RECONFIGURABLE HARDWARE SAT SOLVING

WANG ZHANQING

(*B.Eng. Beijing University of Aeronautics and Astronautics)*

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2003

To My Parents

Acknowledgements

First and foremost, I would like to thank my supervisors, Assoc. Prof. Roland Yap and Dr. Martin Henz, for their inspiration and continuous support of my Master of Science research, a great combination who are always willing to listen, encourage, and give insightful comments and valuable criticism. They read all the drafts of my thesis and taught me to be thorough in analyzing problems and rigorous in presenting ideas. This thesis would not have been possible without their support and guidance. I also thank my previous supervisor Prof. Joxan Jaffar who got me started in research.

My gratitude is also conveyed to all my previous and current colleagues in Programming Languages and Systems lab of NUS, for their cooperation and support during the time I studies here.

I am deeply grateful to my parents for their everlasting patience and love. I wish to thank my younger sister, my brother-in-law, my nephew, for just being there and providing me love and support. I also thank my husband for his encouragement and support. I wish to express my deepest appreciation to my lovely daughter for the happiness her smiling face and sweet words bring me.

Finally, to my new friend who kept me company and gave me support which have been my source of strength and the reason why I have come this far, I all of thank you!

Contents

List of Figures

List of Tables

Summary

Boolean satisfiability (SAT) problems are NP-complete problems that are well-known in areas of operations research, artificial intelligence and computer-aided design. Algorithms for solving NP-complete problems may have long running times. To improve the performance of SAT solvers, hardware processing elements are used to accelerate execution. There has been considerable recent interest in the application of Field Programmable Gate Arrays (FPGAs) devices as accelerators for solving SAT problems.

There are two main types of SAT solvers, complete solvers, e.g. Davis-Putnam (DP), and incomplete Stochastic Local Search (SLS) methdos. The DP procedure is a complete branch and bound algorithm that is able to prove both satisfiability and unsatisfiability; whereas the SLS procedure is an incomplete algorithm and may not find a solution even if one exists. SLS algorithms have been successful for solving SAT problems. The WalkSAT family of algorithms contains some of the best performing SLS algorithms and has a very simple structure, thus can be improved by extracting more parallelism. There are a number of such hardware designs and implementations using reconfigurable FPGAs in the existing literature.

The use of hardware SAT solver only makes sense if there is significant performance advantage compared to software. Software can make use of state of the art processors built with the latest processor technology. A hardware SAT solver, on the other hand, is less likely to have the same level of process technology, and hence longer cycle times. Earlier hardware implementations did not outperform optimized software. One new instance-specific approach was to maximize performance by making full use of parallelism and enabled a performance of one flip per clock cycle, more than two orders of magnitude faster than software. However, an important limitation of all these previous work is that they generated a high level description of a circuit customized for a particular SAT problem. Since the time needed to re-synthesis, map, place and route the new design is likely to significantly exceed the runtime improvement from faster software SAT solver, the approach of custom design specific to a particular SAT problem instance is not practical.

This thesis explores FPGA-based hardware designs for WalkSAT, which are not instance-specific and thus not require re-synthesis. In addition to this requirement, a hardware implementation faces interesting design tradeoffs due to the inherently limited logic resources on the chip. We propose two versions of WalkSAT, which allow real-time reconfiguration. The differences of the two WalkSAT versions lead to different design choices for maximal performance. The first design emphasizes fast cycle times (one flip per clock cycle), employing random variable selection to allow for a pipelined design. The second uses a greedy variable selection heuristic, which precludes pipelining, exemplifying a tradeoff between flip rate and effectiveness of variable selection. Both design have improved performance over other published non-re-synthesis SLS FPGA implementations.

Chapter 1

Introduction

Recent improvements of Field Programmable Gate Array (FPGA) technology have made FPGA’s a viable platform for development of hardware accelerators, while still allowing design flexibility and promise of design migration to future technologies. Many members of the computing community are eyeing FPGA-based platforms as a way to provide rapidly deployable, flexible, and portable hardware solutions.

Using FPGA components in the content of propositional satisfiability problem (SAT) solving introduces challenges in system architecture and logic design. Stochastic local search (SLS) algorithms have been a successful approach for solving SAT problems. The WalkSAT family of algorithms [SKC94, MSK97] contains some of the best performing SLS algorithms. SLS algorithms like WalkSAT have a very simple structure and are composed of essentially three steps which are iterated until a satisfiable solution is found: (i) evaluate clauses; (ii) choose a variable; and (iii) flip the variable’s Boolean value.

Since each of the steps is simple, moreover SAT clauses can be directly represented in hardware, it is tempting to build a hardware SLS solver. There are a number of such hardware designs and implementations [HM97, YSLL99, LSW01, HTY01] using reconfigurable FPGA hardware. Hardware approaches to systematic search procedures for SAT problems are beyond the scope of this thesis; see [AS00] for an overview.

The use of hardware SAT solvers only makes sense if there is significant performance advantage compared to software. Software can make use of state of the art processors built with the latest processor technology. A hardware SAT solver, on the other hand, is less likely to have the same level of process technology, and hence longer cycle times. Earlier hardware implementations like [HM97, YSLL99] did not outperform optimized software. For example, a reimplementation of the design in [HM97] which was done in [HTY01] had flip rates between 98 – 962 Kflips/s. In some problems, this was a bit faster than software and in other cases slower. In [HTY01], it was shown that GSAT SLS solvers running at one flip per clock cycle was achievable with performance gains of about two orders of magnitude over software. That implementation makes use of the reconfigurable nature of FPGAs to build a custom design specific to a particular SAT problem instance. While [HTY01] shows that very large speedups are feasible, this approach is not practical as a general SAT problem solver, because the time to re-synthesize, place and route the new design for an FPGA is likely to significantly exceed the runtime improvement from the faster solver.

In the brief survey above of relevant work, we have observed that while some of these efforts have focused on the design of instance-specific solving system, there has been less work in the area of implementing a practical design in a real time environment. Typically an instance-specific hardware accelerator is not practical, because the re-synthesis requirements are often time consuming, it is necessary to find a solution.

To help address this challenge we have created the design without re-synthesis. In this thesis, we explores hardware designs for WalkSAT, which are not instance-specific and thus do not require re-synthesis. In addition to this requirement, a hardware implementation faces interesting design tradeoffs due to the inherently limited logic resources on the chip. We propose two versions of WalkSAT, which allow real-time reconfiguration. The differences of the WalkSAT versions lead to different design choices for maximal performance. The first design emphasizes fast cycle times (one flip per clock cycle), employing random variable selection to allow for a pipelined design. The second uses a greedy variable selection heuristic, which precludes pipelining, exemplifying a tradeoff between flip rate and effectiveness of variable selection. Both designs have improved performance over published SLS FPGA implementations without re-synthesis.

The remainder of this thesis is structured as follows: Chapter 2 introduces the background related to stochastic local search. Chapter 3 gives an overview on FPGA technology and its usage in reconfigurable computing and design prototyping. Chapter 4 discusses some of the current reconfigurable implementations of SAT solvers. From their design limitations, we presented a clause evaluator without re-synthesis in Chapter 5. Chapter 6 addresses our implementation platform. Chapter 7 describes our two WalkSAT implementations based on two strategies. Chapter 8 reports the experimental results. Finally, Chapter 9 concludes and offers suggestions for future work.

Chapter 2

Stochastic Local Search

Local search algorithms are among the standard methods for solving propositional satisfiability problems from various areas of computer science. After its introduction by Selman, Levesque, and Mitchell [SLM92] and Gu [Gu92], a large number of such algorithms were proposed and investigated. In this thesis, we focus on WalkSAT family of stochastic local search. WalkSAT algorithms are in general sound. In this thesis we will discuss variants of WalkSAT family.

2.1 Propositional Satisfiability (SAT)

In 1971, propositional satisfiability (SAT) was introduced as the first computational task to be NP-complete [Coo71]. As SAT is the conceptually simplest NP-complete problem, a wide range of other problems can be encoded into SAT; which make SAT a useful problem.

SAT problems can be presented as a set of propositional clauses in conjunctive normal form (*cnf*). In this form, the problem is basically a conjunction of clauses, wherein each clause is a disjunction of literals. A literal is then a propositional variable or its negation. An example of a *cnf* problem is shown in Table 2.1. A solution to a

SAT problem is a variable assignment that satisfies all the clauses according to a rule of interpretation. For the example *cnf* problem below, one possible solution has an assignment of $v1 = 1$, $v2 = 0$, $v3 = 1$, $v4 = 0$. The *cnf* is a popular standard format for encoding SAT problems.

variables	$v1, v2, v3, v4$		
literals	$v1, \neg v1, v2, \neg v2, v3, \neg v3, v4, \neg v4$		
cnf	$clause1 \wedge clause2 \wedge clause3 \wedge \ldots \wedge clause8$		
clause1	$v1 \vee v2 \vee v3$	*clause5*	$v1 \vee v3 \vee v4$
clause2	$v1 \vee v2 \vee v4$	*clause6*	$\neg v2 \vee v3 \vee \neg v4$
clause3	$\neg v1 \vee \neg v2 \vee \neg v3$	*clause7*	$v1 \vee \neg v3 \vee \neg v4$
clause4	$\neg v1 \vee \neg v2 \vee v3$	*clause8*	$v2 \vee v3 \vee v4$

Table 2.1: Example of a SAT Problem in *cnf*

2.2 Stochastic Local Search (SLS)

Stochastic local search is best viewed as a model-finding procedure wherein finding a solution to a problem determines its satisfiability. This is different from other theorem-proving procedures that look for a sound and formal proof of the satisfiability. In order to understand this model-finding procedure, we define *variable space* to be the set of all the possible combinations on truth value assignments for each variable in a given SAT problem. A procedure like Davis-Putnam [DP60] or ASAT [DABC93] performs deterministic search over the whole problem. These are called as complete procedures which can determine either the satisfiability or unsatisfiability of the SAT problem. SLS algorithms on the other hand are incomplete procedures with the advantage of having a more efficient search traversal that could solve the problem with less time. An incomplete procedure might be capable of prove satisfiability by finding a solution but will never establish unsatisfiability. Their main idea is to perform an indeterministic non-backtracking local search over the *variable space* to find a solution that satisfies the *cnf*. This local search strategy has shown to be robust and could outperform other systematic SAT solvers as presented in [SLM92], [Gu92], and [HS99].

The local search starts with an initial variable assignment or initial state. If the current state does not satisfy the *cnf*, the search strategy is to move to an adjacent state that has a difference of one or more variables depending on its preset Hamming distance. For a Hamming distance of one, the neighboring states would be the states that only have one different variable assignment. The search strategy will do repeated moves until a satisfiable assignment is found or the time-out limit on moves is reached. The limit imposed for this type of algorithms should be high enough that satisfiable problems are detected with high accuracy. For the WalkSAT and GSAT algorithms investigated in this thesis, the Hamming Distance is set to one.

```
procedure SLSSAT(cnf, maxtries, maxflips)
    output: satisfying variable assignment for cnf
    for i := 1 to maxtries do /* outer loop */
        INIT_ASSIGN(V);
        for j := 1 to maxflips do /* inner loop */
            if V satisfies cnf then
                return V
            else
                CHOOSE_FLIP(f, V, cnf);
                V := V with variable f flipped;
            end
        end
    end
end
```

Figure 2.1: Stochastic Local Search Algorithm

A general outline for the Stochastic Local Search algorithm **SLSSAT** is given in Figure 2.1. **SLSSAT** algorithms are different in two aspects, namely: the generation of the initial assignment (**INIT_ASSIGN**) and the selection for the next state

(**CHOOSE_FLIP**). All the investigated **SLSSAT** algorithms have a common **INIT_ASSIGN** procedure that randomly chooses the initial assignment from the variable space according to a uniform distribution. Hence, we concentrate on the **CHOOSE_FLIP** procedure that differentiate the investigated **SLSSAT** algorithms. As shown in Figure 2.1, there are two limits imposed in the algorithms. As the algorithm repeatedly performs flips to the current state, we limit the number of repetitions to *maxflips*. When it reaches *maxflips* with no solution found, the algorithm would exit the inner loop and restart with a new initial assignment. This stage is essential for the algorithm to escape from the local minima in the variable space. It means that for **SLSSAT** algorithms there exists a state in the *variable space* from which a solution will not be reached without reinitializing the search. The second time-out stage ends the execution of the algorithm when a certain number of tries (*maxtries*) has been reached. In that case, the algorithm fails to prove satisfiability.

For the **CHOOSE_FLIP** procedure, the *score*, which is the number of clauses satisfied by variable assignment *V*, plays a crucial role in the selection for the next variable to flip. We declare some *score* and additional functions that will be used in the following sections.

1. The function *score(cnf, V)* returns the number of clauses satisfied as a results of using a variable assignment *V* in *cnf*.
2. The function $score_f(i, cnf, V)$ returns the number of clauses in *cnf* that are satisfied by using the modification of the assignment *V* where the truth value of the *i*-th variable is inverted.
3. The function $score_b(i, cnf, V)$ returns the number of clauses in *cnf* that would be broken (unsatisfied) when the truth value for the *i*-th variable in *V* is flipped.

4. The function **CHOOSE_ONE** returns an element from a sequence using uniform distribution.

5. The function **UNSATISFIED** returns a sequence of unsatisfied clauses from *cnf* for the variable assignment of *V*.

2.2.1 The GSAT Architecture

The greedy local search procedure called GSAT was first introduced by Selman, Levesque, and Mitchell [SLM92] and Gu [Gu92] in 1992. Since then, a number of GSAT variants have been derived such as GSAT with Tabu Search (GSAT/TABU) [MSK97, MSG97, SSS97] and GSAT with History (HSAT) [GW93]. Figure 2.2 shows the **CHOOSE_FLIP** procedure used by GSAT. The procedure **CHOOSE_FLIP** gathers the variables that produce the highest $score_f$ in the sequence named *scores* and performs a random selection in function **CHOOSE_MAX** to determine the next variable *f* to flip. This algorithm is referred to as 'greedy' since it assumes that a neighboring *state* with the highest $score_f$ would have the highest probability leading to a solution.

```
procedure CHOOSE_FLIP(f, V, cnf)
    output: variable f that produces the maximum score
    for i := 1 to n do      /* for all variables */
        scores[i] := score_f(i, cnf, V);
    end
    return CHOOSE_MAX(scores);
end
```

Figure 2.2: CHOOSE_FLIP Algorithm for GSAT

A straightforward implementation of GSAT in Figure 2.2 from [SLM92] is rather inefficient, since for each call to CHOOSE_FLIP the scores for all the variables are recalculated. An implementation of GSAT by Selman and Kautz version 41 (GSAT41) is an optimized software implementation that usually serves as a reference benchmark implementation. Their method to efficiently implement GSAT is to evaluate the affected *scores* of some variable after each variable flip. A detailed description of GSAT41 together with a complexity analysis is given in [Hoo96].

2.2.2 The WalkSAT Architecture

The WalkSAT architecture is based on ideas first published by Selman, Kautz, and Cohen in 1994 [SKC94] and it was later formally defined as an algorithmic framework by McAllester, Selman, and Kautz in 1997 [MSK97]. WalkSAT is a family of stochastic algorithms that assigns all the variables a random truth assignment and then attempts to heuristically refine the assignment until all the clauses evaluate to true. WalkSAT is based on a 2-stage variable selection process focused on the variables occurring in currently unsatisfied clauses. For each local search step, in a first stage a currently unsatisfied clause *c'* is randomly selected. In a second step, one of the variables appearing in *c'* is then flipped to obtain the new assignment. Thus, while the GSAT architecture is characterized by a static neighborhood relation between assignments with Hamming distance one, WalkSAT algorithms are effectively based on a dynamically determined subset of the GSAT neighborhood relation.

WalkSAT family is in general a kind of robust stochastic local search algorithm. In WalkSAT family, the specific method of varying the truth assignment defines the variant of WalkSAT. All variants share the common behavior of occasionally ignoring their heuristic and making a random refinement according to some fixed probability.

In our FPGA-based WalkSAT implementations described in Chapter 7, the algorithm we adapted is based on a variant called WalkSAT-B [MSK97]. Figure 2.3 briefly describes this algorithm.

In Figure 2.3, given a SAT problem instance in format *cnf*, a random truth assignment *V*, and a noise setting *p*, the procedure will return a variable *f* which will be the next to be flipped. The function **UNSATISFIED** returns a list of clauses that are unsatisfied by the assignment of *V*. Then randomly choose an unsatisfied clause *c* in this list. Following, with probability *p* choose *f* in *c* randomly; with probability *1-p* choose *f* with the smallest $score_b$.

```
        procedure CHOOSE_FLIP(f, V, p, cnf)
            output: variable f
            c := CHOOSE_ONE(UNSATISFIED(cnf, V));
            min := m; /* number of clauses */
            flip := 0; /* 0-list whose length is n (number of  variables) */
            with probability p choose f in c randomly;
            with probability 1-p choose f in c with following heuristic:
                for i := 1 to k do /* for each variable found in c */
                    $v_i$ := i-th variable in c;
                    $c_i$ := $score_b(v_i, cnf, V)$;
                    if $c_i < min$ then
                        flip[$v_i$] := 1;
                        min := $c_i$;
                    else if $c_i = min$ then
                        flip[$v_i$] := 1;
                    end
                end
                  f := CHOOSE_ONE(flip);
            return f
    end
```

Figure 2.3: Algorithm for WalkSAT-B Variant in WalkSAT Family

As discussed in [MSK97], it is well known that the performance of a stochastic local search procedure depends upon the setting of its noise parameter, and that the optimal setting varies with the problem distribution. It is therefore desirable to develop general principles for tuning the procedures. In [MSK97], they presented two statistical measures of the local search process that allow one to quickly find the optimal noise settings. These properties are independent of the fine details of the local search strategies, and appear to be relatively independent of the structure of the problem domains.

In Chapter 7, we investigate two extreme implementations based on the above WalkSAT variants by setting p to 1 (Random-Strategy) and 0 (Greedy-Strategy) respectively.

2.2.3 WalkSAT Variants

2.2.3.1 WalkSAT/TABU

A well-known search mechanism in WalkSAT family is called WalkSAT/TABU which uses Tabu Search [MSK97]. It uses the same two-stage selection mechanism and the same scoring function $score_b$ as WalkSAT and additionally enforces a *tabu* tenure. A local search can be stuck at a local minima when it actually performs variable flips over a certain variable pattern. In order to avoid the repeating patterns, all recently flipped variables are restricted from getting flipped again for a certain duration. This duration is usually based on the number of variable flips, which is often referred to as *tabu* tenure. With the addition of the *tabu* mechanism the local search will hopefully be forced to flip a different variable that breaks the pattern and escapes the local minima. This however is not a guaranteed performance and is only a heuristic. As for the length of the *tabu* tenure, there is still no formal function for it to attain the Probabilistic Approximate Completeness (PAC) property.

2.2.3.2 History Mechanism

The history mechanism, as the name implies, makes use of history information in guiding the local search of SLSSAT. Typically, in the situation where several variables with the same *score* arise, a random selection over uniform distribution is done. In this procedure, it would be possible to have variables that are never chosen even though they have been eligible many times. The history information eliminates this scenario by adding an additional step in the variable selection process whenever tie-breaking between variables is needed. This step would select the variables that are the least recently flipped. Although this may appear to be an unimportant addition to the algorithm, results from [GW93] show that SLSSAT combined with history provides superior performance.

2.2.3.3 Self-Tuning Implementation of WalkSAT

The ability of stochastic satisfiability solvers to successfully find a problem's solution depends on how the trade-off between random decisions and heuristic decisions is managed during the solution search. This trade-off is controlled by a parameter setting, typically called the noise, which ranges from 0% to 100%. The optimal noise setting can vary greatly depending on the specifics of the algorithm used and the problem being solved. For a particularly hard problem, whose solution is unknown, it would be very useful to know the optimal noise setting.

In [PK01], Donald J. Patterson and Henry Kautz presented an algorithm that uses a variant of WalkSAT [SCK94] to probe the parameter space of noise settings for the value which will maximize the probability of finding a solution. In [PK01], they introduce *Auto-WalkSAT* which is a general algorithm that automatically tunes any variant of the WalkSat family of stochastic satisfiability solvers.

In [PK01], their algorithm *Auto-WalkSAT* is able to successfully minimize the invariant ratio using a bracketed search supplemented with parabolic interpolation. The additional overhead of minimizing this ratio is very small, adding approximately one minute to the running time of the algorithm. Using a heuristic of adding ten percent noise to this value, *Auto-WalkSat* then efficiently solves many problems which critically depend on a proper noise setting.

2.2.3.4 Davis-Putnam Procedure + WalkSAT

WalkSAT is an incomplete method and is claimed to be more efficient than Davis-Putnam Procedure [DLL62] which is a complete method. However, WalkSAT may come into difficulties on big SAT instances with many variables. In [ZHZ02], Wenhui Zhang *et al.* improved the efficiency by combining the Davis-Putnam procedure and the WalkSAT algorithm.

In 1960, Davis Putnam introduced a resolution algorithm for solving propositional satisfiabilty, which is called as the Davis-Putnam algorithm [DP60]. After two years, Davis, Logemann and Loveland improved on the algorithm and developed the Davis-Putnam procedure [DLL62]. The former algorithm uses an elimination rule, while the latter which became more famous uses backtracking. Further references to both works became ambiguous, but are likely to refer to the Davis-Putnam Procedure. The detailed algorithm for Davis-Putnam procedure can be found in [DLL62] which is the backtracking search algorithm.

Davis-Putnam procedure is one of the most efficient complete search algorithm for SAT. Many systems based on this procedure have been implemented and many interesting problems have been solved by these tools. A major problem with DP is that it may have to go through a very large search space.

In [ZHZ02], a hybrid approach was adopted. Firstly, use the DP procedure partially, and produce some subproblems. Then the subproblems are given to WalkSAT. In [ZHZ02], there are two parameters for controlling the number of subproblems. One is the maximum depth to be searched by DP, the other is the maximum number of subproblems.

If a subproblem is proven to be satisfiable within the given depth, the satisfiability checking is also finished. Otherwise, the subproblems which have not yet been proven to be unsatisfiable are recorded in files. In each subproblem, the propositional variables are renumbered consecutively from 1 to the number of remaining variables. These subproblems are given to WalkSAT in a loop until a solution is found or the maximum number of repetitions is reached.

The advantage of partitioning a problem into subproblems compared to using WalkSAT alone is that each subproblem is much smaller than the original problem. The implication of this is that the time needed for each trial of such a subproblem with WalkSAT is much shorter; and a solution of such a subproblem is expected to be found with much less time, if this subproblem indeed has a solution. For hard SAT instances, the speed up with their approaches is significant.

Chapter 3

Reconfigurable Computing Paradigm

Reconfigurable computing is a new and emerging computing paradigm that uses reconfigurable hardware, like Field Programmable Gate Arrays (FPGAs), to implement computationally intensive tasks. An FPGA provides the benefits of a customized CMOS-VLSI chip, and at the same time, avoids the fabrication cost and inherent risk of using conventional masked gate array. Similar to current application-specific hardware accelerators, reconfigurable hardware benefits from the customization of data widths, instructions, memory access, etc. as compared to general-purpose computer. The resulting hardware can be optimally designed for the target application and exploits fine-grain parallelism.

3.1 General-Purpose Computer vs. Special-Purpose Computer

When we use the word "computer", we are normally referring to a general-purpose computer. By definition, general-purpose computers are computing machines that can be used for a wide range of applications. On the other hand, there are also special purpose computers used for a single application or a class of similar applications.

The design of a general-purpose computer takes into account a wide range of considerations and constraints. Through several generations, a family of general-purpose computers often maintains a relatively stable instruction set. There are many applications available for these computers. In addition, programming for such computers is very easy because there are many software tools available. General-purpose computers offer good performance on wide range of applications at a very reasonable price.

For a particular application, however, a general-purpose computer does not always provide the highest performance. When the performance requirement of certain applications exceeds the performance of the available general-purpose computer, there are different approaches to create higher performance computing machines to provide the necessary computing power. One way is through parallel computing. A number of general-purpose processors can be combined to form a parallel computer. Very high performance can be achieved by partitioning the problem into small pieces and letting many computers work in parallel to solve the problem. However, the application should be suitable for such parallel computing. Another approach is to build specialized computers according to the application to provide higher performance specially for this application. The application-specific approach may provide very high performance for the targeted application, often with less hardware usage than the parallel computing approach.

There is one major obstacle in building application-specific computing machines. That is the cost for designing and building such a computer. The initial cost for designing and manufacturing integrated circuit (ICs) is very high and the subsequent cost for fabricating the IC is relatively small. When an integrated circuit is fabricated in large quantities, the initial cost can be amortized and each chip produced is only responsible for a small portion of the initial cost. This is the major reason that popular general-purpose computers can be sold at relatively low prices. On the other hand, special-purpose computers require special-purpose integrated circuits. The initial cost is so high that it may dominate the total cost of building such system. It lacks the economy of scale.

Another difficulty with the special-purpose approach is the development time. It often takes a very long time to develop such a system, because a large amount of work is involved. Because the performance of general-purpose computer improves very quickly, special-purpose hardware may become obsolete very soon.

Taking into the cost and short life, special-purpose computers are not an attractive approach unless the need for such hardware is very strong. However, if the cost and development time can be significantly reduced, this can be a viable approach for many problems.

3.2 Field Programmable Gate Array (FPGA)

Reconfigurable computing is a novel approach that combines the strengths of general purpose computing and the special-purpose approach. The research for reconfigurable computing is motivated by pursuit of higher computing performance with modest hardware cost. The advances in integrated circuits has brought about the class of programmable logic devices that can achieve high computing performance and yet provide the flexibility of gate-level programming. The typical hardware device used for reconfigurable computing is Field Programmable Gate Array (FPGA) [BFRV92, Sha99]. The basic idea of reconfigurable computing is to build a hardware system based on FPGAs or other programmable devices. This system is configured, or programmed, according to a particular application to achieve high performance. On the other hand, the hardware should be general enough that many different applications can be mapped to the same hardware and run quickly.

The advent of reconfigurable computing bridges the gap between general-purpose processors and special-purpose computers or accelerators. It blurs the distinction between hardware and software. The study of reconfigurable computing also brings together knowledge on computer architecture, parallel computing, compilers, software development, hardware and IC design, and VLSI CAD.

A general-purpose computer has a fixed instruction set. Different applications are implemented using different software programs. User programming is performed at the instruction level. Reconfigurable computing takes a different approach. There is no fixed instruction set. Instead of a general-purpose processor, reconfigurable computing uses FPGAs as the computing elements. The FPGAs are essentially integrated circuits that can be configured into specific logic functions. Different applications are realized by different configurations for hardware. The user programming can be performed at the logic gate level. Reconfigurable computing achieves high performance by creating specific functional units and better exploiting the parallelism.

A reconfigurable hardware system normally cannot operate as a stand-alone machine. It should work in tandem with a general-purpose processor, called a host machine. The host machine should handle the operating system and many basic functions such as program loading, file I/O and control functions. There can be different coupling mechanisms between the reconfigurable computer and the host. There have been proposals and recent design work on very closely coupled architectures, in which the processor and the FPGA are located on the same chip

[RS94, RLG98]. There are less closely coupled systems, in which the FPGA communicates with the processor through some I/O bus [GHK91, VBR96]. This has an impact on the communication bandwidth and latency, hence how the application is implemented. It will affect the programming model and performance model of the implementation.

Secondly, there are differences in total logic capacity of reconfigurable systems. The number of FPGA chips ranges from one to a few dozens or even thousands. The logic capacity determines the maximum complexity of application that can be mapped to hardware. It places an upper limit of parallelism that can be exploited.

There are also differences in the programming model of reconfigurable hardware. In some systems, all instructions are compiled into an FPGA hardware configuration. In other systems the reconfigurable hardware supports a limited instruction set. In this case, the programming model bares some similarity with general-purpose processor with the added flexibility in the instruction set. An application can be either fully implemented on reconfigurable hardware or partitioned between reconfigurable hardware and a general-purpose computer.

An FPGA is a type of programmable device, wherein a general-purpose chip can be configured to perform a wide variety of applications. The first programmable device that has achieved widespread use was the PROM (Programmable Read-Only Memory). PROMs, a one-time programmable device come in two basic versions: the Mask-Programmable Chip programmed only by manufacturer, and the Field-Programmable Chip programmed by the end-user. The Field Programmable PROM developed into two types, the Erasable Programmable Read-Only Memory (EPROM) and the Electrically Erasable Programmable Read-Only Memory (E^2PROM). The E^2PROM has the advantage of being erasable and re-programmable many times.

Another step took place in this field which lead to the development of the Programmable Logic Device (PLD). These devices were constructed to implement logic circuits. The PLD include an array of AND-gates connected to an array of OR-gates. The PAL (Programmable Array Logic) is a commonly used PLD consisting of a programmable AND-plane followed by a fixed OR-plane. PALs come in both mask and field versions. The PAL was designed for small logic circuits.

The Mask-Programmable Gate Array (MPGA) was developed to handle larger logic circuits. A common MPGA consists of rows of transistors that can be interconnected to implement desired logic circuits. User specified connects are available both within the rows and between the rows. This enables implementation of basic logic gates and the ability to interconnect the gates. As the metal layers are defined at the manufacturer, significant time and cost are incurred in producing the run. In 1985, Xilinx Inc. introduced the FPGA (Field Programmable gate Array). An FPGA is a universal logic device structures as an array of user programmable logic and I/O cells interconnected by a programmable routing network.

There are four FPGA technologies in use: static Ram cells, anti-fuse, EPROM transistors, and E^2PROM transistors. For this discussion, we focus on the static RAM technology on symmetrical array configuration developed by Xilinx. In the static RAM FPGA, programmable connections are made using pass-transistors, transmission gates, or multiplexers that are controlled by SRAM cells. Only SRAM cells allow fast in-circuit reconfiguration for any number of times. The major disadvantage, on the other hand, is the size requirement of the RAM technology.

3.2.1 Principle of FPGA

FPGAs are based on the structure of Look-Up-Table (LUT), and LUT is essentially a RAM. Currently, most FPGAs adopt 4-input LUT, thus each LUT can be viewed as a 16-deep and 1-bit RAM with a 4-input address line. From a schematic or VHDL code, the synthesis tool computes all possible results and writes these results into the RAM.

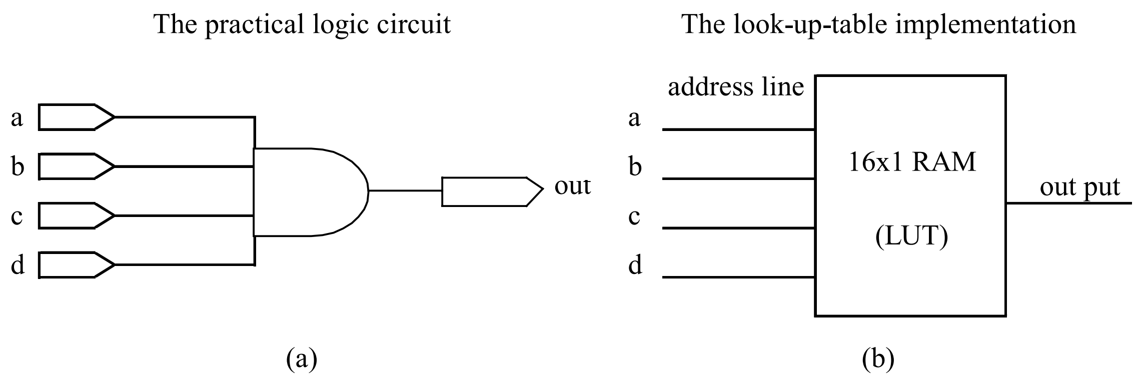

Figure 3.1: A Four- Input AND Gate Example

Figure 3.1 shows a 4-input AND gate example. Sub-figure (a) describes the schematic of the practical logic circuit of a 4-input AND gate; sub-figure (b) is the Look-Up-Table implementation respondent to sub-figure (a).

4-input AND Gate		16-deep and 1-bit RAM	
input of "abcd"	logic output	address line "abcd"	data in 16x1RAM
0000	0	0000	0
0001	0	0001	0
0010	0	0010	0
0011	0	0011	0
0100	0	0100	0
0101	0	0101	0
0110	0	0110	0
0111	0	0111	0
1000	0	1000	0
1001	0	1001	0
1010	0	1010	0
1011	0	1011	0
1100	0	1100	0
1101	0	1101	0
1110	0	1110	0
1111	1	1111	1

Table 3.1: Implementing a Four-Input AND Gate with the LUT in FPGA

In Table 3.1, column 1 shows the input signals “abcd” of the four-input AND gate, column 2 shows the expected output of the four-input AND gate when input signals are as in column 1. Column 3 shows signals on the 4-bit address line of the 16x1 RAM, column 4 shows the data stored in this 16x1 RAM and addressed by “abcd” shown in column 3. Thus, a four-input AND gate can be implemented with the Look-Up-Table structure in FPGA.

3.2.2 Structure of FPGA

An FPGA is an integrated circuit (IC) that can be programmed after manufacture. Since it is re-programmable on the field, it is a kind of reconfigurable hardware. Typical architecture of an FPGA comprises a regular array of *Configurable Logic Blocks* (CLBs) with routing resources for interconnection and surrounded by programmable *Input/Output Blocks* (IOBs). CLBs provide the functional elements for constructing logic while IOBs provide the interface between the pins of the package and the CLBs. FPGAs are widely used as a prototype before fabricating a VLSI design, or can be used directly in a product. Figure 3.2 shows the basic structure of Xilinx SRAM-based FPGAs.

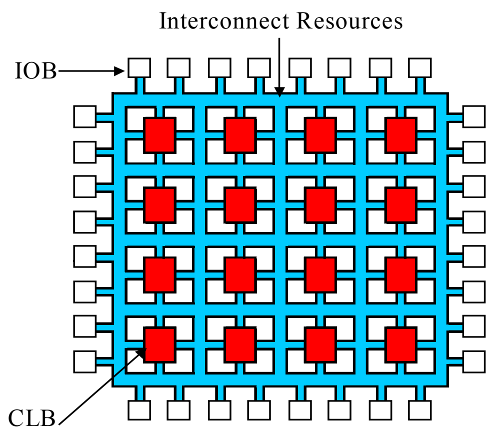

Figure 3.2: Basic Structure of Xilinx SRAM-based FPGAs

The structure of Xilinx Virtex IOB is shown in Figure 3.3. The three D-type flip-flops are synchronized on the same clock. Two of them are for input and output, and the other one is for the control to the output tri-state buffer. The input signal can be routed to the internal logic either directly or through an input flip-flop. A programmable delay element at the D-input of the input flip-flop is to eliminate the pad-to-pad hold time. Moreover, by configuring the threshold voltage V_{ref} at the input buffer, the device can support designs with different voltage level. Similarly, the output from the internal logic can be routed to the pad either directly or through the optional output flip-flop. All I/O pins involved in configuration are set to high impedance state so that the internal logic is isolated.

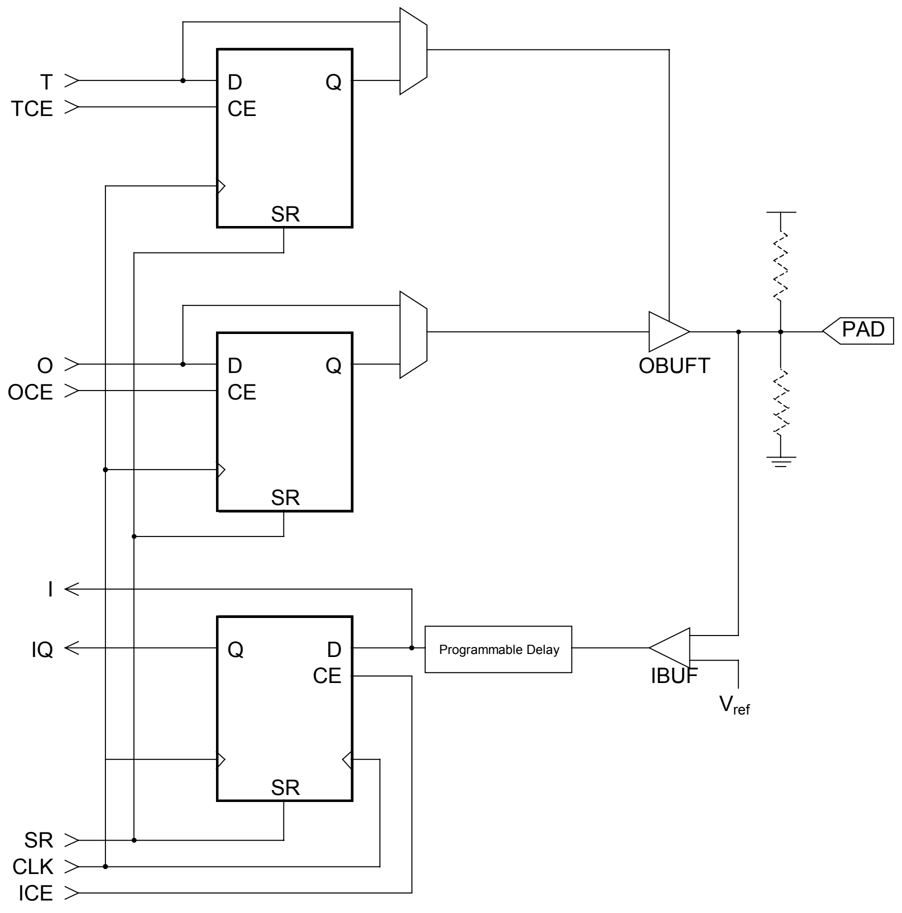

Figure 3.3: Structure of Xilinx Virtex IOB

The basic building block of the Xilinx Virtex FPGA is the Logic Cell (LC). A LC includes a 4-input function generator, carry logic and a storage element. Each Virtex CLB contains four LCs, organized in two slices (Figure 3.4). The 4-input function generator are implemented as 4-input look-up tables (LUTs). Each of them can provide the functions of one 4-input LUT or a 16x1-bit synchronous RAM(called "distributed RAM"). Furthermore, two LUTs in a slice can be combined to create a 16x2-bit or 32x1bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM [Xil00].

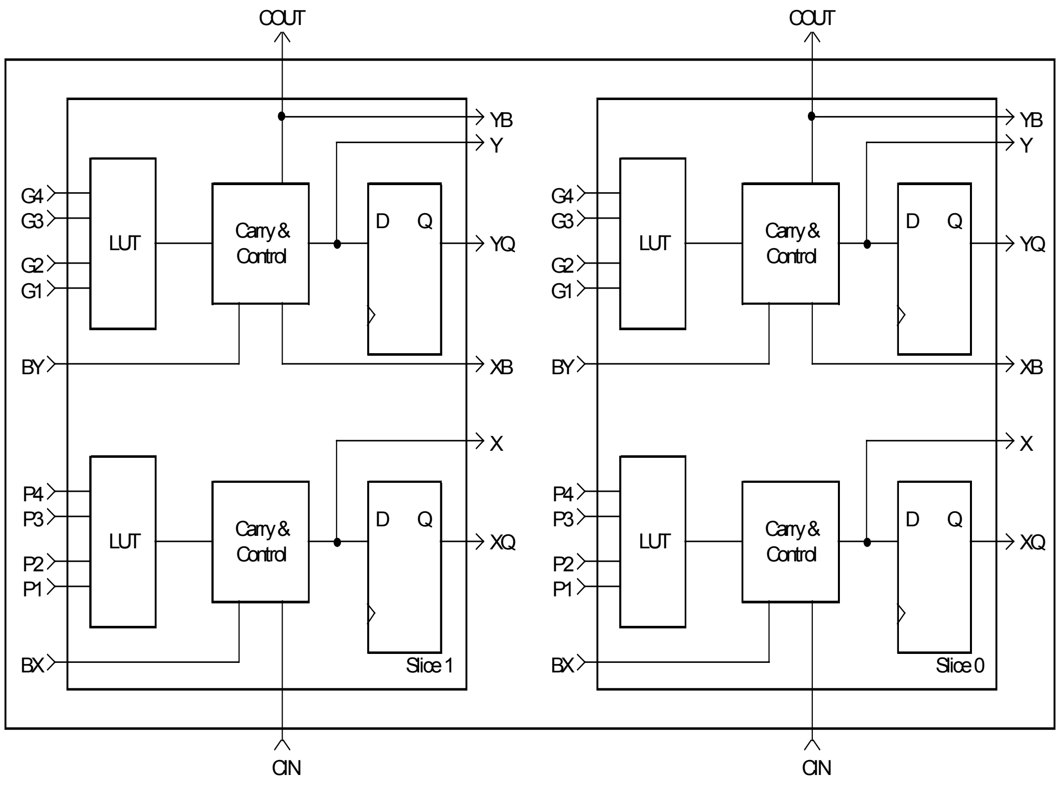

Figure 3.4: Simplified Structure of Xilinx Virtex CLB

Chapter 4

Current SLS SAT Hardware Implementations

There has been considerable recent interest in the application of FPGAs as accelerators for solving SAT problems. Most previous research on using FPGAs as accelerators for solving SAT problems has concentrated on complete algorithms. Complete algorithms are guaranteed to find a solution if one exists, whereas incomplete algorithms like stochastic local search may not find a solution even if one exist as we have discussed in Chapter 2.

For the complete algorithms, Zhong *et al.* developed a design for SAT problems utilizing the Davis-Putnam algorithm [ZMAM98a] as well as an unimplemented design which used nonchronological backtracking [ZAMM98].

Yokoo et al [YSS96] developed a machine based on FPGAs which implemented a tree search with forward checking for SAT problems. Implementations from Abramovici and Saab [AS97] can also be used to solve for SAT problems. A path-oriented decision making (PODEM) algorithm [Goe81] was used to solve for an encoded SAT problem. This algorithm was developed primarily for Automatic Test-Pattern Generation (ATPG) problems and does not perform quite well with SAT problems. In addition, Suyama et al [SYS98] developed a machine with a dynamic variable ordering heuristic. These approaches are less efficient than the Davis-Putnam procedure as stated in their paper. All of these implementations didn't outperform state of the art DP based algorithms.

Due to the inherent algorithm complexity of the DP SAT algorithm, it is not feasible to extract more parallelism than the implementation in [ZMAM98a]. Our research will focus on the FPGA implementations of WalkSAT algorithms which is a robust family in stochastic local search. In this chapter, we first review two recent implementations of GSAT [HTY01] and WalkSAT [Tan02]. These two implementations can achieve "one flip per clock cycle" performance. After that, another two implementations for GSAT [YSLL99] and WalkSAT from [LSW01] are discussed.

4.1 One Flip per Clock Cycle for GSAT

This section reviews the implementation of GSAT given by Henz, Tan, and Yap [HTY01]. In their work, they showed how GSAT can be implemented to be as fast as possible in hardware. Their implementation using FPGA achieves one flip per clock cycle by exploiting maximal parallelism and at the same time avoiding excessive hardware cost in terms of gates.

The speed of the GSAT implementations given in Hamadi and Merceron [HM97] and Yung et al. [YSLL99] is limited, because only clause evaluation is parallelized but variable scoring is not, hence the minimal depth of CHOOSE_FLIP after applying pipelining will still have a factor of n (n is the number of variables).

In the algorithm shown in Figure 2.2, Henz *et al.* found that there is no dependency between the score computation of different variables. Thus, this is obviously another parallelism opportunity. Figure 4.1 shows this naive maximum parallelism strategy.

```
procedure CHOOSE_FLIP($f$, $V$, $cnf$)
    output: variable f that produces the maximum score
    par (for $i$ := 1 to $n$ ) do /* for all variables */
        $scores[i] := score_f(i, cnf, V)$;
    end
    return CHOOSE_MAX($scores$);
end
```

Figure 4.1: Basic CHOOSE_FLIP Design with Parallelized Variable Scoring

In Figure 4.1, with key word **par,** the algorithm compute $score_f\,[1]$ to $score_f\,[n]$ in parallel. The depth of the this algorithm is *O(log m)* (m : the number of clauses), since the $score_f$ computation is bounded by *O(log m + log n)*, the CHOOSE_MAX computation is bounded by *O(log n)*, and we assume $n < m$. While this is closer to achieving their goal, the drawback is that the cost in gate increases by a factor of *n* to $O(mn^2)$. With the exception of small problems, this design will not be practical.

In [HTY01], they turned to an alternative hardware design. The ideas are related to the software optimizations for GSAT but the rationale is to decrease the circuit size while keeping parallel score evaluation. The key observations are:

1. The selection of the flip variable can be done on the basis of relative contribution to the score of that variable when flipped.
2. The number of clauses which will be affected by a change to one variable is small and typically bounded.

In [HTY01], Henz *et al.* developed a new procedure as shown in Figure 4.2. As only the affected clauses should be referred, function $score_c(i,\ cnf_{c(i)},\ V)$ and function $score_c(i,\ cnf_{c(i)},\ V'[\neg V(i)/i])$ are used. Function $score_c(i,\ cnf_{c(i)},\ V)$ returns the number of clauses satisfied as a result of using a variable assignment *V* in $cnf_{c(i)}$, while function $score_c(i, cnf_{c(i)}, V'[\neg V(i)/i])$ returns the number of clauses satisfied as a result of using a new variable assignment $V'[\neg V(i)/i]$. The new variable assignment $V'[\neg V(i)/i]$ is generated from the old variable assignment *V* when *V* is changed with the *i*-the variable is flipped. The notation $cnf_{c(i)}$ represents the set of clauses which contain variable *i*. For a particular SAT problem, $cnf_{c(i)}$ is constant. Thus, for each variable *i*, a fixed Boolean function can be extracted from $cnf_{c(i)}$ in order to get *OldS[i]* and *NewS[i]*.

The bound on the maximum number of clauses per variable can be denoted by *MaxClauses*. In practice, most SAT problems have also a bound on the number of variables per clause, which can be denoted by *MaxVar*. For example, for 3-SAT, *MaxVars* is 3. Thus, the number of gates for procedure in Figure 4.2 is *O(MaxVars MaxClauses n)*. The depth for it is *O(log MaxClauses + log MaxVars)*, which for practical SAT problems is much smaller than *O(log m)*. One more advantage of their design is that the circuit for $score_c$ is also smaller because the actual size of the numbers to be considered requires less bit.

```
procedure CHOOSE_FLIP(f, V, cnf)
        output: variable f that produces the maximum score
s1:   par (for i := 1 to n ) do /* for all variables */
            NewS[i] := $score_c(i, cnf_{c(i)}, V'[\neg V(i)/i])$;
      end
s1:   par (for i := 1 to n ) do /* for all variables */
            OldS[i] := $score_c(i, cnf_{c(i)}, V)$;
      end
s2:   par (for i := 1 to n ) do /* for all variables */
            Diff[i] := NewS[i] – OldS[i];
      end
s3:   f := CHOOSE_MAX(Diff);
end
```

Figure 4.2: Parallel CHOOSE_FLIP with Relative Scoring

With the above procedure the innermost loop of GSAT is over flips. Unfortunately, it is not possible to pipeline the different flip iterations of CHOOSE_FLIP, since each iteration is dependent on the flip of the previous iteration. Instead, pipelining the outer loop of the procedure show in Figure 2.1 is available, which is called multi-try pipelining in [HTY01]. Since there is no dependency between different tries in GSAT, essentially one can parallelize each try independently. Each pipeline stage deals in parallel with the work for a different try. For simplicity, *maxtries* should be a multiple of the number of stages in the pipeline.

In practice, in the actual implementation it is feasible in one clock cycle to accommodate the $score_C$ for all variables. Therefore, to achieve one flip per clock cycle for GSAT is only need to allocate each design block in the procedure in Figure 4.2 to a pipeline stage *s*, leading to a pipeline with four stages. The first three stages, *s1* to *s3* are labeled in the procedure in Figure 4.2. The last stage, *s4*, which is not in the CHOOSE_FLIP procedure, is the circuit to make actual flip. This is illustrated in Figure 4.3, where procedure in Figure 4.2 is implemented as a four-stage pipeline which gives one flip per clock cycle.

Tries	time1	time2	time3	time4	time5	time6	time7	time8	…
Try1	*s1*	*s2*	*s3*	*s4*	*s1*	*S2*	*s3*	*s4*	…
Try2		*s1*	*s2*	*s3*	*s4*	*S1*	*s2*	*s3*	…
Try3			*s1*	*s2*	*s3*	*S4*	*s1*	*s2*	…
Try4				*s1*	*s2*	*S3*	*s4*	*s1*	…

Figure 4.3: A Four Stage Pipeline for GSAT

4.2 One Flip per Clock Cycle for WalkSAT

In this section, we review another FPGA-based implementation which is of WalkSAT algorithm and also achieved one flip per clock cycle in [Tan02].

The WalkSAT algorithm is technically an offspring of a GSAT variant, GSAT with random walk [SKC94]. For this reason, Tan *et al.* [Tan02] adapted many implementation details from GSAT in [HTY01] which is reviewed in the previous section. The algorithm they used is as the procedure shown in Figure 2.3, and they set the noise parameter N to 0%.

WalkSAT uses a function $score_b$ that counts for the number of clause that will be unsatisfied when a variable is flipped. It is found that the clause evaluation as the procedure in Figure 4.1 is ideal for a fast WalkSAT solver design. For GSAT, the procedure in Figure 4.1 is truly impractical due to the large size increase to a factor of n. But for a WalkSAT implementation of a 3-SAT problem, the increase of the hardware size is only a factor of 3.

Figure 4.4 shows the complete instance-specific WalkSAT hardware design in [Tan02]. The main computation is divided into six data dependent stages, labeled *s1* to *s6*. In stage *s1*, the function CHOOSE_ONE selects one unsatisfied clause *c* from *Cp* using uniform distribution; *Cp* contains the sequence of unsatisfied clauses from the last iteration. This stage also determines whether the last iteration has produced a satisfying solution; all the clauses are satisfied when SUM(*Cp*) is equal to zero. For the next stage *s2*, the VARIABLE_LIST(*Vp*, *j*, *i*) returns the variable sequence *V'p* with the *i*-th variable in clause *j* inverted. It is assumed that there are three variables per clause, therefore, there are *V[1], V[2], V[3]* to store the variable assignments with different variable flipped. The next stage *s3* evaluates the variable assignments to the *cnf* and then forms a list of unsatisfied clauses for each of the variable assignments. Stage *s4* computes for the $score_b$ for each of the three variable assignments (Function $score_b$ is discussed in Section 2.2). The next stage *s5* determines the variable assignment that produced the least $score_b$. In the next stage *s6*, the new variable assignment will be updated, as well as the list of unsatisfied clauses. This loop would repeat until a satisfying solution is found or the *maxflips* number of iterations is reached.

```
MAIN():
        $V_p$ := RECEIVE_INITIAL_ASSIGNMENT();
        $C_p$ := {1: $j \in [1\ldots m]$};
        for $i$ := 1 to maxflips do
s1:         par{
                if SUM($C_p$) = 0 then BREAK;
                $c$ := CHOOSE_ONE($C_p$);
            };
s2:         par{
                $V[1]$ := VARIABLE_LIST($V_p$, $c$, 1);
                $V[2]$ := VARIABLE_LIST($V_p$, $c$, 2);
                $V[3]$ := VARIABLE_LIST($V_p$, c, 3);
            };
s3:         par{
                $C[1]$ := {$\neg$EVAL$_j$($V[1]$) : $j \in [1\ldots m]$};
                $C[2]$ := {$\neg$EVAL$_j$($V[2]$) : $j \in [1\ldots m]$};
                $C[3]$ := {$\neg$EVAL$_j$($V[3]$) : $j \in [1\ldots m]$};
            }
s4:         par{
                $S[1]$ := SUM( $\neg C_p \wedge C[1]$);
                $S[2]$ := SUM( $\neg C_p \wedge C[2]$);
                $S[3]$ := SUM( $\neg C_p \wedge C[3]$);
            }
s5:         $i$ := OBTAIN_MIN_INDEX($S$);
s6:         par{
                $V_p$ := $V[i]$;
                $C_p$ := $C[i]$;
            }
        end;
        SEND_ASSIGNMENT($V_p$)
```

Figure 4.4: Instance Specific Implementation of the WalkSAT Algorithm

As we can see from the procedure in Figure 4.4, the innermost loop of WalkSAT is also over flips. Just like the implementation of GSAT, it is impossible to pipeline the different flip iterations of CHOOSE_FLIP since the data dependency between the consecutive flips. Instead, pipelining the outer loop of the procedure show in Figure 2.1 is also available for WalkSAT, which is called multi-try pipelining in [HTY01]. Since there is no dependency between different tries in WalkSAT, essentially one can parallelize each try independently, in this way, one flip per clock cycle for WalkSAT is achieved.

4.3 GSAT Variant by Yung *et al.*

In this section, we will review another FPGA-based GSAT implementation which was given by Yung *et al.* [YSLL99].

Although the implementations discussed in section 4.1 and 4.2 can run at one flip per clock cycle and can get performance gains of about two orders of magnitude over software, their approach are not practical as a general SAT problem solver, because the time to re-synthesize, place and route the new design for a new SAT problem is likely to significantly exceed the runtime improvement from the faster solvers. In section 4.3 and 4.4, we will review two implementations which address this problem.

From 1999, bitstream reconfigurable systems have been employed to address the re-synthesis problem occurring in instance-specific implementations for solving SAT problems. In [YSL99], Yung *et al.* provide a method of modifying the bitstream in a problem specific fashion without requiring re-synthesis. Like [ASS99], the runtime configurable systems in [YSL99] also used Xilinx XC6200 series devices [Xil6200] 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 capability (The largest reported bitstream reconfigurable system only supports 13 variables and 29 clauses [ASS99].).

The difference in the work by Yung et al. [YSLL99] is the use of partial re-synthesis of the design that bypasses the synthesis. Their design technique is only possible with two assumptions. First, the device vendor like in their case Xilinx Inc. has provided enough information to reconstruct their configuration file for the Xilinx XC6216 FPGA. Secondly, the changes to their design should be simple and should do not affect the timing constraints

Yung *et al.* was able to provide partial reconfiguration to the FPGA given the advantage of knowing how to construct the configuration file. Their approach allows reconfiguration that skips the synthesis tool and allows directly changing the configuration of the FPGA. Current FPGA chips do not provide an open architecture thus rendering this technique useless. Xilinx has currently announced that they would release future FPGA chips that would allow partial reconfiguration. Partial reconfiguration will allow CLB rows to be configured separately and could reduce synthesis time by a factor. This technology has yet to come out and it would improve the performance of instance-specific design implementation.

Since the algorithm used in [YSLL99] was patterned after the algorithm provided by Sleman, Levesque and Mitchell in [SLM92] rather than the optimized version as in GSAT41 [Hoo96], the respondent FPGA-based implementation in [YSLL99], like that in [HM97], was not fully parallelized. Thus the implementation in [YSLL99] didn't provide enough performance increase compared with the GSAT implementation we discussed in the previous section which significantly improved over GSAT41 running on fast CPUs.

4.4 WalkSAT based on ROM Array

In 2001, Leong *et al.* [LSW01] achieved a bitstream reconfigurable FPGA implementation for WalkSAT. The algorithm they adopted is as the procedure shown in Figure 2.3. In their implementations, the noise *N* is set to 100%. Their implementation stores clauses for a SAT problem in the 16x1-bit ROM available in the Logic Cells (LCs) of the Xilinx FPGA. A different SAT instance requires various ROM definitions to be modified. Normally this would require re-synthesis of the FPGA to generate a new bitstream configuration for downloading. Leong *et al.* were able to achieve an implementation without requiring re-synthesis by designing a transformer for the ROM configuration.

In their scheme, 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 file has an extension *.ncd* which means Native Circuit Description.) is generated. This file can be opened with Xilinx tool *FPGA Editor*. *FPGA Editor* is a graphical application for displaying and configuring FPGAs. The *FPGA Editor* can read from and write to NCD files, macro files (NMC), and Physical Constraints Files (PCF). Under the environment of *FPGA Editor*, the names and physical locations of those LCs, by which the ROM arrays of the clause checker are implemented, can be found.

At the same time, with another kind of Xilinx tool named *ncd2xdl*, the binary-format bitstream file *.ncd,* which stores the contents of the circuit, can be converted into a human readable format, and then, with the information regarding the names and the physical locations of the LCs of the ROM array acquired under *FPGA Editor*, data stored in these LCs can be extracted and modified.

In [LSW01], a program was written which takes as input the normal *.ncd* file and the specification of a specific SAT problem in the standard DIMACS benchmark format [DIMACS]. For each SAT problem, this transformer designed modifies the bitstream *.ncd* file according to the SAT problem specification by customizing the ROM values and recomputed the Cyclic Redundant Check (CRC) of the *.ncd* file. After that, the resulting bitstream file *.bit* generated by Xilinx tool *bitgen* can be downloaded to a Virtex FPGA to find a solution for this SAT problem instance.

In their scheme, they elect to recalculate the CRC checksum inside their software transformer. In this way, they can avoid running the Design Rules Checker (DRC) when recreating the configuration *.bit* file. CRC bits are checksum bits that the FPGA uses to verify that the bitstream transmitted correctly.

This approach requires analysis of the bitstream *.ncd* file to figure out how to rebuild the configuration without re-synthesis. Like [YSLL99], the implementation in [LSW01] simulates re-synthesis in a very efficient fashion. However, it is also dependent on the ability to modify the FPGA configuration.

Chapter 5

Clause Evaluator without Re-Synthesis

Instance specific implementations for SAT problems have provided an outstanding performance from their compact sizes. This is achieved by using a customized design that is specific for each problem. The disadvantage of these implementations is that a high level description of a circuit customized for a particular SAT problem is needed. In order to execute the design, an entire iteration of the synthesis, map, place and route (P&R) cycle was required for each problem. These steps are time consuming (it can take several hours to synthesize, map, place and route a large design.) and preclude their use in real time systems. Our goal is to develop a general system which avoids these steps. We develop a general clause evaluator for WalkSAT solvers, which fits well within an FPGA architecture and can be reconfigured according to different SAT problems quickly in a portable fashion. In this Chapter, Section 5.1 discusses the compilation (synthesis) time on current platform in order to demonstrate the shortcoming of the instance-specific implementations. Section 5.2 describes our general clause evaluator.

5.1 Compilation Time on Current Platform

For instance-specific designs, since the circuit is generated according to the specific SAT problem to be solved, the problem solving time should take into account compilation time. In this section, we investigate the actual compilation time for instance-specific designs by reviewing the implementations which achieved one flip per clock cycle in [Tan02]. These implementations are based on GSAT and WalkSAT strategies respectively.

As shown in Table 5.1 and Table 5.2, for instance-specific implementations using FPGAs, the following steps contribute to the total compilation time.

1. **Handel-C Synthesis (Syn):** This is a process called logic synthesis which compiles a Handel-C project into a Electronic Design Interface Format (EDIF) netlist file. EDIF netlist is a standard netlist format which describes a circuit including the basic elements and their connections. This process takes the Handel-C project as input and then generates the circuit structure implementing the functions described in the Handel-C.
2. **Xilinx mapping (Map):** The EDIF netlist uses generic constructs to describe the circuit while the FPGAs have their own logic functional units. For example, a netlist can express combinational circuits in terms of AND, OR and inverter gates. The target FPGA uses the CLBs to realize logic functions. Fitting the logic gates into the LUTs in the CLBs is called technology mapping. After mapping, the circuit is represented by the functions of the CLBs and the routing newtwork between these CLBs.
3. **Xilinx placement and routing (Par):** This is the placement and routing of physical design. The task of placement is to determine the location of the

logic functions on the target FPGA. The placement of the logic functions should facilitate later routing. A good placement should minimize the routing congestion and routing delay. Typically, placement is optimized through iterative improvement after an initial constructive placement. With the logic elements in place, routing takes care of creating the connections between these elements. Since the routing resources are limited, there is no guarantee that a circuit can be routed. It may take several tries to get an acceptable routing.

4. Xilinx bitstream generation (Bitg): After the logic functions and routing are all determined, Xilinx's bitstream generation program, BitGen, takes a fully routed circuit description file as its input and produces a configuration *bitstream* – a binary file. This *bitstream* file contains all of the configuration information.

5. Download configuration: This process download the bitstream file into the FPGA's memory cell. On our current AMD Athlon 1.2GHz CPU it takes about 0.14 seconds or so.

SAT Problems	Var	Cla	Slices	Syn (min)	Map (min)	Par (min)	Bitg (min)	Total (min)
uf20-01	20	91	17%	10	1	2	2	15
aim-50-1_6-yes1-1	50	80	18%	10	1	1	2	14
aim-50-2_0-yes1-1	50	100	20%	13	1	2	2	18
aim-50-3_4-yes1-1	50	170	31%	40	3	3	2	48
aim-50-6_0-yes1-1	50	300	54%	171	6	6	2	185
aim-100-1_6-yes1-1	100	160	34%	43	3	4	2	52
aim-100-2_0-yes1-1	100	200	40%	63	5	4	2	74
aim-100-3_4-yes1-1	100	340	64%	199	11	14	2	226
flat30-01	90	300	57%	198	7	8	2	217
BMS_k3_n100_m429_0	100	286	56%	199	7	8	2	216
RTI_k3_n100_429_0	100	429	79%	293	19	12	2	326
uf50-01	50	218	39%	55	5	3	2	65
uf100-01	100	430	89%	294	19	12	2	327

Table 5.1: Time Spent on Re-synthesis for GSAT in Section 4.1

Table 5.1 shows the time spent on these steps for different problems for instance-specific one flip per clock cycle implementation for GSAT achieved in [HTY01]. Table 5.2 shows the time spent on the re-synthesis steps for different problems for the instance-specific one flip per clock implementation for WalkSAT achieved in [Tan02].

The compilation tools used in both Table 5.1 and Table 5.2 are Celoxica's Handel-C 3.0 and Xilinx Foundation Series ISE 3.1i; and the PC on which these tools ran is of an AMD Athlon 1.2GHz CPU.

Table 5.1 and Table 5.2 demonstrate compile-time statistics for two subsets of the SAT suite. The total compilation time to generate hardware solution for a specific big SAT problem can be of several hours for big problems.

SAT Problem	Var	Cla	Slices	Syn (min)	Map (min)	Par (min)	Bitg (min)	Total (min)
aim-100-1_6-yes1-1	100	160	28%	13	2	2	2	19
aim-100-2_0-yes1-1	100	200	25%	13	2	2	2	19
aim-100-3_4-yes1-1	100	340	38%	18	11	2	2	34
aim-100-6_0-yes1-1	100	600	51%	33	6	4	2	45
aim-200-1_6-yes1-1	200	320	52%	74	7	5	2	88
aim-200-2_0-yes1-1	200	400	53%	79	5	5	2	91
aim-200-3_4-yes1-1	200	680	74%	113	21	8	2	144
aim-200-6_0-yes1-1	200	1200	99%	170	94	52	2	318
aim-50-1_6-yes1-1	50	80	14%	3	1	1	2	7
aim-50-2_0-yes1-1	50	100	15%	4	1	1	2	8
aim-50-3_4-yes1-1	50	170	20%	5	2	1	2	10
aim-50-6_0-yes1-1	50	300	27%	8	3	2	2	15
BMS_k3_n100_m429_0	100	286	37%	25	3	3	2	33
Flat30-1	90	300	29%	13	3	2	2	20
RTI_k3_n100_m429_0	100	429	48%	33	4	4	2	43
uf100-01	100	430	47%	23	4	4	2	33
uf200-01	200	860	96%	120	31	14	2	167
uf20-01	20	91	12%	2	1	1	2	6
uf50-01	50	218	24%	6	2	1	2	11

Table 5.2: Time Spent on Re-synthesis for WalkSAT in Section 4.2

Since the compilation time is on the order of hours, the implementations in [Tan02] will not provide practical speedups for problems that can be solved in minutes or less by software approach. As we can see from Table 5.1 and Table 5.2, for instance-specific implementations, basically the hardware space cost is proportional to the size of the SAT problem; and the more the space cost, the longer the compilation time. Figure 5.1 and Figure 5.2 demonstrate a relation between the space cost of a design and the compilation time of this design.

As we can see, hardware compilation problems such as optimal partitioning and placement are quite complicated, and hardware compilation time can be on order of hours, research for means to reduce synthesis time is being done. Our method is to develop a general clause evaluator in WalkSAT rather than instance-specific, which fits well within an FPGA architecture and can be reconfigured quickly in a portable fashion.

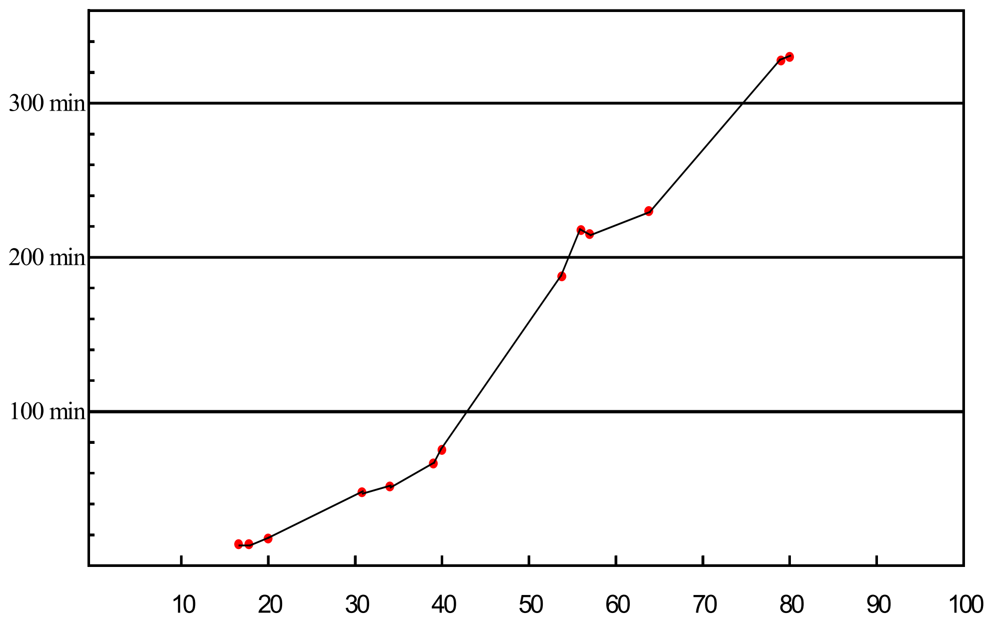

Figure 5.1: Space Cost & Compilation Time for GSAT Instance-Specific Designs

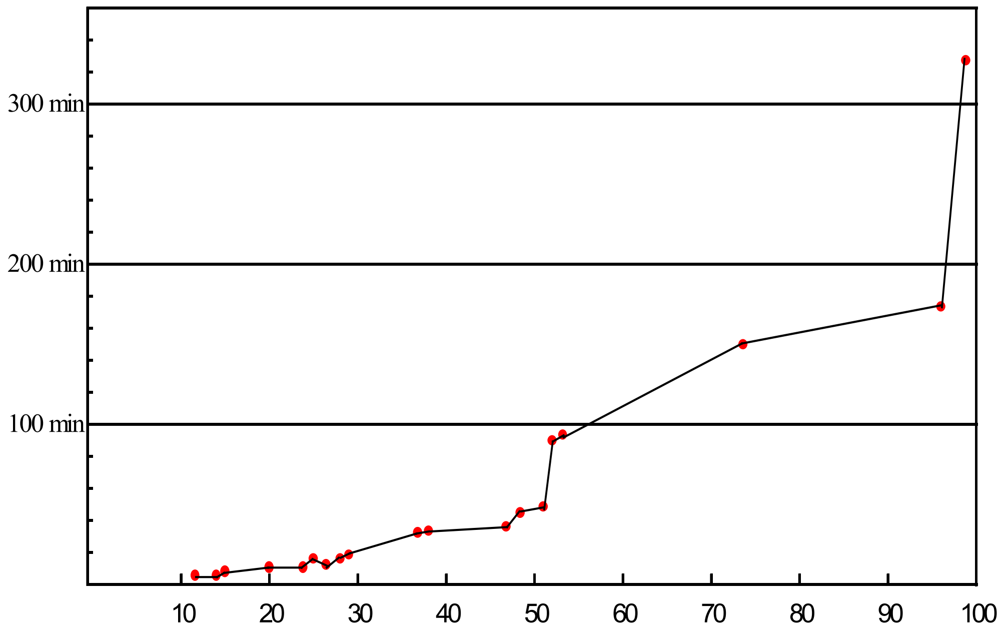

Figure 5.2: Space Cost & Compilation Time for WalkSAT Instance-Specific Designs

5.2 A General Clause Evaluator

As we have discussed in the previous section, the compilation overhead limits the usage of instance-specific implementations. Our method is to develop a general clause evaluator in a SAT solver, which avoids those re-synthesis steps described in Section 5.1, and at the same time this clause evaluator should fit well in an FPGA architecture and can be reconfigured quickly in a portable fashion.

5.2.1 Decompose one Clause into Small Boolean Function Blocks

In our design, we will focus on the Xilinx Virtex FPGA chips. As we mentioned in Chapter 3, two LUTs in a slice can be combined to create a 16x1-bit dual port RAM.

Our clause evaluator represents the clauses in a SAT instance in a 16x1-bit dual port RAM array, which can be generated from the Xilinx RAM16x1D primitive.

5.2.1.1 Function of RAM16X1D

RAM16x1D is a 16-word by 1-bit static dual port random access memory with synchronous write capability. The device has two separate address ports: the read address port (DPRA3 – DPRA0) and the write/read address port (A3 – A0). These two address ports are completely asynchronous. The read address controls the location of the data driven out of the output pin (DPO), and the write/read address controls the destination of a valid write transaction and also the data driven out of the output pin (SPO). This means SPO output reflects the data in the memory cell addressed by A3–A0. DPO output reflects the data in the memory cell addressed by DPRA3–DPRA0. The write process on the write/read port won't be affected by the address on the read address port. Figure 5.3 shows the external pins of RAM16x1D. Figure 5.4 gives the function block diagram of RAM16x1D.

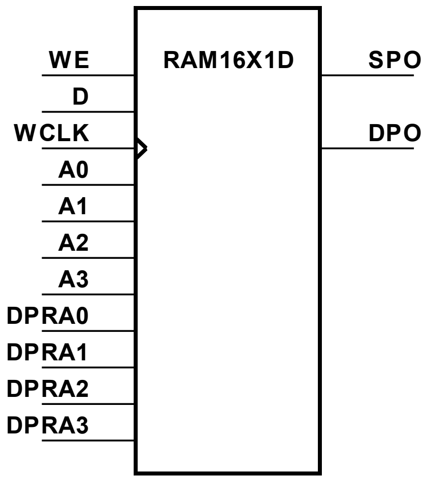

Figure 5.3: External Pins

Inputs			Outputs	
WE (mode)	**WCLK**	**D**	**SPO**	**DPO**
0 (read)	X	X	data_a	data_d
1 (read)	0	X	data_a	data_d
1 (read)	1	X	data_a	data_d
1 (write)	↑	D	D	data_d
1 (read)	↓	X	data_a	data_d
data_a = word addressed by bits A3-A0 data_d = word addressed by bits DPRA3-DPRA0				

Table 5.3: Mode Selection of RAM16x1D

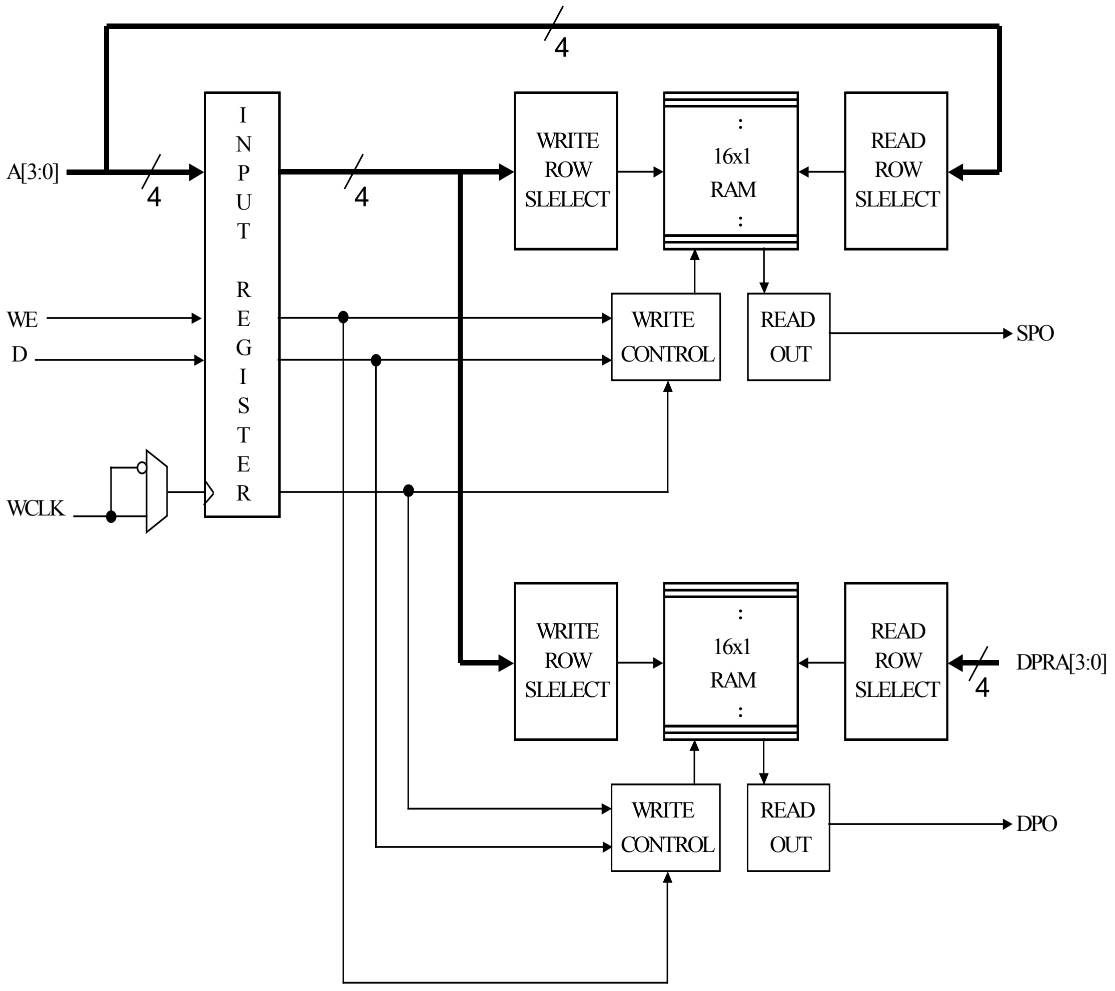

Figure 5.4: Function Block Diagram of RAM16x1D

When the write enable (WE) is Low, transitions on the write clock (WCLK) are ignored and data stored in the RAM is not affected. If we assume an active-High WCLK, when WE is high, any positive transition on WCLK loads the data on the data input (D) into the word selected by the 4-bit write address. For predictable performance, write address and data inputs must be stable before a Low-to-High transition. Table 5.3 shows the mode selection for RAM16X1D. WCLK can be active-High or active-Low.

5.2.1.2 Map Boolean Functions to RAM16x1Ds

After describing the function of RAM16x1D, we describe how to map a clause in a SAT problem to RAM16x1Ds by an example. Consider a SAT clause, *c3*, of the form, $x1 \vee x2 \vee \neg x15$, and let us assume that *c3* is a clause of a SAT problem over 20 variables. The clause can be written as $f_{3,0}(x0, x1, x2, x3) \vee f_{3,1}(x4, x5, x6, x7) \vee f_{3,2}(x8, x9, x10, x11) \vee f_{3,3}(x12, x13, x14, x15) \vee f_{3,4}(x16, x17, x18, x19)$, where $f_{3,0}(x0, x1, x2, x3) = x1 \vee x2$, $f_{3,1}(x4, x5, x6, x7) =$ **FALSE**, $f_{3,2}(x8, x9, x10, x11) =$ **FALSE**, $f_{3,3}(x12, x13, x14, x15) = \neg x15$, $f_{3,4}(x16, x17, x18, x19) =$ **FALSE**; thus, clause *c3* can be decomposed into a disjunction of 5 small Boolean functions, each of these functions is over four consecutive Boolean variables. In this way, all the clauses of a SAT problem can be decomposed into a set of small Boolean functions.

signals on address A3A2A1A0	a row of 5 RAM16x1Ds respondent to clause : $x1 \vee x2 \vee \neg x15$				
	RAM_0	RAM_1	RAM_2	RAM_3	RAM_4
Input signal of each small function block	*x0x1x2x3*	*x4x5x6x7*	*x8x9x10x11*	*x12x13x14x15*	*x16x17x18x19*
	$x1 \vee x2$	FALSE	FALSE	$\neg x15$	FALSE
0000	0	0	0	1	0
0001	0	0	0	0	0
0010	1	0	0	1	0
0011	1	0	0	0	0
0100	1	0	0	1	0
0101	1	0	0	0	0
0110	1	0	0	1	0
0111	1	0	0	0	0
1000	0	0	0	1	0
1001	0	0	0	0	0
1010	1	0	0	1	0
1011	1	0	0	0	0
1100	1	0	0	1	0
1101	1	0	0	0	0
1110	1	0	0	1	0
1111	1	0	0	0	0

Table 5.4: Decompose a Clause into Small Boolean Function Blocks

We map each Boolean function $f_{i,j}$ arising from the j-th part of clause i to a RAM16x1D primitive, treating the four variables as the address to the read port (DPRA3 – DPRA0). The function $f_{i,j}$ is configured by using the write port (A3 – A0) to define its truth table. Note that one advantage of this representation is that negated variables are handled automatically inside the $f_{i,j}$ function block.

Table 5.4 shows the truth tables of the 5 Boolean functions whose disjunction represents the clause *c3* we mentioned above.

5.2.2 Hierarchical Structure of our Clause Evaluator

In the previous section, we described the principle of our general clause evaluator. With this principle, we can construct a clause whose variables vary over all the variables occurring in a SAT problem. In this section, we develop our general clause evaluator without requiring re-synthesis.

First of all, since more detailed descriptions in the register-transfer level (RTL) can be available in VHSIC (an acronym for Very High Speed Integrared Circuits) Hardware Description Language (VHDL), we designed our general clause evaluator in VHDL language. VHDL is a hardware description language that can be used to model a digital system. The digital system can be as simple as a logic gate or as complex as a complete electronic system. A hardware abstraction of this digital system is called *entity.* An entity X, when used in another entity Y, becomes a component for entity Y. Therefore, a component is also an entity, depending on the level at which you are trying to model. VHDL supports three basic different description styles: structural, dataflow, and behavioral.

Our clause evaluator is of hierarchical design and expressed in combination of these three descriptive styles. To understand our structural style of modeling, we should first understand the concept of entity in VHDL.

In structural design, an entity is modeled as a set of components connected by signals, that is, as a netlist. The behavior of the entity is not explicitly apparent from its structural model. The component instantiation statement is the primary mechanism used for describing structural model of an entity.

Our clause evaluator is hierarchically a three-level design. In this design, we developed three kinds of entities (components), plus a Xilinx RAM16x1D entity. The entity declaration specifies the name of the entity being modeled and lists the set of interface port. *Ports* are signals through which the entity communicates with the other models in its external environment. Our final clause evaluator is for up to 100 variables and 220 clauses. (See the Appendix for the declarations of the entities in our final clause evaluator.)

Figure 5.5 shows the relationship of the four entities inside our clause evaluator. Entity *clause_checker* has its architecture body *clause_checker_A*. In our design, architecture body *clause_checker_A* is a purely structural model with a hierarchy of three levels, and *clause_checker_A* itself is of the top-level. It contains 221 components, one is called *ctrl*, while the other 220 components are named *clause_0*, *clause_1*, … , *clause_219*.

The dashed lines represent the bindings of components used in the architecture body with other entities. Component *ctrl* in architecture body *clause_checker_A* is bound to entity *cc_ctrl*; component *clause_0*, clause_1, … , *clause_219* are all bound to entity *cc_clause*. Architectures of both entity *cc_trl* and entity *cc_clause* are of second-level.

The architecture body of entity *cc_ctrl* is *cc_trl_A*. It is of a mixed style of behavioral modeling and dataflow modeling, its function will be discussed in Section 5.2.3.

The architecture body of entity *cc_clause* is *cc_clause_A*. It is of a mixed style of structural modeling and behavioral modeling. Inside *cc_clause_A*, component *ram_0*, *ram_1*, … , *ram_24* are all bound to entity *ram16x1d*.

Hierarchically, the architecture body of entity *ram16x1d*, *ram16x1d_A*, is of third level. We have discussed the function of *ram16x1d* in Section 5.2.1.

Figure 5.6 shows the structure of the *i*_th clause in our clause evaluator. Since our clause evaluator is for SAT problems of up to 100-variable/220-clause, each clause is over 100 variables, thus it takes 100/4 = 25 RAM16x1Ds to store the Boolean function truth tables for each clause.

In Figure 5.6, 4-bit address line *ADDR[11..8]* are connected to the write address line *A3-A0* of all the 25 RAM16x1Ds. Data line *row_wdata[0], row_wdata[1]*, …, *row_wdata[24]* are connected to the data input *D*s of the 25 RAM16X1Ds in the clause row consecutively. *V0, V1, …, V99* are connected to the 4-bit read address line *DPRA3-DPRA0*s of the 25 RAM16X1d in each clasue row 4 by 4 consecutively. 25 *DPO*s in this clause row are "Or"ed together, the output of the 25-input OR gate is connected to *row_out[i]*, *i* means the *i*-th clause. *ROW_WEN[i]* is the write enable signal comes from the read/write controller (see Section 5.2.3).

Figure 5.7 shows the entire structure of our general clause evaluator for SAT problems up to 100 variables and 220 clauses. Our clause evaluator evaluates *all_clauses[0]* to *all_clauses[219]* in parallel in one clock cycle.

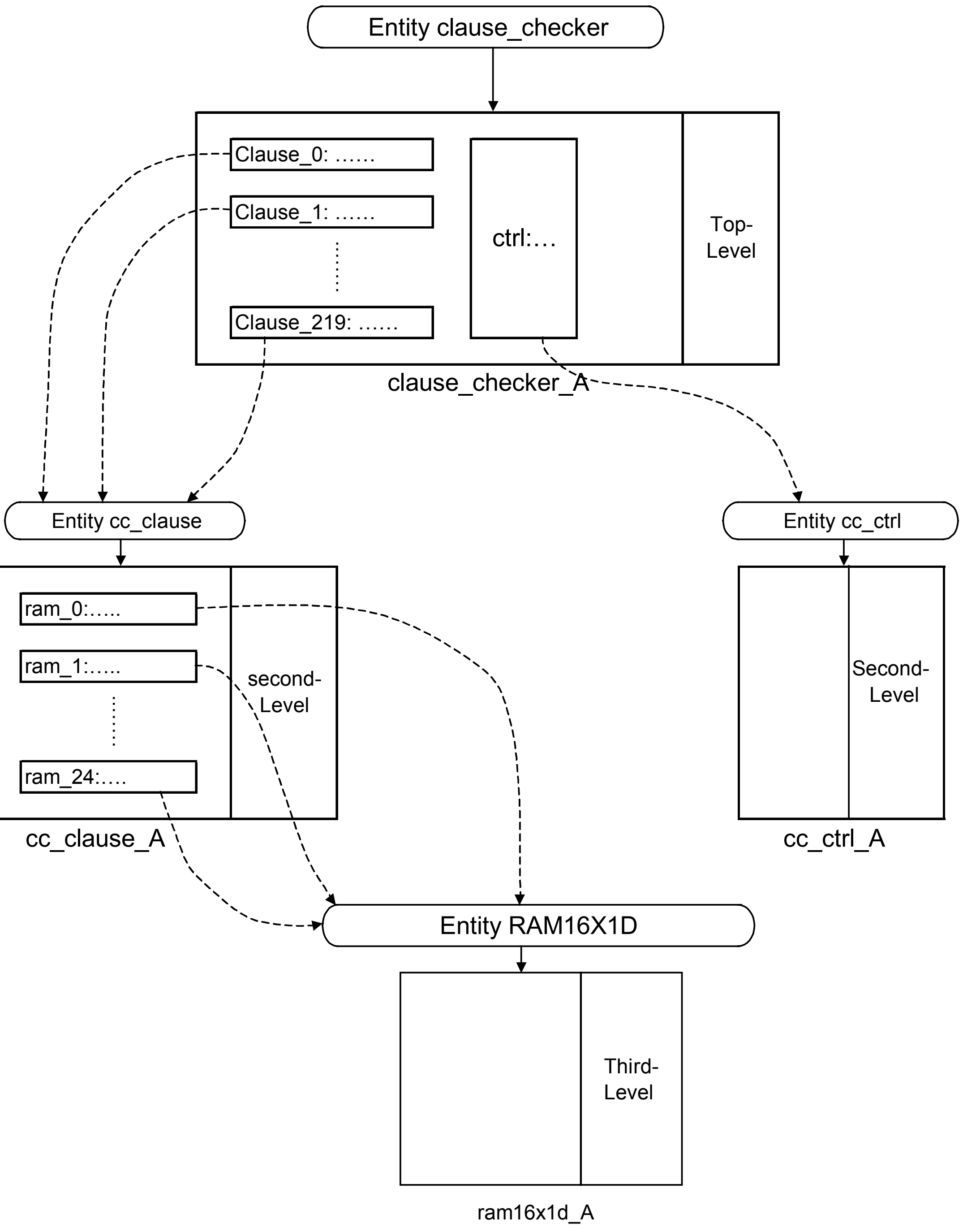

Figure 5.5: Hierarchy of a General Clause Evaluator

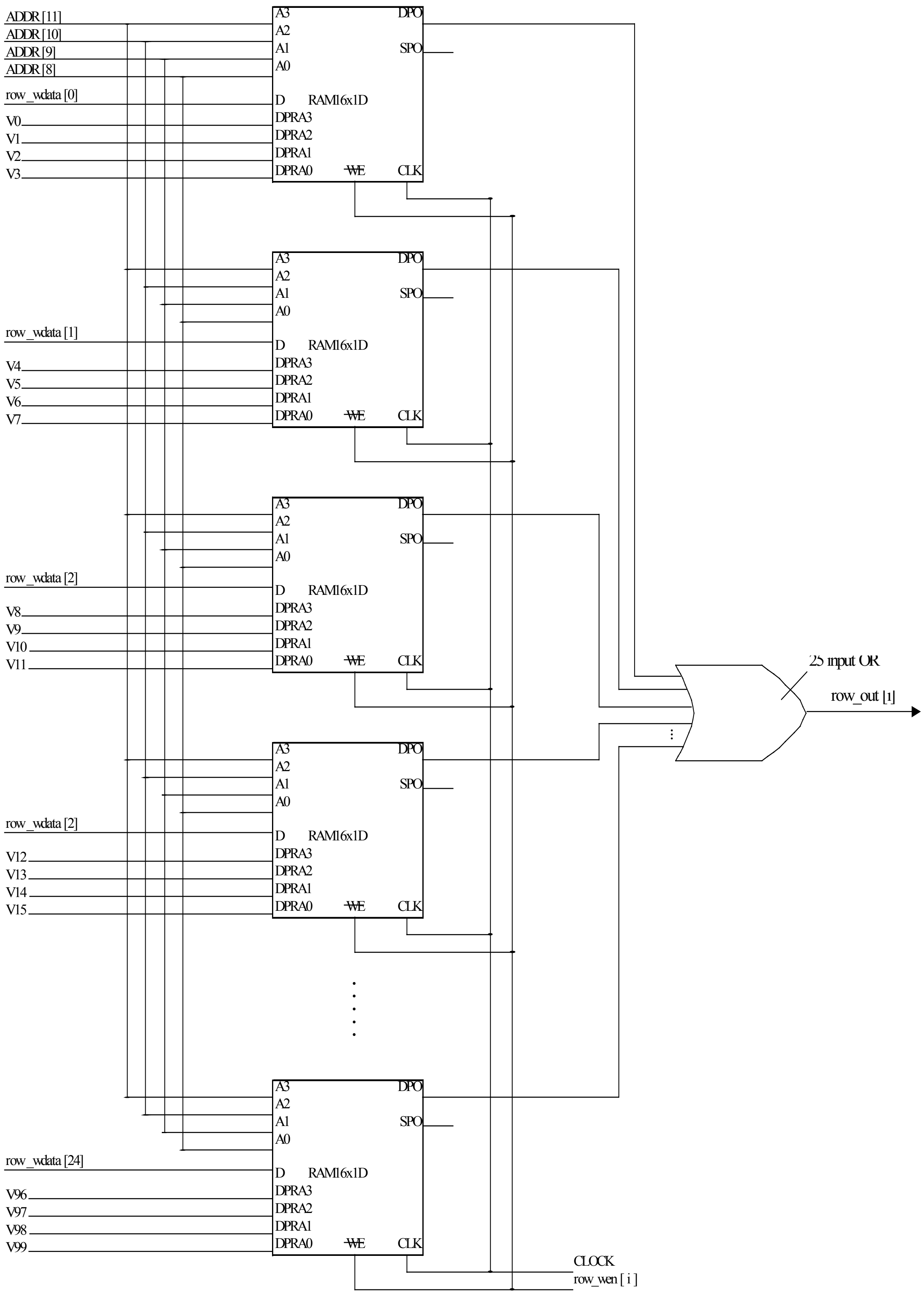

Figure 5.6: Structure of Clause *i*

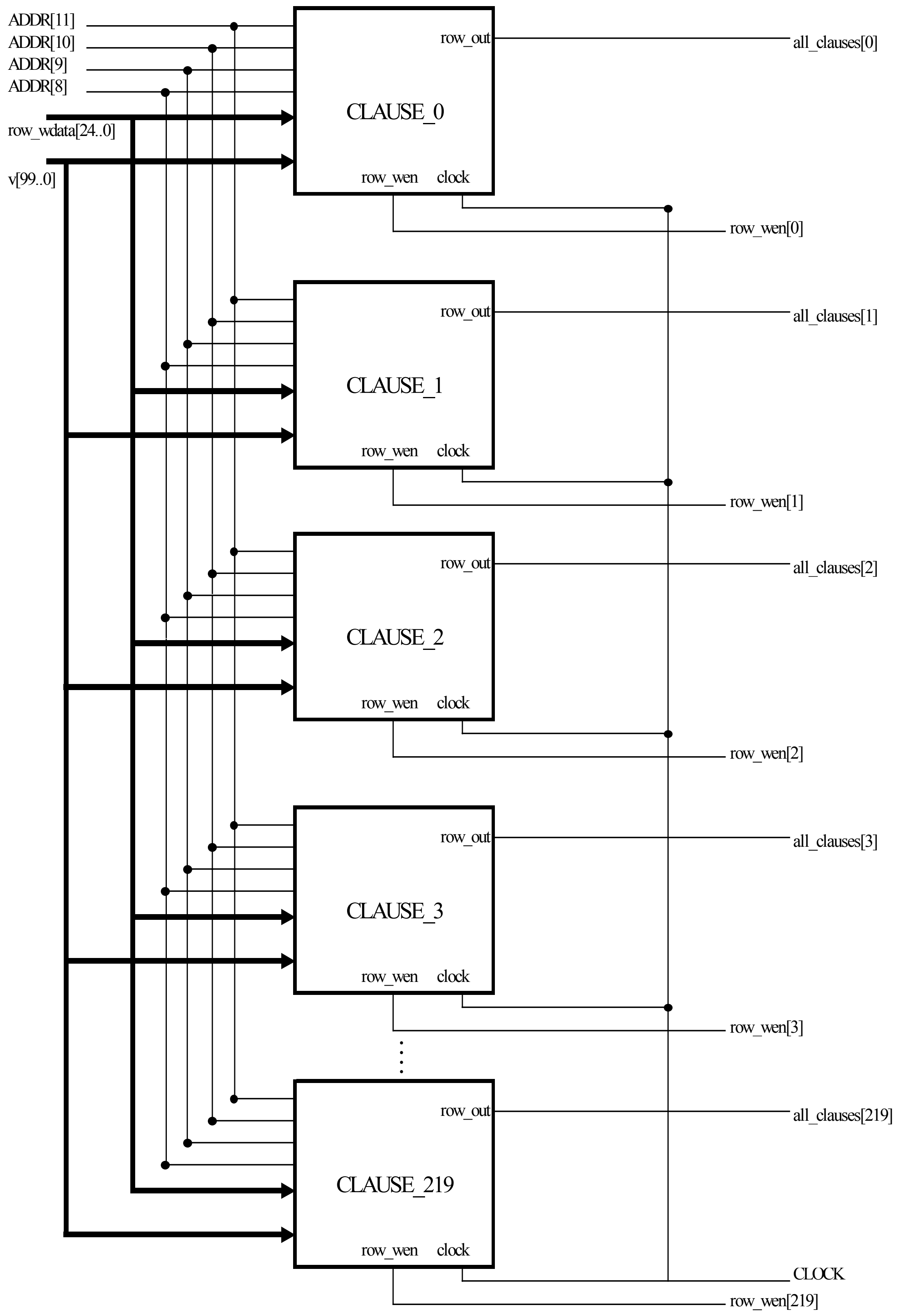

Figure 5.7: Structure of our General Clause Evaluator

5.2.3 Control Logic inside our Clause Evaluator

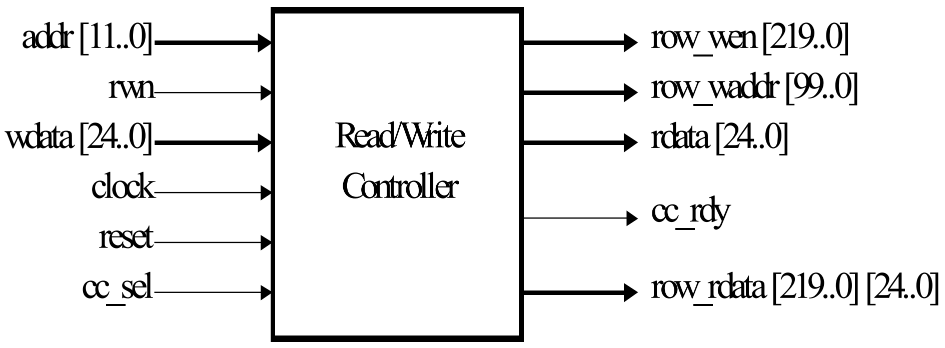

Figure 5.8: External Connections of Read/Write Controller

Figure 5.8 shows the external connections of the read/write controller around the 220x25 RAM16x1D arrays of our clause evaluator.

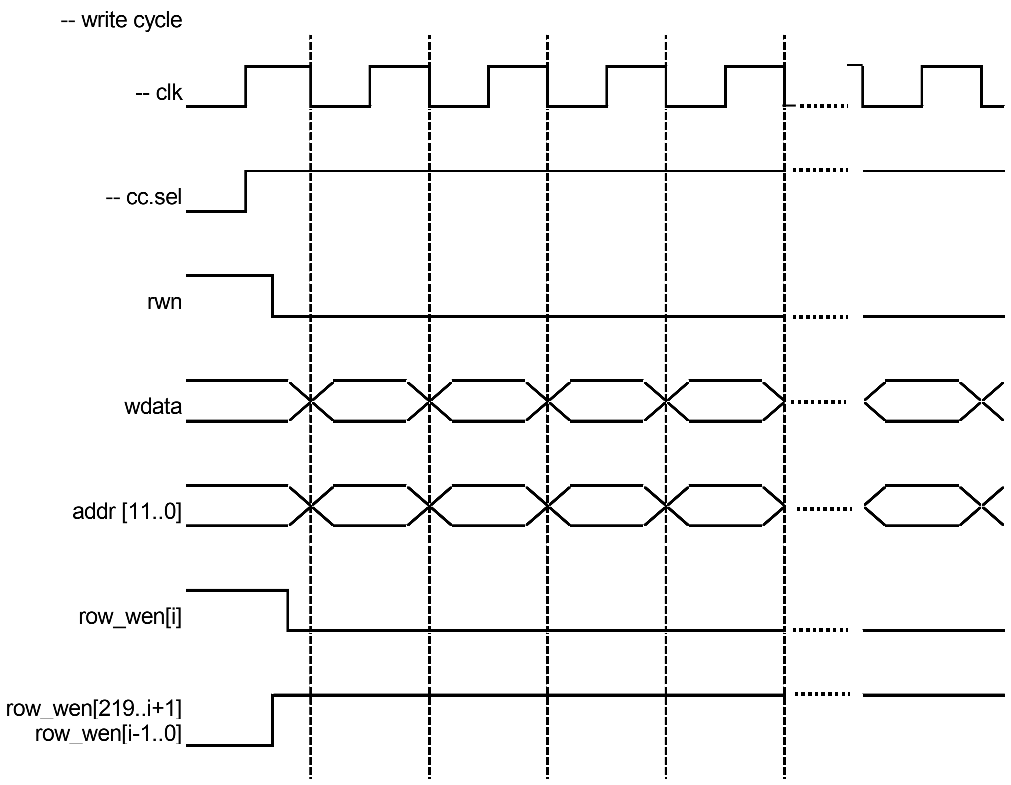

Figure 5.9: Waveform of Write Cycle

Figure 5.9 demonstrates the waveform of the write process. Signal *cc_sel* is the clause_checker ram array select signal, the clause_cheker ram array is selected when this signal is high. If we want to write data into the ram array of the clause checker, we should first set *cc_sel* to high level. After setting this signal we should flip read/write control signal *rwn* from high level to low level (we define this signal is a high active read signal that means: if it is high, data will be read out from the clause checker; if it is low, data will be write into the clause checker.). The controller has *addr[11..0]* as its address input. Among the 12 signals, *addr[7..0]* select which clause among the 220 clauses will be written, while *addr[11..8]* select one unit from the 16 units of each RAM16X1D. During each write clock cycle, 25-bit data on the writing date line *wdata[24..0]* will be written into the 25 RAM16X1Ds in parallel. Each bit represents the content of the unit addressed by *addr[11..8]* of the respondent RAM16x1D. Since we connect *addr[11..8]* to the 4-bit write port *A3 – A0* of all 25 RAM16X1Ds in each clause and connect *wdata[24..0]* to the input D pin of all the 25 RAM16X1D in this same clause, when we flip write clock from low to high, the 25-bit data *wdata[24..0]* will be written into each unit of the 25 RAM16X1d in the selected clause respectively.

Besides the write logic inside the controller, we also designed read logic whose function is to read the data addressed by the write address port *A3 – A0*. Although in our SAT implementations we needn't to read data from this port, but during the process of system development, we can use this read function to read the data we have already written into the RAM16X1D in order to check if our write function works correctly.

Chapter 6

Implementation Platform

For this chapter, we discuss the implementation platform in order to give enough background information to issues that influenced the development of the designs. In both of our implementations that we will discuss in this chapter, we divide the system into a software programming part and a hardware design part. The software program, which includes the FPGA software driver, were written in the "C++" language and compiled using the "Microsoft Visual C++" version 6.0. On the other hand, the hardware part is designed using a combination of a new high-level hardware programming language Handel-C and VHDL. In section 6.1, we firstly introduce Handel-C and VHDL, then we discuss the combination of these two languages. Section 6.2 gives the configuration of the RC1000-PP prototype board developed by Celoxica which is used as target FPGA platform of our implementations.

6.1 Handel-C vs. VHDL

6.1.1 The Handel-C Programming Language

Handel-C is a programming language for rapid prototyping of synchronous hardware designs that uses a similar syntax with conventional "C" with the addition of inherent parallelism. The output from Handel-C is a file that is used to create the configuration data for FPGAs.

The C-like syntax makes the tool appealing to software engineers with little or no experience of hardware. They can quickly translate a software algorithm into hardware, without having to learn about VHDL or FPGAs in detail. Fundamentally, as a functional language, Handel-C allows you to code complex algorithms without having to consider lower-level designs. Using Handel-C constructs, the development cycle for the creation and testing of FPGA designs can be accelerated. In addition, the package includes a library of basic functions and a memory controller to access the external memory on the FPGA board.

The timing model used by Handel-C is relatively simple and adheres to an idea that all instructions execute in one clock cycle. Handel-C allows arbitrary length for sequences and includes a parallel construction which can easily implement the parallel evaluation for these sequences. As individual statements execute in one clock cycle, the sequencing for instructions and loops fits accordingly. Variables are declared with fixed bit sizes, which is consistent with $O(1)$ assumptions for operations on integers. Handle-C is convenient for rapid prototyping as we observed a shorter development cycle than with traditional hardware design languages such as VHDL or Verilog. While VHDL and Verilog give finer control and possibly better performance, the GSAT implementations in [HTY01] demonstrated the efficiency of Handel-C designs.

6.1.2 VHDL Language Issues

VHDL is a hardware description language that can be used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description.

VHDL is a language, strongly influenced by Ada and Modula-2, originally intended for hardware documentation and modeling, whereupon due to growing popularity it was standardized in 1987.

VHDL is widely adopted as hardware modeling and design language in current electronic industry. VHDL features in hierarchical structures and both behavior and Register Transfer level (RTL) modeling and simulation.

However, VHDL does not completely succeed in abstracting from hardware and indeed requires knowledge of circuitry. VHDL talks of components, sensitivity to signals, and ports. There is also a timing model which specifies the time allocated to statements when simulating code.

Normally, more detailed descriptions in the register-transfer level (RTL) of VHDL are instantiated as components at the structural level. One approach to VHDL's rich expressiveness is to write only in structured VHDL whereby components from a pre-written library are selected from within the code. In particular, at RTL a restricted set of VHDL constructs can be employed to suit a particular synthesizer toolkit. Thus, in practice, VHDL introduces a multi-stepped programming environment.

While VHDL development follows the code, compile, execute cycle common with software development, the runtime environment is a hardware simulator instead of a host microprocessor. VHDL code once checked for syntax and type integrity will be compiled into a repository of hardware entities. Simulation consists of executing the top-level hardware entity (the test bench) and saving the time ordered sequence of events and outputs generated by test bench stimuli. Subsequent analysis and verification is facilitated by a variety of waveform viewers and output capture tools. However, it is essential to check correct working by means of runtime capture and verification routines coded into test bench modules, as visual checking of waveform display is insufficient.

Once stimulation has verified the functional correctness of VHDL source, the next phase is Logic Synthesis. Historically, synthesis was the step of translating the top-level design into a schematic diagram of gates and flip-flops. Software synthesis tools now automate this process by compiling source code into a netlist of gates and gate interconnections. The final stage involves passing the netlist into a layout tools, which is responsible for mapping logic onto device resources.

6.1.3 Discussion

Handel-C is best suited to rapid prototyping and proof of concept engineering rather than high-performance, optimal solutions. Handel-C allows hardware to be implemented without a digital background, although awareness of the underlying synchronous finite machine model and digital design tradeoffs is beneficial. The *par* statement in Handel-C provides a flexible and code-efficient mechanism for specifying parallelism of arbitrary granularity.

Specification at behavioral-level in Handel-C minimizes the amount of code compared to VHDL and insulates the software engineer from low-level hardware detail. The trade-off is that the Handel-C compiler controls implementation detail so there is no opportunity to perform the gate-level optimization that is available to

VHDL-trained engineers. Consequently, gate counts may be higher than a VHDL solution, and the inability to dictate logic to the FPGA technology mapping can result in lower clock speeds.

Handel-C is a product which has only just made the transition from a research phase to commercial exploitation. There has probably simply not been time to develop extensive support for modularity and debugging facilities, as is present in VHDL toolkits. As larger-scale projects are tackled, these two issues will become significant. Equally, long place-and-route times will make debugging aids imperative as programs become larger.

6.1.4 Combination of Handel-C and VHDL

Although there does a convergence of opinion now favor high-level languages as a means of creating high-level behavioral models for hardware and system-on-a-chip designers prefer C/C++ because partitioning between hardware and software can remain uncommitted for a longer time, we still choose a combination of Handel-C and VHDL as our hardware implementation platform.

There are two major advantages of this combination. Firstly, a number of macro procedures written in Handel-C are provided with the RC1000-PP prototype board which we used as our FPGA platform. These macros include: memory access macros which can be used to request ownership of one or more memory banks, to access the external memory banks on an RC1000-PP card, to release ownership of one or more memory banks; macro procedure to read a byte from the control port; macro procedure to write a byte to the status port; and *et al*. With these macro procedures we can design a wrapper in a succinct

Handel-C style, and achieve on board memory access functions and read/write control/status functions, in this way we can save a lot of time to design the interface between the FPGA and HOST in VHDL language. Secondly, on the other hand, by designing components in VHDL inside the wrapper, we can take advantage of capabilities of VHDL language and design the hierarchical structure of our general clause evaluator that we have described in Chapter 5. In this combinatorial way, we can bridge the gap between the high-level abstract version of a design and its actual implementation in a Xilinx FPGA, thus we can achieve high-performance SAT solving systems in Xilinx FPGAs. Figure 6.1 illustrates the design flow of our system.

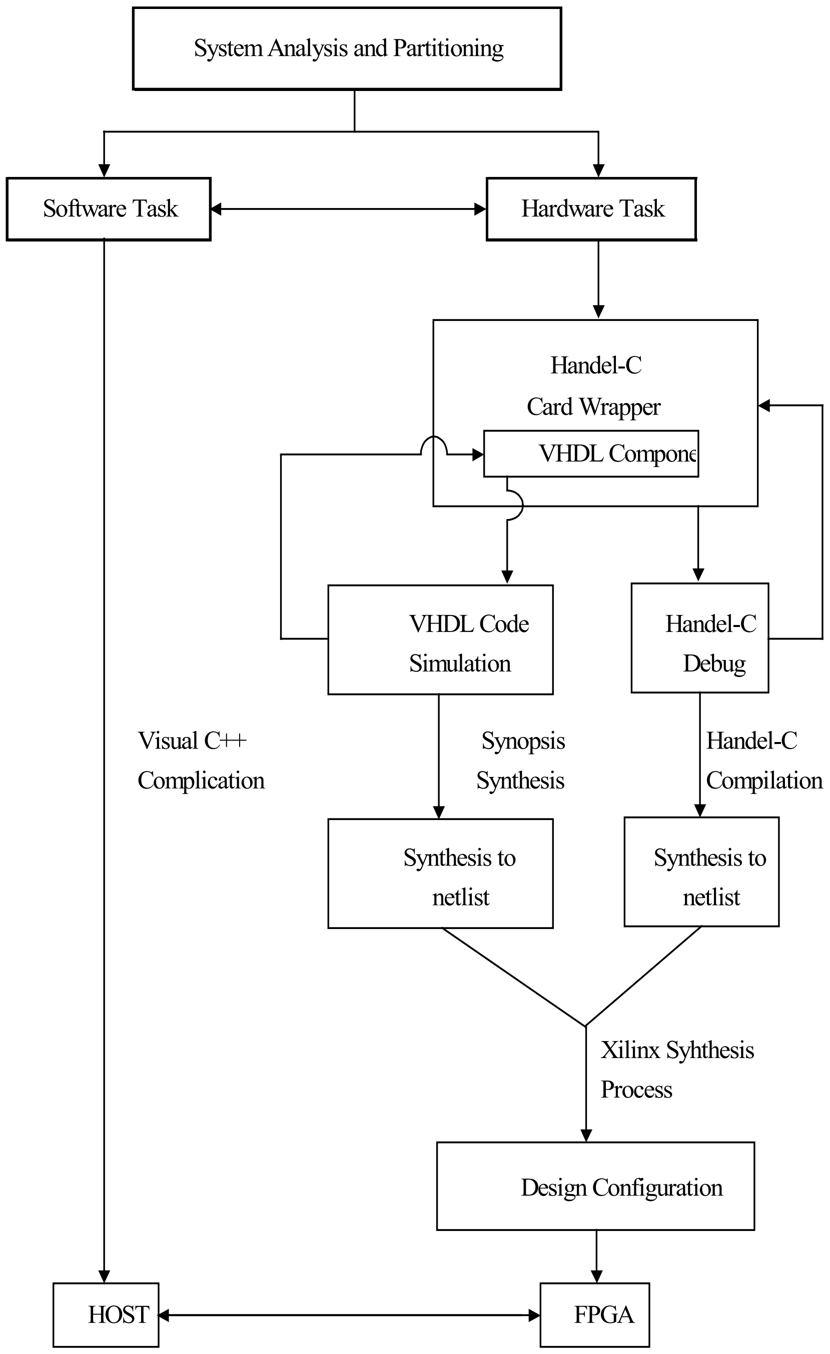

Figure 6.1: Design Flow of Handel-C and VHDL Combinatorial Method

6.2 RC1000-PP Prototyping Board

Currently, there are a lot of different FPGA prototyping board that can be found in the market, with each having different architecture and FPGA chips installed. For our implementations, we used the RC1000-PP development board specially made by Celoxica for use with the DK1 development suite that includes a Handel-C compiler. The practical advantage, when using this board, is the packaged communication library that is written in Handel-C. The communication library provides pre-built hardware design to gain access to the PCI bus and the on-board memories of the RC1000-PP. The block diagram for the RC1000-PP board is shown in Figure 6.2.

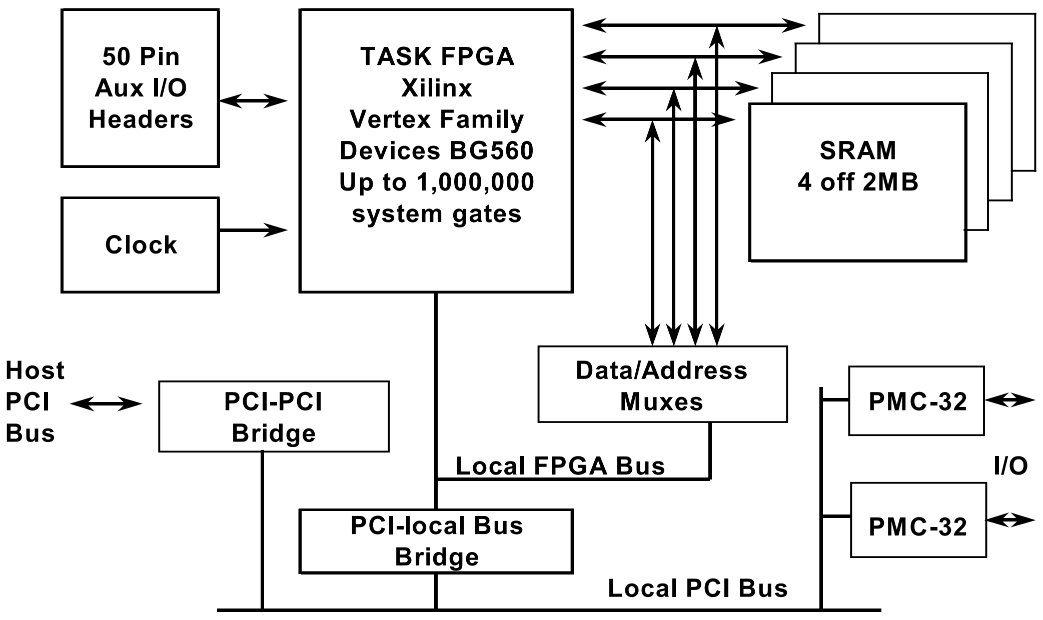

Figure 6.2: RC1000-PP Block Diagram

RC1000-PP board includes a PCI bridge, a clock generator, 8Mbytes SDRAM and an XCV1000 FPGA chip. The board is designed to allow single byte transfers to and from the FPGA chip through a dedicated address port in the PCI bus. Multi-byte transfers are possible only by redirecting the data using direct memory access (DMA) transfer to the external memory, before being read by the FPGA chip. Theoretically, the XCV1000 itself is capable of running at clock speeds of up to 200MHz, but the memory controller restricts the clock speed down to a maximum of 33 MHz when using the on-board RAM. A single XCV1000 chip contains 1 Mbyte of internal distributed RAM and 6144 CLBs (configurable logic blocks) that roughly amounts to 1.5 million system gates. Each CLB in the Virtex series are divided into 2 programmable slices that is roughly around 127 system gates each. Therefore, we can program 12,288 individual slices in the XCV1000 chip.

Chapter 7

Two Implementations of WalkSAT

In Chapter 5, we present the structure of our general non-re-synthesis clause evaluator. Based on this kind of clause evaluator, and the techniques discussed in Section 4.1 and 4.2 for improving the parallelism of SAT solving systems, we hope to achieve a high performance SAT solver without requiring re-synthesis. As we have described in Chapter 5, the non-re-synthesis reconfigurable clause evaluator requires *O(mn)* CLBs for an implementation with *m* clauses and *n* variables. This component consumes a significant fraction of the available CLBs (as much as 80%). As we would like to be able to handle as large a problem as feasible within the constraints of the FPGA, it is impractical to consider implementations that require multiple clause evaluators. This would consume too much of the chip real estate, even though there is considerable parallelism gain.

Within the constraints of our hardware platform, only one non-re-synthesis clause evaluator is available. For GSAT, we find that the procedure shown in Figure 4.1 is truly impractical due to the large size increase to a factor of n. As for the general GSAT procedure shown in Figure 2.2, although we can design a general hardware system which containing only one non-re-synthesis clause evaluator, the performance of such a system isn't expected to be good since the scoring procedure is to be implemented sequentially.

In contrast, when considering WalkSAT implementations for 3-SAT problems, it is of a different nature. As we have mentioned in Section 2.2.2, based on the algorithm given in Figure 2.3, by setting noise parameter *p* to 100% (Random-strategy) and 0% (Greedy-strategy) respectively, we can get two WalkSAT variants, the so-called Random-strategy and Greedy-strategy implementations. Since the computing complexities are different for the two variants, we adopted different hardware techniques when designing systems with our current FPGA.

For this chapter, Section 7.1 describes Random-strategy-based WalkSAT implementation; Section 7.2 describes Greedy-strategy-based WalkSAT implementation. The two implementations represent different tradeoffs in using a single reconfigurable clause evaluator.

7.1 Pipelined Random-strategy-based FPGA Implementation

Implementing algorithmic parallelism or pipelining is a frequently used technique in hardware design that reduces the number of clock cycles needed to perform complex operations.

Given that we are constrained to a single clause evaluator, we are left with pipelining as the only option for increasing the flip rate. For maximal reuse of the clause evaluator, it is important that the pipeline be well balanced with simple pipeline stages. Given that we already have a fully parallel clause evaluator, the most expensive step in WalkSAT is variable selection. A particularly simple WalkSAT variant is to set *p* to 100% in algorithm shown in Figure 2.3, thus this variant it to choose the variable randomly in a selected unsatisfied clause. This strategy is also used in the WalkSAT implementation of Leong *et al.* [LSW01].

7.1.1 Five-stage Pipelined Random Implementation

The program shown in Figure 7.1 gives the complete WalkSAT hardware design of Random-strategy. The main computation is divided into five stages, labeled *s1* to *s5*. In stage *s1*, function *Clause_Selector* selects one unsatisfied clause from *Cp* using uniform distribution; *Cp* is a sequence of 1s and 0s, 1 means the respondent clause is satisfied and 0 means the respondent clause is unsatisfied. The index number of the selected clause is stored in *Clause_Number.* Since the system is designed for SAT problems of up to 220 clauses, *Clause_Number* is designed as 8-bit wide. For the next stage *s2*, three parallel *Variable_Indexes_Generator*s generate three index numbers for the three variables occurring in the selected unsatisfied clause. It is assumed that there are three variables per clause, therefore, we have *Variable_0*, *Variable_1*, and *Variable_2* to store the indices of the three variables respectively. Since the system can solve SAT problems of up to 100 variables, *Variable_0*, Variable_1, and *Variable_2* are designed as 7-bit wide. In the next stage *s3*, *Pseudo_Random_Number_Generator* generates a pseudo random number among 0, 1 and 2. With the generated random number *PRN*, function *SelectFlip* flips one variable upon variable assignment *Vp*, thus generates a new variable assignment *Vars*. In stage *s4* , *Clause_Checker* evaluates if all the clauses are satisfied with variable assignment *Vars*. The final stage *s5* transfers *Vars* to *Vp*. This stage also determines whether this iteration has produced a satisfying solution; all the clauses are satisfied when *Sum(Cp)* is equal to *m.* (*m* is the total number of clauses.)

```
     main():
         Vp:= Receive_Initial_Assignment() ;
         Cp:= Clause_Checker(Vp);
         if Sum(Cp) = $m$ then break;
         for $i$ :=1 to $maxflips$ do
s1:          Clause_Number := Clause_Selector(Cp);

s2:          par {
                 Variable_0:=Variable_Indexes_Generator(Clause_Number,0);
                 Variable_1:=Variable_Indexes_Generator(Clause_Number,1);
                 Variable_2:=Variable_Indexes_Generator(Clause_Number,2);
             }

s3:          PRN := Pseudo_Random_Number_Generator(0, 1, 2);
             Vars := SelectFlip(Vp, Variable_0, Variable_1, Variable_2, PRN);

s4:          Cp := Clause_Checker(Vars);

s5:          if Sum(Cp) = $m$ then break;
             Vp := Vars;
         end
         Send_Assignment(Vp)
```

Figure 7.1: Random-strategy-based Implementation of the WalkSAT-B Variant

Figure 7.2 depicts the five-stage Random-strategy-based pipelined implementation. Stage 1 finds a random unsatisfied clause (this checks all clauses in parallel). Stage 2 generates three variable indices for the selected clause. Stage 3 implements the random selection heuristic and flips the selected variable in the selected unsatisfied clause. Stage 5 checks for satisfiability. There are a number of on-board storage buffers used. Buffer 1 stores the clause table which gives the mapping of clause to variables used within that clause as represented by variable indices. The SAT problem is initially loaded into buffer 2 which is then used to initialize the $f_{i,j}$ blocks in the clause evaluator (See Section 5.2.1). The result of this implementation is a one flip per clock cycle implementation.

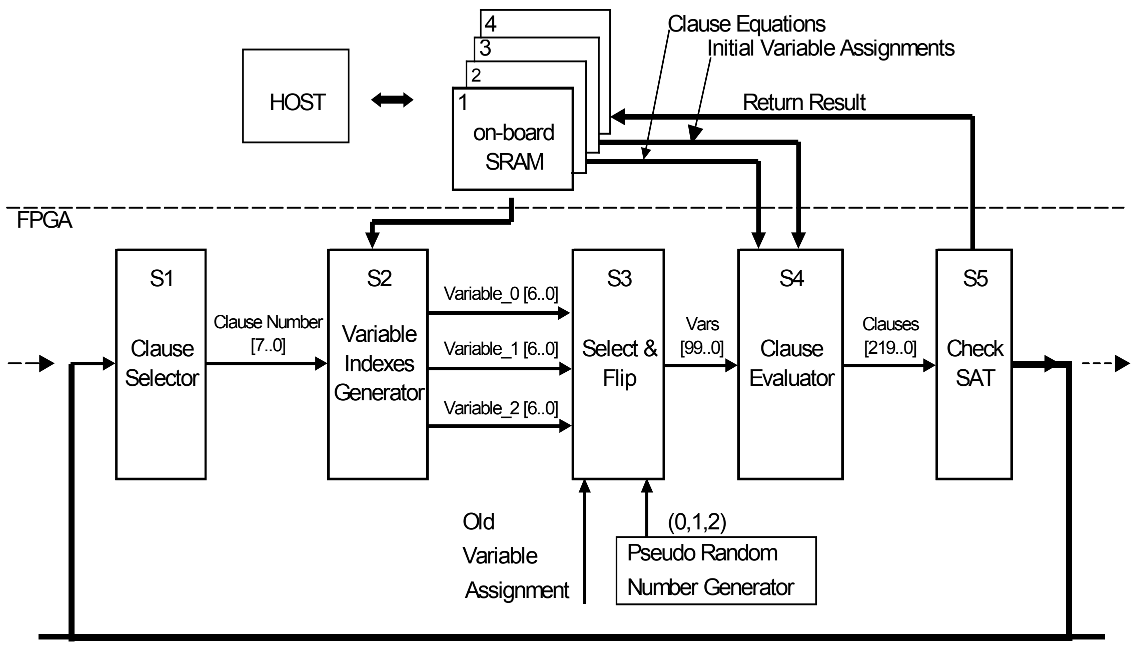

Figure 7.2: Pipelined Random-Strategy WalkSAT

7.1.2 A Pseudo Random Number Generator

As we can see from the previous section, there is a Pseudo Random Number Generator (PRNG) in the stage 3 of our pipelined Random-strategy-based WalkSAT implementation. In the past, the random number generation was mostly done by software. However, as digital systems become faster and denser, it is feasible, and frequently necessary, to implement the generator directly in hardware. Although the software-based method are well understood [James90, Knuth81, LE88, Mar85], they frequently require complex arithmetic operations and thus are not feasible to be constructed in hardware.

Ideally, the generated random numbers should be uncorrelated and satisfy any statistical test for randomness. A generator can be either “truly random” or “pseudo random". The former exhibit true randomness and the value of next number is unpredictable. The later only appears to be random. The sequence is actually based on specific mathematical algorithms and thus the pattern is repetitive and predictable. However, if the cycle period is very large, the sequence appears to be non-repetitive and random.

True randomness can be derived from certain physical phenomena, such as the time between tics from a Geiger counter exposed to radioactive materials. In electronic circuit, thermal noise is frequently used as the source of randomness because of its well-qualified spectral and statistical properties. A representative implementation [Quan98] is shown in Figure 7.3. In this circuit, the source of the noise is the thermal noise of a precision resistor, which is represented as V_{noise}. It is amplified by a low-noise amplifier and then passed to a high-speed comparator. The threshold of the comparator (V_{ref}) correspond to the mean voltage of the input noise signal. The output of the comparator is sampled and latched to a register. The latched signal is a one-bit binary signal that exhibits true randomness.

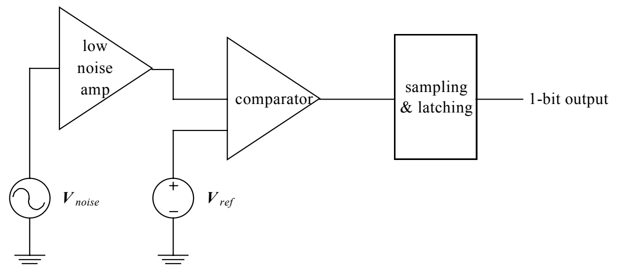

Figure 7.3: A True 1-bit Random Number Generator

The true random number generator is fairly involved since it needs to preserve and amplify the thermal noise, and at the same time shield all the external disturbances. It consists of mainly analog components and cannot be implemented by pure digital circuitry. The mixed-signal implementation significantly increases the system complexity. This implementation is also relatively slow and cannot match the high-speed digital circuit.

There are many methods to generate pseudo random numbers. As our SAT accelerating hardware are FPGA based, it is more desirable to have FPGA PRNG (Pseudo Random Number Generator). Nova Engineering Inc. developed a so-called Linear Feedback Shift Register (LFSR) Megafunction [Nova96], this function is designed for application in digital signal processing (DSP) and wireless telecommunication systems). P. Chu et al [CJ99] described techniques suitable for hardware implementations. In our implementation, we designed our LFSR PRNG by adopting the LFSR Megafunction from Nova Engineering, Inc. [Nova96] and combining the techniques described in [CJ99]. Figure 7.4 shows the block diagram of our 8-bit LFSR PRNG. Here, the pattern of the random number sequence repeats itself after 256 numbers. This period is sufficient for our implementation.

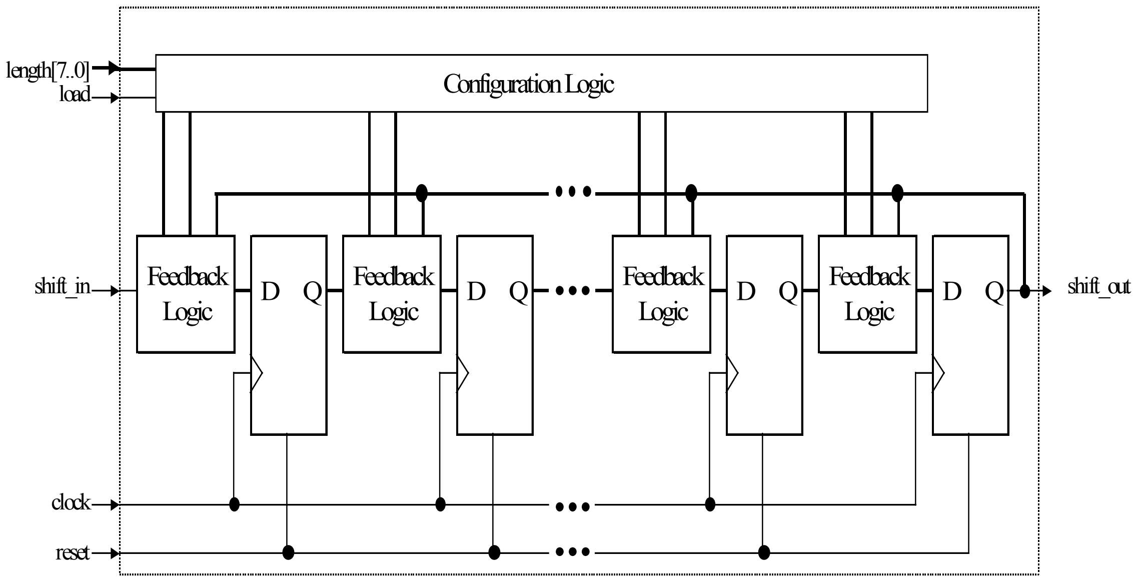

Figure 7.4: 8-bit LFSR PRNG Block Diagram

7.2 FPGA Implementation of Greedy Selection

A more typical WalkSAT variable selection heuristic is to select the variable, which best improves the score. In terms of the constraints of the hardware, this corresponds to a design with more complex operations. We have chosen to use a pure greedy heuristic without noise, a so-called Greedy-strategy-based implementation (but a noise component can be easily added).

Figure 7.5 shows the block diagram of our sequential Greedy-strategy-based implementation. Since we are dealing with 3-SAT, it is only necessary to determine at most which of the three variables in the selected unsatisfied clause to flip. However, any kind of parallel implementation (duplication or pipelining) of this step would require evaluating the score of each of the three possibilities. This would require three clause evaluator units in order to keep the resource independency, which we deem too much space consuming for the targeted SAT problem size.

Thus, we are restricted to a sequential implementation for the variable selection (Stages 4-6), which reduces the flip rate. Our current implementation performs one flip in nine cycles, as opposed to one cycle achieved by the design for random selection heuristic described in the previous section.

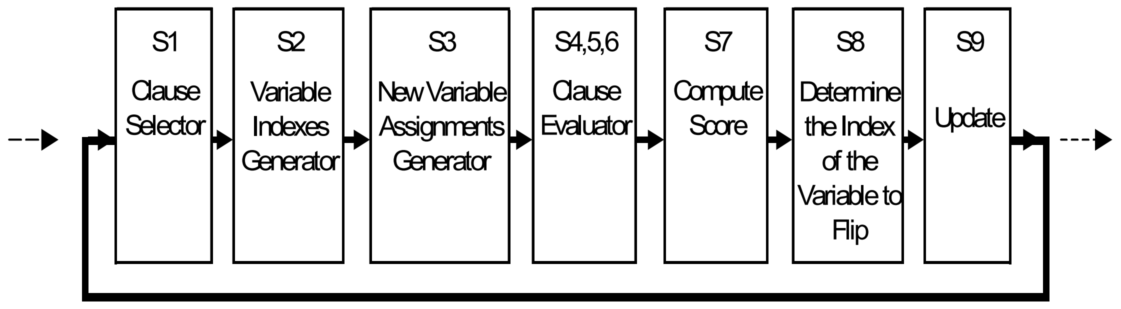
S1
Clause Selector
S2
Variable Indexes Generator
S3
New Variable Assignments Generator
S4,5,6
Clause Evaluator
S7
Compute Score
S8
Determine the Index of the Variable to Flip
S9
Update

Figure 7.5: Sequential Greedy-Strategy WalkSAT

Chapter 8

Experimental Results

The preceding chapters have described our general clause evaluator without requiring re-synthesis and developed two WalkSAT implementations which adopted different strategies based on this same clause evaluator. This chapter, we investigate on the experimental results gathered by running a set of benchmarks on the two implementations. Our prototype implementations investigate the two designs on two SAT problem sizes; a 50-variable/170-clause format and a 100-variable/220-clause format, the latter chosen to such that its reconfigurable clause evaluator fits on the FPGA which we used. Section 8.1 introduces the benchmarks. Section 8.2 describes the comparison schemes. Section 8.3 compares the flip rate performance between software implementation and hardware implementation. Section 8.4 compares the timing performance between software implementation and hardware implementation. Section 8.5 discusses the time/space cost about designs of different sizes.

8.1 Benchmark Selection

Our benchmark set is just a sub-set of a more comprehensive set compiled by Hoos and Stützle in [HS00], which includes the AIM problems and the uniform random 3-SAT problems.

AIM problems were produced from a Random-3-SAT generator, which was created by Asahio, Iwama, and Miyano [AIM]. These instances are particularly special due to its wide range of satisfiable and unsatisfiable instances. These instances ranges from m = 1.6 n to 6.0 n for both satisfiable and unsatisfiable problems. For our benchmark set (see Table 8.1) we include satisfiable problems ('aim-{*n*}-{*ratio*}-yes1-1') with n = 50 and 100; *m/n ratio* = 1.6, 2.0, and 3.4.

Problem Set	Representative	Variables	Clauses
Uniform Random-3-SAT	uf20-09	20	91
	uf20-031	20	91
	uf20-037	20	91
	uf50-01	50	218
	uf50-010	50	218
	uf50-0100	50	218
	uf50-01000	50	218
AIM Random-3-SAT	aim-50-1_6-yes1-1	50	80
	aim-50-1_6-yes1-2	50	80
	aim-50-1_6-yes1-3	50	80
	aim-50-1-6-yes1-4	50	80
	aim-50-2_0-yes1-1	50	100
	aim-50-2_0-yes1-2	50	100
	aim-50-2_0-yes1-3	50	100
	aim-50-2_0-yes1-4	50	100
	aim-50-3_4-yes1-1	50	170
	aim-50-3_4-yes1-2	50	170
	aim-50-3_4-yes1-3	50	170
	aim-50-3_4-yes1-4	50	170
	aim-100-1_6-yes1-1	100	160
	aim-100-1_6-yes1-2	100	160
	aim-100-1_6-yes1-3	100	160
	aim-100-1-6-yes1-4	100	160
	aim-100-2_0-yes1-1	100	200
	aim-100-2_0-yes1-2	100	200
	aim-100-2_0-yes1-3	100	200
	aim-100-2_0-yes1-4	100	200

Table 8.1: The Benchmark Set

The Uniform Random-3-SAT distributions are instances wherein the point of satisfiability is found at the phase transition region (i.e., 4.26 clauses per variable) [KS94, MSL92]. At this region, the average instance hardness for both systematic and stochastic local search algorithms is maximal [Yok97, CKT91, DLL62]. In Table 8.1, we included the problems ('uf{*n*}-01') with *n* = 20 and 50 to our benchmark set.

The benchmarks shown in Table 8.1 are simply those, which fit within the required problem sizes. As the main purpose of the benchmarks is to measure flip rate performance, the difficulty of the benchmark is not relevant.

8.2 Performance Comparison Scheme

To properly compare the performance of different WalkSAT implementations, we must first consider their underlying nature. In the current scenario, there are two types of implementation of WalkSAT algorithms, namely: pure-software and hardware accelerated implementation. The hardware accelerated implementations for the WalkSAT variants are generally patterned after the pure software implementation except for some trivial compiler functions. In this sense, algorithm specific factors like downward and sideway moves would generally be the same for both implementations. Performance factors used in empirical studies for WalkSAT that deals with probability of solubility are also not important. Our intension for adding the hardware accelerator is basically to improve the performance by doing computationally intensive task in hardware and reduce run-time. The major performance factor would be the "*flip rate*" performance that is measured by dividing the total number of flips by the total time. The resulting performance value would have the unit of flips-per-second or *fps*.

On the other hand, we also compare the timing performance between the hardware accelerated implementations and the pure software implementations. This is to verify the "*flip rate*" performance further.

8.3 Flip Rate Performance Comparison: Software vs. Hardware

In this section, we establish the basis of comparison by gathering performance results from the software implementation of WalkSAT algorithm and the FPGA-based implementations.

We utilized WalkSAT35 (WalkSAT version 35) by Bart Selman as the software implementation. The Linux workstation that we used to run the programs has an Intel Pentium 4 1500MHz processor and 899644 Kbytes of SDRAM.

As for the FPGA-based SAT implementation, they are from our pipelined Random-strategy-based WalkSAT implementation and our sequential Greedy-strategy-based WalkSAT. Our FPGA chip is Virtex 1000 and the work frequency of FPGA is 20Mhz.

In Table 8.2, there are two sets of flip rate performance comparison. One for the Greedy-strategy-based WalkSAT, the other is for the Random-strategy-based WalkSAT. For the Greedy-strategy-based implementation, the second column shows the flip rate performance of the pure software implementation, the third column shows the flip rate performance of the FPGA-based hardware implementation, the fourth column shows the speedup of column three versus column two. As for the Random-strategy-based implementation, the fifth column shows the flip rate performance of the pure software implementation, the sixth column shows the flip rate performance of the

FPGA-based hardware implementation, the seventh column shows the speedup of column six versus column five.

The resulting *flip rate, fps,* shows the program execution speed by dividing the number of flips performed with the execution time in seconds. The *flip rate* in Table 8.2 are shown in Kfps which is in thousands flip per second.

SAT Problem	Greedy-Strategy			Random-Strategy		
	Software *Kfps*	Hardware *Kfps*	Speed Up	Software *Kfps*	Hardware *Kfps*	Speed Up
uf20-09	265.8	2227	8.38	407.6	20345	49.91
uf20-031	251.8	2225	8.84	390.9	20350	52.06
uf20-037	303.2	2225	7.34	405.8	20335	50.11
uf50-01	409.4	2230	5.45	536.2	20347	37.95
uf50-010	459.6	2224	4.84	466.2	20371	43.70
uf50-0100	474.3	2237	4.72	598.3	20468	34.21
uf50-01000	476.9	2238	4.69	570.7	20367	35.69
aim-50-1_6-yes1-1	832.6	2208	2.65	947.9	20444	21.57
aim-50-1_6-yes1-2	814.1	2224	2.73	994.8	20521	20.63
aim-50-1_6-yes1-3	793.7	2227	2.81	960.1	20483	21.33
aim-50-1_6-yes1-4	868.1	2226	2.56	956.7	20461	21.39
aim-50-2_0-yes1-1	775.4	2227	2.87	865.4	20332	23.49
aim-50-2_0-yes1-2	775.3	2238	2.89	859.8	20407	23.73
aim-50-2_0-yes1-3	805.1	2224	2.76	877.7	20355	23.19
aim-50-2_0-yes1-4	783.2	2237	2.86	870.4	20433	23.48
aim-50-3_4-yes1-1	609.2	2227	3.66	618.6	20635	33.36
aim-50-3_4-yes1-2	596	2228	3.74	612.6	20557	33.56
aim-50-3_4-yes1-3	572.4	2228	3.89	613.1	20570	33.55
aim-50-3_4-yes1-4	561.2	2227	3.97	609.3	20613	33.83
aim-100-1_6-yes1-1	814.2	2228	2.74	962.5	20569	21.37
aim-100-1_6-yes1-2	787.4	2253	2.86	968.4	20427	21.09
aim-100-1_6-yes1-3	805.4	2222	2.76	972.9	20480	21.05
aim-100-1_6-yes1-4	872.2	2225	2.55	1014.4	20579	20.29
aim-100-2_0-yes1-1	744.9	2233	3.00	838.5	20184	24.07
aim-100-2_0-yes1-2	747.7	2245	3.00	814.3	20289	24.92
aim-100-2_0-yes1-3	731.4	2236	3.06	812.6	20262	24.93
aim-100-2_0-yes1-4	744.6	2233	3.00	834.4	20237	24.25

Table 8.2. Flip Rate Performance Comparison: Software versus FPGA-based

Analysis of the results from the UF and AIM problems suggest that the performance of the pure software WalkSAT algorithm is directly proportional to the variable/clause ratio of the SAT problem. The resulting graph in Figure 8.1 clearly shows the dominance in raw flip rate of the Random-strategy-based WalkSAT variant over Greedy-strategy-based WalkSAT variant. This is an obvious case since the Random-strategy does less computation, which results in shorter execution time.

Since our FPGA chip is clocked at 20 MHz, thus the flip rate for the random and greedy variable selection heuristics is constant throughout the problems – 20M for random, and 2.2M for greedy heuristics, due to its 9-stage implementation for each iteration inside the inner loop. In Table 8.2, the measured actual flip rate is shown in column 3 and column 6 respectively for Random-strategy and Greedy-strategy implementations.

Compared to the WalkSAT reconfigurable FPGA implementation from Leong *et al.* [LSW01], which also uses a random strategy, our Random-strategy-based implementation is much better than theirs. Their implementation uses a smaller FPGA and can only solve problems of up to 50-variable/170-clause and hence could be clocked at a faster speed of 33 Mhz. A major difference between their implementation and our pipelined implementation is that we use a constant flip rate. Their implementation, on the other hand, has a variable flip rate, because the use of sequential clause selection, and is bounded by maximum flip rate of 364Kfps.

With the random variable selection heuristic, the preliminary results show that our reconfigurable FPGA implementation is faster than software and previous hardware implementations. This implementation achieves one flip per clock cycle at 20Mhz. The greedy variable selection implementation has more modest speedups. The speedup is likely comparable to software or slightly faster, if the fastest state of art microprocessors are used, since performance scales at a lower rate with clock speed for microprocessors. However, the reduced flip rate may be offset by the increased effectiveness of the variable selection strategy. The greedy heuristic typically gives a better success rate than a random heuristic for WalkSAT. A detailed analysis of the effect of different variable selection heuristics is given in [HS00].

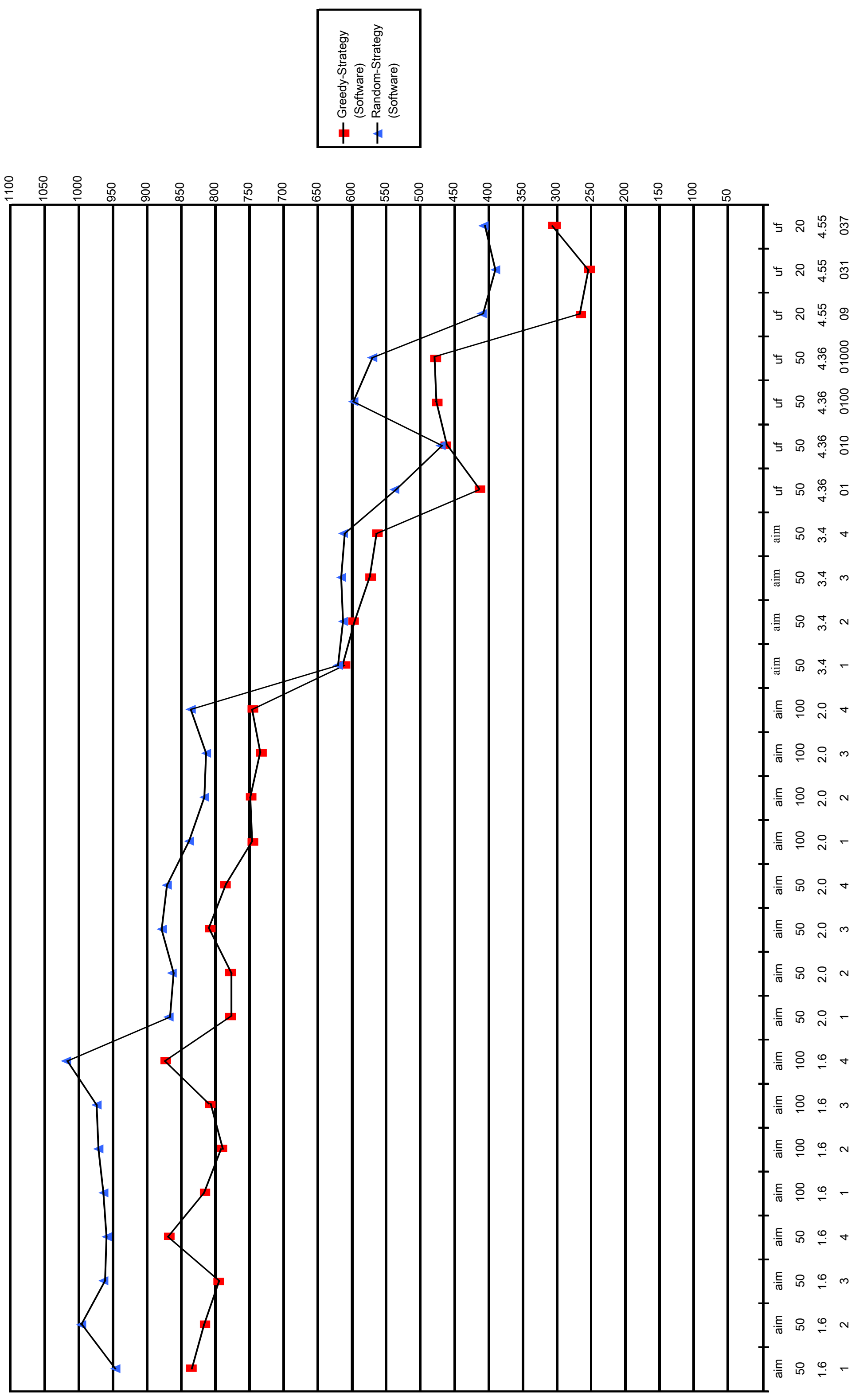

Figure 8.1: Pure Software Flip Rate Performance Chart

8.4 Timing Performance

In the previous section, we compared FPGA-based hardware implementations versus software for various 3-SAT benchmarks. In this section, we will compare the run-time timing performance.

WalkSAT algorithm, like all SLS algorithms, strongly involves random decisions such as the choice of the initial assignment, random tie-breaking, or biased random moves. Due to this inherent randomness, given a specific soluble problem instance, the time needed by a WalkSAT algorithm to find a solution varies from run to run. Consequently, the most detailed characterization of such an algorithm's behavior is given by its run-time distribution (RTD), which for a given instance maps the run-time *t* to the probability of finding a solution within time *t* [HS00].

To measure RTDs, one has to take into account that most SLS algorithms have some cutoff parameter bounding their run-time, like the *maxtries* and *maxflips* parameters in the generic algorithm schema of the procedure shown in Figure 2.1.

As argued in [HS98], this RTD will generally suggest the existence of an optimal setting of the maxSteps (Maxtries) parameter. Using this setting will indeed maximize the algorithm's performance – but only in the sense that within a given time period, the number of problem instances randomly drawn from the instance set (or distribution) which are solved within this time period will be maximal. However, as an RTD-based analysis shows, in this case the "optimal parameter setting" will not affect the performance on any individual instance from the set – it will only make sure that not too much is wasted trying to solve hard instances. Thus, using the optimal setting will effectively introduce a bias for solving easier problems – an effect which, except for very special application situations, will most likely be undesirable and can potentially give rise to erroneous interpretations of the observed behavior. This problem becomes very relevant when the inter-instance difficulty within the test-set has a high variance, as is the case for Random-3-SAT.

In our work, our objective is not to find an optimal setting for each SAT problem, instead, we map the software algorithm into hardware and see the speedup by using hardware. What is important is the consistency between the software implementation and the hardware implementation. In order to provide a consistency comparison between the hardware and the software, we compare the timing performance in this section.

Table 8.3 Compares the run-time timing performance between our FPGA-based pipelined random-strategy-based WalkSAT implementation and the software random-strategy-based WalkSAT implementation. Table 8.4 gives the timing performance comparison between the software implementation and FPGA-based hardware implementation based on greedy strategy.

Both our two kinds of FPGA-based hardware implementations are clocked at 20Mhz, and the overhead work time is the same. As host we use a PC with an AMD Athlon 1.2GHz CPU. Our prototype generates the clause configuration for a new SAT instance in software in about 7ms (This is unoptimized and is probably dominated by file I/O and hence could possibly be faster). Transferring the clause configuration from the host PC to the on-board SRAM takes 0.6ms. The FPGA takes 220 x 16 clock cycles to read the SRAM. With an FPGA clock frequency of 20MHz, this corresponds to 0.176ms. Thus the configuration overhead for solving a new SAT instance is 7.776ms. In contrast, the time to download a new bitstream to the FPGA is around 0.14s.

As for the pure software WalkSAT implementations, for all problems, the software execution time was measured on an Intel Pentium 4 1500MHz CPU.

Additionally, both of the software and the hardware implementations set *maxtries* to 256000 and set *maxflips* to 4000. Table 8.3 and Table 8.4 show the average software and hardware execution times computed over 100 trials respectively for random and greedy implementations.

SAT Problems	Hardware			Software			Speed Up
	Time (s)	Avg-Tries	Suc %	Time (s)	Avg-Ttries	Suc %	
uf20-09	0.0004	1	100	0.0039	1	100	9.75
uf20-031	0.0004	1	100	0.0042	1	100	10.5
uf20-037	0.0005	1	100	0.0071	1	100	14.2
uf50-01	0.0028	14	100	0.07834	10.5	100	27.98
uf50-010	0.0012	5	100	0.04818	5.6	100	40.15
uf50-0100	0.0188	91	100	0.51749	77	100	27.53
uf50-01000	0.0049	25	100	0.13001	19	100	26.53
aim-50-1_6-yes1-1	1.6880	7763	100	39.6707	9401	100	23.50
aim-50-1_6-yes1-2	4.1922	19262	100	81.4133	20247	100	19.42
aim-50-1_6-yes1-3	0.6535	3009	100	21.6208	5190	100	33.08
aim-50-1_6-yes1-4	1.3824	6355	100	26.2875	6287	100	19.02
aim-50-2_0-yes1-1	0.0196	97	100	1.8083	391	100	92.26
aim-50-2_0-yes1-2	0.0580	272	100	1.3615	293	100	23.47
aim-50-2_0-yes1-3	0.0386	185	100	1.6067	353	100	41.62
aim-50-2_0-yes1-4	0.0771	359	100	3.0008	653	100	38.92
aim-50-3_4-yes1-1	0.0205	107	100	0.9381	145	100	45.76
aim-50-3_4-yes1-2	0.0264	127	100	0.7892	121	100	29.89
aim-50-3_4-yes1-3	0.0308	146	100	0.7958	122	100	25.84
aim-50-3_4-yes1-4	0.0128	65	100	0.4849	74	100	37.88
aim-100-1_6-yes1-1	56.8889	256000	timeout	1064.34	256000	timeout	18.71
aim-100-1_6-yes1-2	57.5281	256000	timeout	1057.00	256000	timeout	18.37
aim-100-1_6-yes1-3	57.7540	256000	timeout	1058.52	256000	timeout	18.33
aim-100-1_6-yes1-4	57.7984	256000	timeout	1029.99	256000	timeout	17.82
aim-100-2_0-yes1-1	56.7512	256000	timeout	1238.59	256000	timeout	21.82
aim-100-2_0-yes1-2	35.2324	174364	3	1270.55	256000	timeout	-
aim-100-2_0-yes1-3	31.1952	154384	3	1269.93	256000	timeout	-
aim-100-2_0-yes1-4	56.8618	256000	timeout	1240.75	256000	timeout	21.82

Table 8.3: Timing Performance Comparison based on Random Strategy

As we can see from Table 8.3 and Table 8.4, the Average Tries (Avg-Tries) and the Success Rate (Suc) 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 WalkSAT algorithm. As can be seen from the "Speed Up" column, the hardware performance for all problems is approximately two to seven times faster than the software for the greedy-strategy-based implementation; and it is almost ten to ninety times faster for the random-strategy-based implementations.

SAT Problems	Hardware			Software			Speed Up
	Time (s)	Avg-Tries	Suc %	Time (s)	Avg-Tries	Suc %	
uf20-09	0.0048	2.6	100	0.0317	2.1	100	6.67
uf20-031	0.0080	4.4	100	0.0567	3.6	100	7.07
uf20-037	0.0029	1.6	100	0.0202	1.5	100	6.83
uf50-01	0.0255	13	100	0.0634	6.5	100	2.48
uf50-010	0.0067	3.4	100	0.0225	2.6	100	3.34
uf50-0100	0.0763	39	100	0.1017	12	100	1.33
uf50-01000	0.0319	15	100	0.0859	10	100	2.69
aim-50-1_6-yes1-1	0.3775	194	100	0.8034	167	100	2.13
aim-50-1_6-yes1-2	1.1509	594	100	2.5934	528	100	2.25
aim-50-1_6-yes1-3	0.2966	153	100	0.4959	98	100	1.67
aim-50-1_6-yes1-4	0.2549	131	100	0.4417	96	100	1.73
aim-50-2_0-yes1-1	0.5868	326	100	2.3624	458	100	4.03
aim-50-2_0-yes1-2	0.4220	217	100	0.8384	162	100	1.99
aim-50-2_0-yes1-3	0.2525	130	100	0.4975	100	100	1.97
aim-50-2_0-yes1-4	0.7510	374	100	1.3009	255	100	1.73
aim-50-3_4-yes1-1	0.0310	15	100	0.0369	5.6	100	1.19
aim-50-3_4-yes1-2	0.0080	4.4	100	0.0513	7.6	100	6.42
aim-50-3_4-yes1-3	0.0230	11	100	0.0646	9.2	100	2.81
aim-50-3_4-yes1-4	0.0122	6.1	100	0.0251	3.5	100	2.05
aim-100-1_6-yes1-1	252.508	131425	43	587.648	119616	45	2.34
aim-100-1_6-yes1-2	182.280	91130	47	554.998	109251	50	3.04
aim-100-1_6-yes1-3	172.350	86166	63	596.606	120127	15	3.46
aim-100-1_6-yes1-4	490.354	256000	timeout	1174.09	256000	timeout	2.39
aim-100-2_0-yes1-1	42.64653	21321	100	131.487	24486	100	3.08
aim-100-2_0-yes1-2	31.04498	15520	100	89.6451	16757	100	2.89
aim-100-2_0-yes1-3	23.93	11928	100	75.8917	13877	100	3.17
aim-100-2_0-yes1-4	72.8159	36404	100	175.518	32673	100	2.41

Table 8.4: Timing Performance Comparison based on Greedy Strategy

SAT Problems	Leong et al. [LSW01]	Ours, Pipelined	Speedup
uf20-9	0.012	0.0004	30
uf20-31	0.009	0.0004	23
uf20-37	0.009	0.0005	23
aim-50-2_0-yes1-1	3.93	0.0196	201
aim-50-2_0-yes1-2	3.44	0.058	59
aim-50-3_4-yes1-1	2.29	0.0205	112
aim-50-3_4-yes1-2	0.98	0.0264	37
aim-50-3_4-yes1-3	2.12	0.0308	69
aim-50-3_4-yes1-4	3.93	0.0128	307

Table 8.5: Running Time Comparison between Random-Strategy Implementations

As can be seen in Table 8.5, when comparing the run-time timing performance between our pipelined random-strategy-based implementation and that from Leong *et al.* [LSW01], which also uses the random strategy, our random-strategy-based implementation is much better than theirs. Although the FPGA is clocked at 33MHz in their implementation, but as we have discussed in Section 8.3, the flip rate in their implementation is much lower than ours, and thus their timing performance is consequently much worse than ours.

8.5 Time/Space Cost Comparison of FPGA-based Implementation

In comparing hardware implementations for WalkSAT, an additional important factor is the size of resulting design. For ASIC implementations the design size is estimated in terms of system gates and for FPGA implementations the design size is based on the number of slice used. A single slice roughly amounts to 127 system gates or 2.25 logic cells.

Our prototype implementations investigate each of the random-strategy-based system and the greedy-strategy-based system on two SAT problem sizes; thus we get four circuits, two circuits are for a 50 variable/170 clause format and the other two are for a 100 variable/220 clause format, the latter format is chosen so that its reconfigurable clause evaluator fits on the FPGA which we used.

Table 8.6 gives the hardware costs in terms of slices for these four implementations. The minimum gate delay is as reported by the Xilinx place and route tools. These delays shown in column two and column five are just the worst delay inside the FPGA. In each system, there is board delay, for examples, along the signal from FPGA to SRAM, set up time of other devices and et al. The total delay is definitely larger than the FPGA delay alone. Experiments show that our systems can work stably when FPGA is clocked at 20 MHz. As we can see in Table 8.6, there is only a small difference in gate delay between the two implementations of the same size while their hardware cost are quite similar. The larger influence is the increased delay due to larger problem sizes. This because the circuits is synthesized to more levels of logic and the routing congestion becomes denser as the system becomes bigger. For a simplest example, there are 25 input signals at the OR gate inside each clause for the bigger system, whereas only 13 inputs for the smaller one.

	Random-Strategy WSAT		Greedy-Strategy WSAT	
System Size	Delay (ns)	Cost of Slices	Delay (ns)	Cost of Slices
50-var/170-c	24.097	4946 (40%)	24.842	6408 (52%)
100-var/220-c	31.005	10396 (85%)	31.639	11834 (96%)

Table 8.6: Time/Space Cost Comparison of FPGA-based Implementation

Chapter 9

Conclusions

This thesis has studied the effects of solving SAT problems using stochastic local search algorithms on an FPGA platform. The goal is to accelerate SAT solving using hardware in a practical way. This chapter summarizes the new techniques developed in the preceding chapters and then discusses the future work of this research.

We demonstrate two prototype hardware solvers implemented on the Xilinx Virtex XCV1000 FPGA with significantly better performance than software and previous hardware WalkSAT solvers. Furthermore, the solvers are reconfigurable in real-time, with a reconfiguration time of a few milliseconds for problems with 100 variables. Our two implementations illustrate the tradeoff between time, space, and effectiveness of the SLS algorithm. The random solver achieves an optimal flip rate at the cost of a simple variable selection strategy, while the greedy solver uses the more expensive and effective strategy but is not amenable to pipelining and is hence slower.

Both implementations are limited by the size of the Xilinx Virtex XCV1000 chip used, which can accommodate a reconfigurable clause checker only for problems with 100 variables and 220 clauses. This chip, dating from 1999, is fabricated using a 5-layer metal 0.22um CMOS process. In comparison, the current Virtex-II generation uses an 8-layer 0.15um CMOS process, this leading-edge process and the Virtex-II architecture are optimized for high speed with low power consumption. Combining a wide variety of flexible features and a large range of densities up to 10 million system gates, the Virtex-II family enhances programmable logic design capabilities and is a powerful alternative to mask-programmed gates arrays. The XC2V10000 has about 10 times more system gates than the XCV1000 and has significantly faster clock speeds. For example, a 100 variable/600 clause evaluator requires about 30K slices and fits in a XC2V6000 which has 6M system gates.

An FPGA implementation will have more limitations on problem sizes even when larger FPGAs are used. A fast hardware based solver can however still be useful for general SAT solving. One approach is with hybrid search and stochastic solvers. For example, Zhang et al. [ZHZ02] combine Davis Putnam with stochastic solvers. Their approach uses Davis Putnam to generate smaller sub-problems which are then solved with WalkSAT.

Another route to deal with larger problems is to use ASICs rather than FPGAs. Our implementation is not restricted to FPGAs since the reconfiguration for different SAT instances is not dependent on the reconfiguable logic of FPGAs. The prototype uses FPGAs simply because they are more cost effective for development. Given the real-time reconfiguration capability, this may be a promising candidate for direct ASIC implementation, which means higher clock speeds and much more resources for dealing with larger problems.

Appendix

Entity Declarations in VHDL

```
ENTITY clause_checker IS
    PORT (
        clk : IN STD_LOGIC;
        rst : IN STD_LOGIC;
        cc_sel : IN STD_LOGIC;
        addr : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        rwn : IN STD_LOGIC;
        wdata : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
        rdata : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
        cc_rdy : OUT STD_LOGIC;
        v : IN STD_LOGIC_VECTOR(99 downto 0);
        all_clauses : OUT STD_LOGIC_VECTOR(219 DOWNTO 0));
 END;


ENTITY cc_ctrl IS
    PORT (
        clk : IN STD_LOGIC;
        reset : IN STD_LOGIC;
        cc_sel : IN STD_LOGIC;
        addr : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        rwn : IN STD_LOGIC;
        wdata : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
        rdata : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
        cc_rdy : OUT STD_LOGIC;
        row_wen : OUT STD_LOGIC_VECTOR(219 DOWNTO 0);
        row_waddr : OUT STD_LOGIC_VECTOR(99 DOWNTO 0);
        row_wdata : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
        row_rdata : IN allrowrdata_type)
    );
 END;

ENTITY cc_clause IS
    PORT (
        row_wen : IN STD_LOGIC;
        row_wdata : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
        row_srdata : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
        row_wclk : IN STD_LOGIC;
        row_waddr : IN STD_LOGIC_VECTOR(99 DOWNTO 0);
        v : IN STD_LOGIC_VECTOR(99 DOWNTO 0);
        row_out : OUT STD_LOGIC
     );
 END;

ENTITY ram16x1d IS
    PORT (
        dpo : OUT STD_LOGIC;
        spo : OUT STD_LOGIC;
        a0  :  IN STD_LOGIC;
        a1  :  IN STD_LOGIC;
        a2  :  IN STD_LOGIC;
        a3  :  IN STD_LOGIC;
        d   :  IN STD_LOGIC;
        dpra0 : IN STD_LOGIC;
        dpra1 : IN STD_LOGIC;
        dpra2 : IN STD_LOGIC;
        dpra3 : IN STD_LOGIC;
        wclk   : IN STD_LOGIC;
        we     :  IN STD_LOGIC
    );
 END;
```

Bibliography

[AIM]	Y. Asahiro, K. Iwama, and E. Miyano, "Random Generation of Test Instances with Controlled Attributes".
[ASS99]	M. Abramovici, J. T. Sousa and D. Saab, "A Massively-Parallel Easily-Scalable Satisfiability Solver Using Reconfigurable Hardware," in Proceedings ACM/IEEE Design Automation Conference, pages 684-690, 1999.
[AS97]	M. Abramovici and D. Saab, "Satisfiablilty on Reconfigurable Hardware," International Workshop on Field Programmable Logic and Applications, 1997.
[AS00]	M. Abramovici and D. Saab, "A Sat Solver Using Reconfigurable Hardware and Virtual Logic," 2000.
[BFRV92]	S. D. Brown, R. Francis, J. Rose, and Z. Vranesic, "Field Programmable Gate Arrays," Kluwer Academic Publishers, Netherland, 1992.
[CJ99]	P. Chu and R. Jones, "Design Techniques of FPGA-Based Random Number Generator," Military and Aerospace Application of Programmable Devices and Technologies Conference, 1999.
[CKT91]	Peter Cheeseman, Bob Kanefsky, and William M. Taylor, "Where the Really Hard Problems Are," in Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia, pages 331-337, 1991.

[Coo71] Stephen A. Cook, "The Complexity of Theorem-Proving Procedures," 3rd Annual ACM Symposium on Theory of Computing, pages 151-158, 1971.

[DABC93] O. Dubois, P. Andre, Y. Boufkhad and J. Carlier, "Sat versus Unsat," in 2nd DIMACS Implementation Challenge, 1993.

[DIMACS] Dimacs Challenge Benchmarks. Online available: ftp://dimacs.Rutgers.edu/pub/challenge

[DLL62] M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem Proving," Communications of the ACM, 5(7):394-397, July 1962.

[DP60] M. Davis, H. Putnam, "A Computing Procedure for Quantification Theory," Journal of the ACM, 7(3), July 1960.

[GHK91] M. Gokhale, W. Holmes, A. Kopser, et al, "Building and Using a Highly Parallel Programmable Logic Array," IEEE Computer, 24(1):81-89, Jan. 1991.

[Goe81] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," IEEE Transactions on Computers, 30(3):215-222, 1981.

[Gu92] J. Gu, "Efficient Local Search for Very Large-Scale Satisifiability Problems," SIGART Bulletin, 3:8-12, 1992.

[GW93] Ian P. Gent and Toby Walsh, "Towards an Understanding of Hill-Climbing Procedures for SAT," in Proceedings of AAAI-93, pages 28-33, 1993.

[HM97] Youssef Hamadi and David Merceron, "Reconfigurable Architectures: A New Vision for Optimization Problems," in Gert Smolka, editor, Principles and Practice of Constraint Programming - CP97, Proceedings of the 3rd International Conference, Lecture Notes in Computer Science 1330, pages 209-221, Linz, Austria, 1997. Springer-Verlag, Berlin.

[Hoo96] Holger Hoos, "Aussagenlogische SAT-Verfahren und ihre Anwendung bei der Lösung des HC-Problems in Gerichteten Graphen," Diplomarbeit. Fachbereich Informatik, Technische Hochschule Darmstadt, Germany, March 1996.

[HS98] H. H. Hoos and T. Stützle, "Evaluating Las Vegas Algorithms – Pitfalls and Remedies," in Proceedings of UAI-98, pages 238-245, 1998.

[HS99] Holger H. Hoos and Thomas Stützle, "Systematic vs. Local Search for SAT," in Proceedings of the 23rd National German Conference of Artificial Intelligence (KI-99), 1999.

[HS00] Holger H. Hoos and Thomas Stützle, "Local Search Algorithms for SAT: An Empirical Evaluation," Journal of Automated Reasoning", 24(4):421-481, 2000.

[HTY01] Martin Henz, Edgar Tan and Roland Yap, "One Flip per Clock Cycle," in Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming, CP2001, Cyprus, Nov/Dec 2001.

[James90] F. James, "A Review of Pseudo-random Number Generators," Computer Physics Communications 60, 1990.

[Knuth81] D.E. Knuth, The Art of Computer Programming Vol. 2: Seminumerical Methods, (2nd edition), Addison-Wesley, reading, Mass., 1981.

[KS94] S. Kirkpatrick and B. Selman, "Critical Behavior in the Satisfiability of Random Boolean Expressions," Science, 264(5163):1297-1301, 27 1994.

[LSW01] P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and

M. P. Leong, "A Bitstream Reconfigurable FPGA Implementation of WSAT Algorithm," IEEE Transactions on Very Large Scale Integration(VLSI) Systems, 9(1), Feb. 2001.

[Mar85] G.A. Marsaglia, "A Current View of Random Number Generators," Computational Science and Statistics: The Interface, ed. L. Balliard, Elsevier, Amsterdam, 1985.

[MSG97] Bertrand Mazure, Lakhdar Sais and Eric Gregoire. "Tabu Search for SAT," in AAAI/IAAI, pages 281-285, 1997.

[MSK97] David McAllester, Bart Selman, and Henry Kautz, "Evidence for Invariants in Local Search," in Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97), 1997.

[MSL92] David G. Mitchell, Bart Selman and Hector J. Levesque, "Hard and Easy Distributions for SAT Problems," in Paul Rosenbloom and Peter Szolovits, editors, Proceedings of the Tenth National Conference on Artificial Intelligence, pages 459-465, Menlo Park, California, 1992. AAAI Press.

[Nova96] Nova Engineering Inc, "Linear Feedback Shift Register Megafunction," http://www.nova-eng.com, 1996.

[PK01] Donald J. Patterson and Henry Kautz, "Auto-Walksat: A Self-Tuning Implementation of Walksat," in Proceedings of SAT2001: Workshop on Theory and Application of Satisfiability Testing, 2001.

[Quan98] Quantum World Corporation, "QNG Model J20KP True Random Number Generator Users Manual," 1998.

[RLG98] C. R. Rupp, M. Landguth, T. Garverick, E. Gomerall, H. Holt, J. M. Arnold and M. Gokhale, "The NAPA Adaptive Processing Architecture," in Proceedings IEEE Symposium on FPGAs for Custom Computing Machines, pages 28-37, Apr. 1998.

[RS94] R. Razdan and M. D. Smith, “A High-Performance Microarchitecture with Hardware-Programmable Functional Units,” in Proceedings of the 27th Annual IEEE/ACM International Symposium on Micro Architecture, pages 172-180, 1994, Nov, 1994.

[Sha99] A.K. Sharma, “Programmable Logic Handbook,” MacGraw Hill, New York, 1999.

[SKC94] B. Selman, H. Kautz and B. Cohen, “Noise Strategies for Improving Local Search,” in Proceedings of AAAI-94, pages 337-343, 1994.

[SLM92] B. Selman, Hector Levesque, and David Mitchell, “A New Method for Solving Hard Satisfiability Problems,” in Proceedings of AAAI-92, pages 440-446, 1992.

[SSS97] O. Steinmann, A. Strohmaier and T. Stützle, “Tabu Search vs Random Walk,” in Advances in Artificial Intelligence (KI97), volume 1303 of LNAI, pages 337-348. Springer Verlag, 1997.

[SYS98] T. Sayama, M. Yokoo and H. Sawada, “Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware,” in proceedings 31st Hawaii International Conference of System Sciences, pages 179-186, 1998

[Tan02] Edgar Tan, “Local Search Algorithms for SAT on Field Programmable Gate Arrays,” MSc thesis, National University of Singapore, 2002.

[VBR96] J. E. Vuillemin, P. Bertin, D. Roncin, et al, “Programmable Active Memories: Reconfigurable Systems Come of Age,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 4(1):56-69, Mar. 1996.

[Xil00] Xilinx, Virtex 2.5V Field Programmable Gate Arrays (Data Sheet), 2000.

[Xil6200] Xilinx, XC6200 Programmable Gate Arrays (Data Sheet), 1997.

[Yok97] Makoto Yokoo, "Why Adding More Constraints Makes a Problem Easier for Hill-climbing Algorithms: Analyzing Landscapes of CSPs," in Principles and Practice of Constraint Programming, pages 356-370, 1997.

[YSLL99] Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee and Philip Heng Wai Leong, "A Runtime Reconfigurable Implementation of the GSAT algorithm," in Patrick Lysaght, James Irvine and Reiner W. Hartenstein, editors, Field-Programmable Logic and Applications, pages 526-531. Springer-Verlag, Berlin, 1999.

[YSS96] M. Yohoo, T. Sayama, and H. Sawada, "Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results," in Proceedings, 2nd International Conference of Principles Practice Constraint Programming, pages 497-509, 1996.

[ZAMM98] Peixin Zhong, Pranav Ashar, Sharad Malik and Margaret Martonosi, "Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability," in Design Automation Conference, pages 194-199, 1998.

[ZHZ02] Wenhui Zhang, Zhou Huang and Jian Zhang, "Parallel Execution of Stochastic Search Procedures on Reduced SAT Instances," PRICAI 2002, the 7th Pacific Rim International Conference on Artificial Intelligence, Tokyo, Japan, Aug, 18-22, 2002, Lecture Notes in Computer Science 2417:108-117, Springer-Verlag. 2002.

[ZMAM98a] Peixin Zhong, Margaret Martonosi, Pranav Ashar, Sharad Malik, "Accelerating Boolean Satisfiability with Configurable Hardware," in

Kenneth L. Pocek and Jeffrey Arnold, editors, IEEE Symposium on FPGAs for Custom Computing Machines, pages 186-195, Los Alamitos, CA, 1998, IEEE Computer Society Press.