Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 11 (2013) 363 – 366

www.elsevier.com/locate/procedia

2nd International Through-life Engineering Services Conference

Creating a Self-Configuring Finite State Machine out of Memory Look-Up Tables

Philipp Schiefer[a]*, Richard McWilliam, Alan Purvis

[a]*Durham University, South Road, Durham, DH1 3LE, United Kingdom*

* Corresponding Philipp Schiefer. Tel.: +44 191 334 2418; fax: +44 191 334 2408. *E-mail address:* philipp.schiefer@durham.ac.uk

Abstract

A finite state machine (FSM) is one of the most used digital logic applications in today's electrical systems. An FSM can be implemented in electrical systems based on programmable logic devices (PLD) or combinatorial logic platforms. Both platforms for a FSM contain advantages and restrictions for the hardware and software design. In regards of coding, FSM can be coded in alternatives styles and programming languages. In this paper we introduce the concept of a self-configuring FSM based on coding data as memory look-up tables. The resulting FSM is then able to self-configure the combinatorial logic of this FSM required to perform the compulsory state sequence. The primary benefit of using memory based look-up table (LUT) FSM is that well established data error correction methods can be applied to protect the FSM behavior, even in the event of single error events (SEE). A high level hardware design of this FSM will be presented in comparison to a PLD FSM implementation.

Finite State machine; Memory Look-Up Tables

1. Introduction

In today's digital environment finite state machines can be found in almost all digital systems due to the fixed way of working at only one single active state or instruction at a time. The transition from one state to another state of a FSM is controlled by input stimulus and stored information. The basic building blocks of an FSM contain decision making logic and memory [1]. This concept of only having one active state at a time makes is possible to use FSMs to control a width spectrum of every day application such as vending machines, turn crosses, voice systems or safety critical systems in automobiles [2]. In this paper a soda vending machine behavior is going to be used as an example FSM. The implementation of an FSM can be done within a programmable logic device (PLD) or combinatorial logic platforms. The design implementation depends on the actual designer of the system. If an FSM is going to be programmed with any type of description language two solutions in the same language will not look the same. The coding of this system will follow one of the three commonly used methods: combined single process (CSP), state separated combinatorial outputs (SCO), and state separated registered outputs (SRO) [3]. PLD based implementation can be done with any microcontroller system platform or in an FPGA based system. FPGA based platforms have the advantage of having a combination of combinatorial logic and memory facility within the chip [4-5]. In this paper the focus of FSM implementation will be done on a PLD and combinatorial logic based platform. FPGAs can be provided with ready-to-use library implementations of FSMs that are part of the development software tools [3-4]. An FPGA based solution offers limited possibilities to improve or alter the given design and this is the reason why it will not part of this investigation. In this paper a solution for showing an approach of creating a unique hardware design with minimal controlling logic and

doi:10.1016/j.procir.2013.07.030

memory LUT will be presented. Comparison between the memory usage and execution times are investigated and the execution time will be used to evaluate the system performance.

2. Different coding styles for FSMs

For coding an FSM with the help of a high level programming language there are three methods commonly used to do so: CSP, SCO, and SRO. Variation of this three coding methods like One-Hot or Gray encoding are in use but associated with specific hardware requirements for FPGAs HDL implementation [3]. The CSP coding method uses a single process for controlling both the state transitions and the output of the FSM. These outputs of the FSM are stored in a register. Due to this storing of the output, a FSM structure cannot be easily synthesized by software. A block diagram of a CSP is illustrated in Figure 1. The SCO coding uses two processes instead of only one used in the CSP style and the outputs are generated directly out of combinatorial logic. No output register gets used in the coding method. A block diagram can be seen in Figure 2. The SRO coding method uses a two process method and uses an output register. The output of the output signal will be done one stage later than it had been inferred out of the output combinatorial logic. A block diagram of this coding method can be seen in Figure 3.

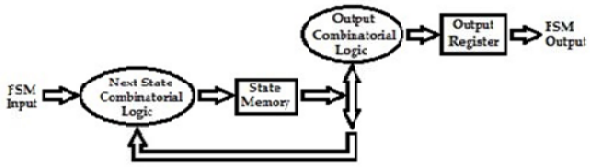

Fig. 1 Coding style combined single process (CSP)[3]

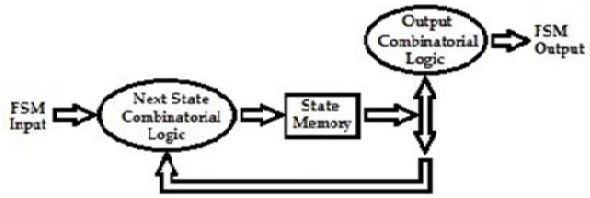

Fig. 2 Coding style state separated combinatorial outputs (SCO)[3]

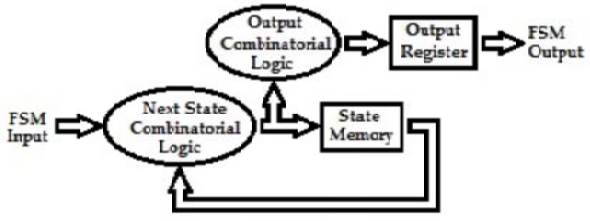

Fig. 3 Coding style state separated registered outputs (SRO)[3]

The goal of this research is to create a FSM design which uses as little as possible controlling combinatorial logic and memory LUTs. It became apparent that the above three coding styles do not generate minimal combinatorial logic overheads. Reduction in this area of logic can be gained with the coding the data stored in the state memory. By including a control bit within the memory data the input and output combinatorial logic can be altered at run-time and logic overhead can be reduced. With this memory data structure the coding style for the FSM can be altered to the block diagram structure shown in Figure 4.

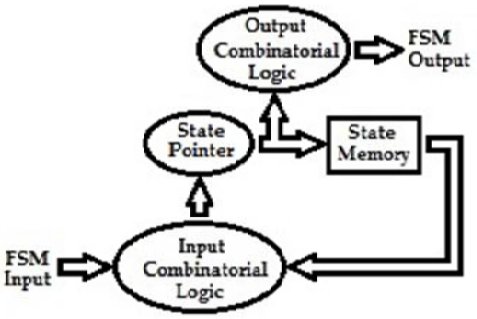

Fig. 4 Memory based LUT coding style

The block diagram shown in Figure 4 discloses another difference between the three other coding styles. Which is a state memory pointer used to access the required state transition stored within the memory LUT. The information of this state pointer can be altered by the FSM inputs and state transitions and this is how the memory stored state transition gets controlled if required.

3. Vending machine implementation

With the example of a soda vending machine FSM implementation the proposed coding style proposed in Figure 4 will be implemented to show the feasibility and the minimum controlling combinatorial logic design. The basic state diagram of the soda vending machine can be seen in Figure 5.

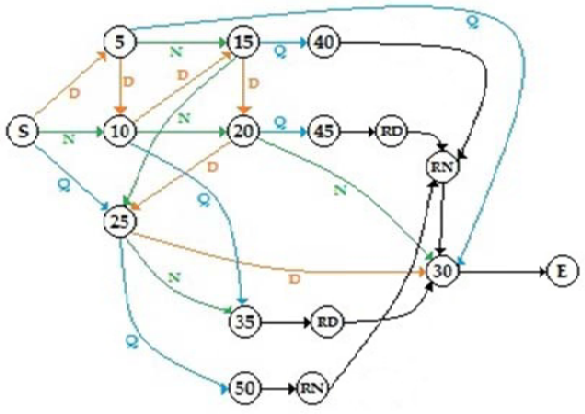

Fig. 5 Basic state diagram of soda vending machine

The function of the machine is a simple task-:. after the correct amount of 30 cents has been tossed into the machine a can of soda is getting ejected. The following denominations in coins are accepted by the machine 5 (D), 10 (N), and 25 (Q) cent coins. The machine accepts any arrangement of these three coins denomination and even handles overpaying. In this case the machine has to perform two tasks: supplying a can of soda and paying out the overpaid amount.

The state diagram for the soda vending machine presented in Figure 5 shows 17 inputs, 10 outputs, and 2 fixed dependent transition states. These different transition states can be codes and mapped into 29 bytes of memory LUTs. The information in this memory is going to be the same to show the overhead required for implementation of this FSM on these two platforms.

3.1. PLD based FSM implementation

For the implementation on a PLD platform an 8051 microcontroller was chosen because it has on-chip input and output capabilities. For coding the FSM the assembler language has been chosen because of the direct link to the hardware and minimal overhead in the programming memory requirements. The coding style was consistent with the coding style presented in Figure 4. From the instruction set of the 8051 microcontroller only the following direct commands were used: MOV, ADD, AND, XOR, relative jump at zero, and jump. The assembler implementation of this FSM uses the following instructions required memory size: 22 bytes for AND, 6 bytes for XOR, 6 bytes for ADD, 27 bytes for relative jumps, and 33 bytes for jumps. The total byte count for the logic memory is 94 bytes. This is a ratio of 3.2 times the memory of the state memory count.

3.2. Combinatorial logic based FSM implementation

The combinatorial logic based FSM has been implemented according to Figure 6. This design uses a single memory address pointer for the next state transition. As a result, the state transition memory requirement is one byte. The combinatorial logic based FSM uses the same memory LUT size and structure as the PLD design. This is a common functional block equal for both designs. The controlling logic of the combinatorial logic FSM design contains an input signal reducer and output signal expander. Additional logic gates are needed to determine the signal adding to the address pointer data and logic for output redirection.

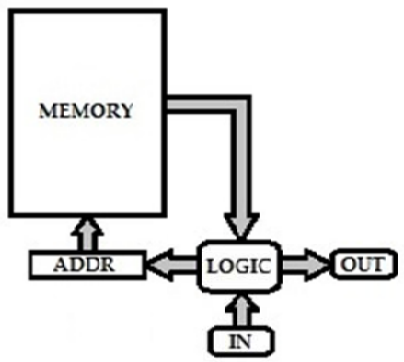

Fig. 6 Basic block diagram of combinatorial logic based FSM design

4. Results

For storing the different state transition information a memory based LUT of 29 bytes has been used for both example implementations. For the PLD the memory requirements of storing the assembler program has been 94 bytes and for the combinatorial logic design only 1 byte. The comparison of the memory ratio is for the PLD design 3.2 and for the combinatorial design 0.03. This indicates that the combinatorial logic design has a better memory ratio. A comparison of these two designs will show the faster system:

Table 1. Comparison of FSM implementation execution time

Coins applied	PLD based design	Combinatorial logic based design
5*5 cent & 25 cent	339 cycles	18 cycles
6*5 cent	305 cycles	14 cycles
3*10 cent	161 cycles	8 cycles
5 cent & 25 cent	113 cycles	6 cycles

The comparison of the performance based on the cycle count is possible due to the fact that a certain programming flow in the assembler code results in a fixed cycle count.

5. Conclusion

The design of the combinatorial logic design using a memory based LUT shows the best results in performance and memory ratio. Another advantage of this design is the reduced use of logic circuits in comparison to a microcontroller. This design solution reduces the logic circuit overhead considerable which has an impact on power consumption and logic element reliability. Due to the simple combinatorial logic block a solution for a fault tolerant design could be part of a future work.

Acknowledgements

This work is supported by the EPSRC Centre in Through-life Engineering Services.

References

[1]. Senhadji-Navarro, R., I. Garcia-Vargas, and J.L. Guisado. Performance evaluation of RAM-based implementation of Finite State Machines in FPGAs. in Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on. 2012.

[2]. Garcia-Vargas, I., et al. ROM-Based Finite State Machine Implementation in Low Cost FPGAs. in Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on. 2007.

[3]. Rafla, N.I. and B.L. Davis. A Study of Finite State Machine Coding Styles for Implementation in FPGAs. in Circuits and Systems, 2006. MWSCAS '06. 49th IEEE International Midwest Symposium on. 2006.

[4]. Tiwari, A. and K.A. Tomko, Saving Power by Mapping Finite-State Machines into Embedded Memory Blocks in FPGAs, in Proceedings of the conference on Design, automation and test in Europe - Volume 2. 2004, IEEE Computer Society. p. 20916.

[5]. Koster, M. and J. Teich. (Self-)reconfigurable finite state machines: theory and implementation. in Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings. 2002.