Test Access Mechanism for Multiple Identical Cores

Grady Giles, Jing Wang, Anuja Sehgal, Kedarnath J. Balakrishnan, and James Wingfield
Advanced Micro Devices Austin, USA and Sunnyvale, USA

Abstract

A new test access mechanism (TAM) for multiple identical embedded cores is proposed. It exploits the identical nature of the cores and modular pipelined circuitry to provide scalable and flexible capabilities to make trade-offs between test time and diagnosis over the manufacturing maturity cycle from low-yield initial production to high-yield, high-volume production. The test throughput gains of various configurations of this TAM are analyzed. Forward and reverse protocol translations for core patterns applied with this TAM are described.

1. Introduction

Modern microprocessor designs have integrated multiple identical CPU core modules [1], [2], [3], [4]. In such designs, the multiple instances of the cores often comprise a large fraction of the total SoC integration. In addition to the benefits of multiprocessing, such microprocessor architectures can afford attractive economical options of redundant cores for manufacturing yield enhancement; that is, the product specification may be for six cores, whereas the SoC actually has eight cores. With a general embedded core test methodology such as IEEE Std. 1500 [5], the cores can be wrapped so that they have no interaction with outside data sources, and there exists the opportunity to generate the test for a single core and apply that test to each of the instances of the core.

Modular SoC testing strategies have been presented in a variety of previous work. Zorian et. al. [6] introduced a conceptual architecture for modular testing of SoCs, which includes three structural elements: test pattern source and sink, test access mechanism (TAM), and core test wrapper. Goel et. al. [7] discussed TAM and wrapper architecture design and optimization. But a significant improvement in efficiency is accomplished if the test can be applied simultaneously to all of the identical cores. It is conceptually simple to fanout the shared scan input data to the multiple cores but, given the limited test access, gathering the unique scan output data of the many cores is more complicated.

The Cell processor [2], the T1 processor [3], and the Niagara2 processor [4] use dedicated scan inputs and outputs for each unique core. This approach enables quick determination of defective cores but leads to higher test time since the number of scan flops in each chain will be higher. The UltraSPARC processor [1], which has two cores, includes a scan lockstep mode in which the scan chains in both the cores receive the same scan input data. The responses scanned out of both the cores are compared internally and a mismatch between any two bits is reported by a fail pin. In the AZSCAN architecture [8], multiple identical processor cores are tested in parallel by broadcasting the scan data inputs to all of them. The responses of each of the cores are then compared on-chip with the expected data, which is also loaded from the tester, identifying bad cores that are then either repaired or turned off.

Our general strategy is to take advantage of the fact that all of the identical cores should respond identically to a pattern when they are isolated from interaction with outside data sources. This allows us to compare the cores' output responses to either the expected response or to each other's responses using on-chip comparators. Thus, this TAM with its on-chip comparators, besides being a vehicle to convey test data to and from the embedded cores, is a sort of compression scheme as well, since it compacts the pass/fail information of the cores using only one core's output scan channels. Early work of on-chip comparison was published in 1989 [9].

The objectives and considerations that have guided our approach include the following:

- Fault detection characteristics of core patterns as comprehended by the original core level ATPG must not be compromised.
- We want our TAM not to be a limiting factor in how fast scan patterns can be applied to the multiple identical cores. Practically, this implies the use of pipelining registers.
- Our solution should be scalable. The core may be used in different SoC's with different numbers of cores. Extension of the solution to handle different numbers of cores should be straightforward.
- The solution should be flexible to accommodate a range of manufacturing optimization criteria and different SoC configurations. This flexibility is primarily related to the trade-off between ease of fault diagnosis and test throughput.
- Though we expect our core's test patterns to have some X data volume, we expect that the masking data volume is small compared to the total test pattern size. That is, for our application, we can optimize for sparse X's.

2. Architecture

Figure 1 is a simplified diagram of one configuration of the basic pipelined architecture for a toy example of treatment for three identical cores shown as the three larger rectangles. The small rectangles are pipeline registers for the various data lanes. The four primary lanes of data are shown in magenta, green, blue, and red for command,

1-4244-4203-0/08/$20.00

scan input (SDI), scan output (SDO), and match data respectively. Data flow in the figure is from top and left to bottom and right. All of the lanes may comprise a plurality of bits. The comparators (circles) and AND gates comprise a cascaded output response comparator. With this pipeline structure, every core receives the same test pattern stimulus in a staggered fashion. Several configurations of this triplet assembly will be discussed in subsequent sections; not every configuration uses all of the assembly's inputs on the left nor all of its outputs on the right. Clocks for this structure are not shown, but consider that the TAM circuitry receives a free-running clock stream (TAM_Clk) of the same frequency as is appropriate to shift data through the scan chains of the cores. All of the TAM pipeline registers are always updating with every cycle of TAM_Clk whether or not the cores are actually shifting.

Figure 1: Simplified TAM Architecture

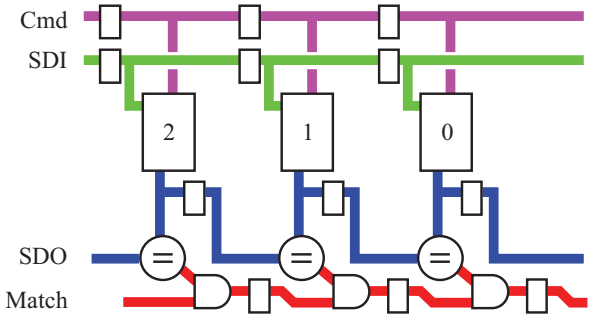


Table 1: TAM Commands

Command	Operation(s)
Shift	• Shift the core's scan chains. • Record any SDO channel miscompares in the core's respective error register.
First Pattern Shift	• Shift the core's scan chains. • Reset the core's error register.
Shift and Clear Mask	• Shift the core's scan chains. • Record any SDO channel miscompares in the core's respective error register. • Clear the mask register.
Load Mask	• Do nothing to the core. • Move the data from the SDI pipeline stage to the mask register.
Capture	• Trigger an at-speed capture sequence in the core.
Nop	• Do nothing to the core. • Move the data from the SDI pipeline stage to the SDO pipeline stage.

The data registered in each core's respective command pipeline stage encodes the operation that the core or its TAM circuitry are to do with the data currently resident in the respective core's SDI pipeline stage during the next TAM_Clk cycle. The commands are for operations like shift the scan chains, apply capture clocks, load a mask value, or do nothing (Nop). The command operations are applied to each core successively, one TAM_Clk cycle apart, according to the cores' positions in the pipeline. To apply high-speed transition tests, the core will need an at-speed capture mechanism that can be triggered by the TAM's capture command, as described in [10]. A set of commands are listed in Table 1 and subsequent sections will discuss how the commands are used in test application protocols.

The SDI lane of Figure 1 conveys the test pattern stimulus to each core in the staggered pipelined manner, but it is also used for mask data and, in one configuration, for expected response data. Extending the pipeline metaphor, if we convey a test pattern through the pipeline with a series of shift commands and if we interrupt for a cycle the application of the pattern and, instead, inject a Load Mask or Nop command, then for that cycle we can be said to have inserted a bubble into the pipeline. The bubble can be used to insert data other than the scan input pattern data.

Figure 2: TAM Architecture Detail

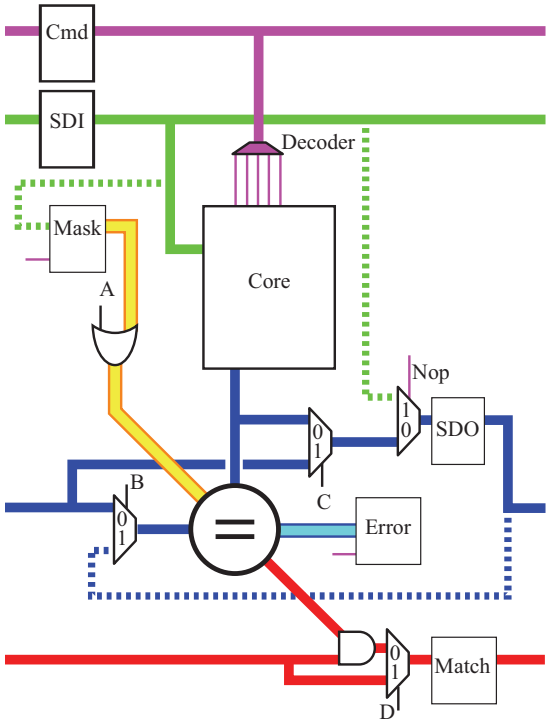


Figure 2 is a more detailed diagram of the TAM circuitry provided for one core. (This circuitry is replicated for each core.) Note that, in addition to the connections from the SDI pipeline register to the core, there are connections to a mask register and to the SDO pipeline register shown with the hashed green lines.

The SDO lane has a comparator of the core's scan output stream and either an upstream core's scan output or a connection from the SDO pipeline register (of the present core) shown as the hashed blue line. The comparator operation is conditioned by a mask value (yellow). Ones in the mask disable the corresponding bit positions from mismatching. The bit-wise outputs of the comparator (light blue) are provided to the error register. The error register bits are "sticky" -- they record if any mismatch occurs

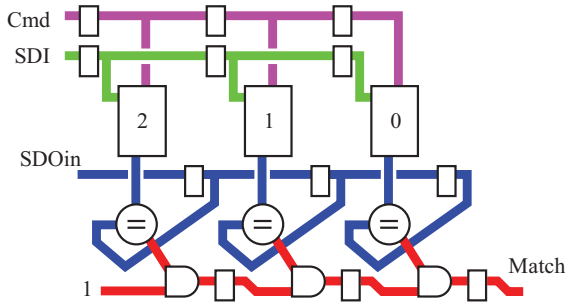
during a whole test application sequence. The total comparator output is ANDed with the upstream pipeline match signal. The MUXes of Figure 2 are described in subsequent sections but the configuration with all of the MUX select signals set to zero corresponds to Figure 1. The signals A, B, C, and D are static configuration controls provided uniquely and separately to each core's TAM circuitry. The MUX selector, Nop, changes in response to the command stream.

Masking is necessary to prevent corruption of the error register if there are any unknown data in the core test pattern. Instead of dedicating chip pins to masking as in [8], this scheme uses the pipeline bubble method to provide data to a mask register (depicted in Figure 2). This works especially well if the pattern data has few masks or infrequently changing masks. The "Shift and Clear Mask" command is an expedient way to clear the mask registers without stalling the shifting. Control signals A also force all channels to be masked if a core is to be removed from the collective test. A consequence of this masking method is that the number of cycles in each pattern, as translated to chip pins, may vary.

In the next few sections we will consider different configurations and usage modes of the individual core TAM circuits of Figure 2 in pipelined assemblies.

2.1. Full-Rate Self-Compare Mode

Figure 3: Full-Rate Self-Compare Mode



By rededicating all the scan output pins of the chip to be input pins for expected pattern data, the TAM circuitry can be configured to have each core's test results compared to the expected results. To do this, control signals B and C for each core are set to one. This configures a pipeline arrangement as depicted in Figure 3. In this mode, for each shift type command cycle, each core successively compares its results to the expected pattern data and passes on its component of the total comparison result down the match chain. The match pipeline chip output indicates on a cycle-by-cycle basis whether any core mismatched on any channel. The error registers record whether their respective cores ever mismatched. This full-rate self-compare mode provides a pass/fail determination for *N* cores individually in the time it takes to test one core (a throughput increase of a factor of *N*).

2.2. Interleaved Self-Compare Mode

The conventional protocol of a scan test pattern application is Shift, Shift, …, Shift, Capture, Shift, …, Shift, etc. We may accomplish a time multiplex of interleaved use of the SDI stream by changing the test application to Nop, Shift, Nop, Shift, ..., Nop, Shift, Capture, Nop, Shift, Nop, Shift, …, Nop, Shift, etc. On the Nop cycles, the MUX in front of the SDO pipeline register loads data from the SDI stream (the right side hashed green line of Figure 2). Setting the control signal B to one allows that data to be presented to the comparator. Alternately, expected data with a Nop command and then stimulus data with a Shift command are applied. On the shift cycles, the MUXes with selector C determine which of either the respective core's scan output stream or an upstream core SDO stage's data are loaded into the respective core's SDO stage. By appropriately setting the collective set of each core's control signals C, any one of the cores' scan output streams can be piped to the chip's SDO outputs.

Figure 4: Interleaved Self-Compare Mode

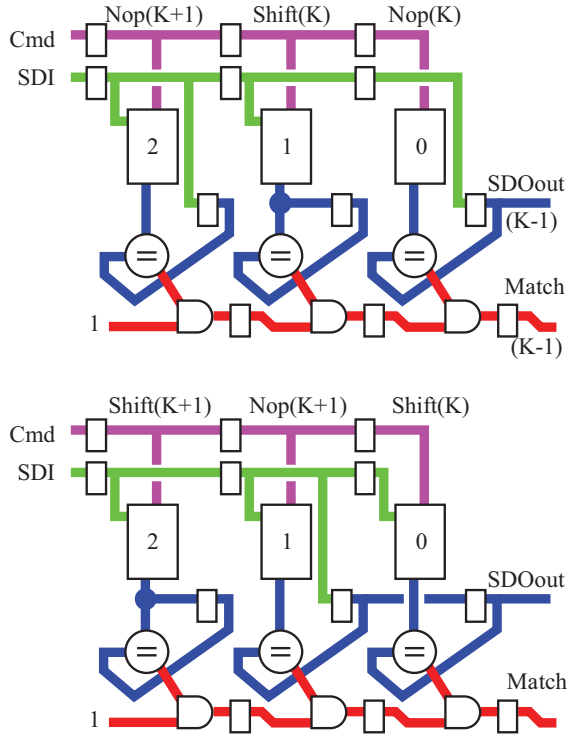


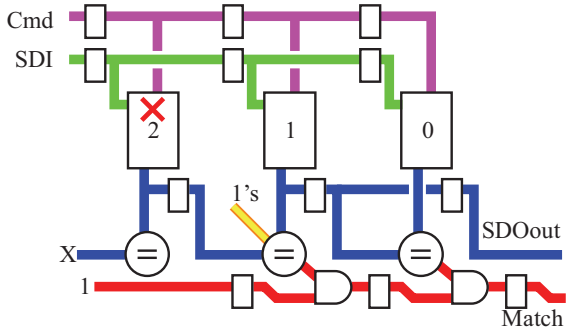
Figure 4 depicts two successive operational states of the TAM circuitry configured to directly observe core 1's scan output stream in interleaved self-compare mode. In this example C[2:0]=(0,0,1). This mechanism of selecting which core's response is to be directly observed has the beneficial characteristic that the test pattern as translated to chip pins is the same irrespective of which core is chosen. The match pipeline chip output indicates on a cycle-by-cycle basis whether any core mismatched on any channel. The error register records whether each core ever mismatched. Interleaved self-compare mode shifts every other cycle so it provides a pass/fail determination for *N* cores

individually in the time it takes to test two cores serially (a throughput increase of a factor of $N/2$).

2.3. Inter-core Compare Mode

If a core is directly observed and confirmed by ATE to be good and if other cores are compared to the verified core and shown to produce the same responses as the directly observed core, then the other cores are also verified. This is the principle of the inter-core compare mode, an arrangement of which is depicted in Figure 5. In this example core 2 has on a previous experiment been directly observed to fail the test pattern. Core 1 has been configured to be directly observed for the current experiment. The match AND gate of core 2 has been bypassed by setting control signal D[2]=1. In inter-core compare mode the error register of the most upstream core participating on the test does not provide any useful test information. In this example the error register of core 0 (not shown) will indicate any mismatches that may occur between it and the directly-observed core 1. The match signal indicates if any cores disagree on a cycle-by-cycle basis.

Figure 5: Inter-core Compare Mode



With inter-core compare, the throughput acceleration is more complicated to reckon because, if the directly observed core turns out to be bad, additional experiments may be needed to determine the disposition of the other cores. Consider the flow for testing a quad-core chip with inter-core compare mode. Suppose there is a policy that there must be at least two good cores to salvage the chip as a usable (but derated) chip. We start by running an experiment with direct observation of core 3. If core 3 is observed to pass the test pattern, then we may determine the disposition of cores 2, 1, and 0 by checking their respective error registers at the conclusion of the experiment. If, on the first experiment, core 3 is bad then we remove core 3 from the configuration by setting control signal A[2]=1 and configure core 2 for direct observation by setting C[2:0]=(0,1,1). Then a second experiment is run and, if core 2 is good, then we know the disposition of cores 1 and 0 as well; otherwise, we must run a third experiment directly observing core 1 by setting C[2:0]=(0,0,1). In the case that the third experiment finds core 1 to be bad, the program exits without a pass/fail determination for core 0 because the chip has failed the deration policy. The average number of experiments, E, that are necessary to determine pass/fail status of each core (up to the choice to exit according to the deration policy) may be calculated as a function of the per core yield, Y. It is pessimistic to assume that the yield of a particular core is the same regardless of whether other cores are defective [11], but let us hold with that pessimistic assumption for the sake of simplicity in this explanation. For this "at least two out of four core" example, Table 2 shows the relative probabilities of all possible experimental outcomes.

Table 2: Four-core Experiment Probability (Pessimistic)

Core[3:0] pass/fail	Number Of Experiments	Probability
PPPP	1	Y^4
PPPF	1	$Y^3(1-Y)$
PPFP	1	$Y^3(1-Y)$
PPFF	1	$Y^2(1-Y)^2$
PFPP	1	$Y^3(1-Y)$
PFPF	1	$Y^2(1-Y)^2$
PFFP	1	$Y^2(1-Y)^2$
PFFF	1	$Y(1-Y)^3$
FPPP	2	$Y^3(1-Y)$
FPPF	2	$Y^2(1-Y)^2$
FPFP	2	$Y^2(1-Y)^2$
FPFF	2	$Y(1-Y)^3$
FFPP	3	$Y^2(1-Y)^2$
FFPF	3	$Y(1-Y)^3$
FFFP	3	$Y(1-Y)^3$
FFFF	3	$(1-Y)^4$

Adding the like terms weighted for the number of experiments we obtain

$$E(Y) = Y^4 + 5Y^3(1-Y) + 10Y^2(1-Y)^2 + 9Y(1-Y)^3 + 3(1-Y)^4 \quad (1)$$

which simplifies to a quadratic

$$E(Y) = Y^2 - 3Y + 3 \quad (2)$$

If the per core yield is greater than 38% then the inter-core compare mode has a throughput advantage over the interleaved compare mode; that is, in this example for $Y > 38\%$, in high volume production we can test the quad-core chip in less time than it takes to test two cores serially. For Y=90%, the average number of experiments is 1.1.

Consider the general case in which there are N cores and we require at least G good cores to sell the chip. We denote the maximum number of experiments to identify all of the good cores on all of the sellable chips as E_{max}. If we conduct N-G experiments in which each of the directly observed cores fails, then there is only one last experiment to determine if the remaining G cores are all good; therefore

$$E_{\max} = N - G + 1 \quad (3)$$

If we require the E_{max} experiments it is because we have failed E_{max}-1 cores by successive direct observation

experiments. The probability of E_{max}-1 cores failing is $(1-Y)^{E_{\max}-1}$. By counting the experiments and the probabilities that the respective experiments will be necessary we obtain

$$E(Y) = 1(1) \leftarrow \text{Prob. of need of } 1^{st} \text{ experiment} \quad (4)$$
$$+ 1(1-Y)^{1} \leftarrow \text{Prob. of need of } 2^{nd} \text{ experiment}$$
$$+ \cdots$$
$$+ 1(1-Y)^{E_{\max}-1} \leftarrow \text{Prob. of need of } E_{max} \text{ experiment}$$

Combining equations (3) and (4) yields the following general formula for the average number of experiments.

$$E(Y,N,G) = \sum_{i=0}^{N-G} (1-Y)^{i} \quad (5)$$

2.4. Direct Core Access using Asymmetric Hardware Compression

For microprocessor cores, the use of on-chip hardware for scan compression is common. The SDI pins of the TAM feed a decompressor and the SDO pins come from a compactor for each core. A common configuration has been one with an equal number of input and output channels entering and leaving each core. However, commercial hardware compression tools have recently started supporting a higher compaction ratio at the scan output of the core compared to the decompression ratio at the input. As a result, as compared to the symmetrical input to output scheme, use of just the higher output compaction liberates outputs that may then be redeployed for other purposes. The proposed TAM architecture can easily be extended to utilize the availability of the liberated output channels for multiple tracks of inter-core compare.

If the output compaction ratio is chosen to be twice that of the input compression ratio, only half as many output channels are required. The remaining half of the output channels can be utilized in at least two ways:

- The liberated channels can be reapportioned to the input and output channels according to the 2:1 ratio respectively to achieve greater compression.
- The spare output channels can be used to directly observe the response of another core. This allows for two sets of cores to be compared in parallel to each observable core.

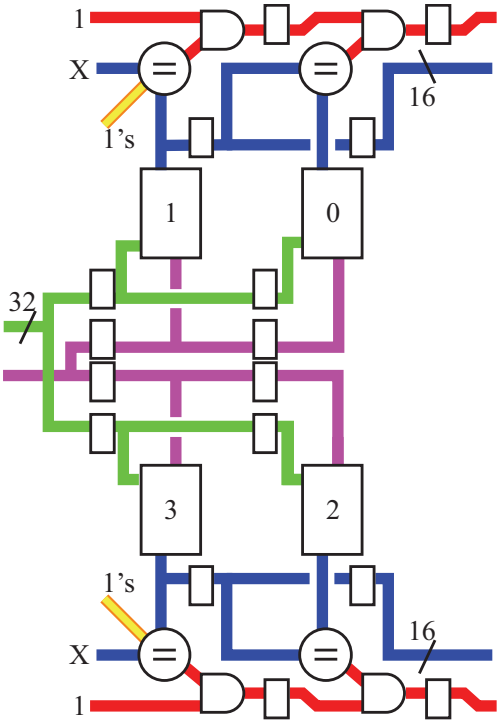
Figure 6 illustrates the second scenario for a chip with four cores. Since there are two available cores that are fully observable, two tracks of two cores configured in inter-core compare mode are used. In this way the pipeline of inter-core comparison is limited to two, instead of four, cores. The shorter pipelines of comparison in parallel allow for lower maximum number of experiments (E_{max}) needed to detect failing cores.

The use of multiple tracks for a direct-compare mode is not necessarily contingent on the use of asymmetric hardware compression. The total number of available TAM channels can be distributed across multiple tracks; however, there are a few tradeoffs involved.

If the compression ratio remains unchanged, going from one track to multiple tracks reduces the number of available channels to each core, which results in an increase in the scan chain lengths; this, in turn, results in an increase in the overall test time.

A second tradeoff is that if the scan chain lengths remain unchanged, going from one track to multiple tracks requires an increase in the effective compression and compaction ratio to account for the fewer number of available scan channels; this, in turn, typically results in an increase in the decompressor and compactor area.

Figure 6: TAM with Asymmetric Hardware Compression



Consider the flow for testing a quad-core chip with inter-core compare mode across two tracks, as shown in Figure 6. Suppose there is a policy that there must be at least two good cores to salvage the chip as a usable derated chip. We start by running an experiment with direct observation of Core 1 and Core 3; in each experiment, the cores in both tracks are tested in parallel. If Core 1 and Core 3 are observed to pass the test pattern, then we may determine the disposition of the remaining cores by checking their respective error registers at the conclusion of the experiment. If, however, Core 1 or Core 3 is bad on the first experiment, then we remove the bad core for observation and configure the other core in the respective track to become the observable core. Thus, similar to the case with a single track of inter-core compare mode, depending on the pass or fail status of the observable core in each track after each experiment, each track can be re-configured independently to switch to a different observable core.

As in the case with a single track, the average number of experiments, E, necessary to determine the pass/fail status of each core may be calculated as a function of the per-core yield, Y. Table 3 shows the relative probabilities of all possible experimental outcomes for a quad-core with two tracks.

Table 3: 4-Core, 2-Track Experiment Probability

Core[1:0], Core[3:2] pass/fail	Number Of Experiments	Probability
PP,PP	1	Y^4
PP,PF	1	$Y^3(1-Y)$
PP,FP	2	$Y^3(1-Y)$
PP,FF	2	$Y^2(1-Y)^2$
PF,PP	1	$Y^3(1-Y)$
PF,PF	1	$Y^2(1-Y)^2$
PF,FP	2	$Y^2(1-Y)^2$
PF,FF	2	$Y(1-Y)^3$
FP,PP	2	$Y^3(1-Y)$
FP,PF	2	$Y^2(1-Y)^2$
FP,FP	2	$Y^2(1-Y)^2$
FP,FF	2	$Y(1-Y)^3$
FF,PP	2	$Y^2(1-Y)^2$
FF,PF	2	$Y(1-Y)^3$
FF,FP	2	$Y(1-Y)^3$
FF,FF	2	$(1-Y)^4$

Again, adding the like terms weighted for the number of experiments we obtain:

$$E(Y) = Y^4 + 6Y^3(1-Y) + 11Y^2(1-Y)^2 + 8Y(1-Y)^3 + 2(1-Y)^4 \quad (6)$$

which simplifies to:

$$E(Y) = 2 - Y^2 \quad (7)$$

which, interestingly, has a different sign in Y^2 compared to the single-track example.

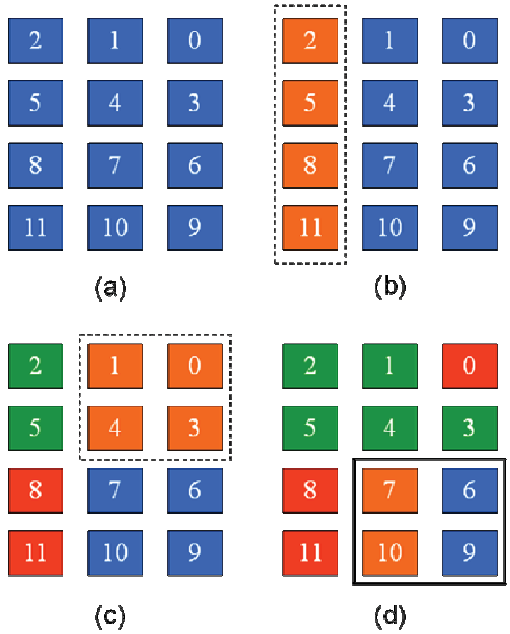
Extending this calculation for the general case of N cores and G required good cores with T tracks requires some analysis. Since multiple cores are tested in each experiment, several outcomes are possible depending on the number of passing cores. Also, the pass/fail information about all downstream cores will be available for the tracks that have passing cores in the current experiment.

$$E_{\max} = \min(N - G + 1, \frac{N}{T}) \quad (8)$$

The maximum number of experiments required is given by equation (8). This is because the worst case for the number of experiments occurs when all the failures are in the same track. The term N-G+1 takes into account the fact that the maximum number of failures allowed can be less than the depth of each track.

The average number of experiments is derived using a recursive algorithm. The recursion has two terms, the first in case no information about any passing core is available (e.g., at the beginning of the experiment) and the second taking into account information about passing cores. These two terms are given in equations (9) and (10), where P represents the number of passing cores after an experiment. If the arrangement of cores in multiple tracks is thought of as a rectangle with height T and width (N/T), the two terms are reducing the width and height of the rectangle respectively.

Figure 7: Recursive Partitioning Example



This algorithm is best illustrated with the help of an example. Consider the example SoC in Figure 7 (a), which has 12 cores in four tracks with three cores in each track. The color blue means no information about the core is available; green and red represent passing and failing cores respectively. Cores that are part of the current evaluation are shown in orange. Let the minimum number of good cores required, G, be 8. In the first experiment, four cores

$$E(N,G,Y,T,P=0) = 1 + \sum_{i=1}^{\min(N-G,T)} Y^{(T-i)}(1-Y)^i C_T^i E(N-T, G-T+i, Y, T, T-i) \quad (9)$$

$$E(N,G,Y,T,P>0) = \sum_{j=0}^{\min(N-G,\frac{PN}{T})} C_{\frac{PN}{T}}^j Y^{(\frac{PN}{T}-j)}(1-Y)^j E(N-\frac{PN}{T}, G+j-\frac{PN}{T}, Y, T-P, 0) \quad (10)$$

(2, 5, 8, and 11) are tested. Since an experiment is done, 1 is added to the term in equation (9). The next expression in equation (9) sums over all the possibilities of a different number of cores passing this experiment. The probability that exactly i cores fail in the current experiment is given by $Y^{(T-i)}(1-Y)^i C_T^i$. In this case, the next term should take into account the (T-i) passing cores among the remaining (N-T) cores to be tested with a requirement of at least G-(T-i) good cores. In the example of Figure 7 (c), i = 2. Hence, the algorithm now moves to equation (10) to calculate $E(N=8,G=6,T=4,P=2)$.

Equation (10) sums over all the possibilities of a different number of cores passing in the tracks with passing cores in the previous experiment. In the case of our example, it is summing over all possibilities for cores 0, 1, 3, and 4. For exactly j cores failing among these cores, the recursion algorithm continues with the remaining cores (N-PN/T) in the remaining tracks (T-P), with the requirement that at least G-(PN/T-j) cores pass. Since we don't know anything about the cores in the remaining tracks, equation (9) is used. If we assume that only core 0 failed, the algorithm will progress to calculate $E(N=4,G=3,T=2,P=0)$ as shown in Figure 7 (d).

3. Trade-offs between Test Throughput, Diagnosis, and TAM Complexity

We have discussed various configurations and modes of use of this modular pipelined TAM for multiple identical cores. Full-rate compare can determine the pass/fail disposition of any number of cores in essentially the same time as it takes to test a single core. Interleaved compare mode takes twice as long to make the determination. The throughput acceleration afforded by inter-core compare modes are strong functions of yield or defectivity. The main difference between these usage modes is the amount of diagnostic information that can be gathered.

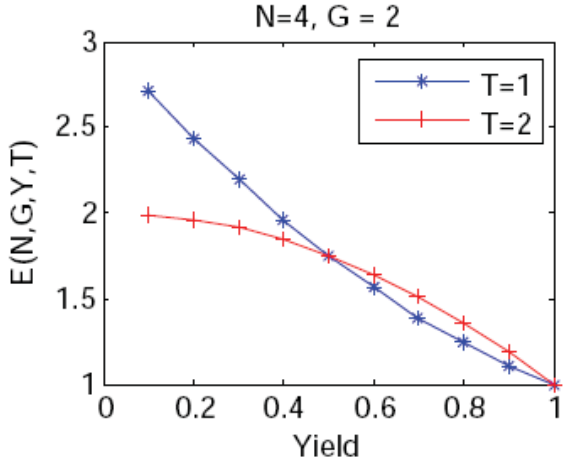
The only diagnosis data that is produced by full-rate compare mode is which scan chains fail on defective cores. That is very low resolution data of very limited use for yield learning. But if the SoC test yield is 99%, there may be no interest in yield improvement.

Debug and other purposes demand the existence of a means to test one core at a time with maximum diagnostic resolution. Such a mode requires the existence of scan output pins. In practice, it may not be acceptable to have some SoC pins be configurable as both inputs for expected response data in full-rate compare mode and scan outputs in a single-core full-access mode. So full-rate configuration may not be attractive or attainable for some SoCs or for some times in a SoC's manufacturing cycle.

Interleaved compare mode can fully observe any single core, but there is only failing chain data for the non-observed cores. The translated test pattern is the same regardless of which core is observed so, for volume production, every core can be directly observed for a fraction of the volume. But the throughput gains of full-rate self-compare mode are not limited to a factor of 2, so the factory might want to choose between interleaved compare and full-rate self-compare depending on the per-core yield. Inter-core compare mode is biased for directly observing the most upstream core. For SoCs with many cores, the downstream cores would be observed very infrequently. This sampling bias can be remedied by providing more configuration circuitry such that the TAM is made into a ring wherein any core can be the most upstream entry point.

The analysis methodology for the expected number of experiments for the different numbers of tracks and deration policies can lead to another regime of optimization criteria. More tracks afford more direct observations and more diagnostic information. We've also seen that, for a given number of cores, the optimal number of tracks (from a throughput perspective) may depend on the per-core yield. This highlights the possibility of providing more TAM circuitry to configure different numbers of tracks.

Figure 8: Expected Experiments vs. Core Yield for 4 Cores

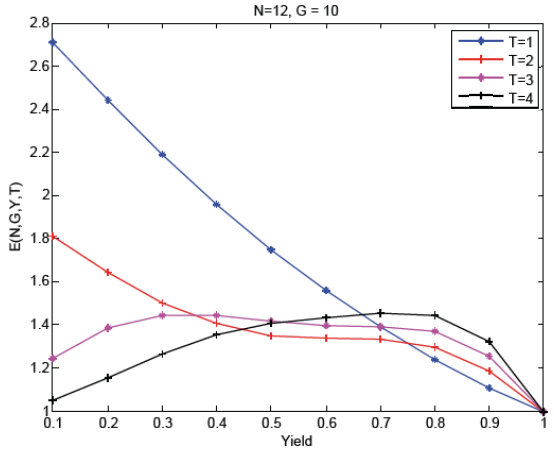


Consider the example of four cores with a requirement that at least two of them must be good. The expected number of experiments required in the inter-core compare mode for a single track (T=1) and two tracks (T=2) are given by formulae (2) and (7) respectively. Figure 8 shows the average number of experiments plotted against the core yield. Lower average number of experiments means the SoC spends less time on the tester and, hence, will lower the test costs. As seen from Figure 8, if the yield per core is less than 50%, two tracks with two cores in each track is better that a single track of four cores.

The same idea can be extended to 12 cores. Using formulae (9) and (10), Figure 9 shows the expected number of experiments vs. yield for the case of 12 cores, where no more than two cores may fail, and for four different track configurations (T=1,2,3,4). This graph shows that as the number of tracks T increases, the expected number of experiments E falls for low yield (Y<40%), but it increases for high yield (Y>70%). Using this analysis, the TAM may be reconfigured dynamically to select an arrangement

of tracks that minimizes the expected experiments for a given yield.

Figure 9: Expected Experiments vs. Core Yield for 12 Cores



4. Preparing TAM Test Patterns

One advantage of a test access mechanism for multiple identical cores is that it may permit the test generation process to focus on generating tests for one instance of the core, and then engage the TAM to apply that test pattern to all of the identical cores without explicitly targeting each core during test generation. The following procedure uses this idea to produce and apply test patterns that utilize the TAM described above:

1. Generate core-level patterns
2. Translate to TAM patterns
3. Apply tests to chip, collect TAM fail data
4. Translate fail data to core level
5. Diagnose failures at core level

Step one refers to generating patterns that are suitable for application at the boundary of the core, such as instance 0 of the core inFigure 1. The translations mentioned in steps 2 and 4 are introduced as a type of pattern data manipulation wrapper around the step of using the TAM when testing the chip. Prior work of TAM pattern translation was presented by Marinissen et. al. [12]. An examination of this translation process is another approach to understanding the proposed TAM. The task of translating patterns and fail data can be segmented into a few basic transformations of the pattern data which are described incrementally in the following subsections.

4.1 Pipelining and TAM Instructions

The most basic transformation required when translating core-level pattern data to TAM patterns is handling the additional pipeline stages that are introduced between the core boundary and the chip pins. As an example, consider the core-level pattern data that is prepared for application at the core boundary for a very simple example core with a single scan chain of four scan flops. This core-level pattern is represented in Table 4, showing pattern data fed into the inputs (In) and expected values at the output pins (Out) for each cycle. The load and unload data for the first pattern is colored **blue**, and the load data for the second pattern is **green**. The "C" represents the cycle in which capture clocks are applied between loading and unloading pattern data.

Table 4: Core-level Pattern Data

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
In:	1	2	3	4	C	1	2	3	4
Out:	-	-	-	-	-	1	2	3	4

Suppose we have a design that contains three instances of this example core as in Figure 5, and we wish to translate the core-level pattern data to use the proposed TAM in inter-core compare mode. From the perspective of core 1, the TAM introduces two pipeline stages to the input stream, and two pipeline stages to the output stream. The transformation to handle the additional pipeline stages is straightforward, as depicted in Table 5.

Table 5: After Pipeline Transform

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
TAM Cmd:	S	S	S	S	C	S	S	S	S	C	S	S	S
SDI:	1	2	3	4	C	1	2	3	4	C	1	2	3
SDOout:	-	-	-	-	-	-	-	-	-	**1**	**2**	**3**	**4**
Match:	-	-	-	-	-	-	-	-	-	1	1	1	1

The data provided at the input pins for this pattern is unchanged, but the pattern data is delivered to the core 2 stages later due to the pipelining of the input stream, as shown in Figure 5. The TAM instructions will be delivered to the core with the same pipelining as the input pattern data, causing the TAM circuitry to execute the shifts and captures on core 1 at the correct times (two cycles after they appear at the chip inputs).

In Table 5, the expected output data is transformed to reflect the delay in data coming out of core 1. The output of core 1 is delayed by two cycles due to input stream pipelining, and an additional 2 cycles due to output stream pipelining, producing a transformation of the output data that delays the expected values by four cycles.

Table 5 also depicts the addition of expected data for the Match signal produced by the TAM comparator system. Adding this match data is a simple matter of expecting the Match output to show its affirmative value for every cycle that produces expected data from the cores. The Match data is pipelined with the rest of the output data, so it receives the same pipeline transformation as the other output data.

It is important to note that, although we have been describing the pattern data from the perspective of core 1, the resulting transformed pattern will work with inter-core compare mode regardless of which core is being observed. This is because the pipeline transformation for TAM-enabled modes depends only on the sum of input and output stream pipeline depths, and that sum is the same for all cores (1+3 for core 3, 2+2 for core 1, and 3+1 for core 0).

Therefore, only one translation is required to prepare patterns for inter-core compare mode, regardless of the configuration of the TAM's core observation muxes.

4.2 X-Masking

We can further extend our working example to a case in which we do not know what the expected value is for one of the cycles of the core-level data. Table 6 shows this unknown value as an **X**.

Table 6: Core-level Pattern Data with X

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
In:	1	2	3	4	C	1	2	3	4
Out:	-	-	-	-	-	1	2	**X**	4

The proposed TAM includes a masking feature (described in section 2.5) to prevent this unknown data from causing a mismatch to be registered by the TAM comparator system. To make use of this X-masking feature, the translator must identify the cycles of unknown output data and insert a new cycle into the input stream with the appropriate TAM command and mask data.

Table 7: After X-Masking Transform

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
TAM Cmd:	S	S	S	S	C	S	S	**L**	SCM	C
In:	1	2	3	4	C	1	2	**M**	3	4
Out:	-	-	-	-	-	1	2	-	**X**	4
Match:	0	0	0	0	0	1	1	0	1	1

Table 7 depicts the translated pattern data for our example, accommodating the X-masking feature (but without the pipeline transformation at this point). Note especially the new data inserted at cycle 8 to load the mask register. The input stream is used to specify the mask register data (M), and the TAM command stream is set to the "load mask" instruction (L). The following cycle (9) is a shift operation, but we change the TAM command to a "Shift and Clear Mask" so that the X-mask register will be cleared to permit the comparator to operate unmasked on the last cycle of unload data. Note that the output stream has a useless cycle, or "bubble," in it at the time when the extra cycle is inserted for loading the mask.

The translated pattern data in Table 7 can be further transformed by the pipeline transformation described earlier, resulting in a translation that handles both pipelining and X-masking. It is easier to conceptualize performing the pipeline transform after the X-masking transform due to the complexity of positioning the inserted cycle, and the resulting "bubble" in the output stream when that extra cycle propagates to the output pins. The "bubble" remains in its relative position when the pipeline transform is applied, but the entire output stream (including match signal) is delayed by the total number of pipeline stages (input + output) as mentioned earlier. The result of applying both transforms is in Table 8.

To optimize the use of the X-mask feature, the X-mask transformation must recognize that multiple consecutive cycles of unload data with the same X pattern do not require multiple loads of the X-mask register. This requirement, along with the task of inserting cycles into the streams, makes the X-mask transformation the most complex transformation required for the proposed TAM.

Table 8: After X-Masking and Pipeline Transforms

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
TAM Cmd:	S	S	S	S	C	S	S	L	SCM	S	C	S	S	S
In:	1	2	3	4	C	1	2	M	3	4	C	1	2	3
Out:	-	-	-	-	-	-	-	-	-	1	2	-	**X**	4
Match:	-	-	-	-	-	-	-	-	-	1	1	-	1	1

4.3 Interleaved Data Insertion

To use interleaved self-compare mode, the translation must insert additional cycles providing the expected data on the input stream. Starting with the core-level pattern data from Table 4, the interleave-transformed data will appear as in Table 9.

Table 9: After Interleave Transform

Cycle:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
TAM Cmd:	S	S	S	S	C	**N**	S	**N**	S	**N**	S	…
In:	1	2	3	4	C	**1**	1	**2**	2	**3**	3	…
Out:	-	-	-	-	-	-	1	-	2	-	3	…
Match:	0	0	0	0	0	0	1	0	1	0	1	…

The interleave transform is simply insertion of the expected data before each cycle when it would normally appear on the output of the core. As with X-masking, this transform inserts cycles that did not exist in the original core-level pattern; thus, it suffers from the same complexity of the X-masking, and is easiest to conceptualize when applied before the pipeline transformation.

A translation process can be composed of a series of these transforms according to the desired mode of TAM operation. For example, to translate core-level patterns for use in interleaved mode, the translation may be composed by applying the X-masking transform, followed by the interleave transform, and finally the pipeline transform.

4.4 Fail Data Translation

When the translated pattern is applied to the chip and the fail data is collected, the fail data will not be readily usable by diagnostic tools due to the translation performed on the pattern data to enable the TAM features. Usually, diagnostic tools utilize the same environment and models that were used in the test generation process. In this case, that test generation process is at the core level, so the fail data must be translated back to a core-level perspective for use by diagnostic tools.

To translate the fail data to a core-level perspective, we have to account for all the cycles that were added to the pattern data by translation (X-masking and interleaving), as well as the offset of the pipeline transformation. Given a failing cycle number (*FailCycle*), the resulting core-level failing cycle number ($FailCycle_{core}$) is calculated according to formula (11).

$$FailCycle_{Core} = FailCycle - (L + I) - PipeStages \quad (11)$$

In formula (11), *L* represents the number of load mask instruction cycles from the beginning of the pattern (cycle 0) up to the *FailCycle*. *I* represents the number of cycles inserted to handle interleaved data up to the *FailCycle*, and *PipeStages* is the sum of pipeline stages accounted for by the translation (input + output).

As an example, suppose a chip is tested that will produce a mismatch on the fourth bit of the first pattern unload. The pattern in Table 8 will produce a fail log showing a failing comparison on cycle 14. To translate this fail cycle number to a core-level perspective, we subtract 1 to account for the cycle inserted by the X-masking transform. Then we subtract 4 to account for the pipeline transform. This results in calculating cycle 9 as the core-level fail cycle. Comparing this to Table 4, we see that cycle 9 corresponds to bit 4 of the first pattern unload, which was the premise of this example, and will permit diagnostic tools to interpret the fail data.

4.5 Final Considerations

When translating patterns for use with the proposed TAM, a few additional considerations must be made. During the first pattern load operation, unknown data may shift out of the cores and into the comparators; thus, the translation process must use the "first pattern shift" command on the TAM command stream during the first load.

Also, care must be taken to ensure that the TAM logic is initialized such that the error registers are cleared before pattern data is applied. The TAM command pipeline should also be cleared to contain the "Nop" instruction so unknown commands do not execute at the cores while the first cycle of data is still shifting through the input pipeline. Both of these can be accomplished by flushing the TAM command pipeline with the "first pattern shift" command followed by "Nop" commands.

5. Conclusions

We have described a Test Access Mechanism that uses on-chip comparison to reduce the amount of test data and time necessary to test an SoC with multiple identical cores. The architecture is modular and scalable in timing and area and easily works for a large number of cores in SoCs. Test generation complexity of the SoC is also reduced since core level patterns can be generated and reused for the different cores. The protocol translation transforms required to convert the core-level patterns to SoC patterns and the reverse for core-level diagnosis of SoC pattern failures have been discussed.

The many usage modes and configurations of this architecture allow for adapting to changing optimization criteria over the manufacturing cycle of a product in a factory. There are a range of different operating points, from early in a product's life when yields are low and diagnostics are very important, to a very mature product that requires only minimal diagnostic monitoring. We have quantified the throughput acceleration that can be expected of the different modes. In particular, for reasonably high yields, the single-track inter-core compare mode provides highest testing throughput and a steady stream of diagnostically useful data.

The proposed test access mechanism is flexible in design, configuration, and application, making it an attractive solution to reduce test time when testing multiple identical cores.

6. References

[1] Parulkar I., et. al., "A Scalable, Low Cost Design-for-test architecture for UltraSPARC Chip Multi-Processors", *Proc. of International Test Conference*, pp. 726-735, 2002.

[2] Riley, M., et. al., "Testability Features of the First-Generation Cell Processor", *Proc. of International Test Conference*, Paper 6.1, 2005.

[3] Tan, P. J., et. al., "Testing of the UltraSPARC T1 Microprocessor and Its Challenges", *Proc. of International Test Conference*, Paper 16.1, 2006.

[4] Molyneaux, R., et. al., "Design for Testability Features of the SUN Microsystems Niagara2 CMP-CMT SPARC Chip", *Proc. of International Test Conference*, Paper 1.2, 2007.

[5] Silva, F. D., et. al., The Core Test Wrapper Handbook: Rationale and Application of IEEE Std. 1500™, Springer-Verlag New York, LLC, 2006.

[6] Zorian, Y., et. al., "Testing Embedded-Core Based System Chips", *Proc. Of International Test Conference,* Paper 6.2, 1998.

[7] Goel, S. K., et. al., "Effective and Efficient Test Architecture Design for SOCs", *Proc. Of International Test Conference,* Paper 19.2, 2002.

[8] Makar S., et. al., "Testing of Vega2, a Chip Multi-Processor with Spare Processors", *Proc. of International Test Conference*, Paper 9.1, 2007.

[9] Atwell W. D. Jr., et. al., "Tester on a Chip (TOAC) or Appratus for Application of Tests for Embedded Test Points", *Journal of Motorola Technical Developments*, Volume 9, 1989.

[10] Wood, T., et. al., "The Test Features of the Quad-Core AMD Opteron™ Microprocessor", *Proc. of International Test Conference*, 2008

[11] Stapper C. H., et. al., "Yield Model for ASIC and Processor Chips", *International Workshop on Defect and Fault Tolerance in VLSI System*, 1993.

[12] Marinissen E. J., et. al., "The Role of Test Protocols in Testing Embedded-Core-Based System ICs", *European Test Workshop*, 1999.