A Fast and Low Cost Testing Technique for Core-based System-on-Chip

Indradeep Ghosh† Sujit Dey‡ Niraj K. Jha†

†Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

‡Department of Electrical & Computer Engineering
University of California, San Diego
La Jolla, CA 92093-0407

Abstract

This paper proposes a new methodology for testing a core-based system-on-chip (SOC), targeting the simultaneous reduction of test area overhead and test application time. Testing of embedded cores is achieved using the transparency properties of surrounding cores. At the core level, testability and transparency can be achieved by reusing existing logic inside the core, and providing different versions of the core having different area overheads and transparency latencies. At the chip level, the technique analyzes the topology of the SOC to select the core versions that best meet the user's desired test area overhead and test application time objectives. Application of the method to example SOCs demonstrates the ability to design highly testable SOCs with minimized test area overhead, minimized test application time, or a desired trade-off between the two. Significant reduction in area overhead and test application time compared to an existing SOC testing technique is also demonstrated.

1 Introduction

System-level integration is evolving as a new paradigm in system design, allowing an entire system to be built on a single chip, using pre-designed functional blocks called cores. While SOC is proving to be very useful in meeting aggressive time-to-market, performance, and cost requirements of today's electronic products, testing such core-based SOCs poses a two-fold challenge. Core-level testing involves making each core testable - inserting the necessary design for testability (DFT) structures and generating test sequences. Typically, for hard and firm cores, testability is ensured, and precomputed test sets provided by the core provider. For soft cores, testability can be addressed and test sets generated by the user. When the cores are integrated into an SOC, chip-level testing needs to be addressed by the SOC designer. The main difficulty in chip-level testing is the problem of justifying precomputed test sequences of a core embedded deep in the design from the chip inputs, and propagating the test responses from the core outputs to the chip outputs.

Core-level testing can be done through automatic test pattern generation (ATPG) at the logic level and by employing a variety of DFT techniques like full or partial scan, and built-in self-test (BIST). At the chip level, a number of DFT techniques have been proposed [1]. In one existing DFT method, referred to as FSCAN-BSCAN, each core is made testable by full scan while chip-level testability is obtained by isolating each core using boundary scan [2]. This scheme may have large area and delay overheads, and prohibitively large test application times. Recently, some work has been done to address this problem [3]. Another existing DFT method, utilizes a combination of full scan and test bus. In this method, an added test bus runs from the PIs of the system to its POs and uses a series of multiplexers to isolate each embedded core, which is full scanned, during testing to provide system-level testability. In this case as well, the area and delay overheads can potentially be quite large. In addition, the test bus architecture is unable to test the interconnect that exists between cores.

In [4], a DFT method is described to test macro blocks inside a circuit with a heavy reliance on full/partial scan and boundary scan whose disadvantages have been stated above. Though functional information of modules is sometimes used to reduce test overhead by utilizing the concept of module transparency, the techniques for introducing transparency are *ad hoc*. In [5], a new technique is introduced to make each core transparent. In the test mode, a transparent core can propagate test data from its inputs to outputs without information loss. During testing of the SOC, the transparency property of cores is used to propagate precomputed core test sequences from the chip inputs to the core inputs, and test responses from the core outputs to the chip outputs. While successful in lowering the test area and delay overheads compared to existing techniques, the technique in [5] suffers from the drawback of relatively large test application times due to the potentially large transparency latency of each core (number of cycles needed to propagate test data from inputs to outputs of the core). Also, functional description of the cores are required to make them transparent which may not be available for many cores, including legacy cores.

In this work, we propose a new DFT technique for core-based SOC testing that simultaneously targets reduction of area overhead and test application time. It has two parts. The first part consists of core-level DFT and test generation to make each core testable and transparent, and generate a precomputed test set for the core. For the core-level DFT, we use the low-cost high-level scan (HSCAN) technique [6]. Transparency of a core is achieved by reusing existing paths in the core, including HSCAN paths. The core-level DFT and transparency method uses only structural information of the core, in contrast to functional information used in [5], which may be harder to extract for some cores. This task needs to be performed by the core provider in case of hard and firm cores, and the user in case of soft cores. During the core testability phase, various versions of the same core are synthesized. For each version, the following are specified: the appropriate DFT structures, testability and transparency area overheads, transparency latency for different input/output pairs, and the test sequence for testing each core. Note that this is a one-time cost incurred by the core provider or the user. The second part consists of chip-level DFT and test generation which is to be performed by the user. For this, we propose a technique that analyzes the given interconnection of various cores in the SOC, the test set size of each core, and the area overhead/transparency latency characteristics of the available core versions, and identifies the most suitable version of each core such that the desired test area overhead/test application time trade-off is achieved for the SOC. Testing of each embedded core can be performed by justifying (propagating) the test sequences (responses) from (to) the chip inputs (outputs) using the transparency mechanism of the cores in the SOC.

The proposed technique can be used to design an SOC with high fault coverage such that: (i) the test area overhead is minimized, (ii) the test application time for testing the SOC is minimized, or (iii) a desired trade-off is obtained between the test area overhead and test application time. This technique is suitable for testing the SOC in a hierarchical fashion. Even if soft cores are used in the SOC, sequential test pattern generation subsequent to integrating

Permissions to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 98, San Francisco, California
(c) 1998 ACM 1-58113-049-x/98/06 ..$5.00

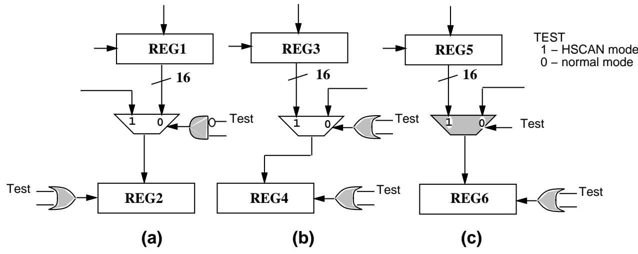

Figure 1: Structures employed by HSCAN

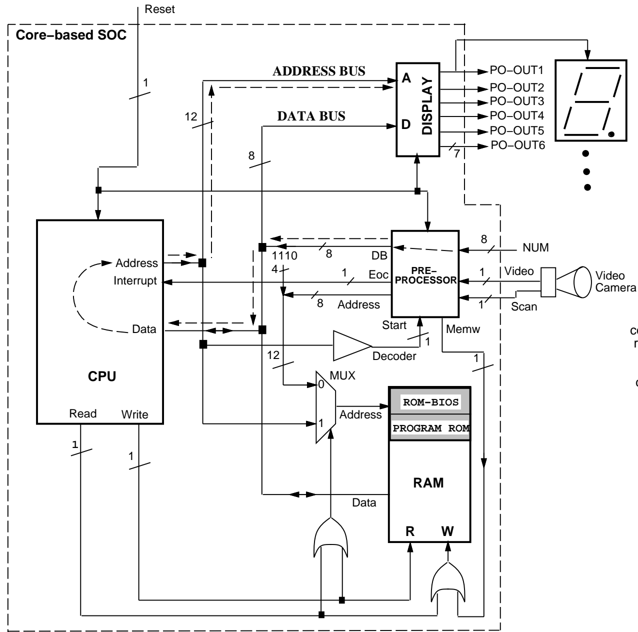

Figure 2: An example core-based embedded system

all cores into a single design can be computationally prohibitive. This problem does not arise in our method. Significant reduction in the area overhead and test application time compared to an existing SOC testing technique is demonstrated for two example SOCs.

2 Preliminaries

In this section, we briefly describe the register-transfer level (RTL) HSCAN methodology and illustrate its use as our underlying core-level testing technique [6]. It utilizes existing paths between registers, through multiplexers (mux paths), to connect registers in parallel scan chains from the circuit inputs to the circuit outputs. Consider the example RTL circuit shown in Figure 1, which shows 16-bit multiplexers and registers. Since a multiplexer path already exists between *REG1* and *REG2*, in the HSCAN mode these registers can be connected in 16 parallel scan chains by using just two extra logic gates, as shown in Figure 1(a). If the select-0 path of an existing multiplexer needs to be chosen during testing then a configuration like Figure 1(b) can be used. If a direct connection exists between two registers, only an OR gate is required at the load signal of the destination register. If no path exists between two registers, or if there is a conflict with already created HSCAN paths, then a scan path is created by adding a test multiplexer, as shown in Figure 1(c). This multiplexer can be integrated with the destination flip-flops to create a set of scan flip-flops to reduce test overheads further. Since it is a full scan technique, only combinational test pattern generation is required which makes it suitable for tackling large designs like cores.

To achieve transparency in a core, the existing parallel paths between registers as well as the ones introduced by HSCAN can be utilized to transfer data from the inputs of the core to its outputs making the HSCAN scheme suitable for obtaining low-cost transparency for each core. This is explained next.

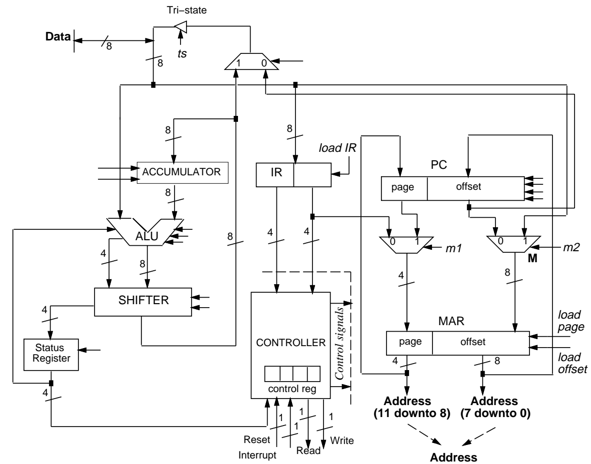

Figure 3: The CPU core (*Version 1*)

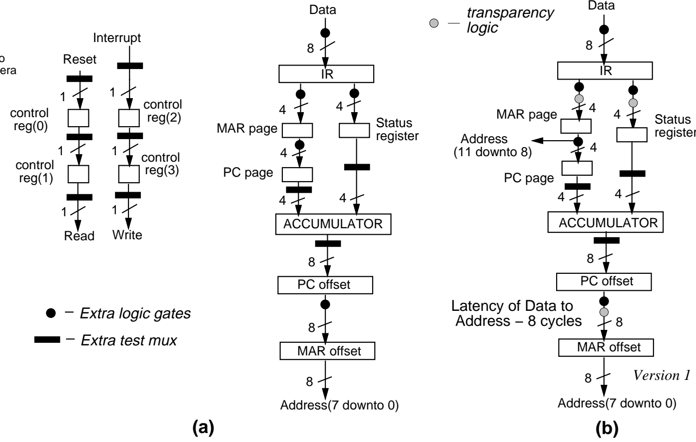

Figure 4: HSCAN and transparency chains in the CPU core

3 Outline of the SOC Test Method

In this section, we describe our core-based testing method using an example core-based embedded system that implements a barcode scanning system. This embedded system consists of the following cores: CPU, PREPROCESSOR, DISPLAY, RAM and ROM, as shown in Figure 2. The PREPROCESSOR receives signals from a video scanner (signal *Video*), processes the barcode scanned, and if no errors are detected, writes the width of the black and white bars in consecutive locations in the RAM. Subsequently, the CPU uses an embedded program in the ROM to convert the barcode in the RAM to a cost. The DISPLAY then converts the binary coded decimal output of the CPU to a series of six seven-segment display codes. The memory space is 4KB long, organized as 16 pages of 256 byte each, and all ports are memory-mapped.

The internal structure of the CPU core is shown in Figure 3 [7]. In the figure, *PC* represents the program counter, *MAR* represents the memory address register, and *IR* represents the instruction register. To make the CPU testable, we use HSCAN to obtain the scan chain configuration shown in Figure 4(a). Each flip flop/register in the circuit now belongs to one scan chain which can shift in the tests and shift out the responses.

Next, we need to make the CPU transparent. This means that each output of the core can be justified from at least one input or a combination of inputs of the core, and each input of the core can be propagated to one output or a combination of outputs of the core, in a fixed number of cycles. First, we try to utilize the HSCAN chains for transparency. Some extra transparency logic may be required, so that we can freeze values along the chain whenever required. If HSCAN is not the underlying test methodology for the core, we can use other techniques given in Section 4 to achieve transparency. The transparency mechanism for the CPU is shown in Figure 4(b). Assume that the core output *Address(7 downto 0)* needs a symbolic value α. If α is applied at the core-input *Data*,

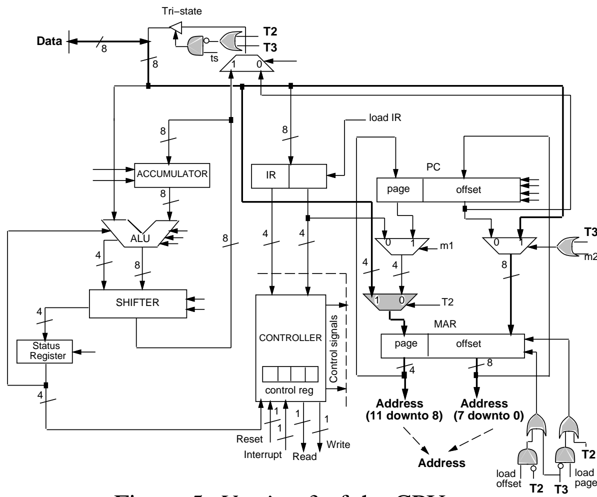

Figure 5: *Version 3* of the CPU core

then the core output *Address(7 downto 0)* will get α after six clock cycles. While α is being shifted from the core input to output, the data in the *Status register* needs to be frozen for a cycle so that the latency through the two parallel chains are balanced. This requires some extra transparency logic. Subsequently, the value required at the other core output *Address(11 downto 8)* can be applied at the core input *Data*. The required value appears in register *MAR page*, which is connected to *Address(11 downto 8)*), after two clock cycles. Also, the HSCAN chains can be used to transfer the value at input *Reset* to output *Read* in two cycles, and input *Interrupt* to output *Write* in two cycles. However, these single-bit chains are not explicitly shown in Figure 4(b). If all the bits of the core output *Address* need to be justified simultaneously from input *Data*, then some extra logic is required to freeze the values at output registers *MAR page* and *MAR offset*.

Next, we briefly describe our methodology of testing each embedded core, using transparency of the other cores in the SOC. We demonstrate the importance of the transparency latency of a core in determining the test application time of a core under test, and that of the entire SOC. We show how the transparency latency can be traded off for area overhead to reduce the test application time. Consider the DISPLAY in the SOC of Figure 2, which needs a test sequence of 525 HSCAN vectors for achieving 100% test efficiency (105 full-scan vectors are required and the sequential depth of the longest HSCAN chain is 4). To test the DISPLAY, its inputs *A* and *D* need to be controlled and its outputs *PO-PORT1* through *PO-PORT6* need to be observed. To control input *A* from the chip inputs, the test vector needs to be propagated through the PREPROCESSOR from its input *NUM* to its output *DB* and then through the CPU from its input *Data* to its output *Address*. This path is shown with the help of a dashed line in Figure 2. Input *D* of the DISPLAY can be controlled by propagating the desired vector through the PREPROCESSOR from its input *NUM* to its output *DB*. The outputs of the DISPLAY can be observed directly as they are the chip outputs.

Suppose the PREPROCESSOR can transfer data from input *NUM* to output *DB* in one cycle. The CPU takes a total of eight cycles to transfer a test vector from its input *Data* to output *Address* (six cycles to transfer a vector from *Data* to *Address(7 downto 0)*, and two cycles to transfer a vector from *Data* to *Address(11 downto 8)*). Here, we have assumed that test data cannot be pipelined through a core. Thus, if two transparency paths in a core share some logic between them, then data through one path can be propagated only after data has been completely propagated through the other path. Hence, a test vector can be propagated from the chip input *NUM* to the core-input *A* in nine cycles, where we have assumed that test data can flow through different cores simultaneously. In between, the test vector at input *D* can also be made ready. A subsequent test vector for the DISPLAY can be made ready at its inputs in another nine cycles. In the intermediate cycles, when the data at the inputs of the DISPLAY are not valid, the scan clock of the DISPLAY needs to be frozen. During this period, the test

CPU	Latency (cycles)			Overhead (cells)
	D -> A(7-0)	D -> A(11-8)	D -> A(11-0)	
Version 1	6	2	8	3
Version 2	1	2	3	10
Version 3	1	1	2	30

A– Address

Figure 6: Transparency latency *vs* overhead trade-off

response can be scanned out through the primary outputs. Thus, the test application time required for testing the DISPLAY alone is $525 \times 9 + 3 = 4,728$ cycles. The last three cycles are required to scan out the last test response. Since the DISPLAY core has 66 flip-flops and 20 internal inputs, an FSCAN-BSCAN approach would have needed $(66+20) \times 105 + (66+20) - 1 = 9,115$ cycles to do the same.

The transparency latency through the CPU can be reduced at the expense of area overhead, by exploiting existing alternative connections in the CPU and some extra transparency logic. In this case, output *Address(7 downto 0)* can be justified from the input *Data*, with some extra logic at the select line of multiplexer **M** in Figure 3, in one cycle. Also, *Address(11 downto 8)* can be justified from *Data* in two cycles. This transparency logic is added in addition to the HSCAN logic. Now, the CPU takes a total of three cycles to transfer a test vector from its input *Data* to output *Address*. If this version (named *Version 2*) of the CPU is used in the chip, the test application time for testing the DISPLAY is now $525 \times 4 + 3 = 2,103$ cycles only. The latency of the CPU core can be further reduced by using one extra transparency multiplexer to form *Version 3* of the CPU as shown in Figure 5. In this configuration, both outputs *Address(7 downto 0)* and *Address(11 downto 8)* can be justified from input *Data* in one cycle. The test application time for testing the DISPLAY is now reduced to $525 \times 3 + 3 = 1,578$ cycles. The results are summarized in the table in Figure 6. In the table, the area overheads are for the extra transparency logic only. The numbers are obtained after technology mapping with a .8µm cell library using an in-house synthesis tool.

In general, more expensive versions can be used for critical cores whose transparency latency affects the test application time of many other cores in the SOC.

4 Making Cores Transparent

In this section, we explain the method we have developed to make cores transparent with some given transparency latency. At first, if HSCAN is the underlying DFT methodology, we try to exploit the HSCAN chains to achieve transparency. If HSCAN is not the underlying test methodology or if transparency is not possible through the HSCAN chains or if any of the transparency latencies for an input or output is not acceptable, we introduce extra logic into the circuit to create transparency paths.

First, we extract a register connectivity graph (RCG) which consists of input nodes, output nodes, and register nodes of the core. An edge is present between two nodes if a direct or multiplexer path exists between them. Figure 7 shows the initial RCG of the CPU core. In the figure, the edges corresponding to the HSCAN paths are darkened. In many RTL circuits, bit-slicing is common, where a subset of bits (bit-slice) of a register is involved in an operation (as source or destination register), instead of the complete register. In such cases, the corresponding register node in the RCG is marked as a split node. When different bit-slices of a register must receive data from different sources, the register node is termed a C-split node. To control a desired value in a C-split node, the different bit-slices need to be controlled through different paths. In Figure 7, *ACCUMULATOR* is a C-split node. Similarly, if the fanout of a node is split into different bit-slices going to different destinations, the node is marked as an O-split node.

From the RCG, we try to find transparency paths from inputs to outputs. To do this, we do a breadth-first search (BFS) from each input node until we reach an output node. During this search, anytime we reach an O-split node, the BFS is done from each of the nodes at the fanout edges of the O-split node, as all the fanout edges need to be used for propagating the data at the O-split node. At first, we only use the HSCAN edges during this search. If we fail

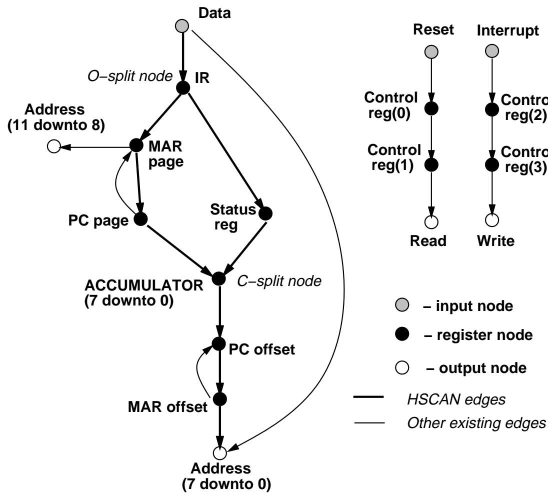

Figure 7: RCG of the CPU core

to get a transparency path, we try to use other existing edges in the RCG. For the CPU core, the BFS from input *Data* reaches node *IR*. Then, different BFSs start along the two fanout edges. The search from the left edge terminates at output node *Address(11 downto 8)*. The BFS from the right edge continues until we reach output node *Address(7 downto 0)*. Thus, we have found a transparency path for propagating the input to the output with a latency of six cycles. We delete this path and do a BFS for any remaining input whose transparency path has not yet been found. If no paths are found, the deleted paths are put back in the RCG and the BFS is repeated with the aim of finding a transparency path by reusing edges from prior transparency paths. If transparency paths are still not found, then a transparency path is created by adding extra logic. Any register reachable from the input in one cycle (found from the BFS) is connected to an output (or outputs if bit-widths mismatch) with a test multiplexer. To keep the transparency paths disjoint, preference is given to an output(s) which has not yet been used for providing transparency to other core inputs. If any edge is used in more than one transparency path, then it may not be possible to propagate data through the two paths in parallel. Finally, a solution is obtained where each input can be propagated to some output(s) in a fixed number of cycles (transparency latency) in the test mode.

Next, we try to find transparency paths so that each output of the core may be justified from some input of the core during the transparency mode. For this, we reverse all the edges in the RCG, and again do a BFS from each output node of the RCG. We select an output and continue the BFS until we reach an input node. For example, for the CPU, we first select *Address(7 downto 0)*, and do a BFS using only the HSCAN edges. When we reach a C-split node, the BFS is done from each of the nodes at the fanout edges (fanin in the original RCG) of the C-split node since all the fanout edges need to be used for justifying the data at the C-split node. When two BFSs originating at the fanout nodes of a C-split node converge at an O-split node, then they again constitute a single search. Thus, the search branches into two different searches at the C-split node *ACCUMULATOR* and reconverges to a single search at the O-split node *IR*, eventually reaching the input node *Data*. If the transparency path thus formed has split nodes, and if the parallel subpaths are not balanced at each fanin of a C-split node in the original RCG, for each fanin which does not fall on the longest subpath we add extra logic to freeze the data there. This balances the parallel sub-paths, so that the data, which have arrived early at the fanin of a C-split node, wait until the data at all the fanin edges are ready.

Next, we repeat the procedure for the other output nodes. For the CPU, we delete the transparency path and do BFS for *Address(11 downto 8)*. Since we cannot find a path, we put back the deleted path and try again. This time we succeed. Thus, the transparency paths of the two outputs are not disjoint, but rather reuse common edges (*MAR*, *IR*) and (*IR*, *Data*). When the same edge(s) in the RCG has to be reused in multiple transparency paths, we transfer data through them in a sequential manner.

If transparency paths are not found using HSCAN edges, then

BARCODE PREPRO-CESSOR	LATENCY (cycles) NUM->DB	NUM->A	Ovhd. (cells)
Ver. 1	5	2	2
Ver. 2	1	2	19
Ver. 3	1	1	37

A – Address

(a)

DISPLAY	LATENCY (cycles) D->OUT	A->OUT	Ovhd. (cells)
Ver. 1	2	3	5
Ver. 2	2	1	20
Ver. 3	1	1	55

OUT –> combination of output ports

(b)

Figure 8: Transparency latency-area overhead trade-off in the PREPROCESSOR and DISPLAY cores

other existing paths in the RCG are used to obtain transparency. In the case of the CPU, transparency paths for all inputs and outputs are achievable using HSCAN paths only. This gives rise to *Version 1* of the CPU core (see Figure 6).

The transparency latency of a core can be reduced by adding extra logic and using existing non-HSCAN paths or by adding transparency multiplexers. Consider Figure 7. Let the non-HSCAN edges existing in the RCG be included in the BFS this time. Since an edge exists from *Data* to *Address(7 downto 0)*, a transparency path can be found for output *Address(7 downto 0)* to input *Data* with a latency of just one cycle. A transparency path exists from *Address(11 downto 8)* to input *Data* with a latency of two cycles as before. Consequently, we have *Version 2* of the CPU core. When non-HSCAN edges are used for transparency, the type of transparency logic required is determined by analyzing the multiplexer tree or bus interconnect that created the edge in the RCG and adding the appropriate extra logic according to the data flow required. This is a straightforward mapping, as explained in Section 2.

Further reduction in transparency latency can be obtained by introducing transparency multiplexers. This is done one at a time for each input/output pair with a transparency latency greater than one. Figure 5 shows a transparency multiplexer (shaded) added to reduce the latency of *Data* → *Address(11 downto 8)* to one, leading to *Version 3* of the CPU. The trade-offs for different core versions for the other two cores in Figure 2 are shown in Figure 8.

Note that input control signals to a core can be treated as data inputs and if direct paths from control inputs to control registers are not present, the random logic in between can be simply bypassed with single-bit multiplexers during testing. The control registers can similarly connect to control outputs as shown in the *Read* and *Write* chain in Figure 4. Thus, expensive test generation techniques are avoided during the evaluation of transparency for such signals.

5 Trading Off Area and Test Application Time

We next present a method to select the best core versions to produce a testable SOC for the following objectives:

(i) Given a test area overhead constraint of x cells, the global test application time of the SOC is minimized.

(ii) Given a test application time constraint of y cycles, the overall test area overhead of the SOC is minimized.

We propose an iterative improvement technique, which can obtain solutions for either of the design objectives by varying the parameters of a cost function. At first, we create a core connectivity graph (CCG) from the given SOC. In this graph, there are primary input (PI) nodes, primary output (PO) nodes, core input nodes and core output nodes. Edges are added between input-output pairs of a core according to the transparency paths in the versions of the core currently in use. The cost associated with an edge represents the transparency latency between the corresponding input-output pair. An input node is split into separate nodes if its fanin is split, *i.e.*, if different bit-slices of the node receive data from different sources exclusively. Similar splitting is done for output nodes also by considering the fanout bits. The CCG of the barcode system in Figure 2 is shown in Figure 9 where the minimum area version (*Version 1*) of each core is used. We do not consider the memory cores in this discussion, as most memory cores use BIST [8].

5.1 Identification of test paths for a core

We next discuss how to identify the justification and propagation paths for each core such that the test application time needed for the

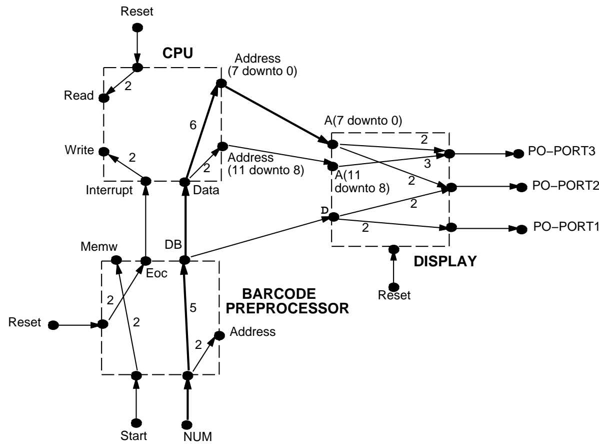

Figure 9: CCG of the barcode system

core is minimized. Let us take the DISPLAY core first. We have to control its inputs *A(7 downto 0)*, *A(11 downto 8)* and *D*. We find out all possible shortest paths from the chip inputs to *A(7 downto 0)* and choose the one with lowest cost (which takes the least number of clock cycles). Any shortest path algorithm like Dijkstra's algorithm can be used for this purpose. Here, only one path is possible (*NUM* → *DB* → *Data* → *Address(7 downto 0)* → *A(7 downto 0)*) which is highlighted in Figure 9. We mark this path and reserve the edges for the cycles in which they will be used. Hence, edge (*NUM*, *DB*) in the PREPROCESSOR is reserved from cycle 0 to 5, edge (*Data*, *Address(7 downto 0)*) is reserved from cycle 5 to 11, and so on. If there is no path possible, we add a system-level test multiplexer to connect the input of the core directly to a PI. Next, we try to find the shortest path from any chip PI to *A(11 downto 8)*. The shortest path algorithm is modified so that if any marked edge is reused, then the cost is automatically modified so that the edge is not reused in the reserved cycles. Here, part of the previous path needs to be reused. Therefore, the edge (*NUM*, *DB*) can only be utilized from cycle 6 onwards, the edge (*Data*, *Address(7 downto 0)*) can only be used from cycle 12 onwards, and so on. The minimum-cost path is again chosen among all possible paths and test multiplexers are added if necessary. Similarly, the path for controlling input *D* can be found.

Next, we try to find the paths for observing the core outputs at the POs of the chip. This process is very similar to the method of finding paths to inputs which is described above. For the DISPLAY, all paths are trivially found as they connect to POs of the chip. Again, test multiplexers are added to increase observability if necessary. This process is repeated for every core in the system. In Figure 9, the output *Address* of the PREPROCESSOR is connected to a PO with a system-level test multiplexer since there is no way of observing it by existing paths through the cores.

Once a test solution is found, the global test application time and area overheads for the system are calculated. If it does not meet the constraints that the user has set, then a different design point which provides the most improvement needs to be obtained.

5.2 Iterative improvement for core selection

Next, we outline the procedure to select the most suitable core to be replaced by the next higher overhead version. First, all core versions are ordered in terms of the area overhead from lowest upwards. Then a test application time improvement number is calculated for each core. To do this, the test solution is examined and the number of times each edge in a core is used is counted. Then this number is multiplied by the latency of the edge and is summed over all edges of the core. For example, the edge (*NUM*, *DB*) in the PREPROCESSOR is used twice to test the DISPLAY and once to test the CPU. Its latency is five. The edge (*NUM*, *Address*) is never used. The edge (*Reset*, *Eoc*) in the PREPROCESSOR is used once to test the CPU. Its latency is two. Hence, the initial latency number for the PREPROCESSOR is $3 \times 5 + 0 \times 2 + 1 \times 2 = 17$. Now if we replace the PREPROCESSOR with the next area-expensive version, the latency number changes to $3 \times 1 + 0 \times 2 + 1 \times 2 = 5$. Thus, the latency improvement, ΔTAT, in the PREPROCESSOR is $17 - 5 = 12$. ΔTAT gives a notion of the improvement in the global test application time if a core is replaced with another version. The area overhead increase, ΔA, for this replacement is 17 cells. For each core, the cost of replacing it with another version is calculated as $C = w_1 \times \Delta TAT + w_2 \times \Delta A$, where w_1 and w_2 are weights that can be adjusted according to the problem solved.

Next, suppose we are trying to meet objective (i) as specified before. We set w_1 to 1 and w_2 to 0 and improve the existing solution by replacing the core that has the highest C value with its next expensive version. We repeat this until the test overhead reaches the preset maximum or the test overhead becomes higher than the overhead of a system-level test multiplexer. In the latter case, we choose the core which contributes the most to the global test application time, examine the time it takes to get an input vector to each of its inputs from the PIs (observe an output vector at each of its outputs at the POs) and place the system-level multiplexer on the input or output that is the most critical. This process is stopped once the test overhead reaches the user-defined maximum allowable test overhead.

If we are trying to meet objective (ii) stated before, we set w_1 to 0 and w_2 to 1 and choose the core with the minimum C that has a non-zero ΔTAT value. This means that we replace the core which is the least expensive to replace but ensure that it will have some effect in reducing the global test application time. The global test application time is again calculated and the process repeated until the preset test application time limit is met. Once again, if the test overhead becomes higher than the overhead of a system-level test multiplexer, we place an extra test multiplexer as before. Thus, in the worst case, the solution will degenerate into a test bus like system in which core inputs are controllable from the PIs and core outputs observable at the POs. Note that this system will have the minimum possible test application time.

The proposed methodology requires that each core can be clocked independently of the other cores as sometimes test data needs to be frozen at certain points while it keeps on flowing at other points. This is done by freezing the clock of the core when necessary with the help of clock gating circuitry. During the test mode, these control signals as well as the transparency mode control signals like *T2* and *T3* in Figure 6 are provided by a test controller which is added to the chip. This usually consists of a small finite-state machine.

6 Experimental Results

We applied the proposed SOC testing methodology to two example core-based systems. The first example, System 1, is the barcode embedded system shown in Figure 2. System 2 consists of a graphics processor core [9], a GCD core [10], and a X25 protocol core [11]. Each core used in the systems has HSCAN DFT, and hence can be treated as a full-scan circuit and tested using combinational ATPG tools. The area numbers represent number of cells, and are obtained after synthesizing and technology mapping different versions of the cores and systems with an in-house synthesis tool. The fault coverage numbers are obtained by fault simulating the logic-level implementation of the system with the system-level test set using a commercial combinational ATPG tool.

The graph shown in Figure 10 demonstrates the chip-level trade-off between the area overhead and the chip test application time for 18 design points of System 1 which are obtained by different combinations of the core versions. Design point 1 (18) corresponds to the SOC consisting of a minimum area (minimum latency) version of each core. Details of some of the design points (1, 17, 18) are given in Table 1, with Columns 2, 3, 4, and 5 showing the area overhead, test application time, fault coverage, and test efficiency, respectively. The first two rows, representing design points 1 and 18, show that about 4.5 times reduction in test application time can be achieved with about two times increase in the test area overhead. The table also shows that selecting

Table 1: Design space exploration for System 1

Circuit description	*A. Ov.* (cells)	*TApp.* (cycles)	*FCov.* (%)	*TEff.* (%)
Each core has min. Area (1)	156	17,387	98.4	99.8
Each core has min. latency (18)	325	3,818	98.4	99.8
Min. chip TApp. (17)	307	3,806	98.4	99.8

Table 2: Area overheads

Circuit	*Orig.* Area (cells)	*Core-level DFT* *FSCAN* *Ovhd.* (%)	*HSCAN* *Ovhd.* (%)	*Chip-level DFT* *BSCAN* *Ovhd.* (%)	*SOCET* *Circuit type*	*Ovhd.* (%)	*Core + Chip-level DFT* *FSCAN-BSCAN* *Ovhd.* (%)	*SOCET* *Ovhd.* (%)
System 1	8014	18.8	10.1	5.2	Min. Area	2.0	24.0	12.1
					Min. TApp.	3.8	24.0	13.9
System 2	5540	15.6	10.3	9.9	Min. Area	1.2	25.5	11.5
					Min. TApp.	4.7	25.5	15.0

Table 3: Testability results

Circuit	*Orig.* *FC* (%)	*TEff.* (%)	*HSCAN* *FC* (%)	*TEff.* (%)	*FSCAN-BSCAN* *FC* (%)	*TEff.* (%)	*TApp.* (cycles)	*SOCET* *Circuit type*	*FC* (%)	*TEff.* (%)	*TApp.* (cycles)
System 1	10.6	10.8	14.6	14.9	98.4	99.8	36,152	Min. Area	98.4	99.8	17,387
								Min. TApp.	98.4	99.8	3,806
System 2	11.2	11.3	13.8	13.8	98.2	99.9	46,394	Min. Area	98.2	99.9	16,435
								Min. TApp.	98.2	99.9	3,998

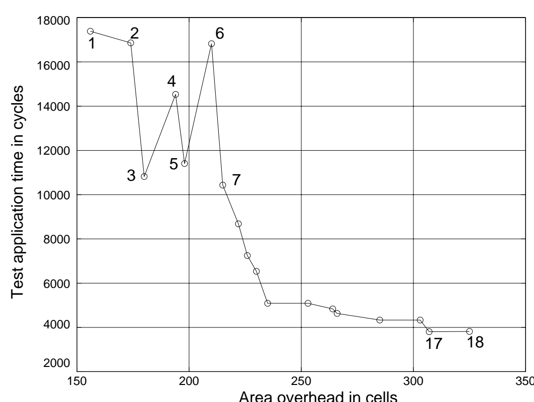

Figure 10: Test application time *vs* area ovhd. for System 1

the minimum latency cores does not necessarily lead to an SOC with minimum test application time. Design point 17 represents the SOC having minimum test application time though it uses a higher latency version of the PREPROCESSOR core than the one used in design point 18.

Table 2 compares the area overheads obtained by our method with the existing core-based SOC testing method FSCAN-BSCAN. We use two extreme design points obtained by our method, termed SOCET, for the comparison - the chip having least area overhead, and the chip having least test application time. In Column 2 of the table, the original area of the chip is given. Columns 3 and 4 give the area overhead needed to make the cores testable using FSCAN and HSCAN, respectively. The next two columns give the area overhead for the chip-level DFT needed to make the SOC testable: Column 5 shows the boundary scan area overhead, while Columns 6 and 7 list the overheads for the two extreme design points using SOCET. The total area overheads including the core-level and chip-level DFT for the two different methods are given in Columns 8 and 9. The area overhead required for SOCET is significantly lower than the area overhead required for FSCAN-BSCAN.

The testability results are reported in Table 3. Column 2 gives the fault coverage of the chip before the addition of any test hardware. This is obtained by running an in-house sequential test generation tool on the original circuit. As expected, the fault coverage is very poor. The test efficiency is given in Column 3. In Columns 4 and 5, the corresponding results are given for the chip when individual cores are made testable using HSCAN, but no chip-level DFT is added. This case shows that the overall fault coverage of the chip may be quite poor even if individual cores are testable. The fault coverage, test efficiency, and test application time for the chip made testable by FSCAN-BSCAN are given in Columns 6, 7, and 8, respectively. The corresponding results for SOCET are given in Columns 10, 11, and 12. While the fault coverage obtained by FSCAN-BSCAN and SOCET are equally high, the test application time required by SOCET is significantly lower than FSCAN-BSCAN.

7 Conclusions

In this paper, we have presented a novel technique for efficient testing of core-based SOCs. Individual cores are first made testable and then transparent using the existing logic inside the core and adding test hardware only if needed. Since the transparency latency of cores is a critical factor in the overall test application time of the core-based system, it is shown how different versions of the same core can be created with different transparency latencies and test overhead values. In the system, each core is tested with its precomputed test set which is propagated through other cores using their transparency modes. A technique is presented here that helps the user to explore the design space and choose a particular version of each core so that the final test overhead and chip test application time goals can be met. Application of the technique to example SOCs demonstrates the feasibility of our approach as well as significant reduction in test overhead and test application times compared to existing SOC test techniques.

Acknowledgments: We would like to thank A. Raghunathan for his help with the example embedded systems and NEC CCRL for supporting I. Ghosh with a summer internship.

References

[1] Y. Zorian, “Test requirements for embedded core-based systems and IEEE P1500,” in *Proc. Int. Test Conf.*, Nov. 1997.

[2] R. Chandramouli and S. Pateras, “Testing systems on a chip,” *IEEE Spectrum*, pp. 42-47, Nov. 1996.

[3] N.A. Touba and B. Pouya, “Testing embedded cores using partial isolation rings,” in *Proc. VLSI Test Symp.* pp. 10-15, Apr. 1997.

[4] F. Bouwman *et al.*, “Macro testability; The results of production device applications,” in *Proc. Int. Test Conf.*, pp. 232-241, Nov. 1992.

[5] I. Ghosh, N.K. Jha, and S. Dey, “A low overhead design for testability and test generation technique for core-based systems,” in *Proc. Int. Test Conf.*, pp. 50-59, Nov. 1997.

[6] S. Bhattacharya and S. Dey “H-SCAN: A high level alternative to full-scan testing with reduced area and test application overheads,” in *Proc. VLSI Test Symp.*, pp. 74-80, Apr. 1996.

[7] Z. Navabi, *VHDL Analysis and Modeling of Digital Systems*, McGraw-Hill, New York, 1993.

[8] Y. Zorian, “A distributed BIST control scheme for complex VLSI devices,” in *Proc. VLSI Test Symp.*, pp. 6-11, Apr. 1993.

[9] A. Raghunathan, S. Dey, and N.K. Jha, “Power management techniques for control-flow intensive designs,” in *Proc. Design Automation Conf.*, pp. 429-434, June 1997.

[10] P.R. Panda and N.D. Dutt, “1995 high-level synthesis design repository,” in *Proc. Int. Symp. System Level Synthesis*, pp. 170-174, Sept. 1995.

[11] S. Bhattacharya, S. Dey, and F. Brglez, “Performance analysis and optimization of schedules for conditional and loop-intensive specifications,” in *Proc. Design Automation Conf.*, pp. 491-496, June 1994.