Software Aspects of BOLD
A System and User's Manual

Jung-Cheun Lien

CENG Technical Report: 91-26

Abstract

This manual describes the software aspects of BOLD in synthesizing test programs for a module. The issues addressed include: the test description languages; the design of the synthesizer; the generation of test programs using the synthesizer; the execution of test programs by a test controller; and a step-by-step demonstration to show the complete process of applying BOLD to a design.

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4469

July, 1991

Contents

1 Introduction

BOLD [10] is a design-for-test (DFT) tool that is applicable at the chip, module and system level. A system designed using BOLD is testable and maintainable at every level of the design hierarchy e.g., chip, module and system level. This is achieved in part by embedding test controllers and test buses into the system. For example, each chip contains an on-chip test controller (CMC) and each module contains a module test controller (MMC). These controllers are used to test and maintain a system throughout its life-cycle. Standard test busses are used to communicate between test controllers. For example, a CMC should contain a Test Access Port (TAP) in order to comply with the IEEE Std. 1149.1 boundary scan architecture [4], and an MMC should contain an IEEE 1149.1 bus master in order to communicate with CMCs. For implementation details of CMCs and an MMC, the reader is referred to [10].

1.1 Hierarchical Test Control

To test a circuit consisting of one or more kernels, both an MMC and an CMC are required. Through a boundary scan bus, which consists of four signal lines, namely TDI, TDO, TMS and TCK, the MMC controls the execution of the test of a chip having a CMC. The MMC also provides all necessary test data for testing the chip. During test mode the control signals of a kernel must be activated in the sequence specified by its associated control graph.

Figure 1 shows how the test control signals for a kernel can be generated and controlled by an MMC with the help of a CMC consisting of a TAP and a BIT-controller. The signals `Ci, Cj, Ck` are controlled by the state signals `Capture, Shift, Update, RunTest` corresponding to their associated TAP controller states, which are in turn controlled by the TMS line. The value of the TMS line is controlled by the state of the FSM1 in the test channel. The FSM1 can control the TAP controller to various states via the TMS line. When sending data to the kernel, a predefined sequence of state transitions is generated by the FSM1 to activate the TAP states such that data can be received. Several predefined sequence of state transitions have been built into FSM1. The processor in the MMC can select a sequence by loading proper data into the internal registers of the test channel. The loading of a register in the test channel is achieved by controlling the processor interfacing signals `/CS, /WR, /RD, PA, PD`. These signals can be controlled by the processor executing a piece of code that contains I/O instructions.

By executing a program stored in the memory unit of an MMC, the processor can control the test of a kernel in the application circuit. The program directs the processor to control the test channel by loading its internal registers. The FSM1 is then activated and produces the proper sequence of values on the TMS line, which drives the TAP controller to appropriate states. This activates the state signals and, with the help of the BIT controller, the control signals `Ci, Cj, Ck` are activated according to the control graph. The kernel is thus tested.

Writing a program to test a kernel is a difficult problem since it involves a high degree of

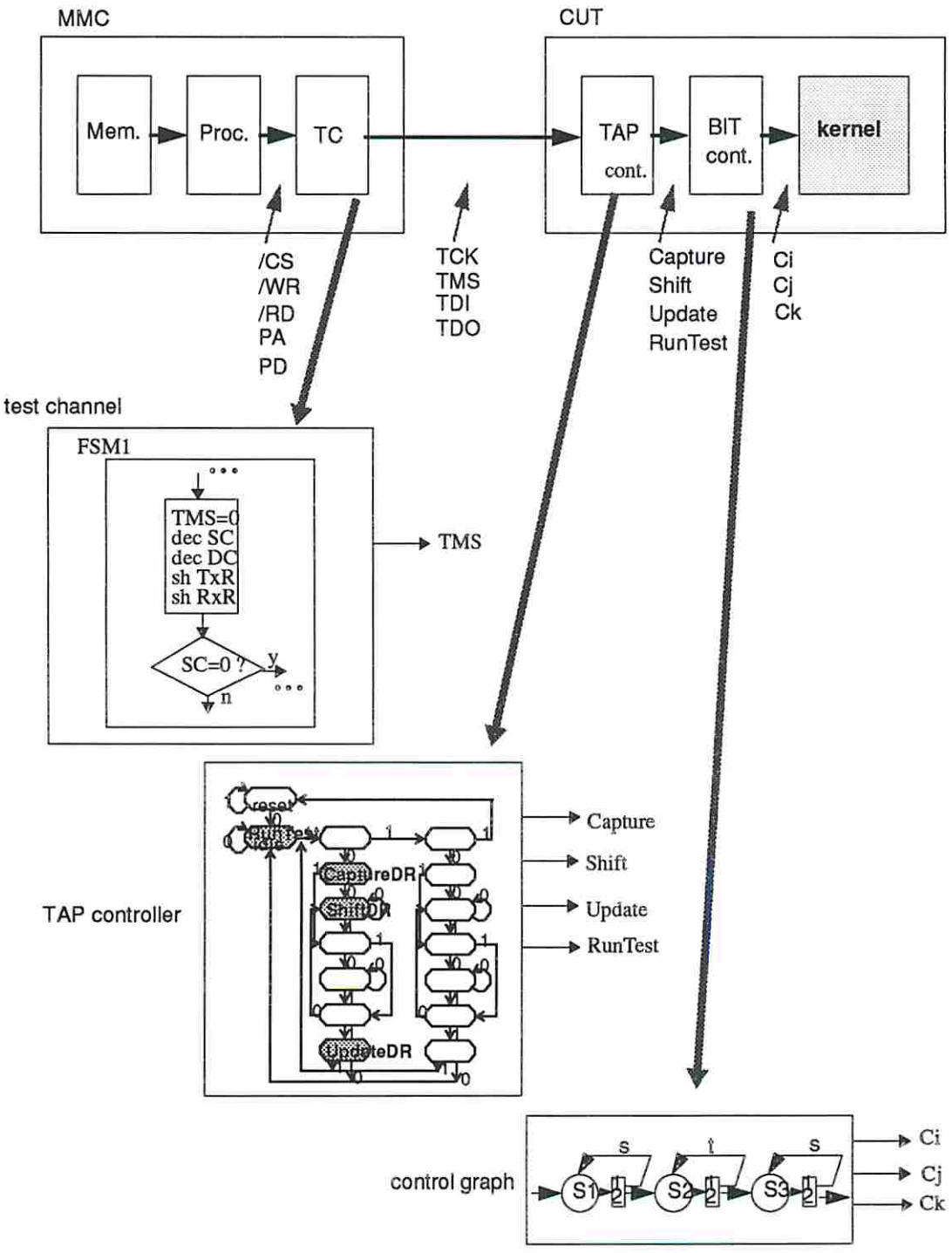


Figure 1: Overview of the test control.

complexity resulting from many details at various level. To solve this problem a test program synthesis technique is used in BOLD. Test description languages are provided for describing test procedures at both chip and module levels. Software tools are also provided so that test programs can be automatically synthesized from the test procedures. The synthesized test programs are compiled and loaded into the memory unit of the MMC. The MMC can then test the kernel by executing the test programs.

1.2 Test Hardware and Software Hierarchy

The relationship between the test hardware and software in an HTM system is shown in Figure 2, where four axes are used to represent the hardware assembly units, the test description languages, the test programs, and the test controllers, respectively. Each axis again is represented by a hierarchy of four levels. The top hardware assembly unit is a system, which consists of several subsystems, which again consists of several modules, which again contains many chips. Each hardware assembly unit has a test controller associated with it. These test controllers include the system maintenance processor (SMP), the subsystem maintenance processor (SuMP), the module test and maintenance controller (MMC), and the chip test and maintenance controller (CMC). The test programs are classified into four levels according to their applications to the hardware units. These are the system test program (STP), the subsystem test program (SuTP), the module test program (MTP), and the chip test program (CTP). The test languages used to describe the test aspects of a hardware unit includes the system test language (STL), the subsystem test language (SuTL), the module test language (MTL) and the chip test language (CTL).

The synthesis process starts with the preparation of the test description files by a designer. These files are constructed using high level description languages that are easily understood by designers with little or no knowledge of testability. A set of synthesizers are then used to translate the input files into appropriate formats (which is in C language) for each hardware unit. These test programs are then translated down to executable code for the test controllers by an appropriate C compiler. A test controller can then test its associated hardware unit by executing the loaded executable codes. In such a manner, the entire system can be tested.

The major advantages of the synthesis approach are (1) consistent test methodology, that is, a chip is tested using the same test set during the chip test, module test, subsystem test and system test; (2) reduced time, effort, and errors in test program development; (3) test programs can be prepared by designers with little knowledge of testability; (4) interconnect testing is included automatically.

1.3 Test Control Representations in BOLD

Figure 3 shows how a circuit gets tested in the BOLD system. The circuit is first made testable by using some testable design methodologies (TDMs) [1]. A test plan is used to described how this circuit can be tested. The test plan of a testable circuit is represented by

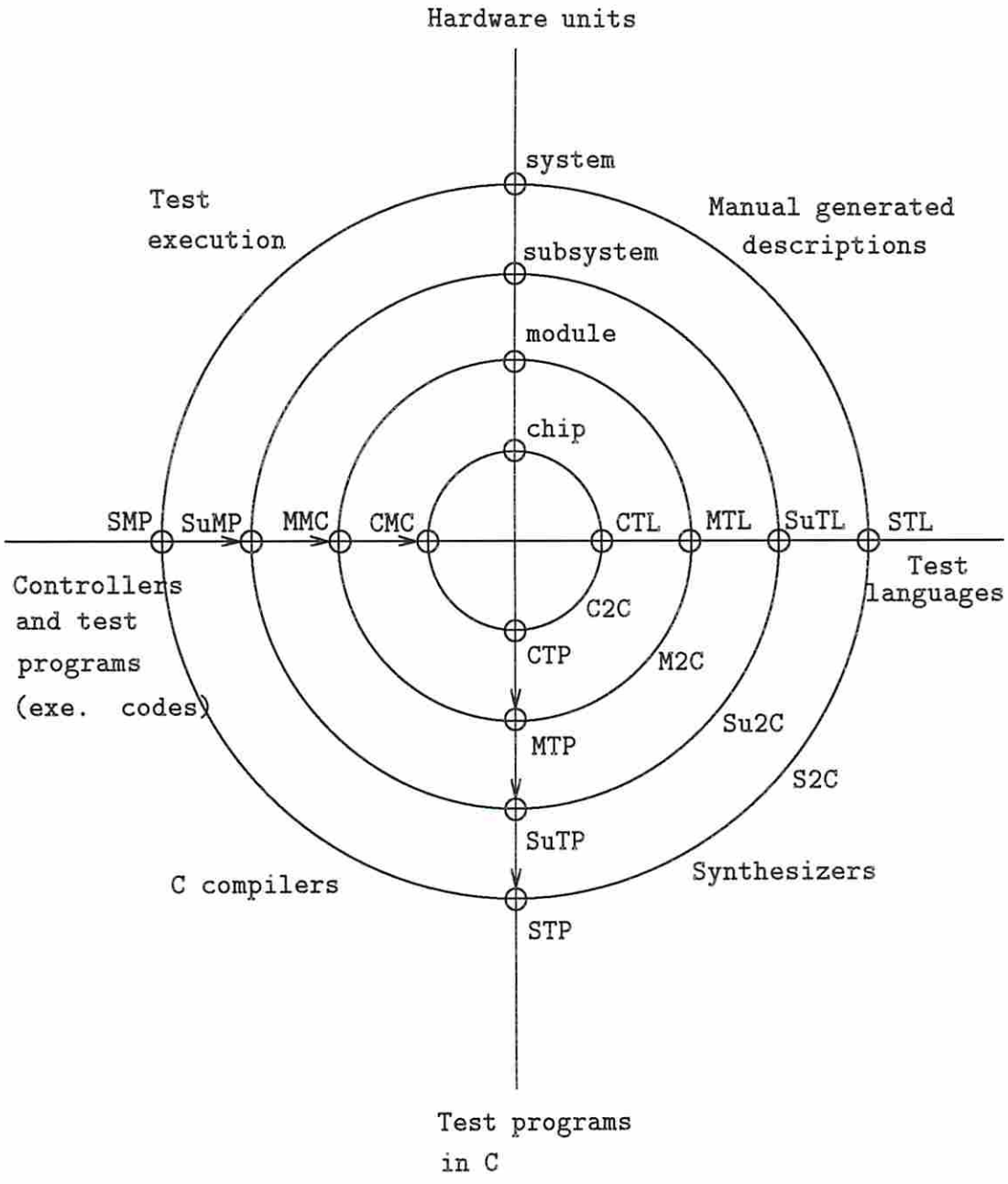


Figure 2: Overview of the test hardware/software hierarchy.

a set of test control signals and its associated control graph.

The test activities that occur at various levels are of different complexity. For example, on the test bus activities are represented by a sequence of binary values on the signal lines. For the test channel activities are represented by a sequence of processor write and read operations to/from the internal registers of the test channel. It is clear that the lower level the representation, the high level of complexity involved in describing the test process of a circuit. To reduce the test program development cost, it is desirable to reduce such complexity. In BOLD, the test activities are represented at a very high level, e.g., a statement **test(chip1)** is used to test a chip called chip1. Minimal complexity is involved in describing the test process of a circuit.

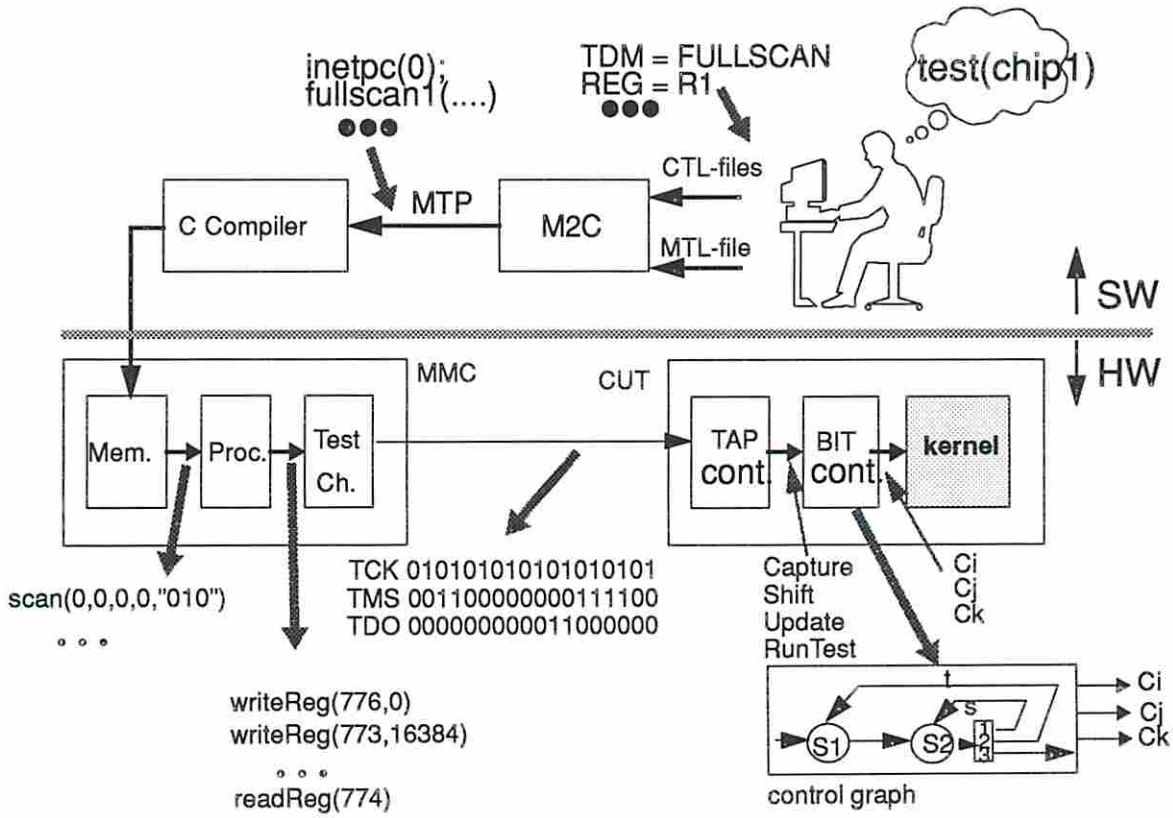


Figure 3: Testing a circuit using BOLD.

Two types of descriptions are needed in testing a circuit, namely one or more CTL-files and an MTL-file. The former is a description of the test aspects of a chip in the chip test language (CTL). The latter is a description of the test aspects of a module in the module test language (MTL). Both CTL and MTL are high level languages and can be understood by a designer with little or no knowledge in testing.

1.4 Test Program Synthesis and Execution Procedure

BOLD provides a synthesizer (called M2C) that can take as input both CTL and MTL files and generates a module test program in the C language that, when properly compiled, can be executed by the MMC to test the entire module, including all chips on the module and the interconnect among them. Figure 4 shows the process flow of synthesizing test programs for a module. Note that a file describing the net information is also generated and sent to the MMC so that the interconnect can be properly tested.

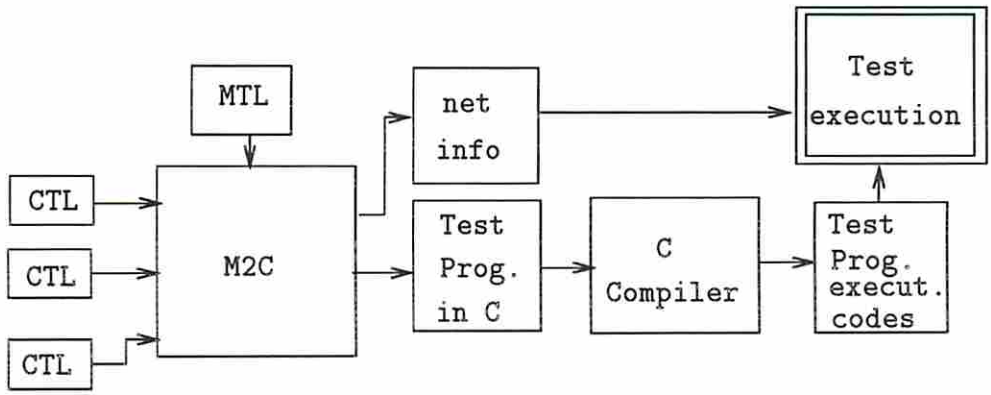


Figure 4: Generating test programs for a module.

1.5 Organization of the Manual

This manual is organized as follows. The preparation of test descriptions are described in section 2. The design of the synthesizer M2C is described in section 3. Problems that might occur while running M2C to synthesize test programs are described in section 4. The compilation and execution of test programs by the MMC is described in section 5. A consistency check for the test controller hardware is described in section 6. Appendix A is a paper describing the details of the BSDL. Appendix B includes a step-by-step demonstration of the BOLD system.

2 Test Description Languages

2.1 CTL

The CTL is based on BSDL [9]. A CTL description (CTL-file) consists two parts, namely the BSDL description and the chip test procedure. A detailed description of the BSDL is included in appendix A of this manual. Here only the test procedure part is described. The test procedure can be incorporated into a BSDL file. The incorporation of the test procedure

is achieved by adding a VHDL attribute called TEST_PROC. The following example shows how a test procedure is incorporated in a CTL-file.

```
attribute TEST_PROC of app1 : entity is
    "Test_Begin" &
    "TDM 1 = FULLSCAN;" &
    "REG=FBR, VECFILE=ap1_in2, RESFILE=ap1_out2;" &
    "REG=BOUNDARY, VECFILE=ap1_in1, RESFILE=ap1_out1;" &
    "CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;" &
    "Test_End ";
```

It is also possible to omit this attribute and describe the test procedure in a separate file, where the quotes and & are omitted. For example, the above test procedure can be written in a file called `app1.ctp` as follows.

```
TEST_BEGIN
TDM 1 = FULLSCAN;
REG=FBR, VECFILE=AP1_IN2, RESFILE=AP1_OUT2;
REG=BOUNDARY, VECFILE=AP1_IN1, RESFILE=AP1_OUT1;
CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;
TEST_END
```

A test procedure consists of one or more test sessions. During a test session different parts of the chip are tested according to a predefined methodology, which is called a testable design methodology (TDM). One can associate the process of testing a kernel using a TDM with a test procedure for that TDM. The procedure can be described as either template-based or user-defined. A template-based TDM procedure is used to describe a procedure to test a circuit designed with a commonly used TDM. Currently BOLD supports templates include Fullscan, FullscanN, BILBO, RUNBIST, and INTEST. A user-defined TDM procedure, on the other hand, is used to describe an arbitrary procedure composed by the user using C code or some test-specific statements provided by CTL.

The test-specific statements that can be used in a user-defined TDM procedure is listed below.

- *ScanIR(outS);*
  The content of the instruction register (IR) of the TAP can be updated by executing this function. The string *outS* is loaded into the IR after this function is executed. When scanning in the new instruction, a string of values called status is also scanned out. The status is the logic value on the parallel data input to the instruction register before the shifting started. By executing this function, a test controller can control the test process of a chip. The format of the instruction is determined by the chip designer except for those that have been predefined by the IEEE Std. 1149.1. The format of the status is also defined by the chip designer.

- *ScanDR(outS);*
  This function is similar to the $ScanIR$ except that the selected data register is scanned. The selection of the data register is determined by the current content of the instruction register, which can be altered by the $ScanIR$ function. When scanning a new string of data, the resulting string from the selected data register is also scanned out. The result is the logic value on the parallel data input to the data register before the shifting started.

- *ApplyClock(tck, n);*
  This function can be used to provide test clocks to the circuit under test. Both the test clock `TCK` and the functional clock `FCK` can be applied using this function. For example, if the value of *tck* is 1 (or true), then the test clock `TCK` is applied for n cycles while the rest of the I/O pins are kept unchanged. If the value of *tck* is 0 (or false), then the functional clock `FCK`, which may consist of several phases, is applied for n cycles while keeping the input to the rest of the I/O pins unchanged.

- *Bring2state(i);*
  The execution of this function drives the bus to state i, defined in the IEEE Std. 1149.1, regardless of the current state of the bus. It is easy to conclude, from the state transition diagram, that putting a 1 on the `TMS` line for five consecutive clock cycles will bring the TAP controller into the `Reset` state. The TAP controller can then be brought to any state i from the `Reset` state. If more than one possible state transition paths exist, the one that goes through the smallest number of states will be taken.

- *State2state(i,j);*
  The execution of this instruction drives the bus from state i to state j. Again, the path taken is through the smallest number of states.

- *RepeatState(i,n);*
  The execution of this instruction enables the bus to stay in state i for n consecutive clock cycles. Note that this instruction can be applied to only some of the states. For the $shiftDR$ and $shiftIR$ states, this instruction can be modified to $RepeatState(i, n, Sout, Sin)$, where $Sout$ is the string sent to the `TDO` line and Sin is the string received from the `TDI` line.

- *RunTest(n);*
  The bus is driven into the `RunTest_Idle` state and held there for n consecutive test clock cycles. During this period, the on-chip test controller executes the predefined test process for the built-in self-test structures.

2.1.1 TDM Examples

An example of the Fullscan TDM template described in CTL is shown below.

```
TDM <tdm_id> = Fullscan;
REG = <reg1>, VECFILE= <file1>, RESFILE= <file2>;
CLOCK= FCK <number1> CYCLES_IN RUN_TEST_IDLE;
```

An example of the FullscanN TDM template in CTL is shown below.

```
TDM <tdm_id> = FullscanN;
REG = <reg1>, VECFILE = <file1>, RESFILE = <file2>;
REG = <reg2>, VECFILE = <file3>, RESFILE = <file4>;
CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;
```

An example of the BILBO TDM template in CTL is shown below.

```
TDM <tdm_id> = BILBO;
INITIALIZE <reg1> = <value1>, <reg2> = <value2>;
USE_INSTRUCTION = <ins1>;
CLOCK = <TCK or FCK> <number1> CYCLES_IN RUN_TEST_IDLE;
EXPECTED_RESULT <reg3> = <value3>, <reg4> = <value4>;
```

An example of the RUNBIST TDM template is shown below.

```
TDM <tdm_id> = RUNBIST;
CLOCK = TCK <number> CYCLES_IN RUN_TEST_IDLE;
EXPECTED_RESULT <reg1> = <value1>;
```

An example of the INTEST TDM is shown below.

```
TDM <tdm_id> = INTEST;
VECFILE = <file1>, RESFILE = <file2>;
CLOCK = FCK <number1> CYCLES_IN RUN_TEST_IDLE;
```

2.1.2 Formal Description of the CTL

The CTL is a superset of the BSDL [9]. The formal definition of the BSDL can be found in Appendix A. The definition for the chip test procedure is listed below using the YACC [5] input format.

```
%%
ctl: BSDL chip_test_procedure;
chip_test_procedure: _TEST_BEGIN test_procs _TEST_END;
test_procs: test_proc | test_procs test_proc;
test_proc
        : _TDM _INT_NUM _EQ _RUNBIST _SEMICOLON runbist_tdm
```

```
        | _TDM _INT_NUM _EQ _INTEST _SEMICOLON intest_tdm
        | _TDM _INT_NUM _EQ _FULLSCAN _SEMICOLON fullscan_tdm
        | _TDM _INT_NUM _EQ _BILBO _SEMICOLON bilbo_tdm
        | _TDM _INT_NUM _EQ _USER_DEFINE _SEMICOLON user_def_proc;
runbist_tdm: clock_part result_part;
clock_part
        : _CLOCK _EQ _TCK nums _CYCLES_IN _RUN_TEST_IDLE _SEMICOLON
        | _CLOCK _EQ _FCK nums _CYCLES_IN _RUN_TEST_IDLE _SEMICOLON
        | _CLOCK _EQ _FCK _SHIFTED
        | _CLOCK _EQ _NONE;
nums: _FLO_NUM | _INT_NUM;
result_part: _EXPECTED_RESULT res_lists  _SEMICOLON;
res_lists: res_list | res_lists _COMMA res_list;
res_list: _IDENTIFIER _EQ _BIN_NUM;
intest_tdm: _VECFILE _EQ _IDENTIFIER _COMMA _RESFILE _EQ
            _IDENTIFIER _SEMICOLON clock_part;
fullscan_tdm: reg_Flists clock_part;
reg_Flists: reg_Flist| reg_Flists reg_Flist;
reg_Flist: _REG _EQ _IDENTIFIER _COMMA _VECFILE _EQ _IDENTIFIER
           _COMMA _RESFILE _EQ _IDENTIFIER _SEMICOLON;
bilbo_tdm: initialize_part use_ins_part clock_part result_part;
initialize_part: _INITIALIZE ini_lists _SEMICOLON;
ini_lists: ini_list | ini_lists _COMMA ini_list;
ini_list: _IDENTIFIER _EQ _BIN_NUM;
use_ins_part: _USE_INSTRUCTION _EQ _IDENTIFIER _SEMICOLON;
user_def_proc: _TOP;
%%
```

Note that the user-defined TDM procedure is not described since the statements in this procedure are directly translated by LEX [6]. In addition, the syntax of C is not listed. Interested readers are referred to the source code of the program.

2.2 MTL

The test aspects of a module described by the MTL include the following parts, namely *library_id, device_list, test bus configuration, net_list*, and *test procedure*. The *library_id* points to the directory containing the CTL-files. The *device_list* associates every device used in the module with a CTL-file in the library. The device_list of an example module consisting of two devices is described as follows.

```
device_list =
(Chip1 adder)(Chip2 multiplier);
```

The *test bus configuration* describes how the devices on the module are connected via the boundary scan bus to the test channel. In the boundary scan architecture, the test bus can be configured as a ring, a star or a combination of both. In MTL a test bus is modeled as a multiple ring, which can be mapped into any one of the above three configurations. A ring configuration is formed when only a single ring is used. A star configuration is formed when every ring contains only one device. All chips under test in a ring are controlled by the same `TMS` line.

An example test bus description is as follows.

```
test bus =
ring 0:  chip1 => chip2 => chip3,
ring 1:  chip4 => chip5;
```

The *net_list* describes how the devices on a module are physically interconnected. The number of nets can be large. Two or more terminals are possible for each net. Each terminal is specified by two names, the first name gives the device name, while the second specifies the I/O port name or the pin number. A net_list consisting of two nets is described below.

```
net_list =
net 1:  (Chip1 inp1) (Chip2 outp1),
net 2:  (Chip2 inp1) (Chip3 inp1) (Chip1 outp1);
```

The *test procedure* contains the information for testing a module by an MMC. This information is represented in terms of standard C code and some test-specific statements. These statements assume no knowledge about the test controller and can be translated to low level functions according to the architectural detail of the MMC. The low level functions are fully supported by a library of C code containing machine-dependent I/O functions. These I/O functions are used to control the test channel, which is responsible for all the low level activities on the boundary scan test bus.

The test-specific statements that can be used to describe a test procedure in an MTL-file are listed below.

- Testchip (chip_id);
  A test controller, such as an MMC, can fully test a chip by executing this statement. If more than one session is required for testing the chip, the test results are reported only after all sessions are completed.

- Testchip (chip_id) Use TDM (tdm_id);
  A test controller can test part of a chip by executing this statement. Usually, a chip may be tested in more than one session. During each session part of the chip is tested using a specific TDM. To fully test the chip, all test sessions must be executed. When a module under test cannot stay offline long enough to allow it to be fully tested, a

partial testing approach is used. Also a piece of circuit can be tested several times using different test patterns or different seeds. In this approach, the chip is tested in different intervals. One or more test sessions are executed in each interval without exceeding the time limit.

- TestInet();
  A test controller can test the interconnect on a module by executing this statement. Every net connecting two boundary scan chips is tested. The test set used in this test is a counting sequence which can determine if the entire interconnect is fault free. However, only the Go/NoGo information is produced in this test. No diagnostic information is provided.

- DiagnosisInet();
  A test controller can test and diagnose the interconnect on a module by executing this statement. To achieve the maximal diagnostic resolution, the test set is a universal test set, which includes a walking ones sequence, a walking zeroes sequence, and the all-0 and all-1 vectors. All diagnosable faults can be identified by this test.

- char *scan(ringid, type, pre, post, outS);
  By executing this statement, a test controller can exchange information with a chip a selected ring specified by the *ringid*. The information sent out is the string *outS*, which can be either instruction (type=1) or data (type=0). This function returns a string that is the incoming information during the shifting.

- sample_ring(ringid);
  By executing this statement, a test controller can achieve a snap-shot of the logic values on the I/O pins of all chips that are connected to the ring specified by "ringid". The returned value of this function is a string of 1s and 0s representing the current status of these chips.

- reset_ring(ring_id);
  The test logic of all chips on the test ring specified by "ringid" are reseted.

- runtest_ring(ringid, len);
  ALl chips on the test ring specified by "ringid" enter the RunTest-Idle state for "len" clock cycles.

- ApplyClock(ringid, n);
  By executing this statement, a test controller can apply the test clock to the DUTs in a selected scan ring for a fixed number of cycles. The bus state of the selected ring is kept in the *RunTest* state so that the last instruction can be executed. For example, when this statement is executed after the public instruction *RUNBIST* has been sent, the chips can be tested.

2.2.1 Formal Definition of the MTL Syntax

The formal definition of the module test language MTL is listed below. The language is described in the input format of YACC.

```
%%
mtl: conf_section test_section;
conf_section : mtl_stmts;
mtl_stmts: mtl_stmt | mtl_stmts mtl_stmt;
mtl_stmt: module | lib | device_list | test_bus | net_list;
module: _MODULE _EQ _IDENTIFIER _SEMICOLON;
lib: _LIB _EQ _IDENTIFIER _LPR _INT_NUM  _RPR _SEMICOLON;
device_list: _DEVICE_LIST _EQ dev_pairs _SEMICOLON;
dev_pairs: dev_pair | dev_pairs dev_pair;
dev_pair: _LPR _IDENTIFIER chip_type _RPR;
chip_type: _IDENTIFIER;
test_bus: _TEST_BUS _EQ test_rings _SEMICOLON;
test_rings: test_ring| test_rings _COMMA test_ring;
test_ring: _RING ring_id _COLON device_chain;
ring_id: _INT_NUM;
device_chain: _IDENTIFIER| device_chain _RARROW _IDENTIFIER;
net_list: _NET_LIST _EQ nets _SEMICOLON;
nets: net| nets _COMMA net;
net: _NET net_id _COLON pins;
net_id: _INT_NUM;
pins: pin| pins pin;
pin: _LPR _IDENTIFIER _IDENTIFIER _RPR;
test_section: _TEST_BEGIN mtp_stmts _TEST_END
mtp_stmts: mtp_stmt | mtp_stmts mtp_stmt;
mtp_stmt
        : test_inet
        | diagnosis_inet
        | test_chip
        | reset_ring
        | runtest_ring
        | sample_ring
        | apply_clock
        | ch_clock_freq;
test_inet: _TESTINET _LPR _RPR _SEMICOLON;
diagnosis_inet: _DIAGNOSISINET _LPR _RPR _SEMICOLON;
test_chip: _TESTCHIP _LPR _IDENTIFIER _RPR _SEMICOLON
 | _TESTCHIP _LPR _IDENTIFIER _RPR _USE
          _TDM _INT_NUM _SEMICOLON;
runtest_ring: _RUNTESTRING _LPR int_num _COMMA int_num _RPR _SEMICOLON;
```

```
reset_ring: _RESETRING _LPR int_num _RPR _SEMICOLON;
sample_ring: _SAMPLERING _LPR int_num _RPR _SEMICOLON;
int_num: _INT_NUM;
apply_clock: _APPLY_CLOCK clock_type int_num _CYCLES _SEMICOLON;
ch_clock_freq: _CHANGE_CLOCK_FREQ _LPR int_num _RPR _SEMICOLON;
clock_type: _TCK | _FCK;
%%
```

Note that the mtp-stmt can also be described in the C language. The formal definition of this part is not listed.

3 Design of the Test Program Synthesizer M2C

The structure of the M2C is shown in Figure 5. The M2C consists of a set of language parsers, a user-defined TDM module, a template-based module, a shift-adjustment module, an interconnect test module, a device driver module, a target manufacturing module, and a global data base. The data base, which consists of three files, namely ctlInfo.h, tdmInfo.h and mtlInfo.h, contains all information required to generate a test program. These three files are used by many functions of M2C.

Figure 6 shows the dependency tree of all the functions that are used in the synthesizer M2C. Each function is represented by two names, i.e., the name of the function and the name of the file that contains this function. Note that some functions are contained in more than one file, for example, the parser mtlInfo is contained in both mtlInfo.y, which is a YACC [5] input file and mtlInfo.l, which is a LEX [6] input file.

3.1 Language Parsers

There are four parsers provided by the synthesizer M2C, namely ctlInfo, tdmInfo, mtlInfo and m2cp. The ctlInfo parses a CTL-file and stores the extracted information into global data structures in the file ctlInfo.h.

The tdmInfo parses the test procedure of a chip and store the extracted TDM information into global data structures in the file tdmInfo.h.

The mtlInfo parses a MTL-file and stores all the extracted information into global data structures in the file mtlInfo.h.

The m2cp parses a MTL-file and translates its test procedure into the module test program in a line-by-line fashion. Each statement in the test procedure of the MTL-file will be translated into a proper function call in the module test program.

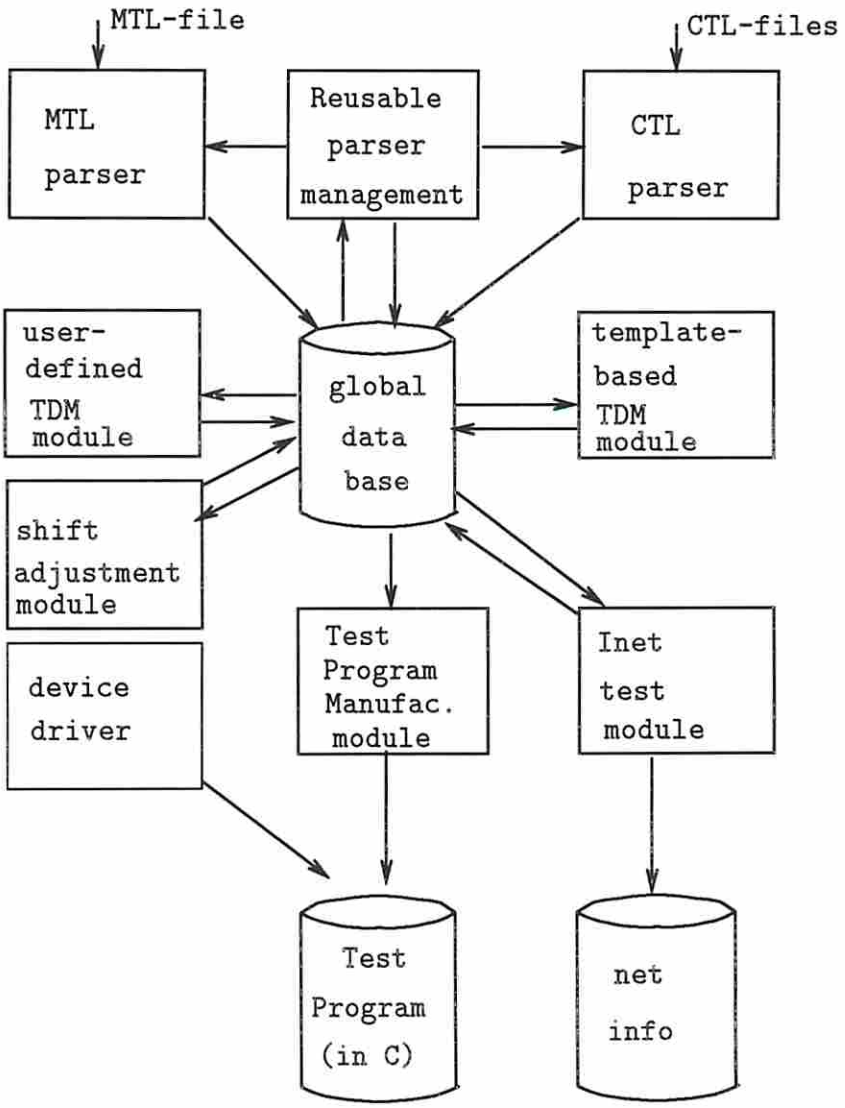


Figure 5: The structure of the M2C.

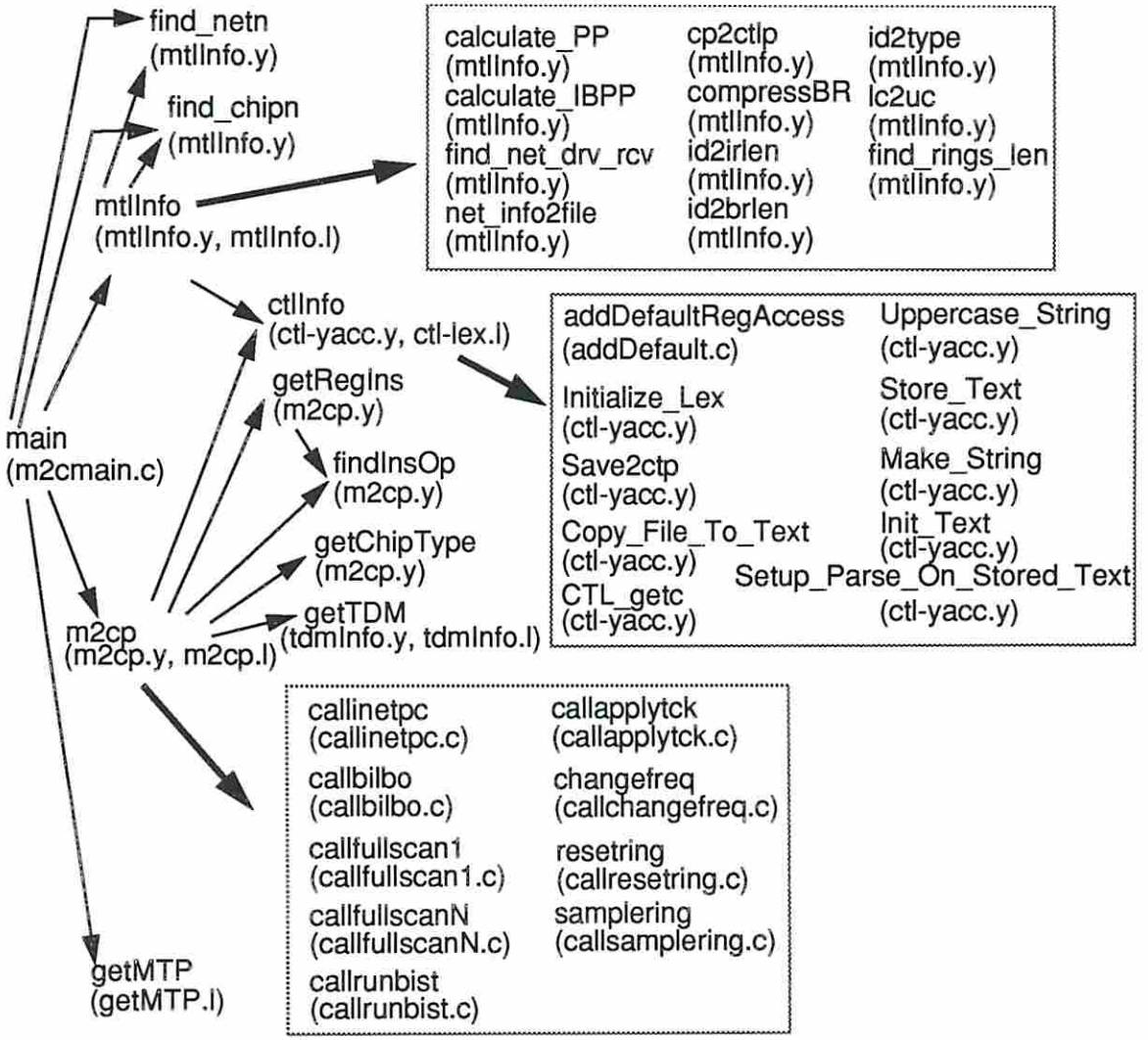


Figure 6: The function dependency tree of M2C.

3.2 Template-based and User-defined TDM Modules

The template-based TDM module consists of a set of functions that can be included in a test program to execute a test procedure using template-based TDM. The user-defined TDM module is used when the test procedure is descirbed by a user using C code and the functions that are provoided by the device driver.

3.3 Interconnect Test Module

This module can extract the interconnect information from the CTL and MTL files and stored the needed information in a file called infofile.net. This information can then be used by a C function such that the execution of this function will test the interconenct. The test set used can be either a counting sequence or a universal test set. The counting sequence can determine whether the entire interconenct is fault free or not. On the other hand, the universal test set can determine the existence of every diagnosable fault.

3.4 Test Program Manufacturing Module

A utility called **genTarget** is included to simplify the actual generation of test programs. All files of a test program is copied into a directory so that they can be correctly send to an IBM PC using a communication utility such as **kermit**.

3.5 Device Driver

The device driver consists of a set of C functions that can control the test controller hardware to perform specific routine functions. All other functions in the M2C use this set of functions to control the test channel. This prevent the M2C from having to deal with the implementation details of the test controller. Any hardware that supports these functions can be a test controller and its test program can be synthesized by M2C.

The device driver for test controllers having a test channel includes the following functions:

1. char *scan(ringid, type, pre, post, outs);
   Set type to 1 to send instruction (outs) to a chip on a test ring specified by "ringid". The pre and post specify the location of the chip on the test ring.

2. reset_ring(ringid, len);
   Put all chips on a test ring specified by "ringid" into Reset state for "len" TCK clock cycles.

3. char *sample_ring(ringid, irslen, brslen);
   Sample the primary I/Os' value of chips on a test ring specified by "ringid". The sampled value is returned.

4. runtest_ring(ringid, len);
   Put all chips in a test ring specified by "ringid" into RunTest state for "len" TCK clock cycles.

5. writeReg(portid, outword);
   Write outword to the test channel's internal register addressed by portid.

6. readReg(portid);
   Read the test channel's internal register address by portid.

3.6 MTL to Test Program Translation

The M2C does a syntax directed translation from a module test procedure in the MTL file to a module test program. Figure 7 shows how the translation is done for a particular example. Both testinet() and diagnosisinet() are translated into the same function call inetpc() in the test program (see line 1 and 2). However, they have different arguments. The user-defined test procedure kernel1() is copied directly into the test program (see line 3). The statement testchip(U3) is translated into a six line code that has a fullscanN() function call (see line 4). Here U3 a chip that is tested using a fullscanN TDM. U4 is also tested using fullscanN() and here the statement testchip(U4) is translated into similar code as for U3 but with different arguments.

3.7 Test Program Hierarchy and its Hardware Dependency

A generated test program consists of many functions that are well structured. Figure 8 shows the hierarchy of these functions for a typical test program. The hierarchy contains four layers, namely Layer 0, Layer 1, Layer 2 and Layer 3.

Layer 0 contains the lowest level functions that can directly access the test control hardware. For example, the readReg and writeReg can be used to access the internal registers of the test channel.

Layer 1 contains a set of functions that are related to the test channel architecture. For example, the function scan() can be used to send a string of data to a data register of a chip.

Layer 2 contains a set of functions that implements the predefined template-based TDM procedure. For example, the function fullscan1 can be used to test a circuit with a single chain full scan TDM. Similarly, the function runbist can be used to test a circuit with a runbist TDM. Also included in this layer are the user-defined test procedures. These procedures are written in the C programming language. For example, kernel1 is a user-defined test procedure.

Layer 3 contains the function main() of the test program. The main function consists of a series of calls to Layer 2 functions.

Note that most of these functions can be used for different modules under test. The

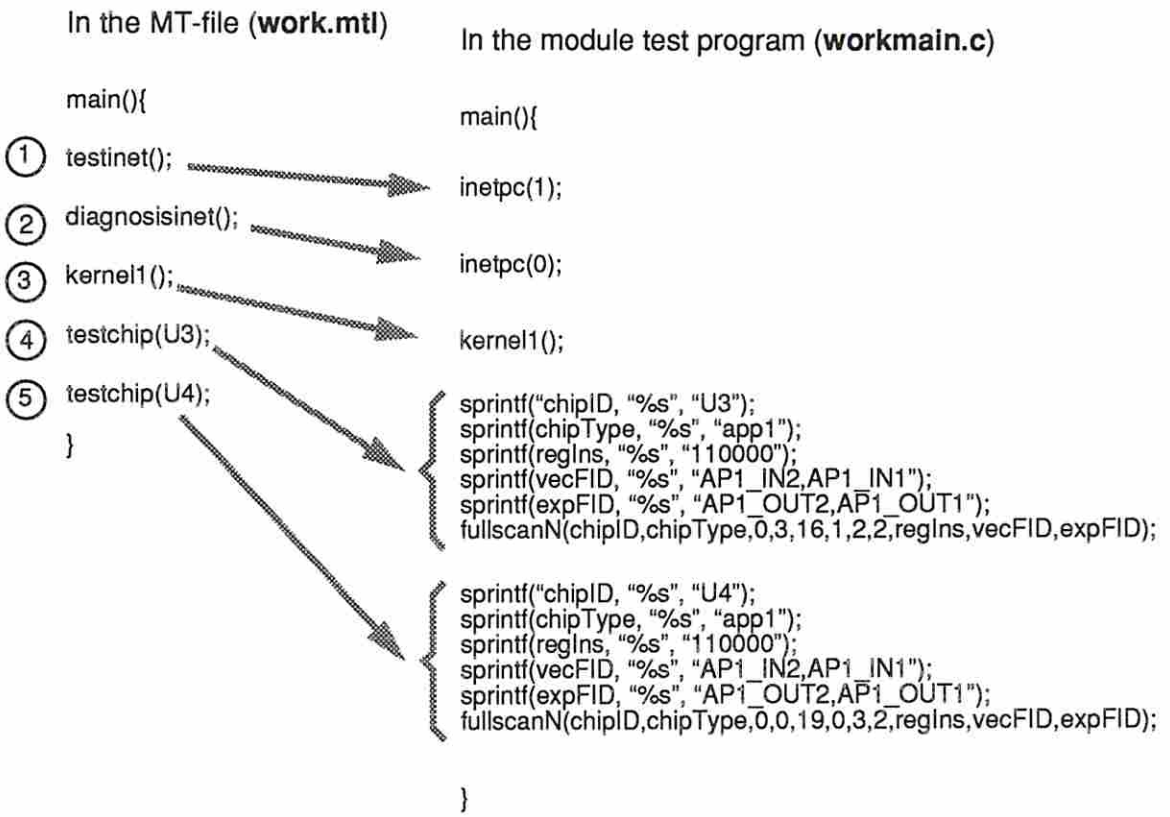
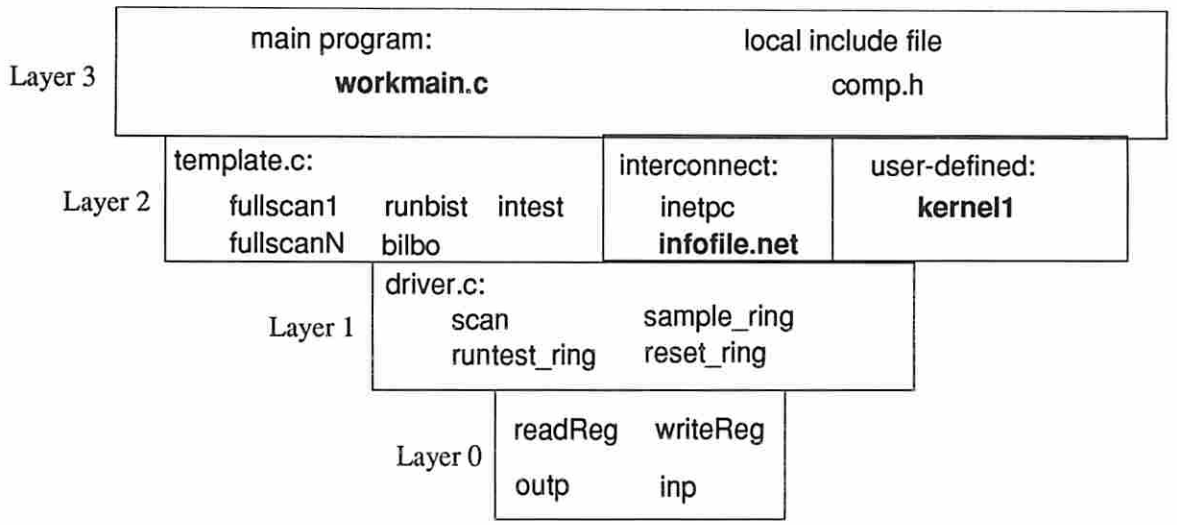


Figure 7: Syntax Directed Translation in M2C.

main components of a test program is shown in Figure 8. The parts that are specific to a test program are workmain.c, infofile.net and kernel1.



Software/Hardware Dependency Matrix

	MMC processor	Test channel	Test channel interface	C compiler
Layer 3	No	No	No	No
Layer 2	No	No	No	No
Layer 1	No	Yes	No	Yes
Layer 0	Yes	Yes	Yes	Yes

Figure 8: Software Dependency of the generated module test program.

The dependency of these functions and the test hardware are also listed in Figure 8. Note that the functions in Layer 2 and 3 are independent of the implementation of the test controller. The functions in Layer 0, such as writeReg, depend not only on the test controller but also the C compiler used.

3.8 Extension and Modification of the M2C

When using a different test controller, part of the M2C must be changed.

3.8.1 Adding a New Template-Based TDM

The addition of a new TDM to the M2C can be carried out as follows.

1. Change the parser tdmInfo. This should be done in both the tdmInfo.l and tdmInfo.y files. New tokens can be recognized by changing the tdmInfo.l file and new grammar rules can be added to the tdmInfo.y file.

2. Add new data structures to the file tdmInfo.h if necessary.

3. Change the parser m2cp. This should be done in both the files m2cp.l and m2cp.y. The modification is easy if the implementation of existing TDMs is consulted.

4. Add a new meta-procedure for generating the function calls for the new TDM. The function name is of the form callxxx, where xxx is the name of the new TDM (see Figure 6). Again, consulting the meta-procedures of the existing TDMs will be helpful.

3.8.2 Interfacing the Test Channel at Different I/O Addresses

Most functions of the M2C can still be used. Only two C functions need to be modified, namely writeReg and readReg. The new I/O address must be correctly incorporated into these two functions.

3.8.3 Using a Different Bus Master

If a bus master other than the test channel is used in the test controller, the device driver, i.e., functions in Layer 0 and 1, need to be rewritten. A similar set of functions provided by the device driver must be provided in order to make use of M2C.

3.8.4 Using a Different MMC

Again, all functions in Layer 0 and 1 must be rewritten. Functions in Layer 2 and 3 remain unchanged.

3.8.5 Recompiling the M2C

A recompilation of M2C is necessary if any functions used by M2C has been changed. The compilation of M2C is helped by a make file called M2C.mak. A new M2C can be generated by typing **make -f M2C.mak** at a SUN workstation.

4 Running M2C

Before actually synthesizing test programs for a module, a user should prepare the MTL description (e.g., in a file called work.mtl, where work is a user-specified name) and the CTL descriptions. Each type of chip (e.g., chip1) should have a CTL description in a file called chip1.ctl.

4.1 Preparing CTL and MTL files

A MTL-file consists of two parts, namely the conf_section and the test_section (see section 2.2). The first part describes the configuration of the module under test in a very straightforward manner. Thus writing the first part of a MTL-file is easy. The second part describe how the module is tested. The statements that can be used in this part include the C code and the test specific statements. Also, the functions that are provided by the device driver can also be used.

A CTL-file also consists of two parts, namely the BSDL part and the test procedure part. Writing the first part requires some knowledge on BSDL. The reader is referred to [9] for a detailed description. The second part consists of one or more TDM procedures, which in turn consists of template-based TDM procedures or user-defined TDM procedures. Writing a template-based procedure is simple since the templates are available. On the other hand, it is harder to write a user-defined TDM procedure. The test specific statements listed in section 2.1 and the full set of C language can be used.

4.2 Preparing Test Programs for the Test Controller

Once CTL and MTL files are prepared one can synthesize the module test programs by typing **M2C work**, where work.mtl is the MTL description of the module under test.

1. At a SUN workstation, type **M2C work** to synthesize test programs for the module called **work**. Then, type **genTarget work** to actually manufacture the test programs. The resulting programs (in C) are put in a directory called **workDir**.
2. Transfer all files in the directory to the IBM PC using the communication program called **kermit**. In addition, all test vectors and results files, if any, should also be send to the PC.
3. Generate executable test programs in PC by typing **make work.mak**. Note that the make file work.mak was generated automatically at the SUN workstation.
4. Execute the test program by typing **work**.

Note that the last three steps are performed at the PC, which is the major part of the MMC prototype.

4.3 Error Messages

Neither LEX nor YACC provide a good error reporting mechanism. To help a user identify the error in the input files, a reporting mechanism has been employed in the language parsers of M2C.

4.3.1 Errors in the ctlInfo

The parser ctlInfo can identify a syntax error to the line number of the source input. For example, the following error message will appear if there is an error in a CTL-file.

Error, Line 19, Syntax Error
CTL file ctype1.ctl contains errors, stopping

The message says that there is a syntax error in line 19 of CTL-file file of ctype1, i.e., ctype1.ctl.

4.3.2 Errors in the tdmInfo

The tdmInfo parser can also identify a syntax error to a line in a MTL-file. For example, the following error message appears when a particular syntax error occur.

curChipType=app1, tdmInfo.l:[error 1] line 2 near 1 =: syntax error.
In mtlInfo: grammar rule: err 1

This message indicates that the tdm description for the chip app1 contains a syntax error in line 2 of the TDM description. The violated grammar rule is err 1. When checking the grammar rule used in the file tdmInfo.y, one can easily located the err 1.

4.3.3 Errors in the mtlInfo

The tdmInfo parser can indicate syntax error to a line in the source input. A typical error message is shown below.

mtlInfo.l:[error 1] line 7 near 3 xxx: syntax error.
In mtlInfo: grammar rule: err 7.

This message indicates that there is a syntax error is line 7 of the mtl input file. The syntax error results a violation of grammar rule identified by err 7. This error can be located in the file mtlInfo.y.

4.3.4 Errors in the m2cp

The m2cp parser cannot indicate errors in the source input since it can accept C statements. Errors in a test procedure of a MTL-file will be detected by the C compiler.

5 Running the Test Programs

5.1 Test Program Compilation

Once the test programs for a module (e.g., described by work.mtl) have been sent to the IBM PC, they can be compiled into executable code (e.g., work.exe). The compilation process is made easy by a make file (e.g., work.mak) generated by a utility of BOLD called genTarget. Currently, the make file is directly usable by the MAKE utility that comes with the Microsoft C compiler. To make the executable code work.exe just type **make work.mak** in the DOS command prompt.

5.2 Executing the Test Program

The module under test must be properly connected to the test controller before executing the test programs. The execution is started by typing in the name of the executable code, e.g., **work** for the code work.exe. If there is no device connected to the test bus of the test controller, or the power is not turn on, then the processor will executing an infinite loop and the PC is stalled. The only way out under such circumstance is to reboot the IBM PC, which can be done by turning the power of the PC off and on.

5.3 Run Time Errors on the IBM PC

In general no problem should occur during the execution of a test program in the IBM PC. However, due to the limitation of the DOS operating system, run time errors can arise.

The run time error in general is hard to locate. However, from experience, most of the errors occur because of stack overflow. This problem can be solved by either increasing the size of the stack or change the memory model used in the compilation.

5.3.1 Increase the Stack Size

The stack size can be changed at the linkage of test program. This can be done by modifying a line of the make file. For example, in the file work.mak the command **LINK /ST:2048 $**, $@;** indicates that the stack size is 2048 bytes. By changing the option to /ST:8096 the stack size can be increased to 8096 bytes. The default size of the stack is 2048 bytes. Note that the stack size cannot exceed 64K bytes.

5.3.2 Change the Memory Model

When compiling a program using CL, the Microsoft C compiler, the default memory model is /AM. If a test program is very large then it is beneficial to change the memory model to /AL or /AH. This can be accomplished by changing a line in the make file. For example, the

default compilation flags used in BOLD is CFLAGS=/AM /Od /FPc /c. One can change it to CFLAGS=/AL /Od /FPc /c for a large memory model. For details of the compiler options, the reader is referred to [11].

6 Checking the Consistency of the Test Controller

Sometimes it is hard to determine whether the problem is due to the test controller or the module under test. A consistency check program is provided for testing the correctness of the test controller. This program is called **one.exe**. Before running this program the test bus connection to the module under test must be disconnected and a jumper that connects the TDO of an appl chip to the TDI of the test channel must be set. The jumper is next to the test channel chip and is clearly marked on the Proto-Board which contains the test channel chip. The successful completion of this program guarantees the correctness of the test controller. Remember to remove the jumper before connecting the test bus to a module under test.

References

[1] M.S. Abadir and M.A. Breuer, "A Knowledge-Based System for Designing Testable VLSI Chips", *IEEE Design & Test of Computers*, pp. 56-68, August 1985.

[2] A. El Gamal, "Protozone: The PC-Based ASIC Design Frame", *EE218 Handout No.7*, Stanford University, Winter 1990.

[3] IEEE Standard 1076-1987, "IEEE Standard VHDL Language Reference", *IEEE Standards Board*, 345 East 47th Street, New York, NY 10017, March 1988.

[4] IEEE Standard 1149.1-1990, "IEEE Standard Test Access Port and Boundary Scan Architecture," *IEEE Standards Board*, 345 East 47th Street, New York, NY 10017, May 1989.

[5] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler", in B.W. Kernighan and M.D. McIlroy, *UNIX Program's Manual*, Bell Laboratories, 7th Edition, 1978.

[6] M.E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator", in B.W. Kernighan and M.D. McIlroy, *UNIX Program's Manual*, Bell Laboratories, 7th Edition, 1978.

[7] J.C. Lien and M.A. Breuer, "A Universal Test and Maintenance Controller for Modules and Boards", *IEEE Trans. on Industrial Electronics*, Vol. 36, No. 2, pp. 231-240, May 1989.

[8] J.C. Lien, "A Module Maintenance Controller Prototype", *Technical Report CENG 90-14*, Department of EE-Systems, University of Southern California, June 1990.

[9] K.P. Parker and S. Oresjo, "A Language for Describing Boundary Scan Devices", *Proc. Int'l Test Conf.*, pp. 222-234, 1990.

[10] J.C. Lien, "Design of Hierarchically Testable and Maintainable Systems", *Ph.D. Dissertation*, University of Southern California, 1991.

[11] "Microsoft C5.0 Optimizing Compiler User's Guide", *Microsoft*, 1987.

Appendix

A. BSDL - Boundary Scan Description Language

B. A Step-by-step Demonstration

C. Input Descriptions for the Demonstration

D. Synthesized Test Programs for the Demonstration

E. Data Sheet of TI SN74BCT8244

A BSDL - Boundary Scan Description Language

A Language for Describing Boundary-Scan Devices

Kenneth P. Parker and Stig Oresjo
Hewlett-Packard Company, Manufacturing Test Division
P. O. Box 301, Loveland Colorado, 80537

ABSTRACT

Boundary-Scan (IEEE Standard 1149.1-1990) technology is beginning to be embraced in chip and board designs. One key need is a way to simply and effectively describe the feature set of a Boundary-Scan compliant device in a manner both user friendly and suitable for software to utilize. A language subset of VHDL is proposed here for this purpose. As with any new standard, the industry is learning how to apply its rules and mistakes will occur. A derivative effect of the language proposed here is that if a device is not describable by the language, then that device does not comply with the 1149.1 standard. While the converse is not true, the language still allows a syntactic check for compliance as well as a number of semantic checks.

INTRODUCTION

IEEE Standard 1149.1-1990[1] was approved in February 1990, and is now available from the IEEE. The Boundary-Scan concept was formally investigated by the Joint Test Action Group, a consortium of European and North American companies starting in 1985, and is often refered to as the JTAG Standard. The standard is rich in options and is open-ended in that user defined features are provided for. This richness can be a source of complication that must be accounted for while utilizing the standard. The testability enhancing attributes of the standard are quite powerful. Many of the barriers that have slowed the adoption of testability technology[2] are directly overcome by Boundary-Scan[3]. For these reasons, expect to see widespread application of the standard. In this paper it is assumed the reader has a passing knowledge[1][3][4] [5] of Boundary-Scan.

As new products become available to support Boundary-Scan designs, each will have the problem of how to describe a designer's unique application of the standard. Some sort of description will be necessary for each device containing Boundary-Scan. This paper describes a language that captures the essential features[6] of an implementation. This language is called the Boundary-Scan Description Language (BSDL) and is written within a subset of the VHSIC Hardware Description Language (IEEE Std 1076-1987 VHDL[7]). It has two criteria to meet: first that it be 'user friendly', since people will have to create the files; and second, it should be simply and unambiguously parsable by computer. This proposal is intended to be a 'straw man' or 'Version 0.0', illustrating a structure and illuminating needs.

It is important to note that the language described here is necessarily evolving. However, it represents a consensus developed from discussions[6] with many individuals within various sectors of the electronics industry as noted in the acknowledgements. Several groups had already begun their own development efforts on proprietary languages suited to their individual needs; of note, AT&T, Hewlett-Packard, Philips, and Texas Instruments. In particular, the Philips work is part of an effort supported by the multinational European Commission ESPRIT Project 2478. It is now the intention that this European activity will merge with this proposal. This process is now underway and in this respect, this proposal reflects both North American and European thinking. While this language definition is expected to change as applications develop, it is our hope that the resulting evolution will differ in minor ways, with a goal of upward compatibility. Thus, software tool developers can make use of this proposal now rather than continue to wait. In so doing they will benefit from compatibility with other segments of the industry. Ultimately, this language should be taken over and maintained by a body devoted to standards, such as the IEEE.

THE SCOPE OF THE LANGUAGE

The BSDL language allows description of the *testability features* in IEEE Std 1149.1-1990 compliant devices. This language can be used by tools that make use of those testability features. Such tools include testability analysis, test generation and failure diagnosis. Note that BSDL itself is not a general purpose hardware description language. With a BSDL description of a device *and knowledge of the standard*, it is possible for tools to completely understand the data transport characteristics of the device. With additional capabilities provided by VHDL, it is possible to perform simulation, verification, compliance analysis, and synthesis functions. Support for these functions is beyond the scope of BSDL alone.

A key characteristic of a BSDL description of the parameters of an implementation is orthogonality to the rules of the standard. As a result, elements of a design absolutely mandated by the standard are not included in BSDL descriptions. For example, the BYPASS Register is not described in BSDL because it is completely described by the standard itself, without option. This eliminates both redundancy and the opportunity for error.

BOUNDARY-SCAN CHARACTERISTICS

What are the characteristics of any Boundary-Scan device that need description? All such devices must have two major features; a Test Access Port (TAP) and a Boundary Register. The parameters of these features are described by BSDL.

The parallel/serial Boundary Register is made up of Boundary Cells which are associated with device inputs, device outputs, device bidirectional signals, and specific embedded device control signals. A great deal of the flexibility of the standard is reflected in the Boundary Register rules.

The TAP possesses either four or five dedicated signals, familiarly labeled TCK, TMS, TDI, TDO and, optionally, TRST*. It must contain an instruction register and a BYPASS register. The TAP implements a minimum set of mandatory instructions which control operation of the Boundary-Scan facility. These instructions operate in conjunction with the dedicated TAP signals in a precisely

prescribed way. The TAP may also contain optional data registers and optional instructions as specified by the 1149.1 standard. Additionally, the TAP may also be endowed by a device designer with additional user-specified data registers and instructions beyond those specified by the standard, but governed by rules of implementation within the standard.

Notice by conspicuous absence that the TAP state diagram is not described here. This information is inherent in the 1149.1 standard itself and does not need to be specified as part of a device adhering to the standard. In essence, stating "1149.1-1990" implies a great deal of information common to any such device. The proposed language is intended to specify those parameters necessarily unique to a given Boundary-Scan device implementation.

As further context, a device should be thought of as a black-box with terminal connections. Inside is the TAP and the *system logic*[1] surrounded on its perimeter by the Boundary Register logic. We want to describe the properties of the Boundary Register and terminal connections without need for describing the system logic. This independence recognizes a major contribution of Boundary-Scan; we can decouple problems such as board test from the system logic of the ICs.

LANGUAGE ELEMENTS

The language consists of a case-insensitive free-form multiline terminated syntax which is a *subset* of VHDL.[7] Comments are any text between a "--" symbol and the end of a line, syntactically terminating that line. Some of the information is conveyed in VHDL strings; sequences of characters between quote marks. This information is associated with a VHDL attribute and has a BSDL syntax requirement. Obviously, this is not checked by VHDL itself, but by applications that consume this information. (This is one reason the BSDL name is retained.) In practice, this information will be used in two environments. The first is a full VHDL-based system. It passes a BSDL description through its VHDL analyser into a compiled design library. From there, VHDL design library based tools can extract Boundary-Scan data by referencing the appropriate attributes. The second environment is a non-VHDL system capable of parsing a limited set of VHDL syntax (simply skipping items it doesn't recognize) to find and parse the BSDL information. In support of these systems, we constrain the full power of VHDL into a *standard practice*. Standard practices will be indicated as they are used in this text. Thus, BSDL is a "subset and standard practice" of VHDL.

BSDL is composed of three sections. These are: **Entity, Package,** and **Package Body.** An entity is the basis for describing a device within VHDL and an example for a real device is shown in Appendix A. Within the entity, the Boundary-Scan parameters of a device are described. The 1149.1 related definitions come from a pre-written, standard VHDL package (and related package body). The definitions for a 1149.1-1990 package and package body are given in Appendix B. The package information is directly related to the 1149.1 standard and is only expected to change when the standard itself is changed. Typically, this information would be write-protected. The development of new standards in the future would require new packages to be created.

A user may add an additional package (and package body), to contain user-specific design information. An example of this would be to contain a library of cell definitions unique to the users application, perhaps dependent upon the silicon technology in use. The reason for breaking out package bodies as seperate units is to allow the updating of the data within these without causing the need for recompilation of all entities that reference the corresponding package.

A simple Backus-Naur Form (BNF)[8] is used to describe the syntax of BSDL data within VHDL strings (see also Appendix C). Where the meaning is obvious without the use of BNF, the description is given by example. Since many of these strings are potentially long, the concatenation operator '&' is used to break them into managable pieces. The syntax descriptions will not show this, and, the concatenation operation may be thought of as a lexicographical pre-processing step before parsing.

THE ENTITY DESCRIPTION

An *entity* describes a device's I/O port and important *attributes* of the device. For BSDL, an entity has the following structure:

```
entity My_IC is          -- an entity for my IC

  [generic parameter]
  [logical port description]
  [use statement(s)]
  [package pin mapping]
  [scan port identification]
  [TAP description]
  [Boundary Register description]

end My_IC;               -- End description
```

The order of the elements within the entity as shown above is a required standard practice to simplify non-VHDL applications. The next few sections will examine each element of the entity.

Generic Parameter

The generic parameter is a VHDL construct used to pass data into a VHDL model. In BSDL it is intended as a method for selecting among several packaging options that a device may have. Each option may have a different mapping between the pins of the package and the bonding pads of the device. Even devices manufactured in a single package will be tested before packaging, with a different mapping possible. We call this the *logical-to-physical* relationship of the signals of the device. The description of the Boundary-Scan architecture of the device is done with the logical signals. Applications such as board testing will need to know how the logical structure of the device maps onto a set of physical pins. A VHDL generic parameter is used for this. It must have the name shown in order for software to seperate it from other parameters that might be passed to the entity. It has this form:

1. The 'system logic' is the same refered to by the 1149.1 document. However, important 'null' logic cases must also be treated as will be discussed.

generic(PHYSICAL_PIN_MAP:string := "undefined");

Note the string is initialized to an arbitrary value ("undefined") that will not allow a package selection if the parameter is not bound to a value, i.e., not passed. The use of this parameter will become clear shortly.

Logical Port Description

The port description uses the VHDL port list in a standard practice. Here, we are assigning meaningful symbolic names to the device's system terminals. These symbolic names are used in subsequent descriptions. This allows the majority of the statements to be 'terminal independent'; that is for example, independent of a renumbering or other reorganization of the terminals of the device. It also allows description of devices which may be packaged in several different forms. It is optional to include non-digital pins such as power, ground, no-connects, or analog signals, but these should be included for completeness. Non-digital pins will not be referenced later in the description, but all pins referenced in the description must have been defined here. The form is:

port(<*PinID*>; <*PinID*>; ... <*PinID*>);

<*PinID*> ::= <*IdentifierList*>: <*Mode*> <*PinType*>

<*IdentifierList*> ::= <*Identifier*> |
<*IdentifierList*> , <*Identifier*>

<*PinType*> ::= <*PinScaler*> | <*PinVector*>

<*PinScaler*> ::= <*Identifier*>

<*PinVector*> ::= <*Identifier*>(<*Range*>)

<*Mode*> ::= in | out | inout | buffer | linkage

<*Range*> ::= <*number*> to <*number*> |
<*number*> downto <*number*>

The <*Mode*> identifies the system usage of a device pin, with *in* for a simple input pin, *out* for an output pin that may participate in buses, *buffer* for an output pin that may not participate in buses, *inout* for a bidirectional signal pin, *linkage* for other pins such as power, ground, analog, or no-connect. A <*PinVector*> is a shorthand for grouping related signals. For example, *Data(1 to 8)* indicates there are 8 signals named *Data* indexed from 1 to 8, like *Data(3)*. A <*PinScaler*> is a single signal. Note, every pin must have a unique name, so if there are several ground pins for example, they must have different names such as *GND1, GND2*, etc, or be expressed as a vector. An example of a port statement for a 22 pin device is:

port(CLK:in bit; CLEAR:in bit; Q:out bit_vector(1 to 8);
DATA:in bit_vector(1 to 8); VCC, GND:linkage bit);

Bit and *bit_vector* are type names known to VHDL.

Use Statement(s)

The **use** statement identifies a VHDL package needed for defining attributes, types, constants, and other items that will be referenced. The following statement is mandatory in BSDL. Others may also be added to support user defined Boundary Register cells. The content of this package and its associated package body is shown in Appendix B.

use STD_1149_1_1990.all; -- Get 1149.1 information

Package Pin Mapping

VHDL **attribute** and **constant** statements are used to show the package pin mapping. These are shown by example:

attribute PIN_MAP of My_IC:entity
is PHYSICAL_PIN_MAP;

constant dw_package:PIN_MAP_STRING :=
<MapString>;

Attribute PIN_MAP is a string that is set to the value of the parameter PHYSICAL_PIN_MAP, already described. VHDL constants are then written, one for each packaging variation, that describe the mapping between the logical and physical pins of the device. (The BSDL syntax for <*MapString*> is given in Appendix C.) In a VHDL design library, the value of PIN_MAP can be used to identify the constant (by name) that contains the mapping of interest. In a non-VHDL implementation, the parse phase would look for the constant with a name matching the value of PIN_MAP. Note, the type of the constant must be PIN_MAP_STRING. This allows such parsers to ignore constants of other types. An example of a mapping is:

"CLK:1, DATA:(6,7,8,9,15,14,13,12), CLEAR:10, " &
"Q:(2,3,4,5,21,20,19,18), VCC:22, GND:11"

Notice it is the concatenation of two smaller strings. This is arbitrary; a string is the result after all concatenations are performed. A BSDL parser will read the content of the string. It matches signal names like *CLK* with the names in the port definition. The symbol on the right of the colon is the physical pin associated with that port signal. It may be a number, or an alphanumeric identifier because some packages such as Pin-Grid Arrays (PGAs) use coordinate identifiers like A07, or H13. If signals like *DATA* are <*PinVector*>'s in the port definition, then a matching list of pins enclosed in parenthesis are required. The physical pin mapped onto *DATA(5)* is pin 15 in the above example.

Scan Port Identification

Next we give the 5 attributes that define the scan port of the device. These are shown by example:

attribute TAP_SCAN_IN of TDI:signal is true;
attribute TAP_SCAN_OUT of TDO:signal is true;
attribute TAP_SCAN_MODE of TMS:signal is true;
attribute TAP_SCAN_RESET of TRST:signal is true;
attribute TAP_SCAN_CLOCK of TCK:signal
is (17.5e6, BOTH);

Here, signal names TDI, TDO, TMS, TRST and TCK must have appeared in the port description. The names chosen here match the 1149.1 standard, but may be arbitrary. The TAP_SCAN_RESET attribute is optional but the others must be specified for a correct implementation. The boolean assigned is arbitrary; the statement is used to bind the attribute to the signal. The TAP_SCAN_CLOCK attribute is a record with a real number field (the first) that gives the maximum operating frequency for TCK. The second field is an enumerated type with values LOW and BOTH which specify which state(s) the TCK signal may be

stopped in without data loss in Boundary-Scan mode.

TAP Description

The next major piece of Boundary-Scan functionality that must be described is the device dependent characteristics of the TAP. It may have four or five control signals, already identified. It may have a user specified instruction set (within the rules) and a number of data registers and options. The following sections show how this is described.

The TAP Instruction Register may have any length 2 bits or longer and is required to support certain opcodes and some (but not all) of these have mandatory bit patterns. A designer may add 1149.1-identified optional instructions and/or new instructions with completely dedicated functions. An instruction may have several bit patterns. Unused bit patterns will default to the BYPASS instruction. Upon resetting the TAP or passing through the *Test-Logic-Reset* state, the instruction register is jam-loaded with a specific instruction. The standard provides for 'private' instructions which need not be documented *except* if their access could create an unsafe condition such as a board level bus conflict. Our language must easily denote these characteristics and take advantage of opportunities for semantic checks.

The characteristics of the instruction register that we capture with the language are *length, opcodes, capture, disable, private* and *usage*. Since these are basically simple, they are introduced by example.

attribute INSTRUCTION_LENGTH of My_IC:entity
is <integer>;
attribute INSTRUCTION_OPCODE of My_IC:entity
is <OpcodeTable>;
attribute INSTRUCTION_CAPTURE of My_IC:entity
is <Pattern>;
attribute INSTRUCTION_DISABLE of My_IC:entity
is <OpcodeName>;
attribute INSTRUCTION_PRIVATE of My_IC:entity
is <OpcodeList>;
attribute INSTRUCTION_USAGE of My_IC:entity
is <UsageString>;

Example:

```
attribute INSTRUCTION_LENGTH of My_IC:entity is 4;

attribute INSTRUCTION_OPCODE of My_IC:entity is
        "Extest (0000), " &
        "Bypass (1111), " &
        "Sample (1100, 1010), " &
        "Preload (1010)," &
        "Hi_Z   (0101), " &
        "Secret (0001) ";

attribute INSTRUCTION_CAPTURE of My_IC:entity is
        "0101";

attribute INSTRUCTION_DISABLE of My_IC:entity is
        "Hi_Z";

attribute INSTRUCTION_PRIVATE of My_IC:entity is
        "Secret";
```

The **instruction_length** attribute defines the length that all opcode bit patterns must have. The **instruction_opcode** attribute is a BSDL string (syntax defined in Appendix C) containing the opcode identifiers and their associated bit patterns. The rightmost bit in the pattern is closest to TDO. The standard mandates the existence of EXTEST, BYPASS, and SAMPLE instructions, with mandatory bit patterns for the first two. Note that other bit patterns may also decode to these same instructions.

The **instruction_capture** attribute string states what bit pattern is jammed into the shift register portion of the instruction register when the TAP passes through the *Capture-IR* state. This bit pattern is shifted out whenever a new instruction is shifted in, and the standard mandates the least 2 significant bits must be a "01". Note, this bit pattern may be design-specific data. Since it is possible, by traversing from *Capture-IR* to *Exit1-IR* to *Update-IR*, to cause this pattern to become the effective instruction, it will act as some instruction (if not simply BYPASS) when it becomes effective. This bit pattern is not the instruction loaded into the instruction register when passing through the *Test-Logic-Reset* state. The standard states that on passing through the reset state, the *effective* instruction is jammed either to BYPASS, or IDCODE if it exists.

The optional **instruction_disable** attribute identifies an opcode that makes a Boundary-Scan device "disappear". In this mode, the 3-state outputs are disabled and the BYPASS register is placed between TDI and TDO. This is not (yet) a specified behavior in the 1149.1 standard, but many devices have this capability today because it is very useful for testing purposes. This attribute allows the opcode to be identified for software use.

The optional **instruction_private** attribute identifies opcodes that are private and potentially unsafe for access. By definition, the results of these instructions are undefined to the general public and should be avoided. Software can monitor the instruction register to issue warnings or errors if a private instruction is loaded during run time.

The optional **instruction_usage** attribute is a BSDL string with the syntax given in Appendix C. The usage concept will be covered in its own section later.

ID Register Values

Next, we need to identify standard prescribed optional registers. These are the IDCODE and USERCODE registers. Note, if an IDCODE instruction exists, an IDCODE register must also exist. Further, the existence of USERCODE implies the existence of IDCODE. To describe these instructions we need two attributes.

```
attribute IDCODE_REGISTER of My_IC:entity is
        "0011" &                -- 4  bit version
        "1111000011110000" &    -- 16 bit part number
        "00000000111" &         -- 11 bit manufacturer
        "1";                    -- mandatory LSB

attribute USERCODE_REGISTER of My_IC:entity is
        "10xx" & "0011110011110000" &
        "00000000111" & "1";
```

The bit patterns must be 32 bits long. The rightmost bit is closest to TDO. In the examples above, concatenation is used to delimit fields within the codes. The "X" values specify a don't-care for that bit position. This is used to nullify subfields within a code that are not important for testing purposes.

Register Access

All TAP instructions must place a shiftable register between TDI and TDO. User-defined instructions may access data registers mandated by the standard; the Boundary Register, the IDCODE register, and the BYPASS register. The standard allows a designer to place additional data registers in the design. These are referenced by user-defined TAP instructions. It is important for software to know the existence and length of these registers and their associated instruction(s). Therefore we need to express these associations in the language. The attribute for this is:

attribute REGISTER_ACCESS of My_IC:entity
is <RegisterString>;

The syntax for <RegisterString> is in Appendix C. Example:

attribute REGISTER_ACCESS of My_IC:entity is
"Boundary (Secret, User1), " &
"Bypass (Hi_Z, User2), " &
"MyReg[7] (LoadSeed, ReadTest)";

In this example, *Secret, Hi_Z, User1, User2, LoadSeed* and *ReadTest* must be previously defined user instructions. Note that a seven bit user-register *MyReg* has been added to the TAP, with two instructions that access it. The 1149.1 standard itself defines the following relationships implicitly, so these need not be given.

attribute REGISTER_ACCESS of My_IC:entity is
"Boundary (Extest, Sample, Intest)," &
"Bypass (Bypass), " &
"Idcode (Idcode, Usercode)";

This ability to identify register access allows software to know the length of a scan sequence, which is dependent on the currently effective instruction. The mandatory Boundary Register, Bypass Register and Instruction Register are known from other statements, as well as their relationship to TAP instructions. Note that a semantic check can be made here ensuring that each instruction has an associated data register as required by the standard. Exceptions to this are the instructions marked 'private' since they are not to be accessed, nor do their target registers need to be identified.

The standard also allows user-instructions to reference several registers at once in a concatenated mode, but also requires them to have a new name in this mode. Here, we would treat this concatenation as if it were a new register with a distinct name and length. The reason is that in any case, the data flowing out of any register after passing through the *Capture-DR* state is not known to BSDL because *it is not a simulation language.* We are not attempting here to completely characterize the entire design so that its behavior is simulatable. This is more properly the domain of VHDL itself. We are simply trying to capture the relevant characteristics of Boundary-Scan devices so that we can intelligently manipulate chains of such devices. Other software can predict what must be coming out of various registers. This allows us to divide testing problems into two parts: calculating tests at an abstract level and manipulating the chain to deliver them. The language described here deals mainly with this second task. This division is important since there are several configurations (even proposed within the standard itself) for setting up Boundary-Scan chains. The abstracted test can be independent of these configurations.

Boundary Register Description

The Boundary Register is an ordered list of Boundary Cells, numbered 0 to N where $N+1$ is the number of cells in the register. Cell 0 is closest to TDO. There are cells of varying design and purpose. The standard, in chapter 10[1], shows fifteen such designs as examples. Others are possible as well. In discussing cell structures we will make heavy use of reference to the standard and its figures depicting cell designs to save space in this paper. To avoid confusion with figure references, a symbol such as *f10-16[1]* will refer to figure 10-16 in the standard. Symbols such as *f10-19c[1]* and *f10-19d[1]* refer to the *control* and *data* cells that make up the structure shown in figure 10-19 of the standard.

Cells must be identified before they are referenced in the Boundary Register description that follows. However, since the standard does give a number of examples that will likely be adopted in a design, we have constructed the language to have *intrinsic* or predefined cells that may be referenced via a simple nomenclature. Cell names are listed in Table 1 and their definitions are shown in the VHDL package body given in Appendix B. However, there will still be a need to define other cells not covered by the intrinsics. The details of these definitions are defered until later. If the intrinsic definitions contain the cells one needs for a description, then no cell definitions are required at all.

BC_1	*f10-12[1], f10-16[1], f10-18c[1], f10-18d[1], f10-21c[1]*
BC_2	*f10-8[1], f10-17[1], f10-19c[1], f10-19d[1], f10-22c[1]*
BC_3	*f10-9[1]*
BC_4	*f10-10[1], f10-11[1]*
BC_5	*f10-20c[1]*
BC_6	*f10-22d[1]*

Table 1. List of intrinsically defined cells and the figures covered in the standard.

Numerous rules must be observed when using the cells to create a Boundary Register as covered in chapter 10[1] of the standard. Some of these may be checked during compilation of a device's description. For example, some cell designs may only be used on a device input. Some will not support the *INTEST* instruction, which is allowable if the device TAP description does not list that instruction. Some cells require the aid of another cell to control 3-state enables. Checks can be performed and problems discovered as soon as a device's Boundary-Scan behavior has been specified and described, which may be well in advance of device fabrication.

A very general cell design from the standard (*f10-16[1]*) is shown in Figure 1. In Figure 2a we show a symbol that captures the essence of this cell needed for discussion. The design in Figure 1 is comprised of a *parallel input*, a *parallel output*, a multiplexer controlled by a *Mode* signal, and two *flip-flops*. The *Mode* signal is a function of the currently effective instruction. Yes, there are other elements such as the signals shifted in from the last cell and to the next cell. Yes, there is a second multiplexer controlled by signal ShiftDR. Yes, there are two clock signals ClockDR and UpdateDR. *But,* all these additional elements are always

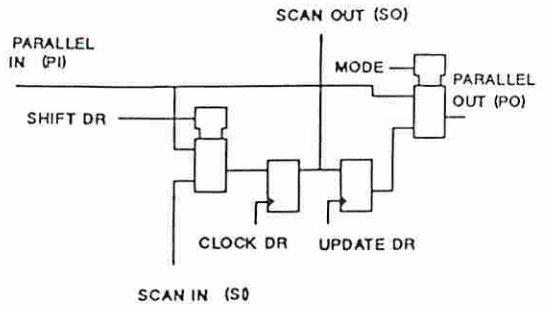


Figure 1. Cell design from f10-16[1].

precisely prescribed by 1149.1 and as such, *may be omitted from our consideration* in this language. This leads us to the symbol in Figure 2a which is simple.

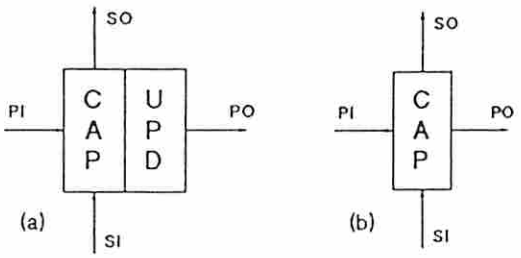


Figure 2. Two symbols for a typical Boundary Cell, one (a) with an UPD flip-flop and one without (b).

The parallel input and output are shown, and these are connected to various places depending on the application. The two flip-flops are labeled CAP and UPD to symbolize their use; the CAP flip-flop captures data in the *Capture-DR* state. The UPD flip-flop captures data in the *Update-DR* state. The shift path is shown because many such cells will be linked together in a shift chain that makes up the Boundary Register. The shift path links *only* the CAP flip-flops. Now, one cell design shown in *f10-11*[1] has a symbol (Figure 2b) with no UPD flip-flop.

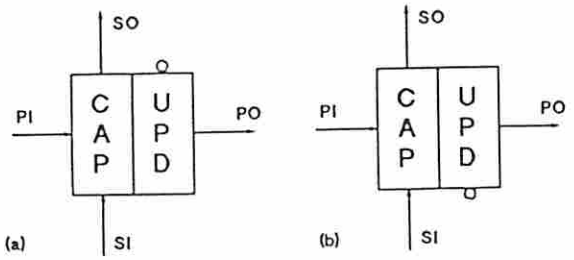


Figure 3. Symbols for a Boundary Cells with preset (a) and clear (b) on the UPD flip-flop.

The symbols in Figure 3 show bubbles on the top or bottom of the UPD flip-flop to indicate that flip-flop may be preset (1) or cleared (0) when passing through the *Test-Logic-Reset* state, as the standard allows in *f10-21c*[1]). No signal connection is made to these bubbles.

Now we show the three attributes needed to define the Boundary Register:

attribute BOUNDARY_CELLS of My_IC:entity is <CellList>;

attribute BOUNDARY_LENGTH of My_IC:entity is <integer>;

attribute BOUNDARY_REGISTER of My_IC:entity is <CellTable>;

The <CellList> and <CellTable> are strings with syntax given in Appendix C. An example of a 3 cell Boundary Register is:

```
attribute BOUNDARY_CELLS of My_IC:entity is
        "BC_1, MyCell";
attribute BOUNDARY_LENGTH of My_IC:entity is 3;
attribute BOUNDARY_REGISTER of My_IC:entity is
  -- num  cell    port  function  safe  [ccell disval rslt]
    " 0  (BC_1,   IN,     input,    X),"    &
    " 1  (BC_1,   *,      control,  0),"    &
    " 2  (MyCell, OUT,    output3,  X,    1,  0,  Z)";
```

The first attribute shows the cells used in the register; *BC_1* from the standard package, and *MyCell*, which must have been described in a user defined package. A semantic check can occur here; do these cells support the standard instructions that are listed for the TAP opcodes? For example, the TAP may support INTEST, but does *MyCell*?

The second attribute defines the number of cells in the Boundary Register. This number must match the number actually found in the third attribute, the register itself. This attribute is a string containing a list of elements, each with two fields. The first field is merely the cell number, which must be between 0 and LENGTH-1. (They may be listed in any order.) The second is a set of subfields within parentheses. There are either four or seven subfields. They are labeled, as in the comment above, **cell, port, function, safe, ccell, disval,** and **rslt.** All cells have the first four subfields. Only cells providing data for device outputs that can be disabled have the remaining three subfields. These three specify how to disable the output.

The cell subfield identifies the cell design used. It must match a cell given in the **boundary_cells** attribute.

The port subfield identifies the port signal actively driven or received by this cell. A cell serving as an output control or internal cell will have an asterisk in this position.

The function subfield shows the primary function of the cell. Table 2 shows the values this subfield may have:

input	a simple input pin receiver (*f10-8*[1])
clock	a cell at a clock input (*f10-11*[1])
output2	supplies data for a 2-state output (*f10-16*[1])
output3	supplies data for a 3-state output (*f10-18d*[1])
control	controls 3-state drive or cell direction (*f10-18c*[1])
controlr	a *control*, disables at *Test-Logic-Reset* (*f10-21c*[1])
internal	captures internal constants (*see page 10-7*[1])
bidir	reversible cell for a bidirectional pin (*f10-22d*[1])

Table 2. Function subfield values, meanings, and a figure reference of a representative cell in the standard[1].

Shortly, we discuss cells with more than one function. Note that the function is with respect to the boundary cell and not the device pin. This reflects the fact that two cells may service a single pin, for example, one serving as an input receiver and the other serving as an output driver, on a

bidirectional pin (*f10-21*[1]). *Internal* cells are used to capture 'constants' (0's and 1's) within a design. They are specifically *not* allowed to be surrounded by system logic (*f10-7*[1]). One proposed use of this is to capture an encoded value (perhaps in the first few bits of the Boundary Register) as an informal identification code. Another was proposed in[9] where sense amplifiers monitor redundant power connections and place the measured results in internal Boundary Register cells. If the power connections are good, the data loaded will be constant. Finally, there may be "extra" cells unused in a programmable device (see page 10-7 of the standard[1]).

The **safe** subfield gives the value that a designer prefers to be loaded into the UPD flip-flop of the cell when software would otherwise choose a value randomly. Two examples are; the value that an output should have that is safe to overdrive during In-Circuit testing; or, the value to present to on-chip logic at a device input during EXTEST. An 'X' signifies that it doesn't matter.

The **ccell** subfield identifies the cell number of the cell that serves as an output enable. The **disval** subfield gives the value the **ccell** must have to disable the output driver. The **rslt** subfield gives the state the disabled driver goes to; a high impedance state (Z), a weak '0' (Weak0), or a weak '1' (Weak1). The last two values correspond to asymetrical drivers such as TTL open-collector drivers or ECL open-emitters. The functions in effect when these three subfields exist must be *output2, output3* or *bidir*. If it is *bidir*, then disabling the driver implies the cell is a receiver.

An Example Boundary Register Description

We now use the device shown in Figure 4 to illustrate a Boundary Register description and how special cases are handled. These special cases arise because the standard allows cells to be merged when *the system logic between them is null.* (See for example, *f10-4*[1], *f10-5*[1].) Cells may be merged if the logic between them is simply a non-inverting data path, like a wire or buffer. When merging is done, the resulting cell must obey a combination of the rules of the merged cells. Here is the definition of the Boundary Register for Figure 4.

```
attribute BOUNDARY_CELLS of Figure4:entity is
        "BC_1, BC_2, BC_6";
attribute BOUNDARY_LENGTH of Figure4:entity is 10;
attribute BOUNDARY_REGISTER of Figure4:entity is
-- num cell port      function safe [ccell disval rslt]
  "0 (BC_1, *,         control, 0)," &
  "1 (BC_1, OUT2,      output2, 1,  1,  1, Weak1),"&
  "2 (BC_6, BIDIR1,    bidir,   X,  3,  0, Z)," &
  "3 (BC_2, *,         control, 0)," &
  "4 (BC_1, *,         control, 0)," &
  "5 (BC_1, BIDIR3,    input,   X)," &
  "5 (BC_1, BIDIR2,    output3, X,  7,  1, Z)," &
  "6 (BC_1, BIDIR2,    input,   X)," &
  "6 (BC_1, BIDIR3,    output3, X,  4,  0, Z)," &
  "7 (BC_1, *,         control, 1)," &
  "8 (BC_1, IN2,       input,   X)," &
  "9 (BC_1, IN1,       input,   X)," &
  "9 (BC_1, OUT1,      output3, X,  0,  0, Z)";
```

Cell 0 is simply a control cell between the system logic and the enable for signal OUT1. Cells 4 and 7 are similar.

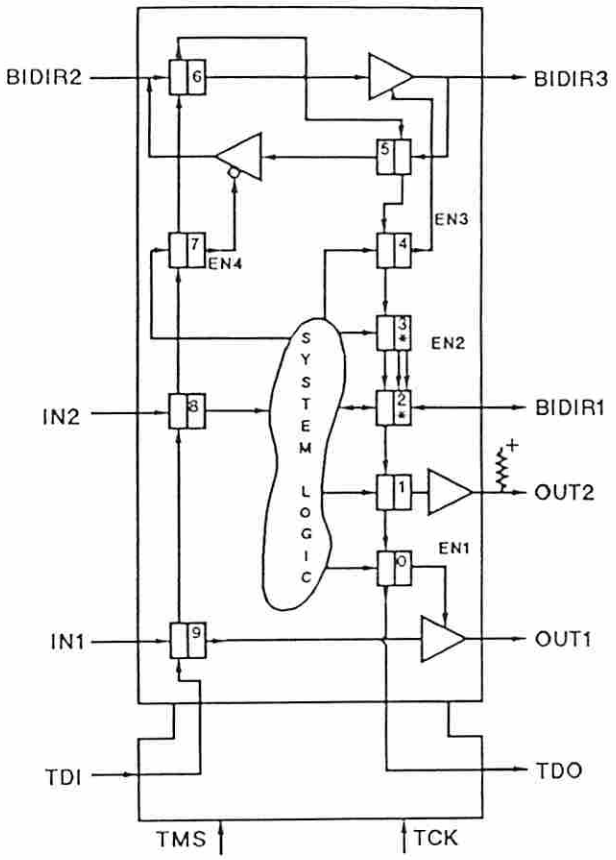


Figure 4. A device illustrating several merged cell situations.

Notice the **safe** bits are assigned to cause the associated drivers to disable. Cell 3 is the control for the reversible cell (*f10-22d*[1]) used on the bidirectional signal BIDIR1.

Cell 1 is a 2-state output data cell. Note that it has the three extra fields indicating that it controls its own open-collector asymetrical driver. Placing a '1' in cell 1 will disable OUT2 by putting it into the 'Weak1' state.

Cell 2 is the reversible cell of figure *f10-22d*[1]. This cell serves as an input if the control cell has turned off the output driver, meaning cell 3 produces a '0'. This cell serves as the data for the driver if the output is enabled. It cannot serve both functions. This is a drawback during test, since the value of BIDIR2 cannot be observed while the driver is turned on. A board level fault could not be seen by this device. Note that the structures for BIDIR2 and BIDIR3 (or *f10-21*[1]) would allow observation of the driver, thus allowing a simple consistancy check.

Cell 5 (and similarly for cells 6 and 9) has merged behavior; it serves as the input receiver for BIDIR3 and as the data source for BIDIR2. It has two lines of description in the Boundary Register definition as a result. The first gives its behavior as an input cell while the second describes its characteristics as an output cell. Note that cell BC_1 used in this capacity must support both *input* and *output3* functions. This is reflected in the definition of BC_1 (see appendix B) where both functions are seen to exist for all instructions.

This example is extreme in dwelling on odd cases. Most device implementations will be quite simple and routine, as the example in Appendix A, the Texas Instruments 74BCT8374[10], illustrates.

PACKAGE DESCRIPTION

The package that describes the Std 1149.1-1990 information needed for BSDL is given in appendix B. This package cannot be modified without changing BSDL itself.

There may be occasion for users to define their own packages for use in conjunction with the 1149.1 package. This is the way to add user-specific Boundary Cell definitions. By placing these in a package, they may be referenced by many entities. While it is possible to place an entire cell description in a package, it is standard practice to place the actual cell description in a *package body* (described next) associated with the package. All that then remains in the user-defined package is the names of the cells. For example, say a user wants to define two new cells for reference in a boundary-scan description. Here is what the package would look like:

```
package My_New_Cells is

    constant MNC_1 : CELL_INFO;  -- My new cell 1
    constant MNC_2 : CELL_INFO;  -- My new cell 2

end My_New_Cells;
```

Of course, to reference these cells, a use statement must appear in an entity description that references *My_New_Cells.all*, and the cell names must appear in the BOUNDARY_CELLS attribute string. The definition of these *deferred constants* goes into the related package body.

PACKAGE BODIES FOR DEFINING BOUNDARY CELLS

Now it is time to discuss the description of Boundary Register cells. We have already skimmed this subject in examining the description of the Boundary Register itself, and, we have benefitted from *intrinsic* cell definitions provided by the 1149.1 package body.

What are the important aspects of a cell we need to describe for BSDL to meet its statement of scope? In looking at the variety of possible cell designs given in the standard and the long list of rules governing these designs, this might seem to be a daunting task. It turns out that all of the cells shown in the standard (excepting *f10-22d*[1]) could be modeled as shown in Figure 5, *for the purposes of BSDL scope.*

For the case of cell *f10-22d*[1], its reversible nature can be represented as if it were two cells; one that is left-to-right and the other that is right-to-left, each modeled by Figure 5. The one to choose is defined by the value of the controlling cell. When enabled to drive, the cell works left-to-right and vice versa.

In BSDL, any cell consists of a Parallel Input (PI), a Parallel Output (PO), CAP and UPD flip-flops and connections to/from CAP flip-flops of other cells. Note, the UPD flip-flop may not exist in certain input cell configuration as allowed by the standard. The CAP flip-flop has eight choices of data source as shown. In looking at any particular cell design, usually only two or three of these choices are actually implemented. The '0' and '1' constants may be

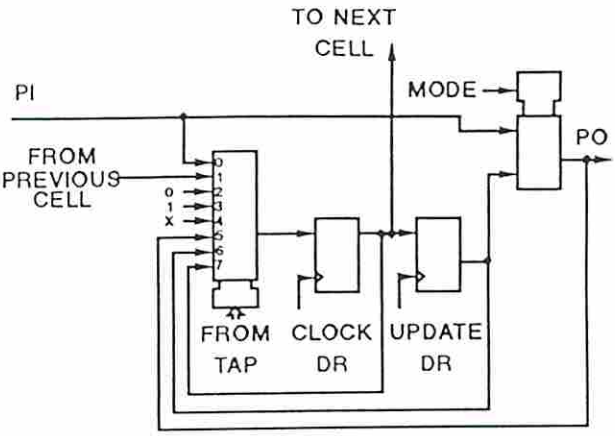


Figure 5. A general BSDL model of a Boundary Register Cell.

loaded into the CAP in certain situations. For example, an output cell design during EXTEST may load a constant into the CAP. The 'X' value denotes a don't-know or don't-care about what is loaded. An example of this is an output cell design during RUNBIST that loads a Linear Feedback Signature bit into the CAP. BSDL alone is not capable of simulating what this value could be. Also, proprietary information about a cell design may be hidden by "X-ing" out the activities of an instruction.

Context is another important factor in analysing Boundary Cells. What is the cell **function**? When a cell is an input cell, then PI **must** be connected to a device pin and PO **must** be connected to the system logic (ignore cell merging here). Now, add the context of the effective instruction. If EXTEST is in effect, then CAP **must** receive PI data. What we are defining here is a *triple* of data:

<function> <instruction> <CAP data source>

A cell description is a collection of these triples in the form of a VHDL array of records. Each triple tells us a piece of a cell behavior; for a given cell **function**, while a certain **instruction** is in effect, what **data** is loaded into the CAP when passing through the *Capture-DR* state. Since the CAP flip-flop data is what is eventually seen when scanning out data, it is important to know what the CAPs will contain. This data is simple to derive. One simply fixes the cell **function**, and then for each **instruction** supported, traces the **data** flowing to the CAP flip-flop.

What about other details? For example, input cell *f10-10*[1] in the standard produces a '1' on PO while EXTEST is in effect. BSDL does not model this because, during EXTEST on an input, we are looking *outward* from the device, not *inward*. Essentially, we do not care what is being fed to PO during EXTEST on input pins. The device designer *did* which is why the '1' is being injected; probably to satisfy some requirement internal to the device. During and after an EXTEST operation, the 1149.1 standard does not specify the what the state of the system logic will be so there is no point in trying to describe inputs to the system logic during EXTEST. There are similar arguments about what it is necessary to model during INTEST, SAMPLE and RUNBIST.

Many details are prescribed by the standard itself. The UPD flip-flops always get the CAP data transfered during the *Update_DR* state, so we need not describe this. If an UPD flip-flop is missing from a design, it can only be used as an input cell. If it still supports INTEST, then the CAP flip-flop will supply data to the system logic (and data ripple due to shifting is guaranteed by design not to matter), or, the input has been specifically designated as a **clock function** as the rules allow.

Defining a Boundary Cell

A cell is defined as a VHDL constant. It is an array of records with the range of the array unspecified, but implicit from the number of records given in the constant definition. Each field of each record must be filled. A standard practice is that these are filled using positional association rather than named association, as shown, to simplify development of non-VHDL based applications.

We give an example of a Boundary Cell C_Ex_1 that supports EXTEST, SAMPLE and INTEST. It loads a '1' into the CAP during EXTEST if the cell is used as an output or control function. The cell may be used as a simple input function. During INTEST as an input, it reloads the CAP with the data value that was shifted into the cell. Its description is:

```
constant C_Ex_1 : CELL_INFO :=
  ( (Output2, Extest, One),   (Output3, Extest, One),
    (Output2, Sample, PI),    (Output3, Sample, PI),
    (Output2, Intest, PI),    (Output3, Intest, PI),
    (Control, Extest, One),   (Input, Extest, PI),
    (Control, Sample, PI),    (Input, Sample, PI),
    (Control, Intest, PI),    (Input, Intest, CAP) );
```

The values allowed for **function** are the same as shown in Table 2 with the exception that *bidir* is replaced by two functions; *bidir_in* and *bidir_out*. A reversible cell such as *f10-22d*[1] is described only with these functions, with both required for every supported instruction as in cell BC_6 of Appendix B. The control cell, when enabling or disabling the cell as a driver, chooses between the *bidir_out* or *bidir_in* functions respectively. This is the only function with this complication.

The values allowed for an instruction are EXTEST, INTEST, SAMPLE and RUNBIST. Others such as BYPASS have no effect on the Boundary Cells. The values allowed for CAP data are *PI, PO, UPD, CAP, ZERO, ONE,* and *X*.

OTHER BSDL FUNCTIONS

There are some other features in BSDL, some of which we defered in previous discussion.

Instruction Usage

Generally, BSDL is a means for describing static design parameters of an 1149.1 implementation. However, the standard contains two instructions with details of operation that are not statically defined. These are RUNBIST and INTEST. The **instruction_usage** attribute gives additional information about the operation of an instruction. While targetted at the two standard instructions, it could be used to document details about a user-defined instruction as well. The types of information needed are; register identification, result identification and clocking information. This information is placed in string *<UsageString>* with syntax given in Appendix C. Here are examples for describing RUNBIST and INTEST and a user-defined instruction MYBIST:

```
attribute INSTRUCTION_USAGE of My_IC:entity is
  "Runbist (registers Boundary, Signature;" &
           " shift Signature;" &
           " result 0011010110000100;" &
           " clock TCK in Run_Test_Idle;" &
           " length 4000 cycles)," &
  "Intest  (clock SYSCLK shifted)," &
  "MyBist  (registers Seed, Boundary, Bypass;" &
           " initialize Seed 001110101;" &
           " shift Bypass;" &
           " result 1;" &
           " clock SYSCLK in Run_Test_Idle;" &
           " length 125.0e-3 seconds)";
```

The RUNBIST usage shows two registers used. Note, the standard states that only the Boundary Register may be initialized for test operation. A second register *Signature* is also used, and will be placed between TDI and TDO for shifting. When the test is completed, the *result* shifted out from *Signature* should match the given pattern (length must match length of *Signature* register), where the rightmost bit is closest to TDO. The test is run by clocking TCK 4000 times while in state *Run_Test_Idle*.

The INTEST usage tells us that clocking is accomplished by shifting the clock states to signal SYSCLK. This implies a cell structure for input signal SYSCLK that supports INTEST. If this cell had been a **clock** function rather than **input**, then the description would read (clock SYSCLK) and we would have to supply the clocking externally.

The MYBIST usage tells us that **registers** *Seed, Boundary*, and *Bypass* are involved in the test and that *Seed* requires initialization to a pattern. This will have to be done using another instruction since MYBIST places the Bypass register between TDI and TDO. Software could look in the **register_access** attribute for such an instruction. When the test is done, the Bypass register should contain the **result** '1'. Clocking is done with the TAP in *Run_Test_Idle*, with SYSCLK freerunning for 125 milliseconds.

Design Warnings

A device designer may know of situations where the system usage of a device can be subverted via the Boundary-Scan feature to cause circuit problems. As a simple example, a device may have dynamic system logic which requires clocking to maintain its state. Thus, clocking must be maintained when bringing the device out of system mode and into test mode for INTEST. The **design_warning** attribute can be assigned a string message to alert future consumers to the potential for problems. For example:

```
attribute DESIGN_WARNING of My_IC:entity is
    "Dynamic device, " &
    "maintain clocking for INTEST."
```

This warning is for application specific display purposes only. It is a textual message with no specified syntax and is not intended for software analysis.

CONCLUSION

BSDL is an extensible language for defining the basic testability features of a device implemented under the IEEE 1149.1-1990 standard. It is specifically designed for describing the numerous options that may be exercised in such implementations, in a way useful for humans and computers. It is also a *subset and standard practice* of IEEE Std 1076-1987 VHDL and as such may be contained within a larger VHDL description of a device used for modeling or simulation. An added benefit is that a number of compliance violations in a design may be discovered either in attempting to code the device features, or, in semantic checks possible during analysis.

Integrated circuit vendors have been reluctant[2] to embed user accessible testability features within their devices, and are now responding to market pressures for it. The 1149.1 standard makes it much easier to add testability in a prescribed way. However, without a simple, complete, and automated way of describing implementations, these vendors rightfully fear that new support difficulties will result. The concept of a standardized description offers them a way of transfering the support burden to the proper segments of the industry, most notably, the ATE vendors. These same ATE vendors will benefit from the assurance that the descriptions they receive are complete, accurate, and uniform across the IC vendors.

Very recently, a new interest has been expressed for BSDL. ASIC vendors could use a BSDL description of a device in conjunction with the description of its system logic to *automatically* add the Boundary-Scan logic during layout. This offers 1149.1 testability to their customers who may be unfamiliar with the details of the standard and, of course, the BSDL description is available immediately.

The advantages of the 1149.1 standard can be more widely enjoyed if there is some commonality in the description of Boundary-Scan devices across tasks and disciplines. We believe BSDL fills this need.

ACKNOWLEDGEMENTS

A large number of companies on several continents have been involved in drafting the 1149.1 standard. Many of these same companies have materially contributed to the development of BSDL in time and travel commitments. In particular we would like to mention AT&T, Bennetts Associates, British Aerospace, British Telecom, DEC, Electronic Tools, ElektronikCentralen, Ericsson, the ESPRIT Consortium, GenRad, Harris, IBM, ICL, Intel, Logic Automation, Marconi, Mentor Graphics, Motorola, NCR, Philips, Siemens, Teradyne, Texas Instruments, Thomson-CSF, and Unisys.

Hewlett-Packard sponsored a worldwide effort to gain consensus and gather comments. Meetings and presentations were held in Amsterdam, Bobligen, Boston, Carmel, Chicago, Dallas, Denver, Karlsruhe, London, Loveland, New York, Osaka, Paris, Philadelphia, Rome, Stanford, Tokyo, Vail and Zurich.

Special thanks go to the chairs of the IEEE 1149.1 working group, Colin Maunder and Rod Tulloss, who responded to scores of transmissions. Larry Saunders, chair of the IEEE Design Automation Standards Subcommittee was instrumental in the development of the VHDL subset.

The authors communicated with many people to great benefit. We wish to thank all of them, and in particular, Elmer Arment, Bill Armstrong, LaNae Avra, Keith Baker, Dave Ballew, Raymond Balzer, James Beausang, Bill Bell, Ben Bennetts, Leon Bentley, Harry Bleeker, Bill Bruce, Mike Bullock, Bill Den Beste, Bulent Dervisoglu, John Deshayes, Gary Dudeck, Lee Fleming, Peter Fleming, Michael Gallup, Vassilios Gerousis, Grady Giles, Luke Girard, Peter Hansen, Vance Harwood, Jay Hiserote, Najmi Jarwala, Doug Kostlan, Dirk van de Lagemaat, William Lattin, Johann Maierhofer, Ralph Marlett, Ken Mason, Mark Mathieu, Don McClean, Ed McCluskey, Randy Morgan, Carsten Mortensen, Roberto Mottola, Math Muris, Paul Ocampo, Dieter Ohnesorge, Anwar Osseyran, Michel Parot, Alain Plassart, Ken Posse, Jeff Rearick, Gordon Robinson, Rick Robinson, Martin Roche, Derek Roskell, Kevin Schofield, Dave Schuler, Rene Segers, Jay Stepleton, Mark Swanson, John Sweeney, Toshio Tamamura, Michael Tchou, Jake Thomas, Hai Vo-Ba, Rolf Wagner, Allen Warren, Ron Waxman, Lee Whetsel, Harry Whittemore, Tom Williams, Akira Yamagiwa, Chi Yau, and Mike Yeager.

REFERENCES

1. IEEE Standard 1149.1-1990, "IEEE Standard Test Access Port and Boundary-Scan Architecture," *IEEE Standards Board, 345 East 47th Street, New York, NY 10017,* May, 1990

2. Parker, K. P., "Testability: Barriers to Acceptance," *IEEE Design and Test of Computers, vol. 3,* October 1986, pp. 11-15

3. Parker, K. P., "The Impact of Boundary-Scan on Board Test," *IEEE Design and Test of Computers, vol. 6,* August 1989, pp. 18-30

4. Maunder, C. and F. Beenker, "Boundary-Scan: A Framework for Structured Design-for-Test", *Proc. Int'l Test Conference,* 1987, pp. 714-723

5. Hansen, P., "The Impact of Boundary-Scan on Board Test Strategies", *Proc. ATE&I Conference East, pp. 35-40,* Boston, June 1989

6. Private Communications: The authors have benefited from over 300 communications in person, by phone, facsimile, mail and E-mail with many individuals. See the acknowledgements.

7. IEEE Standard 1076-1987, "IEEE Standard VHDL Language Reference", *IEEE Standards Board, 345 East 47th Street, New York, NY 10017,* March, 1988

8. Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", *Proc. Intnl. Conf. on Information Processing,* UNESCO, 1959, pp 125-132

9. van de Lagemaat, D., "Testing Multiple Power Connections with Boundary-Scan", *Proc. Ist European Test Conference,* Paris, April 1989, pp. 127-130

10. Texas Instruments Data Sheet (Preliminary) SN54BCT8374, SN74BCT8374 Boundary-Scan Device with Octal D-Type Flip-Flop, Texas Instruments Inc, Dallas Tx. 1988

[Appendices start on the following page.]

Appendix A, An Example: This example is the Texas Instruments 74BCT8374 Octal D-Type Flip-Flop[10] (see Figure 6). This device has an unusually rich set of user defined instructions.

```
entity ttl74bct8374 is
  generic (PHYSICAL_PIN_MAP : string := "UNDEFINED");

  port (CLK:in bit;  Q:out bit_vector(1 to 8);  D:in bit_vector(1 to 8); GND, VCC:linkage bit;
        OC_NEG:in bit;  TDO:out bit;  TMS, TDI, TCK:in bit);
  use STD_1149_1_1990.all;    -- Get Std 1149.1-1990 attributes and definitions

  attribute PIN_MAP of ttl74bct8374 : entity is PHYSICAL_PIN_MAP;

  constant DW_PACKAGE : PIN_MAP_STRING := "CLK:1,  Q:(2,3,4,5,7,8,9,10),  D:(23,22,21,20,19,17,16,15)," &
                        "GND:6,  VCC:18,  OC_NEG:24,  TDO:11,  TMS:12, TCK:13,  TDI:14";

  constant FK_PACKAGE : PIN_MAP_STRING := "CLK:9,  Q:(10,11,12,13,16,17,18,19),  D:(6,5,4,3,2,27,26,25)," &
                        "GND:14,  VCC:28,  OC_NEG:7,  TDO:20,  TMS:21,  TCK:23,  TDI:24";

  attribute TAP_SCAN_IN    of TDI : signal is true;
  attribute TAP_SCAN_MODE  of TMS : signal is true;
  attribute TAP_SCAN_OUT of TDO : signal is true;
  attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

  attribute INSTRUCTION_LENGTH of ttl74bct8374 : entity is 8;

  attribute INSTRUCTION_OPCODE of ttl74bct8374 : entity is
   "BYPASS     (11111111, 10001000, 00000101, 10000100, 00000001),"  &
   "EXTEST     (00000000, 10000000),"  &
   "SAMPLE     (00000010, 10000010),"  &
   "INTEST     (00000011, 10000011),"  &
   "TRIBYP     (00000110, 10000110),"  &   -- Boundary Hi-Z
   "SETBYP     (00000111, 10000111),"  &   -- Boundary 1/0
   "RUNT       (00001001, 10001001),"  &   -- Boundary run test
   "READBN     (00001010, 10001010),"  &   -- Boundary read normal
   "READBT     (00001011, 10001011),"  &   -- Boundary read test
   "CELLTST    (00001100, 10001100),"  &   -- Boundary selftest normal
   "TOPHIP     (00001101, 10001101),"  &   -- Boundary toggle out test
   "SCANCN     (00001110, 10001110),"  &   -- BCR Scan normal
   "SCANCT     (00001111, 10001111),"  &   -- BCR Scan test

  attribute INSTRUCTION_CAPTURE of ttl74bct8374 : entity is "01010101";
  attribute INSTRUCTION_DISABLE of ttl74bct8374 : entity is "TRIBYP";

  attribute REGISTER_ACCESS of ttl74bct8374 : entity is
   "BOUNDARY (READBN, READBT, CELLTST),"  &
   "BYPASS (TOPHIP, SETBYP, RUNT, TRIBYP),"  &
   "BCR[2] (SCANCN, SCANCT)";  -- 2-bit Boundary Control Register

  attribute BOUNDARY_CELLS of ttl74bct8374 : entity is "BC_1";
  attribute BOUNDARY_LENGTH of ttl74bct8374 : entity is 18;

  attribute BOUNDARY_REGISTER of ttl74bct8374 : entity is
   -- num  cell     port     function  safe [ccell disval rslt]
    "17   (BC_1,  CLK,     input,    X),"   &
    "16   (BC_1,  OC_NEG,  input,    X),"   &  -- Merged Input/Control
    "16   (BC_1,  *,       control,  0),"   &  -- Merged Input/Control
    "15   (BC_1,  D(1),    input,    X),"   &
    "14   (BC_1,  D(2),    input,    X),"   &
    "13   (BC_1,  D(3),    input,    X),"   &
    "12   (BC_1,  D(4),    input,    X),"   &
    "11   (BC_1,  D(5),    input,    X),"   &
    "10   (BC_1,  D(6),    input,    X),"   &
    "9    (BC_1,  D(7),    input,    X),"   &
    "8    (BC_1,  D(8),    input,    X),"   &
    "7    (BC_1,  Q(1),    output3,  X,  16,  0,  Z),"  &
    "6    (BC_1,  Q(2),    output3,  X,  16,  0,  Z),"  &
    "5    (BC_1,  Q(3),    output3,  X,  16,  0,  Z),"  &
    "4    (BC_1,  Q(4),    output3,  X,  16,  0,  Z),"  &
    "3    (BC_1,  Q(5),    output3,  X,  16,  0,  Z),"  &
    "2    (BC_1,  Q(6),    output3,  X,  16,  0,  Z),"  &
    "1    (BC_1,  Q(7),    output3,  X,  16,  0,  Z),"  &  -- outputs controlled from cell 16 set to 0 are Hi-Z.
    "0    (BC_1,  Q(8),    output3,  X,  16,  0,  Z)";   -- cell 16 has a merged function, both input and control.

end ttl74bct8374;
```

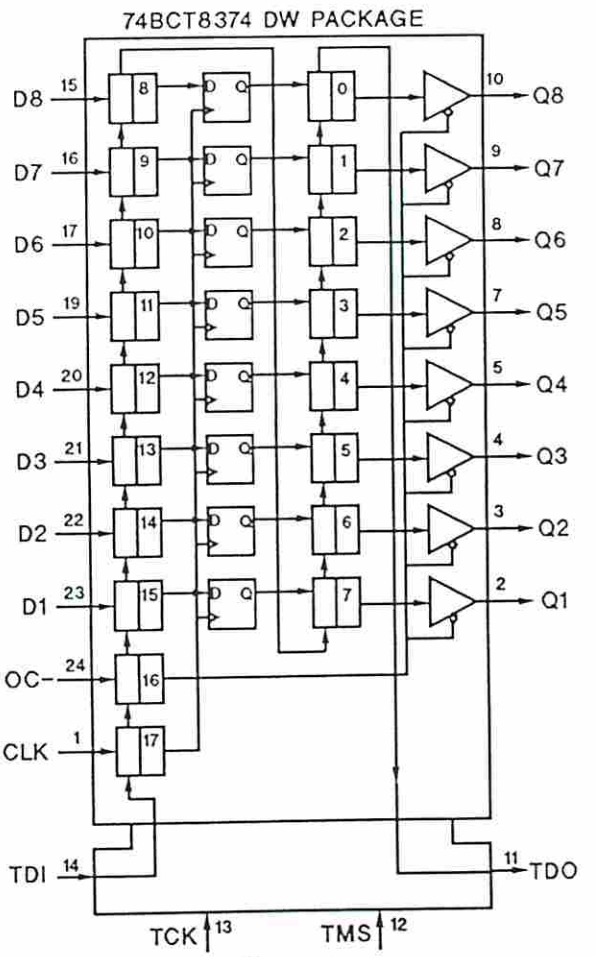


Figure 6

OC = 1 disable Q1-Q8
OC = 0 enable Q1-Q8

Appendix B, IEEE Std 1149.1-1990 VHDL Package/Package Body definition.

This is the definition of the VHDL package and supporting package body for IEEE Std 1149.1-1990 attributes, types, subtypes, and constants of BSDL.

```
package STD_1149_1_1990 is

-- Give pin mapping declarations
attribute PIN_MAP : string;
subtype PIN_MAP_STRING is string;

-- Give TAP control declarations
type CLOCK_LEVEL is (LOW, BOTH);
type CLOCK_INFO  is record
  FREQ : real;
  LEVEL: CLOCK_LEVEL;
end record;

attribute TAP_SCAN_IN   : boolean;
attribute TAP_SCAN_OUT  : boolean;
attribute TAP_SCAN_CLOCK  : CLOCK_INFO;
attribute TAP_SCAN_MODE   : boolean;
attribute TAP_SCAN_RESET  : boolean;

-- Give instruction register declarations
attribute INSTRUCTION_LENGTH : integer;
attribute INSTRUCTION_OPCODE : string;
attribute INSTRUCTION_CAPTURE : string;
attribute INSTRUCTION_DISABLE : string;
attribute INSTRUCTION_PRIVATE : string;
attribute INSTRUCTION_USAGE : string;

-- Give ID and USER code declarations
type ID_BITS is ('0', '1', 'x', 'X');
type ID_STRING is array (31 downto 0) of ID_BITS;
attribute IDCODE_REGISTER  :  ID_STRING;
attribute USERCODE_REGISTER  :  ID_STRING;

-- Give register declarations
attribute REGISTER_ACCESS : string;

-- Give boundary cell declarations
type BSCAN_INST is (EXTEST, SAMPLE, INTEST,
                    RUNBIST);
type CELL_TYPE is (INPUT, INTERNAL, CLOCK,
                   CONTROL, CONTROLR, OUTPUT2,
                   OUTPUT3, BIDIR_IN, BIDIR_OUT);
type CAP_DATA is (PI, PO, UPD, CAP, X, ZERO, ONE);
type CELL_DATA is record
  CT : CELL_TYPE;
  I   : BSCAN_INST;
  CD : CAP_DATA;
end record;
type CELL_INFO is array of CELL_DATA;

-- Boundary Cell defered constants (see package body)

constant BC_1  : CELL_INFO;
constant BC_2  : CELL_INFO;
constant BC_3  : CELL_INFO;
constant BC_4  : CELL_INFO;
constant BC_5  : CELL_INFO;
constant BC_6  : CELL_INFO;

-- Boundary Register declarations

attribute BOUNDARY_CELLS : string;
attribute BOUNDARY_LENGTH : integer;
attribute BOUNDARY_REGISTER : string;

-- Miscellaneous
attribute DESIGN_WARNING : string;

end STD_1149_1_1990;  -- End of 1149.1-1990 Package
```

```
package body STD_1149_1_1990 is  -- Standard Boundary Cells

-- Description for f10-12, f10-16, f10-18c, f10-18d, f10-21c

constant BC_1 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),      (OUTPUT2, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),      (OUTPUT2, SAMPLE,  PI),
  (INPUT, INTEST,  PI),      (OUTPUT2, INTEST,  PI),
  (INPUT, RUNBIST, PI),      (OUTPUT2, RUNBIST, PI),
  (OUTPUT3, EXTEST,  PI),    (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),    (INTERNAL, INTEST,  PI),
  (OUTPUT3, RUNBIST, PI),    (INTERNAL, RUNBIST, PI),
  (CONTROL, EXTEST,  PI),    (CONTROLR, EXTEST,  PI),
  (CONTROL, SAMPLE,  PI),    (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),    (CONTROLR, INTEST,  PI),
  (CONTROL, RUNBIST, PI),    (CONTROLR, RUNBIST, PI) );

-- Description for f10-8, f10-17, f10-19c, f10-19d, f10-22c

constant BC_2 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),      (OUTPUT2, EXTEST,  UPD),
  (INPUT, SAMPLE,  PI),      (OUTPUT2, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),  -- Intest on output2 not supported
  (INPUT, RUNBIST, UPD),     (OUTPUT2, RUNBIST, UPD),
  (OUTPUT3, EXTEST,  UPD), (INTERNAL, EXTEST,  PI),
  (OUTPUT3, SAMPLE,  PI),    (INTERNAL, SAMPLE,  PI),
  (OUTPUT3, INTEST,  PI),    (INTERNAL, INTEST,  UPD),
  (OUTPUT3, RUNBIST, PI),    (INTERNAL, RUNBIST, UPD),
  (CONTROL, EXTEST,  UPD), (CONTROLR, EXTEST,  UPD),
  (CONTROL, SAMPLE,  PI),    (CONTROLR, SAMPLE,  PI),
  (CONTROL, INTEST,  PI),    (CONTROLR, INTEST,  PI),
  (CONTROL, RUNBIST, PI),    (CONTROLR, RUNBIST, PI) );

-- Description for f10-9

constant BC_3 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),      (INTERNAL, EXTEST, PI),
  (INPUT, SAMPLE,  PI),      (INTERNAL, SAMPLE, PI),
  (INPUT, INTEST,  PI),      (INTERNAL, INTEST,  PI),
  (INPUT, RUNBIST, PI),      (INTERNAL, RUNBIST, PI) );

-- Description for f10-10, f10-11

constant BC_4 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),  -- Intest on input not supported
  (INPUT, SAMPLE,  PI),  -- Runbist on input not supported
  (CLOCK, EXTEST,  PI),   (INTERNAL, EXTEST,  PI),
  (CLOCK, SAMPLE,  PI),   (INTERNAL, SAMPLE,  PI),
  (CLOCK, INTEST,  PI),   (INTERNAL, INTEST,  PI),
  (CLOCK, RUNBIST, PI),   (INTERNAL, RUNBIST, PI) );

-- Description for f10-20c, a combined Input/Control

constant BC_5 : CELL_INFO :=
 ((INPUT, EXTEST,  PI),      (CONTROL, EXTEST,  PI),
  (INPUT, SAMPLE,  PI),      (CONTROL, SAMPLE,  PI),
  (INPUT, INTEST,  UPD),     (CONTROL, INTEST,  UPD),
  (INPUT, RUNBIST, PI),      (CONTROL, RUNBIST, PI) );

-- Description for f10-22d, a reversible cell

constant BC_6 : CELL_INFO :=
 ((BIDIR_IN, EXTEST,  PI),    (BIDIR_OUT, EXTEST, UPD),
  (BIDIR_IN, SAMPLE,  PI),    (BIDIR_OUT, SAMPLE, PI),
  (BIDIR_IN, INTEST,  UPD),   (BIDIR_OUT, INTEST,  PI),
  (BIDIR_IN, RUNBIST, UPD),   (BIDIR_OUT, RUNBIST, PI) );

end STD_1149_1_1990;  -- End of 1149.1-1990 Package Body
```

Appendix C, BSDL Syntax Specification

The BNF syntax descriptions are shown in this appendix. The items described are those contained within VHDL strings and as such, are not part of VHDL syntax. Syntactic items are shown in italics, surrounded by '<' and '>' characters. Keywords such as 'Extest' or 'Output3' are in normal font. Boldface items are BSDL terminals such as **INTEGER**, **VHDL IDENTIFIER**, or other description. The symbol **NULL** is the empty expansion. All BSDL elements contained within VHDL strings are treated as single, contiguous strings even though they may be expressed as the concatenation of smaller strings. All concatenations should be removed during lexicographical analysis. An asterisk in the leftmost column marks the start of an BNF expression promised in the text of this paper.

* *<MapString>* ::= *<PinMapping>* | *<MapString>* , *<PinMapping>*

<PinMapping> ::= *<PortName>* : *<PhysicalPinDesc>*

<PortName> ::= **VHDL IDENTIFIER**

<PhysicalPinDesc> ::= *<PhysicalPin>* | (*<PhysicalPinList>*)

<PhysicalPinList> ::= *<PhysicalPin>* | *<PhysicalPinList>* , *<PhysicalPin>*

<PhysicalPin> ::= **INTEGER** | **VHDL IDENTIFIER**

* *<OpcodeTable>* ::= *<OpcodeDesc>* | *<OpcodeTable>* , *<OpcodeDesc>*

<OpcodeDesc> ::= *<OpcodeName>* (*<PatternList>*)

<PatternList> ::= *<Pattern>* | *<PatternList>* , *<Pattern>*

* *<Pattern>* ::= **BINARY STRING**

* *<UsageString>* ::= *<UsageDesc>* | *<UsageString>* , *<UsageDesc>*

<UsageDesc> ::= *<OpcodeName>* (*<UsageList>*)

* *<OpcodeName>* ::= Extest | Sample | Intest | Runbist | **VHDL IDENTIFIER**

<UsageList> ::= *<Usage>* | *<UsageList>* ; *<Usage>*

<Usage> ::= *<RegisterDecl>* | *<InitializeDecl>* | *<ShiftDecl>* | *<ResultDecl>* | *<ClockDecl>* | *<LengthDecl>*

<RegisterDecl> ::= Registers *<RegisterList>*

<RegisterList> ::= *<Register>* | *<RegisterList>* , *<Register>*

<Register> ::= **VHDL IDENTIFIER**

<InitializeDecl> ::= Initialize *<Register>* *<Pattern>*

<ShiftDecl> ::= Shift *<Register>*

<ResultDecl> ::= Result *<Pattern>*

<LengthDecl> ::= Length *<LengthSpec>*

<LengthSpec> ::= **INTEGER** cycles | **REAL** seconds

<ClockDecl> ::= Clock **VHDL IDENTIFIER** *<ClockSpec>*

<ClockSpec> ::= in *<TapState>* | shifted | **NULL**

<TapState> ::= Run_Test_Idle

* *<RegisterString>* ::= *<RegisterAssoc>* | *<RegisterString>* , *<RegisterAssoc>*

<RegisterAssoc> ::= *<Register>* (*<OpcodeList>*)

* *<OpcodeList>* ::= *<OpcodeName>* | *<OpcodeList>* , *<OpcodeName>*

* *<CellList>* ::= *<CellName>* | *<CellList>* , *<CellName>*

<CellName> ::= **VHDL IDENTIFIER**

* *<CellTable>* ::= *<CellEntry>* | *<CellTable>* , *<CellEntry>*

<CellEntry> ::= *<CellNumber>* (*<CellInfo>*)

<CellInfo> ::= *<CellSpec>* | *<CellSpec>* , *<DisableSpec>*

<CellSpec> ::= *<CellID>* , *<PortID>* , *<Function>* , *<SafeValue>*

<CellNumber> ::= **INTEGER**

<CellID> ::= **VHDL IDENTIFIER**

<PortID> ::= *<PortName>* | *

<Function> ::= Input | Output2 | Output3 | Control | Controlr | Internal | Clock | Bidir

<SafeValue> ::= 0 | 1 | X

<DisableSpec> ::= *<DisableCell>* , *<DisableVal>* , *<DisableResult>*

<DisableCell> ::= *<CellNumber>*

<DisableVal> ::= 0 | 1

<DisableResult> ::= Z | Weak0 | Weak1

B A Step-by-step Demonstration

A Demonstration of the BOLD System

Jung-Cheun Lien

Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-0781

June 27, 1991

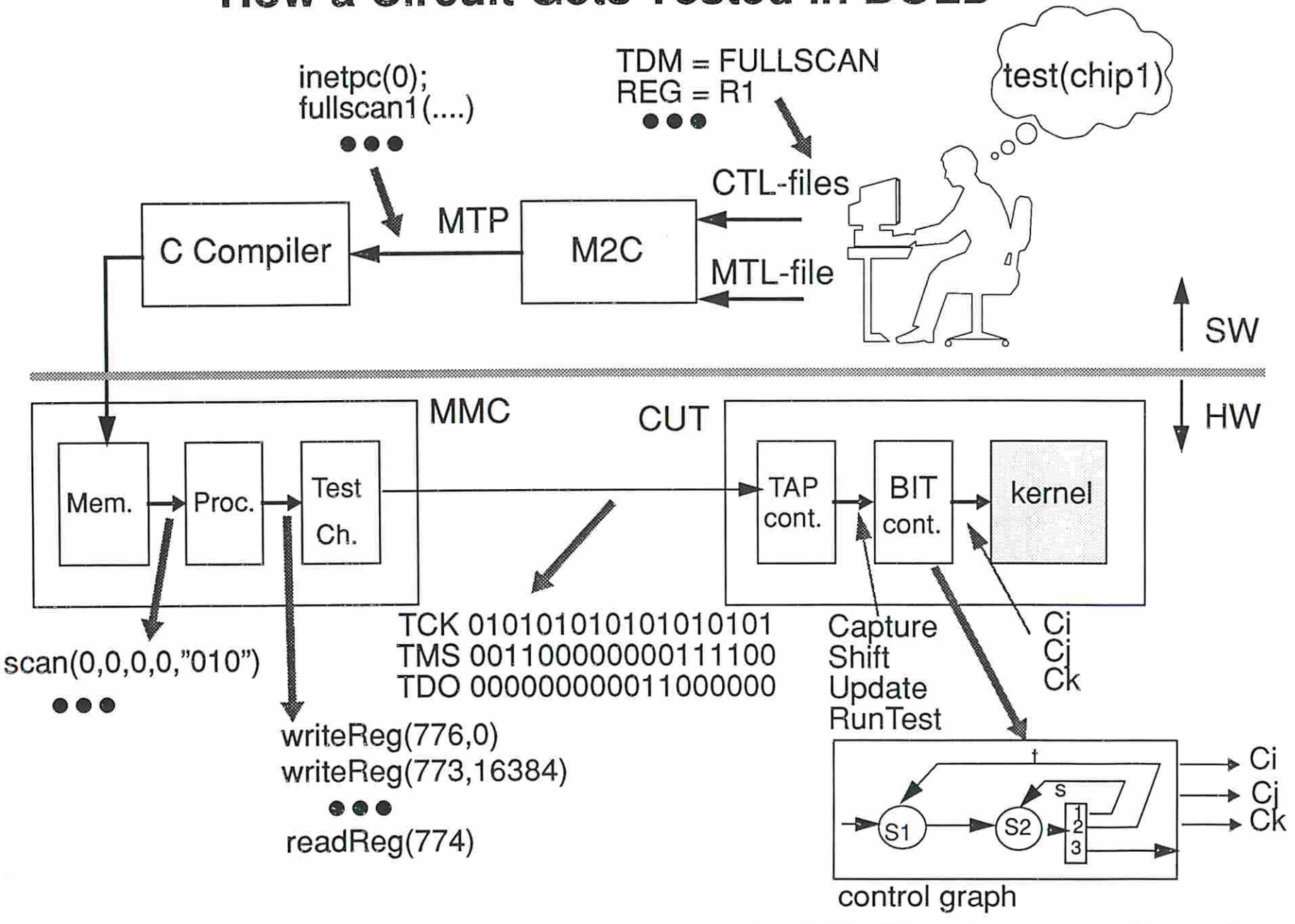
Objective: Demonstrate the capability of the BOLD system.

System Design Methodology: test controllers and test busses.

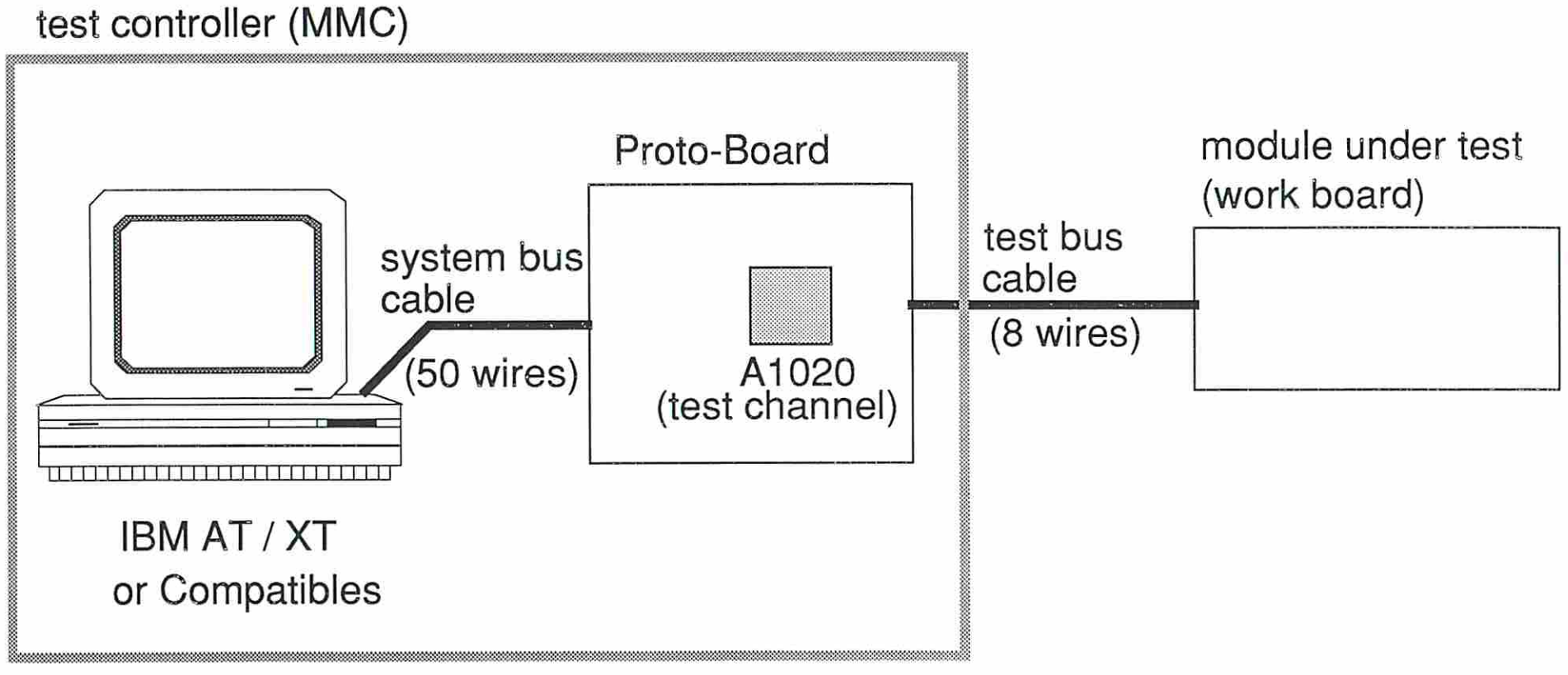
What the BOLD can do:

1. Provide chip and module design and test methodology.
2. Provide module test controller (MMC).
3. Provide Chip Test Language and Module Test Language.
4. Provide test program synthesizer M2C.

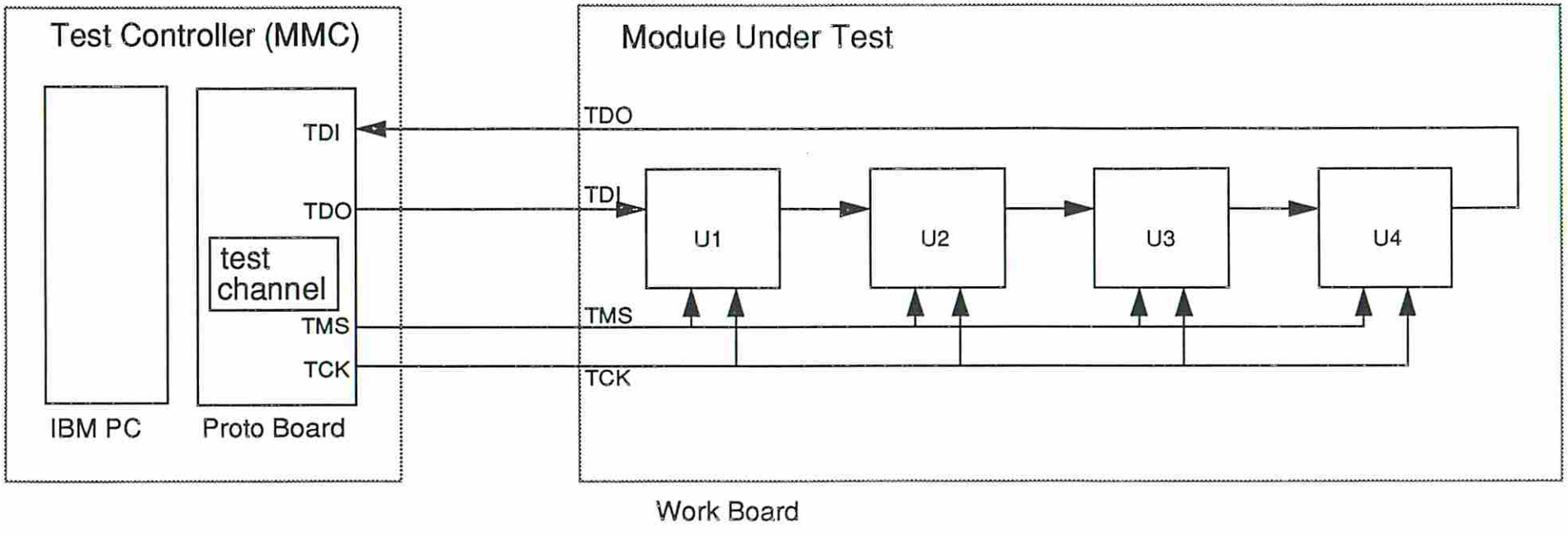
How a Circuit Gets Tested in BOLD



Hardware Components of the Demonstration



Schematic of the Demo System Hardware



System Set Up for the Demonstration

System Requirements:

1. An IBM PC or compatible.
2. Microsoft C compiler.
3. An available I/O bus slot.
4. A link from the PC to the SUN workstation. (Either a direct RS232 line or a modem can be used.)
5. A communication program such as Kermit for transferring data from SUN to the PC.

Hardware Set-up Procedure:

The hardware used in this demonstration includes: an IBM PC/AT and three boards, referred to as Host Board, Proto Board and work Board.

1. Insert the Host Board into a bus I/O slot of the IBM AT.
   (Side A of the Host Board must be connected to Side A of the bus slot.).

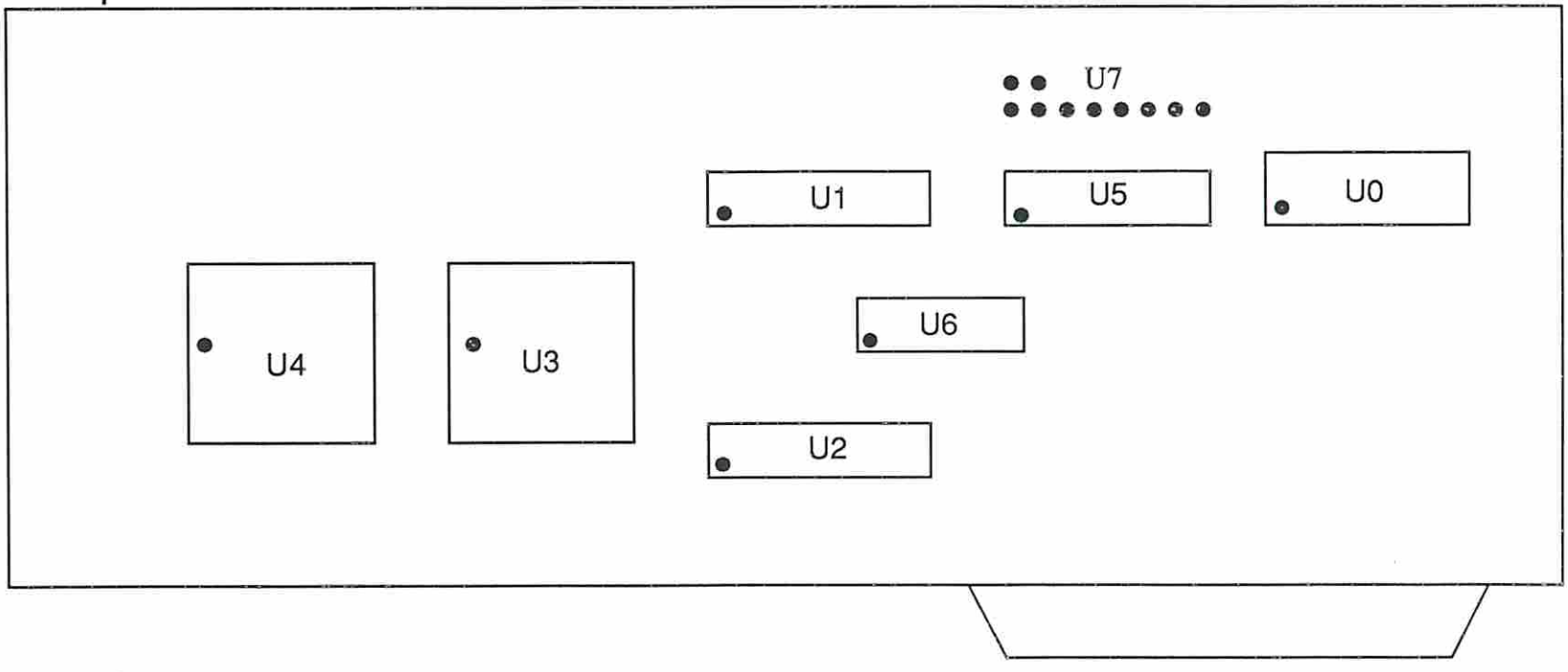
2. Connect the Host Board and the Proto Board using a flat cable.
   (The connection is easy since only the correct connection can be made.)

3. Connect the work Board to the Proto-Board using an 8-wire flat cable.
   (The pin next to the red dot on the Proto Board must be connected to the pin next to the red dot of the work Board.)

4. **Double check the connection.**
   Turning on the power of the Proto Board. The system set up is completed.

Physical View of the Work Board

Component Side

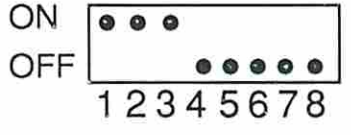


Components

U0: Switch
U1: 74BCT8244 (TI8244)
U2: 74BCT8244
U3: A1010 (app1)
U4: A1010
U5: 74LS244
U6: 74LS00
U7: Connector

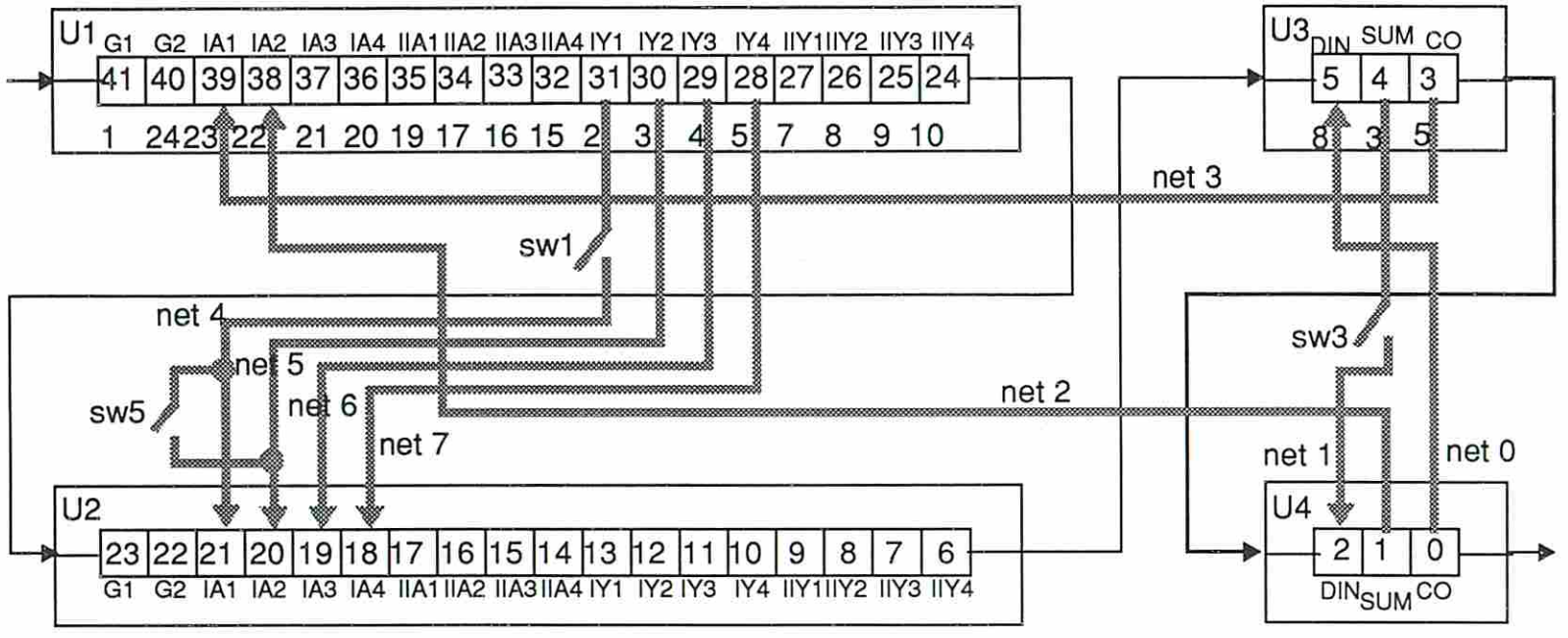
U7 pin names

U0 normal setting



Note: sw1 and sw5 cannot both be on.

Testing Interconnects of the Demo Board

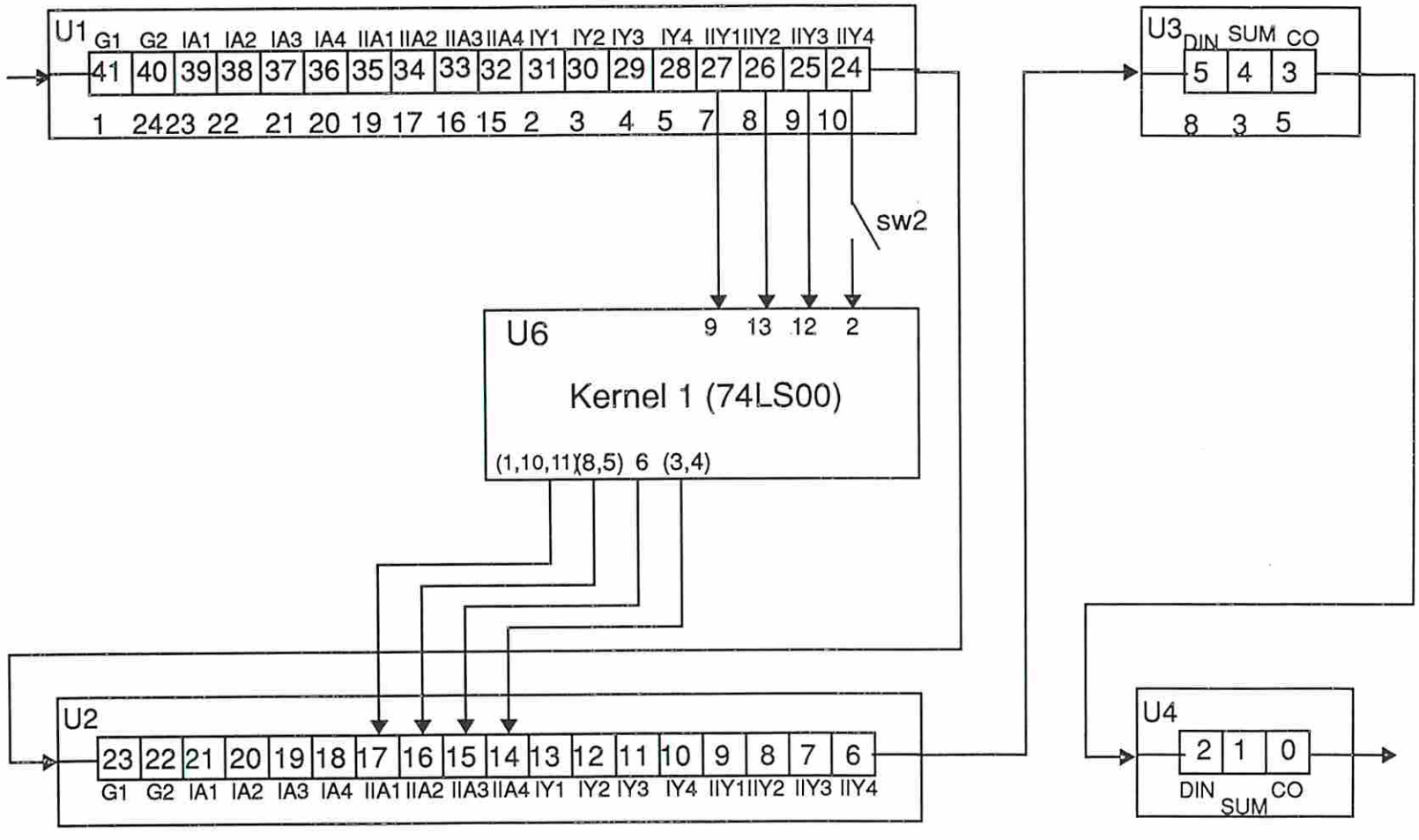


List of interconnects

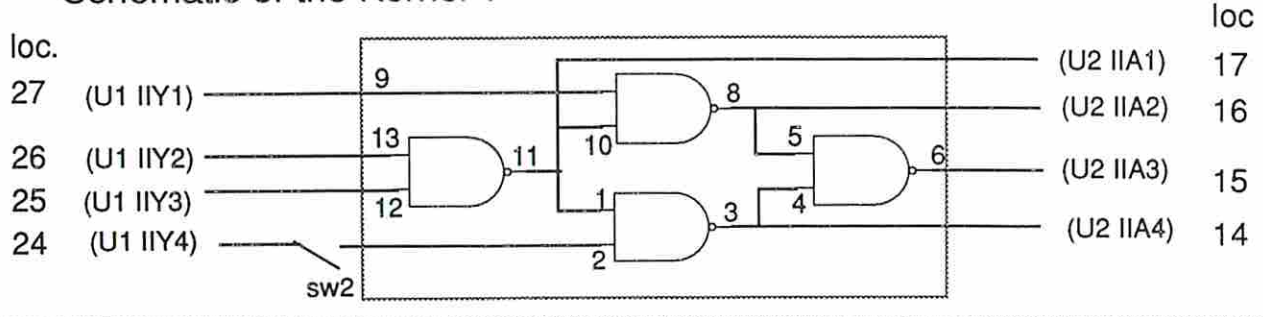
net	driver	location	receiver	location
0	(U4 CO)	0	(U3 DIN)	5
1	(U3 SUM)	4	(U4 DIN)	2
2	(U4 SUM)	1	(U1 IA2)	38
3	(U3 CO)	3	(U1 IA1)	39
4	(U1 IY1)	31	(U2 IA1)	21
5	(U1 IY2)	30	(U2 IA2)	20
6	(U1 IY3)	29	(U2 IA3)	19
7	(U1 IY4)	28	(U2 IA4)	18

Fault Injection:
1. sw3 off ---> open net 1.
2. sw1 off ---> open net 4.
3. sw1 off and sw5 on ---> short net 4 and 5.

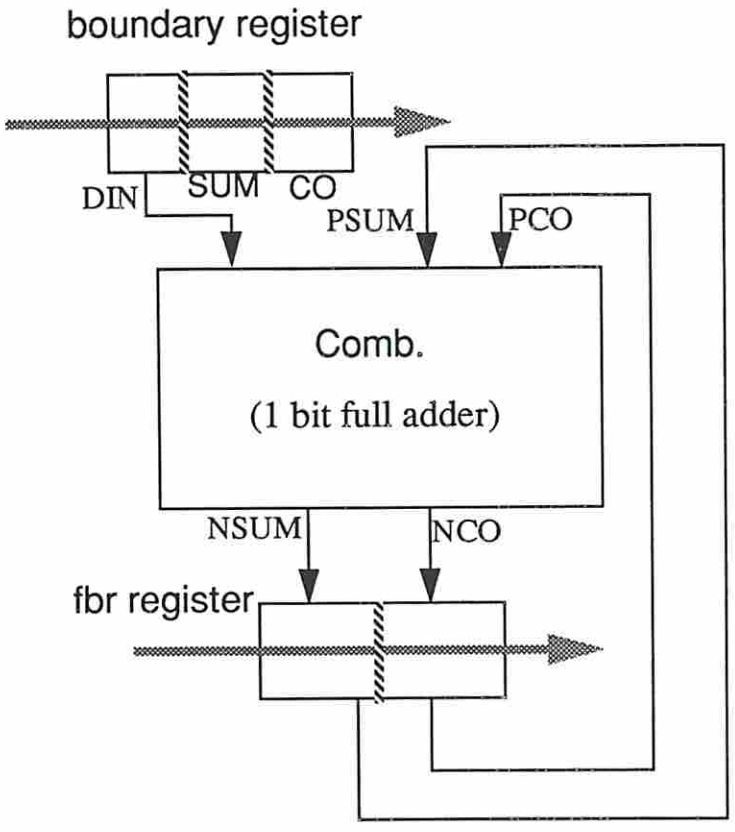
Testing Kernel 1 of the Demo Board



Schematic of the Kernel 1



The App1 Boundary Scan Chip



TDM decsription:

TDM 0 = FULLSCAN;
REG = FBR, VECFILE=AP1_IN2, RESFILE=AP1_OUT2;
REG = Boundary, VECFILE=AP1_IN1, RESFILE=AP1_OUT1;
CLOCK = FCK 1.0 cycles_in Run_Test_IDLE;

Contents of the files

AP1_IN1	AP1_OUT1	AP1_IN2	AP1_OUT2
8 3		8 2	
0 0 0	2 2 2	0 0	0 0
1 0 0	2 2 2	0 1	1 0
0 1 0	2 2 2	1 0	1 0
1 1 0	2 2 2	1 1	0 1
0 0 1	2 2 2	0 0	1 0
1 0 1	2 2 2	0 1	0 1
0 1 1	2 2 2	1 0	0 1
1 1 1	2 2 2	1 1	1 1

The digit 2 represents a don't care value.

For detailed description please see USC Technical Report No. CENG 90-14 (by Jung-Cheun Lien).

Test Program Synthesis Procedure

At the SUN:

1. Prepare the MTL description, the CTL description and the associated test vectors and results files.
   These files include: **work.mtl**, app1.ctl, TI8244.ctl, AP1_IN1, AP1_OUT1, AP1_IN2, AP1_OUT2, kernel1.vec and kernel1.res.
2. Synthesize module test programs using the command: %**M2C work** <cr>.
3. Generate test programs in C using the command: %**genTarget work** <cr>.
   Drectory workDir now contains the files: workmain.c comp.h kernel1.c inetpc.c template.c driver.c and infofile.net.
   Note that the files comp.h, inetpc.c, template.c and driver.c remain unchanged for testing different modules.

At the IBM PC:

4. Transfer these files from the SUN to the IBM PC using Kermit.
   In addition, the test vectors and results files should also be transferred to the PC.
5. Compile the test program using the microsoft C in the PC. The command used is: **make work.mak**<cr>.
6. The final executable test program is in **work.exe**.
7. To execute the test program type:**work**<cr>.

Test Program Execution Results

Test Session 1

Conditions:

Switch (U0) setting: on:1,2,3; off:4,5,6,78.

No fault exists in Demo Board.

Responses when type in **work**<cr>.

Testing interconnect...
Completed with no fault.
Diagnosing interconnect...
Completed with no fault.
Testing kernel 1...
Kernel1 tested OK.
Testing U3 (app1) in fullscanN...
chip=U3 tested OK.
Testing U4 (app1) in fullscanN....
chip=U4 tested OK.

Test Session 2

Conditions:

Switch(U0) setting: on:1,2; off:3,4,5,6,7,8.
The fault "net 1 open" is injected.

Responses when type in **work**<cr>.

Testing interconnect...
err: net=1, drv val=1, 0th rcv val=0, (rid=0,loc=2).
test failed in 1th vector, errs in 1 rcvs.
1 faults detected.
Diagnosing interconnect...
sa0::net 1, drv:rid=0,loc=4,rcv:rid=0,loc=2
w1:open: net 1, drv:rid=0,loc=4,rcv:rid=0,loc=2
2 faults detected.
Testing kernel 1...
Kernel1 tested OK.
Testing U3 (app1) in fullscanN...
chip=U3 tested OK.
Testing U4 (app1) in fullscanN....
chip=U4 tested OK.

Test Session 3

Conditions:

Switch(U0) setting: on:2,3; off:1,4,5,6,7,8.
The fault "net 4 open" is injected.

Responses when type in **work**<cr>.

Testing interconnect...
err:net=4, drv val=0, 0th rcv val=1,(rdi=0,loc=21).
test failed in 1th vector, errs in 1 rcvs.
err:net=4,drv val=0, 0th rcv val=1,(rid=0,loc=21).
test failed in 3th vector, errs in 1 rcvs.
2 faults detected.
Diagnosing interconnect...
sa1::net 4, drv:rid=0, loc=31, rcv:rid=0, loc=21
w0: open: net 4, drv:rid=0, loc=31, rcv:rid=0, loc=21
2 faults detected
Testing kernel 1...
Kernel1 tested OK.
Testing U3 (app1) in fullscanN...
chip=U3 tested OK.
Testing U4 (app1) in fullscanN....
chip=U4 tested OK.

Test Program Execution Results (Cont.)

Test Session 4

Conditions:

Switch(U0) setting: on:2,3,5; off:1,4,6,7,8.
The fault "net4 open" and "net 4 and 5 short" are injected..

Responses when type in **work**<cr>.

```
Testing interconnect...
err: net=4, drv val=1, 0th rcv val=0, (rid=0,loc=21).
test failed in 0th vector, errs in 1 rcvs.
err: net=4, drv val=1, 0th rcv val=0, (rid=0,loc=21).
test failed in 1th vector, errs in 1 rcvs.
2 faults detected.
Diagnosing interconnect...
w1:open: net 4, drv:rid=0,loc=31,rcv:rid=0,loc=21
w1:short:(net=5,net=4), D:rid=0,loc=30,R:rid=0,loc=20
2 faults detected.
Testing kernel 1...
Kernel1 tested OK.
Testing U3 (app1) in fullscanN...
chip=U3 tested OK.
Testing U4 (app1) in fullscanN....
chip=U4 tested OK.
```

Test Session 5

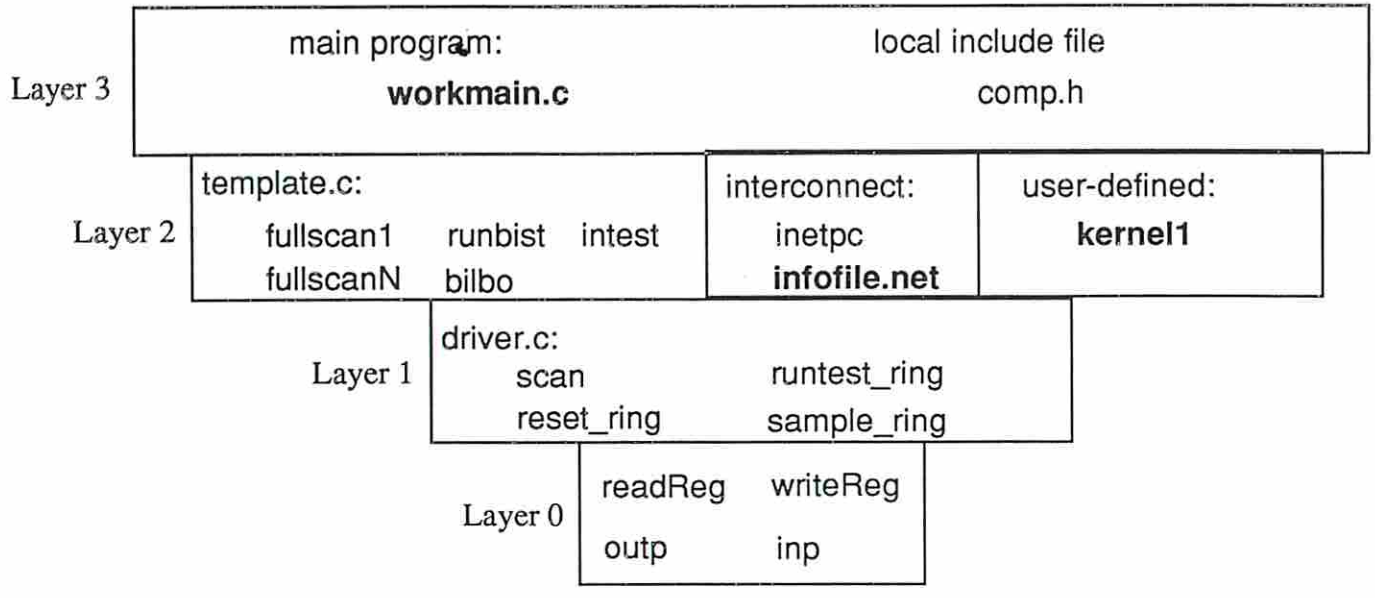
Conditions:

Switch(U0) setting: on:1,3; off:2,4,5,6,7,8.
An input open fault is injected to kernel 1...

Responses when type in **work**<cr>.

```
Testing interconnect...
Completed with no fault.
Diagnosing interconnect...
Completed with no fault.
Test kernel 1...
err:v0,
     outs=000000000000000000000000000000000000000000
    exps=22222222222221011222222222222222222222222
    ins =00000000000000011100000000000001110000000
continue?(1:yes) :1<cr>
err:v1,
    outs=00000000000000000000000000100000000000000
    exps=22222222222221101222222222222222222222222
    ins = 00000000000000010100000000000001110000000
continue?(1:yes) :0<cr>
...
aborted.
```

Software Hierarchy of the Generated Module Test Programs



Software/Hardware Dependency Matrix

	MMC processor	Test channel	Test channel interface	C compiler
Layer 3	No	No	No	No
Layer 2	No	No	No	No
Layer 1	No	Yes	No	Yes
Layer 0	Yes	Yes	Yes	Yes

Syntax-Directed Translation in M2C

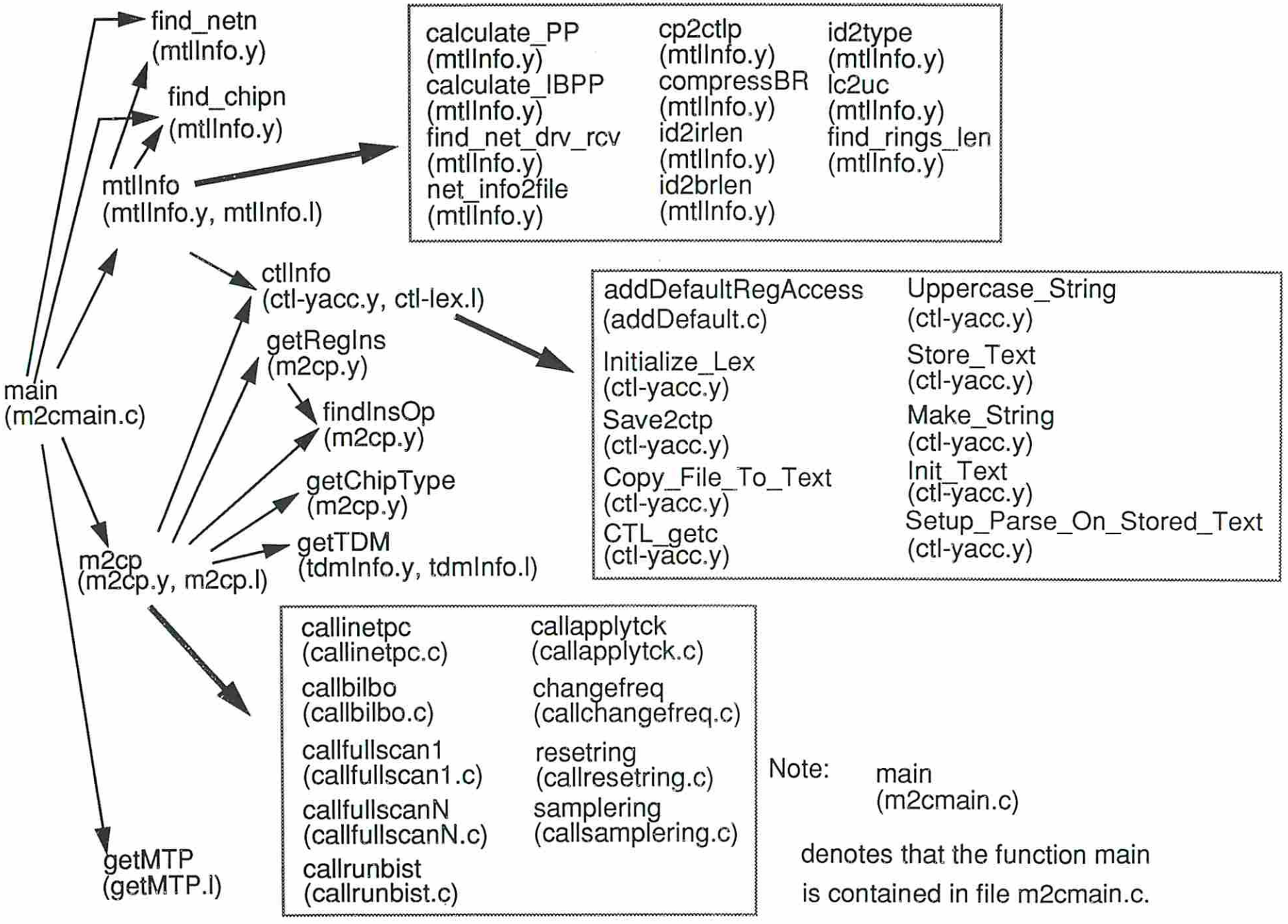
In the MT-file (**work.mtl**)

```
main(){
① testinet();
② diagnosisinet();
③ kernel1();
④ testchip(U3);
⑤ testchip(U4);
}
```

In the module test program (**workmain.c**)

```
main(){

inetpc(1);

inetpc(0);

kernel1();

sprintf("chipID, "%s", "U3");
sprintf(chipType, "%s", "app1");
sprintf(regIns, "%s", "110000");
sprintf(vecFID, "%s", "AP1_IN2,AP1_IN1");
sprintf(expFID, "%s", "AP1_OUT2,AP1_OUT1");
fullscanN(chipID,chipType,0,3,16,1,2,2,regIns,vecFID,expFID);

sprintf("chipID, "%s", "U4");
sprintf(chipType, "%s", "app1");
sprintf(regIns, "%s", "110000");
sprintf(vecFID, "%s", "AP1_IN2,AP1_IN1");
sprintf(expFID, "%s", "AP1_OUT2,AP1_OUT1");
fullscanN(chipID,chipType,0,0,19,0,3,2,regIns,vecFID,expFID);

}
```

Function Dependeny Tree of M2C



C Input Descriptions for the Demonstration

```
MODULE = module_1;
LIB  = lib1(2);
DEVICE_LIST = (U1 TI8244) (U2 TI8244) (U3 app1) (U4 app1);
TEST_BUS =
  RING 0: U1 => U2 => U3 => U4;
NET_LIST =
  NET 1: (U3 DIN) (U4 CO),
  NET 2: (U3 SUM) (U4 DIN),
  NET 3: (U4 SUM) (U1 IA2),
  NET 4: (U3 CO) (U1 IA1),
  NET 5: (U1 IY1) (U2 IA1),
  NET 6: (U1 IY2) (U2 IA2),
  NET 7: (U1 IY3) (U2 IA3),
  NET 8: (U1 IY4) (U2 IA4);
MODULE_TEST
#include <stdio.h>
#include <string.h>
#include "comp.h"
main()
{
  testinet();
  diagnosisinet();
  kernel1();
  testchip(U3);
  testchip(U4);
}
int kernel1()   /* Test non-boundary scan device using EXTEST */
{
char outs[50], exps[50], ins[50];
int i, j, ans, x, faults=0;
FILE *kvecF, *kresF;
extern char *scan(); /*scan(rid,IR,ipre,ipost,optr);*/

  if((kvecF=fopen("kernel1.vec", "r"))==NULL){
    printf("file kernel1.vec doesn't exist.\n");
    exit(2);
  }
  if((kresF=fopen("kernel1.res","r"))==NULL){
    printf("file kernel1.res doesn't exits.\n");
    exit(2);
  }
  sprintf(outs,"%s","000000000000000000000000");
  scan(0,1,0,0,outs);   /* set all chips to EXTEST, irlen=24. */
  printf("Testing kernel 1 ... \n");
  for(i=0; i<16; i++){ /* for each test vector */
    for(j=0; j<42; j++){ /* the bs chain is 42 bits long */
      fscanf(kvecF, "%d", &x);
      if(x==1) outs[j]='1'; else outs[j]='0';
      fscanf(kresF, "%d", &x);
      if(x==1){exps[j]='1';} else{if(x==0) exps[j]='0';else exps[j]='2';}
    } /* for j */
    outs[j]='\0'; /* now the output string is ready */
    exps[j]='\0';
    scan(0,0,0,0,outs);             /* send vector */
    strcpy(ins, scan(0,0,0,0,outs));/* get result */
    /*compare exps with ins */
    for(j=0; j<42; j++){if((exps[j]!='2') && (exps[j]!=ins[j])) break;}
    if(j<42) {
      faults++;
      printf("err:v%d,\n outs=%s,\n exps=%s,\n ins =%s\n",i,outs,exps,ins);
      printf("continue?(1:yes) :");
      scanf("%d",&ans); printf("  ... \n");
      if(ans!=1) {printf("aborted.\n"); exit(2);}
    } /* if */
  } /* for i */
  fclose(kvecF); fclose(kresF);
  if (faults!=0) printf("Faults detected in kernel1.\n");
  else printf("Kernel1 tested OK.\n");
}
END_TEST
```

```
-- CTL description of App1, Lien 6/20/91
entity app1 is
  generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");

  port (DIN: in bit; CO, SUM:out bit; VCC, GND:linkage bit);

  use STD_1149_1_1990.all;   -- Get Std 1149.1-1990 attributes and definitions

  attribute PIN_MAP of app1: entity is PHYSICAL_PIN_MAP;

  constant DW_PACKAGE:PIN_MAP_STRING:="RESET:48, DIN:8, CO:5, SUM:3, " &
         "TDI:45, TDO:44, TMS:46, TCK:52, TRST:47, VCC:25, GND:49";

  attribute TAP_SCAN_IN    of TDI : signal is true;
  attribute TAP_SCAN_MODE  of TMS : signal is true;
  attribute TAP_SCAN_OUT   of TDO : signal is true;
  attribute TAP_SCAN_CLOCK of TCK : signal is (2.0e6, BOTH);

  attribute INSTRUCTION_LENGTH of app1 : entity is 3;

  attribute INSTRUCTION_OPCODE of app1 : entity is
       "BYPASS (001, 101, 011, 111),"   &
       "EXTEST (000),"  &
       "INTEST (100),"  &
       "SAMPLE (010),"  &
       "SCANFB (110)";

  attribute INSTRUCTION_CAPTURE of app1 : entity is "101";
-- attribute INSTRUCTION_DISABLE of app1 : entity is "TRIBPY";

  attribute REGISTER_ACCESS of app1 : entity is
--implicit       "BOUNDARY (EXTEST, INTEST, SAMPLE),"   &
--implicit       "BYPASS (BYPASS),"   &
       "FBR[2] (SCANFB)";   -- 2-bit FeedBack Register

  attribute BOUNDARY_CELLS of app1 : entity is "BC_2";
  attribute BOUNDARY_LENGTH of app1 : entity is 3;

  attribute BOUNDARY_REGISTER of app1 : entity is
      -- num cell  port    function safe [ccell disval rslt]
        "2  (BC_2, DIN,    input, X)," &
        "1  (BC_2, SUM,    output2, X), "&
        "0  (BC_2, CO,    output2, X)";

  attribute TEST_PROC of app1 : entity is
        "Test_Begin" &
        "TDM 0 = FULLSCAN;" &
        "REG=FBR, VECFILE=ap1_in2, RESFILE=ap1_out2;" &
        "REG=BOUNDARY, VECFILE=ap1_in1, RESFILE=ap1_out1;" &
        "CLOCK = FCK 1.0 CYCLES_IN RUN_TEST_IDLE;" &
        "Test_End ";
end app1;
```

```
-- CTL description of TI 74bct8244 Octal D Flip-Flop, Lien 6/23/91
entity ttl74bct8244 is
  generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");
  port(G1, G2: in bit; IA1, IA2, IA3, IA4: in bit;
       IIA1, IIA2, IIA3, IIA4: in bit;
       IY1, IY2, IY3, IY4, IIY1, IIY2, IIY3, IIY4: in bit;
       GND, VCC: linkage bit;
       TDO:out bit; TMS, TDI, TCK:in bit);
  use STD_1149_1_1990.all;   -- Get Std 1149.1-1990 attributes and definitions

  attribute PIN_MAP of ttl74bct8244 : entity is PHYSICAL_PIN_MAP;

  constant DW_PACKAGE:PIN_MAP_STRING:="G1:1, G2:24, IA1:23, IA2:22, "&
       "IA3:21, IA4:20, IIA1:19, IIA2:17, IIA3:16, IIA4:15, "&
       "IY1:2, IY2:3, IY3:4, IY4:5, IIY1:7, IIY2:8, IIY3:9, IIY4:10, "&
       "GND:6, VCC:18, TDO:11, TMS:12, TCK:13, TDI:14";

  constant FK_PACKAGE:PIN_MAP_STRING:="G1:1, G2:24, IA1:23, IA2:22, "&
       "IA3:21, IA4:20, IIA1:19, IIA2:17, IIA3:16, IIA4:15, "&
       "IY1:2, IY2:3, IY3:4, IY4:5, IIY1:7, IIY2:8, IIY3:9, IIY4:10, "&
       "GND:6, VCC:18, TDO:11, TMS:12, TCK:13, TDI:14";

  attribute TAP_SCAN_IN    of TDI : signal is true;
  attribute TAP_SCAN_MODE  of TMS : signal is true;
  attribute TAP_SCAN_OUT   of TDO : signal is true;
  attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

  attribute INSTRUCTION_LENGTH of ttl74bct8244 : entity is 8;

  attribute INSTRUCTION_OPCODE of ttl74bct8244 : entity is
       "BYPASS (11111111, 10001000, 00000101, 10000100, 00000001),"   &
       "EXTEST (00000000, 10000000),"  &
       "SAMPLE (00000010, 10000010),"  &
       "INTEST (00000011, 10000011),"  &
       "TRIBYP (00000110, 10000110),"  &      -- Boundary Hi-Z
       "SETBYP (00000111, 10000111),"  &      -- Boundary 1/0
       "RUNT   (00001001, 10001001),"  &      -- Boundary run test
       "READBN (00001010, 10001010),"  &      -- Boundary read normal
       "READBT (00001011, 10001011),"  &      -- Boundary read test
       "CELLTST(00001100, 10001100),"  &      -- Boundary selftest normal
       "TOPHIP (00001101, 10001101),"  &      -- Boundary toggle out test
       "SCANCN (00001110, 10001110),"  &      -- BCR Scan normal
       "SCANCT (00001111, 10001111)";         -- BCR Scan test

  attribute INSTRUCTION_CAPTURE of ttl74bct8244 : entity is "10000001";
-- attribute INSTRUCTION_DISABLE of ttl74bct8244 : entity is "TRIBYP";

  attribute REGISTER_ACCESS of ttl74bct8244 : entity is
       "BOUNDARY (READBN, READBT, CELLTST),"   &
       "BYPASS (TOPHIP, SETBYP, RUNT, TRIBYP),"   &
       "BCR[2] (SCANCT, SCANCN)";   -- 2-bit Boundary Control Register

  attribute BOUNDARY_CELLS of ttl74bct8244 : entity is "BC_1";
  attribute BOUNDARY_LENGTH of ttl74bct8244 : entity is 18;

  attribute BOUNDARY_REGISTER of ttl74bct8244 : entity is
      -- num cell  port    function safe [ccell disval rslt]
        "17 (BC_1, G1,     input,   X)," &
        "16 (BC_1, G2, input,    X)," &
        "15 (BC_1, IA1,    input,   X)," &
        "14 (BC_1, IA2,    input,   X)," &
        "13 (BC_1, IA3,    input,   X)," &
        "12 (BC_1, IA4,    input,   X)," &
        "11 (BC_1, IIA1,    input,   X)," &
        "10 (BC_1, IIA2,    input,   X)," &
        "9  (BC_1, IIA3,    input,   X)," &
        "8  (BC_1, IIA4,    input,   X)," &
        "7  (BC_1, IY1,    output3, X,  17, 1, Z)," & --cell 17 @ 1 -> Hi-Z.
        "6  (BC_1, IY2,    output3, X,  17, 1, Z)," &
        "5  (BC_1, IY3,    output3, X,  17, 1, Z)," &
        "4  (BC_1, IY4,    output3, X,  17, 1, Z)," &
        "3  (BC_1, IIY1,    output3, X,  16, 1, Z)," &
        "2  (BC_1, IIY2,    output3, X,  16, 1, Z)," &
        "1  (BC_1, IIY3,    output3, X,  16, 1, Z)," &
        "0  (BC_1, IIY4,    output3, X,  16, 1, Z)";
end ttl74bct8244;
```

```
MODULE = module_1;
LIB  = lib1(1);
DEVICE_LIST = (U3 app1);
TEST_BUS =
  RING 0: U3;
NET_LIST =
  NET 1: (U3 SUM) (U3 DIN);
MODULE_TEST
#include <stdio.h>
#include <string.h>
#include "comp.h"
main()
{
  testchip(U3);
}
END_TEST
```

```
-- BSDL description of Texas Instruments 74bct8374 Octal D Flip-Flop
entity ttl74bct8374 is
  generic (PHYSICAL_PIN_MAP : string := "DW_PACKAGE");

  port (CLK:in bit; Q:out bit_vector(1 to 8); D:in bit_vector(1 to 8);
        GND, VCC:linkage bit; OC_NEG:in bit; TDO:out bit; TMS, TDI, TCK:in bit);

  use STD_1149_1_1990.all;   -- Get Std 1149.1-1990 attributes and definitions

  attribute PIN_MAP of ttl74bct8374 : entity is PHYSICAL_PIN_MAP;

  constant DW_PACKAGE:PIN_MAP_STRING:="CLK:1, Q:(2,3,4,5,7,8,9,10), " &
          "D:(23,22,21,20,19,17,16,15)," &
          "GND:6, VCC:18, OC_NEG:24, TDO:11, TMS:12, TCK:13, TDI:14";

  constant FK_PACKAGE:PIN_MAP_STRING:="CLK:9, Q:(10,11,12,13,16,17,18,19)," &
          "D:(6,5,4,3,2,27,26,25)," &
          "GND:14, VCC:28, OC_NEG:7, TDO:20, TMS:21, TCK:23, TDI:24";

  attribute TAP_SCAN_IN    of TDI : signal is true;
  attribute TAP_SCAN_MODE  of TMS : signal is true;
  attribute TAP_SCAN_OUT   of TDO : signal is true;
  attribute TAP_SCAN_CLOCK of TCK : signal is (20.0e6, BOTH);

  attribute INSTRUCTION_LENGTH of ttl74bct8374 : entity is 8;

  attribute INSTRUCTION_OPCODE of ttl74bct8374 : entity is
       "BYPASS (11111111, 10001000, 00000101, 10000100, 00000001),"   &
       "EXTEST (00000000, 10000000)," &
       "SAMPLE (00000010, 10000010)," &
       "INTEST (00000011, 10000011)," &
       "TRIBYP (00000110, 10000110)," &      -- Boundary Hi-Z
       "SETBYP (00000111, 10000111)," &      -- Boundary 1/0
       "RUNT   (00001001, 10001001)," &      -- Boundary run test
       "READBN (00001010, 10001010)," &      -- Boundary read normal
       "READBT (00001011, 10001011)," &      -- Boundary read test
       "CELLTST(00001100, 10001100)," &      -- Boundary selftest normal
       "TOPHIP (00001101, 10001101)," &      -- Boundary toggle out test
       "SCANCN (00001110, 10001110)," &      -- BCR Scan normal
       "SCANCT (00001111, 10001111)";        -- BCR Scan test

  attribute INSTRUCTION_CAPTURE of ttl74bct8374 : entity is "10000001";
  attribute INSTRUCTION_DISABLE of ttl74bct8374 : entity is "TRIBYP";

  attribute REGISTER_ACCESS of ttl74bct8374 : entity is
       "BOUNDARY (READBN, READBT, CELLTST),"   &
       "BYPASS (TOPHIP, SETBYP, RUNT, TRIBYP),"   &
       "BCR[2] (SCANCN, SCANCT)";   -- 2-bit Boundary Control Register

  attribute BOUNDARY_CELLS of ttl74bct8374 : entity is "BC_1";
  attribute BOUNDARY_LENGTH of ttl74bct8374 : entity is 18;

  attribute BOUNDARY_REGISTER of ttl74bct8374 : entity is
      -- num cell  port    function safe [ccell disval rslt]
        "17 (BC_1, CLK,     input,    X)," &
        "16 (BC_1, OC_NEG,  input,    X)," &  -- Merged Input/Control
        "16 (BC_1, *,       control,  1)," &  -- Merged Input/Control
        "15 (BC_1, D(1),    input,    X)," &
        "14 (BC_1, D(2),    input,    X)," &
        "13 (BC_1, D(3),    input,    X)," &
        "12 (BC_1, D(4),    input,    X)," &
        "11 (BC_1, D(5),    input,    X)," &
        "10 (BC_1, D(6),    input,    X)," &
        "9  (BC_1, D(7),    input,    X)," &
        "8  (BC_1, D(8),    input,    X)," &
        "7  (BC_1, Q(1),    output3, X,  16, 1, Z)," &  -- cell 16 @ 1 -> Hi-Z.
        "6  (BC_1, Q(2),    output3, X,  16, 1, Z)," &
        "5  (BC_1, Q(3),    output3, X,  16, 1, Z)," &
        "4  (BC_1, Q(4),    output3, X,  16, 1, Z)," &
        "3  (BC_1, Q(5),    output3, X,  16, 1, Z)," &
        "2  (BC_1, Q(6),    output3, X,  16, 1, Z)," &
        "1  (BC_1, Q(7),    output3, X,  16, 1, Z)," &
        "0  (BC_1, Q(8),    output3, X,  16, 1, Z)";

end ttl74bct8374;
```

D Synthesized Test Programs for the Demonstration

```
# Makefile for Microsoft MAKE
# For testing module described by work.mtl
CFLAGS=/AM /Gs /Od /Zi /FPc /c
CL = cl $(CFLAGS)
.C.OBJ:
        $(CL) $*.C

inetpc.obj: inetpc.c

template.obj: template.c

driver.obj: driver.c

workmain.obj: workmain.c

OBJ1= driver.obj inetpc.obj template.obj workmain.obj

work.exe: $(OBJ1)
        LINK /ST:2048 $**, $@;
```

```
#include <stdio.h>
#include <string.h>
#include "comp.h"
main()
{
  int ans;
  /*---- do interconnect test --------- */
inetpc(1); /* test only */

  /*---- do interconnect test --------- */
inetpc(0); /* diagnosis */

  kernel1();
  /* --call fullscanN in chipID=U3, chipType = app1---- */
sprintf(chipID, "%s", "U3");
sprintf(chipType, "%s", "app1");
sprintf(regIns, "%s", "110000");
sprintf(vecFID, "%s", "AP1_IN2,AP1_IN1");
sprintf(expFID, "%s", "AP1_OUT2,AP1_OUT1");
fullscanN(chipID,chipType,0,3,16,1,2,2,regIns,vecFID,expFID);
/* ----- end call       ----- */

  /* --call fullscanN in chipID=U4, chipType = app1---- */
sprintf(chipID, "%s", "U4");
sprintf(chipType, "%s", "app1");
sprintf(regIns, "%s", "110000");
sprintf(vecFID, "%s", "AP1_IN2,AP1_IN1");
sprintf(expFID, "%s", "AP1_OUT2,AP1_OUT1");
fullscanN(chipID,chipType,0,0,19,0,3,2,regIns,vecFID,expFID);
/* ----- end call       ----- */

}
int kernel1()
{
char outs[50], exps[50], ins[50];
int i, j, ans, x, faults=0;
FILE *kvecF, *kresF;
extern char *scan();

  if((kvecF=fopen("kernel1.vec", "r"))==NULL){
    printf("file kernel1.vec doesn't exist.\n");
    exit(2);
  }
  if((kresF=fopen("kernel1.res","r"))==NULL){
    printf("file kernel1.res doesn't exits.\n");
    exit(2);
  }
  sprintf(outs,"%s","00000000000000000000000");
  scan(0,1,0,0,outs);
  printf("Testing kernel 1 ... \n");
  for(i=0; i<16; i++){
    for(j=0; j<42; j++){
      fscanf(kvecF, "%d", &x);
      if(x==1) outs[j]='1'; else outs[j]='0';
      fscanf(kresF, "%d", &x);
      if(x==1){exps[j]='1';} else{if(x==0) exps[j]='0';else exps[j]='2';}
    }
    outs[j]='\0';
    exps[j]='\0';
    scan(0,0,0,0,outs);
    strcpy(ins, scan(0,0,0,0,outs));

    for(j=0; j<42; j++){if((exps[j]!='2') && (exps[j]!=ins[j])) break;}
    if(j<42) {
      faults++;
      printf("err:v%d,\n outs=%s,\n exps=%s,\n ins =%s\n",i,outs,exps,ins);
      printf("continue?(1:yes) :");
      scanf("%d",&ans); printf("  ... \n");
      if(ans!=1) {printf("aborted.\n"); exit(2);}
    }
  }
  fclose(kvecF); fclose(kresF);
  if (faults!=0) printf("Faults detected in kernel1.\n");
  else printf("Kernel1 tested OK.\n");
}
```

```
8 1 0 22 42
1 1     0 0 0       1 0 5
2 1     0 4 0       1 0 2
3 1     0 1 0       1 0 38
4 1     0 3 0       1 0 39
5 1     0 31 1 41 1       1 0 21
6 1     0 30 1 41 1       1 0 20
7 1     0 29 1 41 1       1 0 19
8 1     0 28 1 41 1       1 0 18
```

The format of this file is as follos. The first line indicates that there are 8 nets and 1 ring in the module. Ring 0 contains 22 instruction cells and 42 boundary scan cells. The second line indicates that net 1 has 1 2-state output driver which is located on ring 0 at the 0th location (cloest to TDO). This net has 1 receiver which is located on ring 0 at the 5th location. There is 1 receiver for this net. The next three lines are similar to the second line. The sixth line indicates that net 5 has 1 3-state driver which is located on ring 0 at the the 31th location. The driver is disabled if the value of the control cell, which is located on the same ring at location 41, is 1.

```
/* comp.h */
/* This file contains declaration of global variables
   that are used in the main program. */
#define LMAX 100
char chipID[30];
char chipType[30];
char regIns[LMAX];
char vecFID[LMAX];
char expFID[LMAX];
char iniIns[LMAX];
char iniVal[LMAX];
char useIns[LMAX];
char resIns[LMAX];
char expVal[LMAX];
#define IR 1
#define DR 0
int BILBO_cycles;
FILE *busf;
```

```
/* inetpc.c: PC version of inet testing ------------- Lien 3/21/91
NEED: input file *infofile.net* on PC, and a function: scan()
available: inetpc(test_only)
*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
/* Remove the need to depend on the global data structures.
   The file  *infofile.net* contains all info needed for inet testing.*/
#define NIL 0
#define IR 1
#define DR 0
int ans;
        /* ---------------- begin of global data ------------ */
/* four global variables: netNumz, ringNumz, rip, nip. */
int ringNumz; /* ring number */
struct STri{ /* ring info struct */
  int brslen;  /* ring brcell num */
  int irslen;  /* ring ircell num */
} *rip; /* pointer to ring info: ring id=i in ith position*/
int netNumz; /* net number */
struct STni{ /* net info struct */
  int nid; /* net id */
  int drvn; /* driver number */
  struct STdrv *drvp;
  int rcvn; /* receiver number */
  struct STrcv *rcvp;
};
struct STni *nip;  /* pointer to net info */
struct STdrv{ /* struct of a driver */
  int rid;   /* the ring the driver is located */
  int cloc;  /* loc in the ring, 0 is next to TDO */
  int contd; /* 0: 2-state drv, 1: 3-state drv */
  int con_cloc; /* if 3-state, the loc of cont cell(in the same ring)*/
  int dis_val;  /* value of the cont cell that disable this drv*/
};
struct STrcv{ /* struct of a receiver */
  int rid;  /* the ring the receiver is located */
  int cloc; /* location in the ring, cloc=0 is next to TDO of ring*/
};          /* ---------------------- end of global data */

struct STrecordset{
int nid; /* net id */
int rid; /* receiver ring id */
int loc; /* receiver location */
};

struct STrecordset *sa1p, *sa0p;
int sa1Num, sa0Num;

             /* record an stuck_at_one receiver */
record_stuck_one(netid, rcv_rid, rcv_loc)
int netid, rcv_rid, rcv_loc;
{
  (sa1p+sa1Num)->nid = netid;
  (sa1p+sa1Num)->rid = rcv_rid;
  (sa1p+sa1Num)->loc = rcv_loc;
  sa1Num++;
}
record_stuck_zero(netid, rcv_rid, rcv_loc)
int netid, rcv_rid, rcv_loc;
{
  (sa1p+sa1Num)->nid = netid;
  (sa1p+sa1Num)->rid = rcv_rid;
  (sa1p+sa1Num)->loc = rcv_loc;
  sa1Num++;
}
             /*see if the receiver was recorded as stuck_at_one*/
int rcv_stuck_one( netid, rcv_rid, rcv_loc)
int netid, rcv_rid, rcv_loc;
{
  int i;
  for(i=0; i<sa1Num; i++){
    if((netid == (sa1p+sa1Num)->nid) &&
       (rcv_rid == (sa1p+sa1Num)->rid) &&
       (rcv_loc == (sa1p+sa1Num)->loc))
    break;
  }
  if(i < sa1Num) return(1);
  else return(0);
}
             /*see if the receiver was recorded as stuck_at_one*/
int rcv_stuck_zero( netid, rcv_rid, rcv_loc)
int netid, rcv_rid, rcv_loc;
{
  int i;
  for(i=0; i<sa0Num; i++){
    if((netid == (sa0p+sa0Num)->nid) &&
       (rcv_rid == (sa0p+sa0Num)->rid) &&
       (rcv_loc == (sa0p+sa0Num)->loc))
    break;
  }
  if(i < sa0Num) return(1);
  else return(0);
}

inetpc(test_only)
int test_only;
{
  int faults;
  int inet_test(), read_netinfo(), show_netinfo_on_ibm();
  printf("Testing interconnect...\n");
  read_netinfo();
#ifdef DEBUG
  show_netinfo_on_ibm();
#endif
  faults=inet_test(test_only);
  if(faults==0) printf("Completed with no fault.\n");
  else printf("%d faults detected.\n",faults);
  return(faults);
}
int inet_test(test_only) /* return num of failed tests, 0 if no fault. */
int test_only;   /* 1: test (use count seq, 0: diagnosis (use w1 seq) */
{
  FILE *fp;
  int log2(), pow2();
  int stvn, ptvn; /* stvn is netNum, ptvn is num of test vector. */
  int i, j, k, mrid, mrlen, mcloc, step, tt, failed_vecn, errn;
  char *get1ptv_count_seq(), *get1ptv_w1_seq(), *get1ptv_w0_seq();
  char *ptvp; /*pointer of ptv(parallel test vector) */
  char *insp, *statusp;
  char *p_sm; /* p_sm: pt to buf mem */
  char **osp, **isp, *mp, *m2p, *tp;
       /* mp: address of allo. mem., osp:out ring strings pointer */
       /* isp: input ring strings pointer, tp: temp pointer*/
  extern char *scan();
  int maxdrvn, drvn; /* max drv num, and drv num */
```

```
  if((fp=fopen("infofile.net", "r")) == NULL) {
    printf("file infonet.net doesn't exist.\n");
    exit(1);
  }
  fscanf(fp, "%d", &stvn); /* get the number of nets as stvn*/
  fclose(fp);
/*  printf("stvn (netNum)= %d\n", stvn);*/
    /* ------ begin counting seq ---------- */
  ptvn = log2(stvn+1)+1;  /* size of a counting sequence as ptvn. */
/*  printf("ptvn(vecNum)=%d in counting seq.\n",ptvn);*/
  p_sm=(char *) malloc((stvn+2)*sizeof(char)); /* p to mem as a buf */
      /* allo. output ring strings */
  for(i=tt=0; i<ringNumz; i++){ /* sum up brslen of all rings */
    tt += ((rip+i)->brslen) +1;  /*extra bit for string terminator */
  }
/*  printf("mp size =%d\n", tt);*/
                    /* allo. mem for intermediate pointers*/
  mp=(char *)malloc(tt*sizeof(char));   /* allo. mem to (osp) strings */
  for(i=0; i<tt; i++) *(mp+i)='0'; /* clear all ring strings*/
  m2p=(char *)malloc(tt*sizeof(char));   /* allo. mem for (isp) strings */
  for(i=0; i<tt; i++) *(m2p+i)='0'; /* clear all ring strings*/
  osp=(char **)malloc(ringNumz*sizeof(char *)); /* out rings data ptr*/
  isp=(char **)malloc(ringNumz*sizeof(char *)); /*in rings data ptr */
  sa1p=(struct STrecordset *)malloc(stvn*10*sizeof(struct STrecordset));
  sa0p=(struct STrecordset *)malloc(stvn*10*sizeof(struct STrecordset));
  sa1Num = sa0Num=0; /* initialize sa1 and sa0ed receivers set */

/*
  for(i=0; i<ringNumz; i++) printf("%d th ring irslen=%d brslen=%d\n",
                                  i, (rip+i)->irslen, (rip+i)->brslen);
*/
                          /* set intermediate pointers for osp*/
  for(i=0, tp=mp; i<ringNumz; i++){  /* for each out ring strings */
    *(osp+i)=tp;  /* assign the address for each intermd. pointers */
    tp += (rip+i)->brslen; /* osp holds bits for br usage */
    *(tp++)='\0';  /* add terminator */
  }
                          /* set intermediate pointers for isp*/
  for(i=0, tp=m2p; i<ringNumz; i++){  /* for each out ring strings */
    *(isp+i)=tp;  /* assign the address for each intermd. pointers */
    tp += (rip+i)->brslen;
    *(tp++)='\0';  /* add terminator */
  }
      /*set IR of all chips to EXTEST=00...00 mode for testing inet */
      /* --- allocate mem for ins and status --- */
  for(i=tt=0; i<ringNumz; i++) /* get max irslen of all rings*/
     if(tt < (rip+i)->irslen) tt = (rip+i)->irslen;
  insp=(char *)malloc(2*(tt+1)*sizeof(char)); /* alloc. mem for ins */
  statusp=insp + (tt+1)*sizeof(char);  /* alloc. mem for status */
  for(i=0; i<ringNumz; i++) { /* set ir=00..0 for each ringid = i */
    for(j=0; j<(rip+i)->irslen; j++) *(insp+j)='0'; /* 0..00 string*/
    *(insp + j)='\0'; /* terminate the string, now the instr. is ready */
    strcpy(statusp, scan(i, IR, 0, 0, insp)); /* send insp to IR*/
       /* ----------- the scan function is hardware realted -------- */
  }  /* Now, all chips are in EXTEST mode */
/*
  printf("tp1: all chips are in EXTEST. Continue(1:yes,0:no)?");
  scanf("%d",&ans);
  printf("\n");
  if(ans==0)exit(1);
*/
  for(i=0, maxdrvn =0; i<netNumz; i++) { /* find maxdrvn */
    if(maxdrvn < (nip+i)->drvn) maxdrvn = (nip+i)->drvn;
  }
```

```
   /*next apply test vectors, the reg selected is BSR for every chip*/
  failed_vecn=0; /* number of failed test vectors */
  if(test_only==1){  /*--- begin: test inet using count seq. ------ */
    for(drvn=0; drvn < maxdrvn; drvn++){
      for(i=0; i<ptvn; i++){ /*test for fault in inet, no diagnosis*/
        ptvp = get1ptv_count_seq(stvn, i, p_sm); /*count seq: get i th ptv */
/*      printf("count_seq: %d th ptv = %s\n", i, ptvp);  */
        /* --- next, map ptv to output rings_trings of a vector */
        mptv2osp(ptvp, stvn, osp, drvn);/* map ptv to output ring strings */
/*      for(j=0;j<ringNumz;j++) printf("%dth ring of osp=%s\n",j, *(osp+j));*/
        scan_bsr_rings(osp, isp); /*actually apply out strings to each ring */
        /* NOTE: no optimal schedule is used */
        /*
        printf("tp2:measure the osp=%s ",osp);
        printf("\n isp=%s\n continue(1:yes)?",isp);
        scanf("%d",&ans); printf("\n"); if(ans==0)exit(1);
        */
/*      for(j=0; j<ringNumz; j++) printf("%dth ring of isp=%s\n",j, *(isp+j));*/
        if((errn=cmp_isp_ptv(ptvp, stvn, isp)) !=0){
          /* check if isp is same as ptv */
          printf("test failed in %d th vector, errs in %d rcvs.\n", i,errn);
/*
            printf("mismatched\n");
            printf("ptv=%s\n", ptvp);
            for(k=0; k<ringNumz; k++)
              printf("received: ring %d,isp=%s\n",k, *(isp+k));
*/
          failed_vecn++;
        }
      } /* i */
    }  /* drvn */
    /* ------ end: test inet using counting seq ------------ */
  } /* if */
  else{ /* diagnosis */
    for(errn=drvn=0; drvn < maxdrvn; drvn++){/*for each driver of each net*/

      /* all-0 vector */
      for(i=0; i<stvn; i++) *(p_sm +i)='0'; /* all-0 vector */
      ptvp = p_sm;
      mptv2osp(ptvp, stvn, osp, drvn);/* map ptv to output ring strings */
      scan_bsr_rings(osp, isp); /*actually apply out strings to each ring */
      for(j=0; j<stvn; j++){  /* check for stuck-at-1 for every net */
        for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
          mrid=(((nip+j)->rcvp)+k)->rid;
          mcloc=(((nip+j)->rcvp)+k)->cloc;
          if ( *(*(isp+mrid)+mcloc) != '0') {  /* stuck-at-1 */
/*
            printf("opened net %d, drv:rid=%d,loc=%d,rcv:rid=%d,loc=%d\n",i,
                  (((nip+i)->drvp)+0)->rid, (((nip+i)->drvp)+0)->cloc,
                  mrid, mcloc);
*/
            record_stuck_one(j, mrid, mcloc);
            errn++;
          } /* if */
        } /* k */
      } /* j */

      /* all-1 vector */
      for(i=0; i<stvn; i++) *(p_sm +i)='1'; /* all-1 vector */
      ptvp = p_sm;
      mptv2osp(ptvp, stvn, osp, drvn);/* map ptv to output ring strings */
      scan_bsr_rings(osp, isp); /*actually apply out strings to each ring */
      for(j=0; j<stvn; j++){  /* check for stuck-at-0 for every net */
        for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
```

```
        mrid=(((nip+j)->rcvp)+k)->rid;
        mcloc=(((nip+j)->rcvp)+k)->cloc;
        if ( *(*(isp+mrid)+mcloc) != '1') {  /* stuck-at-0 */
          printf("opened net %d, drv:rid=%d,loc=%d,rcv:rid=%d,loc=%d\n",i,
                 (((nip+i)->drvp)+0)->rid, (((nip+i)->drvp)+0)->cloc,
                 mrid, mcloc);
          record_stuck_zero(j, mrid, mcloc);
                /*net j, rcv at ring mrid, loc=mcl at loc is opened */
          errn++;
        } /* if */
      } /* k */
    } /* j */

  /* walking-1 seq */
    for(i=0; i<stvn; i++){/*generate ith ptv,NOTE ptvn=stvn in w1. */
      ptvp=get1ptv_w1_seq(stvn, i, p_sm);
/*    printf("w1_seq: %d th ptv = %s\n", i, ptvp); */
      mptv2osp(ptvp, stvn, osp, drvn);/* map ptv to output ring strings */
/*    for(j=0; j<ringNumz; j++) printf("%dth ring osp=%s\n",j, *(osp+j)); */
      scan_bsr_rings(osp, isp); /*actually apply out strings to each ring */
      /* NOTE: no optimal schedule is used */
/*    for(j=0; j<ringNumz; j++) printf("%dth ring of isp=%s\n",j, *(isp+j));*/
      /* --- now do diagnosis (based on ith net drives) ------ */
      if(*(ptvp+i) != '1'){ /* drive value is '1' */
        printf("drive value is wrong, %dth vector in diagnosis w1 seq\n",i);
        exit(1);
      }
      for(j=0; j<stvn; j++){
        if(j==i){ /*check for open */
          for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
            mrid=(((nip+j)->rcvp)+k)->rid;
            mcloc=(((nip+j)->rcvp)+k)->cloc;
            if ( *(*(isp+mrid)+mcloc) != '1') {  /* open fault: list it */
              printf("opened net %d, drv:rid=%d,loc=%d,rcv:rid=%d,loc=%d\n",i,
                 (((nip+i)->drvp)+0)->rid, (((nip+i)->drvp)+0)->cloc,
                 mrid, mcloc);
              errn++; /*--NOTE: the fault may also be outputed to a file*/
            }
          } /* k */
        } /* if */
        else{  /* check for short */
          for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
            mrid=(((nip+j)->rcvp)+k)->rid;
            mcloc=(((nip+j)->rcvp)+k)->cloc;
            if ( *(*(isp+mrid)+mcloc)=='1' && !rcv_stuck_one(j,mrid,mcloc)){
                      /* short fault: list it */
              printf("short(net=%d,net=%d),D:rid=%d,loc=%d,R:rid=%d,loc=%d\n",
                     i,j,(((nip+i)->drvp)+0)->rid,(((nip+i)->drvp)+0)->cloc,
                     mrid,mcloc);
              errn++;
            }
          } /* k */
        } /* else */
      } /* j */
    } /* i */

  /* walking-0 seq */
    for(i=0; i<stvn; i++){/*generate ith ptv,NOTE ptvn=stvn in w1. */
      ptvp=get1ptv_w0_seq(stvn, i, p_sm);
/*    printf("w0_seq: %d th ptv = %s\n", i, ptvp); */
      mptv2osp(ptvp, stvn, osp, drvn);/* map ptv to output ring strings */
/*    for(j=0; j<ringNumz; j++) printf("%dth ring osp=%s\n",j, *(osp+j)); */
      scan_bsr_rings(osp, isp); /*actually apply out strings to each ring */
      /* NOTE: no optimal schedule is used */
/*    for(j=0; j<ringNumz; j++) printf("%dth ring of isp=%s\n",j, *(isp+j));*/
      /* --- now do diagnosis (based on ith net drives) ------ */
      if(*(ptvp+i) != '0'){ /* drive value is '0' */
        printf("drive value is wrong, %dth vector in diagnosis w0 seq\n",i);
        exit(1);
      }
      for(j=0; j<stvn; j++){
        if(j==i){ /*check for open */
          for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
            mrid=(((nip+j)->rcvp)+k)->rid;
            mcloc=(((nip+j)->rcvp)+k)->cloc;
            if ( *(*(isp+mrid)+mcloc) != '0') {  /* open fault: list it */
              printf("opened net %d, drv:rid=%d,loc=%d,rcv:rid=%d,loc=%d\n",i,
                 (((nip+i)->drvp)+0)->rid, (((nip+i)->drvp)+0)->cloc,
                 mrid, mcloc);
              errn++; /*--NOTE: the fault may also be outputed to a file*/
            }
          } /* k */
        } /* if */
        else{  /* check for short */
          for(k=0; k<(nip+j)->rcvn; k++){ /*for every rcv of jth net*/
            mrid=(((nip+j)->rcvp)+k)->rid;
            mcloc=(((nip+j)->rcvp)+k)->cloc;
            if ( *(*(isp+mrid)+mcloc)=='0' && !rcv_stuck_zero(j,mrid,mcloc)){
                      /* short fault: list it */
              printf("short(net=%d,net=%d),D:rid=%d,loc=%d,R:rid=%d,loc=%d\n",
                     i,j,(((nip+i)->drvp)+0)->rid,(((nip+i)->drvp)+0)->cloc,
                     mrid,mcloc);
              errn++;
            }
          } /* k */
        } /* else */
      } /* j */
    } /* i */


    } /* drvn */
  } /* else */
  return((test_only==1) ? failed_vecn: errn);
}

int cmp_isp_ptv(ptvp, stvn, isp)   /*cmp recieved rings data(isp) with ptv*/
char *ptvp, **isp;
int stvn;
{
  int i, j, errn=0;
  int mrid, mcloc; /* rid and cloc(cell location) for current rcv*/
  char rcv_val, drv_val;
  for(i=0; i<stvn; i++){  /*check for each net*/
    for(j=0; j<(nip+i)->rcvn; j++){     /*check for jth receiver*/
      mrid = (((nip+i)->rcvp)+j)->rid;  /* jth rcv ringid, location*/
      mcloc = (((nip+i)->rcvp)+j)->cloc;
      drv_val= *(ptvp+i); /* value on driver of ith net */
      rcv_val= *(*(isp+mrid)+mcloc); /*rcvval of ring:mid, loc:cloc*/
      if(drv_val != rcv_val) {  /* is value of drv = rcv ? */
        errn++;  /* add one more error */
        printf("err:net=%d,drv val=%c, %dth rcv val=%c,(rid=%d,loc=%d).\n",
               i, drv_val, j, rcv_val, mrid, mcloc);
      }
    } /* j */
  } /* i */
  return(errn);
}
scan_bsr_rings(osp, isp)
```

```
char **osp, **isp;    /* test schedule is NOT optimized */
{           /* rewrite this function if optimal schedule is needed */
  int i, j, tt;
  char *insp, *statusp;
  extern char *scan(); /* the device driver scan function for IBM PC only */
                /* for testing on a SUN need a "pseudo" scan */
        /* ---------- next sent inet test data ---------- */
  for(i=0; i<ringNumz; i++){  /* send out data: received garbage */
    strcpy(*(isp+i), scan(i, DR, 0, 0, *(osp+i)));
  }
  for(i=0; i<ringNumz; i++){ /* send out data again: received results*/
    strcpy(*(isp+i), scan(i, DR, 0, 0, *(osp+i)));
  }
        /* data in osp sent and received result in isp */
        /* ---- end of one test vector and one result vector --- */
}

mptv2osp(ptvp, stvn, p, cdrvn)   /* map a ptv to the output ring strings */
int stvn;  /* len of a ptv, i.e., num of stv (=netNumz) */
char *ptvp, **p; /* ptvp: pt to a ptv, p the filled data structure */
int cdrvn; /* the current drvn th driver is enabled */
{
int i, j, k, dn;
int mrid, mcloc, mcontd, mcon_cloc, mdis_val;
  /*printf("ptv=%s\n", ptvp);*/
  for(i=0; i<stvn; i++){ /* do  for each bit in the ptv for ith net */
    dn = ((nip+i)->drvn < cdrvn)? 0 : cdrvn; /* find the enabled driver */
    mrid=(((nip+i)->drvp)+dn)->rid; /* rid of driver dn */
    mcloc=(((nip+i)->drvp)+dn)->cloc; /*cloc of driver dn */
        /* assigned it to the rs */
    *(*(p+mrid)+mcloc)=*(ptvp+i);  /* ptv value for ith net */
    /*printf("ptv %dth bit=%c,to ring=%d,loc=%d\n",i,*(ptvp+i),mrid,mcloc);*/
    mcontd= (((nip+i)->drvp)+dn)->contd; /*contd of driver dn*/
    if(mcontd==1){ /* 3-state, get concell loc and dis_val*/
      mcon_cloc=(((nip+i)->drvp)+dn)->con_cloc;
      mdis_val=(((nip+i)->drvp)+dn)->dis_val;
      if (mdis_val==0) *(*(p+mrid)+mcon_cloc)='1';
      else *(*(p+mrid)+mcon_cloc)='0';
      /* printf("ptv %d the bit enb conval=%d, assign to rin=%d, loc=%d\n",
            i, 1- mdis_val, mrid, mcon_cloc); */
    }
  }
}

char *get1ptv_w1_seq(stvn,i,pw1) /*get next vector in walking-1 seq*/
int stvn, i;
char *pw1;
{
  int j;
  if(i==0){ /* if yes, set up first vector as 100000...000 */
    *pw1='1';
    for(j=1; j<stvn; j++) *(pw1+j)='0';
    *(pw1+j)='\0'; /* terminate the seq*/
   return(pw1);
  }
  else{ /* ith vector use i-1 th vector */
    *(pw1+i-1)='0';
    *(pw1+i)='1';
    return(pw1);
  }
}

char *get1ptv_w0_seq(stvn,i,pw1) /*get next vector in walking-0 seq*/
int stvn, i;
char *pw1;
{
  int j;
  if(i==0){ /* if yes, set up first vector as 01111...1111 */
    *pw1='0';
    for(j=1; j<stvn; j++) *(pw1+j)='1';
    *(pw1+j)='\0'; /* terminate the seq*/
   return(pw1);
  }
  else{ /* ith vector use i-1 th vector */
    *(pw1+i-1)='1';
    *(pw1+i)='0';
    return(pw1);
  }
}

char *get1ptv_count_seq(stvn,i, p) /*get next vector in counting seq*/
int stvn, i;
char *p;
{
  char curValue;
  int j, k, step;
    curValue='0';
    step = pow2(i);
    for(j=0, k=1; j <= (stvn+1); j++, k++){ /*j counts netNum (stvn)*/
      if(k>step){
        k=1;
        if(curValue == '0') curValue = '1';
        else curValue='0';
      }
      *(p+j)=curValue; /* get 1 bit */
    } /* j */
    *(p+stvn+1) = '\0'; /* terminate the string */
    return((p+1));  /* discard 1st bit, return a pointer */
}

int log2(n)
{
  int i,j=0;
  for(i=n; i>1; j++, i /= 2);
  return(j);
}

int pow2(n)
int n;
{
  int i, sum=1;
  if(n<1) return(sum);
  else for(i=n; i>0; i--) sum *= 2;
  return(sum);
}

int show_netinfo_on_ibm()
{
  int i, j;
  printf("%d %d ", netNumz, ringNumz);
  for(i=0; i<ringNumz; i++){ /* show ith ring, irslen, brslen */
    printf("%d %d %d ", i, (rip+i)->irslen, (rip+i)->brslen);
  }  /* showed rip */
  printf("\n");
  for(i=0; i<netNumz; i++){  /* do for every net */
    printf("%d %d ", (nip+i)->nid, (nip+i)->drvn);
    for(j=0; j < ((nip+i)->drvn); j++){  /* do for every driver */
      printf("  %d %d %d ", (((nip+i)->drvp)+j)->rid,
```

```
          (((nip+i)->drvp)+j)->cloc, (((nip+i)->drvp)+j)->contd);
      if((((nip+i)->drvp)+j)->contd == 1){
        printf("%d %d ", (((nip+i)->drvp)+j)->con_cloc,
              (((nip+i)->drvp)+j)->dis_val);
      }
    }

    printf("  %d", (nip+i)->rcvn);
    for(j=0; j < (nip+i)->rcvn; j++){
      printf(" %d %d ", (((nip+i)->rcvp)+j)->rid,(((nip+i)->rcvp)+j)->cloc);
    }
    printf(" \n");
  }
}

int read_netinfo()   /* read net "infofile.net" into nip */
{
  FILE *fp;
  int i,j;
  int v1, v2, v3, v4, v5; /* tmp values */

  if((fp=fopen("infofile.net", "r"))==NULL){
    printf("file <infofile.net> doesn't exist. \n");
    exit(1);
  }
  fscanf(fp, "%d %d", &netNumz, &ringNumz); /*get number of nets and rings*/
  if ((rip=(struct STri *) malloc(ringNumz*sizeof(struct STri)))==NIL){
    printf("not enough memory for rip, ringNumz=%d. \n", ringNumz);
    exit(1);
  }  /* allocated memory for rip: ring info pointer */
  for(i=0; i<ringNumz; i++){ /* get info for every ring */
    if(fscanf(fp, "%d %d %d", &v1, &v2, &v3)==0){ /* not enough nets*/
      printf("not enough nets, i=%d, netNumz=%d.\n", i, netNumz);
      exit(1);
    }
    if(v1>= ringNumz) { /* v1 is ringid, v2 is irslen, v3 is brslen */
      printf("ring id=%d out of range in read_netinfo()\n",v1);
      exit(1);
    }
    (rip+v1)->irslen = v2; /* irslen of ring rid*/
    (rip+v1)->brslen = v3; /* brlen of ring rid*/
  } /* rip ith bit is info for ring id=i */

  if ((nip=(struct STni *) malloc(netNumz*sizeof(struct STni)))==NIL){
    printf("not enough memory for nip, netNumz=%d. \n", netNumz);
    exit(1);
  } /* allocated memory for nip: net info pointer */
  for(i=0; i<netNumz; i++){  /* do for every net */
    fscanf(fp, "%d", &v1);
    (nip+i)->nid = v1; /* get ith net id */
    fscanf(fp, "%d", &v2);
    (nip+i)->drvn = v2; /* get drv number for ith net */
    (nip+i)->drvp=(struct STdrv *) malloc((v2+1)*sizeof(struct STdrv));
                        /*allo. drv mem*/
    for(j=0; j < ((nip+i)->drvn); j++){  /* do for every driver of ith net */
      fscanf(fp, "%d %d %d", &v1, &v2, &v3); /* get fixed info */
      (((nip+i)->drvp)+j)->rid = v1;  /* get ring id */
      (((nip+i)->drvp)+j)->cloc = v2; /* get loc in the ring */
      (((nip+i)->drvp)+j)->contd = v3; /* v3==1 means 3-state drv */
      if(v3==1){   /* fill variable part only if needed */
        fscanf(fp, "%d %d", &v4, &v5);
        (((nip+i)->drvp)+j)->con_cloc = v4;
        (((nip+i)->drvp)+j)->dis_val = v5;
      }
    }
    fscanf(fp, "%d", &v1);
    (nip+i)->rcvn = v1; /* get rcv number for ith net */
    (nip+i)->rcvp=(struct STrcv *) malloc((v1+1)*sizeof(struct STrcv));
                        /*allo. rcv mem*/
    for(j=0; j < ((nip+i)->rcvn); j++){  /* do for every rcv of ith net */
      fscanf(fp, "%d %d", &v2, &v3);
      (((nip+i)->rcvp)+j)->rid = v2;
      (((nip+i)->rcvp)+j)->cloc = v3;
    }
  } /* i */
  fclose(fp);
}
```

```
/* template.c */
/* bilbo.c      template for BILBO TDM --- Lien 8/15/90 */
#include <stdio.h>
#include <string.h>
#define IR 1
#define DR 0
#define MAXCHAINNUM 5
#define BUFSIZE 1024
extern char *scan();
int xstrlen(s)
char *s;
{
  int i=0;
  while(*s++) i++;
  return(i);
}

int xextract(t, s)
char *t, *s;
{
  int i=0;
  while((*s != '\0') && (*s != ',')){
    *t++ = *s++;
    i++;
  }
  *t = '\0';
  return(i);
}

int xstrncpy(s, t, n)
char*s, *t;
int n;
{
  int i;
        for(i=0; i<n; i++) {
                if (*t == '\0') {*s='\0'; break;}
                else *s++ = *t++;
        }
        *s='\0';
}

int xstrcmp(s, t)
char *s, *t;
{
int i;
        for( ; (*s=='x') || (*s=='X') || (*s == *t); s++, t++)
                if (*s == '\0') return(0);
        return( *s - *t);
}

x2one(s)
char *s;
{
int i;
        while (*s !='\0') {
                if((*s=='x') || (*s=='X')) *s++='1';
                else s++;
        }
}

int bilbo(chipID,chipType,ringId,ipre,ipost,pre,post,iniNum,iniIns,iniVal,
                useIns, ckType, ckNum, ckfNum, resNum, resIns, expVal)
int ringId, ipre, ipost, pre, post, iniNum, ckType, ckNum,  resNum;
double ckfNum;
char *chipID, *chipType, *iniIns, *iniVal, *useIns, *resIns, *expVal;
{
   int i, len1, len2, faults=0;
   char *inbuf, outbuf[BUFSIZ];

        len1=strlen(iniIns)/iniNum;
        printf("ins_len1=%d \n", len1);
        printf("iniIns=%s\n", iniIns);
        printf("iniVal=%s\n", iniVal);
        printf("useIns=%s\n", useIns);
        printf("resIns=%s\n", resIns);
        printf("expVal=%s\n", expVal);

        for(i=0; i<iniNum; i++){
           xstrncpy(outbuf, iniIns, len1);
           iniIns += len1;
           printf("extracted iniIns=%s \n", outbuf);
           inbuf=scan(ringId, IR, ipre, ipost, outbuf);
           len2 = xextract(outbuf, iniVal);
           iniVal += len2+1;
           printf("extracted iniVal=%s \n", outbuf);
           inbuf=scan(ringId, DR, pre, post, outbuf);
        }
        inbuf=scan(ringId, IR, ipre, ipost, useIns);
        if (ckType) {           /* TCK */
                waittck(ckNum); }
        else {                  /* FCK */
                waitfck(ckNum); }
        len1= strlen(resIns)/resNum;
        for(i=0; i<resNum; i++){
           xstrncpy(outbuf, resIns, len1);
           resIns += len1;
           printf("extracted resIns=%s \n", outbuf);
           inbuf=scan(ringId, IR, ipre, ipost, outbuf);
           len2 = xextract(outbuf, expVal);
           expVal += len2 +1;
           printf("extracted expVal=%s \n", outbuf);
           inbuf=scan(ringId, DR, pre, post, outbuf);
           if (xstrcmp(outbuf, inbuf)) { /* not the same */
                faults++;
                printf("diff in result and expected result \n");
                printf("expect  = %s \n", outbuf);
                printf("received= %s \n", inbuf);
           }
        }
        if (faults) printf("chip %s failed in test using BILBO.\n", chipID);
        else printf("chip %s tested OK using BILBO.\n", chipID);
        return(faults);
}

/*------------ fscan1.c ---------------------- */

int fullscan1(chipID,chipType,ringID,ipre, ipost,pre,post,regIns,
                                                vecFID,expFID)
char *chipID, *chipType, *regIns, *vecFID, *expFID;
int ringID, ipre, ipost,pre, post;
{
   int xstrcmp();
   int i, j, bitVal, regLen, vecNum, ans, faults=0;
   FILE *vecf, *expf;
   char ovecBuf[BUFSIZ], vecBuf[BUFSIZ], resBuf[BUFSIZ], expBuf[BUFSIZ];
        if ((vecf=fopen(vecFID, "r")) == NULL) {
                printf("vec file: %s deos not exist!\n", vecFID);
                exit(1);
```

```
}
if((expf=fopen(expFID, "r")) == NULL) {
        printf("exp file: %s deos not exist!\n", expFID);
        exit(2);
}
fscanf(vecf, "%d %d", &vecNum, &regLen);
/* set up fullscan instruction */
scan(ringID, IR, ipre, ipost, regIns);
        /* ---- 1st vactor ---------------------------------- */
for(j=0; j<regLen; j++) {       /* read in 1st vector */
        fscanf(vecf, "%d", &bitVal);
        switch (bitVal) {
          case 0: vecBuf[j]='0'; break;
          case 1: vecBuf[j]='1'; break;
          default:
             printf("error vecfile:%s,vec=%d,bit=%d\n",vecFID,i,j);
             exit(3);
        }
}
vecBuf[regLen]='\0';
strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf));
        /* --- n-1 consecutive vectors ---------------------- */
strcpy(ovecBuf, vecBuf);
for(i=1; i<vecNum; i++) {
   for(j=0; j<regLen; j++) {    /* read in next vector */
        fscanf(vecf, "%d", &bitVal);
        switch (bitVal) {
          case 0: vecBuf[j]='0'; break;
          case 1: vecBuf[j]='1'; break;
          default:
             printf("error vecfile:%s,vec=%d,bit=%d\n",vecFID,i,j);
             exit(4);
        }
   }
   vecBuf[regLen]='\0';
   strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf));
   for(j=0; j<regLen; j++) {    /* read in expected result */
        fscanf(expf, "%d", &bitVal);
        switch (bitVal) {
          case 0: expBuf[j]='0'; break;
          case 1: expBuf[j]='1'; break;
          case 2: expBuf[j]='x'; break;
          default:
             printf("error expfile:%s,vec=%d,bit=%d\n",expFID,i,j);
             exit(5);
        }
   }
   expBuf[regLen]='\0';

   if (xstrcmp(expBuf, resBuf)) {
        printf("error in %dth vector,chip=%s,type=%s\n",i-1,
                                              chipID,chipType);
        printf("vector = %s \n", ovecBuf);
        printf("result = %s \n", resBuf);
        printf("expect = %s \n", expBuf);
        printf(" apply more vector? (no = 0)?");
        scanf("%d", &ans);
        faults++;
        if (ans==0) exit(6);
   }
}
        /* ------ last result    ------------------------- */
strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf)); /*last result */
for(j=0; j<regLen; j++) {       /* read in expected result */
        fscanf(expf, "%d", &bitVal);
        switch (bitVal) {
          case 0: expBuf[j]='0'; break;
          case 1: expBuf[j]='1'; break;
          case 2: expBuf[j]='x'; break;
          default:
             printf("error expfile:%s,vec=%d,bit=%d\n",expFID,i,j);
             exit(7);
        }
}
expBuf[regLen]='\0';
if (xstrcmp(expBuf, resBuf)){
        printf("error detcted last vec,chip=%s,type=%s\n",
                                              chipID,chipType);
        printf("vector = %s \n", vecBuf);
        printf("result = %s \n", resBuf);
        printf("expect = %s \n", expBuf);
        faults++;
}
        /* ---------------------------------------- */
if(faults) printf("%d error(s) detected in chip %s\n",faults,chipID);
else printf("chip=%s (chipType=%s )tested OK.\n", chipID, chipType);
fclose(vecf); fclose(expf);
return(faults);
}

/* ---------------------- fscanN.c ------------ */

int fullscanN(chipID,chipType,ringID,ipre, ipost,pre,post,regNum,
regIns,vecFID,expFID)
char *chipID, *chipType, *regIns, *vecFID, *expFID;
int ringID, ipre, ipost, pre, post, regNum;
{
int xstrcmp();
int i, j, k, bitVal;
int vecNum[MAXCHAINNUM], vecLen[MAXCHAINNUM];
FILE *vecf[MAXCHAINNUM], *expf[MAXCHAINNUM];
char vecBuf[MAXCHAINNUM][BUFSIZ], resBuf[MAXCHAINNUM][BUFSIZ];
char expBuf[MAXCHAINNUM][BUFSIZ];
char vecfid[MAXCHAINNUM][BUFSIZ], expfid[MAXCHAINNUM][BUFSIZ];
char regins[MAXCHAINNUM][BUFSIZ];
char tmpbuf[BUFSIZ];
int faults=0, ans=0, len1;

   i=j=0;                          /* get vecfid[] */
   while(*vecFID !='\0') {
        if(*vecFID==',')(vecfid[i++][j]='\0'; j=0; vecFID++;)
        else(vecfid[i][j++]=*vecFID++;)
   }
   vecfid[i][j++]='\0';
   for(k=0; k<=i; k++)printf("vecfid[%d]=%s\n",k,vecfid[k]);
   i=j=0;                          /* get expfid[] */
   while(*expFID !='\0') {
        if(*expFID==',')(expfid[i++][j]='\0'; j=0; expFID++;)
        else(expfid[i][j++]=*expFID++;)
   }
   expfid[i][j++]='\0';
   for(k=0; k<=i; k++)printf("expfid[%d]=%s\n",k,expfid[k]);
   len1=strlen(regIns)/regNum;
   for(k=0; k<regNum; k++){
        strncpy(regins[k], regIns, len1);
        regins[k][len1]='\0';
        printf("regins[%d]=%s\n",k, regins[k]);
        regIns += len1;
```

```
}
for(i=0; i<regNum; i++) {  /*open files and get vecNum[] and vecLen[]*/
     if ((vecf[i]=fopen(vecfid[i], "r")) == NULL) {
            printf("in fullscanN: file %s does not exist!\n", vecfid[i]);
            exit(1);
     }
     fscanf(vecf[i], "%d %d", &vecNum[i], &vecLen[i]);
     printf("vecf[%d]=%s,vecNum=%d,vecLen=%d\n",i,vecfid[i],vecNum[i],vecLen[i]);
     if((expf[i]=fopen(expfid[i], "r")) == NULL) {
            printf("file: %s deos not exist!\n", expfid[i]);
            exit(1);
     }
}
for(i=0; i<vecNum[0]; i++){
   for(j=0; j<regNum; j++){
     for(k=0; k<vecLen[j]; k++) {    /* read in a vector */
            fscanf(vecf[j], "%d", &bitVal);
            switch (bitVal) {
              case 0: vecBuf[j][k]='0'; break;
              case 1: vecBuf[j][k]='1'; break;
              default:
                 printf("error in reading file: %s, vec=%d, bit=%d\n",
                                 vecfid[j], i, k);
                 vecBuf[j][k]='1';
                 break;
            }
     } /* for k*/
     vecBuf[j][vecLen[j]]='\0';
     scan(ringID,IR,ipre,ipost,regins[j]);
     strcpy(resBuf[j],scan(ringID,DR,pre,post,vecBuf[j]));/*apply vector*/
   } /* for j*/
   for(j=0; j<regNum; j++){
     strcpy(tmpbuf, scan(ringID,IR,ipre,ipost,regins[j]));
     strcpy(tmpbuf,scan(ringID,DR,pre,post,vecBuf[j]));/*get result*/
     printf("tmpbuf=%s\n", tmpbuf);
     strcpy(resBuf[j], tmpbuf);
     for(k=0; k<vecLen[j]; k++){ /* read in expected values */
            fscanf(expf[j], "%d", &bitVal);
            switch (bitVal) {
              case 0: expBuf[j][k]='0'; break;
              case 1: expBuf[j][k]='1'; break;
              case 2: expBuf[j][k]='x'; break;
              default: printf("error reading file:%s,vec=%d,bit=%d\n",
                                        expfid[j], i, k);
                       expBuf[j][k]='x';
                       break;
            }
     }
     expBuf[j][vecLen[j]]='\0';
     printf("before xstrcmp: tmpbuf=%s\n", tmpbuf);
     if (xstrcmp(expBuf[j],tmpbuf) !=0) {    /* check results */
            printf("error detcted,chip=%s,type=%s",chipID,chipType);
            printf("vecNum=%d, regNum=%d, ", i, j);
            printf("vector = %s \n", vecBuf[j]);
            printf("result = %s \n", tmpbuf);
            printf("expect = %s \n", expBuf[j]);
            faults++;
            printf("continue? (1=yes):");
            scanf("%d", &ans);
            if (ans==0) exit(1);
     }
     else {
            printf("matched: ");
            printf("vecNum=%d, regNum=%d, ", i, j);
            printf("vector = %s \n", vecBuf[j]);
            printf("result = %s \n", tmpbuf);
            printf("expect = %s \n", expBuf[j]);
        }
      } /* for j */
   }  /* for i*/
   for(i=0; i<regNum; i++){
     fclose(vecf[i]);
     fclose(expf[i]);
   }
   if(faults) printf("%d error(s) detected in chip %s\n",faults,chipID);
   else printf("chip=%s (chipType=%s )tested OK.\n", chipID, chipType);
   return(faults);
}

/* ---------------- intest.c ---------------------- */
int intest(chipID,chipType,ringID,ipre, ipost,pre,post,regIns,
               vecFID,expFID,ckType, ckNum, ckfNum)
char *chipID, *chipType, *regIns, *vecFID, *expFID;
int ringID, ipre, ipost,pre, post, ckType, ckNum;
double ckfNum;
{
   int xstrcmp();
   int i, j, bitVal, regLen, vecNum, ans, faults=0;
   FILE *vecf, *expf;
   char ovecBuf[BUFSIZ], vecBuf[BUFSIZ], resBuf[BUFSIZ], expBuf[BUFSIZ];
        if ((vecf=fopen(vecFID, "r")) == NULL) {
               printf("vec file: %s deos not exist!\n", vecFID);
               exit(1);
        }
        if((expf=fopen(expFID, "r")) == NULL) {
               printf("exp file: %s deos not exist!\n", expFID);
               exit(2);
        }
        fscanf(vecf, "%d %d", &vecNum, &regLen);
        /* set up intest instruction */
        scan(ringID, IR, ipre, ipost, regIns);
               /* ---- 1st vactor ---------- */
        for(j=0; j<regLen; j++) {       /* read in 1st vector */
               fscanf(vecf, "%d", &bitVal);
               switch (bitVal) {
                 case 0: vecBuf[j]='0'; break;
                 case 1: vecBuf[j]='1'; break;
                 default:
                    printf("error vecfile:%s,vec=%d,bit=%d\n",vecFID,i,j);
                    exit(3);
               }
        }
        vecBuf[regLen]='\0';
        strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf));
               /* --- n-1 consecutive vectors ---------- */
        strcpy(ovecBuf, vecBuf);
        for(i=1; i<vecNum; i++) {
                        /* wait for clock applied */
          if (ckType) {           /* TCK */
               waittck(ckNum); }
          else {                  /* FCK */
               waitfck(ckNum); }

          for(j=0; j<regLen; j++) {    /* read in next vector */
               fscanf(vecf, "%d", &bitVal);
               switch (bitVal) {
                 case 0: vecBuf[j]='0'; break;
                 case 1: vecBuf[j]='1'; break;
```

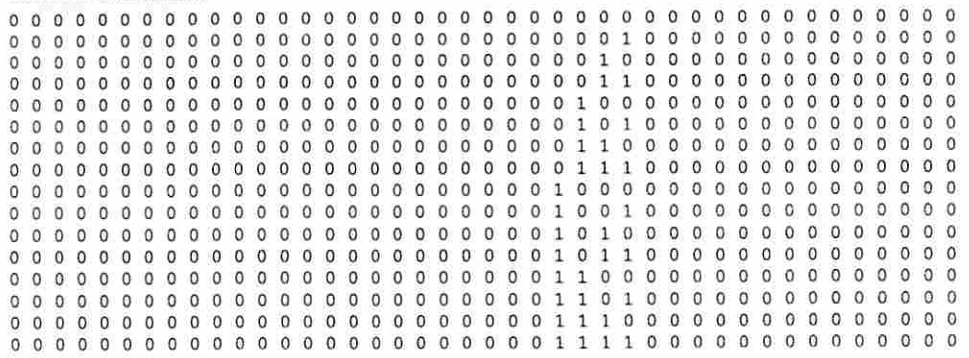
```
            default:
               printf("error vecfile:%s,vec=%d,bit=%d\n",vecFID,i,j);
               exit(4);
         }
   }
   vecBuf[regLen]='\0';
   strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf));
   for(j=0; j<regLen; j++) {    /* read in expected result */
        fscanf(expf, "%d", &bitVal);
        switch (bitVal) {
          case 0: expBuf[j]='0'; break;
          case 1: expBuf[j]='1'; break;
          case 2: expBuf[j]='x'; break;
          default:
             printf("error expfile:%s,vec=%d,bit=%d\n",expFID,i,j);
             exit(5);
        }
   }
   expBuf[regLen]='\0';

   if (xstrcmp(expBuf, resBuf)) {
        printf("error in %dth vector,chip=%s,type=%s\n",i-1,
                                                chipID,chipType);
        printf("vector = %s \n", ovecBuf);
        printf("result = %s \n", resBuf);
        printf("expect = %s \n", expBuf);
        printf(" apply more vector? (no = 0)?");
        scanf("%d", &ans);
        faults++;
        if (ans==0) exit(6);
   }
}
        /* ------ last result    -------------------------  */
if(ckType) {            /* TCK */
        waittck(ckNum); }
else {                  /* FCK*/
        waitfck(ckfNum); }
strcpy(resBuf,scan(ringID, DR, pre, post, vecBuf)); /*last result */
for(j=0; j<regLen; j++) {       /* read in expected result */
        fscanf(expf, "%d", &bitVal);
        switch (bitVal) {
          case 0: expBuf[j]='0'; break;
          case 1: expBuf[j]='1'; break;
          case 2: expBuf[j]='x'; break;
          default:
             printf("error expfile:%s,vec=%d,bit=%d\n",expFID,i,j);
             exit(7);
        }
}
expBuf[regLen]='\0';
if (xstrcmp(expBuf, resBuf)){
        printf("error detcted last vec,chip=%s,type=%s\n",
                                                chipID,chipType);
        printf("vector = %s \n", vecBuf);
        printf("result = %s \n", resBuf);
        printf("expect = %s \n", expBuf);
        faults++;
}
        /* ------------------------------------- */
fclose(vecf); fclose(expf);
if(faults) printf("%d error(s) detected in chip %s\n",faults,chipID);
else printf("chip=%s(chipType=%s)tested OK using INTEST.\n",
                                                chipID,chipType);
return(faults);
}

/* -------------- runbist.c ------------ */
int runbist(chipID,chipType,ringId,ipre,ipost,pre,post,
                useIns, ckType, ckNum, ckfNum, resNum, resIns, expVal)
int ringId, ipre, ipost, pre, post, ckType, ckNum,  resNum;
double ckfNum;
char *chipID, *chipType, *useIns, *resIns, *expVal;
{
   int i, len1, len2, faults=0;
   char *inbuf, outbuf[BUFSIZ];

        printf("runbist: useIns=%s\n", useIns);
        printf("         resIns=%s\n", resIns);
        printf("         expVal=%s\n", expVal);

        inbuf=scan(ringId, IR, ipre, ipost, useIns);
        if (ckType) {           /* TCK */
                waittck(ckNum); }
        else {                  /* FCK */
                waitfck(ckNum); }
        len1= strlen(resIns)/resNum;
        for(i=0; i<resNum; i++){
           xstrncpy(outbuf, resIns, len1);
           resIns += len1;
           printf("extracted resIns=%s \n", outbuf);
           inbuf=scan(ringId, IR, ipre, ipost, outbuf);
           len2 = xextract(outbuf, expVal);
           expVal += len2 +1;
           printf("extracted expVal =%s \n", outbuf);
           inbuf=scan(ringId, DR, pre, post, outbuf);
           if (xstrcmp(outbuf, inbuf)) { /* not the same */
                faults++;
                printf("diff in result and expected result \n");
                printf("expect  = %s \n", outbuf);
                printf("received= %s \n", inbuf);
           }
        }
        if (faults) printf("chip %s failed inrunbist.\n", chipID);
        else printf("chip %s tested OK using runbist.\n", chipID);
        return(faults);
}

waittck(num)
int num;
{
int i;
        for(i=0; i<num; i++);
}
waitfck(fnum)
double fnum;
{
double i;
        for(i=1.0; i<fnum; i +=1.0);
}
```

```
/* driver.c             -- working version -- Lien 8/5/90 */
/*          -- change lead and trailing zeros to ones 4/2/91 */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#define MAXVECLEN 1024
#define IR 1
#define DR 0    /* check this again ?????? */
#define WAITTCLEN 30    /* wait cycle for TC sync */

char *scan(ringID, type, preamble, postamble, optr)
  int ringID, type, preamble, postamble;
  char *optr;
    /*ringID 0 is default, type = 1 is sanIR, (0 for DR)  */
    /*amble:# of extra stages,optr points to strings of 0s, 1s */
{
  extern unsigned int getOutWord(), min(), readReg();
  extern void saveInWord(), writeReg();
  unsigned int outWord, inWord, i, j, shifts, total_shifts;
  unsigned int crCode, vecLength;
  char *tmp1,*tmp2,buf[MAXVECLEN],xmtArray[MAXVECLEN],rcvArray[MAXVECLEN];
  unsigned int tc, cnr, bitCount;
#ifdef DEBUG
  char *rxrp, rxr[MAXVECLEN];
#endif

   vecLength=strlen(optr);
  if (type == IR) {
    total_shifts = vecLength + preamble + postamble;
    for(i=0; i<preamble; i++) xmtArray[i]='1';
    xmtArray[i]='\0';
    for(i=0; i<vecLength; i++) xmtArray[preamble+i] = *optr++;
    for(i=0; i<postamble; i++) xmtArray[preamble+vecLength+i]= '1';
    xmtArray[preamble+vecLength+postamble]='\0';
  }
  else{
   if (preamble > postamble) {
        total_shifts = vecLength + preamble;
        for(i=0; i<(preamble-postamble); i++)
                xmtArray[i]='1';                /* insert leading 1 */
        xmtArray[i]='\0';
        tmp1=&xmtArray[i];
        strcpy(tmp1, optr);
        }
   else if (preamble < postamble) {
        total_shifts = vecLength + postamble;
        strcpy(xmtArray, optr); /* no leading 1 */
        for (i=vecLength; i<vecLength + postamble; i++)
                xmtArray[i]='1';
        xmtArray[vecLength+postamble]='\0';
        }
   else {       /* preamble=postamble */
        total_shifts = vecLength + postamble;
        strcpy(xmtArray, optr); /* no leading 1 */
        for (i=vecLength; i<vecLength + postamble; i++)
                xmtArray[i]='1';
        xmtArray[vecLength+postamble]='\0';
   }
 } /* if type == IR */
/* now output string is ready in xmtArray, modified len=total_shifts */
/* -------- tc can't work total_shifts % 16 == 1   ------ */
   if ((total_shifts % 16) == 1) {
        buf[0]='1'; /* add 1 bit leading 1 */
        buf[1]='\0';
        strcat(buf, xmtArray);
        tmp1=buf;
   }
   else   tmp1 = xmtArray;
#ifdef DEBUG
   printf("formatted output string is %s \n", tmp1);
#endif
   bitCount = strlen(tmp1);
   tmp2 = rcvArray;
   writeReg(0x308, 0);       /* disable FEN */
        /* CR= function of (ins, ringID) */
   if (ringID==0) crCode=0x20; else crCode=0x30;
   if (type == 1)  crCode += 4; /* type =1 scanIR, type =0 scanDR */
   writeReg(0x300, crCode);    /* load CR = 100100 */
   tc = bitCount-2;       /* determine the value of tc and cnr */
   cnr=14;
/*
   if (bitCount < 16) cnr = bitCount;
   else cnr = 14;
*/
#ifdef DEBUG
   printf("tc=%d, cnr=%d \n", tc, cnr);
#endif
   writeReg(0x301, cnr);    /* CNR=1110,or 14 for shifting 16 bits*/
   writeReg(0x304, tc); /* TCL=# shifts-2, TCH not exist */
   outWord=getOutWord(tmp1);
   shifts=min(16,strlen(tmp1));
   tmp1 += shifts;
   writeReg(0x305, outWord);/* load TxR=outword */
   writeReg(0x309, 0);       /* clear SR */
   writeReg(0x306, 0);       /* clear RxR */
   writeReg(0x308, 1);       /* enable FEN */
   for(;(inp(0x300)& 0x0f)==0;);/* wait for event occurs */
              /* ----- need error checking for total_shifts <3 */
   writeReg(0x308, 0); /*disable FEN */
   writeReg(0x309, 0); /* clear SR */
   inWord=readReg(0x306); /* read RxR */
#ifdef DEBUG
   rxrp=rxr;
   saveInWord(inWord, rxrp, 16);
   rxr[16]='\0';
   printf("rxr=%s \n",rxrp);
#endif
   saveInWord(inWord, tmp2, shifts);
   tmp2 += shifts;
   while (strlen(tmp1) >0) {
        outWord=getOutWord(tmp1);
        shifts=min(16,strlen(tmp1));
        tmp1 += shifts;
        writeReg(0x305, outWord);/* load TxR=outword */
        writeReg(0x308, 1);       /* enable FEN */
        for(;(inp(0x300)& 0x0f) ==0;);/* wait for event occurs */
        writeReg(0x308, 0); /*disable FEN */
        writeReg(0x309, 0); /* clear SR */
        inWord=readReg(0x306); /* read RxR */
#ifdef DEBUG
        rxrp=rxr;
        saveInWord(inWord, rxrp, 16);
        rxr[16]='\0';
        printf("RxR= %s \n", rxrp);
#endif
        writeReg(0x306, 0);    /* clear RxR */
        saveInWord(inWord, tmp2, shifts);
        tmp2 += shifts;
```

```
   } /* while */
   *(tmp2)='\0';
   if ((total_shifts % 16) == 1) *(tmp2-1)='\0';/* discard 1 trailing bit */
  if(type==IR){
    tmp2=rcvArray + preamble;  /* discard preamble bits */
    *(tmp2 + vecLength) = '\0'; /* discard postamble bits */
  } /* if type ==IR */
  else{
   tmp2 = rcvArray+preamble;/*discard leading bits (preamble) received */
   if (preamble < postamble) *(tmp2+vecLength)='\0'; /* omit trailing bits*/
  }
   return(tmp2);
}

unsigned int getOutWord(ptr)
  char *ptr;
{
  unsigned int intPower();
  unsigned int i,j,len, outWord=0;
    len=strlen(ptr);
    for(i=16; i>0 && len>0; i--, ptr++, len--){
        j= *ptr - '0';
        outWord += j* intPower(2, i-1);
    }
    return(outWord);
}

void saveInWord(inWord, ptr, len)
unsigned int inWord, len;
char *ptr;
{
  unsigned int template, i, j;
  unsigned int intPower();
  char newBit;
     if (len <16) {
        j=intPower(2,len);
        if (inWord >= j) template = inWord % j;  /*delete leading bits*/
        else template = inWord;
     }
     else template=inWord;
     for(i=len; i>0; i--, ptr++) {
        j=intPower(2,i-1);
        if (template >= j )     newBit='1';
        else    newBit='0';
        *ptr = newBit;
        if (newBit='1') template %= j;
     }
}

unsigned int intPower (base, pow)
unsigned int base, pow;
{
  unsigned int i, j, value=1;
       for (i=0; i<pow; i++) value *= base;
       return(value);
}
unsigned int min(len1, len2)
unsigned int len1, len2;
{
  if (len1 <= len2) return(len1);
  else return(len2);
}

/* rwReg.c              --- by J.C. Lien 7/26/90 */
```

```
              /* This is the only machine dependent part
                 of the test channel control programs. */
                    /* write to a 16 bit TC register
                       using 8 bit host data bus    */
void writeReg(portid, outword)
unsigned int portid,outword;
{
   int i;
   outp(0x30f, outword / 256); /* high byte */
   outp(portid, outword % 256);/* low byte */
   wait(WAITTCLEN);  /* wait for tc synchronized */
}

wait(period)
   int period;
{
   int i;
   for(i=0; i<period; i++);
}
                          /* read from a 16 bit regsiter in TC */
unsigned int readReg(portid)
unsigned int portid;
{
   unsigned int lowByte, highByte;
   lowByte=inp(portid);
   highByte=inp(0x30f);
   return(lowByte+highByte*256);
}

/* other functions */
/* ------------------------------------- */
char *sample_ring(ringid, irslen, brslen)
int ringid, irslen, brslen;
{
  char p1[BUFSIZE], p2[BUFSIZE];
  int i;
  for(i=0; i<irslen; i++) p1[i]='1'; /* set bypass mode */
  p1[irslen]='\0'; /* terminate the string */
  strcpy(p2, scan(ringid, IR, 0, 0, p1)); /*set up sample mode*/
                  /* p2 is status now, throw it away */
  for(i=0; i<brslen; i++) p2[i]='1';
  p2[brslen]='\0'; /* terminate the string */
  strcpy(p1, scan(ringid, DR, 0, 0, p2));
  printf("ring=%d, sampled BSR value=%s \n", ringid, p1);
  return(p1); /*return the sampled BSR value*/
}

/* ------------------------------------------- */
void reset_ring(ringID, period)
unsigned int ringID, period;
{
int i, j;
   void writeReg();
   unsigned int crCode;

   writeReg(0x308, 0);       /* disable FEN */
        /* CR= function of (ins, ringID) */
   if (ringID==0) crCode=0x27;
   else crCode=0x37;
   writeReg(0x300, crCode);   /* load CR = 100111 */
   writeReg(0x301, 0x0e);   /* CNR=1110,or 14 for shifting 16 bits*/
   writeReg(0x304, period-1); /* TCL=# period-1, TCH not exist */
   writeReg(0x309, 0);       /* clear SR */
   writeReg(0x308, 1);       /* enable FEN */
```

```
   for(;(inp(0x300)& 0x0f)==0;);/* wait for event occurs */
   writeReg(0x308, 0); /*disable FEN */
   writeReg(0x309, 0); /* clear SR */
}

/* ------------------------------------------- */
void runtest_ring(ringID, period)
unsigned int ringID, period;
{
   void writeReg();
   unsigned int crCode;
   writeReg(0x308, 0);      /* disable FEN */
        /* CR= function of (ins, ringID) */
   if (ringID==0) crCode=0x25;
   else crCode=0x35;
   writeReg(0x300, crCode);   /* load CR = 100101 */
   writeReg(0x301, 0x0e);   /* CNR=1110,or 14 for shifting 16 bits*/
   writeReg(0x304, period-1); /* TCL=# period-1, TCH not exist */
   writeReg(0x309, 0);      /* clear SR */
   writeReg(0x308, 1);      /* enable FEN */
   for(;(inp(0x300)& 0x0f)==0;);/* wait for event occurs */
   writeReg(0x308, 0); /*disable FEN */
   writeReg(0x309, 0); /* clear SR */
}

/* ------------------------------------------- */

/* change the TCK frequency */
/* fac=0: TCK = clock div 2
   fac=1: TCK = clock div 4
   fac=2: TCK = clock div 8
   fac=3: TCK = clock div 16 */
void changeTCK(fac)
int fac;
{
  int i;
  if((fac <0) || (fac >3)){
    printf("changeTCK:factor=%d out of range.\n", fac);
    exit(2);
  }
  outp(0x307, fac);  /* address for clock freq selection*/
  for(i=0; i=30; i++); /* wait for synchronization */
}

/* ------------------------------------------- */
/*
  This functions cannot be realized in the current prototype of
  MMC since the TCK is a free-running clock. If the clock TCk
  can be controlled (or disabled) then this function can be
  implemented accordingly.  ------ Lien 3/22/91
*/

applytck(clkn) /* apply a number of tck clock */
int clkn;
{
  int i, j=0;    /*just wait so that the TCK cycles is enough.*/
  for(i=0; i<clkn; i++) j=j;
}
```



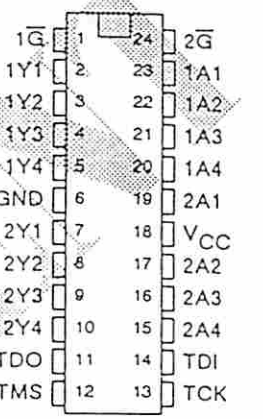
```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 0 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

E Data Sheet of TI SN74BCT8244

REV 0.4

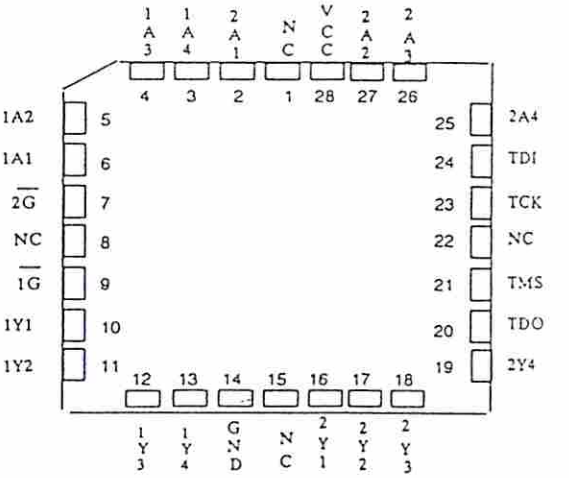
- Device is a member of Texas Instruments SCOPE™ Family of Testability Products
- Octal Test Integrated Circuit
- Compatible with the Proposed IEEE P1149.1 Serial Test Bus
- Functionally Equivalent to 54/74F244 and 54/74BCT244 in the Normal Function Mode
- Implements Optional "Test Reset" Signal on TAP by Recognizing a Double-High on TMS Pin
- Test Operation Synchronous to Test Access Port (TAP)
- 16 Test Instructions – Conforms to the Proposed JTAG Boundary Scan – Provides Data Compression of Inputs – Provides Pseudo-Random Pattern Generation from Outputs – Output Toggle Boundary Mode – Outputs to High Impedance State Mode
- Package Options Include "Small Outline" Packages, Ceramic Chip Carriers, and Standard Plastic and Ceramic 300-mil DIPs

SN54BCT8244...JT PACKAGE
SN74BCT8244...DW OR NT PACKAGE
(TOP VIEW)



SN54BCT8244 . . . FK Package

(TOP VIEW)



description

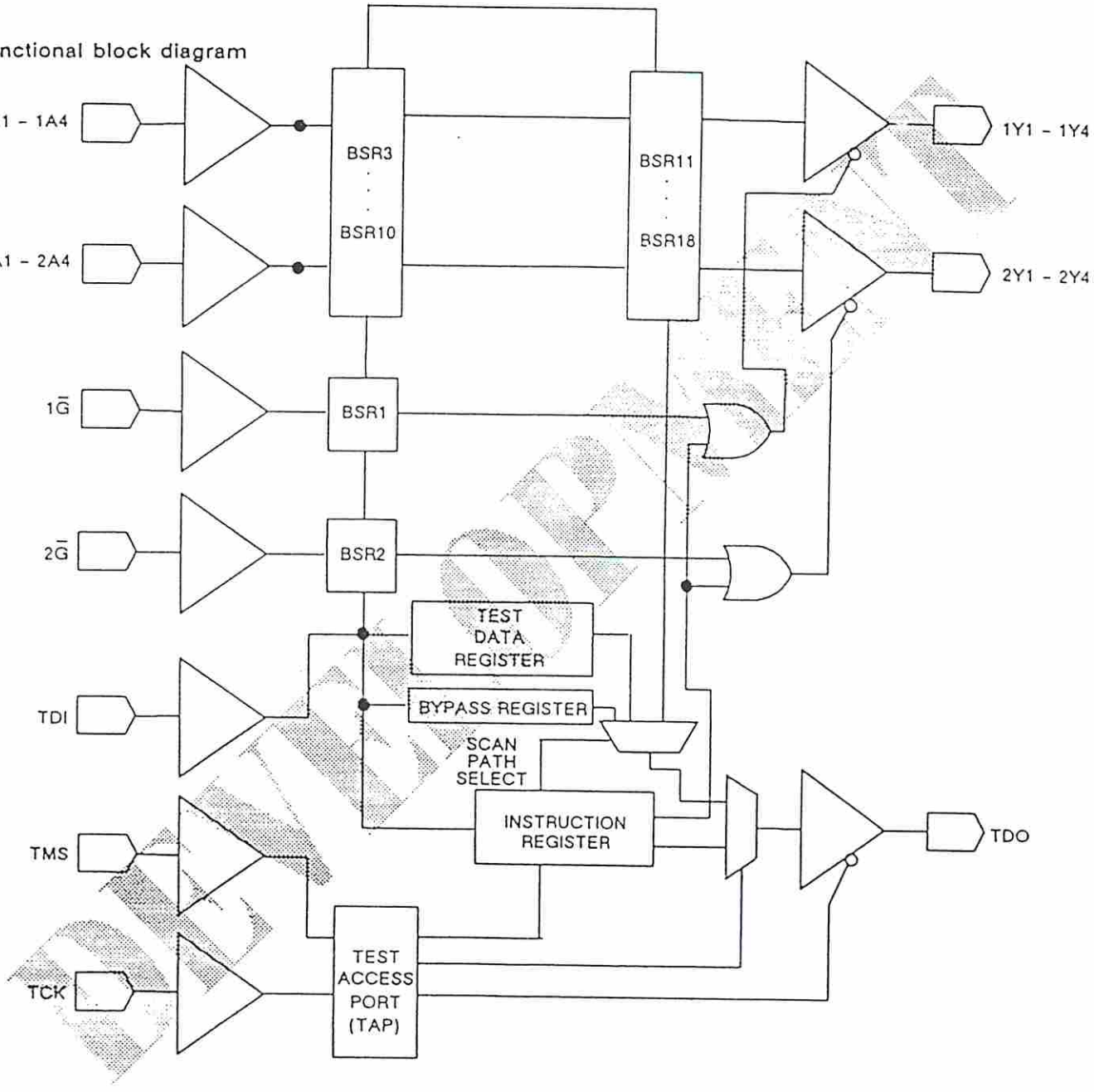
The SN54BCT8244 and SN74BCT8244 are members of Texas Instruments SCOPE™ testability IC family. This family of components blend test circuitry with standard logic functions to facilitate testing of complex circuit board assemblies. Scan access to the test circuitry is accomplished via the 4-wire Test Access Port (TAP) interface.

In normal mode these devices are functionally equivalent to the SN54/74F244 and SN54/74BCT244 octal buffers. In normal mode the test circuitry can be activated by the TAP to take snapshot samples of the data appearing at the device pins or to perform a self test on the boundary test cells. Activating the TAP in normal mode does not affect the functional operation of the SCOPE octal buffers.

In test mode the normal operation of the SCOPE octal buffer is inhibited and the test circuitry is enabled to observe and control the device's I/O boundary. When enabled, the test circuitry can perform boundary scan test operations as described in the proposed JTAG/P1149.1 specification. Additionally, the test circuitry can perform other testing functions such as: parallel signature analysis on data inputs and pseudo-random pattern generation from data outputs. All testing and scan operations are synchronized to the TAP interface.

The SN54BCT8244 is characterized for operation over the full military temperature range of −55° C to 125° C. The SN74BCT8244 is characterized for operation from 0 °C to 70° C.

functional block diagram



FUNCTION TABLE

OUTPUT CONTROL	DATA INPUT	OUTPUT
1$\overline{G}$, 2$\overline{G}$	A	Y
H	X	Z
L	L	L
L	H	H

absolute maximum ratings over operating free-air temperature range (unless otherwise noted) †

Supply voltage, V_{CC} . . . . . -0.5 V to 7 V
Input voltage . . . . . -0.5 V to 7 V
Voltage applied to any output in the disabled or power-off state . . . . . -0.5 V to 5.5 V
Voltage applied to any output in the high state . . . . . -0.5 V to V_{CC}
Current into any output in the low state: SN54BCT8244 . . . . . 96 mA
SN74BCT8244 . . . . . 128 mA
Operating free-air temperature range: SN54BCT8244 . . . . . -55°C to 125°C
SN74BCT8244 . . . . . 0°C to 70°C
Storage temperature range. . . . . -65°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

recommended operating conditions

		SN54BCT8244			SN74BCT8244			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC} Supply voltage		4.5	5	5.5	4.5	5	5.5	V
V_{IH} High-level input voltage		2		5.5	2		5.5	V
V_{IHH} Double high-level input voltage	TMS	10.25	10.50	10.75	10.25	10.50	10.75	V
V_{IL} Low-level input voltage				0.8			0.8	V
I_{IK} Input clamp current				-18			-18	mA
I_{OH} High-level output current				-12			-15	mA
I_{OL} Low-level output current				48			64	mA
T_A Operating free-air temperature		-55		125	0		70	°C

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		SN54BCT8244			SN74BCT8244			UNIT
			MIN	TYP†	MAX	MIN	TYP†	MAX	
V_{IK}	V_{CC} = 4.5V, I_I = -18 mA				-1.2			-1.2	V
V_{OH}	V_{CC} = 4.5 V, I_{OH} = -3 mA		2.4	3.3		2.4	3.3		V
	V_{CC} = 4.5 V, I_{OH} = -12 mA		2	3.2					V
	V_{CC} = 4.5 V, I_{OH} = -15 mA					2	3.1		V
V_{OL}	V_{CC} = 4.5 V, I_{OL} = 48 mA			0.38	0.55				V
	V_{CC} = 4.5 V, I_{OL} = 64 mA						0.42	0.55	V
I_I	V_{CC} = 5.5 V, V_I = 5.5 V				0.1			0.1	mA
I_{IH}	V_{CC} = 5.5 V, V_I = 2.7 V				20			20	uA
I_{IHH}	TMS	V_{CC} = 5.5 V, V_I = 10.50 V			1			1	mA
I_{IL}	V_{CC} = 5.5 V, V_I = 0.5 V				-1			-1	mA
I_{OZH}	V_{CC} = 5.5 V V_O = 2.7 V				50			50	uA
I_{OZL}	V_{CC} = 5.5 V V_O = 0.5 V				-50			-50	uA
I_{OS}‡	V_{CC} = 5.5 V V_O = 0		-100		-225	-100		-225	mA
I_{CC}	V_{CC} = 5.5 V, Outputs open	Outputs high		5.5			5.5		mA
		Outputs low		52			52		mA
		Outputs disabled		2.3			2.3		mA
C_I	V_{CC} = 5.0 V, V_I = 2.5 V or 0.5 V								pF
C_o	V_{CC} = 5.0 V, V_O = 2.5 V or 0.5 V								pF

†All typical values are at V_{CC} = 5 V, T_A = 25°C.
‡Not more than one output should be shorted at a time, and the duration of the short circuit should not exceed one second.

'BCT8244 switching characteristics (see Note 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC} = 5 V, C_L = 50 pF, R1 = 500 Ω, R2 = 500 Ω, T_A = 25°C			V_{CC} = 4.5 V to 5.5 V, C_L = 50 pF, R1 = 500 Ω, R2 = 500 Ω, T_A = MIN to MAX†				UNIT
			'BCT8244			SN54BCT8244		SN74BCT8244		
			MIN	TYP	MAX	MIN	MAX	MIN	MAX	
f_{max}	TCK			20						MHz
t_{PLH}	ANY A	Y		5.9						ns
t_{PHL}	ANY A	Y		6.5						ns
t_{PLH}	TCK↓	Y		11.5						ns
t_{PHL}	TCK↓	Y		11						ns
t_{PLH}	TCK↓	TDO		9.8						ns
t_{PHL}	TCK↓	TDO		9.7						ns
t_{PZH}	Any $\overline{G}$	Y		5.6						ns
t_{PZH}	TCK↓	Y		13						ns
t_{PZH}	TCK↓	TDO		8						ns
t_{PZL}	Any $\overline{G}$	Y		7.7						ns
t_{PZL}	TCK↓	Y		14						ns
t_{PZL}	TCK↓	TDO		9.2						ns
t_{PHZ}	Any $\overline{G}$	Y		7						ns
t_{PHZ}	TCK↓	Y		13						ns
t_{PHZ}	TCK↓	TDO		7						ns
t_{PLZ}	Any $\overline{G}$	Y		6.7						ns
t_{PLZ}	TCK↓	Y		13						ns
t_{PLZ}	TCK↓	TDO		7.8						ns

† For conditions shown as MIN or MAX, use the appropriate value specified under Recommended Operating Conditions.
NOTE 1: See General Information for load circuits and waveforms.

timing requirements

		V_{CC} = 5 V, T_A = 25°C			V_{CC} = 4.5 V to 5.5 V, T_A = MIN to MAX†				UNIT
		'BCT8244			SN54BCT8244		SN74BCT8244		
		MIN	TYP	MAX	MIN	MAX	MIN	MAX	
f_{clock}	TCK	0							MHz
t_w	pulse duration TCK high or low								ns
t_{su}	Setup time, TMS before TCK↑	9							ns
t_{su}	Setup time, TDI before TCK↑	9							ns
t_{su}	Setup time, Any A before TCK↑	9							ns
t_{su}	Setup time, Any $\overline{G}$ before TCK↑	9							ns
t_h	Hold time, TMS after TCK↑	5							ns
t_h	Hold time, TDI after TCK↑	5							ns
t_h	Hold time, Any A after TCK↑	5							ns
t_h	Hold time, Any $\overline{G}$ after TCK↑	5							ns
t_{pu}	Wait time, power-up to TCK↑	100							ns

† For conditions shown as MIN or MAX, use the appropriate value specified under Recommended Operating Conditions.
NOTE 1: See General Information for load circuits and waveforms.

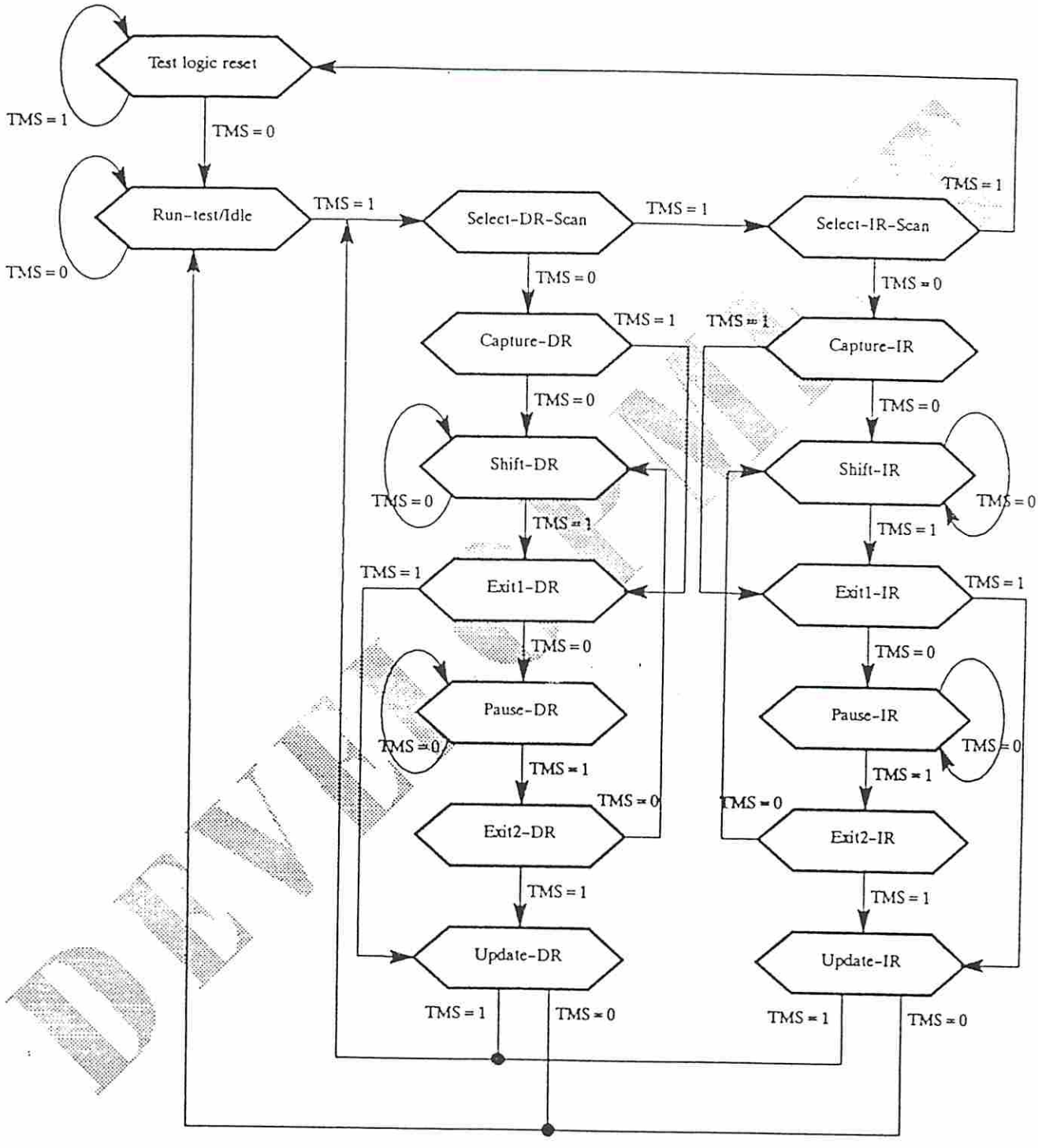


FIGURE 1: TAP State Diagram

functional description

JTAG test information is conveyed by means of a 4-wire test bus. Test commands, test data, circuit state control instructions and synchronous control signals are all passed along the 4-wire bus. The function of the TAP is to extract the state control information and synchronous control signals from the 4-wire test bus, and generate the appropriate on-chip control signals for the JTAG test structures on the device. To accomplish this, the TAP cell monitors two signals from the 4-wire test bus - TCK (the JTAG Test Clock) and TMS (the JTAG Test Mode Select line). The functional block diagram on page 2 illustrates the JTAG 4-wire test bus and boundary scan architecture, and the relationship between the TAP cell, the 4-wire bus, and the various boundary scan test elements.

architectural elements

boundary scan register BSR0 - BSR17

The boundary scan register contains eighteen (18) bits - one for each functional input or output pin of the device. FIGURE 2 illustrates the order of bits in the boundary scan register scan path. The boundary scan registers allow for board interconnect testing, defining conditions at the device logic periphery, and sampling data on the functional input or output pins without disturbing normal device operations.

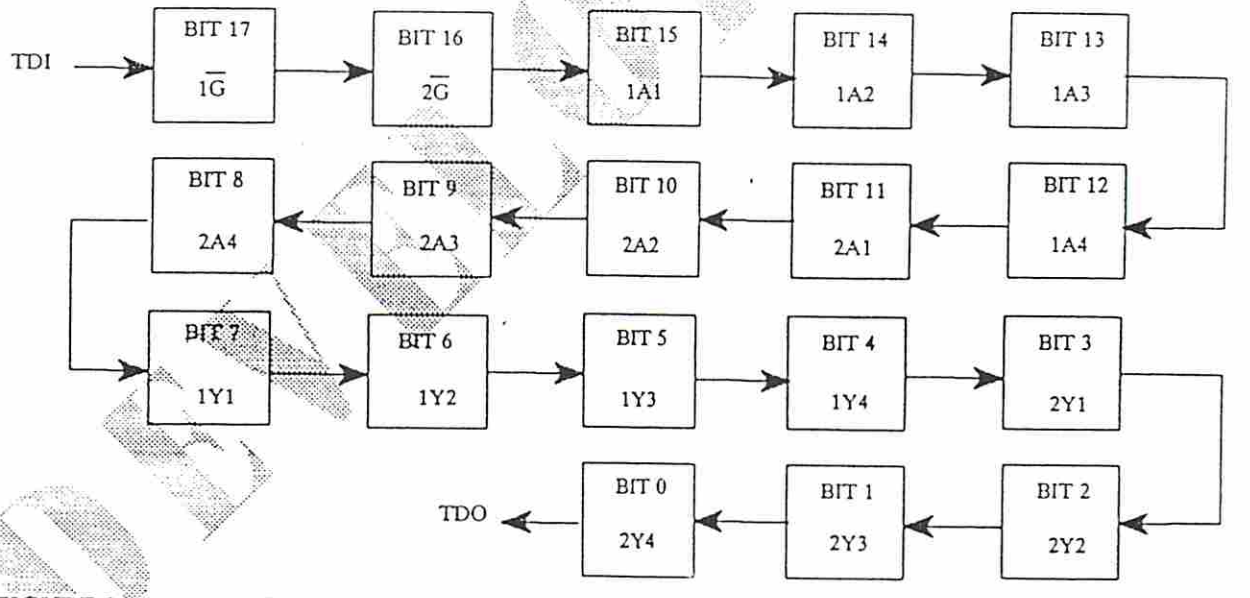


FIGURE 2: Boundary Scan Register Order of Scan

bypass register

The bypass register contains one (1) bit for use when the device is in the bypass scan mode as defined in TABLE 3. FIGURE 3 illustrates the flow through the bypass register. This register provides a short one bit scan path through the device rather than scanning through the eighteen bit boundary scan register path. This is especially useful for decreasing test access times to a particular device on a board with several JTAG compatible devices which are not required for a specific test.

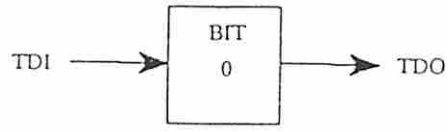


FIGURE 3: Bypass Register Order of Scan

test data register

The test data register contains two (2) bits used to control test operations occurring at the boundary. FIGURE 4 illustrates the order of bits in the test data register scan path.

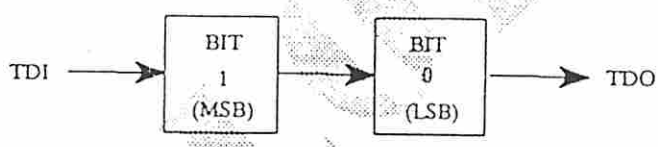


FIGURE 4: Test Data Register Order of Scan

In addition to the boundary test instructions shown in TABLE 3, additional test operations shown in TABLE 1 can be performed when the run test opcode is installed in the instruction register. These test operations include: pseudo-random pattern generation (PRPG) and parallel signature analysis (PSA) as shown in TABLE 1 when the run test opcode is installed in the instruction register.

OPCODE MSB->LSB	TEST
00	SAMPLE INPUTS / TOGGLE OUTPUTS
01	PRPG / 16 BIT MODE
10	PSA / 16 BIT MODE
11	SIMULTANEOUS PRPG AND PSA / 8 BIT MODE

TABLE 1: Run Test Opcodes

example

In order to implement the sample inputs / toggle outputs opcode from TABLE 1, a series of operations must be performed. Refer to FIGURE 1 to trace these operations through the TAP state diagram. The select-IR path is used to shift opcodes into the instruction register. The select-DR path is used to shift data into the boundary scan register or bypass register, or to shift opcodes into the test data register. To shift data or opcodes into the registers, after entering the appropriate shift-DR or shift-IR state TMS must be held low for enough TCK pulses to shift the correct number of bits into the registers.

First, the boundary read opcode (test or normal mode) must be loaded into the instruction register using the select-IR scan path, then the boundary scan registers may be initialized using the select-DR scan path. Load the test data register scan opcode (test or normal mode) into the instruction register using the select-IR scan path, then the sample inputs / toggle outputs opcode (00) may be entered into the test data register using the select-DR-scan path. Finally, the boundary run test opcode (test or normal mode) must be entered into the instruction register using the select-IR scan path. Exiting the select-IR-scan path to the run-test / idle state starts the outputs toggling. As long as the device remains in the run-test / idle state, each TCK pulse will cause the device's function outputs to toggle to the opposite state.

tap bits

Tap bit settings used for PSA and PRPG test operations are shown in TABLE 2. The use of these tap bits as well as the shift operations necessary to perform 8 bit and 16 bit PSA and PRPG test operations is described in FIGURE 5 through FIGURE 7.

OPERATION	MODE	TAP BITS A - - > Y
PSA	8 BIT	1A2, 1A3, 1A4, 2A4
	16 BIT	2A3, 1Y1, 1Y4, 2Y4
PRPG	8 BIT	1Y2, 1Y3, 1Y4, 2Y4
	16 BIT	2A3, 1Y1, 1Y4, 2Y4

TABLE 2: Tap Bit Settings for PSA and PRPG Test Operations

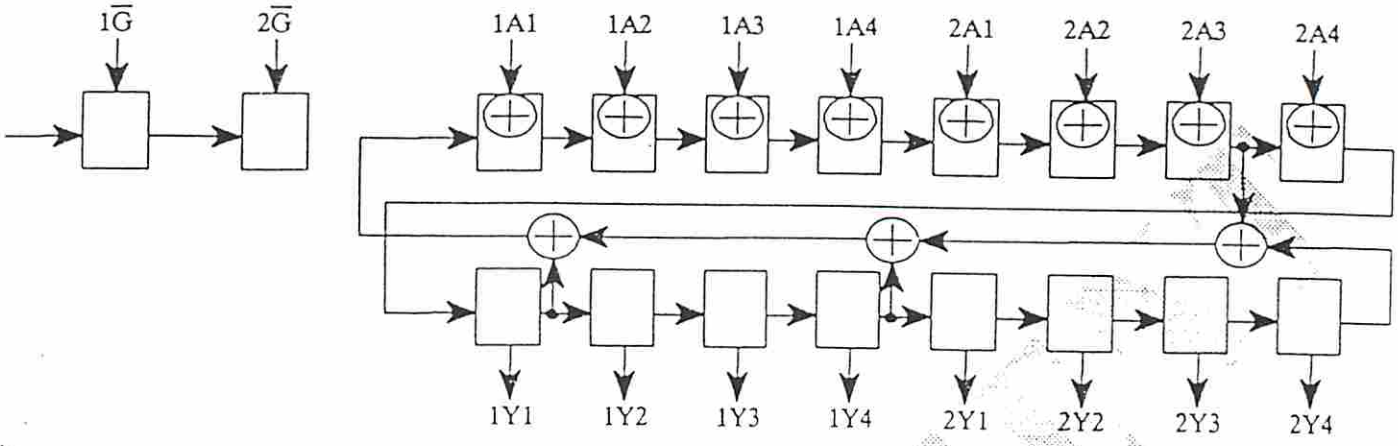


FIGURE 5: 16 Bit PSA configuration during RUN TEST/IDLE state.
A PSA operation on the 8 data inputs proceeds as the 8 data outputs are held static.

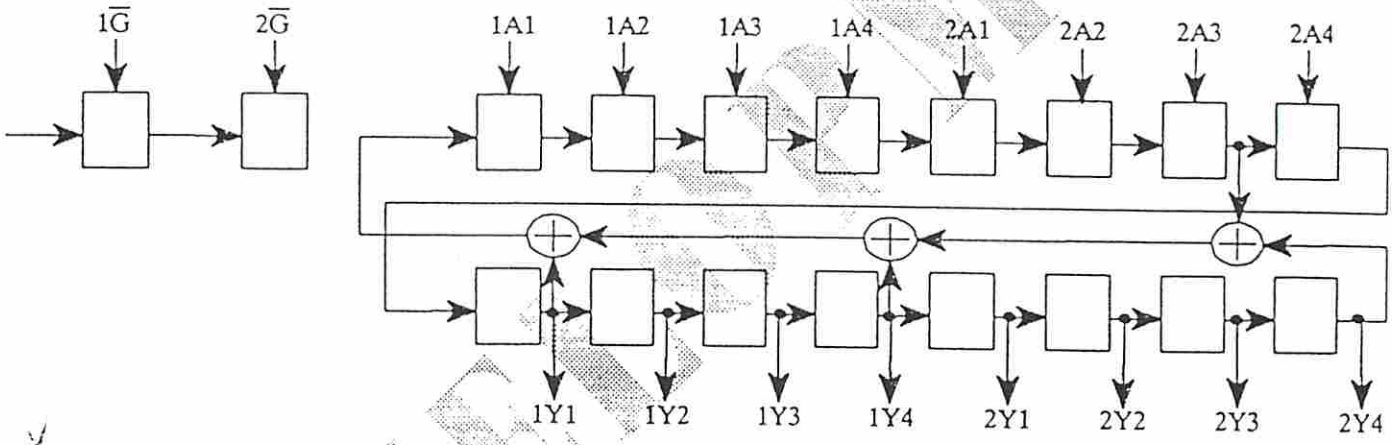


FIGURE 6: 16 Bit PRPG configuration during RUN TEST/IDLE state.
A PRPG operation from the 8 data outputs proceeds while the 8 data inputs are ignored.

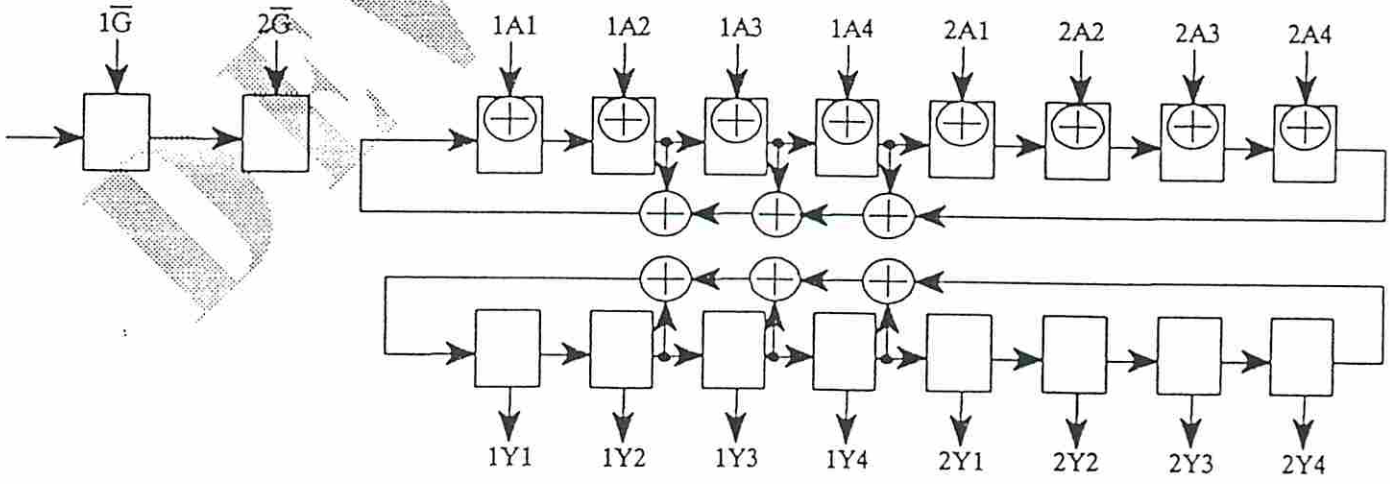
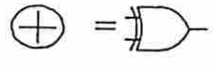


FIGURE 7: 8 Bit PSA and PRPG configuration during RUN TEST/IDLE state.
Simultaneously, an 8 bit PSA operation proceeds on the 8 data inputs, while an 8 bit PRPG operation proceeds from the 8 data outputs.



instruction register

The test device instruction register is 8 bits in length. When in the Shift–IR state, data can be scanned into the register from the most significant bit (MSB) to the least significant bit (LSB) as shown in FIGURE 8. The instruction register controls the internal device structures and test operations according to the opcodes listed in TABLE 3.

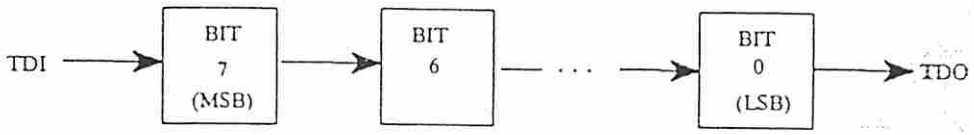


FIGURE 8: Instruction Register Order of Scan

instruction set

The 'BCT8244 uses the 8-bit serial instruction register as its instruction input. TABLE 3 summarizes the 8-bit opcodes and corresponding tests.

TABLE 3: Opcodes (See Note 1)

OPCODE BIT 7 -BIT 0 MSB – LSB	FUNCTION
X0000000	BOUNDARY SCAN
X0000001	ID REGISTER SCAN
X0000010	SAMPLE BOUNDARY
X0000011	BOUNDARY SCAN
X0000100	BYPASS SCAN MODE
X0000101	BYPASS SCAN MODE
X0000110	CONTROL BOUNDARY TO HIGH IMPEDANCE TRIBYP
X0000111	CONTROL BOUNDARY TO 1/0 JTFBYP
X0001000	BYPASS SCAN MODE
X0001001	BOUNDARY RUN TEST/TEST MODE RUNT
X0001010	BOUNDARY READ/NORMAL MODE READBN
X0001011	BOUNDARY READ/TEST MODE READBT
X0001100	BOUNDARY SELFTEST/NORMAL MODE CELLST
X0001101	BOUNDARY TOGGLE OUTPUTS/TEST MODE TOPHIP
X0001110	TEST DATA REGISTER SCAN/NORMAL MODE SCANCN
X0001111	TEST DATA REGISTER SCAN/TEST MODE SCANCT
ALL OTHER	BYPASS SCAN MODE

X = Parity Bit (Even Parity)

NOTE 1: If Bit 4 through Bit 6 are all 0, then Bit 0 through Bit 3 are decoded as shown in TABLE 3.

The test functions which are identified in TABLE 3 and performed by the test integrated circuits are defined as follows:

boundary scan

A boundary scan of the boundary scan register is performed according to the methodology designated by the proposed JTAG or the proposed IEEE P1149.1 specifications. This instruction performs a combination of sample boundary and control boundary to 1/0 tests as specified below.

ID register scan

The test circuit is placed in the bypass mode as defined by JTAG in the absence of an ID Register. A logic 0 is loaded into the bypass register before scanning.

sample boundary

Data appearing at the device's function inputs and outputs is sampled and scanned out the TDO pin. This operation is performed in a functional mode without disturbing normal device operations.

control boundary to high impedance

The device's function outputs are placed in the high impedance state. The bypass register is selected in the scan path. Function inputs remain operational.

control boundary to 1/0

Function inputs and outputs are controlled by the boundary register. The bypass register is selected in the scan path.

boundary run test

Test operations controlled by the test data register are performed while the device is in the idle mode. Operations performed in this mode include the following:

parallel signature analysis of inputs

Data appearing on the device's data inputs is compressed by a parallel signature analysis (PSA) operation with fixed tap bits. Data shall be compressed into sixteen bits. The initial seed value for the PSA operation should be scanned into the boundary scan register prior to performing the test operation.

pseudo–random pattern generation from outputs

A pseudo–random data pattern (PRPG) is output from the outputs of the test device. The initial seed value for the PRPG should be scanned into the data register prior to performing the test operation.

simultaneous PSA and PRPG

An 8 Bit PSA of inputs and an 8 Bit PRPG from outputs is performed simultaneously as specified above.

sample inputs / toggle outputs

The devices inputs are sampled on successive rising edges of TCK, while the device's function output pins are toggled simultaneously on successive falling edges of TCK while the device is in the idle mode. This test is intended to be used for parametric testing and pattern generation purposes. The initial pattern should be scanned into the boundary scan register prior to performing the test operation.

boundary read (test mode and normal mode)

Data is scanned in and out of the boundary scan register without first preloading the boundary condition. This function is particularly useful after a PSA operation – the results can be scanned out for review by the test controller.

test data register scan (test mode and normal mode)

The test data register is selected for scan access.

boundary self test

The boundary scan register is preloaded with the inverted contents of the latch memory elements of each boundary scan register bit. The boundary scan register is then scanned out. Prior to performing this test known data should be scanned into the boundary scan register.

boundary toggle outputs

The device's function output pins are toggled simultaneously on successive TCK clock inputs. This test is intended to be used for parametric testing and pattern generation purposes. The initial pattern should be scanned into the data register prior to performing the test operation.

bypass scan mode

The bypass register is selected in the scan path and the device is placed in the normal mode.