

# Self-Correction of FPGA-based Control Units

Iouliia Skliarova

University of Aveiro, Department of Electronics and Telecommunications, IEETA  
3810-193 Aveiro, Portugal

[iouliia@det.ua.pt](mailto:iouliia@det.ua.pt), <http://www.ieeta.pt/~iouliia/>

**Abstract.** This paper presents a self-correcting control unit design using Hamming codes for finite state machine (FSM) state encoding. The adopted technique can correct single-bit errors and detect two-bit errors in the FSM register within the same clock cycle. The main contribution is the development of a parameterizable VHDL package and the respective error-correcting modules, which can easily be added to an FSM specification using any state assignment strategy and having any number of inputs, outputs and states. Besides of application to FSM error correction, the developed tools can easily be adapted to other applications where error detection and correction is required.

**Keywords:** self-correcting finite state machines, Hamming codes, specification in VHDL

## 1 Introduction

Concurrent error detection and correction is very important in many high-reliability applications. Nowadays, FPGA are increasingly being used for such applications, working in hazardous operating environments. In such circumstances, radiation or overheating can cause either a temporary or a permanent fault in a system prohibiting that it functions correctly.

A control part of the system is the most critical part since it plays the central role in correct functioning of the whole system. Therefore, providing the control units with the properties of self-checking and self-correction is very important.

A number of synthesis techniques have been proposed aimed at design of control units with concurrent error detection [1-3]. These techniques permit a circuit to be synthesized that is capable of providing the identification of an erroneous behavior as soon as it is observable. In this paper, we propose using error-correcting codes that allow for changing the illegal system state to the correct state.

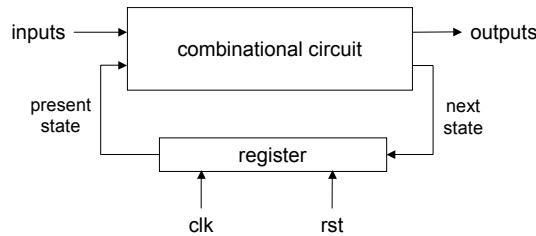
For such purposes, a VHDL package has been developed that includes functions required for generating error-correcting codes and a number of VHDL modules have been designed that make use of these functions and can be employed for constructing a self-correcting control circuit. Besides of specifying self-correcting control units, the developed tools can directly be used for providing error correction and detection properties in other application domains, such as communication systems (data networks, memory caches, etc.).

This paper is organized in 5 sections. Section 2 that follows this introduction is devoted to an overview of the adopted error-correcting codes. Section 3 proposes a general structure of self-correcting control unit and includes the detailed description of the developed VHDL functions and modules. The results of experiments are reported in section 4. Finally, concluding remarks are given in section 5.

## 2 Error Correction

The control units are usually modeled with the aid of *Finite State Machines* (FSMs). An FSM can be defined as a 6-tuple  $M = (S, X, Y, \varphi, \psi, s_0)$ , where  $S = \{s_0, \dots, s_{M-1}\}$  is a finite set of states,  $X = \{x_0, \dots, x_{L-1}\}$  is a finite set of inputs,  $Y = \{y_0, \dots, y_{N-1}\}$  is a finite set of outputs,  $\varphi: S \times X \rightarrow S$  is the next state function,  $\psi: S \times X \rightarrow Y$  is the output function, and  $s_0 \in S$  is the initial state.

The hardware model of FSM is shown in fig. 1. The FSM consists of a combinational circuit (that produces the primary outputs and calculates the next state based on the input values and the present state) and a register (a number of flip-flops or latches) that stores the present FSM state. If the FSM register experiences a fault, it could place the FSM in either a legal (but not correct) state or an illegal state. To allow for recovering from such erroneous state transitions, Hamming codes [4] can be used for state encoding. Hamming codes have a minimum distance of 3 (between different code words) and can therefore be employed for correcting any single-bit fault.



**Fig. 1.** Hardware model of FSM

Hamming codes can easily be constructed for any FSM encoding scheme, such as one-hot, Gray, etc. For  $d$  data bits, the Hamming method requires adding  $p$  parity bits, such that  $d \leq 2^p - 1 - p$ , thus yielding  $(d+p)$ -bit codes. The bit positions in a Hamming code can be numbered from 1 through  $d+p$ . In this case, those bit positions whose number is a power of 2 are parity bits, and the remaining positions are data bits.

Fig. 2 illustrates how a 10-bit state code can be augmented with parity bits. Each parity bit is grouped with a subset of those data bits whose numbers have a 1 in the same bit when expressed in binary. For instance, parity bit  $p_0$  with position  $1_{10}$  ( $0001_2$ ) is grouped with data bits with positions 3 ( $0011_2$ ), 5 ( $0101_2$ ), 7 ( $0111_2$ ), 9 ( $1001_2$ ), 11 ( $1011_2$ ), and 13 ( $1101_2$ ) as illustrated by dashed arrows in the upper part of fig. 2. Then, such a value is assigned to each parity bit as to guarantee that the respective group produces even parity (has an even number of 1s).

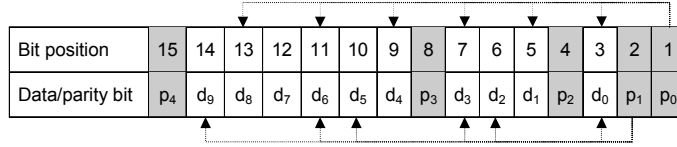


Fig. 2. Hamming code data and parity bits

A distance-3 Hamming code can easily be modified to obtain a distance-4 code by adding one more parity bit (overall parity bit  $p_4$  in fig. 2) chosen so that the parity of all the bits including the new one is even. This new code can detect double errors that are not correctable.

For error correction, all the parity groups are checked. The possible error types and the respective actions to take are summarized in table 1. If one or more parity groups have odd parity and the overall parity bit is 0, then a double-bit error has occurred, which is not correctable. If all the parity groups have even parity then the state code is assumed to be correct. Otherwise, if one or more groups have odd parity and the overall parity bit is 1, then a single-bit error is supposed to have occurred. In this case, a syndrome (the pattern of groups that have odd parity) is created indicating the bit position whose value is assumed to be wrong and consequently has to be complemented. The syndrome can be calculated by XOR-ing the parity bits read out of the FSM register with the new parity bits generated from the data stored in the register. For example, if the FSM register outputs the code `000000000000100`, then the new parity bits  $p_0$  (with position 1) and  $p_1$  (with position 2) will have odd parity corresponding to the position 3 ( $0001_2 \oplus 0010_2 = 0011_2$ ) whose value has to be complemented producing the correct state code `000000000000000`.

Table 1. Error detection and correction

Syndrome	Overall parity bit	Error type and actions to take
= 0	0	no error
= 0	1	overall parity bit error; no problem for correct FSM operation
≠ 0	0	double-bit error; not correctable
≠ 0	1	single-bit error; correctable by calculating the syndrome and inverting the respective bit position

### 3 Self-correcting FSM

#### 3.1 Hardware Model

The hardware model of self-correcting FSM is presented in fig. 3. The *Parity Encoder* block creates the parity bits to store with the FSM state bits in the FSM register. The parity bits can trivially be calculated by XOR-ing the relevant FSM state bits.

The *Code Corrector* block receives the present state from the FSM register and corrects it if required. For such purposes the syndrome is generated by calculating the new parity bits and XOR-ing them with the parity bits read out of the register. Then, a correction mask is produced based on the result of the syndrome.

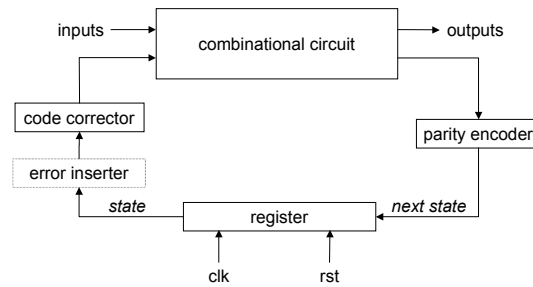


Fig. 3. Hardware model of self-correcting FSM

When either no error is detected or a double error is detected, all the bits of the mask are set to 0. Otherwise the mask is generated so as to conceal all the state bits except for the erroneous bit. Finally, the mask is XOR-ed with the data read out of the FSM register. As a result, when no single-bit errors are detected, the FSM state passes through the *Code Corrector* without any changes, whereas when a single-bit error is detected the corrupted bit is inverted to the correct value. It is important that error correction is performed within the same clock cycle.

To allow for diagnosis, *Error Inserter* block is used to introduce single or multiple bit errors to the FSM state (see fig. 3).

### 3.2 Specification in VHDL

To facilitate the VHDL description of the blocks introduced in section 3.1, a package `parity_types` was developed, which includes the following functions:

- function **calculate\_parity** (constant state : in std\_logic\_vector; constant n : in natural) return std\_logic – this function receives two parameters: an FSM state and a number of a parity group, and calculates the required parity bit value for this parity group;
- function **calculate\_mask** (constant syndrome : in std\_logic\_vector; constant n : in natural) return std\_logic\_vector – this function receives a syndrome and the number of bits used for FSM state encoding and generates the mask used for correcting single-bit errors;

These functions are described in VHDL as indicated in fig. 4 for FSMs using at most 11 bits for state encoding and consequently requiring at most 4 parity bits (not counting the overall parity bit). This restriction is due to the space limitations. In order to support more bits for state encoding only two constant arrays

parity\_table and mask\_table (and the respective data types) have to be modified; the functions themselves do not require any changes to be introduced.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

package parity_types is

    type PAR is array (natural range 0 to 3) of std_logic_vector(10 downto 0);
    type MASK is array (natural range 0 to 15) of std_logic_vector(10 downto 0);

    constant parity_table : PAR := (0 => "10101011011", 1 => "11001101101",
                                     2 => "11110001110", 3 => "11111110000");

    constant mask_table : MASK := ( 0 => "00000000000", 1 => "00000000000",
                                     2 => "00000000000", 3 => "00000000001",
                                     4 => "00000000000", 5 => "00000000010",
                                     6 => "00000000100", 7 => "00000001000",
                                     8 => "00000000000", 9 => "00000010000",
                                     10 => "00000100000", 11 => "00001000000",
                                     12 => "00010000000", 13 => "00100000000",
                                     14 => "01000000000", 15 => "10000000000");

    function calculate_parity (constant state : in std_logic_vector;
                              constant n : in natural) return std_logic;
    function calculate_mask   (constant syndrome : in std_logic_vector;
                              constant n : in natural) return std_logic_vector;
end parity_types;

package body parity_types is

function calculate_parity (constant state : in std_logic_vector;
                          constant n : in natural) return std_logic is
    variable masked : std_logic_vector(state'high downto 0);
    variable result : std_logic;
begin
    masked := state and parity_table(n)(state'high downto 0);
    result := '0';

    for i in masked'range loop
        result := result xor (masked(i));
    end loop;

    return result;
end function calculate_parity;

function calculate_mask (constant syndrome : in std_logic_vector;
                        constant n : in natural) return std_logic_vector is
    variable mask : std_logic_vector(n downto 0);
    variable address : natural range mask_table'range;
begin
    address := conv_integer(syndrome);
    mask := mask_table(address)(n downto 0);
    return mask;
end function calculate_mask;

end parity_types;

```

**Fig. 4.** VHDL package defining functions and data types required for parity calculation and error correction

In the developed package two data types are declared. The first data type `PAR` is an array of four 11-bit standard logic vectors. A constant `parity_table` of type `PAR`, for every parity group (from 0 to 3), marks by 1s the FSM state bits that have to be used for calculating the parity of the group. For example, in order to calculate the parity bit  $p_0$  state bits 0, 1, 3, 4, 6, 8, and 10 have to be analyzed, resulting in a vector 10101011011. The remaining FSM state bits, denoted by 0s in the constant `parity_table`, are not relevant for the considered parity groups. The function `calculate_parity` firstly masks out with the aid of the constant `parity_table` not relevant state bits and stores the result in the variable `masked`. Then, in a loop statement, all the relevant state bits are XOR-ed together calculating in such a way a value to be assigned to the respective parity bit. This value is kept in a variable `result` and is returned from the function. Note that the number of state bits to be analyzed is not fixed and is selected with the attribute `high` applied to the FSM state. Consequently, the function can be used for calculating the parity bit values for FSM states encoded with an arbitrary number of bits.

The second data type `MASK` is an array of sixteen 11-bit standard logic vectors. A constant `mask_table` of type `MASK` stores for each possible syndrome to be calculated in the *Code Corrector* block, the relevant mask that has to be applied to the FSM state in order to correct possible errors. For example, if the generated syndrome is equal to  $0011_2$ , the mask `00000000001` indicates that state bit 0 is in error and has to be complemented. The function `calculate_mask` firstly converts a received syndrome from standard logic vector to a natural value with the aid of function `conv_integer` (defined in package `std_logic_unsigned` of `ieee` library). Then, the calculated value is used for accessing one of the vectors declared in the constant `mask_table`, which is subsequently returned from the function.

With the aid of the developed package, the *Parity Encoder* and the *Code Corrector* block can be described in VHDL as shown in the code below.

The *Parity Encoder* block is parameterizable with the aid of two generic constants ( $n$  and  $m$ ) which indicate respectively the number of bits used for FSM state encoding and the number of the required parity bits. Inside the block, the value to be assigned to each parity bit is calculated with the aid of a generate statement which permits the function `calculate_parity` to be invoked the required number ( $m$ ) of times. The resulting parity bit values are written to the output parity vector.

The *Code Corrector* block is similarly parameterizable with the aid of two generic constants ( $n$  and  $m$ ) which indicate the number of bits used for FSM state encoding and the number of the required parity bits, respectively. Inside the block, first of all the new parity bits are calculated with the aid of a generate statement invoking the `calculate_parity` function  $m$  times. After that a syndrome is generated by XOR-ing the previously calculated parity vector (read out of the FSM register) with the newly calculated parity vector. Based on the syndrome, a mask is produced with the aid of the function `calculate_mask`. Finally, the mask is applied to the FSM state read from the FSM register allowing a possible error to be corrected.

It is very important that the developed functions and modules are parameterizable (with the aid of attributes and generic constants [5]) and can therefore be used for providing error correction ability for any number of data bits (currently at most 120 data bits are supported). Consequently, the proposed modules can directly be

employed for any FSM with any number of states and using any state encoding technique (as far as the number of state bits does not exceed the currently imposed 120-bit limitation) and also for other applications.

```

library IEEE;          use IEEE.STD_LOGIC_1164.ALL;
library corr_codes;   use corr_codes.parity_types.all;

entity encoder is
    generic(n : natural := 11; m : natural := 3);
    port ( state : in std_logic_vector(n downto 0);
          parity : out std_logic_vector(m downto 0));
end encoder;

architecture Behavioral of encoder is
begin
    calc_parity: for i in 0 to m generate
        parity(i) <= calculate_parity(state, i);
    end generate;
end Behavioral;

-----

library IEEE;          use IEEE.STD_LOGIC_1164.ALL;
library corr_codes;   use corr_codes.parity_types.all;

entity corrector is
    generic ( n : natural := 11; m : natural := 3);
    port ( state : in std_logic_vector(n downto 0);
          in_parity : in std_logic_vector(m downto 0);
          corr_state : out std_logic_vector(n downto 0));
end corrector;

architecture Behavioral of corrector is
    signal new_parity : std_logic_vector(m downto 0);
    signal syndrome : std_logic_vector(m downto 0);
    signal mask : std_logic_vector(n downto 0);
begin
    calc_parity: for i in 0 to m generate
        new_parity(i) <= calculate_parity(state, i);
    end generate;

    syndrome <= in_parity xor new_parity;
    mask <= calculate_mask(syndrome, state'high);
    corr_state <= mask xor state;
end Behavioral;

```

The complete self-checking FSM can be constructed from the designed modules as shown in fig. 5 for a simple sequence detector having one input, one output, and 5 states and using Gray state encoding technique. The respective state diagram and state/output table are shown in fig. 6. The developed parameterizable modules and the `parity_types` package were put in a library `corr_codes`.

## 8 Iouliia Skliarova

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity secl_gray is
    Port ( clk, reset : in std_logic;
          X : in std_logic;
          Y : out std_logic);
end secl_gray;

architecture mixed of secl_gray is
    signal state, next_state, corr_state : std_logic_vector(2 downto 0);
    signal parity, next_parity : std_logic_vector(2 downto 0);
begin

    FSM_register: process(clk, reset)
    begin
        if reset = '0' then
            state <= (others => '0'); parity <= (others => '0');
        elsif rising_edge(clk) then
            state <= next_state;
            parity <= next_parity;
        end if;
    end process FSM_register;

    --error inserter, not shown here for the sake of clarity

    par_encoder: entity corr_codes.encoder(behavioral)
        generic map (n => state'high, m => parity'high)
        port map(state => next_state, parity => next_parity);

    corrector: entity corr_codes.corrector(behavioral)
        generic map (n => state'high, m => parity'high)
        port map(state => state, in_parity => parity, corr_state => corr_state);

    combinational_circuit: process (corr_state, X)
    begin
        case corr_state is
            when "000" => --S0
                Y <= '0';
                if (X = '0') then next_state <= "000"; --S0
                else next_state <= "001"; end if; --S1
            when "001" => --S1
                Y <= '0';
                if (X = '0') then next_state <= "000"; --S0
                else next_state <= "011"; end if; --S2
            when "011" => --S2
                Y <= '0';
                if (X = '0') then next_state <= "010"; --S3
                else next_state <= "011"; end if; --S2
            when "010" => --S3
                Y <= '0';
                if (X = '1') then next_state <= "110"; --S4
                else next_state <= "000"; end if; --S0
            when "110" => --S4
                Y <= '1';
                if (X = '1') then next_state <= "011"; --S2
                else next_state <= "000"; end if; --S0
            when others => next_state <= "000"; --S0
                Y <= '0';
        end case;
    end process combinational_circuit;

end mixed;

```

**Fig. 5.** VHDL code describing a Moore self-correcting FSM with the aid of the developed modules

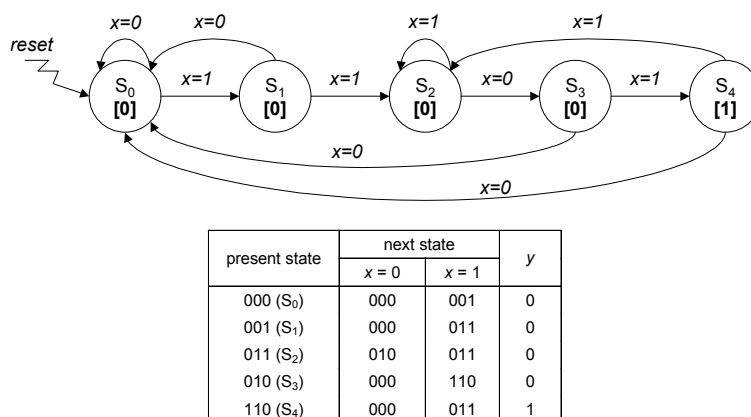


Fig. 6. State diagram and state/output table of a simple 4-bit sequence detector

### 4 Experiments

Obviously, the introduced error correction facility leads to area overhead and overall performance degradation. To estimate the influence of the added modules on the required FSM resources and the resulting clock frequency, two FSMs (*sec1* and *sec2*) have been selected.

Each FSM was synthesized using the Xilinx Synthesis Technology (XST) tool [6] targeted to Spartan-IIe xc2s300e –6 speed grade FPGA. For each FSM three types of state encoding have been examined (one-hot, binary, and Gray). After that each FSM was modified so as to provide for single-bit error correction as described in section 3, and the resulting VHDL descriptions were also synthesized. The obtained results expressed in terms of the required FPGA slices and the maximum attainable clock frequency, are presented in table 2. The synthesis process for both cases (i.e. for FSMs with and without error correction facilities) was optimized for speed.

Table 2. The results of experiments

FSM	inputs	outputs	states	state encoding	without error correction		with error correction	
					FPGA slices	performance	FPGA slices	performance
<i>sec1</i>	1	1	5	binary	2	285 MHz	7	133 MHz
				Gray	2	270 MHz	10	97 MHz
				one-hot	4	245 MHz	20	83 MHz
<i>sec2</i>	4	3	17	binary	18	123 MHz	39	65 MHz
				Gray	17	142 MHz	30	65 MHz
				one-hot	16	183 MHz	112	45 MHz

The worst results, in terms of both the increase in the number of FPGA slices and the performance degradation, were received for one-hot state encoding technique. This can be explained by the fact that one-hot state encoding technique requires more bits to represent FSM states and therefore always obligates more parity bits to be introduced than in the case of compact binary or Gray state encoding techniques. For example, *sec1* FSM has 5 states, which can be encoded with 3 bits (binary or Gray codes) supplemented by 3 parity bits (not counting the overall parity bit), whereas in the case of one-hot state encoding, 5 data bits and 4 parity bits are required. Augmenting the total number of bits from 6 (3+3) to 9 (5+4) leads obviously to increasing the complexity of both *Parity Encoder* and *Code Corrector* blocks and consequently increments the latency and the resource consumption of the whole control circuit.

From table 2 we can also see that as the FSM complexity increases, the error correction logic overhead becomes less noticeable and can be tolerated for high-reliability applications.

## 5 Conclusion

This paper proposed a design methodology for implementation of self-correcting control circuits derived from a VHDL specification. The entire approach has been presented, focusing the attention on the developed parameterizable VHDL package and the error-correcting modules, which can easily augment any VHDL FSM description with a single-bit error correction facility. The results obtained for a number of control circuits have been presented, which show that the inevitable area and performance overhead can be tolerated for high-reliability applications.

## References

1. Bolchini, C., Montandon, R., Salice, F., Sciuto, D.: Design of VHDL based Totally Self-Checking Finite-State Machine and Data-Path Descriptions. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 8, no. 1 (2000) 98-103
2. Zeng, C., Saxena, N., McCluskey, E.J.: Finite State Machine Synthesis with Concurrent Error Detection. In: Proc. Int. Test Conf. (1999) 672-679
3. Levin, I., Sinelnikov, V.: Self-Checking of FPGA-based Control Units. In: Proc. of the 9th Great Lakes Symposium on VLSI (1999) 292-295
4. Wakerly, J.F.: Digital Design. Principles and Practices. 3<sup>rd</sup> ed. Prentice Hall (2000)
5. Ashenden P.J.: The Designer's Guide to VHDL. Morgan Kaufmann Publishers, Inc. (1996)
6. ISE, FPGAs. [Online]. Available: [www.xilinx.com](http://www.xilinx.com)