Using an Embedded Processor for Efficient Deterministic Testing of Systems-on-a-Chip

Abhijit Jas and Nur A. Touba

Computer Engineering Research Center
Department of Electrical and Computer Engineering
University of Texas, Austin, TX 78712-1084
E-mail: {jas, touba}@cat.ece.utexas.edu

Abstract

If a system-on-a-chip (SOC) contains an embedded processor, this paper presents a novel approach for using the processor to aid in testing the other components of the SOC. The basic idea is that the tester loads a program along with compressed test data into the processor's on-chip memory. The processor executes the program which decompresses the test data and applies it to scan chains in the other components of the SOC to test them. This approach both reduces the amount of data that must be stored on the tester and reduces the test time. Moreover, it enables at-speed scan shifting even with a slow tester (i.e., a tester whose maximum clock rate is slower than the SOC's normal operating clock rate). A procedure is described for converting a set of test cubes (i.e., test vectors where unspecified inputs are left as X's) into a compressed form. A program that can be run on an embedded processor is given for decompressing the test cubes and applying them to scan chains on the chip. Experimental results indicate significant amount of compression can be achieved.

1. Introduction

An increasingly difficult challenge in testing systems-on-a-chip (SOC) is dealing with the large amount of test data that must be transferred between the tester and the chip [Zorian 97]. The entire set of test vectors for all components of the SOC must be stored on the tester and transferred to the chip during testing (as illustrated in Fig. 1). This poses a serious problem because of the cost and limitations of ATE (automated test equipment). Testers have limited speed, channel capacity, and memory. The amount of time required to test a chip depends on how much test data needs to be transferred to the chip and how fast the data can be transferred (i.e., the test data bandwidth). This depends on the speed and channel capacity of the tester and the organization of the scan chains on the chip. Both test time and test storage are major concerns for SOCs.

Usually, an SOC will contain one or more embedded processors. This paper presents a novel approach for using an embedded processor to aid in testing the other components of the SOC. The basic idea is that the tester loads a program along with compressed test data into the processor's on-chip memory (e.g., cache or other scratch pad memory). The processor executes the program which decompresses the test data and applies it to scan chains in the other components of the SOC to test them. This approach both reduces the amount of data that must be stored on the tester *and* reduces the test time. Moreover, it can enable at-speed shifting using a slow tester (i.e., a tester whose maximum clock rate is slower than the SOC's normal operating clock rate).

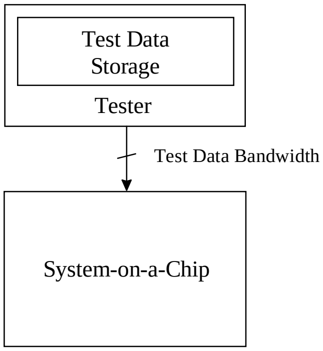

Figure 1. Block Diagram for Transferring Test Data between Tester and Chip

Previous research in using embedded processors to aid in testing has focused on performing memory tests [Saxena 98], [Rajsuman 99], or pseudo-random built-in self-test (BIST). Techniques for generating pseudo-random patterns and compacting test responses using simple programs have been proposed in [Rajski 93], [Gupta 94], and [Stroele 95, 96, 98]. Techniques for mixed-mode BIST using embedded processors have been described in [Hellebrand 96] and [Dorsch 98]. The approach presented here is a fully deterministic test approach which supports either external testing or BIST.

The advantage of decompressing deterministic vectors is that a targeted fault coverage can be achieved with a short test time. The number of vectors that need to be applied to the circuit is much less than that required for pseudo-random BIST. Moreover, it supports structured delay fault testing and testing of intellectual property (IP) blocks. In some cases, an IP provider may not be willing to provide any information of the internal logic of an IP block and thus fault simulation is not possible thereby precluding pseudo-random BIST.

Previous research has also been done in compressing/decompressing test data. Novel approaches for compressing test data using the Burrows-Wheeler transform and run-length coding were presented in [Yamaguchi 97], [Inshida 98]. These schemes were developed for reducing the time to transfer test data from a workstation to a tester (not for use on chips). It is too complex and slow for an on-chip implementation. A scheme for compression/decompression of test data using cyclical scan circuits is described in [Jas 98]. It uses careful ordering of the test set and formation of cyclical scan chains to achieve compression with run-length codes. Compression schemes based on statistical codes were presented in [Iyengar 98] for non-scan circuits and [Jas 99] for scan circuits.

The compression/decompression scheme presented in this paper is very well suited for implementation on an embedded processor. The decompression process requires very few processor instructions and thus can be done very quickly. The decompression process is pipelined to maximize the throughput of decompressed test vectors and thereby minimize the test time.

2. Proposed Scheme

The compression/decompression scheme described in this paper is based on generating the next test vector from the previous one by storing only the information about how the vectors differ. In this scheme, each test vector is divided into fixed length blocks as shown in Fig. 2.

The size of the test vector blocks depends on the word size of the processor. The way this block size is determined will be explained later. The next test vector is built from the previous test vector by replacing the blocks in which they differ. For example, in Fig. 2, the blocks in which test vector *t+1* differs from test vector *t* are shaded. Hence test vector *t+1* can be built from test vector *t* by replacing only the shaded blocks.

Block #:	000	001	010	011	100	101	110	111
Vector (t):								
Vector (t+1):		▨	▨			▨		▨

Figure 2. Dividing Test Vector into Blocks

Last Block	Block #	New Block Pattern
1	m=$\lceil \log_2 N \rceil$	b=block size

Figure 3. Replacement Words

Last Flag	Block #	New Block Pattern
0	001	1011010011010011010001101010
0	010	1011010011010011010001101010
0	101	1011010011010011010001101010
1	111	1011010011010011010001101010

Word Size of Processor (e.g., 32 bits)

Figure 4. Replacement Words for Example in Fig. 2

Because of the structural relationship among faults in a circuit, there will be a lot of similarity between the test vectors. The test vectors can be ordered in an optimal way such that two successive test vectors differ in a relatively fewer number of blocks. Hence the amount of information required to store these differences will be less than that required for storing the entire test vector. These differences are represented by "replacement words" which are encoded pieces of information that tell the processor how to build the next test vector from the previous one. Each replacement word has three fields as shown in Fig. 3. A single bit field called the *last flag*, a log_2N bit field called the *block number* (where N is the number of blocks into which the test vector is divided) and a b bit field called the *new block pattern*. The block number field contains the address of the block that is to be replaced with the new pattern contained in the new block pattern field. If two successive test vectors differ in x blocks then this information is represented as a sequence of x replacement words where the last flag field of the x-th replacement word (the last replacement word in the sequence) has its last flag bit set (1). All other replacement words have their last flags turned off (0). The sequence of replacement words for the example of Fig. 2 is shown in Fig. 4. The processor reads these replacement words and then replaces the appropriate blocks with the new block patterns. When it sees the last flag bit set, then it knows that the next test vector formation has been completed. It then shifts the test vector into the scan chain(s) and applies it to the core-under-test. The block size, b, is chosen in such a way that $1 + \lceil log_2N \rceil + b = W$, where W is the word size of the processor.

As mentioned before, if successive test vectors differ in a small number of blocks, then the total number of bits required for representing all the replacement words (to build the next test vector) will be less than that required for representing the entire test vector (which is the same as the number of bits in a test vector). We thus obtain compression. Note that the amount of compression depends on the ordering of the test vectors. The better the ordering, the fewer the number of blocks in which successive test vectors differ and consequently the fewer the number of replacement words.

The optimal ordering of the test vectors to maximize the compression can be obtained as follows. Form a complete undirected graph where each vertex v corresponds to a test vector t_v and the weight of each edge (x,y) is the number of bits required to encode the information of how to produce t_x from t_y or vice versa. The minimum cost Hamitonian path in the graph corresponds to the optimal ordering of the test vectors. Finding the minimum cost Hamitonian path in a graph is a well known NP-complete problem [Cormen 89]. However, a polynomial time approximation algorithm producing a near optimal solution exists when the triangle inequality holds. In our application, the triangle inequality holds. Let $w(x,y)$ denote the weight of an edge between vertices x and y. The triangle inequality implies that for any three vertices x, y, and z in the graph, $w(x,y) + w(y,z) \geq w(x,z)$. The reason why this is true is as follows. Let x and y differ in blocks $b_1, b_2, \ldots, b_n$ and y and z differ in blocks $b_{1'}, b_{2'}, \ldots, b_{m'}$. Then x and z will differ in $m+n$ blocks if $\forall i\ (b_i \neq b_{i'})$ or they will differ in less than $m+n$ blocks if some b_i and $b_{i'}$ are the same.

3. Performing Decompression Using an Embedded Processor

The compression/decompression technique described in the previous section can be implemented by using an embedded processor in an SOC. This section describes the implementation details.

The block diagram in Fig. 5 gives an overview of the architectural set-up of the scheme. The processor is used to concurrently load multiple scan chains on the chip. The on-chip memory holds the instructions which the processor executes and also the data on which the processor operates. The tester initially downloads a software program into the memory which the processor executes. The tester then supplies the processor with a stream of encoded data (replacement words) about blocks that are to be replaced with new patterns to produce the next test vector. The processor just decodes the replacement words and acts accordingly.

In the initialization phase (before testing begins) the tester downloads the software program (compiled machine code) into the on-chip memory. Then a fixed set of locations are reserved in the memory for the tester to continuously download the encoded data (replacement words) which the program uses. For example, let us assume that memory locations $0, 1, \ldots, M\text{-}1$ are M memory locations which are reserved for the tester to write encoded data into in a modulo M manner, i.e., the tester first writes encoded data into locations $0, 1, \ldots, M\text{-}1$ and then again starts from 0. After the initialization phase is over, the tester starts loading replacement words into the specified memory locations. The processor now starts running the program. The program is very simple. It directs the processor to routinely read the replacement words from the memory locations $0, 1, \ldots, M\text{-}1$ in a modulo M manner and act accordingly. While the processor is running the program, which results in test vectors being generated and applied to the scan chains, the tester continues to load the memory with new replacement words for the processor to work on.

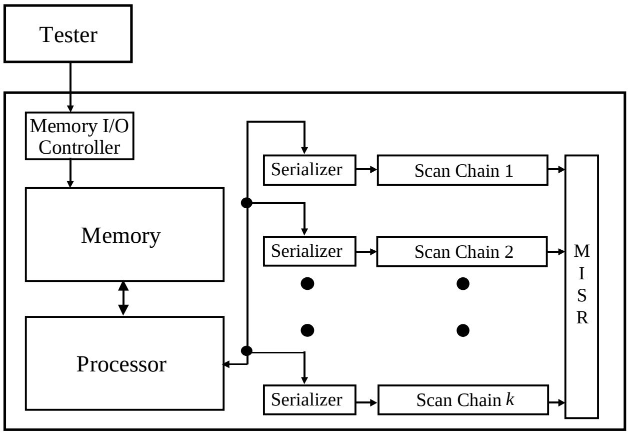

Figure 5. Block Diagram of Test Architecture

Note that, in general, there will be a memory I/O controller on the SOC for interfacing with the outside world during normal system operation. This memory I/O controller is used by the tester to load the data into the memory during testing. The memory I/O controller is typically capable of handling a slower clock rate when interfacing with the outside world. Thus, the tester can run at a slower clock rate than the normal system clock rate of the processor. Thus the processor can be operating at-speed even though the tester may be slower. Because of the tremendous cost of high-speed ATE equipment, this is a major advantage because it allows at-speed scan shifting while using slower (and cheaper) ATE equipment.

The current set of blocks is stored in the on-chip memory along with the replacement words. The program that runs on the processor causes the processor to execute a while loop until the end of the test data. In each iteration of the loop, it fetches a replacement word from the next memory location from which it is supposed to read the data and replaces the appropriate block. When the processor sees that the last flag bit is set, it knows that it has seen the last replacement word for that particular test vector and now needs to apply the test vector to the scan chain. The mechanism for applying a test vector to a scan chain is controlled by a *serializer*. The processor downloads each block to the appropriate serializer one at a time and the serializer shifts the block into the scan chain. The serializer is a register with a small finite state controller which shifts in one bit per clock cycle into the scan chain. When the entire test vector has been shifted into the scan chain, the system clock is applied and the response is loaded back into the scan chain. The response is shifted out into a multi-input signature register (MISR) for compaction as the next test vector is shifted into the scan chain.

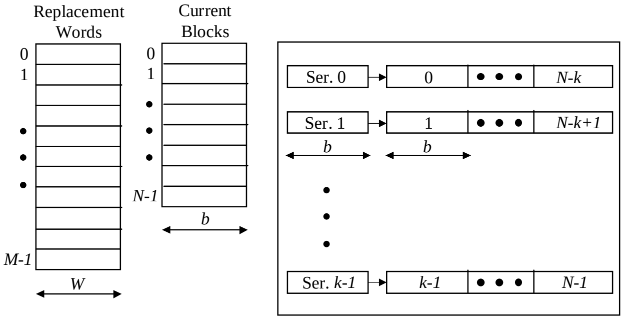

Figure 6. Organization of Data in On-Chip Memory

As shown in Fig. 6, a test vector is stored as *N* words in the on-chip memory (where *N* is the number of blocks into which the test vectors have been divided). Note that there are two distinct areas (sequence of addresses) in the memory from which the processor reads/writes data. There is one area from which the processor only reads the data. This is the area into which the tester writes the replacement words and the processor reads it to update the blocks. The other area is where the patterns for the blocks of the test vector are stored. The processor both reads and writes data from and to this area. It writes data into this area when it is replacing a block with a new pattern, and it reads data from this area when it downloads them to the serializers for shifting them into the scan chains. When the processor sees the *last flag* bit set, it begins the process of applying the vector by downloading the 0th block into the serializer for the 0th scan chain and starts the serializer. The serializer then starts shifting the block into the scan chain. While it is doing so the processor continues to download the 1st block into the serializer for the 1st scan chain and so on and so forth. Thus if there are *k* scan chains and *N* blocks ($k < N$) then the *i*-th block gets downloaded into the (i mod k)-th serializer. When the processor has finished downloading the block into the *k*-th serializer, it comes back to the 0th serializer. Now there can be two situations. If the 0th serializer has finished shifting in the earlier block the processor can immediately download the next block and start the serializer again. Otherwise it has to wait for the serializer to finish shifting before it can download another block into it. Once the last block has been downloaded and shifted in, the test vector is applied to the scan chains and the processor again continues to read replacement words from the on-chip memory and update the blocks.

In Fig. 7, '*C*' like pseudo-code for the decompression program that runs on the processor is given. It is a high level view of how the assembly code for the program running on the processor may look like. Note that it is just an abstraction and tries to convey the basic algorithm for the decoding. The macro calls in most cases will be just a few processor instructions. For example, the *write_memory* is actually a representative of a STORE instruction with indirect register addressing (base and offset register). The variable *mem_index* is an offset that points to the address from which the processor should get the next data to decode. MEM_START is

```
void test (void)
{
 int i;  // looping variable
 int k; // number of scan chains
 int N; // total number of blocks in a test vector
 unsigned int mem_index = MEM_START;
 int last_block,block_to_replace,pattern;

// continue until read sees "end_of_test" instruction
while(read_memory (MEM_START+mem_index,&last_block,
                            &block_to_update,&pattern))
 {
  // replace appropriate block with new block pattern
  write_memory (base_address+block_to_replace,pattern);
  // load test vector into scan chains
  if(last_block)
  {
    for(i=0;i < N; i++)
    {
      while(serializer_busy (i%k));  // wait until serializer is ready
      load_serializer_from_memory (&serializer[i%k],base_address+i);
      start_serializer (i%k);
    }
  }
  // point to next memory location for reading replacement word
  mem_index = (mem_index+1)% MEM_SIZE;
 }
}
```

Figure 7. Pseudo-Code for Program Run on Processor

defined to be the starting location for reading the replacement words. It is assumed that the very last instruction of the test session causes the processor to set some kind of a flag which is abstracted by the condition of the *while* loop. MEM_SIZE is the total size of the address space reserved for the tester to write the replacement words i.e., the tester writes to locations MEM_START, (MEM_START+1),..., (MEM_START+ MEM_SIZE-1). The *read_memory* macro implements reading a replacement block from a memory location and obtains the information about the address of the block to replace, pattern with which to replace, and whether it is the last block for that test vector. The *load_serializer_from_ memory* macro downloads a block from the memory location addressed by the block number as offset from some base address, into the serializer for that scan chain and starts the serializer (by setting an indicator line abstracted by the *start_serializer* routine) to shift that into the scan chain.

4. Avoiding Memory Overflow

Since the tester is constantly transferring replacement words to the on-chip memory, one potential problem is that if the processor falls too far behind in processing the replacement words, there could be a "memory overflow." This would result in the tester overwriting a replacement word in the memory that has not yet been processed. Care must be taken to ensure that no memory overflow will occur.

The possibility for a memory overflow depends on the relative speed of the processor to the tester. The rate at which the tester puts data into the memory depends on the tester clock rate and the number of channels. The rate at which the processor processes data in the memory depends on its clock rate, word size, and instruction set. The instruction set determines how many clock cycles are required to process the replacement words.

The tester continuously writes test data in a cyclic fashion into locations $0,1,\ldots,M-1$ of the memory. So after every MW (where W is the word size) bits each location of the memory gets overwritten by the tester. Hence the processor should finish processing all the replacement words in locations $0,1,\ldots,M-1$ within the time taken by the tester to shift in MW bits. If the tester has n scan channels and has a clock period T_T, the time taken by the tester to shift in MW bits is $[WM/n]T_T$. Hence the processor should process all M replacement words within this time. All the processor does for each replacement word is to read the new pattern and the block address and replace the appropriate block. However for the blocks which have the last flag set, the processor has to download all the blocks into the serializers to shift them into the scan chains. This is the most time consuming part for the processor. So the speed at which the processor can process M replacement words depends on how many of them have their last flag set. Let this be denoted by e. Then the time taken by the processor to process M replacement words is $[Mu+(eNb/k)]\ T_P$ where T_P is the clock period of the processor, N is the total number of blocks, b is the block size, k is the number of scan chains, and u is the number of cycles taken by the processor to read a replacement word and replace the appropriate block. The value of u depends on the instruction set architecture of the processor. No memory overflow will occur if the following condition is satisfied: $[Mu+(eNb/k)]T_P < [WM/n]T_T$

If the condition above is not satisfied, then for a given compressed test set, a quick check can be made to see if a memory overflow will occur. If a memory overflow would occur, then something must be done to avoid this. One solution would be to re-order the test vectors when constructing the compressed test set so that e will become smaller. This will likely result in less test data compression, but it will avoid memory overflow. Another solution may be to insert NOP's in the tester program at carefully selected locations to slow it down so no memory overflow will occur, or to simply run the tester at a slower clock rate that ensures no memory overflow. These solutions would not reduce the amount of test data compression and thus would still minimize tester memory requirements.

Note that if the processor is sufficiently faster than the tester, then the above condition will be satisfied and the maximal reduction of both test data and test time can be achieved. Note also that as the number of scan chains (i.e., k) is increased, the condition becomes easier to satisfy.

5. Experimental Results

We used our scheme to compress test sets for the largest ISCAS benchmark circuits. An ATPG tool was used to generate test cubes that provided 100% coverage of detectable faults in each circuit. Unspecified input assignments were left as X's to enable better compression. Static compaction of the test cubes was performed by merging the test cubes when possible and doing reverse fault simulation to remove superfluous test cubes. The block size in each case was derived by the formula: $1 + \lceil log_2 N \rceil + \ b = W$. Where N is the number of blocks in the scan chain, b is the block size, and W is the width of the addressable element of the memory. N is the number of blocks, so we need $\lceil log_2 N \rceil$ bits to address each block. One bit is needed as the *last block* indicator and the update pattern for each block is *b* bits wide. So the above inequality results from the fact that the memory addressable element (word) should be big enough to hold this data. The results obtained in the table below are for $W = 32$. The percentage of compression is computed as:

(Original Bits - Compressed Bits)/(Original Bits) x 100

As can be seen from the results, a significant amount of compression can be achieve with the proposed scheme. Note that this compression results in both less storage requirements on the tester and less test time. As explained in Sec. 4, the actual test time reduction could be somewhat less depending on the number of scan chains and the relative speed of the processor compared to the tester. However, with a sufficiently fast processor, the percentage compression figure shown in Table 1 would also be the percentage reduction in test time.

Table 1. Compression Obtained for Benchmark Circuits Using the Proposed Scheme

Circuit	Scan Size	Original Test Data (Bits)	Block Size (Bits)	Percent Comp.
c2670	233	35183	27	58.45
c5315	178	24742	28	41.52
c7552	207	62721	28	42.39
s5378	199	29850	28	39.00
s9234	247	48906	27	26.60
s13207	700	186200	26	73.32
s15850	611	86151	26	46.65
s38417	1664	247936	24	59.06

6. Conclusion

The proposed approach makes use of existing functional circuitry on the chip (i.e., an embedded processor) to aid in testing the SOC. By harnessing the computational power of an embedded processor, decompression of test data can be performed in software. This reduces both the amount of test storage and test time, thereby reducing the tester memory and channel capacity requirements. Such techniques are needed to keep down the cost of the ATE equipment for testing future SOC's.

A basic framework for how an embedded processor can be used for decompressing test data is presented here. One specific compression/decompression algorithm which gives good results is described. However, there is a lot of scope for future research in other compression/ decompression algorithms for test data. The ability to use a processor to perform the decompression in software opens the door to many possible techniques.

Acknowledgements

This material is based on work supported in part by the National Science Foundation under Grant No. MIP-9702236, and in part by the Texas Advanced Research Program under Grant No. 1997-003658-369.

References

[Cormen 89] Cormen, T.H., C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1989.

[Dorsch 98] Dorsch, R., and H.-J. Wunderlich, "Accumulator Based Deterministic BIST", *Proc. of International Test Conference*, pp. 412-421, 1998.

[Gupta 94] Gupta, S., J. Rajski, and J. Tyszer, "Test Pattern Generation Based on Arithmetic Operations," *Proc. of Int. Conf. on Comp.-Aided Design (ICCAD)*, pp. 117-124, 1994.

[Jas 98] Jas, A., and N.A. Touba, "Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs," *Proc. of Int. Test Conf.*, pp. 458-464, 1998.

[Jas 99] Jas, A., J. Ghosh-Dastidar, and N.A. Touba, "Scan Vector Compression/Decompression Using Statistical Coding," *Proc. of Int. Test Conference*, pp. 458-464, 1998.

[Hellebrand 96] Hellebrand, S., H.-J. Wunderlich, and A. Hertwig, "Mixed-Mode BIST Using Embedded Processors," *Proc. of Int. Test Conf.*, pp. 195-204, 1996.

[Ishida 98] Ishida, M., D.S. Ha, T. Yamaguchi, "COMPACT: A Hybrid Method for Compressing Test Data," *Proc. of VLSI Test Symposium*, pp. 62-69, 1998.

[Iyengar 98] Iyengar, V., K. Chakraborty, and B. T. Murray, "Built-in Self Testing of Sequential Circuits Using Precomputed Test Sets," *Proc. of VLSI Test Symposium*, pp. 418-423, 1998.

[Rajski 93] Rajski, J., and J. Tyszer, "Accumulator-Based Compaction of Test Responses," *IEEE Transactions on Computers*, Vol. 42, No. 6, pp. 643-650, Jun. 1993.

[Rajsuman 98] Rajsuman, "Testing a System on a Chip with Embedded Microprocessor," *Proc. of Int. Test Conf.*, 1999.

[Saxena 98] Saxena, J., P. Ploicke, K. Cyr, A. Benavides, and M. Malpass, "Test Strategy for TI's TMS320AV7100 Device," *IEEE Int. Workshop on Testing Embedded Core Based Systems*, 1998.

[Stroele 95] Stroele, A.P., "A Self-Test Approach Using Accumulators as Test Pattern Generators," *Proc. of Int. Symposium on Circuits and Systems*, pp. 2010-2013, 1995.

[Stroele 96] Stroele, A.P., "Test Response Compaction Using Arithmetic Functions," *Proc. of VLSI Test Symposium*, pp. 380-386, 1996.

[Stroele 98] Stroele, A.P., "Bit Serial Pattern Generation and Response Compaction Using Arithmetic Functions," *Proc. of VLSI Test Symposium*, pp. 78-84, 1998.

[Yamaguchi 97] Yamaguchi, T., M. Tilgner, M. Ishida, and D. S. Ha, "An Efficient Method for Compressing Test Data," *Proc. of International Test Conference*, pp. 191-199, 1996.

[Zorian 97] Zorian, Y., "Test Requirements for Embedded Core-based Systems and IEEE P1500," *Proc. of International Test Conference*, pp. 191-199, 1996.