Acknowledgements

I would like to thank my supervisor Zebo Peng for all support in the work towards this thesis. Zebo has always been enthusiastic and spent a lot of time helping me to move forward in my research. A special thank should also go to Krzystof Kuchcinski, my assistant supervisor. I would also like to thank all past and present colleagues of the ESLAB and CADLAB groups for providing a very pleasant social atmosphere.

Outside the university I would like to thank Gunnar Carlsson at Ericsson Telecom for his contributions to and criticism of my work.

Jan Håkegård
Linköping
February 1998

Contents

1 Introduction

This thesis deals with the hierarchical test of hardware and specially focuses on testing aspects at the board level.

Hardware testing is the technique used to ensure the correct and fault free operation of electronic chips, circuit boards and complete systems. Hardware testing is performed several times through out the lifetime of a hardware product. The first time a product is subject to being tested is when it is manufactured, and the test that is performed then is called production test. Later, when the unit is in operation, operation and maintenance tests are performed. This latter kind of test is preformed to ensure that no faults have occurred in the circuit after the manufacturing, and may be applied more or less regularly.

It should be noted that there is an important conceptual difference between hardware testing and hardware verification which in contrary to hardware testing is used to verify/prove that a design is correct. Thus hardware verification does not require access to any physical units, and once made it holds for all units manufactured. The purpose of hardware testing is thus not, as for hardware verification, to prove the functional correctness of a circuit, although a side-effect when initially developing the hardware test methods is actually sometimes the discovery of design faults.

This thesis has its focus on hierarchical testing, which is the technique used to access, utilize and abstract embedded hardware testing capabilities in a large system. For a complete system consisting of a number of circuit boards each consisting of a number of chips, a strategy has to be developed to access test capabilities of the boards and chips in the system. In a hierarchical self-test architecture the system itself should know how it should be tested. This means basically that it should know how to order its underlying boards to perform testing. Each board in turn should handle the testing of itself and order its underlying chips to perform their self-tests. This makes the system hierarchically self-testable, and has several advantages. One of the advantages is that at the highest level in the system no information about how the chips should be tested need to be stored, and the same holds for the internal properties of each board since this can be abstracted by simple self-test commands. Another advantage is that the built in self-tests at the different levels can be utilized without dismounting the system and that no complicated external test equipment is necessary (external test equipment is equipment used to test e.g. a board or a chip and may be used either to probe into a system or it may be necessary to remove the device from the system and put it physically in the tester).

1.1 Motivation

With the increasing complexity and miniaturization of today's Printed Circuit Boards (PCBs) it is becoming more and more difficult to apply traditional test techniques such as In Circuit Testing (ICT) where external test fixtures have to be connected to the board. The introduction of Surface Mount Technology (SMT), Multi Chip Modules (MCMs) and double-sided circuit boards makes the situation even worse. Therefore an increasing interest is taken in Design For Testability (DFT) techniques, where test functions are integrated into the products, thus relieving the need for external test equipment.

A major contribution to DFT at the board level has been the standardization of Boundary Scan (BScan) by IEEE [IEEE 90], which is a standard that among other things describes how to access the pads on the integrated circuits by putting shift registers at the boundary of each integrated circuit. Boundary scan can also be used to initiate Built-In Self-Test (BIST) actions within the circuit, i.e. test engines that autonomously perform internal integrity tests within the circuit. The BScan technique, as it reduces the need for ICT tests, is often supplemented with a Test Controller (TC). The idea of such a device is to enable control of the test functions without using special Automatic Test Equipment (ATE). In this way, self test as well as operation and maintenance test at the board level can be achieved. Several test controllers have been developed, ranging from simple BScan interfaces such as the Boundary Scan Master described in [Jar 91b] to complete controllers running test programs on their own [Mat 92].

Hierarchical test is often associated with test controllers. In a hierarchical test environment test can be performed at different levels. As the lowest hierarchical level the individual chip is usually considered. Today more complex chips often have built in autonomous test facilities, such as the BIST scheme. The built in self-test capability of a chip relives the environment of the chip (i.e. the board) of knowing any details about how it should be tested. The only information needed is how to initiate the testing and how to judge if the testing was successful or not. Regarding hierarchy, the next higher hierarchical level of a chip will be the board on which it is mounted (but it could also be e.g. a Multi Chip Module, where several chips are mounted and interconnected within a single package). By providing the board with the logic to control the BIST function of the chips even the knowledge of how to initialize BIST in the circuit does not have to be known to the outside of the board, and for most, the chips do not have to be dismounted from the board to be tested. Thus it is now only necessary to know how to initiate testing of the board, and this will automatically perform the test functions both implemented inside the different chips as well as the test functions implemented at the board level (e.g. boundary scan, P-BIST etc.).

Test cost will be reduced while test quality may be improved. Today testing is a very expensive part of the manufacturing of a system. In systems design, test costs have been estimated to be as much as one third of the production cost of the complete product. If a hierarchical test concept is built in, testing will be much simpler and thus reduce the cost of the whole system. A further advantage of the built in self-test technique is its availability in the system through the whole lifetime of the system. Thus self-test (and if implemented, also diagnosis) routines can easily be run at the site where the system resides, requiring neither expensive test equipment nor disassembly of the system.

1.2 Problem Formulation

Most test controllers that have been proposed or implemented are relatively fixed designs, where sometimes only a few design parameters are supposed to be tuned [Jar 92]. Usually the proposed test controllers are not targeted to any particular application, but there are also examples of test controllers that are targeted for e.g. telecom applications [Jag 95]. A possible strength of these controllers is that they should be so general that they can be applied in a very wide spectrum of applications without having to be modified. The idea they are based on is that they should run a microprogram to control the test and that it is the microprogram that integrates them into the system and adjusts them to the requirements.

The problem we put forward in this thesis is how to synthesize the whole controller to the specifics of the system and the board to support a particular hierarchical test architecture. This means that we can be much more flexible than what is possible with the traditional approach. Further we are not limited to using e.g. a specific test bus since the test bus used in the system can be synthesized into the test controller. With a fixed design the test bus in the system must coincide with the one implemented in the system, or an additional external bridge circuit has to be used.

We are aware of some work that has been done in synthesis of test controllers [Hab 95]. Their approach differs from ours though, since they specify a complete hierarchical system in their approach and they incorporate test facilities at all levels. This means that even at the chip level their approach specifies what test facilities should be implemented in the chip. In our approach we do not want to make such a strong assumption about what the system should look like. Our approach is to semi automatically synthesize a test contoller given an existing hierarchical test system, and the board on which the test controller should reside can contain any type of components with differing test support. A further benefit of this approach is that the test controller can be optimized in terms of price/performance for each specific board, and trade offs can thus be made weather we want a cheap test controller for mass production of a simple system or a very complex one for larger low volume systems that are not as cost sensitive.

1.3 Contributions

This thesis describes our approach to the design of a Board-level Test Controller (BTC), which is used to support hierarchical test based on boundary scan techniques. Previous research on Board-level Test Controllers that have been presented has focused either on design proposals for how a test controller chip should be implemented, or proposed hierarchical concepts going from the system level down to the chip level [Hab 95]. The formulation of our problem is how to develop a framework in which test controllers can be synthesized to adhere both to the requirements the designer puts on the test controller in a particular design and also to the environment in which the controller will operate. Further, our controller is designed to support efficient self-test during the whole life cycle of the product, and this is supported by our hierarchical architecture in which a test controller at a board embedded deeply in a large system can be easily accessed from outside. Thus the controller proposed will not be limited to testing with the board removed from the system which would be the case without our hierarchical architecture.

The test controller is built from what we refer to as lego pieces [Håk 96b], i.e. small building blocks found in a library that can be put together to make up a complete controller. Examples of lego pieces can be a test pattern generator or interface logic for a protocol. These lego pieces can be pre-written code, but may still be parameterized where e.g. a linear feedback shift register lego piece can be parameterized regarding both its length and its polynomial. The lego pieces may contain control logic such as internal finite state machines (typically a lego piece implementing a complicated communications protocol may require quite complex finite state machines), but the overall control of the test controller is controlled by a finite state machine external to the lego pieces. The external finite state machine is to be generated concurrently with the lego piece construction.

We have developed a design representation for the test controller that supports the lego piece concept. We call it a Test Design Graph (TDG) [Håk 97], and it is based on the same ideas as the Extended Timed Petri Nets (ETPN) [Pen 94] which was developed as a general hardware design representation. In our case we need a more specialized design representation that does not only capture a hardware design, but also describes the test actions that we want to take. This is important for both the synthesis and optimization since it enables them to do more intelligent reasoning, as well as redesigning the controller in ways that would not be possible with the semantics preserving transformations defined for ETPN. This difference comes from the fact that since our design representation captures what is to be done functionally, it can choose to implement it in totally different ways, far beyond what is possible with the general purpose ETPN representation.

As with ETPN our TDG will be optimized through transformations. That means that initially the tool will construct a first version working test controller implementation, which will then be optimized in different respects (such as hardware cost, testing speed etc.) until it is accepted by the designer as a final design ready for implementation. We plan to use tabu search for the optimization process since this optimization technique is well known and has generally performed well in the area of hardware synthesis. It is also based on the neighborhood search technique which suits our transformational approach well.

1.4 Thesis Overview

In the next chapter we describe some background to the work presented in this thesis. We also describe some work related to this thesis that has been done elsewhere. Some of the differences between previous work and ours is also highlighted. In chapter 3 we describe the hierarchical test architecture that has been developed. This architecture is important for motivating the other work presented in this thesis. Chapter 3 is based on a paper [Håk 96] co-authored with Gunnar Carlsson, and much of the contributions for the framework is done by him. In chapter 4 the concept of the Board-Level test controller is presented along with some information about how it is to be synthesized to a hardware description. Chapter 5 shows some more concrete work by describing our representation of the test controller and how it can be synthesized. Finally a quite realistic but simplified board example is given in chapter 6 and it is shown how our test controller design representation can be used to design and optimize a board-level test controller for the board. The thesis is rounded up in chapter 7 where conclusions are drawn and some future work is pointed out.

2 Background and Related Work

2.1 Introduction

The methods usually used for board level tests are built-in self-test where the circuit is able to test itself when a test request is made. BIST is usually implemented inside integrated circuits, but the concept extends to several circuits controlled by a BIST engine located on the board. Processor based BIST (P-BIST) is a (usually functional) technique where the BIST test is preformed by a microprocessor. Boundary scan is a standardized technique that links together the circuits in a scan chain. This chain can be used in conjunction with BIST for controlling when to perform BIST and also analyzing the results of the BIST test. Its main, and original purpose, however is to perform interconnection test between the circuits, something that is not covered by traditional BIST schemes that only work inside the circuits. Some discussion will be devoted to In Circuit Tests (ICT). This is the traditional technique that uses bed-of-nails to physically access the circuit under test. This is a technique that is becoming obsolete for several reasons. Examples of these reasons are that it requires external test equipment and can thus not be built into the board and that it is increasingly difficult to use it as miniaturization of the electronics makes physical access at best hard if not impossible.

Solutions to this problem are the use of built-in self-test, boundary scan and board-level test controllers which will be discussed in this chapter.

2.2 BIST

Built-in self-test is a test technique where a circuit tests itself. Originally BIST was used to test an integrated circuit, but has since been generalized to MCMs, boards and whole systems.

2.2.1 General Concept

The BIST concept can be divided into two sub-groups, on-line BIST and off-line BIST. With on-line BIST is meant tests which are run during normal operation. Thus the circuit does not have to be put into a special test mode. This method can be used without disrupting operation. Implementation of on-line BIST are done in different ways, such as coding that can be checked for correctness during normal operation or by duplication where the response of the duplicate parts are compared. With off-line BIST on the other hand, the system will not function properly while test operations are performed.

The BIST implementation can be done for either functional or structural testing. In functional testing the tests are designed to test that the intended functions work regardless of the way the implementation is done. If an adder is tested functionally a set of numbers are picked and provided to the adder to see that the sums coming out are correct. One problem with this technique, however, is that it does not guarantee that the circuit will pass the test if another set of numbers are picked, and the time needed for testing all combinations grows exponentially with the number of bits used for the adder.

For structural testing a fault-model is introduced [Abr 90]. A fault model is a model of the kind of faults that are expected. The most common and most widely used fault model is the single stuck-at-fault model [Abr 90]. This fault model assumes that an interconnection in the design is either intact or permanently stuck-at zero or one. It is further assumed that there is no more than one fault in each design, an assumption that allows the test pattern generation algorithm to find a way of detecting each fault independently. Driven by such a fault model, test patterns can be chosen. And it is straightforward to calculate a measure of the fault coverage achieved, since with structural testing we can calculate how many of the faults in the fault model that are actually detected by the structural test, which is not the case with functional testing. Since the choice of test patterns can be made smarter for structural testing, a structural test usually gives higher fault coverage than a functional one with the same number of test vectors.

In BIST test patterns are often generated by special BIST logic in the circuit. For simplicity often Linear Feedback Shift (LFSR) Registers are used for the dominating test philosophy, structural testing. Sometimes this is expanded to weighted random patterns, in which the probability of ones and zeroes are differentiated. This is useful in cases where a random pattern performs poor structural testing such as for a multiple input AND gate where structural testing will want to test the all one and the only one zero sequences.

2.2.2 P-BIST

Processor based BIST is the BIST technique where a system is tested by a microprocessor running a test program. By P-BIST the microprocessor can test either itself or peripheral devices such as ASICs and memories. The testing of the microprocessor itself can only be performed functionally since the only thing that can be done is to functionally check that all instructions, registers etc. work. To enable structural tests of microprocessor hardware, BIST is required, but that does not sort under P-BIST. External devices may be tested either functionally or structurally depending on what is supported by the particular device. Typical areas where functional P-BIST is performed are memories, where functional test methods, such as the March test [Goo 91], have been developed to cover faults in a functional model of the memory. Structural tests are possible if the structure is known and the processor is able to feed test patterns to the device. Thus structural tests can e.g. be performed on glue logic built from discrete components, while the more complex devices have to go with functional testing.

On-line BIST can be achieved in P-BIST by alternating the execution of the task load and test routines. When the processor is idle, test execution can be performed and thus does not necessarily intrude on the task load at all.

2.3 Board-Level Test and Boundary Scan

Board level tests involve the testing of a complete circuit board with the integrated circuits mounted on it. This test is initially performed during the production test phase to ensure that the circuit board itself is correct and that the components are mounted properly on it. These two aspects can be tested by interconnection tests between the circuits. Further the components mounted on the circuit board may be tested in place, i.e. BIST techniques may be activated when the circuit is in position on the board.

In circuit testing is the traditional test method where a fixture with test probes (often called bed-of-nails) is attached to the Circuit Under Test (CUT). This test method requires (usually expensive) external test equipment and the CUT has to be physically mounted in the tester. Thus ICT testing is mainly used for production testing or to perform off-line fault diagnosis of boards that have been physically removed from the system. Besides being expensive and limited requiring special test equipment, equipment that must include high precision mechanics to access the board, ICT is becoming less viable (or even impossible) for the new circuit board generation. It is less viable because the dimensions constantly shrink with new packaging and manufacturing technologies. It is impossible to access when multi layer circuit boards are used and packaging techniques such as BGA (Ball Grid Array) are used[1].

To solve the problems with ICT testing the boundary scan technique was developed. Boundary scan is a technique to enable direct observation and control of the logical values on the inputs and outputs of different integrated circuits. This control is gained through the insertion of latches at the pads of the ICs. Information can be obtained by serially shifting data out from these latches and at the same time control is gained as new data is shifted in. This feature has been standardized by IEEE and is described in [IEEE 90]. In order to comply to the standard four additional boundary scan control pads are required on each IC. Two of these are the input (TDI) and output (TDO) from the scan chain. There is also a control signal (TMS) through which the operation of the boundary scan chain is controlled. The fourth mandatory pad is the test clock (TCK) used to synchronize the operations. A schematic over a boundary scan compliant IC is given in figure 2.1. The scan cells are drawn as dashed squares while the Test Access Port (TAP), which is controlling the operations, is found to the left.

[1] On a Ball Grid Array chip the pads are located in a matrix underneath the chip and thus the chip itself covers the soldering points and out rules probing techniques.

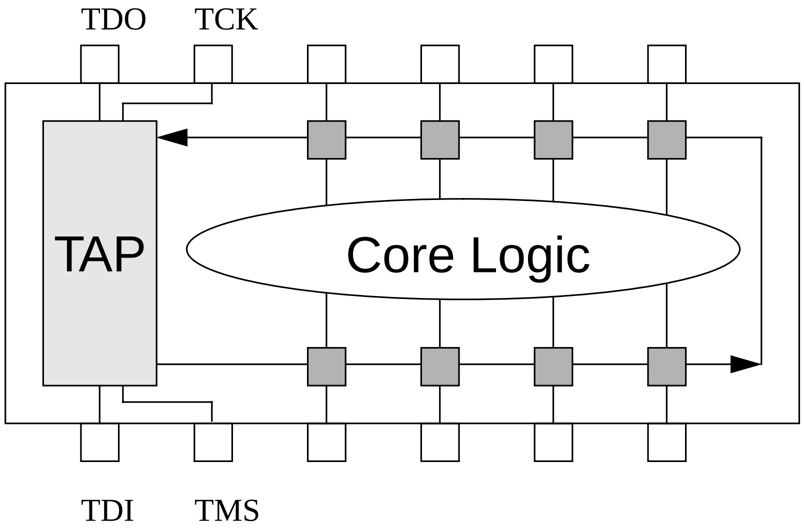


Figure 2.1: Schematic of an IEEE 1149.1 boundary scan IC.

The TAP controller is implemented as a finite state machine, whose state transitions are controlled by the TMS signal. The different states can be seen in figure 2.2 along with the TMS values used to select the transitions. It can further be seen in the figure that there exist two main branches starting with Select-DR-Scan and Select-IR-Scan respectively. The purpose of these branches is to shift data through the boundary scan cells and the instruction register respectively. The instruction register is used to store an instruction that in combination with the current state in the diagram will affect the behaviour of the boundary scan. Shifts are performed in the states marked Shift-DR and Shift-IR, but shifting can be temporarily interrupted by entering the pause states. The Capture states capture the values on the inputs of the boundary scan cells into the latches of these cells, while the Update states feed the new state of the internal latches to the outputs of the circuit.

If there exist non BScan ICs on the board, there will be logic on the board that is not directly accessed through BS. These parts are referred to as non BScan clusters, and if no external test equipment is used to test them, testing must be carried out by use of the BScan chain in the surrounding ICs. In terms of design for testability efforts it is important to consider how these non BScan clusters are interconnected. Attempts should be made to reduce the logical depth of the clusters, which will decrease the number of test patterns that have to be applied.

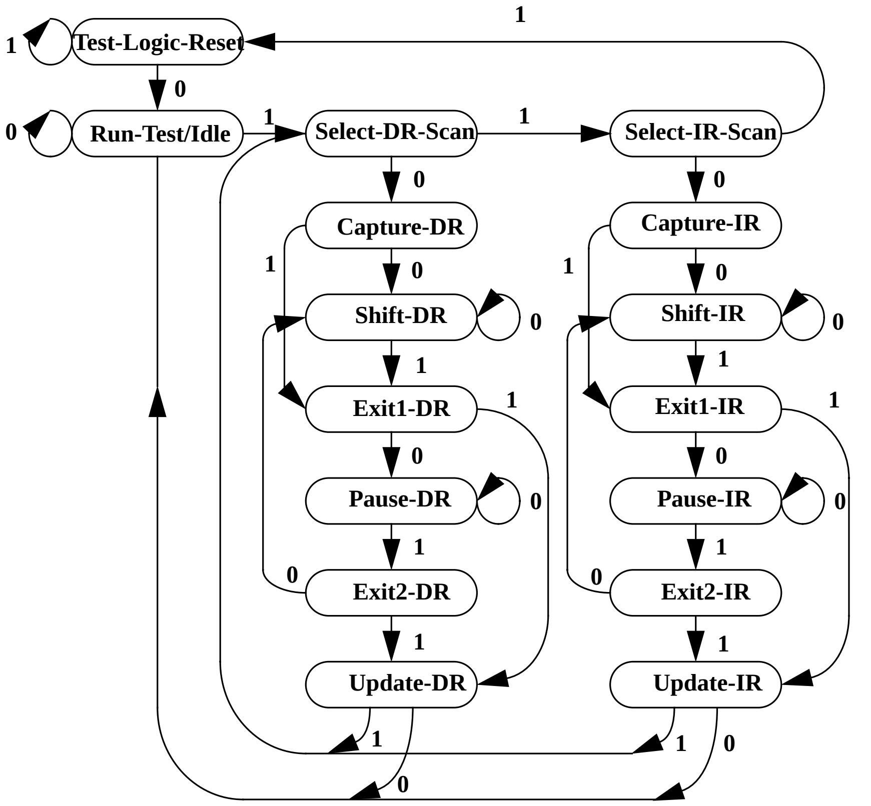


Figure 2.2: State diagram of the Test Access Port.

2.4 Test Controller

The concept of a test controller (or subsets of it) has been proposed by many researchers although varying names have been used such as the Boundary Scan Master (BSM) [Jar 91][Jar 91b], System Test Access Port Master (STAPM) [Jag 95], Module test and Maintenance Controller (MMC) [Bre 88], Test Controller/Test Processor [Mat 92] etc.

The purpose of a full featured test controller is to control test activities at a certain level of a system. In this thesis we deal with board level test controllers, which means that the purpose of our test controller is to control the individual board’s test. The rest of this description will concern board test controllers. In a hierarchical test environment there can be test controllers both at a lower level, i.e. inside a chip or MCM-module or at higher levels such as in a sub-rack or even a test controller for the entire system.

A test controller is used to coordinate and control test facilities built into the circuit board. By doing so it will abstract the access to all test facilities built into the board, such as boundary scan, BIST and P-BIST. Testing of the board will then require only simple commands that are independent both of the boards function and its implementation. By an external request the test controller should thus be able to automatically control and sequentialize a complete test of the board. This test will utilize test facilities present on the board, such as BScan chains on the board, BIST schemes built in to the ICs and also it may utilize an on board processor if such a device is present for performing P-BIST. Further the TC may contain special board test capabilities in itself, such as a memory exerciser to perform the March [Goo 91] test on a memory, if it for some reason is preferred not to use a microprocessor, or if a microprocessor is not present.

Test controller designs published in the literature vary from basically interface circuits which are totally controlled by a microprocessor [Jar 91][Jar 91b] (in such cases the combination of the interface circuit and the microprocessor makes up the test controller) to autonomously running devices which provide their own control [Mat 92][Jag 95]. Our concept of a test controller is very flexible, but for us to call it a test controller it should behave as a complete controller with its own interface to the external device requesting the test, while the actual implementation (hidden on the board) may rely on e.g. a microprocessor not only for specific tests but also for the overall control.

The above description has only described the board abstraction aspects of the test controller as its advantage. The major benefit of using a test controller comes, however, from the fact that when a test controller is incorporated on a board that resides in a hierarchical test system it can be accessed from within the system without removing the board and putting it into special external test equipment. Thus test controllers at the board level make implementations of complete system built-in self-tests possible. It also enables testing that does not require any (or at least minimal) external test equipment. Further since the test controller is built in to the board it can be utilized during the whole life time of the product.

2.5 Related Work

Several research groups have made contributions in the areas of hierarchical self-test, board-level test and techniques to utilize the boundary scan standard. A brief description of some of the important work in this area is presented below. We feel that our approach is unique though, since it is the only work that we know of that deals with the synthesis of a complete and unique test controller for the board without requiring hierarchical insertion of specific test features in the integrated circuits on the board.

2.5.1 AT&T Bell Laboratories

At AT&T several research papers have been published on the use of boundary scan testing [Jar 92], and a boundary scan control device called the Boundary Scan Master (BSM) has been developed [Jar 91][Jar 91b]. The purpose of the BSM is to act as a protocol converter between a generic microprocessor and the boundary scan chain. In [Jar 92] it is shown how the BSM can be utilized in a hierarchical test environment. The proposed hierarchical approach is motivated by its ability to abstract board test information as well as offering revision independence as seen from higher hierarchical levels.

The BSM has two important limitations compared with other test controllers. The first limitation is that it does not contain a test bus interface. Rather it only contains registers that can be read and written to control it (through a generic microprocessor interface), and external interface logic have to be used to interface the controller with the test bus (or system bus) which is necessary if the controller is to be used in a hierarchical test environment. The second limitation is that the BSM is not in itself what we would call a complete test controller. To construct, roughly, the equivalent of a test controller in our terminology it should be used in their so called Boundary Scan Virtual Machine (BVM) configuration which means that the BSM is supplemented by a BSM interpreter (BIN). A BIN is a test controller command interpreter which usually resides on a higher hierarchical level. A microprocessor will have to execute a program that interprets BVM programs. This interpreter will have to have specific knowledge of the interface logic used to interface the BSM to the test bus (or system bus) in order to access it. Although test programs will have to be run at this higher hierarchical level, it can be selected if the BIN will fetch its program from a data base at the system level or if it should be fetched from a ROM residing on the specific board. By the latter approach the board will be better abstracted and revision handling will be greatly simplified.

The approach taken supports a hierarchical test architecture, and it also moves testing details to the board level to make board implementation details transparent to the system. However, the BSM proposed is quite a complex, hardware design. Although it is stated that the architecture is generic [Jar 92], a quite fixed BSM design has been presented. In [Jar 91] where a more detailed description of the BSM is given, the generic parts are restricted to the processor interface. Very limited attention is payed to the reliability aspect, where the only aspect mentioned is that nearly 96.4% of the BSM itself is covered by a BIST scheme.

2.5.2 Bell-Nothern Research, Canada

At Bell-Nothern Research (BNR) they have developed what they call a System Test Access Port Master (STAPM) [Jag 95]. The STAPM is what we would refer to as a test controller, and was designed to operate at the board level to handle BIST and boundary scan operations on a board with several ASICs and one microprocessor. An implementation of the controller was actually made and the system was tested. The STAPM is presented as having basically two purposes, one being to act as a test coordinator and abstractor on the board level (which is also the purpose of our test controller work) but also to perform fault insertion which they think is important in the telecom application that it was used in.

Fault insertion is the process in which a fault is simulated to see the results of it. For BNRs’ telecommunication applications it is desired to insert stuck-at faults to verify that the design is robust enough to continue operation with a single stuck-at fault present. The STAPM itself is not able to introduce these faults on the board, instead the fault insertion is achieved by a proprietary, modified, implementation of the boundary scan chain.

The STAPM has two interfaces, one interface to the backplane or test bus which they refer to as the parallel port and one interface to external test equipment which they call the serial test interface. The serial test interface is basically intended to be connected to a PC that will request test through a character based interface, An interesting property of the design is that they have embedded an EEPROM (Electrically Erasable Programmable Read Only Memory) into the design that logs all errors that are detected in its non-volatile memory. Thus faults can be analyzed later when the system is off-line.

Although it is stressed in the end of the article that the STAPM could be adopted for different products their current work focuses very much on their own telecommunication applications.

2.5.3 University of Porto / INESC

Matos *et al.* at the University of Porto / INESC have proposed a test controller design [Mat 92]. They have even manufactured an ASIC circuit that implements their design. The design is a fixed one that is designed only for handling boundary scan compliant ICs. It has two boundary scan chains built in, and it is argued that it can be used in a hierarchical test framework by using one of the scan chains as a test bus. There is no specific support for interaction or cooperation with an onboard microprocessor and there are no possibilities to adopt to any test bus other than the IEEE 1149.1.

Related to their test controller design they have carried out work on an ATPG tool that will generate test vectors for the circuit board. The tool will however not only generate the test vectors themselves but instead a complete test program for the test controller. To summarize, their test controller is only for boundary scan and is too limited to be considered as a general purpose test controller with our concept of a test controller. It does however support a hierarchical test environment for boundary scan, but not for general purpose test techniques. Since many test techniques such as BIST often can be initiated from boundary scan some expansion from the pure boundary scan tests can be done.

2.5.4 University of Karlsruhe

Haberl *et al.* at the University of Karlsruhe [Hab 95] describe a hierarchical test architecture. This architecture relies on a uniform hierarchical insertion of self-test hardware at all levels. Actual self-test implementations are done at each block inside the integrated circuits. Each BIST block has a standard test interface (STI). The STIs are connected at the next hierarchical level (still inside a chip) by a test management unit (TMU). The chip is finally equipped with an IEEE1149.1

[IEEE 90] TAP controller for a uniform communication with the rest of the board. Finally at the board level a hierarchical boundary-scan controller chip (HBSC) or programmable module self test controller chip (PMSC) [Hab 94] is found. These two chips (HBSC and PMSC) can be compared with the board test controller in our work.

Two major areas of difference between this work and our work can be identified. The first difference is that although this is a hierarchical approach it focuses more on the lower levels, i.e. how to make the board hierarchical itself rather than presenting a hierarchical test concept for a complex system such as a telephone switch, while in our work we propose such a concept. Second, in our work we allow the board to be built from standardized integrated circuits. Thus our approach does not require that the board is built solely from specially designed in house self testable integrated circuits in order to work with the PMSC/HBSC chip. Our test controller will work not only with boundary scan but also with e.g. P-BIST, something that the work by Haberl. *et al.* do not address in their hierarchical approach. If the hierarchical approach is disregarded, however the PMSC chip can be used as a more general purpose [Hab 94] test controller, although it is still very tied to a particular framework which incorporates the use of an IEEE 1149.5 [IEEE 95] test bus.

2.5.5 British Telecom Laboratories

At British Telecom (BT) the need for a well developed, hierarchical, test framework is argued [Mau 93]. The systems the article deals with ranges from circuit boards to interconnected computer systems, and the term system is thus very broad. One of the main argumentation the article puts forward for such systems is the need to abstract the implementation details for a smaller part of the system. No matter weather you view the system at the board level or at the interconnected computer systems level it would be beneficial to be able to use a set of standardized test commands to test all the underlying components. In [Mau 93] the solution to such a problem is proposed to be the inclusion of test management units at different hierarchical levels. These units should have a command repertoire including commands such as “test yourself”, “abort testing yourself” etc. In contrary to the HIST approach described above [Hab 95], it is not required that all chips contain a test management unit. It could e.g. be that the last test management unit found is at the board level, and in such a case this test management unit must act as a test controller to control testing of the board and utilize the non test management unit compliant test facilities found, such as BIST and Boundary-Scan.

As with our hierarchical proposal the range of standards considered ranges from the low level boundary scan standard [IEEE 90] to test bus standards such as [IEEE 95]. Maunder introduces a new level in the system and handles also the interconnection between completely different systems, such as a number of networked computers, and refers to the ISO10164 standard [ISO 10164] which describes test management regarding interconnected systems.

2.5.6 Bell Northern Research Department

A proposal for an improved boundary scan implementation is argued for in [Has 92]. It is to be used in a test controller environment (called TDC for Test and Diagnosis Controller in [Has 92]) to improve test efficiency. The motivation for the work is to improve test efficiency by allowing a walking zero/walking one interconnect test to be performed without having to shift the whole scan chain as required with traditional boundary scan.

Although the idea is novel, the main drawback of the approach is not only that a modification to boundary scan is required but mainly that all ICs in the boundary scan chain must comply with the new ideas. Therefore this approach is only feasible for boards manufactured by a company with in-house production of boundary scan ICs. In our approach of designing a test controller no modifications of the boundary scan cells are required, although if modifications are made it can easily be utilized by our test controller since lego pieces that take advantage of the new features can be designed and incorporated in our database.

2.5.7 University of Southern California

Lien *et al.* at the University of Southern California present a hierarchical test controller concept in [Lie 89]. The component in this framework that corresponds to our test controller is the Module test and Maintenance Controller (MMC) [Bre 88]. Just as with our BTC the MMC is supposed to reside on each board, controlling mainly the boundary scan bus but also other test methods such as P-BIST. The idea here is that the MMC should be targeted towards the board on which it is to reside. It is constructed from quite a big core that contains a processor, RAM, ROM, L1-slave (what we would call test bus interface), Test Channel (usually an IEEE 1149.1 interface) and a Bus Driver/Receiver. The processor is used to control the operation of the MMC, but could be any 16-bit processor either general purpose of special purpose. The Bus Driver/Receiver is used to drive an internal bus that connects to expansions to the proposed MMC core. Expansion units proposed are functional bus interface, testability registers (for inserting test points at the board), additional test channels (e.g. additional BScan interfaces) etc.

The main drawbacks of this work in respect to ours is that it contains quite a fixed and relatively big core that is not necessary in our much more dynamic test controller design. Secondly, although the MMC can be targeted towards the requirements on a specific board there is no automatic MMC generator presented.

2.5.8 Dassault Electronique

At Dassault work has been done both in formulating a hierarchical test concept [Rig 89] and in the testability analysis of boards (and MCMs) [Per 97]. The hierarchical test framework is specified (similarly to our model) as a number of hierarchical levels each containing self-test capabilities. In [Rig 89] a system test processor is found at the highest hierarchical level. At intermediate levels (subassemblies) a test controller is found followed by local test controllers at the board level. The lowest level considered is the ASIC level where a BIST function controlled by an elementary test controller is assumed. The whole design relies heavily on ASIC BIST as well as implementation of the elementary test controller within the ASIC instead of accepting diverse BIST implementations, although it has later been homogenized with boundary scan [Per 97]. No mention about functional test or microprocessor based tests are made, and the example presented in the paper contains some 240 ASICs but no microprocessor.

In [Per 97] a qualitative testability is described. It is based on the controllability/observability concept augmented with so called testability rules to incorporate some expert system ideas. The idea of the algorithm is to quickly pinpoint testability improvements that can be applied to the board. Initially a netlist is read and the components are classified as PALs, FPGAs, JTAG, Combinational etc. For the simpler components such as combinational controllability/observability propagation can be made directly while e.g. a JTAG equipped ASIC will be seen as a black box with internal tests performed by an integral BIST engine.

2.5.9 Ericsson Telecom

At Ericsson Telecom processor based BIST is used for functional testing to supplement the structural testing performed by boundary scan and ATE equipment. The basic method is briefly described in [Ang 97]. Currently Ericssons implementation of P-BIST can not be controlled hierarchically in the system. Instead a special pad is used at power up to set the board into test mode, i.e. to boot the processor with the P-BIST program. Communication with the P-BIST program is further carried out by direct connection of a text terminal through a serial interface. It is pointed out in [Ang 97], however, that future work will target towards the ability of initiating P-BIST from the system software which will make it more similar to our way of using P-BIST, i.e. to initiate it during operation.

2.6 Discussion

There are several techniques developed to support the testing of a circuit board. These ranges from the by now outdated ICT test to more modern techniques such as the BIST for testing integrated circuits and P-BIST for performing tests coordinated by a microprocessor. An important addition to the board level test methods was the standardization of boundary scan by IEEE. This test method is capable of replacing the ICT test for digital interconnections, and can also be used in areas for which ICT was never be intended, such as internally within MCMs. The future for boundary scan in a growing number of products is expected since being completely built into the circuits it is neither dependent on the chip packaging nor their size. Further as boundary scan is implemented with the same implementation technique as the rest of the chip its speed and abilities will be able to improve along with the rest of the development in the integrated circuit area.

To support board testing in a hierarchical system several researchers have proposed test controllers. The most common implementation of a test controller is a chip which is capable of coordinating the testing of a board and report the results of the tests to a higher hierarchical level. Most research this far has, however, concerned the design of a more or less fixed test controller. Our approach differs from these in that we propose a synthesis tool which will, semi automatically, design test controllers for us. These test controllers will be targeted to the particular system for which they are being synthesized. This means that we, in contrary to other proposals, can adopt arbitrary test techniques and protocols. We believe that our approach is more flexible and also that it will yield a much better price/performance ratio since the test controller is integrated with the board.

3 A Hierarchical Test Architecture

This chapter describes the hierarchical test framework that our test controller will work within. Much of the framework has been contributed by Gunnar Carlsson at Ericsson Cadlab Research Center. This chapter is actually based on an elaborated version of a paper written in cooperation with him [Håk 96].

In the chapter a description of our test architecture and what requirements it puts on the test controller will be given. The hierarchical system is based on several levels of busses using different protocols when propagating test related information throughout the system from the highest system level down to the board level. The idea is that at any higher hierarchical level it should always be possible to invoke test actions at lower levels in a coherent way. The methodology is developed to support not only production testing but also operation and maintenance testing, and the whole hierarchical test support mechanism will be fully integrated within the system, which means that minimal or no external hardware is required to support it. The framework will support structural testing by invoking the structural test facilities incorporated, but it can also be used for functional (or ad hoc) testing purposes such as processor based BIST.

3.1 The Hierarchical Structure

A sketch over the hierarchical framework can be seen in figure 3.1. Our main focus is put here on the light grey rack which constitutes of three darker colored subracks. The idea is that at the system level (e.g. a complete AXE[1] telephone switch) there should be test facilities that can request tests at each subrack. At the top level in the hierarchical system there is the Operations and Maintenance Processor (O&M Processor), which can be seen as a small grey box in figure 3.1.

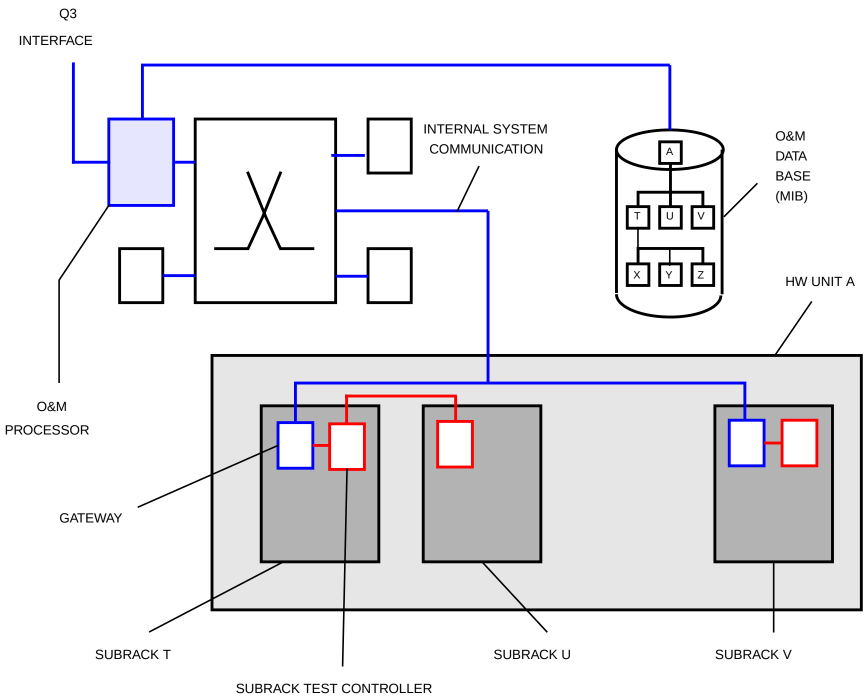


Figure 3.1: Basic structure of a system with a hierarchical test framework.

The operations and maintenance processor is executing the maintenance software for the whole system, and it is also this processor that directs test requests to the subracks. Its communication traffic with the subracks does not have to be limited to pure test information, it may very well include additional features such as board revision requests and the ability to take a subrack or circuit board off-line.

As mentioned above several levels of test busses are incorporated in the system. The highest level bus is the internal system communication bus running between the operations and maintenance processor and the racks. This bus is terminated in the rack by a gateway function residing in one of the subracks (as indicated in figure 3.1 there may be more than one gateway, in the example two gateway chips are shown). From the

[1] AXE is a digital telephone switch developed and marketed by Ericsson Telecom. Throughout this work it has been used for exemplification as well as a case study of a possible target system for our hierarchical framework.

gateway there is an internal bus that runs between the subrack test controllers. Both the gateway and the subrack test controllers can be seen in figure 3.2, and it is also shown how they are interconnected. After the subrack test controller comes a test and maintenance bus that runs between the board test controllers that coordinate the board testing. On Board Z in figure 3.2 it is shown how even a component test controller has been introduced in one of the chips. This is fully feasible, and the board test controller could then have a bus to interact with that level of test controller as well (for example through the boundary scan chain).

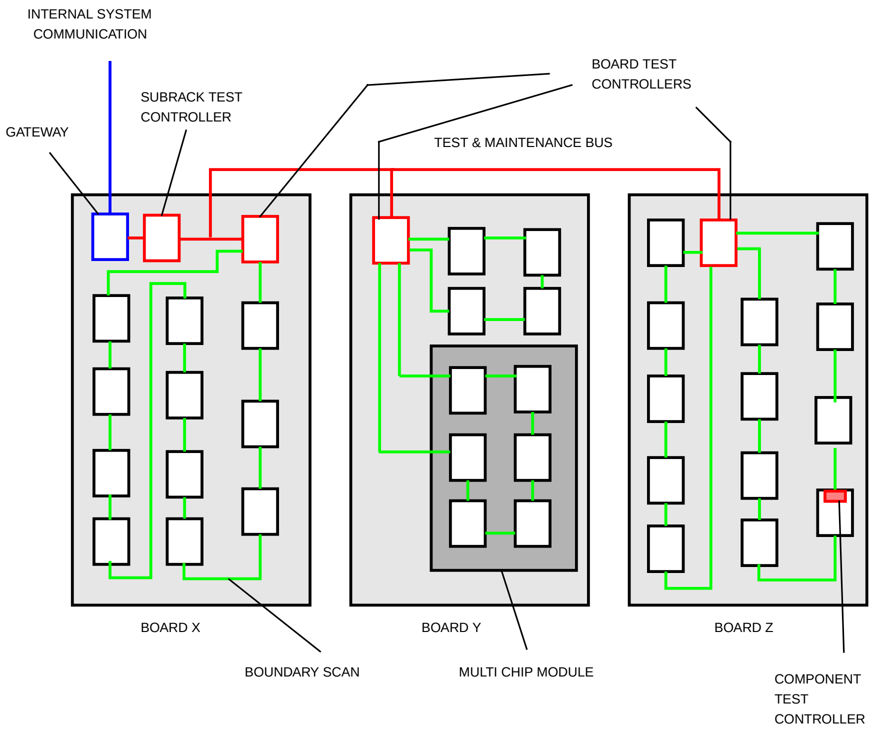


Figure 3.2: The test bus and controller hierarchy in a subrack.

The darker colored part of the middle circuit board in figure 3.2 is an MCM module, and as can be seen it is fully feasible to test it internally by using a boundary scan chain. Thus from the test controller's point of view (and when generating test patterns) the dies in the MCM can be seen as any ordinary chips and they are as accessible as a set of normal chips. In figure 3.2 it can also be seen how the test controllers can handle varying numbers of boundary scan chains, where the two right most boards have two boundary scan chains while the left most one only has a single chain. Control busses within the circuit board do not have to conform to any specific system wide design specification.

In the above picture it is shown how all of the integrated circuits are equipped with boundary scan, but this is not a requirement, although it will improve the ability for the board test controller to perform testing. Further as can be seen the built in test controller in one of the chips on the right most circuit board is assumed to be controlled by the boundary scan chain (i.e. the TAP controller), but this is not a requirement either, since the specific board test controller on that board can very well be equipped with a special interface for direct connection to the on board test controller. Since all these things are abstracted by the board test controller they do not complicate testing of the complete system.

3.2 Principles of operation

The idea of the hierarchical structure is that each level should be able to handle testing without information from higher hierarchical levels. This means that when coming down to the board level the board test controller can autonomously control the testing. Usually the test program for the board test controller would be stored on the board itself in e.g. a ROM memory. Sometimes this may not be a feasible solution however, since one concern may be that stored test patterns grow to big to be stored in ROM. In such situations it may be better to keep them on a disk connected to the operations and maintenance processor at the highest hierarchical level. This is supported by the model and has been indicated in figure 3.1 as the database to the right. This database should contain test programs/test data that can be requested for access by a board test controller. The board level test controller should of course know which file to access to relieve this information from the operations and maintenance processor, but there is still an administrative need to keep all necessary files in the database. If two versions of a particular board are to be used in a system they may very well request different files, and for the system to support both realizations of the board both files have to be in the database. This problem is a cost trade-off that has to be made, since attaching mass storage directly to the circuit board would be both expensive as well as make the whole system much more complex.

In our framework no specific choice of busses are required, i.e. it is up to the designer of the system to choose what test bus and system bus are to be used. We believe it is important to leave this to the designer as the cost of implementing different busses will vary. In our approach of synthesizing specific board test controllers to fit into a board and eventually a system we make trade-offs between cost and performance. Since we have not selected any mandatory busses to be used in the system, we will be able to make similar cost/performance trade-offs for these as well. The requirements on a bus should differ vastly between e.g. an AXE telephone switch and busses used in cheaper consumer electronics (such as personal computers etc.), but with this freedom we are able to use the same framework for both. Without this approach we believe that our test controller synthesis ideas would be too limited.

To summarize the discussion some important key features of our approach are as follows:

- The interface to maintenance software is abstract and simple. Maintenance software developers do not have to know any specifics about the hardware implementation.
- There is little sensitivity to revision status of lower level units at higher levels. This means that similar boards with different implementation can freely be interchanged without any reconfiguration of the test facilities (unless central software has to be loaded as discussed above).
- The test data to be communicated between different levels is usually very limited (unless it has to be loaded from the database) and consists mainly of instructions and simple response data. This puts minimal requirements on the test bus capacity, speed and complexity, and therefore allows simple, inexpensive and robust solutions.

To make the interface to the test controller as generic as possible between the test controllers used in a system the set of operations and their behavior should be standardized as far as possible. This means that we should have mandatory instructions for the most common operations that can be performed on all boards, while leaving options for implementing more board specific instructions. We propose the following three subtypes of operations: *mandatory operations* with standardized behavior regarding both the instructions and the response given from them; *optional operations* which are standardized if they are present, but do not need to be implemented; and *private operations* which implement properties that are not covered by the mandatory or optional operations, but the format of them can still follow some guidelines about how to behave.

A further division of operations can be made between the ones that have impacts on the normal operation of the circuit board and those that can be done transparently with the operation. This is sometimes referred to as intrusive operations and non-intrusive operations respectively. Non-intrusive operations are operations which can be performed in parallel with normal system operation. This means that the operation of the system does not have to be disturbed at all when the test is performed. Intrusive operations refer to the operations which disallow normal system operation while executed. Intrusive operations have different recovery times, which means that given that an intrusive test operation is running, the recovery time specifies the time required to come back to normal operation.

The recovery time is important in e.g a telephone switch where during low traffic self tests are performed on parts of the system that are idle due to over-capacity of the switch. When eventually more calls arrive recovery time for additional switching capacity must not be too long. Sometimes intrusive operations are classified according to their recovery time and this classification can be used to decide when the operation is allowed to be performed, in order to guarantee fast enough response from the system.

Test data is propagated between the different parts in the system by a dedicated test bus. The use of a separate test bus makes it possible to perform fault detection and isolation even in situations where ordinary signal paths (i.e. the system bus) are failing. It also allows testing before the subsystem is taken into operation. The protocol for backplane test communication is as for most more complex communication protocols (for example TCP/IP[1]) divided into several layers. An instruction layer is found at the top, and it is this layer that specifies the instructions that can be used within the system. The instruction layer is implemented on top of the link layer which is the actual underlying communication protocol that the instruction layer is implemented on. Finally at the lowest level is the physical layer, which deals with the electrical properties of

[1] TCP/IP stands for Transmission Control Protocol/Internet Protocol and is a communication protocol frequently used for communication between computers. It is in itself implemented in several layers. Further it must be implemented on top of a hardware protocol to be realized, since it does not specify or require any particular transport medium. Usually it is implemented on top of the ethernet protocol, but it can also be used over e.g. serial connections such as a telephone line with a modem.

the protocol. The instruction layer is standardized for all products, and is thus independent of the bus chosen. This means that we will not see any difference in the way we handle our test requests, regardless of the actual bus used. The other two layers vary depending on requirements of the products as discussed above. However, existing standards can be used and the choice of such will be guided by recommendations for product categories, i.e. different kinds of busses will be used for different target systems. Examples of standard busses [Håk 95] worth mentioning are IEEE 1149.1 [IEEE 90] and IEEE 1149.5 [IEEE 95].

The IEEE 1149.1 and 1149.5 busses are both serial test busses, where the former is intended for the interconnection of chips on a board while the 1149.5 bus was developed for interconnecting different circuit boards. There is nothing in the IEEE 1149.1 standard that prevents it from being used for communication between different circuit boards. The 1149.5 uses a master slave concept just as the 1149.1 which means that there is a master on each bus which controls the slaves. The intelligence of the slaves may vary. The simplest slaves will be totally controlled from the master including being fed with test patterns. More sophisticated slaves (such as our test controller) will operate autonomously after having been requested to perform a test operation by the overlying hierarchical level (which contains the bus master).

4 Board-Level Test Controller and Its Design

4.1 The Main Function of the BTC

The board test controller is a chip (or other implementation such as part of a chip or a mixed hardware/software implementation) that implements the overall self-test functionality on a circuit board. It works in an autonomous way accepting commands either from the system as described in chapter 3 or from an external tester if the board is to be tested off-line.

Coordinating and controlling test functions on the board is the main task of the test controller, but it may be employed for other functions as well. These functions include identification of the board in which the test controller is used to (upon request) indicate to the system what kind of circuit board it resides on, as well as further data of the board such as version or configuration etc. Another possible use of a test controller is to take the board on- or off-line. This means that a board that is not used at the moment may be taken off-line by the test controller. This is useful for situations such as when intrusive tests are to be run on the board or when it is going to be removed from the system.

4.2 Design of the BTC

The main work in this thesis is in the definition of and work towards a semi-automatic design tool to build efficient targeted test controllers for any circuit board located in a hierarchical system.

4.2.1 Synthesis Approach

Our vision in the work towards board test controller synthesis is to eventually automate the whole BTC design process by use of the tool that has been sketched in figure 4.1.

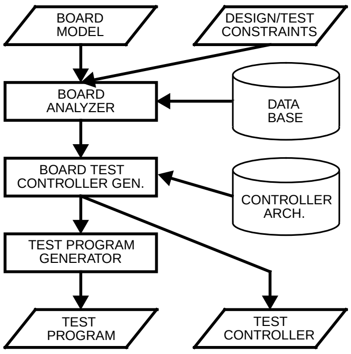


Figure 4.1: Proposed synthesis tool.

The idea of the tool is to take as input a description of the circuit board for which the test controller is to be designed. Along with this input some guidance information will be provided by the designer. The guidance that has to be given to the tool includes bounds on the desired test speed and hardware cost. Additional guidance may also be desirable such as choosing between implementation alternatives etc. The intended purpose of the tool is not to make the designer unnecessary, but rather to relief him by automating what is feasible and does not intrude on the designer’s preferences about the design.

The tool is structured into three main parts, where the first one is the board analyzer. The intended purpose of the board analyzer is to provide a recipe of a test controller to the next stage. This recipe is produced by analyzing the board to see what components are on the board and how the complete board as well as the individual components can be tested. A further responsibility of the board analyzer is to obtain information from the designer regarding test requirements including guidance from the test engineer about how the tests should be performed if there are any specific requirements.

To be able to analyze the board the board analyzer will need to access a data base containing information about all the allowed components on the boards in the system. For each component, information about such properties as if boundary scan or BIST is available is stored. If boundary scan is implemented on the chip information about how to access it should be stored. This includes information such as on what pads of the circuit we can find the boundary scan interface, what the scan chain length is, which boundary scan commands are available and so on. If BIST is implemented on the circuit, information about how to operate it will be provided. This information includes indication of if the BIST facility is invoked from boundary scan (and if so by what boundary scan command) or if the circuit has other means of doing it, such as raising the value on a separate control pad dedicated for BIST. It will also be included in the data base how to interpret the response from the BIST engine and how much time the BIST test requires. In practice if the BIST is operated from boundary scan we must know what the correct pattern to expect is from the boundary scan chain and after how many clock cycles it will be available.

The second part of the tool is the test controller generator. The purpose of this part is to generate a VHDL model of the test controller that is ready for simulation and/or synthesis. The test controller will then be synthesized based on the information obtained from the board analyzer and will therefore be specific to the requirements of each board. The generated VHDL model will be a structural interconnection of predefined VHDL blocks which are found in the “Controller Architecture” database seen in figure 4.1. This database contains predefined lego pieces that implements all necessary parts of the test controllers. This includes both the test controller core for handling overall control and lego pieces for performing specific test functions such as boundary scan interfaces, pseudo-random test pattern generators, test response compactors and different types of microprocessor interfaces etc.

All lego pieces need not be defined in a fixed way but can either be parameterized by VHDLs built-in ability to handle generic constructs or they can be modified with the UNIX m4[1] tool, which is intended to be used as a preprocessor step between our tool and the actual compilation of the model by a standard VHDL compiler. Experiments with such a backend have successfully been carried out for very simple test controller designs (i.e. a limited data base) and the resulting output was simulated in a standard Mentor Graphics environment. Attempts for

[1] m4 is a macro processor supplied as part of the UNIX operating system, and is also available for many other operating systems. It is intended as a general purpose front-end for computer languages, and thus fits very well for our VHDL preprocessing step.

synthesis were not made since the blocks were written in behavioral VHDL which could not be synthesized by the Mentor Graphics tool. For synthesis purposes the blocks have to be written to support the specific synthesis tool that is to be used, since different VHDL restrictions are defined by different vendors' synthesis tools. As we use Mentor Graphics we will target our block library implementation towards their tools, but from a research point of view there should be no limitation in that.

After the tool has created an initial implementation of a test controller, optimization is performed. Optimization is carried out with a transformational approach. This means that transformations are made from one correct (working) solution to another. Through these transformations the solution can be optimized for the specific trade-offs set up by the designer regarding the amount of required hardware and test speed (and eventually even other parameters). The optimization strategy also has the freedom of moving functionality between hardware and software, thus if the hardware cost is too high for the specific application the actual test controller implementation can degenerate to a very simple circuit where most of the functionality is provided by an on-board microprocessor.

The optimization has to be guided by an optimization strategy, and for this we have chosen the tabu search algorithm [Glo 93]. Tabu search is a heuristic optimization technique which means that we can get tractable computation times for optimizations that with exhaustive search would be exponential with the number of possible solutions. Tabu search is also based on neighborhood search which fits very well with our transformational approach since a neighbor can be defined as a solution with a single simple transformation applied to it. To escape getting trapped in cyclic loops when performing the transformations tabu search has the concept of tabus. A tabu is a dynamically inserted restriction that says that a certain neighbor (reached by a transformation) can not be chosen since the algorithm has recently performed that transformation.

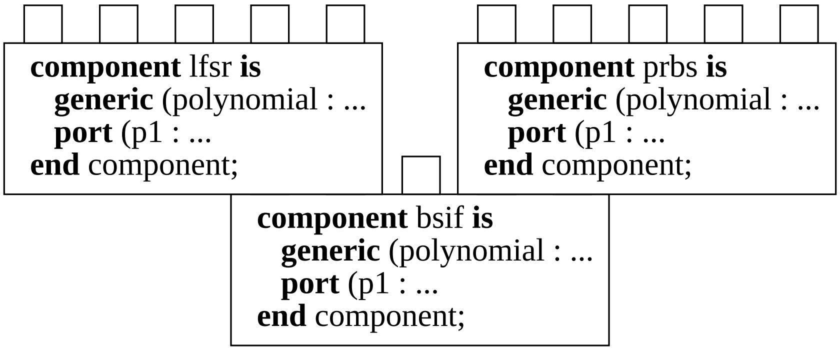
The third and final part of the tool is the test program generator. This generator generates the external data that should be provided with the test controller, if the test controller is constructed in such a way that it will read a machine code program from external memory. Usually however, our test controller will use a finite state machine for control, which means that no external test program is required for the test controller chip itself. If the test controller chip relies on a microprocessor for control this means that the control program for the microprocessor still has to be put in external memory by the test program generator. This also includes P-BIST test programs for the microprocessor. When the control programs for the test controller have been allocated space it still remains to place stored test data in external memory. This data includes stored test patterns, expected correct test vectors, topological information about the board such as what pads are bidirectional or three-state etc.

4.2.2 Lego Piece Approach

With the Lego piece model of the board test controller we mean the approach of assembling the test controller from a set of predefined lego pieces found in a data base. These lego pieces may be implemented in very different ways without affecting our general approach. The environment we are developing will however implement the lego pieces in VHDL. The description may be either structural and direct synthesisable by a synthesis tool or behavioral for simulation. This decision is fully up to the designer of the database. Actually prototyping of the test controller can be done with behavioral lego pieces. When the test controller is ready for synthesis the required lego pieces may be rewritten to structural VHDL (maybe with assistance from CAMAD [Pen 94]) to support synthesis by the desired tool.

With the Lego piece model of the BTC, we can synthesize a large spectrum of controllers emphasizing different aspects such as the relative degree of completeness and adaptability. In terms of completeness BTCs may range from simple BScan interfaces [Jar 91b] that are there only to support the on-board microprocessor which actually performs the overall control, to complete, autonomously running controllers [Mat 92] running test programs of their own. With different levels of adaptability a BTC can be targeted completely to a specific design, without any options to alter scan chain lengths etc., while at the other extreme the generality of the lego piece design can be utilized to produce fully programmable controllers, and can therefore be optimized to handle more than one board design.

We do not expect any major overhead with our lego piece model since implementing the BTC model with a lego approach is expected to result in the same kind of model description as a very structural approach would have. A sketch of our lego approach is drawn in figure 4.2, where three VHDL component declarations are put together. The idea of the picture is to show how a boundary scan interface, implemented as a lego piece, is combined with two lego pieces that implement a Linear Feedback Shift Register (LFSR) for test pattern compaction and a Pseudo Random Binary Sequence (PRBS) generator for test pattern generation.



```
component lfsr is
  generic (polynomial : ...
  port (p1 : ...
end component;
```

```
component prbs is
  generic (polynomial : ...
  port (p1 : ...
end component;
```

```
component bsif is
  generic (polynomial : ...
  port (p1 : ...
end component;
```

Figure 4.2: Sketch of the lego approach, in which a number of VHDL component declarations are drawn as lego pieces.

It should be noted that the VHDL components in figure 4.2 are far from a complete description even of the boundary scan part of the test controller. To make up a complete VHDL description we also need to describe the implementation of the component, instantiate the components, and finally interconnect them, which is here symbolized with the tags of the lego pieces, but is in VHDL done by means of signals.

For a more complete example of both what a lego piece may look like and how it is interconnected in the board test controller a code example is given below. This code was automatically generated by our experimental tool from a database containing amongst others the behavioral description of the LFSR (named LegoComp_3 by the tool). The example code is not complete but just lists the parts of the design that describes the LFSR. Where code has been skipped this is indicated with three dots (...).

```
USE work.std_logic_1164.ALL;
ENTITY LegoComp_3 IS -- LFSRx
  PORT (  Clock        : IN  STD_LOGIC;
          DataIn       : IN  STD_LOGIC ; -- Data Scanned In
          CaptureData  : IN  STD_LOGIC ; -- Request Data
          Data7        : OUT STD_LOGIC ; -- Data Bits in Output
          Data6        : OUT STD_LOGIC ;
          Data5        : OUT STD_LOGIC ;
          Data4        : OUT STD_LOGIC ;
          Data3        : OUT STD_LOGIC ;
          Data2        : OUT STD_LOGIC ;
          Data1        : OUT STD_LOGIC ;
          Data0        : OUT STD_LOGIC);
END LegoComp_3;
...
USE work.std_logic_1164.ALL;
ARCHITECTURE Behave OF LegoComp_3 IS
BEGIN
  p1: PROCESS
  VARIABLE poly : STD_LOGIC_VECTOR(8 DOWNTO 1) := “00000000”;
  VARIABLE tmp : STD_LOGIC;
  BEGIN
    WAIT ON CaptureData;

    IF CaptureData = ‘1’ THEN
      tmp := DataIn XOR poly(1);
      poly(7 DOWNTO 1) := poly(8 DOWNTO 2);
      poly(8) := tmp;
    END IF;

    Data7 <= poly(8);
    Data6 <= poly(7);
    Data5 <= poly(6);
    Data4 <= poly(5);
    Data3 <= poly(4);
    Data2 <= poly(3);
    Data1 <= poly(2);
    Data0 <= poly(1);

  END PROCESS;
END Behave;
...
-- Lego Piece id: 3
COMPONENT LegoComp_3 -- LFSRx
  PORT (  Clock        : IN  STD_LOGIC;
          DataIn       : IN  STD_LOGIC ; -- Data Scanned In
          CaptureData  : IN  STD_LOGIC ; -- Request Data
          Data7        : OUT STD_LOGIC ; -- Data Bits in Output
          Data6        : OUT STD_LOGIC ;
          Data5        : OUT STD_LOGIC ;
          Data4        : OUT STD_LOGIC ;
          Data3        : OUT STD_LOGIC ;
          Data2        : OUT STD_LOGIC ;
          Data1        : OUT STD_LOGIC ;
          Data0        : OUT STD_LOGIC);
END COMPONENT;
...
LegoPiece_3 : LegoComp_3 -- LFSRx
  PORT MAP (Clock       => Clock,
            DataIn      => Si2p4s0i3p1s0,
            CaptureData => Si2p4s1i3p1s1,
            Data7       => Si3p2s0i5p1s0,
            Data6       => Si3p2s1i5p1s1,
            Data5       => Si3p2s2i5p1s2,
            Data4       => Si3p2s3i5p1s3,
            Data3       => Si3p2s4i5p1s4,
            Data2       => Si3p2s5i5p1s5,
            Data1       => Si3p2s6i5p1s6,
            Data0       => Si3p2s7i5p1s7);
```

LegoPiece_3 in the example code refers to the LFSR compactor, which in the library is called “LFSRx” which can be seen from the comment lines. The name LegoPiece_3 is generated by the tool as a unique lego piece name. The same is true for the signals starting with a big “S” followed by a sequence of automatically generated, unique signal names derived from the numbers of the lego pieces and the cardinal number of the ports between which the signal is to be connected. These signals are also automatically declared by the tool[1], but this part of the example code has been cut. The code for the “LFSRx” lego piece, as it is stored in the database, is shown in appendix A.

With our approach of a stored library of lego pieces, the hardware complexity of a lego piece has to be calculated and stored along with the lego piece in order to allow efficient hardware cost optimization. If such a cost estimation would not be stored, it would be necessary to synthesize the whole test controller for each change in lego piece for each change of the design, something that would make the optimization too complex and slow. The disadvantage of estimating each lego piece in isolation is that it can be argued that if the whole design is given to the synthesis tool at once it may be possible for the synthesis tool to perform sharing between different lego pieces as well as optimize some inter lego piece communication away. However, providing the complete design at once to the synthesis tool is supported, it will only make the implementation somewhat more cost effective than what was estimated. We do not expect any problems with this approach. Experimentation will be carried out as soon as we have a working tool.

The optimization that can be done will mainly be selection of lego pieces for the implementation as well as parameter selection for generic lego pieces. This selection can, however, be quite complex if we have

[1] It should be noted that the simple experimental prototype implementation of the tool did not handle array interconnections between lego pieces. This makes the code unnecessarily long and complex, but is a straight forward programming exercise to correct.

much freedom of choice as well as a complex optimization criteria which considers several aspects of the design. The basic idea is that tabu search will, through neighborhood search, find transformations to apply in order to optimize the BTC. In figure 4.3 three BScan interfaces are shown.

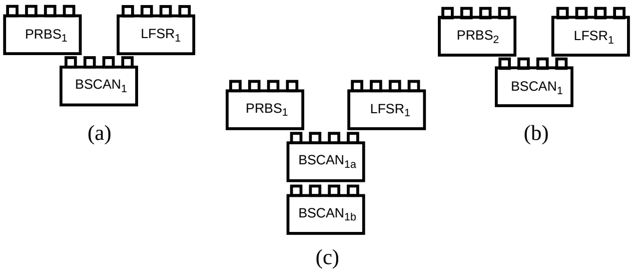


Figure 4.3: Boundary Scan interface undergoing transformations

All of the BScan interfaces provide both a Pseudo Random Binary Sequence (PRBS) pattern generator, and a Linear Feedback Shift Register (LFSR) pattern compactor, the difference being only with which lego pieces they are implemented. In figure 4.3a the implementation chosen for the respective lego pieces are called “PRBS$_1$”, “LFSR$_1$” and “BSIF$_1$” respectively. This means that the specific implementation with cardinal number one has been chosen for implementation. For figure 4.3b the design has undergone a transformation in which “PRBS$_1$” has been substituted by “PRBS$_2$”. This is a straight forward replacement of a lego piece for another that can perform the same job at a more attractive cost given some cost function. The third example in figure 4.3c shows how the boundary scan interface “BSIF$_1$” has been replaced by two lego pieces “BSIF$_{1a}$” and “BSIF$_{1b}$” that together implements the same functionality as “BSIF$_1$” does alone. The advantage of this is that considering a complete design the possibilities for different functions to share hardware may increase, i.e. assume that another boundary scan interface is built by a “BSIF$_{1a}$” and a “BSIF$_{1c}$”. In such a case “BSIF$_{1a}$” may be shared while the whole implementation can not if neither “BSIF$_{1b}$” nor “BSIF$_{1c}$” is general enough to replace the other. There are more transformations that can be performed and in the example no parameterized lego pieces are shown.

4.3 The Tool

As has been mentioned a simple prototype tool was implemented early during this work. This was done before it was decided upon a more specific design representation of the test controller (see chapter 5). The purpose of the prototype implementation was to demonstrate that we can synthesize working test controllers from a very abstract lego piece representation of the controller.

4.3.1 Building Blocks

There is a large freedom in our test controller model to chose any types of lego pieces for inclusion in the test controller. This means that our modeling of the test controller can be adapted both to upcoming test tasks and techniques as well as be extended in directions that we have not directly considered in our work. The main functions that we have considered for this work are boundary scan testing with different types of test pattern generators and compactors. We have also considered BIST in general both when it is initialized through the boundary scan interface and when it requires other means of initialization which can then be done by having special lego pieces send the initialization signals. Further we have CPU interfaces for the purposes of handling CPU BIST testing as well as movement of functionality from the hardware implementation of the test controller to a software implementation within the microprocessor. Further we have lego pieces not directly related to the testing itself, but rather for additional support of the hierarchical test concept. These lego pieces may include identification of a board, which means that the board reports what kind of board it is and what revision status it has. There may also be lego pieces to handle the powering up/down of a board so that maintenance software running at a higher hierarchical level can take the board on or off line automatically. There can also be lego pieces for watch dog timers which are used to interrupt operation after a certain time has elapsed if no result was produced. These can be used to guarantee that the test controller does not get stuck and locks infinitely because of some external problems (e.g. a CPU BIST program that never returns because of a failing ROM memory or jammed address bus).

5 Test Controller Representation and Synthesis

5.1 TDG Representation

The Test Design Graph (TDG) is the graphical notation we use to represent a test controller. A TDG is capable of capturing three individual test controller properties; the test tasks to be performed by the test controller, the control program (or FSM) that is to control the test activities and the hardware required for the implementation. Roughly the last part corresponds to the lego pieces discussed earlier. The finite state machine is used to coordinate the operation of the lego pieces, and the test tasks describe which lego pieces are necessary and how they are to be used.

5.1.1 Introduction to TDG

A TDG consists of two parts, a control part that handles the overall control of the test controller and a test path that implements the functions that are performed by the test controller. The control part is based on Petri net theory [Pet 81]. A Petri net is a graphical and mathematical representation of systems. It can be used to capture the partially ordered events of the given system. Thus Petri nets can be used to formally describe how control flows from one state to another.

An example of a very simple Petri net is given in figure 5.1. This Petri net consists of five places which are the circles marked with a “P” and a number. In a Petri net one or more places can be active at the same time. An active place means that the system is in a specific state, and a place is active iff there is a token in the place[1]. In figure 5.1 there is a token in place P1 (the token is shown as a small grey circle within the place). One or more places can be deactivated when one or more new places are activated. This is done via firing transitions which are shown as black, horizontal lines on the interconnection between the places in figure 5.1. A transition will “consume” all the tokens in its input places (as can be seen in the figure a Petri net is a directed graph) and at the same time put tokens in all its outgoing places. In figure 5.1 there are only transitions with single inputs/outputs. Therefore there will always be only one token in the net, since there is only one token initially.

Transitions can be fired only when a transition is activated. In TDG a transition is activated if there is a token in each of its incoming places and the conditions associated with the transition are fulfilled. Examples of such conditions can be that a transition is activated when a test has finished execution or when a watch dog timer signals that time is up. Thus it is possible to make decisions in a TDG, by having fan-out on a place to several transitions with different conditions. In figure 5.1 there is a choice for the token to move between P_1 and P_2 or between P_1 and P_4 depending on the conditions associated with the T_1 and T_4 transitions.

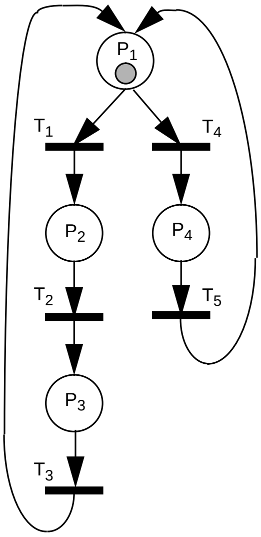


Figure 5.1: A small Petri net with four places and five transitions.

[1] This assumes a safe Petri net. A safe Petri net can not have more than one token in a place at the same time.

Based on Petri net theory a design representation for hardware design has been developed [Pen 87][Pen 94]. This design representation is called ETPN for Extended Timed Petri Nets. The basic idea is to represent the control part of a hardware design by a Petri net, and to add a data path that describes the data manipulating functions, such as adders and registers etc. The ETPN design can then be synthesized to RTL-level designs in a straight forward manner, where the control part is synthesized to a finite state machine and the nodes in the data path can be taken directly from the component library of the given technology.

An example of an ETPN graph can be seen in figure 5.2. The control part is the Petri net to the left in the figure, while the data path can be seen to the right. In this example the values of the two input registers R_1 and R_2 are multiplied to form a product in output register R_3. Each of the nodes in the data path is assumed to correspond to a cell in a module library that will eventually be used for implementing the data part on real hardware. Thus the library of nodes for the data path is specific to the technology that will be used to implement the circuit.

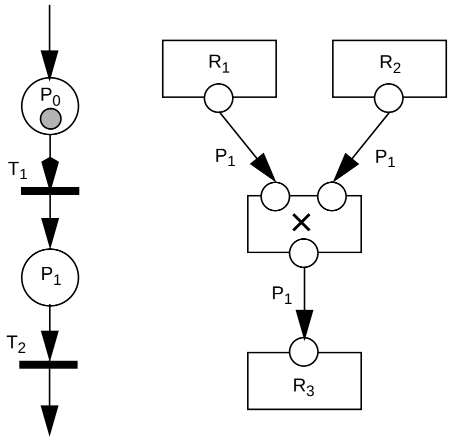


Figure 5.2: An example ETPN performing a multiplication operation.

Since an ETPN is a synchronous representation, transitions in the control part can be fired on every clock cycle given that the conditions associated with the transitions are fulfilled. Although it is not the case in this simple example, conditional expressions are usually formed by the data path and associated with transitions in the control part, thus the calculations in the data path can control the flow of the control part. Control of the data path is obtained by enabling the vertices in the data path. In this example the multiplication is performed when state P_1 is activated since P_1 is associated with all three vertices that connects to the multiplier. A formal definition of ETPN syntax and semantics can be found in [Pen 94].

The purpose of the ETPN representation is to get an easily visualized graphical representation of hardware that can be reasoned about. For ETPN there is a tool, CAMAD [Pen 94], that can be used to compile VHDL or ADDL [Fje 92] to ETPN and ETPN to VHDL, as well as optimizing the Petri net in terms of performance (execution time) and implementation cost (area).

In this thesis we present yet another design representation, the Test Design Graph (TDG), which is based on the ideas developed for ETPN, but that better captures our specific needs when we want to reason about test controller designs. The purpose of the ETPN and the TDG is quite different, where ETPN is used as a general target representation for compiled VHDL or ADDL, the TDG representation is only supposed to capture a test controller design, and is not a target for compilation from VHDL or ADDL, while it still should be a representation from which VHDL (or other high level description languages) should be able to be generated. For the TDG representation it is more important to reason at a higher level than it is for the application areas that ETPN was developed for. Optimization in the TDG and test controller generation case is rather a question of choosing between alternative ways of implementing bigger functional blocks than tuning the design at the register transfer level. Thus ETPN should be seen as a lower level design representation than the TDG, and it would be feasible to compile TDG representation into ETPN representation after optimization and reasoning have been performed at the TDG level.

5.1.2 Formal Definition of TDG

The TDG is based on two interconnected subgraphs. The first is the *control part*, captured as a Petri net which holds all necessary information about test controller test scheduling. The control part is defined in definition 5.1.

Definition 5.1: A control part, Γ, is a four-tuple, $\Gamma = <\mathbf{S}, \mathbf{T}, \mathbf{F}, M_0>$, where

$\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ is a finite set of *control states* (*places*);
$\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ is a finite set of *transitions*;
$\mathbf{F} \subseteq (\mathbf{S}\times\mathbf{T})\cup(\mathbf{T}\times\mathbf{S})$ is a binary relation, the *flow relation*.

M_0: S⟶{0,1} is an *initial marking* function.

A control part consists of interconnected control states (places) and transitions. One or more control states are active at a time. This part contains the information used when the finite state machine that controls the test controller is to be synthesized. Alternatively it can also be used to generate a microcode program for a more flexible, programmable test controller or even a machine code program if the test controller is developed as a slave device to a microprocessor.

The next part of the TDG is the test path. This is the part of the representation that indicates what lego pieces (called implementation nodes) are to be used within the test controller. It also shows how the lego pieces are interconnected with each other, and how the lego pieces connect to the pads of the test controller (called interface nodes) as well as what purpose the lego piece is there fore (each service the test controller can perform is referred to as a test task). The formal definition of the test path is given in definition 5.2.

Definition 5.2: A test path, Λ, is a six-tuple, Λ=<**TA**, **IM**, **IN**, **NP**, **BN**, **BT**>, where

TA = $\{TA_1, TA_2, \ldots, TA_n\}$ is a finite set of *test tasks*;

IM = $\{IM_1, IM_2, \ldots, IM_m\}$ is a finite set of *implementation nodes*;

IN = $\{IN_1, IN_2, \ldots, IN_p\}$ is a finite set of *interface nodes;*

NO = **IM**∪**IN** is the set of *nodes* in the design graph;

NP = $\mathbf{NP}(NO_1)\cup\mathbf{NP}(NO_2)\cup\ldots\cup\mathbf{NP}(NO_q)$ where $\mathbf{NP}(NO_i)$ is the set of *node ports* associated with *node i*;

BN $\subseteq (\mathbf{NP}\times\mathbf{NP})=\{<NP_1,NP_2> \mid NP_1\in\mathbf{NP}(NO_i), NP_2\in\mathbf{NP}(NO_j), NP_1\cup NP_2\not\subset\mathbf{NP(IN)}\}$ is a finite set of arcs that connects two *node ports*;

BT $\subseteq (\mathbf{TA}\times\mathbf{NO})$ is a finite set of edges that associates *test tasks* with *nodes*.

An example of the graphically notation used for visualizing a test path is shown in figure 5.3. It has been indicated in the figure how the graphical objects relates to the objects defined in definition 5.2.

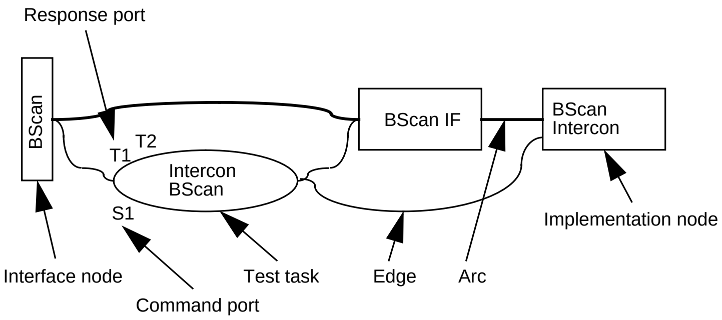


Figure 5.3: Subset of a test path with additional explanations

The Test Design Graph is the over all data structure that incorporates both the control part and the test path. It also serves the purpose of integrating them with each other through so called command ports and response ports which are the means through which the control part commands the test path, respectively gets feedback from the test path.

Definition 5.3: A Test Design Graph, TDG, is a six-tuple, TDG=<Γ, Λ, **CP**, **RP**, **BC**, **BR**>, where

Γ = <**S**, **T**, **F**, M_0> is a *control part*;

Λ = <**TA**, **IM**, **IN**, **NP**, **BN**, **BT**> is a *test path*;

CP = $\mathbf{CP}(TA_1)\cup\mathbf{CP}(TA_2)\cup\ldots\cup\mathbf{CP}(TA_n)$ where $\mathbf{CP}(TA_i)$ is the set of *command ports* associated with *test task i*;

RP = $\mathbf{RP}(TA_1)\cup\mathbf{RP}(TA_2)\cup\ldots\cup\mathbf{RP}(TA_m)$ where $\mathbf{RP}(TA_i)$ is the set of *response ports* associated with *test task i*;

BC: $\mathbf{S}\rightarrow\mathbf{2}^{\mathbf{CP}}$ is a mapping from *control states* in the *control part* to a set of *command ports* in the *test path*;

BR: $\mathbf{RP}\rightarrow\mathbf{2}^{\mathbf{T}}$ is a mapping from *response port*s in the *test path* to a set of *transitions* in the *control part*.

5.1.3 TDG Semantics

The semantics of the control part is defined below:

Definition 5.3: Given a Test Design Graph, its behavior is defined as follows:

1. A function M: **S** ⟶ {0, 1} is called a *marking* of Γ. A marking is an

assignment of tokens to the control states.

2. Initially there is a token in each control state S_i such that $M_0(S_i)=1$ as defined by the initial marking M_0.

3. A transition T is said to be enabled at a marking M iff $M(P_i)=1$ for every P_i such that $<P_i,T> \in$ **F**

4. A transition may be fired when it is enabled and the guard condition is true. If a transition has more than one guard condition, an AND operation is applied to them; therefore, all guard conditions must be true in order for the transition's guard condition as a whole to be true.

5. Firing an enabled transition T removes a token from each of its input control states and deposits a token in each of its output control states.

6. When a control state, S, holds a token the command ports on one or more test tasks associated with the control state is true. If a command port has more than one control state associated to it an OR operation is applied to them.

7. A response port, RP, can be activated iff its corresponding command port, CP, is active.

8. An enabled response port, RP, is deactivated iff the command port, CP, that activated the response port is deactivated.

Related to the formal definition of semantics is the very special facility of a TDG to reset itself. This is not directly part of the semantics, but is possible since the test path allows basically any operation. This means that it is fully allowed to let an implementation node in the test path reset the whole test controller. This means that the TDG may be re-initialized by itself by activating a reset function.

In figure 5.4 a simple test design graph is shown. This TDG describes a test controller that is capable of interacting with a test bus from which it receives its commands of what to do as well as reports success/failure status for the tests that it performs. Further it is capable of carrying out two different tests; a boundary scan interconnect test and a P-BIST test as the test path graph has test tasks for these two actions. Finally an overall reset of the test controller can be performed via the "Reset" test task shown at the bottom of figure 5.4.

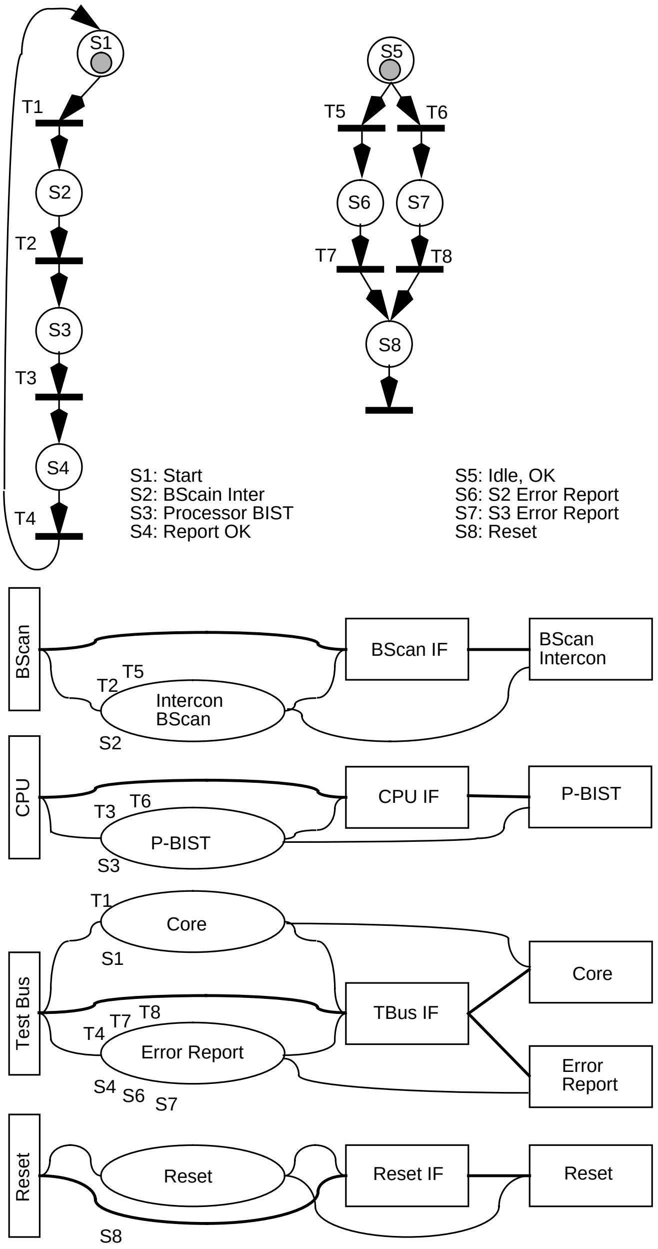


Figure 5.4: A simple TDG example.

In the control part all states have been marked with “S*x*” where *x* is used to enumerate the states and ranges from one to eight. Similarly all transitions have been marked with “T*y*” where *y* enumerates the transitions. There is an association between the command ports in the test path and the states in the control part. To indicate this the states that control command ports are printed under the test task whose command port it controls. If more than one state is put under the same test task this means that the test task has the same number of command ports as the number of states put there, where each command port is controlled by its own state. In the same manner the response ports have their associated transitions printed over the test tasks.

When a TDG is initiated (or when it has been reset) there are tokens in the initial states. In the TDG given in figure 5.4 the initial states are states S1 and S5. This activates the Core test task which awaits commands to arrive from the test bus. In this model it only has one command defined, and that is the command to start testing. When such a command is received on the test bus the core will signal this on the response port and thus enable transition T1 which is associated with it. As S1 now holds a token and T1 is activated, T1 is fired which means that the token is removed from S1 and put in S2 instead.

When S2 is activated boundary scan interconnect test starts since S2 is controlling the command port of the “Intercon BScan” test task. The outcome of this test can be either a success in which case testing is supposed to proceed with the next test, or a failure where testing will have to be aborted and the error reported. To let the test task indicate these two possible outcomes it has two response ports, one for success and one for failure. We assume this time that the testing succeeds, and thus T2 is enabled by the response that is associated with it, which means that state S2 is deactivated and S3 is activated. If testing fails, however, transition T5 is enabled instead. When T5 is enabled S5 is deactivated and S6 is activated. S6 will invoke the error report test task.

The error report test task has three command ports, one for reporting success (which is reported when all test have been performed successfully) and two for reporting failure, i.e. one failure report for each of the two tests that are performed in this example. After having reported the failure control proceeds from S6 to S8, where S8 controls the “Reset” test task which performs an overall reset of the test controller. This means that the test controller is put in its initial state with tokens in S1 and S5 only, and the test controller is thus ready to accept a new command via its “Core” test task that is re-enabled by state S1.

Going back to the case of successful boundary scan test, S3 now holds control and thus starts the “P-BIST” test task which will report success or failure in the same way as the boundary scan interconnect test. For success S4 is activated and the “Error Report” test task is here used to report success to the test bus instead of error as was the case when a report was requested from S6. After the success report state S1 regains control and the test controller is ready for the next command. If the P-BIST test failed, control would have gone to state S7 in the same way as failure of the boundary scan test gave control to S6. From S7 an error would be reported. As S7 has a different command port on the “Error Report” test task than S6, an error report different from the S6 error can be given when the P-BIST test fails. Finally the test controller is again reset in state S8 and the test controller is again ready for the next command.

5.2 TDG Transformations

Three basic transformations have been defined to perform transformation on the *implementation nodes*. They are described below.

5.2.1 Substitution

A transformation of type substitution is employed to exchange a set of implementation nodes for an alternative set that performs the same function but with a different hardware realization. Graphically a substitution operation is depicted in figure 5.5. There, a type A realization of a BScan interface is substituted for the alternative type B implementation. The type B implementation, however, requires a special parameter node to be functionally equivalent. Therefore the substitution must handle sets instead of single implementation nodes. In the example a set with only one member is substituted for a set with two members. It is required that each set is connected (by arcs) in such a way that no two disconnected subsets can be formed within the set. The two sets must also contain the same arcs to the rest of the test path. An algorithmic description of how the transformation affects the test path is given below:

1. Find a set of implementation nodes $\mathbf{IM_1}$ to be replaced by another set $\mathbf{IM_2}$.

2. Check that all implementation nodes in $\mathbf{IM_1}$ have exactly the same edge connections to test tasks. Otherwise sharing is done within the set $\mathbf{IM_1}$ and substitution can not be performed.

3. Replace $\mathbf{IM_1}$ with $\mathbf{IM_2}$.

4. All external arcs are preserved, and all nodes within $\mathbf{IM_2}$ get the same edges as the nodes in $\mathbf{IM_1}$

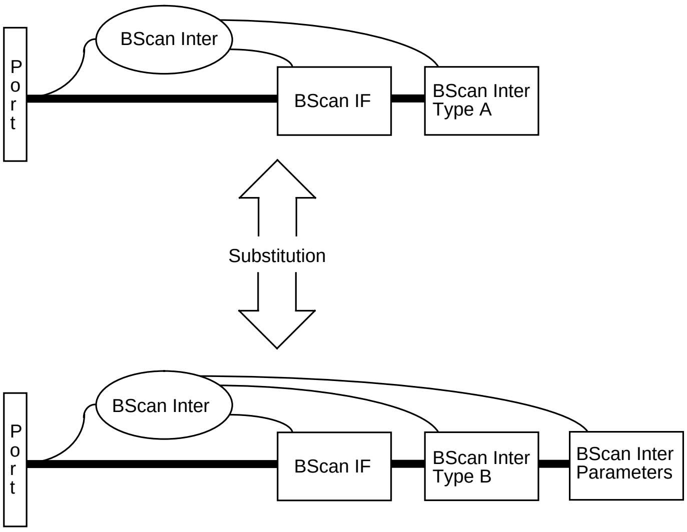


Figure 5.5: Example of a substitution transformation.

5.2.2 Merging

Merging provides the ability for two or more test tasks to share common hardware resources in the test path. The motivation for this ability is a reduction in hardware cost. Merging, is used to merge two implementation nodes to one. There is a reverse transformation to merging which is called splitting. The split transformation splits a merged node into two distinct nodes.

The merge transformation will merge two implementation nodes of the same type. A merge operation has been depicted in figure 5.6 and the merge operation corresponds to a move downwards in the figure. Unlike the substitution transformation, a merge (and also split) transformation always operates on single implementation nodes. An algorithmic description of how the transformation affects the test path is given below:

1. Identify two implementation nodes IM_1 and IM_2 to be merged.
2. Copy all node arcs from IM_2 that are not on IM_1.
3. Copy all edges from IM_2 to IM_1.
4. Remove IM_2 along with its arcs and edges.

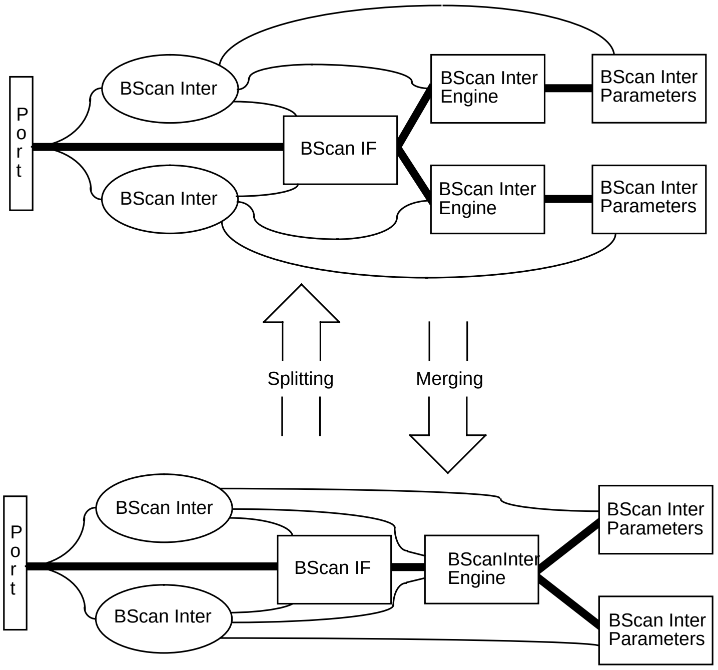


Figure 5.6: Example of a sharing transformation.

5.2.3 Splitting

The second transformation to implement the sharing concept is splitting. It is basically a reverse merge transformation in which a single implementation node is split into two distinct nodes. After a node has been split, the newly created node will always be associated with exactly one test task. Thus if an implementation node is shared by N test tasks, after the split the original node will be shared by N-1 test tasks while the N:th test task will have its own implementation node. To split in other ways successive split and merge transformation will have to be performed. An algorithmic description of how the transformation affects the test path is given below:

1. Identify an implementation node, IM_1, that is to be split. It is required that this node has at least two edge connections since otherwise it is not shared.
2. Choose which edge, BT_1, that is to get its own implementation node.
3. Make a copy of IM_1 and call this copy IM_2.
4. Disconnect edge BT_1 from IM_1 and reconnect it to IM_2.
5. All arcs connecting nodes to the left of IM_1 to IM_1 are duplicated to also connect to IM_2.
6. All nodes to the right of IM_1 that are connected to the edge BT_1 get arc connections to IM_2. If such a node does not have any edge left that points to the same test task that there exists an edge for M_1 for the arc between IM_1 and this node is removed.

5.3 Test Controller Parameters

When designing a TDG, there is one essential piece of information that is not captured graphically. This refers to the different parameters that are necessary for fully describing the operations. Most operations are not fully described for optimization or synthesis without additional parameters. An example of parameters for a boundary scan interconnect test task would be parameters describing the length of the boundary scan chain, the properties of the boundary scan chain (i.e inputs/outputs/bidirectionals etc.). These test task related parameters will have to be transformed into parameters suitable for association with the implementation nodes, since otherwise it will neither be possible to know what transformations can be allowed nor how to finally synthesize the nodes.

5.3.1 Parameters and Transformations

Associated with each node is a set of parameters. To exemplify this consider a part of a test controller that will do boundary scan interconnect test on a very simple, short boundary scan chain with two directly interconnected ICs, figure 5.7.

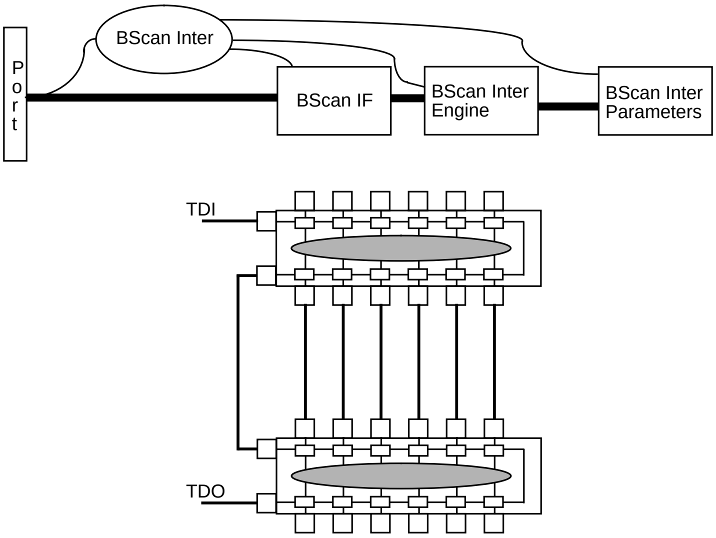


Figure 5.7: A boundary scan interconnect example.

The information the TDG would have to capture about this design is listed in table 5.1. First of all the type of test that is to be performed has to be captured in some parameter. In this case we simply call this parameter "Test Method" and associates "Walking Zero" with it, which means that the interconnection test that we want to perform is the walking zero test. The test method parameter belongs to the test task, which means that the parameter is fixed regardless of the implementation choices. Other parameters are related directly to the boundary scan chain and describe in detail how the scan chain will be handled. These parameters are associated with the "BScan Inter Parameters" implementation node, and consist of the "Scan Length" parameter which contains the length of the boundary scan chain, the "Ignore" parameter indicating which cells in the boundary scan chain should be masked out during testing. In this case cells 1 to 6 and 13 to 18 should be masked out since they are not used for interconnection within the scan chain. Further we have the “Runs” parameter which describes how many walking zero patterns will be used, which in this example is six and equals the number of interconnected pads between the circuits.

TDG Object	Parameter	Value
Port	none	
BScan Inter	Test Method	Walking Zero
BScan IF	none	
BScan Inter Engine	none	
BScan Inter Parameters	Scan Length	24
	Ignore	{1:6,13:18}
	Runs	6

Table 5.1: Parameters associated with an example BScan interconnect test.

It can be seen that no parameters need to be associated with the “BScan Inter Engine” implementation node. This makes this node more general and allows for sharing as shown in section 5.2.2. We do not claim that the table contains all necessary parameters, and what is necessary depends also on the reasoning capabilities of the environment, which we have not built yet. An example of this is whether it will be necessary to explicitly indicate which cells in the boundary scan chain that are interconnected to each other, or if this can be derived automatically from a netlist. If it can be automatically derived, table 5.1 would indicate what is given by the designer, and eventually the table will be refined automatically by the tool to contain further derivable details.

5.3.2 Cost Function

The cost function used when optimizing the TDG requires calculation of both the hardware cost of the TDG and the time required for the test operations in the TDG.

Hardware Cost

The hardware cost of the TDG is the sum of the cost of the control part and the test path.

The cost of the control part is taken as an estimation of the complexity of the Petri net. Thus the number of arcs is counted instead of places/ transitions since the complexity is more closely related to the number of arcs. The formula looks as follows:

$$\mathrm{Cost}(\Gamma)=\sum_i \left[\, |S_i^{\bullet}| + |{}^{\bullet}S_i| \,\right] + \sum_j \left[\, |T_j^{\bullet}| + |{}^{\bullet}T_j| \,\right]$$

where

${}^{\bullet}S_i = \{\, T \mid (T, S_i) \in \mathbf{F} \,\}$, the pre-set of $S_i \in$ **S;**

$S_i^{\bullet} = \{\, T \mid (S_i, T) \in \mathbf{F} \,\}$, the post-set of $S_i \in$ **S;**

${}^{\bullet}T_j = \{\, S \mid (S, T_j) \in \mathbf{F} \,\}$, the pre-set of $T_j \in$ **T;**

$T_j^{\bullet} = \{\, S \mid (T_j, S) \in \mathbf{F} \,\}$, the post-set of $T_j \in$ **T;**

The cost of the test path is calculated as:

$$\mathrm{Cost}() = \sum_i \mathrm{Cost}(IM_i) + \sum_j \mathrm{Cost}(IN_j) + \sum_k \mathrm{Cost}(TA_k)$$

where

$\mathrm{Cost}(IM_i)$ = a function over the type of the node and its associated parameters.

$\mathrm{Cost}(IN_j)$ = the number of pads in IN_j

$\mathrm{Cost}(TA_k) = |\mathbf{CP}(TA_k)| + |\mathbf{RP}(TA_k)|$

The hardware cost of an implementation node is calculated by a special function associated with the type of the implementation node. This function differs for each type of node and is specially designed to take parameters of the actual node type into account when performing the calculation. Since the parameters associated with different types of nodes differ, no general cost formula has been developed.

Execution Time

The cost related to the execution time of the test task is estimated by analyzing the Petri net to see what test tasks that are activated. For each test task that is found to be activated a time can be estimated by a cost function associated with the test task in much the same way as with the hardware cost function.

A typical Petri net for a small test can be seen in figure 5.8. We want to estimate the execution time of either the test operation (the left most path in the left Petri net) or the identification operation (the right most path in the left Petri net). We do not consider the error exception Petri net (the right one) since our estimation of time will only measure the normal case in which tests are assumed to be successful. We make this choice since most tests are successful (if not there is a quality problem).

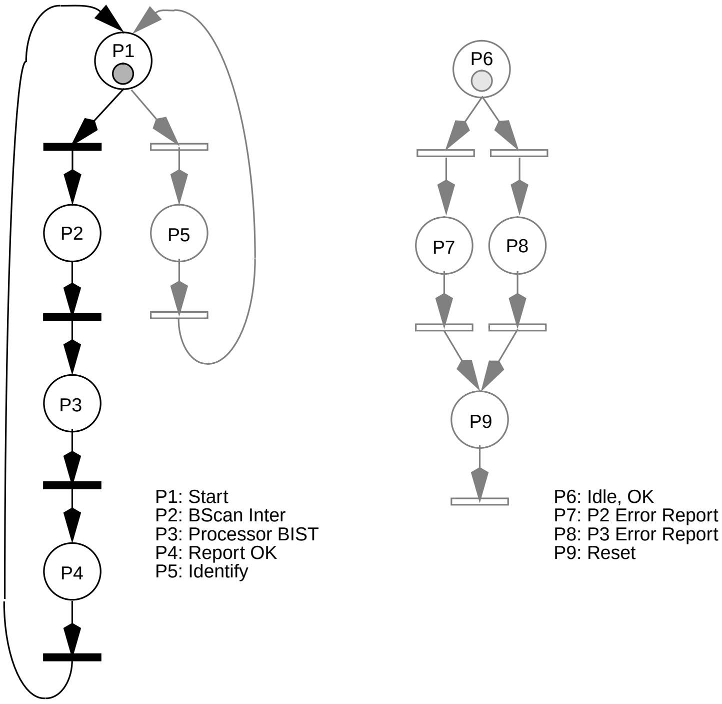


Figure 5.8: A typical control part for a very simple TDG.

As the execution time can be difficult to estimate in situations where there are loops involved in the control part bounds have to be put on the maximum number of iteration each loop will be allowed to take.

6 A Board Example

A board example has been developed. This example contains important components extracted from a real telecommunications board designed by Ericsson. The complete board design is not included though, since that would be unnecessarily complex and too detailed for the exemplification purpose.

6.1 Board Example

The heart of the board is a Motorola M68360 QUICC (Quad Integrated Communication Controller) 32-bit microprocessor. It is a processor originally derived from the Motorola 68020 design but targeted for embedded control. The processor can be seen in many real telecom applications. It also has the benefit of containing a standardized IEEE 1149.1 BScan interface [IEEE 90].

The external devices connected to this processor are assumed to be designed for 8 and 16-bit data busses. Since the M68360 processor has dynamic data bus sizing between 8, 16 and 32 bits this kind of devices can easily be handled. Nonvolatile memory for the processor is stored in a 64 kilobyte PROM memory contained in one single chip with a 16-bit data bus. RAM memory is constructed from two 128 kilobyte DRAM chips that together make up 256 kilobyte storage with a 16-bit data bus interface.

Application specific hardware implementation is constructed from one FPGA (a Xilinx) and one ASIC. Both of these contain boundary scan. The FPGA is connected directly to the data bus through a 16-bit interface, while the ASIC lacks proper bus interfacing and is therefore connected to the bus through a bidirectional three-state driver, a BScan enabled 74BCT8244 chip. This gives the ASIC only 8-bit access to the bus. To load the Xilinx FPGA chip a dedicated serial ROM is employed. Loading is not performed at power up since we want to use the BScan logic that is assumed to be present in our Xilinx model at power up.

Later when initial BScan testing has been performed, the Xilinx is loaded. A sketch of the board can be seen in figure 6.1.

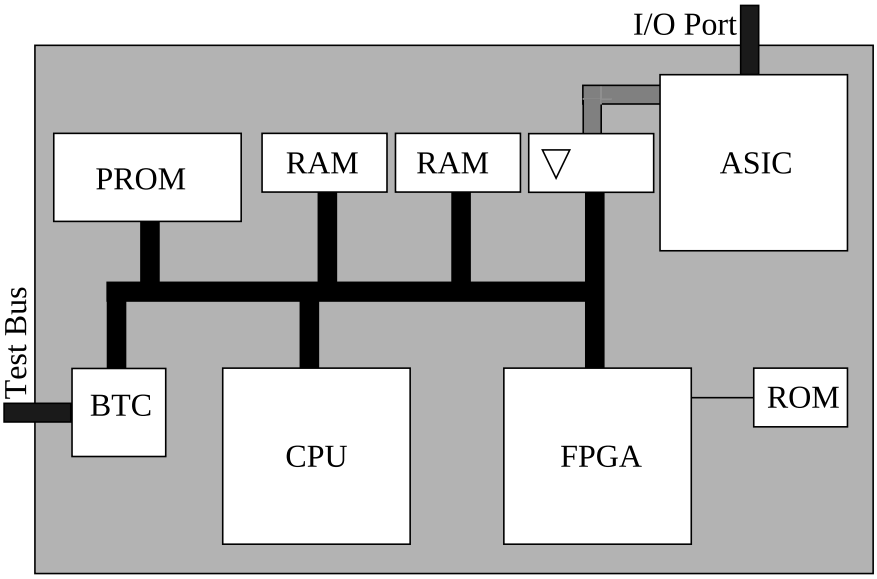


Figure 6.1: Example board derived from a real telecom application.

6.2 Test Requirements

The first step to take in designing the test controller is to perform a test requirements analysis. In this step we decide what we want to test. In the example given in figure 6.1 the internals of the ASIC, RAM, ROM, Xilinx (with its dedicated ROM) and the CPU should be tested. Further we want to test the interconnections on the board. We must also decide the test method that is to be used to perform the test. Here we choose to use the BIST capability of the ASIC for its testing, while the RAM memory is tested by the Mach scheme [Goo 91]. The ROM is tested by calculating its checksum. When we come to the Xilinx chip we have no BIST scheme so the only way to test it is to do it functionally through P-BIST where we let the microprocessor exercise it to see that it is both correct and also correctly loaded from its serial ROM memory. In this way we also get an indirect test of the Xilinx dedicated ROM memory, since if it would not work properly the Xilinx would not have been correctly loaded. The final chip to test is the CPU itself, which is also tested through P-BIST. We let it execute some code to exercise some of its instruction set as well as its interfaces to the environment.

After the individual components on the board have been tested their interconnects are to be tested. This will be done by boundary scan interconnect test. Since all major components on the example board is equipped with boundary scan, except for the memory chips, this will work efficiently.

Finally we want to add an identification feature to the test controller. This means that the test controller will be able to identify itself upon request by sending a special code on the test bus. A summary of the test requirements can be seen in table 6.1.

Circuit to Test	Test Method
ASIC	BIST
RAM	Mach Test
ROM	Check Sum
Xilinx (with ROM)	P-BIST
CPU	P-BIST
Board Interconnect	Boundary Scan
Identification	-

Table 6.1: Test requirements for the example board.

6.3 An Initial Test Design Graph

The designer now has to design an initial TDG. This graph will be the origin from which the tool will be able to perform optimizations. As usual the graph is divided into a test flow, which can be seen in figure 6.2, and a control part, which can be seen in figure 6.3.

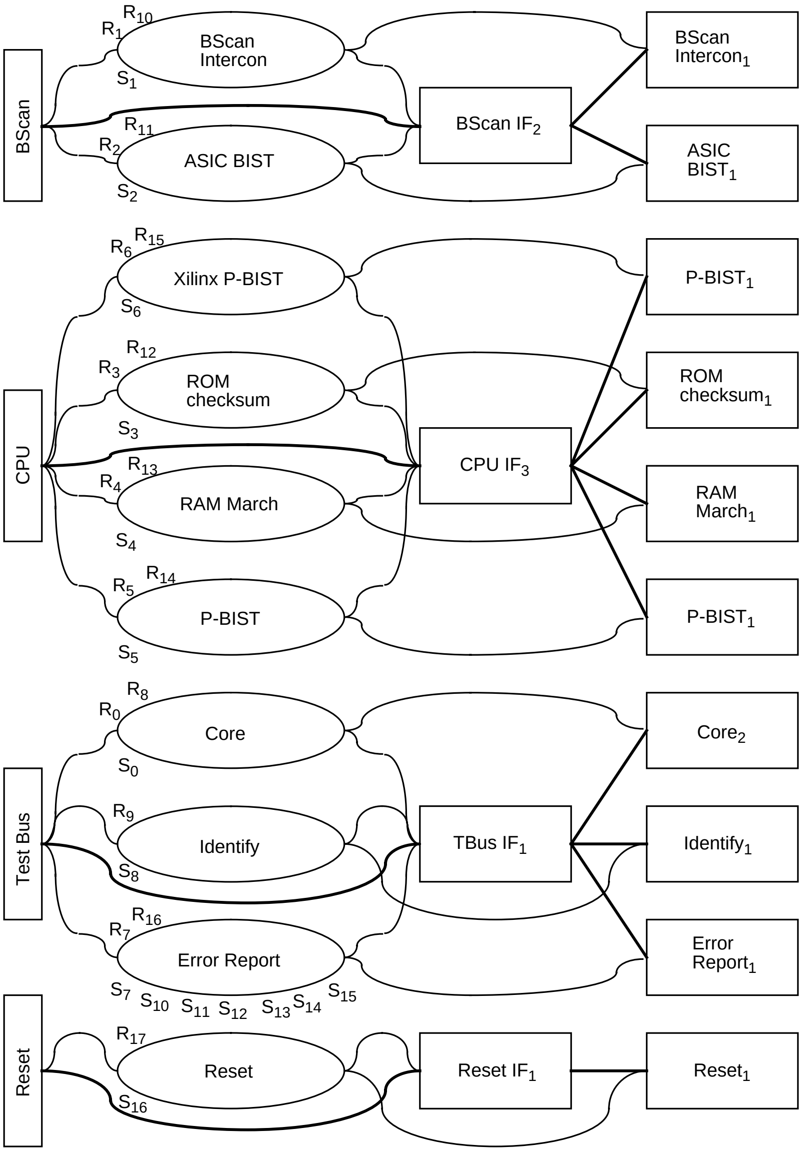


Figure 6.2: First version test path for the board example.

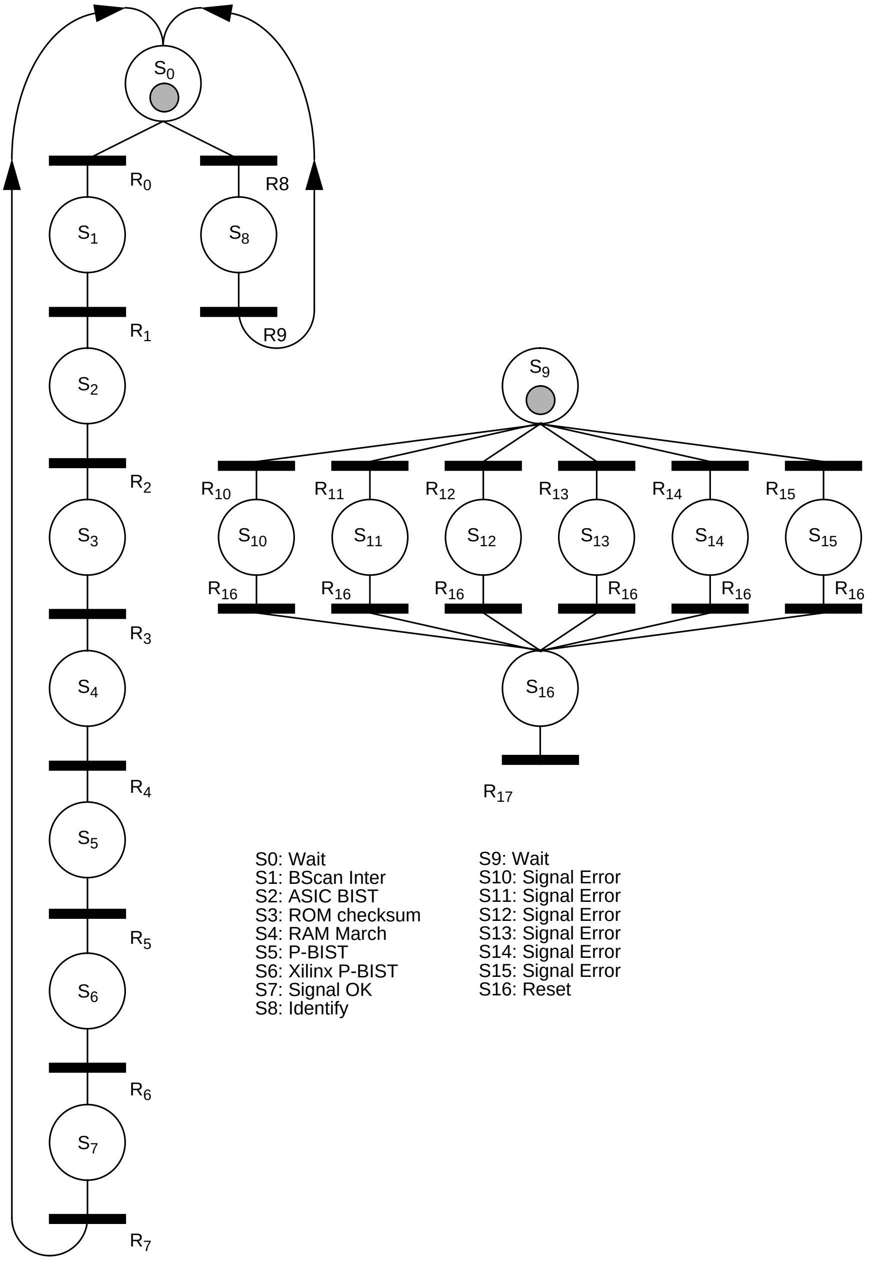


Figure 6.3: Control part for the board example.

In figure 6.2 we can see that four different interface nodes have been introduced on the test controller. These are BScan, CPU, test bus and reset ports. Since we assume that we connect the boundary scan chain on the board into one single chain we only need one BScan interface node. The CPU and test bus interface nodes are for communication with the microprocessor and the test bus respectively and they must be specially designed to operate with the CPU and test bus at hand. Finally a reset node is incorporated. This node is used to signal reset to the board, which means that it should reset the whole board including the test controller itself. This is used when a fault is found during testing, and solves the problem of leaving the board in an undefined state.

The second layer in the TDG (and finally the BTC) that communicates with the board is the left most implementation nodes. Each interface node is controlled by a corresponding implementation node, which provides the logic interface. Just as with the interface nodes themselves, the implementation nodes depend also on the device with which they are to interface. Implementation nodes can be changed during the optimization step, whereas interface nodes can not. This difference comes from the fact that there can be several implementation alternatives of one implementation node, and each implementation can also differ in which level of functionality it is suited to handle. An example of this can be the CPU implementation node which in its simplest form would only be able to interrupt the processor while in its more complex implementations perform efficient bi-directional data transfer between the CPU and the BTC. The initial configuration is chosen such that the implementation nodes provide the minimum required functionality given the test task that it belongs to. It may not be the optimum choice however.

The hardware realization of the test operation is captured in the right most implementation nodes. While the (in this example) left most implementation nodes will handle the protocol conversion between the interface nodes and the other implementation nodes, the latter are used for implementing the actual test. As mentioned above a complex implementation node may require a more advanced implementation node that controls the interface node to support the demands that are put on the protocol. In other situations different protocol handling implementation nodes may be used with the same test realization implementation node, but different performance may be experienced since some protocols are more efficient than others.

6.4 Lego Piece Alternatives

To allow for optimization in the example, a set of interface and implementation nodes are proposed. There are two major alternations possible when designing interface nodes; it can be chosen which protocol to interface with and what level of features the chosen interface will support. The protocol can not be changed during the optimization (although its implementation can) since the board is given with external protocols such as, in this case, the M68360 QUICC CPU. The second issue, the feature support, can be optimized since choices in the implementation may put different requirements on the interface, e.g. if the check sum of a ROM is to be calculated this can be done either by the CPU (which does not require DMA capability of the CPU interface) or by the test controller through a DMA capable CPU interface. The interface nodes follow below. Note that we do not propose many alternatives regarding the protocol issue, thus the choice of interface node in this library is itself targeted to the example board.

Boundary Scan Interface

- BScan IF_1 supports only ASIC BIST
- BScan IF_2 supports ASIC BIST and Interconnect Tests

CPU Interface

- CPU IF_1 QUICC single interrupt vector
- CPU IF_2 QUICC multiple interrupt vectors
- CPU IF_3 QUICC DMA data transfer (multiple vector)
- CPU IF_4 QUICC CPU data transfer (multiple vector)

Test Bus Interface

- TBUS IF_1 IEEE 1149.5 interface
- TBUS IF_2 IEEE 1394 interface

Reset Interface

- Reset IF_1 Basic reset interface

The implementation nodes differ from the interface nodes as they do not support a specific protocol since they talk to the protocol indirectly through the interface nodes. Therefore the optimization choose the cheapest implementation alternative that provides the desired functionality. Another important optimization aspect is the possibility of sharing implementation nodes between different tests. This is more natural for the implementation nodes than for the interface nodes since a protocol always has to be available (at least the portion of it that can support an idle situation). The following alternatives exist:

Boundary Scan Interconnect

- BScan Intercon$_1$ Basic BScan interconnect test without intelligence
- BScan Intercon$_2$ BScan with knowledge of in, out, 3-state and bidir

ASIC BIST

- ASIC BIST$_1$ BScan based BIST supporting one ASIC
- ASIC BIST$_2$ BScan based BIST supporting multiple ASICs

Processor BIST

- P-BIST$_1$ Generic P-BIST
- P-BIST$_2$ Generic P-BIST without parameters
- P-BIST Parameters$_1$ Parameters for P-BIST$_2$

ROM checksum

- ROM checksum$_1$ CPU based checksum calculation
- ROM checksum$_2$ DMA based checksum calculation

RAM March

- RAM March$_1$ CPU based March test
- RAM March$_2$ DMA based March test

Core

- Core$_1$ Accepts commands from the test bus interface
- Core$_2$ Accepts commands with arguments from the bus interface

Identification

- Identify$_1$ Identifies board with a code
- Identify$_2$ Advanced identification that supports simple board queries

Error Reporting

- Error Report$_1$ Reports errors

Reset

- Reset$_1$ Basic reset

6.5 Test Flow Optimization

Optimization is performed from the first version test path in figure 6.2 to the final test path in figure 6.4. The optimization will be carried out with a tabu search algorithm, but a tabu search optimization will not be described here; rather a description of what changes are made will be done. The first optimization that can be made is the change of the CPU IF$_3$ implementation to a CPU IF$_2$ implementation. This means that a CPU interface capable of performing data transfers to and from the test controller (or in other ways access the CPU busses since it has a DMA interface) is changed for a simpler one that can only request the CPU to perform a test or receive an ok or fail signal all of which can be done through interrupt vectors and thus not requiring data transfers. This change is possible since all of the test operations performed through the CPU interface have fully CPU controlled implementations and thus it is only necessary for the CPU test program to signal passed or failed to the test controller without additional data. By doing this optimization we gain area since the CPU interface lacking DMA capabilities will be a smaller design.

The second optimization that can be performed is the unification of the two P-BIST tests. Both are performed on the same CPU interface and it would be feasible to share as much functionality between them as possible. This can be achieved by first splitting each of them into P-BIST$_2$ and P-BIST Parameters$_1$ nodes. The P-BIST$_2$ implementation is a subset of P-BIST$_1$ lacking all parameters that are associated with P-BIST$_1$. To hold the parameters the parameters node P-BIST Parameters$_1$ is used in conjunction with it. Since we now have two P-BIST$_2$ nodes without any parameters specific to their respective P-BIST tests we can merge them and come up with one P-BIST$_2$ and two P-BIST Parameters$_1$ nodes. By doing this we will reduce area (even if we now have one more node than initially).

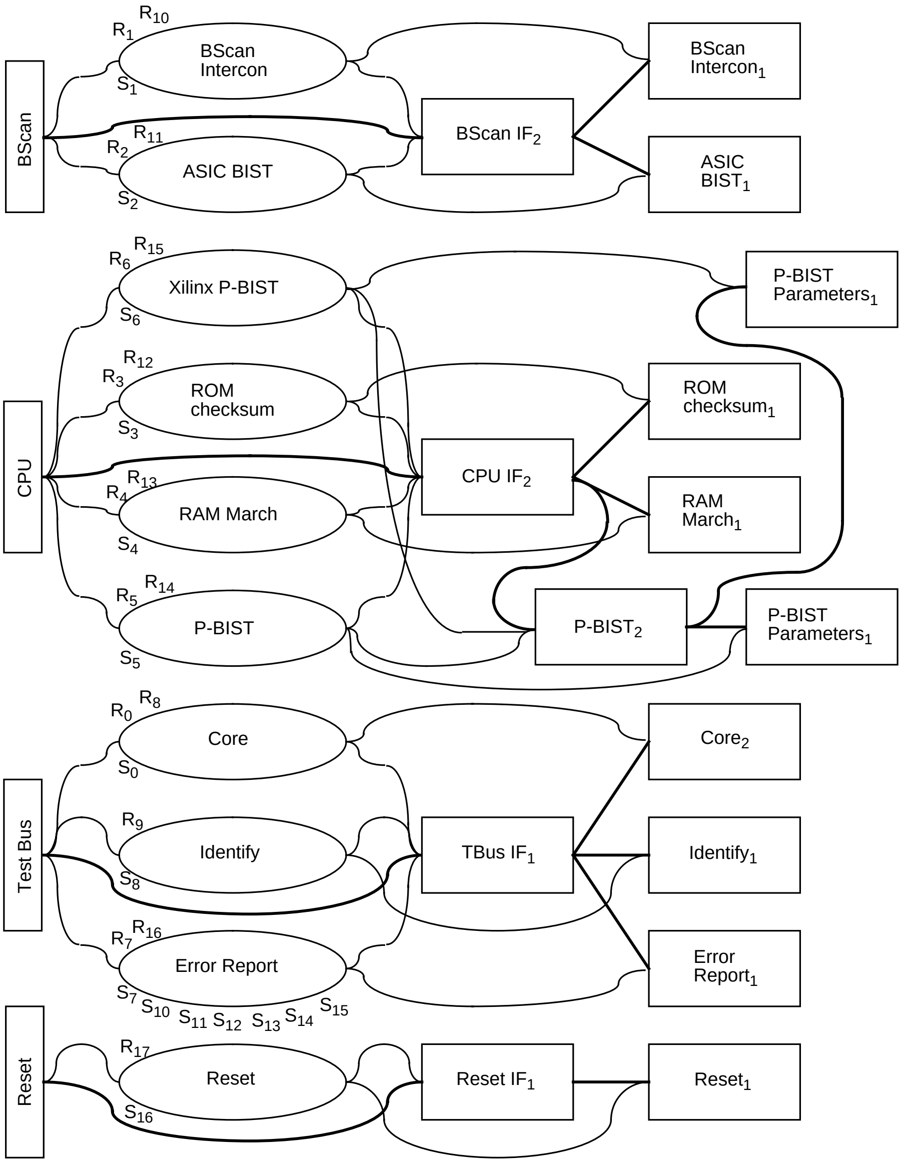


Figure 6.4: Final, optimized test path.

7 Conclusions

7.1 Summary

Hierarchical test architecture is important for hardware test of complex systems. Several research groups and industrial companies have been developing both hierarchical test concepts and board-level test controllers. All approaches this far have either developed synthesis strategies for the whole system where everything is prescribed from BIST functionality within the circuits to the over all system [Hab 95] or proposed more or less fixed test controllers. We believe that our approach in synthesizing the test controller to fit into a more freely designed hierarchical test framework fills a big gap between these two approaches. Further our approach will generate efficient test controllers that can be fitted into mature products as they evolve, and it does not put any constraints on what components must be used in the systems. With a full synthesis technique for the whole system high volume, cost effective components may be inappropriate since they do not fit into the full synthesis strategy. Therefore we think that our work is very important and has realistic uses in many cases where the more visionary ideas do not.

Our main contributions through the development of our board-level test controller strategy are:

- We have, in close cooperation with the industry, developed a hierarchical test architecture that supports self-test at the system, subsystem and board levels through the use of test controllers. The test architecture supports both production test and operation and maintenance test.
- A lego piece approach has been developed in which a test controller can be constructed from a set of pre-designed (although parameterized) code blocks, called lego pieces.
- We have developed a design representation, called the test design graph, that supports the lego piece concept. The TDG representation is suitable for capturing a complete test controller, and is

designed with support for reasoning and optimization. The TDG is built on the same ideas as ETPN, but is specially targeted for use in test controller design.
- Transformations have been defined to allow for a future inclusion of a transformational optimization heuristic. This heuristic has yet to be developed, but our plans are to base it on tabu search.
- A simplified board example from the industry has been used to demonstrate that the synthesis flow from a test requirement to a final BTC design can be achieved with the proposed technique.

7.2 Future Work

The ideas and techniques developed in this thesis aim at an implementation of a tool that will semi-automatically construct board level test controllers. The implementation of such a tool should be one of the highest priority tasks in our continuing work. For such a semi-automatic tool to work conveniently a well designed user interface will have to be designed. This is important both since it must be taken into consideration how the designer will influence the design process but also because a well designed and easy to work with graphical user interface will enhance the usability of such a tool.

The first step to take in the implementation of the tool is to make the test controller generation part (see figure 4.1). This part of the tool is the one that is most well described in this thesis and should therefore be the first one to be implemented. Through implementing that part a larger amount of lego pieces and parameters for them will have to be developed. Further more parameters associated with the different test tasks will be discovered, and relevance of the already proposed parameters will be verified.

Further work has to be done on the optimization issue which is only conceptually described in the thesis but where additional work and experimentation have to be made to ensure efficient cost/performance ratios of the generated controllers. Tabu search is planned as the most probable choice of optimization heuristic, as it has shown good results previously in the electronic design automation area. It also fits well to our transformation based approach of optimization with its neighborhood search concept.

Appendix A The LFSRx Lego Piece VHDL Code

This appendix shows the implementation of the “LFSRx” lego piece, which is a linear feedback shift register. Information about a lego piece is captured in several independent files. The first one is the file “_mapLFSRx.miv” which contains the VHDL port map of the lego piece.

```
__LegoPiece__ : __CompName__ -- PRBSx
  PORT MAP (Clock       => Clock,
            RequestData => RequestDataX,
            Data        => DataX);
```

The second file is “_compLFSRx.miv” which contains the VHDL component declaration of the lego piece.

```
COMPONENT __CompName__ -- LFSRx
  PORT (  Clock        : IN  STD_LOGIC;
         DataIn      : IN  STD_LOGIC ; -- Data Scanned In (port
1)
          CaptureData  : IN  STD_LOGIC ; -- Request Data
         Data7: OUT STD_LOGIC ; -- Data Bits in Output (port 2)
          Data6        : OUT STD_LOGIC ;
          Data5        : OUT STD_LOGIC ;
          Data4        : OUT STD_LOGIC ;
          Data3        : OUT STD_LOGIC ;
          Data2        : OUT STD_LOGIC ;
          Data1        : OUT STD_LOGIC ;
          Data0        : OUT STD_LOGIC);
END COMPONENT;
```

The third file is “_archLFSRx.miv” which contains the actual architecture of the lego piece, this is also written in VHDL.

```
USE work.std_logic_1164.ALL;
ENTITY __CompName__ IS -- LFSRx
  PORT (  Clock        : IN  STD_LOGIC;
         DataIn      : IN  STD_LOGIC ; -- Data Scanned In (port
1)
          CaptureData  : IN  STD_LOGIC ; -- Request Data
          Data7       : OUT STD_LOGIC ; -- Data Bits in Output
(port 2)
          Data6        : OUT STD_LOGIC ;
          Data5        : OUT STD_LOGIC ;
          Data4        : OUT STD_LOGIC ;
          Data3        : OUT STD_LOGIC ;
          Data2        : OUT STD_LOGIC ;
          Data1        : OUT STD_LOGIC ;
          Data0        : OUT STD_LOGIC);
END __CompName__;

USE work.std_logic_1164.ALL;
ARCHITECTURE Behave OF __CompName__ IS
BEGIN
  p1: PROCESS
  VARIABLE poly : STD_LOGIC_VECTOR(8 DOWNTO 1) := "00000000";
  VARIABLE tmp : STD_LOGIC;
  BEGIN
    WAIT ON CaptureData;

    IF CaptureData = '1' THEN
      tmp := DataIn XOR poly(1);
      poly(7 DOWNTO 1) := poly(8 DOWNTO 2);
      poly(8) := tmp;
    END IF;

    Data7 <= poly(8);
    Data6 <= poly(7);
    Data5 <= poly(6);
    Data4 <= poly(5);
    Data3 <= poly(4);
    Data2 <= poly(3);
    Data1 <= poly(2);
    Data0 <= poly(1);

  END PROCESS;
END Behave;
```

Along with the VHDL definitions used for each lego piece our tool uses two additional files in which all lego pieces are listed with some information about them. The first file is “_archlist” which lists all lego pieces along with their identity number and the files in which the port map, component and architecture files can be found.

```
...
7 LFSRx _compLFSRx.miv _mapLFSRx.miv _archLFSRx.miv
8 Const25 _compConst25.miv _mapConst25.miv _archConst25.miv
9 Compx _compCompx.miv _mapCompx.miv _archCompx.miv
10 BSIFx _compBSIFx.miv _mapBSIFx.miv _archBSIFx.miv
...
```

The final file contains information about which ports on the lego pieces that exist. It also indicates if the port is for input or output.

```
...
7 LFSRx 2 2 In DataInX In CaptureDataX 8 Out Data7X Out Data6X
Out Data5X Out Data4X Out Data3X Out Data2X Out Data1X Out
Data0X
8 Const25 1 8 Out Data7X Out Data6X Out Data5X Out Data4X Out
Data3X Out Data2X Out Data1X Out Data0X
9 Compx 3 8 In DataA7X In DataA6X In DataA5X In DataA4X In
DataA3X In DataA2X In DataA1X In DataA0X 8 In DataB7X In DataB6X
In DataB5X In DataB4X In DataB3X In DataB2X In DataB1X In
DataB0X 1 Out EqualX
10 BSIFx 7 8 In DataA7X In DataA6X In DataA5X In DataA4X In
DataA3X In DataA2X In DataA1X In DataA0X 8 In DataB7X In DataB6X
In DataB5X In DataB4X In DataB3X In DataB2X In DataB1X In
DataB0X 2 In DataInX Out RequestDataX 2 Out DataOutX Out Cap-
tureDataX 1 In StartX 1 Out FinishX 4 Out TDOX In TDIX Out TMSX
Out TCKX
...
```

References

[Abr 90] M. Abramovice, M. Breuer, A. Friedman,” Digital Systems Testing and Testable Design,” *IEEE Press*, ISBN 0-7803-1062-4, 1990.

[Ang 97] C. Angelbro, “P-BIST Test Method Conquer the Ericsson World,” *IEEE European Test Workshop*, Cagliari, May 28-30, 1997.

[Bre 88] M. A. Breuer, J. C. Lien, “A Test and Maintenance Controller for a Module Containing Testable Chips,” *International Test Conference*, pp. 502-513, 1988.

[Fje 92] B. Fjellborg, “Pipeline extraction for VLSI data path synthesis,” *Ph. D. dissertation No. 273*, Dept. of Computer and Information Science, Linköping University, 1992.

[Glo 93] F. Glover, E. Taillard and D. de Werra, “A user’s guide to tabu search,” *Annals of Operations Research*, vol. 41, pp. 3-28, 1993.

[Goo 91] A.J. van de Goor, “Testing Semiconductor Memories: Theory and Practice,” *John Wiley & Sons*, 1991.

[Hab 94] O. F. Haberl, T. Kropf, “Selft Testable Boards with Standard IEEE 1149.5 Module Test and Maintenance (MTM) Bus Interface,” *The European Design and Test Conference*, pp. 220-225, 1994.

[Hab 95] O. F. Haberl, T. Kropf, “HIST: A Hierarchical Self Test Methodology for Chips, Boards, and Systems,” *Journal of Electronic Testing: Theory and Applications*, pp. 85-116, vol. 6, no. 1, 1995.

[Has 92] A. S. M. Hassan, V. K. Agarwal, B. Nadeau-Dostie, J. Rajski, “BIST of PCB Interconnects Using Boundary-Scan Architecture,” *IEEE Transactions On Computer-Aided Design*, pp. 1278-1288, vol. 11, no. 10, 1992.

[Håk 95] J. Håkegård, “Board Level Boundary Scan Testing and Test Controllers,” *CADLAB Memo 95-01*, Department of Computer and Information Science, Linköping University, 1995.

[Håk 96] J. Håkegård, G. Carlsson, Z. Peng, “A Board-Level Test Controller to Support a Hierarchical DFT Architecture,” *IEEE European Test Workshop*, June 1996.

[Håk 96b] J. Håkegård, Z. Peng, “A Board-Level Test Controller to Support a Hierarchical DFT Architecture,” *The Third International Test Synthesis Workshop*, Santa Barbara, USA, 1996.

[Håk 97] J. Håkegård, Z. Peng, “Design and Synthesis of a Generic Board-Level Test Controller,” *Short Contribution of the 23rd Euromicro Conference*, Budapest, Hungrary, September 1-4, 1997.

[IEEE 90] IEEE Std. 1149.1-1990, “IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture,” *IEEE Computer Society*, 1990.

[IEEE 95] IEEE Std. 1149.5, “Standard for Module Test and Maintenance Bus (MTM-Bus) Protocol,” *IEEE Computer Society*, 1995.

[ISO 10164] ISO, “*Information Technology - Open Systems Interconnection - System Management: Confidence and Diagnostic Test Classes*,” ISO/IEC DIS 10164-14 draft.

[Jag 95] W. Jager, H. Hulvershorn, S. Adham, “*Self-Testing Hardware to Improve Fault Detection and Isolation in Switching Systems*,” International Switching Symposium, April 1995.

[Jar 91] N. Jarwala, C. W. Yau, “Achieving Board-Level BIST Using the Boundary-Scan Master,” *International Test Conference*, pp. 649-658, 1991.

[Jar 91b] N. Jarwala, C. W. Yau, “The Boundary-Scan Master: Architecture and Implementation,” *2nd European Test Conference*, pp. 1-10, April 1991.

[Jar 92] N. Jarwala, C.W. Yau, P. Stiling, E. Tammaru, “A Framework for Boundary-Scan Based System Test and Diagnosis,” *International Test Conference*, pp. 993-998, 1992.

[Lie 89] J. C. Lien, M. A. Breuer, “A Universal Test and Maintenance Controller for Modules and Boards,” *International Test Conference*, pp. 231-240, May 1988.

[Mat 92] J. S. Matos, F. S. Pinto, J. M. M. Ferreira, “A Boundary Scan Test Controller for Hierarchical BIST,” *International Test Conference*, pp. 217-223, 1992.

[Mau 93] C. Maunder, “A Universal Framework for Managed Built-In Test,” *International Test Conference*, pp. 21-29, 1993.

[Pen 87] Z. Peng, “A Formal Methodology for Automated Synthesis of VLSI Systems,” *Ph. D. Dissertation No. 170*, Linköping University, Dept. of Computer and Information Science, 1987.

[Pen 94] Z. Peng, K. Kuchcinski, “Automated Transformation of Algorithms into Register-Transfer Level Implementations,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 2, February 1994.

[Per 97] M. Perbost, L. Le Lan, C. Landrault, “Testability Analysis of digital boards and MCMs,” *IEEE European Test Workshop*, Cagliari, May 28-30, 1997.

[Pet 81] J. L. Peterson, “Petri Net Theory and the Modeling of Systems,” *Prentice-Hall, Englewood Cliffs, N.J.*, ISBN 0-13-662983-5, 1981.

[Rig 89] J.-L. Rigo, “Hierarchical Built-In Self Test,” *International Conference on Radar*, Paris, April, 1989.