5	7	72
1	8	4	7	45	1	12	8	7	83	3	9	5	8	58
1	8	4	8	31	1	12	8	8	45	**3**	**9**	**6**	**7**	**54**
1	8	5	6	38	2	6	4	5	25	**3**	**9**	**6**	**8**	**40**
1	8	5	7	26	2	8	4	7	24	3	10	5	8	101
1	8	5	8	20	2	8	4	8	20	3	10	6	7	97
1	8	6	5	25	2	8	5	6	23	**3**	**10**	**6**	**8**	**79**
1	8	6	6	22	2	9	4	8	53	3	10	8	5	35
1	8	7	4	23	2	9	5	7	38	3	10	8	6	32
1	9	4	8	35	2	9	5	8	34	3	11	6	8	88
1	9	5	7	29	2	9	6	6	40	3	11	8	6	38
1	9	5	8	23	2	9	6	7	37	4	7	4	6	40
1	9	6	6	26	2	9	6	8	22	4	7	4	7	32
1	9	7	5	20	2	10	5	8	72	4	7	4	8	20
1	10	5	8	55	2	10	6	7	75	4	8	4	7	77
1	10	6	7	50	2	10	6	8	57	4	8	4	8	62
1	10	6	8	36	**2**	**10**	**7**	**7**	**36**	4	8	5	6	41
1	10	7	6	44	2	10	7	8	31	4	8	5	7	47
1	10	7	7	34	2	11	6	8	91	4	8	5	8	38
1	10	7	8	28	2	11	7	7	66	4	8	6	5	22
1	10	8	5	22	2	11	7	8	61	4	9	4	8	65
1	10	8	7	20	2	12	7	8	138	4	9	5	7	51
1	11	6	8	101	2	12	8	7	46	4	9	5	8	41
1	11	7	7	97	3	6	4	5	22	4	10	5	8	66
1	11	7	8	89	3	8	4	7	29	4	10	7	6	22
1	11	8	6	42	3	8	4	8	22					
1	11	8	7	77	3	8	5	6	21					

Table 2 Comparison between FSM and FVSM implementation of a set of testbenches.

FSM testbench				Implementation			
Name	s	t	in	FSM	FVSM		imp
				seb	feb	iueb	(%)
fsm01	567	1066	2	10	7	7	36
fsm03	424	657	3	9	6	8	40
fsm06	274	406	3	9	6	7	54
fsm07	276	471	3	9	6	7	54
fsm08	518	947	2	10	7	7	36
fsm13	401	630	3	9	6	8	40
fsm16	526	872	3	10	6	8	79
fsm17	542	861	3	10	6	8	79
fsm18	325	495	3	9	6	7	54

values required to implement the same state machine in both architectures. The column *imp* represents the percentage of increase in clock frequency of the FVSM over the RAM-based FSM architecture. None of the architectures has been parametrized by the number of outputs because their performance is not affected by this parameter (the depth of the memories is independent of this parameter). Both architectures were implemented in a Xilinx xc2v8000-4 FPGA using distributed RAM. The results show the important speed improvement that can be achieved by using the FVSM architecture (Table 1 only contains those cases whose improvement is equal or greater than 20%). As we can see in Table 1, the same state machine could be implemented with different values of *feb* and *iueb* parameters depending on the effectiveness of the virtualization algorithm. Therefore, this algorithm is a key to improve the operation speed.

In order to demonstrate that the speed improvements of the FVSM architecture shown in Table 1 can be achieved in state machine implementations, a set of 20 FSM testbenches has been virtualized and implemented in the same FPGA device. In almost half the cases, the speed improvement is greater than 33%. For each case, Table 2 shows the number of states (*s*), the number of transitions (*t*), the number of inputs (*in*), the parameters of the FSM and FVSM implementations (*seb*, *feb* and *iueb*) and the speed improvement (*imp*). This speed improvement depends on the FVSM parameter values reached by the virtualization algorithm for each testbench. The values of the parameters *in*, *seb*, *feb*, and *iueb* shown in Table 2 correspond to those values shown in bold type in Table 1.

5. Conclusions and Future Work

The experimental results show that the proposed approach offers a significant improve in the operation speed with respect to conventional RAM-based FSM implementations. In general, more important improvements are obtained for large FSMs, what makes this approach very suitable for hardware implementation of problems traditionally solved by software (like pattern matching and packet routing). Once the usefulness of the approach is proved, the authors are studying how to take further advantage of the capabilities of the proposed model. In many cases, the virtualization obtained by the branch-and-bound algorithm is not optimal. New strategies to refine the algorithm are being studied. On other hand, as explained in Sect. 2, the aspect ratio of an FVSM grows with the number of FSM inputs. Different approaches are being studied to reduce the number of inputs connected to the main memory (such as FSMs with input multiplexing [1] or dummy states [5]). This eases the implementation of FVSMs by using embedded memory blocks which have a limited aspect ratio. Another approach for reducing the required ratio consists in doing write operations at a clock frequency multiple of that of read operations. This allows the reduction of the write data port width.

Furthermore, the authors are considering the use of more than one port in secondary memory (e.g., Xilinx true dual-port memory blocks). This architecture improvement would allow a relaxation of the restriction imposed by the single port over the implementability of FVSMs. The number of update states could then be higher and thus the instances would have a smaller size. Therefore, a speed increase would be obtained. In addition, the power consumption of a FVSM implementation can be reduced respect to the conventional RAM-based implementation if the secondary memory is only enabled when a transfer is required. Based on this strategy, the authors are studying the use of the FVSM in low-power design. Moreover, the authors are studying the addition of a new level of hierarchy to the model in order to extend the capacity of the target device with a new memory level stored in external off-chip memory. This could allow the implementation of large FSMs that exceed the capacity of the target device.

References

[1] I. Garcia-Vargas et al., "Rom-based finite state machine implementation in low cost FPGAs," 2007. ISIE 2007. IEEE International Symposium on, Industrial Electronics, pp.2342–2347, June 2007.

[2] A. Tiwari and K.A. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in FPGAs," Proc. Conference on Design, automation and test in Europe, 2004.

[3] G. Borowik et al., "Cost-efficient synthesis for sequential circuits implemented using embedded memory blocks of FPGA's," Proc. IEEE DDECS '07, pp.99–104, 2007.

[4] V. Sklyarov, "Synthesis and implementation of ram-based finite state machines in FPGAs," Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing, Lect. Notes Comput. Sci., vol.1896, pp.718–727, 2000.

[5] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs," J. Systems Architecture, vol.47, no.14-15, pp.1043–1064, 2002.

[6] R. Senhadji-Navarro et al., "Fpga-based implementation of ram with asymmetric port widths for run-time reconfiguration," Proc. IEEE Int. Conf. Electronics, Circuits and Systems, pp.178–181, 2007.

[7] M. Koster and J. Teich, "(self-)reconfigurable finite state machines: theory and implementation," Design, Automation and Test in Europe Conference and Exhibition, 2002. Proc., pp.559–566, 2002.

[8] M. Desai et al., "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.

[9] N. Rafla and I. Gauba, "A reconfigurable pattern matching hardware implementation using on-chip ram-based FSM," IEEE Int. Midwest Symposium on, Circuits and Systems, pp.49–52, Aug. 2010.

[10] R. Tessier et al., "Power-aware ram mapping for FPGA embedded memory blocks," Proc. ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, pp.189–198, 2006.