September 1996

UILU-ENG-96-2221
CRHC-96-11

University of Illinois at Urbana-Champaign

A Genetic-Algorithm Approach to Architectural-Level Justification of Precomputed Vectors

Michael S. Hsiao and Janak H. Patel

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB NO. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE A Genetic-Algorithm Approach to Architectural-Level Justification of Precomputed Vectors	5. FUNDING NUMBERS
6. AUTHOR(S) Michael S. Hsiao and Janak H. Patel	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES) Coordinated Science Laboratory University of Illinois 1308 W. Main St. Urbana, IL 61801	8. PERFORMING ORGANIZATION REPORT NUMBER (CRHC-96-11)
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Semiconductor Research Corp. P.O. Box 12053 Research Triangle Park, NC 27709-2053	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Test generation is an NP-complete problem in which thebottle-neck of the task is in value justification. A new technique isproposed in which the architectural information is efficiently used while gate-level description is unnecessary to guide valuejustifications. Justification of values consists of excitation phase and propagation phase. Previous work done have always been centered around branch-and-bound backtracing techniques, ARGAJUS (ARchitectural-level Genetic-Algorithm-based JUStification) takes the genetic-algorithms approach involving only forward computation to solve this problem.

14. SUBJECT TERMS genetic-algorithm, architectural-level, justification, precomputed vectors	15. NUMBER IF PAGES 23
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

A genetic-algorithm approach to architectural-level justification of precomputed vectors[1]

Michael S. Hsiao and Janak H. Patel

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois, Urbana, IL 61801
1308 W. Main Street

Abstract

Test generation is an NP-complete problem in which the bottle-neck of the task is in value justification. A *new* technique is proposed in which the *architectural* information is efficiently used while gate-level description is unnecessary to guide value justifications. Justification of values consists of excitation phase and propagation phase. Previous work done have always been centered around branch-and-bound backtracing techniques, ARGAJUS (ARchitectural-level Genetic-Algorithm-based JUStification) takes the genetic-algorithms approach involving only forward computation to solve this problem.

[1]This research was supported in part by the Semiconductor Research Corporation under contract SRC 95-DP-109, in part by DARPA under contract DABT63-95-C-0069, and by Hewlett-Packard under an equipment grant.

1 Introduction

There has been much research effort devoted to the test generation problem for combinational and sequential circuits in the recent years. Based on the stuck-at fault model, algorithms such as PODEM[1], D-algorithm[2], along with recent developments [3, 4, 5, 6] have been developed for the gate-level automatic test pattern generation (ATPG). Due to the hierarchical design style of modern VLSI circuits, high-level and hierarchical test generation have extended the gate-level ATPG to a new level of abstraction, using functional blocks as primitives in place of logic AND and OR gates[7, 8, 9]. With the functional information added to the global knowledge of the circuit, ATPG is no longer *hierarchically blind* to the upper-level information as did the gate-level test generators.

In addition to obtaining global high-level knowledge, other motivations for a higher level of ATPG are the desire to speed up the test generation process and reduce the complexity by hierarchically dividing the circuit into smaller partitions or modules. Furthermore, while a gate-level ATPG may be used to generate test vectors for the individual modules in the circuit, a hierarchical high-level ATPG may later be used to justify the **precomputed** vectors for the module. Gate level structures are not needed for justifying the values at high level; instead, equivalent behavioral C-like functions are used to guide the justification and propagation of values. This enables testing of the module prior to the implementation of the entire circuit is finished.

Based on the forward and backward value implication as well as fault effect propagation algorithms for the gate-level ATPG, new algorithms have been developed for higher level primitives so that the ATPG process can be accelerated. However, because most of these algorithms were extended from the D-algorithm or PODEM, the performance would be constrained by the properties of the search algorithms as well as the size of module partitions.

The proposed work, ARGAJUS (ARchitectural-level Genetic-Algorithm based JUStification), deviates from the traditional branch-and-bound approach of requiring both forward and backward implications during excitation and propagation of the fault. Instead, only forward calculations are involved to justify precomputed vectors for the internal modules. Genetic algorithm is used to implement the value justification process. The circuits that have been used are full-scan sequential circuits. ARGAJUS, however, can easily be extended to general non-scan sequential circuits.

This paper is organized as follows: an overview of architectural-level test generation and value justification is presented in Section 2, previous work in high-level ATPG discussed in Section 3, a tutorial of genetic algorithms given in Section 4, description of the framework and algorithm of ARGAJUS explained in Section 5, followed by experiments and conclusion in Sections 6 and 7, respectively.

2 Overview of Architectural-level ATPG and Value Justification

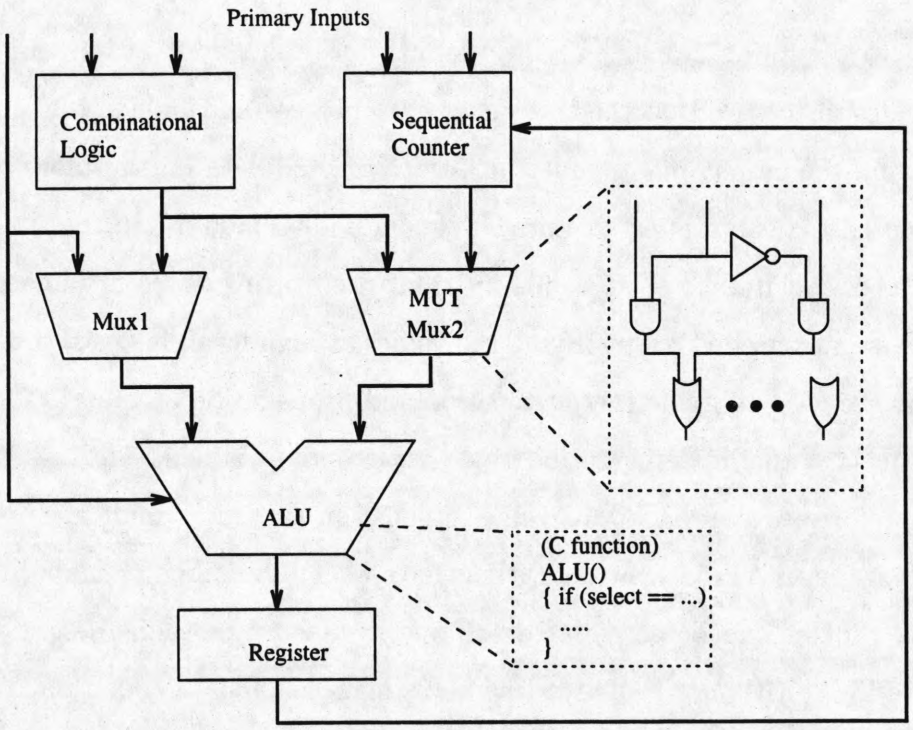

Figure 1: Architectural-level circuit.

The concept of architectural-level test generation is illustrated in Figure 1. The module with the gate-level structure, namely **Mux2**, is the module under test (MUT). Hierarchical test generation requires gate-level structure only of the MUT while all other modules' structure need not be present. Moreover, if the test vectors have been precomputed for the MUT, even the gate-level structure of the MUT needs not be present.

The vector that is either derived or precomputed for the MUT needs to be justified from the primary inputs and its corresponding fault effect propagated to the primary outputs at the architectural level. Value justification at the architectural level, however, is non-trivial and has not any standard forward or backward implication methods.

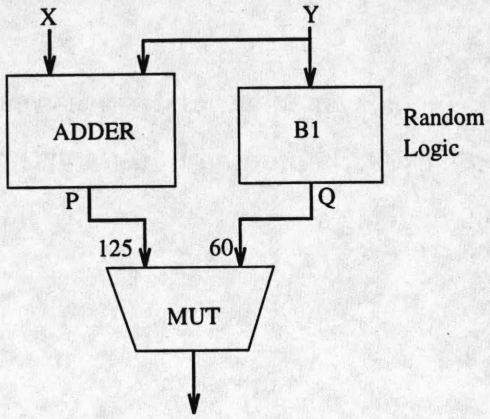

Figure 2: Example of value justification on a module.

Take the situation given in Figure 2 as a simple example. In order to justify the two values 125 and 60 at the inputs of the MUT, what values should the primary inputs **X** and **Y** take? For the ADDER to have 125 as the the output, many possible combinations may exist (ie. (**X**=50, **Y**=75), (**X**=80, **Y**=45), etc). Together with the global constraint that the output **Q**, which depends on **Y**, of block **B1** needs to be 60, the justification becomes more complicated due to data flow and data value conflicts [10].

Furthermore, unlike the simplicity of backward implication at the gate level, backward processing at the architectural level is much more cumbersome and complex. The user often has to hand-code the reverse behavioral function for the modules as in [10, 11]. Using the same example of the ADDER, the test generator may have to be given that **X = P - Y** and **Y = P - X** as its reverse functions to be able to derive appropriate input values for an output of 125. Similarly, the reverse function for block B1 would have to be coded. These reverse behavioral functions for the modules, especially those that involve state machines, are complex and hard to derive automatically, thus often avoided.

Therefore, the method that has often been proposed are the branch-and-bound techniques. They derive justification by reverse directional path search with forward behavioral function evaluations. Using the ADDER as an example again, in order to have an output of 125, random values are assigned to the inputs of the ADDER and hopefully to gradually converge to the desired value 125. The backtrack is continued through the predecessor modules until the primary inputs are reached. This method, however, does not overcome the problem of data flow and value conflicts described earlier.

In attempt to overcome these global constraints and expensive reverse behavioral functions, this paper proposes a *genetic algorithms* (GA) approach to the solution. A brief tutorial of GA will be presented in a later section. Before doing so, previous work in the area of high-level ATPG will be discussed.

3 Previous Work

[12, 13, 14, 15, 16] are some of the work that centered around branch-and-bound techniques. Brahme and Abraham [12] assumed a reverse device function already present for each module of the circuit to help backtracing through the circuit. Murray and Hayes [13] assumes the test vectors for the modules are precomputed and used the branch-and-bound D-Algorithm for value justification and propagation at the high level. Sarfert et. al. [14] implemented the non-trivial functions such as multiplexer, decoder, etc., but these functions still adopted single *bit-level* signals at the high level; thus speedup is limited. Kunda et. al. [15] took advantage of both high-level primitives and bit-vectors still using branch-and-bound approach. Anirudhan and Menon [16] derived symbolic constraints to help guide the test generation; however, these constraints are often computational prohibitive and computation becomes non-trivial as commented in [11].

Lee et. al. [10, 11], as briefly mentioned earlier, did not use the branch-and-bound technique; instead, an equation-solving technique using discrete relaxation Gauss-Siedal iterations is taken, similar to that of the SPICE technique. This technique, however, requires complex reverse behavioral functions and manually coded constraints for each module in the circuit prior to test generation.

For these previous work, the circuits were divided into data-path and control sections, and only the data-path portions were tested. In addition, they had an assumption that states represented by flip-flops are somehow extracted from the sequential circuit. However, these assumptions are not true in today's hierarchical design in general. Modules such as controllers have flip-flops embedded in them, which are difficult to be separated. ARGAJUS, the proposed algorithm, does not make these same assumptions; instead, it handles both combinational and sequential modules, in either data-path or control section of the circuit, during value justification.

Regarding the usage of genetic algorithms in the area of test generation, GA's have been used as a framework for test generation in [17, 18, 19]; however, [17] only handled gate-level combinational circuits while [18, 19] handled gate-level sequential circuits. No hierarchical information was explored in these work.

4 A Tutorial of Genetic Algorithms

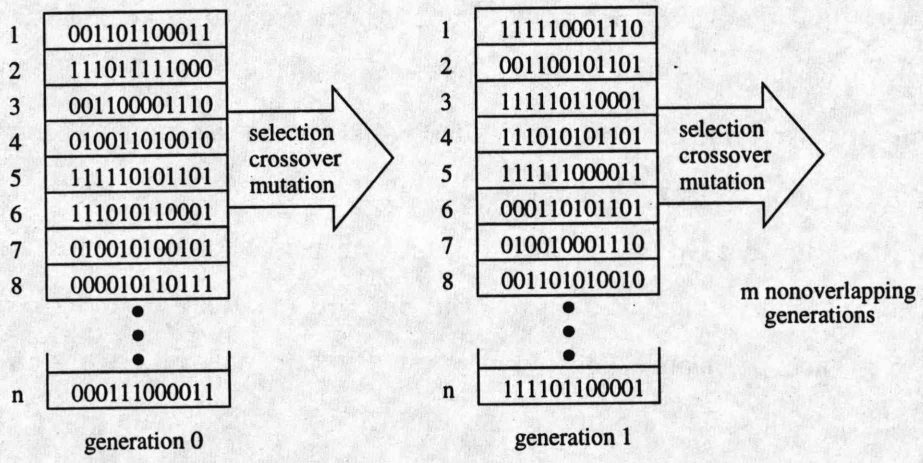

Figure 3: The simple genetic algorithm.

Figure 3 illustrates a simple example of genetic algorithm (GA). GA's are composed of a population of n strings, or *chromosomes*, and three evolutionary operators, namely *selection*, *crossover*, and *mutation* [20]. The population size n will vary with the chromosome length; the longer the length, the more diverse representation of chromosomes the population needs to be, hence resulting in a larger population. Each chromosome, or individual in the population, is an encoding of a solution to the problem, and it has an associated fitness value which depends on the nature of the problem.

The initial generation may be generated randomly or provided by the user. A new individual in the succeeding generation is evolved by *selecting* two individuals from the current generation, *crossing* the two individuals, and *mutating* random bits in the resulting individual with a given mutation probability. In a simple GA, distinct generations are evolved each time and each new generation involves repeated processes of selection, crossover, and mutation until all entries in this new generation are filled.

To guarantee monotonic increase in the fitness between successive generations, one has to make sure selection is to be biased toward more highly fit individuals. The evolutionary process ends when either the number of generations has exceeded a preset number m or the fitness has stopped to increase. Various selection schemes have been proposed [21, 22], but we will focus on the *binary tournament selection*. In binary tournament selection, two individuals are picked at random from the parent generation and the better-fit individual is selected. The two parents may or may not be replaced back into the parent population for the next selection.

Once two individuals are selected, crossover operator is used to produce two offsprings. Several crossover schemes have been studied, namely the one-point, two-point, and uniform crossovers. In one or two-point crossover, one or two chromosome positions are randomly chosen within the length of the chromosome, then the two parents are crossed at those points. For example, in a one-point crossover, if the position chosen was l, then the first offspring has an identical bit pattern up to position l as the first parent, and the rest of the bits are identical with those of the second. In uniform crossover, each position is crossed between the two parents with a probability, usually set to 0.5. As the two offspring are generated, mutation is done by flipping a random bit in the chromosome.

The fitness function is used to evaluate how highly fit an individual is. This function must be carefully chosen so that selection of individuals during the evolutionary process will guarantee a better-fit population of solutions in each successive generation. The fitness function is problem-dependent and is designed by the user.

5 The ARGAJUS Framework and Algorithm

Figure 4 depicts a scenario of justifying the MUT's input values as well as propagating the fault effect during one time frame of test generation. The circles in the figure represent other modules in the circuit.

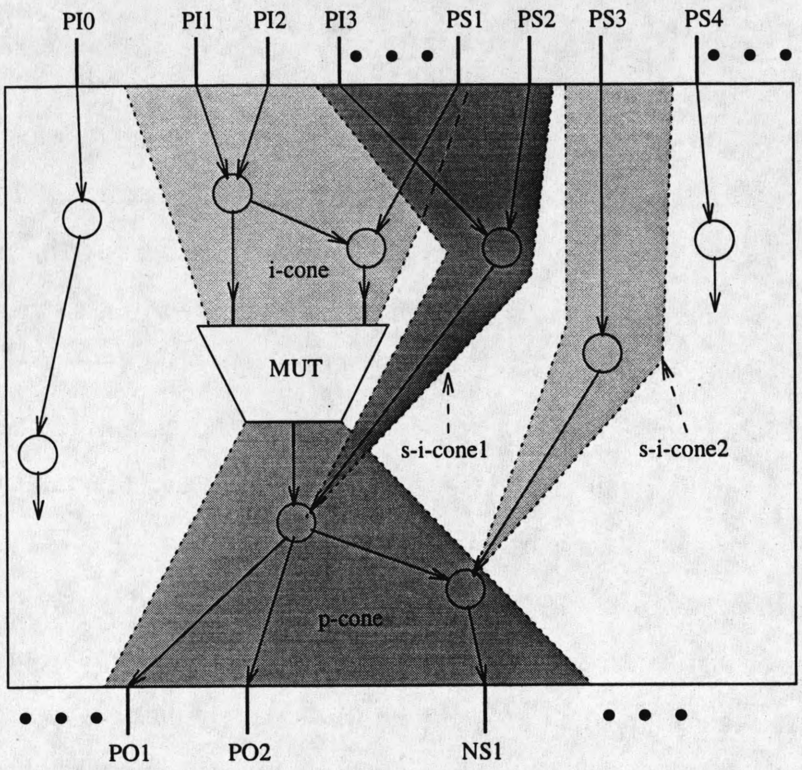

Figure 4: Value justification in one time frame.

Unlike traditional approaches of separating the excitation of the MUT's inputs and the propagation of the fault effect into two phases, ARGAJUS computes both the value excitation and fault effect propagation simultaneously in one single phase. This will save much computation because in traditional two-phase evaluations, if the fault effect is not propagated to a primary output or a state in the second phase, the excitation derived in the first phase is inapplicable and needs to be recomputed. So in order to avoid expensive, complex backward behavioral functions and backtracing algorithms, ARGAJUS proposes a forward-only justification algorithm involving GA and forward behavioral C functions for the modules.

As will be discussed in the following subsection, long individual string lengths will need a large population size. So in order to avoid long individual string lengths, only those primary inputs and state busses that directly influence the excitation and propagation are to be considered. Using Figure 4, the primary inputs and previous state busses that directly influence the inputs of the MUT, together with the corresponding paths form a cone, defined as the *influencing cone*, or *i-cone* for short. In this figure, the busses at the input of the i-cone are *PI1*, *PI2*, and previous state signal bus *PS1*. Note that *PI3* is not included in the i-cone. Similarly, the primary outputs and state busses that the fault effect may propagate to, along with their corresponding paths, form the *propagating cone*, or *p-cone*. Busses *PO1*, *PO2*, and next state signal *NS1* are at the boundary of the p-cone in this figure. There may be input paths to the modules in the p-cone which have not been included in the i-cone, thus *secondary* i-cones will be needed to assist the propagation of fault effects in the p-cone. Notice that there may be more than one secondary i-cone for the MUT. Paths formed from signals *PI3* and *PS2* make up s-i-cone1, and the path from *PS3* make up s-i-cone2. Using these cones, the individuals in the GA framework represent the values of the primary input and the previous state value in the current time frame.

ARGAJUS starts by taking a precomputed vector for the module under test, and this vector becomes the target for justification. A GA having a random initial population of test vectors is used to evolve succeeding generations, and an architectural-level simulator is used to simulate the circuit at high level for each test vector with the corresponding fitness computed. Evolution of generations of the candidate test vectors is repeated until either a justifying vector is found or if the fitness of the best individual has ceased to increase. Below is the pseudo-code for the ARGAJUS algorithm:

```
for each module in the circuit do
begin
        for each vector justification do
        begin
                while no justification is found and best fitness still increasing
                begin
                        evolve a new generation in GA;
                        update new fitness values for the new generation;
                end;
        end;
end;
```

Details on the parameters of the GA are described in the following subsections.

5.1 Population size, individual length, and number of generations

A diverse population is needed to represent individuals of a given length to escape local maximums. For instance, with an individual length of L, the population size n should be sufficiently large to represent a diverse mixture of individuals. Shown below is the relationship between individual length and population size used by ARGAJUS.

Individual length	Population size
less than 4	8
between 4 and 16	16
between 16 and 32	24
between 32 and 64	32
between 64 and 100	40
over 100	48

Because population size is directly proportional to the individual length, it is essential to limit the individual length to be as short as possible in order to have shorter run times without loss of accuracy. The length of individuals is bounded by the inputs to the i-cone and s-i-cones for the MUT as discussed in the previous section.

The bound on the number of generations in the case where the fitness value reaches a plateau depends on the depth of the circuit. The depth of the circuit is defined as the number of levels from the primary inputs and previous states to the primary outputs and the next states. The deeper the circuit, the more levels it needs to go through to excite the value and propagate the fault effect. This generally would involve a greater number, perhaps also more complicated, of paths before arriving at the MUT and propagating the fault to the output. More generations would be needed to converge to the target goal value in this case. GA stops when no more fitness improvement occurs in this bound of consecutive generations. The minimum bound on the number of generations without improvement is set to 8 and the maximum set to 64. If a solution is found in the initial, or the 0^{th}, generation, no more iterations need to be run.

5.2 Fitness function

Fitness function should be carefully chosen such that improvement between successive generations may be guaranteed. Fitness value, in the case of ARGAJUS, is composed of two parts: 1. justification of the excitation of the MUT inputs and 2. propagation of the fault effect. Starting with justifying the input excitation of the MUT: The individual vector is applied to the circuit and a value is arrived at the input of the MUT via the i-cone as illustrated in Figure 5. The i-cone of the MUT, consisting of modules described by their behavioral functions, may be represented theoretically by a function g. With this function, the input vector X of the MUT may be written as $X = g(Individual)$. If G denotes the target, or the objective, vector for the MUT's input, then the fitness function would evaluate how close the vector X is from G.

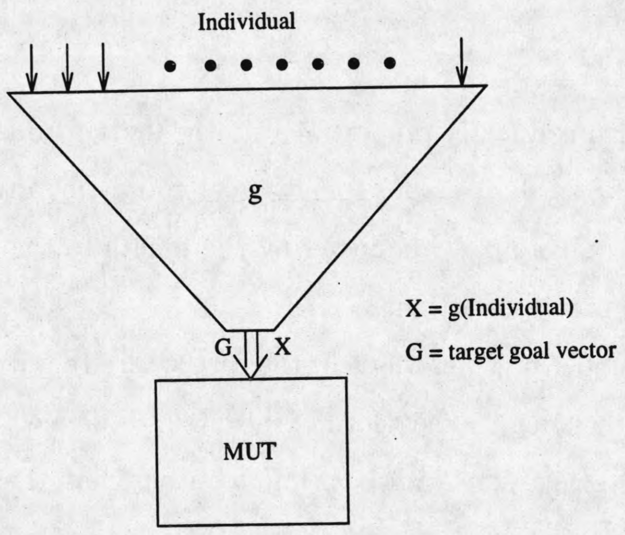

Figure 5: Computing fitness.

In the case that the function g is primarily arithmetic, it would make sense to have the fitness function related to the arithmetic distance:

$$fitness = 1 \, / \, (arithmetic\ distance + 1)$$

where *arithmetic distance* is defined as the *absolute value of the arithmetic difference* between G and X.

On the other hand, if the function g is primarily logical (ie. composed of *mux's*, *decoders*, *buffers*, etc.), it would make sense to have the fitness function measure the logical distance between G and X. Logical distance is the count of bit differences between two vectors. For instance, the vectors **101101** and **011001** have a distance of 3 because the two vectors differ in three positions, namely the first, second, and the fourth bit positions. Mathematically, the fitness function would be

$$\mathit{fitness} = L - d$$

where L is the length of an individual and d is the logical distance.

For general circuits, however, the modules that make up the function g are of both arithmetic and logical nature, composed of **ALU's**, **Mux's**, and **Decoders**. If either of the above fitness functions is taken to evaluate how well an individual fits the desired goal, the vector $X = g(\mathit{individual})$ will not approach the vector G monotonically. The reason behind the non-monotonicity in improvement is the lack of commonly matched patterns among selected individuals. The parent vectors may give similar fitness but match uncommon bits in the target vector G. Hence crossing uncommonly matched patterns would likely result in an individual that matches yet another set of bit positions in G.

The solution to a good fitness function is one which will maintain the commonly matched pattern as well as increasing the common pattern length from one generation to the next. One such solution is to use a weighted-sum of the matched bit positions that takes the advantages of both the arithmetic and logical distances. With this fitness function, more weight is given to the more significant bit positions in the vector as in the arithmetic difference approach, and one bit-position is evaluated at a time as in the logical difference method.

$$\mathit{excitation\_match} = \Big(\sum_{i=1}^{\mathit{total\ \#\ of\ bits}} i^{th}\ \mathit{bit\ value} * 2^{i} \Big) / \big(2^{\mathit{total\ \#\ of\ bits}} - 1\big)$$

The second and minor component of the fitness function in ARGAJUS is to measure how well the fault effect is propagated. In this case, more weight is given to fault effect propagated to the primary output than to those propagated only to a state flip flop. In addition, it doesn't make any difference if the fault effect was propagated to merely one PO, or had it propagated to more PO's, because detection at one primary output is sufficient to detect the fault. On the other hand, if the fault effect is not detected at a primary output, the more flip flops the fault effect reaches, the higher the probability that the fault will be detected in the next time frame.

propagation_match = (detection at PO) + (# of ff's reached) / (total # ff's in p-cone)

Between value excitation and fault effect propagation, more emphasis is given to value excitation because the precomputed inputs for the MUT must be justified. Thus the combined fitness function used in ARGAJUS becomes

*fitness = 10 * excitation_match + propagation_match.*

5.3 Other GA parameters

The parameters used for selection and crossover are tournament selection and uniform crossover. Tournament selection, as explained earlier, selects the better individual of two randomly picked individuals. This selection scheme has shown to perform better than other selection schemes [19]. The various crossover schemes, on the other hand, have not shown significant difference in performance, thus uniform crossover was chosen for this work. The mutation rate is chosen to be the inverse of the individual length such that it is also less than $1/n$, the inverse of the population size. It is important not to mutate more than 1 bit in an individual string, so to preserve the heredity from the parents. It is also important to keep the number of mutants in a population low, preferably less than one individual per population. Small mutation rates encourage exploitation in the current search space; however, higher mutation rates would encourage exploration across search spaces.

6 Experiments

Figure 6 shows the architectural block diagrams of three of the five circuits used for the experiment; and the architectural composition of all five circuits are shown in Table 1. Three of the circuits, namely Am2910, Mp1, and Div, are of both control and data-path structures, while PCONT2 and PIIR8 are data-path dominant digital filter circuits. Am2910 is a microprogram sequence controller; Mp1 is a 16-bit two's-complement multiplier using shift-store; Div is a 16-bit two's-complement divider; PCONT2 and PIIR8 are both 8-bit digital filters. All five circuits are full-scan circuits.

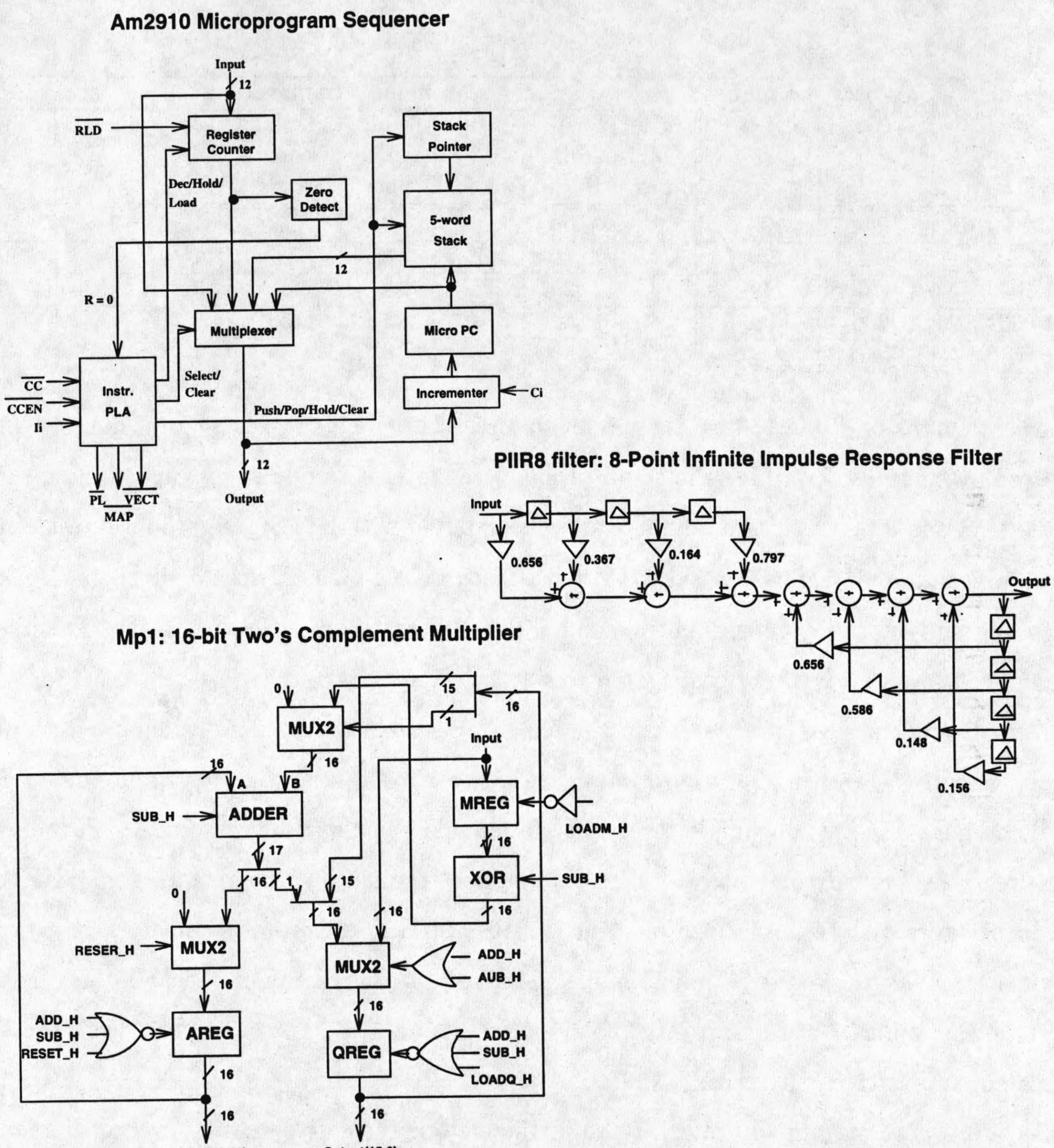

Figure 6: Block diagrams of three architectural circuits

However, the flip flops are not extracted from the sequential modules during value justification to preserve the sequential nature of the circuits.

Table 1: Architectural-level circuit information

circuit	# gates	# faults	module composition							
			8,16-bit mux	8,16-bit buffer	8-bit adder	8-bit multiplier	random comb logic	reg	counter	random seq logic
Am2910	1054	2391	1	0	1	0	3	1	1	3
Mp1	754	1708	4	1	1	0	3	1	0	1
Div	973	2147	3	2	1	0	5	3	0	1
PCONT2	4290	11300	3	4	8	2	5	3	0	0
PIIR8	11437	29801	7	9	7	8	4	7	0	0

The columns of Table 1 show the circuit name, total number of gates, total number of faults, followed by columns describing the composition of the internal modules for each circuit.

Test vectors for each internal module are precomputed with HITEC, a gate-level test generator. These precomputed vectors are fed to ARGAJUS to be justified at the architectural level. Each precomputed vector may detect multiple faults, thus different fault effects may arise. Different fault effects need to be successfully propagated to verify detection of each fault.

Not all precomputed vectors are justifiable. Unjustifiable vectors are different from hard-to-justify vectors. Unjustifiable vectors may be due to inexcitable inputs or unpropagatable fault effects. Figure 7 shows two simple cases where a module input may be inexcitable. In the case of a fault-free *Decoder*, only one bit can take the value 1 at the output. Therefore, if the MUT's input requires vectors with two or more 1 bits, such as **1001**, **0101**, etc. from the output of the preceding Decoder, this vector would be unjustifiable. Similarly, if the inputs of the MUT are fed by the output of a *Control Unit*, a vector would be unjustifiable if the vector is not one of the Control Unit's possible output patterns.

Secondly, if the inputs of the MUT are justifiable but the corresponding fault effect cannot be propagated due to conflicts with the influencing cone value assignments, no justification would be found either.

The results will focus on all the justifiable precomputed vectors. Two runs per circuit were performed using two separate random seeds. Tables 2 to 6 show the results of the number of

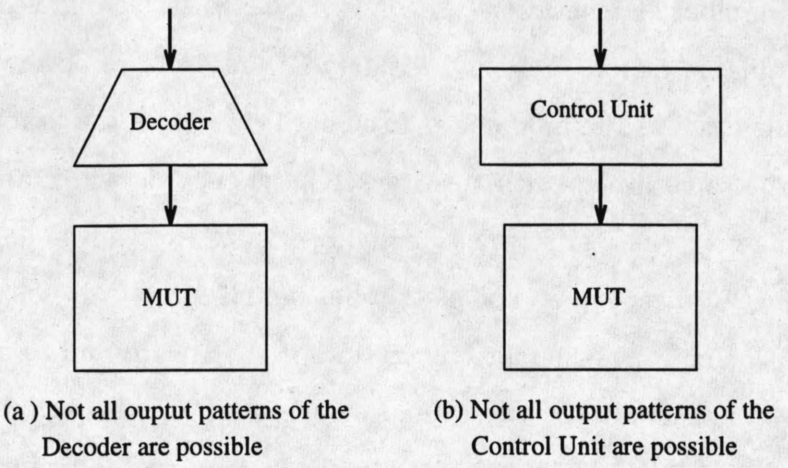
Decoder
MUT
Control Unit
MUT
(a) Not all ouptut patterns of the Decoder are possible
(b) Not all output patterns of the Control Unit are possible

Figure 7: Examples of unjustifiable values.

vectors justified and the number of generations used to justify them for each module n the five circuits. The columns in the tables display the lengths of the individuals in the GA, the i-cone depth, the number of collapsed faults in the module, the number of vectors generated by the gate-level generator HITEC, the corresponding number of vectors justified by ARGAJUS, followed by the maximum and average numbers of GA generations in the two runs of ARGAJUS. The length of the individual is the number of primary inputs and flip-flops in the primary and secondary i-cones. The i-cone depth shows only the depth of the primary i-cone.

As indicated by the results, most of the vectors in the three non-filter circuits are justifiable. In some modules the justification is so easy in which a successful vector is found in the initial randomly produced generation for the precomputed vector; these are the modules which have both the maximum and average number of generations equal to 0. In the filter circuits PCONT2 and PIIR8, however, some modules have many unjustifiable precomputed vectors. For the multiplier modules in these filter circuits, one of the two inputs is always supplied by a constant buffer as shown in the block diagram of Pcont2 in Figure 6. This architectural information, however, is blind to the gate-level generator during the precomputation of the vectors for it. As a result, the precomputed vectors become unjustifiable at high-level due to these constant inputs. These constants in the circuits may influence other modules in the circuit to have unjustifiable precomputed vectors as well.

In terms of the number of generations ARGAJUS's GA takes to find a justification, the results show that most justifiable vectors are found within a few generations on the average. In the circuit Am2910, the module Stack is the module with the hardest vectors to justify because it has a deep i-cone and shares the same inputs with the Stack Pointer module which are fed from the Control Unit. Many conflicts may, then, arise when justifying a vector for this module, thus requiring more generations for the GA to converge. For most other modules in all five circuits, the trend is that the deeper the influencing cone, the more generations it requires to find a solution. Nevertheless, a solution is found in a few number of generations on the average. This is different from the branch-and-bound approach in which its backtracking algorithm may involve exploration of many different paths.

Table 2: Number of generations for Am2910

module name	indiv. length	i-cone depth	# faults	# HITEC vectors	# vectors just.	# GA generations			
						first run		second run	
						max	average	max	average
Reg Cntr	95	3	402	120	120	111.	4.31	53	4.03
Zero Det	107	3	17	34	34	19	10.24	19	10.59
Mux	107	6	294	44	40	256	18.57	176	16.97
Incr	107	6	232	42	42	30	1.95	31	1.91
PC	95	6	48	10	10	35	2.69	37	3.53
Stack Ptr	104	3	294	54	49	26	4.06	26	4.00
Stack	47	6	1008	91	55	114	30.31	104	30.13
Instr PLA	107	3	198	58	54	64	8.58	56	8.39
buffer1	31	4	6	6	5	0	0.00	0	0.00
buffer2	107	3	14	6	4	0	0.00	1	0.50

7 Conclusion

ARGAJUS, a new approach to value justification at the architectural level using behavioral functions and genetic algorithms has been presented. Complicated and cumbersome reverse functions for the modules are unnecessary and no backtracing algorithms are used. Instead, genetic algorithms with good fitness functions are explored to solve the value-justification problem at high-level. Results indicated that convergence of value justification and fault effect propagation in the GA framework to be very promising. ARGAJUS can easily be adapted in an architectural-level ATPG environment

Table 3: Number of generations for Mp1

module name	indiv. length	i-cone depth	# faults	# HITEC vectors	# vectors just.	# GA generations			
						first run		second run	
						max	average	max	average
MREG	57	1	64	8	8	44	2.78	151	5.24
AREG	57	7	64	8	4	77	2.41	93	5.20
QREG	57	8	64	8	6	71	3.67	65	3.07
XOR	73	2	226	12	12	136	10.25	146	10.25
Pass1	73	7	66	14	14	138	21.68	137	17.25
Pass2	73	4	66	14	14	93	6.38	116	7.37
ADDER	73	4	652	28	20	159	1.72	100	1.36
MR_Mux	41	3	130	20	19	149	10.62	143	11.82
AR_Mux	73	7	130	20	8	158	8.05	125	7.35
QR_Mux	73	8	130	20	20	129	8.36	142	8.85
Q_in_mux	73	4	130	20	20	146	8.47	113	7.82
Cntrl unit	66	8	230	75	75	32	4.83	54	5.17
Join	73	5	32	8	8	30	3.13	24	2.88
Invert	73	2	2	2	1	23	17.50	21	15.50

and easily extendible to multi-frame justification problem for justifying sequences of vectors.

Table 4: Number of generations for Div

module name	indiv. length	i-cone depth	# faults	# HITEC vectors	# vectors just.	# GA generations first run max	first run average	second run max	second run average
Alu	67	2	952	86	19	57	0.74	67	0.91
Cntrl unit	81	0	52	22	18	0	0.00	0	0.00
Cmp	67	4	422	114	101	121	13.09	109	13.73
Incr	19	1	311	62	52	97	3.44	74	3.41
Mux1	67	3	130	20	20	107	8.70	89	9.02
Mux2	67	3	130	20	20	123	10.70	113	10.60
Pass1	19	2	66	14	13	65	5.55	60	6.25
Pass2	19	1	66	14	12	54	4.81	71	4.93
Invert	19	4	2	2	1	17	15.50	22	16.50
RegA	67	4	64	8	8	28	1.70	27	1.66
RegB	67	4	64	8	8	23	1.28	20	1.27
RegC	51	4	64	8	0	-	-	-	-
Zero det	67	2	22	44	44	30	22.35	42	22.95
buffer1	67	0	20	4	2	0	0.00	0	0.00
buffer2	67	0	14	6	3	0	0.00	0	0.00

Table 5: Number of generations for PCONT2

module name	indiv. length	i-cone depth	# faults	# HITEC vectors	# vectors just.	# GA generations			
						first run		second run	
						max	average	max	average
nz1	33	1	2	2	1	0	0.00	0	0.00
nz2	33	2	4	4	2	0	0.00	0	0.00
nz3	33	2	4	4	2	0	0.00	0	0.00
nz4	8	3	2	2	1	0	0.00	0	0.00
nz5	33	3	2	2	1	0	0.00	0	0.00
reg1	25	9	32	8	8	12	0.63	15	0.94
reg2	25	9	32	8	8	15	1.53	21	1.06
reg3	25	9	32	8	8	39	3.25	34	1.47
mult1,2	33	4	3143	142	0	-	-	-	-
add1	33	8	633	40	9	62	1.76	68	2.03
add2	33	3	633	40	10	62	5.72	56	5.53
add3	33	4	633	40	14	16	0.10	16	0.14
add4	33	5	633	40	0	-	-	-	-
add5	33	7	633	40	11	63	2.11	48	1.16
add6	33	8	633	40	0	-	-	-	-
add7	33	6	633	40	1	10	10.00	7	7.00
add8	33	7	633	40	12	95	5.73	120	7.00
mux1-3	33	6	66	18	0	-	-	-	-
buffer1	33	9	16	8	8	8	1.38	13	1.31
const1	33	3	16	6	2	0	0.00	0	0.00
const2	33	3	16	6	2	0	0.00	0	0.00
z8	33	3	16	6	3	22	22.00	22	22.00

Table 6: Number of generations for PIIR8

module name	indiv. length	i-cone depth	# faults	# HITEC vectors	# vectors just.	# GA generations first run max	first run average	second run max	second run average
nz1	65	1	2	2	1	0	0.00	0	0.00
nz2	65	2	4	4	2	0	0.00	0	0.00
nz3	65	2	4	4	2	0	0.00	0	0.00
nz4	65	3	2	2	1	0	0.00	0	0.00
reg1	57	12	32	8	8	34	9.09	47	9.63
reg2	57	12	32	8	8	37	11.09	32	10.97
reg3	57	12	32	8	8	34	2.09	32	2.09
reg4	57	12	32	8	8	40	4.53	62	7.43
reg5	57	12	32	8	8	28	2.31	28	2.31
reg6	57	12	32	8	8	34	10.84	36	11.22
reg7	41	12	32	8	8	34	10.94	38	11.00
mult1-6	65	4	3143	142	0	-	-	-	-
mult7	65	4	3143	142	10	1	0.06	2	0.13
add1,2	65	5	633	40	0	-	-	-	-
add3	65	7	633	40	6	52	0.65	35	0.44
add4	65	8	633	40	6	9	0.19	30	0.53
add5	65	9	633	40	7	31	0.65	50	0.93
add6,7	65	10	633	40	0	-	-	-	-
mux1-7	17	4	66	18	0	-	-	-	-
buffer1	65	12	16	8	8	5	0.75	4	0.75
const1	65	3	16	6	1	0	0.00	0	0.00
const2	65	3	16	6	2	0	0.00	0	0.00
const3	65	3	16	6	2	0	0.00	0	0.00
const4	65	3	16	6	3	0	0.00	0	0.00
const5	65	3	16	6	2	0	0.00	0	0.00
const6,7	65	3	16	6	0	-	-	-	-
z8	65	3	52	16	8	28	28.00	28	28.00

References

[1] P. Goel, "An implicit enumeration algorithm to generate tests for combinational circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215-222, Mar. 1981.

[2] J.P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM J. Res. Develop*, vol. 10, pp. 278-281, 1966.

[3] H-K.T. Ma, S. Devadas, A.R. Newton, and A. Sangiovanni-Vincentelli, "Test generation for the sequential circuits," *IEEE Trans. Computer-Aided Design*, pp. 1081-1093, Oct. 1988.

[4] W.T. Cheng, "The back algorithm for sequential test generation," *Proc. IEEE Int. Conf. on Computer Design*, pp. 66-69, Oct. 1988.

[5] A. Ghosh, S. Devadas, and A.R. Newton, "Test generation for highly sequential circuits," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 362-365, Nov. 1989.

[6] T. Niermann and J.H. Patel, "HITEC: A test generation package for sequential circuits," *Proc. European Design Automation Conf.*, Feb. 1991.

[7] S.J. Chandra and J.H. Patel, "A hierarchical approach to test vector generation," *Proc. 24th IEEE/ACM Design Automation Conf.*, pp. 495-501, June 1987.

[8] J.D. Calhoun and F. Brgles, "A framework and method for hierarchical test generation," *Proc. Int. Test Conf.*, pp. 480-490, Sept. 1989.

[9] D. Bhattacharya and J.P. Hayes, "A hierarchical test generation methodology for digital circuits," *J. Elect. Testing: Theory and Appl.*, vol. 1, pp.103-123, 1990.

[10] J. Lee and J.H. Patel, "A signal-driven discrete relaxation technique for architectural level test generation," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 458-461, Nov. 1991.

[11] J. Lee and J.H. Patel, "Hierarchical test generation under intensive global functional constraints," *Proc. 29th IEEE/ACM Design Automation Conf.* pp. 261-266, 1992.

[12] D. Brahme and J. Abraham, "Knowledge based test generation for VLSI circuits," *Intl. Conference on Computer Aided Design*, pp. 292-295, 1987.

[13] B. Murray and J. Hayes, "Hierarchical test generation using precomputed tests for modules," *Intl. Test Conference*, pp. 221-229, 1988.

[14] T.M. Sarfert, R. Markgraf, E. Trischler, and M.H. Schulz, "Hierarchical test pattern generation based on high-level primitives," *Proc. Intl. Test Conference*, pp. 470-479, Sept. 1989.

[15] R.P. Kunda, P. Narain, J.A. Abraham, and B.D. Rathi, "Speedup of test generation using high-level primitives," *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 580-586, June 1990

[16] P.N. Anirudhan and P.R. Menon, “Symbolic test generation for hierarchically modeled digital systems,” *Intl. Test Conference*, pp. 461-469, Sept. 1989.

[17] M. Srinivas and L.M. Patnaik, “A simulation-based test generation scheme using genetic algorithms,” *Proc. Int. Conf. VLSI Design*, pp. 132-135, 1993.

[18] D.G. Saab, Y.G. Saab, and J.A. Abraham, “CRIS: A test cultivation program for sequential VLSI circuits,” *Proc. Int. Conf. Computer-Aided Design*, pp. 216-219, 1992.

[19] E.M. Rudnick, J.H. Patel, G.S. Greestein, and T.M. Niermann, “Sequential circuit test generation in a genetic algorithm framework,” *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 698-704, 1994.

[20] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.

[21] D.E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms,” in *Foundations of Genetic Algorithms,* G. Rawlins, Morgan Kaufmann, pp.69-93, San Mateo, CA, 1991.

[22] J.E. Baker, “Reducing bias and inefficiency in the selection algorithm,” *Proc. Second Int. Cont. Genetic Algorithms*, pp. 14-21, 1987.