Institute of Computer Architecture and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Master Thesis Nr. 3304

Modeling of Design-for-test infrastructure in complex Systems-on-chips

David Prasetyo Buntoro

Course of Study: INFOTECH

Examiner: Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich

Supervisor: M.Sc. Alejandro Cook

Dipl.-Inf. Laura Rodríguez Gómez

Dipl.-Inf. Dominik Ull

Commenced: 17.02.2012

Completed: 18.08.2012

CR-Classification: B5.1,B5.3,C5.3

Every integrated circuit contains a piece of design-for-test (DFT) infrastructure in order to guarantee the chip quality after manufacture. The DFT resources are employed only once in the fab and are usually not available during regular system operation.

In order to assess the hardware integrity of a chip over its complete life-cycle, it is promising to reuse the DFT infrastructure as part of system-level test.

In this thesis, the provided system, a Tricore processor from Infineon, must be partitioned and modified in order to enable the autonomous structural test of every component of the system in the field without expensive external tester.

Contents

List of Figures V

1 Introduction 1

2 Literature Review 4
- 2.1 Electronic System in Automotive . . . 4
  - 2.1.1 CAN Bus . . . 4
- 2.2 SOC Test . . . 6
- 2.3 Design for Test . . . 7
  - 2.3.1 Core-Based Test . . . 7
  - 2.3.2 Scan Design . . . 10
- 2.4 Related Work . . . 15

3 Proposed Method 17

4 Implementation 19
- 4.1 System . . . 19
- 4.2 TriCore Programming . . . 19
  - 4.2.1 Memory Mapped I/O . . . 20
  - 4.2.2 Inserting Constant into SRE File . . . 21
- 4.3 Peripheral Connection . . . 21
  - 4.3.1 FPI Bus . . . 21
  - 4.3.2 FPI Bus Multiplexer . . . 22
- 4.4 Test Data Transfer . . . 22
  - 4.4.1 DFT Infrastructure Bridge . . . 22
  - 4.4.2 CAN Controller . . . 25
- 4.5 DFT Infrastructure for Peripherals . . . 26
  - 4.5.1 Tools . . . 27
  - 4.5.2 EDT Logic Generation . . . 27

5 Test Application 30
- 5.1 Internal Test . . . 31
- 5.2 External Test . . . 32

5.3 Experiment Analysis 33
5.4 Future Work 33

6 Conlusion **35**

Literatur **A**

List of Figures

1.1 "Bathtub" curve represents the relationship between failure rate and time . . . 2
1.2 Traditional and proposed approach . . . 3

2.1 Numerous ECUs in a car . . . 4
2.2 CAN data frame . . . 5
2.3 CAN remote frame . . . 6
2.4 Boundary scan architecture [WST08] . . . 8
2.5 Typical boundary scan cell [WWW06] . . . 9
2.6 IEEE 1500 system architecture[WWW06] . . . 11
2.7 Scan design architecture . . . 12
2.8 Muxed-D scan cell . . . 12
2.9 LSSD scan cell . . . 13
2.10 System of the logic built-in self-test . . . 13
2.11 EDT architecture . . . 14
2.12 EDT decompressor . . . 14
2.13 SOC test platform using an embedded processor . . . 16
2.14 Architecture diagram from [JT99] . . . 16

3.1 Generic test architecture for the proposed approach . . . 18

4.1 Original TriCore SOC from Infineon . . . 20
4.2 Timing diagram for single data transfer of FPI bus read and write . 22
4.3 FPI bus multiplexer configuration . . . 23
4.4 Connect between the DFT infrastructure bridge and the DUT . . . 24
4.5 Two concurrent Finite State Machines . . . 25
4.6 Timing diagram of wishbone bus . . . 26
4.7 Timing diagram for translating FPI and wishbone bus . . . 27
4.8 TestKompress flow to generate the DFT infrastructure . . . 28
4.9 Counter toplevel transformation: (a) Original design. (b) Counter after scan insertion. (c) Counter after EDT logic insertion. . . . 29

5.1 Complete test environment . . . 30
5.2 Internal test scenario . . . 31

5.3 External test scenario 33

List of Tables

2.1 CAN data frame for base frame format 6

3.1 Relationship between test scenarios and the test requirement 17

4.1 Available address space in the TriCore simulation platform 20
4.2 An example on how to insert a constant value into a SRE file 21
4.3 Registers in the DFT infrastructure bridge 24

5.1 CAN message overhead and DUT test data 31
5.2 Internal test simulation result 32
5.3 External test simulation result 32

1 Introduction

Nowadays, automotive systems offer numerous features, mostly realized in software, which are executed on multiple interacting electronic control unit (ECU). To achieve a high level of sophistication, the automotive industry has taken advantage of the processing capabilities made available by continuous advances in semiconductor technology. According to [Cha09], current cars dedicate 35-40 percent of total cost to its ECUs, which control areas like powertrain, chassis, interior or infotainment.

Considering the vital role of the ECUs, one defect might disrupt the whole system operation. Therefore, the ISO 26262 standard has been proposed as a guidance for the development of all system components, covering the concept phase, system design, hardware & software development, validation, and even later device production[JCJ+11] . Despite following the safety standard during development and production phases, ECUs are still prone to hardware failures in the field.

According to the bathtub curve shown in Fig.1.1, the failure rate is characterized by early lifetime, constant, and wear-out phases. Early lifetime failures occur due to latent defects [Mak07], which manifest themselves after manufacturing test due to electrical and thermal stress. Random problems such as electrostatic discharge (ESD), account for the microelectronic constant failure phase, whereas the advancement of microelectronics to deep sub-micron technology and beyond introduces a wear-out failure factor due to electromigration, stress migration, time-dependent dielectric breakdown, and thermal cycling[SABR04]. Similarly, aging degrades the transistor performance because of hot-carrier degradation effects [Bor05].

Such failures can occur within one of the integrated circuits (IC) in the ECU, which can happen at any point in the lifetime of the system. In order to assure the system safety, hardware failures must be promptly identified. The chip must be tested continuously while operating in the field, so that it is possible to detect a defect before some critical system fails.

One way to assess the IC's condition is to test the functional and performance specifications[BA00]. This method, which is called functional test, applies a large set of input combinations to the IC and checks its output. While this is possible for designs with a few number of inputs, thoroughly test an IC with hundreds of inputs is not feasible and leads to excessive test time.

Another method to test an IC is called structural test. Instead of analyzing the IC functionality, this method checks the hardware integrity of the SOC. Structural test is more exhaustive than functional test because of the ability to control and observe internal signals of an SOC. The test data is generated based on a specific fault

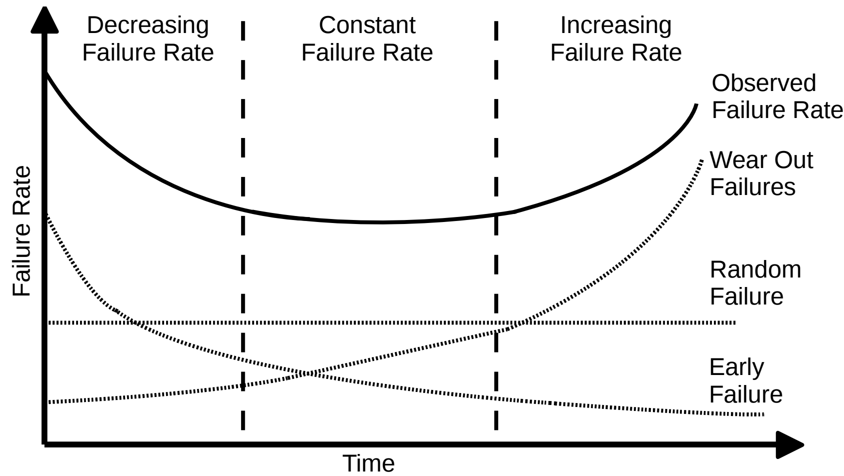

Figure 1.1: "Bathtub" curve represents the relationship between failure rate and time

model by a tool called automatic test pattern generator (ATPG). The fault model represents the behavior of the defect, with stuck-at faults as the most successful fault models [WWW06]. Stuck-at fault model assumes a defect exists and makes the signal lines to be stuck at logical '1' or '0'.

Traditionally, IC is once tested structurally in the fab. After it is assembled into an electronic system like an ECU, there is no direct access to the resources required to perform structural test. Therefore, failure analysis has to proceed using functional tests, which may not achieve sufficient fault coverage.

To allow the verification of an IC integrity in the field, a suitable test architecture for an SOC has been developed in this thesis. The architecture is able to structurally test the core inside the SOC by reusing the available test infrastructure. While existing test architectures require the presence of an external controller, the developed test solution can either be performed independently or activated by an external, low-cost device. For this purpose, the existing functional resources in the system are employed to access a device under test (DUT) and to transfer the corresponding test information. Developing the test architecture involves insertion of a test interface, test infrastructure, test vector generation, as well as test response analysis. Fig.1.2 illustrates the difference between the traditional approach and the proposed method in this thesis.

This thesis report is organized as follows. Chapter two presents the theoretical background of the test architecture, together with other existing in-field test architectures. Chapter three explains the general overview of the proposed solution. Chapter four discusses the detailed implementation of the system. Chapter five explains the test application process and the test result. Conclusion is presented in

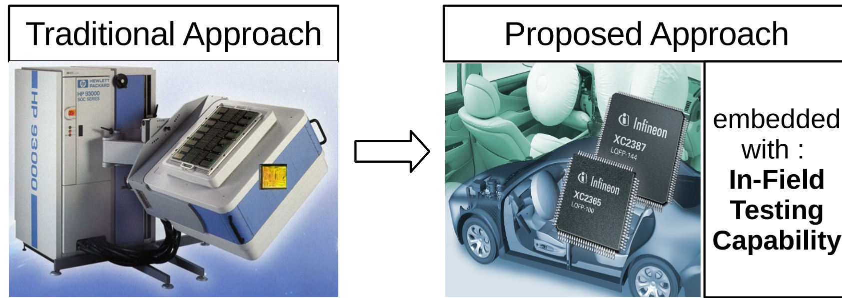

Figure 1.2: Traditional and proposed approach

the last chapter.

2 Literature Review

2.1 Electronic System in Automotive

Modern automotive electronic systems are composed of multiple independent computer systems. These computer systems, which are called electronic control unit (ECU), have specialized tasks. They monitor a specific component in the system such as engine and react according to an external event. Figure 2.1 shows some of the available ECUs inside a modern car.

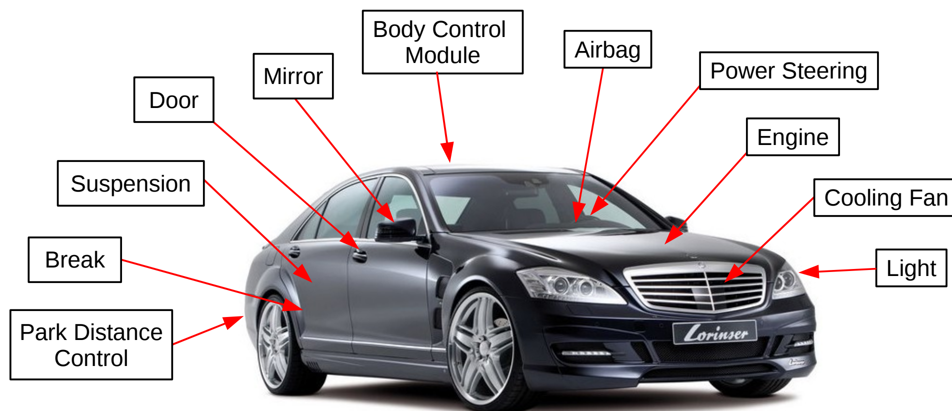

Figure 2.1: Numerous ECUs in a car

This ECUs is connected through a communication channel to form a distributed computing system. The common bus protocols in automotive domain are controller area network (CAN), local interconnect network (LIN), media oriented system transport (MOST), and FlexRay. CAN is a complex bus standard and the most popular communication network in the automotive domain. LIN is a bus standard that has less complexity and cheaper than CAN bus. FlexRay is designed to be more reliable than CAN bus, therefore it is more expensive. MOST is a multimedia bus standard that is optimized for automotive application.

2.1.1 CAN Bus

CAN is a very popular bus standard in the automotive domain. Taken directly from the specification, CAN is defined as a serial communications protocol which

efficiently supports distributed real time control with a very high level of security [Gmb91]. Originally developed in 1983 at Robert Bosch GmbH, CAN is used to create a communication link between multiple ECUs in an automobile.

The data transmission in the CAN bus supports sequential data transfer, i.e. there is only one message transmitted or received at the same time. To determine which message owns the bus, the CAN protocol checks the dominant (logic low) and recessive bits (logic high) in the messages. When two nodes in the CAN network transmit messages at the same time, message with more dominant bits in the beginning wins the bus arbitration. This means, message priority is reflected by first field of CAN frame, i.e. the device id inside the arbitration field. The lower the device ID, the higher its priority.

CAN message, which is also known as CAN frame, has two different standards called base frame format (CAN 2.0 A) and extended frame format (CAN 2.0 B). The only difference between those standards is the size of message ID, with 11 bit and 29 bit to the former and latter standard, respectively. Data transfers between nodes are facilitated by data frame and remote frame.

2.1.1.1 CAN Data Frame

CAN data frame functionality is implied from its name, i.e. to transfer data between CAN nodes. It consists of seven different fields, as shown in Figure 2.2. The start of frame field marks the beginning of the CAN transmission using one dominant bit. Depending on the standard, the arbitration field contains 11/29 bit device ID and 1 dominant bit of remote transmission request (RTR). The control field consists of two reserved bits and four data length bits. The unit for the data length is byte, and the maximum data length size is 8 bytes. The CRC field contains the cyclic redundancy check (CRC) and one bit CRC delimiter. The transmitter sets the acknowledge (ACK) field with two recessive bits so that when the target receiver gets the message correctly, it can acknowledge to the transmitter by sending two domain bits at the ACK field. The end of frame field contains a sequence of 7 recessive bits, defined as the end of the message. This explanation is summarized at Table 2.1.

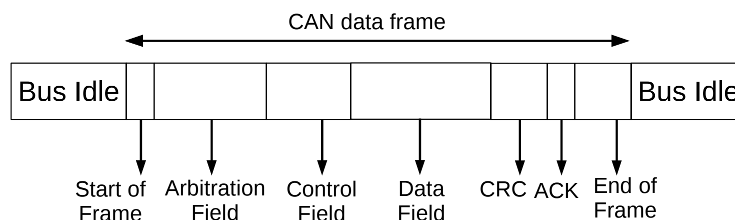

Figure 2.2: CAN data frame

Field name	Length (bit)	Description
Start of frame	1	Start of transmission
Arbitration	12	Contains 11 bit Device ID and 1 bit Remote transmission request(RTR)
Control field	6	2 reserved bit and 4 bit data length code
Data field	0-64	Data to be transmitted
CRC field	16	15 bit CRC sequence and 1 bit CRC delimiter
ACK field	2	Contains ACK slot and ACK delimiter
End-of-frame (EOF)	7	Contains 7 recessive bits

Table 2.1: CAN data frame for base frame format

2.1.1.2 CAN Remote Frame

A remote frame is transmitted by a node who wants to request some information from another node. The structure of remote frame is the same as the data frame, except that the RTR bit is recessive and there is no data field. Figure 2.3 shows the CAN remote frame.

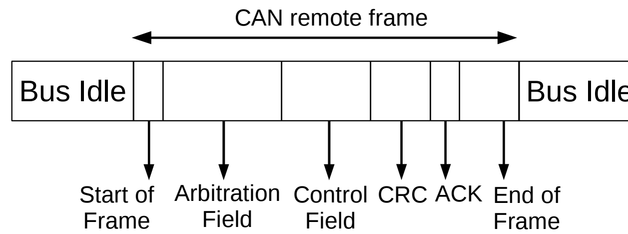

Figure 2.3: CAN remote frame

2.2 SOC Test

Failures can happen anytime during an ECU's lifetime. Hence, it must be continuously tested so that a safety mechanism can be launched. For the safety mechanism, some systems initiate a system shut down when an error is detected, while other systems reconfigure itself so that it can continue working by degrading the system performance. To avoid system error because of hardware failures, test procedures include not only the ECU system as a whole, but also individual components that compose the system, such as SOC. Depending on the time when structural tests take place, test is categorized into two groups, online and offline test.

Online test is used to detect faults during normal system operation [AAMH98].

One simple method to justify the system functionality is to use a watchdog timer, i.e., a peripheral that counts up and gets cleared by the system periodically. If the software, for any kind of reasons, freezes, the watchdog timer will reach to a certain value. When this condition occurs, it means that the system is faulty. Besides in system level, online test is also exist in the core level. [PGSR10] proposes software based self-test (SBST) for a microprocessor for online test, whereas [VGPH05] supports concurrent built-in self-test (BIST).

Offline test at the core level is performed when the device is inactive. One example of offline test is a BIST, a design for testability (DFT) technique that provides test capability within the component itself [Wun98]. Normally, offline test is performed once in the factory for manufacturing test. As there is a need to perform this test in the field [AS11], it is possible to run the offline test periodically or sporadically. Periodic testing is a prescheduled test activity[PG05], while sporadic testing happens only at specific time, such as during device start up or shut down.

2.3 Design for Test

This thesis focuses on nonconcurrent online test to detect a hardware fault using structural test. Structural test consists of three steps, which are applying the test pattern to the core under test (CUT), getting the test response from the CUT, and analyzing the received test response. Such test is covered by a method called DFT. DFT is a design technique that adds a test infrastructure to the design, so that controllability and observability of internal signals are acquired.

2.3.1 Core-Based Test

2.3.1.1 Boundary Scan Standard

Boundary scan standard, which is also known as IEEE 1149.1 or JTAG, is a widespread and successful standard. Originally designed to assist board level test of digital logic circuits, the standard makes its way to other application like power management, clock control, verification and debugging [WWW06]. Using the JTAG standard, it is possible to control the internal register of the chip through the standardized test interface.

JTAG standard consists of four hardware components: a TAP controller (TAPC), a test access port (TAP), instruction registers, and other type of data registers such as bypass register, boundary scan register, or device ID register. TAPC, which is controlled using test clock (TCK), test mode select (TMS), and test reset (TRST) pin, controls the mode of operation through the finite state machine (FSM) logic. Together with test data in (TDI) and test data out (TDO), the five pins are the necessary signals for the JTAG standard. The FSM logic schedules the access to the JTAG's instruction and data register.

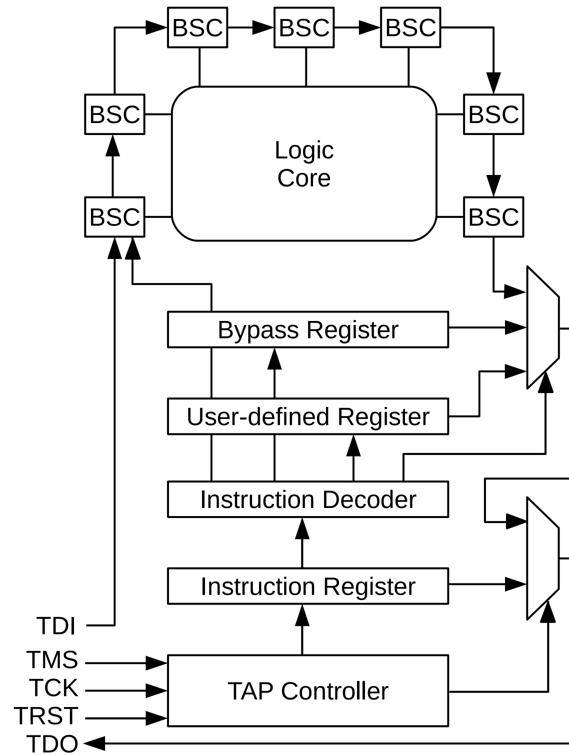

Figure 2.4: Boundary scan architecture [WST08]

TAP Controller TAPC is made from a state machine that has 16 states. This state machine generates reset signal, as well as capture, update, and shift signals for the instruction and data register. The signals are generated based on the 16 different states of the FSM, which are divided into three groups: idle state, control data register state, and control instruction register state. In essence, TAPC provides three functionalities:

- Resets the boundary scan architecture.
- Generates a control signal to load a value into instruction register.
- Generates signals to manipulate the content of the boundary register using operation such as capture, update, and shift.

Instruction register The instruction register determines the operation of the JTAG. The standard specifies four mandatory instructions, which are SAMPLE, PRELOAD, BYPASS, and EXTEST. SAMPLE instruction creates a snapshot of the current value at both the input and output pins. PRELOAD instruction allows the test

data to be shifted into/out of the data register. BYPASS instruction is used to channel the data directly from TDI to TDO. EXTEST instruction enables a tester to check the external hardware connection between chips and boards. Beside these four instructions, JTAG standard also accommodates some other user-defined instructions.

Boundary Scan Figure 2.5 shows a typical boundary scan cell. Mode, shift-DR and clock-DR pins are controlled by TAPC signals. When mode is set to 0, the scan cell enters normal mode and there is a direct connection between the input and the output. Test mode is entered when mode is set to 1, making the out pin to be connected with the output of flip-flop R2.

Capture operation saves the input pin value to the flip-flop R1 by setting the shift-DR pin to 0 and applies a clock pulse to clock-DR pin. Shift operation moves the data from pin scan-in to scan-out. This is performed by setting the shift-DR pin to 1 and applies the clock-DR pin with one clock pulse. For update operation, the data from flip-flop R1 is copied to flip-flop R2 by applying a clock to pin update-DR.

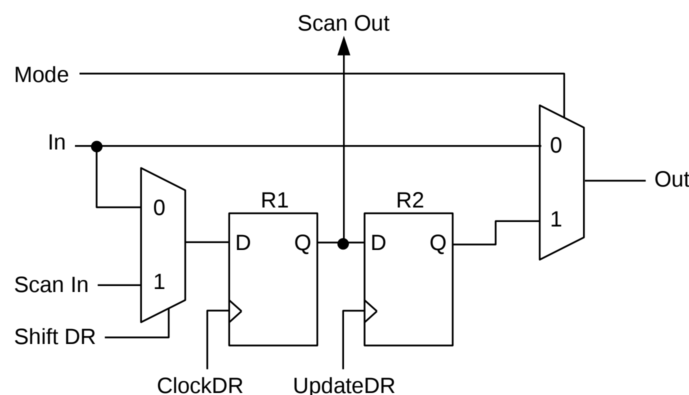

Figure 2.5: Typical boundary scan cell [WWW06]

2.3.1.2 IEEE 1500

IEEE 1149.1 standard was originally designed for testing the connection between ICs in a PCB. As trends go to many core era, testing cores in a SOC seems to be analogous to testing many chips on a PCB. However, the two test scenarios pose different characteristics and challenges. [WWW06] mentions some problems for core-based test that leads to the development of IEEE 1500:

- Mixing technologies – As SOC contains lots of different cores from multiple vendors, it is not possible for system integrators to generate all of the test

data by themselves. Core providers should contribute with the test generation process. Therefore, there is a need to standardize the test mechanism.

- Expensive external ATE – If all testing mechanisms for cores are conducted by an external tester, test time will be high. Considering that an ATE is very expensive, longer test time leads to higher test cost. By embedding some of the core test capabilities within the SOC, test time can be reduced.
- IP protection – A core with IP protection doesn't provide internal structure. Reusing the test data from the core vendor without/little modification is the preferable method. Hence, a standard test interface is needed.

IEEE 1500 standard wraps each core with a wrapper, a component that is used to standardize the test interface and to execute test commands. Figure 2.6 displays n number of cores with IEEE 1500 wrapper. In total, IEEE 1500 spefication defines five hardware components, which are:

- Wrapper Serial Port (WSP) – consists of wrapper serial input (WSI), wrapper serial output (WSO), and wrapper serial control (WSC). WSI and WSO have the same functionality as TDI and TDO in JTAG standard, which are to interface data input and data output. WSC produces six signals to control the WBR and 1500 standard functions.
- Wrapper Parallel Port (WPP) – an optional component for core parallel access. WPP consists of three terminals: wrapper parallel input (WPI), wrapper parallel output (WPO), and wrapper parallel control (WPC).
- Wrapper Instruction Register (WIR) – similar to the JTAG's instruction register, which is to store instruction to be executed.
- Wrapper Bypass Register (WBY) – is used to directly bypass test data from the input to the output pin.
- Wrapper Boundary Register (WBR) – has the same functionality as boundary scan cell in JTAG, with four additional modes These are normal mode, inward facing mode, outward facing mode, and nonhazardous (safe) mode.

2.3.2 Scan Design

The purpose of scan design is to gain observability and controllability over the sequential element, i.e. flip-flops (FF). Scan design is built by replacing normal FFs with scannable FFs (SFF), so that it is possible to serially connect all FFs. The configuration, which is called scan chain, is automatically generated by an electronic design automation (EDA) tools.

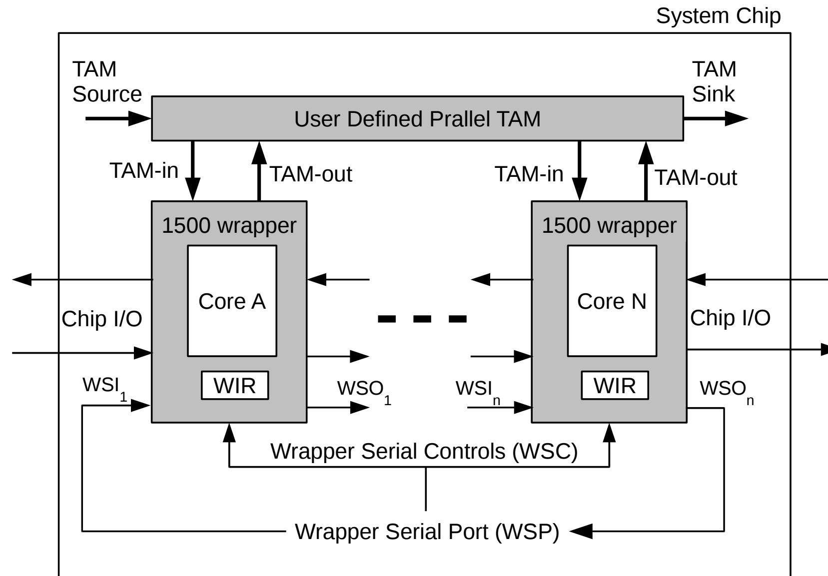

Figure 2.6: IEEE 1500 system architecture[WWW06]

SFF differs from normal FF, in that it has three additional pins: a scan-input (SI), a scan-output(SO), and a scan enable(SE). Normal mode and test mode are active when SE pin is deasserted and asserted, respectively. In normal mode, SFF behaves as normal FF. In test mode, FF content can be controlled by shifting in data through SI and observed from from reading the SO value. There are two popular scan cell design: muxed-D scan cell and level sensitive scan design (LSSD).

Mux-D scan cell As the most popular scan cell design [WWW06], a mux-D scan cell consists of a multiplexer and a D flip-flop. When SE is set to 0, the scan cell works at normal mode. Test mode is activated when SE is set to 1. New data is shifted in through SI, and the FF data is shifted out through SO. Data shift occurs during rising edge of the clock input.

LSSD scan cell Mux-D scan cell is suitable for an edge-triggered flip-flop design, while LSSD scan cell is used for a latch based design. LSSD scan cell, shown in Figure 2.9, consists of multiplexer, FF, and latch. Because of the latch, data capture happens when CLK input is 1. Data is shifted out during falling edge of the FF.

2.3.2.1 Logic Built-in Self-Test

Logic BIST is a DFT technique that provides a self test capability to the design. Testing digital logic requires two elements, a test pattern source and a test response

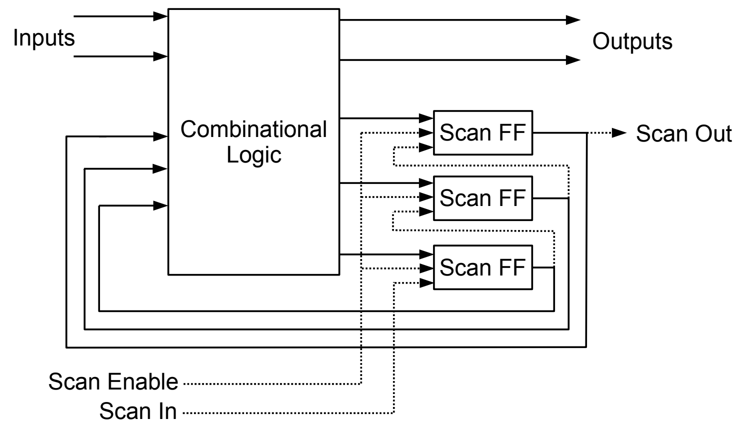

Figure 2.7: Scan design architecture

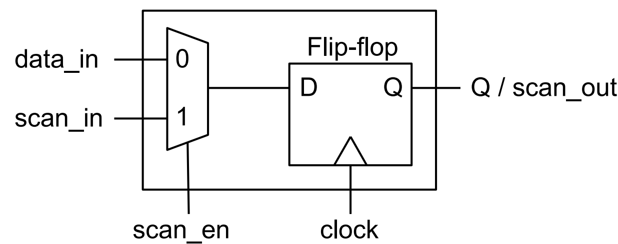

Figure 2.8: Muxed-D scan cell

analyzer. Besides, the test must be controlled by some kind of logic controller to create control signals, such as scan enable and scan clock, as well as to determine whether the core under test (CUT) passes or fails the test. The architecture of logic BIST is shown at Figure 2.10.

Test pattern generator (TPG) generates the necessary test data that is applied to the CUT. There are many kinds of test pattern generator sources. Deterministic testing stores the predetermined test pattern into a test storage, therefore it is very expensive in terms of resource area. Exhaustive testing is able to generate all input combinations using binary counter or complete linear feedback shift register (LFSR), but it takes too much time to complete a core test. Compared to deterministic testing, pseudo random testing reduces the resource requirement to generate the test pattern by sacrificing the test coverage. Pseudo random testing typically uses maximum length LFSR as the pattern generator.

Output response analyzer (ORA) collects and compacts the CUT response into a signature. Response compaction is necessary, as it is not feasible to store all circuit's fault free responses. By only comparing the signature, the system saves massive amount of data storage. Popular ORA methods are ones count testing,

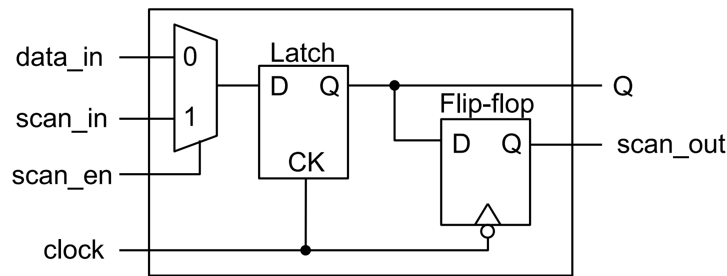

Figure 2.9: LSSD scan cell

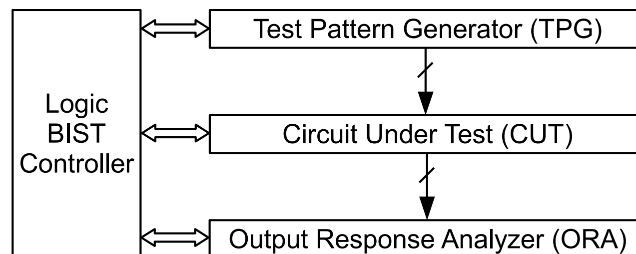

Figure 2.10: System of the logic built-in self-test

transition count testing, and signature analysis.

2.3.2.2 Embedded Deterministic Test

Embedded Deterministic Test (EDT) is a DFT method that aims to reduce the test data by using a compression technique [RTKM04]. The EDT method minimizes the need to modify the original hardware and follows the same test procedure that is based on scan and ATPG. As shown in Figure 2.11, the EDT architecture consists of scan chains, a compactor, and a decompressor. The test procedure is composed of EDT logic insertion, test vector generation using an ATPG, and test application using an Automatic Test Equipment (ATE). The test vector, which is compressed and predetermined, is stored in an Automatic Test Equipment (ATE).

Decompressor The decompressor receives the compressed test pattern from an ATE through scan channels. As the test pattern is shifted in, it is also decompressed and fed into a large number of scan chains. Shown at Figure 2.12, the decompression process is carried out by a ring generator and a phase shifter. Ring generator is another form of linear finite state machine (LFSM). The phase shifter enables the ring generator to control the various scan paths by mutually displacing the produced sequences.

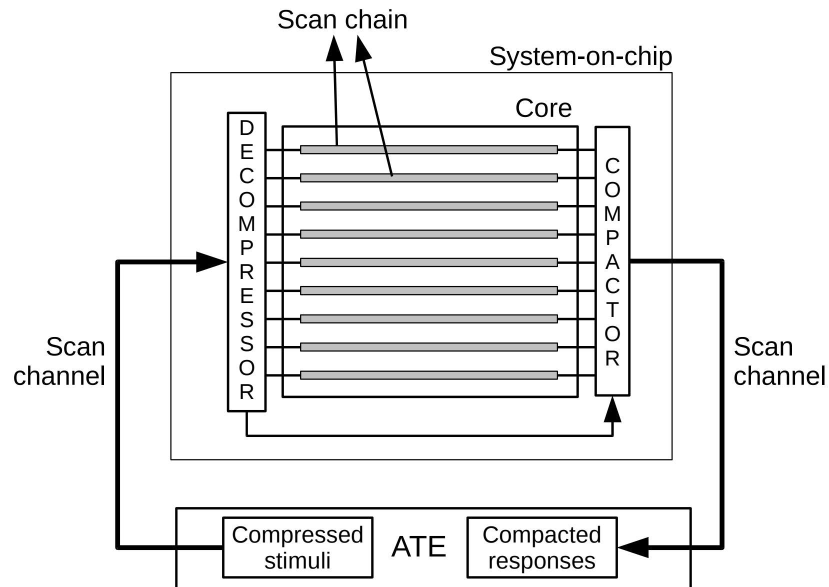

Figure 2.11: EDT architecture

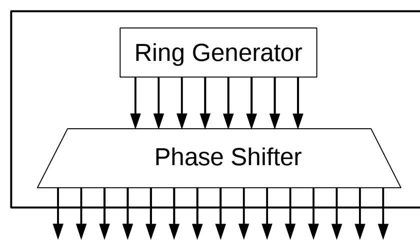

Figure 2.12: EDT decompressor

Compactor The compactor is located between the scan chains output and the scan channels. It compacts the test response from scan chains and transfers it to the scan channels output. The compactor is controlled by the decompressor and is made of spatial compactor(s) and gating logic.

2.4 Related Work

Currently, the method to test the ECUs in a vehicle is to test its functionality as a whole. This test is not sufficient to detect hardware defects that can occur inside a SOC. As a result, it is necessary to structurally test the SOC in the field. Several authors have proposed test architectures for this purpose.

[LCH05] proposes a test architecture using an embedded processor with a dedicated test access mechanism (TAM) controller. The test system, which is illustrated in Figure 2.13, supports numerous core wrappers such as IEEE 1149.1, IEEE 1500, and BIST. Beside a processor and a TAM controller, the system needs a test bus, a test access port (TAP) controller, an embedded memory, an external memory, and an external memory control. By using this configuration, the test system supports scan, analog-digital conversion (ADC), memory BIST, and hierarchical test control.

The test starts when test program is loaded and executed by the processor. First, the test program reads some of the test data from the external memory and then transfers it to the embedded memory. Next, the processor configures the TAM controller. After the configuration, the TAM controller autonomously fetches the test data from the embedded memory and delivers it to the designated core. Then, the processor verifies the test response. This procedure is repeated until all test data have already been applied. When all test responses are correct, the chip is determined to be defect free.

[JT99] introduces a test architecture that minimizes the test cost by reducing the test storage data and the test time. The idea behind this architecture is based on the fact that the test time is related to the test data. They argues that test time can be decreased if there are less test data to be applied. Therefore, the test data comes in the form of compressed deterministic test vector and stored in the external tester.

The test flow begins when an external tester transfer the test data to the SOC through communication channel. The test data is already modified in a way that it contains a processor instruction and test pattern. A processor inside the SOC receives and decodes the test data and applies the test pattern to the scan chain through the test data serializer. The test response from the target core goes through a multiple-input shift register (MISR) for compacting the test response. The compacted test response, which is called response signature, is sent back by the processor to the tester. Figure 2.14 shows the test architecture together with the CUT.

The presented test solutions prove that testing a core in the field without an ATE

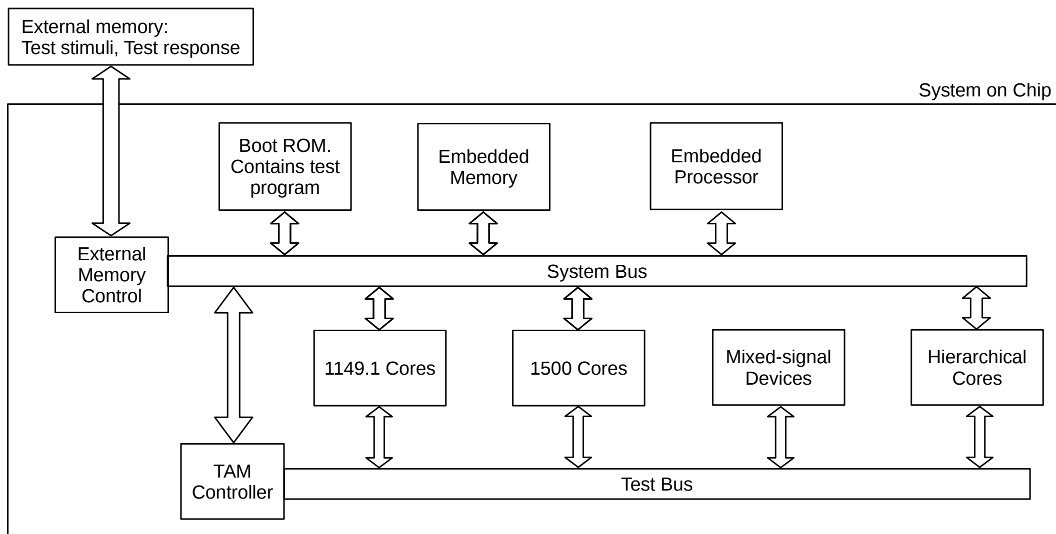

Figure 2.13: SOC test platform using an embedded processor

is feasible. Both approaches, unfortunately, require an external test component to store the test data. Therefore, the proposed test architecture should be able to run independently without an external component. Two ideas from each approaches, i.e. the test data compression and the TAM controller functionality, should be included into the proposed approach.

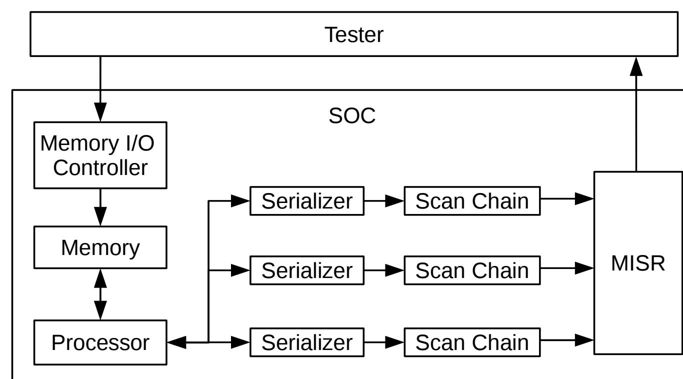

Figure 2.14: Architecture diagram from [JT99]

3 Proposed Method

Industry experts have discussed and stated that there are four possible scenarios to test the SOC in an ECU. The easiest test scenario is to take the whole ECU out of the vehicle and send it to the laboratory for examination. The second test scenario is conducted when the vehicle is in the workshop for maintenance. There, the ECU stays in the vehicle and is tested using a special device. When the vehicle is in the field, it should be possible to test the SOC autonomously either during the system operation or the system power up/down. Table 3.1 shows the relationship between the four scenarios and some factors to consider: cost, test time, test location, and suitable test type.

	Test Scenarios			
	Laboratory	Workshop	During Operation	Power On/Off
Test Time	high/low	high/low	low	low
Cost	high/low	high/low	low	low
Data location	ext/int	ext/int	internal	internal
Test Type	external	external	internal	internal

Table 3.1: Relationship between test scenarios and the test requirement

The devised test architecture should support all four test scenarios. By carefully considering the other test architectures from previous chapter, two types of SOC test are proposed: internal and external test. Internal test embeds all test components within the SOC to maintain low test time, while external test uses an external tester to keep the system cost low. The user chooses between two implementations depending on the test scenario they have.

External test is a suitable test scenario in a laboratory and in a workshop, because there is no maximum time limit to execute the test there. To run this test mode, external test requires a communication channel that connects the CUT to an external tester. The external tester can be a laptop or a low-cost portable handled device. The external tester, besides storing the fault-free response, has two other functionalities in this test mode. First, it supplies the CUT with test patterns and retrieves test responses from the CUT. Second, it compares the received and the fault free test responses and then declares whether the CUT passes or fails the core test.

Internal test, on the other hand, is the suitable test solution in the field. The test

can be conducted either when the device is in operational condition or is powered on/off. In this test mode, an external tester is unnecessary, as both of its functionalities are integrated inside the SOC. For storing the test vector, the SOC provides a storage element such as a ROM. The SOC needs a control logic for comparing the test response.

As there are many cores in a SOC, core partitioning is needed to exclusively test only one of the cores. The CUT in this thesis, which is a digital core in a SOC, is equipped with a DFT infrastructure to execute the structural test. This DFT infrastructure is used once during manufacturing test, and will be reused to run the internal and external test.

A processor, which is a common core in a SOC, has three main tasks to accommodate both test modes. It controls the communication controller to enable the data transfer between an external tester and the SOC. In addition, it controls the DFT infrastructure bridge, a core that creates a test interface to the DFT infrastructure. Furthermore, it is responsible for comparing the test response and determining if the core pass or fail the test. By using a processor as the test controller, it is possible to structurally test all but processor core. For the processor core, processor can run a self test using SBST. The general system overview of the proposed approach is shown in Figure 3.1.

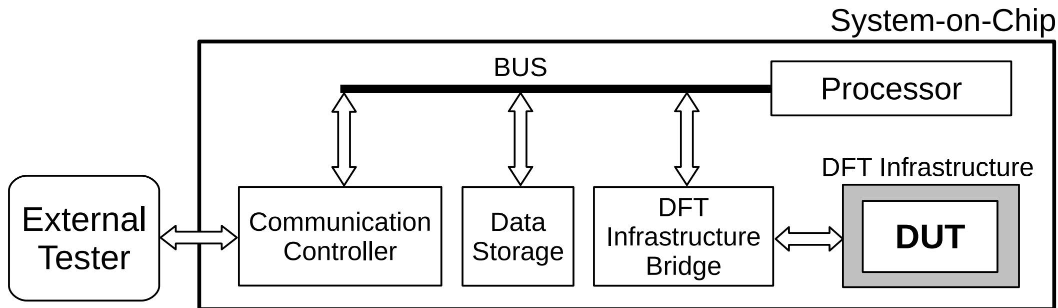

Figure 3.1: Generic test architecture for the proposed approach

4 Implementation

After introducing the system's general overview from the previous chapter, this chapter presents the implementation for each components. The explanation begins with the target system architecture, TriCore SOC, followed by the method to program it. Then, the way to connect peripherals to the system bus is presented. The data transfer in the system, which is between the tester–SOC and the SOC–DUT, is explained later. Finally, the DFT infrastructure generation is then discussed.

The architecture implementation is a big challenge for this thesis. The target architecture is a SOC platform from Infineon that comes only with source code. There is no complete, well-defined documentation resource for the TriCore SOC, except several user manuals and one example program file. To implement the proposed system, reverse engineering is used to understand how to program the TriCore and how to connect multiple peripherals to the system bus.

4.1 System

TriCore is a SOC platform that is designed for automotive application. While the released commercial version contains numerous peripherals, the essential components in the system are the TriCore processor, the data memory (DMEM), and the program memory (PMEM). The system architecture shown at Figure 4.1 is the provided simulation platform from Infineon. In addition to the main three components, it also has a LFI bridge, a Flexible Peripheral Interface (FPI) RAM, and a FPI multiplexer for supporting multiple peripherals. The FPI RAM is simply a RAM memory; it is not used in the final architecture implementation. The LFI bridge enables the TriCore processor to control peripherals that are connected through the FPI bus.

4.2 TriCore Programming

The TriCore SOC is simulated using a ModelSim, a mixed hardware description language (HDL) simulation tool. To run the simulation, the program memory and data memory must contain some data, so that the processor can execute a program. The program and data memory is generated using a Perl script, which comes together with the simulation platform. The Perl script itself needs a SRE file to generate the program and data memory. The SRE file, which is generated by Tasking compiler

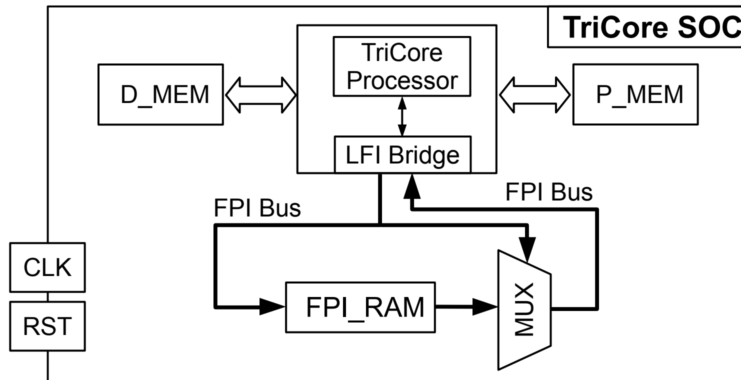

Figure 4.1: Original TriCore SOC from Infineon

from Altium, contains a series of hexadecimal number that represents processor instruction.

During the architecture development, it is known that the simulation platform address space is different from the commercial TriCore address space. The simulation parameters that are outside from the value shown in Table 4.1 must be changed. Therefore, the reset address value is changed from 0xA0000000 to 0x800B3000. The later address serves also as the boot address. In addition, the trap table address is set from 0xA00F2000 to 0x800E0000. These two configurations are available in the linker file of the compiler.

Table 4.1: Available address space in the TriCore simulation platform

0x80081000 – 0x80082000
0x800B3000 – 0x800F3000
0xD0000000 – 0xD7FFFFFF
0xF0000000 – 0xF7FFFFFF

4.2.1 Memory Mapped I/O

Memory mapped I/O is the only known method to control a peripheral through the FPI bus. This method shares the same address space for both the memory and the peripheral registers. The CPU can control the peripheral the same way as manipulating the memory content. In C programming language, the memory content is easily manipulated using a pointer.

4.2.2 Inserting Constant into SRE File

By modifying the SRE file, it is possible to insert a constant value directly inside the program memory. Table 4.2 shows an example to insert 4 byte constant at address 0x800EEB00. This constant value can be accessed by the CPU using a pointer. The valid address for the constant insertion follows the address specified in Table 4.1.

Table 4.2: An example on how to insert a constant value into a SRE file

Target address	800EEB00
Target data	012AE36E
SRE format	constant + byte count + address + switched data + CRC
Constant	S3
byte count	address length + data length + CRC length = 04 + 04 + 01 = 09
Address	800EEB00
Switched data	6EE32A01
CRC	char(09+80+0E+EB+00+6E+E3+2A+01) = char(0x2FE) = FE
Result	S3 09 800EEB00 6EE32A01 FE

4.3 Peripheral Connection

4.3.1 FPI Bus

Among many buses inside the TriCore SOC, the FPI bus is the one that interconnects the CPU and peripherals[tc109]. The bus is able transfer data up to 320 Mbytes/s and supports multiple masters and slaves. While the FPI bus slave can only accepts bus transaction, the FPI bus master is able to accept and initiate bus transaction.

The FPI bus transaction is initiated when the bus master asks for bus ownership to the system peripheral bus control unit (SBCU). The SBCU is the one that handles bus arbitration, as well as bus error. When the bus is free, the SBCU grants the bus ownership so that the bus master can send a message to the bus slave. The bus slave then receives and responses to the bus message accordingly.

There are two types of bus transfer, which are single and block transfer. Single bus transfer can transmit only one byte/half word/word data at a time. Block data transfer can transmit up to 8 words data at once.

As shown in Figure 4.2, single data transfer from the bus master to the bus slave requires at least three clock cycles. At the first clock cycle, the bus master requests a bus ownership to the SBCU. Provided that the bus is free, the bus master asserts the request signal, then applies the address and the data at cycle two and three,

respectively. The data transfer is finished at the fourth clock cycle, and the bus becomes idle.

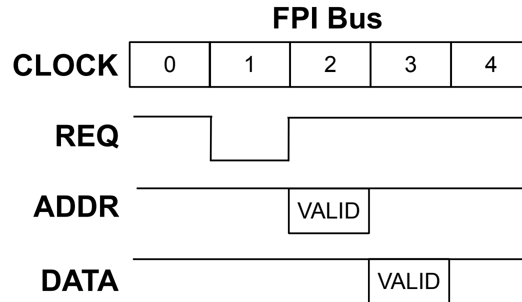

Figure 4.2: Timing diagram for single data transfer of FPI bus read and write

4.3.2 FPI Bus Multiplexer

The configuration to connect multiple slave peripherals in the FPI bus is shown in Figure 4.3. The output from LFI bridge is connected to core A, core B, and core C. The multiplexer is used to select which core is connected to the LFI bridge.

In total, three multiplexers are used to select the ready, acknowledge, and data out signals that come out from the bus slave. The multiplexer selects the active data using the respective enable signals. The multiplexer employs a priority control; when two enable are asserted, the data that has higher priority is chosen.

Ready signal indicates whether the bus slave is available to handle the upcoming data. When one data transfer is finished, the bus slave can send a feedback to the bus master whether no special condition occurs, message retry is requested, or bus error occurs. This feedback message is encoded using 2-bit acknowledge signal. Data out is simply a 32-bit data output from the FPI slave.

4.4 Test Data Transfer

The test data must be transferred between an external tester, the SOC, and the DUT. While CAN bus is used to transfer data between the external tester and the SOC, the DFT infrastructure bridge transfers the test data between the SOC and the DUT.

4.4.1 DFT Infrastructure Bridge

The DFT infrastructure bridge (DFT-IB) is developed as an interface between the CPU and the DUT. This core is designed to support both the internal and external test mode. The main difference between the two test modes is the location of the test

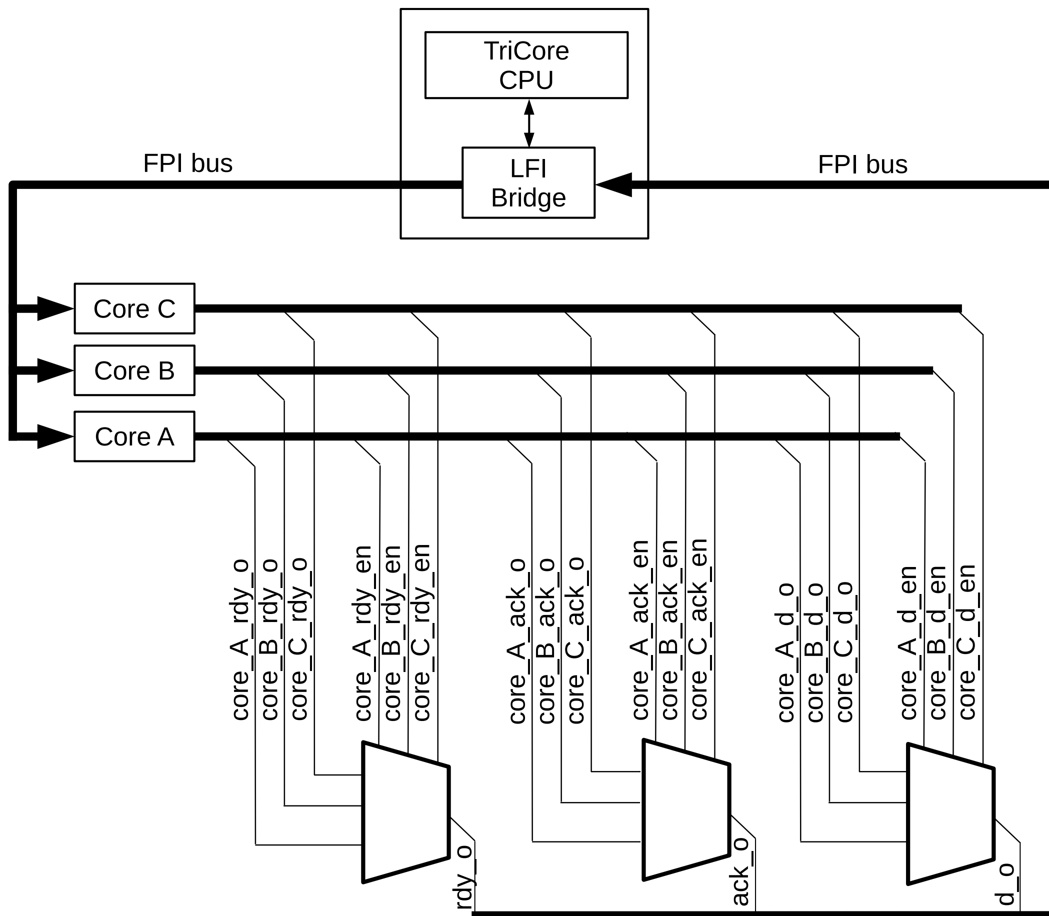

Figure 4.3: FPI bus multiplexer configuration

data. In internal test, the test data is available internally. In external test, the test data is supplied from the CPU. The test data for both test modes is predetermined.

For external test, the DFT-IB acts as a gateway between the CPU and the DUT to transfer the test data. To be more precise, the DFT-IB gets the test pattern from the CPU, applies it to the DUT, reads the DUT's test response and stores it into a register. The CPU can obtain the test response by reading the corresponding register.

For internal test, the DFT-IB runs independently from the CPU, as it has an internal access to the test data. The DFT-IB applies the internal test pattern to the DUT, gets the DUT's test response, then compares it with the fault-free test response from the internal storage. After exercising all the test data, the DFT-IB writes the test result to a register. The CPU then reads the test result through this register.

4.4.1.1 Architecture

Table 4.3 shows the four registers inside the DFT-IB. To run an internal or external test, the CPU writes the request to the command register. After executing the command, DFT-IB writes a response to register command response. Register command and command response are used for the internal and external test modes, while register test response and test stimuli are only used by the external test mode. The size of the test stimuli and test response register is flexible, depending on the input and output pins of the DUT.

Table 4.3: Registers in the DFT infrastructure bridge

Command	contains the command to be executed by the core
Command response	core response after executing the command
Test Stimuli	contains the test pattern that is applied to the DUT
Test Response	contains the test response from the DUT

Figure 4.4 shows the connection diagram between the DFT-IB and the DUT. For this configuration, the minimum size of register test stimuli and test response is 9 bit and 7 bit, respectively. Netlist t_out is used to set the DUT's input, while netlist t_in is used to get the DUT's output.

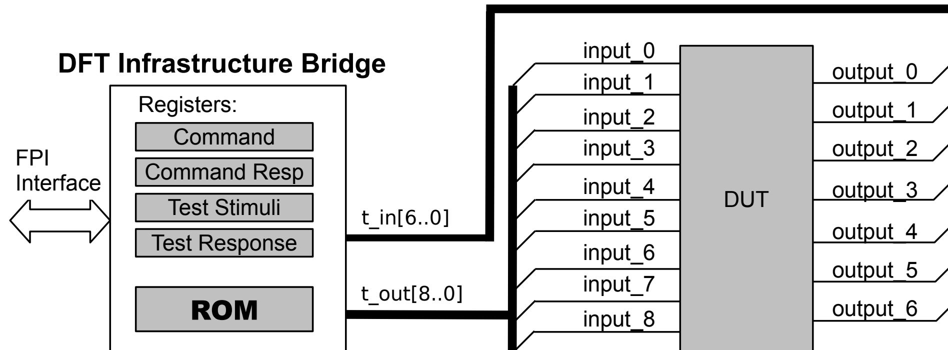

Figure 4.4: Connect between the DFT infrastructure bridge and the DUT

4.4.1.2 State Machine

Because this core is a FPI slave, it is unable to initiate a message transfer or send an interrupt to the CPU to inform a device status update. Hence, the CPU uses polling operation for controlling the DFT-IB. To read the device status register, the CPU needs to directly access the respective register through the FPI bus using

memory mapped IO. The DFT-IB must be ready to answer any inquiry from the processor, even if the device is busy performing the internal or external test. To do such mentioned tasks, the core implements two concurrent Finite State Machines (FSM) shown in Figure 4.5 for the circuit logic.

The first FSM manages the data transfer between the CPU and the device. Read-/write register access are granted only when the message has the correct address. The second FSM monitors the command register and executes the respective command. Test run time depends whether it is an internal or external test, as well as test data size. After executing the command, a response is written into the command response register

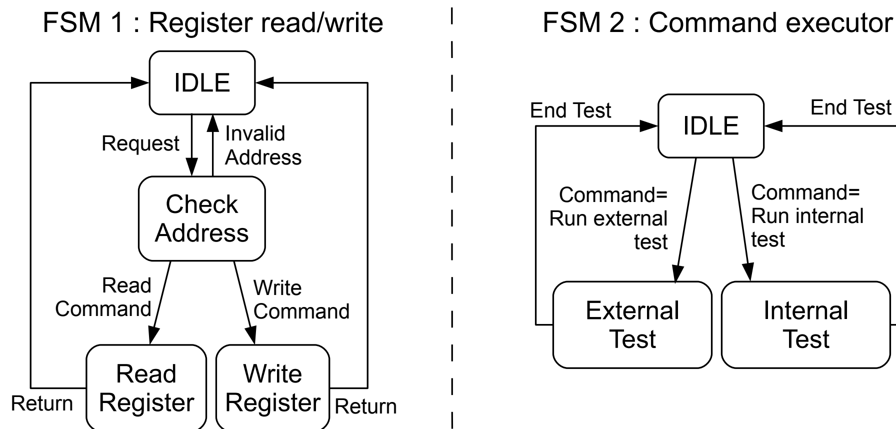

Figure 4.5: Two concurrent Finite State Machines

4.4.2 CAN Controller

In automotive domain, there are several popular bus standards: CAN, LIN, and Flexray. CAN bus is chosen as the data communication standard, as it is the most popular bus. Hence, a CAN controller is inserted to the system bus, so that processor can control the data transfer.

The CAN controller, which is a project from opencores website, supports the 8051 bus and the wishbone bus[Moh12]. Therefore, a bus translator that changes the signal from the FPI bus to the 8051/wishbone bus must be created. The wishbone bus is selected over the 8051 bus in this implementation, as it has similar behaviour as the FPI bus. The wishbone timing diagram is shown in Figure 4.6

4.4.2.1 FPI-Wishbone Bus Translator

When the bus translator detects a request signal, it collects the address and the data information. After receiving the FPI request and FPI address, the bus translator

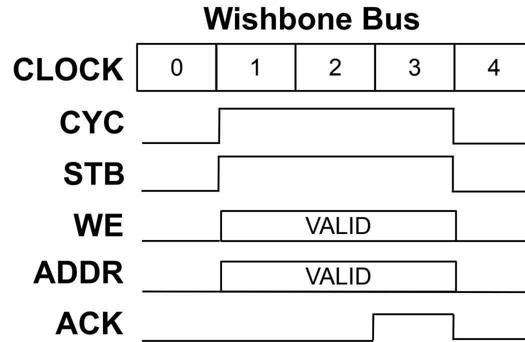

Figure 4.6: Timing diagram of wishbone bus

informs the bus master that it is busy by deasserting the FPI ready signal, so that bus master does not try to send another message. The bus translator then produces the needed wishbone signal, and waits until the wishbone slave generates the acknowledge signal. The FPI ready signal is deasserted as soon as the acknowledge signal is available. For the read procedure, in addition to this, the bus translator puts the data to the FPI line. After all of these procedures, the FPI master is free to send another message.

During implementation, sometimes the FPI master produces FPI signals that is different from the one in Figure 4.7. To avoid unknown signal sequence, a precaution is made. At clock 2 and 11, the FPI slave will request a message retry using the FPI acknowledge signal if the FPI request line is still deasserted. By doing this, the FPI master repeats sending the same message so that the generated timing diagram is similar to the one in Figure 4.7.

4.5 DFT Infrastructure for Peripherals

Design for testability for a peripheral includes a wrapper, test pattern generation, and test response compaction. For this thesis, IEEE 1149.1 and IEEE 1500 are not considered because both cover only the wrapper problem. The DFT technique that covers these three parameters is BIST and EDT. They provide the necessary wrapper, the solution to generate the test pattern, and the method to compact the test response.

In BIST, pattern generation using PRPG is a low cost solution, but not an optimum method to achieve very high test coverage in a short time. To achieve such goal, the pattern generation can use a ROM to store the deterministic test pattern. The EDT architecture provides the same functionality as the logic BIST, but requires less storage area to store the test pattern, due to the compression technique. Hence, the EDT logic is chosen over the logic BIST.

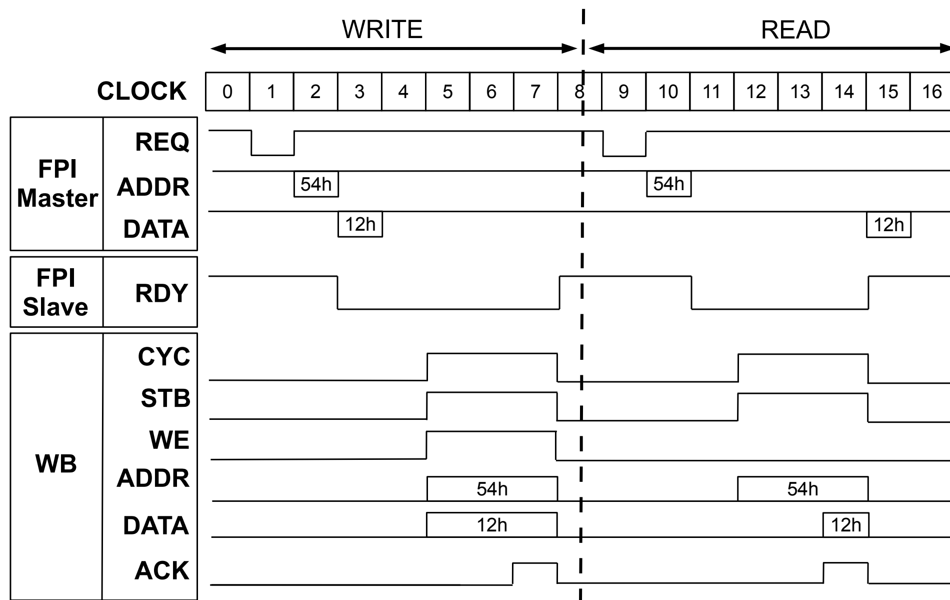

Figure 4.7: Timing diagram for translating FPI and wishbone bus

4.5.1 Tools

Mentor Graphics Tessent TestKompress is the tool used to automatically generate the EDT logic[Cor11b]. In addition, TestKompress is also an ATPG tool that is able to generate the compressed test vector for the EDT logic. To generate the EDT logic, TestKompress requires DFT advisor for inserting the scan chains [Cor11a] and a synthesis tool for synthesizing the RTL code. For this implementation, Synopsys Design Compiler (SDC)[Cor10] is used.

4.5.2 EDT Logic Generation

Figure 4.8 shows the simplified version of the DFT logic generation. First, the target core's RTL code is compiled using SDC, with core's gate level description as the result. The scan chain is then inserted into the gate level using DFT advisor. Later, TestKompress is executed to insert the EDT logic. The EDT logic is in RTL, hence it needs to be compiled once more by SDC. The second compilation produces the toplevel description of the DUT with its EDT logic. Finally, TestKompress uses this gate level description to generate the test vectors.

A very simple and small 10-bit counter is used as the target DUT, because this core takes minimum running time in terms of logic insertion and test vector generation. Figure 4.9 shows the complete transformation of the DUT. Originally, the counter has 3 inputs and 32 outputs. For the scan insertion, the DFT advisor is set to

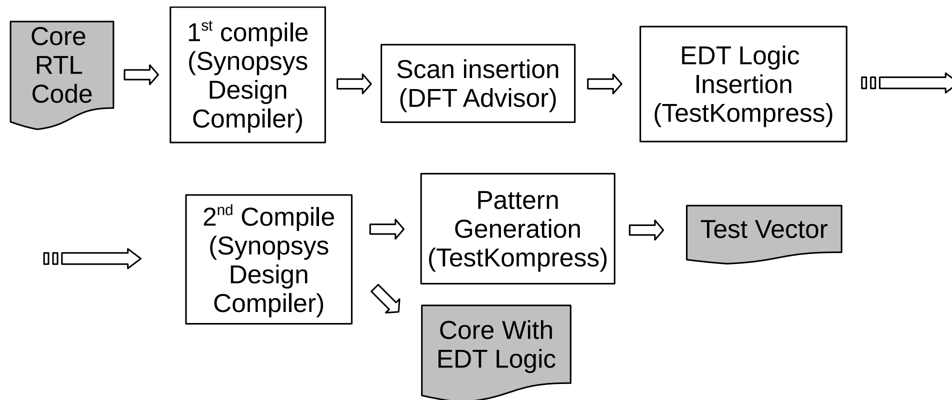

Figure 4.8: TestKompress flow to generate the DFT infrastructure

generate 4 scan channels. Hence, 1 scan enable, 4 scan input, and 4 scan output are added to the design. After the EDT logic insertion, with respect to the original design, the core has 5 additional pins: a scan enable, one pair of EDT scan channel, an EDT bypass, and a low-pin count-test (LPCT) clock. This 5 signals are necessary for controlling the EDT logic.

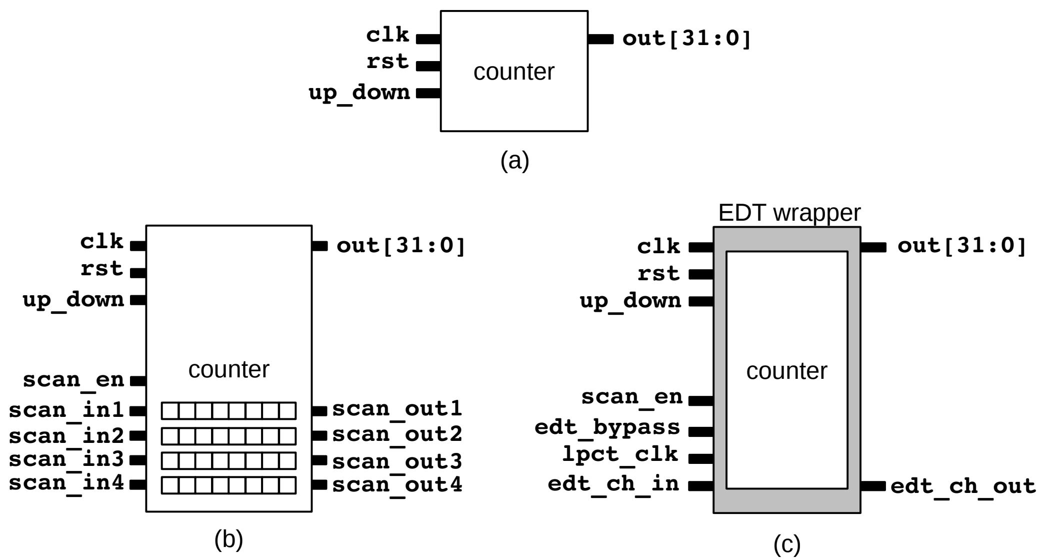

Figure 4.9: Counter toplevel transformation: (a) Original design. (b) Counter after scan insertion. (c) Counter after EDT logic insertion.

5 Test Application

Figure 5.1 shows the complete implementation of the SOC after all components are integrated. An external tester, which is simulated using another TriCore SOC, is necessary to show how the system operates. The CAN message is used as the communication medium between the external tester and the TriCore SOC. When one CAN node initiates a CAN data frame, the other CAN node replies with a CAN remote frame. The additional peripherals in the system are a counter as the DUT, an EDT logic for the DUT, a DFT-IB, a CAN controller, and two data storages for test data. The test data is available in two locations of the system: one is integrated with the program memory, the other one is included inside the DFT-IB.

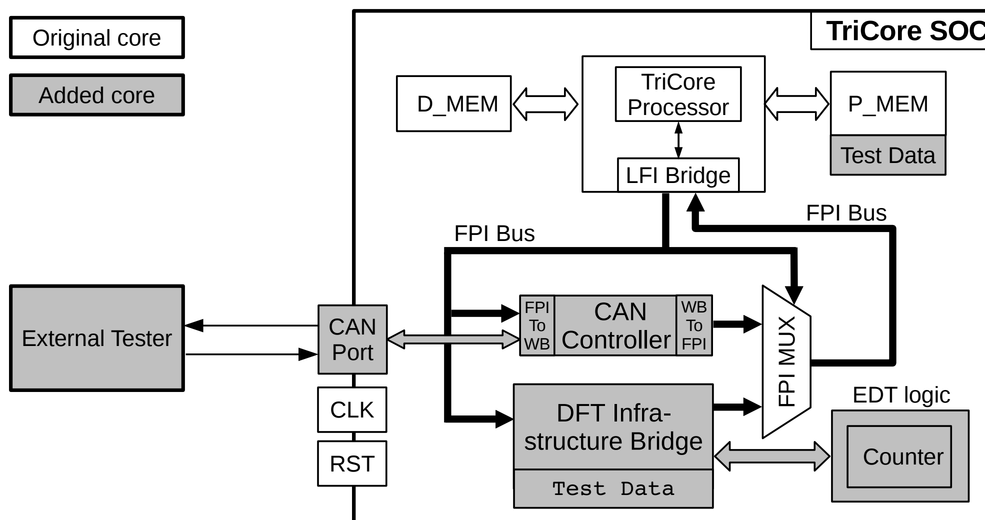

Figure 5.1: Complete test environment

The SOC runs with 80 MHz clock frequency. At this rated clock speed, transferring one CAN data frame with 8 bytes of data and CAN remote frame takes 225 uS and 80 uS, respectively. The TestKompress ATPG tool originally produces 28 test vectors. For this application, the test vectors are transformed into 1095 sets of test data by tapping the DUT I/O value during simulation. One set of test data consists of 7 bits of test pattern, 33 bits of test response, and 33 bits of test mask. This means, it takes

$$(7 + 33 + 33) * 1095 = 79935$$

79935 bit or 78 kByte to store all test data inside the DFT-IB. This information is shown in Table 5.1.

Table 5.1: CAN message overhead and DUT test data

CAN	Data frame overhead	225 uS
	Remote frame overhead	80 uS
DUT test data	Test pattern width	7 bit
	Test response width	33 bit
	Test mask width	33 bit
	Number of test data	1095 set

5.1 Internal Test

In internal test mode, the external tester presence is optional. The SOC test can be started internally by the CPU, or triggered by an external tester. Figure 5.2 shows the FSM for internal test where an external tester is available.

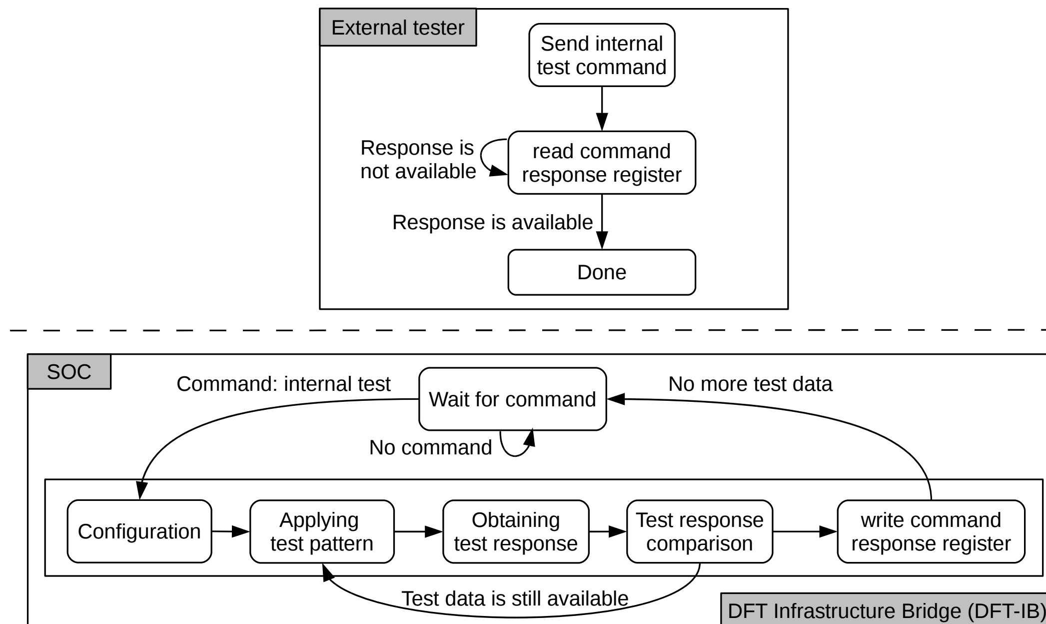

Figure 5.2: Internal test scenario

As soon as an internal test mode command is received by the SOC through the

CAN message, the CPU replies using a CAN remote frame and configures the DFT-IB register to prepare for internal test. The test data stored inside the DFT-IB is then applied to the DUT. The DFT-IB obtains the test response and compares it with the with the fault free one. Once there is a mismatch for the test response, the DFT-IB indicates the test failure in the report register. However, if there is no mismatch after all test patterns are applied, the DFT-IB writes a fault free response in the report register and notifies the external tester using CAN message. Table 5.2 presents the test result for internal test.

Table 5.2: Internal test simulation result

Number of data frame	2
Number of remote frame	2
Total test time	809 uS
DFT infrastructure test time	54.76 uS

5.2 External Test

In this test mode, the external tester provides the test data and analyzes the test response. The test process, which is shown in Figure 5.3, begins when an external tester commands the SOC to run the external test mode. The CPU receives this command and configures the DFT-IB to execute the external test. The test pattern stored in the program memory is sent by the external tester using a CAN message and the test response is sent back to the external tester for analysis. The external tester generates a command to end the test as soon as all test patterns are applied. The test result for external test is shown in Table 5.3.

Table 5.3: External test simulation result

Number of CAN data frame	1097
Number of CAN remote frame	1097
Test time for one set of test data	582 uS
Total test time	640 mS

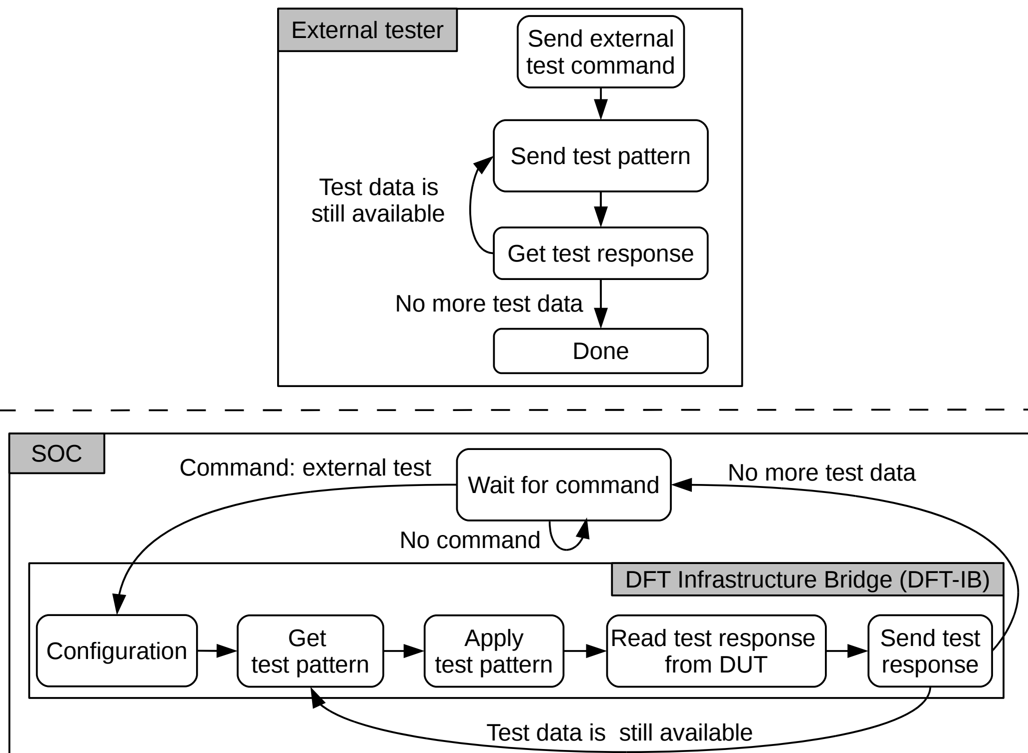

Figure 5.3: External test scenario

5.3 Experiment Analysis

The external test is proven to be a lot slower than the internal test, because of the CAN communication overhead for transferring the test data. Therefore, the external test is a suitable test case when there is no test time limitation, like in the laboratory and in the car workshop. When the long test execution time is an issue, such as testing during device operation or power up/down, the internal test can be used. The internal test can has very fast time due to the additional test storage inside the SOC.

5.4 Future Work

The DUT in this implementation is a very simple digital core, hence the test data size is negligible. However, test data size will increase dramatically when the DUT is a complex digital core. To reduce the test data, the system can only store the test response signature, i.e. a compressed test response data created by a MISR.

The developed DFT-IB is able to partition and test only one DUT. Using this

approach, testing n cores in the SOC requires n number of DFT-IB. As each DFT-IBs need its own address space, the current solution will use too much memory address space. The next version should be able to interface not only one core, but all cores. To do this, the hardware connection between the DFT-IB and the core should be simplified. This can be realized by using the boundary scan concept from IEEE 1149.1 and IEEE 1500.

6 Conlusion

This thesis has successfully developed a method to conduct in-field tests for digital cores inside a SOC. The method is realized by inserting an internal test interface to gain control of the existing DFT infrastructure. By connecting the test interface to the system bus, test controller acquires access to the DFT infrastructure to conduct a core test. The test controller, which is a processor, can execute the structural in-field tests as long as test data are available in the system.

The test architecture is able to run core tests with or without an external tester. Internal test integrates all components within the SOC to run the test and requires an optional external tester. This method has very low test time, as the communication overhead between the DUT and the tester is kept to a minimum. The disadvantage of internal test is the need of additional storage inside the SOC to store the test data. External test requires a dedicated external component to store the test data, which is later transferred to the DUT through the CAN bus. Test time is considerably high compared to the internal test because of the CAN communication overhead. Nevertheless, this method demands minimal modification at hardware level.

Bibliography

[AAMH98] H. Al-Asaad, B.T. Murray, and J.P. Hayes. Online bist for embedded systems. *IEEE Design Test of Computers*, 15(4):17–24, Oct-Dec 1998.

[AS11] S. Arslan and G. Shah. A flexible in-field test controller. In *IEEE 14th International Multitopic Conference (INMIC)*, pages 71–75, Dec 2011.

[BA00] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing*. Kluwer Academic Publishers, 2000.

[Bor05] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov-Dec 2005.

[Cha09] Robert N. Charette. *This Car Runs on Code*. IEEE Spectrum, 2009.

[Cor10] Mentor Graphics Corporation. *Design Compiler User Guide*. 2010.

[Cor11a] Mentor Graphics Corporation. *Scan and ATPG Process Guide*. 1999-2011.

[Cor11b] Mentor Graphics Corporation. *Tessent TestKompress User's Guide*. 2011.

[Gmb91] Robert Bosch GmbH. *CAN specification Version 2.0*. 1991.

[JCJ+11] Seo-Hyun Jeon, Jin-Hee Cho, Yangjae Jung, Sachoun Park, and Tae-Man Han. *Automotive hardware development according to ISO 26262*. 13th International Conference on Advanced Communication Technology (ICACT), 2011.

[JT99] A. Jas and N.A. Touba. Using an embedded processor for efficient deterministic testing of systems-on-a-chip. In *International Conference on Computer Design, 1999. (ICCD '99)*, pages 418–423, 1999.

[LCH05] Kuen-Jong Lee, Chia-Yi Chu, and Yu-Ting Hong. An embedded processor based soc test platform. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 2983–2986, May 2005.

[Mak07] T.M. Mak. *Infant Mortality - The Lesser Known Reliability Issue*. 13th IEEE International On-Line Testing Symposium, 2007.

[Moh12] Igor Mohor. *CAN protocol controller.* http://opencores.com/, 2003-2012.

[PG05] A. Paschalis and D. Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):88–99, Jan 2005.

[PGSR10] M. Psarakis, D. Gizopoulos, E. Sanchez, and M.S. Reorda. Microprocessor software-based self-testing. *Design Test of Computers, IEEE*, 27(3):4–19, May-June 2010.

[RTKM04] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee. Embedded deterministic test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(5):776–792, May 2004.

[SABR04] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, June-July 2004.

[tc109] *TC 1797 User's Manual v1.1.* Infineon Technologies AG, 2009.

[VGPH05] I. Voyiatzis, D. Gizopoulos, A. Paschalis, and C. Halatsis. A concurrent bist scheme for on-line/off-line testing based on a pre-computed test set. In *Proc. IEEE International Test Conference (ITC)*, Nov 2005.

[WST08] Laung-Terng Wang, Charles E. Stroud, and Nur A. Touba. *System on Chip Test Architecture.* Morgan Kaufmann, 2008.

[Wun98] H.-J. Wunderlich. Bist for systems-on-a-chip. *INTEGRATION: The VLSI Journal*, Vol.26:pp.55–78, 1998.

[WWW06] Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures Design for Testability.* Morgan Kaufmann, 2006.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(David Prasetyo Buntoro)