Brunel UNIVERSITY WEST LONDON

Name: Honnet
Student n°: 0531984

EE2071 Micro electronic workshop:

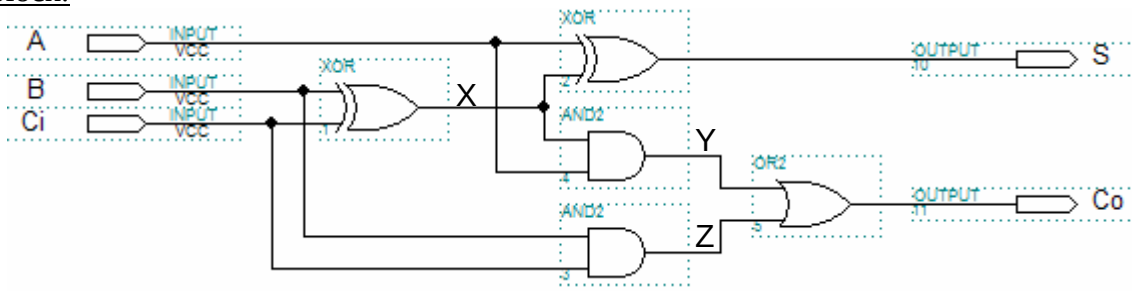
Fault simulations and test pattern generation

Purpose:

The purpose of this assignment is for us to understand the importance and complexity of designing logic to successful test fault. For simple designs, 100% fault cover should be possible, although with larger designs this may be impossible due to the existence of undetectable faults because of redundancy in a circuit.
First of all I am going to try to obtain 100% fault coverage for the simple full adder circuit.

Task 1:

Full adder block:



Exhaustive fault matrix: (I've highlighted the errors in grey and bold)

INPUTS:			OK		A@0		A@1		B@0		B@1		Ci@0		Ci@1		X@0		X@1		Y@0		Y@1		Z@0		Z@1	
A	B	Ci	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S	Co	S
0	0	0	0	0	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	**1**	0	0	0	**1**	0
0	0	1	0	1	0	1	**1**	**0**	0	1	**1**	**0**	0	**0**	0	1	0	**0**	0	**0**	0	1	**1**	1	0	1	**1**	1
0	1	0	0	1	0	1	**1**	**0**	0	**0**	0	1	0	1	**1**	**0**	0	**0**	0	**0**	0	1	**1**	1	0	1	**1**	1
0	1	1	1	0	1	0	1	**1**	**0**	**1**	1	0	**0**	**1**	1	0	1	0	1	0	1	0	1	0	**0**	0	1	0
1	0	0	0	1	0	**0**	0	1	0	1	**1**	**0**	0	1	**1**	**0**	0	1	**1**	1	0	1	**1**	1	**1**	1	**1**	1
1	0	1	1	0	**0**	**1**	1	0	1	0	1	**1**	**0**	**1**	1	0	**0**	**1**	**0**	**1**	**0**	0	1	0	**0**	0	1	0
1	1	0	1	0	**0**	**1**	1	0	**0**	**1**	1	0	1	0	1	**1**	**0**	**1**	**0**	**1**	**0**	0	1	0	**0**	0	1	0
1	1	1	1	1	1	**0**	1	1	1	**0**	1	1	1	**0**	1	1	1	**1**	1	1	1	1	**1**	1	**1**	1	**1**	1

As we can see all the stuck@0 and stuck@1 in the nodes Y and Z don't change the output S. Hopefully these stuck-at faults are detectable with the Co output.

I'm going to simplify this fault matrix, but as my knowledge in Excel is not yet wonderful, I'll do it simply.

♣ *Fault matrix for the output **Co:*** (I've marked the errors by " x " and deleted the rest)

INPUTS:			OK	A@0	A@1	B@0	B@1	Ci@0	Ci@1	X@0	X@1	Y@0	Y@1	Z@0	Z@1
A	B	Ci	Co	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
0	0	0	0										X		X
0	0	1	0		X		X						X		X
0	1	0	0		X				X				X		X
0	1	1	1			X		X						X	
1	0	0	0				X		X		X		X	X	X
1	0	1	1	X				X		X	X	X		X	
1	1	0	1	X		X				X	X	X		X	
1	1	1	1										X	X	X

Here I've just kept all the data concerning the output Co.

Simplification for the output **Co**:

I've used the Quine-McKluskey technique equivalent to simplify the previous fault matrix.

	A	B	Ci	Co	C1,7,9	C2	C3	**C4**	C5	**C6**	**C8**	**C10,12**	**C11**
V1	0	0	0	0								X	
V2	0	0	1	0		X		X				X	
V3	0	1	0	0		X				X		X	
V4	0	1	1	1			X		X				X
V5	1	0	0	0				X		X	X	X	X
V6	1	0	1	1	X				X		X		X
V7	1	1	0	1	X		X				X		X
V8	1	1	1	1								X	X

The vector V5 can detect the faults **C4**, **C6**, **C8**, **C10**, **C11** and **C12**.
The vector V3 can detect C2, other faults detectable being already detected.
The vector V4 can detect C3 and C5 (other faults detectable being already detected).
And the vector V6 can detect C1, C7 and C9 (other faults detectable being already detected).

These 4 vectors are sufficient to detect all the faults

♣ *Fault matrix for the output **S:***

Now I've just kept all the data concerning the output S:

INPUTS:			OK	A@0	A@1	B@0	B@1	Ci@0	Ci@1	X@0	X@1	Y@0	Y@1	Z@0	Z@1
A	B	Ci	S	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
0	0	0	0		X		X		X						
0	0	1	1		X		X	X		X	X				
0	1	0	1		X	X			X	X	X				
0	1	1	0		X	X		X							
1	0	0	1	X			X		X						
1	0	1	0	X			X	X		X	X				
1	1	0	0	X		X			X	X	X				
1	1	1	1	X		X		X							

As I said previously, the stuck@0 and stuck@1 in the nodes Y and Z don't change the output S.

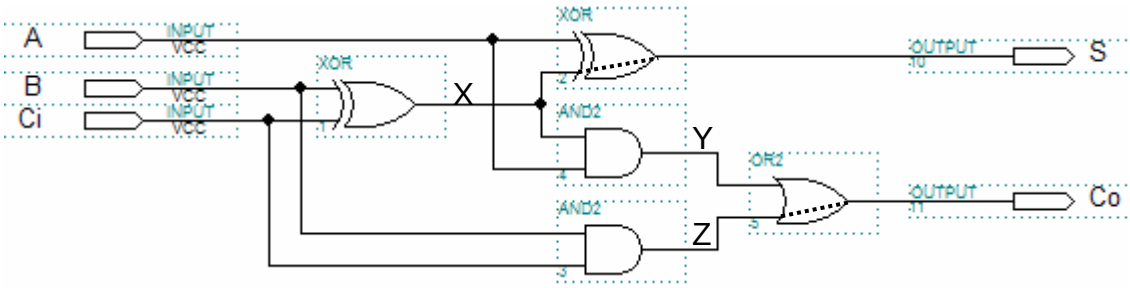
Simplification for the output **S**: (same process)

	A	B	Ci	S	S1	S2	S3	S4	S5	S6	S7,8	S9,10,11,12
V1	0	0	0	0		X		X		X		
V2	0	0	1	1		X		X	X		X	
V3	0	1	0	1		X	X			X	X	
V4	0	1	1	0		X	X		X			
V5	1	0	0	1	X			X		X		
V6	1	0	1	0	X			X	X		X	
V7	1	1	0	0	X		X			X	X	
V8	1	1	1	1	X		X		X			

The vector **V3** can detect the faults S2, S3, S6, S7 and **V6** can detect S1, S4, S5.
These 2 vectors are sufficient to detect all the faults, and that's perfect because we are already going to use V3, V4, V5 and V6 to monitor faults on the output Co.

♣ We can conclude that we just need the vectors ***V3, V4, V5*** and ***V6*** to detect if the full adder is faulty.

Last minute modification: SCAN PATH SENSITIZATION



α	β	$\alpha \oplus \beta$
0	0	0
0	1	1
1	0	1
1	1	0
0/D	**D/0**	**D**
1/!D	**!D/1**	**D**
Note: "**!D**" = "**not D**"		

If we want to test the stuck@fault in the wire X, we need to "open up" the good path. We thus have to select special values to be able to see them effect in output.

- As the "xor" table show, if we have A = 0 and X = D, then in the output S we'll have the same value **S = A ⊕ X = 0 ⊕ D = D**.

- But to obtain this value D in the wire X, we need to choose between {B,Ci} = {!D,1} and {B,Ci} = {1,!D} => **X = B ⊕ Ci = 1 ⊕ !D = D**.

- As we need either B =1 or Ci = 1 (the other being !D) thus **Z = B × Ci = 1 × !D = !D**. Then as Z = !D and Y = 0 thus **Co = Z + Y = !D + 0 = !D**

Note: "×" designate the AND; "+" designate the OR; "⊕" designate the XOR.

=> We thus obtain the Test vector **T1** in the following table:

	A	B	Ci	X	Y	Z	Co	S
	0			D	0			D
		!D/1	1/!D	D		!D	!D	
T1	**0**	**!D/1**	**1/!D**	**D**	**0**	**!D**	**!D**	**D**
					D	0	D	
	D	**0/1**	**1/0**	1				
T2	**D**	**0/1**	**1/0**	**1**	**D**	**0**	**D**	

But the path sensitization is not complete with the vector **T1** because a stuck@fault in the wire Y is not detectable.

We thus need to create another test vector **T2** and we can process with the same technique:

We want to see what's happening in Y then we need Z = 0 to have **Co = D + 0 = D**.

With A = D, to obtain Y = D we need X = 1 (**Y = 1 × D = D**) then we need B ≠ Ci => **X = 0 ⊕ 1 = 1 ⊕ 0 =1**.
With these values, we finally obtain **S = Y + Z = 0 + D = D.**

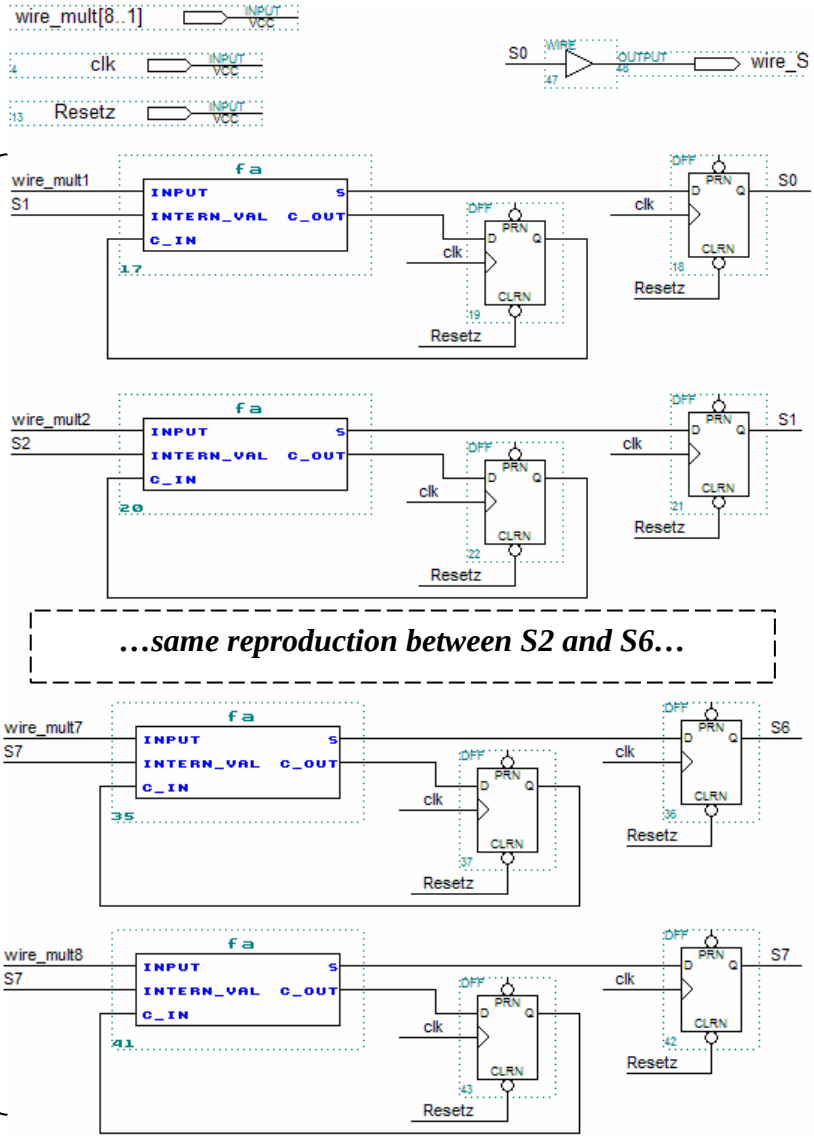
Note: I didn't draw the path for the vector **T2** but the principle is the same.

Task 2:

♣ In this task we are going to demonstrate how the test set produced in task 1 can be applied to a full adder embedded in the final multiplier. As we have seen in task 1, we can detect if the full adder is faulty with 4 test vectors. The systolic multiplier contains 8 full adders (fa). Each of them is connected similarly except the last one that feeds back the MSB:

As we can see, the pins A and B are called "INPUT" and "INTERNAL_VAL" but the full adder is obviously the same.

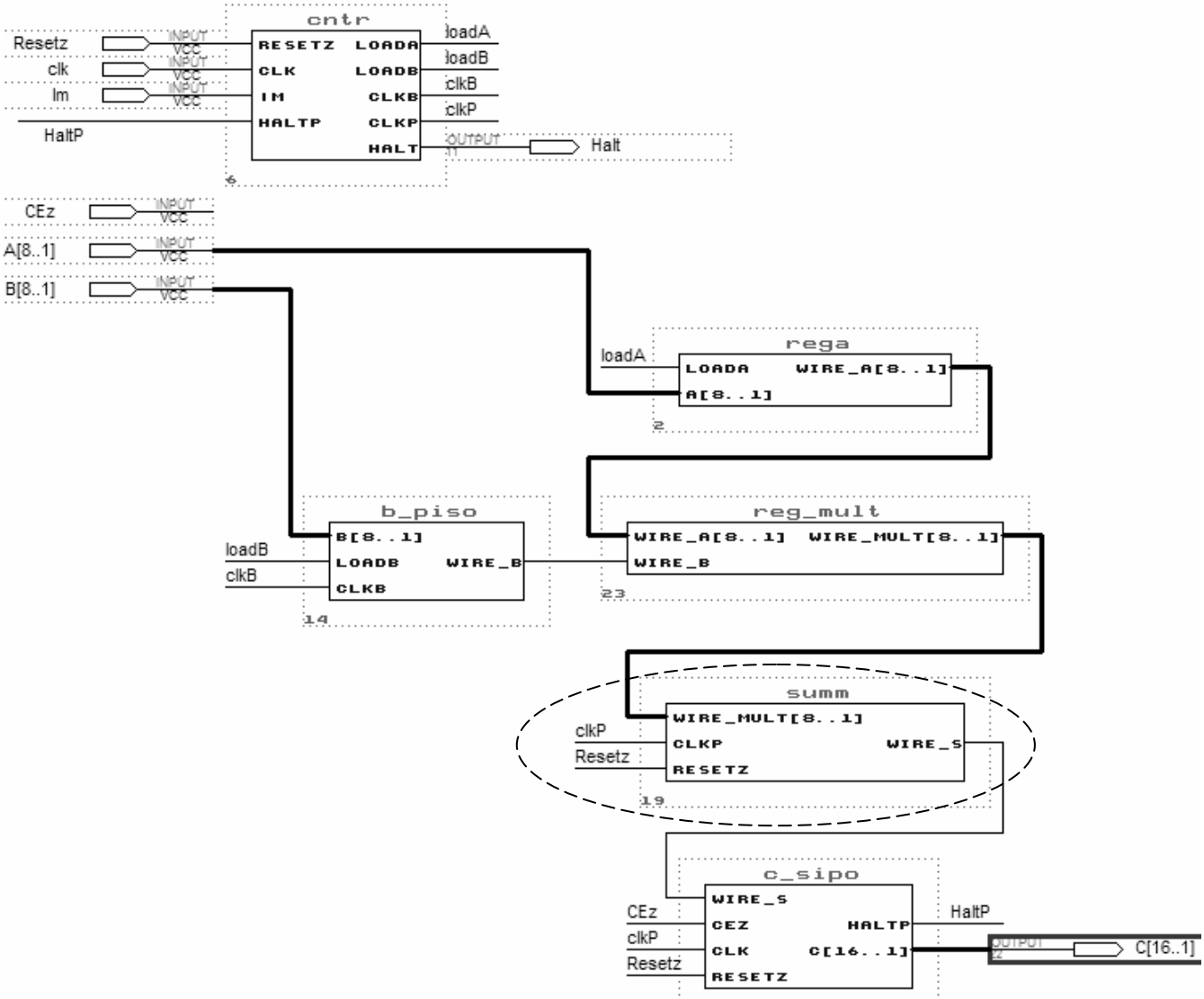
We normally have 8 blocs, but we just show the 2 least and most significant.



The test becomes really harder because C_out is fed back in the C_in (with a D flip flop on the way) and S is injected in the next full adder (also with a D flip flop on its way!)

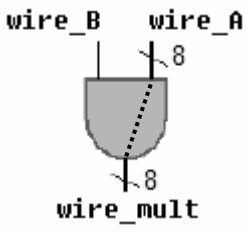
The problem is not finished; we control just 1 input of each full adder and only 1 output of the 8 blocks! We thus have to generate a sequence of vectors to monitor the result in the output.

For that we will assume that the output of the component summ, called wire_S, will be observable (if not, it's not a nightmare to test, it's a full day in hell).



Other delicate point, before arriving in input of summ, this sequence of vectors has to pass by the register A (regA) and to be multiplied by the LSB of the regiser B (b_piso) ! ! !

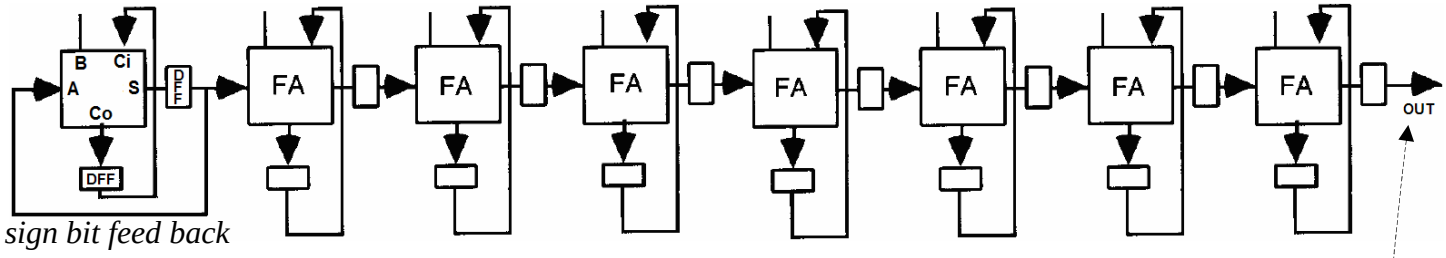
Hopefully, the path sensitisation is not really hard to manage, the multiplication part is done by a kind of AND gate which input are wire_B on 1bit and wire_A on 8bits:



We just need to load 11111111 in the register B thus the value in wire_B will always be 1 which is the neutral element of the logical AND.

This component will thus finally behave like an 8 bits bus with wire_A in input and wire_mult in output.

By keeping the previous discussion in mind, we can finally consider that we just need to test the component described in the following diagram. We just need to pulse the loadA signal to allow the register A releasing its internal value to the next module: (the intermediate AND gate being "path-sensitized")



We now have to think about sequences that can reproduce the 4 test vectors found in task 1:

	A	B	Ci
V3	0	1	0
V4	0	1	1
V5	1	0	0
V6	1	0	1

The sequence to get the test vector V3 is:

- load 0x00, clock the dff. => all the latch inputs are thus set at 0.
- load 0x00, clock the dff. => all the latch outputs are thus set at 0 then Ci = A = 0.
- load 0xFF. => all the inputs B are at 1 (with Ci = A = 0), the test vector [010] is thus sent.

The result in S must be 0+1+0 = **1** if no fault, we just need to load 0x00 and clock 8 times.
The next 8 **OUT** bits should thus be **1** which is the 8 results S of each addition in each cell.

The sequence to get the test vector V4, V5 and V6 can be obtained similarly; I thus won't treat them.

♣ This module is hard to test compared to the highly testable register A or the "big AND gate".
But it's nothing compared to the register B in witch the own internal clock generator is an headache, the 8 PISO dff (that can be considered as combinatorial), the multiplexers (that are also combinatorial)… or the register C that has tristate gates, 16 SIPO dff with a set for the MSB and reset otherwise…

It appears quite obvious after this task, that if a component is sequential its test can really be hard, but if it is combinatorial (without too much logic feedbacks) the testability becomes easily really high.

Task 3:

♣ In this task we are going to try to generate a test strategy plan for our multiplier design. We will discuss possible standard designs for testability methods that could be applied to the multiplier to make it more testable. As assumed in task 2, to make the multiplier more testable, we need more observability.
This observability can be increased by with several techniques:
The simplest one is to add outputs to monitor our internal components; but if we have a 64 bits multiplication (like on my computer), the result has to be on 128 bits then internal registers of this multiplier can be on 128 bits as well and several others on 64. That's not feasible, the power consumption would be terrible.
To give an example, the *.rpt file created by Max+plus after compilation of my gate level multiplier reports a quantity of 35 pins used over 48 (the 2 others being probably for the power supply):

Logic Array Block	Logic Cells	I/O Pins	Shareable Expanders	External Interconnect
A: LC1 - LC16	4/16(25%)	12/12(100%)	1/16(6%)	6/36(16%)
B: LC17 - LC32	16/16(100%)	8/12(66%)	3/16(18%)	28/36(77%)
C: LC33 - LC48	16/16(100%)	4/12(33%)	12/16(75%)	26/36(72%)
D: LC49 - LC64	16/16(100%)	11/12(91%)	4/16(25%)	20/36(55%)

Total dedicated input pins used:	2/4	(50%)
Total I/O pins used:	35/48	(72%)
Total logic cells used:	52/64	(81%)
Total shareable expanders used:	4/64	(6%)
Total Turbo logic cells used:	52/64	(81%)
Total shareable expanders not available (n/a):	16/64	(25%)
Average fan-in:	5.11	
Total fan-in:	266	
Total input pins required:	20	
Total output pins required:	17	
Total bidirectional pins required:	0	
Total logic cells required:	52	
Total flipflops required:	50	
Total product terms required:	180	
Total logic cells lending parallel expanders:	0	
Total shareable expanders in database:	3	

But as we have just used 50 storage elements and we could think to another technique:

Scan Path

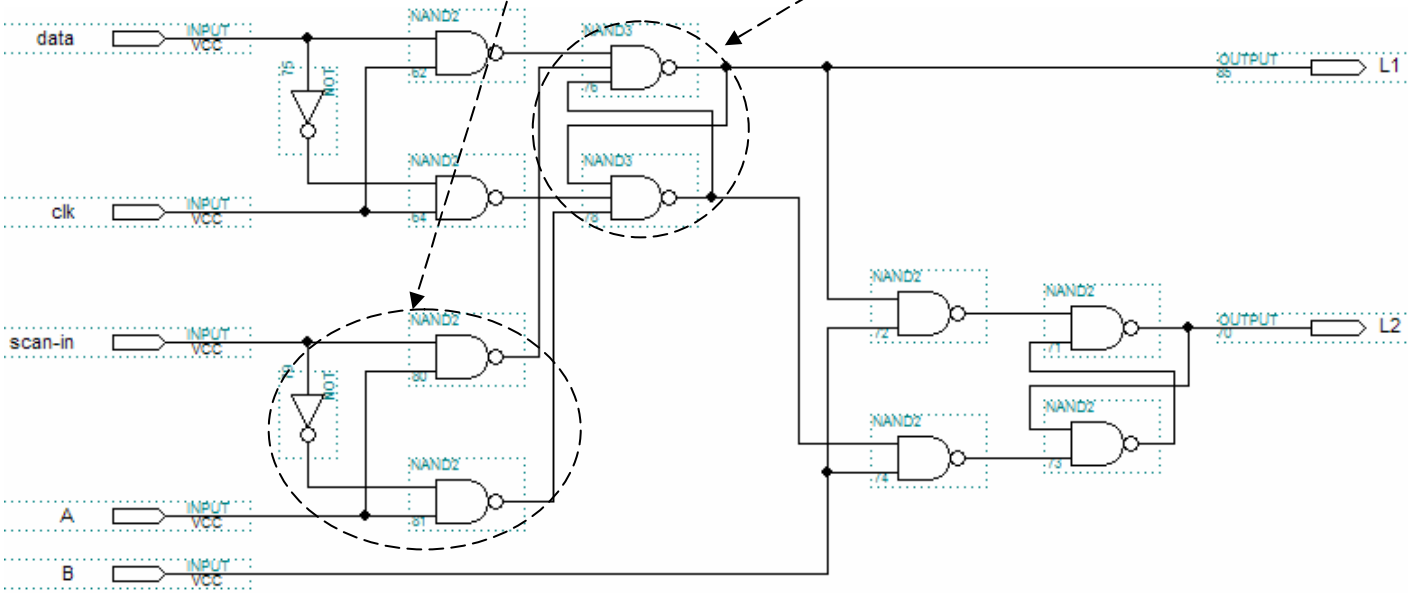
The principle of the scan path is to slightly modify a sequential circuit to test it, into a combinatorial circuit. As a sequential circuit is based on a combinatorial circuit and some storage elements (also combinatorial from another point of view), the scan path consists in connecting together all the modified storage elements to form a long serial shift register with 2 different purposes. The fist part allows initialising the combinatorial circuit to monitor its output using the second part. The internal state of the circuit can thus be observed and controlled by shifting (scanning) the contents of the storage elements.
The shift register is then called a scan path, its storage elements being a kind of multiplexed flip flop, that can be called MDFF or LSSD latch (for level-sensitive scan design).
The sequential circuit containing a scan path has two modes of operation: a normal mode and a test mode which configure the storage elements in the scan path.

-In the normal mode, the storage elements are connected to the combinatorial circuit, in the loops of the global sequential circuit, which is considered then as a finite state machine.

-In the test mode, the loops are broken and the storage elements are connected together as a serial shift register (scan path), receiving the same clock signal. The input of the scan path is called scan-in and the output scan-out (or L2 in the next diagram). Several scan paths can be implemented in one same complex circuit if it is necessary, though having several scan-in inputs and scan-out outputs.

As I said previously, we have just used 50 storage elements and we don't need LSSD latches for 100% of them. As a multiplexer is made with 2"and gate" + 1"or gate" (+1 inverter) we can realise a simple scan paths by just using 4 other I/O pins per register tested and 3 other gates per MDFF. It seems that the LSSD latch is not that much far of a DFF, differences: 2 NAND + 1 INVERTER + 2 NAND **3** inputs instead of NAND **2** inputs

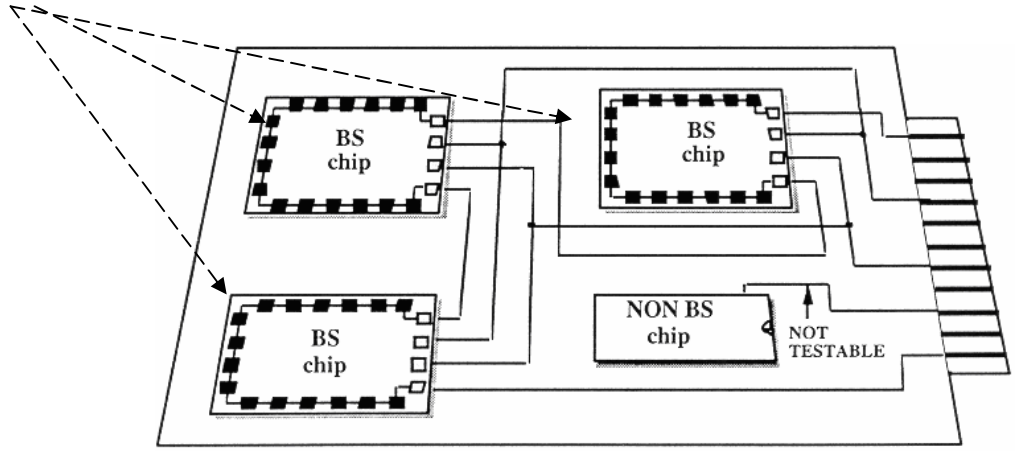
LSSD latch:



Boundary Scan

Boundary Scan Test (BST) is a technique involving scan path and self-testing techniques to resolve the problem of testing boards carrying FPGA or VLSI integrated circuits... Printed circuit boards (PCB) are very dense and complex, that most test equipment cannot guarantee good fault coverage.

BST consists in placing a scan path adjacent to each component pin and to interconnect the cells in order to form a **chain** around the border of the circuit.



The BST circuits contained on a board are then connected together to form a single path through the board (require less I/O that my previous idea, but it's more complex to use). The boundary scan path is provided with serial input and output pads and appropriate clock pads which make it possible to:

-Test the interconnections between the various chip
-Deliver test data to the chips on board for self-testing
-Test the chips themselves with internal self-test

The advantages of Boundary scan techniques are that there is no need for complex testers in PCB testing, the time to spend on test pattern generation is less, the fault coverage is increased…

This efficient solution, commonly used in the industry, can obviously be a great help for our multiplier test. I wouldn’t use it by time lack for the design, but obviously, I'll use it later.

Task 4:

In this task we are going to try to demonstrate the use of a LSSD latch. The Level-Sensitive Scan Design technique was developed and pioneered by IBM, and forms the basis for a structured approach to the design of testable circuits. This component requires following at least 4 rules to use it correctly:

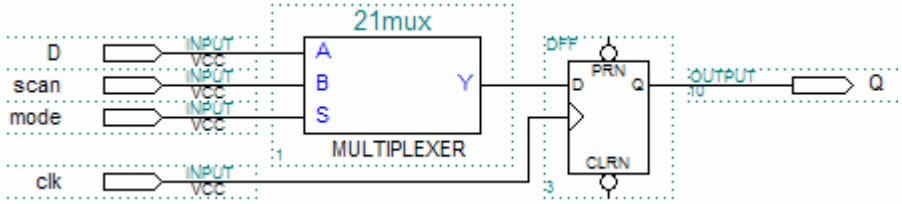
Rule 1: All internal storage is implemented in hazard-free polarity-hold latches.

Rule 2: The latches are controlled by two or more non-overlapping clocks such that latches that feed one another can not have the same clock.

Rule 3: It must be possible to identify a set of clock primary inputs from which the clock inputs to SRLs are controlled either through simple powering trees or through logic that is gated by SRLs and/or non-clock primary inputs.

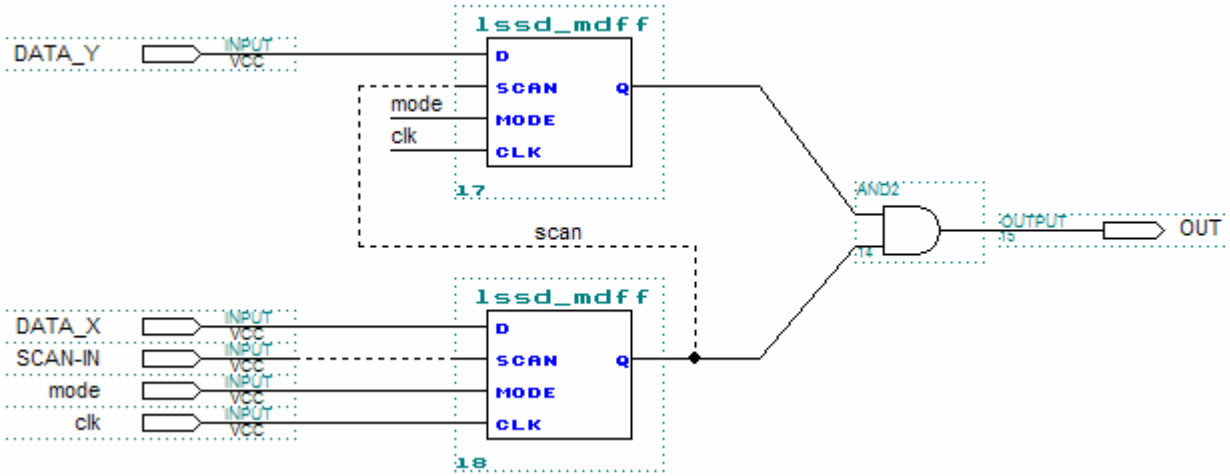
Rule 4: Clock primary inputs may not feed the data inputs to latches either directly or through combinational logic, but may only feed the clock input to the latches or the primary outputs.

As my experience in this domain is not yet phenomenal, I'll use a simplified version of this component:



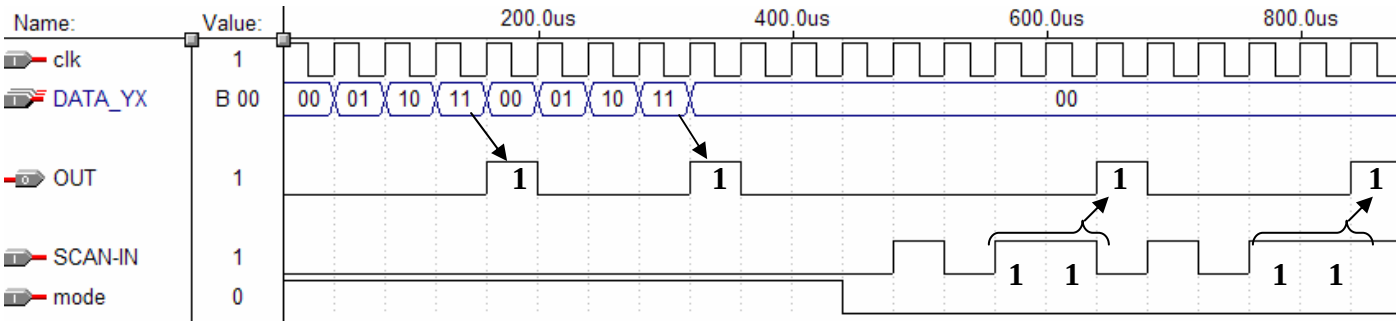
This component is nothing but MDFF, the principle being the same but simplified. I know that this version is not the IBM one as required and there is time problems for this component (that are not occurring in the IBM version) but the time allowed for this report was seriously not realist.

To show the use of this circuit, I've "tested" an AND gate:



I know this circuit is really too much simplified, but the principle is shown.
If mode = 1 (normal mode) the input D is chosen, else SCAN is chosen (test mode).
I'm going to test the circuit firstly in normal mode, I'll use all combinations possible for the input values of X and Y; then I'll pass in test mode, I'll inject serially 2 times the sequence "01011".
The 01 in the beginning, are to test 0 AND 1, then recycling the "1" I'll test 1 AND 0.
=> 2 clk periods later, the injected values will thus be 11 and, as for all this test, the result will occur 1 clk period after to give 1.

Chronogram



The results expected are thus obtained.

Advantages:
With LSSD, the testing problem is transformed from one of sequential circuit testing to one of combinational circuit testing.
By adding controllability/observability to the state variables, LSSD also eases functional testing

Disadvantages:
Additional area is required to fabricate the LSSD latches (area overhead).
Additional time is required to latch the next state into the LSSD registers (speed overhead).
Additional time is required to scan in/out test vectors and responses - at-speed testing is not supported (testing overhead).
Clock generation and distribution for LSSD is more difficult.

The major advantage of LSSD is that it transforms the testing problem from sequential to combinational testing, which is a much more tractable problem.
The major disadvantage of LSSD is probably the speed overhead because it adds several gate delays to the critical path of the design. The testing overhead can be a big problem too because some ASIC vendors charge by the clock cycle for test application.

Conclusion:

This lab report made me generate test patterns for a small combinatorial circuit block and I've got to admit that it's not as easy as it seemed to me. I also learn to identify hard to test and non testable faults, mainly function of the sequential nature of a component. I've thus investigated test strategies and generated it for my multiplier design. And finally, I've implemented a scan path testing method on a tiny combinatorial circuit.
I'm really happy to have learned all of this, I'm sincere, but it took a big part of my revision period (not really fair) and in better condition I would obviously have investigated more and simplify less…