An Architecture for solving boolean satisfiability using runtime configurable hardware

C. K. Chung and P. H. W. Leong
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

An architecture is proposed for a forward checking tree search which is used for solving satisfiability problems. In this design, the FPGA on-chip RAM feature is used to achieve a large improvement in density over a straight forward implementation using configurable logic blocks, enabling much larger problems to be solved. In addition, the boolean function to be satisfied is runtime configurable. A prototype implementation of the design operated successfully at 10MHz for a 50 variable, 80 clause 3-SAT problem.

1 Introduction

A conjunctive normal form (CNF) formula on m binary variables $x_1, x_2, ..., x_m$ is the conjunction (AND) of n *clauses* $C_1, ..., C_n$ each of which is the disjunction (OR) of one or more literals, where a literal is the occurrence of a variable or its complement. Such a formula denotes a n-variable Boolean function $f(x_1, ..., x_m)$. The boolean satisfiability problem (SAT) is concerned with finding a variable assignment so that $f(x_1, ..., x_m) = 1$ or proving that no solution exists.

The SAT problem appears in many fields, such as automatic test pattern generation, timing analysis, logic verification, etc. Algorithms for solving SAT can be complete or incomplete. Complete algorithms, by definition, can find all of the solutions to a problem and typically involve pruned tree searches. However, since SAT belongs to the class of NP-complete problems, in practice the long execution times involved make it difficult to find any solution.

The simplest method of solving SAT is a brute force search where all possible solutions are generated and tested. This method, of course, is grossly inefficient. A more efficient way is backtracking tree search [4], which performs a depth-first search of the space of the potential solutions. The performance of backtracking, although better than generate-and-test, is still poor. A better method is to employ forward checking, which will backtrack and change the values of the committed variables, if no possible solution can be found. This method can avoid searching subtrees which cannot have a solution and hence significantly reduces the search space.

This paper presents our application of field programmable custom computing machines to the acceleration of the forward checking tree search to SAT problems. In our approach, the indexes of the variables and the negation information can be specified at runtime so the very costly process of generating a bitstream tailored to a specified problem is avoided.

Fully parallel designs [5, 6], although extremely fast compared with serial software implementations cannot be used for large computational problems due to limitations on hardware resources. In our approach, we tradeoff the parallelism for increased clock rate and circuit density. Advanced search methods such as the Davis–Putnam method [6] can significantly reduce the search time, and although we use a simple forward checking search, other more sophisticated techniques can be used with the evaluation technique presented. The benefits of our architecture are that it uses FPGA RAM to achieve a 16× reduction in hardware and the design is runtime configurable for different SAT problems.

The rest of the paper is organized as follows. Section 2 describes the tree search algorithm. Section 3 describes the implementation. Results are presented in Section 4 and finally, conclusions are drawn in Section 5.

2 Algorithm

The algorithm we have implemented uses depth first search with forward checking. Figure 1 shows a tree representation of the search for a problem with 4 variables.

To illustrate the concept of forward checking, consider an example with 4 variables and 3 clauses. An array $x[p]$

0-7695-0353-5/99 $10.00 © 1999 IEEE

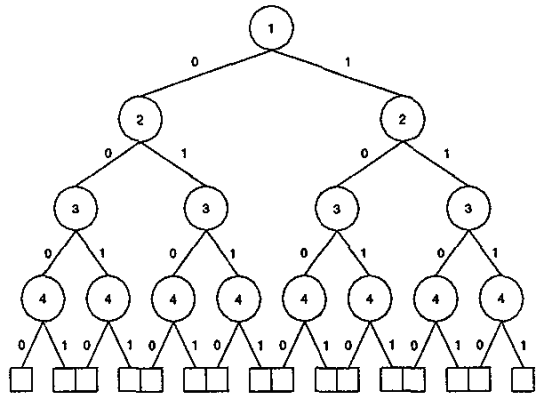

Figure 1: The tree representation of 4-variable SAT problem

is used to keep the current state of each variable. A *Global Pointer*, $GPointer$, is used to index into array x and a Global Counter, $GCounter$, iterates through the variables for the purpose of evaluation.

$$\begin{array}{lcl} \text{clause 1} & : & x_1 + x_2 + x_3 \\ \text{clause 2} & : & \overline{x_1} + x_2 + x_3 \\ \text{clause 3} & : & x_2 + x_3 + \overline{x_4} \end{array}$$

Initially, all variables are free and $GPointer$ is reset to 0. The system will fetch the previous assignment of variable 1 (initially free) and generate the new assignment '0' for it. Then, the value of the 4 variables will be fetched in 4 consecutive cycles and tested in the *Evaluator* (see Section 3) to determine if any constraints are violated. In this example, clause 2 is satisfied and the remaining clauses are undetermined. The next two steps assign '0' to variables 2 and 3. During the evaluation step, after the values of variables $x_1 - x_3$ have been fetched, clause 1 is determined but is '0'. Therefore, clause 1 is not satisfied and backtracking should be executed to search the next subtree, further searching being unnecessary.

The following pseudocode describes the search algorithm.

```
search()
{
    while (true)
    {
        if (GPointer = 0 AND backtrack)
            search completed;
        end if;
        out_val = previous assignment(GPointer);
        in_val = generate(out_val);
        GCounter = 0;
        do {
            out_val = fetch assignment(GCounter);
            backtrack = evaluate(out_val);
            GCounter = GCounter + 1;
        }
        while(GCounter < NO_VAR AND !backtrack);

        if (!backtrack)
            GPointer = GPointer + 1;
        else
            GPointer = GPointer - 1;
        end if;
        if (all clauses are satisfied)
            Solution found;
            exit the loop;
        end if;
    }
}
```

3 Implementation

3.1 Overview

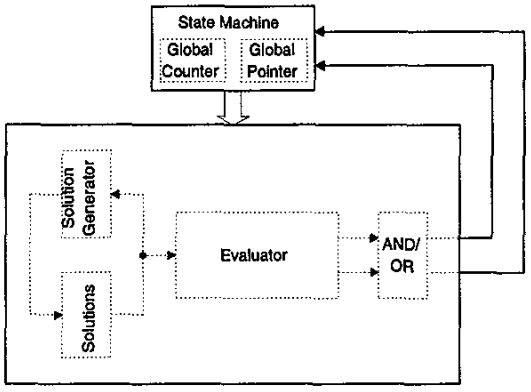

Figure 2: Block diagram of the search machine.

Figure 2 shows the block diagram of our architecture. The diagram consists of five modules, *Solution Generator*, *Solutions*, *Evaluator*, *AND/OR* and *State Machine*. The function of the *Solution Generator* module is to generate a new variable assignment to test from the previous assignment. The *Solutions* module stores the assignment of each variable. The *Evaluator* module is used to evaluate a new assignment to see if it violates constraints. The *AND/OR* module is used to determine whether or not a solution has been found as well as whether to backtrack. It receives outputs from the evaluator, $output_i$ and $back_i$. The *State Machine* controls the execution of the forward checking algorithm. It uses two global counters, namely *Global Counter* and *Global Pointer*. The *Global Pointer* is used for storing the indexes of current variable and the *Global Counter* is used to count the number of cycles required for an evaluation.

A DIMACS [2] 3-SAT benchmark problem (*aim-50-1_6-yes1-1*) with 50 variables and 80 clauses will be used as an example in this paper. The following sub-sections will describe each module based on this problem. Expanding our design to any sized problem is straightforward provided there are sufficient hardware resources.

3.2 Solutions

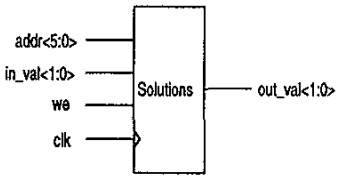

Figure 3: Block diagram of the *Solutions* module.

Figure 3 shows the block diagram of the *Solutions* module. It is built using the distributed RAM [3] feature of Xilinx 4000 series FPGAs. The assignment of each variable consists of two bits, $b_1 b_0$. The bit b_0 indicates whether the variable is free or assigned. The bit b_1 stores the assigned value. If the variable is free, the value of b_1 should be '0'. For 50 variables each 2-bits in size, 4 Xilinx 32 × 1 RAMs are used.

3.3 Solution Generator

To determine the current index of the variable, a *Global Pointer* is used. Based on this value, the previous assignment, *out_val*, of that variable is copied from the *Solutions* module to the *Generator*. The *Generator* will produce a new assignment of that variable, *in_val*, and save it. When the variable is free or '0', the *Generator* will produce a '0' and '1' respectively. If the value of the variable is '1', no next possible assignment is available. In this case, backtracking should be executed and the previous assignment should be removed.

The following pseudocode describes the mechanism of the *Solution Generator*.

```
generate()
{
-- free to '0'
       if out_val = "00" then
              in_val = "01";
              jump to next state;
-- '0' to '1'
       elsif out_val = "01" then
              in_val = "11";
              jump to next state;
-- '1' to free, backtrack
       elsif out_val = "11" then
              in_val = "00";
              if GPointer = 0 then
                  finish searching;
                  jump to idle state;
              else
                  GPointer = GPointer - 1;
                  remain in current state;
              end if;
       end if;
}
```

3.4 Evaluator

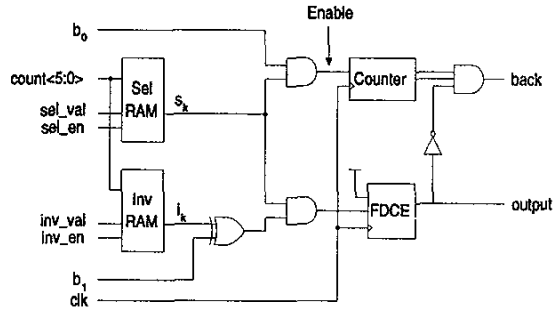

Figure 4: The gate level diagram of a 1-bit evaluator

Our implementation uses a serial evaluator for a clause. Therefore, the *Evaluator* consists of 80 1-bit evaluators shown in Figure 4. The evaluator consists of two independent memory modules, *Sel* and *Inv*. For the clause, $x_i + \overline{x_j} + \overline{x_k}$ the *Sel* memory is set to '1' for the addresses corresponding to variables i, j and k and '0' otherwise. Since variables x_j and x_k are inverted, the addresses of the *Inv* memory corresponding to these variables is set to '1' and the other addresses set to '0'. The *Sel* and *Inv* memories each require 2 Xilinx 32 × 1*s* RAMs. The values of the *Sel* and *Inv* memories are configured at runtime.

To evaluate each clause, a summation circuit is used. The *output* is '1' if the value of any variable in the clause is '1'. The boolean function computed in each cycle is $(I_k \oplus b_1) \wedge S_k)$ where b is the variable assignment described in Section 3.2, S is the output of the *Sel* memory and I is the output of the *Inv* memory. After n cycles have been executed (where n is the number of variables in the problem), the *output* of the evaluator is

$$(I_1 \oplus b_1) \wedge S_1) + (I_2 \oplus b_1) \wedge S_2) + \ldots + (I_n \oplus b_1) \wedge S_n)$$

A 2-bit counter is used to check whether the clause is determined or undetermined. If the clause consists of a

variable that is free, the clause is undetermined. Otherwise the clause is determined since all variables have been assigned a value. A counter is used to count the number of variables that have been assigned to values (see Figure 4). If, after iterating through all of the variables, this is equal to the number of literals in the clause, the clause is determined, otherwise it is undetermined. If the clause is determined then *back* will be set to '1' if the output is '0', a '0' output meaning the partial assignment of variables cannot satisfy the clause.

Each evaluator requires 8 CLBs. Therefore, the total number of CLBs required for the *Evaluator* module is 80×8 or 640 CLBs.

3.5 AND/OR

There are two outputs from the evaluator, *back* and *output*. To see if backtracking should be executed, all the outputs, *back*, are OR'ed together. If the result, namely *tot_back*, is '1', at least one clause is not satisfied. If *tot_back*, is '0', searching can continue and an assignment is found for the next variable. 13 CLBs are required to compute *tot_back*.

To check if a solution has been found, all the outputs, *output*, are AND'ed together. If the result, namely *tot_out*, is '1', all the individual outputs are '1' and the values of the variables correspond to a solution to the problem. 13 CLBs are used to compute *tot_out*.

3.6 State Machine

There are three main parts of state machine. The first part is used to configure the system for a particular SAT problem and involves the host computer configuring the *Sel* and *Inv* RAMs (see Section 3.4) with the appropriate values. The second part implements the main evaluation process and the third part writes the solution to an external memory.

Figure 5 shows the state diagram for problem configuration. Initially, the FPGA sends an interrupt request to the host to indicate that it is ready to receive data (state 0). Before an interrupt acknowledge is sent by the host, the host writes the data to a static RAM. Refer to the Figure 6 for the 2-way hand-shaking protocol. Following this, the FPGA requests an access to the local memory bus (state 1). After the FPGA has received the grant from the memory controller, it will fetch the data from the external static memory and write them to the Xilinx RAMs, *Sel* and *Inv*, inside the evaluator. The data arrives 2 cycles after the addresses are sent to the dual port memory controller (refer to the Figure 7 for details). State 2 and 3 are wait states. The state machine will remain in state 4 until 250 ($\frac{2\times80\times50}{32}$)

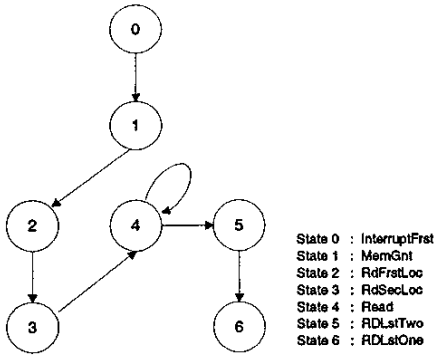

Figure 5: The state diagram for problem configuration

consecutive addresses have been sent to the memory controller. The next two states, 5 and 6, receive the remaining data. After that, all the relevant data have been written to the Xilinx RAMs.

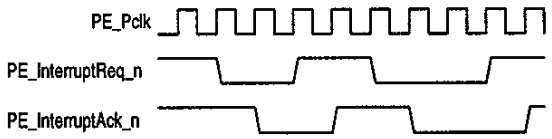

Figure 6: The timing diagram for the hand-shaking

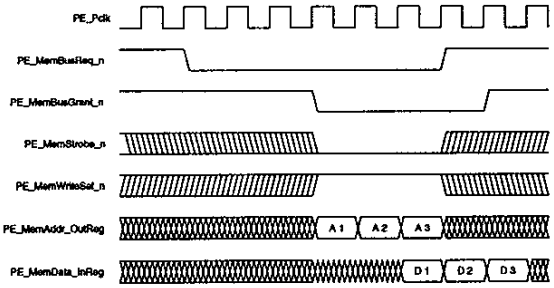

Figure 7: The timing diagram for consecutive memory read accesses

Figure 8 shows the state diagram of the evaluation process. Initially, all the variables are free and reset to "00". In State 7, a new assignment to the current variable is generated. At the same time, the new assignment will be updated in the *Solutions* module. The search is completed if the pointer points to the first variable and no possible assignment exists in which case it will jump to state 14 which is a idle state. The system remains in state 8 for 50 cycles

during which time the variables are tested in the *Evaluator*. During each these 50 cycles, the value of *tot_back* from the output of the *AND/OR* module is checked. If the value is '1', backtracking should be executed and the state machine will change to state 9. In state 9, the pointer will be updated if backtracking is not required. The system will also check the value of *tot_out* from the *AND/OR* module. If equal to '1', a solution has been found.

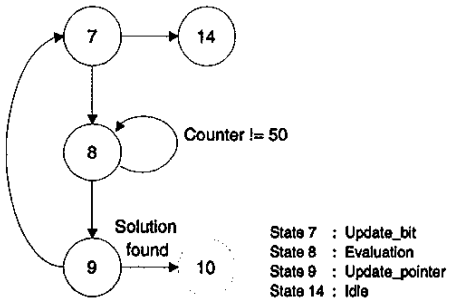

Figure 8: The state diagram of the evaluation

Figure 9 shows the state diagram of solution writeback to the external memory. State 10 is the memory bus request. State 11 will last for 50 cycles to fetch the assignments, x_i, of every variable. State 12 and 13 are the first and second write cycles respectively, to the external memory. Only two cycles are required since the data bus width is 32 bits.

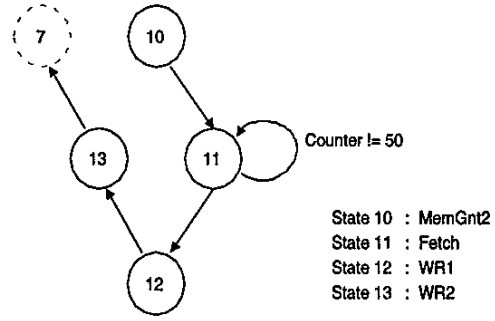

Figure 9: The state diagram of the solution write-back

3.7 Hardware Resource

To estimate the hardware resources required by our design, the equations in the following table can be used.

Modules	Number of CLBs used
Solutions	$\lceil \frac{variable}{32} \rceil \times 2$
Solutions Generator	~ 10
Evaluator	$clause \times (2 \times \lceil \frac{variable}{32} \rceil + 4)$
AND/OR	$\sim (\lceil \frac{clause}{8} \rceil + \lceil \frac{clause}{64} \rceil) \times 2$
State Machine	$state \times 4$
Interface	~ 200
Total	$(clause + 1) \times \lceil \frac{variable}{32} \rceil \times 2 +$ $(\lceil \frac{clause}{8} \rceil + \lceil \frac{clause}{64} \rceil) \times 2 +$ $(clause + state) \times 4 + 210$

In the case of 50 variable, 80 clause 3-SAT problem, the estimated number of CLBs using this equation is 956 compared with an actual value of 977 (see Section 4).

Table 1 shows estimates of CLBs for several standard DIMACS benchmark SAT problems [2].

Problem	Variables	Clauses	CLBs used
aim-50-1_6-yes1-1	50	80	956
aim-50-2_0-yes1-1	50	100	1084
aim-100-2_0-yes1-1	100	200	2716
aim-100-6_0-yes1-1	100	600	7628
aim-200-1_6-yes1-1	200	320	6114
aim-200-6_0-yes1-1	200	600	22242
dubois20	60	160	1580
dubois30	90	240	2724
hole6	42	133	1358
ii8a2	180	800	9838
ii32c1	225	1280	26226
par8-1-c	64	254	2358
pret60_25	60	160	1580
pret150_25	150	400	5974

Table 1: Number of CLBs used for several DIMACS SAT problems

4 Results

The SAT solving architecture was implemented on an Annapolis Micro Systems Inc, Wildforce PCI board [1]. The PCI bus runs at a clock speed of 33MHz and employs a 32-bit data bus such that the peak bandwidth can be up to 133MB/s. The board consists of one Control Processing Element (CPE), a Xilinx's XC4085XL FPGA and 4 Array Processing Elements (PEs), Xilinx's XC4062XL FPGA. The total gate counts can be up to 333K equivalent gates. Each PE consists of its own 4Mb dual ported SRAMs.

A C program generates the contents of the *Sel* and *Inv* RAMs of the evaluator. Using this system we configured the hardware to solve the DIMACS aim-50-1_6-yes1-1. In fact we can solve any 3-SAT problem with size up to ≤ 50 variables and ≤ 80 clauses using only runtime configuration.

The 50 variable and 80 clause 3-SAT problem described in Section 3 was implemented using a single FPGA on the Wildforce board using Xilinx 1.4 Foundation Tools with

the Foundation Express VHDL compiler. The number of CLBs and IOBs used in a Xilinx XC4062XL FPGA were 977 and 77 respectively, corresponding to 42% and 39% utilization. The Xilinx timing analyzer reported a maximum of 10MHz at room temperature and the implementation was successfully tested on the Wildforce board at this frequency.

5 Conclusions

An architecture was presented which allows satisfiability problems to be solved using reconfigurable hardware. The problem is stored in memory and is hence runtime configurable, avoiding costly synthesis and place and route steps required by other approaches. Furthermore, parallelism and speed were sacrificed for better utilization of logic resources enabling very large problems to be solved with modest hardware requirements. It is hoped that such an architecture could be used to implement real-time systems with embedded constraint solving engines.

References

[1] In *http://www.annapmicro.com/wfhtm.html.*

[2] *Dimacs challenge benchmarks.* ftp://dimacs.rutgers.edu/pub/challenge.

[3] Xilinx Inc. *The Programmable Logic Data Book.* 1998.

[4] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[5] M. Yokoo, T. Suyama, and H. Sawada. Solving satisfiability problems using field programmable gate arrays: First results. In *Proceedings of the 2nd Inter. Conf. on Principles and Practice of Constraint programming*, pages 497–509, 1996.

[6] P. Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating boolean satisfiability with configurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 186–195, 1998.