National Institute of Technology

Warangal

Department of Electronics and Communication Engineering

Digital System and Design-ll

A Mini_project on

Self Repairing Adder Using Fault Localisation

Under the guidance of

Dr.B.Lakshmi Ma'am

Submitted by-
Princy Jacob Mesapam
Roll No.164138

INDEX

PROBLEM STATEMENT :

To design and develop a structural VHDL code for a “self repairing adder using fault localisation” and simulate using xilinx tool.

ABSTRACT :

An area-efficient self-repairing adder that can repair multiple faults and identify the particular faulty full adder is proposed. Fault detection and recovery has been carried out using self-checking full adders that can diagnose the fault based on internal functionality, independent of a fault propagated through carry.a fault can create problems in detecting the particular faulty full adder, and we need to replace the entire adder when an error is detected. We apply our self-checking full adder to a carry-select adder (CSeA) and show that the resulting self-checking CSeA consumes 15% less area compared to the previously proposed self-checking CSeA approach without fault localization. After observing fault localization with reduced area overhead, we utilize the self-checking full adder in constructing a self-repairing adder. The proposed self-repairing 16-bit adder can handle up to four faults effectively, with an 80% probability of error recovery compared to triple modular redundancy, which can handle only a single fault at a time.

INTRODUCTION :

A system will be fault secure if it remains unaffected by a fault or if it indicates a fault as soon as it occurs . A system will be self-testing if it produces a non-coded output in response to every generated fault. A system will be totally self-checking (TSC) if it is both fault secure and self-testing.In the past, many approaches have been adopted to introduce self-checking in adder circuits either by hardware- or time-based redundancy. These approaches can detect an error without indicating its exact location because of fault propagation due to carry. In this paper, we propose a new self-checking and self-repairing full adder in which the faulty full adder module can be identified.

PREVIOUS DESIGN APPROACHES

1. TIME REDUNDANCY :

The basic concept of time redundancy is that the same hardware will perform a single operation in different intervals of time, and we can detect an error by comparing the two outputs obtained at different time instances.The single full adder will work such that it is used twice to perform the same computation by following different logical paths, and the comparison of the final results will indicate the presence of a fault. The drawback in this ise, if the first computed result is faulty, then before detecting the fault, that result will be used for other computations. The propagated fault will also cause faulty results in other modules.

2. HARDWARE REDUNDANCY :

The basic idea for this approach is to use more than one hardware to produce either the same, inverted or coded output. The comparison between the outputs of the actual and the redundant hardware will be used to indicate the fault. The two most commonly used approaches are double modular redundancy (DMR) and triple modular redundancy (TMR). In a duplex system (i.e., DMR) the same hardware is repeated once, whereas TMR requires three modules with which to vote to determine the final output. The drawback in this is that instead of detecting faults in individual modules, it can detect the overall fault of a system. This kind of checking creates many complexities in terms of fault recovery, because we cannot detect which individual block is faulty. The other drawback of this approach is that straightforward implementation will create an area overhead of more than 100% . Secondly, it cannot detect a common mode failure in which both modules experience the same set of errors .

BASIC PRINCIPLE :

We utilize the following relations for the full adder :

- The Sum and Carry bits will be equal to each other when all three inputs are equal.
- The Sum and Carry bits will be complemented when any of the three inputs is different.

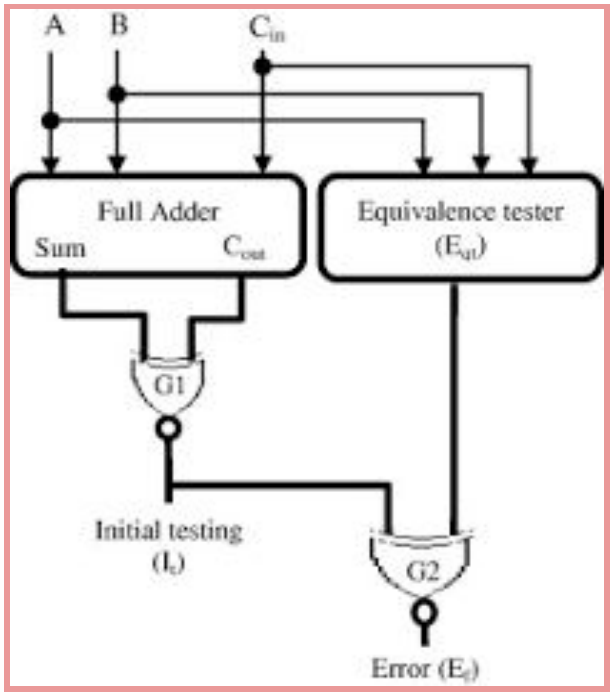
A full adder can be self checked with only the expense of an equivalence tester (Eqt). The purpose of the equivalence tester is to check the equivalence of all inputs.The functional block Eqt has been designed to produce an active low output so that we can use the fault-secure exclusive nor logical function (XNOR) gate for comparison.

Sum = A xor B xor Cin.
Cout = A.B + Cin .(A+B)

Equivalence tester (Eqt) = NOT((NOT A . NOT B . NOT Cin) + (A . B . Cin)).
Error (Ef) = (Sum) xnor (Cout) xnor (Eqt).

The final fault is computed by using two XNOR gates. The purpose of the first XNOR gate (G1) is to check whether the Sum and Cout bits are equal or complemented.
We need a second XNOR gate (G2) because of the previously mentioned observation that Sum and Cout will always complement each other except when all inputs are equal.

- When Eqt is zero, the output of G1 and G2 should be logic 1 and 0, respectively.
- If Eqt indicates logic 1, then both XNOR gates should generate logic 0 (i.e., It = 0 and Ef = 0).
- In other cases a fault is detected.

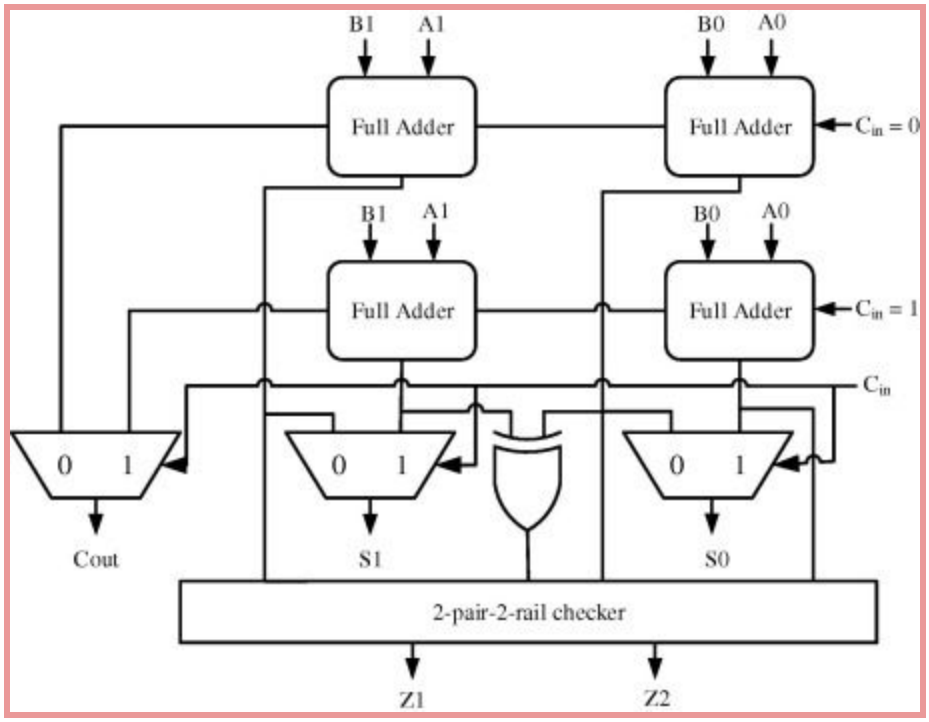


SELF-CHECKING CARRY SELECT ADDER :

Here two adders and a 2-pair-2-rail checker, along with some combinational logic, are used for self-checking. We design a self-checking CSeA with single adder, assuming the initial Cin to be zero and where the individual full adder has been replaced by our proposed self-checking full adders. We then generate Sum and Carry output for the initial Cin equal to one using the following relation, where S i j and Ci j represent the

jth position of the Sum and Carry bits, and i is either 0 or 1, indicating the value of the initial Cin.

- S1 0 is always a complement of S0 0, i.e. S1 0 =NOT(S0 0).
- S1 1 depends on S0 0, S0 1:
  If S0 0 = 0 then S1 1 = S0 1
  If S0 0 = 1 then S1 1 = NOT(S0 1).
- C1 1 depends on C0 1, S0 0 and S0 1:
  If C0 1= 1 OR (S0 0 . S0 1) = 1 then C1 1 = 1;



The above equations can be summarised as

- S1 0 = NOT(S0 0)
- S1 1= S0 0 xor S0 1
- C1 1 = C0 1 xor (S0 1 . S0 0)

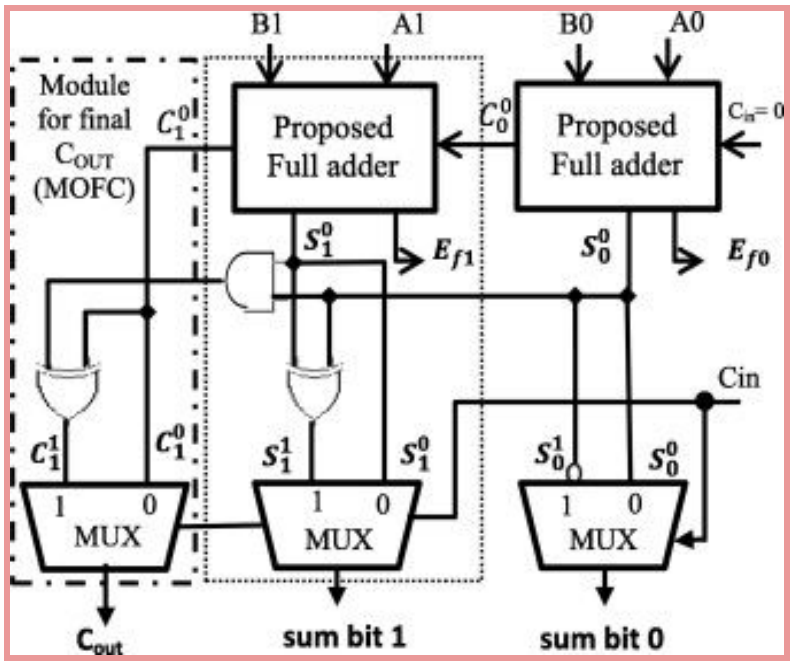
Except for the least significant bit (LSB), S0 j and S1 j has the following relation:

- If (S0 1 S0 2 S0 3 ..... S0 (i-1))=1
  Then S1 j=NOT(S0 j) ;
  Else S 1 j = S 0 j ;

where j indicates the bit position.

The design module used to extend the 2-bit CSeA design to an n-bit CSeA design is

- S 1 0 = NOT(S0 0)
- S 1 n = (S 0 n) xor(S 0 1 . S 0 2 . S 0 3 ..... S 0 (n-1))
- C1 n = C0 n xor (S0 1 . S0 2 . S0 3 ..... S0 (n-1))



Two bit proposed self checking adder.

FAULT COVERAGE :

The logic-high of the final error (Ef) will indicate a fault. We can easily see that the designed approach guarantees self-checking only when any one of the Sum, Cout or

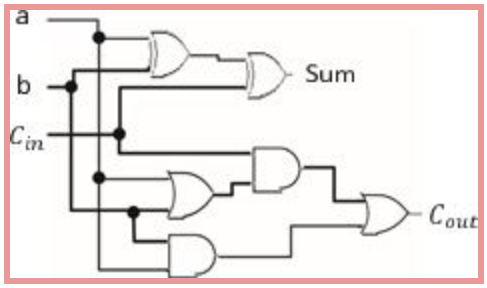
Eqt lines becomes faulty. If two of them have a fault at the same time, thenthat fault cannot be detected. The assumption of having a single fault at one time becomes valid because the fault-secure property assumes that between two consecutive faults, we have enough time to detect the error at its first occurrence.

The conditions for fault coverage of the design have been summarized as

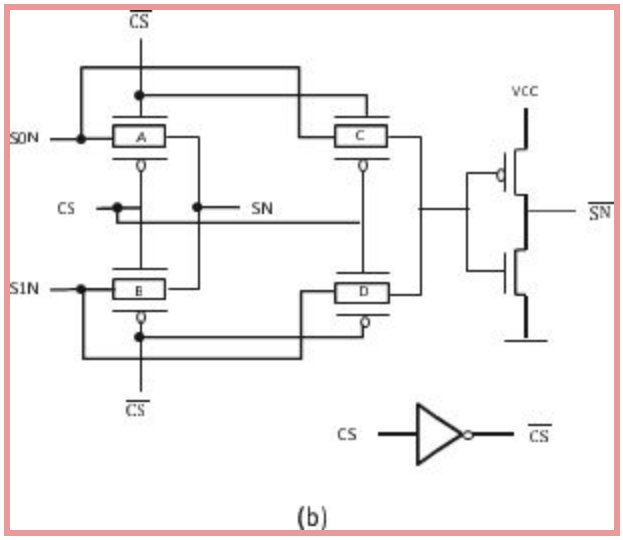
Conditions	Status
If Eqt = 0 and Sum = Cout	No fault
If Eqt = 1 and Sum = NOT(Cout)	No fault
If Eqt = 0 and Sum != Cout	Fault
If Eqt = 1 and Sum != NOT(Cout)	Fault

For example: A = 1, B = 0 and Cin = 1 are three respective inputs in our proposed full adder. The corresponding outputs for these inputs without fault are: Sum = 0, Carry = 1 and Eqt =1. Let us assume that the Sum bit flips from 0 to 1, and the outputs, after the fault, will thus become Sum = 1, Carry = 1 and Eqt =1. This condition has been indicated as a fault.

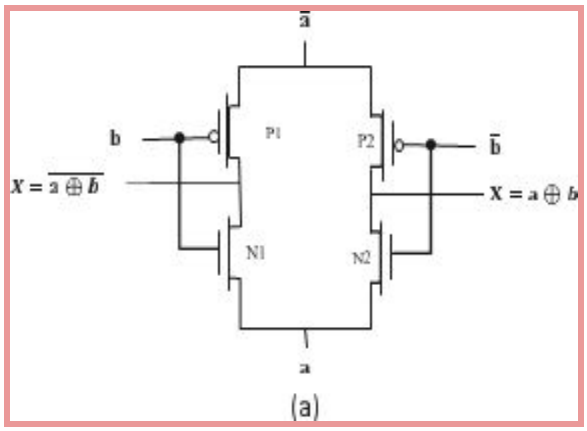
➔ With our proposed approach for a self-checking adder, a fault generated in any full adder can be detected individually. Even if the generated carry has a fault, it will not create a fault indication in subsequent full adders because all the full adders are self-checking with respect to their individual functionality and are independent on their carry input. Hence, the faulty propagated carry will not create a problem in self-checking in any subsequent full adder.



➔ In addition to a self-checking full adder, we need to have a self checking XOR gate and multiplexer (MUX).



Self checking MUX



Self checking XOR.

COMPARISON :

. The area overhead in both cases was computed in terms of transistor count. The area for proposed adder is less as compared to DMF and TMF.

The final transistor count for individual modules is

AND	6
XOR	4

MUX	12
Adder	28
Equivalence tester (Eqt)	12
Checker (2-XOR)	8
Module for final Cout (MOFC) (shown in Fig. 4)	16

The transistor count for our proposed self-checking CSeA with multiple bit data is 58. Our designed self-checking CSeA has only negligible area overhead compared to a normal CSeA design and also requires a 15–16% lower transistor count compared to the self-checking CSeA.

SELF REPAIRING ADDER

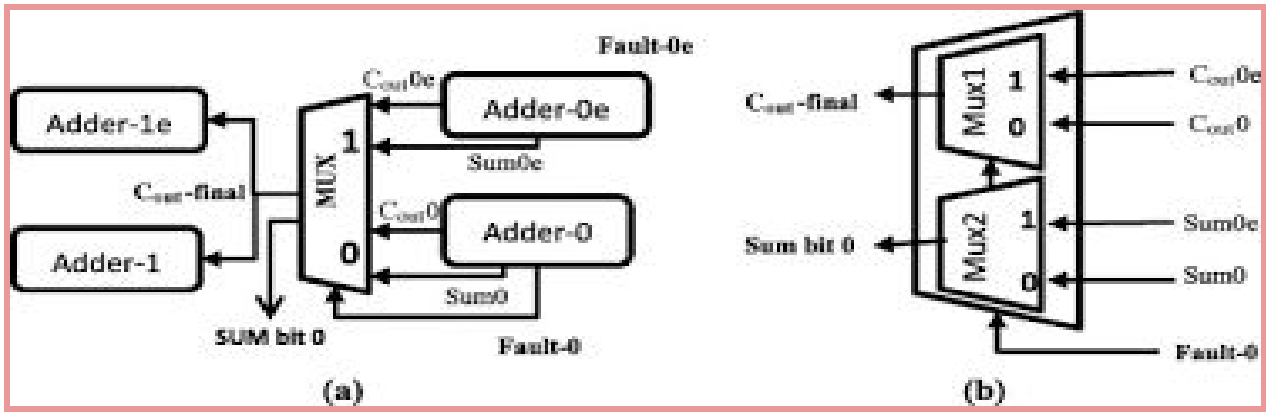
DESIGN CHALLENGES AND SOLUTION :

Along with the area, power consumption and other limitations for a self-reliable adder, there is one important challenging factor that is also related to carry propagation. Although our proposed self-checking adder is independent of fault propagation due to carry, for fault recovery at run time, the propagated carry is crucial. If a fault is detected in any full adder, then because of fault propagation due to carry, we cannot trust the output of the corresponding full adders. Therefore, if we replace the particular faulty module, we need to re-execute all addition processes to get true and reliable output. However, this whole process will increase processing time.

In our proposed design, we provide a solution for this problem by designing a reliable carry propagation chain using a dedicated self-checking adder for replacements. If both the adders are faulty at the same time, this will be indicated as a critical situation. It is possible to have two faults at a time, like the above-mentioned problem, but there is a low probability of the fault being at the same adder position.

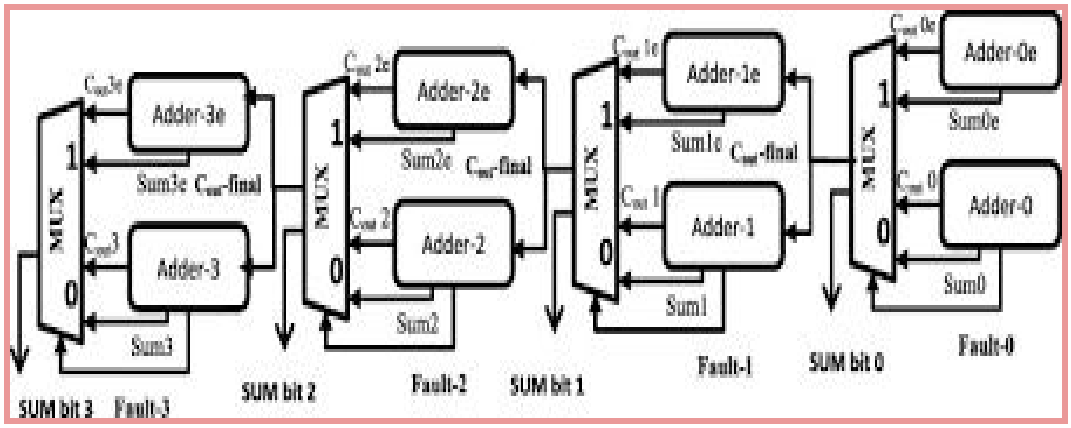
PROPOSED SELF REPAIRING ADDER :

The design approach requires two proposed self-checking adders working at one time on the same input bits.



The operation of the proposed design is such that one adder behaves as a normal adder (indicated by Adder-0) while the second adder will work for recovery at run time (indicated by Adder-0e). The suffix ‘‘e’’ indicates an ‘‘extra’’ adder used for fault recovery. The MUX will only be dependent on the fault of the normal adder (i.e., Adder-0). If any fault occurs in the normal adder, then the MUX will export the Sum and Carry-out from the extra adder module (i.e., Adder-0e).
The final Carry-out obtained from the MUX will be fed to both full adders present in order to compute the next Sum bit.

The Proposed 4-bit Self repairing Adder is shown in the below figure



Comparison with TMR in area overhead :

In order to introduce self-repair in DMR, we need to have either a complex architecture, which requires huge area overhead, or time redundancy, which slows

system performance. On the other hand, TMR is able to provide reliable output but cannot assure reliability for more than one fault. . In order to make TMR indicate a faulty module, we need extra circuitry.Our proposed design, with relatively less area overhead, can repair more than one fault, compared to TMR.

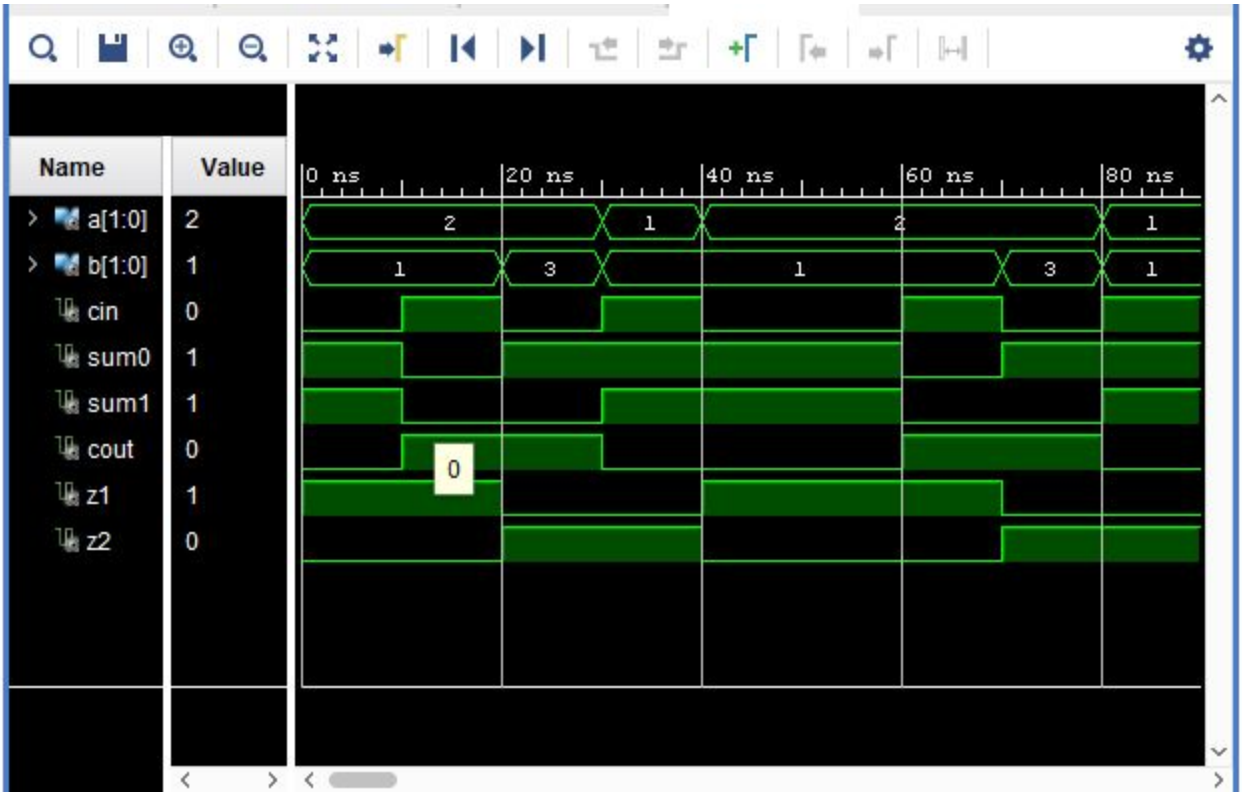
VHDL Code :

1 . SELF TESTING CARRY SELECT ADDER

```
entity self_testing_csa is
   Port ( a,b : in STD_LOGIC_VECTOR (1 downto 0);
          cin : in STD_LOGIC;
          sum0 : out STD_LOGIC;
          sum1 : out STD_LOGIC;
          cout : out STD_LOGIC;
          z1,z2 : out STD_LOGIC);
end self_testing_csa;

architecture Behavioral of self_testing_csa is

component full_adder is
   Port ( a,b : in STD_LOGIC_VECTOR(1 downto 0);
          cin : in STD_LOGIC;
          sum : out STD_LOGIC_VECTOR (1 downto 0);
          carry : out STD_LOGIC_VECTOR (1 downto 0));
end component;

component mux_2 is
   Port ( i0,i1 : in STD_LOGIC;
          s0 : in STD_LOGIC;
          y : out STD_LOGIC);
end component;

signal  s0,s1,c0,c1 :  STD_LOGIC_VECTOR (1 downto 0);
```

```
signal y1 : std_logic;
begin

u0 : full_adder port map(a,b,'0',s0,c0);
u1 : full_adder port map(a,b,'1',s1,c1);
u2 : mux_2 port map(s0(0),s1(0),cin,sum0);
u3 : mux_2 port map(s0(1),s1(1),cin,sum1);
u4 :mux_2 port map(c0(1),c1(1),cin,cout);

y1 <= s1(1) xor s0(0);

z1<= (s0(0) and y1) or (not(s0(1)) and s1(0));
z2 <= (s0(0) and not(s0(1))) or (s1(0) and y1);

end Behavioral;
```

SIMULATION RESULTS :

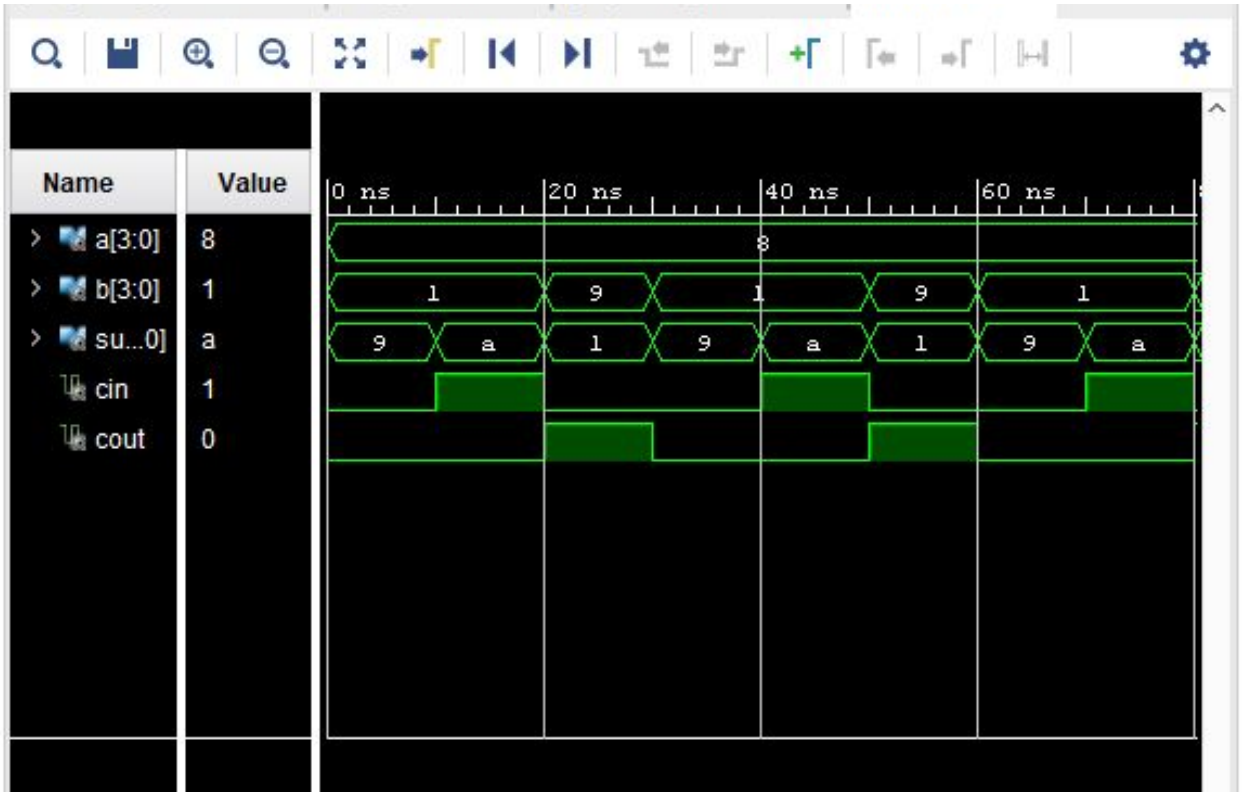


2 . 4_BIT PROPOSED CARRY SELECT ADDER

```
entity proposed_csa is
    Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           cin : in STD_LOGIC;
           cout : out STD_LOGIC;
           sum : inout STD_LOGIC_VECTOR (3 downto 0));
end proposed_csa;

architecture Behavioral of proposed_csa is

component self_fa is
    Port ( a,b,cin : in STD_LOGIC;
            sum : inout STD_LOGIC;
            cout : inout STD_LOGIC;
           Ef : out STD_LOGIC);
end component;

component mux_2 is
    Port ( i0,i1 : in STD_LOGIC;
       s0 : in STD_LOGIC;
       y : out STD_LOGIC);
end component;

signal ef,s0,s1,c0,x : std_logic_vector(3 downto 0);
signal c1 : std_logic;
begin

u0 : self_fa port map(a(0),b(0),'0',s0(0),c0(0),ef(0));
x(0)<=s0(0);
s1(0) <= not(s0(0));
s1(1) <= x(0) xor s0(1);
x(1) <= (x(0) and s0(1));
u1 : self_fa port map(a(1),b(1),c0(0),s0(1),c0(1),ef(1));
u2 : self_fa port map(a(2),b(2),c0(1),s0(2),c0(2),ef(2));
x(2)<= x(1) and s0(2);
s1(2)<= x(1) xor s0(2);
u3 : self_fa port map(a(3),b(3),c0(2),s0(3),c0(3),ef(3));
x(3)<= x(2) and s0(3);
```

```
s1(3)<= x(2) xor s0(3);
c1 <= x(3) xor c0(3);
u4 : mux_2 port map(s0(0),s1(0),cin,sum(0));
u5 : mux_2 port map(s0(1),s1(1),cin,sum(1));
u6 : mux_2 port map(s0(2),s1(2),cin,sum(2));
u7 : mux_2 port map(s0(3),s1(3),cin,sum(3));
u8 : mux_2 port map(c0(3),c1,cin,cout);

end Behavioral;
```

SIMULATION RESULTS :



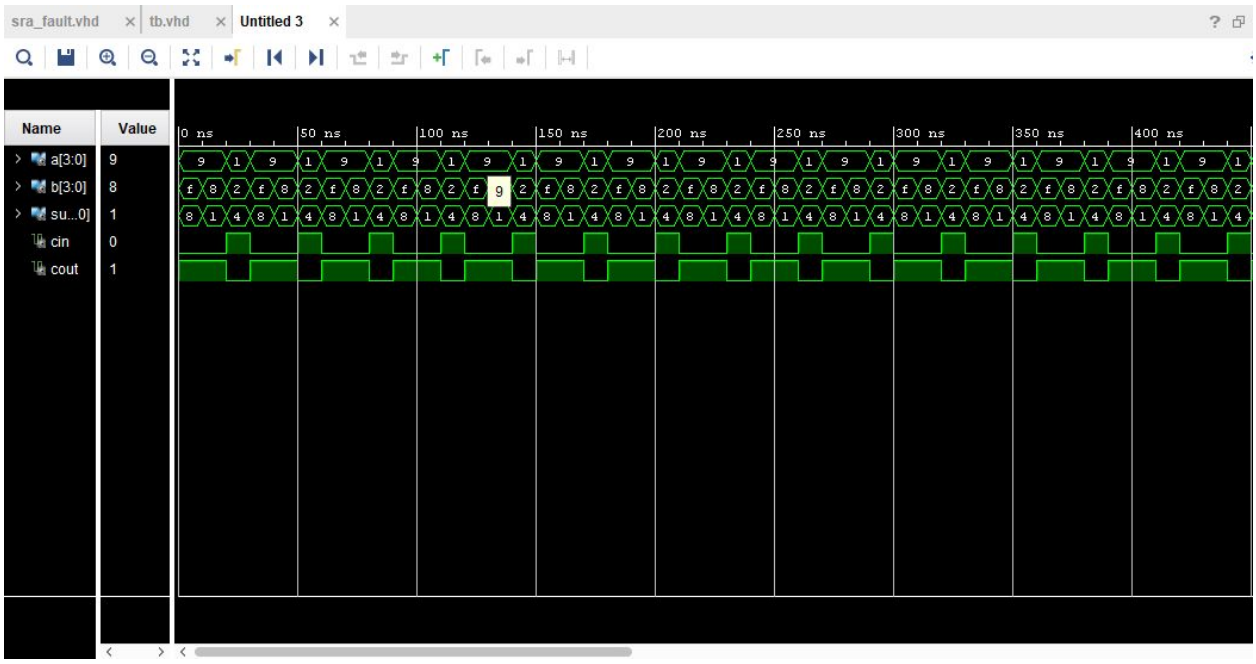
3. SELF REPAIRING FULL ADDER :

```
entity sra_fault is
    Port ( a,b : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        cin : in STD_LOGIC;
        cout : out STD_LOGIC;
        sum : out STD_LOGIC_VECTOR (3 downto 0));
end sra_fault;

architecture Behavioral of sra_fault is

component proposed_csa is
    Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        cin : in STD_LOGIC;
        c : inout STD_LOGIC_VECTOR(3 downto 0);
        ef : inout STD_LOGIC_VECTOR( 3 downto 0);
        sum : inout STD_LOGIC_VECTOR (3 downto 0);
        cout : out STD_LOGIC);
end component;

component mux_2 is
    Port ( i0,i1 : in STD_LOGIC;
        s0 : in STD_LOGIC;
        y : out STD_LOGIC);
end component;

component full_adder is
    Port ( a,b : in STD_LOGIC_VECTOR(3 downto 0);
        cin : in STD_LOGIC;
        sum : out STD_LOGIC_VECTOR (3 downto 0);
        carry : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal ce,se,carry,s,ef : STD_LOGIC_VECTOR(3 downto 0);
signal coute :STD_LOGIC;
begin

u1 : proposed_csa port map(a,b,cin,carry,ef,s,coute);
u2 : full_adder port map(a,b,cin,se,ce);

u3 : mux_2 port map(s(0),se(0),ef(0),sum(0));
u4 : mux_2 port map(s(1),se(1),ef(1),sum(1));
```

```
u5 : mux_2 port map(s(2),se(2),ef(2),sum(2));
u6 : mux_2 port map(s(3),se(3),ef(3),sum(3));

u7 : mux_2 port map(ce(3),carry(3),ef(3),cout);


end Behavioral;
```

SIMULATION RESULTS :



VHDL codes of components used above :

- MUX :

```
entity mux_2 is
    Port ( i0,i1 : in STD_LOGIC;
           s0 : in STD_LOGIC;
           y : out STD_LOGIC);
```

```
end mux_2;

architecture Behavioral of mux_2 is

begin
process(i0,i1,s0)
begin
if (s0 = '0') then y<=i0;
else y<=i1;
end if;
end process;
end Behavioral;
```

- Full Adder :

```
entity full_adder is
    Port ( a,b : in STD_LOGIC_VECTOR(1 downto 0);
           cin : in STD_LOGIC;
           sum : out STD_LOGIC_VECTOR (1 downto 0);
           carry : out STD_LOGIC_VECTOR (1 downto 0));
end full_adder;

architecture Behavioral of full_adder is


begin
process (a,b) is
variable c : std_logic_vector(2 downto 0);
begin
  c(0) := cin;
  for i in 0 to 1 loop
   sum(i) <= a(i) xor b(i) xor c(i);
   c(i+1) := (a(i) and b(i)) or (b(i) and c(i)) or (c(i) and a(i));
  end loop;
  carry(1)<= c(2);
  carry(0) <= c(1);
end process;
```

```
end Behavioral;
```

- Proposed Full Adder :

```
entity self_fa is
    Port ( a,b,cin : in STD_LOGIC;
           Ef : out STD_LOGIC;
           sum : inout STD_LOGIC;
           cout:inout STD_LOGIC);
           end self_fa;

architecture Behavioral of self_fa is

signal eqt : std_logic;
begin
sum <= a xor b xor cin;
cout <= (a and b) or (b and cin) or (cin and a);
eqt <= not((not(a) and not(b) and not(cin))or ( a and b and cin));
Ef <= (sum xnor cout) xnor eqt;

end Behavioral;
```

CONCLUSION :

With our proposed design of a self-checking full adder, a fault generated in any full adder can be detected individually. Even the error propagated through carry will not create a fault indication in subsequent full adders because all full adders are self-checking with respect to their individual functionality. We then proposed a self-repairing adder by using the fault localization property present in our proposed self-checking full adder. The designed self-repairing adder can provide reliable output for more than one fault, with reduced area overhead. This is because, instead of replacing the whole system of adders, we replace the particular faulty full adder only. Moreover, our proposed 4-bit self-repairing adder can repair up to 4 faults with 80% probability of error recovery.