A Quantitative Approach to Functional Debugging

Abstract

Functional debugging dominates both development time and cost of modern design process. Two dominant debugging techniques are simulation and emulation. Neither of them provides an adequate debugging solution. Design simulation is two to ten orders of magnitude slower than functional execution with respect to the fabricated design. Design emulation and fabrication provide high execution speed, but also demonstrate very serious disadvantage: they significantly restrict the observability and controllability of majority of variables in the design.

To facilitate debugging, in particular error detection, we introduce a novel cut-based functional debugging paradigm. It coordinates design emulation and simulation and enables fast transition from one to another. Therefore, we leverage on the advantages of both debugging domains in order to provide fast functional debugging as well as full observability and controllability of all variables. Emulation or functional implementation is used for fast application execution during debugging, while simulation provides complete observability and controllability.

The effective implementation of the new debugging approach poses several CAD tasks. We formulate the optimization tasks, establish their complexity, and develop least-constraining most-constrained heuristics to solve them. The efficiency and effectiveness of the new approach and accompanying algorithms is demonstrated on a set of designs where full observability and controllability are accomplished with low hardware overhead.

1 Introduction

1.1 Motivation

Functional debugging dominates both development time and cost of modern design process. For example, the UltraSPARC design team reported that debugging efforts (mainly architecture and functional verification) took two times longer than the design activities [Yan95]. Similarly, the Hitachi design team reported that specification of the GMICRO 500 microprocessor took less than 36 man-months, functional design took 80 man-months, standard cell design took only 3 man-months, physical design took less than 24 man-months, and finally, debugging efforts took more than 270 man-months [Nar93]. Extensive consultation with designers and CAD tool developers in large and small, US and Japanese, IC and CAD companies suggested similar design effort distributions [Gue97], [Ku97], [Wak96].

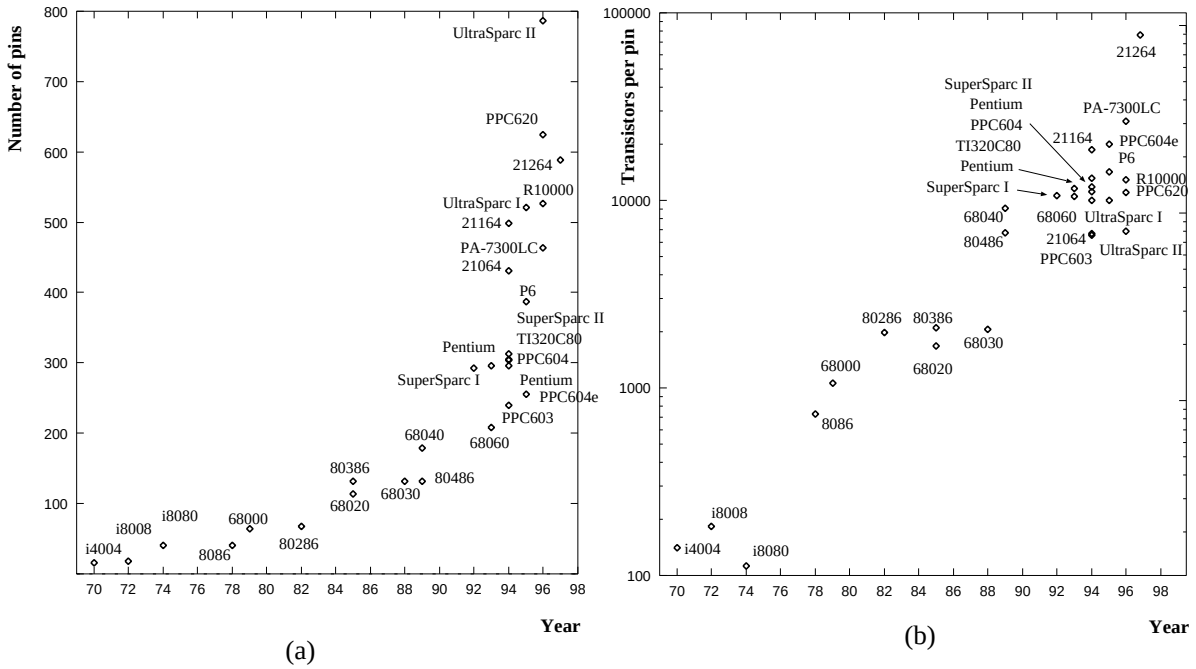


Figure 1: The key physical microprocessor trends.

The key technological and application trends indicate that the cost and time expenses of debugging follow sharply ascending trajectories. The technological trends inducing additional

constraints on debugging are mainly related to increasingly reduced design observability and controllability. Two most directly related factors are rapid growth in the number of transistors per pin in each new generation of designs and the increasing levels of hardware sharing due to increasing clock speeds. Higher levels of hardware sharing are associated with more complex information flows in application specific systems.

For example, the analysis of physical data of state of the art microprocessors (following Microprocessor Report data) indicates that in less than two years (from late 1994 to middle of 1996) the number of transistors per pin increased by more than a factor of two, from slightly more than 7,000 to 14,100 transistors per pin. While in 1994 eight microprocessors had a total of 16.1 million transistors and 2,296 pins, in 1996 nine microprocessors had a total of 53.4 million transistors and 3,781 pins. Wider perspective on the trend of huge discrepancy in growth of number of transistors and pins per chip is shown in Figure 1. Figure 1a points to the fact that the packaging technology has showed significant improvements over the past two decades. However, the rapid decrease of the silicon feature size has resulted in even higher ascending amount of available resources (transistors/gates) on chip. In Figure 1b, collected transistor per pin ratios for the most popular general purpose processors over the past three decades clearly point to the resulting exponential semiconductor over packaging technology trend.

The size of an average embedded or DSP application has been approximately doubling each year, the time to market has been getting shorter for each new product generation, and there has been a strong market need for user customization of application specific systems. These three application factors have resulted in shorter available debugging time of increasingly more complex designs. Finally, design and CAD trends which additionally emphasize the importance of debugging include design reuse, introduction of system software layer, and increased importance of collaborative design. These factors result in increasingly intricate functional errors, often due to interaction of parts of designs/programs written by several designers/programmers.

Although, in software compilers, debugging has been long recognized as one of the crucial compilation utility tools, until now only a very limited effort has been devoted to research on debugging as part of synthesis process. This is mainly due to the limited controllability and observability of storage elements in IC designs, which makes design debugging dramatically more difficult than software debugging.

The existing, widely employed, approaches for functional debugging are either design simulation, or chip emulation or fabrication. Design simulation is typically run on a workstation environment and results in an arbitrary accurate overview of the simulated architecture. Unfortunately, simulation is an extremely computation intensive process and results in two to ten orders slower functional execution in comparison to the fabricated chip. Most important, the actual simulation speed heavily depends on the simulation accuracy. Emulated or fabricated designs on the other hand provide real-life execution speed and yet extremely limited observability and controllability of the design internal structure.

The novel ideas proposed in this work advocate development of a new paradigm for debugging and design-for-debugging of integrated circuits. The paradigm integrates design emulation and simulation in such a way that the advantages of the two are combined, while the disadvantages are eliminated. The debugging technique is fully transparent in a sense that it can be used in conjunction with any existing or future synthesis system or manual design approach as a postprocessing step. The presented debugging approach aims at including debugging criteria in the design process as integral design objectives.

1.2 The New Approach

In order to explain further the proposed debugging process we introduce the notion of a *complete cut* of a computation. A *complete cut* is a set of variables generated within one computation iteration which bisects all possible paths in the control data flow graph. Such set of variables fully determines the state of the machine between two iterations, i.e. the set of primary inputs and the optimal cut of a particular iteration fully determine the set of primary outputs of the same computation iteration.

The functional debugging process, depicted in Figure 2, includes four standard debugging procedures: test input generation, error detection, error diagnosis, and error correction. The test input vectors generated in the first phase are used to drive the design emulation. While primary inputs, cuts and outputs are stored in the permanent storage of the supervising workstation, error detection techniques are deployed to signal to the workstation whether the emulation should be terminated. Upon error detection, the error has to be localized and characterized. This is done

by running the design simulation on a workstation starting from a designer selected breakpoint. The computation is successfully continued from the selected breakpoint due to the recorded cuts and their appropriate primary inputs and outputs.

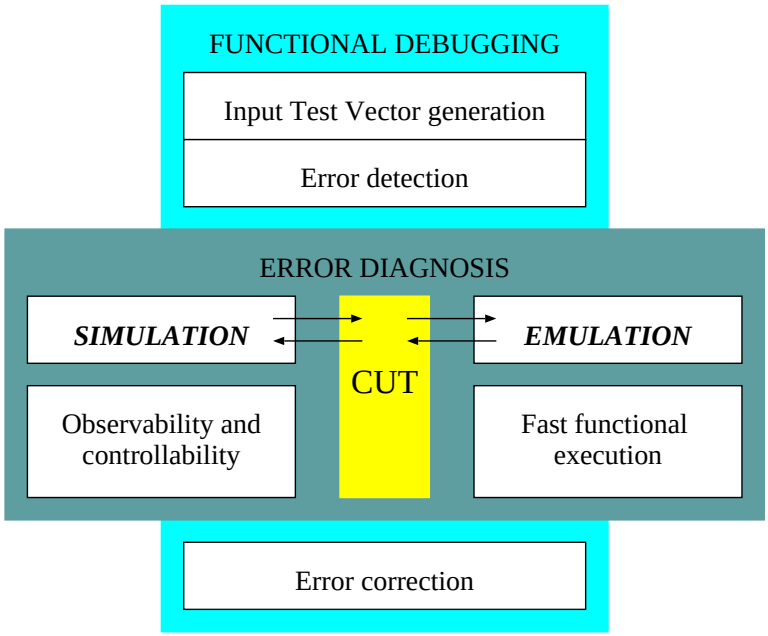


Figure 2: The concept of functional debugging.

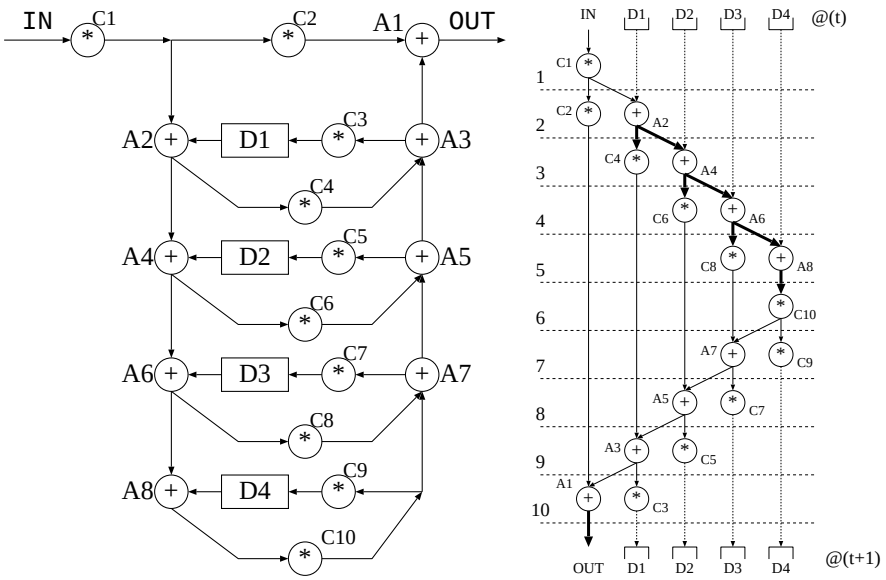
Error is diagnosed using a symbiosis of the design simulation and emulation. Design simulation is used for accurate, but slow, system function test, while the emulation is used for fast execution of the test input vectors. The exchange of information between the simulation and emulation domain is accomplished by controlling and observing breakpoints' complete cuts of both test domains. Upon error diagnosis either the emulator is updated or built-in fault tolerance mechanisms are activated. The ability to read/write breakpoint cuts from/to the emulated/fabricated design is enabled by the design-for-debugging postprocessing step. It inserts hardware resources into the initial design specification.

In this paper we define and establish the complexity of the optimization problems involved in the design-for-debugging postprocessing phase. The developed set of optimization tools aims to add minimum hardware overhead, and still provide efficient integration of the two functional testing domains. The applied algorithms are constructed using a least-constraining most-constrained heuristic methodology. The efficiency of the developed algorithms is tested on a set of real-life

examples where full design observability and controllability is provided with exceptionally low implementation overhead.

1.3 Motivational Example

The diagnosis approach and accompanying optimization issues are illustrated using 4th order continued fraction infinite impulse response (CF IIR) filter [Cro75]. Figure 3a shows the control data flow graph for this popular filter structure. On Figure 3b the computation control flow graph is presented with respect to the particular control step.



(a) Control data flow graph.

(b) Scheduled CDFG.

Figure 3: Optimal cut example for 4th order CF IIR filter.

Recall that the goal of the design-for-debugging step is to allocate minimal hardware resources to enable computation observability and controllability. One possible solution would be to select variables $D1$, $D2$, $D3$, and $D4$ (dotted lines in Figure 3b) as a complete cut. Since all variables of that cut are concurrently alive, they have to be stored in four different registers. In order to provide design full controllability and observability, the designer has to have the ability to read/write into those registers from the designated I/O pins. Since four registers are in the cut, four sets of

register-to-I/O connections have to be allocated in order to enable variable observability and controllability.

If the cut is defined among the output variables of adders $A2$, $A4$, $A6$, and $A8$ (bold lines in Figure 3b), only one register is required to hold the values of all those variables, since they are not alive simultaneously during the computation. In this case only one instance of the selection hardware is dedicated to the register which holds the cut. Cut dispensing is performed in four consecutive control steps (cycles 2, 3, 4, and 5). Note that in both cases the variable included in the cut but not mentioned is the actual output of the chip. The primary output of the computation is always used as a part of the complete cut since its dispensing is inevitable.

1.4 Paper Organization

The rest of the paper is organized in the following way. In order to properly position the proposed research, we survey related work on hardware and applicable software debugging techniques in Section 2. Section 3 presents the assumed hardware and computational models, outlines a typical debugging flow and states the key assumptions drawn for our debugging technique. The idea of a cut-based integrated debugging is extrapolated in Section 4. Section 5 describes the design-for-debugging postprocessing step, the associated optimization problems and the developed heuristic design tools. Section 6 and 7 present the experimental results and this paper's conclusion respectively.

2 Related Work

Debugging is as old as building digital computing systems. Initially debugging was relatively rarely addressed in research literature due to its exceptional conceptual complexity [Bal69], [Hen82]. The situation changed and, consequently, debugging was recognized as a crucial digital system design activity.

By far the most comprehensive treatment of debugging has been conducted in the software compiler domain. Several debuggers, such as VAX DEBUG [Bea83], GDB [Sta91], and more recently Purify [Bro92] have been widely used. The majority of efforts related to debugging in the compiler domain have been dedicated to symbolic level debugging on uniprocessor computers.

The observation that debugging often significantly slows down the execution of the program [Kau88] has directed significant efforts towards the development of efficient support mechanisms for debugging. For example, the Intel 80386 [Int86] provides support for debugging using monitor registers. Currently, in the software compilation debugging domain, major research emphasis is on aggressive scheduling techniques, such as speculative scheduling, and transformation based optimization techniques [Cou88], [Bro92].

Numerous controllers and DSP processors are supported by in-circuit emulators which enable an efficient debugging process [Phi84], [Wyn86]. An in-circuit emulator of a processor is a system that can imitate the mechanical (e.g. pin compatibility), electrical (e.g. voltage level, loading), and functional (e.g. memory read/write cycle) behavior of the microprocessor. Moreover, the internal states and operations of the emulated processor are fully observable and controllable by the user [Chi94]. While some in-circuit emulators are provided by manufacturing houses, a number of in-circuit emulators are built by developers of applications [Chi94]. The major drawback of in-circuit emulation is high cost. An interesting alternative to in-circuit emulation is used during development of new microprocessors (e.g. as one offered by Quickturn, Mentorgraphics). The processor model is ported onto a logic design hardware model comprised of an array of rapidly prototyping modules. Hence, both high speed execution and relatively high observability and controllability of all registers are provided. For example, this methodology has been used during development of Intel's Pentium processor [Sai93]. While this approach is significantly faster than typical software simulation, it is still slower by 2 orders of magnitude than the real operation speed [Sai93].

In the CAD domain, Powley and De Groat had recently developed a VHDL model for an embedded controller [Pow94]. The model supported debugging of the application software. In order to speed-up the design validation process, Naganuma et al. [Nag94] combined structured analysis approaches [Nar91] with algorithmic debugging techniques from logic programming [Sha83] to speed-up design validation process.

Potkonjak et al. [Pot95] propose a technique for design-for-debugging. Their technique focuses only on the error diagnosis phase. It assumes that the designer specifies the debugging variables at design time and only provides controllability and observability of the specified variables. The technique is applicable only to hardwired ASIC designs, and assumes a single functional fault

tolerance.

3 Preliminaries

In order to make the paper self-contained, in this section we outline all relevant design debugging and design-for-debugging background material.

3.1 Computation and Hardware Model

For the sake of brevity and conceptual simplicity all our illustration examples follow a synchronous computation data flow model [Lee87]. Nevertheless, the restrictions imposed on the assumed computation model by the proposed set of debugging techniques are very mild. We assume fully deterministic hardware behavior and its continuous semi-infinite operation mode (not necessarily periodic).

We do not impose any register-transfer level restrictions on the interconnection scheme of the assumed hardware model. Registers may or may not be grouped in register files. Each hardware resource can be connected in an arbitrary way to another hardware resource.

3.2 Debugging Process

After the functional design of a hardware component is specified, functional debugging is used to partially or fully verify and validate the design scheme. In search for an error, a number of steps are taken. They can be partitioned into the following four debugging phases.

Test Input Generation. In the first phase of debugging, the goal is to generate and execute the input data likely to make functional errors visible. In order to create the most promising set of input vectors designer's intuition can be used as well as such sophisticated methods as specialized databases or expert systems.

Error Detection. The designer discovers that the design does not function correctly for a particular input. The discovery can be obtained either by output matching software run by the supervising workstation or defining design-internal error events at design time. Upon an event which undoubtly specifies functional error, the chip signals the supervising station to terminate the emulation.

Error Diagnosis. In the third phase the designer identifies the statement or section of code which causes incorrect behavior. To facilitate error correction we develop a technique to integrate emulation and simulation in such a way as to combine advantages of the two and to eliminate their disadvantages. Design emulation is used for a fast testing of the targeted functionality while simulation is used to provide complete controllability and observability of all variables.

Error Correction. In the final phase the faulty section or statement responsible for the observed fault is replaced by the corrected section. The design is recompiled if simulated, or its specification is updated and the design is, again, emulated or fabricated.

3.3 Debugging: Constraints and Objectives

The four key debugging assumptions are listed bellow.

- The design is fully specified and its functionality and realization is not disturbed by the debugging process with the exception of enabling the user to write into specific controllable variables. A fully specified design is one, where each operation, variable, and data transfer is scheduled and assigned to a particular instance of hardware resource in one or more control steps. In many cases, in software debuggers, the usage of debugging methods modifies the program flow of execution and sometimes even its behavior. This is often quoted as an exceptionally unpleasant phenomenon, named Heisenberg Uncertainty Principle ("Heisenbug") as applied to debugging [Gai85]. Our key premise is that an addition of a small amount of debugging hardware is sufficient to completely eliminate the Heisenberg Debugging Uncertainty Principle.

- In order to support debugging, we allocate additional debugging hardware to satisfy all debugging requirements. The goal is to add as little as possible hardware. In particular, we do not allow increase in the number of I/O pins, since this is a constraint which dominates other hardware constraints in modern designs.

- For proper debugging support, all variables in the optimal cut should be controllable and observable in the same iteration of the computation.

- Controllable and observable variable selection decisions are not known at compile or syn-

thesis time. Therefore, the goal is to enable simultaneous complete controllability and observability of all user defined variables.

4 The New Approach: Cut-Based Integrated Debugging Flow

The key idea behind the cut-based approach for the integration of simulation and emulation is to leverage on the strong aspects of each of the execution domains and to eliminate their disadvantages. Simulation provides complete controllability and observability of all variables in the design. However, this comes at the expense of very slow execution. Depending on the simulation modeling accuracy, it is reported that simulation results in a two to ten order of magnitude slowdown [Ros95], [Ziv96]. On the other hand a fabricated design or its emulation provides real-time or near real-time speed of execution. Due to strict pin limitations, the speed of execution comes at the expense of a very limited and at best cumbersome controllability and observability.

Our new approach combines the two execution domains to provide fast, observable and controllable functional debugging. The essence of the idea is the establishment of the concept of a complete cut. This concept provides information exchange between the two execution domains with small hardware overhead. In addition, the functional and timing specification design aspects are not affected.

Definition. *A complete cut is subset of variables which contains sufficient information required to correctly continue the computation with respect to any incoming set of primary inputs.*

Thus, a complete cut corresponds to a subset of variables in the computation which bisect all paths between the states that delimit successive iterations of the computation. Clearly, if one has complete controllability and observability over the values for all variables in the complete cut for a specific breakpoint, the computation can be continued functionwise correctly from that breakpoint. In a sense, the cut contains the complete information about the complete history of the computation process and its primary inputs until a given point in time (breakpoint). For the sake of brevity, from now on when we say cut, we mean a complete cut.

Our cut-based functional debugging approach is conducted using the following two phases of the design and debugging process:

- Design postprocessing - Defining the cut from the initial design specification at the functional level; - Augmentation of the design specification with cut statements which support controllability and observability when the design is executed in a debug mode; and
- Error diagnosis - Simultaneous and coordinated design execution of the fabricated or emulated design and the appropriate simulator for efficient error diagnosis.

In the first phase, a computation iteration at the behavioral-level of specification is logically partitioned into two or more components such that the cut between the partitions is complete. If the cut bipartitions the computation, we call the cut simple. The synthesis support for exchange of information between simulation and emulation has the following three degrees of design freedom:

- The determination of variables which form the cut;
- The determination of the exact control step when a particular variable is read from its register or replaced by a user specified value.
- The assignment of specific sets of I/O pins used to transfer variables to or from the chip.

It is important to notice the optimization tradeoffs involved in determining the optimal cut. The optimization procedure has a unique goal: to add minimum hardware resources into the initial design, while obtaining its full controllability and observability.

A cut with minimum number of variables seems to be an attractive solution. If those variables are simultaneously alive, more registers storing the cut-set variables have to be provided with register-to-I/O pin interconnects. Cuts which encapsulate the life-time flexibility of its variables and the smallest number of machine registers which store them, are favorable. Once an optimal cut is found, the next design problem is to define the sequence of control steps in which the variables are dispensed out of the chip. The dispensing freedom is limited due to the control steps when the I/O pins are busy.

The second phase of the synthesis approach is feasible once cut selection and output variable scheduling is performed, i.e. slicing information is obtained. Then, the initial specification is augmented with resources which enable design observability and controllability. For example,

the following input operation is incorporated to provide complete controllability of variable Var_1 using user specified input variable $Input_1$:

$$if\ (DEBUG)\ then\ Var_1 = Input_1;$$

After the chip prototype is fabricated or emulated, number of actions are taken in order to debug the functionality of the design. The debugging environment consists of a workstation and fabricated integrated circuit or emulation testbed which interfaces with the workstation. Although in the proposed approach all described four phases of the debugging process share some information, the phases are conceptually self-contained and the software is fully modular.

The designer starts the debugging process by preparing a set of test inputs intuitively or using a software tool which searches the test input generation database. Once the test vectors are determined, functional testing of the fabricated or emulated chip starts at the emulation testbed. While test inputs are forwarded to the chip, the workstation stores the debugging information in its permanent storage using its file system. For each iteration of the computation, its primary inputs, complete cut and resulting outputs are stored into a workstation supervised cut-database.

An efficient error detection mechanism has to be provided in order to terminate the emulation upon functional misbehavior of the design. There are several ways how this can be accomplished. One approach is to compare the outputs with the precomputed expected output values. Sometimes (as in compression/decompression or sorting) it is easy to check the correctness of the result. If the computation result is not predictable, the debugged chip, as an alterantive, signals the workstation on a user predefined design-internal event (overflow, sign of variables, exception handling etc.). In both cases the emulation is terminated upon mismatched output or signalized internal error.

If error detection results in discovering chip malfunction, the next step in the debugging flow is to provide the designer with a set of tools to easily diagnose, i.e. locate and characterize the error. At this point the cut-database is searched for cuts which might point and clear out the error. The designer, guided by a debugging guidance tool, selects the most suspicious cut and runs a cycle-accurate chip simulation on the workstation. If the simulator provides sufficient information, the error can be discovered at that point. Otherwise the designer continues the investigation by intuitively modifying internal registers which are likely to discover the error. The modification is followed by the simulation of the next iteration's cut. Once the cut is retrieved from the

cycle-accurate simulator, all variables in the cut-set are injected into the design emulator while tempering the appropriate input vector. The generated outputs in the following few iterations are likely to uncover the suspected error.

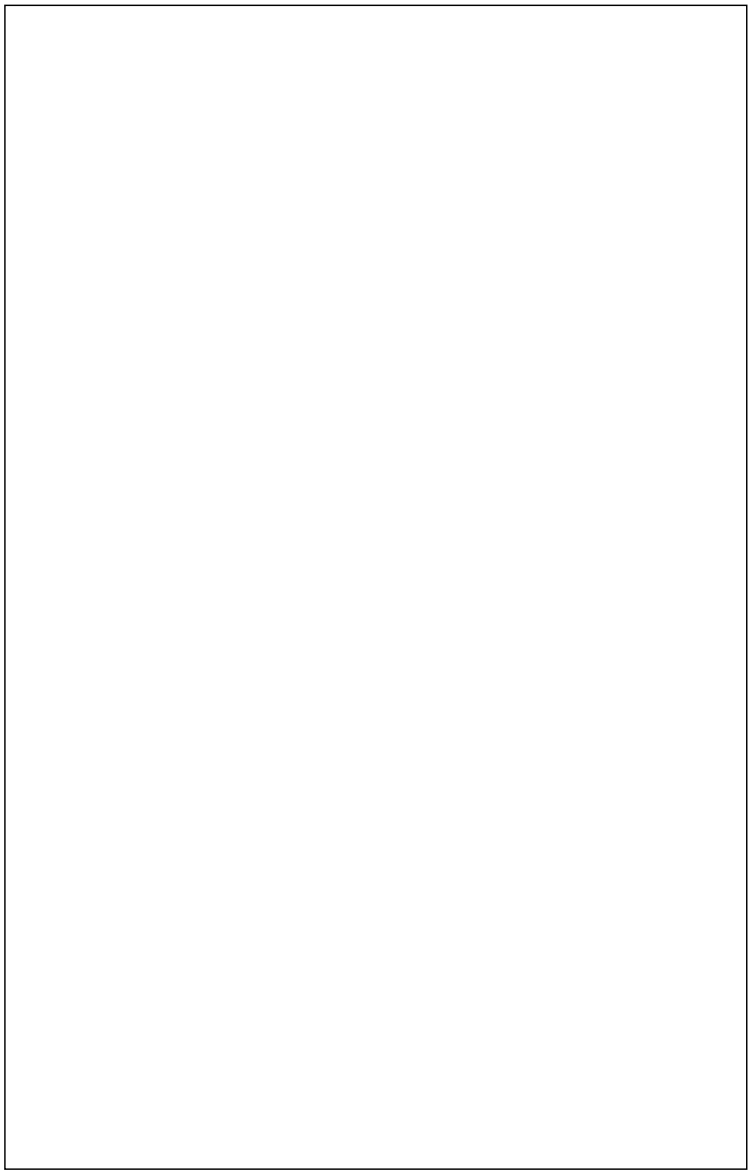


Figure 4: Functional debugging flow.

If the design is emulated, its specification is updated and the emulation is repeated. On the other hand, if the design is fabricated, error correction is expensive due manufacturing costs. In that case, built-in functional fault tolerant hardware resources might help overcoming some

errors. Finally, the designer iterates the described functional debugging process until certain level of confidence in the functional correctness of the design is not established.

5 Synthesis for Debugging

In this section we overview the key optimization aspects for integrating debugging resources into the initial design. The key issues in the design-for-debugging flow are recognized and their computational complexity is established. Problem descriptions are followed by a set of global least-constraining most-constrained optimization techniques developed to provide low cost support for functional debugging.

5.1 Synthesis Flow

As concluded in the previous section, the debugging flow relies entirely on the computation controllability and observability. On the other hand, the main property of the proposed approach is to keep the initial design specification intact. Therefore, a set of CAD tools is developed to add minimal hardware resources to the existing design specification so that it becomes fully controllable and observable. The determination and integration of the inserted debugging resources is performed by a number of fully modular tasks. First, the optimal cut is selected based on the analysis of the computation control data flow graph (CDFG). The goal is to select a set of variables which represent a CDFG cut such that all variables in the set are stored in minimal number of registers. In addition to that, the computation graph and timing bounds have to allow all variables in the cut-set to be output from the chip through a designated set of I/O pads within a single computation iteration.

The synthesis flow resumes searching for an optimal output scheduling of the cut-set variables. The schedule has to be determined so that the number of connections between the output pins and registers containing the cut-set variables is minimal. After cut-set variables are assigned and scheduled, the initial specification is updated with the set of resources which enable cut output. The chip is then ready to be fabricated or emulated.

5.2 Synthesis Optimization Problems Formulation and Their Complexity

The design flow points out to three fundamental problems which enable optimal integration of debugging resources with the initial design specification. In the first phase of the design-for-debugging flow we consider two fundamental problems: cut selection and checking whether the cut-set can be scheduled out from the chip. In the second phase, when the optimal cut-set is defined, the problem that arises is how to schedule the cut-set variables to specific I/O pins in such a way that minimal number of register-to-port connections are inserted into the initial design specification.

Cut Selection

The goal is to find a cut-set of variables S which bipartitions the control data flow graph into two subgraphs such that there exists no path in the initial CDFG which connects the two partitions and does not require computation of at least one variable in the set. In addition to that, the variable set has to be of minimal cardinality of registers that store the cut variables. Since even the problem of proving whether an arbitrary set of variables represents a CDFG cut is a polynomial problem with linear complexity, we choose problem transformation and partitioning as the fundamental approach to search the solution space. In order to explain this problem in detail and describe the developed heuristics, we introduce a set of definitions.

Definition 1. *A scheduled CDFG is given for each variable V_i: the control step C_i^{start} when it is created and the control steps C_i^{first}, C_i^{last} when V_i is used for the first and last time respectively.*

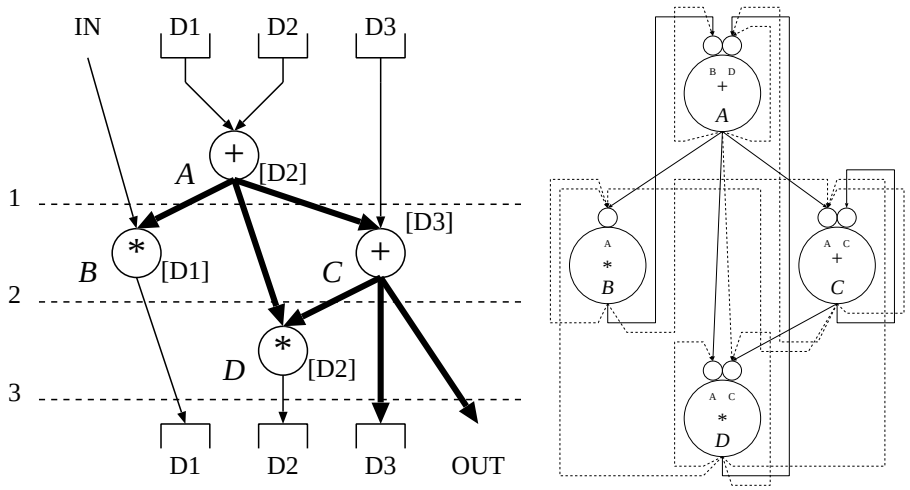
Definition 2. *An assigned CDFG has all its variables assigned to distinct registers.*

Definition 3. *The read life-time of variable V stored in register R begins at the control step when variable V is computed until the control step when variable V is overwritten by another variable W or it is recomputed.* In the latter case the read life-time of a variable spans across the entire iteration.

Definition 4. *Write life-time of variable V stored in register R starts at the control step when variable V is computed and ends at the control step when variable V is used for the first time.*

Definition 5. *An input sensitive graph - ISG - is a directed graph with multiple input nodes and a single output node. An arbitrary number of edges is connected to a single input or single output. At most one edge connects one output and one input.*

Definition 6. *A port is a set of K I/O pins. When variable V is assigned to port P, then V is output or input in its entirety through port P in one control step.*



(a) Scheduled and assigned CDFG. (b) Associate ISG.

Figure 5: Example of a control data flow graph and associated input sensitive graph.

Definitions are depicted using Figure 5. In Figure 5a, a scheduled and assigned CDFG is presented. Registers which store the variables are $[D1]$, $[D2]$, and $[D3]$. Exact clock cycles when operations are executed are also depicted. The read life time of a variable can be easily described using Figure 5a. For example, the last control step when variable stored in $[D1]$ is used, is $Cycle1$. Since no value is stored in $[D1]$ after $Cycle1$ until $Cycle3$, it is said that the read life-time of the variable stored in $[D1]$ spreads over the entire iteration. Variable's write life-time can be observed on the example of the variable stored in $[D2]$. It is computed at control step $Cycle_1$ and used for the first time in the next consecutive control step. Hence, its write life-time period includes only the control step $Cycle_2$. Figure 5b presents an example of an input sensitive graph.

Before we conclude with the algorithm description, note that only observability-related algorithms are presented further in the text. Algorithms that support controllability are absolutely identical with the exception that write life-times are used instead of read life-times. If the same cut is used for observing and controlling the computation, it should be determined by the controllable version of the proposed algorithms. For the sake of brevity and simplicity, the general

case, when different control and observation cut-sets may be employed, is not addressed in this paper.

The initial problem formulation is determined using the standard Garey-Johnson format [Gar79]:

PROBLEM: Optimal Cut-Set for Debugging.

INSTANCE: *Control data flow graph G with read life-times of its variables, variable-to-register assignments, P ports, and the associated set S of control steps when each port is busy.*

QUESTION: *Is there a subset of variables V such that each path in the control data flow graph G contains at least one variable $V_i \in V$, the cardinality of the set of registers that contains each variable $V_j \in V$ equals K, and there exists such schedule that each variable $V_j \in V$ can be output through P ports at control steps not included in S?*

The NP-completeness of this problem is proved by applying Karp's polynomial set of transformations on an arbitrary restricted instance of the initial problem, reduced to a well known NP-complete problem. In our case we restrict the problem of the optimal cut-set for debugging to FEEDBACK ARC SET (GT8, pp.192, [Gar79]) by assuming that no hardware sharing is possible, i.e. each variable V_i in the control data flow graph is stored in separate register R_i.

```
1  ISG = Construct_ISG(CDFG).
2  ISG = Input_Sensitive_Transitive_Closure(ISG, CDFG).
3  Perform search in the solution space for the optimal
   input-sensitive dominating set D in the ISG. Optimality of
   the set is described as minimal number of registers storing
   all variables in D. There has to exist a schedule such that
   all variables in D can be output through a given set of ports.
```

Figure 6: Pseudo-code for the optimal cut search algorithm.

We develop a global least-constraining most-constrained heuristic which constructs its solution based on the analysis of the ISG presentation of the initial CDFG. The pseudo-code of the heuristic is presented in Figure 6. The ISG is built from the CDFG according to the pseudo-code in Figure 7. An important step in the algorithm is the input-sensitive transitive closure operation performed on the ISG. This operation is described using the pseudo-code in Figure 8.

1 For each node $N_i \in CDFG$ 1.1 Create a node $M_i \in ISG$. 2 For each edge E_{N_i,N_j} directed from N_i to N_j 2.1 Create an input port $M_{i,m}$ for M_i, where m is the index of the input port. 3 For each edge E_{N_i,N_j} directed from N_i to N_j 3.1 Create an edge $E_{M_i^o,M_{j,m}}$ which connects M_i^o and $M_{j,m}$, where M_i^o is the output port of M_i, and $M_{j,m}$ is the m^{th} input of M_j.
Comment: Each primary input P_i of the $CDFG$ is ignored. Schedule and assign each ISG node (variable) as its parent $CDFG$ node (variable).

Figure 7: $Construct_ISG(CDFG)$.

An example how ISG is constructed can be observed from Figure 5. It shows a CDFG and its associated ISG. The dotted edges are added while applying the input-sensitive transitive closure procedure.

Using the definition of scheduled and assigned input sensitive graph, the initial problem can be reformulated into the following standard Garey-Johnson format:

PROBLEM: Optimal Input Dominating Set of an Input Sensitive Graph.

INSTANCE: *Input sensitive graph G with read life-times of its variables, variable-to-register assignments. P ports and associated set S of control steps when each port is busy.*

QUESTION: *Is there an input dominating set of variables V such that each input is covered with at least one variable in V, the cardinality of the set of registers that contains all variables from V equals K, and there exists a schedule such that all variables in V can be output through P ports at control steps not included in S?*

1 For each pair of edges $E_{M_a^o,M_{b,m}}, E_{M_b^o,M_{c,m}} \in ISG$ such that $E_{M_a^o,M_{b,m}}$ connects M_a^o to M_b and $E_{M_b^o,M_{c,m}}$ connects M_b^o to M_c 1.1 If the difference in control steps between the starts of read life-times of nodes (variables) A and C is less or equal than the total number of control steps in one iteration 1.1.1 Insert edge $E_{M_a^o,M_{c,m}}$ connecting M_a^o and $M_{c,m}$.

Figure 8: $Input_Sensitive_Transitive_Closure(ISG, CDFG)$.

The NP-completeness of this problem is proved by applying Karp's polynomial set of transformations on an arbitrary restricted instance of the initial problem, reduced to a well known NP-complete problem. In our case we restrict the problem of an optimal input dominating set of an input sensitive graph to GRAPH DOMINATING SET (GT2, pp.190, [Gar79]). The restriction simplifies the input sensitive graph ISG in such a way that each node has only one input and each variable $V_i \in ISG$ is stored in separate register R_i.

Preprocessing: Set of primary output variables $V_i^{output} \in ISG$ is static part of the cut.
Variables read-alive during the entire iteration are static part of the optimal cut.

1 Select a random subset of nodes $CS \in ISG$, such that
CS covers all node inputs in ISG. Set best solution $CS* = CS$.
1.1 If there does not exist a schedule such that CS can be output
through P ports at control steps not included in S go to step 1.
2 Repeat $GLOBAL$ times
2.1 Unselect random subset of nodes $\in CS$ such that at least one
node input remains uncovered.
2.2 Randomly select a subset of nodes $subCS$ from (ISG - CS) which covers
the uncovered set of inputs. Merge $subCS$ and CS.
2.3 If Cost(CS) < Cost($CS*$)
2.3.1 If there exists a schedule such that CS can be output through P ports set $CS* = CS$.
2.3.2 Repeat $LOCAL$ times
2.3.2.1 $CS+ = CS*$
2.3.2.2 Unselect random subset of nodes $\in CS+$ such that at least one
node input remains uncovered.
2.3.2.3 Randomly select a subset of nodes $subCS$ from (ISG - $CS+$) which covers
the uncovered set of inputs. Merge $subCS$ and $CS+$.
2.3.2.4 If Cost($CS+$) < Cost($CS*$)
2.3.2.4.1 If there exists a schedule such that CS
can be output through P ports set $CS* = CS+$.

Return: $CS*$ as the optimal input dominating set (optimal cut-set).

Figure 9: Algorithm that searches for the optimal input dominating set.

Although greedy heuristic algorithms can be employed, experimentation suggests that the following global probabilistic most-constrained least-constraining heuristic obtains the best results. The pseudo-code of the proposed heuristic technique is presented in Figure 9.

The objective function that evaluates the proposed cut-sets is run-time dependent. In the beginning, the algorithm favors solutions with as few as possible storing registers. As the search

progresses, the cost's dominating factor becomes the equilibrium of read life-time periods of all selected variables along the computation iteration. This approach enables more assignment freedom when the final variable-to-port scheduling is performed.

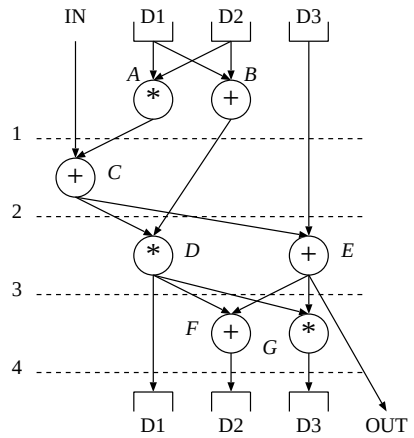


Figure 10: Example of an optimal cut-set.

The algorithm can be easily described using Figure 10 where several cuts can be proposed during the most-constrained least-constraining probabilistic search (e.g. $\{C, B\}$, $\{C, D\}$,...) but the one with the lowest cost is $\{D\}$. Note that the primary output E is always part of the optimal cut.

Cut-Set Output Scheduling

The chip has to be able to output all variables in the cut-set through a limited number of I/O ports within a single computation iteration. Since this procedure is invoked every time a candidate cut-set is found, we propose a most-constrained least-constraining heuristic technique to solve this problem. The problem is formulated using the following format:

PROBLEM: Output Scheduling of a Set of Variables in a Control Data Flow Graph.

INSTANCE: *Set of variables V, each with its read life-time. P ports and associated set S of control steps when each port is busy.*

QUESTION: *Is there a schedule such that all variables in V can be output through P ports at control steps not included in S?*

The NP-completeness of this problem is proved by applying Karp's polynomial set of trans-

formations on an arbitrary restricted instance of the initial problem, reduced to a well known NP-complete problem. In our case we restrict the problem of output scheduling of a set of variables in a CDFG to SEQUENCING WITH RELEASE TIMES AND DEADLINES (SS1, pp.236, [Gar79]) by selecting only those variables in V that are not in the set of state (delay) variables D.

The heuristic developed for this problem assigns variables to ports in the following greedy way. First, the constraint of each control step C_i in the scheduled and assigned CDFG is calculated as the number C_i^{var} of variables being read-alive during that control step. For each variable V_i, its constraint is computed as a sum of C_j^{var}, where j is in the set of control steps when V_i is read alive. Consequently, the most constrained variable is assigned to the least constraining control step. The process of computing constraints and scheduling variables to distinct control steps is iterated until all variables in the cut-set are not output scheduled. Pseudo-code of the proposed most-constrained least-constraining heuristic is presented in Figure 11.

1 Repeat until all variables are not scheduled 1.1 For each variable V_i 1.1.1 If V_i can be scheduled only in one control step C_j, schedule V_i to C_j and any port P_x available at C_j. 1.2 For each control step C_i 1.2.1 Set C_i^{var} as number of all variables which are read-alive at C_i. 1.3 Schedule variable V_i with $max(Cost(V_i)) = max(\frac{\sum_{j=AllControlSteps}(C_j^{var} \cdot ReadAlive(V_i, C_j)}{ReadLifeTime(V_i)-1})$ to control step C_k which has $min(C_k^{var})$ and any port P_x still available at C_k.
$ReadAlive(V_i, C_j)$ returns 1 if V_i is read-alive at C_j. Otherwise, it returns zero. $ReadLifeTime(V_i)$ returns the number of control steps for which V_i is read-alive.

Figure 11: Pseudo-code for the cut-set output scheduling heuristic.

The algorithm can be easily described using the example in Figure 12. There are 12 variables in the cut-set and three output ports which are all busy during control step C_4. The schedule is found using the described heuristic by sequentially assigning variables to ports as depicted. First, variables V_1, V_8, and V_{12} are scheduled to C_5, and V_{11} to C_3 since they do not have other choice. In the next step, since all ports are used in C_5, we schedule V_9 to C_1 and V_6 to C_3. Next, variable V_{10} is the most-constrained according to the formula in pseudo-code so we schedule it to its least-constraining control step C_2. Then V_7 and V_5 have no choice and have to be scheduled

at C_2. Finally, V_2, V_3, and V_4 are scheduled according to already described principles to C_1 and C_2.

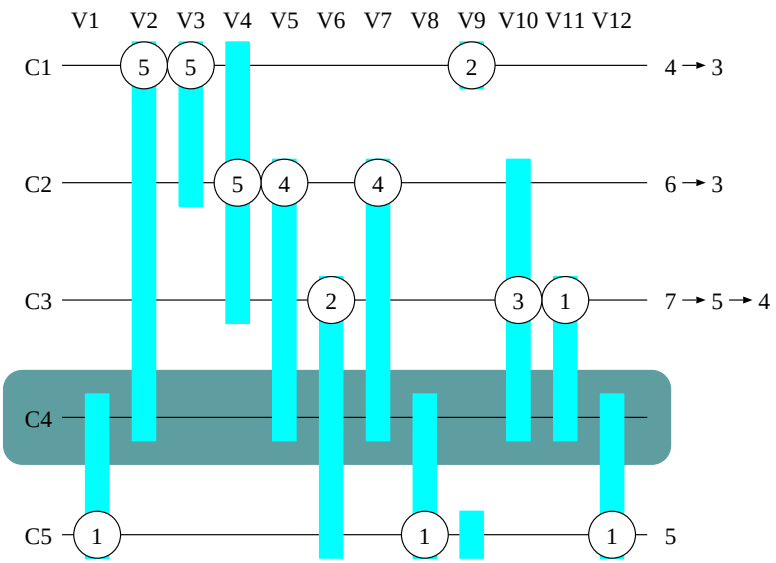


Figure 12: Example of output scheduling.

Variable-to-Port Scheduling

The second phase of the synthesis for debugging process has as an input the selected cut-set from the first phase and the information about the read life-times of each variable as well as its storing register. The available number of output ports is also available. The key design question is to assign each cut-set variable to a specific output port in such a way that the number of connections between registers and I/O ports is minimal. We present the problem in the standard Garey-Johnson format:

PROBLEM: Optimal Output Scheduling of a Set of Variables in a Control Data Flow Graph for Debugging.

INSTANCE: *Set of variables V, each with its read life-time and designated register. P ports and associated set S of control steps when each port is busy.*

QUESTION: *Is there a schedule such that all variables can be output from the chip through P ports at control steps not included in S, and the cardinality of register-port connections equals K?*

If the optimization demand in the problem is ignored, the proof that this problem is NP-

complete is equivalent to the NP-completeness proof for the scheduling problem described in the previous subsection.

The developed algorithm for this problem first partitions the problem two fully modular optimization subproblems. Initially, register to port assignment is performed such that optimization requirements are met and variables of the proposed cut-set can be output through all designated ports. In the second phase, for each port, all variables that are assigned to it are scheduled for output.

For the assignment problem we develop a global most-constrained least-constraining heuristic described using the pseudo-code in Figure 13.

1 For each control step C_i set C_i^{var} as number of variables which are read-alive during C_i.
2 Repeat until all variables are assigned to ports
2.1 Assign variable V_i with

$$max(Cost(V_i)) = max(\sum_{j=AllControlSteps} \frac{C_j^{var}}{ReadLifeTime(V_i)-1})$$

to port P_x which has

$$min(Cost(P_x)) = min(\sum_{j=AllVariables} \frac{ReadLifeTime(V_j) \cdot Assigned(V_j,P_x)}{1+Numberofregistersalreadyassignedto P_x}).$$

Figure 13: Pseudo-code for variable-to-port matching heuristic.

The heuristic iteratively assigns most-constrained variables (registers) to least-constraining ports. Objective function that quantifies the constraint of variable V_i is given as:

$$Cost(V_i) = \sum_{j=AllControlSteps} \frac{C_j^{var}}{ReadLifeTime(V_i)-1}$$

where $ReadLifeTime(V_i)$ returns the number of control steps for which V_i is read-alive. Similarly, the objective function that quantifies the constraint of port P_x is:

$$Cost(P_x) = \sum_{j=AllVariables} \frac{ReadLifeTime(V_j) \cdot Assigned(V_j,P_x)}{1+NumberOfRegistersAlreadyAssignedToP_x}$$

where $Assigned(V_j, P_x)$ returns 1 if V_j is already assigned to P_x. Otherwise, it returns 0.

In the conducted experiments we used a probabilistic version of the described heuristic which iteratively generated randomized solutions. The solutions were guided by the described objective functions augmented with a random offset.

The problem of assigning a set of variables with their read life-times to a single port is equivalent to the problem of **maximum bipartite matching** (pp. 601, [Cor90]) which can be efficiently solved in polynomial time. A bipartite graph G for each port P is constructed. The first partition of G contains a node for each control step in the computation iteration for which port P is not busy. A node in the other partition is created for each variable assigned to port P. Edges are drawn between each node which represents variable V and all nodes in the first control-step partition associated with control steps for which V is read-alive. To efficiently solve this problem we use Ford-Fulkerson method which runs in $O(VE)$, where V is the number of vertices and E is the number of edges in graph G [Cor90].

6 Experimental Results

In order to evaluate the developed debugging technique and synthesized algorithms and tools, we applied the new design-for-debugging approach on several real-life designs [Rab91]. Table 1 shows the experimental results. The first column indicates the name of the evaluated design. For each design we conducted four different hardware overhead experimentations. The first and second row stand for non-optimized design specification, whereas the third and fourth row stand for highly optimized designs. We used HYPER [Rab91] as a behavioral level synthesis tool. Each design was synthesized using the initial and HYPER optimized behavioral specification. Optimization in HYPER was performed using powerful scripts for speed optimization [Hon97].

Next five columns describe the behavioral structure of the design. In order to explore the impact of different timing constraints on the debugging overhead, for each design, optimized and non-optimized, two studies were performed. In the odd rows we evaluate the case when the number of available control steps is equal to the length of the critical path, while in even rows designs are given twice as many control steps. Then, the length of the critical path in the computation is shown followed by the total number of variables. Note that optimized designs have much shorter critical paths and therefore, higher sampling rates. In addition, the number of registers required to accomplish the computation as well as the number of boundary variables are presented in the sixth and seventh column respectively.

The performed design-for-debugging analysis generated complete cuts displayed in columns

eight and nine. First, number of variables included in the cut is presented and then the number of register-to-port connections required to schedule the injection and dispensing of the selected cut.

Design		Structure					Complete Cut		
Description	Hyper optimized	Available control steps	Critical path	Variables	Registers	States	Variables	Registers	Ports added
Linear GE Controller 1	NO	12	12	48	19	5	8	3	0
	NO	24	12	48	23	5	13	1	0
	YES	6	6	48	27	5	5	5	1
	YES	12	6	48	26	5	8	4	0
Wavelet Filter	NO	16	16	31	20	15	15	1	0
	NO	32	16	31	20	15	15	1	0
	YES	1	1	31	31	15	15	15	15
	YES	2	1	31	31	15	15	15	7
Modem Filter	NO	10	10	33	16	8	12	2	0
	NO	20	10	33	15	8	12	1	0
	YES	4	4	47	29	8	8	6	2
	YES	8	4	47	27	8	11	4	1
Digital to Analog Converter	NO	132	132	354	167	74	76	3	0
	NO	132	264	354	171	74	77	2	0
	YES	5	5	398	189	74	76	29	18
	YES	10	5	398	178	74	76	28	8
8th Order Continued Function IIR Filter	NO	18	18	35	19	8	8	1	0
	NO	36	18	35	19	8	8	1	0
	YES	4	4	49	30	8	8	6	2
	YES	8	4	49	29	8	10	5	1
Long Echo Canceller	NO	2566	2566	1082	1056	1024	1027	5	0
	NO	5132	2566	1082	1061	1024	1027	4	0
	YES	1088	1088	1107	1064	1024	1028	6	0
	YES	2176	1088	1107	1059	1024	1027	5	0

Table 1: Experimental results: low hardware overhead for functional debugging.

In most cases, including all designs obtained without optimizations using transformations, only one already available I/O port was sufficient to support full observability and controllability. However, in several cases, we needed to add ports to the initial design specification in order to obtain with feasible I/O schedules. The number of added ports is presented in the last column. Note that port addition occurred only for highly optimized designs with exceptionally high sampling rates. For example, the optimized wavelet filter and digital to analog converter have complete cuts with 15 and 78 variables which had to be dispensed in 1 and 5 control steps respectively. Under the specified control circumstances, scheduling was infeasible due to the drastically reduced

number of control steps for which I/O ports were not busy conveying primary input and output data. This inevitably resulted in adding 15 and 18 ports to the optimized initial specifications of the wavelet filter and digital to analog converter respectively. Most importantly, such drastically optimized designs are rarely met in real-life applications.

To summarize, as an example, the non-optimized modem filter, shown in row nine, computes 33 variables, stores them in 16 registers, has 8 states between two iterations, has a critical path of 10 variables, and has 10 control steps per single iteration, was provided full design controllability and observability with a complete cut which consisted of 12 variables. Only two registers were used to store these 12 variables which resulted in low hardware overhead. No additional ports were required to support cut-set scheduling.

7 Conclusion

The run-time of a design simulation results in several orders of magnitude slower functional execution than chip emulation or ASIC implementation. Design emulation and implementation significantly restrict design controllability and observability during functional debugging. We introduced a new cut-based functional debugging paradigm which integrates design emulation and simulation in such a way that advantages of both domains are fully utilized. These features require a specific design approach to enable high speed debugging with complete observability and controllability. We identified the associated optimization synthesis tasks, established their computational complexity, and developed global heuristics to solve them. The experimental results clearly indicate the power of the new approach and debugging tool on a number of real-life design examples with minimal hardware overhead.

References

[Bal69] R.M. Balzer, "Exdams - Extendable debugging and monitoring systems", Proceedings of AFIPS Spring Joint Computer Conference, Washington, D.C., pp. 125-134, 1969.

[Bea83] B. Beander, "VAX DEBUG", SIGPLAN Notices, Vol.18, No.8, pp. 173-179, 1991.

[Bro92] G. Brooks, et al. "A new approach to debugging optimized code", SIGPLAN Notices, Vol.27, No.7, pp.1-11, 1992.

[Chi94] P.C. Ching, Y.C. Cheng, M.H. Ko, "An In-Circuit Emulator for TMS320C25", IEEE Transactions on Education, Vol.37, No.1, pp. 51-56, 1994.

[Cor90] T.H. Cormen, C.E. Leisserson, R.L. Rivest, "Introduction to Algorithms", The MIT Press, Cambridge, MA, 1990.

[Cou88] D. Coutant, et al. "Doc: A Practical Approach to Source-level Debugging of optimized code", Proc. of SIGPLAN Conference on Programming Language Design and Implementation, ACM , New York, NY, pp. 125-134, 1988.

[Cro75] R.E. Crochiere, A.V. Oppenheim, "Analysis of Linear Networks", Proc. of IEEE, Vol.63, No.4, pp. 581-595, 1975.

[Gai85] J. Gait, "A debugger for concurrent programs", Software Practice and Exp., Vol.15, No.6, pp. 539-554, 1985.

[Gue97] L.M. Guerra, Rockwell, Personal Communication, 1997.

[Hen82] J. Hennessy, "Symbolic Debugging of Optimized Code", ACM Transactions on Programming Languages and Systems, Vol.4, No.3, pp. 323-344, 1982.

[Hon97] I. Hong, D. Kirovski, M. Potkonjak, "Potential-Driven Statistical Ordering of Transformation", accepted for publication, 34th Design Automation Conference, Annaheim, CA, USA, 1997.

[Int86] Intel Corporation, "Intel 80386 Programmer's Reference Manual", Intel, Santa Clara, CA 1986.

[Kau88] S. Kaufer, R. Lopez, S. Pratap, "Saber-C: an interpreter-based programming environment for the C language", Proceedings of the Summer 1988 USENIX Conference, pp. 161-71, 1988.

[Ku97] D. Ku, Escalade, Personal Communication, 1997.

[Lee87] E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", IEEE Transactions on Computers, Vol. 36, No. 1, pp. 24-35, 1987.

[Nag94] J. Naganuma, et al. "High-Level Design Validation Using Algorithmic Debugging", EDAC-94, pp. 474-480, 1994.

[Nar91] S. Narayan, et al. "System Specification and Synthesis with the SpecCharts Language", ICCAD, pp. 266-269, 1991.

[Phi84] D. Phillips, "Advanced in-circuit emulation design", Mini/Micro Southwest, LA, CA, USA, p. 8/2/1-4, 1984.

[Pot95] M. Potkonjak, S. Dey, K. Wakabayashi, "Design-for-Debugging of Application Specific Designs", ICCAD, San Jose, CA, pp. 295-301, November 1995.

[Pow94] G.S. Powley, J. E. DeGroat, "Experience in Testing and Debugging the i960 MX VHDL Model", VHDL International Users Forum, pp. 130-135, 1994.

[Rab91] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures", IEEE Design and Test of Computers, Vol.8, No. 2, pp. 40-51, June 1991.

[Ros95] M. Rosenblum, S.A. Herrod, E. Witchel, A.Gupta, "Complete computer system simulation: the SimOS approach", IEEE Parallel and Distributed Technology: Systems and Applications, Vol.3, No.4, pp. 34-43, 1995.

[Sai93] A. Saini, "Design of the Intel Pentium (tm) Processor", ICCAD, pp. 258-261, 1993.

[Sha83] E.Y. Shapiro, "Algorithmic Program Debugging", The MIT Press, 1983.

[Sta91] R.M. Stallman, et al. "A guide to the GNU source-level debugger", GDB v4.0, Free Software Foundation, Cambridge, MA, 1991.

[Wak96] K. Wakayabashi, NEC, Personal Communication, 1996.

[Wyn86] P. Wynn, "In-circuit emulation for ASIC-based designs", VLSI Systems Design, Vol.7, No.10, pp. 38-45, 1986.

[Yan95] L. Yang, et al. "System design methodology of UltraSPARC-I", Design Automation Conference, pp. 7-12, 1995.

[Ziv96] V. Zivojnovic, H. Meyr, "Compiled HW/SW co-simulation", Design Automation Conference, pp. 690-695, 1996.