Microprocessor Based Testing for Core-Based System on Chip

C. A. Papachristou F. Martin M. Nourani
Computer Engineering Program, EECS Dept.
Case Western Reserve University
Cleveland, OH 44106

Abstract

The purpose of this paper is to develop a flexible design for test methodology for testing a core-based system on chip (SOC). The novel feature of the approach is the use an embedded microprocessor/memory pair to test the remaining components of the SOC. Test data is downloaded using DMA techniques directly into memory while the microprocessor uses the test data to test the core. The test results are tranferred to a MISR for evaluation. The approach has several important advantages over conventional ATPG such as achieving at-speed testing, not limiting the chip speed to the tester speed during test and achieving great flexibility since most of the testing process is based on software. Experimental results on an example system are discussed.

1 Introduction

Recent developments in semiconductor technology have made possible to design entire systems onto a single chip, commonly known as *System-On-Chip* (SOC) [10]. A related practice which is evolving is the use of predefined logic blocks called *Cores* or Macros [1]. A Core is a highly complex logic block which is fully defined in terms of its behavior, also predictable and reusable [4]. System designers can purchase cores from core-vendors and integrate them with their own user-defined logic (UDL) [8] to implement SOCs. We refer to these designs as core-based systems.

Core-based SOCs have significant advantages. Due to the fact that most of the system is on the same chip SOCs can operate faster with lesser power. SOCs reduce the number of discrete components used, thereby reducing the total size and cost of the end-product. Furthermore, using embedded cores in SOCs has the potential of greatly reducing the time-to-market because of the design re-use involved.

Testing core-based systems is a major challenge. The main factor is that the accessibility of the cores and blocks is greatly reduced. Furthermore, the system designer might have a restricted knowledge of the core internals due to the protection of the Intellectual Property (IP) of the cores [3]. The following factors should be considered to determine an effective test strategy when integrating cores into an SOC.
a) *Fault Coverage* : To obtain a high fault coverage, all the system blocks should be thoroughly tested. The interconnects between the blocks should also be tested. Other tests should be done as to whether the individual blocks function properly when they are interacting with each other.
b) *Overall test time* : Testers are very expensive thus it is important that the test time be kept as small as possible.
c) *Area Overhead* : The amount of additional silicon area needed to implement the test scheme should also be low.
d) *Performance Overhead* : Performance penalties should also be considered. The speed of the system and the amount of power consumed might be affected by the test scheme.

Unlike the way we test smaller designs, we cannot test SOCs as one whole unit. This is because such a test solution would give a poor fault-coverage and the overall test generation is impractical, and often not possible. A better way to test SOCs is by testing each of the cores separately, along with other tests to determine whether the system functions as a whole. Many types of isolation methods have been proposed for testing the SOC cores separately. A direct access scheme, [6] [2], isolates embedded cores by accessing all the I/Os of the core in parallel through a test bus. The method is very effective in terms of the total test time, however there is considerable area overhead in routing of the test bus. In [8, 9], a method is proposed using isolation rings to test embedded cores together with a serial scan chain to access the embedded cores. This method requires a much lesser area overhead, however the overall test time might be very high due to the serial access of the cores. Some work on using a microprocessor to test a core-based system is in [5].

• **Our Approach.** Our test scheme is based on using a microprocessor in the SOC to test the remaining embedded cores. Most system designs have some sort of microcontroller (or a similar component) in them. Hence our test scheme can be applied to a majority of core-based systems.

Using the microprocessor to test cores has significant advantages. The testing process could be done at-speed or near at-speed. Since testing is done inside the chip, the tester host can be run at lower speeds thereby reducing the need for high-speed expensive testers. The size of the test controller, if any, is very small because the microprocessor performs most of the test controller function. Also since most of the testing is specified in software the method is very flexible. With some modifications, the method can be even used for field testing.

2 Key Concept

• **Requirements.** The major components required for implementing our test strategy can already be found in most SOCs. The main component required is a microprocessor. Some memory should be available which can be used during test. We will refer to this as the test memory. Of course, this memory will be free for use during normal operation of the design. A path should be available from the system I/O pins to the test memory, so that the test data can be downloaded into the memory.

Testing the microprocessor and the test memory is not the focus of our work. We assume that the microprocessor and the test memory are tested by some other method [11].

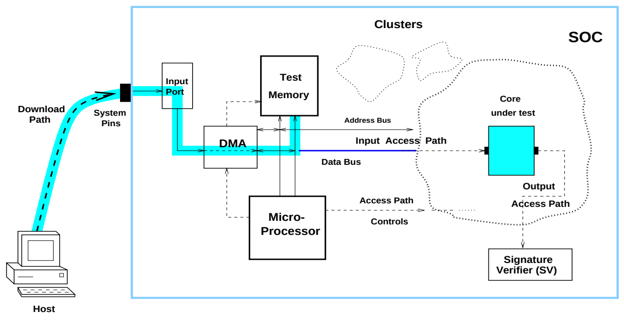


Figure 1: Structural test scheme

• Test Scheme. The key concept is illustrated in Fig. 1. The compressed test data is downloaded into the test memory from an external host, e.g. an inexpensive tester. In our example the download is done using a DMA (direct memory access) technique, so that the microprocessor does not have to worry about the download. Next, the microprocessor tests the cores using the test data in test memory. The cores can be separated into *clusters* of cores. Cores in the same cluster cannot be tested at the same time, whereas cores in different clusters can be tested in parallel. Cores in the same cluster can be directly connected to each other, however, cores from different clusters would not be interconnected directly. The advantage of separating cores into clusters is that potentially we can test cores in different clusters in parallel. In our example, we use the bypass approach [7] to access the cores. However, any other type of access mechanism can be used [12]. To test a core in a cluster, we provide an access path from the microprocessor to the core inputs and we also provide an access path from the core outputs to a block called the *Signature Verifier(SV)*. The latter is a programmable multiple input signature register (MISR) with capabilities of verifying the signature with the data provided by the microprocessor. The microprocessor enables the access path with the help of a simplified test controller and sends the test data to the core. After applying the test pattern to the core, the core outputs are propagated to the SV.

Figure 1 shows that there are three different paths: the download path, the input access path and the output access path. The input access path includes the path from the test memory to the microprocessor and the microprocessor to the core inputs. To make the testing faster, we overlap in a pipeline form the download and testing phases. The test memory acts as the buffer between the two stages (download and test) of the pipeline. It will be seen later that the ordering of the cores to be tested plays an important role in decreasing the total test time.

• Testing Issues. The key issue is the synchronization of the downloading and testing. Since the test memory size is limited, we should ensure that the amount of data downloaded does not exceed the test memory capacity. We do this by packetizing the test vectors into frames. The size of a frame is smaller than the size of the test memory. Each frame is downloaded only when there is enough space for that frame in memory. To read a frame, the microprocessor waits until the whole frame is downloaded.

In order to understand the test scenario, we will now describe in more detail the function of the microprocessor. The microprocessor should perform the following functions:
1) Before accessing each frame, it should be checking whether the DMA has downloaded already the required frame.
2) It should instruct the DMA to load the next frame as long as the new frame will not overwrite the frame currentnly being accessed. This is done using interrupts.
3) While testing a core, the microprocessor should configure the bypass paths, so that the correct input and output access paths to the core(s) under test are enabled. The microprocessor should also configure the SV to compact/verify the outputs as required.
4) In the case of pseudorandom testing, the microprocessor could generate the test patterns by itself. Hence a separate BIST controller is not needed for testing any BISTable cores, i.e. cores amenable to pseudorandom testing. This is a form of software BIST and it will done be much faster than any external pseudorandom testing. However, if there already exists a BIST hardware scheme for the core under test, the the microprocessor can control the BIST controller.

The following three aspects are considered for our testing scheme. They are described in detail later. i) Download Phase. ii) Testing Phase. iii) Test Management.

3 Download Phase

To download data into the test memory, we should ensure that there exists an access path the test memory from the system inputs ports. Usually such a path would exist in a system design. However, for designs which do not have direct access to the test memory, we should provide an alternate access path which will be enabled during testing. As mentioned earlier, the downloading is done using direct memory access (DMA). In general, DMA is a scheme which can be used to transfer data to/from the memory without using the microprocessor. During these non-memory cycles, the DMA takes control of the memory bus and reads/writes in the memory. This process of using the memory when the microprocessor is not using it is known as *cycle stealing*.

• Test Information Compression. The test vectors are of two types, *deterministic* and *pseudo-random*. For deterministic test patterns, the entire set of test vectors should be downloaded to the test memory. For pseudorandom test patterns, it is sufficient to download the seed(s) and polynomial(s) to generate random test patterns. Thus the downloading would be much faster for the case of pseudorandom patterns. The microprocessor can use the seed and the polynomial to generate the psuedo-random test patterns.

Test pattern compression or compaction prior to downloading is beneficial in that it reduces the download time. The tester memory requirements are also minimized. The compression is done at the test generation level. Decompression takes place inside the chip by the microprocessor. In order to keep the testing process effective, the decompression should take as little time as possible. This compression/decompression strategy will be very effective if the the microprosessor is much faster than the host.

One form of compression which is very useful is the elimination of duplicate patterns. Test patterns usually contain many duplicate entries. This is especially true in control-dominated cores. A series of duplicate input patterns may apply to the core inputs for several cycles, depending on the sequential depth of the core. However, in our approach we do not need to download duplicate patterns in test memory. We only need to download one pattern out of a series of N duplicates, as well as the series length N. The microprocessor can be instructed in software to reproduce the duplicates for the corresponding cores.

• Frame Packetization techniques. After the test data is compressed, it is packetized for download. The basic packetization technique for the test information to be downloaded is shown in Figure 2. The first step is arranging the test vectors of each core in the order they should be downloaded. This arrangement depends upon the test scheduling which

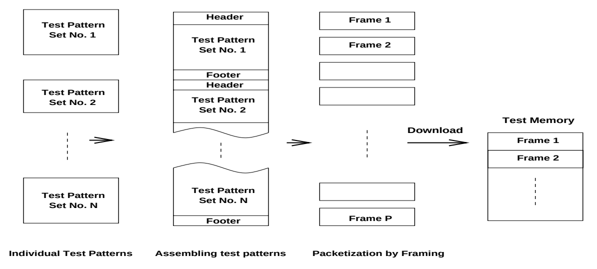


Figure 2: Test Data Packetization

is described later. The next step is to add headers and footers to the test vector set of each core in order to identify and separate them. The header contains the following information – the core to be tested, size of the test vector set, whether the vector is deterministic/pseudo-random, how to control the test controller and the signature verifier configuration. The headers also provide initialization information to the microprocessor.

The next step is to split the test data of each core into frames. A frame is a packet of test data which is smaller than the size of the test memory. The following rules ensure that downloading and testing synchronize with each other.
1) Each frame is transmitted to the memory in one stretch.
2) After a frame is downloaded, the DMA interrupts the microprocessor to inform that the frame download has been done. The microprocessor makes note of this.
3) Whenever the microprocessor starts to use the test data it verifies that the corresponding frame has been downloaded. If the frame is still being downloaded, then the microprocessor waits till the frame has been downloaded.
4) Also when the microprocessor is accessing a frame, no download should take place which will overwrite the frame being accessed.

The frame size f has to be decided carefully as it affects the overall testing time. If f is equal to the test memory size, M, then the microprocessor has to wait till the entire M has been downloaded. A new download cannot occur until the microprocessor has finished accessing the existing frame. This means both download and testing are never done in parallel. Hence $f = M$ represents entirely serial way of testing. On the other hand if $f = 1$, then the microprocessor starts accessing as soon as one test pattern has been downloaded. Downloading takes place as long as it does not overwrite the test pattern currently being accessed. Hence this represents the highest form of overlapping. However this means that for every test pattern the microprocessor should check whether the next frame has been downloaded, which would clearly slow down the testing process. An effective frame size is an intermediate value between 1 to M and should be chosen carefully by experimentation.

4 Testing Phase

In this phase, the microprocessor gets the test data from the memory, decompresses it and dispatches the data to the corresponding cores. This is done as follows. First the microprocessor decodes the header to identify which core it is supposed to test, the size of the test vector, decompression details and other relevant information. It then decompresses the compressed data into its original form. It then configures the system so that it can access the primary inputs of the core to be tested. The outputs of the core are also propagated to the signature verifier (SV), which is programmed accordingly to compare the outputs with the given response. After processing the header, the microprocessor reads in the test vector one by one and sends them to the core inputs. The outputs are verified/compacted by the SV. If the outputs have been compacted, then the microprocessor finally verifies this compacted signature once the footer is reached. It then proceeds to test the next core.

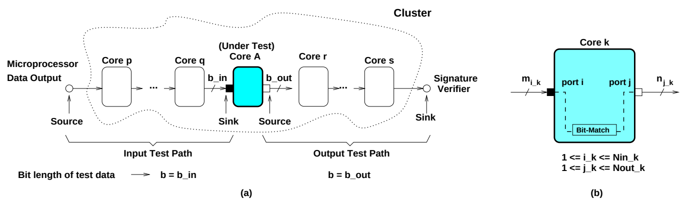


Figure 3: The Bypass Approach

4.1 Accessing the cores - The Bypass Approach

The bypass approach, adapted from [7], is briefly illustrated in Figure 3. The overall objective is to use the existing wires and topology of a cluster of cores to establish a path to carry test data between the core under test and the two test points in the system. Thus, there are two types of test paths, i.e. *input test path* accessing core input ports from the microprocessor data bus and *output test path* accessing core output ports from the signature verifier. Figure 3 shows a core under test (Core A) and the I/O test paths. Nin_k and $Nout\_k$ are the total number of input and output ports of the core k. All the cores shown in Fig. 3 belong to the same cluster.

Figure 3 also shows a general view of our bypass model, and Figure 3(b) the blowup picture of a core bypassing data in the path. We symbolically showed that the inputs are bypassed to the output without interfering with the core circuitry which are used in normal mode. The basic idea of the bypass mode for each core is to have an independent route around a core to carry *test data* (predefined test patterns or core signatures) between $port_i$ (m_{i_k} bit wide) and $port_j$ (n_{j_k} bit wide) of that core. $Port_i$ and $Port_j$ represent two ports of a core and m_{i_k} and n_{j_k} are the bit-widths of $port_i$ and $port_j$ of core k. This is shown in Figure 3(b). The objective here is to establish the shortest path (fastest route) to carry the packed test data between source and sink. Note that by this formulation, accessibility of the core inputs from the microprocessor and of the core outputs from signature verifier, are similar problems.

Figure 3(a) clearly shows that the bit width of core inputs or outputs change in a path between two test points. This requires a sort of bit matching. Let's assume that we need to transfer a b bit pattern between source and sink. In general, to transfer b bit test data from $port_i(m_{i\_k}$ bit) to $port_j(n_{j\_k}$ bit) of *Core k* we need to pack the data (to match the available bit width) and send it in several iterations. For example, a core with a $m = 4$ bit input port bypasses $b = 16$ bit test data in 4 iterations. The time cost for serial to parallel or parallel to serial transfer is:

$$Time\ Cost:\quad t_{ij\_k} = \lceil \frac{b}{min\{m_{i\_k}, n_{j\_k}\}} \rceil \ \ cycles \qquad (1)$$

Using these cost values, we find the shortest (fastest) path to access the cores. To do this, the cluster is modeled as a graph as explained in the following.

• **Graph modeling and the shortest path problem.** The objective here is to model the port accessibility of cores within a cluster as a directed weighted graph in which the shortest path between any two points (called source and sink) reflects the fastest route to transfer packed test data between those two points. From testing perspective, with such model we

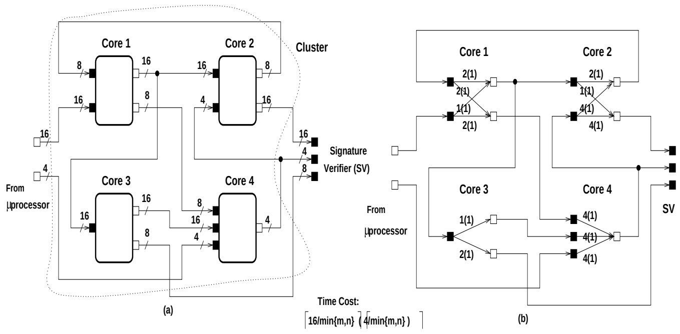


Figure 4: A cluster and its CBG graph.

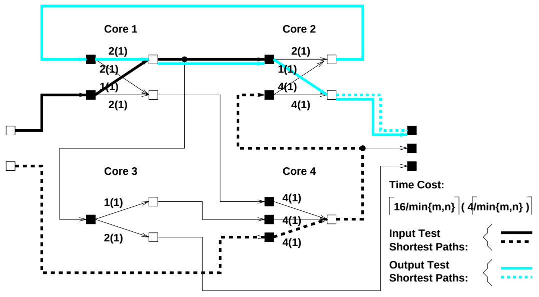


Figure 5: For input/output shortest paths for *Core 2* using Branch-and-bound Algorithm.

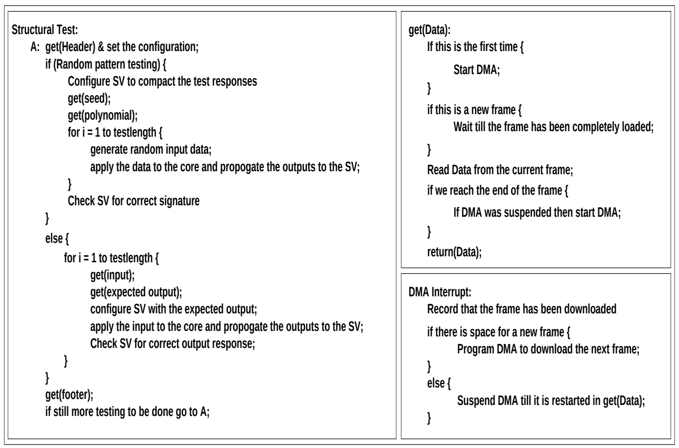
can find the fastest route to transfer test data (predefined or random pattern) from the microprocessor to any of the core input ports. Similarly, we can find the fastest route to transfer test data (signature) from any of the core output ports to the SV.

In our graph model, a node corresponds to a *port* and an edge corresponds to the interconnects between ports or the bypass possibilities. From Equation 1, the cost of an edge corresponds to the bypass delay or the transfer time of the packed data from one point to another. The time cost of the existing interconnects between cores is assumed to be zero since no additional circuit/delay for packing or transfer-control is needed. We call the above graph model the *Core Bypass Graph*.

Figure 4(a) shows a cluster, made of four cores with different ports and bit widths, under test. The cluster has two primary inputs going to *Core 1, 3* and three primary outputs from *Core 2, 3, 4*.

Figure 4(b) shows the corresponding *Core Bypass Graph* (CBG). Depending on the bitwidth of test data (b) different cost values should be used in finding the shortest paths. In Figure 4(b) near each edge we show two cost values. The cost values outside parenthesis are $t_{ij\_k} = \lceil \frac{16}{min\{m_{i\_k}, n_{j\_k}\}} \rceil$ showing the time overhead to transfer 16-bit test data. Similarly, the cost values inside parenthesis are $t_{ij\_k} = \lceil \frac{4}{min\{m_{i\_k}, n_{j\_k}\}} \rceil$ reflecting the time overhead to transfer 8-bit test data. All edges without a cost value correspond to the existing interconnect between cores and are assumed to have time cost of zero because no packing cost is involved. We use a branch and bound algorithm to find the shortest path from the microprocessor to the core primary inputs and from the core primary outputs to the SV.

To show the process, we continue our running example by applying the branch and bound algorithm to the CBG graph of Figure 4(b) for testing *Core 2* only. The algorithm found four paths, two to reach the two primary inputs of



```
Structural Test:
    A:  get(Header) & set the configuration;
        if (Random pattern testing) {
            Configure SV to compact the test responses
            get(seed);
            get(polynomial);
            for i = 1 to testlength {
                generate random input data;
                apply the data to the core and propogate the outputs to the SV;
            }
            Check SV for correct signature
        }
        else {
           for i = 1 to testlength {
                get(input);
                get(expected output);
                configure SV with the expected output;
                apply the input to the core and propogate the outputs to the SV;
                Check SV for correct output response;
           }
        }
        get(footer);
        if still more testing to be done go to A;
```

```
get(Data):
    If this is the first time {
        Start DMA;
    }
    if this is a new frame {
        Wait till the frame has been completely loaded;
    }
    Read Data from the current frame;
    if we reach the end of the frame {
        If DMA was suspended then start DMA;
    }
    return(Data);
```

```
DMA Interrupt:
    Record that the frame has been downloaded
    if there is space for a new frame {
        Program DMA to download the next frame;
    }
    else {
        Suspend DMA till it is restarted in get(Data);
    }
```

Figure 6: Pseudocode of the test software

Core 2 from the microprocessor and two to observe the two primary outputs of *Core 2* to the SV. Note that *Core 2* has one 4-bit, one 8-bit and two 16-bit ports. So, we consider the appropriate cost of edges accordingly. That is, the cost values outside parenthesis when a test point is the 16-bit port and the cost values inside parenthesis when a test point is the 4-bit port. The result is summarized in Figure 5. that shows the shortest paths between microprocessor and core input ports, with thick solid and broken lines. It also shows the shortest paths between two output ports of *Core 2* and the SV using the two different lines.

4.2 The function of the microprocessor

The microprocessor acts as the main test controller, coordinating the downloading and the testing of the cores. It also controls the bypass paths to access the cores. All these functions can be implemented in software. A separate section of the memory should be allocated for this test software.

A pseudocode of the test software present in the memory is shown is Figure 6. First of all the module gets the header of the test data. This is obtained by calling the `Get(Data)` procedure. Then the `Structural Test` module configures itself and prepares to test the required core. If it is random pattern testing, it first configures the SV in order to compact the output test responses. The random patterns are generated by the microprocessor and applied to the core inputs and propagated to the outputs of the SV. After the required `testlength`, the SV is then checked for the correct signature. In the case of deterministic patterns, each of the test vector is obtained from the test memory and applied to the cores. Then the core outputs are checked by using the SV. The `Get(Data)` block gets the data from the test memory. During the first time, the `Get(Data)` block starts the DMA and instructs it to download the first frame. Whenever the `Get(Data)` is reading from a new frame, it checks and waits if necessary for the next frame to get completely loaded. This is done by checking whether the frame marker has been downloaded. Also when the `Get(Data)` has finished reading all the data in a frame, it checks if the DMA is waiting for the test memory to clear up. If so it informs the DMA that it has finished reading the current frame and that the DMA can download the next one.

The `DMA interrupt` procedure is called whenever the DMA interrupts. This happens when a frame has been downloaded and the DMA wants to signal the completion of the download of a frame. The procedure instructs the DMA to download the next frame as long as the test memory has enough space. If the test memory does not have enough space, then it suspends the downloading till the DMA gets activated again by the `Get(DATA)` block.

5 Test Management

The order in which we test the cores can affect the overall core testing time. Consider tesing two cores C_1 and C_2. Suppose the dowload time for the test vectors of C_1 and C_2 are 100 and 50 milliseconds respectively. This is just the raw download time i.e. the time taken without considering the memory size and the microprocessor testing. The time taken for the microprocessor to test the cores with the downloaded test data is estimated to be 50 and 100 milliseconds, without consideration of the download phase. These times can be calculated analytically or by simulating the download and testing individually.

Let us also assume a frame size of $f = M/10$ where M is the test memory size. If we download and test C_1 followed by C_2, then the total test time for both C_1 and C_2 would be 210 milliseconds. On the other hand, if we download and test C_2 followed by C_1, then the total test time would only be 160 milliseconds. This is seen in the scheduling chart of Figure 7. The chart shows when each of the frames are downloaded and tested. Note that the time to download is a constant, since we use the same download path for testing all cores. This is seen by the constant width of the downloads in Figure 7. We have assumed that a single frame takes 10 milliseconds to download. However, the time taken to test a frame depends upon the core. This is because the accessing time due to bypass may vary from core to core. In this case, for C_1 it is 5 milliseconds and for C_2 it is 20 milliseconds per frame. In Figure 7 the widths of the testing for C_2 is longer that that of C_1 to reflect the different testing times.

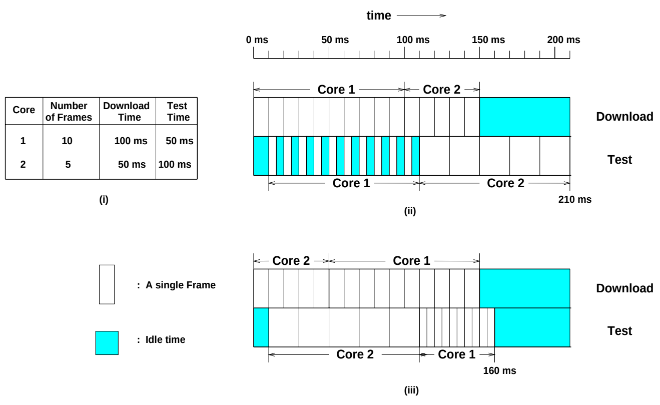


Figure 7: Test management example (i) Testing and Downloading times; (ii) Schedule 1 (iii) Schedule 2

The primary reason why the order in which we test the cores affects the overall test time is because different cores have different download and test times. For those cores which have a shorter download than test time, the test time is the bottleneck. For cores which have download time greater than their test time, downloading is the bottleneck. Hence if we schedule the testing in such a way that the cores with download bottleneck come only after cores with test time bottleneck, then we would have the shortest test time possible, if the memory were unlimited. This is shown in Figure 8. If D represents the download times and T represents the testing times, first we compute the difference $g = D - T$, then we arrange them in increasing order. This ordering would give a test schedule with minimal test time.

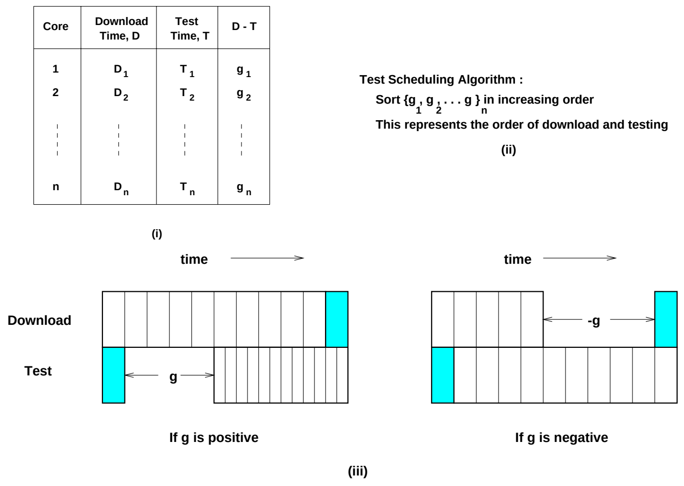


Figure 8: Test Scheduling Algorithm (i) Dowload, Test times; (ii) The algorithm; (iii) Interpretation of g

Note that we have not considered the test memory size in this algorithm. The test memory size determines the maximum time between the download and testing of a frame. Hence if the test memory is full, no further download will take place until a frame has been tested.

6 Experimental Results

A sample design modeling an alarm clock (Synopsys documentation) was tested using the above methodology. The top level blocks in this design were considered to be individual cores. This circuit was assumed to be one of the clusters of a SOC as shown in Fig. 9.

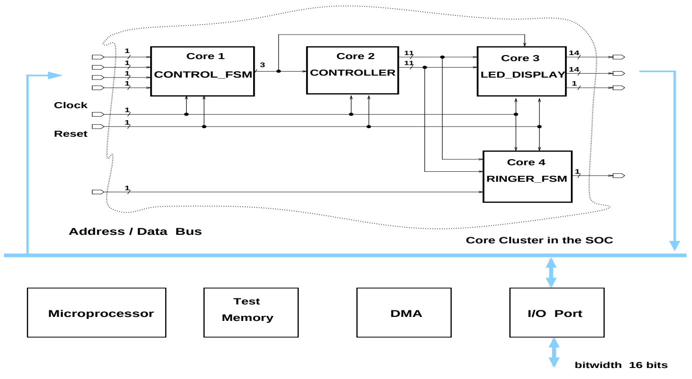


Figure 9: The Alarm Clock Circuit Embedded in a SOC

The test patterns for each of the cores of the alarm clock were generated using an ATPG tool from Synopsys. The fault coverage of the individual cores are shown in Table 1. The faults at the ports of the cores were removed as they are considered interconnect faults when the cores are embedded in the system. Now the structural description of the cluster was analyzed, a core bypass graph was obtained and the shortest path required to access each core was determined. The bypass paths for accessing all the four cores of this circuit are shown in Figure 10. The input and output access paths for testing Core 2 are also shown. In order to test the entire cluster of cores, a simple microprocessor model was assumed. The test software (Figure 6) was simplified for this example and the software size was estimated. The test pat-

Core	Test Length	Total No. of faults	Untested faults	% Fault Coverg
Control_FSM	71	196	2	98.98
Controller	610	1356	39	97.12
Led_Display	147	2378	25	98.95
Ringer_FSM	435	530	83	84.34

Table 1: Testing Information of Individual Cores

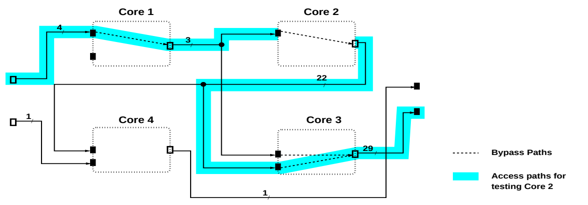


Figure 10: Bypass paths necessary for testing the cluster

terns for the cores were analyzed for removing the duplicate patterns that were generated by ATPG. The compression results are shown in table 2. Column 2 shows the test pattern length of the individual cores. In order to download them, the test patterns are re-arranged to 16-bit patterns. This is because the width of the system IO pins is 16-bit. This is shown in column 3. Column 4 shows the length of the test patterns after removing the duplicates. While removing the duplicates, the duplicate vectors are replaced by a pointer which points to the real value. Hence removing the duplicates might sometimes increase the test pattern length. This is seen for Cores 1 and 2. However the test patterns for Cores 3 and 4 could be compressed this way. For the core test, we compress the test patterns only for cores 2 and 4.

The entire cluster was fault simulated with the patterns from the microprocessor. The overall fault coverage of the cluster is 96.23%. The complete result is tabulated in Table 3. The table shows the result when all the four cores were tested by the bypass approach. The embedded cores have the same fault coverage as they had individually table 1.

The interconnects are completely tested since they were used to send the test vectors to test the cores. The test (bypass) circuitry was also tested well with a fault coverage of 96.84%. The overall faults and fault coverage is shown in the last row of Table 3.

In order to compute the overall test time, the following assumptions were made: The external tester speed was taken to be 50 Mhz. The number of microprocessor cycles necessary to send a test pattern to the core was estimated to be 4 cycles. The overall test time was estimated for different chip speeds. These results are shown in Table 4. Note that the the first row gives the overall test time if the chip was tested directly from the tester. Hence the tester speed and the chip speed are the same. The next three rows show the overall test time, if the chip was tested using the microprocessor. Note that there is significant advantage, when the chip speed is increased. By using our method, the chip can be tested at much higher speeds with a low speed tester without sacrificing the overall test time.

Core	Test (a)	Test (b)	Compressed (c)	Use Compression
Control_FSM	71	26	35	No
Controller	610	190	160	Yes
Led_Display	147	248	4090	No
Ringer_FSM	435	706	647	Yes

Table 2: Test Pattern Compression. (a) Core test pattern length; (b) test pattern length using chip I/Os (16-bit); (c) Compressed test pattern length (w/o duplicates)

Block	Total Number of faults	Untested faults	Fault Coverage
Control_FSM	196	2	98.98%
Controller	1356	39	97.12%
Led_Display	2378	25	98.95%
Ringer_FSM	530	83	84.34%
Bypass Circuitry	1236	39	96.84%
Interconnects	96	0	100%
Complete Circuit	5792	188	96.75%

Table 3: Test results of the entire cluster

Test Method	Tester Speed	Chip Speed	Overall Test time
Regular Method	50 MHz	50 MHz	98.16 μsec
Our Method	50 MHz	50 MHz	101.48 μsec
	50 MHz	100 MHz	51.24 μsec
	50 MHz	200 MHz	27.64 μsec

Table 4: Overall test time

7 Conclusion

Microprocessor-based testing has important advantages over traditional testing methods for core-based SOCs because we exploit the regular functionality of the chip during testing. Fault coverage also improves, because an embedded microprocessor has far greater accessibility than an external tester. Since we divide the testing process into the download and testing phase, the chip speed is not limited to the tester speeds. Hence the overall testing time is reduced.

References

[1] F.P.M. Beenker, R.G. Bennetts and A.P. Thijssen, "Testability Concepts for Digital ICs, The Macro Test Approach," *Kluwer Acad. Publishers*, 1995.

[2] L. Whetsel, "An IEEE 1149.1 Based Test Architecture for ICs with Embedded IP Cores," *Intern. Test Conf. (ITC-97)*, Nov. 1997.

[3] K. De, "Test methodology for embedded cores which protects intellectual property," *VLSI Test Sym. (VTS-97)*, pp. 2-9, May 1997.

[4] R. Chandramouli and S. Pateras, "Testing Systems on a Chip," *IEEE Spectrum*, pp. 42-47, Nov. 1996.

[5] I. Ghosh, N. Jha and S. Dey "A Low Overhead Design for Testability and Test Generation Technique for Core-Based Systems" *Intern. Test Conf. (ITC-97)*, Nov. 1997.

[6] V. Immaneni and S. Raman, "Direct Access Test Scheme – Design of Block and Core Cells for Embedded ASICs," *Intern. Test Conf. (ITC-90)*, pp. 488-492, Oct. 1990.

[7] M. Nourani and C. Papachristou, "Parallelism in Structural Fault Testing of Embedded Cores," *16th VLSI Test Sym. (VTS-98)*, pp. 15-20, April 1998.

[8] N. Touba and B. Pouya, "Testing embedded cores using partial isolation rings," *VLSI Test Sym. (VTS-97)*, pp. 1016, May 1997.

[9] N. Touba and B. Pouya, "Modifying User-defined Logic for Test Access to Embedded Cores," *Intern. Test Conf. (ITC-97)*, Nov. 1997.

[10] "VSI Alliance", *Architecture Document*, Version 1.0, 1997.

[11] A.J. van de Goor and Th. J. Verhallen, "Functional Testing of Current Microprocessors," *Intern. Test Conference (ITC-92)*, pp. 684-695, Sept. 1992.

[12] J. Aerts and E. J. Marinissen, "Scan Chain Design for Test Time Reduction in Core-Based ICs," *Intern. Test Conference (ITC-98)*, Oct. 1998.