A Dedicated Reconfigurable Architecture for Finite State Machines

Johann Glaser, Markus Damm, Jan Haase, and Christoph Grimm

Institute of Computer Technology, Vienna University of Technology, Austria
{glaser,damm,haase,grimm}@ict.tuwien.ac.at

Abstract

For ultra-low-power sensor networks, finite state machines are used for simple tasks where the system's microprocessor would be overqualified. This allows the microprocessor to remain in a sleep state, thus saving energy. This paper presents a new architecture that is specifically optimized for implementing reconfigurable Finite State Machines: Transition-based Reconfigurable FSM (TR-FSM). The proposed architecture combines low use of resources with a (nearly) FPGA-like reconfigurability. [1]

1 Introduction

Wireless sensor networks are applied in numerous fields, including building automation, automotive systems, container tracking, and geological surveillance. To ensure autonomous operation over a long period of time (up to several years), the power consumption of such a node must be very low (some μW). A possible strategy to reduce power consumption is to keep the microprocessor (CPU) in an inactive low power mode as long as possible, and to use a separate, hard-wired Finite State Machines (FSMs) for simple, periodic tasks such as:

- Power management by switching on (or off) peripheral units like sensors, A/D converters and the microprocessor itself.
- Controlling the periodic measurement of sensor values, A/D conversion and decision upon further processing.
- Handling communication at physical and even MAC layer in wake-up receivers.

Although a hard-wired FSM permits significant reduction of power consumption, its application is restricted by the fact that it is not programmable as the firmware is. This restricts the applicability of FSMs to tasks that don't have to be changed later on. The exertion of programmable logic such as an embedded FPGA would permit a tradeoff, but unfortunately introduces a large area and power overhead [1].

In this paper, a new architecture for implementing re-configurable FSMs is presented: Transition-based Reconfigurable FSM (TR-FSM). TR-FSMs have a low logic,

[1] This work is conducted as part of the Sensor Network Optimization through Power Simulation (SNOPS) project which is funded by the Austrian government via FIT-IT (grant number 815069/13511) within the European ITEA2 project GEODES (grant number 07013).

wiring and configuration effort and are thereby applicable in ultra-low power applications while offering re-configurability.

The rest of the paper is organized as follows: The following section surveys various hardware implementations of FSMs. Then the new approach is introduced and implementation details are described. This is followed by an analysis of the architecture and its advantages compared to FPGAs, and a conclusion.

2 Related Work

For the implementation of an FSM in an ASIC, synthesis tools use logic minimization algorithms to find optimal combinational circuits which realize the state transfer function and the output function. An alternative implementation is to read from a ROM to retrieve the results of the two functions. Both, the input signals and the state signals are used as address inputs. The data output of the ROM is split into the FSM output signals and the next state signals, where the latter are fed back to the inputs through a clocked register [2].

A RAM in read mode is a simple approach to realize a *reconfigurable* FSM. By writing the RAM content to a specific set of data the FSM functions are specified. The disadvantage of the memory approach is the required size. For n_I input signals, n_S state bits and n_O output signals, the memory has to offer $2^{n_I+n_S}$ words with a width of $n_S + n_O$ bits.

The first reconfigurable logic structures were implemented as sum-of-product term structures in PALs and PLAs [3]. Another widely used circuit design for reconfigurable logic are FPGAs. These comprise look-up tables, flip-flops and rich routing resources to universally implement any logic function [3].

The synthesis results for FSMs in an FPGA are suboptimal, according to [4]. Therefore, new synthesis methods using multi-level architectural decomposition and utilization of Block RAMs to reduce the amount of logic resources are introduced. However, this approach still relies on FPGAs and their disadvantages for low-power applications as outlined in [1].

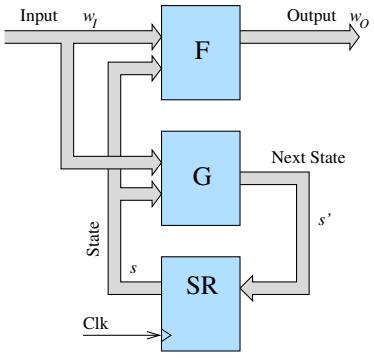

Fig. 1. Generic FSM (SR: state register).

In [5], an unbalanced and unsymmetrical architecture for reconfigurable FSMs is introduced. The FSM is split into a sequential section which implements the state transition function and a logic section which implements the output signals, both connected via routing resources. Although this approach achieves an area reduction of 43 % and a power consumption decrease of 82 %, it has overhead due to its concentration on the logic functions for the next state and output signals.

While the previously mentioned approaches implement the full FSM at once, [6] only implements the logic required to calculate the next state (and output signals) for the current state. After every state transition, the internal logic is reconfigured to realize the next state logic for the current state. While this approach greatly reduces the total amount of logic elements by increasing their utilization, the permanent reconfiguration effort is not a low power approach.

3 Transition-Based Reconfigurable FSM

We consider FSMs with n_S state bits, n_I input signals, and n_O output signals, with the respective signals being Boolean. For $w_I \in \{0,1\}^{n_I}$, $s, s' \in \{0,1\}^{n_S}$ and $w_O \in \{0,1\}^{n_O}$, the state transition function and the output function is defined by $G(s, w_I) = s'$ and $F(s, w_I) = w_O$, respectively (see Figure 1). That is, we generally assume Mealy automata, and denote a transition by $s \xrightarrow{w_I/w_O} s'$.

The number of possible transitions per state equals 2^{n_I}. Since there are 2^{n_S} possible states, we get an upper bound of $n_T \leq 2^{n_S+n_I}$ for the total number of transitions for such an FSM. However, FSMs in concrete applications have a number of transitions which is considerably lower than this bound. This is especially true for FSMs with many inputs signals, where certain states are only inspecting a subset of these signals. An example is shown in Figure 2, which shows the state transition graph (with outputs omitted) of the FSM "opus" from the LGSynth93 benchmark [7]. Only the state "IOwait" inspects all 5 input signals, while all other states consider at most 2. As a result, the number of transitions in the example is considerably lower (30 transitions)

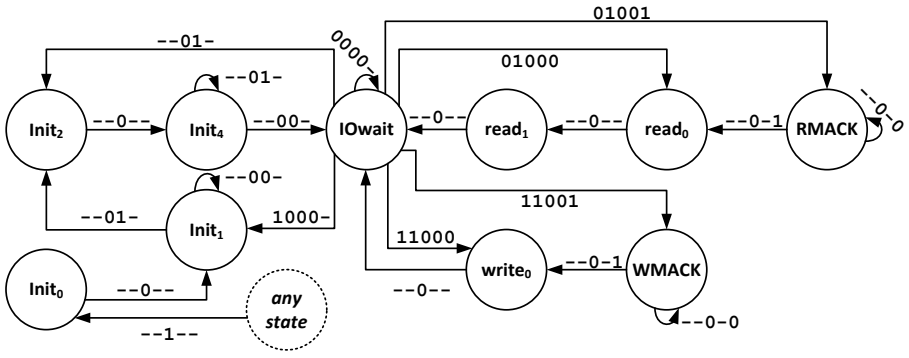

Fig. 2. Example finite state machine.

than the possible number regarding the 10 states (320 transitions) or regarding the 4 state bits necessary for state encoding ($2^{n_S+n_I} = 2^{5+4} = 512$ transitions).

For FSMs with such low transition-per-state-ratios, we propose an approach which focuses on the *transitions* rather than on the state transition function G, and call this *Transition-based Reconfigurable FSM (TR-FSM)*. Instead of providing a big reconfigurable block for implementing the whole state transition function, we provide several smaller reconfigurable blocks for implementing each transition.

Applying TR-FSM for ASIC or SoC-design is a two-stage process. In the first stage, the necessary resources for the TR-FSM are specified. This can either be an estimation based on FSM prototypes for a certain application class, or it is based on a collection of FSMs which the resulting TR-FSM must be able to be configured to. This can be compared to choosing a sufficiently large FPGA (in terms of slices and BRAM) for the FSMs. However, in TR-FSMs there are more size parameters to be considered, but these parameters depend *directly* on the FSMs in question and not on the state encoding or the quality of logic minimization and synthesis algorithms.

From this specification, the semiconductor chip containing the TR-FSM is produced. In the second stage, this TR-FSM embedded in the chip now can be configured with an actual FSM analogous to configuring an FPGA. However, since the TR-FSM is already structured like an FSM, the synthesis process is considerably less complex.

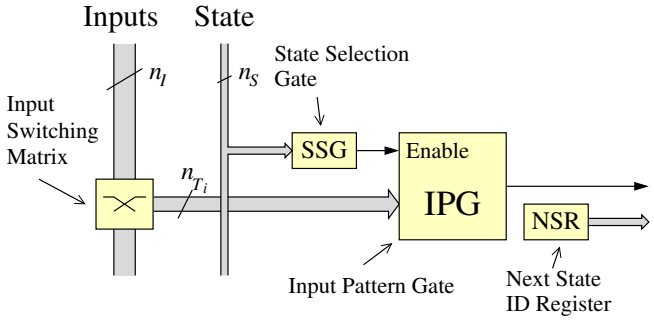

Fig. 3. A transition row.

3.1 Architecture

Note that for the following considerations we omit the FSM outputs. That is, for our purposes a transition is a starting state s, a target state s' and the collection of all input patterns which cause the FSM to change from state s to s', even if the associated outputs are different for each input pattern. The output function F is computed independently, e.g. with a dedicated reconfigurable block. In Section 3.2, we extend the TR-FSM to an architecture which also computes the output in line.

The basis of the proposed architecture is the so called *transition row* (see Figure 3). A state selection gate (SSG) compares the current state to its configured value and enables the transition row. A reconfigurable input witching matrix (ISM) selects a subset

of n_T out of the n_I input signals which then serve as the inputs for the input pattern gate (IPG), which is a reconfigurable combinational logic block, e.g. a look-up table (LUT). If activated, the IPG outputs a "1" iff the input pattern matches a certain transition from the recognized state. We also refer to n_T as the *size* of the transition row. Consider for example an FSM which changes from state s to s' on the input patterns `10---` and `0-1--`.

The ISM of the transition row for this transition would select the leftmost 3 signals out of the 5 input signals, and the SSG would check for the encoding of the state s. If s is the current state, the IPG would output a "1" on the input patterns `10-` and `0-1`, indicating that the transition $s \longrightarrow s'$ is active, and would output "0" otherwise.

The reconfigurable *next state ID register* (NSR) of the corresponding transition row is then selected by a multiplexer ("Select") and fed back to the *state register* (SR) as the new state of the FSM.

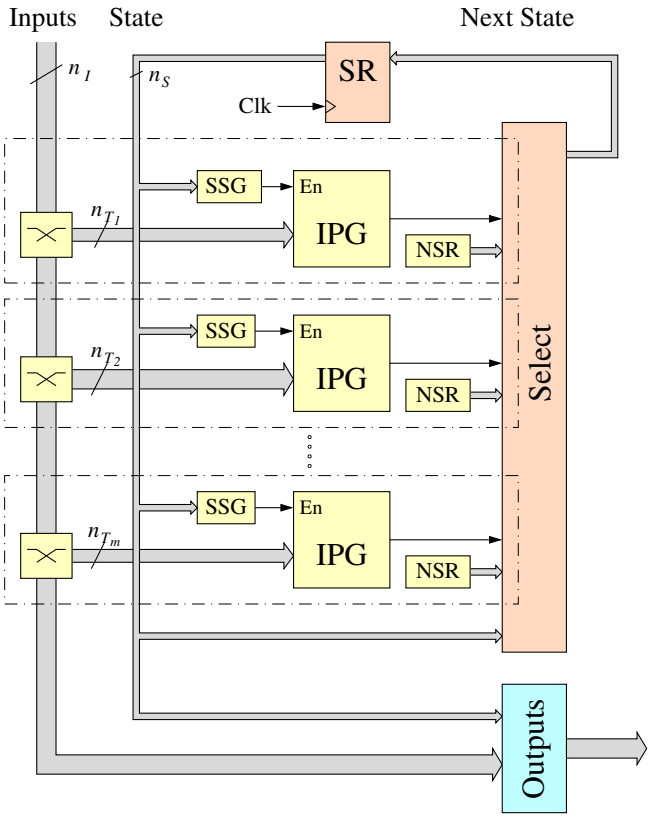

Fig. 4. The overall TR-FSM architecture

The overall architecture contains many such transition rows with varying input width (see Figure 4). The number of transition rows and their input widths are derived during

the specification stage mentioned above. The output signals are computed in a separate configurable logic block ("Outputs" in Figure 4), which for example can be a LUT or an embedded FPGA IP. Note that the state encoding can be chosen freely with respect to the transition logic, since both, the SSG and the NSR are fully configurable. That is, a different state encoding won't yield any benefits like reducing the number of transition rows needed.

As depicted in Figure 4, the current state is also fed into the next state selection logic block ("Select"). This allows for a very simple treatment of loops, i.e. transitions where the FSM remains in the same state. In the case that none of the transition rows are activated, the select logic chooses the current state as new state. Therefore, no transition rows have to be used for loops.

In some cases, one of the input signals of an FSM serves as a reset signal (e.g. the third input signal of the "opus" example in Figure 2). If it is set, the FSM resets to its starting state, which we can assume to be encoded by the `0`-vector. For such cases, we propose a special transition row whose ISM selects this signal and directly feeds it to the state register reset input. Since this reset overrides all transition rows, we can exclude the reset signal from all other transition rows.

To summarize, the reconfigurable parts of the architecture are the state selection gate (SSG), the input switching matrix (ISM), the input pattern gate (IPG), and the next state ID register (NSR).

3.2 Including Output Computation

Instead of computing the output function F with a separate reconfigurable combinational logic block, the output associated to a transition can also be included in the transition row as output pattern register (OPR), similar to the NSR (see Figure 5).

The disadvantage is that more transition rows have to be included for this approach. Consider for example two transitions $s \xrightarrow{\texttt{10/1}} s'$ and $s \xrightarrow{\texttt{11/0}} s'$ in an FSM. With the architecture presented in Section 3.1, these two transitions could be covered by one transition row, since the starting state and the target state are identical. Also, the transition row would only need to inspect one input bit by checking for the input pattern `1-`.

If the outputs are considered, two transition rows have to be used here, and both transition rows have to inspect both input signals. Also, loops can't be treated as a default

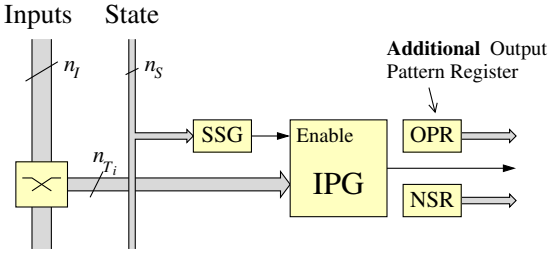

Fig. 5. A transition row with output pattern register (OPR).

case anymore, since each loop will in general produce a different output. Therefore, each loop now needs to be implemented with a transition row, too.

The possible output functions F computable with this TR-FSM variant are restricted. However, if an FSM rarely outputs different values when changing between two specific states, this approach saves the overly generic reconfigurable block computing F. The overall TR-FSM architecture in this case would look very similar to Figure 4, except that the current state is not fed to the next state selection block. Additionally a second selection block for the output signal replaces reconfigurable combinational logic block for F.

3.3 Implementation Details

State Selection Gate (SSG). Since the SSG has to recognize one specific state, it can be implemented with an n_S bit wide AND gate preceded by configurable input inverters. However, similar to the reset signal discussed in Section 3.1, an FSM might have a common error state which is reached from several (but not all) states if an error input is active. For such multi-source transitions, an SSG implemented as an n_S input LUT instead of an AND gate can be used. Now, the respective transition row can be enabled for multiple states.

Input Pattern Gate (IPG). The IPG can be implemented in different ways. The first implementation is similar to that of the SSG, i.e., a n_T wide AND gate with configurable input inverters. This gate can recognize only one specific input pattern.

The second implementation is to use a LUT, such that it is possible to activate the transition for multiple different input patterns. This is beneficial either if the transitions from one state have mixed don't care conditions or if an IPG with more inputs than necessary is used (e.g., because all rows with less inputs are used by other transitions).

Input Switch Matrix (ISM) Consider an ISM with n_I input signals of which $n_T < n_i$ are connected to the IPG, requiring n_I switches per connected signal. Since only one (or none) of the n_I input signals is selected, there are $n_I + 1$ possible switch combinations, which can be encoded as binary numbers with $\lceil \log_2(n_I + 1) \rceil$ bits. Altogether, this yields $n_T \cdot \lceil \log_2(n_I + 1) \rceil$ configuration bits per transition row. If we exclude the possibility to select no signal (e.g. if the IPG is realized as a LUT such that it can implement don't cares), this number reduces to $n_T \cdot \lceil \log_2(n_I) \rceil$.

Another variant to encode the configuration of an ISM exploits that the order of the input signals for the IPG is not important. Therefore, a bit vector of size n_I of which a maximum of n_T bits are set to "1" is sufficient. Every bit represents a primary input which is connected to the IPG if its respective bit is "1". However, this encoding is only more concise than the previous one if $n_I + 1 < n_T \cdot \lceil \log_2(n_I + 1) \rceil$. That is, if a TR-FSM on average has transition rows of small sizes, the previous encoding is more beneficial.

The number $n_I \cdot n_T$ of switches per ISM can also be optimized due to the insignificance of the order of the selected inputs (see Figure 6): If the leftmost input signal is selected, it can be selected as first IPG input signal, and since it can be selected only

once, no further switches are required for the leftmost input signal. The same holds for the second input signal on the left and the second IPG input signal, and so on. Also, the rightmost input signal can always be handled by the last IPG input signal, therefore the respective switch can be removed from the previous IPG input signals, and so on. Altogether, from every selection signal, $n_T - 1$ switches can be removed, such that the total number of switches is $[n_I - (n_T - 1)]n_T$, which gives a reduction of $n_T(n_T - 1)$ switches.

Next State Selection. Every transition row holds the configurable next state ID register (NSR). A multiplexer selects the next state ID associated with the transition row which outputs a logical "1". The configuration applied by the bit stream must ensure, that for every combination of inputs and states at most one of the IPG emits a logic "1" and all others emit a logic "0". The case that all outputs are "0" is legal because in this case the current state is maintained (note the connection of the current state to the "Select" box in Figure 4).

We expect the total number of transitions to be rather high (several tens to several hundreds, see Table 1). For such huge multiplexers the heterogeneous tree approach [8] was proposed. An implementation with pass transistors or transmission gates provides a purely combinational circuit [9].

4 Analysis

We analyzed the proposed approach using several benchmark FSMs taken from the LGSynth93 Benchmark [7]. We chose FSMs which need at least 4 state bits and had relatively few transitions per state. These FSMs were grouped into 3 sets with roughly similar values regarding state vector width, input signals, and output signals. For each group, a TR-FSM was specified by the following algorithm:

Let $\Gamma = \{A_1, \ldots, A_k\}$ be the FSM-group, $n_I(A_i)$ the number of input signals of an FSM A_i, and $N = \max\{n_I(A_1), \ldots, n_I(A_k)\}$. Denote by $m_{i,j}$ the number of transitions in A_i which need to inspect exactly j input signals (that is, they can be implemented with a transition row of size at least j). This results in a $k \times N$ matrix $(m_{i,j})$, where each row effectively specifies an optimal TR-FSM for the respective FSM.

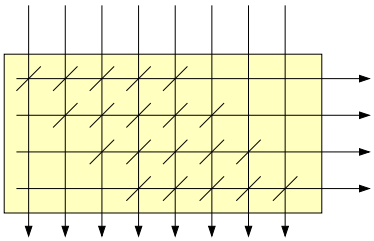

Fig. 6. Optimizing the Input Switch Matrix.

The TR-FSM for the group Γ is now specified by a vector $(M_N, \ldots, M_0)$, where M_j is the number of transition rows of size j needed. This vector must satisfy the equations

$$M_j = \max_{i=1,\ldots,k} \left\{ m_{i,j} - \sum_{j<l\leq N} M_l - m_{i,l} \right\}$$

for each entry. That is, $M_N = \max_{i=1,\ldots,k} \{m_{i,N}\}$, and the rest of the M_j can be computed iteratively. This formula takes into account that a transition row of size n_T can also implement a transition which needs to inspect less than n_T input signals.

Table 1. Example implementations without outputs

Name	#states	#state-bits	#inputs	#outputs	#transitions	#non-loop transitions	input signals to inspect per bitwidth									configuration bits	utilization (in %)
							8	7	6	5	4	3	2	1	0		
planet	48	6	7	19	71	70					6	7	9	17	31		32
s208	18	5	11	2	35	34							31	2	1		20
s420	18	5	19	2	35	34							31	2	1		20
s510	47	6	19	7	75	52							10	18	24	↓	20
s820	25	5	18	19	107	85	2	3	1	5	9	19	25	21			82
sand	32	5	11	9	90	60			3	14	11	8	4	19	1		68
scf	121	7	27	56	152	152	2				16	2		30	102		100
TR-FSM	n.a.	**7**	**27**	**56**	n.a.	**152**	**2**	**3**	**1**	**11**	**11**	**11**	**25**	**21**	**67**	**5087**	n.a.
ex1	20	5	9	19	73	57					11	12	31	1	2		68
ex4	14	4	6	9	18	16						2	4	10			9
mark1	15	4	5	16	22	22						6	1	2	13	↓	12
s1488	49	6	8	19	118	116				6	7	9	17	76	1		100
styr	30	5	9	10	92	73			3	24	17	10	3	7	9		81
TR-FSM	n.a.	**6**	**9**	**19**	n.a.	**116**			**3**	**24**	**17**	**10**	**3**	**58**	**1**	**4037**	n.a.
bbsse	16	4	7	7	42	35					7	5	19	4			25
cse	16	4	7	7	55	39				14	11	10	3	1			80
keyb	19	5	7	2	46	45		8	8	8	2	6	7	4	2		85
opus	10	4	5	6	22	16					4	1		5	6	↓	11
pma	25	5	8	8	38	38				8	18	7	2	3			82
s1	20	5	8	6	80	68		2	1	4	18	16	19	6	2		100
s386	14	4	7	7	40	32					7	6	18	1			27
TR-FSM	n.a.	**5**	**8**	**8**	n.a.	**68**		**8**	**8**	**8**	**2**	**15**	**19**	**6**	**2**	**3439**	n.a.

This algorithm was applied to the three groups to specify TR-FSMs with a stand-alone output logic (Table 1) as well as TR-FSMs which include output signal generation as described in Section 3.2 (Table 2). In the second case, the FSMs where taken "as is" without any preprocessing. In the first case, a simple optimization algorithm was applied to the FSMs in the groups which combined transitions with equal start- and target-state and different outputs to one transition, which yields less transitions rows. Also, the transition rows tend to be smaller, since this optimization process produces more don't cares. Also, loops didn't contribute to the entries of the matrix $(m_{i,j})$, since they are treated as default transitions and don't need to be implemented with a transition row. In both cases, if an FSM possessed a reset input signal, this signal was not considered and a special transition row as outlined in Section 3.1 was added to the TR-FSM.

Table 2. Example implementations with outputs

Name	#states	#state-bits	#inputs	#outputs	#transitions	input signals to inspect per bitwidth									configuration bits	utilization (in %)
						8	7	6	5	4	3	2	1	0		
planet	48	6	7	19	111					14	11	34	33	19		60
s208	18	5	11	2	70					64	6					58
s420	18	5	19	2	70					64	6					58
s510	47	6	19	7	75							10	41	24	↓	39
s820	25	5	18	19	149	6	5	1	6	45	46	37	3			98
sand	32	5	11	9	102			5	17	13	15	8	44			71
scf	121	7	27	56	152	2				16	2		30	102		100
TR-FSM	n.a.	**7**	**27**	**56**	**152**	**6**	**5**	**1**	**10**	**42**	**45**	**37**	**3**	**3**	**17058**	n.a.
ex1	20	5	9	19	136			24	20	34	33	20	5			72
ex4	14	4	6	9	20						2	10	8			7
mark1	15	4	5	16	22						6	1	2	13	↓	7
s1488	49	6	8	19	225			24	23	13	62	63	39	1		100
styr	30	5	9	10	101			6	41	26	8	4	7	9		58
TR-FSM	n.a.	**6**	**9**	**19**	**225**			**24**	**23**	**31**	**44**	**63**	**39**	**1**	**13150**	n.a.
bbsse	16	4	7	7	44					8	13	18	5			33
cse	16	4	7	7	58				19	18	15	5	1			88
keyb	19	5	7	2	46		8	8	8	3	6	7	4	2		74
opus	10	4	5	6	22				4	2		9	7		↓	18
pma	25	5	8	8	40				8	18	7	3	4			46
s1	20	5	8	6	80		2	2	4	22	18	22	8	2		100
s386	14	4	7	7	44					11	14	18	1			34
TR-FSM	n.a.	**5**	**8**	**8**	**80**		**8**	**8**	**8**	**13**	**15**	**18**	**8**	**2**	**4507**	n.a.

The number of configuration bits needed for each TR-FSM was assessed as follows: Each transition row needs n_S configuration bits for the SSG and the NSR, respectively. Each transition row T of size n_T needs $\lceil \log_2 n_I \rceil \cdot n_T$ configuration bits for the ISM (binary encoding) and 2^{n_T} configuration bits for the IPG (implemented as LUT). If the output is computed in the transition rows, we have to add n_O bits for each transition row as well. Summarizing, if $T(A)$ denotes the transition rows of an RT-FSM A, the number of configuration bits is

$$T(A) \cdot 2 \cdot n_S + \sum_{T \in T(A)} (\lceil \log_2 n_I \rceil \cdot n_T + 2^{n_T})$$

if no outputs are considered. The leftmost summand changes to $T(A)(n_O + 2 \cdot n_S)$ if the output is also computed in the transition rows. Each of the TR-FSMs specified need a reset transition row, which requires $\lceil \log_2 n_I \rceil$ configuration bits for the ISM, and one bit to configure possible inversion of the reset bit.

Table 1 shows the result for TR-FSMs without output computation. Here, the column "#non-loop transitions" gives the number of transitions which actually need transition rows for implementation. Table 2 treats the case where the outputs are computed within the transition rows. In both Tables, below of each of the three groups, the data for the respective groups' RT-FSM is given. Note that only relatively few of the input signals have to be inspected, at most 8 in any case.

Each of the FSMs in each group was then synthesized to the respective group TR-FSM, which is a simple mapping process of transitions to transition rows with enough

resources. The column to the right shows the resource utilization by comparing the number of configuration bits needed for the used transition rows to the total number of configuration bits of the respective group TR-FSM.

Since no silicon implementation of the TR-FSM approach has been done yet, we don't compare to an FPGA implementation regarding chip-space and power consumption. Therefore we take the size of the configuration bit stream of both approaches as a cost indicator. While a LUT in an FPGA only needs $2^4 = 16$ configuration bits, there are considerably more configuration bits needed to configure the whole slice and especially the routing between the CLBs. According to [10] the configuration bit stream of the Virtex family dedicates 864 bits per CLB and therefore 432 bits per slice.

For every FSM in our analysis we took the lower number of slices occupied by the VHDL or Verilog implementation according to Table 5.7 (p. 83) of [4]. Within each group, the FSM with the most used slices was picked (see Table 3). With this amount of slices any other FSM in the particular group can be implemented. The length of the configuration bit stream was calculated for the maximum FSM, and then compared to length of the configuration bit stream of the respective group's TR-FSM. The last column of Table 3 shows the size of the TR-FSM configuration bit stream compared to the FPGA configuration bit stream in percent.

Table 3. Configuration bits needed for TR-FSM and FPGA

Group	Conf.Bits TR-FSM	max. FSM [4]	Slices [4]	Conf.Bits FPGA	Comparison
1	17,058	scf	168	145,152	11.8 %
2	13,150	styr	119	102,816	12.7 %
3	4,507	pma	71	61,344	7.3 %

This shows that TR-FSMs need considerably less configuration bits than FPGAs (7.3 to 12.7 %). While we see this as an indicator for a prospective improvement regarding area and power consumption, this is also beneficial by itself regarding run-time reconfigurability and for applications with limited memory resources like wireless sensor nodes. Note that we don't compare directly to the improvements of [4] since this approach involves BRAMs.

FPGAs achieve the high flexibility with an high amount of rich and flexible routing resources. On the other hand the involved short and long wires impose a high capacitive load and include numerous pass transistors along the path. Both result in increased power consumption and area overhead. TR-FSM, on the other hand, have a very low routing overhead, since the general FSM structure is already implemented. Also, the unused transition rows of an TR-FSM can be switched off by voltage gating to save power.

Other advantages are the constant latencies of the TR-FSM, since the routing is fixed, and the fact that the synthesis of an FSM to a specific TR-FSM is very easy, since arbitrary state encodings can be used and no sophisticated minimization problems are involved.

5 Conclusion and Future Work

This paper introduces a novel, fixed latency architecture for reconfigurable finite state machines, which is based on the state transitions instead of the state transfer function (transition-based reconfigurable FSMs, TR-FSM), which is beneficial if the FSMs in question have relatively few transitions.

Synthesizing an FSM onto a given TR-FSM is straightforward, and only few configuration bits are needed. Compared to an implementation of FSMs in an (embedded) FPGA, the approach also promises several advantages regarding power consumption and area. However, this has to be quantified yet by concrete implementations.

Further extensions of our approach range from multi-bit LUTs for the IPG, such that a single transition row can handle transitions to multiple different states and/or with varying output patterns, with only a little overhead in the LUT. For linear sequences simplified transition rows with zero or one input signal, or specialized time-out transition rows with an integrated counter are considered.

Transitions which have to consider a high number of input signals should be mapped to special transition rows with IPGs implemented as sum-of-product logic. Another option is to combine two transition rows and place a MUX to select between the outputs of the two IPGs. Its select input is then driven by another FSM input, so increasing the total input signal sensitivity list by one. Finally, by enhancing the NSR to latch the current state, sub-state-machines which are able to return to the caller may be implemented.

References

1. Hartenstein, R.: Coarse Grain Reconfigurable Architecture. In: Proceedings of the 2001 Conference on Asia South Pacific Design Automation, Yokohama, Japan (2001) 564–570
2. Katz, R.H.: Contemporary Logic Design. The Benjamin/Cummings Publishing Company, Inc. (1994)
3. Rabaey, J.M., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits. Prentice Hall, Upper Saddle River (2003)
4. Bukowiec, A.: Synthesis of Finite State Machines for Programmable Devices Based on Multi-Level Implementation. PhD thesis, University of Zielona Gora, Poland (2008) http://zbc.uz.zgora.pl/Content/14528/PhD-ABukowiec.pdf (retrieved 2009-11-03).
5. Liu, Z., Arslan, T., Khawam, S., Lindsay, I.: A High Performance Synthesisable Unsymmetricaı Reconfigurable Fabric For Heterogeneous Finite State Machines. In: Proceedings of the ASP-DAC. Volume 1. (18.-21. January 2005) 639–644
6. Milligan, G., Vanderbauwhede, W.: Implementation of Finite State Machines on a Reconfigurable Device. In: Proceedings of the second NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh (5-8 August 2007) 386–396
7. McElvain, K.: LGSynth93 Benchmark Set: Version 4.0. online: http://www.cbl.ncsu.edu/-pub/Benchmark_dirs/LGSynth93/ (1993)
8. Lin, M.B.: On the design of fast large fan-in CMOS multiplexers. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19**(8) (August 2000) 963–967
9. Alioto, M., Palumbo, G.: Interconnect-Aware Design of Fast Large Fan-In CMOS Multiplexers. IEEE Transactions on Circuits and Systems II: Express Briefs **54**(6) (June 2007) 484–488
10. Xilinx, Inc.: Virtex Series Configuration Architecture User Guide (XAPP151). v1.7 edn. (20 October 2004)