

# Protocol Requirements in an SJTAG/IJTAG Environment

Gunnar Carlsson<sup>1</sup>, Johan Holmqvist<sup>2</sup>, Erik Larsson<sup>2</sup>

PDU Base Station<sup>1</sup>  
Ericsson AB  
Sweden

Department of Computer Science<sup>2</sup>  
Linköping University  
Sweden

contact: gunnar.carlsson@ericsson.com, erila@ida.liu.se

**Abstract -- Integrated Circuits, Printed Circuits Boards, and Multi-board systems are becoming increasingly complex to test. A major obstacle is test access, which would be eased by effective standards for the communication between devices-under-test (DUTs) and the test manager. Currently, the Internal Joint Test Access Group (IJTAG) work at micro-level on a standard for interfacing embedded on-chip instruments while the System JTAG (SJTAG) work at macro-level on a standard for system-level test management that connects IJTAG compatible instruments with the system test manager. In this paper we discuss requirements on a test protocol to be used in an SJTAG/IJTAG environment. We have from a number of use scenarios made an analysis and defined protocol requirements. We have taken the Standard Test and Programming Language (STAPL), which is built around a player (interpreter), and defined required extensions. The extensions have been implemented in an extended version of STAPL and we have made experiments with a PC acting as test controller and an FPGA being the DUT.**

focus on how IEEE 1149.1 can be used for test access to a variety of test-and-measurement instruments embedded on-chip such as LBIST (Logic Built-In Self-Test) controllers [2], [7], [8]. The instruments can support device characterization, as well as structural and functional test at device level. The standard will probably include a description language for characterization of the embedded instruments, a protocol language for communicating with the instruments and a method for interfacing to the on-chip instruments.

The System JTAG (SJTAG) group with focus on macro-level tries to find system-level solutions to address testing and troubleshooting [4]. Multi-board implementations are often based on IEEE 1149.1 as the backplane test bus for accessing tests embedded on-board or for downloading tests that are applied as part of a field service application. The SJTAG group has identified the need for a language or communications protocol for command control and for test data delivery and response. The communication takes place between a test manager, which is responsible for the overall test application management, and a test controller, which is an embedded function in control of the vector delivery, see Figure 1. The test manager in such a system-test deployment could be local or remote. The language should be independent of any vendor of IEEE 1149.1 based test development platforms and the language should support the following basic requirements on the test controller:

- embed test data (vectors) efficiently,
- read, write and manage test data stored on the board,
- run embedded test,
- configure and validate an on-board programmable logic devices,
- capture the result of the test and compare result with expected result,
- log test execution details such as time, date, result, etc.,
- send specific test reports and service logs to the test manager.

In this paper we focus on the test data delivery and response aspects. In particular, we:

- define use scenarios that must be handled,
- analyze the scenarios in order to define protocol requirements,
- investigate to what extent Standard Test and Programming Language (STAPL), which is built around a player (inter-

## I. INTRODUCTION<sup>1</sup>

The IEEE 1149.1 Boundary scan Standard, developed by the Joint Test Action Group (JTAG), has become successful for testing Printed Circuit Boards (PCBs) [5]. The standard addresses the access problem for the increasingly crowded PCBs. However, IEEE 1149.1 is not only used for PCB testing; it is nowadays also used for the testing of Integrated Circuits (ICs) and as the test bus in the backplane of multi-board systems.

As ICs, PCBs and multi-board systems are becoming increasingly complex to test there is a need for further standardization both at device-level and at system-level. There is an especial need for standards on test access. These standardization initiatives will most likely be based on IEEE 1149.1 due to its wide usage and acceptance.

At device-level, micro-level, the Internal JTAG (IJTAG)

---

<sup>1</sup> The research is partially supported by the Strategic Integrated Electronic Systems Research (STRINGENT) programme.

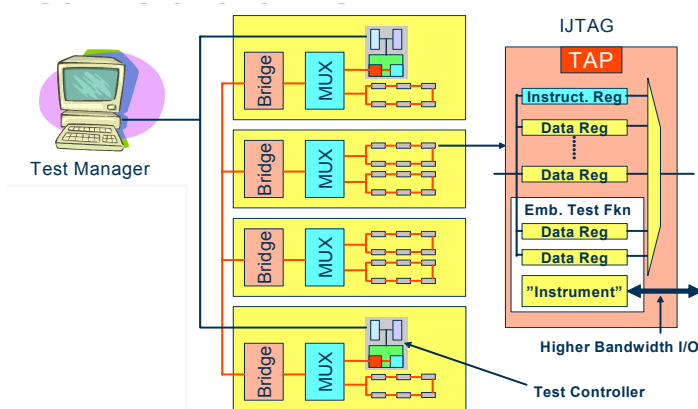


Figure 1: System perspective: A test manager connected to multi-board system where boards are equipped with IEEE 1149.1 and IJTAG instruments.

- preter) and a language, can handle the requirements,
- define required extensions to STAPL, the player and the language, to meet the requirements,
- implement the required extensions in STAPL and
- make experiments with a PC as test controller and an FPGA as DUT to show the feasibility.

The paper is organized as follows. Section II describes the SJTAG environment; test requirements and Section III gives an overview of IEEE 1149.1, IJTAG, and STAPL. The modifications to STAPL are introduced in Section IV and an example to show the usefulness of extending STAPL is in Section V. A discussion is in Section VI and conclusions and future work are in Section VII.

## II. SYSTEM JTAG ENVIRONMENT

Figure 1 shows the test manager communicating with a multi-board system. For such a system, a number of use scenarios can be defined. The following use scenarios has been proposed by the SJTAG group:

- Low Level HW and SW Debug during Development
- Production Test and Programming
- Test in the Field
- Repair Diagnosis
- Test Program Verification and Debug

The use scenarios are detailed below.

### 1) Low Level HW and SW Debug during Development

This scenario applies when only HW and low level SW, e.g. drivers and similar functionality, are available. Drivers and other low level SW are downloaded to the system processor(s) along with functional test routines and a simple command interpreter. The test program is mainly a set of low level procedures. Some of these may be interacting functions, which need to run in parallel, and be synchronized by TCK. External

instruments may also be involved. The procedures are called from scripts via the test manager. Flow control is mainly done through the scripts. Response data from the procedures are mainly collected as raw TDO data via the test manager. Diagnosis is done manually, using scripts or by ad hoc SW.

The debug operations are typically fairly low level operations, e.g. reading and writing to registers, single stepping of instructions, etc. This requires that the user can use the test manager to apply low level operations on the system through the test manager.

### 2) Production Test and Programming

The production test and programming at the system level is often divided into:

- Structural test, verifying interconnects and IC internal structures.
- Functional test, mainly verifying performance and interfaces.
- Measurements of critical parameters and conformance to standards and regulations.
- Programming of ID, application SW and parameters.

*Structural test* typically consists of a Boundary scan interconnect test, including inter-board connections and other system level interconnect, Boundary scan based at-speed interface test, memory tests, and IC BIST tests.

The Boundary scan interconnect test is ATPG generated, and makes use of functions for configuration of scan chain segments (e.g. bridges and linkers), if present in the system. Procedures to support such functions should be provided by the configuration element (components and IP) vendors, and understood by the tools. Diagnosis of the interconnect test has dependencies on the ATPG algorithms, and is best suited to take place on a platform related to the ATPG. The test program should take care of comparisons, and the test controller should forward bad return TDO data to the test manager.

The structural at-speed interface test requires that procedures that operate on related functions can run in parallel and be coordinated. The requirement on parallelism has implications on the test program language, but the co-ordination is mainly a design issue.

Memory test and BIST operations also need parallel execution of procedures, but rather from an efficiency viewpoint. On the other hand, it may not be possible to execute all procedures at the same time, e.g. for power consumption reasons. For test program development, procedures to configure scan chain segments at the system level as well as procedures to configure the component level setup are needed. The component level configuration may even be hierarchical. To diagnose parallel operations, information on the current configuration of the registers, which constitutes the scan chain at the point of failure, is needed. However, the leaf test procedures are expected to be responsible for comparison and report to the test manager, via the test controller. Only data from the faulty segment of the scan chain needs to be transferred to the test manager.

*Functional test and Measurements* are similar to the HW and

SW Debug, as described above. A difference is that comparisons are made rather than collection of values.

*Programming* becomes easier using procedures for the programming operations and data that is independent of the programming algorithm. Data formats of the test program language, and flow control features, such as loops, may contribute to make the test programs more compact. Parallel programming of several devices may enhance efficiency, thus requiring the test program language to support parallel execution of procedures.

### 3) Test in the Field

Test in the field may be part of fault management in the operation and maintenance (O&M) subsystem. In this case, the test manager is an integral part of the O&M SW. In other cases, an external test manager may be connected to the system, at field service, to take control of the test operations done by the test controller. This may typically be the case with small embedded systems without great O&M capabilities.

The major reasons for test in the field are to find latent faults before they get an impact on system operation, and to verify HW integrity at fault indications during system operation. The test efficiency, i.e. test time vs. coverage, is often a crucial characteristic. Beside verifying whether a fault exists, the purpose is to identify either which part to replace or which unit to fault-mark and take out of operation. The latter is important if the system operation can continue with reduced capacity even though a unit is faulty, given that the faulty unit is excluded from the system operation (often called graceful degradation).

Tests may be executed both under O&M control and on operator control. The tests may be directed to units at different levels of the system hierarchy. For this reason, the system hierarchy must be reflected in some sense in the test program, and supported by the procedures, which in turn may be hierarchically arranged. A simple example is procedures for operations on different BIST controllers in a component, called by a procedure for test of the component, which in turn may be called by a procedure for test of a board, etc.

The test program may be structured in phases, where the first phase is to verify whether a fault exists (go/nogo). This phase is time critical. The next phases are aiming at pinpointing the failing unit. The latter, diagnostic parts of the test, may even be adaptive, i.e. depending on test results the test flow may take different directions. Since the diagnostic parts of the test may have an impact on the operation of the system (parts of the system may be allowed to be turned on again), it is reasonable that the test manager (which in turn is controlled by O&M), has control of the test execution flow. This also allows the operator to perform directed tests.

Programming in the field of volatile devices, such as FPGA, takes place at system or board start/re-start. The programming files may be updated remotely. Programming of other devices, e.g. PLD and flash memories may take place during system upgrade, which could be remotely controlled. In both cases the same mechanisms as in test operations could be used, i.e. the

test manager and test controller(s). In the field, similar requirements on programming as in production, as described above, apply.

Debug of microprocessor code and DSP code may also occur in the field. The debug procedure as well as the related requirements are quite similar as those for debug during development, as described above.

### 4) Repair Diagnosis

At repair diagnosis, field returns or failing units from system test in production are inserted in a known good reference system, and tested in an environment which resembles the one in which the unit failed. The purpose is to verify the fault and to find the root cause down to a level which translates to a repair action.

The faults may be sensitive to environmental conditions, which makes them difficult to repeat in the reference system. Hence, sections of the test program need to be able to loop while the environmental conditions are stressed. Further, external instruments may be needed, which also may require looping and other active control over the program flow.

While production test programs usually stops on failure, repair tests sometimes need to continue at failure, but report failing results during execution. Such results may be used with fault dictionaries to support diagnosis. Other types of reporting may also be required during execution, e.g. report of measurement values (analog-to-digital converted results from embedded instruments).

The operator usually wants more flexibility than in production test in terms of changing the test program on the fly. It can be to change an algorithm or add a test. This requires opportunities to send additions and changes from the test manager to the test controller, which in turn must be able to change the stored test program.

### 5) Test Program Verification and Debug

Test program verification and debug is similar to conventional SW verification and debug. The addition is that faults need to be inserted in the DUT, which can be a challenge in some cases. Typical actions are setting breakpoints, writing trace messages from the test program, and read and write to variables.

## III. OVERVIEW OF STANDARDS

In this section we give a short introduction to the IEEE 1149.1 (Boundary scan) standard, the IEEE P1687 (IJTAG) standard under development, and the Jam Standard Test and Programming Language (STAPL) which is adopted as JEDEC standard JESD-7.

### A. IEEE 1149.1

The IEEE 1149.1 (Boundary scan) was designed for testing PCBs; however, today it is also used for testing sub-blocks of integrated circuits, and is useful as a mechanism for debugging

embedded systems, providing a convenient "back door" into the system. The standard adds a special four/five-pin interface to a chip, designed such that multiple chips on a board can have their lines daisy-chained together; a test probe need only connect to a single "JTAG port" to have access to all chips on a circuit board. The connector pins are TDI (Test Data In), TDO (Test Data Out), TCK (Test Clock), TMS (Test Mode Select), and an optional TRST (Test ReSeT). Figure 2 shows two devices equipped with Boundary scan.

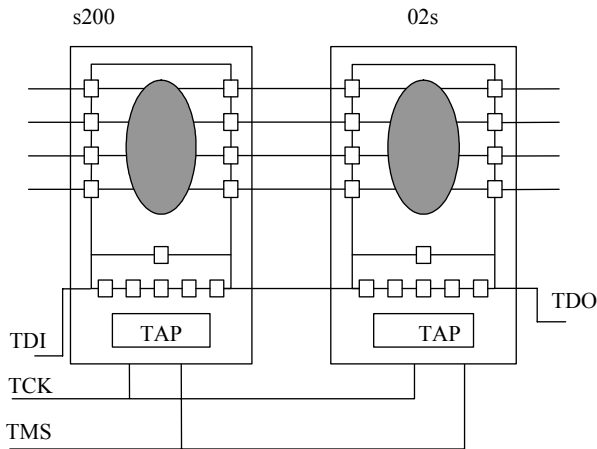


Figure 2: Two devices (s200 and 02s) with Boundary scan.

### B. IEEE P1687 - Internal JTAG

The IJTAG P1687 statement of scope: "This standard will develop a methodology for access to embedded test and debug features, (but not the features themselves) via the IEEE 1149.1 Test Access Port (TAP) and additional signals that may be required. The elements of the methodology include a description language for the characteristics of the features and for communication with the features, and requirements for interfacing to the features" [2], [3].

The P1687 standard focuses on access and control of instrumentation embedded within a semiconductor device; hence defining the interface to instruments. Examples of instruments are scan, BIST, Memory BIST, compressors, debug logic, and so on. Note that P1687 will define the interface to instruments but not the instruments.

### C. Standard Test and Programming Language (STAPL)

The Jam Standard Test and Programming Language (STAPL) developed by Altera [1] was adopted as JEDEC standard JESD-71 in August, 1999. The Jam Player is software that reads the Jam File and applies vectors for programming and testing devices in a IEEE 1149.1 Boundary scan chain.

A STAPL program consists of the following elements:

- NOTE statement
- ACTION statement
- PROCEDURE blocks and DATA blocks
- CRC statement

The NOTE statements include description on documentation and functionality of the program. The ACTION statement is a sequence of procedures describing an operation. A PROCEDURE includes STAPL statements, and DATA block includes variable declarations. CRC is for verification of the file. An example of a STAPL program is in Figure 3.

#### 1) Program Flow

The execution starts by a user selecting an ACTION that is to be executed. When the ACTION is terminated, the program stops. If another action is to be executed, the player has to be restarted. PROCEDURE blocks are called in order of appearance. A PROCEDURE block is terminated by an ENDPROC statement. CALL statement can be used to initiate execution of procedures from a procedure. GOTO statement can be used to jump within a procedure.

There is no support for linking several STAPL files and it is not possible to include other files in a STAPL file.

#### 2) Data Management

All variables that are to be used must be declared. A variable declared in a PROCEDURE can only be used within that PROCEDURE block. A variable declared in a DATA block can on the other hand be shared. STAPL supports 32-bit signed numbers and BOOLEANS. Single dimension arrays can be used.

#### 3) Input and Output

The input and output is handled through IEEE 1149.1.

## IV. EXTENSIONS

We have extended the STAPL language and the way test programs are executed (the STAPL player; interpreter). The language extensions include object oriented constructs for handling hierarchy and parallel constructs for handling parallelism. We have introduced INSTANTIATE, CLASS, BODY, PARALLEL, and we have made modifications to IRSCAN, DRSCAN, IF, and EXPORT. The introduced extensions are detailed below. All extensions in the language have been done such that the language remains backwards compatible with the original STAPL. Hence, programs written for standard STAPL can be used in our extended STAPL version.

#### A. Program execution modifications

Executing a STAPL program means that ACTIONS are called and when EXIT is reached, the program terminates. In order to run a new ACTION, the program must be restarted. We have therefore extended the program flow such that the program is not terminated. Hence, several ACTIONS can be called after each other without having to restart the player. Further, as we would like to allow the possibility to execute parts of the tests in an interactive way, alternatively under control by diagnostic SW, we have made it possible to directly call procedures and access variables.

```

ACTION run_test = bypass,check_id;

DATA maindata;
  BOOLEAN test_fail;
ENDDATA;

PROCEDURE bypass USES maindata;
  BOOLEAN result_2[2];
  'Load the BYPASS instruction
  IRSCAN 14, $3FFF;
  'Scan "11", compare "00"
  DRSCAN 2, #11, CAPTURE result_2[], COMPARE #00, #11, test_fail;
  IF test_fail THEN GOTO 11;
  EXPORT "FAILURE TYPE BYPASS_TEST", result_2[];
  EXIT 17;
  11:
ENDPROC; 'bypass

PROCEDURE check_id USES maindata;
  BOOLEAN result_64[64];
  'Load the IDCODE instruction
  IRSCAN 14, $09FE;
  'Check the ID code
  DRSCAN 64, $0000000000000000, CAPTURE result_64[], COMPARE $F1414093F5045093, $FFFFFFFFFFFFFFFF,
test_fail;
  IF test_fail THEN GOTO 12;
  EXPORT "FAILURE TYPE IDCODE_TEST", result_64[];
  EXIT 2;
  12:
ENDPROC; 'check_id

```

Figure 3: STAPL program with an action that does bypass and check\_id.

## B. Hierarchy

A hierarchical representation is introduced to:

- ease the reuse of code,
- enable context free operation on only a segment of the TDI-TDO chain,
- ease access of variables and procedures from the test manager, and
- to make the test program more readable.

Reuse of code is useful for embedded instruments such as LBIST and MBIST where there usually are a number of instances. Instead of reproducing code for each such instrument, the same code can be used for several of the same type. The test programs also become more readable and easier to overview if a hierarchical representation is used. The structural representation also makes it easy to access variables and procedures at any level in the hierarchy from a Test Manager, which is useful at manual operation of the test features of a system, e.g. at debug or diagnosis. Further, diagnosis is simplified since the processes can return error messages and data only relevant for the particular part of the TDI-TDO they operate on.

INSTANTIATE is used to create and define boards, components, and embedded instruments (also hierarchical implementations) as CLASS objects.

CLASS defines a PCB, a component and so on and INSTANTI-

ATE creates an instance of the class. CLASS declaration includes a PROCEDURE and a DATA block. In the DATA block, variables common for the class are declared. These can be seen as variables global to the CLASS, but local in the test program scope. At instantiation of a class, the instance is given a unique name. This name is used to access variables and procedures. For example, to access the variable result, device.result\_32 is used. And to run the procedure id\_check, CALL board.device2.id\_check is used.

A BODY expression is introduced to allow CLASS objects to be instantiated at different hierarchical levels. Every CLASS declaration includes a BODY expression that is executed when the CLASS object is instantiated.

In the original STAPL, all variables are global and every variable name must be unique. We have, in order to ease the development of test programs, changed the scope of variables. We have introduced local variables. These local variables do not need to have unique names. The same variable name can be used in other classes or in other hierarchical levels. It means that local variables are invisible outside of the class. Test programs developed for standard STAPL where no hierarchy is used can still be executed.

### C. Parallel block

Procedures sometimes need to run in parallel for efficiency reasons or to coordinate test activities. Further, the parallel procedure concept also supports dynamic configuration of the Boundary scan chains, both at system level through bridges and routers, and within components. Parallel execution means that the procedures operate on the IEEE 1149.1 pins simultaneously, and that their TDI and TDO operations at IRSCAN and DRSCAN are concatenated. To support parallel execution, the PARALLEL block has been introduced.

Parallel execution implies that procedures that operate on the Boundary scan chain are synchronized and that the TDI data from IRSCAN and DRSCAN are connected to a TDI data sequence and that the long TDO data sequence is partitioned into sequences that corresponds to the TDI data.

In order to implement these operations we have introduced the PARALLEL statement. Procedure calls within a PARALLEL block will be executed pseudo parallel, which means that the procedures within the block are executed sequentially until an operation on the Boundary scan chain is to be executed. As soon as an operation on the Boundary scan chain is detected, all procedures that operates on the Boundary scan chain are stitched together as the TDI data. The TDI data is formed according to the order in which the procedures are called; which is the same as in which the components are connected in the Boundary scan chain. The created TDI data is executed and the collected TDO data is partitioned according to the procedure calls such that the TDO data for each component can be distinguished. The TDI data is concatenated in the order of the procedure calls, and the composite scan operation is executed. The TDO data is then distributed in the same fashion between the procedures to the CAPTURE and COMPARE part of the statements. After this, the execution continues in pseudo-parallel until the next IEEE 1149.1 pin operation is detected.

Procedures in PARALLEL blocks may call other procedures, which in turn contains parallel blocks. This allows the TDI-TDO data stream to be expanded when diving through the system hierarchy (which is exploited in the example), e.g. when lower level instance registers are linked into the TDI-TDO path.

EXIT statement in between scan operations are collected and returned together to the calling procedure simultaneously. The idea behind this is that no statement should be lost or missed in the case that the program execution is stopped due to a test failure. Other parallel procedures that do not contain EXIT statements will continue until next IEEE 1149.1 operation or ENDPROC is found.

### D. Other extensions

We have also made minor modifications to the statements IRSCAN, DRSCAN, IF, EXPORT and EXIT.

#### 1) IrsCAN and DrsCAN

In standard STAPL IRSCAN and DRSCAN can take zero, one

or two arguments. In order to be able to perform a check if the result from a scan operation is correct or not, and, if it is not correct, store the response, there is need for several arguments in the test program.

We have done a modification of the scan commands such that they can take both CAPTURE and COMPARE as arguments. Thus, it is possible on a single line to both check the data that is returned and in the case of a fault, store it for later export.

#### 2) IF statement

In order to make the IF statement more powerful we have extended the IF statement such that it not only can handle single statements but also a BEGIN/END block.

#### 3) EXPORT

The export statement is in STAPL limited to a single tuple, e.g. <key string> <value>. In order to make the return possibilities more powerful we have extended EXPORT to include an arbitrary number of tuples.

## V. EXAMPLE

We make use of the very small example in Figure 2 to demonstrate the principal differences between standard STAPL and the extended STAPL. First we discuss test program development and then we discuss test program modifications.

### A. Developing programs

The standard STAPL program in Figure 3 does the same test on the example (Figure 2) as the extended STAPL program that is described in Figure 4, Figure 5, and Figure 6.

The extended STAPL program is longer in text; however, note that much of the code is identical. The differences between the classes are the length of the instruction register, the name of the class, and the instruction code to perform id check.

Note also that in extended STAPL, each class is only describing itself. Hence, there is no system level information at a class/component. The advantage is that each component can be developed without system knowledge. In the case of standard STAPL, each procedure contains information on which part of the Boundary scan it operates on. In extended STAPL, the respective position for each component in the Boundary scan chain is given by the Procedure instance, which instantiate each component and defines its place in the Boundary scan chain (Figure 4).

### B. Adding an additional PCB

Suppose we would like to extend our system by adding an additional PCB that is identical to the previous one. We have then a system with two identical PCBs as in Figure 2. It means that the Boundary scan chain must be extended with the additional PCB. Changing the Boundary scan chain implies that some modifications must be made to the programs that performs the tests. Every statement that operates on the Boundary-

```

ACTION run_test = instance,bypass,check_id;

PROCEDURE instance;
  INSTANTIATE s200 device1;
  INSTANTIATE 02s device2;
ENDPROC; 'instance

PROCEDURE bypass;
  PARALLEL;
    CALL device1.bypass;
    CALL device2.bypass;
  ENDPARALLEL;
ENDPROC; 'bypass

PROCEDURE check_id;
  PARALLEL;
    CALL device1.id_check;
    CALL device2.id_check;
  ENDPARALLEL;
ENDPROC; 'check_id

```

Figure 4: The actions and procedures.

Scan chain, i.e. the DRSCAN and IRSCAN statements, in the program written in standard STAPL must be modified. The length of the scan operations, the data sent to chain and the reference data must be modified.

For the program written in extended STAPL where there is no system information at each component/class, and therefore there is nothing to modify at each component/class description. Instead, all modification are to be made at system level. In this example, it is enough to add two statements in the instantiate procedure and two statements in each PARALLEL block. This is obviously much easier than editing hexadecimal vectors.

For the example, we have to make 14 modifications in the standard STAPL program and only six modifications to the extended STAPL program. The modifications to the STAPL program consists most of editing hexadecimal vectors while the modifications to the extended STAPL program consists of modifying object instantiation.

The example is written for a Boundary scan chain consisting of two ICs only, which is to be considered as an extremely small design. It is not unusual that PCBs consist a much higher number of components (ICs) where each have several BIST controllers. Normally the BIST controllers are identical which increases the possibility for reusing code. Writing code in standard STAPL for larger systems soon becomes confusing; however, in extended STAPL it is more about adding/creating more instances, which is done in a single line, since the code for the BIST controllers often are the same.

It is possible in standard STAPL to write code similar to the extended STAPL by using the PRE and POST statements where each DRSCAN and IRSCAN statement operate only on a part of the Boundary scan chain. There are disadvantages with using the PRE and POST statements. A Boundary scan chain would demand twice as many operations on the Boundary scan chain using PRE and POST statements as if one of the

two solutions (standard STAPL without PRE and POST and extended STAPL). The number of operations on the Boundary scan chain increases for every new component added to the scan chain, e.g. four components demands four times as many operations on the Boundary scan chain. Another disadvantage is that two or more components cannot be executed in parallel, only in sequential. It means that data from other components, which corresponds to the PRE and POST statements, would be lost. Using PRE and POST statements also presumes that the programmer has knowledge of how the scan chain is mapped at all time.

## VI. DISCUSSION

The rationale behind this work is to elaborate on requirements for embedded Boundary scan based tests in a system context. We chose STAPL as a basis for a test language since a set of public domain supporting SW is available, and since most commercial Boundary scan development platforms can generate test programs, which are easily converted to STAPL. This gave us a jump start into building a demonstrator to carry out experiments on. However, STAPL may not be the ideal candidate language, but since it is well established in the board test and programming community, it should not be immediately excluded either.

Our preliminary rough requirements included parallelism in test execution, reuse of test procedure code, and management of hierarchy, which in turn implies some form of structural representation of the system under test. Most of these requirements have been met with our experimental setup, however somewhat clumsy, due to the restrictions set by the original STAPL player and language constructs.

Some of our earlier findings are related to requesting higher level chain operations from lower level embedded test functions, e.g. requesting a series of DR scan operations on different DR register segments, which in turn requires interleaving different IR scan operations at a level, not known by the lower level function. This is not yet properly handled by our demonstrator. Another issue is to manage parallel operation of procedures in different components, which in turn have different number of required DR or IR scan operations. Today, this is handled implicitly in the demonstrator, guided by a set of rules.

The plan is to continue refining the requirements on the embedded test language, but using a more versatile experimental platform. One possibility would be to build a new extended STAPL player from scratch, but it is probably better to use a platform which already has inherent capabilities to handle objects and processes, such as a conventional object oriented language. To further extend this idea, we are discussing to model both the test program and the vector delivery mechanism in xUML (executable Unified Modelling Language). With this approach, we can both run the model, and also generate e.g. C++ or Java code, to execute in a real system.

```

CLASS s200;
  DATA maindata;
    BOOLEAN test_fail;
    BOOLEAN result[1];
    BOOLEAN result_32[32];
    'Instruction codes
    BOOLEAN bypass_instruction [6] = $3F;
    BOOLEAN idcode_instruction [6] = $09;
    'ID code
    BOOLEAN idcode [32] = $F1414093;
  ENDDATA;

  PROCEDURE bypass;
    'Load the BYPASS instruction
    IRSCAN 6, bypass_instruction[];
    'Scan "1", compare "0"
    DRSCAN 1, #1, CAPTURE result[], COMPARE #0, #1, test_fail;
    IF !test_fail THEN BEGIN;
      EXPORT "FAILURE", NAME, "TYPE", "BYPASS_TEST", result[];
      EXIT 17;
    ENDIF;
  ENDPROC; 'bypass

  PROCEDURE id_check;
    'Load the IDCODE instruction
    IRSCAN 6, idcode_instruction[];
    'Check the ID code
    DRSCAN 32, $00000000, CAPTURE result_32[], COMPARE idcode[], $0FFFFFFF, test_fail;
    IF !test_fail THEN BEGIN;
      EXPORT "FAILURE", NAME, "TYPE", "IDCODE_TEST", result_32[];
      EXIT 2;
    ENDIF;
  ENDPROC; 'id_check

  BODY; 'Executes when s200 is instantiated
  'Empty
  ENDBODY;
ENDCLASS;

```

Figure 6: Device1: s200.

## VII. CONCLUSIONS AND FUTURE WORK

Standards are important for effective testing. For printed circuit boards the IEEE 1149.1 standard developed by the JTAG (Joint Test Action Group) has been widely accepted. Integrated circuits and multi-boards have become increasingly complex and additional standards are required. At micro-level, the Internal Joint Test Access Group (IJTAG) is currently working on a standard for embedded on-chip instruments and at macro-level the System JTAG (SJTAG) is aiming at defining a standard for system-level test management; mainly connecting the IJTAG standard with the system test manager. The base for both IJTAG and SJTAG is the IEEE 1149.1. In this paper we have extended the Standard Test and Programming Language (STAPL) such that it can be used as an SJTAG engine; handling access between the test manager and embedded instruments. We have identified a number of required extensions that we have implemented in an extended version of STAPL. We have performed initial experiments where we simulated an embedded environment with a PC running the extended

STAPL player (interpreter) and an FPGA serving as device-under-test.

Future work includes to continue the refinement of requirements using a more versatile platform, and to demonstrate the proposed approach in a real embedded environment. We also aim at finding a protocol for the interaction between the Test Manager and the Test Controllers.

## VIII. ACKNOWLEDGMENT

We would like to thank Ann Chen at Altera for the technical support on the Jam STAPL player.

## REFERENCES

- [1] Altera Jam STAPL Software, <https://www.altera.com/support/software/download/programming/jam/jam-index.jsp>
- [2] IJTAG P1687, <http://grouper.ieee.org/groups/1687/>
- [3] B. Eklow and B. Bennetts, "New Techniques for Accessing Embedded Instrumentation: IEEE P1687 (IJTAG)", Proceedings of



```

CLASS 02s;
  DATA maindata;
    BOOLEAN test_fail;
    BOOLEAN result[1];
    BOOLEAN result_32[32];
    'Instruction codes
    BOOLEAN bypass_instruction [8] = $FF;
    BOOLEAN idcode_instruction [8] = $FE;
    'ID code
    BOOLEAN idcode [32] = $F5045093;
  ENDDATA;

  PROCEDURE bypass;
    'Load the BYPASS instruction
    IRSCAN 8, bypass_instruction[];
    'scan "1", compare "0"
    DRSCAN 1, #1, CAPTURE result[], COMPARE #0, #1, test_fail;
    IF !test_fail THEN BEGIN;
      EXPORT "FAILURE", NAME, "TYPE", "BYPASS_TEST", result[];
      EXIT 17;
    ENDIF;
  ENDPROC; 'bypass

  PROCEDURE id_check;
    'Load the IDCODE instruction
    IRSCAN 8, idcode_instruction[];
    'Check the ID code
    DRSCAN 32, $00000000, CAPTURE result_32[], COMPARE idcode[], $0FFFFFFF, test_fail;
    IF !test_fail THEN BEGIN;
      EXPORT "FAILURE", NAME, "TYPE", "IDCODE_TEST", result_32[];
      EXIT 2;
    ENDIF;
  ENDPROC; 'id_check

  BODY; 'Executes when 02s is instantiated
  'Empty
  ENDBODY;
ENDCLASS;

```

Figure 5: Device2: 02s.

European Test Symposium (ETS'06), pp. 253-254, Southampton, UK, May 2006.

- [4] SJTAG, <http://www.dft.co.uk/SJTAG/>
- [5] IEEE Std 1149.1-2001, "IEEE Standard Test Access Port and Boundary-Scan Architecture," IEEE, USA, 2001.
- [6] J Holmqvist, G. Carlsson, and E. Larsson, "Extended STAPL as SJTAG engine", Informal Digest of Papers at European Test Symposium (ETS), pp. 119-124, Freiburg, Germany, May 2007.
- [7] J. Rearick, B. Eklow, K. Posse, A. Crouch, B. Bennetts, "IJTAG (Internal JTAG): A Step Toward a DFT Standard", Proceedings of International Test Conference (ITC), paper 32.4, Austin, Texas, USA, November 2005.
- [8] K. Posse, A. Crouch, J. Rearick, B. Eklow, M. Laisne, B. Bennetts, J. Doege, M. Ricchetti, J.-F. Cote, "IEEE P1687: Toward Standardized Access of Embedded Instrumentation", Proceedings of IEEE International Test Conference, Santa Clara, CA, USA, October 2006, pages 1-8.