ROM-based FSM implementation using input multiplexing in FPGA devices

R. Senhadji-Navarro, I. García-Vargas, G. Jiménez-Moreno and A. Civit-Ballcels

A new approach for ROM implementation of finite state machines (FSMs) is proposed, based on the selection of a subset of inputs in each state using multiplexers. This technique has been applied to different FSM standard benchmarks and very good results have been obtained.

Introduction: The growing interest in the implementation of finite state machines (FSMs) based on ROM has been motivated by the inclusion of embedded memory blocks in modern field programmable gate array (FPGA) devices. In many cases, these resources are not used and can be exploited to implement sequential circuits, saving logic cells for other purposes [1, 2]. However, in a ROM-based FSM implementation, the size of the required memory increases exponentially with the number of inputs and state encoding bits. This aspect is critical because little FSMs may require more memory than available in the FPGA.

Different techniques for reducing the amount of memory used by a ROM-based FSM implementation have been reported in the literature [2, 3]. They make use of additional resources to reduce the ROM size. Some techniques, which are applied to microcode optimisation, exploit the fact that not all inputs of an FSM have influence in all states. Only the inputs that have influence on each state are selected by multiplexing. Thus, the ROM address space depends on a smaller number of inputs.

A new approach based on input multiplexing is proposed. Besides reducing the number of inputs, the approach improves the memory usage reduction by decreasing the number of state encoding bits. Moreover, an algorithm to simplify the complexity of the multiplexers is presented. Small multiplexers can be efficiently implemented in modern FPGA devices [4, 5]. For example, a 32:1 multiplexer occupies only two configurable logic blocks (CLB) in Xilinx Virtex-II FPGAs [6]. In some FPGA families, tri-state buffers can be used to implement multiplexers without consuming logic cells [7].

Approach description: A FSM is a 5-tuple (X, Y, S, f, g) where X, Y and S are a finite set of input variables, output variables, and states, respectively; f: $XS \rightarrow S$ is the transition function and g: $XS \rightarrow Y$ is the output function. A FSM can be expressed using a tabular representation named state transition table (STT). Each row in the STT contains the 4-tuple (in, ps, ns, out), where in and out are an assignment of the input and output, respectively; ps is the present state, and ns is the next state. An STT example is shown in Fig. 1*a*.

in *a b c*	ps	ns	out
1 - -	s_0	s_1	0
0 - -	s_0	s_0	0
- 1 0	s_1	s_2	0
- 0 0	s_1	s_1	0
- - 1	s_1	s_0	0
- - 1	s_2	s_0	1
- - 0	s_2	s_2	0

a

pis in_1	pis in_2	in in_1	in in_2	ps	nis in_1	nis in_2	ns	out
a	-	1	-	s_0	*b*	*c*	s_1	0
a	-	0	-	s_0	*a*	-	s_0	0
b	*c*	1	0	s_1	*c*	-	s_2	0
b	*c*	0	0	s_1	*b*	*c*	s_1	0
b	*c*	-	1	s_1	*a*	-	s_0	0
c	-	1	-	s_2	*a*	-	s_0	1
c	-	0	-	s_2	*c*	-	s_2	0

b

pis in_1	pis in_2	in in_1	in in_2	ps	nis in_1	nis in_2	ns	out
a	0	1	0	s_{02}	*b*	*c*	s_1	0
a	0	0	0	s_{02}	*a*	0	s_{02}	0
b	*c*	1	0	s_1	*c*	1	s_{02}	0
b	*c*	0	0	s_1	*b*	*c*	s_1	0
b	*c*	-	1	s_1	*a*	0	s_{02}	0
c	1	1	1	s_{02}	*a*	0	s_{02}	1
c	1	0	1	s_{02}	*c*	1	s_{02}	0

c

Fig. 1 *FSM example*

a STT representation *b* ESTT representation
c ESTT representation after state encoding bit reduction

A FSM can be implemented using ROM [1]. Fig. 2*a* shows the general architecture of the traditional implementation. The ROM addresses are composed of the encoding present state and the FSM inputs. The ROM words contain the encoding next state and the FSM outputs. The amount of memory used by this implementation is expressed by $2^{m+p} \times (n+p)$, where m is number of inputs, n is number of outputs, and p is number of bits needed for binary encoding states.

The previous architecture can be modified as shown in Fig. 2*b* to reduce the address space of the ROM. This idea has been used in microprogramming. The multiplexer bank selects for each state the subset of input which has influence on it. These inputs are named effective inputs. Each state has a different number of effective inputs associated to it. However, the number of inputs selected by the multiplexers is fixed. Those selected inputs that are not effective are denoted by *don't care selected inputs*. Some bits, which are named selection bits, are added to the ROM word to control the multiplexers. The size of the ROM is expressed by $2^{m'+p} \times (n+p+r)$, where m' is maximum number of effective inputs per state, n is number of outputs, p is number of state encoding bits, and r is number of selection bits.

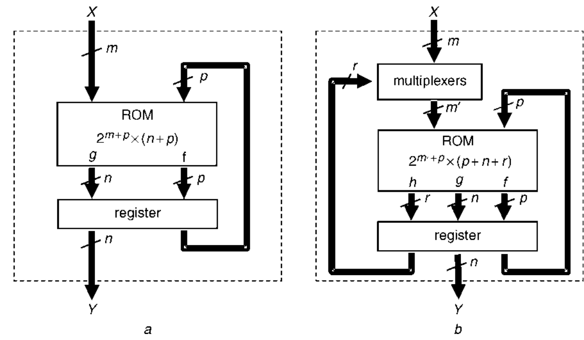

Fig. 2 *ROM-based FSM implementation*

a Traditional implementation *b* Implementation with input multiplexing

To represent a finite state machine with input multiplexing (FSMIM), the STT must be extended. The rows of this extended state transition table (ESTT) contain the 6-tuple (pis, in, ps, nis, ns, out), where the new elements pis and nis are the present and next input selection, respectively. For each state, *pis* contains information about the inputs selected and nis specifies the inputs to be selected for the next state. Fig. 1*b* shows the ESTT of the FSM described. Unlike a FSM, the states of a FSMIM are identified not only by ps, but also by pis.

In the proposed approach, the amount of memory usage is reduced by decreasing the number of state encoding bits. For this purpose, the *don't care selected inputs* in a state can also be employed to codify it. Let A and B be two states each of them with almost one *don't care selected inputs*. Let state A be defined by the pair (PS_j, PIS_j), where PS_j is the present state and PIS_j is the present input selection. In the same way, let state B be defined by (PS_k, PIS_k). Both states can be identified by the same present state PS_r. To distinguish them, the *don't care selected input* is fixed to '0' in one state and to '1' in the other. So, the states A and B are identified by (PS_r, PIS'_j) and (PS_r, PIS'_k), respectively, where PIS'_j and PIS'_k have the *don't care selected input* fixed to different values. In Fig. 1*c*, the states (s_0, *a*-) and (s_2, *c*-) have been identified by the same symbol s_{02} and distinguished by the input in_2 ('0' for s_0 and '1' for s_2). The resultant states are (s_{02}, a_0) and (s_{02}, c_1).

The complexity of the multiplexer bank and thus the number of selection bits stored in ROM depend on the manner of assigning inputs to each multiplexer. An appropriate input assignment allows reducing the complexity of multiplexers. Fig. 3 shows two possible assignments related to the FSM example. An algorithm that is based on a modification of the Knapsack algorithm [8] has been proposed for this purpose.

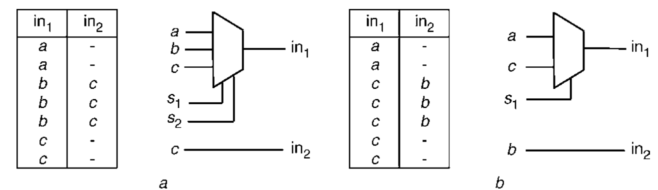

Fig. 3 *Example of multiplexer bank with and without simplification*

a With simplification *b* Without simplification

Results: The FSMIM can be implemented in modern FPGA devices using synchronous memory blocks to implement the ROM and other resources to implement the multiplexers. When tri-state buffers are available, the multiplexers can be implemented without using logic cells. Alternatively, the multiplexers can be implemented using logic cells. Despite of the tri-state buffer based implementation not requiring logic cells, the ROM size grows due to the necessary one-hot codification.

Table 1: Optimisation results (columns MUX contain list of numbers that represent count of channels of each multiplexer)

FSM	Traditional approach ROM size (bits)	Input multiplexing approach: Tri-state buffer based implementation				
		Before SEBR		After SEBR		
		ROM size (bits)	TSB	ROM size (bits)	TSB	ROM reduction (%)
bbsse	22528	7168	3	4864	7	**78.4**
cse	22 528	13 312	2	9216	6	**59.1**
ex1	393 216	61 440	6	32 768	8	**91.7**
ex4	13 312	2176	4	1216	6	**90.9**
ex6	2816	960	4	960	4	**65.9**
keyb	28 672	28 672	0	11 264	4	**60.7**
mark1	10 240	5632	2	1792	7	**82.5**
mc	224	144	2	144	2	**35.7**
opus	5120	5120	0	1792	4	**65.0**
planet	204 800	59 392	4	17 920	9	**91.3**
pma	106 496	32 768	3	18 432	5	**82.7**
s1	90 112	90 112	0	36 864	6	**59.1**
s1488	409 600	118 784	4	19 968	12	**95.1**
s1494	409 600	118 784	4	19 968	12	**95.1**
s27	512	512	0	384	2	**25.0**
s386	22 528	7168	3	4096	5	**81.8**
s510	436 207 616	8192	19	4224	20	**~100.0**
s820	201 326 592	311 296	14	88 064	18	**~100.0**
s832	201 326 592	311 296	14	88 064	18	**~100.0**
sand	917 504	102 400	11	30 720	15	**96.7**
scf	$\sim 10^{12}$	5 570 560	22	196 608	32	**~100.0**
sse	22 528	7168	3	4864	7	**78.4**
styr	245 760	77 824	4	27 648	10	**88.8**

FSM	Traditional approach ROM size (bits)	Input multiplexing approach: Logic cell based implementation				
		Before SEBR		After SEBR		
		ROM size (bits)	MUX (channels)	ROM size (bits)	MUX (channels)	ROM reduction (%)
bbsse	22 528	6656	3	4096	3, 2, 2	**81.8**
Cse	22 528	12 288	2	7680	2, 2, 2	**65.9**
ex1	393 216	55 296	2, 2, 2	28 672	2, 2, 2, 2	**92.7**
ex4	13 312	1920	4	1024	4, 2	**92.3**
ex6	2816	832	2, 2	832	2, 2	**70.5**
keyb	28 672	28 672	0	9216	2, 2	**67.9**
mark1	10 240	5376	2	1536	4, 2, 2	**85.0**
mc	224	128	2	128	2	**42.9**
opus	5120	5120	0	1536	2, 2	**70.0**
planet	204 800	55 296	2, 2	15 360	2, 2, 2, 4	**92.5**
pma	106 496	30 720	3	16 384	3, 2	**84.6**
s1	90 112	90 112	0	30 720	2, 2, 2	**65.9**
s1488	409 600	110 592	2, 2	16 384	4, 4, 2, 2, 2	**96.0**
s1494	409 600	110 592	2, 2	16 384	4, 4, 2, 2, 2	**96.0**
s27	512	512	0	320	2	**37.5**
s386	22 528	6656	3	3584	3, 2	**84.1**
s510	436 207 616	5120	14, 5	2432	14, 7	**~100.0**
s820	201 326 592	262 144	5, 4, 3, 2	69 632	5, 4, 4, 4, 2	**~100.0**
s832	201 326 592	262 144	5, 4, 3, 2	69 632	5, 4, 4, 4, 2	**~100.0**
sand	91 7504	81 920	3, 2, 2, 2, 2	22 528	4, 2, 4, 2, 2, 2	**97.5**
scf	$\sim 10^{12}$	4 784 128	13, 3, 3, 3	159 744	13, 3, 4, 4, 2, 2, 2, 2	**~100.0**
sse	22 528	6656	3	4096	3, 2, 2	**81.8**
styr	245 760	69 632	2, 2	21 504	4, 4, 2, 2	**91.3**

SEBR: state encoding bit reduction; TSB: tri-state buffer; MUX: multiplexer

The proposed techniques have been applied to a set of FSM standard benchmarks [9]. Both tri-state buffer and logic cell based implementations have been used. Table 1 shows the obtained results. The optimisation process consists of two steps. The multiplexer bank simplification has been applied to the generated FSMIMs before using the state encoding bit reduction (SEBR) technique (see columns 'Before SEBR'). Next, the SEBR technique has been applied to the results obtained in the previous step (see columns 'After SEBR'). The memory reduction ratio (see columns 'ROM reduction') has been calculated with respect to the traditional approach.

The test results show that it is not necessary to have many extra resources to obtain a large memory reduction (about 83% of the test cases consume only one CLB). The best reduction ratio has been obtained for the scf example. In the traditional implementation, there are not enough memory resources in any FPGA devices. However, after reduction, the amount of memory usage is suitable for real implementation in modern FPGAs. For example, in a Xilinx Virtex-II, 10 SelectRAM blocks plus 3 CLBs or 12 SelectRAM blocks plus 32 tri-state buffers are used. Some tests (keyb, opus, s21 and s27) demonstrate that the proposed approach allows ROM size reduction even when the input multiplexing technique alone is useless.

Conclusion: We have proposed a new approach which exploits the *don't care inputs* to implement ROM-based FSMs using much fewer memory resources than the traditional implementation. The benchmark results shown demonstrate that large FSMs may be implemented in FPGA devices using a reduced number of memory blocks.

R. Senhadji-Navarro, I. García-Vargas, G. Jiménez-Moreno and A. Civit-Ballcels (*Departamento de Arquitectura y Tecnología de Computadores, Escuela Superior de Ingeniería en Informática, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012 Sevilla, Spain*)

References

1 Garcia, E.: 'Xilinx: creating finite state machines', *Xcell J.*, 2000, **38**
2 Rawski, M., Selvaraj, H., and Łuba, T.: 'An application of functional decomposition in ROM-based FSM implementation in FPGA devices'. Proc. Euromicro Symp. on Digital System Design, Belek-Antalya, Turkey, 2003, pp. 104–110
3 Katz, R.H.: 'Contemporary logic design' (The Benjamin/Cummings Publishing Company, Inc., California, 1994)
4 Krueger, R.: 'Xilinx Virtex devices: variable input LUT architecture', 2004, **4**, (1), (The Syndicated)
5 Altera, Corp.: 'Stratix II device handbook', 2004, (Ver. 1.0, Chapter 2)
6 Xilinx, Inc.: 'Virtex-II Platform FPGAs: Detailed description', 2004, Ver. 3.2
7 Xilinx, Inc.: 'HDL synthesis for FPGAs design guide', 1995
8 Kellerer, H.: 'Knapsack problems' (Springer-Verlag, New York, 2004)
9 McElvain, K.: 'IWLS'93 Benchmark Set: Version 4.0', 1993 (http://www.cbl.ncsu.edu/CBL_Docs/lgs93.html)