The design of a Master Test Controller for a system-level Embedded Boundary-Scan Test architecture

Sven-Åke Andersson
Ericsson AB
Stockholm, Sweden
sven-ake.andersson@ericsson.com

Abstract

Boundary scan (IEEE 1194.1) is today implemented in every ASIC and FPGA and is only used during production test of chips and boards. It is then left unused for the rest of the product life cycle. That is a waste of resources. Adding an on-board JTAG tester will change that scenario. Tests can then be run at power-up of the system or as maintenance tests when the system is up and running. If the test program is stored on the board, tests can also be run at a repair center. The tester can also be used when debugging the board in the lab. All tests controlled from the test access port (TAP) can be executed from the on-board tester.

This paper describes an implementation of an on-board JTAG tester called a Master Test Controller (MTC). The MTC has been designed as an IP block in an system-on-chip design. The ASIC is manufactured by IBM in their 80nm CMOS process. The design takes up no more than 60k gates and can easily be fitted into any ASIC or FPGA design. The MTC is controlled from an embedded PowerPC microprocessor, but can easily be adopted to use any type of processor.

To support the programming of the MTC a new JTAG test language (JTL) has been defined. The test language uses the same syntax as verilog and test programs can be compiled using a standard verilog simulator. There are 13 tasks that can be used to build any kind of JTAG test sequence.

1 Introduction

The Master Test Controller (MTC) is a fully IEEE 1149.1 compliant test generator, that can be used for running JTAG test sequences both on internal logic and external devices. The test sequence setup is controlled from the on-chip microprocessor, a PowerPC 440. The MTC and the IBM JTAG Test Controller (JTC) uses a common standard 5 pin bidirectional testbus plus one extra testpin (TDI_EX). The MTC enable signal (MTC_ENB) will only be activated when running the MTC in external mode. In all other cases this bus will look like a standard JTAG bus. After power-up reset the MTC will be deactivated and can only be activated by the programming interface from the PowerPC microprocessor. The MTC has been implemented in a SOC design called DBC (Device Board Controller).

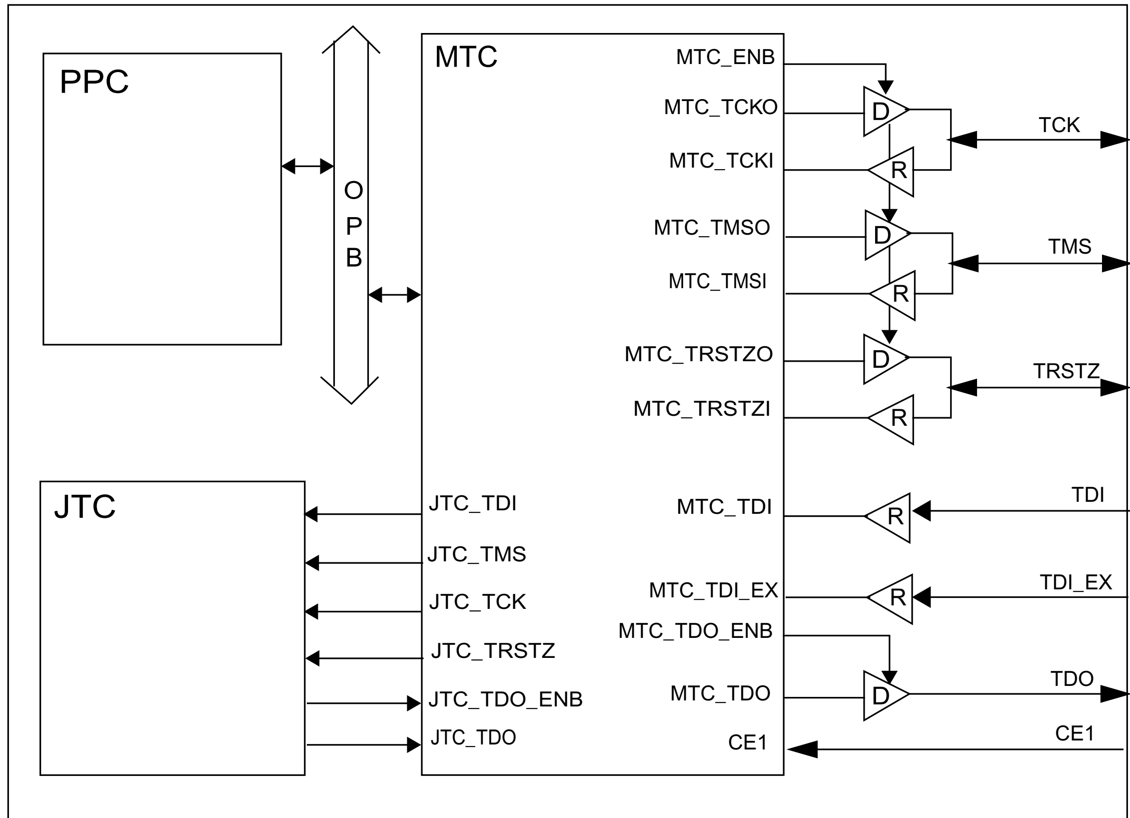


Figure 1 The Master Test Controller

1.1 Running the MTC

The MTC when activated can be used for running internal JTAG tests on the DBC or external tests on other JTAG compliant devices on the board or in the system.

The PPC device driver loads compiled test programs into the MTC_TESTPROGRAM_RAM register array (*Figure 8, page 8)*. The MTC_TESTGEN reads the test program and executes the different tasks in sequence. When the test is running, the relevant TDO data will be recorded by the MTC_TDO_RECORD block and stored in the MTC_DATA_RAM register array. When the test has finished the TDO data can be read by the PPC and compared to expected data. The status register will hold the number of TDO bits recorded.

1.1.1 PPC to MTC data transfer

The MTC exchanges information between the PPC device driver and the MTC using the On-chip Peripheral Bus (OPB) see Ref. [1].

1.1.2 Interrupt handling

The MTC will generate an interrupt, pulling O_INTERRUPT high when the test has finished. An interrupt will also be generated after an pause shiftdr has occurred. The status register should be read after an interrupt to find out the cause of the interrupt. The interrupt is enabled by setting one bit in the control register.

1.2 MTC Operating Modes

The MTC can be programmed to operate in the following modes:

Table 1 MTC operating modes

Mode	Config	MTC	JTC	External Testbus
Board test	0	Inactive	Active	Driven from external tester
MTC external test (DBC excluded)	1	Active	Test-reset-state	Driven from MTC
MTC external test (DBC included)	2	Active	Active	Driven from MTC
MTC internal test	3	Active	Active	Disabled
MTC system test (DBC excluded)	4	Active	Test-reset-state	Driven from MTC
MTC system test (DBC included)	5	Active	Active	Driven from MTC

1.2.1 External test

The MTC can run external tests on other JTAG compliant device on the same board. You can for example setup and run MBIST to test memories in other ASICs on the board.

To TDO output from the last device in the JTAG scan chain will be connected to the TDI_EX pin (*Figure 5, page 6)*.

In external test mode the DBC can be excluded or included in the JTAG board scan chain (*Figure 6, page 7)*. If the DBC is excluded the JTC will be kept in test reset state.

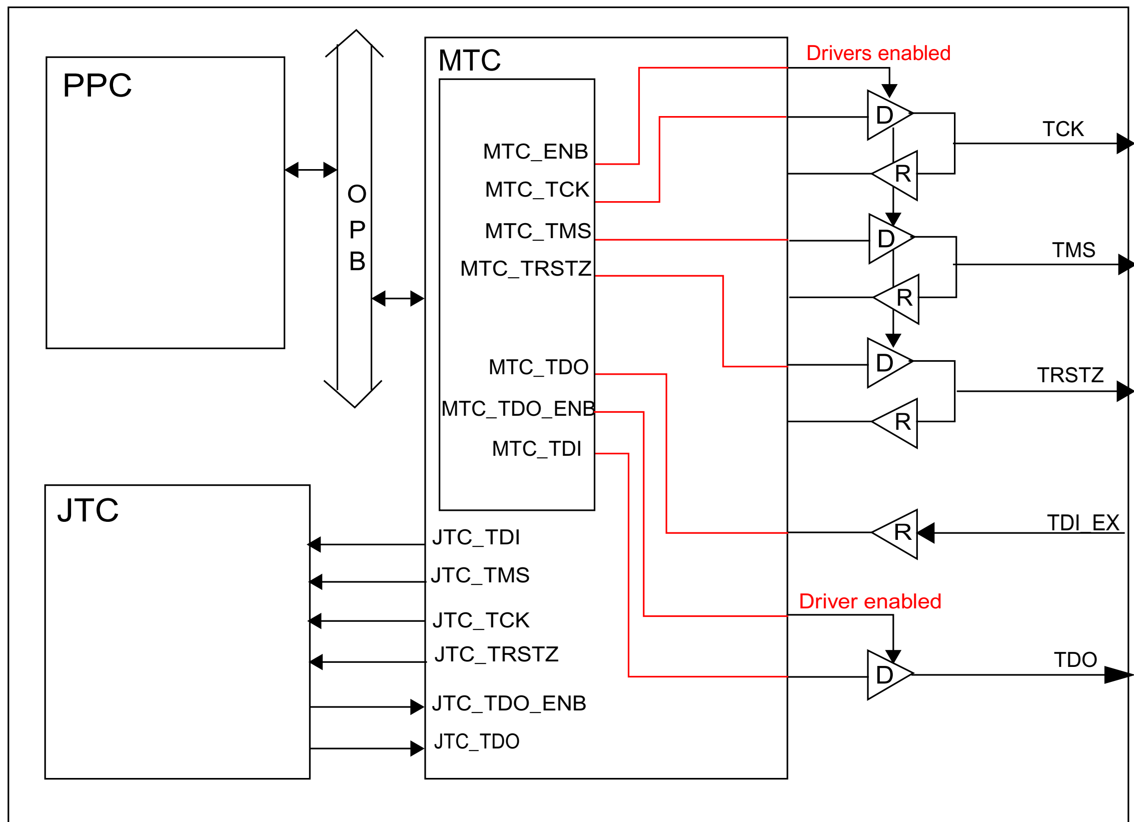


Figure 2 External Test Operating Mode

1.2.2 Internal test

The MTC can run internal JTAG tests on the DBC ASIC. It is possible to setup and run MBIST to test memories, read IDCODE, sample boundary scan register and execute other JTAG instructions. When the MTC is used in internal mode, the testbus drivers will be disabled. When the testbus drivers are disabled, by keeping MTC_ENB low, the pull-down resistor on TRSTZ will hold TRSTZ low and thereby setting all other JTAG devices on the board in Test-Logic-Reset-State.

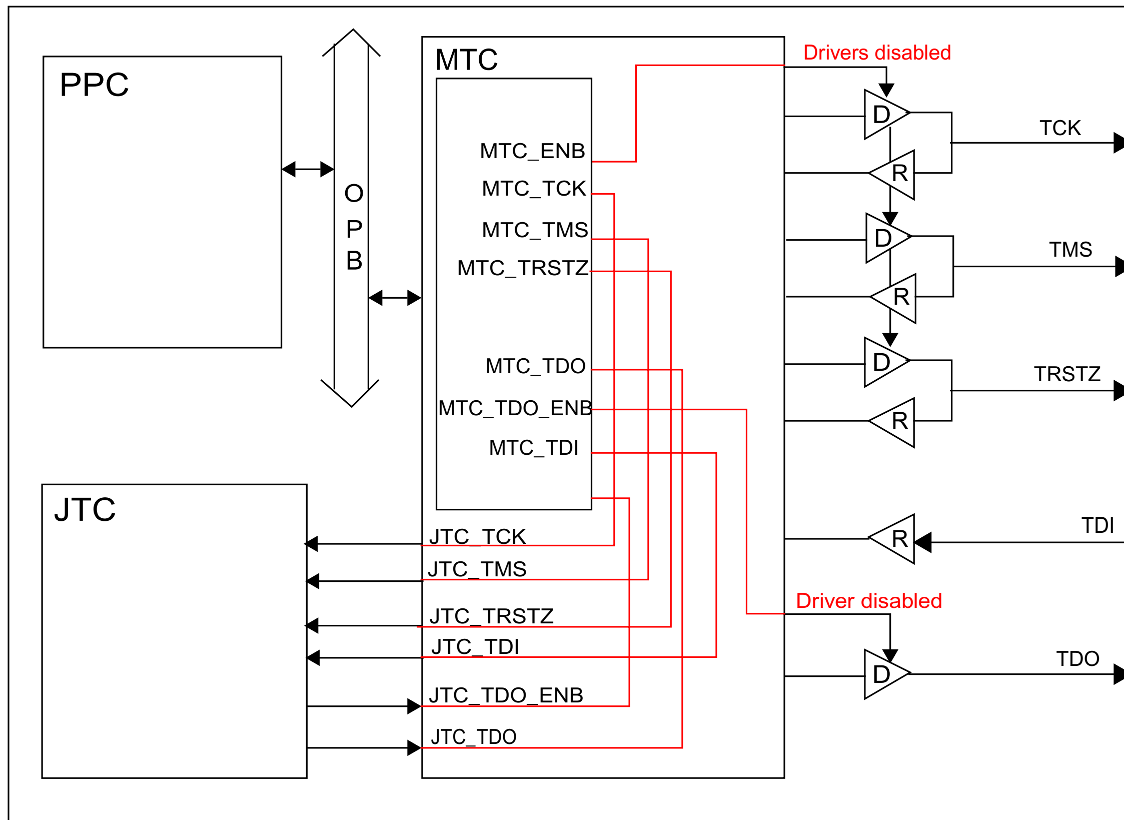


Figure 3 Internal Test Operating Mode

1.2.3 System test

In system test mode the MTC can control other boards in the system. The JTAG scan chain can include any number of devices on any number of boards (*Figure 7, page 7)*. The limiting factor is the drive capability of the MTC. In system test mode the DBC can be excluded or included in the JTAG board scan chain. If the DBC is excluded the JTC will be kept in test reset state.

1.2.4 Board test

In board test mode the MTC is deactivated and all the 5 test signals will connect to the JTC only. All JTAG compatible devices on the board will be connected into one scan chain (*Figure 4, page 6)*. An external JTAG tester will drive the JTAG test sequences.

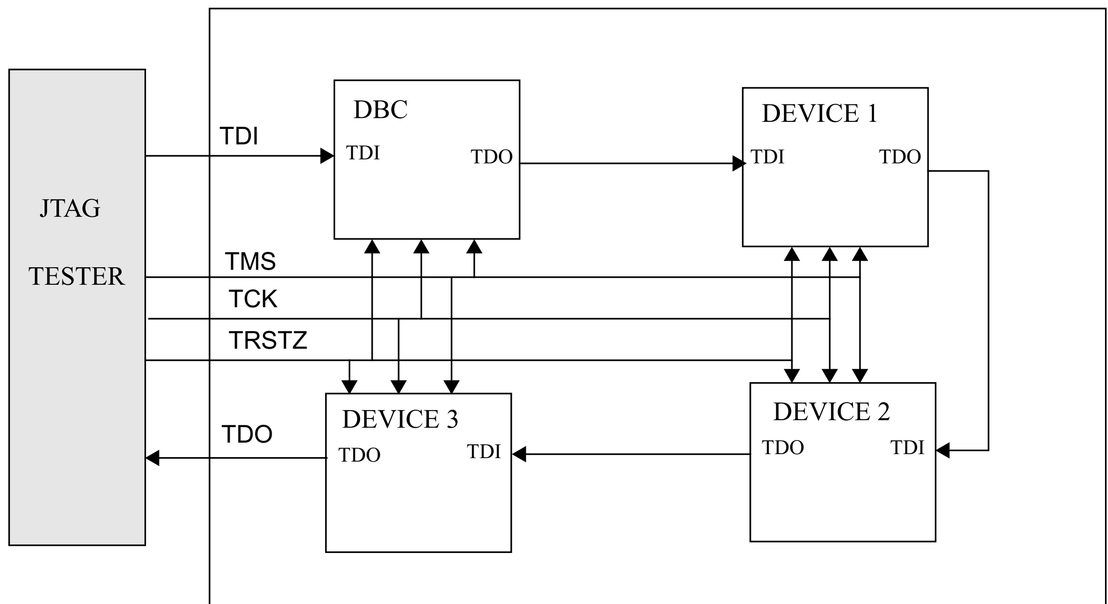


Figure 4 Board test setup

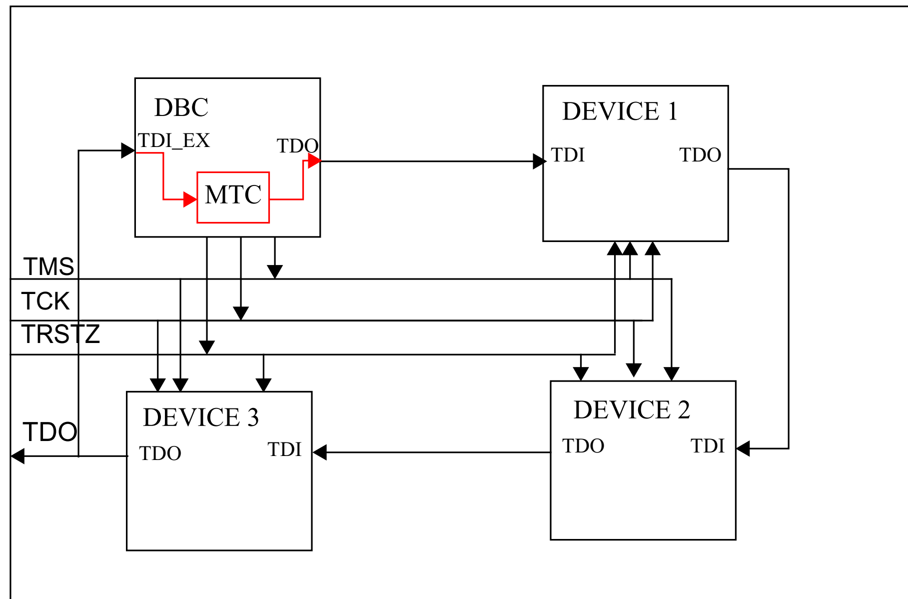


Figure 5 MTC external test setup (DBC excluded)

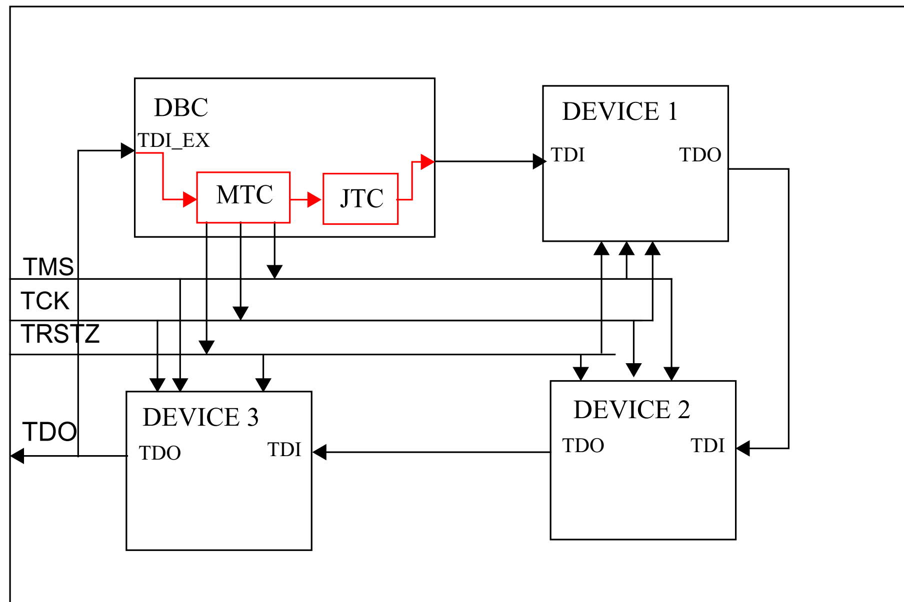


Figure 6 MTC external test setup (DBC included)

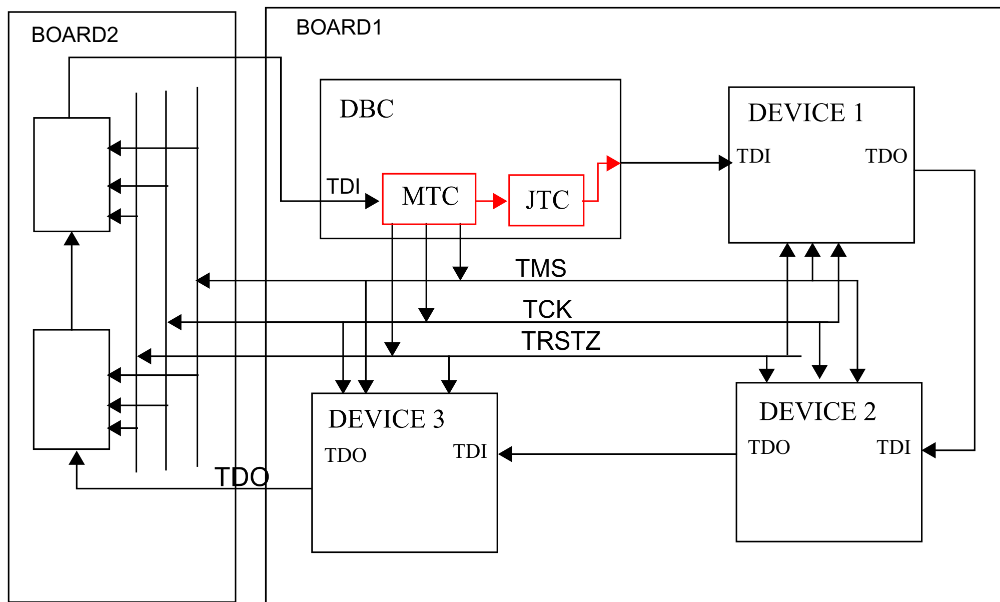


Figure 7 MTC system test setup (DBC included)

1.3 Interface Description

1.3.1 Block Diagram

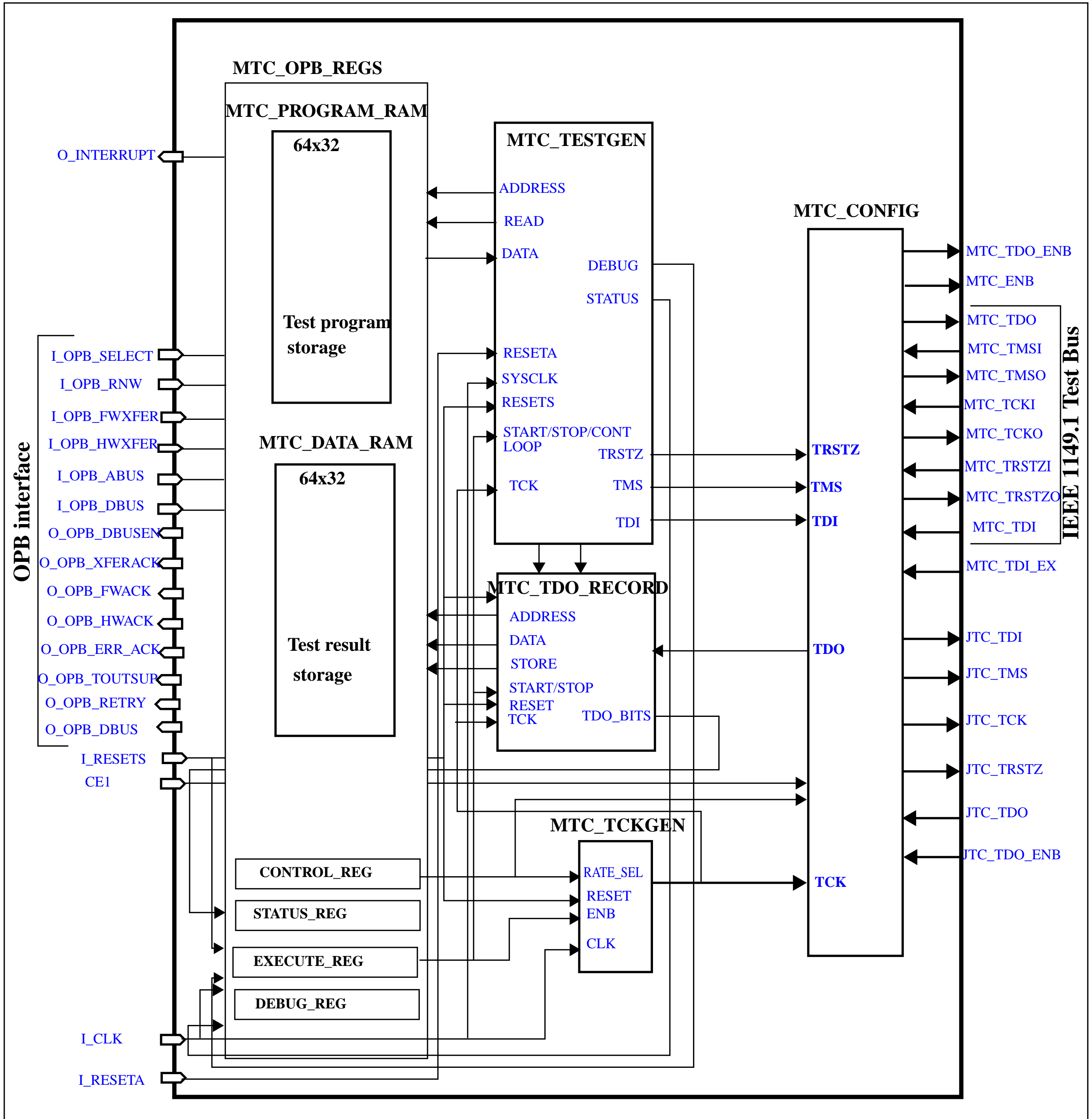


Figure 8 MTC block diagram

1.4 Block Description

1.4.1 MTC_OPB_REGS

The MTC_OPB_REGS block has 4 addressable registers and two register arrays.

Table 2. MTC address map

Register name	Register size [bits]	OPB type of access	Register Address (hex)
MTC_TESTPROGRAM_RAM	64x32	WRITE	000 - 0fc
MTC_DATA_RAM	64x32	READ	100 - 1fc
Control	32	READ/WRITE	200
Status	32	READ	204
Execute	32	READ/WRITE	208
Debug	32	READ	20c

The control register will be used for setting up the MTC operating mode, before the test starts (see table 5). The status register keeps track of the MTC status and can be read at any time. The execute register controls the start, stop, continue after pause and single stepping of the test program. The debug register can be used when debugging a failing test. It will hold information about the internal logic of the MTC that can be of help when trying to isolate the failure.

One register array MTC_TESTPROGRAM_RAM is used for storing the compiled test program and one other array MTC_DATA_RAM is used for storing the test result.

Table 3. Control register description

Bit	Bit name	Description
0	Config bit 0	Controls the MTC_CONFIG block
1	Config bit 1	(See table 1)
2	Config bit 2	
3	Config bit 3	
4	TCK clock divide bit 0	Controls MTC_TCKGEN block
5	TCK clock divide bit 1	
6	TCK clock divide bit 2	
7	Enable TCK	Enables TCK clock
8	Single step mode	1 = enabled

Table 3. Control register description

Bit	Bit name	Description
9	TDO no record	Don’t save TDO data during shiftir
10	TDO no record	Don’t save TDO data during shiftdr
11	Loop mode	1 = enabled
12	Interrupt enable	1 = enabled

Table 4. Status register description

Bit	Bit name	Description
0	Busy bit	0 = Test generation stopped. 1 = running
1	Pause bit	0 = not paused, 1 = paused
2	Test finished	1 = test have finished (completed)
6:3	Instruction code (4 bits)	Current loaded task
7	Pause in shiftdr	1 = when paused during shift data
13:8	Result RAM address (6 bits)	Last address written in TDO recording RAM
25:14	TDO bit counter (12 bits)	Counts the number of TDO bits received
31:26	Reserved	For future use

Table 5. Execute register description

Bit[7:0]	Description
0	Start / stop of MTC
1	Single step MTC (one task)
2	Continue MTC after pause
3	Reset MTC_TESTGEN

1.4.2 MTC_TESTGEN

The MTC_TESTGEN block contains the JTAG test generator. The test program will be read from the MTC_TESTPROGRAM_RAM and executed by the

MTC_TESTGEN test generator. The test program is built from a number of tasks that will operate on different JTAG registers. All tasks in the test program will be executed in sequence and the test execution will stop when the "End of task" instruction is executed or a stop signal is sent to MTC_TESTGEN. Task comes in two formats. Task format 1, 2 and 3 are used for tasks that do not use any TDI data, and task format 4 for all tasks that use TDI data. Figure 9 shows the tap controller states supported by the test generator. The shaded states are not supported and will never be entered. Dotted lines shows state transitions not supported.
The MTC_TESTGEN block performs a reclocking of the TMS, TDI and TRSTZ signals to provide save timing margins. The negative edge of the TCK clock is used as the reclocking clock.

Table 6. MTC_TESTGEN task format 1

Bit 31	Bit 30	Bits [29:4]	Bits [3:0]
1	0	Not used	Instruction code

Table 7. MTC_TESTGEN task format 2

Bit 31	Bit 30	Bits [29:16]	Bits [15:4]	Bits [3:0]
1	0	Not used	Number of TCK cycles (12 bits used)	Instruction code

Table 8. MTC_TESTGEN task format 3

Bit 31	Bit 30	Bits [29:4]	Bits [3:0]
1	0	Number of TCK cycles (26 bits used)	Instruction code

Table 9. MTC_TESTGEN task format 4

Bit 31	Bit 30	Bits [29:28]	Bits [27:16]	Bits [15:4]	Bits [3:0]
1	1	Not used	Number of shift data	Number of shift instruction	Instruction code
0	1	TDI test data			
0	1	TDI data			
0	0	Last TDI data (bit 30 = 0)			

Table 10. MTC_TESTGEN Instruction Codes

Value	Description	Task format
4’b0000	No operation	1
4’b0001	Generate test reset 1 (TSTSZ low for x TCK cycles)	2
4’b0010	Generate test reset 2 (TMS high for x TCK cycles)	2
4’b0011	Load JTAG instruction register	4
4’b0100	Load JTAG data register	4
4’b0101	Load JTAG instruction register and read/write data register	4
4’b0110	Load JTAG data register and load JTAG instruction register	4
4’b0111	Load JTAG data register and pause in pause-dr	4
4’b1000	Load JTAG data register (continue from pause-dr)	4
4’b1001	Load JTAG data register (continue from pause-dr and stop in pause-dr)	4
4’b1010	Wait in Run-Test-Idle state a number of TCK cycles	3
4’b1011	Pause in Run-Test-Idle state	1
4’b1100	End of test	1

Table 11. MTC_TESTGEN Instruction Register (task format 2)

Bit	Description
3:0	Instructions Code (see above)
15:4	Number of TCK cycles
29:16	Not used
30	0 = no more words of data follows for this task
31	1 = instruction word

Table 12. MTC_TESTGEN Instruction Register (task format 4)

Bit	Description
3:0	Instructions Code (see above)
15:4	Number of bits to load instruction

Table 12. MTC_TESTGEN Instruction Register (task format 4)

Bit	Description
27:16	Number of bits to load of TDI data
29:28	Not used
30	1 = more words of data follows for this task
31	1 = instruction word

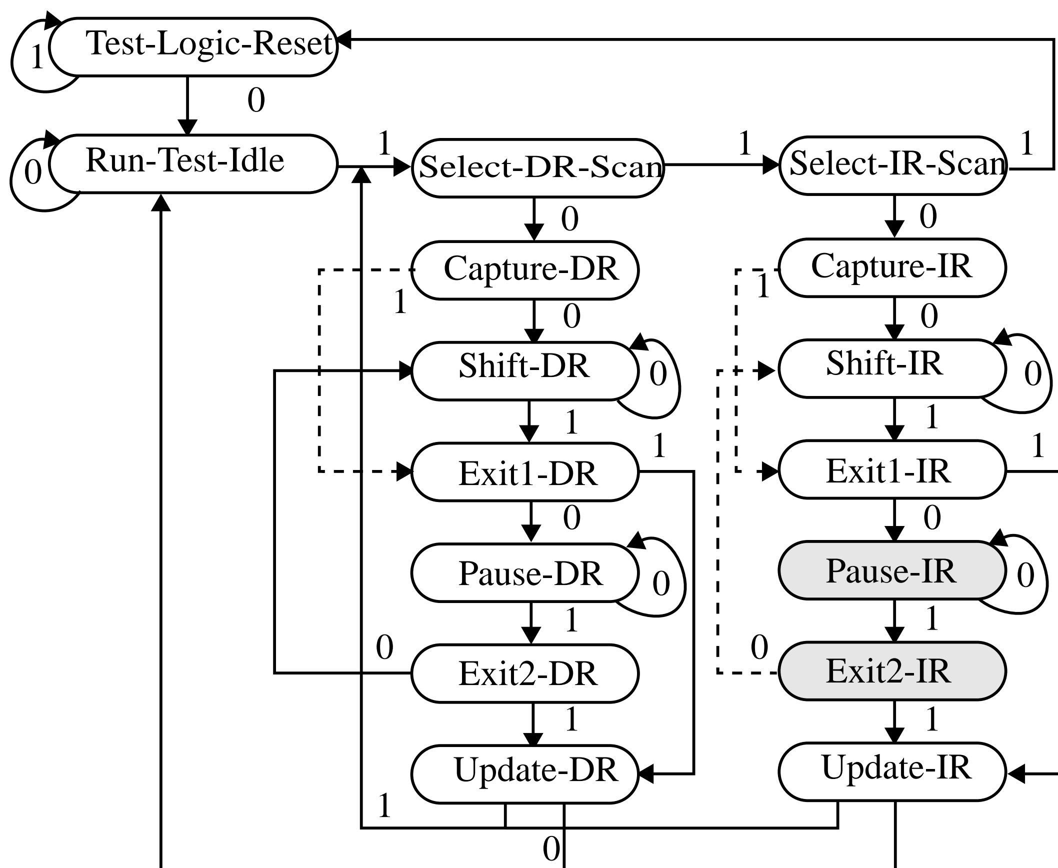


Figure 9 TAPC flow chart supported by MTC_TESTGEN

1.4.3 MTC_TDO_RECORD

The MTC_TDO_RECORD block is used as a recorder for TDO data. The TDO data will be saved when in shiftir or shiftdr state in the MTC_TESTGEN block. The saving of TDO data is enabled from the control register. The TDO data will be stored into a 32 bit register, right justified. When the 32 bit register has been filled it will be saved in the MTC_DATA_RAM register array.

1.4.4 MTC_TCKGEN

The MTC_TCKGEN block generates the TCK clock to be used as the JTAG test clock. The TCK clock frequency can be programmed to six different frequencies. With the SYS_CLK running at 48MHz the maximum TCK clock frequency will be 12MHz an the minimum clock frequency will be 750 kHz. The TCK clock can be turned on or off from the programming interface using a bit in the control register.

Table 13. TCK programmable clock frequencies

RATE_SEL	TCK clock frequency	Actual clock frequency [MHz]
0	SYS_CLK/4	12
1	SYS_CLK/8	6
2	SYS_CLK/16	3
3	SYS_CLK/32	1.5
4	SYS_CLK/64	0.75

1.4.5 MTC_CONFIG

The MTC_CONFIG block will be used to setup the different operating modes of the MTC. The config register will be written from the PPC with the selected operating mode and will control the connections of signals between the MTC and the JTC and the JTAG test port.

2 Writing test programs for MTC

The MTC will use a test program stored in the test program memory, to run the actual JTAG testing. This program consists of a number of tasks, that will perform the testing of the JTAG logic in the DBC, other ASICs on the board or in the system.

2.1 Compiling the test program

Before the test program can be loaded to the test program memory, it must be compiled into a binary format (32 bit words). This can be done in many different ways, but a simple solution is to use the verilog simulator to generate the compiled code. Every task is implemented as a verilog task and for every test program a verilog testcase using the tasks are constructed (see Appendix). Running the verilog simulator with the testcase will generate the compiled code. The compiled code will be stored in a load module file, ready to be loaded to the PPC program memory.

2.2 Load module description

After compilation of the test program, a load module will be generated. (see table below). The mask data will be calculated from the expected data. When the expected data is "0" or "1" the mask data will be set to "1". When the expected data is "x" the mask data will be set to "0".

Table 14. Load module

Address	**Content (32 bit word)**
0	Load module identification
1	Load module size (number of words)
2	TDO recording mode
3	Number of TDO bits expected
4	Number of TCK cycles to complete the test
5	Address to test program (Start_test_program)
6	Test program size (number of words)
7	Address to expected data (Start_expected_data)
8	Expected data size (number of words)
9	Address to mask data (Start_mask_data)
10	Mask data size (number of words)
Start_test_program:	Compiled test program
Start_expected_data:	Expected data
Start_mask_data:	Mask data (0 = mask)

2.3 JTAG Test Language (JTL)

The following 13 tasks should be sufficient for all kinds of JTAG tests that will be needed. All tasks except for the test reset tasks will start from the Run-Test-Idle state and return to the Run-Test-Idle state when finished.

2.3.1 TestResetKeepingTrstzLow

Generate a test reset by keeping TRSTZ low for a specified number of TCK clock cycles.

Syntax :

TestResetKeepingTrstzLow(TCK_cycles);

TCK_cycles is an integer between 1 and 4095

2.3.2 TestResetKeepingTmsHigh

Generate a test reset by keeping TMS high for a specified number of TCK clock cycles.

Syntax :

TestResetKeepingTmsHigh(TCK_cycles);

TCK_cycles is an integer between 5 and 4095

2.3.3 LoadInstructionRegister

Load JTAG instructions into all instruction registers in the scan chain.

Syntax :

LoadInstructionRegister(instruction_length,instruction_code);

instruction_length is an integer between 1 and 4095
instruction_code is a verilog register value with maximum 4095 bits

Examples:
20’b10101101110111110000;
64’h123456789abcde0;
{10’b1111111111,5’00000}; (concatenation)

2.3.4 LoadDataRegister

Load data into all data registers in the scan chain.

Syntax : LoadDataRegister(data_length,data_content);

data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.5 LoadDataRegisterPause

Load data into all data registers in the scan chain. Pause in Pause-Dr state when finished. This way a new testprogram can be loaded. The new testprogram must start with LoadDataRegisterContinue.

Syntax : LoadDataRegisterPause(data_length,data_content);

data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.6 LoadDataRegisterContinue

Load data into all data registers in the scan chain. Used after the loading has been paused. Will continue from the Pause-Dr state.

Syntax : LoadDataRegisterPause(data_length,data_content);

data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.7 LoadDataRegisterAgain

Load data into all data registers in the scan chain. Used after the loading has been paused. Will continue from the Pause-Dr state and will stop in pause-dr.

Syntax : LoadDataRegisterAgain(data_length,data_content);

data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.8 LoadInstructionAndData

Load JTAG instructions into all instruction registers in the scan chain and thereafter load all selected data registers without going to Run-Test-Idle state in between loads.

Syntax :

LoadInstructionAndData (instruction_length, instruction_code, data_length, data_content);

instruction_length is an integer between 1 and 4095
instruction_code a verilog register value with maximum 4095 bits
data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.9 LoadDataAndInstruction

Load all selected data registers and thereafter load JTAG instructions into all instruction registers in the scan chain without going to Run-Test-Idle state in between loads.

Syntax :

LoadDataAndInstruction (instruction_length, instruction_code, data_length, data_content);

instruction_length is an integer between 1 and 4095
instruction_code a verilog register value with maximum 4095 bits
data_length is an integer between 1 and 4095
data_content is a verilog register value with maximum 4095 bits

2.3.10 WaitInRunTestIdle

Generate a wait in Run-Test-Idle state for a specified number of TCK clock cycles.

Syntax :

WaitInRunTestIdle(TCK_cycles);

TCK_cycles is an integer between 1 and 67108863

2.3.11 PauseInRunTestIdle

Generate a pause in Run-Test-Idle state until the continue bit is set in the execute register.

Syntax :

PauseInRunTestIdle;

2.3.12 EndOfTest

The EndOfTest task should be the last task in the test program. When encountered it generates an interrupt by setting the O_INTERRRUPT signal high if the interrupt enable bit is set

Syntax :

EndOfTest;

2.3.13 NoOperation

The NoOperation task can be inserted anywhere in the test program. It will generate a few TCK clock cycles and the test generator will stay in Run-Test-Idle mode.

Syntax :

NoOperation;

2.4 Auxiliary tasks

The following task will only put data into the load module file. See Load Module description for more information about the data format.

2.4.1 LoadModuleIdentity

Stores information about the design, the testcase in the first word in the load module.

Syntax :

LoadModuleIdentity (design_identity,testcase_identity,testcase_revision);

design_identy is an integer between 0 and 1023
testcase_identity is an integer between 0 and 1023
testcase_revision is an integer between 0 and 4095

2.4.2 SetTdoRecordingMode

Stores information about the TDO recording mode of the MTC. The corresponding recording mode must be programmed in the control register setup.

Syntax : SetTdoRecordingMode(Mode);

Mode	Recording
0	Record TDO data during instruction and data shift
1	Record TDO data during data shift only
2	Record TDO data during instruction shift only

3 No TDO data recorded.

2.4.3 SetExpectedData

Use the SetExpectedData task to specify the expected TDO data after every instruction that will generate a TDO output.

Syntax : SetExpectedData(number_of_bits,expected_data);

number_of_bits is an integer between 1 and 4095
expected_data is a verilog register value with maximum 4095 bits.
"x" can be used to specify not known or don't care data bits.

3 Operating the MTC

When the test program is loaded the MTC can be activated. Perform the following steps to run a JTAG test using the MTC.

3.1 Setup the MTC

Load the config register to set the TCK clock rate, enable TCK, disable saving of TDO data, enable/disable single step mode and loop mode.

3.1.1 Starting the MTC

Write to the execute register with the start bit set (bit 0).

3.1.2 Stopping the MTC

Write to the execute register with the start bit cleared (bit 0). This will also clear the interrupt if set.

3.1.3 Resetting the MTC

Write to the execute register with the reset bit set (bit 3). The TCK clock must be running for reset to fully reset the MTC.

3.1.4 Looping the MTC

When the loop mode is enabled the MTC test will stay in a loop until the start bit or loop mode bit is cleared.

3.2 Test result comparison

When the test has been stopped the result memory can be read and compared against expected data. Read the status register bits [31:20] to find out how many bits of TDO data that have been stored. Use the following sequence to find out if an error has occurred:

```
loop (i = 0; i < ExpectedDataSize; i++) {
    errors[i] = (TestResult[i] xor ExpectedData[i]) and MaskData[i]; }
```

If the error array contains a "1" an error has occurred.

4 Writing a software driver for MTC

The MTC must operate in a number of testing application. The software driver must be written to support all these application (*Table 15., page 21)*. The tests will be of two kinds, automated tests and debug tests. For every test application a test set will be generated. When a test application is listed as destructive it will destroy the functional setup of the board.

4.1 Test sets

A test set is a set of test programs (load modules), that will be executed in sequence. Every test application can have its own test set or share test set with another application. A test set is a list of addresses to the test programs used (*Figure 10, page 20)*. All test sets and load modules should be stored in a flash memory on the board. This way the board will carry all its test application even when removed from the system at a repair center.

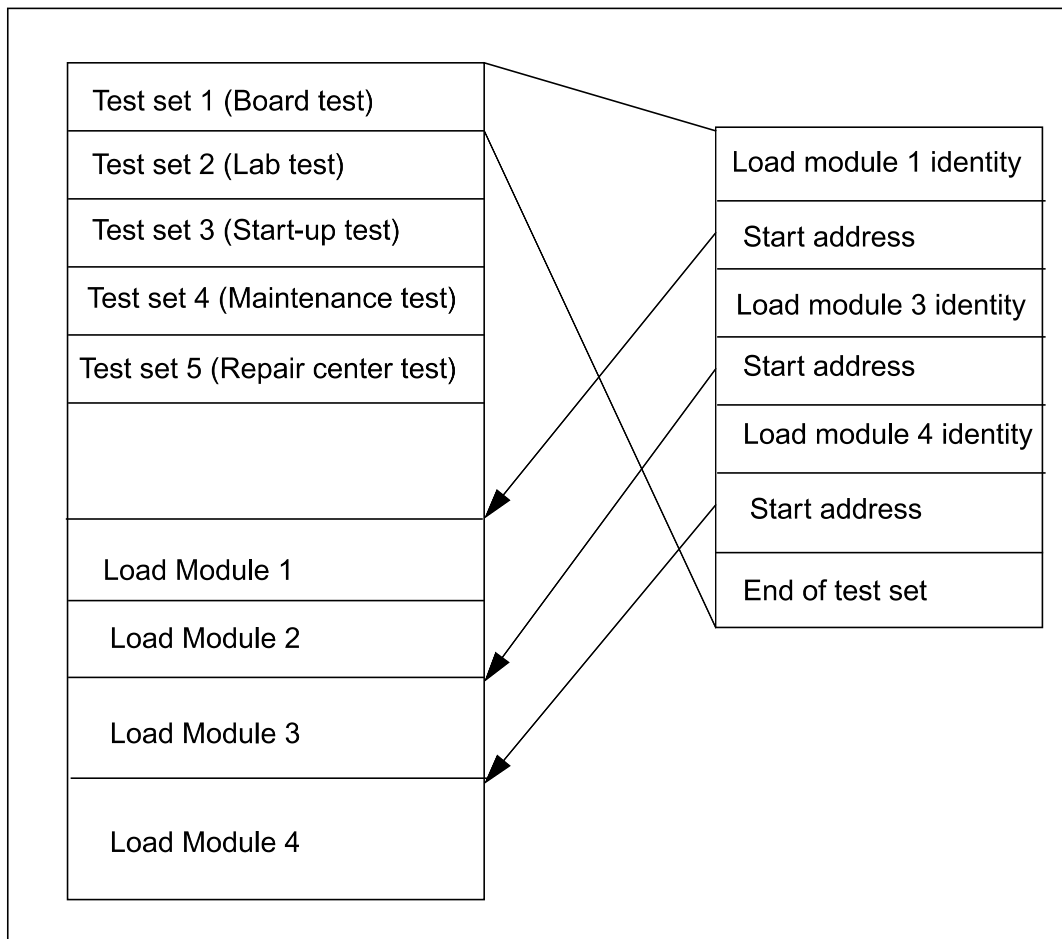


Figure 10 Flush memory data storage

4.2 Automated test

An automated test will run all tests in sequence after a start signal is issued. The test result from the first test will be compared to expected data, and if there is an error the test will stop and report the error. If there is no error the next test will start and if no errors are reported, this will go on until all tests have been run. The end of test set word (= all zeros) signals the end of the test set.

4.3 Debug test

When used in debug mode the MTC will be controlled from a terminal. Debug commands will be issued to control the MTC behaviour. All debug commands that are needed are listed in (*Table 16., page 21)*

Table 15. MTC test applications

Application	Description	Mode	Destructive
Board test	Used during production testing of the board	Automated	Yes
Lab test	Used during debugging of the board in the lab	Debug	Yes
Start-up test	Used when powering up the system	Automated	Yes
Maintenance test	Used when the system is up and running	Automated	No
Fault isolation	Used during fault isolation in the system	Debug	Yes
Repair center test	Used in the board repair center	Automated, debug	Yes

Table 16. Debug command

Short command	Description	Arguments	Register affected
TSET	Select test set	1..N	None
RUN	Run complete test set	1..N	None
LOAD	Load test program (Load module)	1..N	None
RESET	Reset MTC	None	All
TCK	Enable or disable TCK clock	ENB DIS	Control_reg[7]
MODE	Select MTC operating mode	BOARD INT EXTI EXTE SYSI SYSE	Control_reg[3:0]
SPEED	Set TCK clock speed	4 8 16 32 64	Control_reg[6:4]
SSTEP	Enable or disable single step mode	ENB DIS	Control_reg[8]
TDOREC	Set TDO recording mode	ALL INSTR DATA NONE	Control_reg[10:9]
LOOP	Enable or disable loop mode	ENB DIS	Control_reg[11]
INT	Enable or disable interrupt	ENB DIS	Control_reg[12]
STATUS	Read status register	None	Status_reg

Table 16. Debug command

Short command	**Description**	**Arguments**	**Register affected**
DEBUG	Read debug register	None	Debug_reg
CONTROL	Read control register	None	Control_reg
START	Start MTC	None	Execute_reg[0]
STOP	Stop MTC	None	Execute_reg[0]
STEP	Single step MTC test program	None	Execute_reg[1]
CONT	Continue MTC execution after pause	None	Execute_reg[2]
RESULT	Read TDO recording RAM	None	
COMPARE	Compare TDO recorded to expected data		

5 Conclusions

The design of the MTC shows that a JTAG tester can easily be fitted into a system-on-chip design. Having a JTAG tester on the board will open up many new testing opportunities during the whole product life cycle. The definition of a simple JTAG test language (JTL) that can be used for writing all kind of JTAG tests will make the programming of MTC an easy task. The debugging features can be used in the lab to help the board designer in the prototype testing. New test features can be added and controlled from the MTC. The possibilities are infinite.

Acknowledgement

I would like to thank Gunnar Carlsson and Tom Stadler for their support and technical assistance during the MTC design phase.

References

[1] IBM On-Chip Peripheral Bus Architecture Specifications v 2.1

[2] IEEE Std 1149.1-1990 with supplements IEEE 1149.1a-1993 and IEEE 1149.1b-1994

[3] PPC440x6 Embedded Processor Core User's Manual IBM SA14-2758-01

Appendix A

Test program example 1

Read identification code from the IDCODE register. Load the IDCODE instruction and shift out the 32 bit identification code.

```
parameter INSTRUCTION_LENGTH        = 4;
parameter DATA_LENGTH               = 32;
parameter IDCODE                    = 4'b0010;
parameter ALL_TDO_DATA              = 2'b0;
parameter DEVICE_IDCODE             = 32'h14012049;

SetTdoRecordingMode(ALL_TDO_DATA);
TestResetKeepingTrstzLow   (10);
LoadInstruction (INSTRUCTION_LENGTH,IDCODE);
SetExpectedData(INSTRUCTION_LENGTH,4'b0001);
ReadWriteDataRegister (DATA_LENGTH,32'h0);
SetExpectedData(DATA_LENGTH,DEVICE_IDCODE);
EndOfTestProgram;
```

Test program example 2

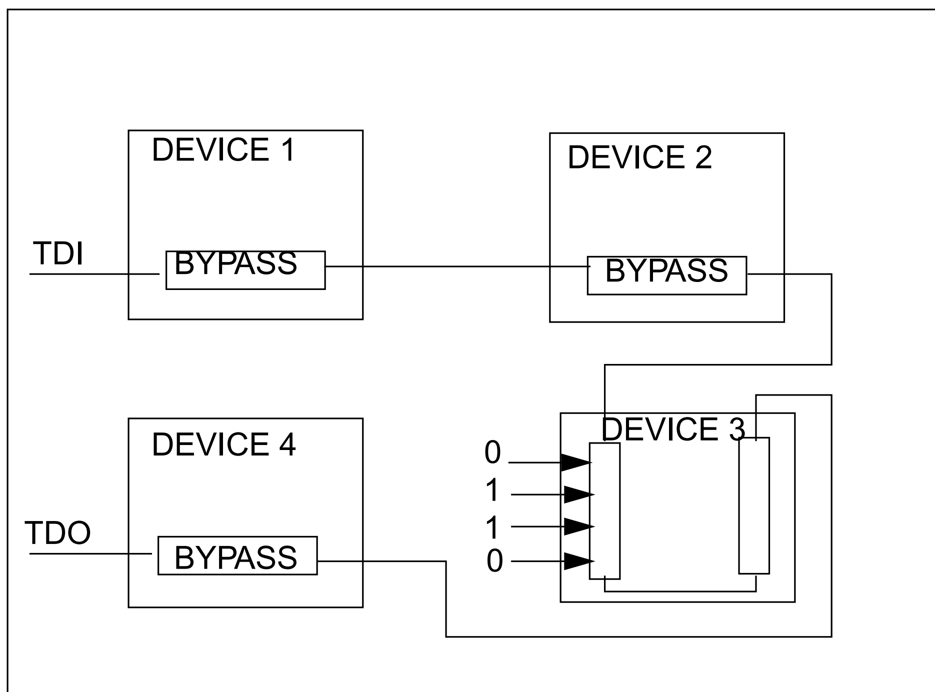
Setup and run a MBIST test. Load the MBIST_ENABLE instruction and enable all 8 MBISTs. Load the RUNBIST instruction to start the MBISTs. Wait in Run-Test-Idle state until the test has finished. Load the instruction MBIST_RESULT. Read the result by shifting out the date from the MBIST_RESULT data register. Expect all bits set to one.

```
parameter INSTRUCTION_LENGTH              = 4;
parameter DATA_LENGTH                     = 8;
parameter TDO_SHIFTDR                     = 2'b01;
parameter MBIST_ENABLE                    = 4'b0100;
parameter MBIST_RESULT                    = 4'b0101;
parameter RUNBIST                         = 4'b0110;
parameter MBIST_RUNTIME                   = 1000000;

SetTdoRecordingMode(TDO_SHIFTDR);
TestResetKeepingTrstzLow (10);
LoadInstruction (INSTRUCTION_LENGTH,MBIST_ENABLE);
ReadWriteDataRegister (DATA_LENGTH,8'b11111111);
SetExpectedData(DATA_LENGTH,8'bxxxxxxxx);
LoadInstruction (INSTRUCTION_LENGTH,RUNBIST);
WaitInRunTestIdle (MBIST_RUNTIME);
LoadInstruction (INSTRUCTION_LENGTH,MBIST_RESULT);
ReadWriteDataRegister(DATA_LENGTH,8'b00000000);
SetExpectedData((DATA_LENGTH,8'b11111111);
EndOfTestProgram;
```

Test program example 3

Setup a test to sample the inputs on device 3 in the board scan chain.




```
parameter INSTRUCTION_LENGTH            = 4;
parameter BOUNDARY_SCAN_LENGTH          = 8;
parameter TDO_SHIFTDR                   = 2'b01;
parameter SAMPLE                        = 4'b0011;
parameter BYPASS                        = 4'b1111;

SetTdoRecordingMode(TDO_SHIFTDR);
TestResetKeepingTrstzLow (10);
LoadInstruction (4 * INSTRUCTION_LENGTH,
{BYPASS,BYPASS,SAMPLE,BYPASS});
ReadWriteDataRegister (1+1+BOUNDARY_SCAN_LENGTH+1,11'b0);
SetExpectedData (1+1+BOUNDARY_SCAN_LENGTH+1,11'bxx0110xxxxx);
EndOfTestProgram;
```