Techniques for Functional Test Pattern Execution

Abstract

Functional debugging of application specific integrated circuits (ASICs) has been recognized as a very labor-intensive and expensive process. It often dominates the time and cost of the ASIC system development. The difficulty of functional debugging is mainly due to the limited controllability and observability of the storage elements in designs, and therefore the intermediate variables in functional specifications.

Debugging process can be logically divided into five phases: functional test generation, functional test execution, error detection, error diagnosis, and error correction. We propose a new approach for the functional test pattern execution phase. The goal is to maximize the simultaneous controllability of an arbitrary set of the user selected variables in the design at the debugging time for facilitating the functional test pattern execution while minimizing the hardware overhead.

The new approach is based on the divide and conquer optimization paradigm. Functional specification of the design is logically partitioned into linear and nonlinear components. The full controllability of each component is achieved by exploiting resource sharing of a set of variables in the components and primary inputs. The most important property of the selected variables is that they enable direct and complete answers on all possible questions related to simultaneous controllability of an arbitrary set of variables in the design. The selection of the variables introduces an interesting combinatorial optimization problem. We establish its computational complexity and propose a non-greedy heuristic algorithm. The approach imposes minimal restriction on register sharing so that the synthesized designs will have the desired characteristic while minimizing the additional hardware overhead and minimizing the disruption of the optimization potential when scheduling, allocation and binding tasks in high-level synthesis are performed. The effectiveness of the proposed approach is demonstrated on a number of designs.

1 Introduction

1.1 Motivation

Functional debugging of hardware and software systems has been recognized as a very labor-intensive and expensive process. For example, the designers of a modern superscalar microprocessor reported that the process took more than 40% of the overall development time [26]. Although the functional debugging has been established as one of the crucial system development processes and tremendous research efforts have been devoted to the topic, especially in software compiler, the exceptional conceptual complexity of the debugging process has prevented the process from being automated.

Functional debugging of ASIC designs is in particular a difficult activity and until now only limited effort has been spent on the subject. This is mainly due to the limited controllability and observability of the storage elements or variables in designs. This situation is likely to become more serious in the future since the key technological trends indicate that the percentage of controllable and observable variables in designs will steadily decrease. There is, however, a key technological factor, the clock rate, likely to help remedy the situation. The clock rate has been doubling every three years. Thus, higher levels of resource sharing become feasible and economically desirable with each new generation of technology, while at the same time resource sharing is becoming increasingly important for satisfying applications requirements.

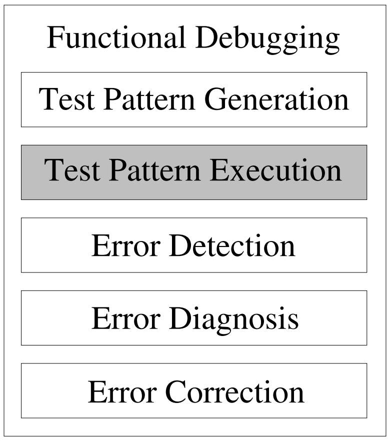


Figure 1. The functional debugging process for ASIC designs

The functional debugging process for ASIC designs can be partitioned to the following five debugging phases as illustrated in Figure 1.

Test Pattern Generation. In the first phase of debugging, the goal is to generate the input patterns which are likely to make functional errors visible. These patterns are given as the specific values of a set of variables in the design.

Test Pattern Execution. In this phase the functional test input patterns generated in the previous phase should be executed. The test input patterns are executed by providing the proper values to the primary inputs.

The goal is to execute the given test pattern whenever possible.

Error Detection. In this phase the designer discovers an error of the design by the discrepancy between the actual values and the desired values of the primary outputs for the given values of primary inputs.

Error Diagnosis. In this phase the designer identifies the part of the design which causes the error detected in the previous phase.

Error Correction. In the final phase the faulty part of the design identified in the previous phase is replaced by the correct one.

In this paper, we address the test pattern execution (FTPE) phase of the functional debugging of ASIC designs in high level synthesis. We have the following four main objectives for the research presented in this paper:

1. to understand the key aspects of the problem and develop sound theoretical foundation.
2. to understand the role of the functional test pattern generation and execution in design process and the interaction with other synthesis tasks such as scheduling, allocation and binding.
3. to develop techniques for design-for-debugging which facilitates the test pattern execution phase while minimizing its overhead.
4. to define the optimization problems involved in the process, establish their computational complexity, and propose efficient algorithms.

1.2 Motivational Example

To illustrate the key ideas behind the new approach, consider the design shown in Figure 2(a). The design is a cascade form Avenhaus filter composed of direct-form II sections designed by Crochiere and Oppenheim [6]. For the simplicity of illustration, we assume that all operations take one control step. The critical path is six control steps. It is simple to verify that the minimum hardware requirement is two adders and two multipliers.

To debug the functionality of the design, in the first step, functional test patterns, which are most likely to make functional errors visible, are generated. These patterns are given as the specific values of a set of variables in the design that need to be set in the same iteration. For example, the variables a and b in Figure 2 should be set to the maximum possible value in order to detect some possible error. The functional test patterns should be executed by providing the proper values to the primary inputs. The goal is to maximize the simultaneous controllability of an arbitrary set of the user selected variables in the design at the debugging time for facilitating the functional test pattern execution while minimizing the hardware overhead.

It is hard to execute functional test patterns in the original design due to several reasons. First, there is only one primary input while there are four functional delays and many intermediate variables, which results in low direct controllability of the design. Secondly, there are four loops in the design due to its sequentiality. It is necessary that at least one of the variables in a loop is controllable in order to control variables in the loop.

To overcome the difficulties, the key design element to be explored is resource sharing, especially register sharing between primary inputs and intermediate variables. Our approach is to impose minimal restriction on register sharing so that the synthesized design will have the desired characteristic while minimizing the additional hardware overhead and minimizing the disruption of the optimization potential when performing scheduling, allocation and binding tasks in high-level synthesis. In order to explain the proposed approach, we need to introduce the notion of a complete cut of a computation.

Definition 1 (Cut) *A cut is a subset of variables in control data flow graph. A variable in a cut is called cut variable.*

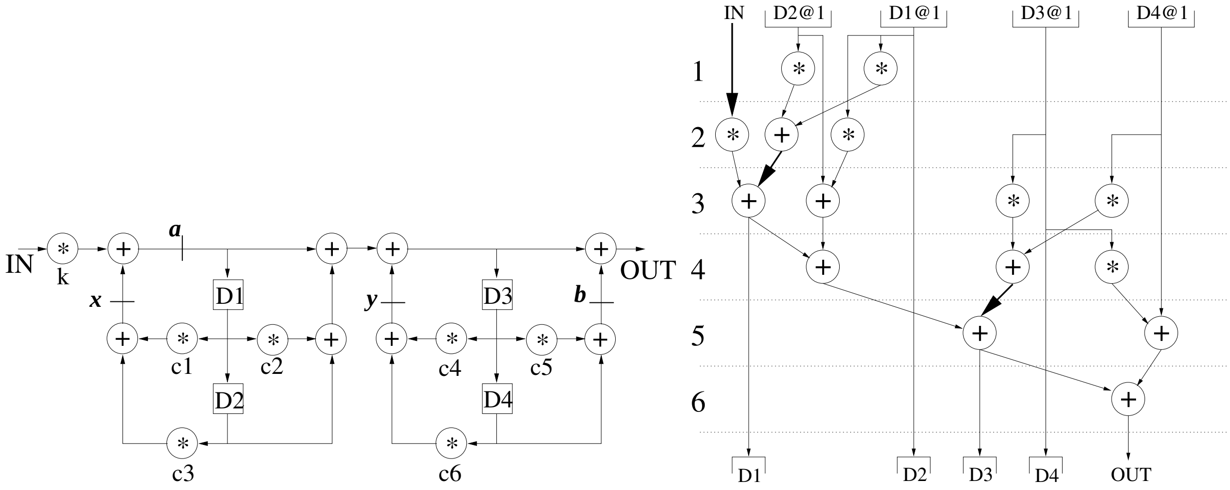
Definition 2 (Complete Cut) *A complete cut is a subset of variables which includes at least one variable from every loop in control data flow graph. Any incomplete cut is called partial cut.*

A complete cut bisects all loops in the control data flow graph. For example, all functional delays or state variables form a complete cut. A complete cut plays an important role to overcome the difficulties mentioned above. Providing controllability to the variables in a complete cut allows us to isolate the consecutive iterations. By sharing registers between the primary inputs and the variables in the complete cut such that every cut variable shares a register with one of the primary input variables, direct controllability to the cut variables is provided. Based on this idea, our approach selects a complete cut and imposes such a restriction on scheduling, allocation and binding phases that every cut variable should share a register with one of the primary input variables. Therefore, the complete cut must be selected to impose minimal restriction on the synthesis phases.

Consider a complete cut composed by all four state variables. Since all variables in the cut and the primary input **IN** are concurrently alive in control step 1 regardless of scheduling algorithm to be used, they have to be stored in five different registers. Therefore, the direct of the cut variables is not provided by the primary input **IN**. Allocation of extra hardware, four sets of register-I/O connections with write ability to the registers for the state variables can provide direct controllability. Without the extra hardware, it is impossible or very hard at best to set arbitrary requested values in the functional delays. Providing the ability of resetting the functional delays to a fixed value, e.g., 0, may reduce the complexity, but the resetting has a serious drawback of interrupting functionality of the computation.

Consider a different complete cut consisting of the variables x and y (the bold lines in Figure 2(b)). In this case, all the variables in the cut and the primary input **IN** can share a register because they can be scheduled such that they are not alive simultaneously. The two phase clocking scheme in which read operations are performed in the first phase and write operations are performed in the second phase is assumed. One such schedule is shown in Figure 2(b). The schedule requires only two adders and two multipliers which are exactly the lower bound.

Now the complete cut can be used to control other variables in the design. Any other variable can be described by a linear equation of the variables in the complete cut and the primary inputs. Thus, by solving



(a) Control data flow graph

(b) Scheduled CDFG

Figure 2. Motivational example: cascade-form Avenhaus filter

a system of linear equations for the requested set of variables to control, the feasibility of the request can be determined. For example, the system of equations for the variables a and b at the iteration t is the following:

$$a[t] = k * IN[t] + x[t]$$

$$b[t] = c_5 * D_3[t] + D_4[t] = c_5 * D_3[t] + D_3[t-1]$$

$$D_3[t] = y[t-1] + k * IN[t-1] + x[t-1] + c_2 * D_1[t-1] + D_2[t-1]$$

$$D_2[t] = D_1[t-1]$$

$$D_1[t] = k * IN[t-1] + x[t-1]$$

$a[t]$ and $b[t]$ are described by linear combinations of $IN[s], x[s], y[s]$ for $s = t, t-1, t-2, t-3, t-4$.

We stress here that our technique is not limited to linear designs. For nonlinear designs, a technique, which isolates all output variables of nonlinear operations through register sharing with the primary input variables, is employed.

1.3 What is New?

To the best of our knowledge this is the first attempt to study the functional test pattern execution problem in high level synthesis and in general CAD techniques. This is also the first approach which addresses the functional test pattern execution problem for the custom ASIC designs in an optimization-intensive way.

1.4 Paper Organization

The rest of the paper is organized in the following way. In the next section we introduce several definitions and present the necessary background material which includes the assumed computational and hardware models, resource sharing and linear algebra. Section 3 outlines the previous work in software debugging, hardware verification, debugging-related CAD, and register sharing in high level synthesis. The proposed approach is described in Section 4. In Section 5, the optimization problem for minimizing the hardware overhead in achieving test pattern execution requirement is defined, its computational complexity is established, and an efficient algorithm is proposed for the problem. In Section 6, we present experimental results. Finally, we draw conclusions in Section 7.

2 Preliminaries

2.1 Definitions

In this subsection, we define several terminologies used in this paper.

Definition 3 (Linear Computation) *A linear computation is a computation which satisfies two axiomatic properties, homogeneity and additivity* [28].

- *Homogeneity: If the response to a signal x is a signal y, then the response to ax is ay for arbitrary constant a.*
- *Additivity: If the response to signals x_1 and x_2 are y_1 and y_2, respectively, then the response to $x_1 + x_2$ is $y_1 + y_2$.*

A computation which uses only addition (subtraction) and multiplication with constant as computational operators is a linear computation. It is important to note that linearity can be exhibited over a different set of operators. A computation with either *min* or *max* in place of addition, and addition in place of multiplication with constant is a linear computation. Any computation which is not a linear computation is a nonlinear computation. Typical nonlinear operators used in CDFG include variable multiplication and variable division.

Definition 4 (Irredundant Complete Cut) *An irredundant complete cut is a complete cut in which removal of any variable in the cut results in an incomplete cut. Any other complete cut is called redundant complete cut.*

Definition 5 (Partially Redundant Complete Cut) *A partially redundant complete cut is a complete cut in which removal of any variable in the cut, originating from a linear operation, results in an incomplete cut.*

A partially redundant complete cut is simply an irredundant complete cut for a linear design.

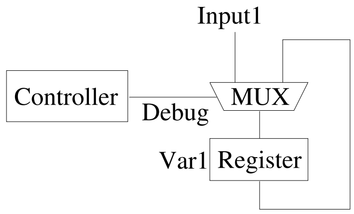


Figure 3. Necessary hardware for controlling variables in the “debug” mode

2.2 Computational and Hardware Model

We represent a computation by a hierarchical control data flow graph (CDFG) consisting of nodes representing data operators or sub-graphs, and edges representing the data, control, and timing precedences [34]. The computations operate on periodic semi-infinite streams of inputs to produce semi-infinite streams of outputs. The underlying computational model is homogeneous synchronous data flow model [19] which is widely used in computationally intensive applications such as image and video processing, multimedia, speech and audio processing, control, and communications. Under this model, the operators consume a single input from each input and produce a single sample on each output on every execution. An important implication of the semantics of the selected computational model is static compile-time scheduling and assignment.

We do not impose any restriction on the interconnect scheme of the assumed hardware model at the register-transfer level. Registers may or may not be grouped in register files. Each hardware resource can be connected in an arbitrary way to another hardware resource. We assume that the two phase clocking scheme in which read operations are performed in the first phase and write operations are performed in the second phase is used. It is important to note that all techniques and algorithms are directly applicable to other hardware models.

The initial design is augmented with the additional hardwares which enable controllability in the “debugging” mode. The following input operation is incorporated to provide complete controllability of a variable *Var1* using user specified input variable *Input1*: *if(Debug) then Var1 = Input1*. This statement can be implemented using a multiplexer and some control logic as illustrated in Figure 3.

2.3 Resource Sharing

Resource sharing methods in high level synthesis can be divided into *unconditional* and *conditional* [21]. In unconditional resource sharing, either functional modules for operations scheduled in different loops or registers for variables without conflicting lifetimes can be shared. In conditional resource sharing, modules for operations and registers for variables at mutually exclusive parts of conditional branches can be shared.

Values of the variables which are generated in one control step and used in a later control step must be stored in registers. In order to minimize the number of registers used, we use register sharing which allows different variables to share the same registers. The lifetime of a variable is the time period in which the value of the variable must be saved in a register. More formally, it is defined as the following:

Definition 6 (Lifetime of a Variable) *The lifetime of a variable spans between the control step at which the value is first computed and the control step at which all variables dependent on its value have been computed.*

Two variables can share a register if their lifetimes do not overlap.

Definition 7 (Compatible Variables) *If the lifetimes of variables x and y do not overlap, then the two variables x and y are compatible variables. Otherwise they are incompatible variables.*

2.4 System of Linear Equations

A system of linear equations with n unknown variables and m equations is considered.

$$\begin{array}{ccc} a_{11}x_1+ & \cdots & +a_{1n}x_n = b_1 \\ & \cdots & \\ a_{m1}x_1+ & \cdots & +a_{mn}x_n = b_m \end{array} \tag{1}$$

Equations (1) can be rewritten in terms of the column vectors of the matrix $A = (a_{ij})$.

$$x_1A^1 + \cdots + x_nA^n = b \tag{2}$$

The equation (2) can be rewritten as $Ax = b$.

Definition 8 (Rank of a Matrix) *Let A be a $m \times n$ matrix. The row rank of A is equal to the number of linear independent rows of A. The column rank of A is equal to the number of linear independent columns of A. The rank of A, Rank(A), is either the row rank or column rank of A because the row rank of a matrix is the same as the column rank of the matrix.*

The known results about a system of equations $Ax = b$ and the augmented matrix (A, b), where A is a $m \times n$ matrix, are summarized in the following [38]:

- If Rank$(A, b) >$ Rank(A), then $Ax = b$ has no solution.
- If Rank$(A, b) =$ Rank$(A) = n$, then there exists a unique solution to $Ax = b$.
- If Rank$(A, b) =$ Rank$(A) < n$, then there exist infinitely many solutions to $Ax = b$.

3 Related Work

We survey the related works along four lines: software debugging, hardware verification, debugging-related CAD, and register sharing in high level synthesis. In software debugging, the related works on test data generation and execution are surveyed. In hardware debugging, we survey the related research effort on functional test data and program generation and execution for functional verification and validation.

Test data generation in software debugging is the process of identifying a set of test data that satisfies a selected testing criterion, such as statement coverage and branch coverage, which requires that certain program elements be evaluated. There are three types of test data generators: *random* test data generators [4], *path-oriented* test data generators [7], and *dynamic* test data generators [16]. The random test data generators often suffer inefficiency when finding test data to execute selected statements. The path-oriented approach consists of the process of selecting a program path to the selected statement and then generating input data for that path. If the test input data are not found for the selected path, a different path is selected. One of the drawbacks with this approach is that infeasible paths are often selected and as a result, significant computational effort can be wasted in analyzing those paths. In the dynamic approach of test data generation, test data are derived based on the actual execution of the program under test. The approach starts by executing a program with random input. During the execution the program flow is dynamically adjusted to lead to the selected statement by analyzing control flow and data dependency. The approach differs from the path-oriented test data generation in that the path selection stage is eliminated.

Jones and Privitera [13] proposed a method for the generation of test data for functional verification using a formal specification to drive the test generation and applied the method for a number of Rambus designs. For the test program generation, Wood, Gibson and Katz [41] proposed a method for verifying multiprocessor cache controller which used a pseudo-random generator to probabilistically generate all multiprocessor interaction cases. Lee and Siewiorek [18] presented a method to generate functional test program specifically designed for detecting design errors in pipelined computer implementations. An approach which combines random generation with user specific sequence generation according to knowledge and experience was proposed [22]. In IBM, a test program generation method based on an expert system consisting of heuristic testing knowledge database and a formal model of processor architecture has been used for functional verification of six IBM PowerPC processors [1]. For multiprocessor design verification of PowerPC 620 microprocessor, a test program generation method, which combines the deterministic and random techniques, has used instruction sequences that resemble commercial code for symmetric multiprocessors in market [23].

In the CAD domain recently Powley and De Groat developed a VHDL model for an embedded controller [33]. The model supports debugging of the application softwares. Naganuma, Ogura and Hoshino [24] combined the structured analysis approaches [25] with algorithmic debugging techniques from logic programming [35] to speed-up design validation processes. Potkonjak, Dey, and Wakabayashi [32] proposed a technique for design-for-debugging. Their technique focuses only on the error detection phase of debugging. The method provides controllability and observability for the variables specified by the designer. The method is applicable only to the hardwired ASICs. Koch, Kebschull, and Rosenstiel [14] proposed an approach for source level debugging of behavioral VHDL in a way similar to software source level debugging through the use of hardware emulation. They also presented an approach for breakpoint detection by hardware means with low overhead [15]. Benner et al. [2] presented a system emulator using rapid prototyping system for verification and evaluation of small embedded hardware/software-systems generated by a hardware/software cosynthesis. Fang, Wu and Yen proposed a method which supports on-line debugging for logic-emulation applications [8]. Simulation has been also used for functional debugging [27, 40]. While the behavioral synthesis for functional debugging has not received much attention, behavioral synthesis for manufacturing testing has been extensively discussed [29, 11, 20].

Register sharing is explored when register allocation is performed. Many well-known register allocation algorithms focus on either unconditional register sharing [39, 31, 12] or conditional register sharing [3, 17] for control data flow graphs that contain no loops. For control data flow graphs with loops, Stok and van den Born [37] proposed a method to break the loops at loop boundaries such that variables whose lifetimes cross a loop boundary are split and treated as two separate variables. When the two splitted variables of a variable are assigned to different registers, a register transfer operation is necessary. Stok [36] proposed a technique to minimize the unnecessary register transfer operations. Park, Kim and Liu [30] presented a transformation-based method for register allocation in the presence of both conditional branches and loops in a control data flow graph. All of these techniques assumed that register allocation is performed after scheduling is finished.

4 Design for Functional Test Pattern Execution

4.1 Global Flow of the Approach

The goal is to maximize the simultaneous controllability of an arbitrary set of the user selected variables in the design at the debugging time for facilitating the functional test pattern execution while minimizing the hardware overhead. There are several problems that make it hard to achieve high controllability. First, there are usually only a few primary inputs in designs. Primary inputs are directly controllable and are instrumental in providing controllability of other variables. Secondly, there are loops in designs. To control variables in a loop, we must be able to control one of the variables in the loop. Thirdly, there are often nonlinear operations in designs. Nonlinear designs are particularly harder to achieve controllability than linear ones. The key design element to address the three problems is resource sharing, especially register sharing between primary inputs and variables in designs. Our approach is to impose minimal restriction on register sharing so that the synthesized designs will have the desired characteristic while minimizing the additional hardware overhead and minimizing the disruption of the optimization potential when performing scheduling, allocation and binding tasks in high-level synthesis.

The idea is to find a complete cut that satisfies the following requirements:

- All output variables of nonlinear operations are included in the cut.
- The cut should be a partially redundant complete cut.

- The cut should minimize the disruption of optimization potential in scheduling, allocation and binding.
- The cut should include the minimum number of variables.

We explain the reasons behind the requirements. First, it is difficult to indirectly control the output variables of nonlinear operations from other directly controllable variables because it may involve solving a system of nonlinear equations. Thus, the output variables of nonlinear operations are directly controlled by the primary inputs through register sharing. In other words, the functional specification of the design is logically partitioned into linear components. Secondly, more variables in the cut impose more restriction on the later synthesis steps. Thus, if a variable can be removed from a complete cut without resulting in a partial cut, it should be removed. Thirdly, because we need to impose the restriction that every variable in the cut should share a register with any one of the primary input variables, the cut should be selected to impose minimal restriction. Finally, because direct controllability of a variable requires hardware overhead described in Section 2, the number of variables in the cut should be minimized to reduce the debugging hardware overhead.

After an optimal complete cut is found, we perform scheduling, allocation and binding such that every variable in the cut shares a register with any one of the primary input variables. If we fail to find such a cut, allocation of extra hardware, register-I/O connections with write ability to the registers can provide direct controllability to the cut variables which are incompatible with all primary inputs. The goal is to minimize the allocation of the extra hardware.

Given functional test patterns, a system of linear equations for the requested control variables with a function of only the variables in the complete cut and the primary input variables is constructed and solved.

4.2 Maximum Number of Iterations for Functional Test Pattern Execution

It is interesting to know the number of iterations required for any functional test pattern execution. A simple maximum bound is the number of functional delays plus 1. The exact number, however, is smaller or equal to the number depending on the dependencies between functional delays. The maximum number of iterations required for any functional test pattern execution can be computed using the algorithm described in Figure 4. In the first phase of the algorithm, all strongly-connected components of the computation are identified and a graph which describes the dependencies between them is constructed. A strongly-connected component is defined as the following.

Definition 9 (Strongly-Connected Component) *For any pair of operations A and B within a strongly-connected component, there exist both path from A to B, and one from B to A.*

The strongly-connected components can be efficiently identified using the standard depth-first search-based algorithm [5]. The weight of the vertex is the number of functional delays in the strongly-connected component. One such graph is shown in Figure 5. The number in bold italic represents the number of the delays that the strongly-connected component depends on. These numbers are computed by a dynamic programming starting from the primary inputs. The maximum of such numbers for the nodes in the graph plus 1 is the maximum number of iterations required for any functional test pattern execution for the computation. For the example, at most eight iterations are required while the simple bound is 13.

All strongly-connected components (SCCs) of the computation are identified;
A graph $G = (V, E)$ which describes the dependencies between SCCs is constructed;
For each vertex v in V
  $W(v) =$ the number of functional delays in v;
For each vertex v in V
  If there are no incoming edges to v
  Then $N(v) = W(v)$;
  Else $N(v) = \max_{\forall w:(w,v)\in E}(N(w) + W(v))$;
Return $\max_{\forall v\in V} N(v)$;

Figure 4. The algorithm for computing the maximum number of iterations required for any functional test pattern execution

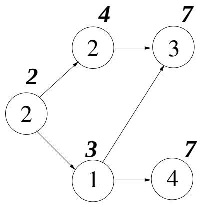


Figure 5. An example of computing the maximum iterations for functional test pattern execution

5 Selection of Optimal Partially Redundant Complete Cut

5.1 Distribution Graphs for Variables

We use *distribution graphs* (DG's) similar to the ones used in force-directed scheduling by Paulin and Knight [31]. The possible control step at which an operation is scheduled spans between its *as soon as possible* (ASAP) control step and *as late as possible* (ALAP) control step. By noting that an operation can be assigned to any control step between its ASAP and ALAP control steps, the lifetime of a variable before scheduling spans between the second phase of the ASAP control step of the source operation and the first phase of the maximum of ALAP control steps among all destination operations. The lifetime of a primary input variable includes only the control steps that the variable is used and spans the first phase of the minimum of ASAP control steps and the first phase of the maximum of ALAP control steps among all destination operations. We assume uniform probability of assigning an operation to any feasible control step. The height of a rectangle in a distribution graph for a variable means the probability that the corresponding control steps of the rectangle belong to the lifetime of the variable after scheduling is performed. Lemmas 1 and 2 show that any variables can share a register if the distribution graphs for the variables do not have such intersecting rectangles that the probabilities of the rectangles for all the variables are 1.

Lemma 1 *If the distribution graphs for variables x and y do not have such intersecting rectangles that the probabilities of the rectangles for the two variables are 1, then the two variables x and y can share a register by scheduling, allocation and binding.*

Proof: The source and destination operations of the variables x and y can be scheduled such that the lifetimes of x and y do not overlap because there is nonzero probability for every control step that at least one of the variables is not alive. $\square$

The Lemma 1 can be easily extended for n variables.

Lemma 2 *If the distribution graphs for variables $x_1, x_2, \cdots, x_n$ do not have such intersecting rectangles that the probabilities of the rectangles for all the variables are 1, then the n variables $x_1, x_2, \cdots, x_n$ can share a register by scheduling, allocation and binding.*

Proof: We can trivially use the same argument in Lemma 1 to prove this lemma. $\square$

We define the cost for a variable x, $COST(x)$, and the cost for a cut C, $COST(C)$ and use the cost measures to compare different variables and cuts, respectively.

Definition 10 (Cost for a Variable) *For a variable x,*

$$COST(x) = \begin{cases} \infty & \textit{if } x \textit{ is incompatible with every primary input} \\ \textit{Intersecting areas between } x \textit{ and primary inputs} & \textit{otherwise} \end{cases}$$

Definition 11 (Cost for a Cut) *For a cut C, $COST(C) = \sum_{\forall x \in C} COST(x)$*

To illustrate the creation of distribution graphs for variables, consider the design shown in Figure 6. The design is a sub-part of a low-pass lattice filter of degree 3, which is designed to be used in a 12-channel transmultiplexer [9]. The critical path of the design has six control steps. The available time is also six control steps. The distribution graphs for the variables **IN**, a, b, c and d in Figure 6 are shown in Figure 7. Note that $\{a\}$, $\{b\}$ and $\{c, d\}$ are irredundant complete cuts.

It is simple to see that the variables a and **IN** can share a register without disrupting the optimization potential because there are no intersecting rectangles. The variables b and **IN** can also share a register, but should disrupt the optimization potential because the destination multiplication operation of the variable b must be scheduled at control step 3 even though it can be scheduled in any control step between 3 and 6. The variable c can share a register with **IN**, but the variable d can not because d and **IN** are both alive at control step 6. Therefore the complete cut $\{a\}$ is clearly the best choice for our objective. Our cost measure reaches the same conclusion because $COST(\{a\}) = 0$, $COST(\{b\}) = 0.5$ and $COST(\{c, d\}) = \infty$.

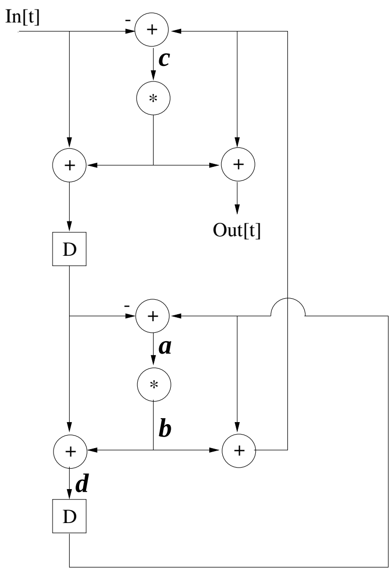
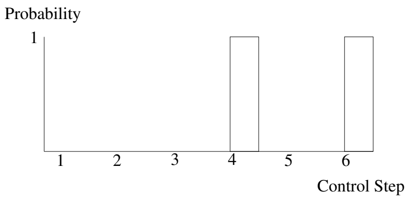
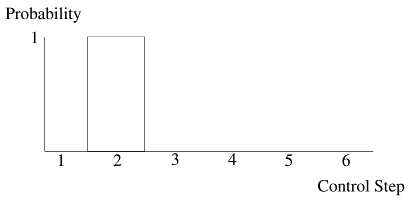


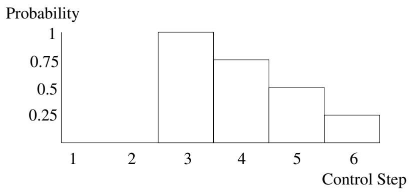
Figure 6. Control data flow graph for a low-pass lattice filter designed for communications application



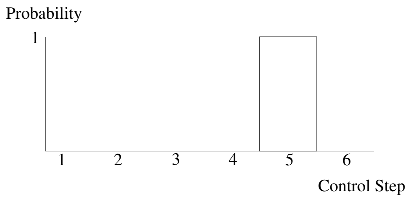
(a) DG for a variable IN



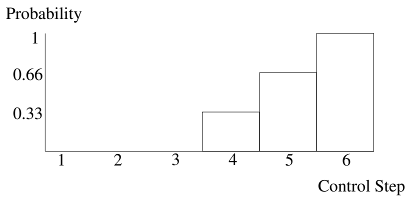
(b) DG for a variable a



(c) DG for a variable b



(d) DG for a variable c



(e) DG for a variable d

Figure 7. Some distribution graphs for the control data flow graph in Figure 6

5.2 Strongly-Connected Components

To facilitate the construction of the complete cut, we use strongly-connected components. Every operation in non-trivial strongly-connected components (those with more than one operation) is part of a loop. [1] By Lemma 3, a complete cut for a computation can be obtained by the separate complete cuts for all strongly-connected components of the computation.

Lemma 3 *The complete cuts for all non-trivial strongly-connected components form a complete cut for the entire computation.*

Proof: Every loop in the computation is included in exactly one of the non-trivial strongly-connected components because all operations in a loop have paths to each other. □

This lemma can be easily extended for partially redundant complete cuts.

5.3 Optimal Cut for Functional Test Pattern Execution

The problem to find an optimal complete cut for functional test pattern execution is defined as the following: **Problem (Optimal Complete Cut for Functional Test Pattern Execution):** Find a complete cut that satisfies the following requirements:

- All output variables of nonlinear operations are included in the cut.
- The cut should be a partially redundant complete cut.
- The cut should minimize the disruption of optimization potential in scheduling, allocation and binding.
- The cut should include the minimum number of variables.

The problem is NP-complete since the FEEDBACK ARC SET problem [10] can be reduced to our problem by considering only the last requirement that the number of variables in the complete cut should be minimized.

Due to the computational complexity, we have to resort to a heuristic method to find a good solution for the problem. The pseudo code of the heuristic for the Optimal Complete Cut for Functional Test Pattern Execution problem is provided in Figure 8. Checking if a cut is a complete cut can be efficiently done by removing all the corresponding edges of the cut from the CDFG and running any directed graph cycle detection algorithm on the graph [5]. It is simple to see that all functional delay variables form a complete cut. Replacing a variable with the output variables of all destination operations in strongly-connected components maintains the completeness of the cut. When performing the replacement, we can also remove all input variables of the operation from the complete cut because all the loops containing the input variables also contain the output variables. As a postprocessing step, we remove all redundant variables to make the complete cut to be partially

[1] There may be more than one loop in a strongly-connected component. See an example in Figure 6. The computation is a strongly-connected component which has two loops.

```
Cuts C_n and C_l are initially set to empty;
Include all output variables of nonlinear operations in the cut C_n;
If the cut C_n is a complete cut
Then return C_n;
Else {
  Include all functional delay variables in the cut C_l;
  C_t = C_l;
  For each variable v in C_t
    Compute COST(v);
  COST_old = COST(C_t);
  COST_best = COST_old;
  Identify all strongly-connected components;
  Repeat {
    Randomly select a variable x from C_t with a probability proportional to its COST;
    Replace x with the output variables of all destination operations in strongly-connected components;
    Remove all input variables of the destination operations of x from C_t;
    For each newly added variable v in C_t
      Compute COST(v);
    COST_new = COST(C_t);
    If (COST_new < COST_best) {
      COST_best = COST_new;
      C_l = C_t;
    }
  } Until (no improvement in k consecutive moves);
  Remove redundant variables from C_l;
  Return C_n ∪ C_l;
}
```

Figure 8. The pseudo code of the heuristic for the Optimal Complete Cut for Functional Test Pattern Execution problem

redundant complete cut. The redundancy of a variable v can be checked by removing all the corresponding edges of the complete cut except v from the CDFG and running any directed graph cycle detection algorithm on the graph. If there is no cycle, v is redundant.

6 Experimental Results

We applied our approach to design for functional test pattern execution on a set of 10 industrial examples. Table 1 provides the characteristics of the considered designs and presents the experimental results for the designs. The examples are the following: second order Volterra filter, third order Volterra filter, 12th order IIR filter, DAC (NEC digital-to-analog converter for audio applications), LMS audio formatter (NEC design for communication), Avenhaus direct-form filter, Avenhaus cascade-form filter, Avenhaus parallel-form filter, Avenhaus continued-fraction filter, and Avenhaus ladder filter. The second order Volterra filter, third order Volterra filter and LMS audio formatter are nonlinear designs while the rest are linear designs. The sixth column corresponds to the timing requirements in terms of the critical path length. The ninth column presents the hardware overhead of the new approach for the functional test pattern execution. The tenth column provides the maximum number of iterations for any functional test pattern execution on the design.

All designs were synthesized using the Hyper high-level synthesis system from University of California, Berkeley [34]. Original synthesized designs without considering functional test pattern execution showed poor functional test pattern execution ability. In all cases, we were required to use extra I/O pins. Due to the interconnect and register file model of the Hyper system, the hardware overhead for controlling registers in debugging mode, as shown in Figure 3, was not incurred. The other hardware overhead in I/O pins and registers was minimal. In all cases, no extra I/O pins were required. In only a few cases, one extra register was required. We note that for 12th order IIR filter, the number of registers actually decreased by 1 when the available time was equal to the critical path length. Since the scheduling, allocation and binding algorithms the Hyper system uses are randomized, adding a few "good" constraints may result in better solutions. The constraints we impose on register sharing are minimal and sometimes beneficial since the selected variables to share registers with primary inputs are highly compatible with primary inputs. For all examples, no area overhead was incurred. The designs who used one more register resulted in less area for interconnect.

7 Conclusion

We proposed a new approach for the functional test pattern execution phase of the ASIC functional debugging process in high level synthesis. The goal was to maximize the simultaneous controllability of an arbitrary set of the user selected variables in the design at the debugging time for facilitating the functional test pattern execution while minimizing the hardware overhead. The new approach, based on the divide and conquer optimization paradigm, provided the full controllability of each component by exploiting resource sharing of a set of variables in the components and primary inputs. The variables were selected so that they could enable direct and complete answers on all possible questions related to simultaneous controllability of an arbitrary set of variables in the design. The selection of the variables introduced an interesting combinatorial optimization

Design	# of Primary Inputs	# of Nonlinear Operations	# of SCCs	# of Variables	Available Time	Initial /Final Area	Clock Cycles	Overhead Pins/ Registers	# of Iterations for FTPE
12th order IIR	1	0	6	56	cp	48.13	13	0/-1	7
					1.1*cp	48.03	14	0/0	7
					5*cp	10.48	65	0/0	7
Avenhaus direct form	1	0	1	40	cp	92.49	10	0/0	9
					1.1*cp	54.17	11	0/0	9
					5*cp	20.32	50	0/0	9
Avenhaus cascade form	1	0	4	34	cp	12.27	10	0/0	9
					1.1*cp	8.97	11	0/0	9
					5*cp	5.98	50	0/0	9
Avenhaus parallel form	1	0	4	39	cp	11.47	8	0/0	3
					1.1*cp	8.67	9	0/0	3
					5*cp	6.71	40	0/0	3
Avenhaus continued fraction	1	0	1	35	cp	37.80	18	0/0	9
					1.1*cp	32.67	20	0/0	9
					5*cp	14.02	90	0/0	9
Avenhaus ladder	1	0	1	50	cp	16.87	27	0/0	9
					1.1*cp	11.45	30	0/0	9
					5*cp	5.73	135	0/0	9
DAC	1	0	2	354	cp	27.06	132	0/1	3
					1.1*cp	24.61	145	0/0	3
					5*cp	16.02	660	0/0	3
2nd order Volterra	1	6	1	29	cp	10.81	12	0/0	1
					1.1*cp	10.81	13	0/0	1
					5*cp	5.40	60	0/0	1
3rd order Volterra	1	12	1	50	cp	12.50	20	0/0	1
					1.1*cp	9.98	22	0/0	1
					5*cp	2.10	100	0/0	1
LMS audio formatter	1	2	3	464	cp	73.26	202	0/1	4
					1.1*cp	56.24	222	0/0	4
					5*cp	38.83	1010	0/0	4

Table 1. The characteristics and experimental results of the examples used to demonstrate the effectiveness of the new approach. cp - critical path length

problem. We established its computational complexity and proposed a non-greedy heuristic algorithm. The approach imposed minimal restriction on register sharing which allowed the synthesized designs to have the desired characteristic while minimizing the additional hardware overhead and minimizing the disruption of the optimization potential when performing scheduling, allocation and binding tasks in high-level synthesis. The effectiveness of the proposed approach was demonstrated on a number of designs.

References

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Design Automation Conference*, pages 279–285, 1995.

[2] T. Benner, R. Ernst, I. Konenkamp, P. Schuler, and H.-C. Schaub. A prototyping system for verification and evaluation in hardware-software cosynthesis. In *International Workshop on Rapid System Prototyping*, pages 54–59, 1995.

[3] R.A. Bergamaschi, R. Camposano, and M. Payer. Data-path synthesis using path analysis. In *Design Automation Conference*, pages 591–596, 1991.

[4] D.L. Bird and C.U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] T.H. Cormen, C.E. Leisserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] R.E. Crochiere and A.V. Oppenheim. Analysis of linear digital networks. *Proceedings of the IEEE*, 63(4):581–595, 1975.

[7] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[8] W. Fang, A.C.H. Wu, and T.-Y. Yen. A real-time RTL engineering-change method supporting on-line debugging for logic-emulation applications. In *Design Automation Conference*, 1997.

[9] A. Fettweis. Wave digital filters: theory and practice. *Proceedings of the IEEE*, 74(2):270–327, 1986.

[10] M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, 1979.

[11] F.F. Hsu, E.M. Rudnick, and J.H. Patel. Enhancing high-level control-flow for improved testability. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 322–328, 1996.

[12] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu. Data path allocation based on bipartite weighted matching. In *Design Automation Conference*, pages 499–504, 1990.

[13] K.D. Jones and J.P. Privitera. The automatic generation of functional test vectors for Rambus designs. In *Design Automation Conference*, pages 415–420, 1993.

[14] G. Koch, U. Kebschull, and W. Rosenstiel. Debugging of behavioral VHDL specifications by source level emulation. In *European Design Automation Conference*, pages 256–261, 1995.

[15] G. Koch, U. Kebschull, and W. Rosenstiel. Breakpoints and breakpoint detection in source level emulation. In *International Symposium on System Synthesis*, pages 26–31, 1996.

[16] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[17] F.J. Kurdahi and A.C. Parker. REAL: a program for REgister ALlocation. In *Design Automation Conference*, pages 210–215, 1987.

[18] D.C. Lee and D.P. Siewiorek. Functional test generation for pipelined computer implementations. In *Fault-Tolerant Computing Symposium*, pages 60–67, 1991.

[19] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[20] T.-C. Lee, W.H. Wolf, and N.K. Jha. Behavioral synthesis for easy testability in data path scheduling. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 616–619, 1992.

[21] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[22] J. Miyake, G. Brown, M. Ueda, and T. Nishiyama. Automatic test generation for functional verification of microprocessors. In *Asian Test Symposium*, pages 292–297, 1994.

[23] C. Montemayor, M. Sullivan, J.-T. Yen, P. Wilson, and R. Evers. Multiprocessor design verification for the PowerPC 620 microprocessor. In *Internation Conference on Computer Design*, pages 188–195, 1995.

[24] J. Naganuma, T. Ogura, and T. Hoshino. High-level design validation using algorithmic debugging. In *European Design and Test Conference*, pages 474–480, 1994.

[25] S. Narayan, F. Vahid, and D.D. Gajski. System specification with the SpecCharts language. *IEEE Design & Test of Computers*, 9(4):6–13, 1992.

[26] S. Narita, F. Arakawa, K. Uchiyama, and I. Kawasaki. Design methodology for GMICRO/500 TRON microprocessor. In *International Conference on Computer Design*, pages 253–257, 1993.

[27] K.A. Olukotun, R. Helaihel, J. Levitt, and R. Ramirez. A software-hardware cosynthesis approach to digital system simulation. *IEEE Micro*, 14(4):48–58, 1994.

[28] A.V. Oppenheim and R.W. Shafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[29] C. Papachristou and J. Carletta. Test synthesis in the behavioral domain. In *International Test Conference*, pages 693–702, 1995.

[30] C. Park, T. Kim, and C.L. Liu. Register allocation for data flow graphs with conditional branches and loops. In *European Design Automation Conference with EURO-VHDL*, pages 232–237, 1993.

[31] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[32] M. Potkonjak, S. Dey, and K. Wakabayashi. Design-for-debugging of application specific designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 295–301, 1995.

[33] G.S. Powley and J.E. DeGroat. Experiences in testing and debugging the i960 MX VHDL model. In *VHDL International Users Forum*, pages 130–135, 1994.

[34] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of data path intensive architectures. *IEEE Design & Test of Computers*, 8(2):40–51, 1991.

[35] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[36] L. Stok. Transfer free register allocation in cyclic data flow graphs. In *European Conference on Design Automation*, pages 181–185, 1992.

[37] L. Stok and R. van den Born. EASY: multiprocessor architecture optimisation. In *International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, pages 313–328, 1989.

[38] G. Strang. *Linear Algebra and its Applications*. Academic Press, New York, NY, 1976.

[39] C. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(3):379–395, 1986.

[40] C.A. Valderrama, F. Nacabal, P. Paulin, and A.A. Jerraya. Automatic generation of interfaces for distributed C-VHDL cosimulation of embedded systems: an industrial experience. In *IEEE International Workshop on Rapid System Prototyping*, pages 72–77, 1996.

[41] D.A. Wood, G.A. Gibson, and R.H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test of Computers*, 7(4):13–25, 1990.