# Project Report

## Testing and Verification

Dr. Rama S. Komaragiri

Ayush Vatsal
E19ECE038

Sukrit Chatterjee
E19ECE039

# Contents

# Introduction

Producing a digital system begins with a designer specifying his or her design in a high-level design language, and ends with manufacturing and shipping parts to the customer. This process involves many simulations, synthesis, and test phases. In this project the design is analysed in RTL level design simulation.

Multiplication is one of the most used algorithms in computer science. The best known time complexity for multiplication is still larger than $O(n.\log_2 n)$. GMP uses a variety of faster algorithms, depending on the operand size. The asymptotically fastest of these, Schönhage–Strassen algorithm, has complexity $O(n.\log_2(n.\log_2(\log_2 n)))$. The complexity of the optimal integer multiplication algorithm is not known. There is an algorithm with better complexity than Schönhage–Strassen algorithm, known as Fürer's algorithm, which uses $n.\log_2 n.2^{O(\log 2n)}$ time. For further larger numbers, karatsuba's algorithm or Tom cook's algorithm are employed. All of the algorithms need binary multipliers and adders at the gate level.

This project implements an RTL design synthesis and testing of a 2-bit binary multiplier, i.e. a module which takes two 2-bit binary numbers as inputs and produces their 4-bit binary product.

# Multiplier

A multiplier is a circuit that returns the multiplication of two numbers. The circuit implemented in this report, uses only NAND, NOR, and NOT gates to implement a 2-bit binary multiplier.

The circuit uses 11 NAND gates, 4 NOR gates and 19 NOT gates. There are 4 primary input lines(PI lines) and 4 output lines. Depth of the circuit is four(excluding the NOT gates), amounting to delay of 4 units.

# RTL design process

In a register transfer level (RTL) design process, the designer first writes his or her design specification in an RT level language such as VHDL. Using standard HDL (Hardware Description Language) descriptions and testbenches in the same HDL, this description will be simulated and tested for design errors.
*RTL Simulation* : The testing of the design here is primarily functional that is extracted from the original specification of the circuit being designed. Detailed timing checks and physical flaws are not addressed at this level of simulation. For analysing the behaviour of the design, the testbench can inject design errors to predict the behaviour of the design under unanticipated circumstances. After a satisfactory simulation, and when the designer is reasonably sure that his or her description of the design meets the design specifications.

*RTL Level Synthesis* : RT level synthesis takes the behavioural description of a design as input, and produces a netlist of the design. The netlist specifies interconnection of low-level basic logic components such as AND and OR gates and D-type flip-flops. The exact set of gates used in this level of description depends on the target library, which is the library of components provided by the chip manufacturers. The format for the netlist can be specified to be the same HDL as the original design. Before going to the next step, this netlist must be tested. The testing done in this project is done by simulating it with GHDL, a VHDL simulation tool. This simulation phase is referred to as post-synthesis simulation. With this simulation we are checking for delay issues, races, clock speed, and errors caused by misinterpretation of the RT level design by the synthesis tool.

# HDL Simulation

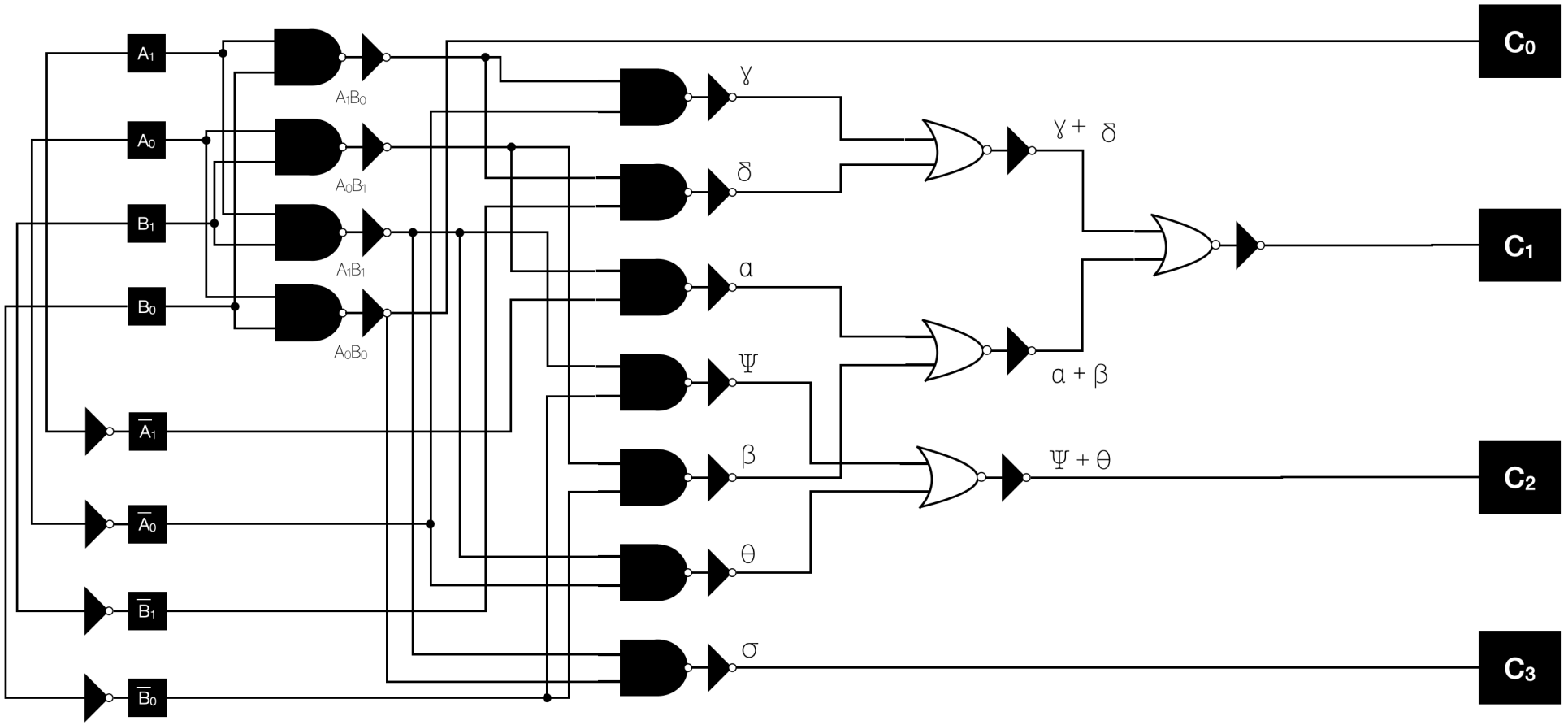
```
library IEEE;
use IEEE.std_logic_1164.all;

entity multiplier is
port(
  A0_in: in std_logic;
  A1_in: in std_logic;
  B0_in: in std_logic;
  B1_in: in std_logic;
  C0: out std_logic;
  C1: out std_logic;
  C2: out std_logic;
  C3: out std_logic;
  C0_check: out std_logic;
  C1_check: out std_logic;
  C2_check: out std_logic;
  C3_check: out std_logic;
  disp_A1B1: out std_logic;
  disp_A0B0: out std_logic;
  disp_A0B1: out std_logic;
  disp_A1B0: out std_logic;
  disp_al: out std_logic;
  disp_be: out std_logic;
  disp_ga: out std_logic;
  disp_de: out std_logic;
  disp_al_or_be: out std_logic;
  disp_ga_or_de: out std_logic;
  disp_si: out std_logic;
  disp_th: out std_logic
  );
end multiplier;

architecture arc_multiplier of multiplier is
begin
  process(A1_in, A0_in, B1_in, B0_in) is
  variable A1: std_logic;
  variable A0: std_logic;
  variable B1: std_logic;
  variable B0: std_logic;
```

```
variable A1B1: std_logic;
variable A0B0: std_logic;
variable A0B1: std_logic;
variable A1B0: std_logic;
variable al: std_logic;
variable be: std_logic;
variable ga: std_logic;
variable de: std_logic;
variable al_or_be: std_logic;
variable ga_or_de: std_logic;
variable si: std_logic;
variable th: std_logic;
variable cc0: std_logic;
variable cc1: std_logic;
variable cc2: std_logic;
variable cc3: std_logic;
variable c0_c: std_logic;
variable c1_c: std_logic;
variable c2_c: std_logic;
variable c3_c: std_logic;

begin
  A1 := A1_in;
  A0 := A0_in;
  B1 := B1_in;
  B0 := B0_in;

  A1B1 := not(A1 nand B1);
  A0B0 := not(A0 nand B0);
  A0B1 := not(A0 nand B1);
  A1B0 := not(A1 nand B0);

  al := not(A0B1 nand not(A1));
  be := '0';
  ga := not(A1B0 nand not(A0));
  de := not('1' nand not(B1));
  si := not(A1B1 nand not(B0));
  th := not(A1B1 nand not(A0));

  al_or_be := not(al nor be);
  ga_or_de := not(ga nor de);
```

```
   -- C1
    C0 <= A0B0;
    cc0 := A0B0;
    C0_check <= A0 and B0;
    c0_c := A0 and B0;
    if cc0/=c0_c then
      C0 <= 'X';
    end if;
    -- C1
    C1 <= not(al_or_be nor ga_or_de);
    cc1 := not(al_or_be nor ga_or_de);
    C1_check <= (A0 and not A1 and B1) or (A0 and not B0 and B1) or (not A0
and A1 and B0) or (A1 and B0 and not B1);
    c1_c := (A0 and not A1 and B1) or (A0 and not B0 and B1) or (not A0 and
A1 and B0) or (A1 and B0 and not B1);
    if cc1/=c1_c then
      C1 <= 'X';
    end if;
    -- C2 not(not(not(A1 nand B1) nand (not B0)) nor not(not(A1 nand B1)
nand (not A0)))
    C2 <= not(si nor th);
    C2_check <= (A1 and B1 and not B0) or (A1 and not A0 and B1);
    cc2 := not(si nor th);
    c2_c := (A1 and B1 and not B0) or (A1 and not A0 and B1);
    if cc2/=c2_c then
      C2 <= 'X';
    end if;
    -- C3 not(not(not(A1 nand A0) nand B1) nand B0)
    C3 <= not(A0B0 nand A1B1);
    C3_check <= A1 and A0 and B1 and B0;
    cc3 := not(A0B0 nand A1B1);
    c3_c := A1 and A0 and B1 and B0;
    if cc3/=c3_c then
      C3 <= 'X';
    end if;
  end process;
end arc_multiplier;
```

VHDL design code

# RTL Schematic of VHDL design




Sukrit Chatterjee
Ayush Vatsal

2-bit MULTIPLIER
Using NAND, NOR, NOT

Testing and Verification
EECE343L

# Output waveforms

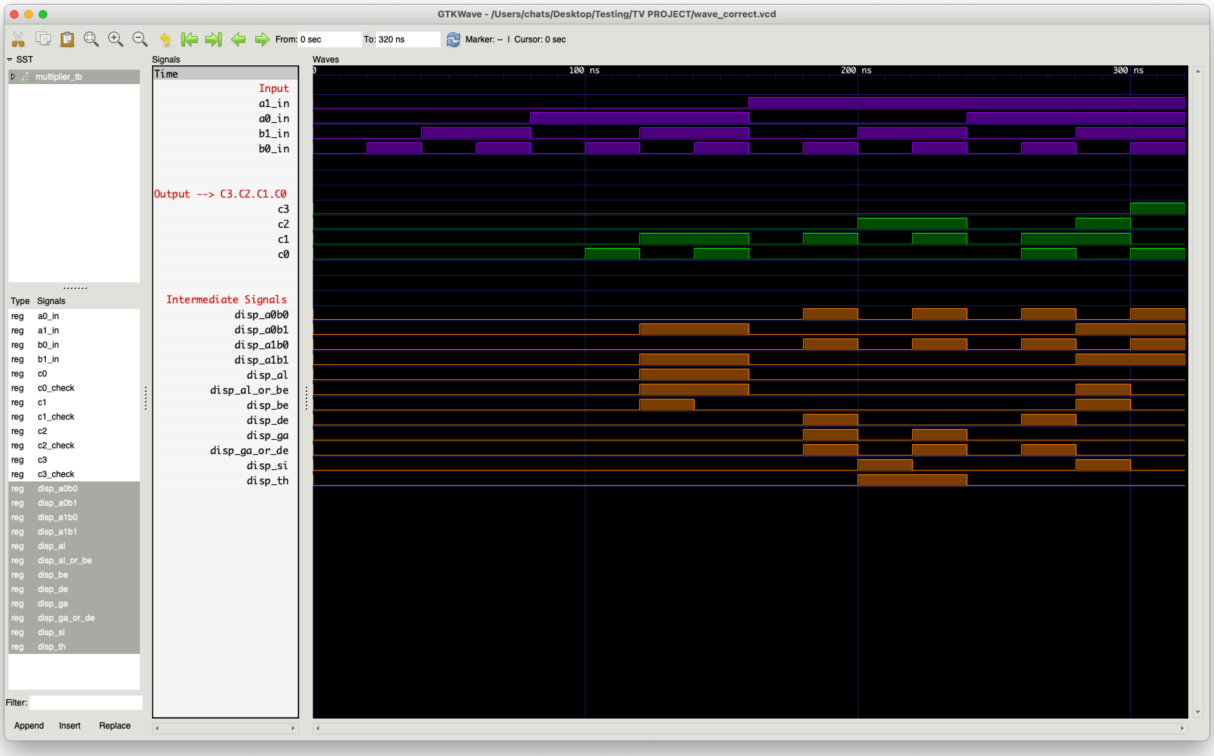
The output bits are calculated using midterm from the generated truth table and verified as correct. The resultant boolean expressions are noted below as well the truth table and the waveform snippet generated from the VHDL simulation.

$$C_0 = A_0 \cdot B_0$$

$$C_1 = A_0 \cdot \overline{A_1} \cdot B_1 \; + \; A_0 \cdot \overline{B_0} \cdot B_1 \; + \; A_1 \cdot \overline{B_1} \cdot B_0$$

$$C_2 = A_1 \cdot \overline{B_0} \cdot B_1 \; + \; A_1 \cdot \overline{A_0} \cdot B_1$$

$$C_3 = A_1 \cdot A_0 \cdot B_1 \cdot B_0$$




Waveform with no faults

| S no | A | B | C |
|---|---|---|---|
| 1 | 00 | 00 | 0000 |
| 2 | 00 | 01 | 0000 |
| 3 | 00 | 10 | 0000 |
| 4 | 00 | 11 | 0000 |
| 5 | 01 | 00 | 0000 |
| 6 | 01 | 01 | 0001 |
| 7 | 01 | 10 | 0010 |
| 8 | 01 | 11 | 0011 |
| 9 | 10 | 00 | 0000 |
| 10 | 10 | 01 | 0010 |
| 11 | 10 | 10 | 0100 |
| 12 | 10 | 11 | 0110 |
| 13 | 11 | 00 | 0000 |
| 14 | 11 | 01 | 0011 |
| 15 | 11 | 10 | 0110 |
| 16 | 11 | 11 | 1001 |

Truth Table with no faults.

# Single sa-faults

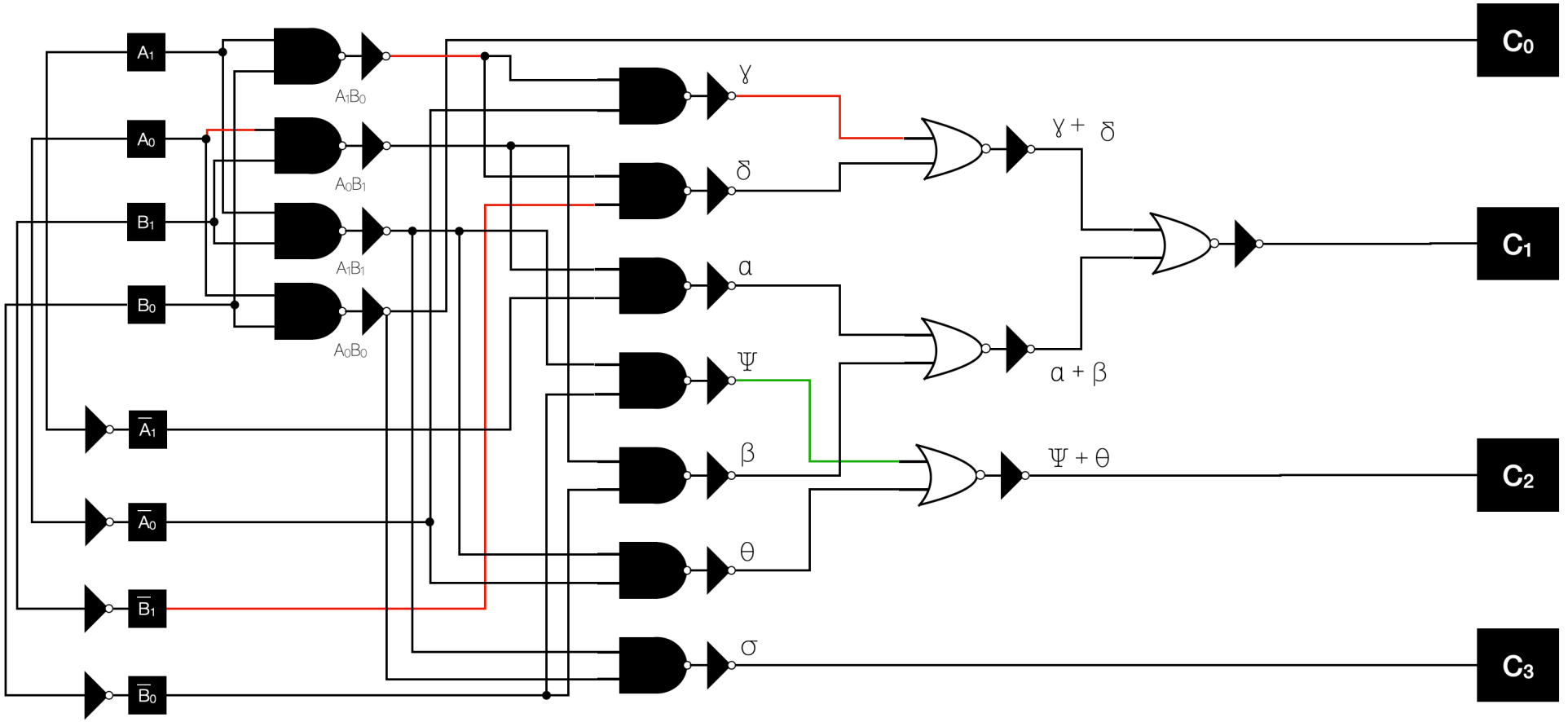
The number of possible single stuck at faults in the schematic are 57, accounting for 11 NAND gate input and output lines, 4 NOR gate input and output lines, primary input and output lines. This project generates and tests a given set of single sa-faults, assuming that there is only 1 fault present in the circuit at a time.
The set of given faults are:
Fault 1: $A_0B_0$ sa-0
Fault 2: gamma sa-0
Fault 3: psi sa-1
Fault 4: not $B_1$ in delta sa-0
Fault 5: $A_0$ in $A_0B_1$.

The test patterns were first generated using a random pattern generator. The results are as follows: (test vector - $A_1A_0B_1B_0$)

| Test Vector | Fault 1 | Fault 2 | Fault 3 | Fault 4 | Fault 5 | Total faults, redundant | FC cumulative |
|---|---|---|---|---|---|---|---|
| 0101 | 0000 - ✅ | 0001 | 0101 - ✅ | 0001 | 0001 | 2, 0 | 40% |
| 0111 | 0010 - ✅ | 0011 | 0111 - ✅ | 0011 | 0001 - ✅ | 3, 2 | 60% |
| 1101 | 0010 - ✅ | 0011 | 0111 - ✅ | 0001- ✅ | 0011 | 3, 2 | 80% |
| 0011 | 0000 | 0000 | 0100 - ✅ | 0000 | 0000 | 1, 1 | 80% |
| 1110 | 0110 | 0110 | 0110 | 0110 | 0100 - ✅ | 1, 1 | 80% |

After 5 test loops, the random test pattern generator results saturated with no (significant) increase in fault coverage. Deterministic test pattern generation was used to find the test vector that detects Fault 2.

# RTL Schematic of faulty VHDL design





Sukrit Chatterjee
Ayush Vatsal


2-bit MULTIPLIER
Using NAND, NOR, NOT

Testing and Verification
EECE343L

stuck-at 0
stuck-at 1

Using single path sensitisation algorithm, the fault at $\gamma$ was activated, and propagated to the output line $C_1$ using the only path that was available.
The resulting test vector is noted in the table below along with the (now) complete test set.

| Test Vector | Fault 1 | Fault 2 | Fault 3 | Fault 4 | Fault 5 | Total faults, redundant | Fault coverage |
|---|---|---|---|---|---|---|---|
| 0111 | 0010 - ✅ | 0011 | 0111 - ✅ | 0011 | 0001 - ✅ | 3, 0 | 60% |
| 1101 | 0010 - ✅ | 0011 | 0111 - ✅ | 0001- ✅ | 0011 | 3, 2 | 20% |
| 1011 | 0110 | 0100 - ✅ | 0110 | 0110 | 0110 | 1, 0 | 20% |

As we can see, all the faults, i.e. faults 1 through 5 are uniquely detectable.

1. [01, 11] — Fault 1 has detectable error in line $C_0$.
   Fault 3 has detectable error in line $C_2$.
   Fault 5 has detectable error in line $C_1$.
2. [11, 01] — Fault 4 has detectable error in line $C_1$.
3. [10, 11] — Fault 2 is the only fault which results in an error.

# Multiple sa-faults

Although the number of possible multiple stuck-at faults are much much larger than single stuck-at faults, the probability of them occurring are exponentially lower.

Here, a given set of 3 possible multiple stuck-at faults are tested for. They are as follows:

Fault 1: $B_1$ sa-1, and theta sa-0

Fault 2: $\alpha+\beta$ sa-1, and $A_0B_0$ sa-0

Fault 3: $A_1B_0$ in delta sa-1, beta sa-0

| S no | A | B | C (No Error) | C — B1 sa-1 and Theta sa-0 | C — Alpha + Beta sa-1 and A0B0 sa-0 | C — A1B0 in de sa-1 and beta sa-0 |
|---|---|---|---|---|---|---|
| 1 | 00 | 00 | 0000 | 0000 | 0010 🚫 | 0010 🚫 |
| 2 | 00 | 01 | 0000 | 0000 | 0010 🚫 | 0010 🚫 |
| 3 | 00 | 10 | 0000 | 0000 | 0010 🚫 | 0000 |
| 4 | 00 | 11 | 0000 | 0000 | 0010 🚫 | 0000 |
| 5 | 01 | 00 | 0000 | 0010 🚫 | 0010 🚫 | 0010 🚫 |
| 6 | 01 | 01 | 0001 | 0011 🚫 | 0010 🚫 | 0011 🚫 |
| 7 | 01 | 10 | 0010 | 0010 | 0010 | 0010 |
| 8 | 01 | 11 | 0011 | 0011 | 0010 🚫 | 0011 |
| 9 | 10 | 00 | 0000 | 0100 🚫 | 0010 🚫 | 0010 🚫 |
| 10 | 10 | 01 | 0010 | 0010 | 0010 | 0010 |
| 11 | 10 | 10 | 0100 | 0100 | 0110 🚫 | 0100 |
| 12 | 10 | 11 | 0110 | 0010 🚫 | 0110 | 0110 |
| 13 | 11 | 00 | 0000 | 0110 🚫 | 0010 🚫 | 0010 🚫 |
| 14 | 11 | 01 | 0011 | 1001 🚫 | 0010 🚫 | 0011 |
| 15 | 11 | 10 | 0110 | 0110 | 0110 | 0100 🚫 |
| 16 | 11 | 11 | 1001 | 1001 | 1010 🚫 | 1001 |

From the above table, where $2^n$ possible inputs were tested, we can see that multiple sa-faults causes chaos.

| A | B | C (No Error) | C — B1 sa-1 and Theta sa-0 | C — Alpha + Beta sa-1 and A0B0 sa-0 | C — A1B0 in de sa-1 and beta sa-0 | Total faults, redundant faults | Fault Coverage (cumulative) |
|---|---|---|---|---|---|---|---|
| 00 | 10 | 0000 | 0000 | 0010 🚫 | 0000 | 1, 0 | 33.33% |
| 01 | 01 | 0001 | 0011 🚫 | 0010 🚫 | 0011 🚫 | 3, 1 | 100% |
| 10 | 01 | 0010 | 0010 | 0010 | 0010 | 0, 0 | 100% |
| 11 | 01 | 0011 | 1001 🚫 | 0010 🚫 | 0011 | 2, 2 | 100% |

Randomly generated test patterns

To detect the given faults, test vectors were randomly generated (RTPG) until the faults became uniquely detectable.

All the faults, i.e. faults 1 through 3 are uniquely detectable.

1. [01, 01] — Fault 1 and 3 has detectable error only in line $C_1$, while fault 2 has an error in both $C_0$ and $C_1$.
2. [11, 01] — Fault 1 becomes uniquely detectable, thus fault 3 also becomes detectable uniquely is the output is error free in this test vector.

# Results and references

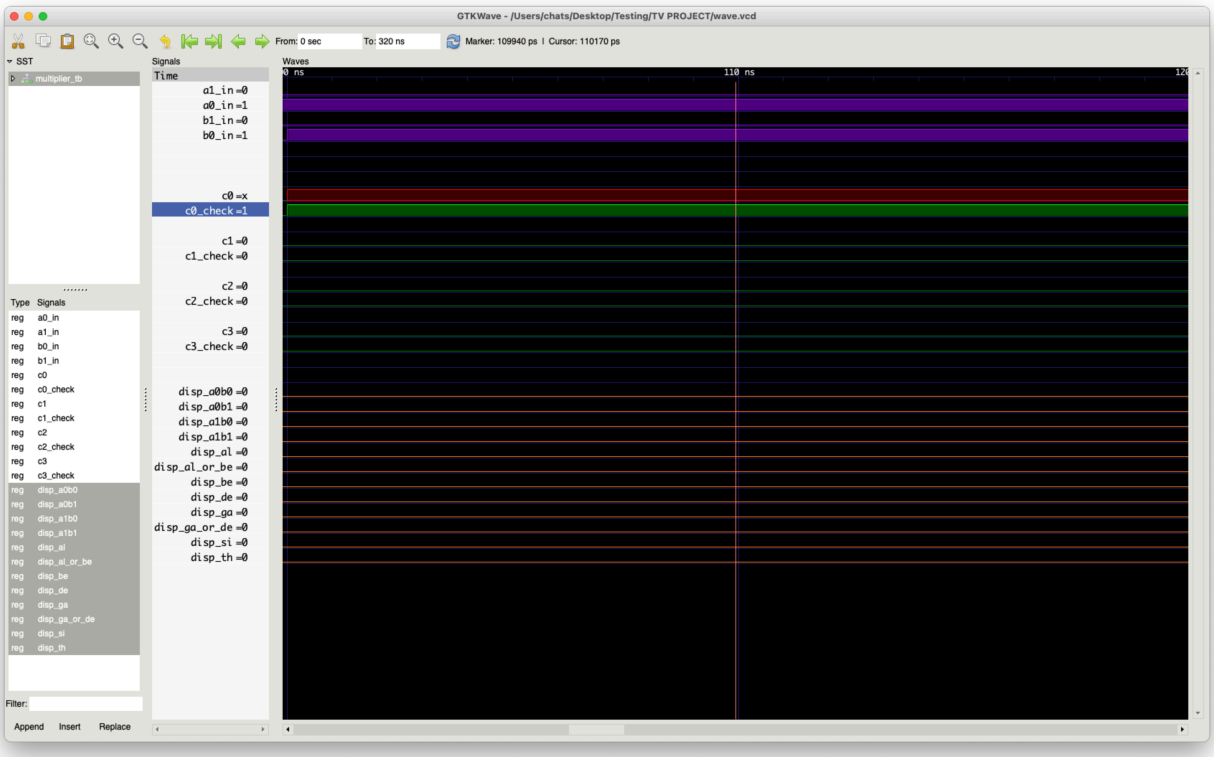
To detect the given faults, different approaches and algorithms were employed due to the fact that resources are limited and checking for all the boolean combinations is exponentially tasking and redundant.

*Single sa-faults test set* : [01,11], [11,01], [10,11]
*Cumulative fault coverage* = 100%
*Number of uniquely detectable faults* = 100%

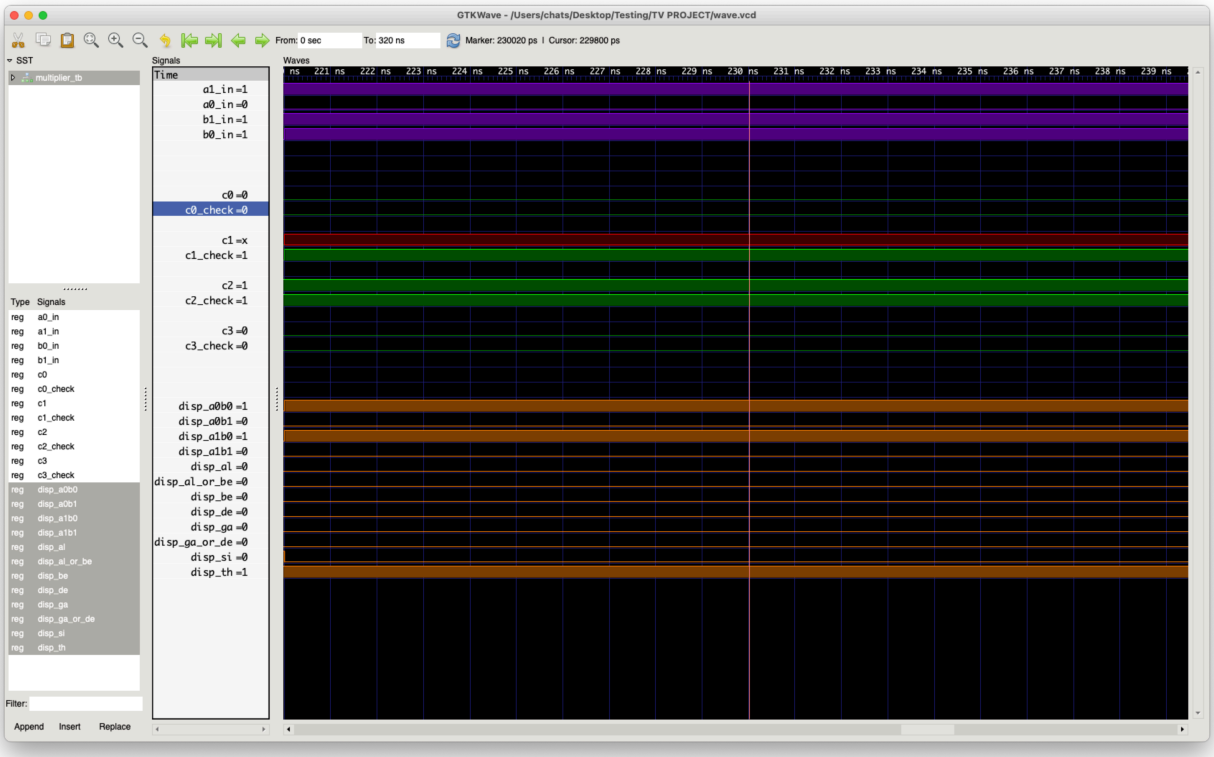
*Multiple sa-faults test set* : [01,01], [11,01]
*Cumulative fault coverage* = 100%
*Number of uniquely detectable faults* = 100%

- It is important to note that while testing for faults in larger circuits, much more sophisticated algorithms like Boolean difference, D-algorithm, PODEM are used.
- The fault coverage is not always 100% for all circuits.
- Testing, timing, verification and validation are the most important and time consuming part of the process of modern silicon designing and manufacturing.

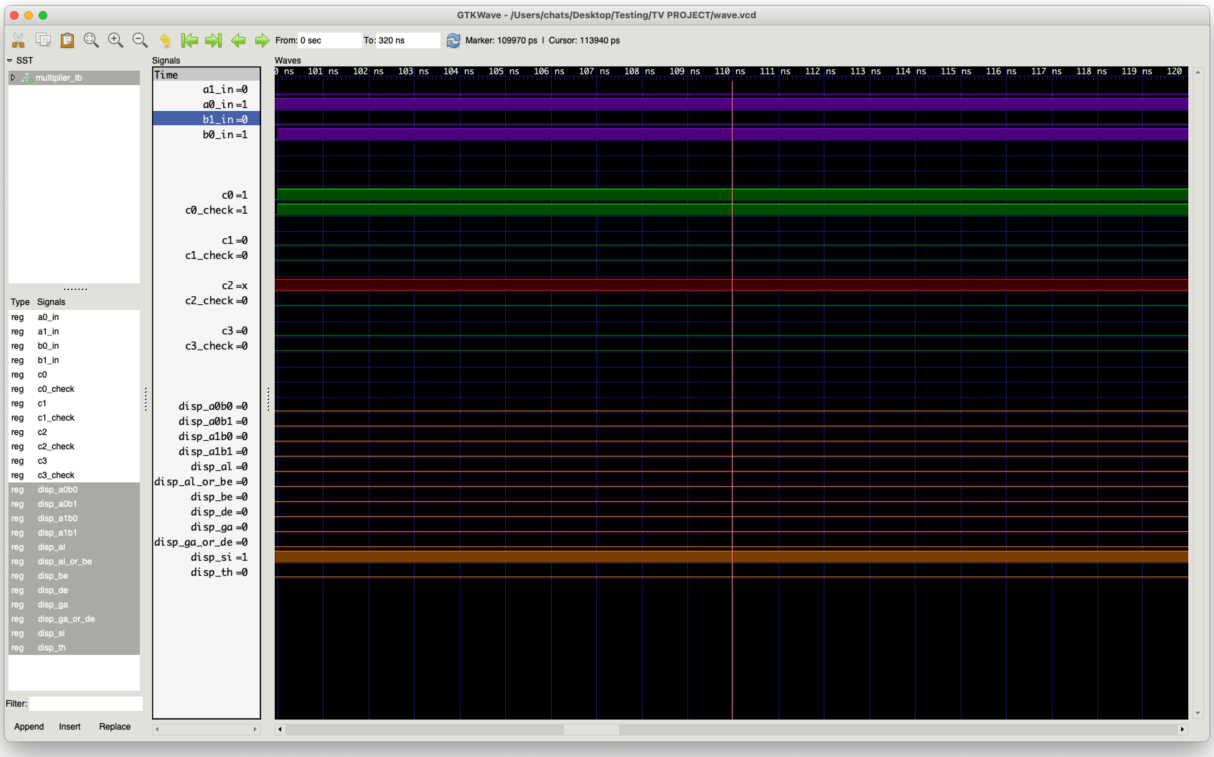
All the truth tables presented in this report are inferred from the waveform generated using the VHDL simulation. All of those waveforms are attached for reference in the following pages.



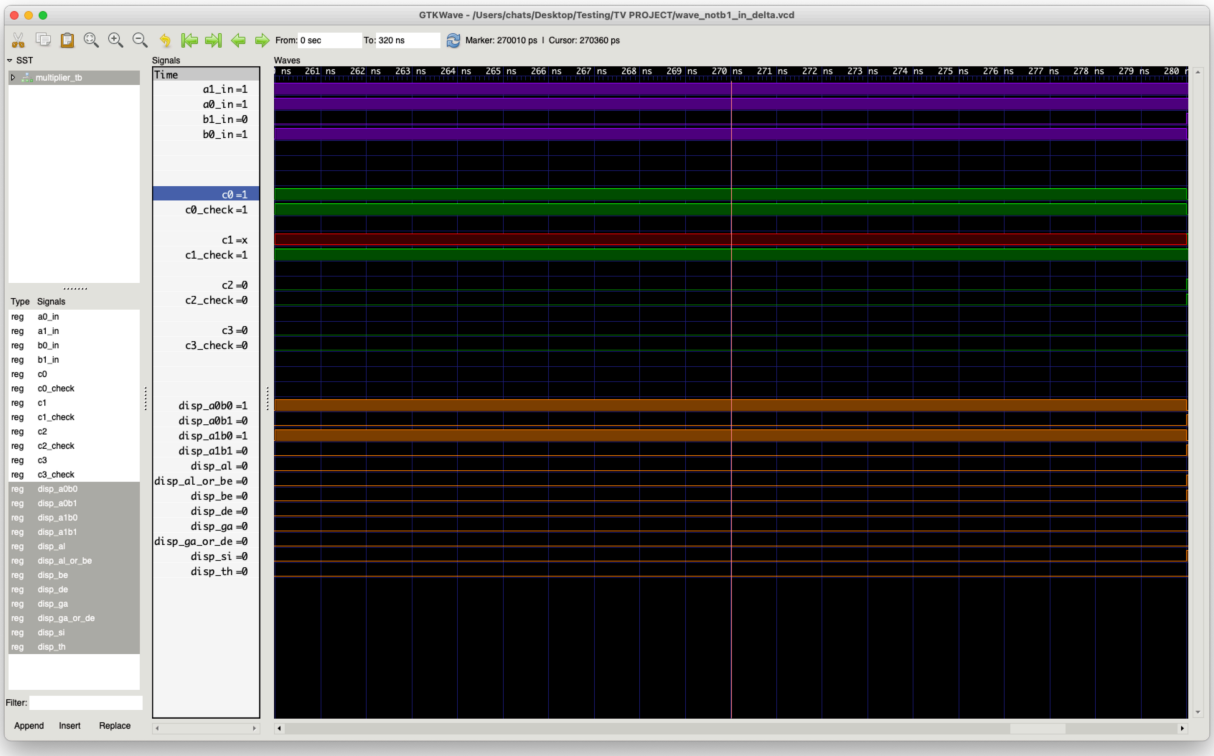

Fault 1: $A_0B_0$ sa-0



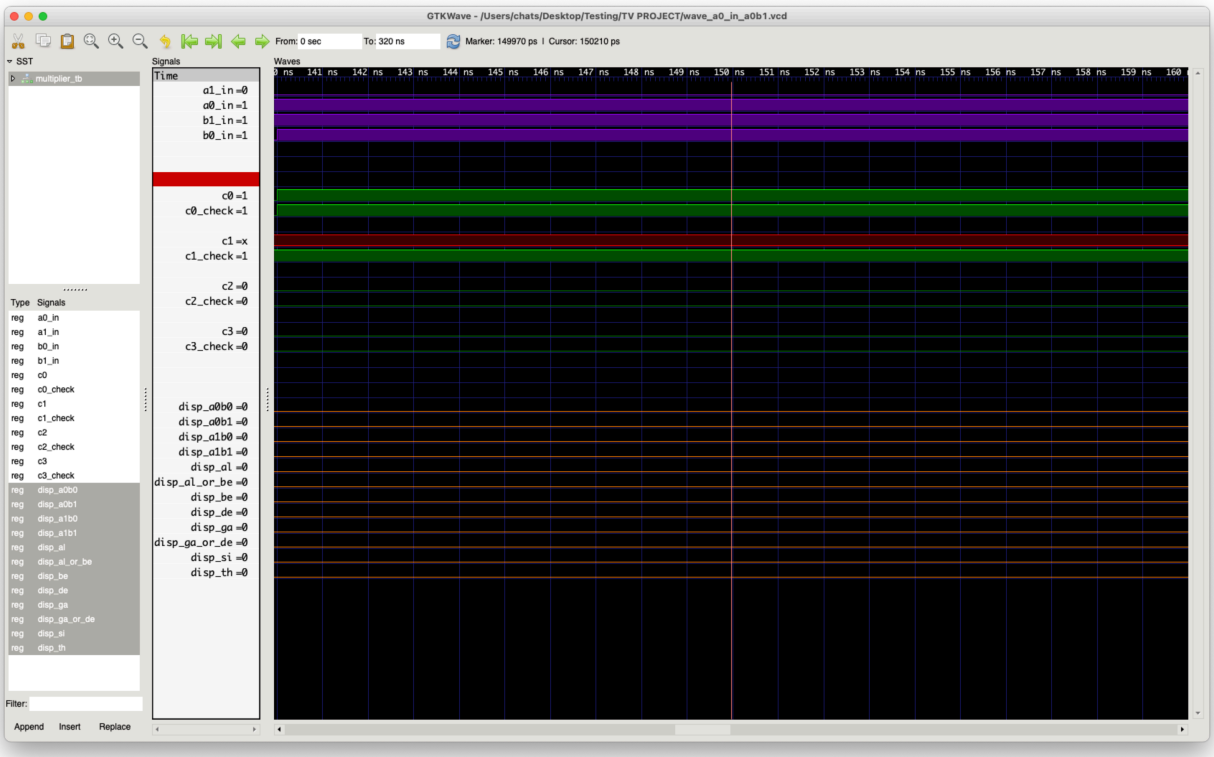

Fault 2: gamma sa-0



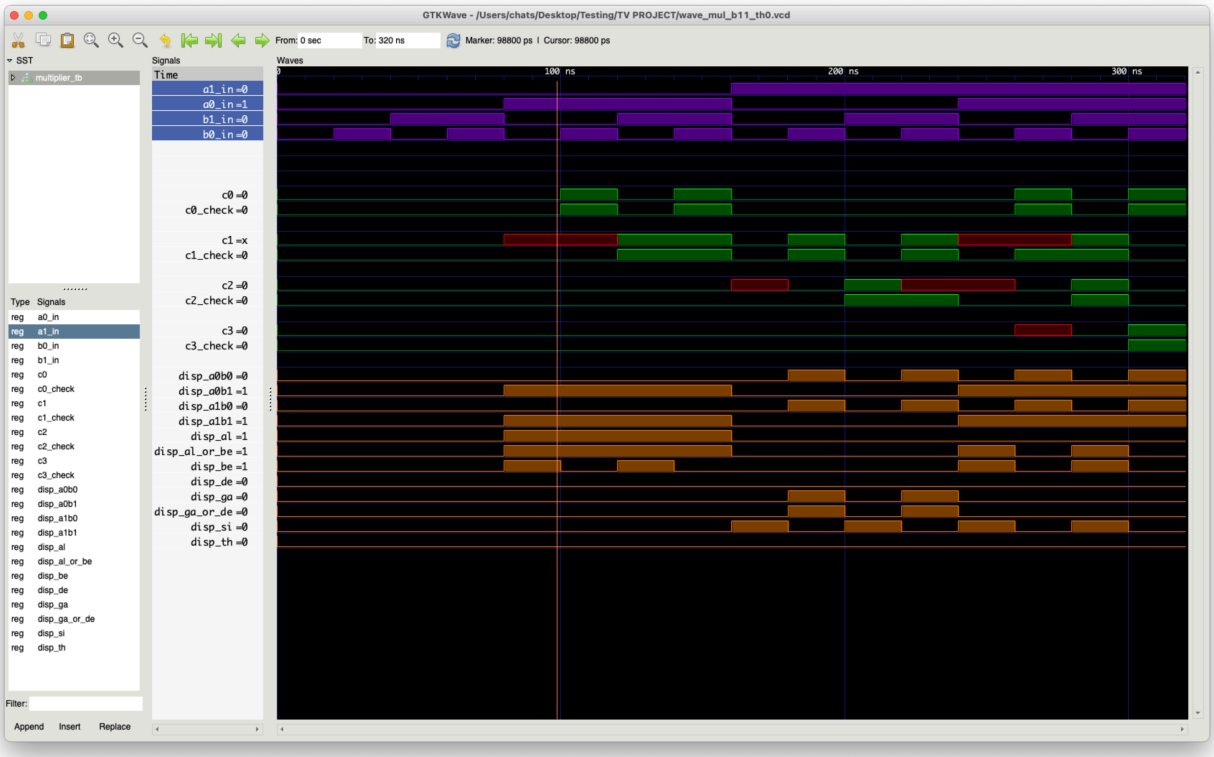

Fault 3: psi sa-1



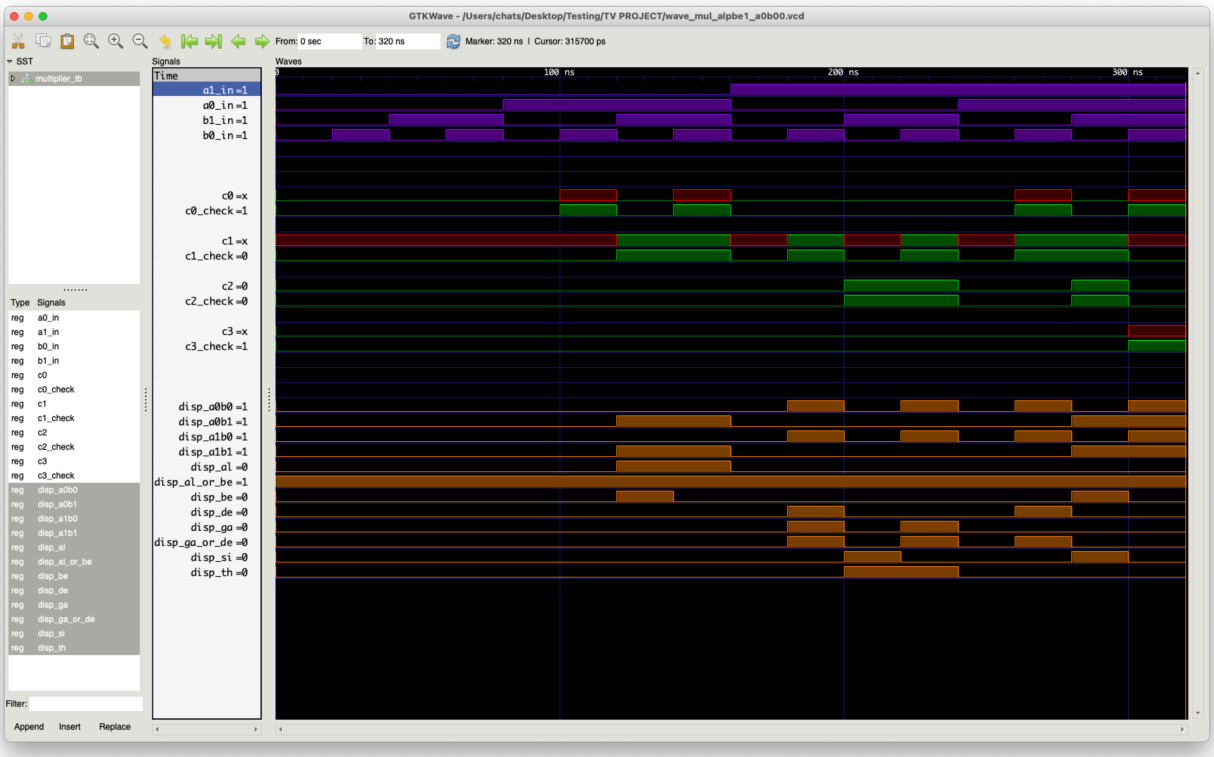

Fault 4: not B1 in delta sa-0



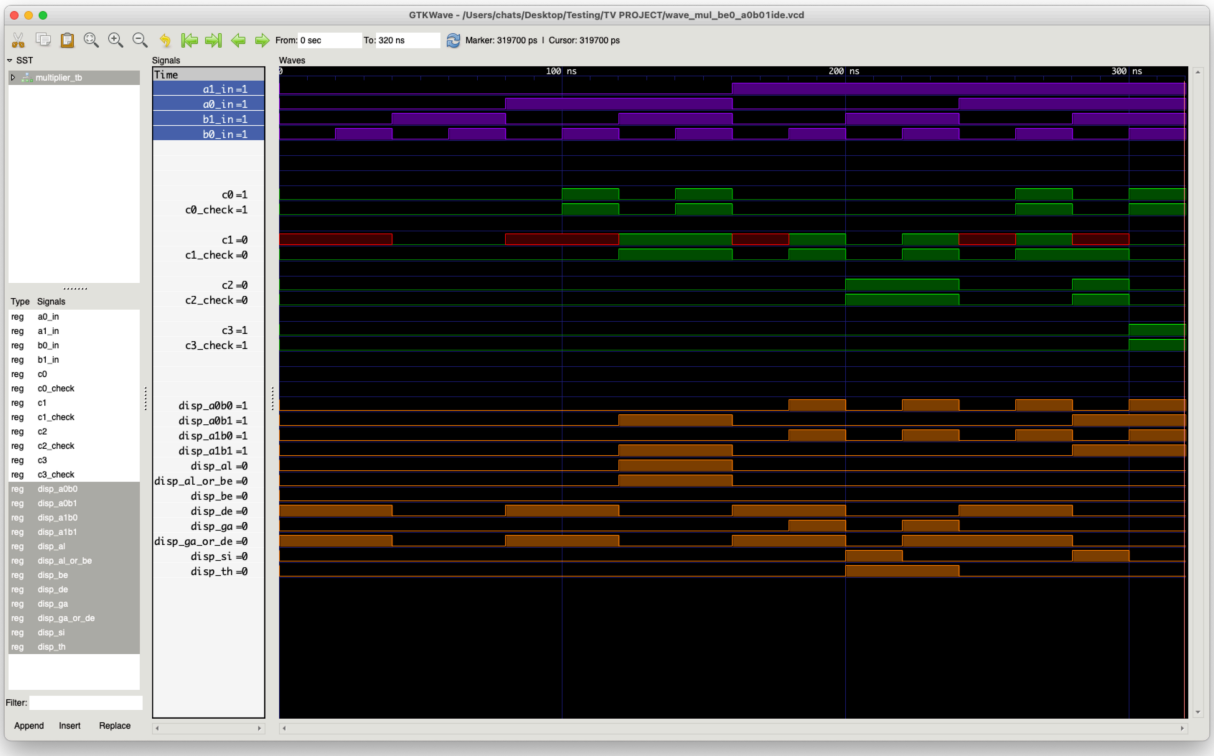

Fault 5: A0 in A0B1




Multiple Fault 1: B1 sa-1 and theta sa-0




Multiple Fault 2: α+β sa-1, and A0B0 sa-0




Multiple Fault 3: A1B0 in delta sa-1, beta sa-0