Design-For-Debugging of Application Specific Designs

ABSTRACT

Debugging can be defined as a process of identifying and correcting errors made during functional specification by observing the functional behavior of the design. It often dominates time and cost of integrated circuits and system development. In modern ASIC designs debugging is particularly difficult problem due to very limited controllability and observability of intermediate variables during the operational mode of the design.

We address the problem of considering debugging requirements during high level synthesis by providing low-cost hardware support and scheduling and assignment methods for ensuring controllability and observability of the user specified variables. Three key conceptually new design ideas that enable efficient debugging are developed: pipelining of debugging variables for improving their scheduling and assignment freedom, use of I/O buffers for improving resource utilization of I/O pins, and increase the debugging periodicity to satisfy all the debugging requirements.

The provably optimal bounds for the maximum cardinality of the set of controllable and observable variables for a given design specification are derived. A simple, polynomial time complexity synthesis algorithm for achieving the bounds is developed. The minimization of hardware overhead gives rise to an interesting combinatorial optimization problem which is solved using a non-greedy heuristic algorithm.

The effectiveness of the proposed Design-for-Debugging approach is demonstrated on several examples.

1.0 Introduction

1.1 Motivation

It is well-known that functional debugging usually dominates the cost of design development. For example, the designers of a modern processor reported that functional debugging took almost 240 man-months or more than 40% of the overall effort [Nar93]. We conducted an informal study at several industrial and design research places which also clearly indicated that most often designers usually spend most of their time debugging the functionality of the design.

The need for debugging consideration and support during high level synthesis is additionally emphasized by the current and future technology trends. With feature scaling in design rules, the areas increases quadratically, while the number of pins only linearly. Also, pins are interfaces to the outside environment and intrinsically do not scale as well as the number of transistors with new generations of implementation technologies. Consequently, the ratio of the number of pins to the computational resources on a chip keeps decreasing. For example, while the area of the state of the art microprocessor, Intel P6, increased by a factor of more than three order of magnitudes compared to the state-of-the-art processor 25 years back, Intel 4004, the number of pins in the same processor increased by only one order of magnitude [Min94].

This trend clearly indicate that debugging is bound to become even a more acute problem, since the percentage of observable/controllable variables in designs has been steadily decreasing. Debugging is in particular a difficult activity when real-time full-custom ASIC designs are targeted, due to the strict timing constraints and a lack of flexibility during execution.

We address the design-for-debugging problem in high level synthesis. We have four main objectives of the research presented in this paper:

1. to formalize an intuitive notion of ASIC debugging so that it can be treated as a design and CAD activity;
2. to identify key design and high level synthesis principles which support debugging;
3. to developed efficient high level synthesis algorithms for optimization problems associated with ASIC debugging; and
4. to give an impetus for creation of compressive design-for-debugging and synthesis-for-debugging methodologies.

1.2 Informal Introduction to Design-for-Debugging

Debugging, as any other highly creative human activity, can not be cleanly and uniquely abstracted. Debugging can be defined as a process of detecting, diagnosing, and correcting errors in the specification of an ASIC implementation. Error is any discrepancy between desired and realized behavior of the specification or the design.

The debugging process can be divided into three phases [Ren89]. The first step is error detection, in which the designer discovers that a program (design) does not function correctly for a particular input. The second phase is error diagnosis in which the programmer/designer identifies the statement or the section of the code which is causing the incorrect behavior. The third step is error correction, in which the faulty section or the statement responsible for the observed fault is replaced by the corrected section.

In the research presented in this paper, we concentrate on the error detection phase. Even when only this phase is considered there can be numerous different strategic approaches. However, it is widely accepted that providing simultaneous controllability and observability of as many as possible variables of the program under execution immensely facilitates the debugging process. Note that although our goal is error detection, enhanced observability and controllability of the computation's variables also greatly helps error diagnosis.

Therefore, we will informally define the design-for-debugging problem in the following way. Given is an ASIC design. The design is fully specified: the control-data flow graph (CDFG) of the computation, timing constraints in terms of the available number of control steps, and the schedule and assignment of each operation, variable and constant, and data transfer are given. Furthermore, a set of desired controllable debug variables (write variables) and observable debug variables (read variables) is specified by the user. The goals of a design-for-debugging (DfD) technique is to modify design such that the set of desired debug variables are made controllable/observable, satisfying given timing constraints, while adding a minimal additional hardware.

The key constraint of the DfD is that functionality of the design should not be altered in any way, except when requested by the user debug variables which should be altered by the user provided values. The key idea is to use available I/O for reading and writing debug variables in control steps when they are not used by the design variables. As shown in Table 1, in many examples I/O pins are very rarely used.

Task	# of operations	critical path	# of I/O variables	I/O var/ #of oper.	I/O var/ crit. path
GE Controller1	48	5	2	0.04	0.40
GE Controller2	108	7	2	0.07	0.29
Honda Controller1	97	8	2	0.08	0.25
Honda Controller2	67	7	2	0.03	0.29
Wavelet filter	31	14	2	0.06	0.14
Low Pass Filter	32	8	2	0.06	0.25
BandPass Filter	38	10	2	0.05	0.20
High Pass Filter	42	11	2	0.05	0.18
BandStop Filter	30	11	2	0.07	0.18
8X8 DCT	46	7	2	0.04	0.29
DAC	354	48	2	0.006	0.04
modem	227	44	2	0.009	0.05
adaptive modem	200	46	2	0.01	0.04
Large Controller	324	19	6	0.02	0.32
LMS audio formatter	464	61	3	0.006	0.05
Echo-Canceller	212	56	3	0.01	0.05

Table 1: Ratios the number of I/O variable vs. total number of variables and vs. the length of the critical path as measured in control steps. Data clearly indicates that in almost all, and in particular in large examples, I/O pins are rarely used. during one iteration of the computation.

We conclude this section by pointing out key differences between testing and debugging. Both debugging and testing are aimed at enhancing controllability and observability of the design. However, there is a number of differences which makes debugging unique and only very distantly related to testing.

The key difference is that while testing targets controllability and observability in the test mode, debugging targets enhanced controllability and observability during the functionally correct mode of operation of the design. Furthermore, while testing has as the goal to make all hardware elements of the design (e.g. all execution units, all registers, and complete control logic) controllable and observable, debugging concentrates only on a selected set of registers at the selected control steps in which user-specified debugging variables are stored. The controllability and observability of those registers is required only during particular control steps of each iteration.

Also, debugging an ASIC design usually requires that all controllable variables are set simultaneously and that all observable variables are simultaneously obtained. This makes debugging, in some sense, significantly more difficult than manufacturing testing.

Finally, note that similar differences as between testing and the considered error detection phase of debugging are between silicon diagnosis and the error diagnosis phase of debugging.

1.3 What is New?

To the best of our knowledge this is the first attempt to study debugging using high level synthesis and in general CAD techniques. This is also the first effort which addresses debugging of custom ASIC design in systematic way.

We formulated the debugging of ASIC designs as an optimization CAD problem. We developed three key conceptual techniques to enable effective and inexpensive debugging: I/O buffers, functional pipelining of debugging variables, and multi iteration debugging. I/O buffers provide easy and powerful mechanism for enabling 100% utilization of all I/O pins for debugging purposes, functional pipelining of variables essentially removes all scheduling constraints from debugging variables by replacing them with register allocation constraints, and the concept of multi iteration debugging significantly enhances the application range of the developed techniques. For additional reduction of debugging hardware overhead, we applied life-time splitting of debug variables, which reduce both memory storage and interconnect hardware requirements.

From the optimization point of view, for majority problems optimal polynomial time algorithms has been developed. We prove that the minimization of debugging hardware using splitting life-time of debugging variables is computationally intractable problem, and propose a novel maximum entropy based heuristic.

1.4 Paper Organization

The rest of the paper is organized as follows. In the next section we review related work. Section 3 introduces all preliminaries, and Section 4 introduces the design-for-debugging process. We present the algorithm for minimization of debugging hardware using life-time spliting of debugging variables in Section 5. After presenting experimental results and outlining potential future directions in Sections 6 and 7, we summarize the DfD method in Section 8.

2.0 Related Work

Debugging is as old as building of digital electronic computing systems. Among numerous speculations concerning the initial use of the word "bug" in the context of computer science, the most popular is one by

Grace Hoper which reports how an actual bug was found in one of the relays of the Harvard Mark II computer impeding correct functioning of the early computer [Hop81]. However, recent evidence indicates that most likely the use of the word "bug" in the language of computer engineering predates the use related to malfunctioning of the Mark-II computer [Tro89, Coh94].

Consequently, debugging has been recognized as a crucial design and compilation activity. However, initially it was relatively rarely addressed in research literature due to its exceptional conceptual complexity [Bal69, Hen82, Zel83]. Recently, the situation changed and the importance of debugging has been documented by a great deal of research in several research and development communities on several levels of abstraction, including compilers, computer architecture, and databases.

By far the most comprehensive treatment of debugging has been conducted in the software compiler domain. Several debuggers, such as VAX DEBUG [Bea83], Dbx [Lin90], GDB [Sta91], and more recently Purify [Has92] has been widely used.

Majority of efforts related to debugging in the compiler domain has been dedicated to symbolic level debugging on uniprocessor computers. It has been noticed that debugging often significantly slows done execution of the program. For example, it has been reported that ANSI C using Saber-C executes approximately 200 times slower than compiled code [Kau88]. This observation directed significant efforts toward development of efficient support mechanisms for debugging. The proposed techniques can be divided in three groups: specialized hardware, virtual memory, and software methods. Few processors, e.g. the Intel 80386 [Int86] and the MIPS 4000 [Kan92], provide support for debugging using monitor registers. Only one general purpose processor, TRON-based TX1 provides an elaborate hardware support for debugging [Miy88]. Virtual memory-based approaches protect all pages indicated by the debugger and detect when programs attempts to write to those pages [Bea83, Sul91]. Finally, software based methods modify the program to check the target location of all write instructions [Kes90, Has92, Wah92, Wah93].

Currently the major emphasis in the software compilation debugging domain is on integration between optimization technique based on transformations or aggressive scheduling techniques such as speculative scheduling and debugging process [Cou88, Coh91, Mil91, Bro92, Shu93].

Numerous controllers and DSP processors are supported by in-circuits emulators which enable an efficient debugging process. An in-circuits emulator for 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) of the microprocessor, with the extension that the internal state and operation of the emulated processor is fully observable and controllable by the user of the emulator [Chi94]. While some in-circuits emulators are provided by manufacturing houses [Tex83, Mic88], a number of in-circuits emulator are build by developers of applications [Raf84, Chi94]. The major drawback of in-circuit emulation is high cost.

An interesting alternative to in-circuit emulation is used sometimes during development of new microprocessors. The processor model is ported onto a logic design hardware model comprising of an array of rapid prototyping modules. This way both high speed execution and complete observability/controllability of all registers is provided. For example, this methodology has been used during development of Intel's Pentium processor [Sai93].

Debugging has been also addressed in several other domains. For example, in the data base literature, debugging of active data bases received considerable attention [Sul91], in theoretical computer science literature debugging is considered under the powerful paradigm of self-checking programs, and in operating systems performance assertion checking is an approach to automate the debugging of performance properties of complex computing systems [Per93].

In the CAD domain recently Powley and De Groat developed a VHDL model for an embedded controller [Pow94]. The model supports debugging of the application software. Also, Naganuma et al. [Nag94] combined structured analysis approaches [Har90, Nar91] with algorithmic debugging techniques from logic programming [Sha83] to speed-up design validation process.

3.0 Design-for-Debugging: Preliminaries and Problem Formulation

In this section we present all the essential assumptions for introducing and developing our approach for design-for-debugging. In particular, we describe the assumed computational and hardware models, and provide a description of targeted debugging process, requirements, and goals. We conclude the section by explicitly stating the considered design-for-debugging problem.

3.1 Computational and Hardware Model

We assume the synchronous data flow model of computations [Lee87] which is widely used in computationally intensive applications, such as image and video processing, multimedia, speech and audio processing. The selected computational model has two important implications for the design-for-debugging approach. First, it states that computation is conducted on infinite stream of data implying a need for periodic controllability and observability of variables in each iteration. Second, it implies static compile-time scheduling and assignment and full predictability of the earliest and the latest time when a particular debugging variable can be observed or controlled.

We do not put any restriction on the interconnect scheme of the assumed hardware model at the register-transfer level, registers may or may not be grouped in registers files, and each hardware resource can be connected in an arbitrary way to other hardware resource. It is important to note that all techniques and algorithms are directly applicable on other hardware models, although they do not in some cases explore all advantages provided by more restricted models which usually produce more compact layouts. We considered two different types of input/output mechanisms. In the first type each pin can be used to both input and output data. In the second scenario, a pin can be used exclusively as either an input or an output unit. While the first type of I/O pins provides higher flexibility, its hardware realization is more expensive.

3.2 Design-for-Debugging Preliminaries and Problem Formulation

The four key debugging assumptions are the following.

1. The design is fully specified and its functionality and realization should not be disturbed by the debugging process, except for bringing the user specified values to the controllable variables. A fully specified design is one where each operation, variable, and data transfer is scheduled and assigned to a particular hardware resource.
2. All controllable/observable variables are known at compile/synthesis time. As we already stated one of our key debugging assumption is that the user provides the list of variables whose controllability and observability facilitates debugging. Each variable either needs to be only controllable or only observable

or both simultaneously controllable and observable. This is the standard debugging assumption for the error detection phase in essentially all debugging tools. Usually debugging variables are one which are states in the functionality of the computations which denote boundaries between successive program or internal loop iterations.

3. For proper support of debugging, all controllable and observable variable should be simultaneously controllable and observable.

4. During design-for-debugging, we allocate additional debugging hardware to t satisfy all (or as many as possible) debugging requirements. The goal is, of course, to add as little as possible hardware. In particular, we do not allow increase in the number of I/O pins, since this is the hardware constraint which usually dominates other hardware constraints in modern designs.

The design-for-debugging problem can be summarized as follows. Given a design and a list of debugging variables. Add as little as possible additional hardware resources and schedule and assign the desired debugging variables and associated data transfers so as to satisfy all the debugging requirements.

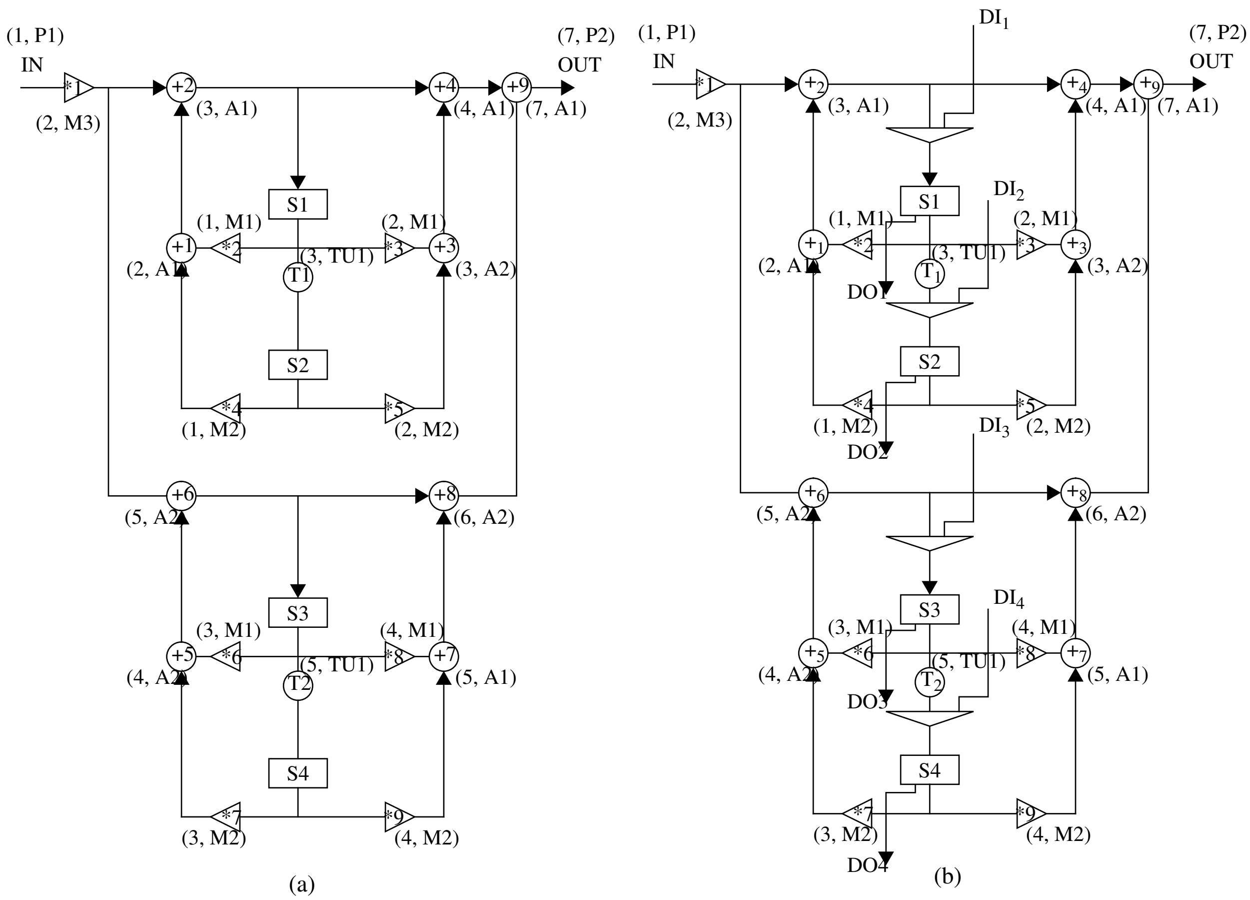
4.0 The Design-for-Debug Process

Consider the Control Data Flow Graph (CDFG) of the computation of a 4th order IIR parallel filter shown in Figure 1(a). The example consists of nine additions represented by +1, ... , +9, nine multiplications represented by *1, ... , *9. The state variables, shown by $S1$, ... , $S4$, denote the state of the computation at the end of the present iteration, to be used in the next iteration. The nodes T1 and T2 represent transfer operations, $(S2 <-S1)$ and $(S4 <-S3)$.

Suppose the designer-specified schedule and assignment of the nodes is as shown in Figure 1(a). The schedule satisfies a performance constraint of seven clock cycles, and uses a minimal set of execution units, two adders ($A1$ and $A2$) and two multipliers ($M1$ and $M2$). For instance, the operation +2 is scheduled in control step 3, and assigned to be executed in adder $A1$, shown in Figure 1(a) by the ordered pair (3, $A1$). Similarly, the transfer operation $T1$ is scheduled in control step 3, and assigned to the transfer unit $TU1$, as shown in Figure 1(a).

Associated with every input (output) variable of the design is an input (output) operation. Similar to scheduling/assigning other operations, an input (output) operation has to be scheduled in a clock cycle in which an available input (output) pin resource can be used to write in (read out) the variable from (to) the environment. For instance, for the CDFG shown in Figure 1(a), the input operation associated with variable IN has been scheduled in clock cycle 1, and assigned to an input pin resource $P1$. Similarly, the output operation of variable *OUT* has been scheduled in clock cycle 7 and assigned to output pin $P2$. Consequently, the specified design has one input pin and one output pin. In the rest of the paper, an input (output) operation of data to (from) an input (output) variable will be referred to by the name of the variable itself.

Without any design-for-debugging, to debug the design, the designer can only write to the primary input *In*, and read from the primary output variable *Out*. To make the design easier to debug, suppose the designer wishes to be able to write and read the state variable $S1$, $S2$, $S3$, and $S4$ during debugging. The ability to write and read the state variables would enable the designer to control and observe the state of the computation after every iteration. Consequently, for the Design-for-Debugging (DfD) technique to be described in this paper, the **debug write and read variables** are {$S1$, $S2$, $S3$, $S4$}, and the **debug requirements** are {WR($S1$),



	ASAP	ALAP
DI1	1	0
DI2	1	0
DI3	1	2
DI4	1	2
In	1	1
DO1	4	7
DO2	4	7
DO3	6	7
DO4	6	7
Out	7	7

(c)

Figure 1: (a) Original CDFG of the 4th order IIR parallel filter, showing the user-specified schedule and assignment to satisfy the available time constraint of 7 control steps, (b) Modified CDFG after incorporating the debug I/O operations, (c) Interval Table of the I/Os

WR($S2$), WR($S3$), WR(S4), RD($S1$), RD($S2$), RD($S3$), RD($S4$)}. Note that in this case, the debug write variables are the same as the debug read variables; but, in general, this may not be the case.

4.1 Incorporating the Debug Requirements

To incorporate the desired debug requirements, the original CDFG in Figure 1(a) is modified to the CDFG of Figure 1(b), with the desired input/output operations added. To satisfy the debug write requirements, each debug write variable S_i will be set to a new **Debug Input variable** DI_i when "Debug = 1" and to its original source otherwise. The introduced triangular nodes represent conditional statements/switches, which can be implemented as multiplexor nodes in the actual implementation. For example, in the new CDFG, the following input operation is incorporated to satisfy WR($S1$):

If (Debug)

then $S1$ <- Out(+2);

else $S1$ <- DI_1;

In general, a separate input variable DI_i and its input operation are needed for each debug write variable DW_i, so that each DW_i can be written independently, DW_i <- DI_i, during the debug mode. Similarly, for each debug read variable DR_i, an output variable DO_i and its output operation is needed to accomplish the debug read DO_i <- DR_i. Note that the debug requirement of independent write (read) of each debug variable is very different from a typical testing requirement, where each variable whose controllability (observability) needs to be assigned can be written from (read to) the same, even existing, input (output) variable.

4.2 Satisfying the Debug Input/Output Operations

After the original CDFG has been modified to incorporate the debug requirements, the next step is to schedule and assign the input/output operations, so as to satisfy the specified clock cycle and input/output pin constraints. Scheduling a given debug input/output operation is constrained by the following two factors:

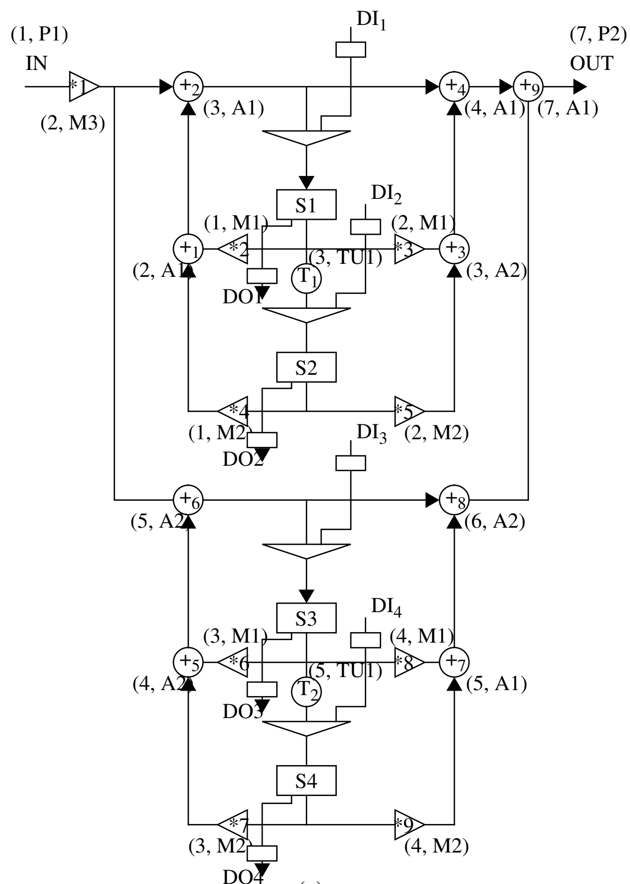
1. The As Soon As Possible (ASAP) and the As Late As Possible (ALAP) control steps that the write/read debug variable can be written/read, and
2. The availability of an input/output pin in that clock cycle.

Let *prod(X)* be the set of control steps in which a variable X in the CDFG is possibly produced. (X can be produced in multiple control steps if it is set in different branches of a conditional). Let *req(X)* be the set of control steps in which a variable X is required. Let *min{prod(X)}*, *max{prod(X)}* *min{req(X)}*, and *max{req(X)}* denote the latest and earliest control steps a variable X is produced and required respectively. For example, in Figure 1(a), *min{prod(S1)} = max{prod(S1)} = 3, min{req(S1)} = 1, max{req(S1)} = 3.*

Consider a debug input operation for the debug write X <- DI, where X is the debug variable to be written, and DI is the input variable that will be used to write X. The debug variable X has to be written anytime from control step 1, and before the earliest control step X is required. Hence, the (ASAP, ALAP) control steps for the input operation associated with input DI are:

$$[ASAP, ALAP]\ (DI) = (1, min\{req(X)\} - 1) \quad \textbf{(EQ 1)}$$

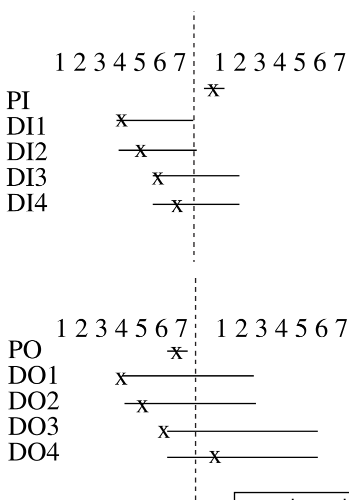
Similarly, the ($ASAP$, $ALAP$) control steps for the output operation associated with the debug read DO <- Y,



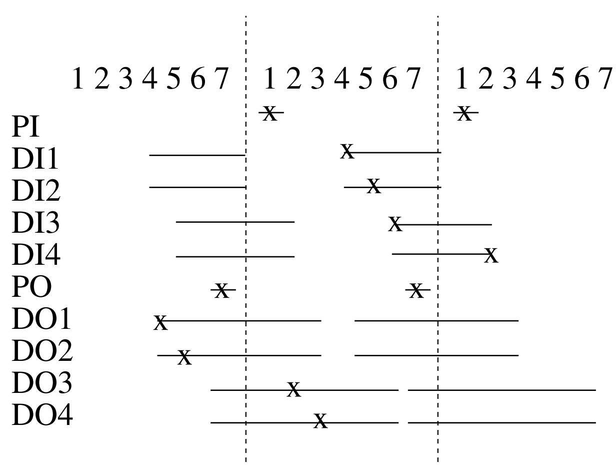
(a)

	ASAP	ALAP
DI1	4@1	0
DI2	4@1	0
DI3	6@1	2
DI4	6@1	2
PI	1	1
DO1	4	3@-1
DO2	4	3@-1
DO3	6	5@-1
DO4	6	5@-1
PO	7	7

(b)



(c)



x : input/output operation scheduled

(d)

Figure 2: (a) Modified CDFG showing pipelining of debug variables, (b) Interval Table, (c) Interval Graph showing solution with 2 pins and debug periodicity of 1, (d) Interval Graph showing solution with 1 pin and debug periodicity of 2.

where Y is a debug read variable, and DO is an output variable, are:

$$[ASAP, ALAP]\ (DO)\ =\ (min\{prod(\dot{Y})\} + 1, last\ \ cs) \qquad \textbf{(EQ 2)}$$

Let us consider the modified CDFG shown in Figure 1(b). The *(ASAP, ALAP)* times for the debug input/output operations are shown in Figure 1(c). For instance, the *(ASAP, ALAP*) for input operation of DI_1, denoted just by DI_1, are (1,0), as $S1$ is needed in control step 1. Hence, input variables DI_1 and DI_2 cannot be scheduled. Since, input In has been scheduled in control step 1, the only available control step to schedule both DI_3 and DI_4 is control step 2. However, this can be done only if an extra input pin is made available. Similarly, it can be seen that to satisfy the debug output requirements, an extra output pin is required for either DO_3 or DO_4. Consequently, when the modified CDFG in Figure 1(b) is given to the high level synthesis system Hyper, it can schedule the input/output operations DI_3, DI_4, DO_1, DO_2, DO_3, and DO_4 using an extra input pin and an extra output pin, but cannot satisfy DI1 and DI2. However, if no I/O pins are available, only 1 input operation and 3 output operations can be satisfied.

4.3 Pipelining Debug Input/Output Variables

We now show how the debug input/output variables can be functionally pipelined to satisfy the desired debug requirements. Pipelining is a widely used transformation technique which changes positioning of a selected set of variables from one iteration to another iteration of the computation. Probably conceptually most simple way to accomplish pipelining is to add simultaneously one delay (state) on all inputs or all outputs of the pipelined computations. Positioning of a variable to previous iteration is usually denoted by adding @1 to the name of the variable. Pipelining the debug I/O variables gives more freedom to schedule the debug I/O operations. Consider the CDFG with the debug I/O variables pipelined, shown in Figure 2(a). For instance, variable S1 is last written in the previous iteration in control step 3, and is required in the current iteration in control step 0. Consequently, if the write operation ($S1$ <- DI_1) can be performed after step 3 in previous iteration, it will not be re-written in the previous iteration. Similarly, the write operation needs to be completed before control step 1 in the current iteration for it to be used. Hence, the (*ASAP, ALAP*) times for scheduling the input operation for DI_1, is (4@1,0). The (*ASAP, ALAP*) times of the other pipelined debug variables/operations are listed in Figure 2(b).

In general, the (*ASAP, ALAP*) times of pipelined input variable *DI* for a debug write *X* <- *DI* are:

$$[ASAP, ALAP]\ (DO)\ =\ (max\{max\{prod(X@1)\}, max\{req(X@1)\}\} + 1, min\{req(X)\} - 1), \qquad \textbf{(EQ 3)}$$

where *X*@1 denotes the variable *X* in the previous iteration. The (*ASAP, ALAP*) times of a pipelined output operation for a debug read *DO* <- *Y* are:

$$[ASAP, ALAP]\ (DO)\ =\ (max\{prod(Y)\} + 1, min\{prod(Y@-1)\} - 1) \qquad \textbf{(EQ 4)}$$

where *Y*@-1 denotes the variable *Y* in the next iteration. The (*ASAP, ALAP*) times for the pipelined debug input/output variables in Figure 2(a) are listed in Figure 2(b). The effect of pipelining the debug I/O variables is clear from Figure 2(b). Each of the I/O variables has now much more time to be scheduled than originally as shown in Figure 1(c).

The scheduling of the debug I/O operations/variables, and their assignment to available pins, is performed using an interval graph representation of the (ASAP, ALAP) times of the I/O operations, and the left-edge algorithm [Kur87], originally proposed for register/module assignment. Figure 2(c) shows the (*ASAP, ALAP*) information for the debug I/O operations in the form of an interval graph, where each horizontal bar associated with an I/O operation/variable represents the interval of the (ASAP, ALAP) times of that operation. Note that the (*ASAP, ALAP*) intervals for the input operations are from the previous to the current iterations, while the intervals for the output operations are from the current to the next iteration.

When the left-edge algorithm is applied to the interval graph shown in Figure 2(c), all the I/O operations can be scheduled and assigned as shown in Figure 2(c). The four input operations are scheduled in the control steps 4, 5, 6, and 7 of the previous iteration respectively, and assigned to input pin $p1$. The four output operations are scheduled in the control steps 4, 5, 6 of the current iteration, and control step 1 of the next iteration, and assigned to output pin $p1$. The net effect is that at the beginning of every iteration, all the debug write variables $S1$, $S2$, $S3$, and $S4$ can been written for use in the current iteration. Also, by the second control step of every iteration, the values of the debug read variables $S1$, $S2$, $S3$, and $S4$ in the previous iteration can be read out.

4.4 Increasing Debug Periodicity

The previous sub-section showed that pipelining the debug input/output operations can be a powerful technique to satisfy the debug requirements. However, pipelining itself may not be sufficient to solve the problem of scheduling the I/O operations under the given performance (control steps) and pins constraints. For example, let us re-visit the CDFG shown in Figure 2(a), under the constraint that only one I/O pin is available. In the 1st and 7th control steps, the pin will be busy with the mandatory input operation of *In* and the mandatory output operation of *Out*. In the remaining 5 control steps, it is not possible to schedule all the eight debug input/output operations with only one available pin.

Note that till now, we have attempted to write and read the desired debug variables at every iteration of the computation. For instance, the solution shown in Figure 2(c) ensures that at every iteration, the variables $S1$, $S2$, $S3$, and $S4$ can be written for use in the current iteration, and their values from the previous iteration can be read out. However, when a solution does not exist with a single iteration write/read, we can increase the periodicity at which the debug variables can be written and read. We define debug periodicity as the number of iterations needed to write and read the desired debug variables.

We next show that the debug requirements for the seven clock cycles, single pin constraint can be satisfied if we increase the debug periodicity to two iterations. The (*ASAP, ALAP*) times of the I/O operations for each iteration remains the same; however, now the operations can be spread over two iterations, as shown by the new interval graph in Figure 2(d). A possible solution using the left-edge algorithm [Kur87] is also shown. Each period starts at the 4th control step of the i-th iteration, and ends with the 3rd control step of the (i+2)-th iteration. The solution implies that by the 2nd control step of iteration (i+1), the desired debug data from the previous period can be read out, and by the 2nd control step of the (i+2)-th iteration, the desired debug data can be written to be used in the next two-iteration period. Note that to satisfy this constraint of a single I/O pin, it is required to schedule at most one I/O operation in any control step.

The above solution shows a trade-off between the valuable resources of input/output pins and debug periodicity. A solution with periodicity of 1 requires two I/O pins, while increasing the periodicity to 2 allows a solution with only one I/O pin. The disadvantage of a solution with debug periodicity greater than one is that the desired variables can be written/read only after every few iterations, determined by the periodicity, and not at every iteration of the computation as can be desirable. However, increasing the debug periodicity is an effective way to provide a solution with lower pin requirements.

Let *dv* be the number of debug variables to be satisfied, *dp* be the debug periodicity, *iop* be the number of I/O pins available, *cc* be the number of control steps available, *I* be the number of primary inputs, and *O* be the number of primary outputs. Then, the maximum number of input/output operations that can be supported is (*cc***iop***dp*). Since in every iteration, all the primary input and primary output operations have to be supported, the number of debug variables that can be satisfied is given by:

$$dv \le cc*iop*dp - (I+O)*dp \qquad \textbf{(EQ 5)}$$

The debug periodicity required to satisfy all the *dv* debug operations is:

$$dp \ge \lceil dv / (cc*iop - (I+O)) \rceil \qquad \textbf{(EQ 6)}$$

4.5 Debug I/O buffering

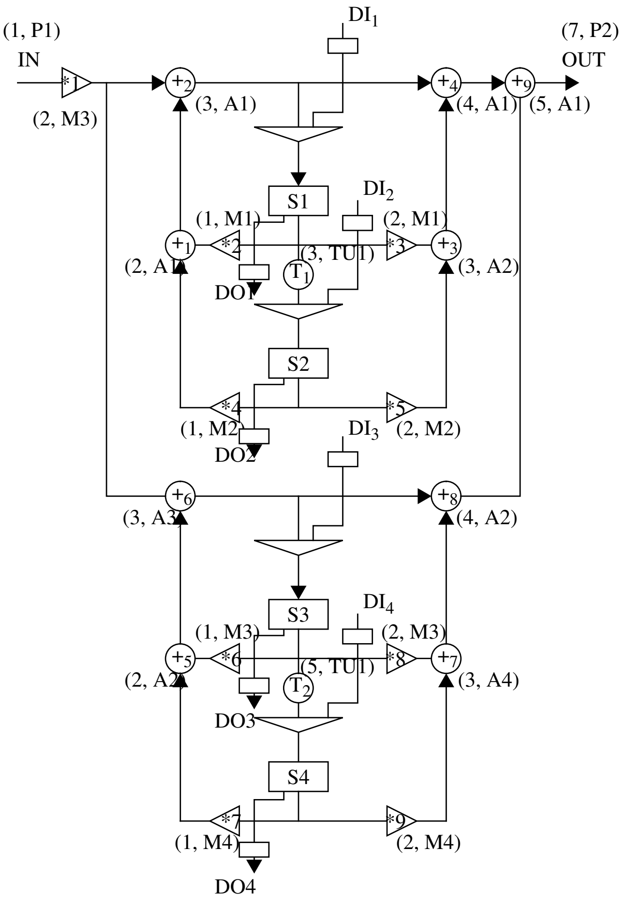
The minimum debug periodicity *dp* given by equation 6 may not be sufficient to satisfy all the debug requirements of *dv* variables. In this section, we introduce I/O buffering as a way to be always able to achieve n debug requirements using $\lceil n / (cc*iop - (I+O)) \rceil$ iterations, where *cc* and *iop* are the available control steps and input/output pins, respectively, and I and O are the number of primary inputs and primary outputs.

Let us consider the same CDFG of the 4th order IIR parallel filter as before, but with a more constrained available time of 5 control steps. Assume there is one input pin, and one output pin available. The debug write/read required are the same as before: { WR(*S*1), WR(*S*2), WR(*S*3), WR(*S*4), RD(*S*1), RD(*S*2), RD(*S*3), RD(*S*4)}. Figure 3(a) shows the given schedule and assignment of the original CDFG nodes, as well as the modification done to add the desired debug write/reads, with the debug input/output variables pipelined.

According to equation 5, with a debug periodicity of 1, the maximum number of debug variables that can be satisfied is (5*2*1 - 2*1) = 8. Since the number of debug variables that we have to satisfy is exactly 8, a periodicity of 1 may be sufficient to satisfy them. In other words, the minimum debug periodicity required to satisfy the given 8 debug reads/writes is 1. Consequently, let us try to schedule/assign the added debug input/output operations with a debug periodicity of 1. The interval table for the debug I/O operations (variables) are shown in Figure 3(b). Application of the left-edge algorithm [Kur87] on the corresponding interval graph is shown in Figure 3(c). As can be seen from Figure 3(c), only two of the four desired debug writes can be satisfied.

Analysis of the interval graph shows that even though the input pin is available in two control steps, 2 and 3 in the previous iteration, it cannot be utilized for the debug inputs because the debug variables cannot still be written, as shown by their *ASAP* times in Figure 3(b). This is because if *S*1, *S*2, *S*3, or *S*4 are written in control steps 2 or 3, they will be re-written by the functional operations, +2, *T*1, +6, or *T*2 respectively. This shows that though control steps and pins may be available, they may not be utilizable because of the *ASAP, ALAP* constraints imposed by the original CDFG on the debug variables.

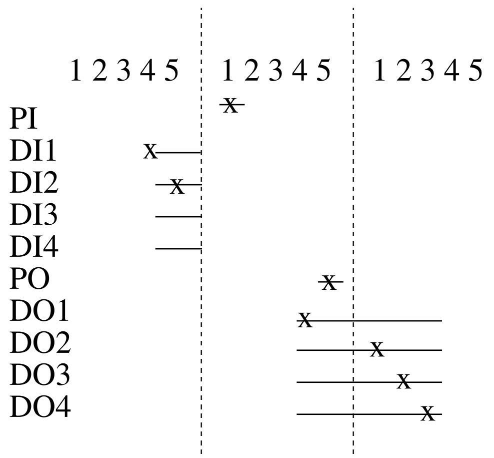
Input/Output buffering is a possible way to eliminate the restriction imposed on the input/output operations by the *ASAP, ALAP* times of the write/read variables. With input buffering, any input operation for a write *X* <- *DI* can be performed with any available input pin at any control step, and then stored in an



(a)

	ASAP	ALAP
DI1	4@1	0
DI2	4@1	0
DI3	6@1	2
DI4	6@1	2
PI	1	1
DO1	4	3@-1
DO2	4	3@-1
DO3	6	5@-1
DO4	6	5@-1
PO	7	7

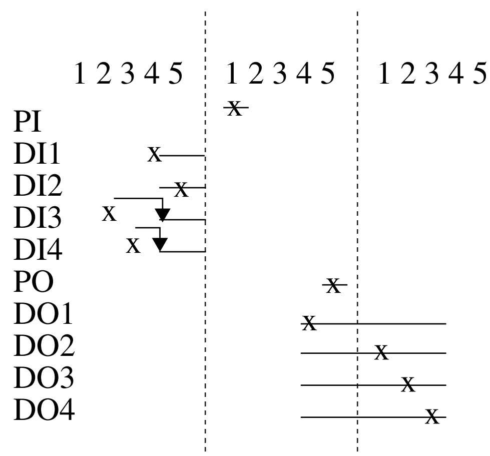
(b)



(c)

x : input/output operation scheduled

: transfer from I/O buffer



(d)

Figure 3: (a) CDFG of IIR parallel filter showing schedule and assignment with available time of 5 control steps, and added nodes for debugging, (b) Interval Table, (c) Interval graph showing solution not possible (pins =1, debug period =1), (d) Interval Graph showing solution using 2 input buffers

input buffer *IB*, to be later transferred to the register storing variable *X* anytime during [*ASAP,ALAP*](*X*). Similarly, output buffering can be used to transfer *Y* to an output buffer, *BO*, during [*ASAP,ALAP*](*Y*), and then use an output pin when it becomes available later. The I/O buffering strategy, while taking up some extra hardware resources of registers and interconnects, allows all the available pins and control steps to be utilized for the desired input/output operations. Hence, if the number of I/O buffers is not limited, a solution to the debug write/read requirements of *dv* variables can be always satisfied with a periodicity of *dp*.

$$dp = \lceil dv/(cc*iop - (I+O)) \rceil \qquad \textbf{(EQ 7)}$$

Using input/output buffering, all the debug requirements for Figure 3(a) can be satisfied as shown by the interval graph in Figure 3(d). The input operations DI_3 and DI_4 are performed in control steps 2 and 3, and stored in input buffers *IB*1 and *IB*2 respectively. Subsequently, in control step 4, the data from *IB*1 and *IB*2 are transferred to the register storing variables *S*3 and *S*4 respectively. Consequently, with 2 input buffers, and two interconnects, (*IB*1 -> Reg(*S*3)) and (*IB*2 -> Reg(*S*4)), all 8 debug read/write can be accomplished, with a periodicity of 1, under the available time and pins constraints of 5 and 2 respectively.

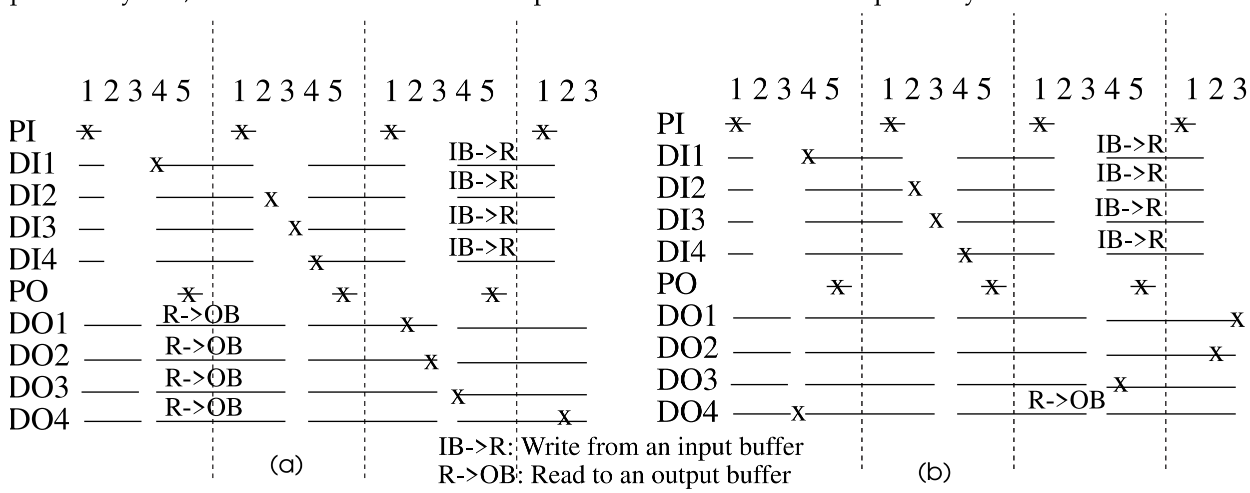


Figure 4: Solution for CDFG of Figure 3(a), with added constraint of 1 I/O pin. (a) Solution with 8 I/O buffers, (b) Solution with 5 I/O buffers

4.6 Optimizing Number of I/O Buffers

As mentioned in the previous section, the I/O buffer solution incurs some hardware penalty in terms of the number of I/O buffers and interconnects that need to be added. This arises a need to optimize the I/O buffers required to satisfy a given debug requirement.

Let us consider the same example in Figure 3(a), with a pin constraint of 1 input/output pin. Using equation 7, the debug periodicity required is $= \lceil 8/((5*1) - 2) \rceil = 3$. The desired solution should provide all the debug inputs to the debug write variables by the beginning of the 1st iteration of the 3-iteration period. Also, reading of all the read variables from the previous period should be completed in the present 3-iteration period.

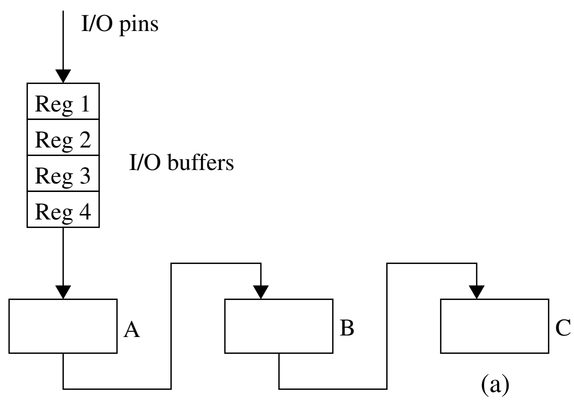
Figure 4(a) shows a possible solution using the left edge algorithm, which requires a periodicity of 3, and 8 input/output buffers. For example, though the input operation for DI_1 is performed in control step 4 which is within the (*ASAP,ALAP*) bounds of DI_1, it has to be stored in an input buffer to avoid being written over by operation +3 in the next iteration, before the beginning of the next 3-iteration period. Similarly, all the other input and output variables need to be stored in I/O buffers. On the other hand, a solution is shown in Figure 4(b) which requires only 5 input/output buffers, using the same periodicity of 3. We describe an algorithm in the next section which obtains solutions do the given debug

5.0 Minimizing Debugging Hardware Overhead using Life-time Splitting of Debugging Variables

5.1 Problem Formulation

As we already indicated, one of the most important features of the behavioral synthesis debugging process is that it usually incurs a relatively small hardware overhead. In this Section we will show a technique which can even further reduce debugging hardware overhead and in many situations even eliminates the need for additional registers and interconnect.

We will introduce the procedure for debugging hardware reduction using the following small example shown in Figure 5. The key idea is to use already available registers in the designs when they are not used for storing design variables and to use already available interconnects for transferring debugging variables among the available registers in the design when they are free.



(a)

Variable	Control steps when register is not used	lifetime of variable *dv*
Reg2	1 - 7	1 -4
A	3 - 11	4 - 6
B	5 - 9, 12 - 15	7 - 9
C	8 - 20	10

(b)

Figure 5: Minimizing Debugging Hardware Overhead using Life-time Splitting of Debugging Variables

The techniques is based on life-time splitting of debugging variables. The idea of life-time splitting for the reduction of registers requirements was first proposed by Krishnamoorthy and Nestor [Kri92]. Consequently, Potkonjak and Dey [Pot94] showed how additions of deflections operations with idempotent elements at the behavioral level can be used to not only reduce the required number of registers, but also to reduce the interconnect requirements at the RT-level by essentially doing splitting of life-times of design variables. The problem and the solution presented in this section differs from those two works, not only by development of a different algorithmic optimization technique and due to its different purpose (debugging), but also in the essence how the problem is treated. While both Krishnamoorthy and Nestor and Potkonjak and Dey assumed life-time splitting of variables as an integral part of the allocation, scheduling and assignment process, the life-time splitting of debugging variables is done after the allocation is finished and the schedule and assignment of all design operations and variables is completed.

Figure 5 illustrate the considered optimization problem. Assume that debug variable *dv* is stored in the first control step in the register 2 of the I/O buffer and that has to be written in register C between control steps 8 and 10. Figure 5(a) shows a part of the datapath of the initial implementation which was obtained without taking into account debugging. Figure 5(b) shows intervals in the registers of the design in which those designs are not used, except for the register C for which the interval during which variable *dv* should be stored in is shown.

The straightforward way to accomplish this part of debugging task is to introduce a new interconnect from the I/O buffer to the register C. However, one can avoid the introduction of a new interconnect by first transferring variable *dv* from I/O to register A, and then consequently to register B and eventually to register C. During this process two requirements must be always satisfied. First, during period the debugging variable is stored in a particular register the register should not be already allocated for either design or another debug variable. Second, each transfer from a register to another register must be accomplished in one of control steps when this interconnect is not used for transfer or any other data.

Assuming, that interconnects I/O -> reg A, reg A -> reg B, and reg B -> reg C are not allocated in control steps 4, 6, 7, and 10 respectively one can transfer variable *dv* from I/O to reg C as it is shown in the last column of Figure 5(b).

So, the problem of debugging hardware minimization using life-time spliting can be now stated in the following way. Given is a design and all debugging variables and their destinations. Reduce I/O buffer and additional interconnect requirements by appropriately scheduling and assigning data-transfers of the debugging variables, without impeding proper functionality of the design.

5.2 Problem Complexity and Algorithm for Debugging Hardware Overhead Minimization using Life-Times Spliting of Debugging variables

It is easy to see that the problem is NP-complete. It is so, because one can reduce an instance of a general scheduling problem to an instance of the new problem. Also, once the solution for the new problem is available, it is easy to verify its correctness, by checking timing constraints for each register and each interconnect.

To efficiently solved the optimization problem, we developed the heuristic algorithm described by the following pseudo-code:

Algorithm for Debugging Hardware Overhead Minimization using Life-Time Splitting of Debugging Variables (){

```
        1. {Assemble_pull_of_free_resources();
        2. Identify_feasible_debugging_variables();
        3. while (non_resolved_debugging_variable() == 1){
                4. dv = Select_debugging_variable();
                5. Assign_and_Schedule(dv);
                6. if (no_feasible_variable) {add_additional_interconnect();
                7. Update_pull_of_resources();}
                8. Update_list_of_debugging_variables();
        }
}
```

The key idea of the algorithm is to select at each stage a debugging variable which will least reduce the number of choices in which debugging variables can be transferred to their destinations, by allocating registers for the shortest amount of time, and by allocating interconnects which are in smallest demand for future possible use by other debug variables. If in a particular stage of the algorithm there is no debug variable which can be transferred using existing resources to its destination, a new interconnect is allocated for directly transferring a debugging variable with the shortest life-time. Consequently, the pull of resources is updated. The run time of the heuristics is $O(n^2m)$, where n is the number of debugging variables, and m is the number of resources in the pool.

6.0 Experimental Results

We applied our approach for design-for-debugging and optimization algorithms on a set of 16 industrial examples. Table 2 gives the size characteristics of the considered designs. The examples are: GE Controller1 and GE Controller2 - two 4 and 5 state linear controllers; Honda Controller1 and Honda Controller2 - two linear mechanical controller; Wavelet filter; - QMB sub-band filter; Low Pass Filter, BandPass Filter, High Pass Filter and Band Stop Filter - four audio filters (pass filters are cascade and parallel IIR structures, two other filters are direct form and transpose form FIR structures); 8X8 DCT - 2 dimensional Lee's algorithm version of discrete cosine transform; DAC - NEC digital-to-analog converter for audio applications; modem and adaptive modem - two low-speed low-cost communication modems; LMS audio formatter and Echo-Canceller - two NEC communication designs and Large Controller - 11 states, 3 input, 3 output linear controller for automotive applications.

Task	# of operations	# of debug variables	additional I/O pins, Hyper	additional registers, DfD method
GE Controller1	48	10	3	0
GE Controller2	108	10	3	1
Honda Controller1	97	10	4	1
Honda Controller2	67	8	1	1
Wavelet filter	31	14	5	4
Low Pass Filter	32	12	2	1
BandPass Filter	38	14	2	0
High Pass Filter	42	20	1	0
BandStop Filter	30	12	1	1
8X8 DCT	46	20	11	2
DAC	354	44	7	0
modem	227	40	3	0
adaptive modem	200	54	4	0
Large Controller	324	22	5	2
LMS audio formatter	464	88	16	0
Echo-Canceller	212	56	16	0

Table 2: Characteristics of examples used to demonstrate the effectiveness of DfD approach

During the derivation of experimental results we applied the following procedure. We selected as debugging variables all the state variables of the corresponding computations. First, all design have been synthesized using the Hyper high level synthesis system [Rab91]. For comparison purposes we considered only implementations in which Hyper was able to produce a feasible solution after addition of debugging variables. The number of additional I/O pins required by Hyper due to the debugging requirements is shown in Table 2, column 3. We see that in a number of examples, this approach resulted in high and unacceptable I/O overhead.

We next applied the proposed DfD approach to the initial designs (with no debugging variables) produced by Hyper. The DfD approach could satisfy all the debugging requirements without addition of any new I/O pins. The number of registers in I/O buffers needed is shown in the last column of Table 2. The area overhead was minimal, in all cases less than 3% of the initial area, clearly indicating low hardware overhead of the DfD approach. For all examples, a debugging period of 1 was obtained, i.e. all debugging variable could be observed and controlled in all iterations.

7.0 Future Research Directions

The introduction of debugging as a synthesis, and in particular behavioral synthesis, design consideration and goal creates a starting point for studying numerous new research and development topics from at least four different points of view: (1) implementation platform; (2) high level synthesis; (3) computational and hardware models; and (4) debugging process itself.

While the ASIC implementation style often provides advantages in terms of mass volume pricing, low power, and overall performance, programmable platforms are superior when flexibility and time-to-market are considered. Of particular interest are not just general purpose microprocessors, microcontrollers and DSP processors, but even more application specific instruction set processors (ASIP) and application specific programmable processors (ASPP). Design methodologies and technologies for both ASIPs and ASPP are currently very popular topics in high level synthesis, and providing debugging facilities for them would certainly improve their chances for wide acceptance in design practice.

Our current implementation of DfD methodologies targets static compilation and synchronous data flow model of computation. It will be interesting to see how it can be modified to satisfy needs of different models of computations, in particular those which are commonly used for decision and conditional intensive applications [Ber91] and those which address complex timing constraints [Ku92]. Also, our methodology did not impose any restrictions on underlying hardware models. In many situations it is advantageous to considered hardware models which enforce efficient implementation by imposing restriction on interconnection strategy between execution units and registers (e.g. dedicated register file model). We anticipate that for this types of hardware models it is possible to develop even more efficient support for debugging, by exploring the properties of the hardware model. Finally, we would like to mention that one of most interesting, challenging, and important related problems is to integrate other two phases of debugging, error diagnosis and error correction, as integral components of (high level) synthesis process.

8.0 Conclusion

We addressed a new and important problem of considering hardware and synthesis support for debugging during high level synthesis. The ASIC debugging process has been defined. Pipelining of debugging variables for increasing their scheduling and assignment freedoms, addition of I/O buffers for intermediate storage of debugging variables, introduction of the concept of increasing debugging periodicity, and approach for minimizing hardware overhead using life-time splitting technique are conceptual and implementation basis for efficient, yet inexpensive design for debugging. Several synthesis problems related to the DfD approach are solved using polynomial complexity algorithms. The practical effectiveness of DfD approach is demonstrated on numerous examples by providing complete observability and controllability of debug variables with a very minimal hardware overhead.

9.0 References

[Bal69] R.M. Balzer, "Exdams - Extendable debugging and monitoring systems", Proceedings of AFIPS Spring Joint Computer Conference 34, AFIPS, Washington, D.C., pp. 125-134, 1969.

[Bea83] B. Beander, " VAX DEBUG: an interactive, symbolic, multilingual debugger", SIGPLAN Notices, Vol. 18, No. 8, pp. 173-179, 1991.

[Ber91] R. Bergamaschi, R. Camposano, M Payer, "Datapath synthesis using path analysis", 28th Design Automatic Conference, pp. 591-596, 1991.

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

[Car87] T.A. Cargill, B.N. Locanthi, "Cheap hardware support for software debugging and profiling", ACM SIGPLAN Notices, Vol. 22, No. 10, pp. 82-83, 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.

[Coh94] I.B. Cohen, "The Use of "Bug" in Computing", Annals of the History of Computing, Vol. 16, No. 2, pp. 54-55, 1994.

[Coh91] R. Cohn, "Source Level Debugging of Automatically Parallelized Code", SIGPLAN Notices, Vol. 26, No. 12, pp. 132-143, 1991.

[Cop94] M. Cooperman, "Debugging Optimized Code Without Being Misled", ACM Transactions on Programming Languages and Systems, Vol. 16, No. 3, pp. 387-427, 1994.

[Cou88] D. Coutant, S. Melov, M. Ruscettia, "Doc: A Practical Approach to Source-level Debugging of optimized code", Proc. of SIGPLAN'88 Conference on Programming Language Design and Implementation, ACM , New York, NY, pp. 125-134.

[Har90] D. Harel et al., "Working Environment for the Development of Complex Reactive Systems", IEEE Trans. on Software Engineering, Vol. 16, No. 4, pp. 403-413, 1990.

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

[Hop81] G. Hopper, "The First Bug", Annals of the History of Computing, Vol. 3, pp. 285-286, 1981.

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

[Kan92] G. Kane, J. Heinrich, "MIPS RISC architecture", Prentice Hall, NJ 1992.

[Kes90] P.B. Kessler, "Fast breakpoints: Design and implementation", SIGPLAN Notices, Vol. 25, No. 6, pp. 78-84, 1993.

{Kri92] G. Krishnamoorthy, J.A. Nestor, "Data Path Allocation using an Extended Binding Model", Design Automation Conference, pp. 279-284, 1992.

[Kur87] F.J. Kurdahi, A.C. Parker, "REAL: A Program for Register Allocation", Design Automation Conference, pp. 210-215, 1987.

[Lin90] M.A. Linton, “The evolution of Dbx”, USENIX Conference, pp. 211-220, 1990.

[Mic88] Microtec International Inc., “New MICE-II Users’ Manual for 16-bit INTEL series Microprocessors, 1988.

[Mil91] B. Miller, J. Choi, “Techniques for debugging parallel programs with flowback analysis”, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp. 491-530, 1991.

[Min94] D. Minoli, R. Keinath, “Distributed Multimedia Through Broadband Communications Services”, Artech House, Boston, MA 1994.

[Miy88] M. Miyata, H. Kishigami, K. Okamuto, S. Kamiya, “The TX1 32-Bit Microprocessor: Performance Analysis, and Debugging Support”, IEEE MICRO, pp. 37- 46, April 1988.

[Nag94] J. Naganuma, T. Ogura, T. Hoshino, “High-Level Design Validation Using Algorithmic Debugging”, EDAC-94, pp. 474-480, 1994.

[Nar91] S. Narayan, F. Vahid, D.D. Gajski, “System Specification and Synthesis with the SpecCharts Language”, ICCAD91, pp. 266-269, 1991.

[Nar93] S. Narita, F. Arikawa, K. Uchiyama, I. Kawasaki, “Design Methodology for Gmicro TM/500 TRON Microprocessor”, ICCD93, pp. 253-257, 1993.

[Per93] S.E. Perl, W.E. Weihl, “Performance Assertion Checking”, 14th ACM Symposium on Operating Systems Principles, pp. 134-145, 1993.

[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.

[Ren89] S. Renner, M.T. Harandi, “Debugging Run-time Errors”, 22nd Annual Hawaii IEEE International Conference on System Science, Vol. 2, pp. 495-503, 1989.

{Sai93] A. Saini, “Design of the Intel Pentium TM Processor”, ICCAD93, pp. 258-261, 1993.

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

[Shu93] W.S. Shu, “Adapting a debugger for optimized programs”, SIGPLAN Notices, Vol. 28, No. 4, pp. 39-44, 1993.

[Sta91] R.M. Stallman, R.H. Pesch, “Using GDB: A guide to the GNU source-level debugger, GDB version 4.0, Technical Report, Free Software Foundation, Cambridge, MA, 1991.

[Sul91] M. Sullivan, M. Stonebraker, “Using write protected data structures to improve software fault tolerance in highly available database management systems”, 17th International Conference on Very Large Data Bases, pp. 171-180, 1991.

[Tex83] Texas Instruments Inc., “XDS/22 TMS32010 Emulator”, 1983.

[Tro89] H.S. Tropp, “Whence the “bug”?”, Annals of the History of Computing, Vol. 10, pp. 341-342, 1989.

[Wah92] R. Wahbe, “Efficient Data Breakpoint”, ACM SIGPLAN Notices, Vol. 27, No. 9, pp. 200-212, 1992.

[Wah93] R. Wahbe, S. Lucco, S.L. Graham, “Practical data points: design and Implementation”, ACM SIGPLAN’93 Conference on Programming Language Design and Implementation”, pp. 1-12, 1993.

[Zel83] P. Zellweger, “An Interactive high-level debugger for control-flow optimized programs”, SIGPLAN Notices, Vol. 18, No. 8, pp. 159-172, 1983.