國立清華大學

超大型積體電路測試

VLSI Testing

國立清華大學

NATIONAL TSING HUA UNIVERSITY

Homework 1

系所級:電子所一年級

學號:111063548

姓名:蕭方凱

指導老師:黃錫瑜教授

目錄

1. Consider the testing of a gate-level circuit as shown below. The primary input signals are {x1, x2, x3} and the primary output signal is {z}. The output signals of logic gates {A, B} are denoted as {a, b}, and the branches of primary input signal x2 is clled y1 and y2, respectively. 3

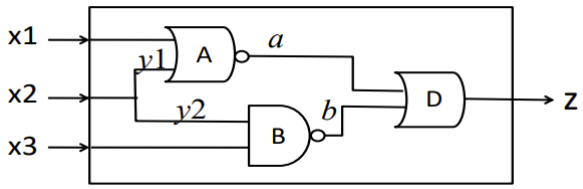
(a) Write a software program (using C, C++, or any other programing language) that can exhaustively simulate the logic behavior of the given circuit under each of the 2 3 = 8 input vectors). Note that this can be done by executing the following Boolean equations in sequence in your program: a = ~(x1 or x2); b = ~(x2 and x3); z = (a or b); List the results as a truth table for output signal z. (Note this truth table contains 8 entries, one for each input combination). 3

(b) Enhance your program so that it can perform exhaustive fault simulation for bridging faults, {AND-bridging between a and b} and {OR-bridging between y1 and x3}. Note that you need to report the total number of possible test patterns for each of the above two bridging faults. (Hint: perform fault injection, run exhaustive simulation on the faulty circuit, and then compare the results with those of the fault-free circuit. The fault injection can be done manually by changing the compiled code.) 4

圖表目錄

FIG 1 PYTHON CODES 3
FIG 2 TRUTH TABLE 4
FIG 3 ENHANCED PART OF CODES 4
FIG 4 TRUTH TABLE OF BRIDGING FAULT 5

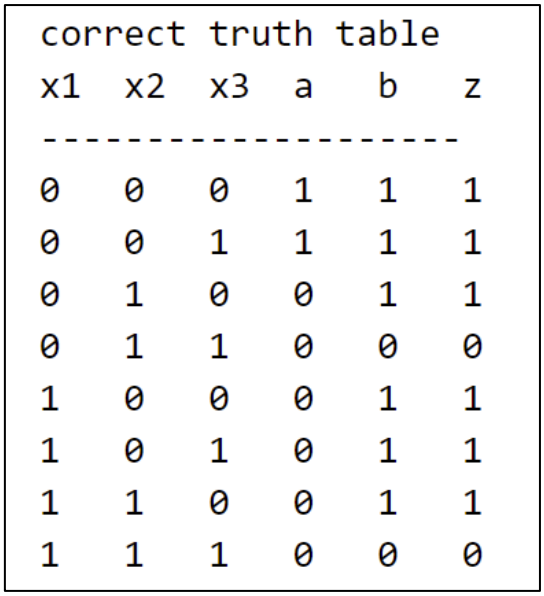
1. Consider the testing of a gate-level circuit as shown below. The primary input signals are {x1, x2, x3} and the primary output signal is {z}. The output signals of logic gates {A, B} are denoted as {a, b}, and the branches of primary input signal x2 is called y1 and y2, respectively.



(a) Write a software program (using C, C++, or any other programing language) that can exhaustively simulate the logic behavior of the given circuit under each of the 2 3 = 8 input vectors). Note that this can be done by executing the following Boolean equations in sequence in your program: a = ~(x1 or x2); b = ~(x2 and x3); z = (a or b); List the results as a truth table for output signal z. (Note this truth table contains 8 entries, one for each input combination).

```
import numpy as np

def circuit(x1, y1, y2, x3):
    a = not (x1 or y1)
    b = not (y2 and x3)
    z = a or b
    return a,b,z

def test_pattern(x1, y1, y2, x3):
    if(x1 == True and y1 == False):
        excepted_a = False
    elif(x1 == False and y1 == True):
        excepted_a = False
    elif(x1 == True and y1 == True):
        excepted_a = False
    elif(x1 == False and y1 == False):
        excepted_a = True

    if(y2 == True and x3 == False):
        excepted_b = True
    elif(y2 == False and x3 == True):
        excepted_b = True
    elif(y2 == True and x3 == True):
        excepted_b = False
    elif(y2 == False and x3 == False):
        excepted_b = True

    return excepted_a,excepted_b

def test_and(x1, y1, y2, x3):
    a = not (x1 or y1)
    b = not (y2 and x3)
    if (a == 1 and b == 1):
        z = True
    else:
        z = False
    return a,b,z

def test_or(x1, y1, y2, x3):
    y1_AB = (y1 or x3)
    a = not (x1 or y1_AB)
    b = not (y2 and y1_AB)
    z = a or b
    return a,b,z

print("correct truth table\t\t\tAND-bridging fault between a and b\t\t\tOR-bridging fault between a and b")
print("x1\tx2\tx3\ta\tb\tz\t\t\tx1\tx2\tx3\ta\tb\tz\t\t\t\t\t\tx1\tx2\tx3\ta\tb\tz")
print("-------------------\t\t\t-------------------\t\t\t\t\t\t-------------------")
```

Fig 1 Python codes



correct truth table

x1	x2	x3	a	b	z
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	0	0	0

Fig 2 Truth Table

(b) Enhance your program so that it can perform exhaustive fault simulation for bridging faults, {AND-bridging between a and b} and {OR-bridging between y1 and x3}. Note that you need to report the total number of possible test patterns for each of the above two bridging faults. (Hint: perform fault injection, run exhaustive simulation on the faulty circuit, and then compare the results with those of the fault-free circuit. The fault injection can be done manually by changing the compiled code.)

```
for x1 in [False, True]:
    for x2 in [False, True]:
        y1 = x2
        y2 = x2
        for x3 in [False, True]:
            correct_result = circuit(x1, y1, y2, x3)
            binary_correct_result = '\t'.join('1' if x else '0' for x in correct_result)
            print(f"{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_correct_result[0]}\t{binary_correct_result[2]}\t{binary_correct_result[4]}\t\t", end = '')

            bridging_fault_and = test_and(x1, y1, y2, x3)
            #print(bridging_fault_and)
            binary_and = '\t'.join('1' if x else '0' for x in bridging_fault_and)
            if (binary_and[0] != binary_correct_result[0]):
                print(f"\t{int(x1)}\t{int(x2)}\t{int(x3)}\t({binary_and[0]})\t{binary_and[2]}\t({binary_and[4]})\t\t", end ='')
            elif (binary_and[2] != binary_correct_result[2]):
                print(f"\t{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_and[0]}\t({binary_and[2]})\t({binary_and[4]})\t\t", end ='')
            elif (binary_and[4] != binary_correct_result[4]):
                print(f"\t{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_and[0]}\t{binary_and[2]}\t({binary_and[4]})\t\t", end ='')
            else:
                print(f"\t{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_and[0]}\t{binary_and[2]}\t{binary_and[4]}\t\t", end ='')


            bridging_fault_or = test_or(x1, y1, y2, x3)
            binary_or = '\t'.join('1' if x else '0' for x in bridging_fault_or)
            if (binary_or[0] != binary_correct_result[0]):
                print(f"\t\t\t\t{int(x1)}\t{int(x2)}\t{int(x3)}\t({binary_or[0]})\t{binary_or[2]}\t{binary_or[4]}\t\t")
            elif (binary_or[2] != binary_correct_result[2]):
                print(f"\t\t\t\t{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_or[0]}\t({binary_or[2]})\t({binary_or[4]})\t\t")
            else:
                print(f"\t\t\t\t{int(x1)}\t{int(x2)}\t{int(x3)}\t{binary_or[0]}\t{binary_or[2]}\t{binary_or[4]}\t\t")
```

Fig 3 Enhanced part of codes

```
correct truth table
x1  x2  x3  a   b   z
------------------
0   0   0   1   1   1
0   0   1   1   1   1
0   1   0   0   1   1
0   1   1   0   0   0
1   0   0   0   1   1
1   0   1   0   1   1
1   1   0   0   1   1
1   1   1   0   0   0
```

```
AND-bridging fault between a and b
x1  x2  x3  a   b   z
------------------
0   0   0   1   1   1
0   0   1   1   1   1
0   1   0   0   1   (0)
0   1   1   0   0   0
1   0   0   0   1   (0)
1   0   1   0   1   (0)
1   1   0   0   1   (0)
1   1   1   0   0   0

AND-bridiging fault num : 4
```

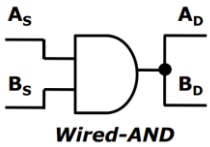
```
OR-bridging fault between a and b
x1  x2  x3  a   b   z
------------------
0   0   0   1   1   1
0   0   1   (0) 1   1
0   1   0   0   (0) (0)
0   1   1   0   0   0
1   0   0   0   1   1
1   0   1   0   1   1
1   1   0   0   (0) (0)
1   1   1   0   0   0

OR-bridiging fault num : 3
```

Fig 4 Truth Table of bridging fault(括號數字代表該值異常)

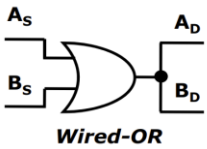
Comment:

Truth Table 從左到右分別為:correct truth table、and bridging fault 的 truth table 及 or bridging fault 的 truth table。數字括號起來代表該值是錯誤的，以下分解釋不同程式碼區塊的功能目的:

1. Def circuit： 將電路用 python 寫出，使 output z 根據輸入的 x1, x2, x3 變化而產生對應的輸出值。
2. Def Test_pattern： 寫一個擁有正確電路行為的 pattern，供電路偵錯時用。
3. Def test_and： 與 def circuit 不同，此副程式的目的是製造 a 和 b 之間的 and bridge。如下圖:



4. Def test_or： 與 def circuit 不同，此副程式的目的是製造 y1 和 x3 之間的 or bridge。如下圖:



5. For 迴圈部分為主要功能，首先套用 def circuit 的正確電路，print 出正確的 correct truth table。接著是偵錯 and bridging fault 的程式碼，套用 test_and 副程式，模擬 a 和 b 後方出現 and gate，進而影響 output z 的正確性。最後是偵錯 or bridging fault，在 def test_or 中畫上一個 y1 or x3 的 or gate，使 a 或 b 不正常輸出，與 correct truth table 比較後，找出錯誤值。

   總結來說，從 Truth Table 觀察 output z 是否正確，若不正確，可檢測 a 和 b 的值，a 和 b 與 correct truth table 相同，代表是 and bridging fault，也就是 a 和 b 後方多出的一個 and gate; a 和 b 與 correct truth table 不同，則代表在電路 Input 訊號處多了一個 y1 or x3 的 or gate。