Fault Insertion using IEEE1149.1

Silicon implementation and tool support.

Ken Filliter: National Semiconductor Ken.Filliter@nsc.com
Pete Collins: JTAG Technologies petec@jtag.co.uk

High availability systems often include fail-over mechanisms that continually monitor system operation. In the event of a hardware fault, the monitoring software will detect the fault, diagnose the possible causes and select appropriate redundant or alternate hardware to ensure minimal disruption of operation. As systems become more complex verification of these platform management systems becomes more challenging. Fault Insertion is a technique used to debug and verify these tools and can be accomplished in hardware or software. Ideally inserted faults should be random and sufficient in number for high fault coverage. Conventional hardware approaches have the drawback of adding additional circuitry and generally a small set of potential faults. Software approaches to simulate faults may not completely model faulty hardware.

Using Boundary Scan as an insertion mechanism offers the advantage of being non-intrusive and potentially pervasive throughout the hardware. This paper describes implementation of stuck-at fault insertion in a commercial device as well as the description in bsdl and considerations on how the circuitry could be activated using commercial boundary scan tools.

Introduction

For some electronic hardware there is no tolerance for downtime and the concept of system failure is unacceptable. These high-availability systems are designed to be fault tolerant, where redundant hardware and back-up systems are available to take over operation in the event of a failure. Mission critical systems in Avionics and Space flight are designed this way as well as some medical and industrial hardware where a hard failure would be catastrophic. The Telecommunications Industry has the famous “five 9s” reliability requirement, where acceptable downtime is measured in a few minutes per year.

To achieve these levels of reliability requires suitable hardware design to eliminate any single point of failure and sufficient redundant hardware to allow back-up systems to assume control in the event of a failure. In addition, sophisticated diagnostic software must be able to monitor the system operation and be able to identify hardware failures, diagnose the likely causes of the problem and then select the appropriate corrective action. The verification and debug of this diagnostic software can be costly and time consuming. A useful tool in the verification process is fault insertion, sometimes referred to as “fault injection”.

Fault Insertion can either be accomplished in hardware or simulated in software. Using software to insert faults is not ideal. Whatever assumptions and constraints were made in the diagnostic software are likely also present in the test software. An

independent (hardware) test would therefore be preferable. Conventional hardware fault insertion requires the manual addition of switches or jumpers to create temporary faults. There may not be physical access to many nodes on the board and the additional circuitry must not compromise the functionality of the mission circuit. For these reasons generally only a small number of fault insertion locations can be added.

Boundary scan (IEEE1149.1) has been proposed as an attractive alternative. Boundary scan has the advantages of being non-intrusive and providing access to numerous nodes throughout the system. Potentially boundary scan could provide extensive insertion capability without the need for any additional hardware. But for practical purposes to be useful, the boundary scan fault insertion approach must be clearly described in bsdl (boundary-scan-description-language) and readily assimilated by 3rd party tools and users.

Review of Prior Work

Wilcox et al [1] describes a boundary scan implementation that adds an additional register to the boundary scan chain for each fault capable pin. This register is used to store the go or no-go flag on whether that pin will be faulted. The output of this register is then sent to a multiplexer with the Select signal used to control whether the core data or the boundary scan value is sent to the output pin. When a FIRE (Fault-Insertion-Register-Enable) instruction is executed, the mux selects the go/no-go flag to determine if that pin will be faulted. The actual fault value for the pin is loaded into the regular boundary scan register.

Wilcox's scheme offers many advantages. The core signals are only required to be routed through one multiplexer as with a conventional boundary scan cell. While the device is in normal operation each and every pin with this capability can be faulted high or low, and single or multiple faults inserted. One drawback to this approach is that additional cells are required in the boundary scan chain, possibly adding shift time during long test and programming sequences.

Nadeau-Dostie et al [2] build on the approach described in [1] and add the concept of sticky faults. As they describe, in the event that after inserting faults into a system, structural testing would then be used to locate these faults, the initial use of EXTEST would clear all inserted faults. They propose an additional constraint on the TAP signal UPDATE such that for pins with asserted faults, the pins will stay faulted until the FIRE Instruction is turned OFF.

While the previous two papers mentioned were concerned with ASICs, Chakraborty et al [3] propose a solution for FPGAs using existing user-defined instructions. The approach described offers the same capability of inserting single or multiple stuck-at high or low faults during operation. The drawback to the approach in [3] is that an additional mux delay is inserted in the signal path.

When considering the advantages of adding fault insertion versus the drawbacks, added die area, additional scan-path length, added mux delay, all three authors make the point that fault insertion capability need only be added

on pins where this is desired, thereby minimizing the cost of implementation.

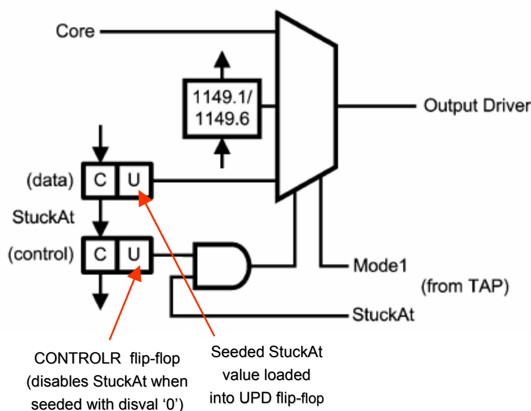
Standard Product Insertion Scheme

When considering implementing fault insertion on a standard product there are different trade-offs compared to the high pin count custom ASICs and FPGAs described in the papers mentioned. For standard products, no assumptions can be made about what pins may be required or optimal for fault insertion, hence all pins should support the feature.

Furthermore since the device may have many users the approach taken must be straightforward, simple to document and understand. Most critical is for the approach to be readily described in bsdl and able to be ported to 3rd party tools.

Factors that remain the same are the strong preference to ensure no increase in multiplexer delays in the signal path, and the ability to insert single or multiple stuck-at faults independently through-out the device.

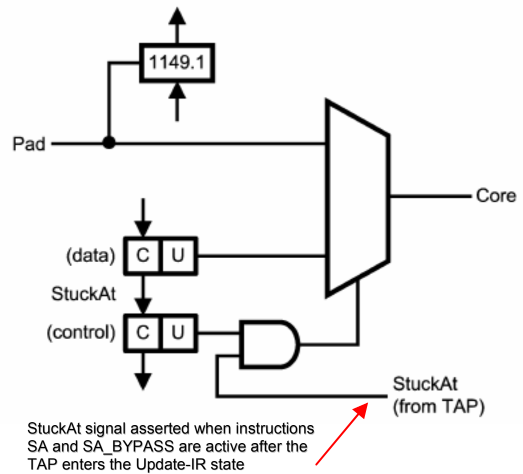
Figure 1: Stuck-at Output Cell



For these reasons the target device, a National Semiconductor SCAN90CP02, [4] uses a completely separate register for fault insertion, with two bits required for each pin, a stuck-at value, and a stuck-at control.

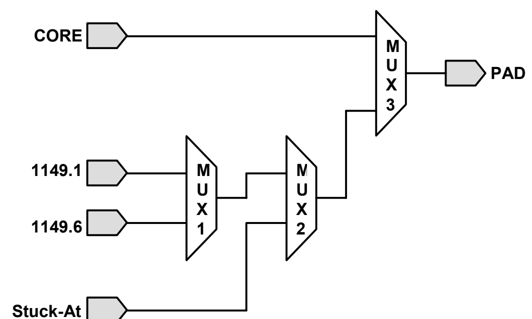
As shown in *figures 1 & 2* a "Stuck-at" instruction generates a TAP signal that is fed into an AND gate along with the stuck-at control flag. The stuck-at Data for each pin will only be presented when both conditions are TRUE, the stuck-at control is asserted and a stuck-at instruction is updated in the TAP.

Figure 2: Stuck-at Input Cell



Since the subject device supports IEEE1149.6 a series of muxes are used to determine the output state at each differential pin *figure 3*.

Figure 3: Multiplexed Output States



The first mux establishes whether IEEE1149.1 or IEEE1149.6 test data is presented to the 2nd mux. The 2nd mux determines whether the stuck-at data or boundary scan test data is presented to the 3rd mux.
The 3rd mux selects between the core mission circuit or the stuck-at and test data. In this manner a single mux delay is retained in the mission path.

Single ended pins use a similar approach but simpler without the requirement of IEEE1149.6

Tool Support

The problem with any extension to the IEEE 1149.1 standard is the guarantee of third party support from boundary-scan tool vendors, for the generation of the bit streams to support the extended functionality.

It is important that any extension to the standard can be described within the Boundary Scan Description Language (BSDL) file in a format that can be easily read by any third party BSDL parser software.

As previously described in this paper, the fault insertion functionality within the National Semiconductor SCAN90CP02 device is implemented using a completely separate **Stuck-at** data register comprising of 20 bits in length. Each of the stuck-at pins has a dedicated cell for selecting the stuck-at value and an associated control cell that enables the stuck-at capability.

The **Stuck-at** data register order can be simply described in a table format as depicted in *Table 1*, but in order for this information to be read sensibly by a BSDL parser it must be converted into a more readable format.

Table 1: Stuck-at Data Register Order

Cell Number	Function	Pin
0 (TDO)	Control	OUT1±
1	Data	OUT1±
2	Control	OUT0±
3	Data	OUT0±
4	Control	PEM11
5	Data	PEM11
6	Control	PEM10
7	Data	PEM10
8	Control	PEM01
9	Data	PEM01
10	Control	PEM00
11	Data	PEM00
12	Control	$\overline{EN1}$
13	Data	$\overline{EN1}$
14	Control	$\overline{EN0}$
15	Data	$\overline{EN0}$
16	Control	SEL0
17	Data	SEL0
18	Control	SEL1
19 (TDI)	Data	SEL1

The IEEE 1149.1 standard does allow users to define their own extensions to the BSDL language to include new syntax for additional capabilities [5].

These BSDL Extensions must be declared within the BSDL file before they can be defined, or alternatively the user-defined package can also be used to declare the BSDL extensions.
By referencing the package with a '**use'** statement, the declaration of the extensions become available to a BSDL description as described below

```
attribute SA_REGISTER_LENGTH:   BSDL_EXTENSION;
attribute SA_REGISTER:          BSDL_EXTENSION;
```

In this example the BSDL Extension attribute 'SA_REGISTER_LENGTH' will define the number of cells in the SA_REGISTER, defined by the attribute 'SA_REGISTER'.

This now allows these extensions to be defined in the BSDL file as detailed in *figure 4*.

Figure 4: BSDL Extension Definition

attribute SA_REGISTER_LENGTH of SCAN90CP02 : entity is 20;

```
attribute SA_REGISTER of SCAN90CP02 : entity is
-- num   cell        port            function     safe [ccell disval rslt]

   "0  (BC_1,        *              ,CONTROLR , 0)," &
   "1  (SA_DATA, OUT_POS1 ,INTERNAL  , X,   0,   0)," &
   "2  (BC_1,     *               ,CONTROLR , 0)," &
   "3  (SA_DATA, OUT_POS0 ,INTERNAL  , X,   2,   0)," &
   "4  (BC_1,     *               ,CONTROLR , 0)," &
   "5  (SA_DATA, PEM11       ,INTERNAL  , X,   4,   0)," &
   "6  (BC_1,     *               ,CONTROLR , 0)," &
   "7  (SA_DATA, PEM10       ,INTERNAL  , X,   6,   0)," &
   "8  (BC_1,     *               ,CONTROLR , 0)," &
   "9  (SA_DATA, PEM01       ,INTERNAL  , X,   8,   0)," &
  "10 (BC_1,     *               ,CONTROLR, 0)," &
  "11 (SA_DATA, PEM00       ,INTERNAL  , X,  10,   0)," &
  "12 (BC_1,     *               ,CONTROLR, 0)," &
  "13 (SA_DATA, EN_N1       ,INTERNAL  , X,  12,   0)," &
  "14 (BC_1,     *               ,CONTROLR, 0)," &
  "15 (SA_DATA, EN_N0       ,INTERNAL  , X,  14,   0)," &
  "16 (BC_1,     *               ,CONTROLR, 0)," &
  "17 (SA_DATA, SEL1         ,INTERNAL  , X,  16,   0)," &
  "18 (BC_1,     *               ,CONTROLR, 0)," &
  "19 (SA_DATA, SEL0         ,INTERNAL  , X,  18,   0)," ;
```

The 'SA_REGISTER_LENGTH is defined to consist of 20 cells and the 'SA_REGISTER' order is now defined in a format similar to that of the boundary-scan data register 'BOUNDARY_REGISTER'.

As already discussed the BSDL Extensions must be declared before they can be defined within the BSDL language. In this example we have chosen to make the declarations within a new **user package** description called *Fault_Insertion_v1_2005.all.*

This user package description referenced within the BSDL file by the statement **use Fault_Insertion_v1_2005.all;** as detailed in *figure.5* also defines a new **SA_DATA** cell defined by the attribute 'SA_REGISTER'.

This cell is used to seed the stuck-at value shifted into the SA_REGISTER during the Shift-DR sequence and latched into the **Stuck-at Data** register as the TAP enters the Update-DR state.

Figure 5: User Package Definition

```
package Fault_Insertion_v1_2005 is
  use STD_1149_1_2001.all;

  attribute SA_REGISTER_LENGTH:   BSDL_EXTENSION;
  attribute SA_REGISTER:                BSDL_EXTENSION;

  constant SA_DATA : Cell_Info; -- Stuck_At Fault Value

end Fault_Insertion_v1_2005;

package body Fault_Insertion_v1_2005 is

  use STD_1149_1_2001.all; -- Refer to BSDL definitions

-- Captures 'Seed' values in the SA_Data FF

constant SA_DATA : Cell_Info:=
((INTERNAL, SA, Zero),  (INTERNAL, SA_PRELOAD, Zero),
 (INTERNAL, SA, One),   (INTERNAL, SA_PRELOAD, One),
 (INTERNAL, SA, X),       (INTERNAL, SA_PRELOAD, X),
 (INTERNAL, SA, CAP),  (INTERNAL, SA_PRELOAD, CAP));

end Fault_Insertion_v1_2005;
```

The cell description for **SA_DATA** defines an array of CELL_DATA records containing three fields that define the CELL TYPE, the BSCAN_INST and the CAP_DATA field i.e.

(INTERNAL, SA, Zero) & (INTERNAL, SA_PRELOAD, Zero) etc.

These cell data descriptions define the cell type as an INTERNAL cell that supports the instructions SA and SA_PRELOAD and the source of the data captured by the capture flip-flop, which can be a Zero, One, X or the previously captured value (CAP).

The SA_REGISTER definition described in *figure.4* defines the use of both the SA_DATA cell and the CONTROLR cell.

The CONTROLR cell is similar to a CONTROL cell with the added feature that it is forced to its disabled state when the TAP enters Test-Logic-Reset.

Each of the stuck-at cells within the SA_REGISTER has its own CONTROLR cell, which is defined within the SA_REGISTER cell description i.e.

```
num   cell          port           function     safe [ccell disval rslt]
"2 (BC_1,           *           ,  CONTROLR,   0),             " &
"3 (SA_DATA, OUT_POS0,  INTERNAL   , X,     2,     0)," &
```

This clearly defines the CONTROLR cell for the SA_DATA cell as cell number 2 and the disable value as logic ‘0’. This is confirmed by defining the *safe* value of ‘0’ to be loaded into the Update (UPD) flip-flop in order to disable the stuck-at condition.

The cell description defined for the SA_DATA cell is different to the description of an AC_1 cell as defined within the BOUNDARY_REGISTER.

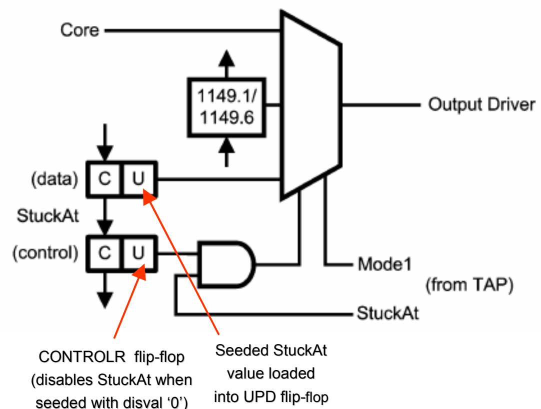
```
num   cell       port          function     safe [ccell disval rslt]
“4 (AC_1, OUT_POS1,     OUTPUT3,   X,     8,    0,    Z)," &
"8 (BC_1,         *       ,    CONTROL,   0),                  " &
```

The AC_1 cell is a three-state output cell that can be driven to one of three states, logic 0, 1 or the Z (high impedance) state, defined by the result field (rslt).

The function of the SA_DATA cell differs because the disable value does not disable an output driver.
It actually disables the seeded value that has been latched into the UPD flip-flop from being multiplexed out on to the output pins as detailed in *figure.6.*

Once the CONTROLR flip-flop is loaded with the *disval* of 0, the AND gate controlling the selection of the seeded Stuck-at value loaded into the UPD flip-flop, is effectively disabled.

Figure 6: Stuck-at Output Cell



There are three instructions that are associated with the stuck-at feature, SA, SA_PRELOAD and SA_BYPASS. The SA and SA_PRELOAD instructions allow access to the SA_REGISTER, whereas the SA_BYPASS instruction will access the BYPASS register as defined within the BSDL file i.e.

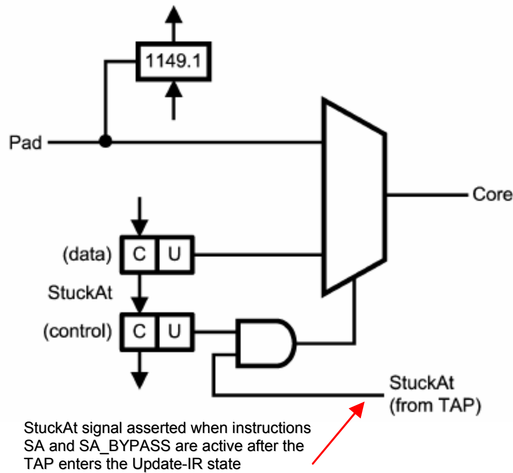
```
"SA_REGISTER     (SA_PRELOAD, SA),"&
"BYPASS          (SA_BYPASS),      "&
```

However, the stuck-at feature is only activated when either the SA or SA_BYPASS instruction is active, once the TAP enters the Update-IR state.

The latching of the SA and SA_BYPASS instructions assert the **Stuck-at** signal shown in *figure.6,* which has the same effect as loading the CONTROLR flip-flop with a logic ‘1’ and enable the seeded Stuck-at value to be multiplexed on to the output pin.

The same principle applies when seeding a stuck-at fault on an input pin, except that the effect of the stuck-at condition will be sensed by the core logic as shown in *figure.7.*

Figure 7: Stuck-at Input Cell



Conclusions

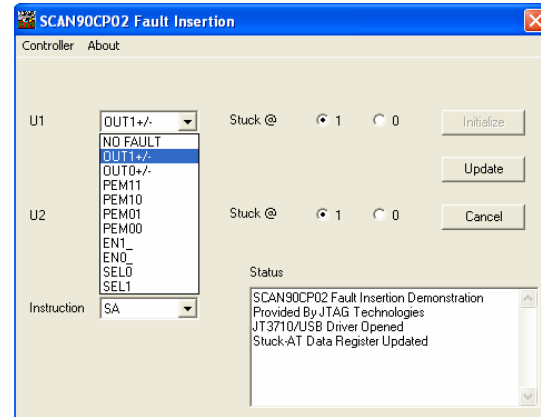
Boundary Scan as a Fault Insertion method is a viable approach for verifying system diagnostic software. Boundary Scan potentially offers a large number of injection sites and is completely non-intrusive.

This technique resolves some of the deficiencies seen with conventional hardware or software approaches. Fault Insertion capability can be added to IEEE1149.1 compliant ICs without any additional effect on the signal path.

The straightforward approach described in this paper can be easily described in bsdl and read by commercial tools.
Once Fault Insertion capability is included in a device bsdl file the next step is converting this information such that it can be readily observed and understood by users.

A tool that provides a pin list or graphical display of all potential injection sites and the means to select and initiate the faults, similar to that shown in *figure*.8 would be a powerful tool. Work in this area is continuing as the technique of using boundary scan for fault insertion becomes a more established methodology.

Figure 8: Stuck-at GUI



References

[1] P. Wilcox, J Hjartarson, Robert Hum, "Software Verification by Fault Insertion", US Patent no 5,130,988, 1992.

[2] B. Nadeau-Dostie, H Hulvershorn,S.M.Adham, "A New Hardware Fault Insertion Scheme for Systems Diagnostics Verification. IEEE International Test Conference Proceedings 1995, pages 994-1002.

3] T. Chakraborty, C-H Chiang, "A Novel Fault Injection Method for System Verification Based on FPGA Boundary Scan Architecture". IEEE International Test Conference Proceedings 2002, pages 923-929.

[4] National Semiconductor product datasheet, SCAN90CP02, http://www.national.com/appinfo/scan/cp02.html

[5] The Boundary-Scan handbook, 2nd Edition Kenneth P. Parker, 2001.