Transactions Briefs

A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm

P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and M. P. Leong

***Abstract*—A field programmable gate array (FPGA) implementation of a coprocessor which uses the WSAT algorithm to solve Boolean satisfiability problems is presented. The input is a SAT problem description file from which a software program directly generates a problem-specific circuit design which can be downloaded to a Xilinx Virtex FPGA device and executed to find a solution. On an XCV300, problems of 50 variables and 170 clauses can be solved. Compared with previous approaches, it avoids the need for resynthesis, placement, and routing for different constraints. Our coprocessor is eminently suitable for embedded applications where energy, weight and real-time response are of concern.**

***Index Terms*—Cost, design, digital, programmable-logic-array, reconfig2000, reconfigurable-computing.**

I. Introduction

A constraint satisfaction problem (CSP) is a problem with a finite set of variables. Although there exist continuous CSPs with infinite domain, this paper is concerned solely with discrete CSPs where variables can take discrete values within a certain finite domain subject to a set of constraints which restricts them. The solution of a CSP involves finding an assignment of the variables which violates no constraints.

There has been considerable recent interest in the application of field programmable gate array (FPGAs) devices as accelerators for solving constraint satisfaction problems and, in particular, the Boolean satisfiability (SAT) problem. The Boolean SAT problem is a CSP in which the constraints are represented by a Boolean function of m binary variables $F(x_0, x_1, \ldots, x_{m-1})$ in a product of sums form. Each sum term is a clause, C_i, and is the sum of single literals, where a literal is a variable or its negation. The Boolean SAT problem is concerned with finding a variable assignment that makes $F = 1$ (satisfiable) or proving that $F = 0$ (unsatisfiable). If there are n literals in each clause, the problem is called an n-SAT problem.

Most previous research on using FPGAs as accelerators for solving SAT problems have concentrated on complete algorithms. Complete algorithms are guaranteed to find a solution if one exists, whereas incomplete algorithms may not find a solution even if one exists (e.g., [1]). In the overview below, unless specifically specified, the solvers are not restricted to n-SAT problems. Yokoo *et al.* [2] developed a machine based on FPGAs which implemented a tree search with forward checking for SAT problems. Suyama *et al.* [3] developed a machine with a dynamic variable ordering heuristic. Zhong *et al.* [4] developed a design for SAT problems utilizing the Davis–Putnam algorithm as well as an unimplemented design which used nonchronological backtracking [5]. Hamadi and Merceron [6] proposed an incomplete SAT solver based on the GSAT algorithm [7], but the design was not tested on hardware. Lee *et al.*, described an architecture for an implementation of the GENET algorithm using FPGA devices [8].

An important limitation of all of the implementations described above is that they generated a high level description of a circuit customized for the particular constraint problem. In order to execute the design, an entire iteration of the synthesis, place, and route (P&R) cycle was required for each new set of constraints. These steps are time consuming (it can take several hours to synthesize, place, and route a large design) and precludes their use in real time systems.

Recently, bitstream reconfigurable systems have been employed to address this problem, modifying the bitstream in a problem specific fashion without requiring resynthesis [9], [10]. To the best of our knowledge, all runtime configurable systems have used Xilinx XC6200 series devices [11] which document the manner in which the bitstream relates to the hardware of the device. However, XC6200 devices have been discontinued by Xilinx and also have very small logic capacity (the largest reported *bitstream reconfigurable* system only supports 13 variables and 29 clauses [10]). An implementation of a bitstream reconfigurable clause checker was previously reported by our group [12]. However, this implementation only addressed the subproblem of clause evaluation and did not implement a complete SAT solving system.

In this paper, we describe a system which implements the walksat (WSAT) algorithm [13] on a single Xilinx Virtex device [14] and enables the direct generation of a problem specific customization of the implementation. The implementation can accomodate 3-SAT problems up to 50 variables and 170 clauses, obviating the need for synthesis, placement and routing of a new circuit for problems of this size and smaller. This results in a three orders of magnitude savings in compilation time. Finding a solution quickly is of great importance in real-time applications and Selman *et al.* [13] showed that incomplete local search algorithms such as GSAT and WSAT outperform the best known complete algorithms on certain classes of large SAT problems. We are not aware of any previous hardware implementations of the WSAT algorithm.

It is envisaged that there are many applications of this work in embedded real-time constraint applications where size, power, and speed are critical, the constraints are changing in real-time, and the limitations of using incomplete algorithms are acceptable. As an example, autonomous robots could use CSP solvers for navigational planning, scene recognition, and scheduling.

II. WSAT Machine Architecture

The WSAT algorithm [13] is a simple, local search-based method for solving Boolean SAT problems. It was shown to have much better performance than the random noise, GSAT, and simulated annealing algorithms for a class of random, planning, circuit synthesis, and circuit diagnosis problems [13]. For a Boolean constraint equation F, the WSAT algorithm can be described by the following pseudocode:

Manuscript received February 18, 2000. This work was supported by a direct grant from the Chinese University of Hong Kong.

The authors are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T. Hong Kong (e-mail: {phwl, cwsham; wcwong2; hywong2; wsyuen; mpleong}@cse.cuhk.edu.hk).

Publisher Item Identifier S 1063-8210(01)00801-0.

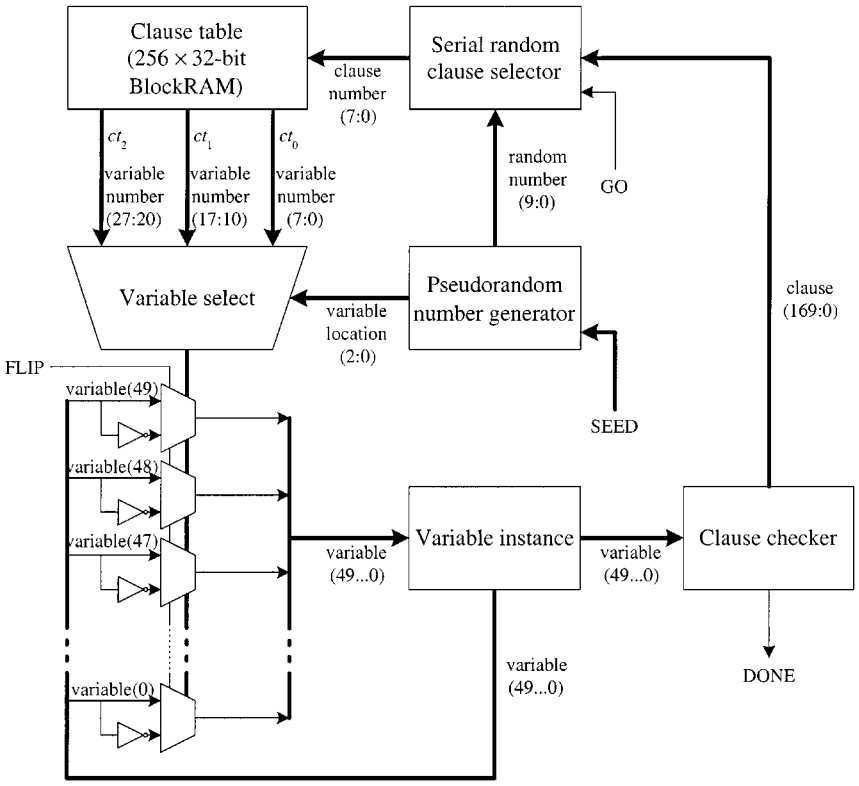

Fig. 1. Datapath of the WSAT core.

```
WSAT(int Maxtries, int Maxflips)
{
    for (tries = 1 to Maxtries) {
        V = a random instantiation
            of the variables;
        for (flips = 1 to Maxflips) {
            C = a random unsatisfied clause;
            p = a random variable in C;
            V = V with p flipped;
            if (F(V) is true)
                return V;
        }
    }
    return(no solution);
}
```

The implementation described in the rest of this paper is for problems of size up to 50 variables and 170 clauses and smaller. A block diagram of the datapath of the WSAT core is shown in Fig. 1. Note that the WSAT core implements only the inner loop of the WSAT algorithm described in the previous section. The host will call the WSAT core "Maxtries" times with random variable assignments to implement the entire WSAT algorithm described earlier.

A random variable assignment is first downloaded from the host machine to *variable(49:0)* in the WSAT core. The clause checker then, in parallel, outputs a vector of all of the clause values, *clause(169:0)*. If all clause values are TRUE, then the variable assignment represents a solution to the SAT problem and the DONE signal is asserted. Otherwise, the random clause selector takes the clause values serially and selects an unsatisfied clause (i.e., one that is FALSE) which is output as *clause_number*. Since the WSAT core supports 170 clauses, *clause_number* is an eight-bit number. *Clause_number* is used to address the clause table memory which is used to determine which variables are used in a clause. Currently the design is for 3-SAT problems so every clause has three variables (ct_0, ct_1, and ct_2 in Fig. 1). The random variable selector simply generates a random number which is used to select a random variable in the chosen clause. This is the variable number which is flipped via multiplexers to create a new variable assignment.

In the following paragraphs, the major blocks of the WSAT machine are described in detail.

Fig. 2 shows a block diagram of the clause checker. It contains an array of logic cells (LC) (see Section III), the four input lookup table (LUT) logic primitives of Xilinx Virtex devices [14]. The LCs are configured as 16 × 1-bit ROM memories. The inputs to the clause checker are 50 bits corresponding to the variables and the outputs are the 170 clause evaluations. Note that although bits 50 and 51 are shown in the figure, they are always tied to zero in the current design.

Each LC in a row has its address lines connected to four consecutive inputs of the variable to be evaluated. The output of the LC is the evaluation of the sum terms for the input variables to which it is connected. The ROM outputs are connected to OR gates which are implemented outside of the array. As an example, for the clause $C_0 = \overline{x_0} + x_2 + x_5$, the first column LC of Fig. 2 implements $\overline{x_0} + x_2$ (as a lookup table) and the second column LC implements x_5. All the outputs along a row are OR'ed together to form the desired equation for the clause.

The clause checker is the only part of the WSAT core which needs to be reconfigurable at runtime. The equations for the clauses are stored in LC 16 × 1-bit ROM primitives so all that is required is to have the ability to alter these values. The circuit is designed in the normal fashion and the ROMs can be placed at arbitrary locations. After synthesis, technology mapping, placing and routing, a circuit description file (for the Xilinx tools this has an extension .ncd) is generated. Using tools provided by Xilinx, the contents of the circuit can be converted into a human readable format, and information regarding the physical

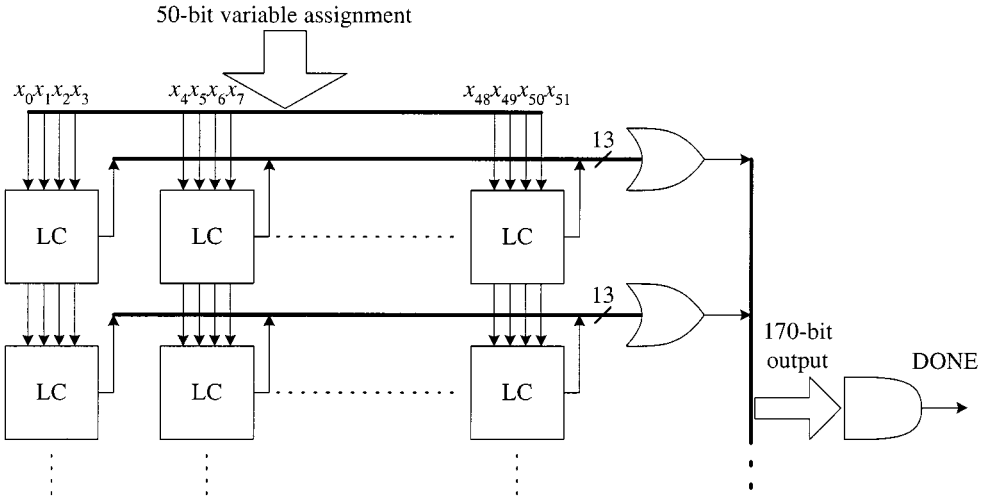

Fig. 2. Block diagram of the clause checker. Note that bits 50 and 51 are unused.

location of the LCs can be extracted. A software program was written which takes as input the bitstream, .ncd file and specification of the SAT problem in the standard DIMACS benchmark format [15]. It modifies the bitstream according to the SAT problem specification by customizing the ROM values and recomputing the CRC of the bitstream. The resulting bitstream can be downloaded to a Virtex FPGA.

The serial random clause selector is used to randomly select an unsatisfied clause in an online fashion. In iteration i, the kth unsatisfied clause, is chosen a with probability $1/k$ ("choose = i" in the pseudocode below). Moreover, the previous unsatisfied clauses have a probability of $1/(k-1) \times (1-1/k) = 1/k$ of being chosen. Thus, the algorithm will choose an unsatisfied clause with equal probability. The selection algorithm is described in pseudocode form that follows:

```
select_clause(int clause[NUMBER_CLAUSES])
{
  k = 1;
  for (i = 1 to NUMBER_CLAUSES) {
    if (clause[i] == '0') {
      r = rand(0, k);
      if (r <= 1)
        choose = i;
      k = k + 1;
    }
  }
  return choose;
}
```

The clause table memory is implemented as a 256 × 32-bit memory using the BlockRAM feature of the Virtex FPGAs (see Section III). Each memory address of the RAM corresponds to a different clause number. The address stores the number of each variable of the clause. For example, for the clause $C_0 = \overline{x_0} + x_2 + x_5$, memory address 0 would have $ct_0 = 0$, $ct_1 = 2$, and $ct_3 = 5$ (refer to Fig. 1). Note that the clause evaluator is not used efficiently since an entire row of LCs is required to implement a single clause. Methods which modify the routing in the bitstream instead of the logic equations could lead to much more compact implementations.

III. Implementation

The WSAT processor was implemented on an Annapolis Micro Systems Wildcard™ board [16], a type II PCMCIA card with 32-bit CardBus interface consisting of a Processing Element (PE, Xilinx Virtex XCV300-4 [14]) and two 64K × 32-bit SDRAMs. The XCV300 has 64Kbits of block RAM (arranged as 8 × 8-Kbit blocks) and 1536 configurable logic blocks (CLB's).

The basic building block of the Virtex FPGA is the the logic cell (LC). An LC includes a four-input function generator, carry logic, and a storage element. Each Virtex CLB contains four LCs, organized in two slices. The four-input function generators are implemented as four-input look-up tables (LUTs). Each of them can provide the functions of one four-input LUT or a 16 × 1-bit synchronous RAM (called "distributed RAM"). Furthermore, two LUTs in a slice can be combined to create a 16 × 2-bit or 32 × 1-bit synchronous RAM, or a 16 × 1-bit dual-port synchronous RAM.

Ultimately, the compactness of the circuit implementation determines the size of the SAT problem that can be solved using the system. The Xilinx Virtex XCV300-4 device used in our implementation has a relatively large number of logic resources which enables a single chip implementation of the WSAT core.

Based on an analysis of the resources required by the clause checker, clause instance, variable instance, flipping logic, random number generator, control logic, registers and multiplexers, for a 3-SAT problem with v variables and n clauses, the number of slices used for a n-clause v-variable problem can be approximated by

$$\text{slices} \approx \frac{1}{2}\left(2n\left\lceil\frac{v}{4}\right\rceil + 11\log_2 n + v + 2\log_2 v + 12\right) + 200.$$

For the 50-variable 170-clause case, the number of slices used was 2382, whereas the number predicted by the formula is 2487. Using a Xilinx Virtex XCV1000 which has 12 288 slices [14], one could solve 100-variable 340-clause SAT problems.

IV. Results

On a Pentium II 333MHz machine, the time required to generate a new bitstream file was 0.45 s and to download a bitstream to the FPGA board was 1.46 s. Using the same Pentium machine, the synthesis and place and route time was 6600 s. Thus the bitstream reconfigurable version enjoys a three orders of magnitude improvement over the standard resynthesis approach. The current program reads an existing bitstream file, computes the new values for the clause ROMs, updates the bitstream, computes a new CRC and writes the result to another file. This file is then downloaded to the board using a download program.

TABLE I
TABLE COMPARING THE EXECUTION TIME FOR SOFTWARE AND HARDWARE IMPLEMENTATIONS OF WSAT. "TIME" IS THE RAW EXECUTION TIME NOT INCLUDING RECONFIGURATION AND DOWNLOADING TIMES (WHICH ARE CONSTANT AND GIVEN AT THE BOTTOM OF THE TABLE), "AVERAGE TRIES" REFERS TO THE AVERAGE NUMBER OF TRIES TO FIND A SOLUTION, "SUCCESS" IS THE PERCENTAGE OF TIMES WSAT WILL FIND A SOLUTION TO THE SAT PROBLEM AND "SPEEDUP IS THE TOTAL SPEEDUP CONSIDERING THE RECONFIGURATION AND DOWNLOAD TIMES

Problem	Hardware			Software			Speedup
	Time (s)	Average tries	Success %	Time (s)	Average tries	Success %	
aim-50-2_0-yes1-1	3.93	14	7	16.9	11	9	2.9
aim-50-2_0-yes1-2	3.44	13.4	8	13.4	9	11	2.5
aim-50-3_4-yes1-1	2.29	13.9	12	13.9	8	13	3.3
aim-50-3_4-yes1-2	0.98	9.5	28	9.5	5	19	3.3
aim-50-3_4-yes1-3	2.12	8.5	13	8.5	5	21	2.1
aim-50-3_4-yes1-4	3.93	10.0	7	10.0	6	18	1.7
uf20-9	0.012	1	100	0.22	1	100	0.1
uf20-31	0.009	1	100	0.23	1	100	0.1
uf20-37	0.009	1	100	0.36	1	100	0.2
(no solution)	27.5	100	0	162	100	0	5.5

For all problems, time to create a new bitstream file: 0.45 s; download a bitstream to FPGA: 1.46 s; and handshake overhead is 0.67 ms per handshake.

It would be possible to avoid writing the intermediate files, avoid computing a CRC (they are optional) and directly download the resulting bitstream to the FPGA. This would significantly reduce the time required for reconfiguration.

The design was tested[1] on the "aim" benchmark problem from the Second DIMACS Implementation challenge on NP Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability [15] and on the "uf20" uniform random 3-SAT benchmark problems [17]. The problems tested (see Table I) were "aim-50-2-*" which have 50 variables, 100 clauses; "aim-50-3-*" which have 50 variables, 170 clauses and "uf20-91" with 20 variables and 91 clauses.

In order to provide a comparison with software performance, an implementation of WSAT written by Selman and Kautz [13] was used. This software implementation is highly optimized and when a variable is flipped, only the clauses affected are recomputed. For all problems, the software execution time was measured on a Sun SparcStation 20, and the hardware WSAT implementation was clocked at 33 MHz. Additionally, the software and hardware both used the parameters *Maxtries = 100* and *Maxflips = 100 000* (see Section II).

The average flips per second (fps) achieved by the software implementation were found to be independent of the problem and measured at 50 000 fps. The hardware achieved 363 700 fps. This measure can be thought of as the maximum performance achievable by our design.

Table I details the average software and hardware execution times (not including the constant configuration and downloading times) computed over 100 trials. On average, a handshake takes 0.67 ms and the overhead for a particular case can be computed by multiplying this value by the "Average Tries" entry of Table I and does not contribute significantly to the overall runtime of the system. The "Success rate" of the hardware and software implementations are quite similar, indicating that the statistics of our clause selection algorithm is similar to that of the software implementation of the WSAT algorithm. As can be seen from the "Speedup" column, the hardware is significantly slower than the software for the "uf20" problems. This is due to the constant 1.91 s overhead incurred through reconfiguration and bitstream download. This overhead becomes less significant as the run time increases and the hardware performance for aim problems is approximately two to three times faster than the software.

[1]All problems are available from SATLIB, http://aida.intellektik.informatik.tu-darmstadt.de/SATLIB.

V. CONCLUSION

An implementation of a fast SAT solving machine which uses the WSAT algorithm was presented. In contrast to previous approaches which require resynthesis when the SAT problem is changed, a problem specific architecture and runtime configuration was used to achieve a three orders of magnitude speedup in the reconfiguration time of the clause checker.

The measured performance of the WSAT core was approximately equivalent to that of a typical workstation. However, this is achieved with a single chip plus a host which does not need to perform computationally demanding tasks. Such a system has advantages in terms of cost, memory, energy, size, weight, and real-time performance compared with software implementations. The WSAT core can be considered an SAT solving coprocessor chip for a microprocessor host and would be eminently suitable for embedded application where energy, weight and real-time response is important.

ACKNOWLEDGMENT

The authors would like to thank Profs. H. M. Lee, K. H. Lee, and T. K. Lee for useful discussions on this work and C. K. Chung for his help with this project.

REFERENCES

[1] E. Tsang, *Foundations of Constraint Satisfaction*. New York: Academic, 1993.
[2] M. Yokoo, T. Sayama, and H. Sawada, "Solving satisfiability problems using field programmable gate arrays: First results," in *Proc. 2nd Int. Conf. Principles Practice Constraint Programming*, 1996, pp. 497–509.
[3] T. Sayama, M. Yokoo, and H. Sawada, "Solving satisfiability problems using logic synthesis and reconfigurable hardware," in *Proc. 31st Hawaii Int. Conf. System Sciences*, 1998, pp. 179–186.
[4] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean satisfiability with configurable hardware," in *IEEE Symp. Field-Programmable Custom Computing Machines*, 1998, pp. 186–195.
[5] P. Zhong, P. Ashar, S. Malik, and M. Martonosi, "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability," in *Proc. Design Automation Conf.*, 1998, pp. 194–199.
[6] Y. Hamadi and D. Merceron, "Reconfigurable architectures: A new vision for optimization problems," in *Principles Practice Constraint Programming CP97*, 1997, pp. 209–215.
[7] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," in *Proc. 10th Nat. Conf. Artificial Intell. (AAAI-92)*, San Jose, CA, 1992, pp. 440–446.

[8] T. K. Lee, P. H. W. Leong, K. H. Lee, K. T. Chan, S. K. Hui, H. K. Yeung, M. F. Lo, and J. H. M. Lee, "An FPGA implementation of GENET for solving graph coloring problems," in *IEEE Symp. Field-Programmable Custom Computing Machines*, 1998, pp. 284–285.
[9] H. Y. Wong, W. S. Yuen, K. H. Lee, and P. H. W. Leong, "A runtime reconfigurable implementation of the GSAT algorithm," in *Proc. Field Programmable Logic Applications Workshop (FPL'99)*, Scotland, 1999, pp. 526–531.
[10] M. Abramovici, J. T. Sousa, and D. Saab, "A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware," in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 684–690.
[11] Xilinx Inc., *XC6200 Programmable Gate Arrays (data sheet)*, 1997.
[12] P. H. W. Leong and C. K. Chung, "FPGA based runtime configurable clause evaluator for SAT problems," *Electron. Lett.*, vol. 35, no. 19, pp. 1618–1619, Aug. 1999.
[13] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proc. 12th Nat. Conf. Artificial Intell. (AAAI-94)*, Seattle, WA, 1994.
[14] Xilinx, *Virtex 2.5V Field Programmable Gate Arrays*, 1999.
[15] Dimacs challenge benchmarks. [Online]. Available: ftp://dimacs.rutgers.edu/pub/challenge
[16] Annapolis Micro Systems, Inc., *Wildcard Reference Manual*, 1999.
[17] P. Cheeseman, B. Kanefsky, and W. M. Taylor, "Where the really hard problems are," *Proc. IJCAI*, pp. 331–337, 1991.

Unifying Simulation and Execution in a Design Environment for FPGA Systems

Brad L. Hutchings and Brent E. Nelson

***Abstract*—Field programmable gate array (FPGA)-based systems provide advantages over conventional hardware including: 1) availability of the hardware during design and debug; 2) programmability; and 3) visibility. These three advantages can greatly shorten the design and verification cycle. This paper discusses a design environment that exploits these three FPGA-specific advantages to create a unified simulation/execution debug environment implemented in the JHDL design system. The described system provides a *hardware* debugging environment with the functionality of a simulator but up to 10 000× faster. In addition, testbenches and other typical verification software used in simulators can be used to verify running hardware.**

***Index Terms*—Debug, field programmable gate array (FPGA), reconfigurable computing.**

I. Introduction

Field programmable gate array (FPGA)-based systems enjoy three unique characteristics that can enhance the development and debugging process relative to conventional custom-hardware-based systems: 1) hardware availability (FPGA hardware is generic and can be used at the earliest point in the design cycle); 2) programmability (FPGA hardware can be modified throughout the design cycle); and 3) visibility (the internal state of many FPGAs is directly accessible). Early availability of the hardware means that designs can be executed on high-speed hardware (as opposed to simulation[1]) and tested in-system much earlier in the design cycle. Programmability not only allows designers to fix bugs in their design, but more importantly allows designers to incrementally develop designs in actual hardware, verifying each circuit block in-system with other circuit blocks. Finally, the ability to view internal FPGA state makes it easier to verify designs and track down and fix bugs. In short, these features suggest that a software-like debugging process containing many repeated compile/execute/modify cycles should be feasible with FPGA-based hardware systems.

In spite of these advantages, current FPGA-based systems remain difficult to use and debug. The debug process is relatively primitive and consists of two disjoint phases. In the first phase, the design is verified using logic simulation. Test vectors, memory initialization files, and command scripts are prepared and a system-level simulation model is written. In the second phase, bitstreams for each FPGA are generated and downloaded to the platform. Moving from the simulation to hardware execution environment, at worst, requires a total rewrite of test vectors and other files. Most platforms also require the creation of a runtime control program (usually written in C) which configures and controls the FPGA board. Additionally, an important tool for hardware debug in this phase is a logic analyzer; a typical approach to find bugs is to make repeated circuit synthesis runs to bring signals of interest to package pins where they can be accessed by the logic analyzer.

Most of the early research on CCM platforms reported rudimentary "built-in" debugging capability. Splash2 [1] and DecPerle-1 [2] could use readback (readback is the ability of Xilinx FPGAs to dump the internal state of memory elements as a user-accessible bitstream) to access the internal FPGA state and could match this state with some of the symbolic signal names found in the original design specification. Teramac [3] and DecPerle-1 both had rudimentary breakpoint capabilities that allowed the designer to define a single hardware event that could be used to stop the global system clock. Finally, the work described in [4] allowed a Verilog simulator to request a hardware readback and could then display the retrieved flip flop values in a table. Triggers and breakpoints were also supported to provide feedback to the simulator as to when doing a readback was warranted.

However, little has been done to further exploit the unique features of FPGAs, especially for 1) incorporating readback data into a simulator to deduce all signal values and 2) providing the same debug capabilities in *simulation* and *hardware execution*. This paper discusses an approach to doing this. That is, whether simulating a circuit or executing it in hardware, the same complete circuit state is presented to the user, and the same GUI, testbenches, commands, and control and data input files can be used in both places, simplifying the move from simulation to hardware execution and back. Further, the resulting CAD tool makes it possible to implement many of the features found in standard software debuggers including signal probing and tracing, single-stepping, breakpointing, and checkpointing. This provides support for hardware experimentation much earlier in the design process than current tools, reducing the user's reliance on much slower simulation.

In the sections which follow we first describe the structure of the CAD system, focusing on the APIs and mechanisms required to support both simulation and execution. We conclude with an example of the tool's use and a description of ongoing and future research in this area.

Manuscript received February 23, 2000. This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under Contact DABT63-96-C-0047.

The authors are with the Department of Electrical and Computer Engineering, Brigham Young University, Provo UT 84604 USA (e-mail: {hutch; nelson}@ee.byu.edu).

Publisher Item Identifier S 1063-8210(01)00697-7.

[1]Although much slower, software simulation is still useful when the FPGA hardware is not available or while waiting for bitstreams to be generated.