University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow, G12 8QQ

Session 2006/2007

Level 4 project within the scope of an academic year abroad

Solving NP-complete problems in hardware

Andreas Koltes

10/04/2007

Supervisors: Dr. Paul W. Cockshott, Dr. John T. O'Donnell

Contents

Abstract **vii**

1 Introduction **1**

2 Project description and hypothesis **3**
2.1 Boolean satisfiability problems 3
2.2 Applications of SAT solvers 4
2.3 Complexity related phenomena 5
2.4 Basic circuit architecture 6
2.5 Introduction to FPGA technology 8

3 Basic experiments and infrastructure **13**
3.1 Basic manual experiments 13
3.2 Modularisation and automation 21
3.3 Acquisition of reference data 53

4 Large-scale experiments **59**
4.1 Basic circuits 59
4.2 Phase transition related experiments 61
4.3 Globally probability driven circuits 62
4.4 Locally probability driven circuits 64

5 Analysis of results **67**
5.1 Asynchronous circuits 67
5.2 Synchronous circuits 67

6 Conclusion and future work **77**

Bibliography **79**

A Infrastructure tools **81**
A.1 Small instance unsatisfiability search tool 81
A.2 Seed generator 83
A.3 Simulated annealing stepping table generator 84
A.4 40-bit LFSR seed checking tool 85
A.5 Sleeping tool 86
A.6 SAT instance generator version 1 86
A.7 SAT instance generator version 2 87

B VHDL Library **91**
B.1 Term evaluators 91
B.2 Variable sources 93
B.3 Fixed distribution bit sources 99
B.4 Pseudo-random random number generators 118
B.5 Support circuitry 121

C Top level circuit setups **135**
C.1 Basic asynchronous circuitry 135
C.2 Basic synchronous circuitry 137
C.3 Basic probability driven asynchronous circuitry 139
C.4 Template for globally probability driven circuitry 142
C.5 Template for single instance batch testruns 144
C.6 Template for simulated annealing experiments 147
C.7 Template for locally probability driven circuitry 149

D Result tables of experiments **153**
D.1 Automated experiments on small SAT instances 153
D.2 Experiments using fixed toggling probabilities 160
D.3 Experiments using derived toggling probabilities 162
D.4 Results of insufficient randomisation 162
D.5 Comparision of different randomisation engines 163
D.6 Phase transition experiments 165
D.7 Runtime statistics 167
D.8 Dynamic probability calculation using simulated annealing 179
D.9 Performance results of locally probability driven circuits 182

List of Figures

2.1 Basic term evaluator module 7
2.2 Basic combinational variable source module 7
2.3 Basic synchronous variable source module 8
2.4 Altera Cyclone device block diagram 9
2.5 Altera Cyclone device logic cell operating in normal mode 10
2.6 Altera Cyclone device logic cell cluster structure 11
2.7 Altera Cyclone device memory block operating in single-port mode 11

3.1 SLS UP3-1C6 Cyclone FPGA development board 14
3.2 Altera Cyclone series EP1C6Q240 FPGA chip 15
3.3 Santa Cruz long expansion headers 15
3.4 Synchronous circuit implementing 4x4 3CNF-SAT instance 18
3.5 Asynchronous circuit implementing 4x4 3CNF-SAT instance 19
3.6 Hardening of variable signals against compiler optimisations 20
3.7 Hardening of feedback signals against compiler optimisations 20
3.8 Example support circuitry layout for automated test case execution 22
3.9 Block diagrams of term evaluator modules 23
3.10 Schematic diagram of basic term evaluator module 24
3.11 Schematic diagram of probabilistic term evaluator module 26
3.12 Schematic diagram of erroneous probabilistic term evaluator module 27
3.13 Block diagrams of variable source modules 28
3.14 Schematic diagram of basic asynchronous variable source module 28
3.15 Schematic diagram of basic asynchronous variable source module 30
3.16 Schematic diagram of basic synchronous variable source module 31
3.17 Schematic diagram of hardened synchronous variable source module 33
3.18 Schematic diagram of hardened compact synchronous variable source module . . . 34
3.19 Schematic diagram of experimental locally probability driven variable source module (example for a variable participating in 5 clauses) 35
3.20 Block diagrams of bit source modules 36
3.21 Schematic diagram of bit source using single bit LFSR 37
3.22 Example of a memory initialisation file (MIF) 39
3.23 Block diagrams of LFSR based pseudo-random number generator modules 40
3.24 Schematic diagram of single bit LFSR (40-bit) 40
3.25 Schematic diagram of parallelised LFSR (40-bit) 41
3.26 Block diagram of delayed startup controller module 43
3.27 Schematic diagram of delayed startup controller for single testruns 44
3.28 Block diagram of timeout controller module 45
3.29 Schematic diagram of timeout controller for single testruns 46
3.30 Block diagram of performance measurement module 47
3.31 Schematic diagram of performance measurement 47
3.32 Block diagrams of memory controller modules 48
3.33 Example script controlling automated operation of the Altera Quartus II development environment 52
3.34 Example script controlling JTAG communication through the Altera Quartus II development environment 53

3.35 Basic algorithm for generation and pseudo-random SAT instances (not recommended for future experiments) 54
3.36 Improved algorithm for generation and pseudo-random SAT instances 55
3.37 Example C/Assembler source for reading the time-stamp counter of Intel IA-32 compatible CPUs 56

5.1 Location of phase transition point (Y-axis) depending of the number of participating variables (X-axis) 69
5.2 Fraction of satisfiable random instances consisting of 10 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis) 69
5.3 Fraction of satisfiable random instances consisting of 50 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis) 70
5.4 Fraction of satisfiable random instances consisting of 100 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis) 70
5.5 Approximated distribution of runtime of WalkSAT solver (X-axis) showing scaled probability approximation (Y-axis) 73
5.6 Peak area of approximated distribution of runtime of hardware SAT solver (X-axis) showing scaled probability approximation (Y-axis) 73
5.7 Beginning of tail area of approximated distribution of runtime of hardware SAT solver (X-axis) showing scaled probability approximation (Y-axis) 74
5.8 Approximated distribution of runtime quotients SAT solvers (X-axis) showing scaled probability approximation (Y-axis) 74

List of Tables

3.1 Timings of asynchronous circuit stabilisation 17
3.2 Basic term evaluator interface 25
3.3 Probabilistic term evaluator interface 26
3.4 Basic asynchronous variable source interface 29
3.5 Hardened asynchronous variable source interface 30
3.6 Basic synchronous variable source interface 32
3.7 Hardened synchronous variable source interface 33
3.8 Experimental locally probability driven variable source interface 35
3.9 Fast modulo computation interface 36
3.10 Interface of bit source using single bit LFSR 37
3.11 Interface of bit source using parallelised LFSR array and preseeding 38
3.12 Interface of single bit LFSR (40-bit) 41
3.13 Interface of parallelised LFSR (40-bit) 42
3.14 Interface of parallelised LFSR supporting variable seed (40-bit) 42
3.15 Interface of parallelised LFSR supporting variable seed (41-bit) 43
3.16 Interface of delayed startup controller for single testruns 44
3.17 Interface of delayed startup controller for batch testruns 45
3.18 Interface of timeout controller for single testruns 45
3.19 Interface of timeout controller for batch testruns 47
3.20 Performance measurement interface 48
3.21 Interface of memory controller for single testruns 49
3.22 Data format produced by memory controller for single testruns 49
3.23 Interface of memory controller for batch testruns 50

D.1 Performance of early circuit variants on SAT instances consisting of 10 variables and 30 clauses 154
D.2 Performance of early circuit variants on SAT instances consisting of 10 variables and 40 clauses 155
D.3 Performance of early circuit variants on SAT instances consisting of 10 variables and 50 clauses 156
D.4 Performance of early circuit variants on SAT instances consisting of 10 variables and 60 clauses 157
D.5 Performance of early circuit variants on SAT instances consisting of 10 variables and 70 clauses 158
D.6 Performance of early circuit variants on SAT instances consisting of 10 variables and 80 clauses 159
D.7 Performance of SAT circuits using fixed toggling probabilities 161
D.8 Performance of SAT circuits using derived toggling probabilities 162
D.9 Performance of SAT circuits using derived toggling probabilities with insufficient randomisation engine 163
D.10 Performance of SAT circuits using different randomisation engines 164
D.11 Fraction of satisfiable random SAT instances regarding ratios of clauses to variables with approximated phase transition points (Part 1) 165
D.12 Fraction of satisfiable random SAT instances regarding ratios of clauses to variables with approximated phase transition points (Part 2) 166

D.13 Runtime statistics of hardware SAT solver engine (Probability multiplier 0.750) . . 168
D.14 Runtime statistics of hardware SAT solver engine (Probability multiplier 0.875) . . 169
D.15 Runtime statistics of hardware SAT solver engine (Probability multiplier 1.000) . . 170
D.16 Runtime statistics of hardware SAT solver engine (Probability multiplier 1.250) . . 171
D.17 Runtime statistics of hardware SAT solver engine (Probability multiplier 1.500) . . 172
D.18 Runtime statistics of hardware SAT solver engine (Probability multiplier 1.750) . . 173
D.19 Runtime statistics of hardware SAT solver engine (Probability multiplier 2.000) . . 174
D.20 Runtime statistics of hardware SAT solver engine (Probability multiplier 2.250) . . 175
D.21 Runtime statistics of hardware SAT solver engine (Probability multiplier 2.500) . . 176
D.22 Runtime statistics of WalkSAT solver engine (flip counts) 177
D.23 Runtime statistics of WalkSAT solver engine (cycle counts) 178
D.24 Performance of SAT circuits using simulated annealing approach (Part 1) 180
D.25 Performance of SAT circuits using simulated annealing approach (Part 2) 181
D.26 Performance of SAT circuits using locally probability driven approach (Part 1) . . 183
D.27 Performance of SAT circuits using locally probability driven approach (Part 2) . . 184

Abstract

In [COP06], Paul Cockshott, John O'Donnell and Patrick Prosser proposed a new design for a hardware based incomplete SAT solver based on highly parallelised circuitry running in eihter a FPGA or a structured ASIC. The design is based on fundamental theories about self-stabilisation of complex systems published in [Kau93]. This project aims at the exploration of the feasability of the proposed basic design investigating different implementation strategies using synchronous as well as asynchronous circuits. It is shown that the proposed design makes it possible to speed up conventional incomplete SAT solver based on algortihms implemented in software by a full order of magnitude. Behavioral properties of different hardware algorithms based on the basic design are investigated and the foundations for future research on this topic layed out.

1 Introduction

During the last century, many fundamental results in computability theory were discovered which are based on mathematical state machines. The type of mathematical concept has been used, for example, to prove the computational equivalence of a variety of mathematical computability models, including Turing Machines, lambda calculus, and the Post Correspondence problem. The Church-Turing Hypothesis even uses them to define the set of computable problems. Based on these foundations a large construct of complexity theory has been constrcuted.

However, some computational models are based on natural phenomena in physics and chemistry, being fundamentally different compared to the mentioned concepts, because they do not operate by moving through a sequence of well-defined states. Examples for this type of model include annealing, protein folding, combinational circuits with feedback as well as quantum computing. Whether these systems are subject to the same comparatively well understood computability limitations as state machines is still an open question. A strong form of the Church-Turing Hypothesis assmues that physical systems are subject to the same computability limitations as state machine models whereas weak forms of the Church-Turing Hypothesis leave room for these systems being eventually able to break the limitations of traditional state machine like concepts.

One of the aims of this project is to perform an experiment designed to provide evidence that will support or weaken the strong Church-Turing Hypothesis. The basic design behind the experiment uses a class of combinational circuits with feedback in order to attempt to solve a problem, 3SAT, which is NP-complete on state machines. In parallel, it is also tried to construct efficient synchronous circuits with feedback for comparision purposes and to eventually explore ways to speed-up computation of SAT problems in hardware which would be of high practical value.

Combinational circuits with feedback do not necessarily behave like state machines: They may settle down in a stable state, they may oscillate among a set of states, or they may vary chaotically, in which case it is hard to predict whether they will ever settle down in the future. Because of this chaotic nature, combinational circuits with feedback are a topic within computer science which is still far away from being fully understood giving plenty of space for research activity. Because of this complex behaviour, most practical digital hardware avoids combinational circuits with feedback, and uses the synchronous model instead.

The computational problem investigated during this project is Boolean satisfiability with clauses consisting of three terms; this is often called 3SAT, and is a standard NP-complete problem. An arbitrary instance of 3SAT will be compiled (in polynomial time) into a corresponding combinational circuit, and the execution of the circuit may solve the 3SAT problem instance. For simplicity reasons, the SAT problems investigated during this project belong to the 3CNF-SAT type which is among the easiest Boolean satisfiability problems still being NP-hard.

Preliminary experimentation with the SAT circuitry was carried out by Paul Cockshott, using an older FPGA board. Initial results show that the circuit can solve some 3SAT problems quickly. To continue the research, it is necessary to reimplement the circuit using a modern and larger scale FPGA, to instrument the hardware so that its performance can be measured, and to experiment with the hardware on a range of randomly chosen problems in an automated way allowing for the collection of statistically meaningful data.

There are effective techniques for proving the correctness of synchronous digital circuits, such as model checking [ECGP99] and equational reasoning [OR04], and a major research topic in computer hardware is the methodology for designing reliable circuits to solve problems. These proof techniques are based on state machine models, and they do not apply to combinational circuits with feedback. Even if applied to synchronous circuits, the mentioned techniques have limits regarding the size and complexity of the circuits practically analysable. This it is impossible to prove the correctness of the hardware 3SAT solver, or to analyse its time complexity precisely. Instead, an experimental approach is needed to evaluate the approach, and to assess its implications for the Strong Church-Turing Hypothesis as well as for new ways to efficiently solve SAT problems in hardware. Thus the proposed research cannot give a definitive answer to the hypothesis, but it will give an enlightening data point.

Previous research has shown that the set of 3SAT problems has an interesting structure, with a phase change from a subset of problems with few solutions to a subset of problems with many solutions [Hay03] [Hay97]. The instances of 3SAT that are hard lie mostly near the phase change. This previous research is also experimental: Large sets of problem instances are generated randomly and their solution times are measured. Investigation of these phase transition related phenomena is carried out where it is applicable.

2 Project description and hypothesis

2.1 Boolean satisfiability problems

The Boolean satisfiability problem (SAT) is the problem of determining whether the variables of a given boolean term can be assigned in a way as to make the term evaluate to $true$. Equally important for many applications is the inverse problem to determine that no truth assignment exists satisfying the boolean formula. This implies, that the given term evaluates to $false$ for any given truth assignment. In the first case the formula is called satisfiable otherwise it is unsatisfiable. The term "boolean" satisfiability refers to the binary nature of the problem which is also known as propositional satisfiability. Often the term "SAT" is used as a shorthand to denote the boolean satisfiability with the implicit understanding that the function as well as its variables are strictly binary valued. A binary value of 1 is commonly used to denote a boolean value of $true$ whereas the value 0 is used to denote $false$. Abstracting from the fact whether a formula is given in a boolean or a binary form, a specific boolean expression is also referred to as being an instance of the boolean satisfiability problem.

2.1.1 Basic definitions and terminology

Formal definitions of SAT usually make use of the function to be expressed being in the so-called conjunctive normal form (CNF). This means that the function consists of a conjunction of disjunctions of literals. A disjunction of literals is a term consisting of an arbitrary number $n \geq 1$ of literals, which are combined using the Boolean OR function. A literal is either a variable (called a positive literal) or its complement (called a negative literal). The disjunctions contained in a SAT instance are referred to as clauses and implicitly act as constraints on the possible values of its variables allowing the instance evaluating to $true$. For example the clause $(\overline{A} \vee B \vee C)$ is satisfied by all truth assignments of the variables A, B and C except $A = true$ and $B = C = false$. All clauses of an instance are combined using the Boolean AND function forming the full function term. This requirement is not a restriction on the representable Boolean functions because every Boolean function can be transformed into an equal Boolean function in CNF. A Boolean formula in CNF can be viewed as a system of simultaneous constraints in the parameter space of the instance consisting of all possible truth assignments of its variables. This is analogous to a system of linear inequalities over real variables modelling the set of feasible assignments (also called the feasible region) in a linear program. The feasible region of a CNF formula therefore contains precisely those truth assignments which make the formula evaluating to $true$. It is very important to understand that the Boolean AND as well as the OR functions are commutative, associative and idempotent. Therefore reordering or duplicating clauses or literals respectively do not change the actual SAT instance.

In complexity theory, the Boolean satisfiability problem is actually a decision problem, whose instance is an arbitrary Boolean expression. The question is: Given the expression, is there a truth assignment of the variables contained in the instance existing, which makes the entire expression evaluating to $true$? The inverse problem, whether there is no such truth assignment is sometimes referred to as the Boolean unsatisfiability problem (UNSAT). Both of these problems are NP-complete.

Even if the SAT problem is significantly restricted to expressions being in 3CNF it remains NP-complete. A Boolean expression is in 3CNF if it is in CNF with each clause containing at most three different literals. The restriction of the SAT problem to 3CNF expressions is often referred to as 3SAT, 3CNFSAT or 3-satisfiability. The proof of the 3SAT problem being NP-complete is known as Cook's theorem and in fact was the first decision problem proved to be NP-complete.

Only by restricting the problem even further, it can be brought below NP-completeness. If the Boolean expression is required to be in 2CNF, the resulting problem, 2SAT, is NL-complete. Alternately, if every clause is required to be a Horn clause, containing at most one positive literal, the resulting problem, Horn-satisfiability, is P-complete.

There are also extensions to the basic SAT problem as for example the QSAT problem which asks the question whether a Boolean expression containing quantifiers is satisfiable. However, all of these problems are at least NP-complete and were not further investigated during this project.

2.2 Applications of SAT solvers

Despite looking like a rather theoretical problem without much practical significance, there are many practical applications of SAT solvers able to decide the satisfiability of a given SAT instance. Over the last decade many scalable algorithms were developed which can efficiently solve many practically occurring instances of SAT even if they reach enormous sizes containing tens of thousands of variables and millions of clauses. Practical applications of SAT solvers include amongst many others:

- Routing in FPGAs
- Combinational equivalence checking
- Model checking
- Formal verification of circuits
- Logic synthesis
- Graph colouring
- Planning problems
- Scheduling problems
- Cryptanalysis of symmetric encryption schemes

In fact, a capable SAT solver is nowadays considered to be an essential component of Electronic Design Automation (EDA) tools and all EDA vendors provide such capabilities (usually employed behind the scenes of the software tools). SAT solvers currently also find their way into many other application domains because more and more ways are developed to efficiently transform or reduce respectively many other problems into SAT problems.

Despite the availability of efficient general purpose SAT solvers as well as SAT solvers specifically optimised for SAT problems originating from specific domains, the underlying SAT problem remains a computationally hard problem. Therefore there are many SAT instances even highly optimised algorithms take a long time to solve (if they are able to solve the instance in reasonable time at all). Because of this fact for many applications it would be beneficial to have some sort of hardware accelerated SAT solving engine available which is able to operate at far higher speeds than a pure software implementation.

In practice, there are two large classes of high-performance algorithms for solving instances of the SAT problem. The first class is known as the class of complete SAT solvers. This type of algorithm guarantees termination after a finite amount of time returning either a truth assignment modelling the investigated expression or guaranteeing the passed SAT instance being unsatisfiable. The time required for this type of algorithm can of course be exponential in the number of variables contained in the instance. Currently, the fastest general purpose SAT solvers belonging to this class implement variants of the DPLL algorithm (for example Zchaff2004, GRASP, BerkMin and MiniSAT). The second class of SAT solvers is known as the class of incomplete SAT solvers. These solvers either return a truth assignment modelling the passed expression or basically run forever or until a certain timeout is reached (analogous to a semi-determinable problem). This implies that this type of solver is not able to prove the unsatisfiability of a problem (but in fact, for many practical applications, this is not necessary). Solvers belonging to this class usually implement probability driven stochastic local search algorithms. Examples for solvers belonging to this class are WalkSAT and its predecessor GSAT having features which are similar to Tabu search.

DPLL SAT solvers employ systematic backtracking search procedures to explore the (exponentially-sized) parameter space of truth assignments looking for satisfying assignments. This type of solver usually also employs some sort of branch-and-bound strategy to exclude truth assignments known as definitely not satisfying the investigated instance. The basic search procedure was proposed in two seminal papers in the earls 1960s and is now commonly referred to as the David-Putnam-Logemann-Loveland (DPLL) algorithm. Modern SAT solvers extend the basic DPLL approach by efficient conflict analysis, clause learning, non-chronological backtracking (also known as back-jumping), “watched-literal” unit propagation, adaptive branching and random restarting to maximise the average speed or to optimise the algorithm for SAT instances originating of specific application domains. These extensions to the basic systematic search strategy proved to be essential for handling very large SAT instances especially arising in EDA. Powerful solvers of this type are readily available in the public domain and are remarkably easy to use. In particular, MiniSAT (which was also used during the project to produce reference data and verify results) is a small but yet highly efficient complete SAT solver which won the 2005 SAT competition. Despite this achievement, the main solver engine of MiniSAT consists of only about 600 lines of C++ code.

Genetic algorithms and other general-purpose or specialised stochastic local search methods are usually being employed by incomplete SAT solvers. These are especially useful when there is no or limited knowledge of the specific structure of the investigated problem instance to be solved. The hardware-based solvers developed during this project are belonging to this class of SAT solvers, too.

2.3 Complexity related phenomena

In [CKT91] Cheeseman, Kanefsky and Taylor observed an abrupt phase transition from solubility to insolubility in graph colouring problems as average degree was increased. In the area of this phase transition a complexity peak was observed leading to a comparatively high computation effort being required to solve problems lying in this area. It was conjectured that this kind of phase transition phenomenon would be algorithm independent and eventually even common to all NP-complete problems. The same phenomenon was observed regarding SAT problems originating from transformed graph colouring problems. Later research showed that incomplete algorithms also experienced this kind of phenomenon including the corresponding complexity peak when applied to satisfiable instances. This means that easily soluable problem instances were easy to solve, hard soluable instances were hard and rare soluable instances found in the easy insoluable region were easy, too. Much research got carried out regarding the location of the 3SAT phase transition and to develop theories about the location of this phase transition for other problems being NP-complete or even belonging to higher complexity classes (e.g. QSAT being PSPACE-complete). Research done to date appears to confirm the algorithm independence of the complexity peak, but this has only been investigated with respect to complete and incomplete algorithms.

It was conjectured that there would be another phase transition, this time between complexity classes. As mentioned above, the 2SAT problem lies below the NP complexity class, whereas 3SAT is NP-complete. Similarly 3COL is NP-complete whereas 2COL is in P. Experiments were performed mixing clauses of lengths 2 and 3 giving an average clause length somewhere in the interval $[2, 3]$. It was observed that SAT problems having an average clause length of 2.4 or above behave as if they were NP-complete, whereas polynomial complexity behaviour was observed below this threshold. This has several implications for algorithm design, because if a process can make decision that when propagated leave the majority of clauses to have a length of 2 then the remaining sub problem becomes polynomial and easily soluable. The transition from P to NP was also observed in a variety of problems by Walsh [Wal02b].

Beside the development of fast SAT solving circuitry another aim of this project was to perform experiments regarding the behaviour of hardware SAT solvers regarding the presented phenomena. The experiments carried out during the project covered a variety of synchronous circuits as well as a few asynchronous circuit variants.

Previous research has shown that the set of 3SAT problems has an interesting structure, with the mentioned phase change from a subset of problems with few solutions to a subset of problems with many solutions [CKT91]. The 3SAT instances being hard lie mostly in the phase transition area. This previous research is also experimental: Large sets of problem instances are generated randomly and their solution times are measured.

During the project the behaviour of various circuit solvers was investigated by observing their results and comparing them to the results obtained using a complete software solver. Problem instances on both sides of the phase change area and at the phase change itself were of special interest during the research.

2.4 Basic circuit architecture

A SAT expression E can be directly implemented as a combinational circuit which determines whether the expression is satisfied, for a given set of inputs. Because of the fact that the Boolean AND as well as the Boolean OR functions are commutative as well as associative the circuit can be implemented forming some sort of tree structure evaluating very rapidly. The average evolution time is roughly proportional to $G \log n_E$ with G being a gate delay and n_E being the number of sum terms in the final product.

In order to find a solution to the given SAT problem, it is necessary to construct a feedback circuit which alters the values of the truth assignment v until E is satisfied. Regarding a fully combinational circuit this can be reposed as "construct a Boolean circuit over v whose only stable states are those satisfying E". This differs from an algorithm iterating in a state machine because the alterations to the variable settings are made by an asynchronous circuit. In the case of a synchronous circuit, the execution model is equal to a software execution of an algorithm as long as there are no random components in the circuit (e.g. introduction of noise to a probability driven strategy).

An execution of the synchronous variant of the circuit is equal to the execution of an incomplete software SAT solver regarding its outcome. Regarding the asynchronous variant of the circuit, the circuit may settle down representing a solution. It may also oscillate indefinitely, when there is no solution (both circuit types will not prove the absence of a solution since they belong to the class of incomplete SAT solvers). It may oscillate between several solutions, or it may just oscillate without finding a solution even if one exists. It may continually change its variable settings without oscillating. In this case it is unclear whether the circuit will eventually find a solution in the future, given enough time (this is analogous to the Halting Problem and an inherent property of all incomplete SAT solvers).

Figure 2.1 on page 7 shows a schematic layout of a combinational circuit evaluating whether a particular clause of a 3SAT instance in the variables a, b and c is satisfied. Modules of this type are cascadable so that, provided that all the prior modules in the chain are satisfied, then the solved signal becomes *true*. To improve execution performance it is also possible to compute the final *solved* signal by a tree-structured sub circuit combining individual solution state signals from all term evaluator modules. If none of a, b and c are *true* the signals $awrong_{out}$, $bwrong_{out}$ and $cwrong_{out}$ are generated. These are propagated through all other modules that use the variables

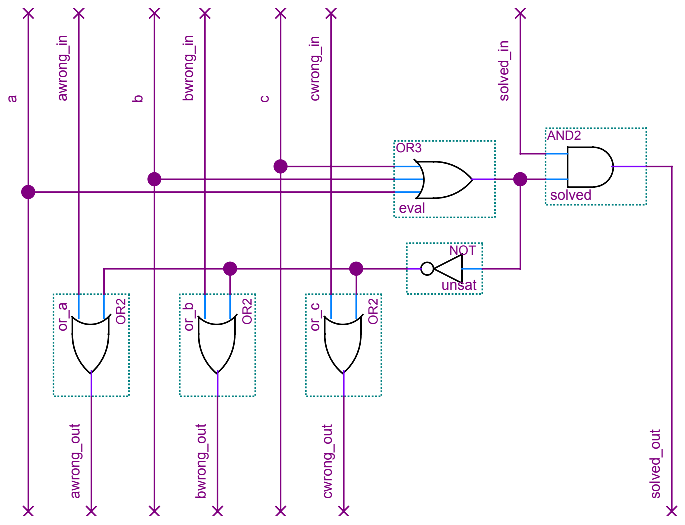

Figure 2.1: Basic term evaluator module

a, b or c.

The entire Boolean expression forming the SAT instance is represented by a collection of these modules, one for each clause in E having the following inputs:

- A signal for each element of v representing the positive literals
- A signal for the complement of each element of v representing the negative literals
- A $wrong_{in}$ signal for the straight and complement versions of each element of v

The circuit representing the entire SAT instance also has a $wrong_{out}$ signal for the straight and complement versions of each element of v. Modules of the basic structure shown in Figure 2.2 on page 7 and Figure 2.3 on page 8 generate the actual values of v on the basis of the feedback from the $wrong_{out}$ signals and optionally further information depending on the specific type of variable source module. If either the straight or the complement version of the variable are found to be wrong, a XOR gate is used to toggle its value.

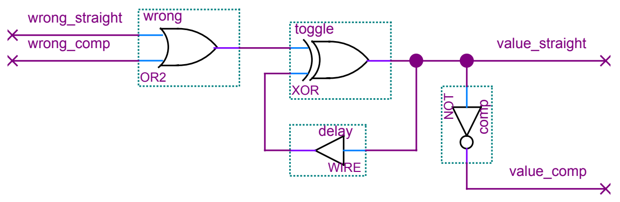

Figure 2.2: Basic combinational variable source module

The precise behaviour of the entire circuit depends of the fact whether the variable source modules are combinational or synchronous modules and their exact implementation. It is also possible to add further logic to the term evaluator modules to improve the circuit's overall performance. In the case of an unclocked circuit it can be expected that the circuit 'oscillates' until a truth assignment satisfying E is found. Simulations of small systems and preliminary experiments done in

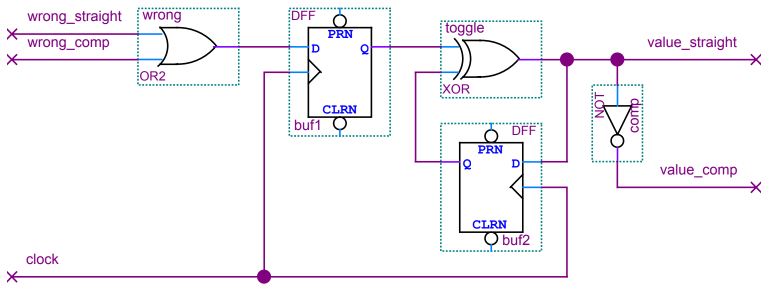

Figure 2.3: Basic synchronous variable source module

1995 using the Space Machine [BCMS92] [SCB96] indicated that such circuits stabilise on solutions to the implemented problem instance.

Prior experiments carried out during other projects indicate that the stabilisation may be too fast for the attached host computer to time so it is sensible to add a clocked on-chip timing circuit to measure the time the circuit requires to stabilise on a solution. In the case of a synchronous circuit this approach allows precise measurement of the number of clock cycles the circuit travels through until a solution is found.

For verification purposes it is also required to be able to read the actual truth assignment of the variables when a solution is found. In smaller experiments this can be achieved by letting the variable signals to external pins so that they can be monitored and verified. To allow for larger experiments and automated testing it is required to implement some sort of memory storage of the variable values to be able to read them using software running on the host computer.

2.5 Introduction to FPGA technology

Because of the enormous amount of different circuits arising during the project and because of the need for fully automated testing facilities, implementation of the circuits in application-specific integrated circuits (ASIC) was not feasible. Instead all circuits investigated where implemented using a field-programmable gate array (FPGA) chip. A FPGA is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic elements (also called logic cells or logic blocks) can be programmed to mimic the functionality of arbitrary small Boolean functions as for example AND, OR, XOR or NOT gates. More complex combinational functions such as decoders or simple mathematical functions can be implemented by cascading multiple logic cells. In most FPGAs, these logic cells also include memory elements, which may be simple flip-flops or more complete blocks of memory. Additionally to these flexible logic cells, many FPGAs also contain dedicated hardware multipliers, memory blocks, phase-locked loops or even small microprocessors to provide high-speed space-saving building blocks for commonly recurring functionalities.

A hierarchical structure of almost freely programmable interconnects allows the logic cells of a FPGA to be interconnected as needed to implement a specific circuit, similar to a one-chip programmable breadboard. These logic cells and interconnects can be programmed after the manufacturing process by the customer or designer (hence the term “field programmable”, i.e. programmable in the field) allowing the FPGA to mimic an almost arbitrary ASIC (or in fact even multiple ASICs since the programming can be changed as needed).

FPGAs are generally slower than their ASIC counterparts, cannot handle as complex a design because the logic density is about ten times lower than that of a corresponding ASIC and draw more power. However, they have several advantages such as a very short time to market, extremely short development and design cycles, the ability to re-program in the field to fix bugs or to mimic different chips as needed, and significantly lower non-recurring engineering costs. Some vendors also offer cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. The development of these designs is made on regular FPGAs and then migrated into a fixed version which more resembles an ASIC (an example for this technique is the Stratix HardCopy chip offered by Altera). Complex programmable logic devices (CPLD) are another alternative.

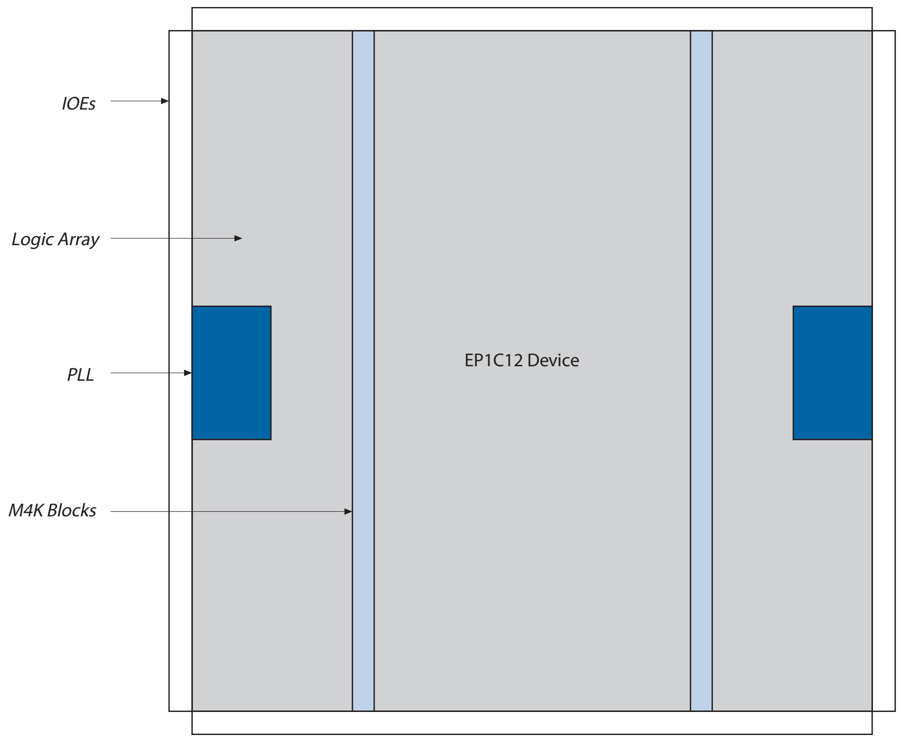

Figure 2.4: Altera Cyclone device block diagram

During the project a development board containing a low-cost Altera Cyclone EP1C6 FPGA in a 240-Pin PQFP package was used. Figure 2.4 on page 9 shows the overall structure of a Cyclone series FPGA device (the only difference to the one used is, that its memory is contained in a single column). This chip offers 5,980 logic cells each containing a 4-input lookup table producing a single output signal which can optionally passed through a flip-flop. The lookup tables and interconnects of the device are configured using SRAM based registers. All logic cells are grouped into clusters of ten cells which are surrounded by a 80-channel interconnect routing matrix. In addition to the logic cells, the device features 20 dedicated SRAM blocks each providing space for 4,608 bits of data (or 4,096 bits respectively without parity) supporting true dual-port memory access. The feature set is completed by two phase-locked loops supporting a wide variety of different frequency multipliers. The chip supports a maximum of 185 pins for data transfer including clock pins.

The logic cells featured by the FPGA device are able to implement logic which is far more complex than a single logic gate. In fact a single lookup table can implement an arbitrary Boolean function in up to four variables. If the implemented functions produce more than one output signal the implied lookup table has to be replicated forming one logic cell per output signal if necessary. One signal input of the lookup table is optionally assignable to an output of the previous logic cell in the same cluster (as displayed in Figure 2.6 on page 11) forming an efficient way for implementing carry chains.

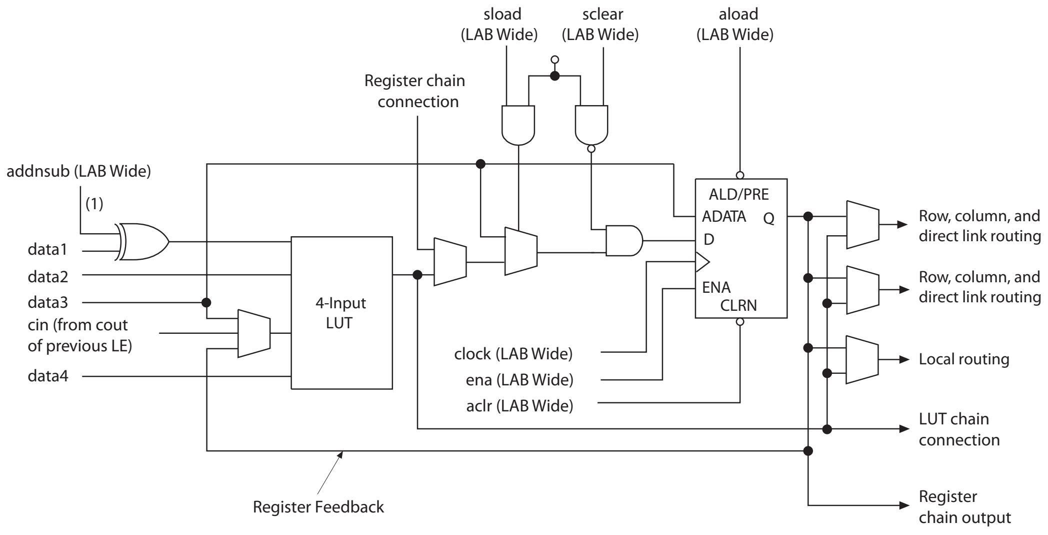

Figure 2.5: Altera Cyclone device logic cell operating in normal mode

Regarding the basic modules proposed in the previous section this means that these modules can be implemented in a very efficient way using the Cyclone FPGA device. The term evaluator module is implementing a binary function of type $(\mathbb{F}_2 \times \mathbb{F}_2 \times \mathbb{F}_2) \to \mathbb{F}_2$ fitting into a single logic cell. Since the combinational variable source module is of type $(\mathbb{F}_2 \times \mathbb{F}_2 \times \mathbb{F}_2) \to (\mathbb{F}_2 \times \mathbb{F}_2)$ it requires two logic cells for producing both output signals. The clocked version of the variable source module requires three logic cells. Two of them contain the flip-flops storing the variable state and a third one is required to produce the complemented variable value. These calculations are of course only theoretical because the synthesis software will combine logic cells where possible. For example the last logic cell implementing the single NOT gate will most likely be combined with the logic cells implementing the connected term evaluator modules fitting the variable source in only two logic cells.

As mentioned before, an automated test environment requires a way to automatically read the resulting truth assignment, the timing information and eventually other data from the FPGA device to the host computer. The easiest way to realise this is to write the data to one of the dedicated memory blocks shown in Figure 2.7 on page 11 embedded in the FPGA device. These memory blocks can be easily read using a standardised software interface (this is explained in detail in section Section 3.2.5).

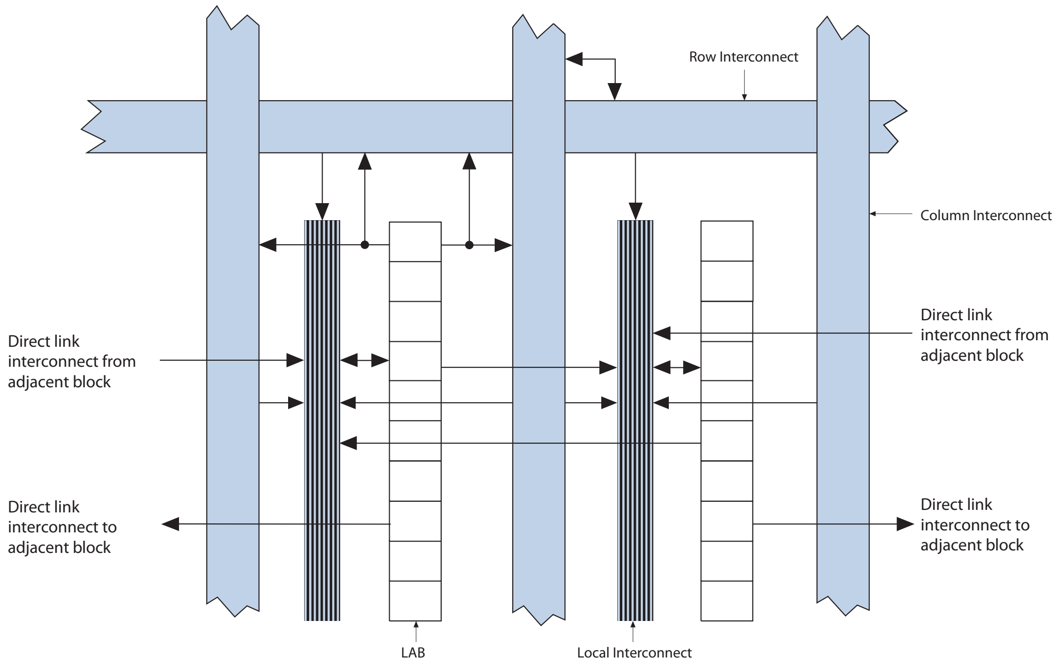
Row Interconnect
Column Interconnect
Direct link interconnect from adjacent block
Direct link interconnect from adjacent block
Direct link interconnect to adjacent block
Direct link interconnect to adjacent block
LAB
Local Interconnect

Figure 2.6: Altera Cyclone device logic cell cluster structure

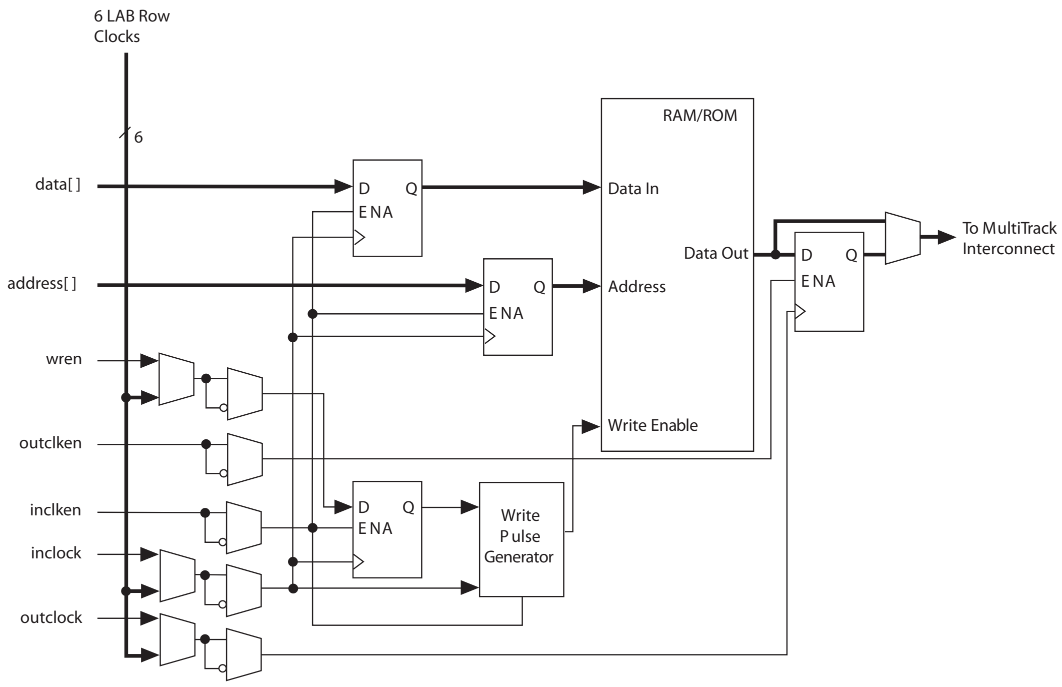

Figure 2.7: Altera Cyclone device memory block operating in single-port mode

3 Basic experiments and infrastructure

3.1 Basic manual experiments

The first step in the project was the manual implementation of the example 3CNF-SAT instance given in [COP06] using the available FPGA hardware. The aim of this was the familiarisation with the equipment and the development environment as well as the proof of the concept presented in Section 2.4. To achieve this an asynchronous as well as a synchronous version of the example was manually implemented and its behaviour investigated. After this the resulting circuits were unitised to prepare future automated experiments.

The example instance presented in [COP06] is the following satisfiable 3CNF-SAT formula containing four variables in four clauses (in fact all 4×4 3CNF-SAT instances are satisfiable as shown by the application in Appendix A.1).

$$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (B \vee C \vee D) \wedge (\overline{A} \vee \overline{C} \vee \overline{D})$$

A synchronous simulation of the circuit assuming that the rows in the circuit array proceed simultaneously showed the following behaviour: To begin, all values at the top of the circuit are initialised to $A = 0, B = 0, C = 0, D = 0$. As these first guesses propagate downwards the first row find the first term formula to be satisfied, so it passes the variable settings down unchanged. The second row proceeds in the same manner. The third row finds the formula unsatisfied, so it changes all the relevant variables, thus settings B, C and D to 1. The fourth row is satisfied.

The feedback now causes the new variable settings to flow through the system. Therefore the whole evaluation process starts again with the variable assignments $A = 0, B = 1, C = 1, D = 1$. The first row is satisfied, but the second fails so the variables A, B and C are flipped. The third and fourth rows are satisfied. The third downward pass initialised by the feedback now starts with the variable assignment $A = 1, B = 0, C = 0, D = 1$. With this assignment all four rows of the circuit array (or all four terms of the instance, respectively) evaluate to *true*. Therefore these values are sent back to the top of the circuit over and over again without changing the truth assignment. The system has therefore settled down to a solution to the problem which can easily be verified:

$$\begin{aligned}
&(false \vee true \vee true) \wedge (true \vee true \vee true) \wedge (false \vee false \vee true) \wedge (false \vee true \vee false)\\
&= true \wedge true \wedge true \wedge true\\
&= true
\end{aligned}$$

3.1.1 Overview over the laboratory equipment used during the experiments

All experiments described in this report were run on an Altera EP1C6Q240 device in combination with an EPCS1 configuration device. These devices were installed on a UP3-1C6 education board. This is a low-cost experimentation board designed for University and small-scale development projects. The board supports multiple on-board clocks with the base clock running at 14.318 MHz. Programming of the FPGA and data access to the on-chip memory are done using a JTAG or an Active Serial interface, respectively which is connected to the parallel port of a host computer (a standard off-the-shelf Pentium IV based Windows XP PC in this case). During all experiments the JTAG based interface was used as described in Section 3.2.4. In addition to these features the board supports several push button switches, a switch block, LEDs and a total of 74 pin headers for directly influencing or investigating signals used or produced by the chip respectively.

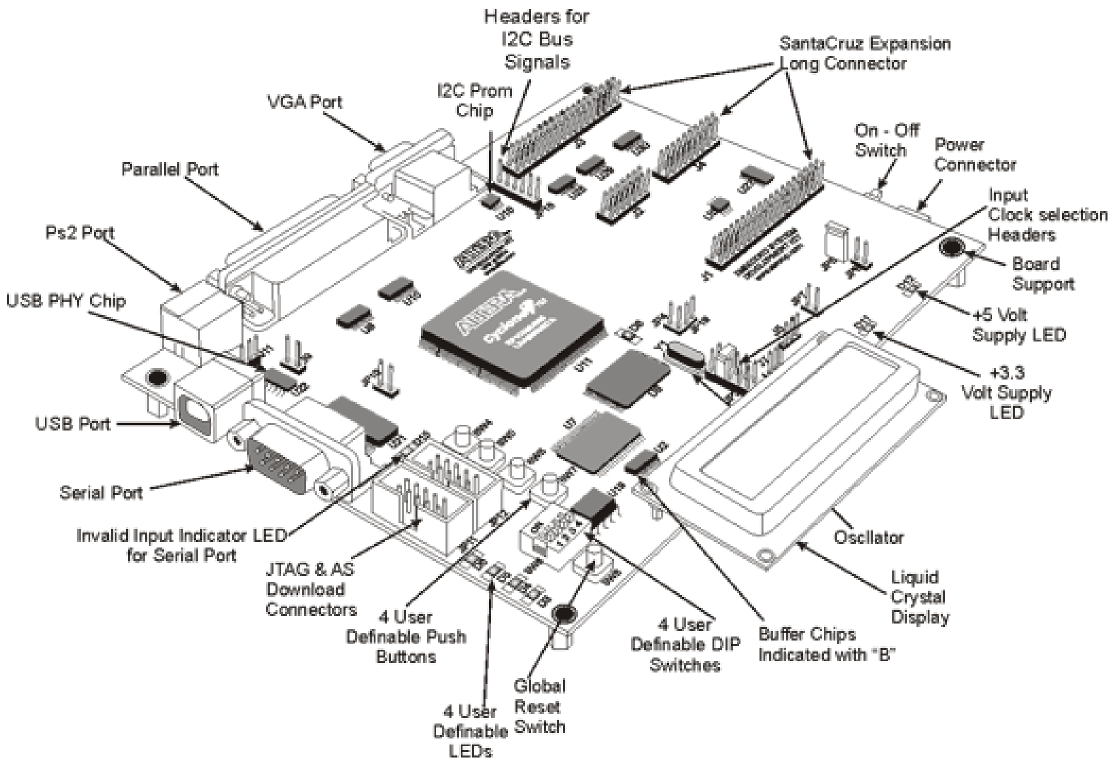
Headers for I2C Bus Signals
SantaCruz Expansion Long Connector
I2C Prom Chip
VGA Port
On - Off Switch
Power Connector
Parallel Port
Input Clock selection Headers
Ps2 Port
Board Support
USB PHY Chip
+5 Volt Supply LED
+3.3 Volt Supply LED
USB Port
Serial Port
Oscllator
Invalid Input Indicator LED for Serial Port
JTAG & AS Download Connectors
Liquid Crystal Display
4 User Definable Push Buttons
4 User Definable DIP Switches
Buffer Chips Indicated with "B"
Global Reset Switch
4 User Definable LEDs

Figure 3.1: SLS UP3-1C6 Cyclone FPGA development board

The employed FPGA provides a total amount of 5980 programmable logic elements amended by 92160 bits of on-chip SRAM divided into 20 memory blocks. It also contains two phase-locked loops for adjusting operation frequencies but these were not used during the experiments.

The 74 directly accessable pin headers are arranged in a standard-footprint called Santa Cruz long expansion headers. All 74 I/O pins directly conect to user I/O pins on the Cyclone FPGA device. The output logic level on the expansion prototype connector pins is 5 Volts. This makes it easy to investigate signals produced by the FPGA in real-time using an oscilloscope. During the manual experiments a digital 500 MHz oscilloscope of type Hewlett & Packard 54616C was used which allowed for a peak detect resolution of 1 ns. It supports optionally trigger based voltage and time measurement features on two distinct input channels.

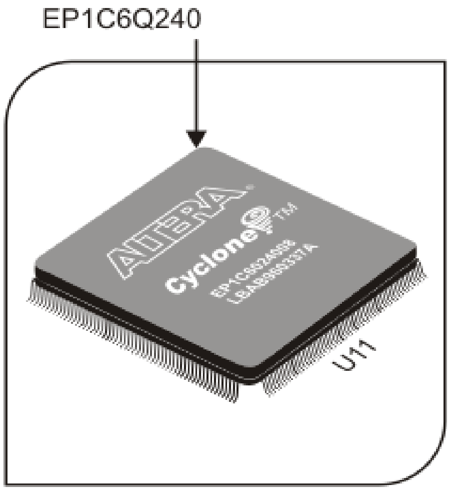

Figure 3.2: Altera Cyclone series EP1C6Q240 FPGA chip

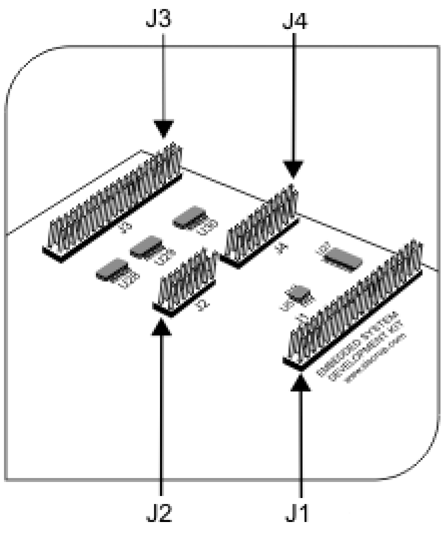

Figure 3.3: Santa Cruz long expansion headers

3.1.2 Synchronous circuit

The first circuit investigated was a synchronous straight-forward implementation of the example instance shown in Section 3.1. Figure 3.4 on page 18 shows a schematic diagram of the circuit. At this point the full implementation was done using a schematic design tool rather than a hardware description language. In addition to the main circuit a counter component from the Altera provided component library was included into the design to measure the number of clock cycles the circuit needs to stabilise. The clock signal was produced by the on-board base clock running at 14.318 MHz (this was kept for all other experiments as well). During the manual experiments the reset signal was produced by one of the push button switches present on the development board. The push button switches generate a logical 1 if they are in their normal state and a logical 1 if they are pressed. Unfortunatly the push button switches on the board proved to be not very well stabilised making it necessary to clear the counter with the reset signal (the FPGA device initialises all of its registers to 0).

The variable as well as the counter value signals where let to pin headers on the board where they could be investigated using the oscilloscope. Analysis of the signals produced by the chip showed that the circuit was behaving exactly as prognosed by the simulation presented in [COP06]. Therefore it produced a variable assignment of $A = 1, B = 0, C = 0, D = 1$ after 2 feedback steps.

3.1.3 Asynchronous circuit

After testing the synchronous design which worked as expected, the design was changed to the asynchronous one shown in Figure 3.5 on page 19. The rest of the setup of the experiment stayed unchanged. This circuit quickly found a satisfying truth assignment, too, but it was different from the one the synchronous circuit found (the synchronous circuit found A and D being set and B and C being cleared whereas the asynchronous circuit found only D being set and the other variables being cleared). Furthermore the stabilisation time of the circuit was so short that the clocked on-chip counter circuit was not able to measure it (it stopped counting after a single clock cycle in all cases).

Because of this, the stabilisation time was measured externally using the oscilloscope. The reset signal generated by the push button was used as trigger to center the oscilloscope image on the rising edge of it. A second signal indicating that a solution was found was superimposed and the timing differences measured. Table 3.1 on page 17 shows the time differences of the two signals reaching a level of 2 Volts as well as the difference to the first peak of the singals (the signal indicating that a solution was found tended to rise slower than the reset signal). Please note that these timings can only be considered as approximations because the maximum resolution of the oscilloscope used is 1 ns.

3.1.4 Hardening against compiler optimisations

After the results of the first two experiments were very promising the next step was to try a synchronous as well as an asynchronous implementation of an unsatisfiable 3CNF-SAT instance. If the concept is fully working the circuits must not come up with a solution for an unsatisfiable instance. For doing this an unsatisfiable 3×8 instance was created using diagonalisation:

$$(A \vee B \vee C) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee C) \wedge (\overline{A} \vee B \vee \overline{C}) \wedge (\overline{A} \vee \overline{B} \vee C) \wedge (\overline{A} \vee \overline{B} \vee \overline{C})$$

On the first attempt to implement this instance directly as circuit the resulting FPGA program just set the output signals to constant values. The reason for this is that the used FPGA compiler which is part of the Altera provided development environment contains a powerful optimisation engine probably featuring a complete software SAT solver. Because of this the compiler detected that the circuit is actually modelling constant output signals and removed most parts of the circuit.

Run	Δt_{rising}	$\Delta t_{firstpeak}$
1	1.52 ns	1.78 ns
2	2.04 ns	1.72 ns
3	1.88 ns	1.62 ns
4	2.04 ns	1.84 ns
5	1.98 ns	2.00 ns
6	1.68 ns	1.58 ns
7	1.28 ns	1.72 ns
8	1.48 ns	1.76 ns
9	1.38 ns	1.82 ns
10	1.42 ns	1.72 ns
11	1.94 ns	1.80 ns
12	2.02 ns	1.80 ns
13	1.72 ns	1.74 ns
14	0.88 ns	1.60 ns
15	1.38 ns	1.86 ns
16	1.64 ns	1.48 ns
17	1.32 ns	1.76 ns
18	1.78 ns	1.84 ns
19	1.20 ns	1.82 ns
20	1.54 ns	1.76 ns
Average	1.61 ns	1.75 ns
Variance	0.10 ns	0.01 ns
Standard deviation	0.32 ns	0.11 ns

Table 3.1: Timings of asynchronous circuit stabilisation

Since this satisfiability analysing optimisation engine could easily tamper future measurement results even on satisfiable instances it was necessary to effectively disable it. This was also the only way to test whether the circuits would come up with solutions for unsatisfiable instances. Since the compiler does not provide the option to entirely disable its optimisation engine it was necessary to circumvent it by the introduction of constant external signal the optimiser does not know.

Two external signals provided by push buttons on the development board were introduced into the circuit. These signals have a constant logical value of 1 as long as they are not pressed. Their complements were combined with the variable signals inside the circuit using XOR gates as shown in Figure 3.6 on page 20.

To further strengthen future circuit designs against the optimisation engine a third external signal was combined with the feedback signals produced by the term evaluation parts of the circuit. This way the optimisation engine of the compiler was no longer able to remove constant parts of the circuit.

After these hardening components were added to both circuits their behaviour was investigated using the oscilloscope. Both circuits produced a constant output signal regarding the satisfiability of the instance set to 0. The signals describing the truth assignment of the variables were floating around without settling down to a specific value. Therefore both circuits were behaving like prognosed providing a proof that the concepts proposed in [COP06] really word at least on very small instances. Therefore the next step in the project was to unitise the SAT circuitry, and to build a framework allowing for automated generation and even automated execution of experiments on the FPGA.

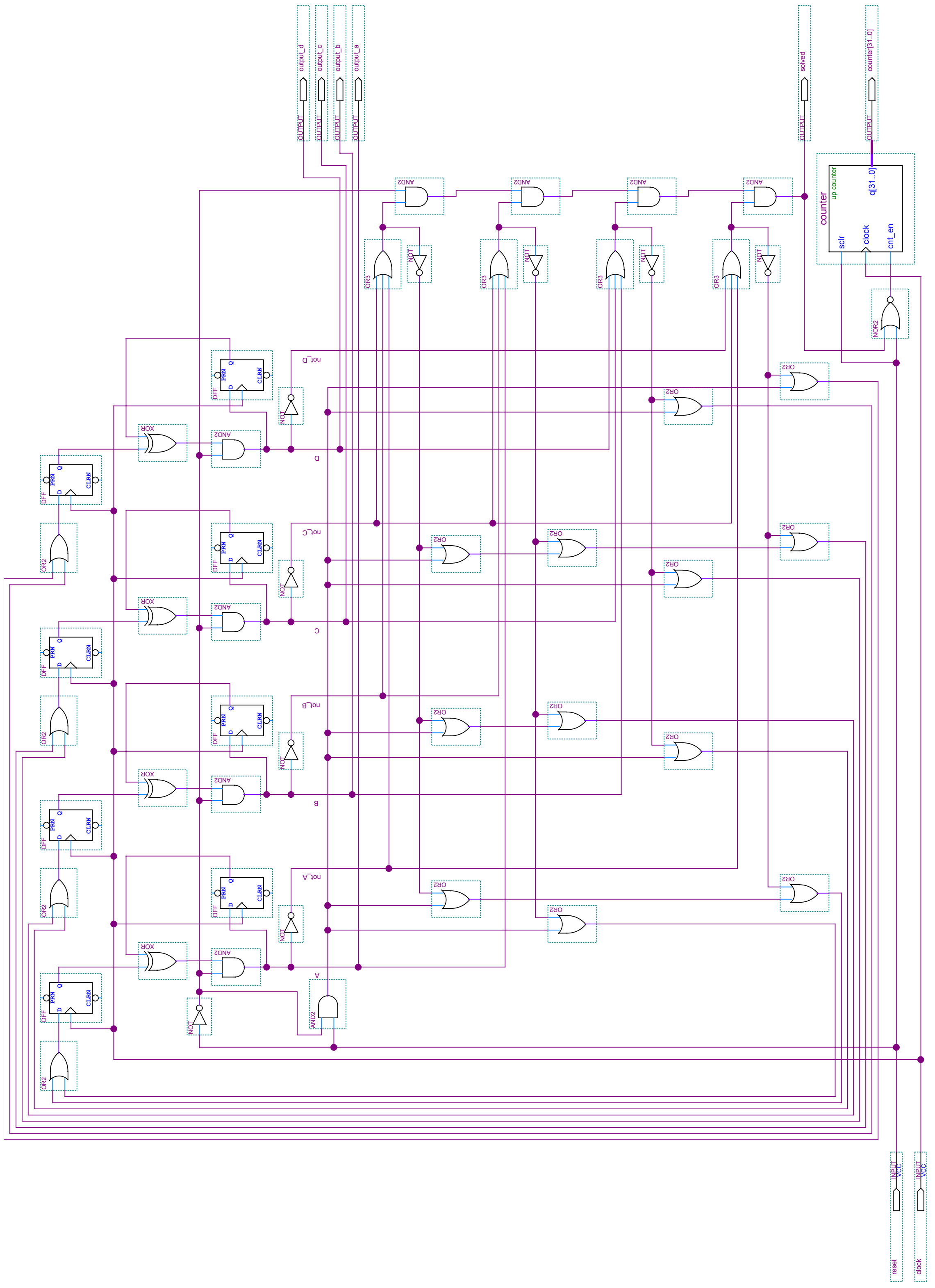

Figure 3.4: Synchronous circuit implementing 4x4 3CNF-SAT instance

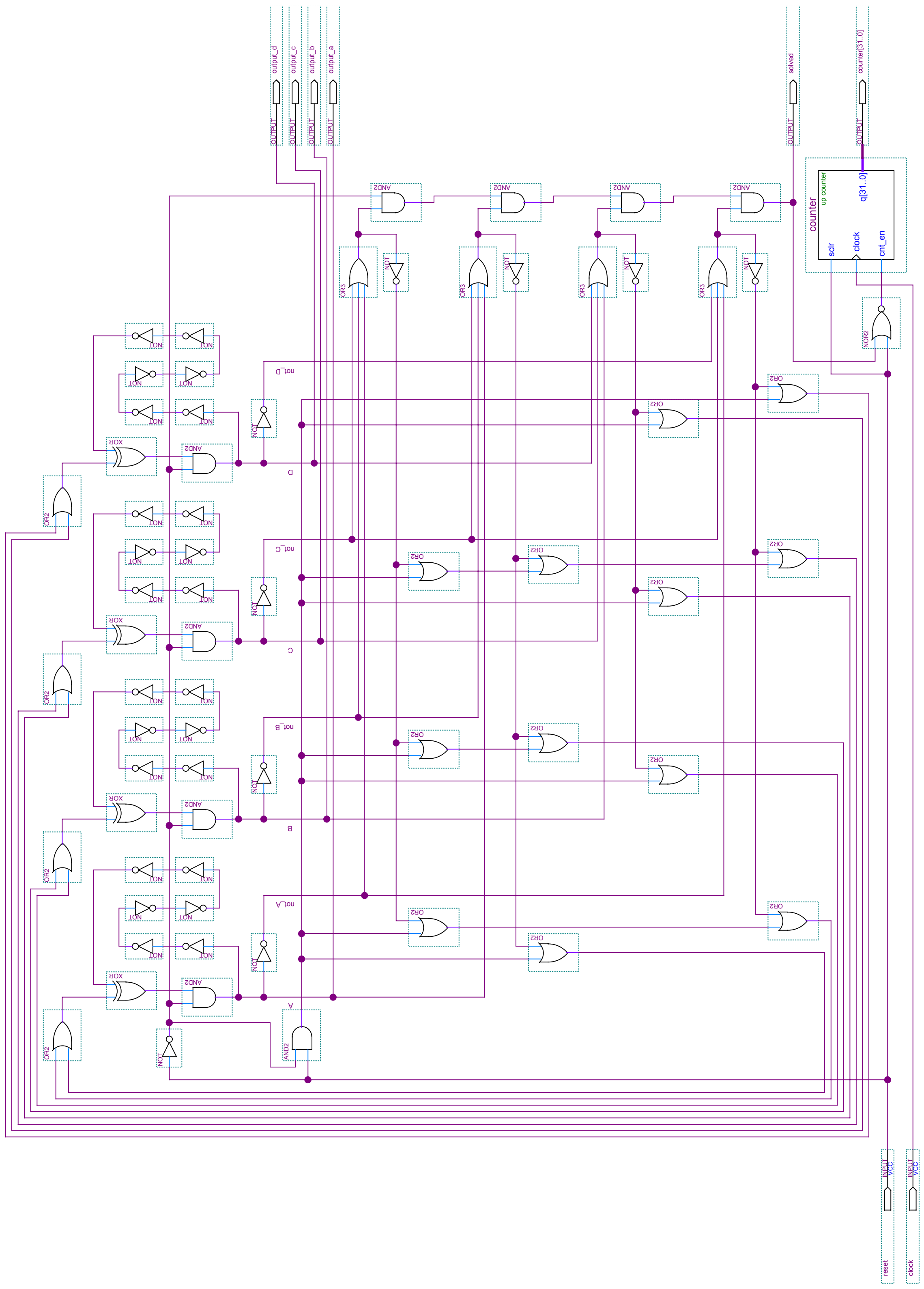

Figure 3.5: Asynchronous circuit implementing 4x4 3CNF-SAT instance

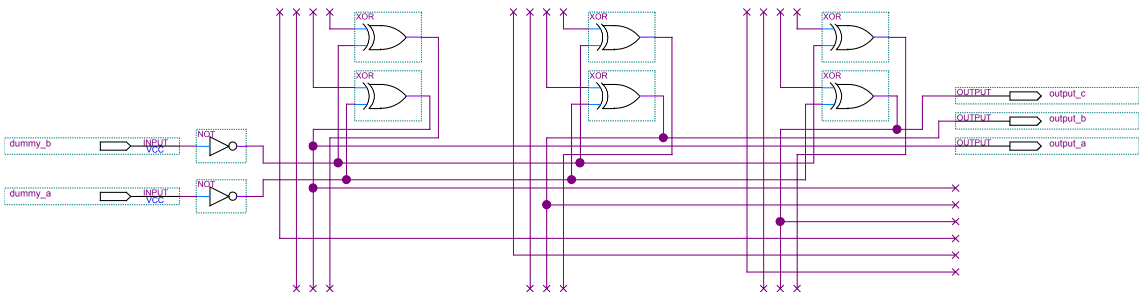

Figure 3.6: Hardening of variable signals against compiler optimisations

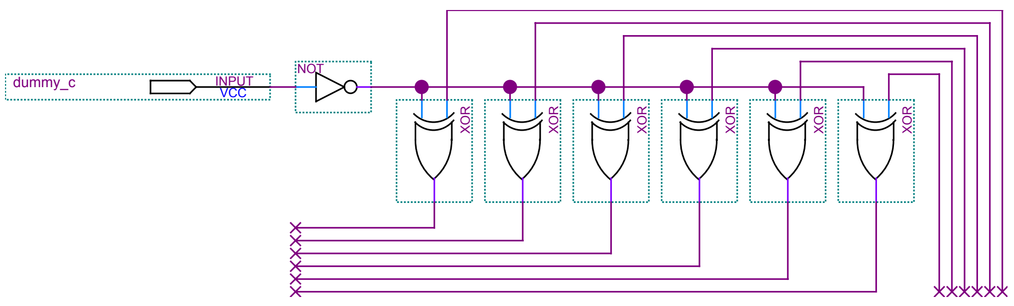

Figure 3.7: Hardening of feedback signals against compiler optimisations

3.2 Modularisation and automation

3.2.1 Unitised SAT circuitry

After the manually created test cases showed a very promising behaviour the decision was taken to prepare the experimental setup for the automated generation and execution of test cases and the underlying circuits, repsectively. The first step in this process was the expression of the different parts of the circuit using a hardware definition language (all previous experiments were set up using a schematic design tool). The Altera provided development environment supports three different languages in different versions each. Besides Altera's own AHDL language, the industry standard languages VHDL and Verilog are supported. VHDL was chosen for this project because of its good support by the Altera software, its modular structure and its compatibility to other design tools making reusing and simulating the created components using non-Altera provided tools possible. It is also well suited for automated code generation.

The SAT circuitry itself was divided into three modules. On the one hand the term evaluator and variable source modules drafted in Section 2.4 were implemented in stand-alone VHDL modules shown in Section 3.2.3 to be easily exchangable in different experiments. This makes these modules also independant from the actually implemented SAT instance. On the other hand the actual SAT instances are implemented by modules combining term evaluators and variable sources (and in some experiments other components as well). These modules are automatically generated by software specifically for each type of experiment as shown in Section 3.2.5.

This design makes the SAT core independant from the measurement circuitry necessary for unattended testing and result collection as shown in Section 3.2.2.

3.2.2 Support circuitry for automated measurements

Since the different experiments on the SAT problems required a large number of different test cases covering an even larger number of single test instances it was not an option to execute all tests manually. Instead the generation of the circuit definitions, their compilation, the programming of the FPGA and the retrieval of the measurement data had to be automated to be executable in an unattended way.

To achieve this goal all measurements had to be done by the circuitry implemented by the FPGA and the result data had to be transferred to the host computer for storage and later analysis. After looking into different possibilities of communication between the host computer and the FPGA the decision was taken to use the provided JTAG interface (see Section 3.2.4) to read the result data back to the host computer. To make this possible the result data had to be stored either directly in logic elements on the chip (using their built-in flip-flops) or in the 4096 bit memory blocks provided on the device. The latter option was selected because it provides much more flexibility regarding the collected data and also requires much less chip space.

The memory blocks provided by the FPGA are accessible in VHDL code through an Altera provided pseudo-component which acts as a wrapper around one or more memory blocks. This pseudo-component also optionally triggers the generation of JTAG interface structures allowing the memory block contents to be read (and optionally even to be written) using the JTAG interface connecting the FPGA development board to the host computer.

Since the memory block component supports only writing data at one (or optionally two) distinct addresses at a time a memory controller had to be implemented which collects the measurement data from other components of the circuit, buffers it, and writes it in a defined structure to the memory block. The actual data written varies between the experiments but most experiments write at least the number of clock cycles the circuit required to stabilise on the result (if not interrupted by a time-out), a flag whether a solution was found before the time-out occurred and the final truth assignment when the solution was found or the time-out occurred. Most experiments also output the number of variables participating in the analysed instance or even a computed checksum for error detection and debug purposes.

To be able to collect these types of data a couple of other components had to be implemented. Delay and time-out controllers were implemented to start the experiment at a specific point in time and to abort it if a solution could not be found after a preset number of clock cycles. A performance counter component uses the signals provided by these components to calculate the exact running time of the experiments in clock cycles. Figure 3.8 on page 22 shows a sketch of the basic layout of the support circuitry. Details about the different experiments are documented in Chapter 4.

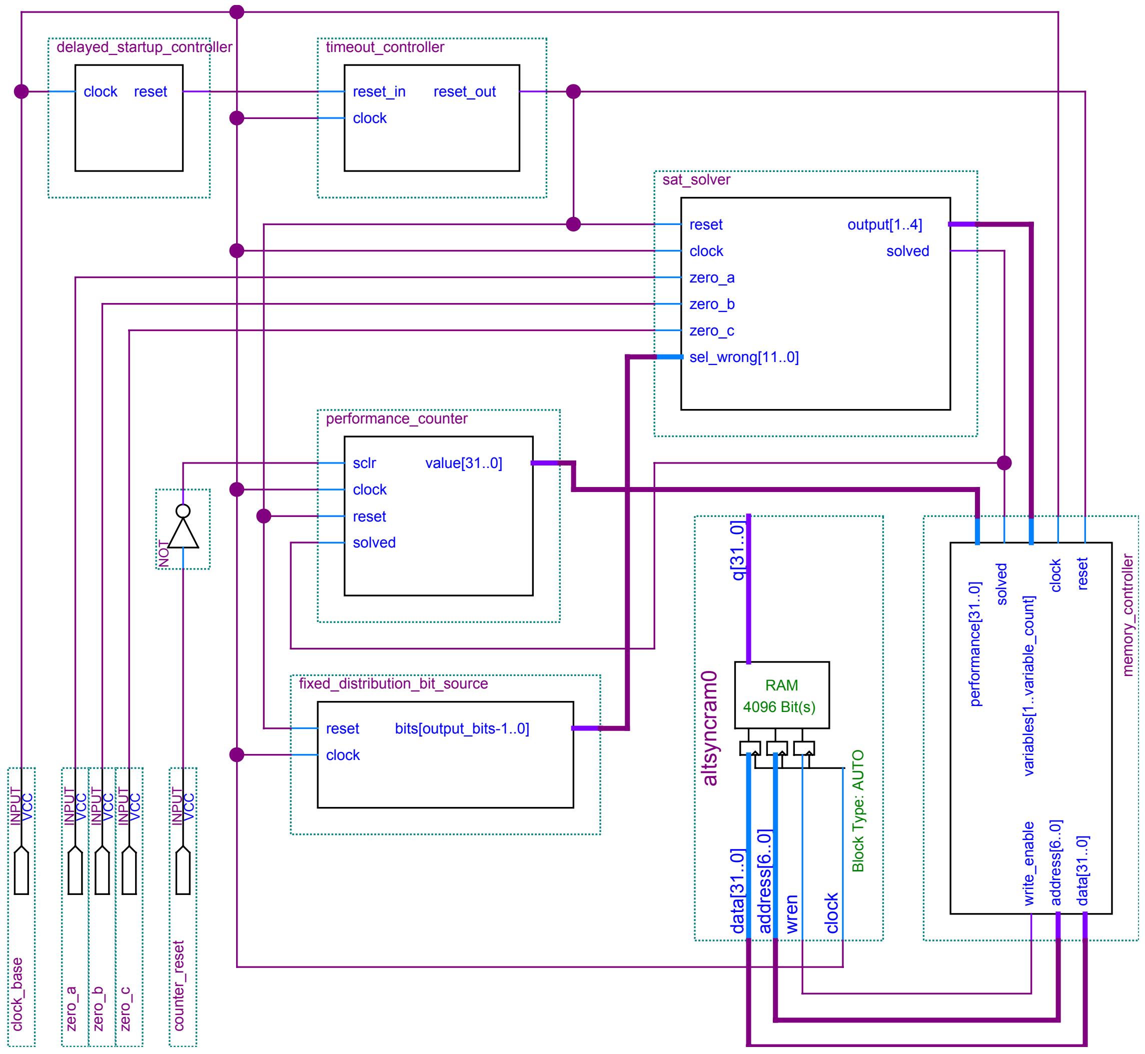

Figure 3.8: Example support circuitry layout for automated test case execution

Some experiments required the implementation of other more experiment-specific modules as well (e.g. randomisation components as shown in Figure 3.8 on page 22). During the development of all components the reusability of the created components through multiple experiments was emphasised. Because of this many components are implemented as VHDL generics providing module templates for different types of experiments and instances (e.g. the memory controller is able to handle different numbers of variable value signals using a VHDL generic).

The delay controller is needed because the circuit basically starts "somehow" after the programming of the FPGA finished. This component ensures that a clear reset signal is emitted and that this reset signal is hold long enough for all components to initialise. Note that all registeres of the FPGA are initialised to 0 when starting up.

3.2.3 Overview over the VHDL library used during the experiments

The following paragraphs give an overview over the VHDL module library created during the project. Please note that the VHDL modules presendet in this section were not created for a single experiment but for a large number of experiments over a time of several months. This section is mainly intended as a reference to facilitate understanding the source codes and diagrams created during the project and to make reusing the created components in future projects as easy as possible.

It should be pointed out up front that the semantics of the reset signals used by many components changed during the project. The first components developed during the project (and also components derived from them) expect the reset signal to be set to a logical 0 if being in reset state and to a logical 1 if being in operational state. This assignment was selected because in the early experiments the reset signal was manually generated by pressing one of the push button switches on the development board. These switches generate a logical 0 signal if pressed and a logical 1 signal if released. Since this assignment is not very intuitive the assignment was swapped later in the course of the project. Because of this there are components expecting a reset signal using the first way and others which expect the reset signal using the second way of assignment. Please pay attention to this fact if reusing and mixing the created components in future projects.

If not otherwise stated, all synchronous modules use registered inputs. The outputs of all modules are unregistered. If necessary, the produced values have to be stored by subsequent modules. The latency of all modules is exactly one clock cycle unless otherwise stated in the module description.

Term evaluators

The term evaluator modules are implemented as VHDL generics supporting an arbitrary number of input signals. Each each signal corresponds to a variable value or its complement, respectively. Figure 3.9 on page 23 shows block diagrams of the available term evaluators. Implementation details are shown by the module sources in Appendix B.1.

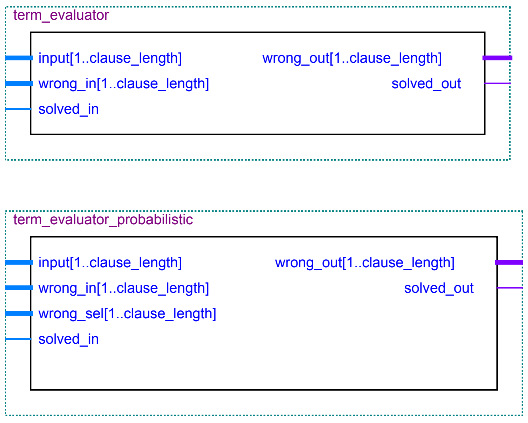
term_evaluator
input[1..clause_length]
wrong_in[1..clause_length]
solved_in
wrong_out[1..clause_length]
solved_out
term_evaluator_probabilistic
input[1..clause_length]
wrong_in[1..clause_length]
wrong_sel[1..clause_length]
solved_in
wrong_out[1..clause_length]
solved_out

Figure 3.9: Block diagrams of term evaluator modules

Basic term evaluator The basic term evaluator module is a straight-forward implementation of the term evaluator module draft shown in Section 2.4. The input signals are combined using an

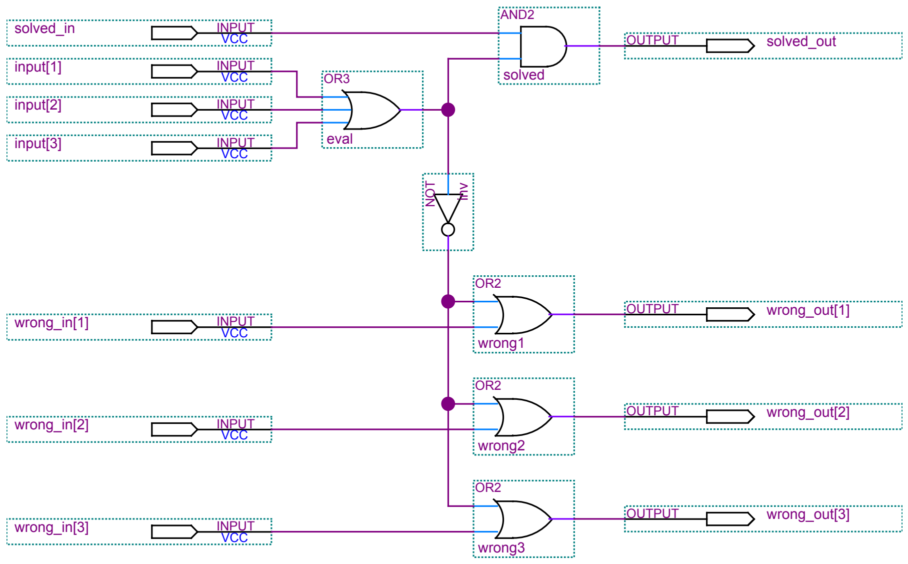

Figure 3.10: Schematic diagram of basic term evaluator module

OR function. If the result of the disjunction is *false*, all outgoing *wrong* signals are set to *true* and the outgoing *solved* signal is set to *false*. Otherwise the incoming *wrong* signals and the incoming *solved* signal are passed through. The source code of this module if available in Appendix B.1.1.

Input port	**Type**	**Required**	**Comments**
`input[]`	STD_LOGIC_VECTOR	Yes	Current truth assignment of the participating variables or their complements, respectively
`wrong_in[]`	STD_LOGIC_VECTOR	Yes	Participation status signals provided by previous evaluator modules
`solved_in`	STD_LOGIC	Yes	Solution status signal provided by previous evaluator modules
Output port	**Type**	**Required**	**Comments**
`wrong_out[]`	STD_LOGIC_VECTOR	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`solved_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
Parameter	**Type**	**Required**	**Comments**
`clause_length`	Integer	No	Number of variables in this clause (default is 3)

Table 3.2: Basic term evaluator interface

Probabilistic term evaluator The probabilistic term evaluator module behaves exaclty like the basic term evaluator module with the only difference that in the case of the clause evaluating to $false$, a *wrong* signal is only set to *true* if the corresponding *select* signal is set. Otherwise the *wrong* signal is passed through just as if the clause would have been satisfied. The source code of this module is available in Appendix B.1.2.

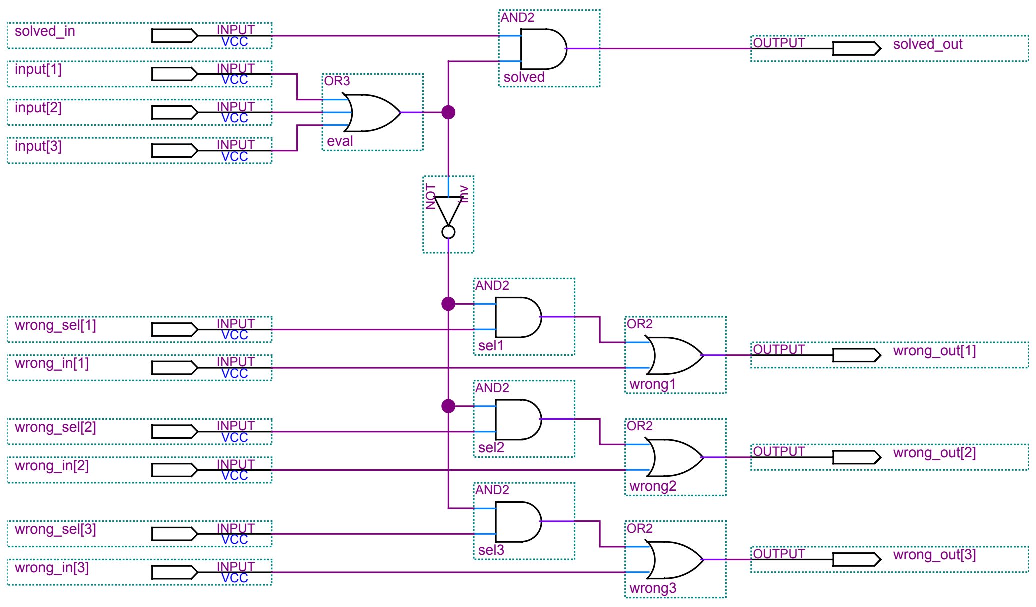

Figure 3.11: Schematic diagram of probabilistic term evaluator module

Input port	**Type**	**Required**	**Comments**
`input[]`	STD_LOGIC_VECTOR	Yes	Current truth assignment of the participating variables or their complements, respectively
`wrong_in[]`	STD_LOGIC_VECTOR	Yes	Participation status signals provided by previous evaluator modules
`wrong_sel[]`	STD_LOGIC_VECTOR	Yes	If a signal of this vector is set to 0 the corresponding *wrong* signal is just passed through regardless of the evaluation result of the clause
`solved_in`	STD_LOGIC	Yes	Solution status signal provided by previous evaluator modules
Output port	**Type**	**Required**	**Comments**
`wrong_out[]`	STD_LOGIC_VECTOR	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`solved_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
Parameter	**Type**	**Required**	**Comments**
`clause_length`	Integer	No	Number of variables in this clause (default is 3)

Table 3.3: Probabilistic term evaluator interface

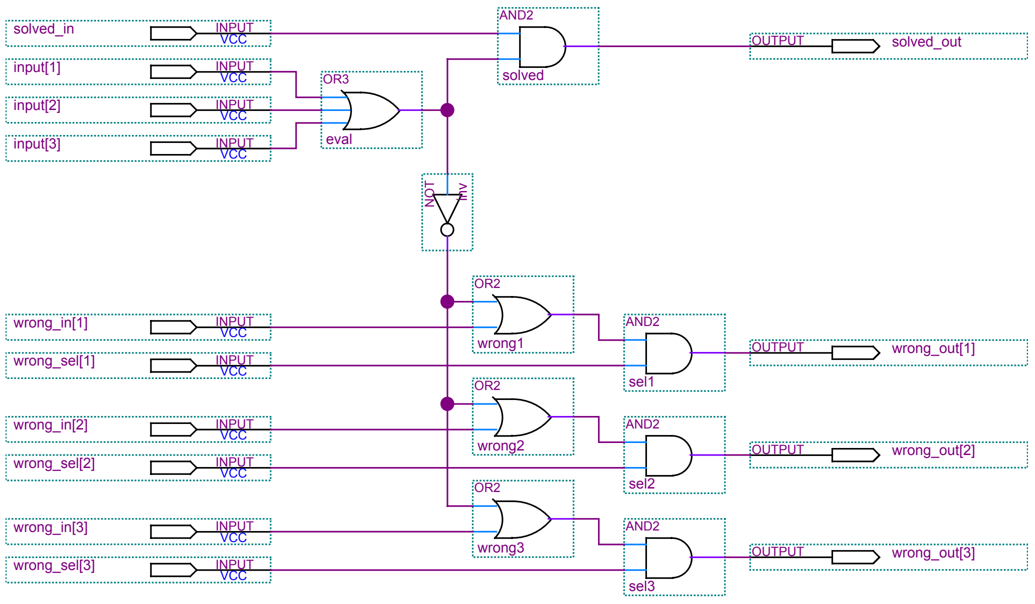

Figure 3.12: Schematic diagram of erroneous probabilistic term evaluator module

Probabilistic term evaluator (buggy) This variant of the term evaluator module is just included for completeness. It was accidently used in some experiments but contains a bug rendering the measurement results useless. If a specific signal in the *select* signal vector is set to 1 with a probability of p, the total probability of a variable being announced for toggling in the correct module is np with n being the number of unsatisfied clauses the variable is participating in. With this buggy variant of the term evaluator module the probability is roughly p^n. The interface of the module is identical to the non-buggy variant. The source code of this module is available in Appendix B.1.3.

Variable sources

The variable source modules heavily differ because one of the most important parts of the research regarding the SAT circuitry focused on different variable source types. The library contains synchronous as well as asynchronous variable sources modules which were used in many different experimental contexts. See Chapter 4 for details regarding the different experiments. Some variable sources are implemented as VHDL generics supporting multiple configurations of the same component template. Figure 3.9 on page 23 shows block diagrams of the available variable sources. Implementation details are shown by the module sources in Appendix B.2.

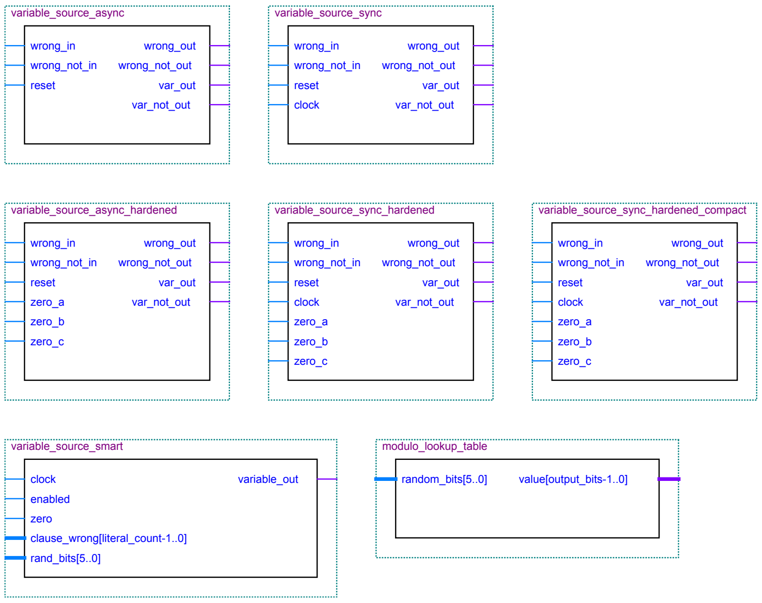

Figure 3.13: Block diagrams of variable source modules

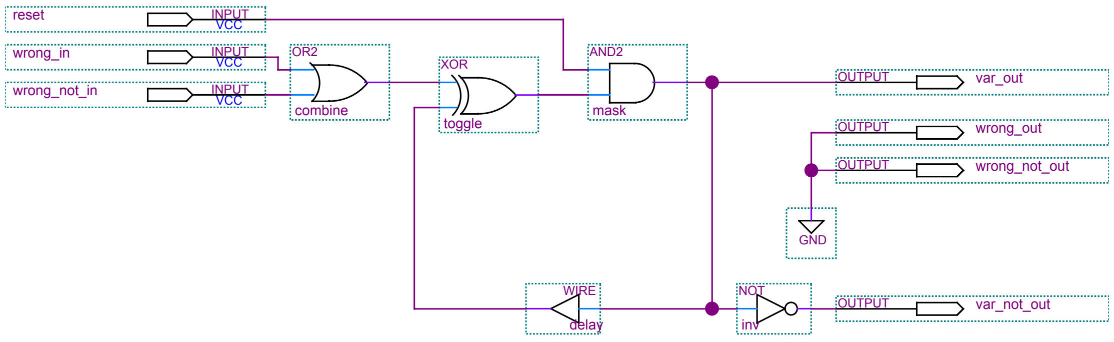

Figure 3.14: Schematic diagram of basic asynchronous variable source module

Basic asynchronous variable source This is the basic asynchronous variable source module used in early experiments before the idea of having asynchronous variable sources was discarded. The toggling of a variable is delayed by a configurable number of delay gates which are implemented as AND gates combining the feedback value with $true$. Unfortunately it could not be verified what the compiler optimisation engine does with this implementation so it is possible that this way of delaying the toggling of variables is completely ineffective. This was not further investigated since the asynchronous circuit variant showed very uncontrollable behaviour evan on smaller SAT instances when watched using the oscilloscope. Besides this, the component is a straight-forward implementation of the asynchronous variable source module drafted in Section 2.4. The source code of this module is available in Appendix B.2.1.

Input port	**Type**	**Required**	**Comments**
`wrong_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the variable participated in an unsatisfied clause
`wrong_not_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the complement of the variable participated in an unsatisfied clause
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 0 if in reset state - in this case the feedback loop is cleared and the variable initialised to 0
Output port	**Type**	**Required**	**Comments**
`wrong_out`	STD_LOGIC	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`wrong_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
Parameter	**Type**	**Required**	**Comments**
`delay_gates`	Natural	No	Number of delay gates used to delay the feedback signal (default is 0)

Table 3.4: Basic asynchronous variable source interface

Asynchronous variable source hardened against compiler optimisations As described in section Section 3.1.4, several parts of the SAT circuitry require special hardening against compiler optimisations. This variant of the variable source module behaves exactly like the basic asynchronous variant with the exception that combines three externally provided signals with the internal signals of the module using a logical XOR function. The source code of this module is available in Appendix B.2.2.

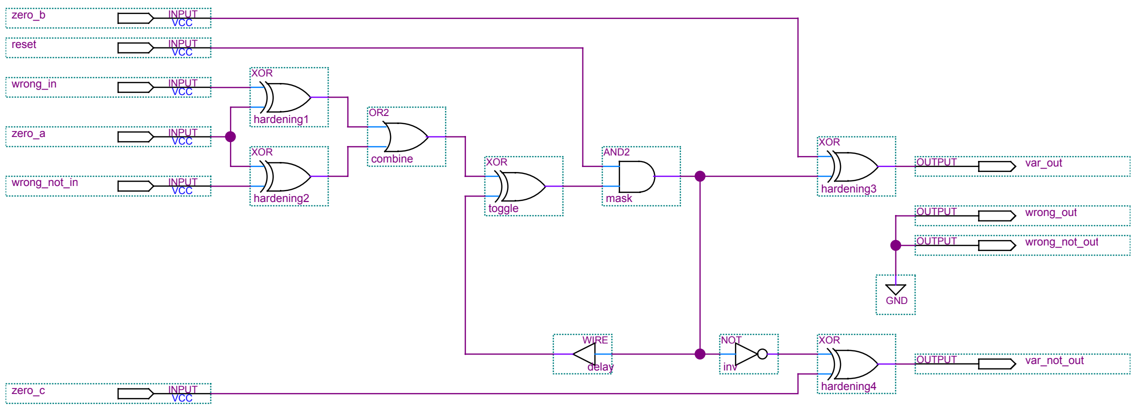

Figure 3.15: Schematic diagram of basic asynchronous variable source module

Input port	**Type**	**Required**	**Comments**
`wrong_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the variable participated in an unsatisfied clause
`wrong_not_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the complement of the variable participated in an unsatisfied clause
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 0 if in reset state - in this case the feedback loop is cleared and the variable initialised to 0
`zero_a`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
`zero_b`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
`zero_c`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
Output port	**Type**	**Required**	**Comments**
`wrong_out`	STD_LOGIC	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`wrong_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
Parameter	**Type**	**Required**	**Comments**
`delay_gates`	Natural	No	Number of delay gates used to delay the feedback signal (default is 0)

Table 3.5: Hardened asynchronous variable source interface

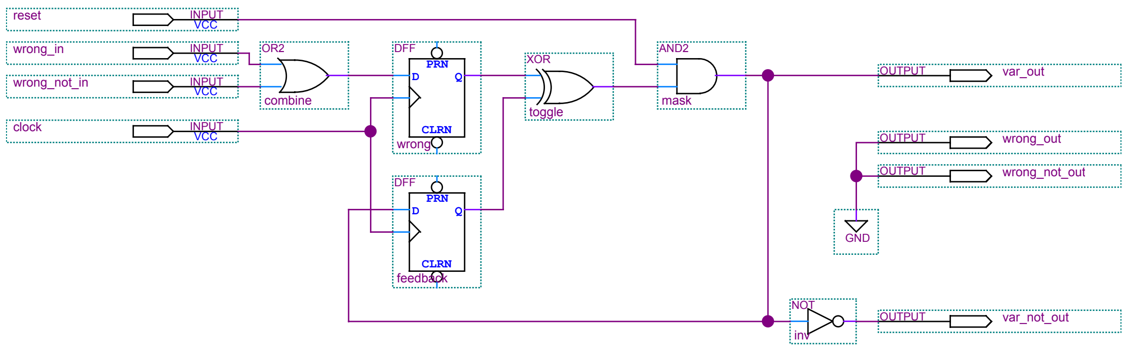

Figure 3.16: Schematic diagram of basic synchronous variable source module

Basic synchronous variable source This is the basic synchronous variable source module used in early experiments. In contrast to the asynchronous variable source modules, the toggling of a variable only occurs on a rising edge of the *clock* signal. The component is a straight-forward implementation of the synchronous variable source module drafted in Section 2.4. The source code of this module is available in Appendix B.2.3.

Input port	**Type**	**Required**	**Comments**
`wrong_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the variable participated in an unsatisfied clause
`wrong_not_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the complement of the variable participated in an unsatisfied clause
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 0 if in reset state - in this case the feedback loop is cleared and the variable initialised to 0
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`wrong_out`	STD_LOGIC	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`wrong_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)

Table 3.6: Basic synchronous variable source interface

Synchronous variable source hardened against compiler optimisations This synchronous variable source module is hardened against compiler optimisations analogous to the hardened asynchronous variable source module. Despite this, the behaviour of the module is identical the the basic synchronous variable source module. The source code of this module is available in Appendix B.2.4.

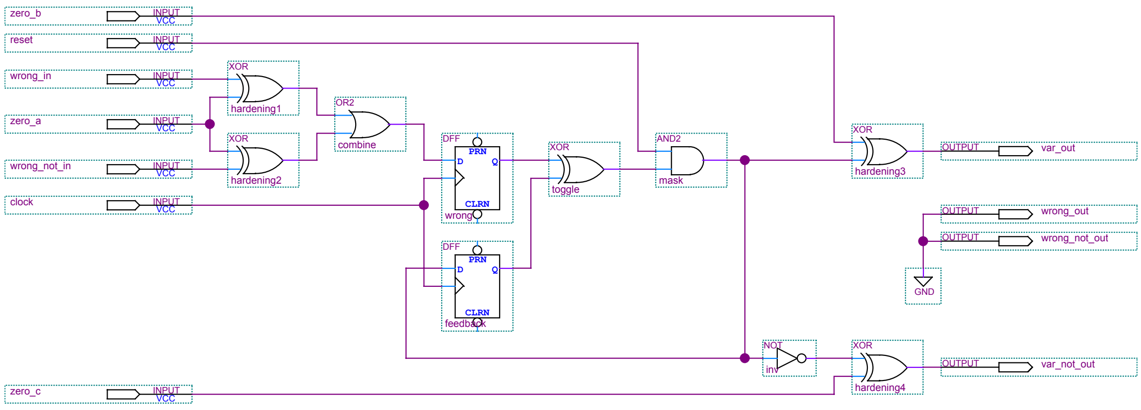

Figure 3.17: Schematic diagram of hardened synchronous variable source module

Input port	**Type**	**Required**	**Comments**
`wrong_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the variable participated in an unsatisfied clause
`wrong_not_in`	STD_LOGIC	Yes	A signal value of 1 indicates that the complement of the variable participated in an unsatisfied clause
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 0 if in reset state - in this case the feedback loop is cleared and the variable initialised to 0
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`zero_a`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
`zero_b`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
`zero_c`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
Output port	**Type**	**Required**	**Comments**
`wrong_out`	STD_LOGIC	Yes	Signal vector signalling that variables participated in wrong clauses (0 means no participation in wrong clause, 1 means participation in at least one wrong clause)
`wrong_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)
`var_not_out`	STD_LOGIC	Yes	Updated signal signalling solution state (0 means solution not found, 1 means possible solution so far)

Table 3.7: Hardened synchronous variable source interface

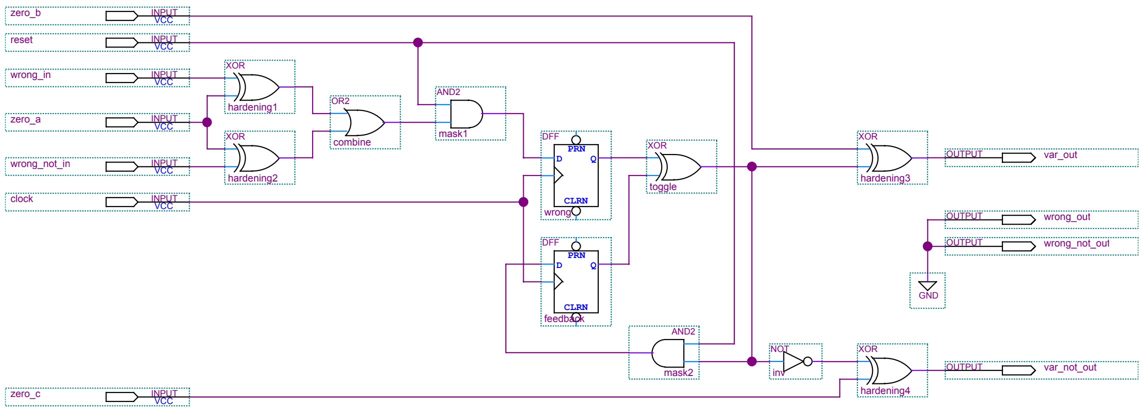

Figure 3.18: Schematic diagram of hardened compact synchronous variable source module

Synchronous variable source hardened against compiler optimisations (compact) This is a slightly compacted version of the hardened synchronous variable source module. If integrated into the SAT circuitry the compiler is able to optimise the solver circuit more compactly if this module is used compared to the previous version of the module. Despite this, the behaviour and the interface of the module are identical the the hardened synchronous variable source module. The source code of this module is available in Appendix B.2.5.

Locally probability driven variable source This synchronous variable source module was used in some experiments regarding locally probability driven SAT solvers. The basic idea behind this is explained in Section 4.4. This module was only used in a few experiments because of its high space requirements which make it hard to build an universal ASIC using this kind of variable source. If using this variable source the probability driven state evaluation is moved from the term evaluators into the variable sources. This means that this module must not be used in combination with the probabilistic term evaluator module. If a variable participates in m clauses with n of them being unsatisfied the probability of the corresponding variable being toggled is roughly ${}^{n}/_{m}$. The source code of this module is available in Appendix B.2.6.

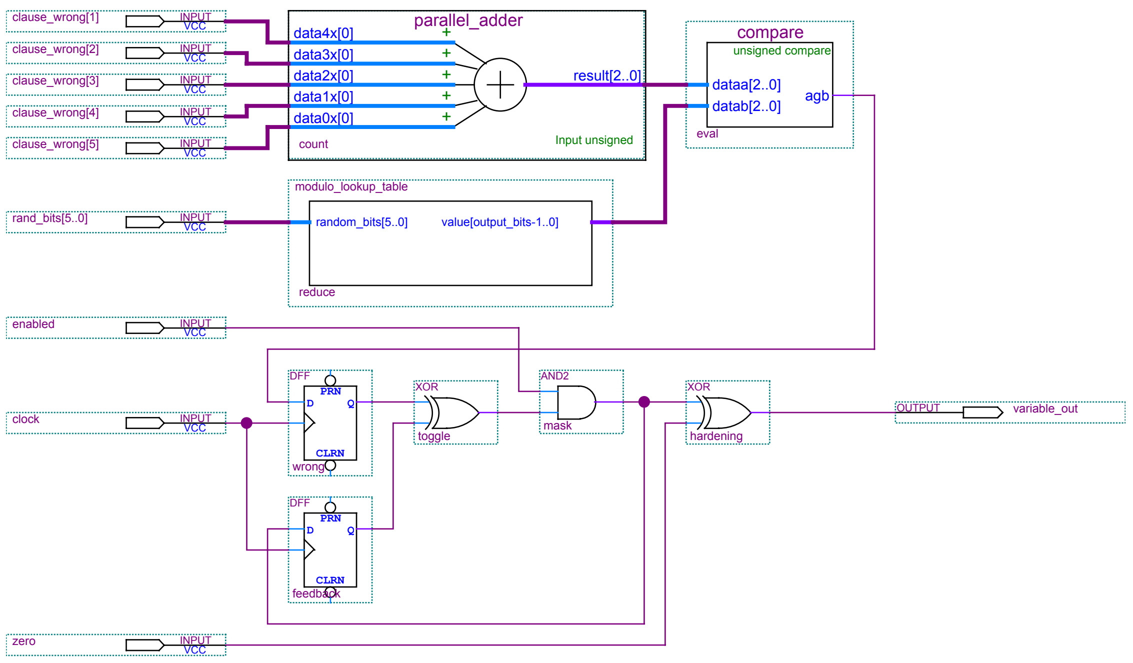

Figure 3.19: Schematic diagram of experimental locally probability driven variable source module (example for a variable participating in 5 clauses)

Input port	**Type**	**Required**	**Comments**
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`enabled`	STD_LOGIC	Yes	The module expects the *enabled* signal being 0 if in reset state - in this case the feedback loop is cleared and the variable initialised to 0
`zero`	STD_LOGIC	Yes	The module expects this signal to be constantly set to 0
`clause_wrong[]`	STD_LOGIC	Yes	A signal value of 1 indicates that the corresponding clause, in which the variable or its complement is participating, is unsatisfied
`rand_bits[]`	STD_LOGIC	Yes	The module expects this signal vector to consist of (pseudo-)randomly generated bits and to contain one bit for each clause this variable participates in
Output port	**Type**	**Required**	**Comments**
`variable_out`	STD_LOGIC	Yes	Updated truth assignment of the corresponding variable
Parameter	**Type**	**Required**	**Comments**
`literal_count`	Integer	Yes	Number of clauses the correspondign variable or its complement participate in
`count_bits`	Integer	Yes	Ceiled binary logarithm of the number of relevant clauses

Table 3.8: Experimental locally probability driven variable source interface

Fast modulo computation for smart variable source This module is used by the experimental locally probablity driven variable source module. It provides a fast combinatorial lookup table for computing the remainder of a natural number passed as bit vector and a constant chosen at compile time. The (shortened) source code of this module is available in Appendix B.2.7.

Input port	**Type**	**Required**	**Comments**
`random_bits[]`	STD_LOGIC	Yes	Signal vector describing a 6-bit wide natural number
Output port	**Type**	**Required**	**Comments**
`value[]`	STD_LOGIC	Yes	Signal vector describing the number described by the input signal vector modulo the output range
Parameter	**Type**	**Required**	**Comments**
`output_range`	Integer	Yes	Modulus (valid numbers are from 1 to 32)
`output_bits`	Integer	Yes	Length of the output signal vector (valid numbers are from 1 to 5)

Table 3.9: Fast modulo computation interface

Fixed distribution bit sources

As early experiments showed that some form of probability driven architecture is necessary to reach good results using the highly parallelised SAT solvers investigated during this project, a number of randomisation components were developed. The fixed distribution bit source modules transform one or multiple streams of (pseudo-)randomly generated bits having a theoretical probability of 0.5 of a bit being set to 1 to a single bit stream in which the probability of a bit being 1 is an arbitrary constant between 0 and 1 preset during compile time. The bit source modules also provide long shift registers serving selector signals to the probabilistic term evaluator modules described earlier. The bit source modules are implemented as VHDL generics supporting an arbitrary number of output bits gated to approximate a given probability distribution. Figure 3.9 on page 23 shows block diagrams of the available bit sources. Implementation details are shown by the module sources in Appendix B.3.

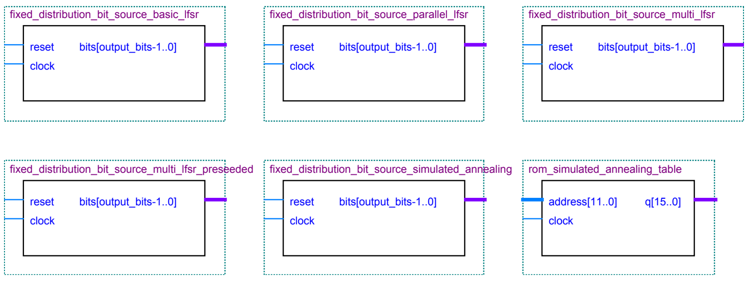
fixed_distribution_bit_source_basic_lfsr
reset
clock
bits[output_bits-1..0]
fixed_distribution_bit_source_parallel_lfsr
reset
clock
bits[output_bits-1..0]
fixed_distribution_bit_source_multi_lfsr
reset
clock
bits[output_bits-1..0]
fixed_distribution_bit_source_multi_lfsr_preseeded
reset
clock
bits[output_bits-1..0]
fixed_distribution_bit_source_simulated_annealing
reset
clock
bits[output_bits-1..0]
rom_simulated_annealing_table
address[11..0]
clock
q[15..0]

Figure 3.20: Block diagrams of bit source modules

Bit source using single bit LFSR This bit source module uses a single linear feedback shift register moving by a single bit each clock cycle. The highest ten bits of the LFSR are gated to produce a preset probability distribution. Since each bit produced by the LFSR influences 10 bits running through the bit source register this basic bit source module proved to be not very well

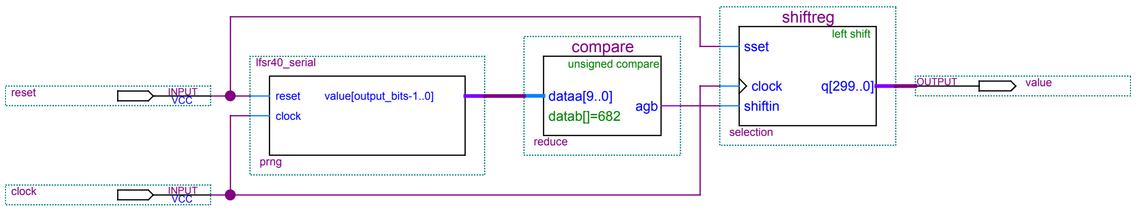

Figure 3.21: Schematic diagram of bit source using single bit LFSR

suited for proper randomisation of the SAT solver circuitry because the bits running through the selection register are closely statiscally dependant from at least nine other bits each. The effects of proper randomisation of the solver engine are discussed in Section 5.2.4. The source code of this module is available in Appendix B.3.1.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state - in this case the LFSR as well as the selection register are cleared
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`bits[]`	STD_LOGIC	Yes	Signal vector representing bits having preset probability distribution
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the selection register
`probability_factor`	Integer	Yes	$\lfloor 2^{10} \cdot (1-p) + 0.5 \rfloor$ with p being the probability of a bit in the selection register being 1

Table 3.10: Interface of bit source using single bit LFSR

Bit source using parallelised LFSR This module is an improved version of the previous single bit. It still uses a single LFSR but this LFSR is implemented in a parallelised manner to generate 10 fresh bits every clock cycle. This way the statistical dependancy of the bits running through the selection register is heavily reduced. However, the statistical properties of this bit source module are still not good enough for representative experiments with the SAT solver engine. Despite this, the behaviour as well as the interface of this module are identical to the previously described single bit variant of the bit source. The source code of this module is available in Appendix B.3.2.

Bit source using parallelised LFSR array This module is the finally used bit source module implementing an array of 10 parallelised LFSRs. Each of these LFSRs produces 10 fresh bits every clock cycle which are reduced to a single bit fulfilling the preset probability distribution. This way 10 fresh bits are sent through the selection registers letting it move with the tenfold speed compared to the previous bit source modules. The bits running through the selection register are still subject to statistical dependancies but these proved to be small enough to produce reliable measurement results. Unfortunately the employed array of 10 equally long LFSRs (each having alnegth of 40 bits) seems to degrade the period of the LFSR. This became a problem when running single instance test cases as described in Section 4.3.4. Despite this, the behaviour as well as the interface of this module are identical to the previously described variants of the bit source. The source code of this module is available in Appendix B.3.3.

Bit source using parallelised LFSR array with shift register preseeding The previously described bit sources all have the problem that the selection register is initialised to all bits being set to 0. This way it in the worst case it can take several hundred clock cycles before the first variables are toggled. This module is a slightly modified variant of the previous module employing an array of LFSRs. In addition to this improvement, this module preseeds the selection register with a preset seed whose proper probability distribution has to be ensured by the developer (source code to generate such a bit sequence is included in Appendix A.2). The selection register is set to the preset seed whenever the *reset* signal is set to 1. Please not that the length of the selection register is hardcoded in the current version of the module. If the module is to be used in future projects this parts should be converted to a VHDL generic. The source code of this module is available in Appendix B.3.4.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state - in this case the LFSR as well as the selection register are cleared
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`bits[]`	STD_LOGIC	Yes	Signal vector representing bits having preset probability distribution
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the selection register
`probability_factor`	Integer	Yes	$\lfloor 2^{10} \cdot (1-p) + 0.5 \rfloor$ with p being the probability of a bit in the selection register being 1
`seed[]`	STD_LOGIC	Yes	Preset seed to be loaded into the selection register if the *reset* signal is set to 1 (currently this has to be of length 1110)

Table 3.11: Interface of bit source using parallelised LFSR array and preseeding

Bit source supporting dynamic probabilities using simulated annealing This module is a modified variant of the bit source module using a parallelised LFSR array. It currently does not support preseeding but instead includes the possibility to dynamically alter the probability of a bit being set to 1 in the selection register. It does this by employing the fixed probability algorithm described previously and adding a dynamic probability component read from a table contained in an on-chip ROM block (this has to be preloaded during compilation). Details about the simulated annealing experiments are documented in Section 4.3.3. Despite this, the behaviour as well as the interface of this module are identical to the previously described variants of the bit source without preseeding. Source code for the generation of the simulated annealing table data can be found in Appendix A.3 along with the source code of this module in Appendix B.3.5.

ROM interface for simulated annealing stepping tables This module provides a wrapper for an on-chip SRAM block configured to operate in ROM mode and is internally used by the previously described module. The ROM block is accessed in units of 16 bits and holds a maximum of 4096 words which are preloaded from file `sa_table.mif`. The source code of this module is available in Appendix B.3.6.

The initialisation file has to be an ASCII text file (with the extension `.mif`) that specifies the initial content of a memory block, that is, the initial values for each address. This file is used during project compilation and/or simulation. A MIF is used as an input file for memory initialization in the Compiler and Simulator (alternatively a Hexadecimal (Intel-Format) File (.hex) can be used to provide memory initialisation data).

A MIF contains the initial values for each address in the memory. In a MIF, it is also required to specify the memory depth and width values. In addition, the radixes used to display and interpret addresses and data values can be specified.

```
DEPTH = 32;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 00000000;
01 : 00000001;
02 : 00000010;
03 : 00000011;
04 : 00000100;
05 : 00000101;
06 : 00000110;
07 : 00000111;
08 : 00001000;
09 : 00001001;
0A : 00001010;
0B : 00001011;
0C : 00001100;

END;
```

Figure 3.22: Example of a memory initialisation file (MIF)

The actual data used for determining the dynamic probability adjustments must consist of 16-bit words using big endian encoding. The data is encoded using a simple run length encoding scheme to save on-chip memory. The lower 10 bits of each word consist of the value $\lfloor 2^{10} \cdot (1-p) + 0.5 \rfloor$ with p being the probability to be added to the preset base probability (note that it is theoretically possible to exceed a probability of 1 using this mechanism, but this case is handled automatically by the circuit). The higher 6 bits of each words are treated as run-length counter. For example, if the first 16-bit word in the table is 0011000010000000, this means, that during the first 001100 = 12 clock cycles, a probability of $^1/_8$ is added to the preset base probability of a bit being sent through the selection register being set to 1. The sequence of code words has to be terminated by a word set to 0000000000000000 leaving a maximum of 4095 slots for table data. The source code provided in Appendix A.3 generates a table in the correct format using an adjustable exponentially declining probability boost curve.

Pseudo-random number generators

The pseudo-random number generators used by generate input bits for the probability distribution gating in front of the selection register are based und simple linear feedback shift registers (LFSR) using Fibonacci-Style layout and XNOR feedback gates. Figure 3.23 on page 40 shows block diagrams of the available LFSRs. Implementation details are shown by the module sources in Appendix B.4.

Please note that the 40-bit variants of the LFSR module contain a problem related to the period of the LFSR states. Section 5.2.4 describes the problem and gives some mathematical background.

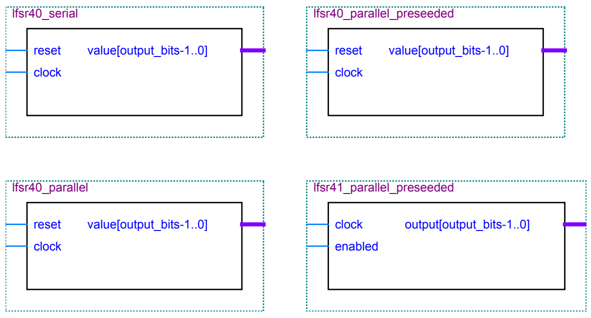

Figure 3.23: Block diagrams of LFSR based pseudo-random number generator modules

The seeds used in combination with the 40-bit LFSRs have been checked to give a reasonable high period in combination with the seeds used in most experiments (source code for simulating the 40-bit LFSR is available in Appendix A.4). Only the batch experiments described in Section 4.3.4 are affected by this flaw. It is strongly recommended to replace the 40-bit LFSR modules for future experiments. However, the 41-bit LFSR module is not affected by this weakness and gives the documented period regardless of the seed used.

Furthermore it is important to include at least one 0 bit in every seed used to initalise a LFSR module (the default seed for all modules consists of a 0 bit vector). If all bits of the register are set by the seed, the LFSR module gets stuck in this single state. Note that all other seed values are not creating this problem (if the seed contains at least one 0 bit, it is guaranteed, that the shift register never gets into a state where all bits are set to 1).

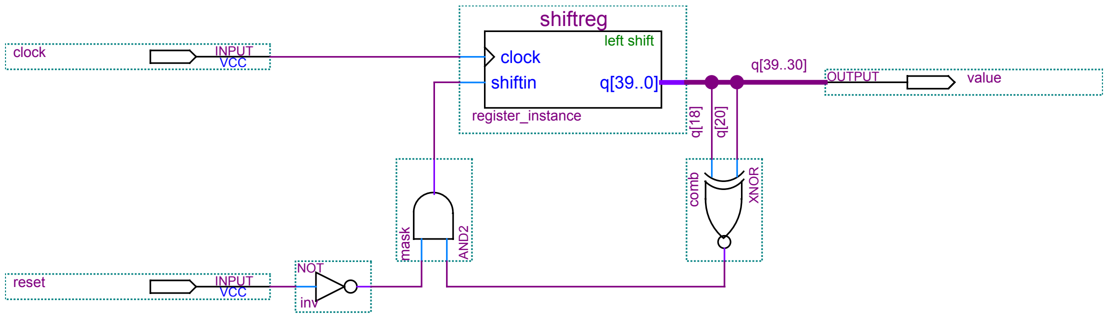

Figure 3.24: Schematic diagram of single bit LFSR (40-bit)

Single bit LFSR (40-bit) This basic single bit LFSR module implements a linear feedback shift register with a length of 40 bits. Please note previous paragraph about problems with the state period of this implementation. This variant of the 40-bit LFSR generates one fresh bit every clock cycle. The source code of this module is available in Appendix B.4.1.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state - in this case the shift register is cleared
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`value[]`	STD_LOGIC	Yes	Signal vector representing the higher part of the shift register
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the higher end shift register part led to the output port (valid values range from 1 to 40)

Table 3.12: Interface of single bit LFSR (40-bit)

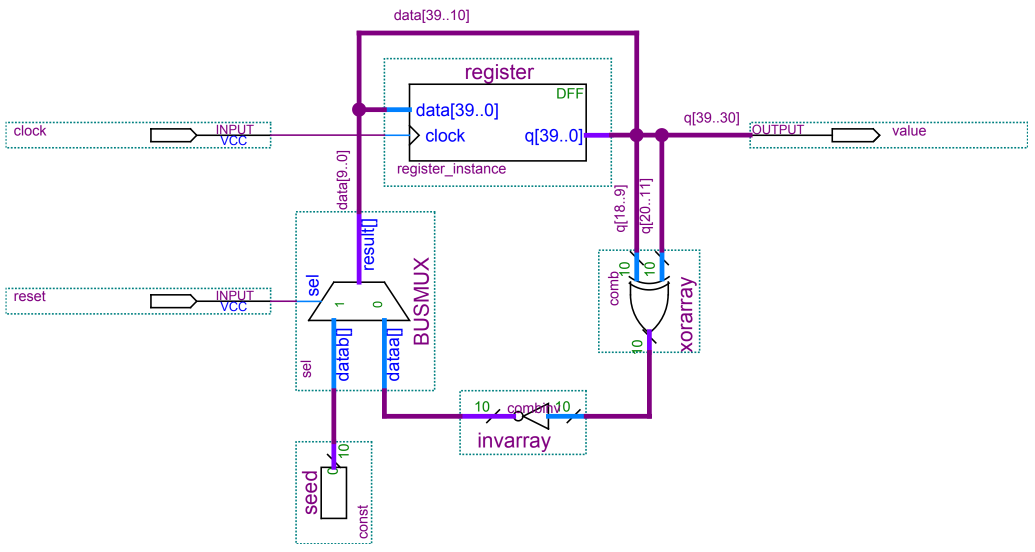

Figure 3.25: Schematic diagram of parallelised LFSR (40-bit)

Parallelised LFSR (40-bit) This parallelised 40-bit LFSR module behaves exactly like the single bit variant of the module described in the previous paragraph. The only exception is that the LFSR generates 10 fresh bits in every clock cycle using a parallelised implementation (which limits the maximum size of the output signal vector). Please note previous paragraph about problems with the state period of this implementation. The source code of this module is available in Appendix B.4.2.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state - in this case the shift register is cleared
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`value[]`	STD_LOGIC	Yes	Signal vector representing the higher part of the shift register
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the higher end shift register part led to the output port (valid values range from 1 to 19)

Table 3.13: Interface of parallelised LFSR (40-bit)

Parallelised LFSR supporting variable seed (40-bit) This module is identical to the parallelised 40-bit LFSR module with the only exception being that the shift register is set to a preset seed value if the *reset* signal is set, instead of just clearing it. Please note previous paragraph about problems with the state period of this implementation. The source code of this module is available in Appendix B.4.3.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state - in this case the shift register is reseeded using a preconfigured seed vector
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`value[]`	STD_LOGIC	Yes	Signal vector representing the higher part of the shift register
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the higher end shift register part led to the output port (valid values range from 1 to 19)
`seed[]`	STD_LOGIC_VECTOR	No	Seed to load into shift register whenever the *reset* signal is set to 1 (the default is filling the register with 0 bits)

Table 3.14: Interface of parallelised LFSR supporting variable seed (40-bit)

Parallelised LFSR supporting variable seed (41-bit) This module is identical to the parallelised 40-bit LFSR module supporting a confiurable seed. The only difference is an extended shift register (41 instead of 40 bits) using a different feedback function. This is currently the only LFSR implementation giving a period which is not dependant on the seed used. The period of the 41-bit LFSR is equal to $2^41 - 1$. The source code of this module is available in Appendix B.4.4.

Input port	**Type**	**Required**	**Comments**
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`enabled`	STD_LOGIC	Yes	The module expects the *enabled* signal being 0 if in reset state - in this case the shift register is reseeded using a preconfigured seed vector
Output port	**Type**	**Required**	**Comments**
`output[]`	STD_LOGIC	Yes	Signal vector representing the higher part of the shift register
Parameter	**Type**	**Required**	**Comments**
`output_bits`	Integer	Yes	Length of the higher end shift register part led to the output port (valid values range from 1 to 37)
`seed[]`	STD_LOGIC_VECTOR	No	Seed to load into shift register whenever the *enabled* signal is set to 0 (the default is filling the register with 0 bits)

Table 3.15: Interface of parallelised LFSR supporting variable seed (41-bit)

Support circuitry

The various support circuitry modules are intended to guarantee a fully defined execution environment for the various SAT solvers and to collect measurement data about their performance. The serialisation of the measurement data is supported by special modules as well which write the collected result data to the on-chip memory allowing it to be read by the host computer. Implementation details are shown by the module sources in Appendix B.5.

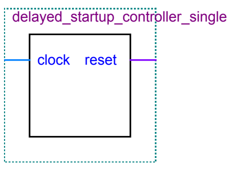

Figure 3.26: Block diagram of delayed startup controller module

Delayed startup controller for single testruns The delayed startup controller module guarantees that a reset signal is automatically issued for a preset number of clock cycles after the circuit powers up. This way it guarantees that all components of the circuit are properly initialised before the actual circuit operation starts. The circuit immediately starts running after programming of the FPGA device finished with all flip-flops and memory blocks, respectively, being initalised to 0 bits, unless otherwise stated in the source code. The component is designed to wait 71590000 clock cycles (which corresponds to 5 seconds assuming the FPGA is running at the base frequency of the development board being 14.318 MHz) during which the *reset* signal is set to 1. After this number of clock cycles passed, the output *reset* signal is set to 0 for 100 clock cycles and set to 1 again after this period of time. The source code of this module is available in Appendix B.5.1.

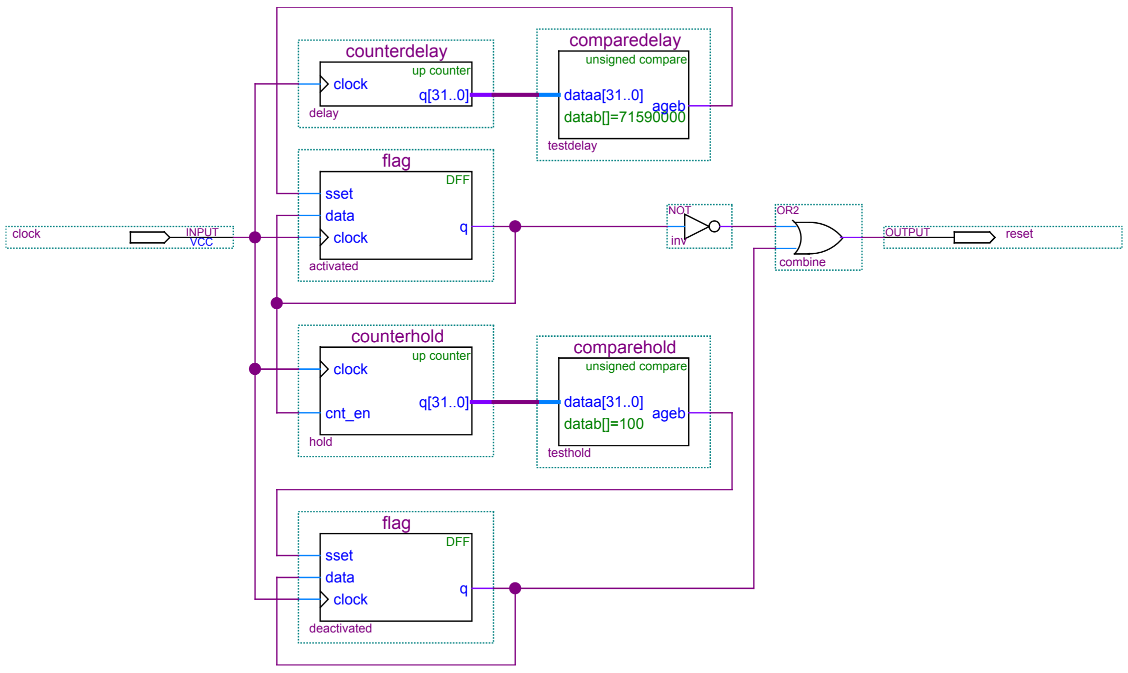

Figure 3.27: Schematic diagram of delayed startup controller for single testruns

Input port	**Type**	**Required**	**Comments**
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module issues the *reset* signal being 0 if in reset state and being 1 otherwise

Table 3.16: Interface of delayed startup controller for single testruns

Delayed startup controller for batch testruns This delayed startup controller module is a variant of the previously described module designed to be used in the batch test environment described in Section 4.3.4. This test environment uses two distinct reset signals, one resetting the whole circuit and another one just restarting a single test run. The delayed startup controller only manages the global reset signal initialising the circuit. This signal is issued for 71590000 + 100 clock cycles and cleared after that. It stays this way until the circuit is powered down. Please note that the semantic of the *reset* signal was swapped compared to the previously described module. The source code of this module is available in Appendix B.5.2.

Input port	**Type**	**Required**	**Comments**
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module issues the *reset* signal being 1 if in global reset state and being 0 otherwise

Table 3.17: Interface of delayed startup controller for batch testruns

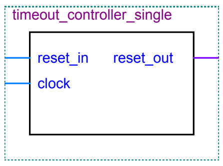

Figure 3.28: Block diagram of timeout controller module

Timeout controller for single testruns The timeout controller module aborts a testrun if a solution has not been found after a configurable number of clock cycles and initiates the writing of the result data to the on-chip memory. If used in manual experiments without the delayed startup controller this module also eliminates problems produced by bouncing or floating reset signals and guarantees precise measurement timeouts (e.g. the push button switches on the development are not sufficiently stabilised). As long as the incoming *reset* signal is set to 1 this settings is just passed through. Whenever the incoming *reset* signal becomes 0 the modules ignores the incoming *reset* signal for the preconfigured amount of clock cycles and sets its outgoing *reset* signal to 0 until the timeout elapses. After this amount of time the outgoing *reset* signal is set to 1 again and the component restarts listening to the incoming *reset* signal. Please note the different semantics of the incoming and outgoing reset signals. The source code of this module is available in Appendix B.5.3.

Input port	**Type**	**Required**	**Comments**
`reset_in`	STD_LOGIC	Yes	The module expects the incoming *reset* signal being 0 if in reset state and being 1 otherwise
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`reset_out`	STD_LOGIC	Yes	The module issues the outgoing *reset* signal being 1 if in reset state and being 0 otherwise
Parameter	**Type**	**Required**	**Comments**
`timeout_cycles`	BIT_VECTOR	No	Natural number specifying the number of clock cycles the SAT solver has to find a solution (the default is 71590000 clock cycles)

Table 3.18: Interface of timeout controller for single testruns

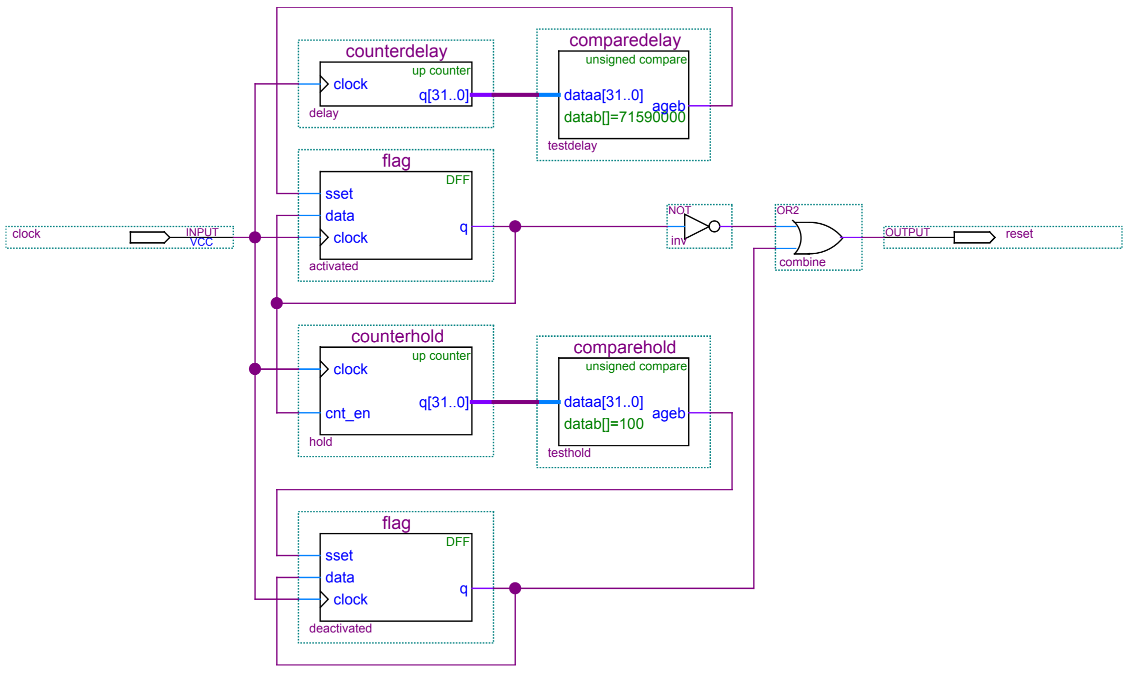

Figure 3.29: Schematic diagram of timeout controller for single testruns

Timeout controller for batch testruns This module is a modified variant of the basic timeout controller which got amended by a small state machine which controls starting and stopping consecutive testruns in a batch test environment. This component was used during the experiments described in Section 4.3.4. Please note that the semantics of the incoming *reset* signal changed (an incoming *reset* signal of 1 now means being in reset state which is compatible with the unchanged semantics of the outgoing *reset* signal). The source code of this module is available in Appendix B.5.4.

Input port	**Type**	**Required**	**Comments**
`reset_in`	STD_LOGIC	Yes	The module expects the incoming *reset* signal being 1 if in reset state and being 0 otherwise
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
Output port	**Type**	**Required**	**Comments**
`reset_out`	STD_LOGIC	Yes	The module issues the outgoing *reset* signal being 1 if in reset state and being 0 otherwise
Parameter	**Type**	**Required**	**Comments**
`timeout_cycles`	BIT_VECTOR	No	Natural number specifying the number of clock cycles the SAT solver has to find a solution (the default is 71590000 clock cycles)

Table 3.19: Interface of timeout controller for batch testruns

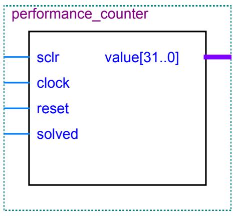

Figure 3.30: Block diagram of performance measurement module

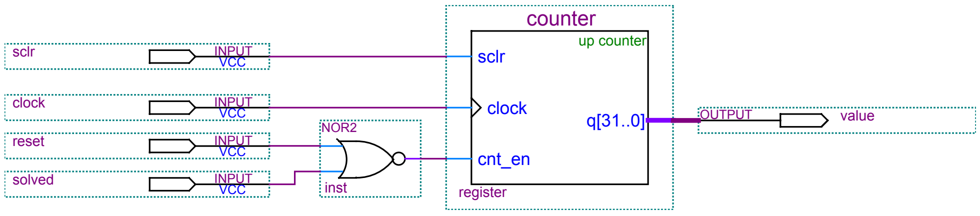

Figure 3.31: Schematic diagram of performance measurement

Performance counter The performance counter module acts as a wrapper around a binary 32-bit counter, incrementing by 1 in every clock cycle. The counter is only running if neither the *reset* nor the *solved* signal is set to 1 (but it still keeps its value if this is not the case). The source code of this module is available in Appendix B.5.5.

Input port	**Type**	**Required**	**Comments**
`sclr`	STD_LOGIC	Yes	Clears the counter register if set to 1
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`reset`	STD_LOGIC	Yes	The module expects the incoming *reset* signal being 1 if in reset state and being 0 otherwise
`solved`	STD_LOGIC	Yes	The module expects the incoming *solved* signal being 1 if the SAT solver found a solution and being 0 otherwise
Output port	**Type**	**Required**	**Comments**
`value[]`	STD_LOGIC_VECTOR	Yes	Signal vector representing the current value of the 32-bit counter register

Table 3.20: Performance measurement interface

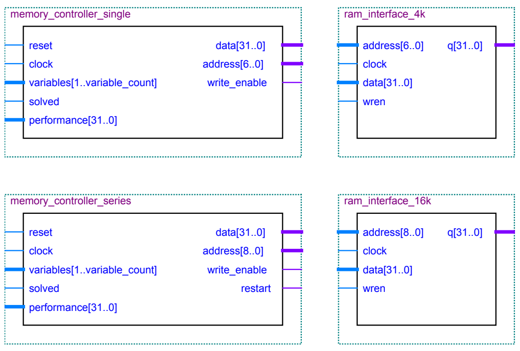

Figure 3.32: Block diagrams of memory controller modules

Memory controller for single testruns The memory controller module implements a circuit responsible for collecting and serialising measurement data which is written to the attached memory block interface. The circuit itself is synthesised from a serialisation algorithm. Serialisation of measurement data is triggered by the *reset* signal being set to 1. The source code of this module is available in Appendix B.5.6.

The memory controller module outputs the measurement data to the on-chip memory according to the following data format. The measurement data is organised in 32-bit words stored in big endian byte order.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state and being 0 otherwise
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`variables[]`	STD_LOGIC_VECTOR	Yes	Final truth assignment established by the SAT solver to be serialised
`solved`	STD_LOGIC	Yes	The module expects the *solved* signal being 1 if the SAT solver claims having found a solution and being 0 otherwise
`performance[]`	STD_LOGIC_VECTOR	Yes	Signal vector describing the number of clock cycles the SAT solver ran
Output port	**Type**	**Required**	**Comments**
`data[]`	STD_LOGIC_VECTOR	Yes	32-bit data port to the attached memory block interface
`address[]`	STD_LOGIC_VECTOR	Yes	7-bit address port to the attached memory block interface
`write_enable`	STD_LOGIC	Yes	Write enable port to the attached memory block interface (set to 1 if the *data* and *address* vectors are valid)
Parameter	**Type**	**Required**	**Comments**
`variable_count`	Integer	Yes	Number of variables participating in the SAT instance

Table 3.21: Interface of memory controller for single testruns

Bit offset	**Content**	**Comment**
0x00	Number of clock cycles the SAT solver ran	This might be the preconfigured timeout of the timeout controller if the SAT solver did not manage to find a solution
0x20	Solution status flag	Set to 1 if the SAT solver claims having found a solution and set to 0 otherwise
0x30	Truth assignment	Final truth assignment established by the SAT solver (variable values are serialised starting with the highest bit of the word and padded with zero bits if the number of variables is not a multiple of 32)
0x30 + $\lceil {}^{count}/_{32} \rceil$	Number of variables participating in the instance	Mainly intended for debug purposes

Table 3.22: Data format produced by memory controller for single testruns

Memory controller for batch testruns The memory controller module for batch testruns is a modified version of the basic memory controller module. It writes a hardcoded number of 256 pairs of solved flags and performance counter values to the attached memory block interface. After each testrun the controller issues a *restart* signal triggering the beginning of the next test run until 256 testruns got executed and their results stored. The circuit itself is synthesised from a serialisation algorithm. Serialisation of measurement data is triggered by the *reset* signal being set to 1. The source code of this module is available in Appendix B.5.7.

Input port	**Type**	**Required**	**Comments**
`reset`	STD_LOGIC	Yes	The module expects the *reset* signal being 1 if in reset state and being 0 otherwise
`clock`	STD_LOGIC	Yes	Module operation is triggered by the rising edge of the *clock* signal
`variables[]`	STD_LOGIC_VECTOR	Yes	Final truth assignment established by the SAT solver to be serialised
`solved`	STD_LOGIC	Yes	The module expects the *solved* signal being 1 if the SAT solver claims having found a solution and being 0 otherwise
`performance[]`	STD_LOGIC_VECTOR	Yes	Signal vector describing the number of clock cycles the SAT solver ran
Output port	**Type**	**Required**	**Comments**
`data[]`	STD_LOGIC_VECTOR	Yes	32-bit data port to the attached memory block interface
`address[]`	STD_LOGIC_VECTOR	Yes	7-bit address port to the attached memory block interface
`write_enable`	STD_LOGIC	Yes	Write enable port to the attached memory block interface (set to 1 if the *data* and *address* vectors are valid)
`restart`	STD_LOGIC	Yes	Set to 1 when the start of the next testrun is requested and set to 0 otherwise
Parameter	**Type**	**Required**	**Comments**
`variable_count`	Integer	Yes	Number of variables participating in the SAT instance

Table 3.23: Interface of memory controller for batch testruns

The modified memory controller module outputs the measurement data to the on-chip memory according to a simplified data format. The measurement data is organised in 32-bit words stored in big endian byte order. The first 256 words each contain a 1-bit flag set to 1 if the SAT solvers claims having found a solution which is stored in the highest bit of the word. The lower 31 bits contain the number of clock cycles the SAT solver ran. The established truth assignments are not stored to the result data memory. The 256 result words are followed by a single checksum word mainly intended for debug purposes. This checksum c is computed as a rotating XOR-based checksum of the data words w_i:

$$c := \bigoplus_{i=0}^{255} \left(w_i \cdot 2^{8(4-(i \mod 4))} + \lfloor w_i \cdot 2^{-8(i \mod 4)} \rfloor \right) \mod 2^{32}$$

RAM interface (4K) This module provides a wrapper for an on-chip SRAM block configured to operate in RAM mode and is used by the memory controller module for single testruns. The ROM block is accessed in units of 32 bits and holds a maximum of 128 words. The source code of this module is available in Appendix B.5.8.

RAM interface (16K) This module provides a wrapper for an on-chip SRAM block configured to operate in RAM mode and is used by the memory controller module for batch testruns. The ROM block is accessed in units of 32 bits and holds a maximum of 512 words. The source code of this module is available in Appendix B.5.9.

3.2.4 Introduction to the JTAG standard

Joint Test Action Group (JTAG) is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture as well as following standards based on this for test access ports used for testing printed circuit boards using boundary scan. Boundary scanning is a technique which allows specifically defined registers and signals, respectively, of a circuit being accessed by an external interface without disturbing the chip operation itself. This way JTAG provides a very convinient way for debugging hardware of various kinds.

While designed for printed circuit boards, it is nowadays primarily used for testing sub-blocks of integrated circuits, and is also useful as a mechanism for debugging embedded systems, providing a convenient "back door" into the system. When used as a debugging tool, an in-circuit emulator which in turn uses JTAG as the transport mechanism enables a programmer to access an on-chip debug module which is integrated into a chip via JTAG. The debug module enables the programmer to debug the behaviour of an embedded system.

Hardware devices communicate to the outside world via a set of I/O pins. By themselves, these pins provide limited visibility into the workings of the device. However, devices that support boundary scan contain a shift-register cell for each signal pin of the device. These registers are connected in a dedicated path around the device's boundary (hence the name). The path creates a virtual access capability that circumvents the normal inputs and provides direct control of the device and detailed visibility at its outputs. Many modern devices are even able to provide this kind of debug facility for structures within the circuitry implemented by the chip. This way a chip can allow debug access to internal strucutres which otherwise would be completely isolated and invisible from the outside world. During testing, I/O signals enter and leave the chip or its components, respectively, through the boundary-scan cells. The boundary-scan cells can be configured to support external testing for interconnection between chips or internal testing for logic within the chip.

To provide the boundary scan capability, IC vendors add additional logic to each of their devices, including scan registers for each of the signal pins, a dedicated scan path connecting these registers, four or five additional pins, and control circuitry. The overhead for this additional logic is minimal and generally well worth the price to have efficient testing at the board level.

Almost all modern FPGA provide powerful JTAG based debugging and even programming capabilities. Using the standardised JTAG interface, it is possible to read and even write register and memory block contents inside the FPGA and to access signals using special control circuitry. Most devices even allow to be partially or fully programmed using the JTAG transport infrastructure. This way the JTAG interface provides an integrated communication platform between a FPGA and a host computer which provides all ways of interactions with the hardware device that are necessary during hardware development. Most FPGA development environments even contain JTAG based communication libraries which allow for automated testing and communication with the FPGA device. For example, the Altera development environment used during this project, provides efficient ways to read the full contents of an on-chip SRAM block to the host computer and if necessary to write new data back into the memory block.

3.2.5 Automatic generation and execution of test cases

To be able to run large-scale experiments on the FPGA equipment covering reasonable numbers of different SAT instances, it was necessary to build an infrastructure able to automatically generate hardware definitions implementing a given SAT instance in the way it was desired in the particular experiment and to automatically compile, run, and measure the generated hardware definitions. Especially compilation and execution of the test cases had to be done in a fully unattended manner since the compilation of a single test case can take up to several minutes depending on the exact scenario.

The generation of the hardware definitions is highly specific for the different experimental setups and described in Chapter 4. However, since the circuitry used was designed with a high level of reusability in mind, the generation in most cases was restricted to the generation of the main SAT solver module. The main modules linking the different components together were adjusted manually in most cases because they just changed between several experimental scenarios but nut depending on the instance being analysed.

Compilation, execution and data collection were performed by simple Windows batch scripts, which were manually adjusted for the different experiments. These scripts called several prebuild tools and script modules to make the setup of the experiments as efficient as possible.

The unattended interaction with the Altera provided development environment was performed using a couple of very powerful command-line interfaces to the Altera software. These interface allow for script controlled operation of nearly the whole development environment. The core script used to compile and run a test case and to read back the result data is shown in Figure 3.33 on page 52.

```
md %1
copy %1.vhd %1\sat_solver.vhd
copy %2 %1\
cd %1

quartus_map Sample --source=Sample.vhd --read_settings_files=on --write_settings_files=off --
    family=Cyclone
quartus_fit Sample --read_settings_files=off --write_settings_files=off --part=EP1C6Q240C8 --
    fmax=14.318MHz
quartus_asm Sample --read_settings_files=off --write_settings_files=off
quartus_tan Sample --read_settings_files=off --write_settings_files=off --timing_analysis_only
quartus_pgm -c "ByteBlaster [LPT1]" Sample.cdf
..\Sleep.exe 15000
quartus_stp -t ..\readmem.tcl > out_stp1.txt
..\Sleep.exe 5000
quartus_stp -t ..\readmem.tcl > out_stp2.txt

cd ..
```

Figure 3.33: Example script controlling automated operation of the Altera Quartus II development environment

The script takes as first parameter the name of the test case to execute. The specification of the SAT solver module is expected to be stored under the name of the test case using the extension `.vhd` in the current directory. As second parameter the script takes a directory, whose contents are copied together with the SAT solver module to a working directory named after the test case. This mechanism is intended to provide a template directory containing all files which are identical for all SAT instances investigated in the current experiment (which are actually all files except the actual SAT solver module).

After compiling the test case (whose master definition files are expected to be named Sample.vhd and Sample.cdf for legacy integration reasons), the script programs the FPGA device. After this the script waits 15 seconds (needs to be adjusted for batch test cases) to allow the FPGA running the specific test case. Waiting is performed using a small application whose source code can be

```
begin_memory_edit -hardware_name "ByteBlasterII␣\[LPT1\]" -device_name "@1:␣EP1C6␣(0x020820DD)"
puts [read_content_from_memory -instance_index 0 -start_address 0 -word_count 128
     -content_in_hex]
end_memory_edit
```

Figure 3.34: Example script controlling JTAG communication through the Altera Quartus II development environment

found in Appendix A.5. The result data is read back using the Altera provided JTAG based communication tool which is controlled using TCL scripts. Figure 3.34 on page 53 shows a TCL script to read a single JTAG enabled memory block and displaying its contents in hexadecimal notation to the screen (needs to be adjusted for batch tests as well since it reads only 128 words in the displayed configuration). Please note that the words contained in the memory are read in the opposite order as they appear in the SRAM block (e.g. the word written to the lowest address of the memory block will be the last word outputted by the script). A second copy of the result data is read after a delay of 5 seconds. This copy is compared by other support tools against the first copy for debug purposes (but actually this comparison did not fail even in a single test case).

The text files created by this script contain the result data according to the formats described in Section 3.2.3. They were processed by various scripts and tools to aggregate them into comma separated value (CSV) table files which are readable by Microsoft Excel and other spreadsheet applications. Many of these tools and scripts are quite specific to the different experiment scenarios. Since all of these tools are very basic text processing and aggregation tools (written in C#), there is no point in discussing them in detail in this report because they are not giving any insights into the matter of the project. The scripts and tools are included on the accompanying CD-ROM to make them available to future projects. The aggregated results of the various experiments are presented in Chapter 5.

3.3 Acquisition of reference data

Acquiring and verifying reference data was a crucial aspect of the project. On the one hand there was the need for randomly generated SAT instances consisting of a defined number of variables and clauses having a specified length to be investigated using different hardware SAT solver approaches. On the other hand, reliable performance data of SAT solvers on these instances was necessary to have a base data set for comparing the hardware performance against.

3.3.1 Generation of random SAT instances

Since the aim of the project mainly was the research on a general purpose SAT solver engine rather than a domain specific engine, the decision was taken to investigate the behaviour of the software and hardware SAT solver engines on pseudo-randomly generated SAT instances. This requirement was served by a manually created SAT instance generator. The generated instances should have four basic properties:

- Distinct variables should appear according to a uniform probability distribution
- A variable must not appear multiple times inside a single clause
- All specified variables must appear inside the SAT instance
- The generated SAT instance must not be easily partitionable

The reason for the first three requirements is quite obvious. If a SAT instance is partitionable, this means, that it is possible to split the set of variables into multiple classes in a way, that no two variables contained in different classes together appear in the same clause. A partitionable SAT instance in 3CNF can easily be split into smaller SAT instances which can be solved individually. The original SAT instance is satisfiable if, and only if, all of these sub-instances are satisfiable.

The format chosen to represent SAT instances is compatible with the one used by many SAT related tools published by other researchers. It is a simple text format in which every line starting with a single letter "c" followed by a space character as well as whitespace lines are treated as comments. The first non-comment line must be starting with the string "p cnf" followed by a space character which is followed by the number of variables and the number of clauses, separated by a space character. Each following line represents a single clause of the SAT instance. Variables are named consecutively starting from 1 to the number of variables available. A clause is defined by a space separated series of variable numbers, optionally prepended by a "-" character which signals an inverted literal. The lines are terminated by an optional "0" character followed by a normal line break. Optionally the last non-comment line of the file can contain a single "0" character to signal the end of the file. Most tools used in this project handle these "0" delimiters in a flexible way by ignoring them. Regarding SAT related tools on the internet, there are some tools which actually require the zeroes and other that do not require or even do not allow them.

1. Repeat for k clauses with 3 out of n variables each:
   a) Get 2 pseudo-random bytes $b_i, i \in \{0..1\}$ from the cryptographic random number generator built into Microsoft Windows (Crypto API)
   b) Convert b_0 to a double precision floating point value μ (53-bit mantissa)
   c) Multiply μ by $(n-1)$ (assuming $n \leq 50$, giving at most 46 significant bits)
   d) Divide μ by $(2^8 - 1)$
   e) Floor the result and use it as variable identifier (in the range of 1 to n)
   f) If variable is already present in the current clause, discard it and restart current iteration
   g) If $b_1 \geq 128$, the variable is inverted in the clause
2. Check whether all n variables occur in the instance - if not discard generated instance and repeat generation process

Figure 3.35: Basic algorithm for generation and pseudo-random SAT instances (not recommended for future experiments)

During the first experiments generation of SAT instances was accomplished by the algorithm show in Figure 3.35 on page 54. Unfortunately, this algorithm provides a resonably uniform probability distribution only for smaller variable counts ($\leq$ 50) which caused problems during the phase transition related experiments. Therefore the algorithm was replaced by the algorithm shown in Figure 3.36 on page 55 in all following experiments. The first version of the generation algorithm should not be used for future experiments. The source codes of the instance generators are available in Appendix A.6 and Appendix A.7, respectively.

The distribution of the variables inside a generated instance as well as whether it is easily partitionable were verified by a secondary tool whose source code is available on the accompanying CD-ROM. The partitionability check works by building the dependancy graph of the variables regarding the clauses they participate in. Each variable participating in the analysed instance corresponds to a vertex in the dependancy graph. The graph contains an (undirected) edge between two vertices if the corresponding variables participate together in a single clause. The instance is easily partitionable if the dependancy graph is not connected (meaning that there exist vertices

1. Repeat for k clauses with 3 out of n variables each:
    a) Get 6 pseudo-random bytes $b_i, i \in \{0..5\}$ from the cryptographic random number generator built into Microsoft Windows (Crypto API)
    b) Concatenate first 5 bytes to form an unsigned 40-bit integer $\lambda := \sum_{k=0}^{4} b_i \cdot 2^{8(4-i)}$
    c) Convert it to a double precision floating point value μ (53-bit mantissa)
    d) Multiply μ by n (assuming $n \leq 256$, giving at most 48 significant bits)
    e) Divide μ by 2^{40}
    f) Floor the result and use it as variable identifier (in the range of 1 to n)
    g) If variable is already present in the current clause, discard it and restart current iteration
    h) If $b_5 \geq 128$, the variable is inverted in the clause
2. Check whether all n variables occur in the instance - if not discard generated instance and repeat generation process

Figure 3.36: Improved algorithm for generation and pseudo-random SAT instances

which do not have a path between them).

3.3.2 Examination of satisfiability using software tools

To be able to verify the correct behaviour of the different hardware SAT solver engines it was important to know whether a particular SAT instance was satisfiable or unsatisfiable. This knowledge was acquired by running all generated instances through a complete software SAT solver known to work reliably. The software solver chosen for this task is MiniSat which is a freely available complete light-weight SAT solver implemented in C. It supports the previously mentioned data format and also makes performance measurements quite easy (see Section 3.3.3). MiniSat operates based on the DPLL algortihm mentioned in Section 2.2. The original version is designed to be used under Linux but there is also a patch available to make it compile under Windows.

The accompanying CD-ROM includes some scripts used to automatically generate large numbers of pseudo-random SAT instaces and running them through MiniSat. There is also a small tool available aggregating the MiniSat results into CSV files to be further processed by other aggregation tools mentioned in Section 3.2.5 or to be used directly within a spreadsheet application, respectively.

3.3.3 Automatic measurement of software solver timings

Since the aim of the project was the research in efficient hardware SAT solver engines which are able to operate faster than existing software based SAT solver engines it was necessary to acquire timing information of various software SAT solvers for comparison. The primary problem of this task is the fact, that on the one hand, the algorithms implemented by the software SAT solver engines are heavily different from the hardware approaches researched during this project. This makes it hard to measure the performance in some sort of "algorithm steps". On the other hand, since most experiments were done on rather small SAT instances due to the limited space available on the provided FPGA device, the SAT solvers found most solutions so quickly, that it was not possible to get meaningful execution timings on the application level. The latter is also undesirable because this method of measurement would include the time the software SAT solver needs to start and to load and to preprocess a particular SAT instance. Regarding the hardware SAT solver engines, this time is absorbed by the compilation stage which is not included into the measurements because this amount of time is negligible if the hard engine is implemented by an ASIC. Therefore another way had to be found to measure the performance of the software engines.

The software SAT solvers used for comparison purposes were the previously mentioned MiniSat solver [ES03] [SE05] on the one hand, which is a complete solver, and the software based WalkSAT solver, which is an incomplete solver more closely related to the algorithms implemented by the hardware engines. Both solvers are freely available through the internet and operate as command-line tools under Linux. To get meaningful performance data about these solvers the decision was taken to slightly modify both solvers enabling them to use the time-stamp counter included in all modern Intel IA-32 compatible CPUs as a timing reference. This time-stamp counter consists of a 64-bit register (even on 32-bit CPUs) which is initialised to 0 at powerering up the CPU and incremented by 1 every clock cycle regardless of the application context. The register contents can be read by a special CPU instruction named RDTSC which is available in all priviledge levels (see [Int06a] and [Int06b]).

Both software SAT solvers were prepared by surrounding their inner search loops by two measurement points reading the time-stamp register to a local variable. The difference of the time-stamps at both measurement points gives the number of clock cycles the search ran through. Figure 3.37 on page 56 shows inline assembly code reading the time-stamp register to a local variable living on the stack. To make measurement results more meaningful, all screen output and unnecessary statistics collection of the SAT solvers which take place in the main search loop were removed (see modified sources available on the accompanying CD-ROM).

```
unsigned int tscStartHigh;
unsigned int tscStartLow;
unsigned int tscEndHigh;
unsigned int tscEndLow;
unsigned long long clockCycles;

__asm__ __volatile__ (
        "rdtsc;"
        "mov␣%%eax,␣%0;"
        "mov␣%%edx,␣%1;"
        : "=m"(tscStartLow), "=m"(tscStartHigh)
        : "m"(tscStartLow), "m"(tscStartHigh)
        : "%eax", "%edx"
);

/*
 * Activity to measure goes here
 */

__asm__ __volatile__ (
        "rdtsc;"
        "mov␣%%eax,␣%0;"
        "mov␣%%edx,␣%1;"
        : "=m"(tscEndLow), "=m"(tscEndHigh)
        : "m"(tscEndLow), "m"(tscEndHigh)
        : "%eax", "%edx"
);

clockCycles = ((((unsigned long long)(tscEndHigh)) << 32) | ((unsigned long long)(tscEndLow)))
        - ((((unsigned long long)(tscStartHigh)) << 32) | ((unsigned long long)(tscStartLow)))
```

Figure 3.37: Example C/Assembler source for reading the time-stamp counter of Intel IA-32 compatible CPUs

The main problem with this measurement technique is the fact that the time-stamp register is independant of execution context and cannot be saved by the operating system or an application. Therefore it is necessary to reduce the number of CPU interrupts and context changes during the measurement interval as much as possible. This was accomplished by booting the computer running the measurements from a bootable Linux CD-ROM into text mode without loading the graphical user interface (Knoppix V5.1.0 English CD edition was used for the measurements).

All unnecessary cables like USB devices, mouse and network connection were unplugged and the bootable CD-ROM removed from the drive (the CD data was entirely loaded to a RAM disk at startup). Each SAT instance was measured 100 times and the minimum timing of all runs taken as the result.

4 Large-scale experiments

4.1 Basic circuits

After reaching a project state allowing for automated large-scale experimentation, the first circuits investigated were straight-forward implementations of small SAT instances consisting of 10 variables following the basic experiments described in Section 3.1.2 and Section 3.1.3. These experiments were the first experiments which used the newly created VHDL component library documented in Section 3.2.3. Rather than manually implementing single instances, these experiments used SAT solver modules automatically generated by software out of SAT instance descriptions generated using the techniques described in Section 3.3.1.

The main goals of these experiments were on the one hand to check that the automated testing facilities described in Section 3.2.5 were working properly. On the other hand the scalability of the basic algorithms proposed in [COP06] on slightly larger instances was of major interest. Until these experiments, the proposed algorithms had only been tested on very small instances consisting of variable and clause counts in ranges where in fact all produced SAT instances are satisfiable.

These early automated experiments were accompanied by the creation of an extensible generator application which is able to generate SAT solver modules in VHDL language based on SAT instance descriptions. The generator tool is included on the accompanying CD-ROM and support a wide variety of options for the creation of the SAT solver modules. Unless otherwise stated, all SAT solver modules used in the automated experiments were created using this tool.

The SAT instances used were created using version 1 of the SAT instance generator which gives a reasonable variable distribution for the given instance sizes. All instances investigated consisted of 10 variables. The number of clauses included were 30, 40, 50, 60, 70 and 80, respectively. For each configuration 30 instances were generated leading to a total of 180 instances which were investigated.

4.1.1 Asynchronous circuits

These were the first automated experiments executed using the newly created automated testing environment. The SAT solver modules used in these experiments were of the asynchronous type described in Section 3.1.3 with additionally added logic described in Section 3.1.4 to harden the circuits against compiler optimisations. The top-level template linking the SAT solver modules with the synchronous support circuitry can be found in Appendix C.1.

Since the asynchronous circuit type showed very promising behaviour through the manual experiments, it was chosen first for the automated tests. Unfortunately, the results were very disappointing because the circuits did not manage to come up with a solution in the given time for most satisfiable instances. A couple of instances could be solved by this type of circuit but the average performance reached was very poor (see Section 5.1 for results and a discussion of the behaviour of this circuit type).

A couple of experiments were carried out testing the circuit type using different numbers of delay gates and insertion of delay logic to other parts of the circuit. However, these modifications were unable to noticeably increase the average performance of the circuit type. The main problem with the asynchronous circuit type is that the Altera provided compiler provides only very limited options to influence the optimisation and the layout of combinational loops. Even for smaller instances the compiler takes large amounts of time apparently trying to optimise the combinational circuit. Doing this it outputs warning messages stating that a combinational loop was found. Since proper support for combinational loops is apparently not integrated into the Altera compiler and because of the fact, that meaningful information about the circuit behaviour is not extractable without precise control over the circuit layout on the FPGA chip, the decision was taken to drop the idea of having fully combinational SAT solver engines for this project. Instead of this, all further efforts were concentrated on the optimisation of the synchronous variants of the SAT solver engine.

4.1.2 Synchronous circuits

The first synchronous circuits investigates in the automated testing environment were of the synchronous type described in Section 3.1.2 with additionally added logic described in Section 3.1.4 to harden the circuits against compiler optimisations. The top-level template linking the SAT solver modules with the support circuitry can be found in Appendix C.2.

Unfortunatly it turned out, that the basic algorithm concept used in the manual experiments is not scalable to larger instances because the fully deterministic synchronous circuit type was unable to solve most instances provided. Only very few instances could be solved and these were limited to instances which were either satisfied by the initial truth assignemnt (all variables set to $false$) or which required only a single cycle truth the circuit flipping some variables. A short discussion of this behaviour is included in Section 5.2.1.

Because of the structure of the SAT instances the synchronous circuit type was able to solve it was conjectured that the synchronous circuit is toggling to many variables at ones continously flipping between truth assignments having most variables set to either *true* or $false$, respectively. This led to the idea of introducing some form of randomisation to the synchronous circuit. The basic idea was to toggle a variable participating in an unsatisfied clause only with a certain probability while variables participating in more unsatisfied clauses than others should have a higher probability of being flipped.

4.1.3 Probabilistic synchronous circuits

The idea behind the first probability driven circuits was that each unsatisfied clause on average should cause only one of its variables to be toggled to prevent the global truth assignment from changing to quickly. To accomplish this task the synchronous circuit was amended by a shift register holding one bit for each literal in each clause (e.g. for 50 clauses, this means 150 bits assuming a SAT instance in 3CNF). This shift register is fed by a pseudo-random number generator (implemented as linear feedback shift register) whose output is postprocessed by gating logic to convert the uniform binary probability distribution of the LFSR to a configurable binary probability distribution (in this case giving a probability of approximately 1/3 for a bit being set to 1). The shift register is running through all term evaluators and shifted by one bit each clock cycle. An unsatisfied clause triggers the toggling of a participating variable only if the corresponding bit in the area of the shift register corresponding to this clause is set to 1. The top-level template linking the SAT solver module with the mentioned shift register and the support circuitry can be found in Appendix C.3.

Despite using early randomisation components later proving to have significant problems regarding various functional aspects and suffering from statistical dependencies and short periods, this synchronous circuit type managed to solve all satisfiable instances which were investigated. Most of them were even solved in significantly less than 1000 clock cycles. Even the use of a erroneous term evaluator component producing wrong toggling probabilities did not significantly obstruct the computation of satisfying truth assignments because the SAT instances used were still very small. A discussion about the behaviour of the circuit type can be found in Section 5.2.2.

4.2 Phase transition related experiments

Because the space on the available FPGA device is very limited, the idea came up to systematically generate SAT instances which will be particularly hard to solve because of their structure. Previous publications [CKT91] [GMPW96] show that large numbers of hard SAT instances can be found at specific ratios of the number of participating variables to the number of clauses as described in Section 2.3.

Since available research publications experimentally show the existence of these kinds of phase transition phenomena, the available documentation does not provide large-scale experimental results about the exact location of the phase transition regarding different numbers of participating variables. Therefore two different experiments were setup to investigate the behaviour of a complete software SAT solver regarding phase transition phenomena taking into account the number of participating variables and to investigate the behaviour of the previously introduced probabilistic hardware SAT solver engine in and around the phase transition area.

4.2.1 Phase transition points

To get more precise data about the location of the phase transition points, the first step was to carry out a purely software based experiment. SAT instances consting of 5 to 250 variables in steps of 5 variables were analysed. For each number of variables 1000 pseudo-random instances were created for every ratio between the number of variables and the number of clauses between 3.5 and 6.0 in steps of 0.1 (e.g. a ratio of 4.0 means taht there are exactly four times more clauses than variables). This leads to a total of 1.3 million SAT instances whose satisfiability was checked using the complete MiniSat software solver engine. For each configuration of the number of variables and the number of clauses, the number of satisfiable and unsatisfiable instances was recorded.

Unfortunately, after executing these experiments, the first version SAT instance generator described in Section 3.3.1 which was used to generate the pseudo-random SAT instances, proved not to generate a reasonably uniform probability distribution of the variables leading to highly unprecise results shown in Section 5.2.3. However, the results were precise enough to get an idea of the location of the phase transition point for smaller instances up to 100 variables. Therefore the next step was, to investigate the behaviour of the hardware SAT solver engine on larger sized SAT instances (compared to the previous experiments) which are located around the phase transition point.

4.2.2 Satisfiability related experiments in hardware

To provide higher quality reference data for the following exepriments, new instances were generated using the second version of the SAT instance generator described in Section 3.3.1. The number of variables for this and in fact all experiments following was fixated to 100. On the one hand, this number of variables is sufficiently high to give good experiment results about the behaviour of the hardware SAT solver at least on mid-sized SAT instances. On the other hand, this number of variables leaves enough room on the FPGA device to carry additional measurement logic as well as future extensions to the SAT solver logic itself. This way a standard set of instances was generated consisting of a total of 700 instances. The ratio of clauses to variables was chosen being 3.7 to 4.3 in steps of 0.1 leading to 100 pseudo-random SAT instances per configuration. However, most experiments (including this one) use only a subset of this standard test set (which is also included on the accompanying CD-ROM), because the compilation time of the test cases took up to 10 minutes for some experiments.

Unfortunately, the basic probability driven SAT solver engine proved to perform very badly on the generated instance only being able to solve only about 2% of the satisfiable instances. This later proved to be caused mainly by the earlier mentioned toggling probability of $1/3$ being still far to high for reasonably sized experiments and the randomisation engine containing severe problems regarding statistical dependencies.

4.3 Globally probability driven circuits

The first step in engaging the previously mentioned problems were experiments on a subset consisting of 20 SAT instances of ratio 3.7. These were tested using the basic probability driven circuit using three different toggling probabilities of $1/2$, $1/3$ and $1/4$, repsectively. The experiments carried out showed a significantly better performance using a probability of $1/2$ while being unable to solve any instance using a probability of $1/4$. Since this behaviour was absolutely contrary to the expected behaviour, this led to a review of all involved parts of the VHDL library documented in Section 3.2.3. While reviewing the term evaluator module used the bug in the term evaluator module mentioned in this section was discovered and fixed. After fixing this bug the performance significantly improved and the behaviour of the circuit was much closer to the expectations.

Since it was likely that the probability factors giving optimal performance were dependant on the actual number of variables and clauses participating in the SAT instance, a basic formula for the calculation of a base toggling priority P_b was defined with n being the number of variables and c being the number of clauses, assuming a fixed clause length of 3:

$$P_b := \frac{1}{\frac{3c}{n}}$$

Since $3c/n$ is the average number of occurencies of a single variable in a pseudo-randomly generated SAT instance in 3CNF, the idea behind this formula was a linear toggling probability regarding the fraction of clauses a variable participates in which are unsatisfied (e.g. if a variable participates only in satisfied clauses, it should never be toggled, if it participates only in unsatisfied clauses it should always be toggled). This is of course only an approximation since most variables do not occur exaclty $3c/n$ times in an arbitrary instance.

4.3.1 Probability factor experiments

The next step during the experiments was testing the fixed circuitry with the derived probability. Since the derived probability was less than 0.1 for the selected SAT instances of the standard set (50 instances of ratio 3.7), the circuit was also run using probabilities derived by multiplying the calculated base probability with factor between 1.0 and 4.0 in steps of 0.5. The results of these experiments are shown in Section 5.2.2.

It turned out that the calculated base probability gave very good performance on some instanes, while the multiplied probabilities gave good performance on some other instances. Since the average performance of the circuit was still rather disappointing and the circuit even failed to solve several instances depending on the probability multiplier used, another design review of the SAT solver circuitry was started.

4.3.2 Pseudo-random number generators

During a discussion in one of the project meetings, the concern came up, that the simple randomisation engine currently used could suffer from statistical dependencies between the bits run through the selection bit register. Another point of concern was the large number of clock cycles a single bit takes for traveling through the whole register until being discarded and the number of toggling decisions it influences on its way through the register (e.g. regarding the SAT instances used in the previous experiments, the number of clock cycles a generated bit remains in the selection register was $3c = 3 \cdot 370 = 1110$).

To tackle possible problems with the randomisation engine, two modified versions of the randomisation system were implemented. In the first step the LFSR used to generate the input bits for the probability gating logic was parallelised to generate 10 fresh bits every clock cycle. Since the probability gating logic documented in Section 3.2.3 reduces ten bits in each clock cycle to a single bit following the preconfigured probability distribution, this way statistical dependencies between the input bits of the probability gating logic were reduced to the level implied by the LFSR used. This modification heavily improved the average performance of the hardware SAT solver and enabled it to solve instances the circuits using the old randomisation engine were unable to solve.

The second modification introduced aimed at the travel time of the bits in the selection bit register. The single LFSR previously used was replaced by an array of 10 LFSR starting with different seeds each producing 10 fresh bits every clock cycle. The bits generated by each of these LFSRs was processed by a dedicated probability gating module leading to the generation of 10 new selection bits each clock cycle. These 10 bits were concatenated and fed into the selection bit register reducing the travel time of the single bits by a factor of 10. This way the number of toggling decisions each bit influences was also reduced by a factor of 10. This variant of the SAT circuit was the first variant able to solve all 50 SAT instances and it also was the first hardware engine giving an average performance lying significantly over the performance of the MiniSat software solver (even if the fact is taken into account, that the hardware engine - even if integrated into an ASIC - cannot be clocked as fast as the pipelines of the Pentium IV CPU used to acquire the reference data). Section 5.2.4 shows results of the experiments along with a discussion of the effects of randomness to the circuit.

Since the calculated base probability still had no experimental evidence of giving optimal performance, the experiments using different probability multipliers were repeated using the modified SAT circuitry. This time, probability multipliers between 0.75 and 2.5 where tested in steps of 0.25 with an additional multiplier of 0.875 being evaluated. All probabilites tested managed to produce a shortest runtime for at least one instance. However, the average performance of the circuit was decreasing for all probability multipliers over 1.0. Starting with a factor of 1.75, the circuit was even unable to solve certain instances. Surprisingly, the performance increased reducing the factor slightly below 1.0 with a factor of 0.875 giving more than twice the average performance of the base probability. However, reducing the multiplier further to 0.75 gave only half average performance compared to the base probability. Detailed results of the experiments are discussed in Section 5.2.4.

4.3.3 Simulated annealing

To further improve the performance of the SAT solver engine, the idea came up to use a simulated annealing approach to dynamically calculate the probability used to toggle a specific variable. This was implemented by modifying the probability gating logic. The basic idea was to start the solving process with a higher probability and to exponentially "cool the process down" during the first s clock cycles. This was achieved by reading probability boost values from a preconfigured table which got added to the base probability dependant on the number of clock cycles the circuit already run through. The starting probability was calculated as

$$P_s := 0.875 \cdot P_b + \omega \cdot 0.875 \cdot P_b$$

with ω being a preconfigured boost factor exponentially decreasing during the first s clock cycles until it reaches 0. Experiments using boost factors between 0.25 and 1.25 in steps of 0.25 were carried out using values of s of about 5000 and 10000, respectively. The formula used to precalculate the boost factor for a specific clock cycle i is (λ and μ are constants, c is the number of clauses and n the number of variables):

$$\omega_i := \frac{\lambda}{\frac{3c}{n}} \cdot e^{-\frac{ci}{n\mu}}$$

In some cases the performance reached was higher than that reached by the previously discussed circuit variant using a probability factor of 0.875 but in 48% of the testruns, none of the circuits using the simulated annealing technique was able to give better performance compared to the circuit not using simulated annealing. Detailed results of the experiments can be found in Section 5.2.6 along with a discussion of the basic idea behind the simulated annealing approach and possible reasons for its bad performance. Because of these rather disappointing results and because of the limited time left in the porject schedule, the decision was taken to drop the simulated annealing approach.

4.3.4 Runtime variance experiments

All experiments carried out so far were measured using only a single run per SAT instance using a constant seed applied to the randomisation engine. To get more meaningful data regarding the statistical behaviour of the SAT solver engine, the SAT circuitry described in Section 4.3.2 was modified to be able to run a total of 256 consecutive testruns on the same instance and record the number of clock cycles needed to find a solution by each iteration. The modifications done to the components of the SAT support circuitry are documented in Section 3.2.3.

The results of these testruns can be found in Section 5.2.5 along with a discussion of the statistical distribution of the runtimes using different seeds. Unfortunately, during these experiments, the period related problems with the 40-bit LFSR became apparent because in many of the testruns the result timings became periodic. However, since these periods are reasonable large compared to the number of testruns per instance, the data generated is still usable to do meaningful analysis about the statistical distribution of the runtimes.

For statitsical comparision the 50 SAT instances used in this experiment were also run through the incomplete WalkSAT software solver that employes a randomised nieghborhood search strategy. This search strategy is different from the highly parallelised search strategy used by the hardware SAT solver engine but it is one of the closest comparable software search strategies compared to the circuit used. The measurement of the timing of the WalkSAT solver were carried out according to Section 3.3.3.

4.4 Locally probability driven circuits

During the experiments with the simulated annealing technique, another idea came up to heavily modify the SAT solver design used so far. The globally probability driven SAT solvers all together have the problem that they do not take into account the actual number of occurencies of a particular variable in the SAT instance analysed. In almost all cases, the number of occurencies of most variables will obviously not match the statistical expectancy. Therefore the idea came up the move the logic doing the toggling decisions for the variables from the term evaluators to the variable source modules. The basic design used in these experiments counted the number of clauses a particular variable was wrong in and compared it to the total number of occurencies of that particular variable in the SAT instance investigated (which was known by the variable sources by preconfiguring it during code generation). The quotient of the number of unsatisified clauses and the total number of clauses the variable particiaptes in was used as the probability of toggling it.

The resulting circuit gave excellent performance on a couple of the 50 instances used in the previously described experiments but the average performance was comparable to that of the globally probability driven ciruit using a probability factor of 1.0. Results of the basic experiments carried out with this circuit can be found in Section 5.2.7. The behaviour of this type of circuit was not investigated further due to a number of reasons:

- Each variable source requires its own randomisation engine making it nearly impossible to express it using compact logic while keeping a good approximation of the described toggling probability.

- The routing of the variable signals gets more complex making it harder to implement the circuit in an universal ASIC.
- The completely changed circuit design would have required significant additional experimentation time to come up with meaningful figures about its behaviour which was not available.
- Since the average performance of the basic experiment was not significantly higher than that of the globally probability driven circuit it was considered to be of greater value to invest the remaining time available for the project in the analysis of the statistical behaviour of the globally probability driven circuit (see Section 4.3.4).

5 Analysis of results

5.1 Asynchronous circuits

As already stated in Section 4.1.1, the asynchronous circuit type proved to be heavily uncontrollable even for smaller SAT instances consisting of only 10 variables. Appendix D.1 shows tables containing the aggregated measurement data collected by the support circuitry ordered by the number of clauses participating in the SAT instances.

Despite the uncontrollable and comparatively poor performance shown by the asynchronous circuits, the tables also show, that the detection of a found solution by the circuit itself is heavily unreliable since multiple instances which were essentially solved by the circuit were not discovered as solved. The cause of this were probably floating signal levels in the asynchronous part of the circuits observable through the oscilloscope.

To be able to do meaningful research in this area of asynchronous circuits it would be necessary to have full control over the circuit layout on the FPGA chip. It would be even better to have some sort of structured ASIC available which has fixed variable sources and term evaluators and allows for the configuration of the signal flow between the different components. Since the output produced by the Altera compiler provided only partial, hardly analysable knowledge about the layout of the circuits on the FPGA chip, the only really meaningful result extractable from these early experiments with fully combinational logic is, that the equipment avaialble is not suitable for their proper analysis. Therefore all following experiments were focussed on synchronous circuits as already stated previously.

5.2 Synchronous circuits

5.2.1 Fully deterministic circuits

The fully deterministic variants of the synchronous circuit type were the first ones investigated. As the exepriments described in Section 4.1.2 showed very poor performance this decision was quickly taken to move on to probability driven circuits. In fact, the probability driven circuits investigated during the following experiments were fully deterministic as well, since the started operation of their randomisation engines using preconfigured seeds, mainly to be able to reproduce experiments in a fully defined environment. However, if implemented in structured ASICs, the design of the randomisation engines would obviously being changed to an effectively random bit source, for example based on temperature or radiation sensors.

As conjectured early and proved by the later experiments, the reason for the original fully deterministic approach not to work even on small SAT instances is, that this approach is flipping far to many variables each clock cycle. Even if taking into account, that each clause participating in the SAT instance has only a probability of $1/8$ assuming 3CNF and a random state in the search process, this means, that the fully deterministic circuit will toggle a variable with a probability of

$$P_t := \frac{1}{8} \cdot \frac{3c}{n}$$

with c being the number of clauses and n being the number of variables participating in the SAT instances (therefore, $3c/n$ is the average number of clauses each variable participates in). Applied to the SAT instance configurations having 100 variables and 370 clauses, which were widely used during this project, this means that each variable toggles with a probability of 1.3875. This effectively means that the circuit toggles almost all variables in each clock cycle never getting even close to a solution. This conjecture is fortified by the results shown in Appendix D.1.

5.2.2 Globally probability driven circuits

The first basic globally probability driven circuits investigated reduced the probability of an arbitrary variable toggling by introducing the selection bit register described in Section 4.1.3. Since the toggling probability in the early experiments with this technique was set to $1/3$, this implies that in the average case, each unsatisfied clause randomly picks one of its participating variables and toggles it. This implies that the probability for a variable toggling is now

$$\tilde{P}_t := \frac{1}{8} \cdot \frac{c}{n}$$

which proved to provide good performance on the small instances investigated. Appendix D.1 shows the results of these experiments. If taking into account the erroneous term evaluator modules used these times, the results would probably be slightly different. The problem with these term evaluator modules was that, instead of giving a probability of $m/3$ for a variable to toggle with m being the number of unsatisfied clauses, it participates in, the modules gave a probability of approximately $1/3^m$ which heavily reduced the toggling probabilities especially in the experiments involving many clauses. However, since later experiments showed that a generic toggling probability of $1/3$ is far too high for larger SAT instances, this may have even helped the search process in this case (the smaller instances were not tested again using the fixed logic). Appendix D.2 shows a summary of the results of these experiments.

Ongoing experiments with this circuit types, which are documented in Section 4.3, showed that the basic probabilistic circuit with a toggling base probability of $1/3$ gives poor performance as the size of the analysed instances grows. Experiments with SAT instances consisting of 100 variables and 370 showed that a toggling base probability of $1/4$ gives higher performance on these SAT instances. Therefore it was conjectured that the optimal probability for toggling a variable is dependant upon the number of variables and clauses participating in the SAT instance being analysed.

Because of this an experimental formula for the base probability was defined which depends on these two paramters as described in Section 4.3. However, the following experiments using this formula for the calculation of the base probability showed, that best average performance results are achieved by a probability which was slightly below the one calculated by this formula as can be seen in Appendix D.3.

The suboptimality of the proposed probability function may have different reasons. On the one hand does this formula assume, that each variable occurs in the same number of clauses which is not the case in randomly generated SAT instances. On the other hand, the linear function used may not be optimal and it may be beneficial to use an exponential function for the summing of the probabilities. Because of the assumption regarding the number of variable occurrencies, the simple derivation function used to calculate the base priority has the problem, that it reaches a toggling probability of 1 as soon as a variable appears in $3c/n$ unsatisfied clauses which is to early for frequently occurring variables and to late for infrequently occurring variables. This fact led to the idea of having locally probability driven circuits described in Section 4.4 and Section 5.2.7.

5.2.3 Phase transition points

The software based experiments regarding the location of the satisfiability/unsatisfiability phase transition were not only done to find hard instances to save logic resources on the FPGA device as described in Section 4.2.2. Another reason for these experiments was to study the dependancy of the phase transition point of the number of variables participating in the generated instances. However, as documented in Section 4.2.1 this aim was failed due to erroneously generated reference data. Due to the high time consumption of the experiments it was decided to move on in the project and to not repeat the epxeriments to get more reliable data since the computed data was at least precise enough to settle future experiments around the phase transition point.

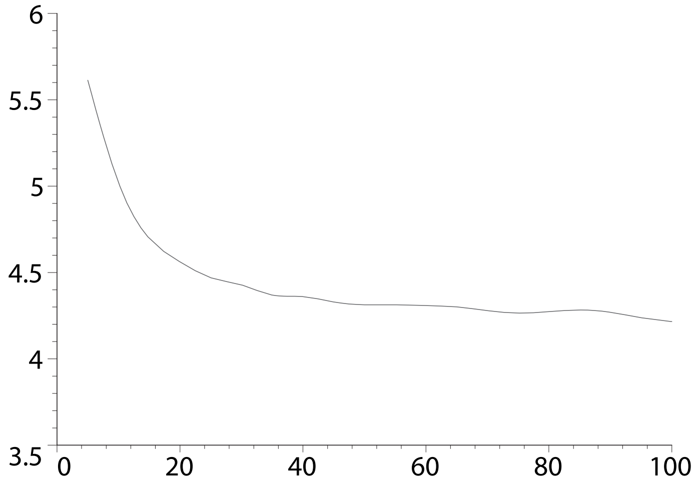

Figure 5.1: Location of phase transition point (Y-axis) depending of the number of participating variables (X-axis)

Figure 5.1 on page 69 shows the phase transition curve computed from the experiment results which are shown in Appendix D.6. Phase transition locations for variable number above 100 are not graphed because the experimental results for larger variable counts are not meaningful. The higher phase transition locations regarding very small SAT instances are most likely caused by the fact that 3CNF-SAT instances with only a small number of participating variables need to reach a certain number of participating clauses (dependant on the actual number of variables), before unsatisfiable instances are even possible.

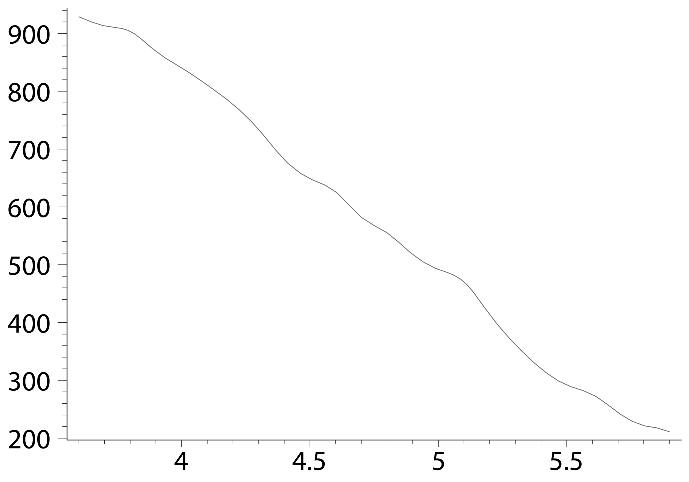

Figure 5.2: Fraction of satisfiable random instances consisting of 10 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis)

Another interesting aspect is, that the size of the phase transition area rapidly declines with an increasing number of participating variables. Figure 5.2 on page 69 shows a comparatively large phase transition area for pseudo-randomly generated SAT instances consisting of 10 variables whereas Figure 5.3 on page 70 and Figure 5.4 on page 70 show declining area sizes for 50 and 100 participating variables.

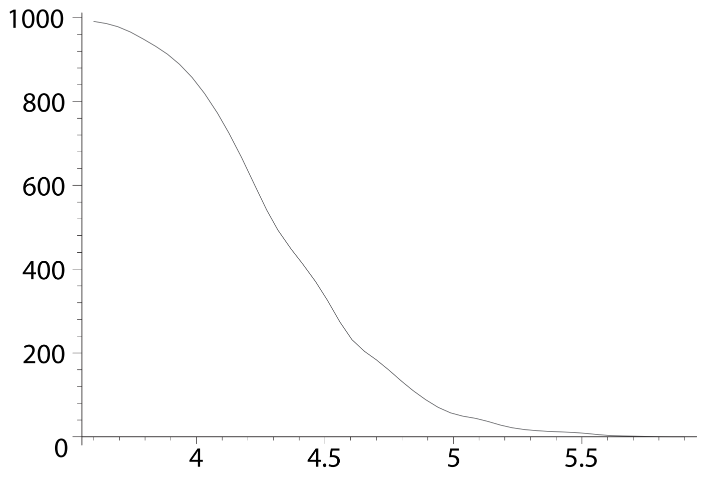

Figure 5.3: Fraction of satisfiable random instances consisting of 50 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis)

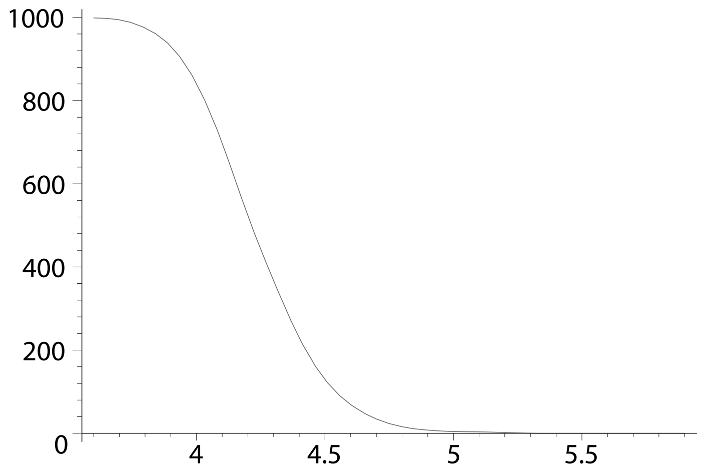

Figure 5.4: Fraction of satisfiable random instances consisting of 100 variables (Y-axis) regarding ratios lying in the phase transition area (X-axis)

5.2.4 Effects of randomness

The experiments described in Section 4.3.2 proved that a strong randomisation engine is absolutely crucial for the performance of the whole SAT solver engine. The randomisation engine was based on linear feedback shift registers from the beginning on because this type of pseudo-random number generator logic is implementable especially compact in FPGAs as well as in ASICs. In the latter case it would nonetheless be advisable to replace this form of pseudo-randomisation by a real hardware randomisation engine (e.g. based on temperature or radiation sensors) or at least a hybrid form of deterministic and non-deterministic randomisation logic.

As the experiments comparing different randomisation engines showed it can be quite hard to produce a LFSR based randomisation engine with good statistical properties. Results of the experiments are shown in Appendix D.4 and Appendix D.5, respectively. Especially if the output of a shift register based randomisation engine is passed through some sort of reduction function (in this case the probability gating logic), the situation can get even worse, because the reduction function may eventually map different series of input values to identical series of output values.

Even if the statistical properties of the randomisation engine are sufficiently good, there can be other problems which might not necessarily be apparent at first glance. During the experiments regarding the statistical runtime behaviour of the hardware SAT solver described in Section 4.3.4 it became apparent, that the randomisation engine used had a significantly shorter period than expected during its design which is shown in Section 5.2.5. The first conjecture was that the shorter periods are produced by the array design linking the outputs of 10 $40 - bit$ LFSRs of the same type. However, later reinspection of the implementation of the single LFSRs brought up the actual reason for the short periods.

The LFSR implementation serving as core component of all randomisation engines used in the globally probability driven circuit variants was designed after information found on the Internet. Due to some vagueness about the actual implementation of the linear feedback function generating input bits for the LFSR, the 40-bit LFSR implementation included in the VHDL library documented in Section 3.2.3 proved to have a significantly shorter period than the period stated on the website.

The linear feedback function of the LFSR using taps at positions 19 and 21 of the register can be described as a linear recursion relationship (assuming all operations taking place in $\mathbb{F}_2$) in which s_i is the ith bit generated by the linear feedback function (the following paragraphs abstract from the usage of a XNOR gate instead of a XOR gate because this simplifies the formulas and does not change the final outcome):

$$s_{k+40} = s_{k+19} + s_{k+21}, k \geq 0$$

or equivalently

$$s_{k+19} + s_{k+21} + s_{k+40} = 0, k \geq 0$$

This allows for the definition of the characterisitc polynomial of the LFSR, which is

$$f(x) = x^{19} + x^{21} + x^{40} = \left(x^{19}\right)\left(1 + x^2 + x^{21}\right)$$

Since this characteristic polynomial is obviously not irreducible, the resulting LFSR has multiple disjoint classes of states not necessarily having the same size. It can travel through each of these classes, depending on the seed used, but is unable to cross the boundaries between these classes during normal operation. Therefore the period length of the LFSR depends on the seed used to initially load it. The seeds used in the experiments with the globally probability driven circuit types were retrospectively checked and found to give periods much higer than 2^{30} which is enough for single testruns. However, the batch testruns, whose results are presented in Section 5.2.5, were affected by this issue.

It is strongly recommended that future projects eventually reusing parts of the created VHDL library use the 41-bit LFSR used in the experiments with the locally probability driven circuit type because this LFSR implementation does not have the mentioned problem. To show this, the following points recapitulate some facts and definitions from Algebra:

- Every polynomial $f(x)$ with coefficients in $\mathbb{F}_2$ having $f(0) = 1$ divides $x^m + 1$ for some m. The smallest value m for which this fact holds is called the period of $f(x)$.

- An irreducible polynomial of degree n has a period which divides $2^n - 1$.
- An irreducible polynomial of degree n whose period is equal to $2^n - 1$ is called a primitive polynomial.

So for a LFSR of length n to produce the maximum possible period length of $2^n - 1$ the characteristic polynomial must be a primitive polynomial (the maximum period is not 2^n because a LFSR based on XOR-gates cannot leave the all-zero state and a LFSR based on XNOR-gates cannot leave the all-one state).

Since the characteristic polynomial of the 41-bit LFSR provided by the VHDL libarary is

$$g(x) = 1 + x^{38} + x^{41}$$

the period of this LFSR implementation if in fact $2^{41} - 1$ because it can be shown that $g(x)$ is a primitive polynomial.

5.2.5 Statistical distribution of solver runtimes

Since most experiments carried out during the project resulted only in per instance "snapshots" of the performance reached, an expriment was set up to investigate the statistical distribution of the runtimes of the globally probability driven SAT solver engine as described in Section 4.3.4.

The results shown in Appendix D.7 are partially subject to periodical behaviour of the randomisation engine producing even periodical runtimes in many cases. However, since the period lengths are relatively long compared to the total number of testruns per instance (which was set to 256), the results still provide meaningful statistical data.

The recorded performance measurements show very large variances in the runtime required to solve the instances provided. For most instances some of the runs finished after only a few hundered clock cycles. The reason for this is probably that the circuit is coincidently placed in a state close to the solution by the choice of the seed. However, on the other hand, most instances also produced testruns running for a long time until the solution was found. In four cases there were even timeouts because the circuit was unable to find a solution in the given time frame (these four instances were excluded for the statistically discussions below). The standard deviation of the runtimes is close to the average runtime in most cases.

For comparison purposes, the instances testes were also run 256 times through the WalkSAT software SAT solver which is an incomplete randomised SAT solver, just like the hardware engine. However, the actual algorithm implemented by it is quite different in many details. The main purpose of this part of the experiment was to observe whether the hardware SAT solver is subject to the same statistical behaviour as a software SAT solver using a comparable approach for solving SAT instances.

The authors of the WalkSAT software SAT solver published a paper [GSCK00] in which they are discussing the statistical distribution of the runtimes WalkSAT needs to solve random SAT instances of different configurations and how these runtimes can be improved. Of particular interest in this context are so-called heavy tailed probability distributions. These distributions are characterised by a high probability peak close to the point of origin. Moving away from the origin in terms of events the probability rapidly declines forming some sort of "tail". Unlike it is the case with most other distributions, this tail is not asymptotically converging against zero. Because of this, heavy tailed distributions can - from a theoretical point of view - have an infinitely large variance. Detailed discussions of these distributions can be found in the paper mentioned. The next pargraphs focus on comparing the behaviour of the comparatively well investigated WalkSAT solver with the behaviour of the hardware SAT solver engine.

Figure 5.5 on page 73 shows an approximation of the distribution of the runtimes required by the WalkSAT software SAT solver to solve pseudo-randomly generated SAT instances consisting of 100 variables and 370 clauses. The distribution was well as the following distributions was

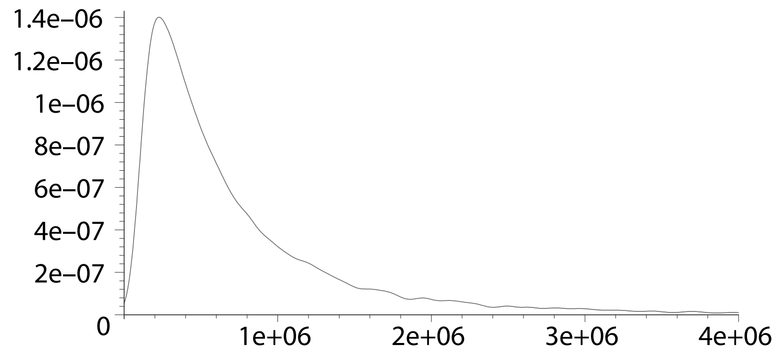

Figure 5.5: Approximated distribution of runtime of WalkSAT solver (X-axis) showing scaled probability approximation (Y-axis)

generated using a kernel density estimation employing a Gaussian kernel function and a frequency-independent smoothening function. The heavy tailed character of the distribution is clearly visible. WalkSAT optionally exploits this distribution when solving larger instances by restarting with a different seed after a processing time threshold is reached.

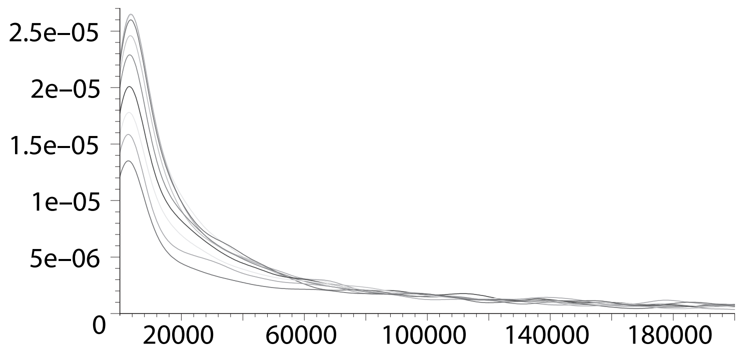

Figure 5.6: Peak area of approximated distribution of runtime of hardware SAT solver (X-axis) showing scaled probability approximation (Y-axis)

Figure 5.6 on page 73 shows an approximation of the peak area of the distribution produced by the hardware SAT solver engine using different probability multipliers. The three closely adjacent peaks belong to the probability factors of 0.75, 0.875 and 1.0 respectively. The curves below them belong to the higher factors in steps of 0.25 in increasing order. The distribution shows the characteristic layout of a heavy tail distribution showing that the randomised hardware SAT solver engine is in fact behaving comparatively to the randomised software SAT solver.

Figure 5.7 on page 74 shows an approximation of the beginning of the tail area of the distribution produced by the hardware SAT solver engine. It shows the typical floating character encountered in the tail areas of heavy tailed distribution. The many spikes visible especially in the right-hand side of the graph are probably produced mainly because of two reasons. On the one hand, the smoothening function used in the kernel density estimation is frequency-independent. This means that the smoothening does not take into account the more chaotic character of the distribution in the tail area. On the other hand many of the spikes might be produced by the periodical parts of the measurement results promoting single events which would not be the case if a better randomisation engine would have been used.

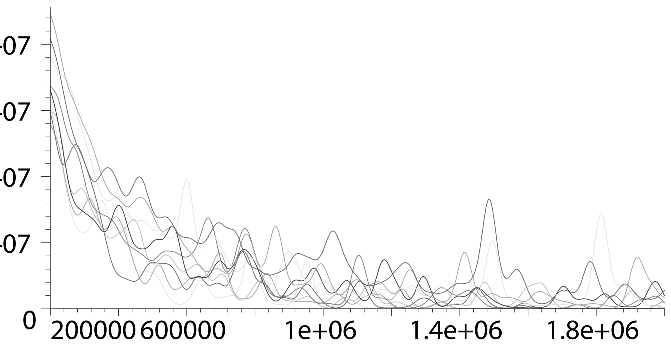

Figure 5.7: Beginning of tail area of approximated distribution of runtime of hardware SAT solver (X-axis) showing scaled probability approximation (Y-axis)

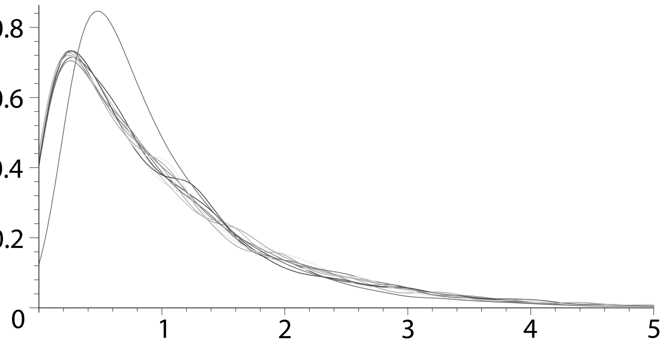

Figure 5.8: Approximated distribution of runtime quotients SAT solvers (X-axis) showing scaled probability approximation (Y-axis)

Since the peaks of the heavy tailed distributions produced by the hardware SAT solver engine for different probability multipliers looked like scaled versions of each other, the idea came up to search for some sort of invariant aspect regarding the distribution produced. As an experiment, the runtime samples recorded from the hardware SAT solver engine as well as those recorded from the WalkSAT solver were normalised by dividing the values in each group of 256 runtime samples belonging to a particular SAT instance and solver configuration by the arithmetic mean of the samples. It was expected to produce different heavy tailed distributions having a peak near 1.0 because this is the expectancy implied by dividing the samples by their airthmetic mean.

Figure 5.8 on page 74 shows an approximation of the resulting distributions. The single free-standing curve is the distribution implied by the WalkSAT samples. Surprisingly, the probability distributions implied by the normalised runtime samples produced by the hardware SAT solver are nearly identical. This leads to the conclusion that the fraction of short or long runs, respectively, observable during multiple runs on a particular SAT instance is not dependent on the global base toggling probability used. In fact, the choice of this probability mainly seems to "scale" the hardness of a particular SAT instance regarding the hardware SAT solver engine.

The distribution also shows that the hardware SAT solver seems to produce more very short runs compared to the software solver. This is likely due to the fact the the hardware solver is able to toggle many variables in parallel in a single operation cycle whereas the algorithm used in WalkSAT only flips a single variable in each iteration. Therefore the hardware SAT solver seems to be able to approach some solutions faster than the WalkSAT solver.

5.2.6 Globally applied simulated annealing

As described in Section 4.3.3, the simulated annealing based approach was unable to noticeably increase the average performance of the hardware SAT solver and was therefore dropped again. Results of the experiments carried out can be found in Appendix D.8.

The reason why the simulated annealing has shown good performance for some instances but degraded performance for others might be found in the actual realisation the the simulated annealing. The selection bits generated using a dynamic probability distribution as described in Section 4.3.3 are still traveling through the selection bit register when the actual generator probability already changed to a lower value. This means, that the effective toggling probability at different positions in the register and therefore for different clauses participating in the instance, respectively, is different. Especially at the beginning of the simulated annealing process, the probability boost rapidly declines, so at the time the first bits having a high probability being set to 1 reach the end of the register, the bits travelling through the start of the register are already having a much lower probability for being set to 1.

Further increasing the speed the bits run through the selection bit register would reduce this problem but this is not a real solution since the problem will reoccur when scaling the circuit to larger instances because the possible speed the selection bits can be run through the register is limited.

5.2.7 Locally probability driven circuits

The locally probability driven circuits described in Section 4.4 were mainly based on the idea to take the actual distribution of the variables in the SAT instance into account rather than other the theoretical average number of occurencies. Like the simulated annealing approach, this circuit type showed performance imporvements for a couple of SAT instances tested but was unable to increase the average performance (in fact the average performace was cut to half compared to the globally probability driven approach). Appendix D.9 shows results for some experiments done with this circuit type.

Because of the various reasons outlined in Section 4.4, this approach was dropped as well as the simulated annealing approach. Unfortunately, the small amount of gathered measurement data makes it impossible to come up with meaningful conclusions about the behaviour of this circuit type. Further investigation including batch test series would be necessary to get presentable results in this direction. However, this would only make sense, if a way would be found to compactly implement this circuit type in a structured ASIC because otherwise the compilation time of the circuit into a FPGA device would be required to be taken into account. This would make this kind of circuit only suitable for very special cases involving large search times compared to the necessary synthesis time.

6 Conclusion and future work

Recapitulating the research and development carried out during the project it could be shown that the techniques proposed in [COP06] are suitable to speed up the computation of SAT problems in hardware by a full order of magnitude. Unfortunately, the original idea of investigating the behaviour of the asynchronous circuit variants and their behaviour in relation to the Church-Turing Hypothesis had to be dropped mainly due to the lack of necessary equipment and the applying time restrictions for the project.

However, the various synchronous variants of the hardware SAT solver engine, which were developed, as well as the experimentation infrastructure built during the project, give plenty of room for future research in this area. Especially the randomisation aspects as well as the emerging heavy tailed runtime distributions shared by the hardware solvers as well as existing software solvers seem to be of particular importance when trying to improve the performance of SAT solvers belonging to this class of algorithms.

Some topics especially interesting for future research include the development of efficient strategies to exploit the heavy tailed nature of the emerging runtime distributions. As shown in [GSCK00], there are many ways to improve serialised software based algorithms based on the assumption of having a heavy tailed runtime distribution. It would be interesting to explore the possibilities to apply the proposed concepts to the parallelised hardware based algorithms and to research new ways of exploitation of these distributions.

Another area of eventual improvement possibilities consists of the inclusion of additional heuristics into the still comparatively simple hardware algorithm. Eventually it would be possible to parallelise some of the already well understood heursitics used in software base complete as well as incomplete SAT solver engines to be applied to the hardware SAT solver in an efficient and logic-saving way.

Large-scale statistical analysis of the observed phase transition phenomena would be possible either directly in hardware or by a software simulation. This way it could be explored whether SAT solvers operating in a highly parallelised way are subject to the same complexity related behaviour as more serialised algorithms. Research in this area could be combined with research on SAT instances originating from specific problem domains (implying specific instance structures) as well as the investigation of related NP-complete problems like graph colouring which are experiencing similiar phase transition phenomena as shown in [Wal02a].

Finally, the asynchronous variants of the SAT solver circuits could be reapproached using appropriate laboratory equipment to gain insights in the behaviour of complex systems not belonging to the class of state machine like systems as mentioned in the introduction. This topic would give much room for fundamental research since these kinds of hardly modelable systems are still far away from being fully understood.

Bibliography

[Alf96] Peter Alfke. Xilinx application note 052 - efficient shift registers, lfsr counters, and long pseudo-random sequence generators. Technical report, Xilinx, Inc., July 1996.

[Alt07] Altera Corporation. *Cyclone Device Handbook, Volume 1*, 2007.

[BCMS92] Peter Barrie, Paul Cockshott, George J. Milne, and Paul Shaw. Design and verification of a highly concurrent machine. *Microprocess. Microsyst.*, 16(3):115–123, 1992.

[CKT91] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, 1991.

[COP06] Paul Cockshott, John O'Donnell, and Patrick Prosser. Experimental investigation of computability bounds in adaptive combinational circuits. Technical report, Department of Computing Science, University of Glasgow, July 2006.

[DHN05] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A clause-based heuristic for sat solvers. Technical report, School of Computer Science, Tel Aviv University, 2005.

[ECGP99] Jr. Edmund, M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT 2003*, LNCS 2919, pages 502–518, 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modelling and Computation*, 2:1–25, 2006.

[GMPW96] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Thirteenth National Conference on Artificial Inteleigence (AAAI'96)*, pages 246–252, 1996.

[GN02] E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *Design Automation and Test in Europe*, pages 142–149, 2002.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.

[Hay97] Brian Hayes. Can't get no satisfaction. *American Scientist*, 85(2):108–112, March 1997.

[Hay03] Brian Hayes. On the threshold. *American Scientist*, 91(1):12–17, January 2003.

[Int06a] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, November 2006.

[Int06b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, November 2006.

[Kau93] Stuart A. Kauffman. *The Origins of Order*. Oxford University Press, 1993.

[KS96] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Thirteenth National Conference on Artificial Inteleigence (AAAI'96)*, pages 1194–1201, 1996.

[KSTW05] Philip Kilby, John Slaney, Sylvie Thiebaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *AAAI-2005*, 2005.

[MFM04] Y.S. Mahajan, Z. Fu, and S. Malik. Zchaff 2004: An efficient sat solver. In *SAT 2004: Theory and Applications of Satisfiability Testing*, LNCS 3542, 2004.

[MKST99] R. Monasson, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic phase transitions. *Nature*, 400, 1999.

[OR04] John T. O'Donnell and Gudula Rünger. Derivation of a logarithmic time carry lookahead addition circuit. *Journal of Functional Programming*, 14(6):697–731, 2004.

[SCB96] Paul Shaw, Paul Cockshott, and Peter Barrie. Implementation of lattice gases using fpgas. *The Journal of VLSI Signal Processing*, 12(51):66, 1996.

[SE05] Niklas Sörensson and Niklas Eén. Minisat v1.13 - a sat solver with conflict-clause minimization. Technical report, Chalmers University of Technology, Sweden, 2005.

[Sys05] System Level Solutions, Inc. *UP3-1C6 Education Kit, Reference Manual, Cyclone Edition*, April 2005.

[Wal02a] Toby Walsh. 2+p-col. Technical report, Cork Constraint Computation Center, University College Cork, 2002.

[Wal02b] Toby Walsh. From p to np: Col, xor, nae, 1-in-k, and horn sat. In *AAAI-2002*, 2002.

Appendix A

Infrastructure tools

A.1 Small instance unsatisfiability search tool

```
#include <stdio.h>

bool evaluate(unsigned int discardTerm1, unsigned int discardTerm2, unsigned int discardTerm3,
     unsigned int discardTerm4, unsigned int literalMask1, unsigned int literalMask2, unsigned
     int literalMask3, unsigned int literalMask4, bool stateA, bool stateB, bool stateC, bool
     stateD) {
        bool result1;
        bool result2;
        bool result3;
        bool result4;

        switch (discardTerm1) {
                case 0: result1 = (((literalMask2 >> 3) == 1) ^ stateB) | (((literalMask3 >> 3)
                     == 1) ^ stateC) | (((literalMask4 >> 3) == 1) ^ stateD); break;
                case 1: result1 = (((literalMask1 >> 3) == 1) ^ stateA) | (((literalMask3 >> 3)
                     == 1) ^ stateC) | (((literalMask4 >> 3) == 1) ^ stateD); break;
                case 2: result1 = (((literalMask1 >> 3) == 1) ^ stateA) | (((literalMask2 >> 3)
                     == 1) ^ stateB) | (((literalMask4 >> 3) == 1) ^ stateD); break;
                case 3: result1 = (((literalMask1 >> 3) == 1) ^ stateA) | (((literalMask2 >> 3)
                     == 1) ^ stateB) | (((literalMask3 >> 3) == 1) ^ stateC); break;
        }

        switch (discardTerm2) {
                case 0: result2 = ((((literalMask2 >> 2) & 1) == 1) ^ stateB) | ((((
                    literalMask3 >> 2) & 1) == 1) ^ stateC) | ((((literalMask4 >> 2) & 1) == 1)
                     ^ stateD); break;
                case 1: result2 = ((((literalMask1 >> 2) & 1) == 1) ^ stateA) | ((((
                    literalMask3 >> 2) & 1) == 1) ^ stateC) | ((((literalMask4 >> 2) & 1) == 1)
                     ^ stateD); break;
                case 2: result2 = ((((literalMask1 >> 2) & 1) == 1) ^ stateA) | ((((
                    literalMask2 >> 2) & 1) == 1) ^ stateB) | ((((literalMask4 >> 2) & 1) == 1)
                     ^ stateD); break;
                case 3: result2 = ((((literalMask1 >> 2) & 1) == 1) ^ stateA) | ((((
                    literalMask2 >> 2) & 1) == 1) ^ stateB) | ((((literalMask3 >> 2) & 1) == 1)
                     ^ stateC); break;
        }

        switch (discardTerm3) {
                case 0: result3 = ((((literalMask2 >> 1) & 1) == 1) ^ stateB) | ((((
                    literalMask3 >> 1) & 1) == 1) ^ stateC) | ((((literalMask4 >> 1) & 1) == 1)
                     ^ stateD); break;
                case 1: result3 = ((((literalMask1 >> 1) & 1) == 1) ^ stateA) | ((((
                    literalMask3 >> 1) & 1) == 1) ^ stateC) | ((((literalMask4 >> 1) & 1) == 1)
                     ^ stateD); break;
                case 2: result3 = ((((literalMask1 >> 1) & 1) == 1) ^ stateA) | ((((
                    literalMask2 >> 1) & 1) == 1) ^ stateB) | ((((literalMask4 >> 1) & 1) == 1)
                     ^ stateD); break;
                case 3: result3 = ((((literalMask1 >> 1) & 1) == 1) ^ stateA) | ((((
                    literalMask2 >> 1) & 1) == 1) ^ stateB) | ((((literalMask3 >> 1) & 1) == 1)
                     ^ stateC); break;
        }

        switch (discardTerm4) {
                case 0: result4 = (((literalMask2 & 1) == 1) ^ stateB) | (((literalMask3 & 1)
                    == 1) ^ stateC) | (((literalMask4 & 1) == 1) ^ stateD); break;
                case 1: result4 = (((literalMask1 & 1) == 1) ^ stateA) | (((literalMask3 & 1)
                    == 1) ^ stateC) | (((literalMask4 & 1) == 1) ^ stateD); break;
                case 2: result4 = (((literalMask1 & 1) == 1) ^ stateA) | (((literalMask2 & 1)
                    == 1) ^ stateB) | (((literalMask4 & 1) == 1) ^ stateD); break;
                case 3: result4 = (((literalMask1 & 1) == 1) ^ stateA) | (((literalMask2 & 1)
                        == 1) ^ stateB) | (((literalMask3 & 1) == 1) ^ stateC); break;
        }

        return result1 && result2 && result3 && result4;
}

void printSAT(unsigned int discardTerm1, unsigned int discardTerm2, unsigned int discardTerm3,
     unsigned int discardTerm4, unsigned int literalMask1, unsigned int literalMask2, unsigned
     int literalMask3, unsigned int literalMask4) {
        switch (discardTerm1) {
                case 0: printf("(%s,%s,%s)␣", ((literalMask2 >> 3) == 1) ? "-B" : "B", ((
                     literalMask3 >> 3) == 1) ? "-C" : "C", ((literalMask4 >> 3) == 1) ? "-D" :
                     "D"); break;
                case 1: printf("(%s,%s,%s)␣", ((literalMask1 >> 3) == 1) ? "-A" : "A", ((
                     literalMask3 >> 3) == 1) ? "-C" : "C", ((literalMask4 >> 3) == 1) ? "-D" :
                     "D"); break;
                case 2: printf("(%s,%s,%s)␣", ((literalMask1 >> 3) == 1) ? "-A" : "A", ((
                     literalMask2 >> 3) == 1) ? "-B" : "B", ((literalMask4 >> 3) == 1) ? "-D" :
                     "D"); break;
                case 3: printf("(%s,%s,%s)␣", ((literalMask1 >> 3) == 1) ? "-A" : "A", ((
                     literalMask2 >> 3) == 1) ? "-B" : "B", ((literalMask3 >> 3) == 1) ? "-C" :
                     "C"); break;
        }

        switch (discardTerm2) {
                case 0: printf("(%s,%s,%s)␣", (((literalMask2 >> 2) & 1) == 1) ? "-B" : "B",
                     (((literalMask3 >> 2) & 1) == 1) ? "-C" : "C", (((literalMask4 >> 2) & 1)
                     == 1) ? "-D" : "D"); break;
                case 1: printf("(%s,%s,%s)␣", (((literalMask1 >> 2) & 1) == 1) ? "-A" : "A",
                     (((literalMask3 >> 2) & 1) == 1) ? "-C" : "C", (((literalMask4 >> 2) & 1)
                     == 1) ? "-D" : "D"); break;
                case 2: printf("(%s,%s,%s)␣", (((literalMask1 >> 2) & 1) == 1) ? "-A" : "A",
                     (((literalMask2 >> 2) & 1) == 1) ? "-B" : "B", (((literalMask4 >> 2) & 1)
                     == 1) ? "-D" : "D"); break;
                case 3: printf("(%s,%s,%s)␣", (((literalMask1 >> 2) & 1) == 1) ? "-A" : "A",
                     (((literalMask2 >> 2) & 1) == 1) ? "-B" : "B", (((literalMask3 >> 2) & 1)
                     == 1) ? "-C" : "C"); break;
        }

        switch (discardTerm3) {
                case 0: printf("(%s,%s,%s)␣", (((literalMask2 >> 1) & 1) == 1) ? "-B" : "B",
                     (((literalMask3 >> 1) & 1) == 1) ? "-C" : "C", (((literalMask4 >> 1) & 1)
                     == 1) ? "-D" : "D"); break;
                case 1: printf("(%s,%s,%s)␣", (((literalMask1 >> 1) & 1) == 1) ? "-A" : "A",
                     (((literalMask3 >> 1) & 1) == 1) ? "-C" : "C", (((literalMask4 >> 1) & 1)
                     == 1) ? "-D" : "D"); break;
                case 2: printf("(%s,%s,%s)␣", (((literalMask1 >> 1) & 1) == 1) ? "-A" : "A",
                     (((literalMask2 >> 1) & 1) == 1) ? "-B" : "B", (((literalMask4 >> 1) & 1)
                     == 1) ? "-D" : "D"); break;
                case 3: printf("(%s,%s,%s)␣", (((literalMask1 >> 1) & 1) == 1) ? "-A" : "A",
                     (((literalMask2 >> 1) & 1) == 1) ? "-B" : "B", (((literalMask3 >> 1) & 1)
                     == 1) ? "-C" : "C"); break;
        }

        switch (discardTerm4) {
                case 0: printf("(%s,%s,%s)\n", ((literalMask2 & 1) == 1) ? "-B" : "B", ((
                     literalMask3 & 1) == 1) ? "-C" : "C", ((literalMask4 & 1) == 1) ? "-D" : "D
                     "); break;
                case 1: printf("(%s,%s,%s)\n", ((literalMask1 & 1) == 1) ? "-A" : "A", ((
                     literalMask3 & 1) == 1) ? "-C" : "C", ((literalMask4 & 1) == 1) ? "-D" : "D
                     "); break;
                case 2: printf("(%s,%s,%s)\n", ((literalMask1 & 1) == 1) ? "-A" : "A", ((
                     literalMask2 & 1) == 1) ? "-B" : "B", ((literalMask4 & 1) == 1) ? "-D" : "D
                     "); break;
                case 3: printf("(%s,%s,%s)\n", ((literalMask1 & 1) == 1) ? "-A" : "A", ((
                     literalMask2 & 1) == 1) ? "-B" : "B", ((literalMask3 & 1) == 1) ? "-C" : "C
                     "); break;
        }
}

void main() {
        for (unsigned int discardSel = 0; discardSel < 256; discardSel++) {
                unsigned int discardTerm1 = discardSel >> 6;
                unsigned int discardTerm2 = (discardSel >> 4) & 0x3;
                unsigned int discardTerm3 = (discardSel >> 2) & 0x3;
                unsigned int discardTerm4 = discardSel & 0x3;
                for (unsigned int literalSel = 0; literalSel < 4096; literalSel++) {
                        unsigned int literalMask1 = literalSel >> 9;
                        unsigned int literalMask2 = (literalSel >> 6) & 0x7;
                        unsigned int literalMask3 = (literalSel >> 3) & 0x7;
                        unsigned int literalMask4 = literalSel & 0x7;

                        bool soluable = false;

                        for (unsigned int valueSel = 0; valueSel < 16; valueSel++) {
                                bool stateA = (valueSel & 0x8) != 0;
                                bool stateB = (valueSel & 0x4) != 0;
                                bool stateC = (valueSel & 0x2) != 0;
                                bool stateD = (valueSel & 0x1) != 0;

                                soluable |= evaluate(discardTerm1, discardTerm2, discardTerm3,
                                    discardTerm4, literalMask1, literalMask2, literalMask3,
                                    literalMask4, stateA, stateB, stateC, stateD);
                        }

                        if (!soluable) {
                                printSAT(discardTerm1, discardTerm2, discardTerm3, discardTerm4
                                    , literalMask1, literalMask2, literalMask3, literalMask4);
                        }
                }
        }
}
```

A.2 Seed generator

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;

namespace GenProbSeed
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length < 2)
            {
                Console.WriteLine("Usage:␣GenProbSeed.exe␣<bit␣count>␣<probability␣for␣one␣bit>
                    ");
                return;
            }

            int bitCount = Int32.Parse(args[0]);
            double probFactor = Double.Parse(args[1]);

            double baseValue = (double)(((long)(1)) << 48);
            baseValue *= probFactor;
            long oneLimit = (long)(Math.Floor(baseValue));

            for (int index = 0; index < bitCount; index++)
            {
                byte[] randomBytes = new byte[6];

                RNGCryptoServiceProvider Gen = new RNGCryptoServiceProvider();
                Gen.GetBytes(randomBytes);

                long randomNumber = randomBytes[0];
                randomNumber = (randomNumber << 8) | randomBytes[1];
                randomNumber = (randomNumber << 8) | randomBytes[2];
                randomNumber = (randomNumber << 8) | randomBytes[3];
                randomNumber = (randomNumber << 8) | randomBytes[4];
                randomNumber = (randomNumber << 8) | randomBytes[5];

                Console.Write((randomNumber <= oneLimit) ? '1' : '0');
            }
        }
    }
}
```

A.3 Simulated annealing stepping table generator

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GenExpTable
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length < 6)
            {
                Console.WriteLine("Usage:␣GenExpTable.exe␣g|s␣<variable␣count>␣<clause␣count>")
                    ;
                Console.WriteLine("␣␣<variables␣per␣clause>␣<global␣factor>␣<exponent␣divisor>"
                    );
                return;
            }

            int variableCount = Int32.Parse(args[1]);
            int clauseCount = Int32.Parse(args[2]);
            int clauseLength = Int32.Parse(args[3]);
            double globalFactor = Double.Parse(args[4]);
            double exponentDivisor = Double.Parse(args[5]);

            double averageOccurencies = ((double)(clauseCount * clauseLength)) / ((double)(
                variableCount));
            double variableClauseRatio = ((double)(clauseCount)) / ((double)(variableCount));

            double baseFactor = globalFactor * ((double)(1024)) / averageOccurencies;
            double exponentFactor = -variableClauseRatio / exponentDivisor;

            int[] values = new int[100001];
            for (int valueIndex = 0; valueIndex <= 100000; valueIndex++)
            {
                values[valueIndex] = (int)(Math.Round(baseFactor * Math.Exp(exponentFactor * ((
                    double)(valueIndex)))));
                if ((values[valueIndex] < 0) || (values[valueIndex] >= 1024))
                {
                    Console.WriteLine("Table␣entry␣" + values[valueIndex].ToString() + "␣
                        exceeds␣valid␣number␣range");
                    return;
                }
            }

            if (args[0].Trim().ToLower().Equals("g"))
            {
                Console.WriteLine("DEPTH␣=␣4096;");
                Console.WriteLine("WIDTH␣=␣16;");
                Console.WriteLine("ADDRESS_RADIX␣=␣HEX;");
                Console.WriteLine("DATA_RADIX␣=␣HEX;");
                Console.WriteLine("CONTENT");
                Console.WriteLine("BEGIN");
                Console.WriteLine();

                uint lastValue = (uint)(values[0]);
                uint currCount = 1;
                uint address = 0;

                for (int rleIndex = 1; rleIndex <= 100000; rleIndex++)
                {
                    if ((values[rleIndex] == lastValue) && (currCount < 63))
                    {
                        currCount++;
                    }
                    else
                    {
                        uint memoryWord = (currCount << 10) | lastValue;
                        Console.WriteLine(address.ToString("X3") + "␣:␣" + memoryWord.ToString(
                            "X4") + ";");
                        lastValue = (uint)(values[rleIndex]);
                        currCount = 1;
                        address++;

                        if (lastValue == 0)
                        {
                            break;
                        }
                    }
                }

                Console.WriteLine(address.ToString("X3") + "␣:␣0000;");
                Console.WriteLine();
                Console.WriteLine("END;");
            }

            if (args[0].Trim().ToLower().Equals("s"))
            {
                int limit = values[0] / 20;
                int steepSequenceLength = -1;
                int totalSequenceLength = -1;

                for (int index = 0; index <= 100000; index++)
                {
                    if ((steepSequenceLength == -1) && (values[index] <= limit))
                    {
                        steepSequenceLength = index;
                    }

                    if (values[index] == 0)
                    {
                        totalSequenceLength = index;
                        break;
                    }
                }

                Console.WriteLine("Maximum␣probability␣boost:␣␣" + values[0].ToString());
                Console.WriteLine("Total␣sequence␣length:␣␣␣␣␣␣" + totalSequenceLength.ToString
                    ());
                Console.WriteLine("High␣boost␣sequence␣length:␣" + steepSequenceLength.ToString
                    ());
            }
        }
    }
}
```

A.4 40-bit LFSR seed checking tool

```
#include <stdio.h>

#define SEED0 0ULL

#define SEEDA 0x9A62DD2287ULL
/*      seed          => "1001101001100010110111010010001010000111"*/
#define SEEDB 0xAC75043ABDULL
/*      seed          => "1010110001110101000001000011101010111101"*/
#define SEEDC 0x7629E20BB3ULL
/*      seed          => "0111011000101001111000100000101110110011"*/
#define SEEDD 0xFD2C8274FCULL
/*      seed          => "1111110100101100100000100111010011111100"*/
#define SEEDE 0xA6C2E4CE5ULL
/*      seed          => "0000101001101100001011100100110011100101"*/
#define SEEDF 0x95DC2030C7ULL
/*      seed          => "1001010111011100001000000011000011000111"*/
#define SEEDG 0x453E5F88E7ULL
/*      seed          => "0100010100111110010111111000100011100111"*/
#define SEEDH 0x40903C3A21ULL
/*      seed          => "0100000010010000001111000011101000100001"*/
#define SEEDI 0xDFF9944C8DULL
/*      seed          => "1101111111111001100101000100110010001101"*/
#define SEEDJ 0x756CF011EBULL
/*      seed          => "0111010101101100111100000001000111101011"*/

void checkseed(unsigned __int64 seed) {
        unsigned __int64 lfsr = seed;
        unsigned __int64 counter = 0;

        do {
                unsigned int tapbit1 = (((unsigned int)(lfsr)) >> 19) & 1;
                unsigned int tapbit2 = (((unsigned int)(lfsr)) >> 21) & 1;
                lfsr = ((lfsr << 1) | ((unsigned __int64)(tapbit1 ^ tapbit2 ^ 1))) & 0
                    xFFFFFFFFFF;
                counter++;
        } while ((counter != 0x100000000ULL) && (lfsr != seed));

        printf("%.10I64X\n", seed);
        printf("%.10I64X\n", lfsr);
        printf("%I64u\n", counter);
        printf("\n");
}

void main() {
        checkseed(SEED0);
        checkseed(SEEDA);
        checkseed(SEEDB);
        checkseed(SEEDC);
        checkseed(SEEDD);
        checkseed(SEEDE);
        checkseed(SEEDF);
        checkseed(SEEDG);
        checkseed(SEEDH);
        checkseed(SEEDI);
        checkseed(SEEDJ);
}
```

A.5 Sleeping tool

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace Sleep
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 1)
            {
                Console.WriteLine("Usage:␣Sleep.exe␣<milliseconds>");
            }
            else
            {
                Thread.Sleep(Int32.Parse(args[0]));
            }
        }
    }
}
```

A.6 SAT instance generator version 1

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;

namespace SATGenerator
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 3)
            {
                Console.WriteLine("Usage:␣SATGenerator.exe␣<clause␣length>␣<variable␣count>␣<
                    clause␣count>");
                return;
            }

            int clauseLength = Int32.Parse(args[0]);
            int varCount = Int32.Parse(args[1]);
            int clauseCount = Int32.Parse(args[2]);
            DateTime timeStamp = DateTime.Now;
            Console.WriteLine("c␣Automatically␣generated␣on␣" + timeStamp.ToShortDateString() +
                 "␣at␣" + timeStamp.ToShortTimeString());
            Console.WriteLine("p␣cnf␣" + varCount.ToString() + "␣" + clauseCount.ToString());

            StringBuilder outputData = null;

            bool done = false;
            while (!done)
            {
                BitArray checkFlags = new BitArray(varCount, false);
                outputData = new StringBuilder();

                for (int index = 0; index < clauseCount; index++)
                {
                    BitArray localCheckFlags = new BitArray(varCount, false);

                    for (int subIndex = 0; subIndex < clauseLength; /* nothing */)
                    {
                        byte[] randomNumber = new byte[2];

                        RNGCryptoServiceProvider Gen = new RNGCryptoServiceProvider();
                        Gen.GetBytes(randomNumber);

                        double rand = Convert.ToDouble(randomNumber[0]);
                        rand *= (((double)(varCount - 1)) / 255.0);

                        int variable = Convert.ToInt32(rand);

                        if (!localCheckFlags.Get(variable))
                        {
                            checkFlags.Set(variable, true);
                            localCheckFlags.Set(variable, true);
                            if (randomNumber[1] >= 0x80)
                            {
                                outputData.Append("-");
                            }
                            outputData.Append((variable + 1).ToString() + "␣");
                            subIndex++;
                        }
                    }

                    outputData.AppendLine("0");
                }

                int checkSum = 0;
                for (int checkIndex = 0; checkIndex < varCount; checkIndex++)
                {
                    if (checkFlags.Get(checkIndex))
                    {
                        checkSum++;
                    }
                }

                done = (checkSum == varCount);
            }

            Console.Write(outputData.ToString());
            //Console.WriteLine("0");
        }
    }
}
```

A.7 SAT instance generator version 2

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;

namespace SATGenerator
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 3)
            {
                Console.WriteLine("Usage:␣SATGenerator.exe␣<clause␣length>␣<variable␣count>␣<
                     clause␣count>");
                return;
            }

            int clauseLength = Int32.Parse(args[0]);
            int varCount = Int32.Parse(args[1]);
            int clauseCount = Int32.Parse(args[2]);

            DateTime timeStamp = DateTime.Now;
            Console.WriteLine("c␣Automatically␣generated␣on␣" + timeStamp.ToShortDateString() +
                  "␣at␣" + timeStamp.ToShortTimeString());
            Console.WriteLine("p␣cnf␣" + varCount.ToString() + "␣" + clauseCount.ToString());

            StringBuilder outputData = null;

            bool done = false;
            while (!done)
            {
                BitArray checkFlags = new BitArray(varCount, false);
                outputData = new StringBuilder();

                for (int index = 0; index < clauseCount; index++)
                {
                    BitArray localCheckFlags = new BitArray(varCount, false);

                    for (int subIndex = 0; subIndex < clauseLength; /* nothing */)
                    {
                        byte[] randomNumber = new byte[6];

                        RNGCryptoServiceProvider Gen = new RNGCryptoServiceProvider();
                        Gen.GetBytes(randomNumber);

                        long randomValue = 0;
                        randomValue = (randomValue << 8) | randomNumber[0];
                        randomValue = (randomValue << 8) | randomNumber[1];
                        randomValue = (randomValue << 8) | randomNumber[2];
                        randomValue = (randomValue << 8) | randomNumber[3];
                        randomValue = (randomValue << 8) | randomNumber[4];
                        byte signIndicator = randomNumber[5];

                        double rand = Convert.ToDouble(randomValue);
                        rand *= (double)(varCount);
                        rand /= (double)(((long)(1)) << 40);

                        int variable = Convert.ToInt32(Math.Floor(rand));

                        if (!localCheckFlags.Get(variable))
                        {
                            checkFlags.Set(variable, true);
                            localCheckFlags.Set(variable, true);
                            if (signIndicator >= 0x80)
                            {
                                outputData.Append("-");
                            }
                            outputData.Append((variable + 1).ToString() + "␣");
                            subIndex++;
                        }
                    }

                    outputData.AppendLine("0");
                }

                int checkSum = 0;
                for (int checkIndex = 0; checkIndex < varCount; checkIndex++)
                {
                    if (checkFlags.Get(checkIndex))
                    {
                        checkSum++;
                    }
                }

                done = (checkSum == varCount);
            }

            Console.Write(outputData.ToString());
            //Console.WriteLine("0");
        }
    }
}
```

Appendix B

VHDL Library

B.1 Term evaluators

B.1.1 Basic term evaluator

```
-- Generic term evaluator component for SAT instances in CNF

library ieee;
use ieee.std_logic_1164.all;

library work;

entity term_evaluator is
  generic (
    clause_length :     integer range 2 to 100 := 3
    );
  port (
    input         : in  std_logic_vector (1 to clause_length);
    wrong_in      : in  std_logic_vector (1 to clause_length);
    wrong_out     : out std_logic_vector (1 to clause_length);
    solved_in     : in  std_logic;
    solved_out    : out std_logic
    );
end term_evaluator;

architecture term_evaluator_architecture of term_evaluator is
  signal     term_result     : std_logic;
  signal     not_term_result : std_logic;
begin
  process(input)
    variable temp_result     : std_logic;
  begin
    temp_result   := input(1);
    for index in 2 to clause_length loop
      temp_result := temp_result or input(index);
    end loop;
    term_result <= temp_result;
  end process;

  not_term_result <= not(term_result);

  process(wrong_in, not_term_result)
    variable temp_wrong : std_logic_vector (1 to clause_length);
  begin
    for index in 1 to clause_length loop
      temp_wrong(index) := wrong_in(index) or not_term_result;
    end loop;
    wrong_out <= temp_wrong;
  end process;

  solved_out <= solved_in and term_result;
end term_evaluator_architecture;
```

B.1.2 Probabilistic term evaluator

```
-- Generic probabilistic term evaluator component for SAT instances in CNF

library ieee;
use ieee.std_logic_1164.all;

library work;
```

```
entity term_evaluator_probabilistic is
  generic (
    clause_length :     integer range 2 to 100 := 3
    );
  port (
    input         : in  std_logic_vector (1 to clause_length);
    wrong_in      : in  std_logic_vector (1 to clause_length);
    wrong_sel     : in  std_logic_vector (1 to clause_length);
    wrong_out     : out std_logic_vector (1 to clause_length);
    solved_in     : in  std_logic;
    solved_out    : out std_logic
    );
end term_evaluator_probabilistic;

architecture term_evaluator_probabilistic_architecture of term_evaluator_probabilistic is
  signal     term_result     : std_logic;
  signal     not_term_result : std_logic;
begin
  process(input)
    variable temp_result     : std_logic;
  begin
    temp_result   := input(1);
    for index in 2 to clause_length loop
      temp_result := temp_result or input(index);
    end loop;
    term_result <= temp_result;
  end process;

  not_term_result <= not(term_result);

  process(wrong_in, wrong_sel, not_term_result)
    variable temp_wrong : std_logic_vector (1 to clause_length);
  begin
    for index in 1 to clause_length loop
      temp_wrong(index) := wrong_in(index) or (not_term_result and wrong_sel(index));
    end loop;
    wrong_out <= temp_wrong;
  end process;

  solved_out <= solved_in and term_result;
end term_evaluator_probabilistic_architecture;
```

B.1.3 Probabilistic term evaluator (buggy)

```
-- Generic probabilistic term evaluator component for SAT instances in CNF
--
-- Probability summing is erroneous, this is just included for completeness

library ieee;
use ieee.std_logic_1164.all;

library work;

entity term_evaluator_probabilistic_buggy is
  generic (
    clause_length :     integer range 2 to 100 := 3
    );
  port (
    input         : in  std_logic_vector (1 to clause_length);
    wrong_in      : in  std_logic_vector (1 to clause_length);
    wrong_sel     : in  std_logic_vector (1 to clause_length);
    wrong_out     : out std_logic_vector (1 to clause_length);
    solved_in     : in  std_logic;
    solved_out    : out std_logic
    );
end term_evaluator_probabilistic_buggy;

architecture term_evaluator_probabilistic_buggy_architecture of
     term_evaluator_probabilistic_buggy is
  signal     term_result     : std_logic;
  signal     not_term_result : std_logic;
begin
  process(input)
    variable temp_result     : std_logic;
  begin
    temp_result   := input(1);
    for index in 2 to clause_length loop
```

```
      temp_result := temp_result or input(index);
    end loop;
    term_result <= temp_result;
  end process;

  not_term_result <= not(term_result);

  process(wrong_in, wrong_sel, not_term_result)
    variable temp_wrong : std_logic_vector (1 to clause_length);
  begin
    for index in 1 to clause_length loop
      temp_wrong(index) := (wrong_in(index) or not_term_result) and wrong_sel(index);
    end loop;
    wrong_out <= temp_wrong;
  end process;

  solved_out <= solved_in and term_result;
end term_evaluator_probabilistic_buggy_architecture;
```

B.2 Variable sources

B.2.1 Basic asynchronous variable source

```
-- Asynchronous variable source component

library ieee;
use ieee.std_logic_1164.all;

library work;

entity variable_source_async is
  generic (
    delay_gates   :     natural := 0
    );
  port (
    wrong_in      : in  std_logic;
    wrong_not_in  : in  std_logic;
    reset         : in  std_logic;
    wrong_out     : out std_logic;
    wrong_not_out : out std_logic;
    var_out       : out std_logic;
    var_not_out   : out std_logic
    );
end variable_source_async;

architecture variable_source_async_architecture of variable_source_async is
  signal wrong_any    : std_logic;
  signal delay_values : std_logic_vector (0 to delay_gates);
  signal new_value    : std_logic;
  signal output_value : std_logic;
begin
  wrong_any <= wrong_in or wrong_not_in;

  process(reset)
  begin
    delay_values(0) <= output_value and reset;

    for index in 1 to delay_gates loop
      delay_values(index) <= delay_values(index - 1) and reset;
    end loop;
  end process;

  new_value    <= delay_values(delay_gates) xor wrong_any;
  output_value <= new_value and reset;

  wrong_out     <= '0';
  wrong_not_out <= '0';

  var_out     <= output_value;
  var_not_out <= not(output_value);
end variable_source_async_architecture;
```

B.2.2 Asynchronous variable source hardened against compiler optimisations

```
-- Asynchronous variable source component
-- Hardened against compiler optimisations

library ieee;
use ieee.std_logic_1164.all;

library work;

entity variable_source_async_hardened is
  generic (
    delay_gates   :     natural := 0
    );
  port (
    wrong_in      : in  std_logic;
    wrong_not_in  : in  std_logic;
    reset         : in  std_logic;
    zero_a        : in  std_logic;
    zero_b        : in  std_logic;
    zero_c        : in  std_logic;
    wrong_out     : out std_logic;
    wrong_not_out : out std_logic;
    var_out       : out std_logic;
    var_not_out   : out std_logic
    );
end variable_source_async_hardened;

architecture variable_source_async_hardened_architecture of variable_source_async_hardened is
  signal wrong_any_a           : std_logic;
  signal wrong_any_b           : std_logic;
  signal wrong_any             : std_logic;
  signal delay_values          : std_logic_vector (0 to delay_gates);
  signal new_value             : std_logic;
  signal output_value          : std_logic;
  signal var_out_effective     : std_logic;
  signal var_not_out_effective : std_logic;
begin
  wrong_any_a <= wrong_in xor zero_a;
  wrong_any_b <= wrong_not_in xor zero_a;
  wrong_any   <= wrong_any_a or wrong_any_b;

  process(reset, output_value, delay_values)
  begin
    delay_values(0) <= output_value and reset;

    for index in 1 to delay_gates loop
      delay_values(index) <= delay_values(index - 1) and reset;
    end loop;
  end process;

  new_value    <= delay_values(delay_gates) xor wrong_any;
  output_value <= new_value and reset;

  wrong_out     <= '0';
  wrong_not_out <= '0';

  var_out_effective     <= output_value;
  var_not_out_effective <= not(output_value);
  var_out               <= var_out_effective xor zero_b;
  var_not_out           <= var_not_out_effective xor zero_c;
end variable_source_async_hardened_architecture;
```

B.2.3 Basic synchronous variable source

```
-- Synchronous variable source component

library ieee;
use ieee.std_logic_1164.all;

library work;

entity variable_source_sync is
  port (
    wrong_in      : in  std_logic;
    wrong_not_in  : in  std_logic;
    reset         : in  std_logic;
    clock         : in  std_logic;
    wrong_out     : out std_logic;
    wrong_not_out : out std_logic;
    var_out       : out std_logic;
    var_not_out   : out std_logic
    );
end variable_source_sync;

architecture variable_source_sync_architecture of variable_source_sync is
  signal wrong_any       : std_logic;
  signal buffer_input    : std_logic;
  signal buffer_feedback : std_logic;
  signal new_value       : std_logic;
  signal output_value    : std_logic;
begin
  wrong_any <= wrong_in or wrong_not_in;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_input <= wrong_any;
    end if;
  end process;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_feedback <= output_value;
    end if;
  end process;

  new_value    <= buffer_feedback xor buffer_input;
  output_value <= new_value and reset;

  wrong_out     <= '0';
  wrong_not_out <= '0';

  var_out     <= output_value;
  var_not_out <= not(output_value);
end variable_source_sync_architecture;
```

B.2.4 Synchronous variable source hardened against compiler optimisations

```
-- Synchronous variable source component
-- Hardened against compiler optimisations

library ieee;
use ieee.std_logic_1164.all;

library work;

entity variable_source_sync_hardened is
  port (
    wrong_in      : in  std_logic;
    wrong_not_in  : in  std_logic;
    reset         : in  std_logic;
    clock         : in  std_logic;
    zero_a        : in  std_logic;
    zero_b        : in  std_logic;
    zero_c        : in  std_logic;
    wrong_out     : out std_logic;
    wrong_not_out : out std_logic;
    var_out       : out std_logic;
    var_not_out   : out std_logic
    );
end variable_source_sync_hardened;

architecture variable_source_sync_hardened_architecture of variable_source_sync_hardened is
  signal wrong_any_a           : std_logic;
  signal wrong_any_b           : std_logic;
  signal wrong_any             : std_logic;
  signal buffer_input          : std_logic;
  signal buffer_feedback       : std_logic;
  signal new_value             : std_logic;
  signal output_value          : std_logic;
  signal var_out_effective     : std_logic;
  signal var_not_out_effective : std_logic;
begin
  wrong_any_a <= wrong_in xor zero_a;
  wrong_any_b <= wrong_not_in xor zero_a;
  wrong_any   <= wrong_any_a or wrong_any_b;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_input <= wrong_any;
    end if;
  end process;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_feedback <= output_value;
    end if;
  end process;

  new_value    <= buffer_feedback xor buffer_input;
  output_value <= new_value and reset;

  wrong_out     <= '0';
  wrong_not_out <= '0';

  var_out_effective     <= output_value;
  var_not_out_effective <= not(output_value);
  var_out               <= var_out_effective xor zero_b;
  var_not_out           <= var_not_out_effective xor zero_c;
end variable_source_sync_hardened_architecture;
```

B.2.5 Synchronous variable source hardened against compiler optimisations (compact)

```
-- Synchronous variable source component
-- Hardened against compiler optimisations
--
-- Slightly compacted version for better space-efficiency

library ieee;
use ieee.std_logic_1164.all;

library work;

entity variable_source_sync_hardened_compact is
  port (
    wrong_in      : in  std_logic;
    wrong_not_in  : in  std_logic;
    reset         : in  std_logic;
    clock         : in  std_logic;
    zero_a        : in  std_logic;
    zero_b        : in  std_logic;
    zero_c        : in  std_logic;
    wrong_out     : out std_logic;
    wrong_not_out : out std_logic;
    var_out       : out std_logic;
    var_not_out   : out std_logic
    );
end variable_source_sync_hardened_compact;

architecture variable_source_sync_hardened_compact_architecture of
     variable_source_sync_hardened_compact is
  signal wrong_any_a           : std_logic;
  signal wrong_any_b           : std_logic;
  signal wrong_any             : std_logic;
  signal buffer_input          : std_logic;
  signal buffer_feedback       : std_logic;
  signal new_value             : std_logic;
  signal output_value          : std_logic;
  signal var_out_effective     : std_logic;
  signal var_not_out_effective : std_logic;
begin
```

```
  wrong_any_a <= wrong_in xor zero_a;
  wrong_any_b <= wrong_not_in xor zero_a;
  wrong_any   <= wrong_any_a or wrong_any_b;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_input <= wrong_any and reset;
    end if;
  end process;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_feedback <= output_value and reset;
    end if;
  end process;

  new_value    <= buffer_feedback xor buffer_input;
  output_value <= new_value;

  wrong_out     <= '0';
  wrong_not_out <= '0';

  var_out_effective     <= output_value;
  var_not_out_effective <= not(output_value);
  var_out               <= var_out_effective xor zero_b;
  var_not_out           <= var_not_out_effective xor zero_c;
end variable_source_sync_hardened_compact_architecture;
```

B.2.6 Locally probability driven variable source

```
-- Synchronous variable source component
-- Hardened against compiler optimisations
--
-- Experimental variable source employing locally probability driven search

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity variable_source_smart is
  generic (
    literal_count :     integer range 1 to 31 := 1;
    count_bits    :     integer range 1 to 5  := 1
    );
  port (
    clock         : in  std_logic;
    enabled       : in  std_logic;
    zero          : in  std_logic;
    clause_wrong  : in  std_logic_vector ((literal_count - 1) downto 0);
    rand_bits     : in  std_logic_vector (5 downto 0);
    variable_out  : out std_logic
    );
end variable_source_smart;

architecture variable_source_smart_architecture of variable_source_smart is
  component modulo_lookup_table
    generic (
      output_range :     integer range 1 to 32 := 1;
      output_bits  :     integer range 1 to 5  := 1
      );
    port (
      random_bits  : in  std_logic_vector (5 downto 0);
      value        : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string
      );
    port (
      dataa               : in  std_logic_vector ((lpm_width - 1) downto 0);
      datab               : in  std_logic_vector ((lpm_width - 1) downto 0);
      AgB                 : out std_logic
      );
  end component;

  signal     wrong_count          : std_logic_vector (count_bits downto 0);
  signal     random_value         : std_logic_vector ((count_bits - 1) downto 0);
  signal     random_value_compare : std_logic_vector (count_bits downto 0);
  signal     toggle_variable      : std_logic;
  signal     buffer_input         : std_logic;
  signal     buffer_feedback      : std_logic;
  signal     new_value            : std_logic;
  signal     output_value         : std_logic;
begin
  process(clause_wrong)
    variable input_sum            : integer range 0 to literal_count;
  begin
    input_sum := 0;

    for index in 0 to (literal_count - 1) loop
      if (clause_wrong(index) = '1') then
        input_sum := input_sum + 1;
      end if;
    end loop;

    wrong_count <= std_logic_vector(To_unsigned(input_sum, count_bits + 1));
  end process;

  modulo_lookup_table_component : modulo_lookup_table
    generic map (
      output_range => literal_count,
      output_bits  => count_bits
      )
    port map (
      random_bits  => rand_bits(5 downto 0),
      value        => random_value
      );

  random_value_compare <= '0' & random_value;

  lpm_compare_component : lpm_compare
    generic map (
      lpm_width          => (count_bits + 1),
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED"
      )
    port map (
      dataa              => wrong_count,
      datab              => random_value_compare,
      AgB                => toggle_variable
      );

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_input <= toggle_variable;
    end if;
  end process;

  process(clock)
  begin
    if (rising_edge(clock)) then
      buffer_feedback <= output_value;
    end if;
  end process;

  new_value <= buffer_feedback xor buffer_input;
  output_value <= new_value and enabled;
  variable_out <= output_value xor zero;
end variable_source_smart_architecture;
```

B.2.7 Fast modulo computation for smart variable source

```
-- Lookup table for fast modulo computations

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity modulo_lookup_table is
  generic (
    output_range :     integer range 1 to 32 := 1;
    output_bits  :     integer range 1 to 5  := 1
    );
  port (
    random_bits  : in  std_logic_vector (5 downto 0);
    value        : out std_logic_vector ((output_bits - 1) downto 0)
    );
end modulo_lookup_table;

architecture modulo_lookup_table_architecture of modulo_lookup_table is
  signal result : std_logic_vector ((output_bits - 1) downto 0);
begin
  process(random_bits)
  begin
    case output_range is
      when 1            =>
        result(0)                <= '0';
      when 2            =>
        result(0)                <= random_bits(5);
      when 4            =>
        result(1 downto 0)       <= random_bits(5 downto 4);
      when 8            =>
        result(2 downto 0)       <= random_bits(5 downto 3);
      when 16           =>
        result(3 downto 0)       <= random_bits(5 downto 2);
      when 32           =>
        result(4 downto 0)       <= random_bits(5 downto 1);
      when 3            =>
        case random_bits is
          when "000000" => result <= "00";
          when "000001" => result <= "01";
          when "000010" => result <= "10";
          when "000011" => result <= "00";
          when "000100" => result <= "01";

          ...

          when "111110" => result <= "10";
          when "111111" => result <= "00";
        end case;

      ...

      when 31 =>
        case random_bits is
          when "000000" => result <= "00000";

          ...

          when "111111" => result <= "00001";
        end case;
    end case;
  end process;

  value <= result;
end modulo_lookup_table_architecture;
```

B.3 Fixed distribution bit sources

B.3.1 Bit source using single bit LFSR

```
-- Fixed distribution bit source
--
```

```
-- Probability of an output bit being 0 is probability_factor / 1024
--
-- Basic LFSR generating one bit each clock cycle

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity fixed_distribution_bit_source_basic_lfsr is
  generic (
    output_bits         :      integer range 1 to 65535 := 1;
    probability_factor :      bit_vector (9 downto 0)  := "1010101010"  -- 682
    );
  port (
    reset               : in  std_logic;
    clock               : in  std_logic;
    bits                : out std_logic_vector ((output_bits - 1) downto 0)
    );
end fixed_distribution_bit_source_basic_lfsr;

architecture fixed_distribution_bit_source_basic_lfsr_architecture of
     fixed_distribution_bit_source_basic_lfsr is
  component lpm_compare
    generic (
      lpm_width           :      natural;
      lpm_type            :      string;
      lpm_representation :      string;
      lpm_hint            :      string
      );
    port (
      dataa               : in  std_logic_vector (9 downto 0);
      datab               : in  std_logic_vector (9 downto 0);
      AgB                 : out std_logic
      );
  end component;

  component lpm_shiftreg
    generic (
      lpm_type      :      string;
      lpm_width     :      natural;
      lpm_direction :      string
      );
    port (
      clock         : in  std_logic;
      q             : out std_logic_vector ((output_bits - 1) downto 0);
      sset          : in  std_logic;
      shiftin       : in  std_logic
      );
  end component;

  component lfsr40_serial
    generic (
      output_bits :      integer range 1 to 40
      );
    port (
      reset       : in  std_logic;
      clock       : in  std_logic;
      value       : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  signal random            : std_logic_vector (9 downto 0);
  signal factor            : std_logic_vector (9 downto 0);
  signal next_bit          : std_logic;
  signal output            : std_logic_vector ((output_bits - 1) downto 0);
begin
  lfsr40_serial_component : lfsr40_serial
    generic map (
      output_bits => 10
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random
      );

  factor <= To_stdlogicvector(probability_factor);

  lpm_compare_component : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random,
      datab              => factor,
      AgB                => next_bit
      );

  lpm_shiftreg_component : lpm_shiftreg
    generic map (
      lpm_type => "LPM_SHIFTREG",
      lpm_width => output_bits,
      lpm_direction => "LEFT"
      )
    port map (
      clock => clock,
      sset => reset,
      shiftin => next_bit,
      q => output
      );

  bits <= output;
end fixed_distribution_bit_source_basic_lfsr_architecture;
```

B.3.2 Bit source using parallelised LFSR

```
-- Fixed distribution bit source
--
-- Probability of an output bit being 0 is probability_factor / 1024
--
-- Parallel LFSR generating 10 bits each clock cycle

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity fixed_distribution_bit_source_parallel_lfsr is
  generic (
    output_bits        :     integer range 1 to 65535 := 1;
    probability_factor :     bit_vector (9 downto 0)  := "1010101010"  -- 682
    );
  port (
    reset              : in  std_logic;
    clock              : in  std_logic;
    bits               : out std_logic_vector ((output_bits - 1) downto 0)
    );
end fixed_distribution_bit_source_parallel_lfsr;

architecture fixed_distribution_bit_source_parallel_lfsr_architecture of
     fixed_distribution_bit_source_parallel_lfsr is
  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (9 downto 0);
      datab              : in  std_logic_vector (9 downto 0);
      AgB                : out std_logic
      );
  end component;
  component lpm_shiftreg
    generic (
      lpm_type      :     string;
      lpm_width     :     natural;
      lpm_direction :     string
      );
    port (
      clock         : in  std_logic;
      q             : out std_logic_vector ((output_bits - 1) downto 0);
      sset          : in  std_logic;
      shiftin       : in  std_logic
      );
  end component;

  component lfsr40_parallel
    generic (
      output_bits :     integer range 1 to 19
      );
    port (
      reset       : in  std_logic;
      clock       : in  std_logic;
      value       : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  signal random             : std_logic_vector (9 downto 0);
  signal factor             : std_logic_vector (9 downto 0);
  signal next_bit           : std_logic;
  signal output             : std_logic_vector ((output_bits - 1) downto 0);
begin
  lfsr40_parallel_component : lfsr40_parallel
    generic map (
      output_bits => 10
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random
      );

  factor <= To_stdlogicvector(probability_factor);

  lpm_compare_component : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random,
      datab              => factor,
      AgB                => next_bit
      );

  lpm_shiftreg_component : lpm_shiftreg
    generic map (
      lpm_type => "LPM_SHIFTREG",
      lpm_width => output_bits,
      lpm_direction => "LEFT"
      )
    port map (
      clock => clock,
      sset => reset,
      shiftin => next_bit,
      q => output
      );

  bits <= output;
end fixed_distribution_bit_source_parallel_lfsr_architecture;
```

B.3.3 Bit source using parallelised LFSR array

```
-- Fixed distribution bit source
--
-- Probability of an output bit being 0 is probability_factor / 1024
```

```
--
-- Parallel LFSR array generating 100 bits each clock cycle

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity fixed_distribution_bit_source_multi_lfsr is
  generic (
    output_bits        :     integer range 1 to 65535 := 1;
    probability_factor :     bit_vector (9 downto 0)  := "1010101010"  -- 682
    );
  port (
    reset              : in  std_logic;
    clock              : in  std_logic;
    bits               : out std_logic_vector ((output_bits - 1) downto 0)
    );
end fixed_distribution_bit_source_multi_lfsr;

architecture fixed_distribution_bit_source_multi_lfsr_architecture of
     fixed_distribution_bit_source_multi_lfsr is
  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (9 downto 0);
      datab              : in  std_logic_vector (9 downto 0);
      AgB                : out std_logic
      );
  end component;

  component lpm_shiftreg
    generic (
      lpm_type      :     string;
      lpm_width     :     natural;
      lpm_direction :     string
      );
    port (
      data          : in  std_logic_vector ((output_bits - 1) downto 0);
      clock         : in  std_logic;
      load          : in  std_logic;
      sclr          : in  std_logic;
      q             : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  component lfsr40_parallel_preseeded
    generic (
      output_bits :     integer range 1 to 19;
      seed        :     bit_vector (39 downto 0)
      );
    port (
      reset       : in  std_logic;
      clock       : in  std_logic;
      value       : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  signal invreset   : std_logic;
  signal random_a   : std_logic_vector (9 downto 0);
  signal random_b   : std_logic_vector (9 downto 0);
  signal random_c   : std_logic_vector (9 downto 0);
  signal random_d   : std_logic_vector (9 downto 0);
  signal random_e   : std_logic_vector (9 downto 0);
  signal random_f   : std_logic_vector (9 downto 0);
  signal random_g   : std_logic_vector (9 downto 0);
  signal random_h   : std_logic_vector (9 downto 0);
  signal random_i   : std_logic_vector (9 downto 0);
  signal random_j   : std_logic_vector (9 downto 0);
  signal factor     : std_logic_vector (9 downto 0);
  signal next_bit_a : std_logic;
  signal next_bit_b : std_logic;
  signal next_bit_c : std_logic;
  signal next_bit_d : std_logic;
  signal next_bit_e : std_logic;
  signal next_bit_f : std_logic;
  signal next_bit_g : std_logic;
  signal next_bit_h : std_logic;
  signal next_bit_i : std_logic;
  signal next_bit_j : std_logic;
  signal next_bits  : std_logic_vector (9 downto 0);
  signal reginput   : std_logic_vector ((output_bits - 1) downto 0);
  signal regoutput  : std_logic_vector ((output_bits - 1) downto 0);
begin
  invreset <= not(reset);

  lfsr40_parallel_preseeded_component_a : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001101001100010110111010010001010000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_a
      );

  lfsr40_parallel_preseeded_component_b : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1010110001110101000001000011101010111101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_b
      );

  lfsr40_parallel_preseeded_component_c : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111011000101001111000100000101110110011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_c
      );

  lfsr40_parallel_preseeded_component_d : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1111110100101100100000100111010011111100"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_d
      );

  lfsr40_parallel_preseeded_component_e : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0000101001101100001011100100110011100101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_e
      );

  lfsr40_parallel_preseeded_component_f : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001010111011100001000000011000011000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_f
      );

  lfsr40_parallel_preseeded_component_g : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100010100111110010111111000100011100111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_g
      );

  lfsr40_parallel_preseeded_component_h : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100000001001000000111100001110100010001"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_h
      );

  lfsr40_parallel_preseeded_component_i : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1101111111111001100101000100110010001101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_i
      );

  lfsr40_parallel_preseeded_component_j : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111010101101100111100000001000111101011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_j
      );

  factor <= To_stdlogicvector(probability_factor);

  lpm_compare_component_a : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_a,
      datab              => factor,
      AgB                => next_bit_a
      );

  lpm_compare_component_b : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_b,
      datab              => factor,
      AgB                => next_bit_b
      );
  lpm_compare_component_c : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_c,
      datab              => factor,
      AgB                => next_bit_c
      );

  lpm_compare_component_d : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_d,
      datab              => factor,
      AgB                => next_bit_d
      );

  lpm_compare_component_e : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_e,
      datab              => factor,
      AgB                => next_bit_e
      );

  lpm_compare_component_f : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_f,
      datab              => factor,
      AgB                => next_bit_f
      );

  lpm_compare_component_g : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_g,
      datab              => factor,
      AgB                => next_bit_g
      );

  lpm_compare_component_h : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_h,
      datab              => factor,
      AgB                => next_bit_h
      );

  lpm_compare_component_i : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => random_i,
      datab => factor,
      AgB => next_bit_i
      );

  lpm_compare_component_j : lpm_compare
    generic map (
      lpm_width => 10,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => random_j,
      datab => factor,
      AgB => next_bit_j
      );

  next_bits <= next_bit_a & next_bit_b & next_bit_c & next_bit_d & next_bit_e & next_bit_f &
      next_bit_g & next_bit_h & next_bit_i & next_bit_j;
  reginput <= regoutput((output_bits - 1 - 10) downto 0) & next_bits;
  bits <= regoutput;

  lpm_shiftreg_component : lpm_shiftreg
    generic map (
      lpm_type => "LPM_SHIFTREG",
      lpm_width => output_bits,
      lpm_direction => "LEFT"
      )
    port map (
      data => reginput,
      clock => clock,
      load => invreset,
      sclr => reset,
      q => regoutput
      );
end fixed_distribution_bit_source_multi_lfsr_architecture;
```

B.3.4 Bit source using parallelised LFSR array with shift register preseeding

```
-- Fixed distribution bit source
--
-- Probability of an output bit being 0 is probability_factor / 1024
--
-- Parallel LFSR array generating 100 bits each clock cycle
-- Selection register is preseeded at startup to stabilise probabilites

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity fixed_distribution_bit_source_multi_lfsr_preseeded is
  generic (
    output_bits        :     integer range 1 to 65535   := 1;
    probability_factor :     bit_vector (9 downto 0)     := "1010101010";  -- 682
    seed               :     bit_vector (1109 downto 0) := "0000␣..."
    );
  port (
    reset              : in  std_logic;
    clock              : in  std_logic;
    bits               : out std_logic_vector ((output_bits - 1) downto 0)
    );
end fixed_distribution_bit_source_multi_lfsr_preseeded;

architecture fixed_distribution_bit_source_multi_lfsr_preseeded_architecture of
     fixed_distribution_bit_source_multi_lfsr_preseeded is
  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (9 downto 0);
      datab              : in  std_logic_vector (9 downto 0);
      AgB                : out std_logic
      );
  end component;

  component lpm_shiftreg
    generic (
      lpm_type      :     string;
      lpm_width     :     natural;
      lpm_direction :     string
      );
    port (
      data          : in  std_logic_vector ((output_bits - 1) downto 0);
      clock         : in  std_logic;
      load          : in  std_logic;
      q             : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  component lfsr40_parallel_preseeded
    generic (
      output_bits :     integer range 1 to 19;
      seed        :     bit_vector (39 downto 0)
      );
    port (
      reset       : in  std_logic;
      clock       : in  std_logic;
      value       : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  signal random_a                       : std_logic_vector (9 downto 0);
  signal random_b                       : std_logic_vector (9 downto 0);
  signal random_c                       : std_logic_vector (9 downto 0);
  signal random_d                       : std_logic_vector (9 downto 0);
  signal random_e                       : std_logic_vector (9 downto 0);
  signal random_f                       : std_logic_vector (9 downto 0);
  signal random_g                       : std_logic_vector (9 downto 0);
  signal random_h                       : std_logic_vector (9 downto 0);
  signal random_i                       : std_logic_vector (9 downto 0);
  signal random_j                       : std_logic_vector (9 downto 0);
  signal factor                         : std_logic_vector (9 downto 0);
  signal next_bit_a                     : std_logic;
  signal next_bit_b                     : std_logic;
  signal next_bit_c                     : std_logic;
  signal next_bit_d                     : std_logic;
  signal next_bit_e                     : std_logic;
  signal next_bit_f                     : std_logic;
  signal next_bit_g                     : std_logic;
  signal next_bit_h                     : std_logic;
  signal next_bit_i                     : std_logic;
  signal next_bit_j                     : std_logic;
  signal next_bits                      : std_logic_vector (9 downto 0);
  signal reginput                       : std_logic_vector ((output_bits - 1) downto 0);
  signal regoutput                      : std_logic_vector ((output_bits - 1) downto 0);
begin
  lfsr40_parallel_preseeded_component_a : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001101001100010110111010010001010000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_a
      );

  lfsr40_parallel_preseeded_component_b : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1010110001110101000001000011101010111101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_b
      );

  lfsr40_parallel_preseeded_component_c : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111011000101001111000100000101110110011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_c
      );

  lfsr40_parallel_preseeded_component_d : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1111110100101100100000100111010011111100"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_d
      );

  lfsr40_parallel_preseeded_component_e : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0000101001101100001011100100110011100101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_e
      );

  lfsr40_parallel_preseeded_component_f : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001010111011100001000000011000011000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_f
      );

  lfsr40_parallel_preseeded_component_g : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100010100111110010111111000100011100111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_g
      );

  lfsr40_parallel_preseeded_component_h : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100000010010000001111000011101000100001"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_h
      );

  lfsr40_parallel_preseeded_component_i : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1101111111111001100101000100110010001101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_i
      );

  lfsr40_parallel_preseeded_component_j : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111010101101100111100000000100011101011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_j
      );

  factor <= To_stdlogicvector(probability_factor);

  lpm_compare_component_a : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_a,
      datab              => factor,
      AgB                => next_bit_a
      );

  lpm_compare_component_b : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_b,
      datab              => factor,
      AgB                => next_bit_b
      );

  lpm_compare_component_c : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_c,
      datab              => factor,
      AgB                => next_bit_c
      );

  lpm_compare_component_d : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_d,
      datab              => factor,
      AgB                 => next_bit_d
      );

  lpm_compare_component_e : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation  => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_e,
      datab               => factor,
      AgB                 => next_bit_e
      );

  lpm_compare_component_f : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation  => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_f,
      datab               => factor,
      AgB                 => next_bit_f
      );

  lpm_compare_component_g : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation  => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_g,
      datab               => factor,
      AgB                 => next_bit_g
      );

  lpm_compare_component_h : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation  => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_h,
      datab               => factor,
      AgB                 => next_bit_h
      );

  lpm_compare_component_i : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation  => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa => random_i,
      datab => factor,
      AgB => next_bit_i
      );

  lpm_compare_component_j : lpm_compare
    generic map (
      lpm_width => 10,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => random_j,
      datab => factor,
      AgB => next_bit_j
      );

  next_bits <= next_bit_a & next_bit_b & next_bit_c & next_bit_d & next_bit_e & next_bit_f &
      next_bit_g & next_bit_h & next_bit_i & next_bit_j;
  reginput <= (regoutput((output_bits - 1 - 10) downto 0) & next_bits) when (reset = '0') else
      To_stdlogicvector(seed);
  bits <= regoutput;

  lpm_shiftreg_component : lpm_shiftreg
    generic map (
      lpm_type => "LPM_SHIFTREG",
      lpm_width => output_bits,
      lpm_direction => "LEFT"
      )
    port map (
      data => reginput,
      clock => clock,
      load => '1',
      q => regoutput
      );
end fixed_distribution_bit_source_multi_lfsr_preseeded_architecture;
```

B.3.5 Bit source supporting dynamic probabilities using simulated annealing

```
-- Fixed distribution bit source
--
-- Probability of an output bit being 0 is probability_factor / 1024
--
-- Modified version for experiments with simulated annealing

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity fixed_distribution_bit_source_simulated_annealing is
  generic (
    output_bits        :     integer range 1 to 65535 := 1;
    probability_factor :     bit_vector (9 downto 0)  := "1010101010"  -- 682
    );
  port (
    reset              : in  std_logic;
    clock              : in  std_logic;
    bits               : out std_logic_vector ((output_bits - 1) downto 0)
    );
end fixed_distribution_bit_source_simulated_annealing;

architecture fixed_distribution_bit_source_simulated_annealing_architecture of
     fixed_distribution_bit_source_simulated_annealing is
  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (9 downto 0);
      datab              : in  std_logic_vector (9 downto 0);
      AgB                : out std_logic
      );
  end component;

  component lpm_shiftreg
    generic (
      lpm_type      :     string;
      lpm_width     :     natural;
      lpm_direction :     string
      );
    port (
      data          : in  std_logic_vector ((output_bits - 1) downto 0);
      clock         : in  std_logic;
      load          : in  std_logic;
      sclr          : in  std_logic;
      q             : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  component lfsr40_parallel_preseeded
    generic (
      output_bits :     integer range 1 to 19;
      seed        :     bit_vector (39 downto 0)
      );
    port (
      reset       : in  std_logic;
      clock       : in  std_logic;
      value       : out std_logic_vector ((output_bits - 1) downto 0)
      );
  end component;

  component rom_simulated_annealing_table
    port (
      clock   : in  std_logic;
      address : in  std_logic_vector(11 downto 0);
      q       : out std_logic_vector(15 downto 0)
      );
  end component;

  signal invreset     : std_logic;
  signal random_a     : std_logic_vector (9 downto 0);
  signal random_b     : std_logic_vector (9 downto 0);
  signal random_c     : std_logic_vector (9 downto 0);
  signal random_d     : std_logic_vector (9 downto 0);
  signal random_e     : std_logic_vector (9 downto 0);
  signal random_f     : std_logic_vector (9 downto 0);
  signal random_g     : std_logic_vector (9 downto 0);
  signal random_h     : std_logic_vector (9 downto 0);
  signal random_i     : std_logic_vector (9 downto 0);
  signal random_j     : std_logic_vector (9 downto 0);
  signal factor       : std_logic_vector (9 downto 0);
  signal next_bit_a   : std_logic;
  signal next_bit_b   : std_logic;
  signal next_bit_c   : std_logic;
  signal next_bit_d   : std_logic;
  signal next_bit_e   : std_logic;
  signal next_bit_f   : std_logic;
  signal next_bit_g   : std_logic;
  signal next_bit_h   : std_logic;
  signal next_bit_i   : std_logic;
  signal next_bit_j   : std_logic;
  signal next_bits    : std_logic_vector (9 downto 0);
  signal reginput     : std_logic_vector ((output_bits - 1) downto 0);
  signal regoutput    : std_logic_vector ((output_bits - 1) downto 0);
  signal next_address : std_logic_vector (11 downto 0);
  signal rom_data     : std_logic_vector (15 downto 0);
begin
  invreset <= not(reset);

  lfsr40_parallel_preseeded_component_a : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001101001100010110111010010001010000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_a
      );

  lfsr40_parallel_preseeded_component_b : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1010110001110101000001000011101010111101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_b
      );

  lfsr40_parallel_preseeded_component_c : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111011000101001111000100000101110110011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_c
      );

  lfsr40_parallel_preseeded_component_d : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1111110100101100100000100111010011111100"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_d
      );

  lfsr40_parallel_preseeded_component_e : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0000101001101100001011100100110011100101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_e
      );

  lfsr40_parallel_preseeded_component_f : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1001010111011100001000000011000011000111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_f
      );

  lfsr40_parallel_preseeded_component_g : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100010100111110010111111000100011100111"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_g
      );

  lfsr40_parallel_preseeded_component_h : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0100000010010000001111000011101000100001"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_h
      );

  lfsr40_parallel_preseeded_component_i : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "1101111111111001100101000100110010001101"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_i
      );

  lfsr40_parallel_preseeded_component_j : lfsr40_parallel_preseeded
    generic map (
      output_bits => 10,
      seed        => "0111010101101100111100000001000111101011"
      )
    port map (
      reset       => reset,
      clock       => clock,
      value       => random_j
      );

  rom_component : rom_simulated_annealing_table
    port map (
      clock   => clock,
      address => next_address,
      q       => rom_data
      );

  process(clock)
    variable current_address  : integer range 0 to 4096;
    variable effective_factor : integer range 0 to 1023;
    variable base_factor      : integer range 0 to 1023;
    variable boost_factor     : integer range 0 to 1023;
    variable rle_counter      : integer range 0 to 63;
  begin
    base_factor := To_integer(unsigned(To_stdlogicvector(probability_factor)));

    if (rising_edge(clock)) then
      if (reset = '1') then
        current_address := 0;
        next_address <= To_stdlogicvector("000000000000");

        boost_factor      := To_integer(unsigned(rom_data(9 downto 0)));
        rle_counter       := To_integer(unsigned(rom_data(15 downto 10)));
      else
        if (rle_counter = 1) then
          current_address := current_address + 1;
          rle_counter     := 0;
        elsif (rle_counter = 0) then
          boost_factor    := To_integer(unsigned(rom_data(9 downto 0)));
          rle_counter     := To_integer(unsigned(rom_data(15 downto 10)));
        else
          rle_counter     := rle_counter - 1;
        end if;

        next_address <= std_logic_vector(To_unsigned(current_address, 12));
      end if;

      effective_factor := base_factor - boost_factor;
      factor <= std_logic_vector(To_unsigned(effective_factor, 10));
    end if;
  end process;

  lpm_compare_component_a : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_a,
      datab              => factor,
      AgB                => next_bit_a
      );

  lpm_compare_component_b : lpm_compare
    generic map (
      lpm_width          => 10,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_b,
      datab               => factor,
      AgB                 => next_bit_b
      );

  lpm_compare_component_c : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_c,
      datab               => factor,
      AgB                 => next_bit_c
      );

  lpm_compare_component_d : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_d,
      datab               => factor,
      AgB                 => next_bit_d
      );

  lpm_compare_component_e : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_e,
      datab               => factor,
      AgB                 => next_bit_e
      );

  lpm_compare_component_f : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_f,
      datab               => factor,
      AgB                 => next_bit_f
      );

  lpm_compare_component_g : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa               => random_g,
      datab               => factor,
      AgB                 => next_bit_g
      );

  lpm_compare_component_h : lpm_compare
    generic map (
      lpm_width           => 10,
      lpm_type            => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint            => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => random_h,
      datab              => factor,
      AgB                => next_bit_h
      );

  lpm_compare_component_i : lpm_compare
    generic map (
      lpm_width => 10,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => random_i,
      datab => factor,
      AgB => next_bit_i
      );

  lpm_compare_component_j : lpm_compare
    generic map (
      lpm_width => 10,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => random_j,
      datab => factor,
      AgB => next_bit_j
      );

  next_bits <= next_bit_a & next_bit_b & next_bit_c & next_bit_d & next_bit_e & next_bit_f &
      next_bit_g & next_bit_h & next_bit_i & next_bit_j;
  reginput <= regoutput((output_bits - 1 - 10) downto 0) & next_bits;
  bits <= regoutput;

  lpm_shiftreg_component : lpm_shiftreg
    generic map (
      lpm_type => "LPM_SHIFTREG",
      lpm_width => output_bits,
      lpm_direction => "LEFT"
      )
    port map (
      data => reginput,
      clock => clock,
      load => invreset,
      sclr => reset,
      q => regoutput
      );
end fixed_distribution_bit_source_simulated_annealing_architecture;
```

B.3.6 ROM interface for simulated annealing stepping tables

```
-- ROM interface providing stepping table for simulated annealing

library ieee;
use ieee.std_logic_1164.all;

library altera_mf;
use altera_mf.all;

entity rom_simulated_annealing_table is
  port (
    address : in  std_logic_vector (11 downto 0);
    clock   : in  std_logic;
    q       : out std_logic_vector (15 downto 0)
    );
end rom_simulated_annealing_table;

architecture SYN of rom_simulated_annealing_table is
  component altsyncram
    generic (
      address_aclr_a         :     string;
      init_file              :     string;
      intended_device_family :     string;
      lpm_type                :     string;
      numwords_a              :     natural;
      operation_mode          :     string;
      outdata_aclr_a          :     string;
      outdata_reg_a           :     string;
      power_up_uninitialized :     string;
      widthad_a               :     natural;
      width_a                 :     natural;
      width_byteena_a         :     natural
      );
    port (
      clock0                  : in  std_logic;
      address_a               : in  std_logic_vector (11 downto 0);
      q_a                     : out std_logic_vector (15 downto 0)
      );
  end component;

  signal output_word : std_logic_vector (15 downto 0);
begin
  altsyncram_component : altsyncram
    generic map (
      address_aclr_a          => "NONE",
      init_file               => "sa_table.mif",
      intended_device_family => "Cyclone",
      lpm_type                => "altsyncram",
      numwords_a              => 4096,
      operation_mode          => "ROM",
      outdata_aclr_a          => "NONE",
      outdata_reg_a           => "UNREGISTERED",
      power_up_uninitialized => "FALSE",
      widthad_a               => 12,
      width_a                 => 16,
      width_byteena_a         => 1
      )
    port map (
      clock0 => clock,
      address_a => address,
      q_a => output_word
      );

  q <= output_word(15 downto 0);
end SYN;
```

B.4 Pseudo-random random number generators

B.4.1 Single bit LFSR (40-bit)

```
-- 40-bit linear feedback shift register
--
-- Taps:   19, 21
-- Period: 1 090 921 693 057
--
-- BEWARE: This implementation is buggy!
--
-- The characteristic polynomial of this LFSR is f(x) = x^40 + x^21 + x^19
-- This is obviously not irreducible leading to a period dependant
-- on the seed used to initialise the LFSR
--
-- Parameters from http://sciencezero.4hv.org/science/lfsr.htm

library ieee;
use ieee.std_logic_1164.all;

library work;

entity lfsr40_serial is
  generic (
    output_bits :     integer range 1 to 40 := 10
    );
  port (
    reset       : in  std_logic;
    clock       : in  std_logic;
    value       : out std_logic_vector ((output_bits - 1) downto 0)
    );
end lfsr40_serial;
```

```
architecture lfsr40_serial_architecture of lfsr40_serial is
  signal tap1    : std_logic;
  signal tap2    : std_logic;
  signal nextbit : std_logic;
  signal shiftin : std_logic;
  signal vector  : std_logic_vector (39 downto 0);
begin
  tap1    <= vector(18);
  tap2    <= vector(20);
  nextbit <= tap1 xnor tap2;
  shiftin <= nextbit and not(reset);

  process(clock)
  begin
    if (rising_edge(clock)) then
      vector <= vector(38 downto 0) & shiftin;
    end if;
  end process;

  value <= vector(39 downto (39 - output_bits + 1));
end lfsr40_serial_architecture;
```

B.4.2 Parallelised LFSR (40-bit)

```
-- 40-bit linear feedback shift register
--
-- Taps:   19, 21
-- Period: 1 090 921 693 057
--
-- BEWARE: This implementation is buggy!
--
-- The characteristic polynomial of this LFSR is f(x) = x^40 + x^21 + x^19
-- This is obviously not irreducible leading to a period dependant
-- on the seed used to initialise the LFSR
--
-- Parameters from http://sciencezero.4hv.org/science/lfsr.htm

library ieee;
use ieee.std_logic_1164.all;

library work;

entity lfsr40_parallel is
  generic (
    output_bits :     integer range 1 to 19 := 10
    );
  port (
    reset       : in  std_logic;
    clock       : in  std_logic;
    value       : out std_logic_vector ((output_bits - 1) downto 0)
    );
end lfsr40_parallel;

architecture lfsr40_parallel_architecture of lfsr40_parallel is
  signal zerobits    : std_logic_vector (39 downto 0);
  signal tap1bits    : std_logic_vector ((output_bits - 1) downto 0);
  signal tap2bits    : std_logic_vector ((output_bits - 1) downto 0);
  signal nextbits    : std_logic_vector ((output_bits - 1) downto 0);
  signal shiftinbits : std_logic_vector ((output_bits - 1) downto 0);
  signal vector      : std_logic_vector (39 downto 0);
begin
  zerobits    <= "0000000000000000000000000000000000000000";
  tap1bits    <= vector(18 downto (18 - (output_bits - 1)));
  tap2bits    <= vector(20 downto (20 - (output_bits - 1)));
  nextbits    <= tap1bits xnor tap2bits;
  shiftinbits <= nextbits when reset = '0' else zerobits((output_bits - 1) downto 0);

  process(clock)
  begin
    if (rising_edge(clock)) then
      vector <= vector((39 - output_bits) downto 0) & shiftinbits;
    end if;
  end process;

  value <= vector(39 downto (39 - output_bits + 1));
end lfsr40_parallel_architecture;
```

B.4.3 Parallelised LFSR supporting variable seed (40-bit)

```
-- 40-bit linear feedback shift register
--
-- Taps:   19, 21
-- Period: 1 090 921 693 057
--
-- BEWARE: This implementation is buggy!
--
-- The characteristic polynomial of this LFSR is f(x) = x^40 + x^21 + x^19
-- This is obviously not irreducible leading to a period dependant
-- on the seed used to initialise the LFSR
--
-- Parameters from http://sciencezero.4hv.org/science/lfsr.htm

library ieee;
use ieee.std_logic_1164.all;

library work;

entity lfsr40_parallel_preseeded is
  generic (
    output_bits :     integer range 1 to 19    := 10;
    seed        :     bit_vector (39 downto 0) := "0000000000000000000000000000000000000000"
    );
  port (
    reset       : in  std_logic;
    clock       : in  std_logic;
    value       : out std_logic_vector ((output_bits - 1) downto 0)
    );
end lfsr40_parallel_preseeded;

architecture lfsr40_parallel_preseeded_architecture of lfsr40_parallel_preseeded is
  signal tap1bits : std_logic_vector ((output_bits - 1) downto 0);
  signal tap2bits : std_logic_vector ((output_bits - 1) downto 0);
  signal nextbits : std_logic_vector ((output_bits - 1) downto 0);
  signal vector   : std_logic_vector (39 downto 0);
begin
  tap1bits <= vector(18 downto (18 - (output_bits - 1)));
  tap2bits <= vector(20 downto (20 - (output_bits - 1)));
  nextbits <= tap1bits xnor tap2bits;

  process(clock)
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        vector <= To_stdlogicvector(seed);
      else
        vector <= vector((39 - output_bits) downto 0) & nextbits;
      end if;
    end if;
  end process;

  value <= vector(39 downto (39 - output_bits + 1));
end lfsr40_parallel_preseeded_architecture;
```

B.4.4 Parallelised LFSR supporting variable seed (41-bit)

```
-- 41-bit linear feedback shift register
--
-- Taps: 41, 38
--
-- Parameters from Xilinx application note about LFSR techniques

library ieee;
use ieee.std_logic_1164.all;

library work;

entity lfsr41_parallel_preseeded is
  generic (
    output_bits :     integer range 1 to 37    := 10;
    seed        :     bit_vector (40 downto 0) := "00000000000000000000000000000000000000000"
    );
  port (
    clock       : in  std_logic;
    enabled      : in   std_logic;
    output       : out  std_logic_vector ((output_bits - 1) downto 0)
    );
end lfsr41_parallel_preseeded;

architecture lfsr41_parallel_preseeded_architecture of lfsr41_parallel_preseeded is
  signal tap1bits : std_logic_vector ((output_bits - 1) downto 0);
  signal tap2bits : std_logic_vector ((output_bits - 1) downto 0);
  signal nextbits : std_logic_vector ((output_bits - 1) downto 0);
  signal vector   : std_logic_vector (40 downto 0);
begin
  tap1bits <= vector(40 downto (40 - (output_bits - 1)));
  tap2bits <= vector(37 downto (37 - (output_bits - 1)));
  nextbits <= tap1bits xnor tap2bits;

  process(clock)
  begin
    if (rising_edge(clock)) then
      if (enabled = '0') then
        vector <= To_stdlogicvector(seed);
      else
        vector <= vector((40 - output_bits) downto 0) & nextbits;
      end if;
    end if;
  end process;

  output <= vector(40 downto (40 - (output_bits - 1)));
end lfsr41_parallel_preseeded_architecture;
```

B.5 Support circuitry

B.5.1 Delayed startup controller for single testruns

```
-- Delayed startup controller
--
-- Automatically initiates circuit startup and shutdown
-- during unattended test runs

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity delayed_startup_controller_single is
  port (
    reset : out std_logic;
    clock : in  std_logic
    );
end delayed_startup_controller_single;

architecture delayed_startup_controller_single_architecture of
     delayed_startup_controller_single is
  component lpm_ff
    generic (
      lpm_width  :     natural;
      lpm_type   :     string;
      lpm_fftype :     string
      );
    port (
      sclr       : in  std_logic;
      clock      : in  std_logic;
      q          : out std_logic_vector (0 downto 0);
      data       : in  std_logic_vector (0 downto 0);
      sset       : in  std_logic
      );
  end component;

  component lpm_counter
    generic (
      lpm_width     :     natural;
      lpm_type      :     string;
      lpm_direction :     string
      );
    port (
      sclr          : in  std_logic;
      clock         : in  std_logic;
      q             : out std_logic_vector (31 downto 0);
      cnt_en        : in  std_logic
      );
  end component;

  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (31 downto 0);
      datab              : in  std_logic_vector (31 downto 0);
      AgeB               : out std_logic
      );
  end component;

  signal activation_timeout_reached   : std_logic;
  signal deactivation_timeout_reached : std_logic;
  signal activated                    : std_logic_vector (0 to 0);
  signal deactivated                  : std_logic_vector (0 to 0);
  signal counter_delay_value          : std_logic_vector (31 downto 0);
  signal counter_hold_value           : std_logic_vector (31 downto 0);
  signal comparator_delay_bits        : bit_vector (31 downto 0);
  signal comparator_delay_value       : std_logic_vector (31 downto 0);
  signal comparator_hold_bits         : bit_vector (31 downto 0);
  signal comparator_hold_value        : std_logic_vector (31 downto 0);
begin
  lpm_ff_activation                   : lpm_ff
    generic map (
      lpm_width  => 1,
      lpm_type   => "LPM_FF",
      lpm_fftype => "DFF"
      )
    port map (
      clock      => clock,
      data       => activated,
      sset       => activation_timeout_reached,
      q          => activated
      );

  lpm_ff_deactivation : lpm_ff
    generic map (
      lpm_width  => 1,
      lpm_type   => "LPM_FF",
      lpm_fftype => "DFF"
      )
    port map (
      clock      => clock,
      data       => deactivated,
      sset       => deactivation_timeout_reached,
      q          => deactivated
      );

  lpm_counter_delay : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      clock         => clock,
      q             => counter_delay_value
      );

  lpm_counter_hold : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      clock         => clock,
      cnt_en        => activated(0),
      q             => counter_hold_value
      );

  comparator_delay_bits(31 downto 0) <= "00000100010001000110000001110000";  -- 5 seconds at
      14.318 MHz
  comparator_delay_value             <= To_stdlogicvector(comparator_delay_bits);

  lpm_compare_delay : lpm_compare
    generic map (
      lpm_width          => 32,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => counter_delay_value,
      datab              => comparator_delay_value,
      AgeB               => activation_timeout_reached
      );

  comparator_hold_bits(31 downto 0) <= "00000000000000000000000001100100";  -- 100 clock cycles
  comparator_hold_value             <= To_stdlogicvector(comparator_hold_bits);

  lpm_compare_hold : lpm_compare
    generic map (
      lpm_width => 32,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => counter_hold_value,
      datab => comparator_hold_value,
      AgeB => deactivation_timeout_reached
      );

  reset <= not(activated(0)) or deactivated(0);
end delayed_startup_controller_single_architecture;
```

B.5.2 Delayed startup controller for batch testruns

```
-- Delayed startup controller
--
-- Automatically initiates circuit startup and shutdown
-- during unattended test runs
--
-- Modified version for multiple runs on a single SAT instance

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity delayed_startup_controller_series is
  port (
    reset : out std_logic;
    clock : in  std_logic
    );
end delayed_startup_controller_series;

architecture delayed_startup_controller_series_architecture of
     delayed_startup_controller_series is
  component lpm_ff
    generic (
      lpm_width  :     natural;
      lpm_type   :     string;
      lpm_fftype :     string
      );
    port (
      sclr       : in  std_logic;
      clock      : in  std_logic;
      q          : out std_logic_vector (0 downto 0);
      data       : in  std_logic_vector (0 downto 0);
      sset       : in  std_logic
      );
  end component;

  component lpm_counter
    generic (
      lpm_width     :     natural;
      lpm_type      :     string;
      lpm_direction :     string
      );
    port (
      sclr          : in  std_logic;
      clock         : in  std_logic;
      q             : out std_logic_vector (31 downto 0);
      cnt_en        : in  std_logic
      );
  end component;

  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (31 downto 0);
      datab              : in  std_logic_vector (31 downto 0);
      AgeB               : out std_logic
      );
  end component;

  signal activation_timeout_reached   : std_logic;
  signal deactivation_timeout_reached : std_logic;
  signal activated                    : std_logic_vector (0 to 0);
  signal deactivated                  : std_logic_vector (0 to 0);
  signal counter_delay_value          : std_logic_vector (31 downto 0);
  signal counter_hold_value           : std_logic_vector (31 downto 0);
  signal comparator_delay_bits        : bit_vector (31 downto 0);
  signal comparator_delay_value       : std_logic_vector (31 downto 0);
  signal comparator_hold_bits         : bit_vector (31 downto 0);
  signal comparator_hold_value        : std_logic_vector (31 downto 0);
begin
  lpm_ff_activation                   : lpm_ff
    generic map (
      lpm_width  => 1,
      lpm_type   => "LPM_FF",
      lpm_fftype => "DFF"
      )
    port map (
      clock      => clock,
      data       => activated,
      sset       => activation_timeout_reached,
      q          => activated
      );

  lpm_ff_deactivation : lpm_ff
    generic map (
      lpm_width  => 1,
      lpm_type   => "LPM_FF",
      lpm_fftype => "DFF"
      )
    port map (
      clock      => clock,
      data       => deactivated,
      sset       => deactivation_timeout_reached,
      q          => deactivated
      );

  lpm_counter_delay : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      clock         => clock,
      q             => counter_delay_value
      );

  lpm_counter_hold : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      clock         => clock,
      cnt_en        => activated(0),
      q             => counter_hold_value
      );

  comparator_delay_bits(31 downto 0) <= "00000100010001000110000001110000";  -- 5 seconds at
       14.318 MHz
  comparator_delay_value             <= To_stdlogicvector(comparator_delay_bits);

  lpm_compare_delay : lpm_compare
    generic map (
      lpm_width          => 32,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => counter_delay_value,
      datab              => comparator_delay_value,
      AgeB               => activation_timeout_reached
      );

  comparator_hold_bits(31 downto 0) <= "00000000000000000000000001100100";  -- 100 clock cycles
  comparator_hold_value             <= To_stdlogicvector(comparator_hold_bits);

  lpm_compare_hold : lpm_compare
    generic map (
      lpm_width => 32,
      lpm_type => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => counter_hold_value,
      datab => comparator_hold_value,
      AgeB => deactivation_timeout_reached
      );

  reset <= not(deactivated(0));
end delayed_startup_controller_series_architecture;
```

B.5.3 Timeout controller for single testruns

```
-- Timeout controller
--
-- Eliminates problems produced by bouncing or floating reset signals
-- and guarantees precise measurement timeouts

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity timeout_controller_single is
  generic (
    timeout_cycles :     bit_vector (31 downto 0) := "00000100010001000110000001110000"  -- 5
         seconds at 14.318 MHz
    );
  port (
    reset_in       : in  std_logic;
    reset_out      : out std_logic;
    clock          : in  std_logic
    );
end timeout_controller_single;

architecture timeout_controller_single_architecture of timeout_controller_single is
  component lpm_ff
    generic (
      lpm_width  :     natural;
      lpm_type   :     string;
      lpm_fftype :     string
      );
    port (
      sclr       : in  std_logic;
      clock      : in  std_logic;
      q          : out std_logic_vector (0 downto 0);
      data       : in  std_logic_vector (0 downto 0);
      sset       : in  std_logic
      );
  end component;

  component lpm_counter
    generic (
      lpm_width     :     natural;
      lpm_type      :     string;
      lpm_direction :     string
      );
    port (
      sclr          : in  std_logic;
      clock         : in  std_logic;
      q             : out std_logic_vector (31 downto 0);
      cnt_en        : in  std_logic
      );
  end component;

  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (31 downto 0);
      datab              : in  std_logic_vector (31 downto 0);
      AgeB               : out std_logic
      );
  end component;

  signal timeout_reached  : std_logic;
  signal inv_reset_in     : std_logic;
  signal inv_reset_out    : std_logic_vector (0 to 0);
  signal counter_value    : std_logic_vector (31 downto 0);
  signal comparator_bits  : bit_vector (31 downto 0);
  signal comparator_value : std_logic_vector (31 downto 0);
begin
  inv_reset_in <= not(reset_in);

  lpm_ff_component : lpm_ff
    generic map (
      lpm_width  => 1,
      lpm_type   => "LPM_FF",
      lpm_fftype => "DFF"
      )
    port map (
      sclr       => timeout_reached,
      clock      => clock,
      data       => inv_reset_out,
      sset       => inv_reset_in,
      q          => inv_reset_out
      );

  lpm_counter_component : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      sclr          => timeout_reached,
      clock         => clock,
      cnt_en        => inv_reset_out(0),
      q             => counter_value
      );

  comparator_bits(31 downto 0) <= timeout_cycles;
  comparator_value             <= To_stdlogicvector(comparator_bits);

  lpm_compare_component : lpm_compare
    generic map (
      lpm_width => 32,
      lpm_type  => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=YES"
      )
    port map (
      dataa => counter_value,
      datab => comparator_value,
      AgeB => timeout_reached
      );

  reset_out <= not(inv_reset_out(0));
end timeout_controller_single_architecture;
```

B.5.4 Timeout controller for batch testruns

```
-- Timeout controller
--
-- Eliminates problems produced by bouncing or floating reset signals
-- and guarantees precise measurement timeouts
--
-- Modified version for multiple runs on a single SAT instance

library ieee;
use ieee.std_logic_1164.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity timeout_controller_series is
  generic (
    timeout_cycles :     bit_vector (31 downto 0) := "00000100010001000110000001110000"  -- 5
         seconds at 14.318 MHz
    );
  port (
    reset_in       : in  std_logic;
    reset_out      : out std_logic;
    clock          : in  std_logic
    );
end timeout_controller_series;

architecture timeout_controller_series_architecture of timeout_controller_series is
  component lpm_counter
    generic (
      lpm_width     :     natural;
      lpm_type      :     string;
      lpm_direction :     string
      );
    port (
      sclr          : in  std_logic;
      clock         : in  std_logic;
      q             : out std_logic_vector (31 downto 0);
      cnt_en        : in  std_logic
      );
  end component;

  component lpm_compare
    generic (
      lpm_width          :     natural;
      lpm_type           :     string;
      lpm_representation :     string;
      lpm_hint           :     string
      );
    port (
      dataa              : in  std_logic_vector (31 downto 0);
      datab              : in  std_logic_vector (31 downto 0);
      AgeB               : out std_logic
      );
  end component;

  signal counter_enabled  : std_logic;
  signal timeout_reached  : std_logic;
  signal clear_counter    : std_logic;
  signal counter_value    : std_logic_vector (31 downto 0);
  signal comparator_bits  : bit_vector (31 downto 0);
  signal comparator_value : std_logic_vector (31 downto 0);
begin
  lpm_counter_component   : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      sclr          => clear_counter,
      clock         => clock,
      cnt_en        => counter_enabled,
      q             => counter_value
      );

  comparator_bits(31 downto 0) <= timeout_cycles;
  comparator_value             <= To_stdlogicvector(comparator_bits);

  lpm_compare_component : lpm_compare
    generic map (
      lpm_width          => 32,
      lpm_type           => "LPM_COMPARE",
      lpm_representation => "UNSIGNED",
      lpm_hint           => "ONE_INPUT_IS_CONSTANT␣=␣YES"
      )
    port map (
      dataa              => counter_value,
      datab              => comparator_value,
      AgeB               => timeout_reached
      );

  process(clock)
    variable reset_state : std_logic;
  begin
    if (rising_edge(clock)) then
      if (reset_in = '1') then
        reset_state := '1';
        reset_out       <= '1';
        counter_enabled <= '0';
        clear_counter   <= '1';
      elsif (reset_state = '1') then
        reset_state := '0';
        reset_out       <= '0';
        counter_enabled <= '1';
        clear_counter   <= '0';
      elsif (timeout_reached = '1') then
        reset_out <= '1';
        counter_enabled <= '0';
        clear_counter <= '0';
      else
        reset_out <= '0';
        counter_enabled <= '1';
        clear_counter <= '0';
      end if;
    end if;
  end process;
end timeout_controller_series_architecture;
```

B.5.5 Performance counter

```
-- Performance counter
--
-- Counts the number of clock cycles the SAT solver needs to stabilise on a result

library ieee;
use ieee.std_logic_1164.all;
```

```
library lpm;
use lpm.lpm_components.all;

library work;

entity performance_counter is
  port (
    sclr   : in  std_logic;
    clock  : in  std_logic;
    reset  : in  std_logic;
    solved : in  std_logic;
    value  : out std_logic_vector (31 downto 0)
    );
end performance_counter;

architecture performance_counter_architecture of performance_counter is
  component lpm_counter
    generic (
      lpm_width     :     natural;
      lpm_type      :     string;
      lpm_direction :     string
      );
    port (
      sclr          : in  std_logic;
      clock         : in  std_logic;
      q             : out std_logic_vector (31 downto 0);
      cnt_en        : in  std_logic
      );
  end component;

  signal counter_enable : std_logic;
  signal output         : std_logic_vector (31 downto 0);
begin
  counter_enable <= reset nor solved;

  lpm_counter_component : lpm_counter
    generic map (
      lpm_width     => 32,
      lpm_type      => "LPM_COUNTER",
      lpm_direction => "UP"
      )
    port map (
      sclr          => sclr,
      clock         => clock,
      cnt_en        => counter_enable,
      q => output
      );

  value <= output;
end performance_counter_architecture;
```

B.5.6 Memory controller for single testruns

```
-- Memory controller
--
-- Writes result data to attached memory block

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity memory_controller_single is
  generic (
    variable_count :     integer range 1 to 4000 := 1
    );
  port (
    reset          : in  std_logic;
    clock          : in  std_logic;
    variables      : in  std_logic_vector (1 to variable_count);
    solved         : in  std_logic;
    performance    : in  std_logic_vector (31 downto 0);
    data           : out std_logic_vector (31 downto 0);
    address        : out std_logic_vector (6 downto 0);
    write_enable   : out std_logic
    );
end memory_controller_single;

architecture memory_controller_single_architecture of memory_controller_single is
  signal     variable_buffer : std_logic_vector (0 to (variable_count + 32 - 1));
  signal     solved_buffer   : std_logic;
  signal     write_address   : std_logic_vector (6 downto 0);
begin
  process(clock)
    variable bits            : integer range 96 to 4096;
    variable slices          : integer range 3 to 125;
    variable current_slice   : integer range 0 to 127;
    variable offset          : integer range 0 to (4096 - 32);
  begin
    bits     := 96 + variable_count;
    slices   := bits / 32;
    if ((bits mod 32) /= 0) then
      slices := slices + 1;
    end if;

    if (rising_edge(clock)) then
      if (reset = '1') then
        current_slice   := To_integer(unsigned(write_address));
        current_slice   := current_slice + 1;
        if (current_slice >= slices) then
          current_slice := 0;
        end if;

        if (current_slice = 0) then
          data               <= std_logic_vector(To_unsigned(variable_count , 32));
        elsif (current_slice = 1) then
          data               <= performance;
        elsif (current_slice = 2) then
          if (solved_buffer = '1') then
            data             <= "00000000000000000000000000000001";
          else
            data             <= "00000000000000000000000000000000";
          end if;
        else
          offset := (current_slice - 3) * 32;
          for index in 0 to 31 loop
            data(31 - index) <= variable_buffer(offset + index);
          end loop;
        end if;

        write_address   <= STD_LOGIC_VECTOR(To_unsigned(current_slice , 7));
        address         <= write_address;
      else
        variable_buffer <= variables & "00000000000000000000000000000000";
        solved_buffer <= solved;
      end if;
    end if;
  end process;

  write_enable <= reset;
end memory_controller_single_architecture;
```

B.5.7 Memory controller for batch testruns

```
-- Memory controller
--
-- Writes result data of single instance test series to attached memory block

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library lpm;
use lpm.lpm_components.all;

library work;

entity memory_controller_series is
  generic (
    variable_count :      integer range 1 to 4000 := 1
    );
  port (
    reset           : in  std_logic;
    clock           : in  std_logic;
    variables       : in  std_logic_vector (1 to variable_count);
    solved          : in  std_logic;
    performance     : in  std_logic_vector (31 downto 0);
    data            : out std_logic_vector (31 downto 0);
    address         : out std_logic_vector (8 downto 0);
    write_enable    : out std_logic;
    restart         : out std_logic
    );
end memory_controller_series;

architecture memory_controller_series_architecture of memory_controller_series is
  signal     write_data      : std_logic_vector (31 downto 0);
  signal     write_address   : std_logic_vector (8 downto 0);
  signal     restart_cycle   : std_logic;
begin
  process(clock)
    variable current_slot   : integer range 0 to 511;
    variable checksum       : std_logic_vector (31 downto 0);
    variable write_checksum : std_logic;
    variable result_written : std_logic;
  begin
    if (rising_edge(clock)) then
      if (reset = '1') then
        write_data    <= "00000000000000000000000000000000";
        write_address <= "000000000";
        current_slot   := 0;
        checksum       := "00000000000000000000000000000000";
        write_checksum := '0';
        result_written := '0';
        restart_cycle <= '1';
      elsif (write_checksum = '1') then
        write_data    <= checksum;
        write_address <= std_logic_vector(To_unsigned(current_slot, 9));
        restart_cycle <= '0';
      elsif ((solved = '1') and (result_written = '0')) then
        write_data    <= solved & performance(30 downto 0);
        write_address <= std_logic_vector(To_unsigned(current_slot, 9));
        checksum       := (checksum(23 downto 0) & checksum(31 downto 24)) xor (solved &
             performance(30 downto 0));
        result_written := '1';
        restart_cycle <= '1';

        current_slot      := current_slot + 1;
        if (current_slot >= 256) then
          write_checksum := '1';
        end if;
      elsif (solved = '0') then
        result_written   := '0';
        restart_cycle <= '0';
      end if;
    end if;
  end process;

  data <= write_data;
  address <= write_address;
  write_enable <= '1';
  restart <= restart_cycle;
end memory_controller_series_architecture;
```

B.5.8 RAM interface (4K)

```
-- RAM interface providing 4K SRAM accessable by host computer

library ieee;
use ieee.std_logic_1164.all;

library altera_mf;
use altera_mf.all;

entity ram_interface_4k is
  port (
    address : in  std_logic_vector (6 downto 0);
    clock   : in  std_logic;
    data    : in  std_logic_vector (31 downto 0);
    wren    : in  std_logic := '1';
    q       : out std_logic_vector (31 downto 0)
    );
end ram_interface_4k;

architecture SYN of ram_interface_4k is
  component altsyncram
    generic (
      address_aclr_a         :     string;
      indata_aclr_a          :     string;
      intended_device_family :     string;
      lpm_hint               :     string;
      lpm_type               :     string;
      numwords_a             :     natural;
      operation_mode         :     string;
      outdata_aclr_a         :     string;
      outdata_reg_a          :     string;
      power_up_uninitialized :     string;
      widthad_a              :     natural;
      width_a                :     natural;
      width_byteena_a        :     natural;
      wrcontrol_aclr_a       :     string
      );
    port (
      wren_a                 : in  std_logic;
      clock0                 : in  std_logic;
      address_a              : in  std_logic_vector (6 downto 0);
      q_a                    : out std_logic_vector (31 downto 0);
      data_a                 : in  std_logic_vector (31 downto 0)
      );
  end component;


  signal output_word   : std_logic_vector (31 downto 0);
begin
  altsyncram_component : altsyncram
    generic map (
      address_aclr_a         => "NONE",
      indata_aclr_a          => "NONE",
      intended_device_family => "Cyclone",
      lpm_hint               => "ENABLE_RUNTIME_MOD␣=␣YES,␣INSTANCE_NAME␣=␣RSLT",
      lpm_type               => "altsyncram",
      numwords_a             => 128,
      operation_mode         => "SINGLE_PORT",
      outdata_aclr_a         => "NONE",
      outdata_reg_a          => "UNREGISTERED",
      power_up_uninitialized => "FALSE",
      widthad_a              => 7,
      width_a                => 32,
      width_byteena_a        => 1,
      wrcontrol_aclr_a       => "NONE"
      )
    PORT MAP (
      wren_a => wren,
      clock0 => clock,
      address_a => address,
      data_a => data,
      q_a => output_word
      );

  q <= output_word(31 downto 0);
end SYN;
```

B.5.9 RAM interface (16K)

```
-- RAM interface providing 16K SRAM accessable by host computer

library ieee;
use ieee.std_logic_1164.all;

library altera_mf;
use altera_mf.all;

entity ram_interface_16k is
  port (
    address : in  std_logic_vector (8 downto 0);
    clock   : in  std_logic;
    data    : in  std_logic_vector (31 downto 0);
    wren    : in  std_logic := '1';
    q       : out std_logic_vector (31 downto 0)
    );
end ram_interface_16k;

architecture SYN of ram_interface_16k is
  component altsyncram
    generic (
      address_aclr_a         :     string;
      indata_aclr_a          :     string;
      intended_device_family :     string;
      lpm_hint               :     string;
      lpm_type               :     string;
      numwords_a             :     natural;
      operation_mode         :     string;
      outdata_aclr_a         :     string;
      outdata_reg_a          :     string;
      power_up_uninitialized :     string;
      widthad_a              :     natural;
      width_a                :     natural;
      width_byteena_a        :     natural;
      wrcontrol_aclr_a       :     string
      );
    port (
      wren_a                 : in  std_logic;
      clock0                 : in  std_logic;
      address_a              : in  std_logic_vector (8 downto 0);
      q_a                    : out std_logic_vector (31 downto 0);
      data_a                 : in  std_logic_vector (31 downto 0)
      );
  end component;

  signal output_word   : std_logic_vector (31 downto 0);
begin
  altsyncram_component : altsyncram
    generic map (
      address_aclr_a         => "NONE",
      indata_aclr_a          => "NONE",
      intended_device_family => "Cyclone",
      lpm_hint               => "ENABLE_RUNTIME_MOD␣=␣YES,␣INSTANCE_NAME␣=␣RSLT",
      lpm_type               => "altsyncram",
      numwords_a             => 512,
      operation_mode         => "SINGLE_PORT",
      outdata_aclr_a         => "NONE",
      outdata_reg_a          => "UNREGISTERED",
      power_up_uninitialized => "FALSE",
      widthad_a              => 9,
      width_a                => 32,
      width_byteena_a        => 1,
      wrcontrol_aclr_a       => "NONE"
      )
    PORT MAP (
      wren_a => wren,
      clock0 => clock,
      address_a => address,
      data_a => data,
      q_a => output_word
      );

  q <= output_word(31 downto 0);
end SYN;
```

Appendix C

Top level circuit setups

C.1 Basic asynchronous circuitry

```
-- Main module used in experiments with
-- basic asynchronous circuits
--
-- Number of variables is set to 10
-- Timeout is set to 71590000 clock cycles

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in    std_logic;
    zero_b        : in    std_logic;
    zero_c        : in    std_logic;
    clock_base    : in    std_logic;
    counter_reset : in    std_logic;
    stabiliser    : inout std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset  : in  std_logic;
      zero_a : in  std_logic;
      zero_b : in  std_logic;
      zero_c : in  std_logic;
      output : out std_logic_vector (1 to 10);
      solved : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 10);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  component opndrn
    port(
      A_IN  : in  std_logic;
      A_OUT : out std_logic
      );
  end component;

  signal solver_reset        : std_logic;
  signal global_reset        : std_logic;
  signal clear_counter       : std_logic;
  signal solution_found      : std_logic;
  signal performance_count   : std_logic_vector(31 downto 0);
  signal truth_assignment    : std_logic_vector(1 to 10);
  signal memory_write_enable : std_logic;
  signal memory_address      : std_logic_vector(6 downto 0);
  signal memory_data         : std_logic_vector(31 downto 0);
begin
  sat_solver_instance        : sat_solver
    port map(
      reset  => solver_reset,
      zero_a => zero_a,
      zero_b => zero_b,
      zero_c => zero_c,
      solved => solution_found,
      output => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);

  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => stabiliser,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 10
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => stabiliser,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );

  tri_state_buffer : opndrn
    port map(
      A_IN  => solution_found,
      A_OUT => stabiliser
      );
end;
```

C.2 Basic synchronous circuitry

```
-- Main module used in experiments with
-- basic synchronous circuits
--
-- Number of variables is set to 10
-- Timeout is set to 71590000 clock cycles

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset  : in  std_logic;
      clock  : in  std_logic;
      zero_a : in  std_logic;
      zero_b : in  std_logic;
      zero_c : in  std_logic;
      output : out std_logic_vector (1 to 10);
      solved : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 10);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal solver_reset        : std_logic;
  signal global_reset        : std_logic;
  signal clear_counter       : std_logic;
  signal solution_found      : std_logic;
  signal performance_count   : std_logic_vector(31 downto 0);
  signal truth_assignment    : std_logic_vector(1 to 10);
  signal memory_write_enable : std_logic;
  signal memory_address      : std_logic_vector(6 downto 0);
  signal memory_data         : std_logic_vector(31 downto 0);
begin
  sat_solver_instance        : sat_solver
    port map(
      reset  => solver_reset,
      clock  => clock_base,
      zero_a => zero_a,
      zero_b => zero_b,
      zero_c => zero_c,
      solved => solution_found,
      output => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);
  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 10
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

C.3 Basic probability driven asynchronous circuitry

```
-- Main module used in experiments with
-- early globally probability driven circuits
--
-- Number of variables is set to 10
-- Number of clauses is set to 50
-- Timeout is set to 71590000 clock cycles
-- Base probability for a selection bit issued is set to 0.3340

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset     : in  std_logic;
      clock     : in  std_logic;
      zero_a    : in  std_logic;
      zero_b    : in  std_logic;
      zero_c    : in  std_logic;
      wrong_sel : in  std_logic_vector (149 downto 0);
      output    : out std_logic_vector (1 to 10);
      solved    : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component fixed_distribution_bit_source_basic_lfsr
    generic (
      output_bits        :     integer;
      probability_factor :     bit_vector (9 downto 0)
      );
    port(
      reset              : in  std_logic;
      clock              : in  std_logic;
      bits               : out std_logic_vector(149 downto 0)
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 10);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal solver_reset         : std_logic;
  signal wrong_selection_bits : std_logic_vector(149 downto 0);
  signal global_reset         : std_logic;
  signal clear_counter        : std_logic;
  signal solution_found       : std_logic;
  signal performance_count    : std_logic_vector(31 downto 0);
  signal truth_assignment     : std_logic_vector(1 to 10);
  signal memory_write_enable  : std_logic;
  signal memory_address       : std_logic_vector(6 downto 0);
  signal memory_data          : std_logic_vector(31 downto 0);
begin
  sat_solver_instance         : sat_solver
    port map(
      reset     => solver_reset,
      clock     => clock_base,
      zero_a    => zero_a,
      zero_b    => zero_b,
      zero_c    => zero_c,
      wrong_sel => wrong_selection_bits,
      solved    => solution_found,
      output    => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);

  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  selection_bit_source_instance : fixed_distribution_bit_source_basic_lfsr
    generic map(
      output_bits        => 150,
      probability_factor => "1010101010"
      )
    port map(
      reset              => solver_reset,
      clock              => clock_base,
      bits               => wrong_selection_bits
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 10
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

C.4 Template for globally probability driven circuitry

```
-- Main module used in most experiments with
-- globally probability driven circuits
--
-- Number of variables is set to 100
-- Number of clauses is set to 370
-- Timeout is set to 71590000 clock cycles
-- Base probability for a selection bit issued is set to 0.3340

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset     : in  std_logic;
      clock     : in  std_logic;
      zero_a    : in  std_logic;
      zero_b    : in  std_logic;
      zero_c    : in  std_logic;
      wrong_sel : in  std_logic_vector (1109 downto 0);
      output    : out std_logic_vector (1 to 100);
      solved    : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component fixed_distribution_bit_source_multi_lfsr
    generic (
      output_bits        :     integer;
      probability_factor :     bit_vector (9 downto 0)
      );
    port(
      reset              : in  std_logic;
      clock              : in  std_logic;
      bits               : out std_logic_vector(1109 downto 0)
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 100);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal solver_reset         : std_logic;
  signal wrong_selection_bits : std_logic_vector(1109 downto 0);
  signal global_reset         : std_logic;
  signal clear_counter        : std_logic;
  signal solution_found       : std_logic;
  signal performance_count    : std_logic_vector(31 downto 0);
  signal truth_assignment     : std_logic_vector(1 to 100);
  signal memory_write_enable  : std_logic;
  signal memory_address       : std_logic_vector(6 downto 0);
  signal memory_data          : std_logic_vector(31 downto 0);
begin
  sat_solver_instance         : sat_solver
    port map(
      reset     => solver_reset,
      clock     => clock_base,
      zero_a    => zero_a,
      zero_b    => zero_b,
      zero_c    => zero_c,
      wrong_sel => wrong_selection_bits,
      solved    => solution_found,
      output    => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);

  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  selection_bit_source_instance : fixed_distribution_bit_source_multi_lfsr
    generic map(
      output_bits        => 1110,
      probability_factor => "1010101010"
      )
    port map(
      reset              => solver_reset,
      clock              => clock_base,
      bits               => wrong_selection_bits
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 100
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

C.5 Template for single instance batch testruns

```
-- Main module used in runtime variance experiments
--
-- Number of variables is set to 100
-- Number of clauses is set to 370
-- Timeout is set to 71590000 clock cycles
-- Base probability for a selection bit issued is set to 0.0908
-- Selection bit source is preseeded according base probability

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset     : in  std_logic;
      clock     : in  std_logic;
      zero_a    : in  std_logic;
      zero_b    : in  std_logic;
      zero_c    : in  std_logic;
      wrong_sel : in  std_logic_vector (1109 downto 0);
      output    : out std_logic_vector (1 to 100);
      solved    : out std_logic
      );
  end component;

  component delayed_startup_controller_series
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_series
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component fixed_distribution_bit_source_multi_lfsr_preseeded
    generic (
      output_bits        :     integer;
      probability_factor :     bit_vector (9 downto 0);
      seed               :     bit_vector (1109 downto 0)
      );
    port(
      reset              : in  std_logic;
      clock              : in  std_logic;
      bits               : out std_logic_vector(1109 downto 0)
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_series
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 100);
      write_enable   : out std_logic;
      address        : out std_logic_vector(8 downto 0);
      data           : out std_logic_vector(31 downto 0);
      restart        : out std_logic
      );
  end component;

  component ram_interface_16k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(8 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal global_reset  : std_logic;
  signal restart_cycle : std_logic;
  signal solver_reset  : std_logic;

  signal wrong_selection_bits : std_logic_vector(1109 downto 0);
```

```
  signal solution_found       : std_logic;
  signal performance_count    : std_logic_vector(31 downto 0);
  signal truth_assignment     : std_logic_vector(1 to 100);
  signal memory_write_enable  : std_logic;
  signal memory_address       : std_logic_vector(8 downto 0);
  signal memory_data          : std_logic_vector(31 downto 0);
begin
  sat_solver_instance          : sat_solver
    port map(
      reset     => solver_reset,
      clock     => clock_base,
      zero_a    => zero_a,
      zero_b    => zero_b,
      zero_c    => zero_c,
      wrong_sel => wrong_selection_bits,
      solved    => solution_found,
      output    => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_series
    port map(
      clock => clock_base,
      reset => global_reset
      );

  timeout_controller_instance : timeout_controller_series
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => restart_cycle,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  selection_bit_source_instance : fixed_distribution_bit_source_multi_lfsr_preseeded
    generic map(
      output_bits        => 1110,
      probability_factor => "1110100011",
      seed               => "
          1000000000000000000000001000001000000000000010000000000100000000000000000000100000001000000000000000010000100000
          "
      )
    port map(
      reset              => global_reset,
      clock              => clock_base,
      bits               => wrong_selection_bits
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => restart_cycle,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_series
    generic map(
      variable_count => 100
      )
    port map(
      reset          => global_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data,
      restart        => restart_cycle
      );

  memory_wrapper : ram_interface_16k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

C.6 Template for simulated annealing experiments

```
-- Main module used in experiments with
-- simulated annealing techniques
--
-- Number of variables is set to 100
-- Number of clauses is set to 370
-- Timeout is set to 71590000 clock cycles
-- Base probability for a selection bit issued is set to 0.0791

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset     : in  std_logic;
      clock     : in  std_logic;
      zero_a    : in  std_logic;
      zero_b    : in  std_logic;
      zero_c    : in  std_logic;
      wrong_sel : in  std_logic_vector (1109 downto 0);
      output    : out std_logic_vector (1 to 100);
      solved    : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component fixed_distribution_bit_source_simulated_annealing
    generic (
      output_bits        :     integer;
      probability_factor :     bit_vector (9 downto 0)
      );
    port(
      reset              : in  std_logic;
      clock              : in  std_logic;
      bits               : out std_logic_vector(1109 downto 0)
      );
  end component;

  component performance_counter
```

```
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 100);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal solver_reset         : std_logic;
  signal wrong_selection_bits : std_logic_vector(1109 downto 0);
  signal global_reset         : std_logic;
  signal clear_counter        : std_logic;
  signal solution_found       : std_logic;
  signal performance_count    : std_logic_vector(31 downto 0);
  signal truth_assignment     : std_logic_vector(1 to 100);
  signal memory_write_enable  : std_logic;
  signal memory_address       : std_logic_vector(6 downto 0);
  signal memory_data          : std_logic_vector(31 downto 0);
begin
  sat_solver_instance         : sat_solver
    port map(
      reset     => solver_reset,
      clock     => clock_base,
      zero_a    => zero_a,
      zero_b    => zero_b,
      zero_c    => zero_c,
      wrong_sel => wrong_selection_bits,
      solved    => solution_found,
      output    => truth_assignment
      );

  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);

  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  selection_bit_source_instance : fixed_distribution_bit_source_simulated_annealing
```

```
    generic map(
      output_bits        => 1110,
      probability_factor => "1110101111"
      )
    port map(
      reset              => solver_reset,
      clock              => clock_base,
      bits               => wrong_selection_bits
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 100
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

C.7 Template for locally probability driven circuitry

```
-- Main module used in experiments with
-- locally probability driven circuits
--
-- Number of variables is set to 100
-- Timeout is set to 71590000 clock cycles

library ieee;
use ieee.std_logic_1164.all;

library work;

entity Sample is
  port (
    zero_a        : in std_logic;
    zero_b        : in std_logic;
    zero_c        : in std_logic;
    clock_base    : in std_logic;
    counter_reset : in std_logic
    );
end Sample;

architecture bdf_type of Sample is
  component sat_solver
    port(
      reset  : in  std_logic;
      clock  : in  std_logic;
      zero_a : in  std_logic;
      zero_b : in  std_logic;
      zero_c : in  std_logic;
      output : out std_logic_vector (1 to 100);
      solved : out std_logic
      );
  end component;

  component delayed_startup_controller_single
    port(
      clock : in  std_logic;
      reset : out std_logic
      );
  end component;

  component timeout_controller_single
    generic (
      timeout_cycles :     bit_vector (31 downto 0)
      );
    port(
      reset_in       : in  std_logic;
      clock          : in  std_logic;
      reset_out      : out std_logic
      );
  end component;

  component performance_counter
    port(
      sclr   : in  std_logic;
      clock  : in  std_logic;
      reset  : in  std_logic;
      solved : in  std_logic;
      value  : out std_logic_vector(31 downto 0)
      );
  end component;

  component memory_controller_single
    generic (
      variable_count :     integer
      );
    port(
      reset          : in  std_logic;
      clock          : in  std_logic;
      solved         : in  std_logic;
      performance    : in  std_logic_vector(31 downto 0);
      variables      : in  std_logic_vector(1 to 100);
      write_enable   : out std_logic;
      address        : out std_logic_vector(6 downto 0);
      data           : out std_logic_vector(31 downto 0)
      );
  end component;

  component ram_interface_4k
    port(
      wren    : in  std_logic;
      clock   : in  std_logic;
      address : in  std_logic_vector(6 downto 0);
      data    : in  std_logic_vector(31 downto 0);
      q       : out std_logic_vector(31 downto 0)
      );
  end component;

  signal solver_reset        : std_logic;
  signal global_reset        : std_logic;
  signal clear_counter       : std_logic;
  signal solution_found      : std_logic;
  signal performance_count   : std_logic_vector(31 downto 0);
  signal truth_assignment    : std_logic_vector(1 to 100);
  signal memory_write_enable : std_logic;
  signal memory_address      : std_logic_vector(6 downto 0);
  signal memory_data         : std_logic_vector(31 downto 0);
begin
  sat_solver_instance        : sat_solver
    port map(
      reset  => solver_reset,
      clock  => clock_base,
      zero_a => zero_a,
      zero_b => zero_b,
      zero_c => zero_c,
      solved => solution_found,
      output => truth_assignment
      );
  delayed_startup_controller_instance : delayed_startup_controller_single
    port map(
      clock => clock_base,
      reset => global_reset
      );

  clear_counter <= not(counter_reset);

  timeout_controller_instance : timeout_controller_single
    generic map(
      timeout_cycles => "00000100010001000110000001110000"
      )
    port map(
      reset_in       => global_reset,
      clock          => clock_base,
      reset_out      => solver_reset
      );

  performance_counter_instance : performance_counter
    port map(
      sclr   => clear_counter,
      clock  => clock_base,
      reset  => solver_reset,
      solved => solution_found,
      value  => performance_count
      );

  memory_controller_instance : memory_controller_single
    generic map(
      variable_count => 100
      )
    port map(
      reset          => solver_reset,
      clock          => clock_base,
      solved         => solution_found,
      performance    => performance_count,
      variables      => truth_assignment,
      write_enable   => memory_write_enable,
      address        => memory_address,
      data           => memory_data
      );

  memory_wrapper : ram_interface_4k
    port map(
      wren    => memory_write_enable,
      clock   => clock_base,
      address => memory_address,
      data    => memory_data
      );
end;
```

Appendix D

Result tables of experiments

D.1 Automated experiments on small SAT instances

			Fully deterministic circuit				**Probability driven circuit**				**Asynchronous circuit**			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	No		No	0111101101	No		No	0100111110	No		No	0111001001	No	
002	Yes	01011001000	No	0011110110	No		Yes	1000100010	Yes	140	Yes	0000110010	Yes	57931
003	Yes	00101001000	No	0010011110	No		Yes	0000110011	Yes	706	No	0010101100	Yes / Undetected	
004	Yes	00111100000	No	1111011110	No		Yes	0001110111	Yes	126	Yes	1011111000	Yes	29560001
005	Yes	00000000000	Yes	0000000000	Yes	0	Yes	0000000000	Yes	0	Yes	0000000000	Yes	0
006	Yes	00000110000	No	1010101010	No		Yes	1001011000	Yes	34	Yes	0000011001	Yes	177454
007	Yes	01001000000	No	0101100000	No		Yes	0100100000	Yes	145	Yes	0111101100	Yes	43948
008	Yes	00010000100	No	0101110111	No		Yes	1011110010	Yes	182	Yes	1011010010	Yes	118750
009	Yes	01100000100	No	1001010011	No		Yes	0110100111	Yes	241	Yes	0010101011	Yes	2243348
010	Yes	00100000100	No	0110111111	No		Yes	1111101100	Yes	142	Yes	0111000001	Yes	354402
011	Yes	11001011000	No	1111111111	No		Yes	1111100100	Yes	77	No	0010110000	No	
012	Yes	00001110100	No	1110011111	No		Yes	1000110110	Yes	118	Yes	1010000101	Yes	15176
013	Yes	00000110000	No	1011111011	No		Yes	1100110101	Yes	207	Yes	1000010101	Yes	5311567
014	Yes	01110000100	No	0001111011	No		Yes	1000011100	Yes	139	Yes	1000011100	Yes	8184
015	Yes	01111111000	No	0110101010	No		Yes	1010111100	Yes	133	Yes	0111011110	Yes	500651
016	Yes	01000000000	No	1100010110	No		Yes	1000011101	Yes	31	No	0000100100	Yes / Undetected	
017	Yes	01010001010	No	1101111110	No		Yes	1000111111	Yes	215	Yes	0101100101	Yes	114642
018	Yes	01101010000	No	1101101010	No		Yes	0001000110	Yes	141	Yes	1000101100	Yes	1205
019	Yes	01101001110	Yes	0111111111	Yes	0	Yes	0111111111	Yes	0	Yes	1011111000	Yes	4391
020	Yes	00101101100	No	0001011111	No		Yes	1101011100	Yes	250	Yes	1001011000	Yes	485289
021	Yes	00001000100	No	0011101010	No		Yes	1010111010	Yes	66	Yes	0000100010	Yes	0
022	Yes	00000000000	Yes	0000000000	Yes	0	Yes	0000000000	Yes	0	Yes	0000000000	Yes	0
023	Yes	00000100100	No	0100111111	No		Yes	0110010010	Yes	130	Yes	1110010010	Yes	172774
024	Yes	00101000010	No	1111100111	No		Yes	1110111110	Yes	132	Yes	0010100001	Yes	339562
025	Yes	00010010000	No	0100101111	No		Yes	0001011010	Yes	135	Yes	0101001100	Yes	9445
026	Yes	01111101000	No	1101101100	No		Yes	1111010110	Yes	122	Yes	0101101110	Yes	404611
027	Yes	00010100000	No	1011111011	No		Yes	1001010100	Yes	184	Yes	0001010001	Yes	448960
028	Yes	00000010000	No	0110000000	No		Yes	0010100111	Yes	81	Yes	0000010111	Yes	2113
029	Yes	01101100000	No	0111111101	No		Yes	1111110111	Yes	68	No	0001011001	No	
030	Yes	10000000000	No	0011010110	No		Yes	1100000100	Yes	1386	Yes	1001100000	Yes	7926

Table D.1: Performance of early circuit variants on SAT instances consisting of 10 variables and 30 clauses

			Fully deterministic circuit				Probability driven circuit				Asynchronous circuit			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	No		No	0111111000	No		No	1011101110	No		No	0010001110	No	
002	Yes	01010010110	No	1111101010	No		Yes	1101001111	Yes	165	Yes	0100101011	Yes	87179
003	No		No	0110111011	No		No	1100010001	No		No	1001001011	No	
004	Yes	10000101110	No	0101111111	No		Yes	1000010111	Yes	279	No	1000111100	No	
005	Yes	00000110010	No	1010010101	No		Yes	1101110001	Yes	321	No	0111010010	No	
006	Yes	00000101110	No	1101111100	No		Yes	1000010110	Yes	185	Yes	1001010111	Yes	2044969
007	Yes	01011001100	No	1001110001	No		Yes	0000111101	Yes	579	No	1010001101	No	
008	Yes	01100011110	No	1011101010	No		Yes	0110001111	Yes	1590	No	0101011010	No	
009	Yes	01000010010	No	1110111101	No		Yes	0101001001	Yes	172	Yes	0100001011	Yes	55185231
010	Yes	01110000010	No	1101001010	No		Yes	0110101111	Yes	183	No	0000101101	No	
011	Yes	00000010000	No	0011101110	No		Yes	0000101000	Yes	241	No	1001101001	No	
012	No		No	1101111100	No		No	0010101111	No		No	1100000010	No	
013	Yes	01111101110	No	1101101011	No		Yes	1110100101	Yes	4026	No	1011100100	No	
014	Yes	11010001100	No	1111110111	No		Yes	1100010111	Yes	196	No	0101001100	No	
015	Yes	01000101000	No	1111110110	No		Yes	0110011100	Yes	165	Yes	0100010100	Yes	7757107
016	Yes	00000000010	No	0000110111	No		Yes	1111110010	Yes	192	Yes	0000000001	Yes	13471061
017	Yes	11110101110	No	0110111110	No		Yes	1111010111	Yes	1970	No	0101111111	No	
018	Yes	00011000000	No	1111111110	No		Yes	0001100100	Yes	188	Yes	0001100000	Yes	50356
019	No		No	0000010110	No		No	1101111101	No		No	1111111101	No	
020	Yes	00111110010	No	0101110111	No		Yes	0110010101	Yes	1125	No	1011111001	Yes / Undetected	
021	Yes	10101011000	No	1110111011	No		Yes	1010101100	Yes	678	No	0101010001	No	
022	Yes	00111001000	No	1110110110	No		Yes	1001010100	Yes	103	Yes	1001010110	Yes	62374043
023	Yes	01001011000	No	1101111011	No		Yes	1110101101	Yes	823	Yes	1001011101	Yes	21561495
024	Yes	00011101100	No	1010111000	No		Yes	0001111110	Yes	201	No	0101110000	No	
025	Yes	01010100100	No	1101111001	No		Yes	1101010011	Yes	219	Yes	1100010010	Yes	893659
026	Yes	01111000000	No	1010100101	No		Yes	0111100000	Yes	238	No	0011000001	No	
027	No		No	0000101001	No		No	0000000000	No		No	0110110010	No	
028	No		No	0011000011	No		No	0111111111	No		No	0100000011	No	
029	Yes	11111010000	No	1100011011	No		Yes	1111101000	Yes	1991	No	0111001010	No	
030	No		No	1110011101	No		No	1110001100	No		No	0010111011	No	

Table D.2: Performance of early circuit variants on SAT instances consisting of 10 variables and 40 clauses

			Fully deterministic circuit				**Probability driven circuit**				**Asynchronous circuit**			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	Yes	10011100000	No	1110111101	No		Yes	1001110000	Yes	2010	No	0111010001	No	
002	No		No	1011111100	No		No	0010000000	No		No	0000010110	No	
003	Yes	01001000110	No	1101110111	No		Yes	0100100011	Yes	5581	No	0111010111	No	
004	No		No	0111100011	No		No	0001101010	No		No	1111000010	No	
005	Yes	01100011000	No	0111101110	No		Yes	1110001100	Yes	7359	No	1110001011	No	
006	No		No	0001111101	No		No	0000101000	No		No	0000000000	No	
007	Yes	11100000000	No	0100101101	No		Yes	1110000000	Yes	6030	No	1111100111	No	
008	Yes	01011101000	No	1011011110	No		Yes	1100110100	Yes	249	No	0110110011	No	
009	No		No	1111110111	No		No	1111011001	No		No	1000100110	No	
010	No		No	0111101100	No		No	0110110000	No		No	1110100101	No	
011	No		No	1011111101	No		No	0111100001	No		No	0010011010	No	
012	No		No	1100111100	No		No	1110111111	No		No	0100000001	No	
013	Yes	11010011100	No	1101111011	No		Yes	1101001110	Yes	4636	No	0001011010	No	
014	No		No	0111111011	No		No	0000100110	No		No	0000111110	No	
015	No		No	0011101101	No		No	0000111101	No		No	1000101000	No	
016	No		No	0111111110	No		No	1110000010	No		No	0000100110	No	
017	No		No	0101010100	No		No	0100000111	No		No	0100100100	No	
018	No		No	0101111111	No		No	0100111101	No		No	1001110111	No	
019	Yes	01100010000	No	0110111110	No		Yes	0110001000	Yes	204	Yes	0110001000	Yes	24853717
020	Yes	01010000000	No	0111111111	No		Yes	0101000000	Yes	7341	No	0100000110	No	
021	Yes	10111000000	No	0111111111	No		Yes	1111001000	Yes	4648	Yes	1011100001	Yes	17609705
022	Yes	11100001100	No	1101101100	No		Yes	1110100110	Yes	1387	No	0010111010	No	
023	Yes	11010010000	No	1101001110	No		Yes	1101001000	Yes	115	No	1000001010	No	
024	Yes	00001110000	No	0110111111	No		Yes	0011010110	Yes	169	No	1111011000	No	
025	No		No	1100111010	No		No	1111110100	No		No	0011011000	No	
026	No		No	1111111111	No		No	0110011100	No		No	1010011000	No	
027	Yes	00100111000	No	0011111101	No		Yes	0010011100	Yes	216	No	0101010001	No	
028	No		No	1111111111	No		No	1000110000	No		No	0100101010	No	
029	Yes	01000100000	No	1000010010	No		Yes	0110001110	Yes	94	Yes	0100010000	Yes	4682
030	No		No	1111111011	No		No	1101100101	No		No	1001001111	No	

Table D.3: Performance of early circuit variants on SAT instances consisting of 10 variables and 50 clauses

			Fully deterministic circuit				Probability driven circuit				Asynchronous circuit			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	No		No	1101101111	No		No	1100000010	No		No	0001100011	No	
002	No		No	1111111111	No		No	0110100100	No		No	1111001001	No	
003	Yes	01100010010	No	1110101101	No		Yes	0110001001	Yes	1460	No	0110110010	No	
004	No		No	1111111111	No		No	0001110110	No		No	1011010101	No	
005	No		No	0111111111	No		No	0001110011	No		No	0111010001	No	
006	No		No	0010011111	No		No	1101101110	No		No	0010100110	No	
007	No		No	0111110111	No		No	0111110000	No		No	1000100100	No	
008	No		No	1111100111	No		No	0000000111	No		No	0011010101	No	
009	No		No	1111111100	No		No	0100001100	No		No	0110001010	No	
010	No		No	0011101110	No		No	1110100001	No		No	0111101110	No	
011	No		No	1111011110	No		No	0111101111	No		No	1011101010	No	
012	No		No	1111111111	No		No	1101100001	No		No	1110100010	No	
013	No		No	0111111110	No		No	1101110110	No		No	1010110011	No	
014	No		No	1111111011	No		No	1001011011	No		No	0101100001	No	
015	No		No	1111111010	No		No	1011011111	No		No	1000011111	No	
016	No		No	1101010110	No		No	0111100010	No		No	0011000110	No	
017	No		No	1111111110	No		No	1110011100	No		No	0010110011	No	
018	Yes	00010010000	No	1101011111	No		Yes	0001001000	Yes	1113	No	0100111100	No	
019	Yes	11111111110	Yes	1111111111	Yes	0	Yes	1111111111	Yes	0	No	1100111111	No	
020	No		No	1111111111	No		No	0101101011	No		No	1001110011	No	
021	No		No	1111111111	No		No	0110110101	No		No	0100001000	No	
022	Yes	00010101010	No	1011111111	No		Yes	0001010101	Yes	512	No	0010001001	No	
023	Yes	00000111000	No	1101111010	No		Yes	0000011100	Yes	8482	No	1100110101	No	
024	Yes	01110110100	No	0111111111	No		Yes	0111011011	Yes	3989	No	1000110011	No	
025	No		No	1111111111	No		No	0101100101	No		No	1100101010	No	
026	No		No	0111111110	No		No	0101000100	No		No	1010101010	No	
027	No		No	1111111111	No		No	0010100110	No		No	1101111010	No	
028	No		No	1111111110	No		No	0111000010	No		No	1101101100	No	
029	No		No	1110111101	No		No	0111011001	No		No	0011111101	No	
030	No		No	0111111111	No		No	0010111010	No		No	1100011010	No	

Table D.4: Performance of early circuit variants on SAT instances consisting of 10 variables and 60 clauses

			Fully deterministic circuit				**Probability driven circuit**				**Asynchronous circuit**			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	No		No	1111111011	No		No	1011111101	No		No	0100101000	No	
002	No		No	1111111111	No		No	1011101000	No		No	0000011111	No	
003	No		No	1111111111	No		No	1010111010	No		No	0011010111	No	
004	No		No	1111111111	No		No	0100101001	No		No	1001000000	No	
005	No		No	0111111010	No		No	1011000100	No		No	1110011110	No	
006	No		No	0111111111	No		No	1011111100	No		No	0011001010	No	
007	No		No	1111111111	No		No	0010111101	No		No	0101001100	No	
008	No		No	0101111111	No		No	1111011010	No		No	1010000000	No	
009	No		No	1111111111	No		No	0110100101	No		No	1000111101	No	
010	No		No	1110111111	No		No	1111111001	No		No	0110011111	No	
011	No		No	0111111111	No		No	1010111110	No		No	1101000101	No	
012	No		No	0111111111	No		No	0111110100	No		No	1100101011	No	
013	No		No	1011111110	No		No	1110100111	No		No	1011111010	No	
014	No		No	1111111110	No		No	0000010100	No		No	1011101011	No	
015	No		No	1011100110	No		No	1111101000	No		No	0011001110	No	
016	Yes	10111100000	No	1111111111	No		Yes	1011110000	Yes	8166	No	0110100110	No	
017	No		No	0111111111	No		No	0010010001	No		No	0000100101	No	
018	No		No	1101101011	No		No	1101000101	No		No	0111000100	No	
019	No		No	0111011111	No		No	1111001010	No		No	1101001101	No	
020	No		No	0101111111	No		No	0101010000	No		No	1011100010	No	
021	No		No	0111111111	No		No	0100100101	No		No	0011111100	No	
022	No		No	1111111111	No		No	0100010100	No		No	1110001001	No	
023	No		No	1111111111	No		No	1110001010	No		No	0111101000	No	
024	No		No	1111111111	No		No	0100001110	No		No	1011101110	No	
025	No		No	0111111111	No		No	0100110111	No		No	1101100010	No	
026	No		No	1110110111	No		No	0010000111	No		No	0100001001	No	
027	No		No	1101111111	No		No	0000110101	No		No	1000111000	No	
028	No		No	1111111111	No		No	0001100010	No		No	1100000001	No	
029	No		No	1100111011	No		No	1010011101	No		No	1110100001	No	
030	No		No	0111111010	No		No	1010101110	No		No	0001111100	No	

Table D.5: Performance of early circuit variants on SAT instances consisting of 10 variables and 70 clauses

			Fully deterministic circuit				Probability driven circuit				Asynchronous circuit			
Tag	Satisfiable	MiniSat solution	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations	Solution found	Final truth assignment	Solution satisfies instance	Iterations
001	No		No	1111110111	No		No	1100010001	No		No	0000100110	No	
002	No		No	0010011111	No		No	0001110110	No		No	0100100100	No	
003	No		No	1111111110	No		No	1000100100	No		No	1001011011	No	
004	No		No	1111111111	No		No	1000101011	No		No	1111010110	No	
005	No		No	1111111111	No		No	1101010000	No		No	1001101010	No	
006	No		No	1111111110	No		No	0011010000	No		No	0011100000	No	
007	No		No	1111111111	No		No	1001111010	No		No	1001110111	No	
008	No		No	1110111111	No		No	0010101000	No		No	1100000000	No	
009	No		No	1111111110	No		No	0001100101	No		No	1001010010	No	
010	No		No	1111111111	No		No	1110010000	No		No	1001110100	No	
011	No		No	1111111111	No		No	1011010010	No		No	1100110100	No	
012	No		No	0111111110	No		No	0000110101	No		No	0100011100	No	
013	No		No	1111110010	No		No	0011000001	No		No	1101110101	No	
014	No		No	1111111111	No		No	1011011110	No		No	1010111100	No	
015	No		No	0111011110	No		No	1110011101	No		No	1000000100	No	
016	No		No	1111111111	No		No	1010000111	No		No	0000000110	No	
017	No		No	1111011110	No		No	1101110111	No		No	1110101111	No	
018	No		No	1010111111	No		No	0000000100	No		No	1011111001	No	
019	No		No	1111111101	No		No	1011001001	No		No	0101110010	No	
020	No		No	0111111111	No		No	0011011010	No		No	1011010001	No	
021	No		No	0111111111	No		No	1000110111	No		No	0011101011	No	
022	No		No	1111111111	No		No	1100000011	No		No	1011111000	No	
023	No		No	1111111111	No		No	0011110010	No		No	0010011100	No	
024	No		No	0110011111	No		No	0110111011	No		No	1001010000	No	
025	No		No	1111111111	No		No	0110000111	No		No	1111011101	No	
026	No		No	0111111111	No		No	1001100011	No		No	1111011001	No	
027	No		No	1111111111	No		No	1011110010	No		No	1001000101	No	
028	No		No	1111111111	No		No	1110001110	No		No	0101110000	No	
029	No		No	1111111110	No		No	0110100100	No		No	0010100111	No	
030	No		No	1111111111	No		No	1101010011	No		No	0110101001	No	

Table D.6: Performance of early circuit variants on SAT instances consisting of 10 variables and 80 clauses

D.2 Experiments using fixed toggling probabilities

					Broken $P_b = 1/4$		**Broken** $P_b = 1/3$		**Broken** $P_b = 1/2$		**Fixed** $P_b = 1/3$		**Fixed** $P_b = n/3c$		**MiniSat**
Tag	Variables	Clauses	Satisfiable	Solution found	Solution satisfies instance	Iterations	Solution satisfies instance	Iterations	Solution satisfies instance	Iterations	Solution satisfies instance	Iterations	Solution satisfies instance	Iterations	Pentium IV CPU Cycles
001	100	370	Yes	Yes	No		No		Yes	1327932	Yes	1913	Yes	7692	568296
002	100	370	Yes	No	No		Yes	964694	No		Yes	19680320	No		1311044
003	100	370	Yes	No	No		Yes	1576389	No		Yes	289512	Yes	14878	3598812
004	100	370	Yes	No	No		No		No		Yes	37691805	Yes	20141	13498924
005	100	370	Yes	No	No		No		No		Yes	1119320	Yes	3788	2073964
006	100	370	Yes	No	No		No		No		No		No		3780780
007	100	370	Yes	No	No		No		No		No		No		4623352
008	100	370	Yes	Yes	No		No		Yes	329703	Yes	65920	Yes	16343	1630848
009	100	370	Yes	No	No		No		No		No		No		4527228
010	100	370	Yes	Yes	No		No		Yes	1077506	Yes	168705	Yes	31670	834584
011	100	370	Yes	No	No		No		No		No		Yes	19962154	290940
012	100	370	Yes	No	No		No		No		No		Yes	77679	942340
013	100	370	Yes	No	No		No		No		Yes	160592	Yes	10006	926816
014	100	370	Yes	No	No		No		No		No		No		970888
015	100	370	Yes	Yes	No		No		Yes	16771839	Yes	1034368	Yes	255037	3653660
016	100	370	Yes	No	No		No		No		Yes	157329	Yes	1643442	324800
017	100	370	Yes	No	No		No		No		Yes	4381172	Yes	9511726	3867540
018	100	370	Yes	No	No		No		No		Yes	24938517	No		1108792
019	100	370	Yes	Yes	No		No		Yes	519272	Yes	9566	Yes	14709	286816
020	100	370	Yes	Yes	No		Yes	456772	Yes	972127	Yes	64641	Yes	21798	395696

Table D.7: Performance of SAT circuits using fixed toggling probabilities

D.3 Experiments using derived toggling probabilities

	FPGA Solver									MiniSat
Tag	Factor 0.75	Factor 0.875	Factor 1.00	Factor 1.25	Factor 1.50	Factor 1.75	Factor 2.00	Factor 2.25	Factor 2.50	Pentium IV CPU Cycles
000	57293	3784	15344	56183	2721	50254	7122	57444	159250	2519040
001	449	1590	358	1250	1997	1048	2734	355	1692	568296
002	214161	103317	276531	125309	170288	1229639	256752	1159660	6926	1311044
003	13391	24386	16346	45065	47822	40078	121980	484129	58420	3598812
004	729090	35032	94865	5099	163958	462868	253997	4201344	5472710	13498924
005	53572	2128	26419	1941	2319	10899	26004	50749	240762	2073964
006	25935	11893	101512	45603	135103	176123	17789	1037645	614514	3780780
007	105346	210857	1095696	1046408	1365045	3033203	3759066	476407	6827247	4623352
008	200	185	539	1140	210	131	1575	1463	2756	1630848
009	8041981	492930	2072110	9948698	5246431	71590001	9322171	39950682	71590001	4527228
010	1321	589	1487	2936	3559	11458	740	10008	15211	834584
011	56399	20698	558489	346593	525291	1421882	217043	2401771	128255	290940
012	8919	108921	226072	107769	778239	436159	511869	1094984	6097624	942340
013	1076	9198	16207	8263	1430	54071	21352	39952	52863	926816
014	51914	283161	1294157	178936	2637863	192242	15427150	22883418	41187274	970888
015	295	49779	70633	205002	25465	346	10454	170949	14357	3653660
016	2745	9021	14077	5797	3639	1069	4413	31651	17910	324800
017	63241	176582	27005	1985	2831	76913	44246	172369	6935	3867540
018	39608	9029	17338	53770	4540	35259	22955	48646	87281	1108792
019	2260	364	348	163	453	1233	4937	148	5143	286816
020	1417	1063	3156	612	361	5471	899	2736	4166	395696
021	368560	382525	642981	1122140	2314387	1058126	2135958	4423893	12425030	3449324
022	15692734	506700	699707	2470792	7601270	8887972	37414205	58842974	43396643	4331364
023	69257	33613	161164	86276	25595	53236	71767	136716	501562	971036
024	145086	4732	49156	601	84918	79985	55125	63428	265367	8340416
025	58982	5377	379166	272421	480892	209231	644300	1720943	89456	1459468
026	368	1107	505	1124	632	216	992	2242	1796	1136504
027	12593	5651	657	28566	75702	98726	303377	234056	197144	588580
028	4922	3354	4968	9239	1350	226	4789	1544	5124	2514512
029	38327	27103	7376	65788	5122	656	46347	210562	31757	522896
030	2611	8361	2156	7442	71919	9276	111415	14844	157979	726344
031	6826	1465	8422	21420	18354	11328	37624	2977	4037	3216612
032	19405	2851	5774	15462	12337	32571	31066	60661	95170	477396
033	17855	33613	2094	4992	17828	14173	2343	37304	13971	206388
034	10730	30370	94858	7309	2918	89137	99106	74064	159232	1727576
035	59709	21014	5336	8111	9632	25093	126698	129162	210299	1914168
036	44529	29401	207858	115037	177914	333755	665485	295886	389284	534384
037	18162	103438	20779	104986	9711	616461	851241	1227467	2000699	1929980
038	10920	4759	6082	6138	6919	16012	2943	12108	32178	1480052
039	31927	24116	33055	40990	37799	45366	181116	26305	1067752	1311956
040	6607	34358	57131	1448	12516	6944	88904	178364	191014	2746300
041	106025	829576	104846	328854	822582	2127369	3832861	4583998	21406539	7170496
042	328332	203390	173685	24872	774446	615381	1573122	250541	507910	3414768
043	38516	125753	33689	2301	20333	180131	460824	84837	1037730	504656
044	104616	714868	2252538	1333978	3596895	1083316	6988364	22796895	28158882	1058380
045	7047	288204	120483	145738	38290	129473	532120	132174	1680063	2374568
046	17551	1498	27948	25522	88413	56772	25034	22003	273630	1648524
047	381865	70142	750751	105251	870488	46987	675073	1574406	5628528	7809788
048	26721	21643	35693	50038	148002	37003	1840	391576	13664	852516
049	22460	8537	51521	377	2065	28764	70640	946239	300950	605992
Sum	27123856	5082026	11869068	18595735	28448794	94724033	87069927	172754679	252834687	116760104
Mean	542477	101641	237381	371915	568976	1894481	1741399	3455094	5056694	2335202

Table D.8: Performance of SAT circuits using derived toggling probabilities

D.4 Results of insufficient randomisation

	FPGA Solver							MiniSat
Tag	Factor 1.0	Factor 1.5	Factor 2.0	Factor 2.5	Factor 3.0	Factor 3.5	Factor 4.0	Pentium IV CPU Cycles
000	120681	7792	61072	105641	226620	112568	937064	2519040
001	1627	1904	1544	2229	1080	2071	5410	568296
002	1085935	71590001	134865	7314572	3179527	71590001	71590001	1311044
003	2983	4990	10333	5712	15808	61264	1015166	3598812
004	43783	122515	160646	1299004	1573646	31115549	15957392	13498924
005	103951	9482	38582	27395	129477	66213	1480881	2073964
006	752055	1035191	306911	948976	1467257	2463013	71590001	3780780
007	71590001	5929593	2651023	6279693	71590001	71590001	71590001	4623352
008	7801	2907	10905	1291	1439	5687	40995	1630848
009	71590001	71590001	26659268	71590001	71590001	71590001	71590001	4527228
010	12794	6536	13646	6040	1834	71286	36429	834584
011	5268891	530682	1289726	1250446	228791	39246692	71590001	290940
012	778997	134369	179677	1014619	3564668	692763	71590001	942340
013	21199	58294	22881	17530	4348	159722	501583	926816
014	71590001	8767855	6754938	5213967	13330252	71590001	71590001	970888
015	671890	109522	26341	147203	170214	881244	6846156	3653660
016	386368	42489	23993	16410	89709	116860	1709595	324800
017	123164	455443	295927	5694	43256	10486016	128875	3867540
018	71590001	785661	311989	37167	447083	18949	202040	1108792
019	8160	5928	1131	1346	2098	3237	3398	286816
020	9719	2914	2275	8434	3566	3978	7007	395696
021	71590001	71590001	9103567	20711878	42016397	71590001	71590001	3449324
022	71590001	71590001	71590001	71590001	71590001	71590001	71590001	4331364
023	166491	54709	318798	79474	38630	751539	19911605	971036
024	32368	133573	73268	260204	1573824	3037874	6970030	8340416
025	71590001	2684369	4244526	7256959	11562566	22468021	71590001	1459468
026	2157	2264	1416	1396	1426	1093	5606	1136504
027	5136	5691	10638	6225	22041	441625	2103358	588580
028	1625	2417	1600	1939	14938	13029	3484	2514512
029	47091	58452	20620	48604	109002	2678014	3758851	522896
030	42248	74283	15010	3077	77320	965885	1874664	726344
031	3166	54241	36907	30205	20401	25059	214371	3216612
032	376876	69914	67453	135237	17892	328997	18354942	477396
033	38449	42234	5208	26222	169677	60618	108632	206388
034	429831	10382	73330	108352	234292	83063	4005348	1727576
035	406258	68361	126046	24712	138564	85033	2190912	1914168
036	177625	203149	140474	184074	480375	1305818	17177511	534384
037	634260	283972	643943	168208	2606217	5665751	17419018	1929980
038	227557	54720	32721	12888	86289	5209	1313412	1480052
039	343577	37879	542955	278802	200736	2721412	16295345	1311956
040	454674	497728	22435	7282	88430	326888	5870778	2746300
041	11267797	12895233	21006399	13745768	29252078	71590001	71590001	7170496
042	650915	40458	375208	2114	1386734	18244623	26893727	3414768
043	71590001	2805040	4678375	4917855	1255948	21697221	71590001	504656
044	71590001	71590001	14558764	39471867	35015747	61998453	71590001	1058380
045	71590001	213941	12687531	43650716	4092635	71590001	71590001	2374568
046	174423	41270	3626	165731	37233	131227	8008048	1648524
047	574552	309138	782297	4542396	826357	6392668	71590001	7809788
048	919911	339487	5823	72012	92301	1260471	7626734	852516
049	191977	42704	5415	37241	489205	1419125	3233922	605992

Table D.9: Performance of SAT circuits using derived toggling probabilities with insufficient randomisation engine

D.5 Comparision of different randomisation engines

	Factor 1.0			**Factor** 2.0			**MiniSat**
Tag	Serial	Parallelised	Array	Serial	Parallelised	Array	MiniSat
000	59047	120681	15344	344988	61072	7122	2519040
001	7692	1627	358	4426	1544	2734	568296
002		1085935	276531	35306933	134865	256752	1311044
003	14878	2983	16346	18072	10333	121980	3598812
004	20141	43783	94865	515886	160646	253997	13498924
005	3788	103951	26419	117734	38582	26004	2073964
006		752055	101512	561480	306911	17789	3780780
007			1095696		2651023	3759066	4623352
008	16343	7801	539	12892	10905	1575	1630848
009			2072110		26659268	9322171	4527228
010	31670	12794	1487	2700	13646	740	834584
011	19962154	5268891	558489	1852993	1289726	217043	290940
012	77679	778997	226072	889843	179677	511869	942340
013	10006	21199	16207	58600	22881	21352	926816
014			1294157	31873865	6754938	15427150	970888
015	255037	671890	70633	24488	26341	10454	3653660
016	1643442	386368	14077	2818	23993	4413	324800
017	9511726	123164	27005	793546	295927	44246	3867540
018			17338	632421	311989	22955	1108792
019	14709	8160	348	7461	1131	4937	286816
020	21798	9719	3156	2616	2275	899	395696
021			642981		9103567	2135958	3449324
022			699707			37414205	4331364
023	1434229	166491	161164	199730	318798	71767	971036
024	580518	32368	49156	36823	73268	55125	8340416
025			379166	1324652	4244526	644300	1459468
026	1669	2157	505	2004	1416	992	1136504
027	34213	5136	657	34101	10638	303377	588580
028	7797	1625	4968	7210	1600	4789	2514512
029	36844	47091	7376	6996	20620	46347	522896
030	13487	42248	2156	292258	15010	111415	726344
031	1060909	3166	8422	11321	36907	37624	3216612
032	2544804	376876	5774	107934	67453	31066	477396
033	295686	38449	2094	125554	5208	2343	206388
034	480833	429831	94858	407569	73330	99106	1727576
035	1959931	406258	5336	933983	126046	126698	1914168
036	2447005	177625	207858	3639346	140474	665485	534384
037	5292101	634260	20779	4053724	643943	851241	1929980
038	304272	227557	6082	235673	32721	2943	1480052
039	860153	343577	33055	1549517	542955	181116	1311956
040	1124972	454674	57131	29949	22435	88904	2746300
041		11267797	104846		21006399	3832861	7170496
042	2047548	650915	173685	122728	375208	1573122	3414768
043			33689	448144	4678375	460824	504656
044			2252538		14558764	6988364	1058380
045			120483		12687531	532120	2374568
046	661202	174423	27948	334883	3626	25034	1648524
047		574552	750751	667958	782297	675073	7809788
048	3057787	919911	35693	934544	5823	1840	852516
049	99301	191977	51521	17648	5415	70640	605992

Table D.10: Performance of SAT circuits using different randomisation engines

D.6 Phase transition experiments

n	3.5	3.6	3.7	3.8	3.9	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	Point
5	922	932	916	911	862	855	839	834	806	775	742	756	718	5.618
10	944	930	909	912	865	842	811	779	737	677	645	635	574	4.959
15	962	941	930	889	849	811	803	732	680	641	584	565	460	4.666
20	971	965	944	917	875	818	759	682	672	589	536	480	404	4.563
25	983	970	938	911	876	825	771	670	607	560	447	411	339	4.454
30	985	989	958	920	887	818	753	662	594	540	445	380	312	4.441
35	990	978	959	931	873	819	758	679	536	462	397	343	251	4.351
40	995	991	970	951	898	818	762	658	591	462	373	307	223	4.372
45	999	996	971	945	888	842	750	608	521	433	317	254	183	4.324
50	998	992	981	947	911	850	758	638	500	424	342	221	188	4.310
55	999	994	984	972	903	858	772	624	522	396	311	232	146	4.316
60	1000	994	991	967	907	846	770	619	513	385	274	200	120	4.308
65	1000	997	991	973	921	872	747	627	501	400	248	161	104	4.305
70	1000	997	990	977	915	860	707	611	468	335	231	146	91	4.278
75	1000	998	990	974	922	839	721	586	439	331	196	161	62	4.259
80	1000	998	997	984	943	865	780	589	469	331	206	99	77	4.273
85	1000	1000	999	988	954	895	797	668	467	336	223	134	72	4.287
90	1000	1000	996	986	953	885	755	611	463	292	168	90	54	4.274
95	1000	1000	996	986	938	849	722	553	402	236	148	101	42	4.236
100	999	999	997	978	939	851	705	516	366	222	122	65	32	4.213
105	1000	1000	997	988	934	821	685	510	331	201	96	43	33	4.205
110	1000	1000	994	983	952	859	688	484	368	207	98	48	20	4.199
115	1000	1000	1000	991	967	890	736	543	332	210	121	49	22	4.220
120	1000	1000	1000	995	977	911	782	563	387	235	126	57	16	4.237
125	1000	1000	1000	996	989	934	829	636	399	234	129	67	27	4.258
130	1000	1000	1000	997	982	916	831	632	400	203	114	50	9	4.257
135	1000	1000	999	991	975	878	722	547	306	159	82	29	14	4.217
140	1000	1000	1000	988	947	839	676	421	239	131	45	17	6	4.171
145	1000	1000	998	986	909	775	606	351	158	85	24	13	1	4.141
150	1000	1000	994	969	899	737	531	305	142	64	29	11	2	4.113
155	1000	999	995	971	878	708	459	252	109	36	17	7	3	4.085
160	1000	999	993	941	863	644	396	200	97	30	13	2	0	4.059
165	1000	1000	992	965	807	624	403	170	72	34	6	2	1	4.056
170	1000	998	988	935	816	593	352	181	70	19	4	4	0	4.039
175	1000	999	991	933	824	571	361	153	62	15	3	0	1	4.035
180	1000	1000	993	952	817	584	321	167	68	20	0	1	0	4.032
185	1000	999	991	941	799	577	318	162	74	14	6	1	1	4.029
190	1000	1000	990	955	816	578	333	150	48	16	5	0	1	4.032
195	1000	1000	993	966	817	610	377	161	68	22	6	0	0	4.047
200	1000	1000	997	966	871	635	349	166	63	24	2	2	0	4.047
205	1000	1000	999	975	861	651	403	168	61	15	1	0	0	4.061
210	1000	1000	998	970	896	702	418	193	68	27	9	2	0	4.072
215	1000	1000	998	992	922	768	453	224	75	25	5	4	0	4.088
220	1000	1000	1000	996	929	794	540	252	94	26	7	1	0	4.113
225	1000	1000	998	997	967	838	588	297	96	34	6	1	0	4.130
230	1000	1000	1000	998	973	872	663	327	122	49	8	2	0	4.148
235	1000	1000	1000	1000	987	918	713	391	155	49	14	4	0	4.167
240	1000	1000	1000	1000	992	953	780	501	175	76	17	3	1	4.198
245	1000	1000	1000	1000	997	967	844	526	229	79	18	2	0	4.210
250	1000	1000	1000	1000	1000	991	866	621	276	85	27	6	1	4.234

Table D.11: Fraction of satisfiable random SAT instances regarding ratios of clauses to variables with approximated phase transition points (Part 1)

n	4.8	4.9	5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	6.0	Point
5	678	666	656	595	592	562	556	526	519	446	454	423	419	5.618
10	561	514	489	481	410	361	318	288	281	242	217	220	169	4.959
15	456	396	353	298	272	225	223	190	142	131	114	90	79	4.666
20	388	326	254	230	182	151	122	118	87	84	65	56	45	4.563
25	338	242	200	161	133	84	85	66	55	31	22	16	14	4.454
30	255	170	160	115	86	67	61	36	24	25	18	13	9	4.441
35	195	142	109	70	50	41	31	23	17	10	9	4	6	4.351
40	164	117	106	71	38	39	24	16	11	9	4	1	1	4.372
45	145	89	57	46	31	21	12	4	6	3	3	2	1	4.324
50	130	83	50	45	23	15	12	10	2	2	0	0	0	4.310
55	96	60	37	21	17	8	4	4	2	3	0	0	1	4.316
60	90	45	31	17	10	2	4	2	1	0	0	0	0	4.308
65	69	35	17	20	7	4	4	0	1	0	0	0	0	4.305
70	49	27	21	11	4	0	0	0	1	0	0	0	0	4.278
75	35	18	16	5	2	3	2	0	0	0	0	0	0	4.259
80	37	30	9	5	3	2	0	0	0	0	0	0	0	4.273
85	25	16	7	6	2	1	0	0	0	0	0	0	0	4.287
90	24	8	7	0	3	0	0	1	0	1	0	0	0	4.274
95	24	6	5	3	0	0	0	0	0	0	0	0	0	4.236
100	14	7	4	4	2	0	0	0	0	0	0	0	0	4.213
105	12	5	3	2	2	0	0	0	0	0	0	0	0	4.205
110	12	3	0	0	0	1	0	0	0	0	0	0	0	4.199
115	10	6	0	0	0	0	0	0	0	0	0	0	0	4.220
120	13	4	2	1	0	0	0	0	0	0	0	0	0	4.237
125	10	4	1	0	0	0	0	0	0	0	0	0	0	4.258
130	6	4	1	0	0	0	0	0	0	0	0	0	0	4.257
135	1	1	0	0	1	0	0	0	0	0	0	0	0	4.217
140	1	0	1	0	0	0	0	0	0	0	0	0	0	4.171
145	3	1	0	0	0	0	0	0	0	0	0	0	0	4.141
150	1	0	0	0	0	0	0	0	0	0	0	0	0	4.113
155	0	0	0	0	0	0	0	0	0	0	0	0	0	4.085
160	0	0	0	0	0	0	0	0	0	0	0	0	0	4.059
165	0	0	0	0	0	0	0	0	0	0	0	0	0	4.056
170	0	0	0	0	0	0	0	0	0	0	0	0	0	4.039
175	0	0	0	0	0	0	0	0	0	0	0	0	0	4.035
180	0	0	0	0	0	0	0	0	0	0	0	0	0	4.032
185	0	0	0	0	0	0	0	0	0	0	0	0	0	4.029
190	0	0	0	0	0	0	0	0	0	0	0	0	0	4.032
195	0	0	0	0	0	0	0	0	0	0	0	0	0	4.047
200	0	0	0	0	0	0	0	0	0	0	0	0	0	4.047
205	0	0	0	0	0	0	0	0	0	0	0	0	0	4.061
210	0	0	0	0	0	0	0	0	0	0	0	0	0	4.072
215	0	0	0	0	0	0	0	0	0	0	0	0	0	4.088
220	0	0	0	0	0	0	0	0	0	0	0	0	0	4.113
225	0	0	0	0	0	0	0	0	0	0	0	0	0	4.130
230	0	0	0	0	0	0	0	0	0	0	0	0	0	4.148
235	0	0	0	0	0	0	0	0	0	0	0	0	0	4.167
240	0	0	0	0	0	0	0	0	0	0	0	0	0	4.198
245	1	0	0	0	0	0	0	0	0	0	0	0	0	4.210
250	0	0	0	0	0	0	0	0	0	0	0	0	0	4.234

Table D.12: Fraction of satisfiable random SAT instances regarding ratios of clauses to variables with approximated phase transition points (Part 2)

D.7 Runtime statistics

	Factor 0.750							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	185	121777	20764	21489	21531			2519040
001	89	5853	941	881	882			568296
002	5203	829325	145593	161153	161469	24	116	1311044
003	222	301059	39043	41354	41435			3598812
004	5542	1429957	239369	253528	254025	40	108	13498924
005	82	173418	23777	24123	24170			2073964
006	253	468697	82499	82526	82687			3780780
007	8173	1257313	305438	213581	213999	32	112	4623352
008	99	5953	1192	1032	1034			1630848
009								4527228
010	119	34675	4214	4487	4496			834584
011	4798	905584	167660	164564	164887	26	150	290940
012	1575	562843	130560	116211	116439	108	92	942340
013	190	81141	13056	13956	13983	94	151	926816
014								970888
015	204	294214	45005	44564	44651			3653660
016	102	46762	8292	7988	8004			324800
017	141	395910	51199	57395	57507			3867540
018	116	90992	20293	18108	18143			1108792
019	102	5412	1130	951	953			286816
020	87	9323	1950	1804	1808			395696
021	950	1676916	437746	416819	417635	28	114	3449324
022								4331364
023	283	223600	44982	45037	45125			971036
024	106	602385	64408	74333	74479			8340416
025	460	648075	132619	127729	127979	33	112	1459468
026	94	4802	970	732	733			1136504
027	119	204985	35341	33253	33318			588580
028	157	14396	2857	2559	2564			2514512
029	161	143613	29981	27330	27383			522896
030	132	81417	15524	14716	14745			726344
031	155	57586	8662	9240	9258			3216612
032	212	106034	18400	18784	18821			477396
033	219	68167	9988	10221	10241			206388
034	335	191640	36488	35194	35263			1727576
035	134	87955	14381	13932	13959			1914168
036	254	413873	90713	85238	85405	153	80	534384
037	379	522149	104676	100881	101078			1929980
038	192	47707	6306	6138	6150			1480052
039	2762	425937	67066	68021	68154	64	96	1311956
040	193	134666	26850	24903	24952			2746300
041	3357	1304092	286259	305247	305845	24	116	7170496
042	901	733711	173227	168352	168682	124	66	3414768
043	474	535984	97453	111070	111288	54	124	504656
044								1058380
045	2513	783214	154832	144038	144320	59	117	2374568
046	153	143317	24971	24535	24583			1648524
047	17145	1028410	675567	267811	268335	19	120	7809788
048	154	158873	26357	26231	26283			852516
049	262	156316	31339	28588	28644			605992
Sum	59538	17520028	3919937					116760104
Mean	1294	380870	85216					2335202

Table D.13: Runtime statistics of hardware SAT solver engine (Probability multiplier 0.750)

	Factor 0.875							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	112	131212	22045	22104	22147			2519040
001	46	5018	889	795	796			568296
002	1162	889437	181847	170035	170368	26	115	1311044
003	519	298004	44924	56211	56321	88	84	3598812
004	3667	1055974	278790	259631	260140	28	114	13498924
005	114	86633	23441	21631	21673	80	88	2073964
006	609	493945	80753	82692	82854	20	152	3780780
007	1454	3537574	641443	794964	796521	40	108	4623352
008	67	5188	1115	918	919			1630848
009								4527228
010	138	20788	3653	3717	3724			834584
011	1976	988636	163108	126289	126536	22	117	290940
012	1396	824758	200320	179397	179748	82	87	942340
013	488	63405	12535	11807	11830			926816
014								970888
015	313	284264	42035	44586	44674			3653660
016	128	43426	7503	7219	7233			324800
017	128	322346	56581	53171	53275			3867540
018	131	143542	18423	19710	19748			1108792
019	53	4515	1054	929	931			286816
020	85	9060	1649	1532	1535			395696
021	2960	2306120	638071	611748	612946	8	130	3449324
022								4331364
023	283	244416	54003	50620	50720	52	117	971036
024	413	469210	75086	75551	75699	42	111	8340416
025	2473	1280030	185689	174516	174858	120	68	1459468
026	50	4140	792	688	690			1136504
027	245	196559	39586	38247	38322			588580
028	135	17276	2661	2534	2539			2514512
029	425	126196	27652	24109	24156			522896
030	113	79549	15129	15519	15549			726344
031	106	40849	8580	8401	8417			3216612
032	142	124259	17178	18395	18431			477396
033	145	86416	10501	10681	10702			206388
034	496	190299	32814	34337	34405			1727576
035	68	77421	16569	16238	16269			1914168
036	2351	445344	91540	90248	90425	10	141	534384
037	1618	604919	117145	101953	102153	90	90	1929980
038	101	50582	7796	8210	8226			1480052
039	254	392587	69208	66609	66740			1311956
040	255	310219	36489	36463	36535			2746300
041	8390	2115869	409969	421428	422253	6	138	7170496
042	2047	1419471	182878	187427	187794	69	96	3414768
043	446	508600	77308	80894	81053			504656
044								1058380
045	1004	681377	129855	129071	129324	36	140	2374568
046	238	155121	25226	25132	25181	92	82	1648524
047	8486	3306536	607194	402963	403752	5	126	7809788
048	121	123470	26871	24402	24450			852516
049	529	211570	35839	36527	36598			605992
Sum	46480	24776130	4723737					116760104
Mean	1010	538612	102690					2335202

Table D.14: Runtime statistics of hardware SAT solver engine (Probability multiplier 0.875)

	Factor 1.000							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	180	116897	24134	23826	23873			2519040
001	50	4212	832	735	736			568296
002	1527	1494971	198440	164127	164449	156	50	1311044
003	240	213368	43572	44377	44464			3598812
004	2231	1588525	273383	340637	341305	39	115	13498924
005	164	148479	25421	23760	23806			2073964
006	107	543223	89260	88651	88825			3780780
007	33739	1444742	687937	395771	396547	9	126	4623352
008	72	4409	981	836	838			1630848
009								4527228
010	73	19914	3740	3314	3321			834584
011	9875	1488552	298767	312884	313497	18	119	290940
012	1053	872891	149587	153081	153381	64	96	942340
013	69	63766	12303	11569	11592			926816
014								970888
015	392	195124	39590	38034	38109	91	106	3653660
016	133	50655	7515	7516	7531			324800
017	105	308623	60419	60494	60612			3867540
018	280	97623	22631	20301	20341			1108792
019	69	4210	914	744	746			286816
020	53	8718	1607	1415	1418			395696
021	36600	1462818	588800	438158	439017	19	126	3449324
022								4331364
023	914	276786	47470	45455	45544	105	132	971036
024	441	539979	89571	102435	102636	84	86	8340416
025	721	792455	137472	127386	127636	104	76	1459468
026	38	4256	795	644	645			1136504
027	422	300887	40235	38416	38491	156	50	588580
028	65	11715	2625	2192	2197			2514512
029	122	136486	28745	25963	26014			522896
030	155	107765	16687	16975	17008			726344
031	81	57094	8901	7926	7941			3216612
032	144	124265	17420	18154	18189			477396
033	104	83631	9417	10566	10587			206388
034	110	253136	39171	45725	45815	158	49	1727576
035	114	66877	13197	12143	12166			1914168
036	507	428915	85623	75840	75989	24	216	534384
037	739	890267	131069	136050	136317	102	77	1929980
038	152	43179	7355	6582	6595			1480052
039	264	391940	80495	76878	77028	3	183	1311956
040	102	208942	29438	29950	30008			2746300
041	12526	2615921	446953	401110	401895	118	69	7170496
042	167	1259526	219516	226606	227050	94	81	3414768
043	1786	587007	128901	110502	110719	96	80	504656
044								1058380
045	4940	699482	169218	159207	159519	18	148	2374568
046	113	185898	21236	22412	22456			1648524
047	7022	1178212	311566	275741	276281	21	119	7809788
048	187	146156	26236	26484	26536			852516
049	184	165123	33975	33536	33601	182	61	605992
Sum	119132	21687620	4673121					116760104
Mean	2590	471470	101590					2335202

Table D.15: Runtime statistics of hardware SAT solver engine (Probability multiplier 1.000)

	Factor 1.250							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	117	119753	23603	22515	22559	68	174	2519040
001	52	3808	725	644	646			568296
002	672	1244858	215854	240515	240986	6	156	1311044
003	235	387486	55350	56422	56533			3598812
004	35556	1467677	515232	423286	424115	19	120	13498924
005	263	172329	32020	31090	31151			2073964
006	158	553346	98908	100167	100364			3780780
007	48499	5310638	667529	600803	601980	32	123	4623352
008	61	5180	957	881	883			1630848
009								4527228
010	34	25430	3784	3542	3549			834584
011	2524	1208636	250477	233092	233549	130	63	290940
012	350	777625	165267	154780	155083	111	132	942340
013	87	96606	12537	13192	13217			926816
014								970888
015	139	255366	52866	48577	48672			3653660
016	108	43351	8062	7123	7137			324800
017	113	480345	66515	71277	71417			3867540
018	212	104539	23327	22139	22182			1108792
019	31	4415	847	698	700			286816
020	21	8260	1479	1345	1348			395696
021	13435	3887920	772745	749916	751385	10	128	3449324
022								4331364
023	436	724308	50632	62476	62598	156	86	971036
024	286	407531	77592	75694	75842			8340416
025	3697	1396597	234397	208001	208409	30	116	1459468
026	31	3842	658	585	586			1136504
027	320	225543	48633	45948	46038	182	37	588580
028	69	12671	2684	2616	2621			2514512
029	362	217921	37219	36744	36816			522896
030	111	82689	17322	17479	17514			726344
031	152	48717	8933	9382	9401			3216612
032	330	98819	19775	18341	18377			477396
033	106	62154	11815	11759	11782			206388
034	307	227348	42784	42410	42493			1727576
035	83	72927	16908	14779	14808			1914168
036	1089	454829	111193	104391	104595	34	111	534384
037	287	1150960	135138	139356	139629			1929980
038	142	60761	7952	8524	8541			1480052
039	282	503129	78476	72512	72654	97	99	1311956
040	113	162164	34027	31086	31147			2746300
041	35314	3186874	1303721	929527	931347	3	128	7170496
042	2586	882014	244910	217823	218249	48	104	3414768
043	104	698084	124613	133996	134258	42	210	504656
044								1058380
045	1649	765240	200687	196538	196923	63	100	2374568
046	64	141683	25361	24313	24361			1648524
047	7104	956209	329408	281943	282496	4	133	7809788
048	168	161057	25242	26485	26537			852516
049	128	268392	39549	40532	40611			605992
Sum	157987	29130031	6197713					116760104
Mean	3435	633262	134733					2335202

Table D.16: Runtime statistics of hardware SAT solver engine (Probability multiplier 1.250)

	Factor 1.500							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	126	162600	28105	27683	27738			2519040
001	49	3476	622	563	564			568296
002	354	1957530	336549	328011	328653	142	57	1311044
003	175	320606	60403	58759	58874			3598812
004	11211	2279537	402533	443193	444061	32	112	13498924
005	214	193788	33829	30869	30930	38	187	2073964
006	2742	473027	152295	155057	155361	40	108	3780780
007	8261	3883699	893227	898225	899984	48	104	4623352
008	38	5231	924	883	884			1630848
009								4527228
010	58	22243	4171	4320	4329			834584
011	2374	1582931	345898	319484	320110	12	122	290940
012	1215	1305263	232138	258037	258543	17	124	942340
013	112	81548	14850	14934	14963			926816
014								970888
015	523	324464	57765	56055	56164			3653660
016	66	56239	9320	9294	9313			324800
017	74	399849	73089	72109	72250			3867540
018	183	144155	27725	28310	28365			1108792
019	35	4198	838	699	700			286816
020	36	8999	1451	1479	1482			395696
021	61116	3147246	719413	423358	424188	16	120	3449324
022								4331364
023	407	699420	73626	90113	90290			971036
024	89	588837	101486	99151	99345			8340416
025	840	1305394	320523	298521	299106	39	112	1459468
026	38	2946	583	472	473			1136504
027	145	347579	54118	51908	52010			588580
028	44	14520	2274	2259	2263			2514512
029	175	201604	42792	41643	41724			522896
030	121	141557	21448	22442	22486			726344
031	84	60817	10432	10870	10891			3216612
032	131	136564	21573	22577	22622			477396
033	78	71930	12455	11241	11263			206388
034	62	304599	57616	53576	53681	25	179	1727576
035	41	105178	18192	18192	18227			1914168
036	232	802472	129649	129678	129932			534384
037	118	921636	166243	171354	171690			1929980
038	84	48428	9589	9054	9072			1480052
039	95	559646	115048	112955	113176	14	147	1311956
040	473	228161	36264	34402	34469			2746300
041	57804	2819358	717771	562504	563606	20	119	7170496
042	801	1587899	304884	261639	262151	94	87	3414768
043	857	932039	183704	164928	165251			504656
044								1058380
045	483	1017519	288382	239562	240031	40	108	2374568
046	188	195759	34204	34336	34403			1648524
047	7427	2715269	892323	493641	494608	40	108	7809788
048	200	174708	34510	33801	33867			852516
049	886	457406	52349	56985	57097			605992
Sum	160865	32797874	7097183					116760104
Mean	3497	712997	154287					2335202

Table D.17: Runtime statistics of hardware SAT solver engine (Probability multiplier 1.500)

	Factor 1.750							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	202	289539	39904	41119	41200			2519040
001	30	2643	611	521	522			568296
002	28284	1771959	518315	447356	448232	10	128	1311044
003	255	407213	80038	78244	78397			3598812
004	5505	1810272	738371	561028	562127	76	90	13498924
005	199	232615	47513	44609	44696			2073964
006	119	1438163	195354	193213	193591			3780780
007	1542	5553701	1567975	1596822	1599950	16	120	4623352
008	41	5209	848	790	791			1630848
009								4527228
010	49	44601	5617	5697	5708			834584
011	19946	2505971	1950885	485110	486061	24	116	290940
012	5455	2731290	348246	373003	373733	180	40	942340
013	87	100628	16785	17256	17290			926816
014								970888
015	137	376158	73863	67618	67750	198	29	3653660
016	52	79613	12835	13426	13452			324800
017	271	707424	126776	127166	127415	24	116	3867540
018	298	291731	39664	40634	40713			1108792
019	27	3452	795	665	666			286816
020	30	9339	1345	1375	1377			395696
021	62936	4706681	1596668	1393473	1396202	16	120	3449324
022								4331364
023	959	431691	95425	96415	96603	63	105	971036
024	141	699669	136675	120671	120908	88	156	8340416
025	2098	1886272	357141	336407	337066	74	91	1459468
026	57	3029	639	524	525			1136504
027	545	359867	71654	66099	66229			588580
028	71	14378	3025	2607	2612			2514512
029	95	366791	63034	61147	61267			522896
030	129	265415	29936	34350	34417			726344
031	63	77935	13956	13588	13615			3216612
032	175	191906	32450	33876	33942			477396
033	69	79020	15822	15788	15819			206388
034	314	545579	66795	68086	68219			1727576
035	201	140147	28271	26469	26521			1914168
036	7196	642011	120713	127396	127645	11	140	534384
037	2159	1310331	262907	255227	255727	108	84	1929980
038	71	62128	10259	9928	9947			1480052
039	3025	987337	207973	222187	222623	59	102	1311956
040	104	198168	42382	39894	39972			2746300
041	16498	3787458	1308286	1121443	1123640	1	128	7170496
042	3650	2853986	457579	474577	475507	49	116	3414768
043	937	1289321	237994	198735	199124	112	72	504656
044								1058380
045	1830	1284805	359321	318785	319410	21	134	2374568
046	158	309883	48502	49671	49769			1648524
047	16649	2826268	1338933	794001	795556	27	119	7809788
048	192	297860	43396	41285	41366			852516
049	185	341524	64494	56057	56166			605992
Sum	183036	44320981	12779971					116760104
Mean	3979	963500	277825					2335202

Table D.18: Runtime statistics of hardware SAT solver engine (Probability multiplier 1.750)

	Factor 2.000							MiniSat
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	183	235614	52495	51206	51307			2519040
001	26	4166	767	718	719			568296
002	417	4621328	757409	683262	684600	66	95	1311044
003	314	763693	145153	145454	145739			3598812
004	15764	4567792	1027014	1070708	1072806	28	114	13498924
005	44	363357	71726	65357	65485			2073964
006	9904	1447572	342989	293133	293707	2	148	3780780
007	337648	12985324	3876251	4275098	4283472	7	126	4623352
008	40	6683	1019	998	1000			1630848
009								4527228
010	91	40671	6269	6285	6298			834584
011	16377	4734165	739541	777348	778870	24	116	290940
012	31008	2285455	593978	474072	475000	33	112	942340
013	198	128729	24874	24452	24500			926816
014								970888
015	158	1003979	115788	114777	115002			3653660
016	50	77913	13508	14007	14035			324800
017	1830	1097993	173452	169744	170077			3867540
018	378	287886	56216	51570	51671			1108792
019	30	5888	869	825	827			286816
020	46	12596	1795	1819	1823			395696
021	71820	11798371	2125594	2942682	2948446	40	108	3449324
022								4331364
023	734	562281	136851	124036	124278	12	122	971036
024	44	1570144	194527	203435	203834			8340416
025	35816	2863242	538715	504386	505374	48	108	1459468
026	28	3136	686	572	573			1136504
027	383	453430	86762	79304	79459			588580
028	105	19498	3667	3613	3620			2514512
029	88	486948	78047	75371	75519			522896
030	161	276183	41186	41974	42056			726344
031	213	124138	18569	19473	19511			3216612
032	655	300521	48267	52763	52866	168	44	477396
033	68	89825	20956	18839	18876			206388
034	629	707243	116426	131370	131628	176	40	1727576
035	61	236702	35688	40229	40308			1914168
036	624	1434173	207742	221324	221757	23	183	534384
037	2775	2330584	373103	378173	378914	80	90	1929980
038	81	80168	16878	15963	15995			1480052
039	563	1237806	284323	253138	253633	141	64	1311956
040	82	352324	64944	59038	59154			2746300
041	24016	6247951	2063372	589328	590482	28	114	7170496
042	6634	3593828	867829	831787	833417	14	132	3414768
043	11699	2324491	399481	360260	360966	76	90	504656
044								1058380
045	10192	2901855	567430	434055	434905	72	92	2374568
046	664	410798	64882	63422	63546			1648524
047	15069	6425412	1115677	673116	674434	24	116	7809788
048	249	347505	57289	60812	60931			852516
049	206	500767	91746	89843	90019			605992
Sum	598165	82350128	17621750					116760104
Mean	13004	1790220	383082					2335202

Table D.19: Runtime statistics of hardware SAT solver engine (Probability multiplier 2.000)

	Factor 2.250							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	728	462013	97021	100538	100735	88	84	2519040
001	24	3867	810	722	723			568296
002	3111	6126290	1421616	1313284	1315857	72	92	1311044
003	202	1032596	215665	203485	203883			3598812
004	3243	8697340	1414539	2524755	2529701	41	108	13498924
005	702	1077274	134117	147171	147460			2073964
006	4365	2490780	513157	475591	476522	111	92	3780780
007	216784	10250416	2147887	2041730	2045730	6	125	4623352
008	31	5130	1233	1106	1108			1630848
009								4527228
010	98	53953	9488	9416	9435			834584
011	45785	6902373	1846640	1636415	1639621	66	95	290940
012	15584	5160931	1323881	1275937	1278436	83	88	942340
013	225	204254	30953	29898	29956			926816
014								970888
015	499	892396	155120	161873	162190			3653660
016	54	143279	22914	23027	23072			324800
017	216	1633373	303247	290556	291125			3867540
018	1067	716286	90165	100099	100296			1108792
019	26	6086	897	918	920			286816
020	30	13527	1998	1921	1925			395696
021	72876	20168577	5004335	4006013	4013860	2	130	3449324
022								4331364
023	2278	1600548	217153	201734	202129			971036
024	597	1238701	307064	295749	296329	114	71	8340416
025	3319	3899152	931431	840001	841647	27	125	1459468
026	29	3427	736	708	709			1136504
027	605	962449	141466	136517	136784			588580
028	40	23734	4443	4477	4486			2514512
029	774	963040	128818	134448	134711			522896
030	53	282965	57807	58185	58299			726344
031	256	144841	25438	25257	25307			3216612
032	105	509982	62682	67003	67135			477396
033	77	154095	31255	28793	28850			206388
034	504	934702	170316	165001	165324	56	197	1727576
035	326	362852	54481	56702	56813			1914168
036	5843	1687193	322946	232254	232709	48	104	534384
037	4043	3423239	623139	673116	674435			1929980
038	89	175151	17538	18884	18921			1480052
039	1094	3029616	419602	497319	498294	118	69	1311956
040	508	686794	85131	103266	103468	10	124	2746300
041	116085	10822132	3127957	2948189	2953964	25	116	7170496
042	26370	2575866	1366787	505136	506125	24	117	3414768
043	1183	3991294	699889	686780	688125	20	144	504656
044								1058380
045	6260	6580168	1084083	1377152	1379849	6	125	2374568
046	1289	693786	111293	112588	112808			1648524
047	2065764	14330874	12430658	1171293	1173588	6	125	7809788
048	452	510454	85616	87093	87263			852516
049	869	750144	153651	161956	162273	38	109	605992
Sum	2604462	126377940	37397061					116760104
Mean	56619	2747347	812980					2335202

Table D.20: Runtime statistics of hardware SAT solver engine (Probability multiplier 2.250)

	Factor 2.500							**MiniSat**
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Period start	Period length	Pentium IV CPU Cycles
000	65	1010531	127269	142608	142887			2519040
001	35	6832	1059	1070	1072			568296
002	58372	13271216	2998527	3073338	3079358	3	128	1311044
003	3443	2347592	374352	370352	371077			3598812
004	939	25529290	3265927	4683531	4692705	40	123	13498924
005	620	745053	180342	165422	165746			2073964
006	6867	4594137	961902	830325	831951	8	149	3780780
007	1205826	53954612	26925341	25722366	25772753	6	126	4623352
008	30	6979	1471	1327	1329			1630848
009								4527228
010	67	100376	13930	14666	14695			834584
011	11352	9823508	2787136	3171959	3178173	16	120	290940
012	94774	7537282	1415516	1408893	1411653	8	124	942340
013	1111	239706	46026	43520	43605			926816
014								970888
015	126	1325621	298546	273196	273731	86	160	3653660
016	143	262021	36115	38462	38538			324800
017	592	2467455	431372	438822	439681			3867540
018	250	1310152	145752	159084	159396			1108792
019	26	6816	1265	1189	1192			286816
020	28	15752	2767	2634	2640			395696
021	548469	32512152	10275968	9653731	9672641	16	120	3449324
022								4331364
023	1421	2262351	370549	387565	388324			971036
024	3796	3899179	651923	674955	676277	163	48	8340416
025	97	7484129	1575917	1718454	1721820	22	117	1459468
026	26	6591	819	781	782			1136504
027	2388	1937485	288778	292080	292652	4	236	588580
028	57	46583	6095	6049	6061			2514512
029	2357	1323111	230093	220595	221027			522896
030	225	361877	86408	72813	72956			726344
031	160	250954	35487	36282	36353			3216612
032	42	566739	99531	102036	102236			477396
033	106	297509	42745	41730	41812			206388
034	775	1603790	299090	276595	277136			1727576
035	311	528638	91340	90042	90218			1914168
036	2119	4585410	656096	688678	690027	126	65	534384
037	76068	2713198	2009362	340626	341293	22	117	1929980
038	210	149761	27397	24450	24498			1480052
039	968	5312958	805760	747171	748635	5	137	1311956
040	69	1766888	164494	172201	172539			2746300
041	97253	25279349	5752853	7351960	7366361	1	132	7170496
042	126697	12071392	1657182	1532371	1535373	6	128	3414768
043	4916	6869204	1458867	1463674	1466542	9	124	504656
044								1058380
045	100637	11530928	2049190	1886767	1890463	50	105	2374568
046	1571	1247880	207137	205375	205777			1648524
047	498014	13491655	4498869	3726642	3733942	24	116	7809788
048	459	674893	129448	140862	141138			852516
049	77	1305270	246235	231610	232064			605992
Sum	2853954	264634805	73732248					116760104
Mean	62042	5752931	1602875					2335202

Table D.21: Runtime statistics of hardware SAT solver engine (Probability multiplier 2.500)

	WalkSAT (flip counts)				
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation
000	53	1351	308	230	230
001	34	514	130	76	76
002	47	3924	604	532	533
003	51	1853	389	326	327
004	53	5902	831	821	822
005	48	2715	334	298	299
006	38	2069	430	361	361
007	41	5795	810	803	804
008	36	1190	150	119	119
009	48	10248	1470	1465	1468
010	38	1409	235	192	193
011	41	2681	564	458	459
012	43	3675	500	443	444
013	31	2394	323	263	263
014	75	5994	1166	969	971
015	41	2185	431	352	353
016	39	2893	344	342	343
017	53	1611	391	300	300
018	55	1889	315	253	254
019	28	572	130	82	82
020	29	669	146	100	100
021	48	3741	730	677	678
022	74	11574	1781	1761	1764
023	39	1848	342	266	267
024	71	3038	505	429	430
025	60	4012	650	645	646
026	27	733	142	100	100
027	53	2905	507	464	465
028	30	782	192	131	132
029	57	1846	381	298	299
030	43	1356	293	232	232
031	37	1023	238	184	184
032	37	1615	323	242	243
033	32	1450	274	195	195
034	38	2400	443	364	365
035	40	2252	318	290	291
036	32	2793	507	465	466
037	39	2266	464	380	381
038	35	1871	267	251	252
039	56	2083	425	346	347
040	55	1980	374	278	278
041	64	2519	655	486	487
042	50	2588	598	505	506
043	78	2699	521	420	420
044	43	5638	994	839	841
045	56	4696	800	781	782
046	42	2225	375	309	309
047	100	5960	996	963	965
048	52	1582	366	292	293
049	45	1864	403	347	348
Sum	2355	142872	24870		
Mean	47	2857	497		

Table D.22: Runtime statistics of WalkSAT solver engine (flip counts)

	WalkSAT (cycle counts)					MiniSat
Tag	Minimum	Maximum	Average	Standard deviation	Sample deviation	Pentium IV CPU Cycles
000	106676	3073664	567357	427382	428220	2519040
001	58244	875524	238492	136830	137098	568296
002	86260	6852716	1065362	922039	923845	1311044
003	92028	3196996	696830	568500	569614	3598812
004	103736	10302092	1489766	1441433	1444257	13498924
005	89144	4578644	602977	517292	518305	2073964
006	70548	3739552	772736	645254	646518	3780780
007	77392	9926096	1452258	1444226	1447055	4623352
008	68716	2065456	277385	210808	211221	1630848
009	92408	22542804	2678697	2822745	2828275	4527228
010	73780	2392796	434383	344768	345443	834584
011	83028	5890440	1020016	832767	834398	290940
012	85872	6305592	897067	772522	774035	942340
013	61504	3974656	571774	449770	450651	926816
014	149876	9767768	2047770	1671740	1675015	970888
015	73384	3693372	775746	621303	622520	3653660
016	74324	5020216	620482	597212	598382	324800
017	100172	2784772	705054	526819	527851	3867540
018	104472	3707568	566145	452651	453538	1108792
019	53184	991656	237057	142231	142509	286816
020	55364	1155320	271886	184140	184500	395696
021	117600	6770776	1313742	1197745	1200092	3449324
022	135380	20767852	3095510	3036393	3042341	4331364
023	70016	3378896	621241	478325	479262	971036
024	130588	5334148	926732	786028	787568	8340416
025	116536	7204444	1182068	1156373	1158638	1459468
026	50600	1329288	269811	186394	186759	1136504
027	97336	5507124	907561	824632	826248	588580
028	56096	1379928	348048	229196	229645	2514512
029	107888	3561828	680362	526380	527412	522896
030	80980	2707624	532076	415393	416206	726344
031	70320	1803944	434070	325702	326340	3216612
032	79460	2750376	571330	414817	415629	477396
033	66700	2512200	487856	334623	335278	206388
034	70264	3962204	786320	622462	623681	1727576
035	72932	3622452	581084	513288	514293	1914168
036	66452	4875160	904824	817819	819421	534384
037	73840	3970092	822434	654591	655874	1929980
038	66260	3170024	481293	437929	438787	1480052
039	105768	3685088	766964	602304	603484	1311956
040	103004	3648244	672589	489340	490299	2746300
041	117788	4236292	1178246	860700	862386	7170496
042	96436	5276216	1086507	925320	927133	3414768
043	137524	4668076	933915	735543	736984	504656
044	84568	9848636	1776433	1478196	1481092	1058380
045	109636	7824140	1412944	1361212	1363878	2374568
046	84080	3857048	686077	550798	551877	1648524
047	180320	10503240	1775237	1685919	1689222	7809788
048	98736	2743340	648706	508720	509717	852516
049	81356	3244732	717222	610302	611497	605992
Sum	4488576	256981112	44590441			116760104
Mean	89772	5139622	891809			2335202

Table D.23: Runtime statistics of WalkSAT solver engine (cycle counts)

D.8 Dynamic probability calculation using simulated annealing

	FPGA Solver											**MiniSat**
Tag	Boost factor 0.25, 5K	Boost factor 0.25, 10K	Boost factor 0.50, 5K	Boost factor 0.50, 10K	Boost factor 0.75, 5K	Boost factor 0.75, 10K	Boost factor 1.00, 5K	Boost factor 1.00, 10K	Boost factor 1.25, 5K	Boost factor 1.25, 10K	Minimum without boost	Pentium IV CPU Cycles
000	13884	41302	7901	5563	53976	52602	26388	13497	10369	33017	2721	2519040
001	2452	3452	1004	644	747	269	1103	794	178	257	355	568296
002	157800	280636	253979	201521	55417	154605	142025	14997	94658	158018	6926	1311044
003	80596	42607	52964	120539	3235	66932	87287	77766	25695	24765	13391	3598812
004	128162	237489	356061	335465	42521	119337	667462	716255	81967	1169927	5099	13498924
005	5473	19891	7269	38701	19172	15719	9439	5099	43353	159727	1941	2073964
006	66404	31648	126621	26881	74995	85422	34312	18425	2814	31961	11893	3780780
007	1343449	244726	163283	81692	185156	770351	194710	376181	300881	997106	105346	4623352
008	398	1515	467	1389	1427	214	154	250	1953	1041	131	1630848
009	492298	2327282	1944130	10082231	2235500	5329016	5936387	1418988	1620023	6570217	492930	4527228
010	4707	2664	371	3659	4518	2844	5825	1940	2410	1462	589	834584
011	67255	139531	271385	129633	251433	416071	116338	105120	233798	457697	20698	290940
012	30837	393306	283042	12761	184838	51561	43939	133750	96273	24577	8919	942340
013	9292	26526	22840	15490	192	255	19612	8146	26109	11057	1076	926816
014	103935	4034693	2402853	2630786	2402853	1989128	2610305	6395388	2363140	123739	51914	970888
015	4684	68627	11687	5841	24270	7806	164628	5777	77383	71696	295	3653660
016	3294	6826	11050	6052	8585	4722	1712	20254	11119	2577	1069	324800
017	23976	85505	40032	28221	48781	37662	10099	143611	2916	42240	1985	3867540
018	20898	14656	5849	5579	7701	26783	14652	88793	3331	10010	4540	1108792
019	800	1684	417	973	193	377	638	192	860	624	148	286816
020	1523	4601	4932	1553	1371	1124	1018	5803	1547	591	361	395696
021	382722	1144314	46970	317527	120816	3123596	280105	48938	306077	187566	368560	3449324
022	3694445	4564238	2584745	555919	179057	7838318	1035261	8820963	6723812	2337274	506700	4331364
023	57095	98579	76664	22987	49854	42112	11059	10816	19908	122522	25595	971036
024	54085	34066	12783	33885	2976	87058	143156	46130	203848	14427	601	8340416

Table D.24: Performance of SAT circuits using simulated annealing approach (Part 1)

	FPGA Solver											MiniSat
Tag	Boost factor 0.25, 5K	Boost factor 0.25, 10K	Boost factor 0.50, 5K	Boost factor 0.50, 10K	Boost factor 0.75, 5K	Boost factor 0.75, 10K	Boost factor 1.00, 5K	Boost factor 1.00, 10K	Boost factor 1.25, 5K	Boost factor 1.25, 10K	Minimum without boost	Pentium IV CPU Cycles
025	312686	167844	133041	49965	91299	24816	9287	423496	103760	144830	5377	1459468
026	231	231	1173	201	342	3552	597	502	311	2492	216	1136504
027	83249	50392	49570	22269	93180	76501	66662	45732	9035	257014	657	588580
028	1328	585	1034	1660	1115	1276	4213	803	5460	3027	226	2514512
029	51774	35847	65951	21969	74252	7786	1068	58163	50337	14855	656	522896
030	20184	17383	1450	19185	26426	17445	38298	38947	13207	13893	2156	726344
031	10054	3162	782	2965	6272	19506	7820	3316	9767	5840	1465	3216612
032	1379	1221	34989	8387	38695	17077	1616	1602	3482	20971	2851	477396
033	7889	16114	15499	11852	51690	7664	536	26903	3049	1788	2094	206388
034	44765	13044	97319	31621	100866	371	109321	27510	40573	52992	2918	1727576
035	67891	3546	4416	1907	15224	32155	4219	30192	50046	9681	5336	1914168
036	61203	22095	11915	42936	88773	33326	224504	1658	14410	1950	29401	534384
037	473256	17832	55149	48627	6217	48627	50762	180397	49028	27727	9711	1929980
038	35425	3930	16314	14700	9878	1954	6661	4227	4226	28045	2943	1480052
039	129122	6504	65645	191782	17915	255793	219834	73078	60220	56167	24116	1311956
040	36036	21356	38434	38651	91411	1233	55455	11107	30316	44004	1448	2746300
041	134386	537993	151882	7036	671865	37791	156058	531553	1380915	149980	104846	7170496
042	85854	33345	551699	275706	29008	9974	87526	172055	7956	178461	24872	3414768
043	111534	11330	44651	39360	49001	126024	27257	34001	211378	183713	2301	504656
044	3089760	652886	4038827	3437294	4207628	1323369	8586555	250072	461612	713423	104616	1058380
045	266989	131008	534871	88069	53026	437978	76638	23311	55976	60275	7047	2374568
046	33519	12335	12240	13203	12486	84208	8768	8015	19179	79540	1498	1648524
047	671681	2014	1389628	379407	369450	431337	1092538	505115	85446	158078	46987	7809788
048	12648	16405	10131	45222	29678	72511	17563	14271	7627	51730	1840	852516
049	68409	10141	61203	20689	62370	34944	36207	12491	45288	39862	377	605992
Sum	12561716	15638907	16077112	19480155	12157651	23331102	22447577	20956390	14977026	14854433	2019738	116760104
Mean	251234	312778	321542	389603	243153	466622	448952	419128	299541	297089	40395	2335202

Table D.25: Performance of SAT circuits using simulated annealing approach (Part 2)

D.9 Performance results of locally probability driven circuits

	FPGA Solver										MiniSat
Tag	Factor 0.75	Factor 0.875	Factor 1.00	Factor 1.25	Factor 1.50	Factor 1.75	Factor 2.00	Factor 2.25	Factor 2.50	Locally	Pentium IV CPU Cycles
000	57293	3784	15344	56183	2721	50254	7122	57444	159250	10854	2519040
001	449	1590	358	1250	1997	1048	2734	355	1692	1671	568296
002	214161	103317	276531	125309	170288	1229639	256752	1159660	6926	45898	1311044
003	13391	24386	16346	45065	47822	40078	121980	484129	58420	782	3598812
004	729090	35032	94865	5099	163958	462868	253997	4201344	5472710	3852	13498924
005	53572	2128	26419	1941	2319	10899	26004	50749	240762	4888	2073964
006	25935	11893	101512	45603	135103	176123	17789	1037645	614514	162706	3780780
007	105346	210857	1095696	1046408	1365045	3033203	3759066	476407	6827247	172418	4623352
008	200	185	539	1140	210	131	1575	1463	2756	2034	1630848
009	8041981	492930	2072110	9948698	5246431	71590001	9322171	39950682	71590001	3515693	4527228
010	1321	589	1487	2936	3559	11458	740	10008	15211	2331	834584
011	56399	20698	558489	346593	525291	1421882	217043	2401771	128255	64165	290940
012	8919	108921	226072	107769	778239	436159	511869	1094984	6097624	8612	942340
013	1076	9198	16207	8263	1430	54071	21352	39952	52863	1835	926816
014	51914	283161	1294157	178936	2637863	192242	15427150	22883418	41187274	1108494	970888
015	295	49779	70633	205002	25465	346	10454	170949	14357	24543	3653660
016	2745	9021	14077	5797	3639	1069	4413	31651	17910	1117	324800
017	63241	176582	27005	1985	2831	76913	44246	172369	6935	15411	3867540
018	39608	9029	17338	53770	4540	35259	22955	48646	87281	3605	1108792
019	2260	364	348	163	453	1233	4937	148	5143	244	286816
020	1417	1063	3156	612	361	5471	899	2736	4166	638	395696
021	368560	382525	642981	1122140	2314387	1058126	2135958	4423893	12425030	145849	3449324
022	15692734	506700	699707	2470792	7601270	8887972	37414205	58842974	43396643	4485085	4331364
023	69257	33613	161164	86276	25595	53236	71767	136716	501562	2660	971036
024	145086	4732	49156	601	84918	79985	55125	63428	265367	33071	8340416

Table D.26: Performance of SAT circuits using locally probability driven approach (Part 1)

	FPGA Solver										**MiniSat**
Tag	Factor 0.75	Factor 0.875	Factor 1.00	Factor 1.25	Factor 1.50	Factor 1.75	Factor 2.00	Factor 2.25	Factor 2.50	Locally	Pentium IV CPU Cycles
025	58982	5377	379166	272421	480892	209231	644300	1720943	89456	90590	1459468
026	368	1107	505	1124	632	216	992	2242	1796	424	1136504
027	12593	5651	657	28566	75702	98726	303377	234056	197144	13841	588580
028	4922	3354	4968	9239	1350	226	4789	1544	5124	4322	2514512
029	38327	27103	7376	65788	5122	656	46347	210562	31757	54586	522896
030	2611	8361	2156	7442	71919	9276	111415	14844	157979	7997	726344
031	6826	1465	8422	21420	18354	11328	37624	2977	4037	6313	3216612
032	19405	2851	5774	15462	12337	32571	31066	60661	95170	16573	477396
033	17855	33613	2094	4992	17828	14173	2343	37304	13971	2835	206388
034	10730	30370	94858	7309	2918	89137	99106	74064	159232	30648	1727576
035	59709	21014	5336	8111	9632	25093	126698	129162	210299	5499	1914168
036	44529	29401	207858	115037	177914	333755	665485	295886	389284	5916	534384
037	18162	103438	20779	104986	9711	616461	851241	1227467	2000699	16302	1929980
038	10920	4759	6082	6138	6919	16012	2943	12108	32178	1919	1480052
039	31927	24116	33055	40990	37799	45366	181116	26305	1067752	66063	1311956
040	6607	34358	57131	1448	12516	6944	88904	178364	191014	10044	2746300
041	106025	829576	104846	328854	822582	2127369	3832861	4583998	21406539	217381	7170496
042	328332	203390	173685	24872	774446	615381	1573122	250541	507910	1512	3414768
043	38516	125753	33689	2301	20333	180131	460824	84837	1037730	111110	504656
044	104616	714868	2252538	1333978	3596895	1083316	6988364	22796895	28158882	217889	1058380
045	7047	288204	120483	145738	38290	129473	532120	132174	1680063	157830	2374568
046	17551	1498	27948	25522	88413	56772	25034	22003	273630	762	1648524
047	381865	70142	750751	105251	870488	46987	675073	1574406	5628528	434012	7809788
048	26721	21643	35693	50038	148002	37003	1840	391576	13664	1278	852516
049	22460	8537	51521	377	2065	28764	70640	946239	300950	39444	605992
Sum	27123856	5082026	11869068	18595735	28448794	94724033	87069927	172754679	252834687	11333546	116760104
Mean	542477	101641	237381	371915	568976	1894481	1741399	3455094	5056694	226671	2335202

Table D.27: Performance of SAT circuits using locally probability driven approach (Part 2)