Design-For-Debugging of Application Specific Designs

Miodrag Potkonjak[†§], Sujit Dey[†], Kazutoshi Wakabayashi[‡]
[†]C&C Research Laboratories, NEC USA, Princeton, NJ 08540
[‡]C&C Research Laboratories, NEC Corp., Tokyo, Japan

ABSTRACT

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. Two key conceptually new design ideas that enable efficient debugging are developed: pipelining of debugging variables for improving their scheduling and assignment freedom and use of I/O buffers for improving resource utilization of I/O pins.

The provably optimal bounds for the maximum cardinality of the set of controllable and observable variables for a given design specification are derived. A polynomial time complexity synthesis algorithm for achieving the bounds is developed. The minimization of hardware overhead gives rise to a 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

It is well-known that functional debugging usually dominates the cost of design development. 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 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 related to ASIC debugging; and
4. to give an impetus for creation of design-for-debugging and synthesis-for-debugging methodologies.

Debugging is 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 of 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.

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 when debugging variables should be altered by the user provided values. The key idea is to use available I/O pins for reading and writing debug variables in control steps when they are not used by the design variables.

We conclude this section by pointing out key differences between testing and debugging. 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. 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

[§]Miodrag Potkonjak is now with UCLA , CS Department

registers at the selected control steps in which user-specified debugging variables are stored. Finally, debugging an ASIC design usually requires that all controllable variables are set simultaneously and that all observable variables are simultaneously obtained.

The rest of the paper is organized as follows. In the next section we review related work. Sections 3 and 4 introduce all preliminaries and 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 in Section 6, we summarize the DfD method in Section 7.

2.0 Related Work

Debugging is as old as building of digital electronic computing systems. Debugging has been recognized as a crucial design and compilation activity. However, initially it was relatively rarely addressed due to its high conceptual complexity [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, including compilers and computer architecture,

By far the most comprehensive treatment of debugging has been conducted in the software compiler domain. Several debuggers, such as Dbx [Lin90], has been widely used. Currently the major emphasis in the software compilation debugging domain is on integration between optimization technique based on transformations or aggressive scheduling techniques and debugging process [Shu93].

Numerous controllers and DSP processors are supported by in-circuits emulators which enable an efficient debugging. An in-circuits emulator for a processor is a system that can imitate the behavior of the microprocessor, with the extension that the internal state and operation of the emulated processor is fully observable and controllable by the user [Chi94].

An interesting alternative to in-circuit emulation is used sometimes during development of new microprocessors [Sei93]. 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.

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 with algorithmic debugging techniques from logic programming to speed-up design validation process.

3.0 Preliminaries and Problem Formulation

In this section we present all the essential assumptions for introducing and developing our approach for design-for-debugging. We conclude the section by explicitly stating the considered design-for-debugging problem.

We assume the synchronous dataflow model of computations [Lee87] which is widely used in many computationally intensive applications. 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. We considered two types of I/O mechanisms. In the first type each pin can be used to both input and output data. In the second case, 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.

The four key debugging assumptions are the following.

1. The design is fully specified (scheduled and assigned) and its functionality and realization should not be disturbed by the debugging process, except for bringing the user specified values to the controllable variables.
2. All controllable/observable variables are known at compile/synthesis time. 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 DfD 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 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 are denoted by $S1$, ... , $S4$. 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 (*A*1 and *A*2) and two multipliers (*M*1 and *M*2). For instance, the operation +2 is scheduled in control step 3, and assigned to be executed in adder *A*1, shown in Figure 1(a) by the ordered pair (3, *A*1).

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. 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 *S*1, *S*2, *S*3, and *S*4 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 DfD technique to be described in this paper, the **debug write and read variables** are {*S*1, *S*2, *S*3, *S*4}, and the **debug requirements** are {WR(*S*1), WR(*S*2), WR(*S*3), WR(S4), RD(*S*1), RD(*S*2), RD(*S*3), RD(*S*4)}. 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(*S*1):

If (Debug) then *S*1 <- *Out*(+2); else *S*1 <- 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.

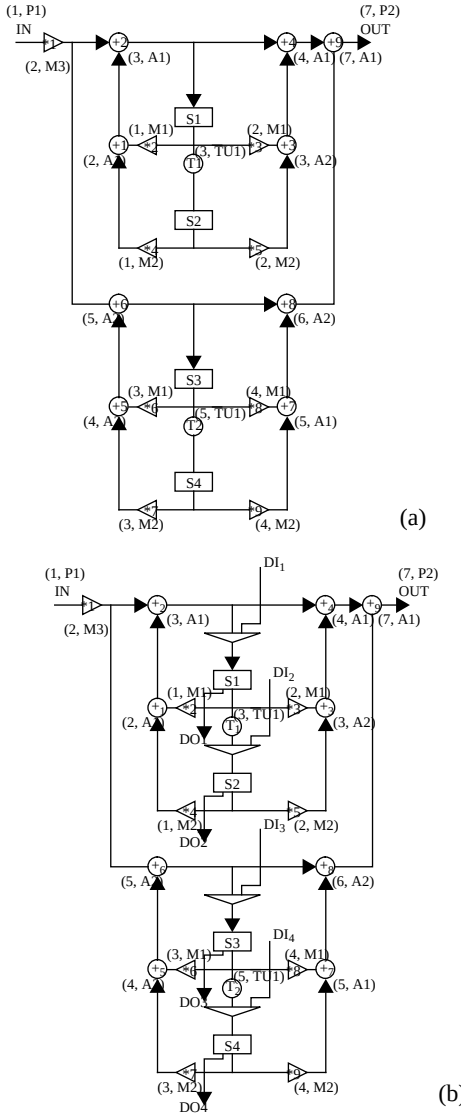

Figure 1: (a) Original CDFG of the 4th order IIR parallel filter, showing the user-specified schedule and assignment to satisfy the available time constraint of 7 control steps, (b) Modified CDFG after incorporating the debug I/O operations.

4.2 Satisfying the Debug I/O 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:

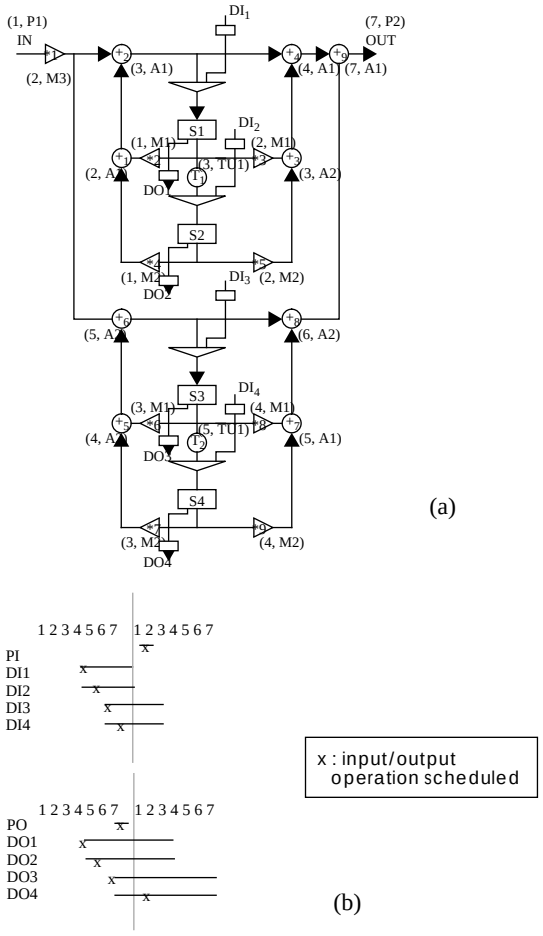

Figure 2: (a) Modified CDFG showing pipelining of debug variables, (b) Interval Graph showing solution with 2 pins (c) Interval Graph showing solution with 1 pin

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

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 *S*1 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 I/O 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. 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 (*S*1 <- 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:

$$[ASAP, ALAP]\ (DO) \ = \ (max\{max\{prod(X@1)\}, max\{req(X@1)\}\} + 1, min\{req(X)\} - 1) \quad \textbf{(EQ 2)}$$

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:

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

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(b) 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.

When the left-edge algorithm is applied to the interval graph shown in Figure 2(b), all the I/O operations can be scheduled and assigned as shown in Figure 2(b). 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 *p*1. 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 *p*1. The net effect is that at the beginning of every iteration, all the debug write variables *S*1, *S*2, *S*3, and *S*4 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 *S*1, *S*2, *S*3, and *S*4 in the previous iteration can be read out.

4.4 Debug I/O buffering

Note that till now, we have attempted to write and read the desired debug variables at every iteration of the computation.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. 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.

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. Application of the left-edge algorithm [Kur87] on the corresponding interval graph is shown in Figure 3(b). As can be seen from Figure 3(b), 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.

I/O 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 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*.

$$dp = \lceil dv/(cc*iop-(I+O))\rceil \qquad \textbf{(EQ 4)}$$

Using input/output buffering, all the debug requirements for Figure 3(a) can be satisfied as shown by the interval graph in Figure 3(c). 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.

5.0 Minimizing Debugging Hardware Overhead

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.

We will introduce the procedure for debugging hardware reduction using the following small example shown in Figure 4. 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.

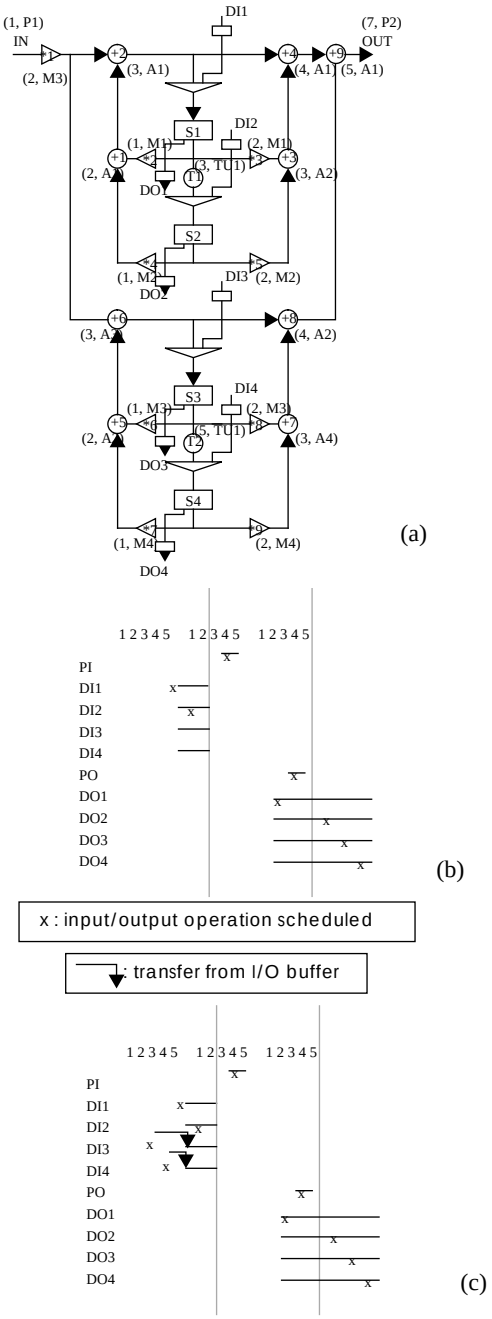

Figure 3: (a) IIR parallel filter: schedule and assignment with available time of 5 control steps, and added nodes for debugging, (b) Interval graph showing solution not possible (c) Interval Graph showing solution using 2 input buffers

The techniques is based on life-time splitting of debugging variables [Kri92, Pot94]. Figure 4 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 4(a) shows a part of the datapath of the initial implementation which was obtained without taking into account debugging. Figure 4(b) shows intervals in the registers of the design in which those registers are not used,

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

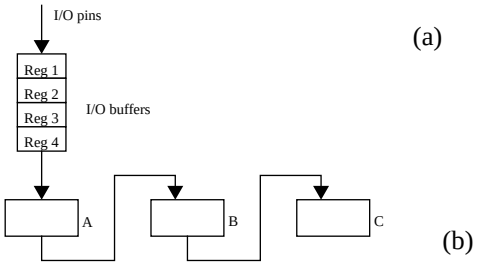

Figure 4: Minimizing Debugging Hardware Overhead using Life-time Splitting of Debugging Variables

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 4(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.

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

Algorithm for Debugging Hardware Overhead Minimization (){
1. Assemble_pull_of_free_resources();
2. Identify_feasible_debug_variables();
3. while (non_resolved_debug_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. 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 6 industrial examples. Table 1 gives the size characteristics of the considered designs. The examples are: Controller - 5 state linear controller, Wavelet - QMB sub-band filter;, High Pass - audio filter, 8X8 DCT - 2 dimensional discrete cosine transform; DAC - NEC digital-to-analog converter and Large Controller - 11 states, 3 input, 3 output linear controller for automotive applications.

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]. The number of additional I/O pins required by Hyper due to the debugging requirements is shown in Table 1, column 3. This approach often resulted in high and unacceptable I/O overhead.

Example	O/D	Hyper (pins)	DfD (registers)
Controller	108/10	3	1
Wavelet	31/14	5	4
High Pass	42 /20	1	0
8X8 DCT	46 /20	11	2
NEC	324 /22	5	2
Echo	212/ 56	16	0

Table 1: Characteristics of examples used to demonstrate the effectiveness of DfD approach

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 1. The area overhead was minimal, in all cases less than 5 % of the initial area.

7.0 Conclusion

We addressed a new and important problem of considering hardware and synthesis support for debugging during behavioral synthesis. The ASIC debugging process has been defined. Pipelining of debug variables, addition of I/O buffers for intermediate storage of debug variables, and approach for minimizing hardware overhead using life-time splitting technique are conceptual and implementation basis for efficient, yet inexpensive design for debugging. The practical effectiveness of DfD approach is demonstrated on several examples by providing observability and controllability of debug variables with a minimal hardware overhead.

8.0 References

[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.

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

{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”, DAC-87, pp. 210-215, 1987.

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

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

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

[Pot94] M. Potkonjak, S. Dey, "Optimizing Resource Utilization and Testabilit using Hot Potato Techniques", DAC-94 31th ACM/IEEE DAC, pp. 201-205, June 1994.

[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, et al. , “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.

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

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