Testing a System-On-a-Chip with Embedded Microprocessor

Rochit Rajsuman
Advantest America R&D Center
3201 Scott Blvd
Santa Clara, CA 95054
(408) 727-2222, ext. 386
r.rajsuman@advantest.com

Abstract

In this paper, we describe the test methodology for embedded cores based system-on-a-chip (SoC) which contains a microprocessor core. First the microprocessor core is tested for correctness of all the instructions and then the computation power of the microprocessor core is used to test the on-chip memories and other cores. A small Iddq test set is also used to detect physical defects, the design features to facilitate Iddq testing are described.

1. Introduction

In the recent years, ASIC technology has evolved from a chip-set philosophy to an embedded cores based system-on-a-chip (SoC) concept. A SoC includes various reusable functional blocks called cores such as microprocessors, memories, DSP, bus control and interfaces like PCI and USB. While the use of cores in SoC serve a broad range of applications, the complexity of these chips is far too complex to be tested by traditional methods [1]. Reference [2] provides a good overview of testing difficulties. A substantial amount of research is underway to address these testing difficulties [1-5]. In the production of SoC, a combination of test methodologies are used such as functional test, full-scan, BIST, Iddq etc. In a broader sense, the individual cores are tested by one or more of the following methods:

1. Testing a core through system-chip's functional test;
2. Direct test application while accessing the core through I/O pin multiplexing;
3. Test application to the core through local boundary scan or a collar register;
4. Scan and Built-in self-test through a variety of access and control mechanisms;
5. Proprietary solutions.

In this paper we describe a test methodology for SoC that contains a microprocessor core. The test methodology can be divided into four subsets: (i) testing of the microprocessor core by ensuring the correctness of all of its instructions; (ii) using microprocessor core to test the embedded memories; (iii) using microprocessor core to test other cores; (iv) implementation and design for Iddq testing. Subsequently, from section 2-to-5, this paper is organized into four sections, each describing one subset of the test methodology and finally, section 6 contains concluding remarks.

2. Testing of Microprocessor Core

A large number of SoC contain one or more microprocessor/micro-controllers. The testing of embedded microprocessors is quite complex. In general, design-for-test and built-in self-test schemes such as full-scan, partial scan, logic BIST, scan-based BIST are used to test various logic blocks within a microprocessor/micro-controller.

In this section, we describe a BIST type method for the testing of microprocessor core used in SoC. This method tests the correctness of instructions and thus, it can be considered as a functional test method.

2.1 Test Structure for μP Core

The broad structure of a microprocessor/micro-controller is shown in Figure 1a. As shown in figure 1a, instruction fetch unit obtains the opcode of the next instruction based upon the address in Program

0-7803-5753-1/99 $10.00 ©1999 IEEE

Counter. This opcode is decoded by the instruction decode logic, which generates function select and control signals for the execution unit. Based upon these control signals, one of the logic blocks within the execution unit computes its function. The operand or data for this computation is obtained from the system memory.

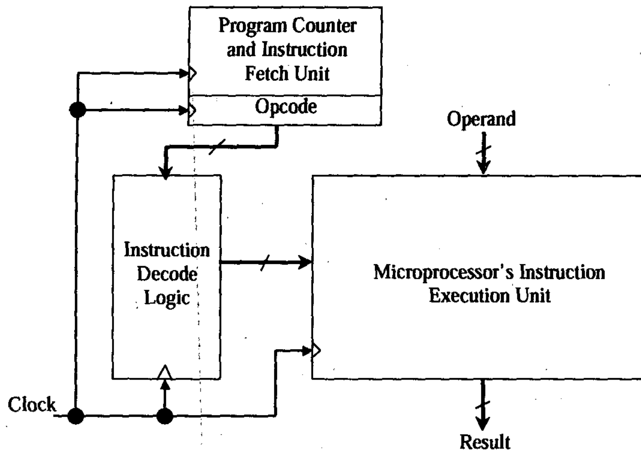

Fig. 1(a): General structure of a microprocessor.

The structure shown in figure 1a is modified by adding three registers as shown in figure 1b. One Test Control Register (TCR) to provide the opcode of the microprocessor's instructions during the test mode, one Linear Feedback Shift Register (LFSR) and another Multi-Input Feedback Shift Register (MISR) to generate random data and to compress test response respectively. During the test mode, the data from LFSR is used as an operand for the instruction provided by the TCR. The computed result is stored in the MISR. It should be noted that, as shown in Figure 1b, the execution unit contains a group of complex blocks implementing integer and floating point arithmetic and logic operations.

2.2 Test Sequence for μP Core

The testing sequence is as follows:

1. Activate the test mode. In this mode, the content of TCR is used as an instruction rather than the opcode from the instruction fetch unit.
2. Initialize TCR, LFSR and MISR via test control signals.
3. Serially load TCR with the opcode of an

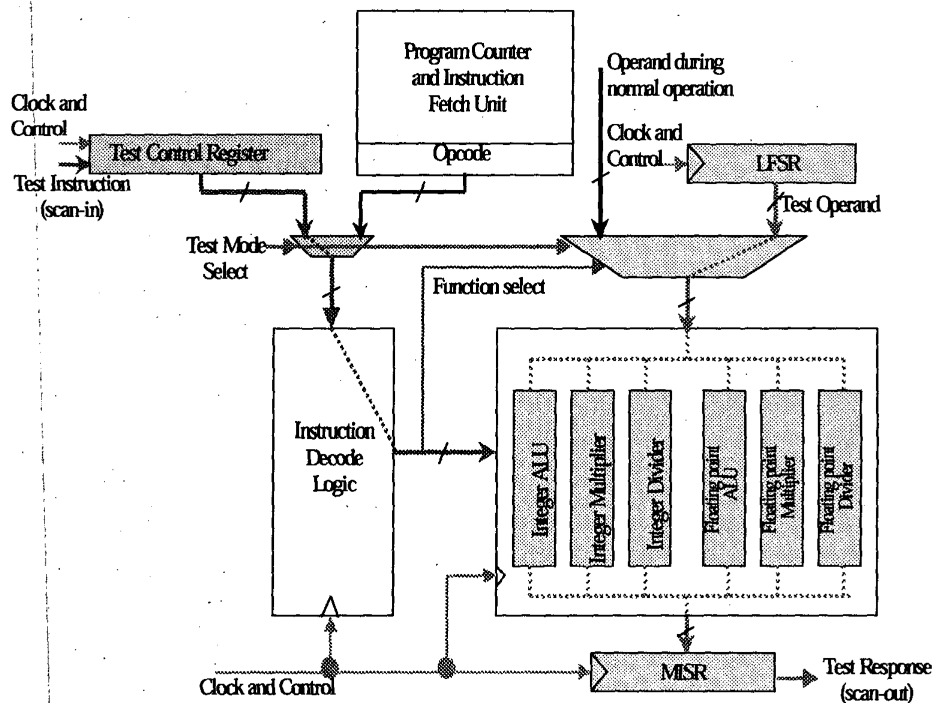

Fig. 1(b): Implementation of BIST-type method for functional testing of microprocessor core.

instruction.

4. Clock LFSR and MISR for a fixed number of cycles. This step repeatedly executes the instruction in TCR with LFSR data. For example, when 1000 clocks are used, the instruction in TCR is executed (1000 times - latency of the block) with different operands, random data provided by the LFSR.
5. Serially take out the content of MISR to determine pass/fail.
6. Compare the content of MISR with a pre-computed simulation signature to determine if there is a fault. The Automatic Test Equipment (ATE) performs this comparison.
7. Repeat steps #2 to #6 with different instructions until all instructions are exercised.

The control of this scheme is implemented through IEEE 1149.1 boundary scan TAP controller [6]. The test control signals and test responses are passed through, as well as controlled by the boundary scan TAP controller. Figure 2 illustrates the overall implementation.

It should be noted that in this scheme, the simulation testbench is used as golden data that contains all instructions with LFSR sequence and MISR signatures after each run. Thus, the fault free MISR content after each instruction run is known apriori for simple comparison during manufacturing test.

The above procedure determines that each instruction is executed correctly and hence, it provides functional coverage. To also estimate the stuck-at fault coverage of an individual block for approximately 1000 random operands, fault simulation is performed on the recorded run. For fault simulation, a commercial EDA tool is used.

3. Testing of Embedded Memories

Once the testing of the microprocessor core is completed, this core is used to test the embedded memory and other on-chip cores. In this section, we describe our method to test the embedded memory using the computation power of microprocessor core. In this method, an assembly language program is executed on the microprocessor core, which generates memory test patterns. Although, we used the

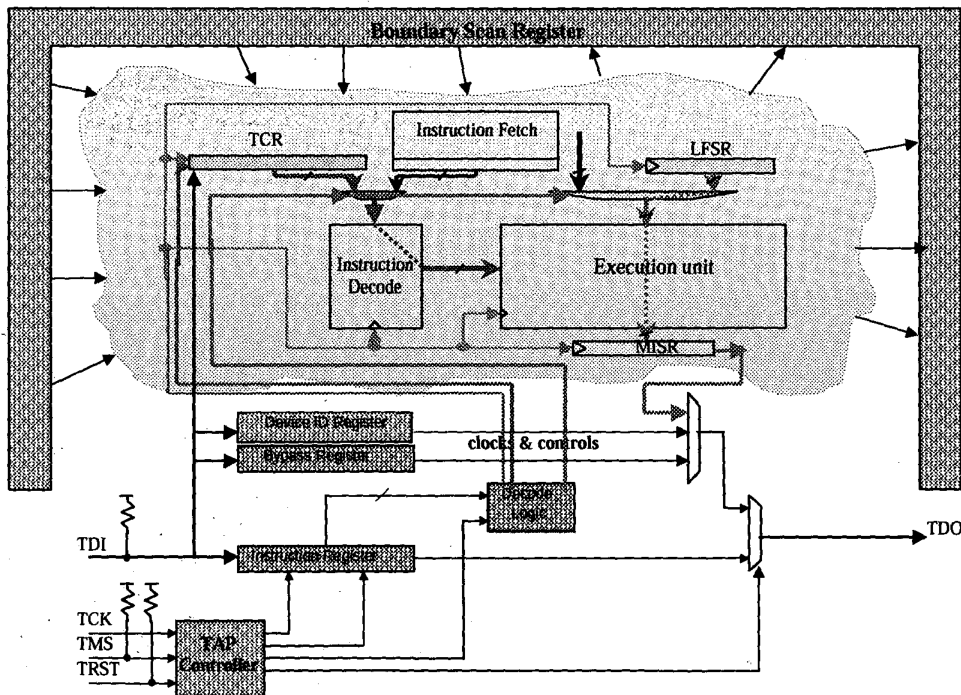

Fig. 2: Implementation of BIST-type method for microprocessor core.

March algorithm, any memory test algorithm can be used for this purpose. The test response is also evaluated by the microprocessor core.

3.1 Test Procedure

The example assembly language procedure using a simplified memory March-algorithm is shown in Figure 3. This example uses word-wide read/write operations with 0101...01 data in increasing order and 1010...10 in decreasing order. This test is for a 2-Mbits RAM, organized as16Kx16.

```
/* Procedure to test embedded RAM */
/*A0 is address counter, D0 contains test data
(write), D1 is used for read data (response);
A0, D0-D2 are general purpose registers */
/* Initialization */
MOVE #0000H, A0
        /*Initializing address counter */
MOVE #0000H, D0
        /*This is initialization data */
MOVE #0000H, D2
        /*This will be used to clear memory
        word after read */

/* Test Procedure */
Initial  MOVE D0, [A0]
        /*The value in D0 is written into
        location addressed by A0 */
 INCR A0
COMP A0, FFFFH
        /* FFFFH is the last address */
 BEQ Test_Incr
BRA Initial
        /*Memory is initialized, Start test */

/* Write/Read in increasing order */
Test_Incr  MOVE #0, A0
MOVE #5555H, D0
        /* This is test data (0101...) */
Cont_Incr  MOVE D0, [A0]
        /* This is write opreation */
MOVE [A0], D1
        /* This is read operation */
MOVE D2, [A0]
        /* This clears memory word */
 COMP D0, D1
 BEQ Next_Incr
BRA Fail
        /* Read data is not 5555H */
Next_Incr  INCR A0
COMP A0, FFFFH
        /* Last address */
 BEQ Test_Decr
 BRA Cont_Incr

/* Write/Read in decreasing order */
Test_Decr  MOVE #AAAAH, D0
Cont_Decr  MOVE D0, [A0]
        /* This is write operation */
MOVE [A0], D1
        /* This is read operation */
MOVE D2, [A0]
        /* This clears memory word */
 COMP D0, D1
 BEQ Next_Decr
BRA Fail
        /* Read data is not AAAAH */
Next_Decr  DECR A0
COMP A0, 0000H
        /* 0000H is the last address */
 BEQ Done
 BRA Cont_Decr

Done    Write Test_Passed
Fail    Write Test_Failed
```

Figure 3: An assembly language program for word-wide March pattern running on the microprocessor core.

The assembly language program shown in figure 3 is converted into binary form by the assembler of the microprocessor core. The assembler binary code is used in a manner similar to the test vectors from ATE for the microprocessor core. The data supplied to the microprocessor core come directly from the tester. In some sense, during this testing, the tester becomes the system memory for the microprocessor core. As the data from tester are the microprocessor's instructions in binary form, the microprocessor core executes operations as intended by the program instruction. The final output from the microprocessor to the tester is a pass/fail status. It should also be noted that the test program shown in figure 3 stops as soon as an error occurs (writes Fail). The tester stores this failure and, hence, the fail-bit location is known without any additional effort.

The method is straightforward and it greatly simplifies ATE requirements. It provides at-speed testing with no performance penalty, and any memory test algorithm can be used. The only drawback to this method is that it requires a special API (application program interface) to handle the binary information generated by the assembler. This API allows the tester to understand the assembler binary code during test program development as well as during actual testing.

It is interesting to note that similar methods have been suggested in the past that a carefully crafted test program can test some memory arrays on-chip if the instruction cache is first fully tested [7-9]. The method described above is fundamentally different from [7-9] with respect to the tester interface as well as it does not require a known good I-cache [10].

4. Testing of Function Specific Cores

Once the microprocessor core and on-chip memories have been tested, the microprocessor core is used to test other cores. In this section we describe the testing of embedded D/A converter (DAC).

4.1 Test Procedure

The basic concept is similar to section 3. An assembly language program is developed, converted into the binary form by the assembler and run on the microprocessor core through tester API.

However, to solve the difficulty of test data application from microprocessor core to DAC, we used one extra register (analog test register, ATR). In the test mode, the contents of ATR can be altered by index addressing, such as by addressing through any one of the microprocessor address register. In the test mode, ATR provides the inputs to DAC via a multiplexer. During normal mode ATR is cut-off via same multiplexer. The concept is illustrated in figure 4.
With the help of the ATR, test stimuli generated by the microprocessor core could be applied to the DAC. The procedures to generate test stimuli for offset voltage (Vos), full-scale range (FSR), missing codes and major transitions, differential non-linearity (DNL) and integral non-linearity (INL) is described as follows:

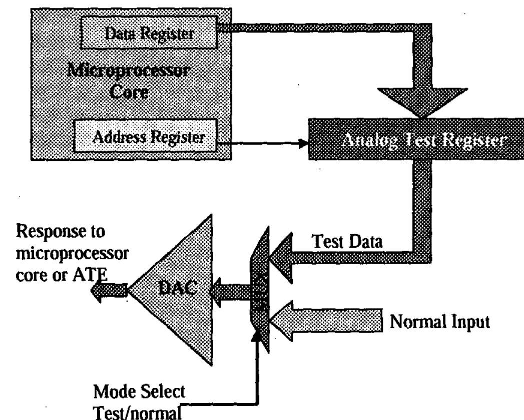

Fig 4: Mechanism for test data application from microprocessor core to DAC.

Offset Voltage (Vos): For DAC, offset voltage is the analog output voltage when a null or all-zero code is applied at the inputs. The width of the test vector is same as the width of DAC resolution bit, while the length is 2^N, such as 16, 32, 64 etc. Further to avoid distortion due to noise, we apply the same vector multiple times and an average of output is taken to compute Vos. The test vector is obtained simply by loading all-0 to one of the microprocessor data register. A single microprocessor instruction is used for this purpose, MVI 0000H, (Ai); where (Ai) is the ith address register.

Full Scale Range (FSR): Full-scale range is the difference between analog output voltage when the full-scale code (all-1) value is applied at the inputs (V_{FS}) to analog output voltage when a null (all-0) code is applied at the inputs (Vos), such as FSR=V_{FS}-Vos. The test stimulus for VFS is all-1 value, hence, the procedure is same as generating test vector for Vos. Two instructions provide the necessary test stimuli for FSR (all-0 and all-1 values).

Missing Codes and Major Transitions: For DAC, a major transition is the transition between codes that causes a carry to flip the least significant non-zero bit and sets the next bit. For N-bits DAC, a counter that counts from 0 to 2^N-1 provides all possible code values, thus, it is sufficient to test for any missing codes and major transitions. A simple assembly language program that provides a counter functionality is sufficient to generate all code values.

Differential Non-linearity: It is the maximum deviation of an actual analog output step between adjacent input codes from the ideal value of 1 LSB (least significant bit). It requires all-0, all-1 and a linear sequence of all code inputs. Thus, the combined procedures given in #2 and #3 provide the necessary test stimuli.

Integral Non-linearity: It is the maximum deviation of the code edges or analog output from a straight line drawn between the first and the last code. Thus, the combined procedure given in #2 and #3 provide the necessary test stimulus.

In summary, the procedure shown in figure 5 generates necessary test stimuli for DAC:

```
MVI 0000H, (A1)
        /* Move all-0 to location addressed
        by A1, it provides all-0 vector */
MVI FFFFH, (A1)
        /* Move all-1 to location addressed
        by A1, it provides all-1 vector */
/* Procedure to generate all code values */
Start   MVI 0000H, D1
        /* It sets register D1 to zero */
MVI 0100H, D2
        /* It sets register D2 to 2^8 for 8-bit
        DAC. This number varies for
        different DAC, for example, for 10-
        bit DAC it needs to be 0400H or 2^10
        */
Cont    CMP D1, D2
        /* Compare the content of D1 to D2
*/
JZ      Stop
MOVE D1, (A1)
        /* Move contents of D1 to the
        location addressed by A1 */
INCR D1
        /* This generates the next code value
*/
JMP     Cont
Stop    HLT
```

Fig. 5: An assembly language program for generating test stimuli for DAC.

For embedded DAC, the evaluation of test response is also a difficult problem. However, the DAC output being the primary pin, mixed-signal unit of the tester is much more efficient and simpler solution rather than designing a method to feed DAC response to microprocessor core and perform various calculations to determine the DAC parameters.

5. Iddq Testing

In Reference [11-12], a design methodology is presented that provides a global control signal to switch-off static current dissipating logic. This design uses a special buffer and a dedicated pin to control this global signal during Iddq testing as well as in the normal operation. The basic design is shown in Figure 6.

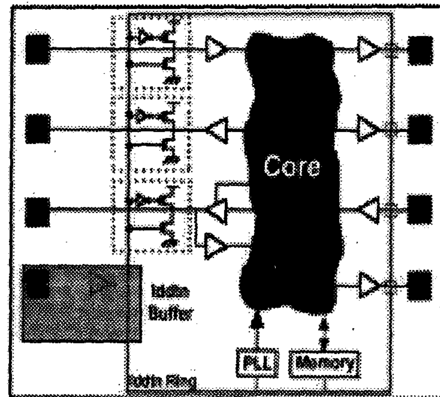

Fig. 6: ASIC design that allows to switch-off static current dissipating logic during Iddq testing [11].

During normal mode, due to the pull-down transistor of special buffer (Iddtn buffer), the control signal (Iddtn-ring) is always "1", keeping the pull-up/pull-down transistors ON

at the signal pins. For Iddq testing a "1" is applied at this dedicated pin, which brings Iddtn-ring to "0" and switches-off all pull-up/pull-down transistors on the signal pins. The connections to the PLL, memories and other switchable internal logic is via an inverter (not shown in figure 6) connected to the gate of a large pMOS transistor in the power supply path.

5.1 Power-Supply Partitioning

To overcome the requirement of a dedicated pin and buffer, we used JTAG Boundary Scan based control. For this purpose a flip-flop is used to control the logic of the global power_down control signal. During normal mode, the flip-flop is always set to "1"; while in the Iddq test mode, it is set to "0". A new private instruction "Power_Down" is implemented along with other mandatory and private JTAG Boundary Scan instructions (extest, intest etc.).

To perform Iddq testing, the Power_Down instruction is loaded into the boundary scan instruction register. This instruction is decoded and provides logic "0" to the flip-flop controlling the power_down control signal.

With power_down control signal being 0, the TAP controller is kept in the RunTest/Idle state for the duration of Iddq testing (for TAP controller state diagram, please refer to [6]). Hence, all the circuits connected to the power_down control signal switch-off and remain off. After Iddq testing is completed, the TAP controller is brought back to the Reset state. This resets the flip-flop to "1" and brings back power_down control signal to "1". Subsequently, the circuit comes back to the normal operating mode. The detailed design is shown in Figure 7.

We further extended this method at the SoC level by implementing multiple Power_Down instructions to selectively switch-off portions of the chip. Specifically, multiple cores and other static current dissipating logic are switched-off and Iddq testing is performed on one core at a time. In a SoC with three cores and one large embedded memory, we required 4 Power_Down instructions:

1. *Power_Down_A*
2. *Power_Down_B*

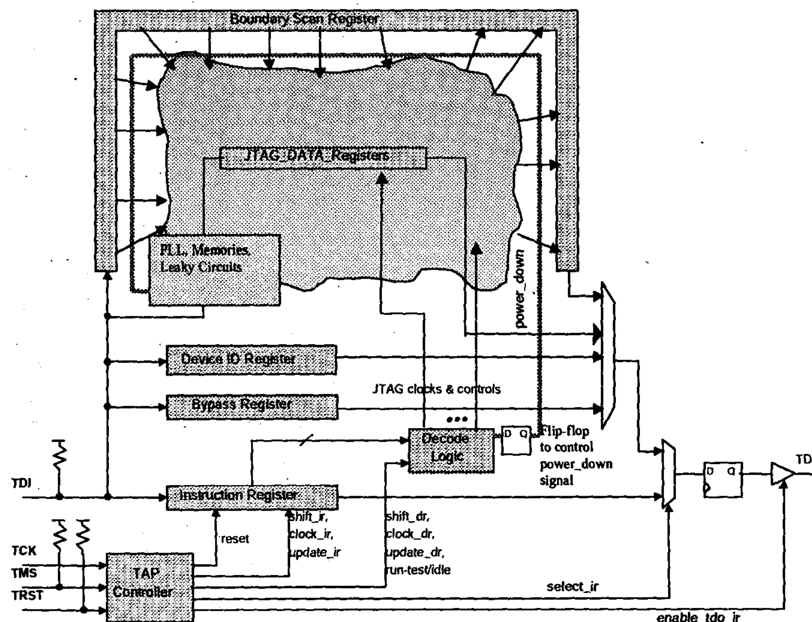

Fig. 7: Implementation of global power_down control signal through TAP

3. *Power_Down_C*
4. *Power_Down_Main.*

The nomenclature is that the instruction allows Iddq testing on the block/core identified in the instruction. It should not be confused with the word power_down, the instruction name indicates power down of the whole SoC except the portion identified by the last character. For example, the Power_Down_A instruction cuts-off the power-supply to core-B, core-C, glue logic, PLL, memory and static current dissipating logic in core-A. Thus, it allows Iddq testing on core-A.

A 4-bit register is used to provide necessary logic values for power_down control signals to selectively perform Iddq testing on any one core or glue logic. When any of the Power_Down_A, Power_Down_B or Power_Down_C instructions is loaded in the Boundary Scan instruction register, after decoding, it keeps one power_down signal to a "1" while all other power_down signals are set to "0". The power-down control signals that are "0", cut-off the power supply of the respective blocks.

The decoding of the Power_Down_Main instruction is slightly different. It sets all power_down signals to 0, and therefore switches-off all individual cores, PLL, memory and other static current dissipating logic in glue logic. Thus, it allows Iddq testing on glue logic.

Besides the above design feature, the choice of package is also considerably useful in Iddq testing. A segmented Vdd plane allows a natural partitioning of the power-supply at chip-level. Such package with 4-segment Vdd plane is shown in figure 8. Although, the each segment has same voltage, but logically we can consider that there are 4 power-supplies: Vdd1=Vdd2=Vdd3=Vdd4.

During testing, each segment is connected to a different tester power-supply or independent source. Iddq testing is performed on one segment at a time. Although, all 4-segments are powered-on for chip operation, but the leakage in circuitry connected to each segment is isolated from each other and limited to independent source. Thus, the leakage as observed at any one segment is significantly less than the cumulative leakage of the whole chip if the whole SoC is powered by one source. This partitioning method allows a very good resolution in Iddq testing as well as it avoids interference due to background leakage without any DFT effort, area overhead or any other type of penalty.

In addition to the above design, a number of other design-rules are necessary to avoid unwanted high Idd states in the SoC IC.

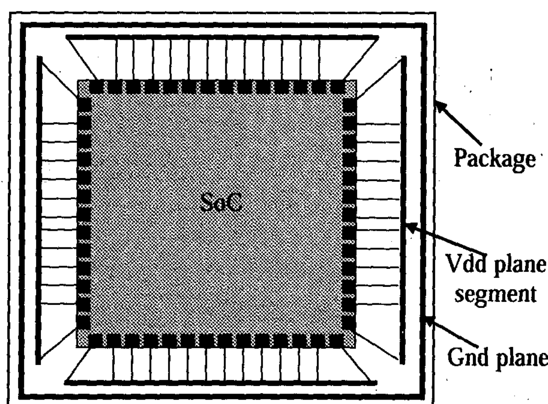

Fig. 8: SoC package with segmented Vdd plane that allows natural partitioning and Iddq testing on one segment at a time. Signal and Gnd connections are not shown in figure.

These rules are summarized as follows [13]:

1. All circuits need to be properly initialized. Full circuit initialization is a fundamental requirement for Iddq testing. Besides using set/reset of flip-flops or a dedicated signal, full-scan or partial scan can be used very effectively to initialize the circuit. It means that all flip-flops (registers) must be placed in a known state, such as flip-flops within a core when core is tested or glue logic when glue logic is tested. This initialization can be done by set/reset signal or through scan operation.
2. All static current dissipating logic within a core or glue logic must be switched-off using power-down control signals. This includes all memory sense-amps, any dynamic logic, asynchronous logic, pull-up/pull-down resistors, special I/O buffers and any analog circuitry.
3. All circuits including individual cores and glue logic must be stable at the strobe point. There should be no pending events during Iddq testing either within a core, from one core to another core, or at the chip-level.
4. All inputs and bi-direct pins of all individual cores as well as pins at the chip level must be at a deterministic 0 or 1.
5. At the core level, if an input, output or bi-direct pin is pulled-up, it should be at a logic 1, connected to Vdd through an on pMOS; if it is pulled down then it should be at a logic 0, connected to Gnd through an on nMOS. All pull-up and pull-down transistors at the chip I/Os must be switched-off.
6. All primitive nets within a core or glue logic with single driver must be checked for the following: (a) all nets are either at a logic 0 or at logic 1; (b) if a net is at x, either the driver should not be tri-stateable or it should not be driven by a tri-stateable gate whose enable pin is active; (c) any net should not be at a high-z state. These conditions are necessary to ensure that there is no internal bus conflict or floating nodes during Iddq testing.
7. When primitive nets are driven by multiple drivers, these must be checked for: (a) net should not be driven both 1 and 0 at any given time; (b) net should not be driven to 0/x, x/x, 0/0/x, 1/x, in all these conditions there is a potential conflict on the net; (c) net should not be driven to x/z, z/z/x, z/x/x, in these situation net is either potentially floating or has a conflict.
8. All nets within a core and glue logic should be checked that there is no weak value feeding to a gate during Iddq measurement. Similarly, there should not be a degraded logic value on a node feeding to a gate during Iddq measurement.
9. Special circuit structures should be avoided as much as possible. When such structures are unavoidable, power_down control signal should be used to switch-off these structures during Iddq testing. The examples of such structures are gate and drain/source of a transistor be driven by the same transistor group; feedback and control loops within one transistor group; substrate connection of nMOS being not at Gnd, substrate connection of pMOS being not at Vdd.
10. A standard cell library which contains components with low power switches and use of separate power supply for digital logic, I/O pad ring and analog circuit is also helpful. In this situation, Iddq testing on digital logic can be done easily.

It is important to mention that when individual components (all cores, PLL, memory sense-amps, pull-up/pull-down, dynamic logic etc.) are designed with a power-down control signal, a consistent naming convention significantly helps in the design. With a consistent naming convention no additional design effort is needed for the implementation of global control signals. During placement and routing, the router sees the same names throughout the chip from the TAP to the cores and connects them

appropriately to automatically form the power_down control signals.

6. Conclusions

In this paper, we described the test methodology for a system-on-a-chip that contains a microprocessor core. In this methodology, the microprocessor core is tested through a BIST-type arrangement for correctness of all the instructions. The opcode of the instruction-under-test is scanned into the TCR register, and the instruction is repeatedly executed for operands obtained by an LFSR; the result is compressed by an MISR and scanned-out for comparison. Once the microprocessor testing is completed, this core is used to test the on-chip memory and the other cores. For this purpose, an assembly language program is written, converted into binary form by the assembler and applied to the microprocessor core through a tester API. In addition to this functional testing, a small Iddq test set is used to detect physical defects. The design features to facilitate Iddq testing are described.

References

1. D&T Roundtable, "Testing embedded cores", IEEE Design and Test of Computers, pp. 81-89, April-June 1997.
2. J. Hutcheson, "Executive advisory: the market for systems-on-a-chip", and "The market for systems-on-a-chip testing", VLSI Research Inc.
3. IEEE P1500 CTAG web page.
4. Y. Zorian, "Test requirements for embedded core based systems and IEEE P1500", IEEE Int. Test Conf., pp. 191-199, 1997.
5. VSI Alliance, Manufacturing related test development WG specifications, 1998.
6. IEEE Standard 1149.1, "IEEE standard test access port and boundary scan architecture", IEEE Press, 1990.
7. D. K. Bhavsar and J. H. Edmondson, "Testability strategy of the alpha AXP 21164 microprocessor", IEEE Int. Test Conf., pp. 50-59, 1994.
8. J. Dreibelbis, J. Barth, H. Kalter and R. Kho, "Processor based built in self test for embedded DRAM", IEEE J. Solid State Circuits, vol. 33(11), pp, 1731-1740, Nov. 1998.
9. J. Saxena, P. Ploicke, K. Cyr, A. Benavides and H. Malpass, "Test stragegy for TI's TMS320AV7100 device", IEEE Int. Workshop on Testing Embedded Cores based Systems, pp. 3.2.1-3.2.6, 1998.
10. R. Rajsuman, "A new test method for testing embedded memories in core based system-on-a-chip ICs", IEEE Int. Workshop on Testing Embedded Cores based Systems, pp. 3.4.1-3.4.6, 1998.
11. F. Zarrinfar and R. Rajsuman, "Automated Iddq testing from CAD to manufacturing", IEEE Int. Workshop on Iddq Testing, pp. 50-55, 1995.
12. M. Colwell, R. Rajsuman, Z. Sarkari and R. Abrishami, US patent No. 5,644,251, July 1, 1997; and US patent No. 5,670,890, Sep. 23, 1997.
13. R. Rajsuman, "Design-for-Iddq-Testing for embedded cores based system-on-a-chip", IEEE Int. Workshop on Iddq Testing, 1998.