Intelligate:
An Algorithm for Learning Boolean Functions for Dynamic Power Reduction

by

Roni Wiener

A thesis submitted in partial fulfilment of the

requirements for the degree of Msc

University Of Haifa

Faculty of Social Sciences

Department of Computer Sciences

November, 2007

Intelligate:
An Algorithm for Learning Boolean Functions for Dynamic Power Reduction

by

Roni Wiener

Supervised by:

Prof. Martin Charles Golumbic

Prof. Moshe Y. Vardi

(Department of Computer Science - Rice University, Houston Texas)

A thesis submitted in partial fulfilment of the

requirements for the degree of Msc

University Of Haifa

Faculty of Social Sciences

Department of Computer Sciences

November, 2007

Approved by: _____ Date: _____

Supervisor

Approved by: _____ Date: _____

Supervisor

Approved by: _____ Date: _____

Chairperson of Msc. committee

Acknowledgements

I would like to thank my supervisors Prof. Moshe Y. Vardi and Prof. Martin Charles Golumbic. Their expert advice, insight and guidance proved invaluable in the writing of this thesis.

A special acknowledgment is offered to Gila Kamhi (Intel Corp.) for her guidance, and contribution of time and energy on her behalf to ensure the successful completion of this work. I would also like to thank to Dr. Ranan Fraer (Intel Corp.), whose technical expertise was of great help.

Contents

Intelligate:

An Algorithm for Learning Boolean Functions for Dynamic Power Reduction

by

Roni Wiener

Abstract

Power dissipation is a major concern in modern VLSI circuit design. Clock gating is a technique which allows substantial dynamic power reduction on the clock network by halting it in parts of the design which are not active. In this work we introduce an algorithm to detect general invariants so as to reveal logical behavior that will allow efficient clock gating in the design. Our approach focuses upon dynamic simulation data as the main source for detection of opportunities for power reduction. We extract and compile gating conditions by using the circuit's existing logic as building blocks by analyzing power simulation benchmarks trace data. Experimental results show the ability to cope with the size of industrial RTL circuit designs and the ability to learn clock gating Boolean functions from dynamic simulation traces.

List of Tables

List of Figures

Chapter 1

Introduction

A CMOS circuit is primarily composed of nets (wires), logic gates and memory units (registers), all of which form its logic behavior. Each net has its electric capacitance, and charges and discharges its capacitance to change its binary value. The amount of power (Watts) that the circuit consumes is a major concern in industrial chip design. The dynamic power consumption of a CMOS circuit depends directly upon the switching frequency of the capacitance (a.k.a. switching activity). Clock nets typically have very high capacitance and switch more frequently than any other net in the design. Without effective power management, this can lead to a large increase in the clock and registers dynamic power consumption of a typical digital logic block being consumed by the clock and register cells combined.

Clock gating is a technique that blocks clock net activity when part of the design is inactive. If applied effectively, clock gating can greatly reduce dynamic power consumption of a typical digital logic block. Clock gating is typically implemented through a combination of two techniques: manual insertion of hand-crafted block-level gated clocks into the RTL code, and automated insertion of clock gating during RTL synthesis. A framework to reveal clock gating opportunities which are not observable

by conventional methods, is beneficial in terms of power reduction and automation of the gating of the logic. Our goal is the detection of general "dynamic invariants" (i.e., global functional relation between signal in the design) based upon trace patterns as a source of design knowledge. The emphasis of our work is upon finding "dynamic invariants" that reveal logical behavior in order to allow efficient clock gating in the design (i.e., idleness condition).

Our approach focus upon dynamic simulation data as the main source for detection of opportunities for power reduction. Validation, power and performance simulation benchmark trace data, if analyzed in-depth, can provide a rich source for characterization of design at hand. While validation tests attempt to capture the corner cases of design behavior, power tests focus upon the common behavior scenarios or, to be more precise, scenarios where there is significant activity in the design. Therefore, conclusions derived from power benchmarks represent more general behavior that holds in almost all the common and most importantly legal execution scenarios.

Power benchmark data mining can be viewed simply as filtering through large amounts of raw data for useful information that gives hints for design improvements from a power consumption standpoint. Since the benchmark trace data is generated by dynamic simulation techniques, it is not exhaustive and does not represent all the execution traces of the design at hand. Nevertheless, power and validation test suites for hardware design are expected to be nearly complete in coverage of all the interesting behaviors of the hardware at hand.

This work introduce a novel method for extracting clock gating functions using the design's existing logic as building blocks. The algorithm learns a Boolean function representing the register's *hold condition* from positive and negative examples derived

from the simulation trace. Positive examples are cycles where the clock can be gated, while negative examples are cycles wherein the clock must be active to allow a value change. The gating function predicts gating opportunities one cycle before the clock should be gated, and it can be composed from any signal in the design derived by data mining and machine learning techniques. Experimental results indicate that the method produces beneficial power saving Boolean functions, even on industrial size designs. Indeed, the global correlations found yield dynamic power reduction up to 70% on the clock network via clock gating.

The thesis is organized as follows: Chapter 2 provides a theoretical background in the fields the method is based upon, where the method's flow and algorithms are elaborated upon in Chapter 3. The experimental method and results are described in Chapter 4, where in Chapter 5 the research conclusions and the further work needed are addressed.

Chapter 2

Background

Chapter 1 explained the need for power saving schemes in logic circuit design. In this chapter, we will address the theoretical aspects of our work, and relate our work with prior art. We address the problem of power reduction of industrial size designs power consumption and cover four main concepts: 1) power consumption 2) clock gating 3) data mining and 4) machine learning. The concept of dynamic power dissipation will be described in Section 2.1. A popular and widely used technique to reduce circuits dynamic power dissipation is clock gating. The concept and recent developments will be explained in Section 2.2. *Learning from examples* is an area studied in *machine learning* dealing with case based reasoning, the ability to derive significant and true concepts from real examples. In our work we try to learn a Boolean function from positive and negative examples. We elaborate upon this topic and introduce related work in Section 2.3. Section 2.4 will clarify the concept of *data mining* and its usage. We will then elaborate upon the more narrow focus of data mining through *associations rules* via the *apriori* algorithm which is the thrust of our work.

2.1 Dynamic power dissipation

A CMOS circuit is primarily composed of nets (electrical wires) ,logic gates and memory units that describe its logic behavior. Each net has its electric capacitance, and charges and discharges it in order to change its binary value, back and forth. Switching power is consumed by the circuits nets charge and discharge operations during its work, for every net in the design the average dynamic power consumption is computed according to Equation 2.1 [22].

$$DynPwr = \frac{1}{2} \cdot C \cdot f \cdot Vdd^2 \cdot AF \tag{2.1}$$

where:

C = the switching capacitance of the net.

f = the design operation frequency.

Vdd = the supply voltage of the design.

AF = the *activity factor* is the average number of switching per cycle, it can be calculated by counting the number of transitions from 0 to 1 a signal made during execution divided by the number of the execution cycles. The activity factor ranges from 0 for nets with constant values, up to 1 for clock nets which change their value in every phase.

The direct relationship between the nets capacitance C, activity factor AF and its dynamic power consumption is easily seen from Equation 2.1. These parameters are net specific, hence can be locally tuned and vary, unlike the operation frequency f and supply voltage Vdd. The later parameters are common for the entire design, and, consequently, subject to global design constraints, complicating their variation.

The overall dynamic power dissipation of a design is the sum of all its nets' dynamic power dissipation as depicted in Equation 2.1; thus introducing Equation 2.2 for total

dynamic power:

$$TotDynPwr = \sum_{i=1}^{N} \frac{1}{2} \cdot Ci \cdot f \cdot Vdd^2 \cdot AFi \tag{2.2}$$

where Ci is the capacitance of the ith net, AFi is the ith net activity factor and N is the total number of nets in the design. Since clock nets have high activity factor and relatively large capacitance, their dynamic power consumption is significantly high. The clock network consumes significant portion of the design's dynamic power (30-40%) [4] therefore reducing activity on these nets will be beneficial in substantial reduction of dynamic power.

The main focus of our work is to reduce the activity factor of the clock nets. One widely adopted technique for clock net activity reduction, is clock gating. This method selectively, halts the clock activity for inactive design blocks, reduces their clock activity and consequently their dynamic power to zero. Section 2.2 describes the clock gating concepts and techniques.

2.2 Clock gating for dynamic power reduction

Clock gating is a technique that blocks clock net activity when parts of the design are inactive. If applied effectively, clock gating can reduce large portions of dynamic power consumption of a typical digital logic block.

Registers are memory units which can hold or change their values. A typical register consists of clock, data and output nets as illustrated in Figure 2.1. A register's functionality can be depicted in Function 1, where *Out'* is denoting the registers previous value.

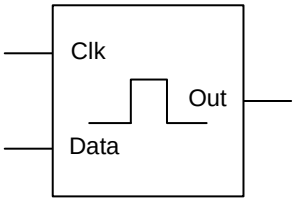


Figure 2.1: Register scheme

Function 1 Register next stage function.

1: **if** clock is high **then**
2: Out = Data.
3: **else**
4: Out = Out'
5: **end if**

The register holds its value when the clock is low. When the clock is high, the registers value is written according to the current data value. Clock gating can be inserted in cases such as when the data value is equal to the previous value held in the register.

A logic condition which describes the design states wherein the register holds its value, is called the *hold condition*. The hold condition logic is expressed by a combination of signals in the design and it is usually implemented to gate the register's clock. An example for a hold condition can be $h = \bar{a}$ where a is a signal, which enables writing to a memory array that the register is part of.

Enable signals are inserted to control the clock activity for groups of registers with common hold conditions. A counter is a good example for clock gating insertion. The counter reset its values to zero whenever *RESET* is asserted high, increment its value by one on every clock cycle when *INC* is asserted high and *RESET* is asserted low, and holds its old value when *INC* and *RESET* are asserted low. A traditional implementation of three bit counter without clock gating is illustrated in Figure 2.2. Note that the clock is routed directly to each of the registers in the design, which

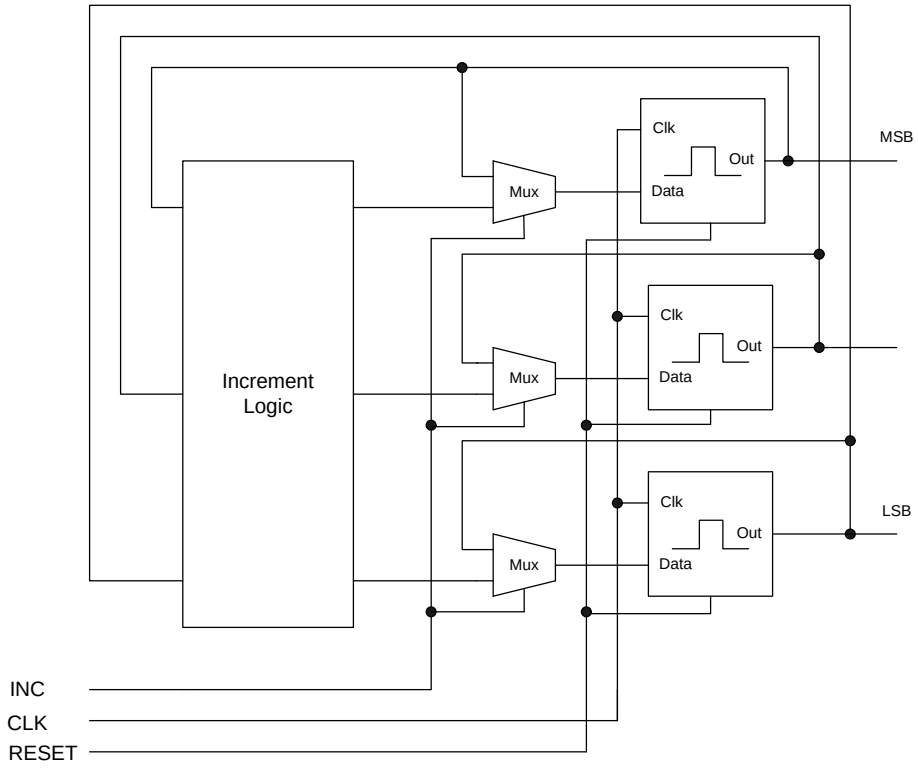


Figure 2.2: 3 bit counter

means that they will be clocked continuously (AF=1), with the old data circulated into the registers through the multiplexers driving the registers data when the *INC* input is low.

In Figure 2.3, the same circuit is implemented with clock gating. The circuit is similar to the traditional implementation except that a logic AND gate is inserted into the clock network to gate the clock. This causes the registers to be clocked only when the *INC* input is high. When the *INC* input is low, the registers are not clocked and thus they retain the old value just as in the original implementation.

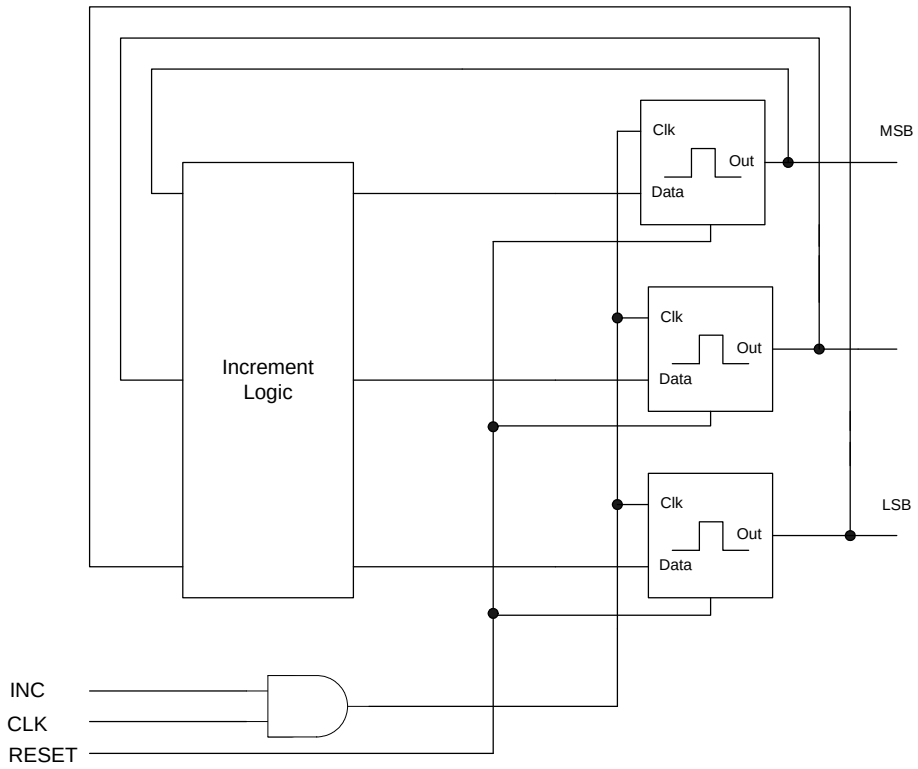


Figure 2.3: 3 bit counter with gated clock

Recalling Equation 2.1, the dynamic power consumption of a clock net has a direct correlation to its activity factor. Therefore the activity difference is a good metric to evaluate the dynamic power reduction achieved by the gating.

A waveform example of 50% dynamic power reduction on the clock network is illustrated in Figure 2.4. In the example, we assume that the counter is incremented every second clock cycle.

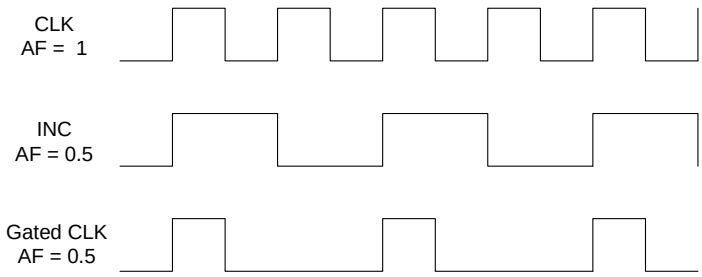


Figure 2.4: Gated clock waveform

Clock gating is typically implemented through a combination of two techniques: manual insertion of hand-crafted block-level gated clocks into the RTL code, and automated insertion of clock gating during RTL synthesis. A framework to reveal clock gating opportunities which are not observable by conventional methods, is beneficial in terms of power reduction and automation of the logic gating.

2.2.1 Previous work

Since power dissipation is one of the major concerns in industrial designs, in the past decade there has been significant research in low power design techniques and methodologies. Due to the large clock network power, several automatic methods have been applied to reveal clock gating opportunities. Clock gating solutions have been implemented via both static and dynamic methods. While static methods extract the hold condition relying on explicit functional representation of the design, dynamic methods relax this requirement and rely upon partial design representation based

upon the incomplete logic behavior illustrated in traces.

2.2.1.1 Static methods

A simple yet effective static method to compute the hold function of individual registers is proposed in [12]. The control signals driving the register's logic are identified and used to approximate the hold condition. A *Finite state machine* (FSM) is a logic representation describing possible transition functions from state to state in a design. Optimization of FSM representation was used in [21] where idle conditions were extracted and optimized from self loops identified on the FSM. This approach is limited to small designs, due to the large number of possible design states in industrial-size designs. Symbolic representation replaced the FSM approach in [23], where the hold condition is extracted from the symbolic representation of the register's next stage functions. The exact symbolic hold condition can be too complex to implement in terms of area and dynamic power consumption, and therefore not suitable for clock gating. in [32] karnaugh maps are used to extract simpler approximations that are still safe for clock gating. The symbolic approach of [23] and [32] does not scale up to large designs as observed by [4]. The detection of *observability don't care* (ODC) described in [30] is used to find opportunities to gate driving logic to buses when their value is not observable (their value is not used). The method extracts a gating function with respect to the multiplexer's selectors only. Usually the implementation of the gating function is too large and complicated so a representative signal from function's fanin is utilized instead of taking into consideration a global view to better approximate the gating condition.

2.2.1.2 Dynamic methods

Extracting clock gating condition by dynamic methods requires implicit or partial knowledge on the design at hand. A dynamic method for detection of redundant

clock switching is introduced in [29]. This method comparers the register's clock, data and output transitions over a given number of cycles. Difference between transitions can reveal redundant clock switching. The method can only reveal the existence of gating opportunities and approximate their power dissipation, but the gating function itself is not extracted. A global method to find clock gating signals from existing logic is introduced in [4]. Register's outputs are paired with each logic node in the design, and logic nodes suitable as gating conditions are searched. In order to resolve the high complexity of pairs combinations, logic nodes are pruned by their gating logic correctness, timing and physical constraints. On the remaining pairs maximum cover techniques are used to group the maximal number of registers to be gated by minimal number of gates. The method generates a *disjunctive normal form* (DNF) expression representing the hold condition, but it is limited to finding only one literal in each clause. Therefore if a gating function of the form $A \wedge B$ exists, but each of its literals is logically pruned by itself, the gating function will not be revealed.

A dynamic method to reveal power saving invariants from simulation traces is introduced in [5]. This method extracts an ODC condition for parts of the design and tries to learn a Boolean function approximating it. First the trace data is divided into two groups of examples (positive and negative, which correlates to *un observable* and *observable*), then a Boolean function that exhibits a False value for all negative examples and a True value for the positive ones is learned. The learning phase is accomplished by the *Extension Matrix Approach* (EMA). The EMA is a covering technique, which represents knowledge in the form of disjunctive Boolean formula by performing a heuristic search through a space of logical expressions, until it finds a decision rule that covers the positive examples while leaving out the negative examples. Since the rule can be too complex, the learning method is applied only on a small subset of signals, usually the control signals in the fanin of the logic cone being

examined. Since the function inferred by the EMA can still be too big and complex for the purpose of dynamic power saving, a signal approximating it is searched in the fanin cone logic. This method is restricted to the control signals in the fanin logic only, global relations between signals in other parts of the design can not be revealed. Since the inferred function covering the positive examples produced by the EMA is too big to save dynamic power, further learning must be made to approximate it. This learning is restricted to only one signal in the fanin logic as well. Hence if there is no such signal or the approximation is a combination of signals, the inferred function can not be applied.

2.3 Machine learning for learning Boolean functions

Machine learning is an automatic task for extracting underlying statistics of a large source of data. By analyzing the source of data, classifiers and decision making schemes can be inferred and applied on future data never seen before.

Supervised learning is a machine learning method applied for a set of training examples with known classification. A mathematical function is inferred from the examples and their outcome, creating a mapping from the input space to one of the possible output targets. Usually the training examples do not cover the entire input space (otherwise the mapping is given by the examples), therefore the given examples are divided for a set of examples used to train the classifier (*training set*) and a *test set*, which is used to test the inferred classifier probabilistic error. The goal is to find a classifier that minimizes the classification error on the test set. *Overfitng* occurs when the function expressed by the learned classifier is too close to the training data points, yielding potentially bad classification results for a new set of data from the same distribution; i.e., the classifier has bad *generalization*.

The learning task can be defined as the problem of guessing a function $h(x)$ (*hypothesis*) based upon the labeled samples in the training set. It can be assumed that the target function is some element of a set of functions C, and that the hypothesis h is an element of a set H of hypotheses (*hypothesis space*), which includes the set C of target functions.

In general h will not be identical to the target function f, but it is desired that for most $x's$ $h(x) = f(x)$, hence h should be approximately correct. The predictive

error of h, ϵ_h, is defined as the probability that a randomly drawn sample will be misclassified. We say that an inferred hypothesis h is *probably approximately correct* (PAC) [19] if the probability that it is approximately correct is greater then $1 - \delta$, where δ is the *confidence parameter*, with an error value less then ϵ. In general a learning algorithm is PAC-learning functions from C in terms of H iff for every function $f \in C$ it outputs a hypothesis $h \in H$ such that with probability at least $(1 - \delta)$, with error $\epsilon_h \leq \epsilon$.

For some classes of functions an exponential learning time is required, a *properly* PAC-learnable class is a class C for which there exists an algorithm that polynomially (in the number of samples needed, sample dimension, ϵ and δ) PAC-learns functions from C in terms of c. For the case of learning Boolean functions, it is known that the classes of terms (conjunctions of literals) and k-DNF (where each term is composed of k literals or less) are properly learnable, where k-term DNF (an expression composed of k terms or less) and arbitrary DNF expressions are classes of functions that are not [6] [24]. The PAC learning model correlates the number of training samples (m), the training error (ϵ_h) and its probability (δ). The relation between m, ϵ and δ is depicted in Equation 2.3, where $|H|$ denotes the number of possible hypotheses in H.

$$m \geq \frac{\ln(|H|) + \ln(\frac{1}{\delta})}{\epsilon} \tag{2.3}$$

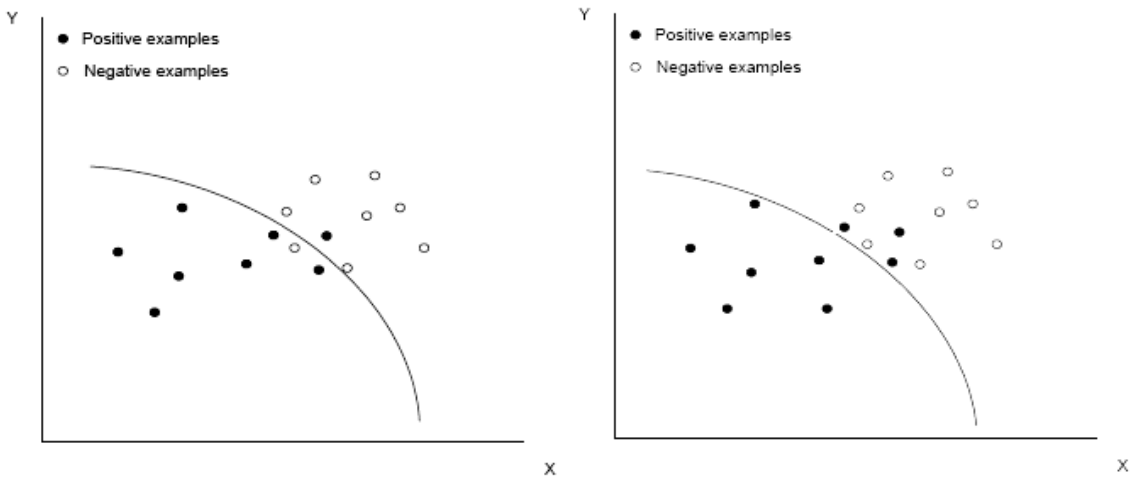
Another important relation is the correlation between the samples dimension and the number of required samples. Samples dimension is the number of features each training sample holds (for the case of Boolean functions, the samples dimension is the number of literals in each training example). For example let $|H|$ be a set of minterms. Then, $|H| = 3^n$, and $m \geq \frac{\ln(3^n)+\ln(\frac{1}{\delta})}{\epsilon} \geq \frac{1.1n+\ln(\frac{1}{\delta})}{\epsilon}$, hence the lower bound of m increases polynomially with n (where n denotes the samples dimension).

It is important to observe the difference between the task of learning two-sided and one-sided error functions. In two-sided learning misclassification is acceptable for both positive and negative examples. Its definition is:
Given a training set $S \subseteq X$ x Y and a hypothesis space H where X is the input space and $Y = \{0, 1\}$ a set of positive and negative labels find hypothesis $h \in H$ such that h minimize the generalization error.

In one-sided error learning misclassification is allowed only for a fraction of the positive examples, means the classifier is never wrong about the negative examples. one-sided error learning is defined as:
Given a training set $S \subseteq X$ x Y and a hypothesis space H where X is the input space and $Y = \{0, 1\}$ a set of positive and negative labels find hypothesis $h \in H$ such that h minimize the positive generalization error and has zero negative error.

The difference between one and two sided error is illustrated in Figure 2.5. The non linear curve that separates the two groups of data points is the classifier (hypothesis) inferred from the positive and negative examples. Each new example is classified according to its position. Data points left to the line are classified as positive and the ones right to the line are classified as negative. 2.5(a) is an example for two-sided error learning, and 2.5(b) is one-sided.

When dealing with a very large or infinite sample space, methods to quantify the generalization quality of the inferred hypothesis are needed. Evaluating the training errors done on the training set is usually not sufficient to quantify the generalization quality of the inferred model, hence in order to predict the generalization error some unseen sample data is used to test the inferred hypothesis (*cross validation*). There



(a) Learning with two-sided errors (b) Learning with one-sided errors

Figure 2.5: Learning with errors.

are several methods to estimate the generalization quality of a classifier by cross validation, where the *holdout* method is the simplest one. In the holdout method the data set is separated into two sets, the training set and the test set. After the learning procedure is done upon the training set, the inferred hypothesis is asked to predict the output values for the data in the test set. The errors it makes are accumulated to give the mean absolute test set error, which is used to evaluate the inferred model. The evaluation may depend heavily on which data points end up in the training set and which end up in the test set, and thus the evaluation may be significantly different depending on how the division is made.

K-fold cross validation is one way to improve over the holdout method. The data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set k-1 times.

Leave-one-out cross validation is K-fold cross validation taken to its logical extreme, with K equal to N, the number of data points in the set. That means that N separate times, the hypothesis if inferred upon all the data samples except for one point and a prediction is made for that point. As before the average error is computed and used to evaluate the model.

The inferred *model complexity* is composed from *bias* and *variance*, where there is a *bias-variance tradeoff* in choosing a model. The model's bias is defined as $\bar{\hat{f}}(x) - f(x)$, where $\bar{\hat{f}}(x) = \lim_{m\to\infty} \frac{1}{m} \sum_{i=1}^{m} \hat{f}_{s_i}(x)$ is the expected predicate value of $f(x)$, where the expectation is taken over all possible training samples S. The model's variance is defined as $E[(\hat{f}(x) - \bar{\hat{f}}(x))^2]$, where the expectation is taken with respect to all training samples S. The variance captures random variations in the inferred model and in the training samples.

The training error tends to decrease whenever we increase the model's complexity, that is, whenever we fit the data harder. However with too much fitting, the model adapts itself too closely to the training data, and will not generalize well (i.e., have large test error). In that case the predictions of h will have large variance. In contrast, if the model is not complex enough, it will under fit and may have large bias, again resulting in poor generalization. More generally, as the model complexity of our procedure is increased, the variance tends to increase and the bias tends to decreases. The opposite behavior occurs as the model complexity is decreased. Typically we would like to choose a model complexity to trade bias off with variance in such a way as to minimize the test set error. One method to do so is *structural risk minimization* (SRM), where models of increasing complexities are tested. Another method is the *Minimum Description Length* (MDL) principle. MDL is a principle of data compression and statistical estimation from information theory, which states that the best

probability model for given data is that which requires the least code length in bits for the encoding of the model itself and the given data observed through it. In the MDL framework, the model description length is an indicator of model complexity, while the data description length indicates goodness of fit to the data. The MDL principle stipulates that the model that minimizes the sum total of the description lengths should be the best model. Both SRM and MDL methods are theory based that must use a priori known assumptions about the hypothesis model in order to generalize beyond the observed data.

For the purpose of clock gating, learning two-sided error Boolean function and thereby capturing the hold condition, is unacceptable. If we allow two-sided errors on our training data (the simulation trace) it is assured that the inferred hold condition will be wrong at some execution paths; i.e, a simulation cycle labeled as negative example will be classified as positive example, therefore the clock will be gated preventing the register from changing its value when it should. One-sided error hold condition is acceptable since the positive error is only resulting in missed gating opportunities. Therefore one-sided error is not optimized in terms of power saving, but according to the trace data, on every cycle that a new value should be stored in the gated register, the clock will not be halted.

2.3.1 Previous work

The PAC learning model is presented in [24], where three classes of Boolean functions are shown to be properly learnable: 1. CNF (*conjunctive normal form*) expression with bounded number of literals in each clause (k-CNF), and therefore also DNF expression with bounded number of literals in each clause (k-DNF), by negation of all literals and swapping the positive and negative example, 2. Monotone DNF expres-

sion, where monotone means no negations of literals are present, and 3. Arbitrary expressions in which each variable occurs only once. The learning procedure in each case output an expression which closely approximates the expression to be learnt. The approximated expression never says *yes* when it should not, but may say *no* on a small fraction of the positive examples.

The size of the inferred function is not bounded by the PAC learning model. For the purpose of dynamic power saving it is important for the gating functions to be small in size, otherwise their logic implementation will consume more dynamic power than the power saved on the clock they gate. Quine-McCluskey algorithm [15] is a method for minimizing Boolean function by finding its *prime implicants*. A minterm m is a prime implicant of a function F if $m \Rightarrow F$ and $m' \not\Rightarrow F$ for any m' obtained by deleting one literal from m. A minterm is defined as a product term in which each of the variables appears once (either complemented, or uncomplemented). Thus, a minterm is a logical expression of variables consisting of only the logical conjunction operator and the complement operator. Unfortunately finding prime implicant is a NP-hard problem, the run time of the Quine-McCluskey algorithm is exponential with the input size and the upper bound for the number of implicants is $\frac{3^n}{n}$, therefore it is not scalable for the amount of literals in industrial designs.

Learning Boolean functions via the Fourier transform is presented in [34], as in the PAC model, this method learns a functions with probabilistic error values. The learning procedure is based upon finding the Fourier coefficients which best approximates the target function, where coefficients with small values (below some threshold value) can be ignored, yielding a smaller function (*Low Degree Algorithm*). The *Sparse Learning Algorithm* learns a sparse target function, i.e a function with small number of non-zero coefficients.

Learning functions with one-sided error is studied in [6], where the family of learnable functions with one side error is introduced. [18] shows that monotone Boolean conjunctions or disjunctions cannot be learned reliably and probably usefully from a polynomial number of examples. A learner that seeks a hypothesis function that agrees with all of the negative examples, and a maximum number of positive examples is studied in [8] and lower mistake bounds are defined for different classes of functions. The size of the inferred functions is not addressed in these works. Rather, the main concern is the bounds for the number of examples and the classification error.

Industrial size designs can have hundreds of thousands, or even millions of signals. It is clear that the complexity of the learning algorithms described in this chapter is too high to be practical for our task. Since we do not want to limit our method to work only on a sub group of signals (for example: signals taken from the immediate inputs of the registers) we use heuristics and methods to filter out important data from a large database of examples. The method used is called *data mining* and it is described in Section 2.4

2.4 Data mining

> "*Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner.*" [11]

Data mining is used to automatically reveal correlation between items in large datasets. Large datasets can contain millions of items, Hence revealing relations between them is impossible by human means. In cases where relations are complex it is impossible for humans to detect them even on very small datasets. Data mining is widely used for commercial applications where economic benefit can be achieved by analyzing customers purchasing history [28].

Clock gating application can utilize data mining algorithms to reveal implicit and non-observable logic relations between design modules, using these relations to infer hold conditions to gate relevant clocks.

2.4.1 Previous work

Data mining upon execution traces is a strategy that has been put to use in the domain of software in order to discover likely program invariants [27, 26] especially in detection of common error patterns [7] and mining formal program specifications to support program verification [13]. Mining invariants has been used in hardware designs to reveal the design's formal properties [14]. The discovery of associations rules [11] has a major focus in the field of data mining research. This method extracts rules of the form $A \Rightarrow B$ from a large dataset and has been demonstrated to be efficient in the discovery of library reuse patterns in software development [2] and the discovery of global constrains of the form $(A \wedge B) \Rightarrow C$ in sequential hardware design to support formal verification [33].

Our work is predicated upon association rules concepts and methods, therefore these methods will be explained and elaborated in section 2.4.2

2.4.2 Associations rules

Associations rules mining is a widely used data mining technique that shows attribute value conditions occuring frequently together in a given dataset. A typical and widely-used example of associations rules mining is *market basket analysis*. For example, consider a database that consists of a large number of transaction records collected from supermarkets shopping baskets. Each record lists all items bought by a customer on a single purchase transaction. It is beneficial to know if certain groups of items are consistently purchased together. This data can be used for adjusting store layouts, cross-selling, promotions and to identify customer segments based on buying patterns. Association rules provide information of this type in form of probabilistic implication such as: If $x_1 \wedge x_2 ... \wedge x_i$ Then y, where the antecedent and consequent are called *itemsets*. Each association rule has two parameters that express the degree of uncertainty about the rule. The first parameter is called the *support* for the rule. The support is simply the number of transactions that include all items in the antecedent and consequent parts of the rule, or a percentage of the total number of records in the database. The second parameter depicts the *confidence* of the rule. Confidence is the ratio of the number of transactions that include all items in the consequent as well as the antecedent (namely, the support) to the number of transactions that include all items in the antecedent.

Apriori [1] is an efficient and popular bottom up approach for mining associations rules. A support threshold is set and itemsets with support above it are called *frequent itemsets*. The algorithm works iteratively with each iteration composed of two

steps: (1) Candidate generation and (2) candidate pruning. In the first iteration only 1-itemsets are taking into account, hence at the first iteration first step all the items in the database are accounted for. In step two of the first iteration, candidates with support below the support threshold are pruned, leaving only items with support above the threshold at the end of the first iteration. In the second iteration, 2-itemset candidates are generated from combinations of the 1-itemsets obtained in the first iteration. There is no need to generate all 2-itemsets from all the items in the database, because of the observation that if an itemset is frequent, then all its subsets must be frequent as well. Therefore all itemsets with infrequent subsets can be discarded before the pruning step. The algorithm iterates where at any given iteration i all frequent i-itemsets will be obtained.

We will use associations rules and apriori concepts in order to find a gating condition of the form: if $x_1 \wedge x_2 ... \wedge x_i \Rightarrow (gate\ the\ clock)$ with probability of one (always true). The method for doing so will be elaborated upon in Chapter 3. The reasoning for using the apriori approach is explained by the venn diagram illustrated in Figure 2.6. Group A consists of all the possible Boolean function over N signals, where B is a subset of A that satisfy the hold condition logic, C is a subset of B that also saves dynamic power. The relation between the groups is $C \subset B \subset A$, and $|C| << |B| << |A|$. Since only the Boolean functions in group C are beneficial for clock gating, there is no reason to explore any of the functions from groups A and B. By applying the apriori bottom up approach, the functions exploration begins at a point in group C, iteratively expanding towards the group boundaries. The exploration remains inside the group boundaries, because functions outside group C are pruned by the algorithm's pruning step.

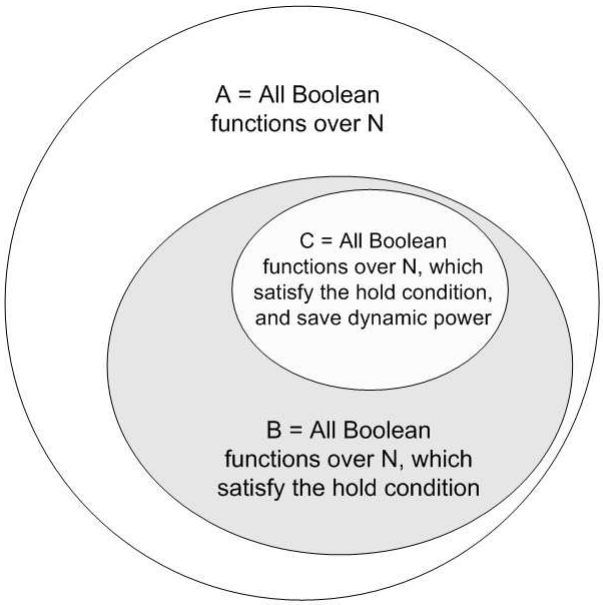


Figure 2.6: Boolean functions venn diagram

2.4.3 Summary

In this chapter we covered the concept of clock gating for reducing dynamic power dissipation in logic circuits. The equations for calculating dynamic power consumption of single nets and the entire design were introduced. Concepts and ideas from the fields of data mining and machine leaning were covered, because the method proposed in this work was inspired by them. The topics of mining associations rules via apriori algorithm and learning Boolean functions from examples were elaborated, and the shortcomings of existing methods for the purpose of clock gating were addressed. In Chapter 3 we will introduce our proposed method for learning clock gating Boolean functions that save dynamic power. To wit: we learned by analyzing selected examples yielded from scrutiny of the aforesaid simulation traces.

Chapter 3

Method

In chapters 1 and 2 we described the need for dynamic power reduction methods in the domain of modern circuits designs. The concept of clock gating was introduced while existing static and dynamic methods were covered, emphasizing their advantages and disadvantages regarding industrial size circuits design. We propose a new solution based upon dynamic method. Since we aim to reveal global circuit correlations, a task, which is not feasible by the static methods described in Section 2.2 a dynamic approach was taken.

This chapter describes a dynamic method for revealing clock gating opportunities. The problem and solution concepts are formalized in Section 3.1, while the solution's framework and stages are described in Section 3.2. The main emphasis and contribution of this work is the learning algorithm for power saving minterms. Elaboration upon the algorithm and execution example will be given in section 3.4. The framework steps and stages in this chapter are demonstrated by the partially gated 3-bit counter shown in Figure 2.3

3.1 Intelligate method

We propose a dynamic method for revealing clock gating opportunities from simulation traces. Traces coming from power and performance simulation benchmarks are rich sources for design information wherein, the logic behavior of the simulated design can be inferred by reverse engineering of the trace data. Knowledge about the design's signals values at each cycle can contribute to the task of revealing implicit relations between design signals and modules. Since modern circuit designs are very large and often comprised from millions of nets, it is impossible to detect correlations between signals or even design modules by human means. Some relations between the design's modules are inherited by the specified operation designed by a hardware engineer. Due to the large capacity and complexity of modern industrial designs, they are divided in to sub designs or *functional unit blocks* (FUB) to ease the design process. Each FUB is fairly large by itself and implicit correlations between FUBs that are inherited from the design's functionality can still be too complex for humans to observe.

We propose to learn a Boolean function from positive and negative examples derived from the simulation trace. Positive examples are cycles where the clock can be gated, while negative examples are cycles where the register changes its value and therefore the clock must be active to allow it. It is clear that errors on negative examples are not acceptable because they will alter the design's logic behavior. On the other hand, errors on positive examples have no effect upon the design's functionality. Positive errors indicate that power saving opportunities were missed. It is important to observe that the gating function should predict gating opportunities one cycle before the clock should be gated, otherwise it will be too late to gate the clock already having toggled. The extracted hold conditions need to be sound but not necessarily complete. The pertinent hold conditions are described by the logic implication in Equation 3.1,

where i is denoting a time scale (cycles in our case) and H means hold the clock for a specified group of registers.

$$f(x)_i \Rightarrow H_{i+1} \tag{3.1}$$

The method is based upon a bottom up approach. The same approach is used for associations rules mining mentioned in Section 2.4. We will use this approach to generate Boolean candidates to be the minterms of a gating DNF expression. The extracted minterms must be power beneficial and their power consumption must be lower than the power consumption of the clock they gate. In order for a minterm to be power beneficial it should be relatively small and with low activity factor. Since gating the clock of a single register is not beneficial, each extracted minterm is associated with a group of registers that it can gate, and the process of verifying the minterms and their corresponding groups of registers initiates . Minterms are verified in two steps, first dynamic verification will filter most of the erroneous minterms, then formal verification techniques will be applied to verify the remaining ones. Since we have formal verification tools at our disposal, the problem of overfitting described in Section 2.3 is not a concern. Our approach is to intelligently generate all the power saving and correct (according to the trace) minterms, so that if one or some of them are always true they will pass the verification process, and a gating DNF expression will be composed out of these minterms.

3.2 Proposed framework

The framework described in this section is a dynamic method developed to discover correlations between signals in the design, for the purpose of dynamic power reduction via clock gating. The main sources of data are the simulation traces and a list of the design's registers. As an output a clock gating function in the form of a DNF expression will be composed using the existing logic in the design as building blocks. The framework flow is described in Figure 3.1.

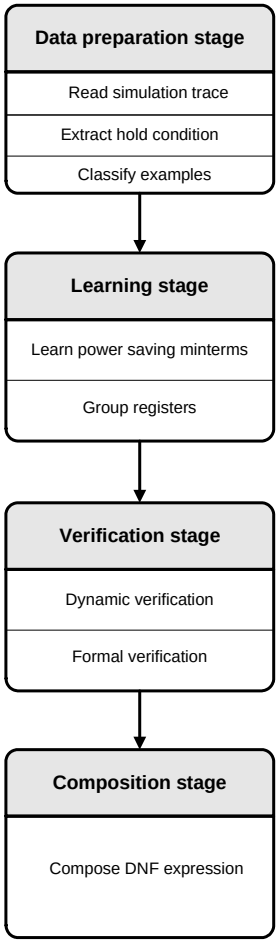


Figure 3.1: Intelligate flow

Each one of the stages will be described and explained in the following sections.

3.3 Data preparation

Our main source of data is simulation traces from power and performance benchmarks. These traces hold a partial logic behavior of the design at hand, and can be used to extract unnoticed and implicit data. In order to use the simulation traces as a useful source of data, some steps are required before a learning mechanism can be applied. The data preparation steps are described in subsections 3.3.1, 3.3.2 and 3.3.3.

3.3.1 Simulation traces

Simulation traces are the source of data for inference of clock gating functions. We can look at the trace as two dimensional matrix, where its columns represent the design's signals and the row represents the trace cycles. Each cell positioned in the i-th row and the j-th column contains the binary value of signal j at cycle i. A trace matrix example for the 3-bit counter illustrated in Figure 2.3 is shown in Table 3.1. For the sake of simplicity, only the values of the registers data, output, enable and reset signals are listed, where o_1 is the counters *most significant bit* (MSB). In the example the counter increases its value by one from cycles 0-1, on cycle 2 the reset signal is high, therefore all the registers are reset to the value 0. The increment continues on cycle 3-5,but the counter holds its current value on cycle 6, since the enable signal is low.

Since our goal is to extract clock gating conditions to save dynamic power, it is important to choose execution traces of common design behaviors. Extracting gating conditions from traces of rare execution paths is redundant, since the inferred gating conditions are unlikely to impact power consumption one way or another, on common design executions. The source of data for the framework comes from the execution of real and common applications. These executions are used as *Thermal Design Power* (TDP) tests, which are typically designed not to represent max-power, but rather

Table 3.1: Trace matrix example

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re
0	0	0	0	0	0	1	1	0
1	0	0	1	0	1	0	1	0
2	0	0	0	0	0	1	1	1
3	0	0	1	0	1	0	1	0
4	0	1	0	0	1	1	1	0
5	0	1	1	1	0	0	1	0
6	0	1	1	1	0	0	0	0
7	1	0	0	1	0	1	1	0
8	1	0	1	1	1	0	1	0
9	1	1	0	1	1	1	1	0

the maximum power that it would consume when running real applications [16]. By extracting gating functions for these applications we ensure to detect dynamic power reduction opportunities that can be applied productively.

3.3.2 Extraction the hold condition logic behavior

The hold condition determines if the clock will be gated or not. By observing the trace matrix the registers value on each trace cycle is known. We say that register j holds its value at cycle i if $M_{[i,j]} = M_{[i+1,j]}$, where M is denoting the trace matrix. In order to extract the logic behavior of the gating condition for a specific register j, we need to go over the j-th column in the trace matrix and construct a vector $H_{[0,1,2...,i-1]}$ according to Equation 3.2, when i is the number of cycles in the trace.

$$H_{[i]} = \begin{cases} 1 & M_{[i,j]} = M_{[i+1,j]} \\ 0 & M_{[i,j]} \neq M_{[i+1,j]} \end{cases} \tag{3.2}$$

An example for extracting the hold behavior of signal o_2 from Table 3.1 is shown in Table 3.2

Table 3.2: Hold condition extraction

Cycle	o_2	test	H_{o_2}
0	0	$M_{[0,1]} = M_{[1,1]}$	1
1	0	$M_{[1,1]} = M_{[2,1]}$	1
2	0	$M_{[2,1]} = M_{[3,1]}$	1
3	0	$M_{[3,1]} \neq M_{[4,1]}$	0
4	1	$M_{[4,1]} = M_{[5,1]}$	1
5	1	$M_{[5,1]} = M_{[6,1]}$	1
6	1	$M_{[6,1]} \neq M_{[7,1]}$	0
7	0	$M_{[7,1]} = M_{[8,1]}$	1
8	0	$M_{[8,1]} \neq M_{[9,1]}$	0
9	1	-	-

The resulting vector $H_{[1,1,1,0,1,1,0,1,0]}$ represents the ideal logic behavior for gating the clock of register o_2, according to the simulation trace. On each cycle it is possible to make a gating decision to the next cycle about to come.

We shall endeavor to learn a Boolean function approximating vector H. The extracted hold condition vector will be used to classify the trace cycles to positive and negative examples, from which we will learn a Boolean function, approximating H's behavior according to trace.

3.3.3 Classifying the examples

In order to be able to learn from trace examples, we need to label the positive and negative examples in the trace. A trace cycle is labeled as positive if the clock can be gated at the next cycle (the register j holds its value). We label a trace cycle as negative when the clock can not be gated at the next cycle since the register j will change its value. The labeling of each trace cycle is done according to Equation 3.3

$$Cycle_{[i]} = \begin{cases} Positive & H_{[i]} = 1 \\ Negative & H_{[i]} = 0 \end{cases} \tag{3.3}$$

According to register's o_2 hold vector H shown in Table 3.2 the trace cycles in Table 3.1 are divided to positive and negative examples as shown in Table 3.3.

Table 3.3: Trace cycles classification

Cycle	H_{o_2}	Label
0	1	Positive
1	1	Positive
2	1	Positive
3	0	Negative
4	1	Positive
5	1	Positive
6	0	Negative
7	1	Positive
8	0	Negative

The cycles classification divides the trace matrix, resulting in two tables: *positive table* shown in Table 3.4, and *negative table* shown in Table 3.5

Table 3.4: Positive table

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re
0	0	0	0	0	0	1	1	0
1	0	0	1	0	1	0	1	0
2	0	0	0	0	0	1	1	1
4	0	1	0	0	1	1	1	0
5	0	1	1	1	0	0	1	0
7	1	0	0	1	0	1	1	0

Table 3.5: Negative table

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re
3	0	0	1	0	1	0	1	0
6	0	1	1	1	0	0	0	0
8	1	0	1	1	1	0	1	0

Since we want to consider the signals negations in our learning process, we add complementary signals to each row in each table. In the bit counter example Tables 3.4 and 3.5 are replaced with Tables 3.6 and 3.7 respectively. The support of each signal at the bottom of the table is simply the number of instances the signal had the value of 1. We will refer to support as *positive support* and *negative support* respectively. For example signal re has positive support of one and zero negative support. The usage of these tables to learn power saving Boolean functions will be described in Section 3.4.

Table 3.6: Positive table with support values and complementary signals

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re	$\bar{o_1}$	$\bar{o_2}$	$\bar{o_3}$	$\bar{d_1}$	$\bar{d_2}$	$\bar{d_3}$	$\bar{en}$	$\bar{re}$
0	0	0	0	0	0	1	1	0	1	1	1	1	1	0	0	1
1	0	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1
2	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0
4	0	1	0	0	1	1	1	0	1	0	1	1	0	0	0	1
5	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1
7	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
Support	1	2	2	2	2	4	6	1	5	4	4	4	4	2	0	5

Table 3.7: Negative table with support values and complementary signals

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re	$\bar{o_1}$	$\bar{o_2}$	$\bar{o_3}$	$\bar{d_1}$	$\bar{d_2}$	$\bar{d_3}$	$\bar{en}$	$\bar{re}$
3	0	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1
6	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1
8	1	0	1	1	1	0	1	0	0	1	0	0	0	1	0	1
Support	1	1	3	2	2	0	2	0	2	2	0	1	1	3	1	3

3.4 The learning algorithm

The learning algorithm is the main contribution and emphasis of this work, its different stages and steps will be elaborated in this section. Subsection 3.4.1 describes the first iteration of the algorithm, when in Subsection 3.4.2 the process of smartly generating clock gating candidates is explained. Since not all the generated candidates are relevant pruning should be done. Pruning method and criteria are described in Subsection 3.4.3. In order to utilize the inferred minterms, groups of registers to be gated together are assigned to each clock gating minterm. The process of grouping is described in Subsection 3.4.4. At the end of this section the algorithms execution is demonstrated on the 3-bit counter mentioned in the previous chapters.

The algorithm is working iteratively in a bottom-up approach trying to find minterms satisfying the hold condition shown in Equation 3.2. Its learning examples are the complementary positive and negative tables introduced in Section 3.3.

The values of Signals/minterms positive and negative support, have a strong effect on the learning procedure. There are two observations regarding signals/minterms support that the algorithm is based upon:

Lemma 1 *Each conjunction of signals/minterms can only reduce or maintain their support.*

Lemma 2 *Each disjunction of signals/minterms can only increase or maintain their support.*

Minterms satisfying Equation 3.1 must have zero negative support. It aligns with the learning of one-sided error model described in Section 2.3, Since $\forall x \in$ Negative examples, $f(x) = 0$ and therefore the total negative error on the training set is

$\sum_{i=1}^{N} f(x) = 0$. Relaying only upon negative support to extract the gating minterms is insufficient, because zero negative support only guarantees logic correctness, but the dynamic power aspect is not addressed thereby . Positive support allows measurement of minterm quality from a perspective of dynamic power consumption. High positive support means that a high percentage of the gating opportunities where utilized, whereas low positive support means that most of the gating opportunities were missed by the minterm. A threshold for minterms positive support is determined by the user according to the desired power scheme. Minterms with positive support below the specified threshold (*support threshold*) are redundant, because they are not power beneficial from the user point of view.

The pseudo code describing the learning process is shown in Algorithm 1. The algorithm inputs are the positive and negative tables, and the trace matrix, a positive support threshold and an upper bound for minterms size . In steps 2-6 the design signals generated and pruned according to their support. These steps will be described in Subsection 3.4.1. Candidates of size up to k are generated on step 8, and elaboration upon the candidates generation process will be given in Subsection 3.4.2. Not all generated candidates are suitable for our purpose, therefore pruning is carried out in steps 9-11 and will be described in Subsection 3.4.3. We reach fixed-point when there are no more generated candidates because all candidates have been pruned or else when the minterms size has reached the upper bound k specified by the user. It is essential to find groups of registers to gate together according to the extracted minterms, since the implementation of a clock gating function for one or even a small number of registers is not beneficial in terms of dynamic power reduction. The Grouping is carried out in step 17 and will be elaborated in Subsection 3.4.4

Algorithm 1 Minterms learning algorithm

Input: Positive tables, negative table, positive support threshold Ps and minterms maximal size k.

Output: A group of gating minterms G and a group of registers to gate.

{Maximal negative support is the number of rows is the negative table}

1: Clear G, C, R

2: C_1 = Genarate candidates of size 1

3: $G = \{\text{c} \mid \text{c} \in C_1 \wedge \text{c negative support} = 0 \wedge \text{c positive support} \geq Ps\}$

4: $C_1 = \{\text{c} \mid \text{c} \in C_1 \wedge \text{c} \notin G\}$

5: $C_1 = \{\text{c} \mid \text{c} \in C_1 \wedge \text{c} \geq Ps\}$

6: $C_1 = \{\text{c} \mid \text{c} \in C_1 \wedge \text{c negative support} < \text{maximal negative support}\}$

7: **for** $i = 2 \rightarrow k$ **do**

8: C_i = Generate candidates of size i

9: $G = G \cup \{\text{c} \mid \text{c} \in C_i \wedge \text{c negative support} = 0 \wedge \text{c positive support} \geq Ps\}$

10: $C_i = \{\text{c} \mid \text{c} \in C_i \wedge \text{c} \notin G\}$

11: $C_i = \{\text{c} \mid \text{c} \in C_i \wedge \text{c} \geq Ps\}$

12: **if** $C_i = \emptyset$ **then**

13: Exit loop

14: **end if**

15: **end for**

16: **for all** $g_i \in G$ **do**

17: R_i = Group registers gated by g_i

18: **end for**

19: Return all pairs (g_i, R_i)

3.4.1 Generating and pruning literals

This Subsection describes the generation and pruning of minterms of size one (literals) and corresponds to steps 2-6 in Algorithm 1. First all the signals in the design are considered as legitimate clock gating candidates (step 2), then pruning of unsuitable signals is accomplished . Signals with zero negative support and above threshold positive support, can all be used as a clock gating signals. Therefore, all such signals are retained for the next stages of the flow (step 3), since according to Lemma 1 any more conjunctions are redundant. Lemma 1 means that conjunctions can only reduce support, in these cases only positive support can be decreased meaning less power saving. In step 5 all signals with positive support below the user defined threshold are pruned, since according to Lemma 1 their positive support will never increase due to conjunctions. Signals with maximal negative support (they have the value of one in each row in the negative table) are pruned in step 6. They can be pruned because $\forall x \in \{0,1\}$, $x \wedge 1 = x$, therefore signals with maximal negative support can not contribute for negative support reduction of any other signal/minterm.

After the generation and pruning of the signals, G contains all the signals that can be utilized for the clock gating DNF expression and C_1 contains potential candidates for clock gating literals. All candidates have positive support values indicating that they will gate enough cycles to be power worthy, but they can not act as gating signals since all of them have non zero negative support. According to Lemma 1 the only way to decrease support is by conjunctions, therefore conjunctions must be made in order to find more gating minterms.

An important observation is that at the end of stage 6 all the signals in the design with constant simulation values are pruned. Signals with constant value of one are

pruned in stage 6 (they have maximal negative support), and signals with constant value of zero are pruned in stage 5 (they have zero positive support). This means that the algorithm complexity does not correlate to the number of signals in the design, but to the number of toggling nets in the simulation trace.

3.4.2 Candidate generation

Candidates generation for minterms of size ≥ 2 is described in Algorithm 2. The procedure is based upon the candidates generated in the previous iteration and the positive and negative tables. In steps 2-8 candidates are generated by conjunctions of un pruned literals to every candidate from the previous iteration. In step 3 l denotes all the literals from the previous iteration, and every literal is conjunct to each minterm from the previous iteration candidates thereby increasing its size by one. Conjunctions between literals and minterms containing them are obviously redundant, and conjunctions between literals and minterms containing their negations are redundant as well, because $\forall x \in \{0,1\}\ x \wedge \bar{x} = 0$. After step 8 C_k contains all the candidates of size k that have unique literals and without literals and their negations in the same minterm. The number of candidates generated in steps 2-8 is shown in Equation 3.4, where k is the minterm size, $m + n$ is the number of all literals. n is the number of literals, which their negation is not present among all literals, and m is the number of literals, for which their negation exists among all literals. The combinatorial explanation is given in Appendix A.

$$Candidates = \sum_{i=0}^{min\{k,m\}} \frac{m!}{i!(m-i)!} \cdot \frac{(n-i)!}{(k-i)!(n-k)!} \tag{3.4}$$

The number of generated candidates is very high, even on relatively small designs. For example generating all candidates of size 3 from a design with only 200 signals can produce more than a million candidates for certain n and m. Luckily not all

Algorithm 2 K-candidates generation

Input: Candidates of size k-1 (C_{k-1}), positive table, negative table.

Output: Candidates of size K.

{Make conjunctions to reduce negative support}

Clear C_k
for all candidates $c_{k-1} \in C_{k-1}$ **do**
  for all literals $l \in C_{k-1}$ **do**
    if $l \notin c_{k-1} \wedge \bar{l} \notin c_{k-1}$ **then**
      $C_k = C_k \cup (c_{k-1} \wedge l)$
    end if
  end for
end for

{Filter infrequent candidates}

for all $c_k \in C_k$ **do**
  for all Sc_{k-1} = subsets of c_k of size k-1 **do**
    if $Sc_{k-1} \notin C_{k-1}$ **then**
      Filter out c_k from C_k
    else if c_k negative support = Sc_{k-1} negative support **then**
      Filter out c_k from C_k
    end if
  end for
end for
Return C_k

candidates are suitable for our cause and many of them can be filtered out. In steps 10-12 every new candidate is broken into all its subsets of size k-1 and it is tested for pruning according to Theorem 3.

Theorem 3 *Generated candidate c_k at iteration k is not power beneficial if not all its subsets of size k-1 are present as candidates of size k-1 at the end of iteration k-1.*

Proof. Candidate of size k (c_k) is composed from a conjunction of minterm m of size k-1 and a literal l. If m is not one of the previous iteration candidates, it means that m is not power beneficial since it was pruned or even not generated in previous iteration. According to Lemma 1 the conjunction $m \wedge l$ can not increase m's positive support, hence if m is not power beneficial then c_k can not be power beneficial as well.

■

In steps 13-15 the new candidates negative support is compared to the one of their subsets. According to Theorem 4 candidate can be pruned if their negative support is not lower then the negative support of all their subsets.

Theorem 4 *Candidate of size k is not power beneficial if its negative support is not lower from all of its subsets of size k-1 negative support.*

Proof. Candidate of size k (c_k) is composed from a conjunction of minterm m of size k-1 and a literal l. According to Lemma 1 c_k's negative support can only be equal or lower than the negative support of m. If it is equal to the negative support of m then the conjunction $m \wedge l$ did not reduced m negative support, hence m and l are identical signals with respect to the negative table. That means m negative support is equal to l negative support and according to Lemma 1, their positive support can only decrease due to the conjunction. Thereby, minterm m has the same negative

support as candidate c_k and higher or equal positive support, hence minterm c_k is redundant.

■

3.4.3 Candidates pruning

At the end of the candidate generation stage, candidates negative and positive support is tested. If a candidate has zero negative support and above threshold positive support it will be saved for later stages of the flow. All candidates with positive support below the user specified threshold are pruned. The pruning is performed in order to filter out logically correct minterms with low power efficiency and to speed up the algorithm convergence. At the end of the pruning process only candidates with non zero negative support and power efficiency potential remain for the next iteration of candidate generation and pruning.

3.4.4 Grouping

The grouping stage is carried out after the candidates generation and pruning iterations are finished. The output of the iterations stage is the group G of clock gating minterms. All the minterms in the group satisfy Equetion 3.2 according to the given simulation matrix and the positive and negative tables. Since the hold condition was extracted on a single register output, all the minterms in the group G were inferred accordingly thereby. Using the minterms to gate only one register will probably increase the designs dynamic power consumption since the implementation of the gating function will consume more power then the power saved by holding the clock. Therefore minterms which gate only small number of registers, are not power beneficial. In order to overcome this inefficiency, a group of registers is matched to each gating minterm. For each minterm m $\in G$ the grouping procedure described in Algorithm 3

is applied. The only cycles needed to be checked for the grouping are cycles where

Algorithm 3 Registers grouping

Input: Clock gating minterm m, trace matrix M a list of design registers L and group size threshold T.

Output: A group of registers R_m satisfied by m.

1: Clear R_m
2: C = all cycles in M, where $f(m) == 1$
3: **for all** register $r \in L$ **do**
4: **if** $\forall$ cycle $c_i \in C$, $r_{[i]} == r_{[i+1]}$ **then**
5: $R_m = R_m \cup r$
6: **end if**
7: **end for**
8: **if** $|R_m| < T$ **then**
9: $R_m = \emptyset$.
10: **end if**
11: Return R_m

the gating minterm is one, meaning the register will not change its value in the next cycle. Registers that can be gated by a minterm are all the registers such that their value is not changed after every cycle that the minterm value is one. Afterwards, the grouping process minterms with groups smaller than a user specified threshold are pruned.

An execution example of the learning algorithm described in Algorithm 1 can be demonstrated by the 3-bit counter design and its corresponding positive and negative tables 3.6 and 3.7. The positive support threshold is set to 20% of the total number of cycles and the maximal minterm size k is set to 3. In step 2

candidates of size 1 are generated, and C_1 contains all the signals in the design. $C_1 = \{o_1, o_2, o_3, d_1, d_2, en, re, \bar{o_1}, , \bar{o_2}, \bar{o_3}, \bar{d_1}, \bar{d_2}, \bar{en}, \bar{re}\}$. Signals d_3 and $\bar{d_3}$ are missing in C_1 since they are equal to $\bar{o_3}$ and o_3 . The relation between these signals is marked and will be restored at the end of the grouping stage. In step 3 literals with zero negative support and positive support $\geq \frac{20}{100} \cdot 10 = 2$ are searched out. In our example signal $\bar{o_3}$ is satisfying these terms ,therefore $G = \{\bar{o_3}\}$ and in step 4, $C_1 = \{o_1, o_2, o_3, d_1, d_2, en, \bar{o_1}, , \bar{o_2}, \bar{d_1}, \bar{d_2}, \bar{en}, \bar{re}\}$. In step 5 all signals with positive support below 2 are pruned, meaning o_1, re, and $\bar{en}$ are filtered from C_1. Signals o_3, and $\bar{re}$ have maximal negative support (3) and therefore are pruned in step 6 resulting in $C_1 = \{o_2, d_1, d_2, en, \bar{o_1}, , \bar{o_2}, \bar{d_1}, \bar{d_2}\}$ at the end of the literals generation and pruning stage.

In step 8 minterms of size 2 are generated based upon the literals in C_1. All the possible combinations of minterms and their status are shown in Table 3.8. We can avoid generating some of the minterms according to Theorem 4. The negative support of the minterms that are not generated is not lower then the negative support of each of the literals from which each minterm is comprised. in our example only the minterms in C_2 passed this criteria. $C_2 = \{o_2 \wedge d_2, o_2 \wedge en, o_2 \wedge \bar{d_1}, d_1 \wedge d_2, d_1 \wedge en, d_1 \wedge \bar{o_1}, d_1 \wedge \bar{o_2}, d_2 \wedge \bar{o_1}, en \wedge \bar{o_1}, en \wedge \bar{d_2}, \bar{o_1} \wedge \bar{o_2}, \bar{o_2} \wedge \bar{d_2}, \bar{d_1} \wedge \bar{d_2}\}$

Minterms with zero negative support and threshold above positive support are searched in C_2. It is easy to see in Table 3.8, that $(o_2 \wedge en)$, $(en \wedge \bar{d_2})$, $(\bar{o_2} \wedge \bar{d_2})$ and $(\bar{d_1} \wedge \bar{d_2})$ are meeting the criteria, thereby $G = \{\bar{o_3}, o_2 \wedge en, en \wedge \bar{d_2}, \bar{o_2} \wedge \bar{d_2}, \bar{d_1} \wedge \bar{d_2}\}$ and $C_2 = \{o_2 \wedge d_2, o_2 \wedge \bar{d_1}, d_1 \wedge d_2, d_1 \wedge en, d_1 \wedge \bar{o_1}, d_1 \wedge \bar{o_2}, d_2 \wedge \bar{o_1}, en \wedge \bar{o_1}, \bar{o_1} \wedge \bar{o_2}\}$.

Since minterms with below threshold positive support are not power beneficial, they are pruned in step 11 of the learning algorithm. After the pruning of minterms with

Table 3.8: Possible minterms of size 2

Minterm	Positive support	Negative support	Generated	Pruned
$o_2 \wedge d_1$	1	1	No	-
$o_2 \wedge d_2$	1	0	Yes	Yes
$o_2 \wedge en$	2	0	Yes	No
$o_2 \wedge \bar{o_1}$	2	1	No	-
$o_2 \wedge \bar{d_1}$	1	0	Yes	Yes
$o_2 \wedge \bar{d_2}$	1	1	No	-
$d_1 \wedge d_2$	0	1	Yes	Yes
$d_1 \wedge en$	2	1	Yes	No
$d_1 \wedge \bar{o_1}$	2	1	Yes	No
$d_1 \wedge \bar{o_2}$	1	1	Yes	Yes
$d_1 \wedge \bar{d_2}$	2	1	No	-
$d_2 \wedge en$	2	2	No	-
$d_2 \wedge \bar{o_1}$	2	1	Yes	No
$d_2 \wedge \bar{o_2}$	1	2	No	-
$d_2 \wedge \bar{d_1}$	2	1	No	-
$en \wedge \bar{o_1}$	5	1	Yes	No
$en \wedge \bar{o_2}$	4	2	No	-
$en \wedge \bar{d_1}$	4	1	No	-
$en \wedge \bar{d_2}$	4	0	Yes	No
$\bar{o_1} \wedge \bar{o_2}$	3	1	Yes	No
$\bar{o_1} \wedge \bar{d_1}$	4	1	No	-
$\bar{o_1} \wedge \bar{d_2}$	3	1	No	-
$\bar{o_2} \wedge \bar{d_1}$	3	1	No	-
$\bar{o_2} \wedge \bar{d_2}$	3	0	Yes	No
$\bar{d_1} \wedge \bar{d_2}$	2	0	Yes	No

positive support ≤ 2, $C_2 = \{d_1 \wedge en, d_1 \wedge \bar{o_1}, d_2 \wedge \bar{o_1}, en \wedge \bar{o_1}, \bar{o_1} \wedge \bar{o_2}\}$ and minterms generation of size 3 begins. Possible minterms combinations of size 3 are shown at Table 3.9. None of the combinations are generated because they are not power beneficial according to Theorem 3 and Theorem 4. Since $C_3 = \emptyset$ the iterations stage is complete yielding 5 clock gating minterms.

Table 3.9: Possible minterms of size 3

Minterm	Positive support	Negative support	Generated	Reason
$d_1 \wedge d_2 \wedge en$	0	1	No	Theorem 4
$d_1 \wedge d_2 \wedge \bar{o_1}$	0	0	No	Theorem 3
$d_1 \wedge d_2 \wedge \bar{o_2}$	0	1	No	Theorem 4
$d_2 \wedge en \wedge \bar{o_1}$	2	2	No	Theorem 4
$d_2 \wedge en \wedge \bar{o_2}$	1	2	No	Theorem 4
$en \wedge \bar{o_1} \wedge \bar{o_2}$	3	1	No	Theorem 3

In the grouping stage a group of registers is matched to each gating minterm $m \in G$. The grouping process for $\bar{o_3}$ is done as follows: $\bar{o_3}$ has the value of 1 at cycles {0,2,4,7}. Each of the designs registers that hold its value from cycles 0 to 1, 2 to 3, 4 to 5 and 7 to 8 can be gated by $\bar{o_3}$. In our design there are only 3 registers o_1, o_2 and o_3. According to the trace matrix, o_1 and o_2 hold their values at the specified cycles, therefore the group $R_{o_3} = \{o_1, o_2\}$ is formed. In the same manner, groups are formed to all the minterms in G, resulting Table 3.10. d_3 is added to the table since it is equal to $\bar{o_3}$, therefore it should be tested in case $\bar{o_3}$ will fail in the verification process (the signals may be equal only in the trace matrix, but different in other executions of the design).

Table 3.10: Gating minterms groups

Minterm	Group
$\bar{o_3}$	$\{o_1, o_2\}$
d_3	$\{o_1, o_2\}$
$o_2 \wedge en$	$\{o_1, o_2\}$
$en \wedge \bar{d_2}$	$\{o_1, o_2\}$
$\bar{o_2} \wedge \bar{d_2}$	$\{o_1, o_2\}$
$\bar{d_1} \wedge \bar{d_2}$	$\{o_1, o_2\}$

3.4.5 Learning Efficiency Approximation

At the end of the minterms learning and grouping stages the efficiency and generalization of the inferred minterms and groups should be approximated. In our work, the approximation should take place in two domains, first we should strive for logic correctness (only one side errors) for unseen scenarios, then the minterms' power reduction efficiency should be approximated.

In order to address the learning efficiency a holdout data for cross-validation can be utilized, where the learning procedure is based upon a single simulation trace and the inferred minterms are tested for logic correctness and dynamic power reduction upon a set of heldout simulation traces. Minterms must be logically correct upon all simulations cycles, where their power reduction efficiency is tested as the average over all traces (training and test sets).

3.4.6 Summary

In this section we described the minterms learning algorithm. The output of the learning process is a group of minterms, each minterm corresponding to a group of

registers it can gate. The minterms are functionally correct only according to the given simulation trace, and a process to verify their global correctness is addressed in Section 3.5.

3.5 Dynamic and formal verification

The output of the learning stage consist of a group of minterms and a group of registers that each minterm gates. The minterms have one-sided positive error greater or equal to zero, and no negative error. Therefore, according to the given traces the minterms are functionally correct and will save dynamic power if they are applied. Since the simulation traces are a subset of all execution possibilities, the inferred minterms can have negative errors on execution paths derived from a different distribution. Before any clock gating functionality can be integrated to the circuit, it is vital to verify its correctness. The logic implication $f(m)_i \Rightarrow (R_{m_i} = R_{m_{i+1}})$ must be verified as correct, where i is denoting the execution cycle, and R_{m_i} is the group of registers to be gated by minterm m.

The verification process is accomplished in two steps. First *dynamic verification* [25] is used to filter erroneous minterms. Dynamic verification is carried out by simulation wherein test vectors are used as stimuli for the tested design and predefined properties are tested throughout the simulation. For our purpose, properties of the form of the logic implication described above are tested in parallel for each minterm and its group of gated registers.

Dynamic verification has high coverage in terms of possible execution paths, but it is not complete. Minterms that are verified by the dynamic verification process are likely to hold in all execution paths, but the low probability of error is unacceptable in the domain of logic design. Since the effect of altering the design functionality is hazardous, clock gating functionality must be completely verified before it can be integrated. *Formal Property Verification* (FPV) [10] is a field in *Formal Verification* (FV) [9], where properties are verified formally, it means that a verified property is

guaranteed to be true.

The reasoning for applying the dynamic verification process before the formal verification process, is due to the complexity involved in the formal property verification, especially on the industrial scale designs. We rely on the high coverage of the dynamic verification process to filter most, if not all of the erroneous minterms, leaving only a handful of minterms to be formally verified. Since the framework extracts global relations in the design and uses them to halt portions of the clock network, it is possible that current state of the art formal verification tools will not be able to cope with such capacity of logic. In this case, a human expert (the circuit's designer for example) can be consulted to verify the gating functions based upon his design knowledge. For example, in the 3-bit counter example, the minterms and their corresponding groups listed in Table 3.10 need to be verified. Since the original trace matrix has only partial coverage of the circuits functionality, dynamic verification is needed to complete as much information as possible. The dynamic verification process should reveal that minterms $(o_2 \wedge en)$, $(en \wedge \bar{o_2})$ and $(\bar{d_1} \wedge \bar{d_2})$ are not true for all execution paths of the circuit. This can be demonstrated by the one cycle transition shown in Table 3.11. This transition can be treated as cycles 10 and 11 of the trace matrix. In cycle 10 $(o_2 \wedge en)$, $(en \wedge \bar{o_2})$ and $(\bar{d_1} \wedge \bar{d_2})$ have the value of 1, meaning that o_1 and o_2 will hold their current values (0), since o_1 and o_2 have the value of 1 in cycle 11, these minterms are erroneous and can be ignored.

Table 3.11: Dynamic verification cycles

Cycle	o_1	o_2	o_3	d_1	d_2	d_3	en	re
10	1	1	1	0	0	0	1	0
11	0	0	0	0	0	1	1	0

The dynamic verification process pruned three erroneous minterm inferred from the partial information given by the trace. The three minterms, which passed the dynamic verification process are o_3, d_3 and $(\bar{o_2} \wedge \bar{d_2})$. Minterms that pass the dynamic verification process have high probability to be correct. Since no functionality can be integrated to the circuit without assurance of correctness, formal verification must be applied as well.

In our case, a human expert familiar with bit-counters functionality can easily verify the inferred gating minterms. The gated group is $\{o_1, o_2\}$, these registers are the counter's two highest bits and o_3 is the *least significant bit* (LSB). It is clear that when the LSB in a bit counter is 0, the only change in the register's values on the next increment is the LSB flipping its value to 1. This inherit logical behavior helps to verify signal $\bar{o_3}$. Signal d_3 is the input signal to the LSB, and its value is always the next value to be loaded into the register upon the next increment. Since o_3 flips its value on every increment, d_3 likewise flips on every increment as well, and $d_3 = \bar{o_3}$, therefore d_3 is verified as well. The minterm $(\bar{o_2} \wedge \bar{d_2})$ is a little more complex because it is composed from a conjunction of two signals. It is obvious that the minterm is correct after interpreting it in the context of the bit counter: when the middle bit is 0 ($\bar{o_2} = 1$) and will be 0 in the next cycle ($\bar{d_2} = 1$), then the middle bit (obvious from the term) and the MSB will hold their values. Adding design information can also reveal that when re is 1 then the group of registers will hold its value since all the registers are reset to 0. The re signal was pruned since in the given trace it does not occur enough times to be implementation worthy.

In this section we described the verification process of the inferred gating minterms and the groups of registers they gate. We demonstrated the dynamic verification process on the 3-bit counter circuit, and showed how information provided through

human knowledge can assist or even replace the formal verification process. After the verification process a DNF expression should be composed from the verified minterms. This process is described in Section 3.6.

3.6 DNF construction

After the verification stage, minterms and their corresponding groups of registers can be combined to form a DNF expression. Each minterm can be implemented as a clock gating function by itself, since it has enough positive support and it gates sufficient number of registers to be power worthy. According to Lemma 2, disjunctions between minterms can increase the overall support, therefore a DNF expression will be composed by building a disjunction between the gating minterms.

Each minterm m_i corresponds to the group of registers $R_i = [r_1, r_2, ..., r_n]$ it can gate. Since R_i might be different from R_j for $m_i \neq m_j$, every disjunction of m_i with m_j, divides R_i and R_j to two subsets: $R_{ii} = \{r \mid r \in R_i \wedge r \notin R_j\}$, $R_{ij} = \{\ r \mid r \in R_i \wedge r \in R_j\}$ and R_{jj}, R_{ji} respectively. $\forall\ i, j,\ R_{ij} \neq \emptyset$ since at least one register is common to all minterms (the register upon the hold condition is extracted. See Section 3.3) and $R_{ii} = \emptyset$ when $R_i \subseteqq R_j$ and vise versa. If $R_{ii} = \emptyset$, $R_{jj} = \emptyset$ ($R_i = R_j$) then one of the minterms m_i or m_j may be redundant. The criteria for redundancy is the amount of positive support added due to the disjunction. If m_i positive support $> m_j$ positive support and (($m_i \vee m_j$) positive support - m_j positive support) $\geq$ user defined threshold T, then m_j is redundant, and vise versa. The first step in composing the DNF expression is to filter out all redundant minterms. Minterms, which satisfy $R_i = R_j$ and pass the user defined threshold are joined by disjunction, which adds enough positive support to justify the addition of the disjunction operation (adding an OR gate in the circuit). The second step is to group minterms with large commonality between their groups of registers. Registers which, are not covered by all the minterms in the minterms group are filtered out. After register filtering, the redundancy check is repeated in each group of minterms. If there are different groups of minterms with large intersection between their gated registers then a DNF

expression is formed to each one of the groups.

The circuits designer's knowledge and reasoning can add a great deal of information to the DNF expression construction, decisions whereof, minterms and groups of registers to add to the expression are best made by the designer, in order to consider knowledge about factors which are beyond the scope of this work. The physical distance between literals in the same minterm, minterms and the registers they gate and the distance between the registers in the same group can cause changes in the minterm's components and the group of registers it gates. Other factors including area and timing can also be addressed by the designer, helping to compose effective clock gating DNF expression.

The composition of the DNF expression can be demonstrated on the 3-bit counter example. The verified minterms and their corresponding groups are listed in Table 3.12.

Table 3.12: Verified minterms and their groups

Minterm	Group	Positive support	Gating cycles
$\bar{o_3}$	$\{o_1, o_2\}$	4	{0,2,4,7}
d_3	$\{o_1, o_2\}$	4	{0,2,4,7}
$\bar{o_2} \wedge \bar{d_2}$	$\{o_1, o_2\}$	3	{0,2,4}

All the minterms gate the same group of registers. It is clear that $(\bar{o_2} \wedge \bar{d_2})$ is redundant because it does not add any positive support to $\bar{o_3}$ or d_3. Each one of the minterms $\bar{o_3}$ and d_3 can be integrated in the design as a clock gating signal to registers $\{o_1, o_2\}$. The DNF expression must be negated before it can be used to gate the clock, therefore the power efficiency of $\bar{o_3}$ as the gating minterm is better than

d_3, since the negation of $\bar{o_3}$ is already implemented in the design (o_3). If d_3 was implemented, then a NOT gate was added, consuming more power and space. The bit counter circuit with the clock gating implementation of o_3 is displayed in Figure 3.2. The implementation with $\bar{d_3}$ as a gating minterm is displayed in Figure 3.3.

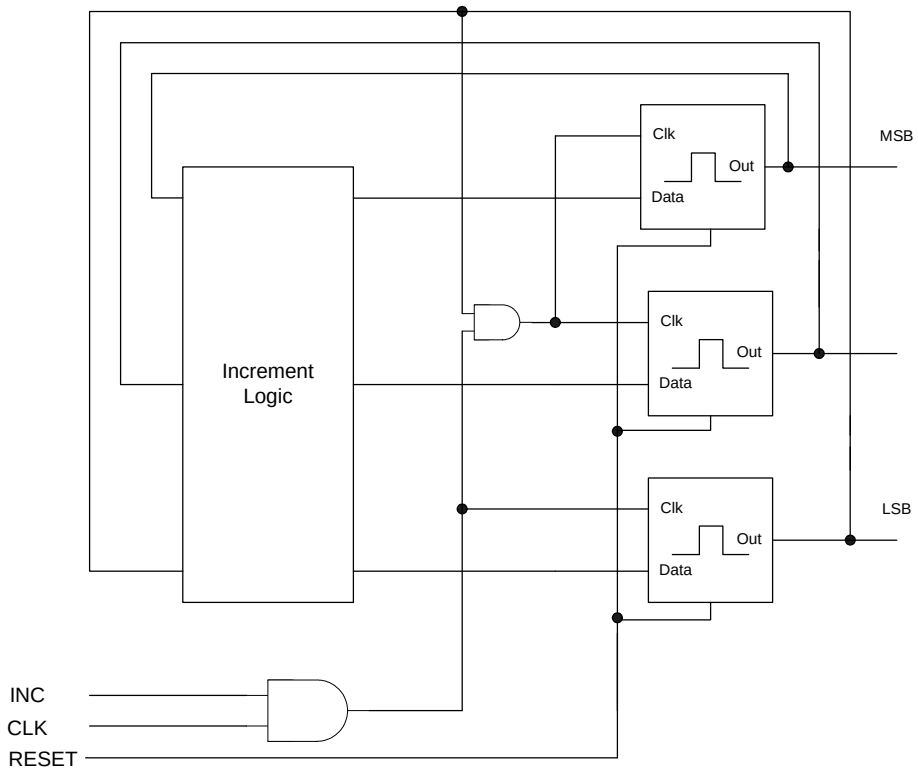


Figure 3.2: 3 bit counter Intelligated 1

3.6.1 Summary

In this chapter we have described a new dynamic method for revealing and implementing clock gating functions for dynamic power reduction. The inferred minterms can be implemented as clock gating functions by themselves, or can be combined and form a gating DNF expression. Human knowledge and reasoning can be applied to choose minterms that are correct and to prune minterms which are not. The algorithm was

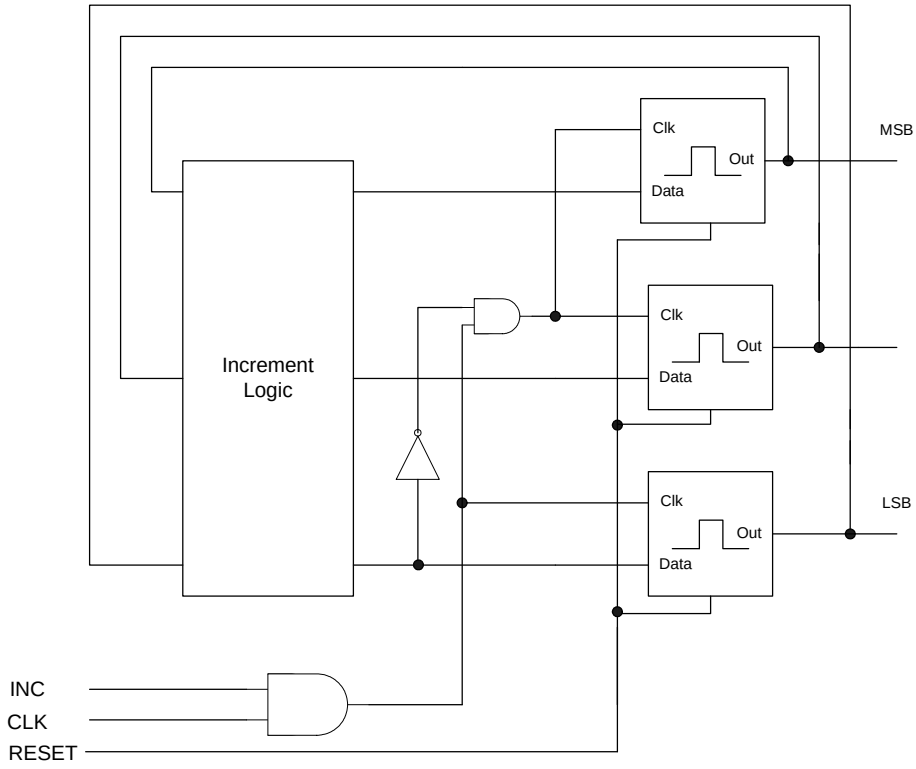


Figure 3.3: 3 bit counter Intelligated 2

implemented and tested on a state of the art next generation INTEL micro-processor design. The experiments method and results are explained in Chapter 4.

Chapter 4

Experimental results

In Chapter 3 we described the learning algorithm method. In this Chapter we report the results of experiments to test the algorithm's feasibility over industrial-size designs and the dynamic power efficiency of the inferred gating conditions. Therefore the experiments where conducted on real industrial state of the art next generation INTEL micro-processor design.

Thereby, the research questions regarding the algorithm's ability to infer global power saving minterms from simulation traces and its feasibility to cope with the capacity of industrial state of the art chip designs are answered.

4.1 Experimental method

The trace data came from power benchmarks simulations carried out during the design's power evaluation. These power tests are known as TDP's (see Section 3.3). We had 8 TDPs at hand to work with. Each test represented simulation trace of a real application execution such as zipping files and reading data through the computer's infrared port. Each simulation consists of millions of cycles, and since building a trace matrix to the entire simulation trace is not feasible, snippets with high power activity were extracted. Each snippet contains 5000 simulation cycles (10,000 phases), representing high power consumption scenario needed to be analyzed.

The learning procedure was applied on one test, where the rest of the tests were used to verify the logic correctness of the inferred minterms (dynamic verification) and to test the minterms average power reduction over all TDPs. The minterms were not formally verified, instead a human expert assisted in the ratification of some of the interesting cases.

4.1.1 Dynamic power saving metric

The power metric for testing dynamic power reduction of a gating function is the percentage of the activity factor reduced from the gated clock. According to the dynamic power consumption formula depicted by Equation 2.1, there is a direct proportion between the nets activity factor and its dynamic power consumption, therefore there is a direct proportion between the activity factor reduction and the nets dynamic power reduction. For example: a *free running clock* is a clock net with activity factor of 1, meaning that it is toggling on every cycle. By gating the clock net with a function that is active on 50% of the simulation cycles, the clock net power consumption is

reduced by 50%.

4.2 Experiments

We started the experiments by applying the learning algorithm on relatively small FUB that consists of 6998 nets. Surprisingly only a small fraction of the FUB's nets toggled during the simulation (159) and the majority of the nets were constant. Since the algorithm's run time complexity depends on the number of toggling nets regardless of the total number of nets in the design (See Subsection 3.4.1), the typical percentage of toggling nets in the TDPs was addressed in Subsection 4.2.1. Out of the 159 toggling nets, a register was chosen at random and the minterms were inferred according to its hold behavior. After the cycles labeling (see Section 3.3) the positive table contained 2433 positive examples and the negative table contained 2566 negative examples. The learning algorithm was applied on the extracted data, when the minterms' maximal size was limited to 4 and several support thresholds were applied. The algorithm's run time and the number of inferred minterms are summarized in Table 4.1.

The learning run time and the number of inferred minterms are decreased as the support threshold is increased. The correlation between the support threshold and the procedure run time is shown in Figure 4.1, and the correlation to the number of inferred minterms is shown in Figure 4.2. The number of candidates at the end of each iteration increases until the third iteration, then the algorithm starts to converge and the number of candidates then decrease. This means that after three conjunctions, the positive support of many of the generated minterms is too low, thereby they are pruned by not meeting the support threshold. An experiment to test the tendency of

Table 4.1: Run results

Threshold	Iteration 1		Iteration 2		Iteration 3		Iteration 4		Total time (sec)	Inferred minterms
	Candidates	Time (sec)	Candidates	Time (sec)	Candidates	Time (sec)	Candidates	Time (sec)		
5%	141	0	6209	2	113720	46	1388961	839	887	28
10%	134	0	4966	1	24747	31	31554	101	133	10
15%	129	0	1514	1	2899	7	1683	33	41	7
20%	124	0	809	1	746	3	90	6	10	5
25%	122	0	109	1	24	0	0	0	1	4

The left column indicates the support threshold used. The Iterations columns indicates the number of candidates at the end of the iteration and its run time.

The total run time and the number of inferred minterms are listed in the right hand columns

convergence in early iterations is described in Subsection 4.2.2.

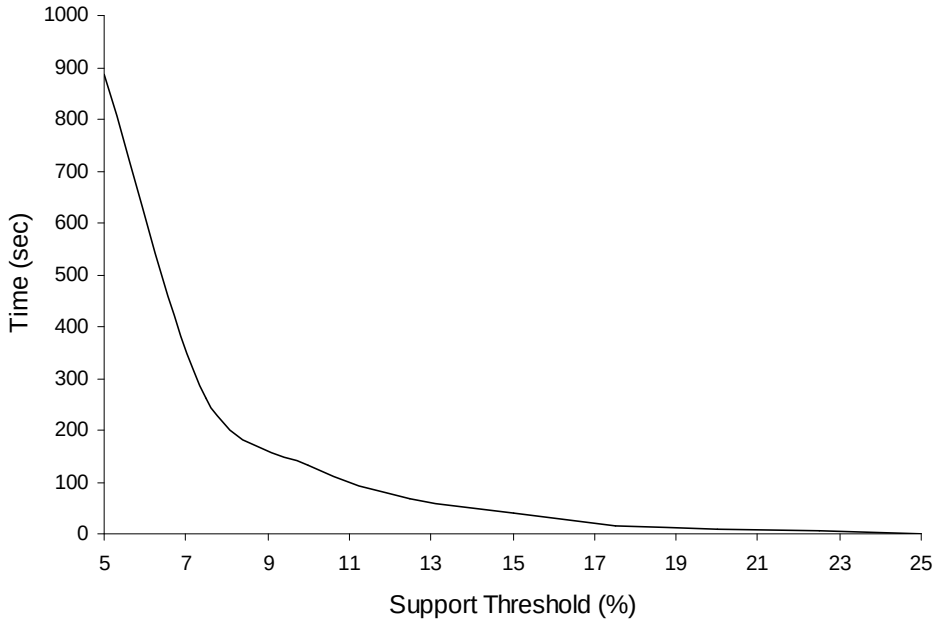


Figure 4.1: Run time Graph

In order to conclude on the learning method usefulness, the inferred minterms must be analyzed in terms of dynamic power reduction. Each minterm is analyzed by the number of registers it gates and its support and thus the percentage of cycles it gates. The analysis of the minterms inferred by applying the learning procedure with support threshold of 25% and minterms maximal size of 4 is summarized in Table 4.2. Since the clock nets of the registers are free running, the support of the gating minterm is equivalent to the activity factor reduction from the clock nets. Hence the gating minterms reduce 23%-37% from the clock nets dynamic power. The group of registers gated by the minterm $b \wedge \bar{c}$ is contained by the group of registers gated by the literal $\bar{a}$, and these minterms were composed to a DNF expression yielding an avarage of 55% dynamic power reduction on the 20 registers gated by both of them. The 18 registers gated by $\bar{a}$ and not by $b \wedge \bar{c}$ can still be gated by only a part of the

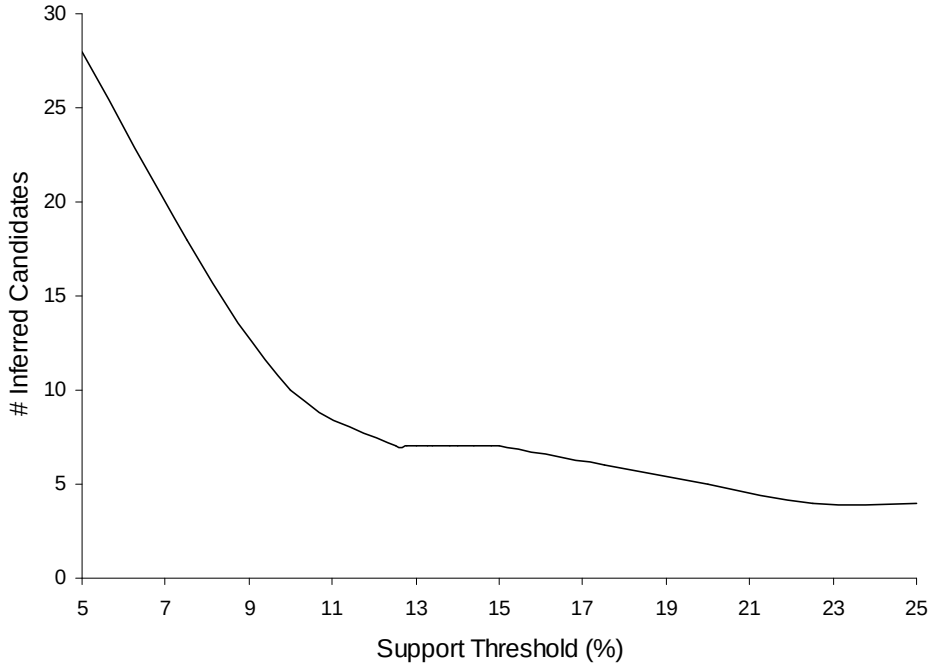


Figure 4.2: Inferred minterms graph

DNF expression. In order to use the gating function as a clock enable, it has to be negated, therefore $F = \neg(b \wedge \bar{c} \vee \bar{a})$. F can be optimized by applying De-morgan's law [17], resulting in $F = (\bar{b} \vee c) \wedge a$. An implementation for the gating function F is illustrated in Figure 4.3, where signal a gates 18 registers covered only thereby, and the function $F = (\bar{b} \vee c) \wedge a$, gates the 20 registers covered by both minterms. The signals a, b and c were present at the design and therefore their implementation does not add any power consumption where the AND, OR and NOT gates and the nets connecting them were added; thus the dynamic power saved by gating the clock nets must be greater than the dynamic power consumed by the added logic (see Appendix B for dynamic power estimation).

Further analysis of the inferred minterms revealed that the signals a, b and c are some of the FUB's input signals. The driving logic of input signals is implemented outside the FUB's boundaries, meaning that the logic implementation for inputs is processed

Table 4.2: Inferred minterms analysis

Minterm	Registers group size	Power reduction								
		TDP 1	TDP 2	TDP 3	TDP 4	TDP 5	TDP 6	TDP 7	TDP 8	Avarage
$\bar{a}$	38	36%	56%	18%	56%	40%	72%	0%	15%	37%
$b \wedge \bar{c}$	20	28%	48%	10%	22%	28%	48%	0%	7%	23%
$d \wedge \bar{c}$	18	25%	35%	11%	30%	27%	48%	0%	7%	23%
$e \wedge f$	10	24%	43%	21%	29%	30%	48%	0%	16%	26%

The minterms column lists the inferred minterms, and the number of registers each minterm gates is listed in the second column. Minterms power reduction from the clock nets is listed in the power reduction columns

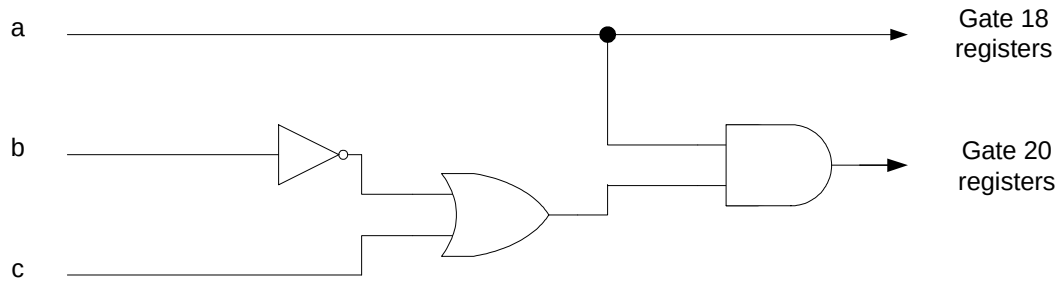


Figure 4.3: Gating implementation

in the FUB or FUBs driving it. Figure 4.4 illustrates *cross-FUB gating*, where the logic implementation of a, b and c is not present in the gated FUB. The discovery of cross-FUB gating is not possible by the static methods described in Subsection 2.2.1, since the gating functionality is implemented outside the FUB's boundary and the explicit representation thereof is not present at the FUB level. As far as we know there is no method, static or dynamic, which is capable of inferring cross-FUB gating functions. In order to further understand the inferred cross-FUB gating, the learning algorithm was applied on a higher logic hierarchical level. The experiment is described in Subsection 4.2.3.

The signals comprising the minterms that are inferred in the cross-FUB experiment

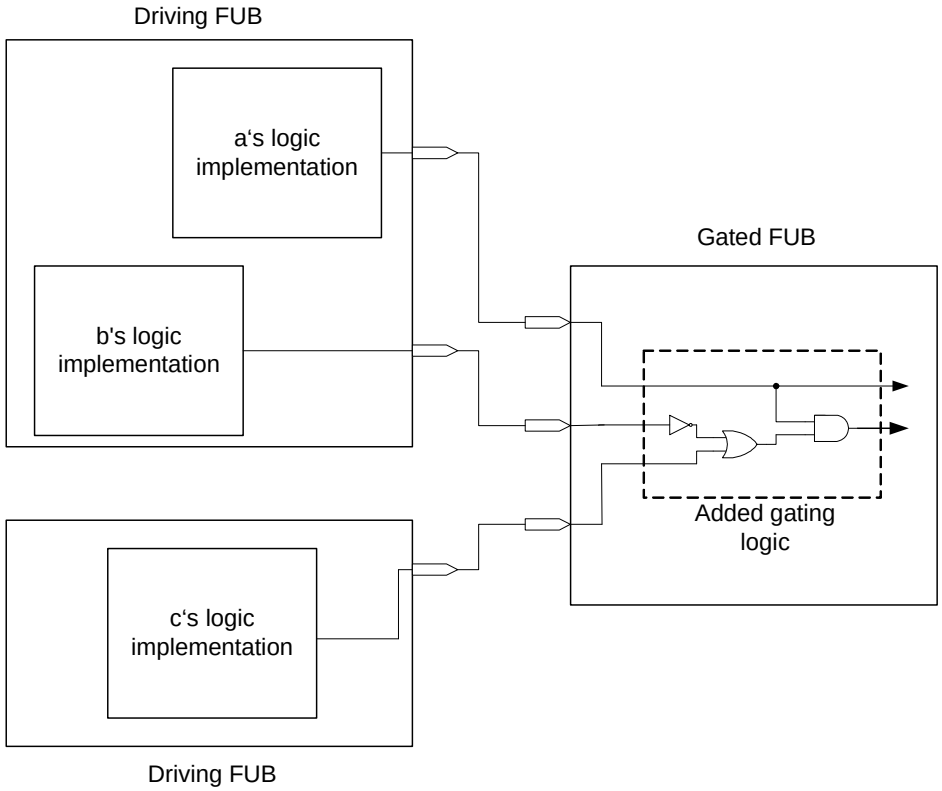


Figure 4.4: Cross-FUB clock gating

contain some of the neighboring logic hierarchy input signals as well, therefore an attempt was made to apply the learning algorithm on the next level of hierarchy. The algorithm was applied on a *unit level* hierarchy. Unit level hierarchy contains a complete logic functionality of the design and it is composed of dozens of FUBs. The unit level clock gating experiment is described in Subsection 4.2.4.

Experiment V tested the algorithm efficiency over five FUBs on each one of the two registers that were chosen at random to be learnt upon their hold behavior. The experiments and results are described in Subsection 4.2.5.

4.2.1 Experiment I - TDPs toggling characteristics

In this experiment the ratio between the toggling signals in the TDP tests and the total number of signals in the design was tested. Ten FUBs from different sizes and

functional units of the design were randomly selected and the percentage of toggling nets was extracted for each one of the TDP benchmarks. The results shown in Table 4.3 indicates that only a relatively small fraction of the FUB's nets are active during typical executions (5.3% on average); thus the learning procedure's ability to cope with very large designs seems feasible.

Table 4.3: Percentage of toggling nets

FUB	Total nets	TDP 1	TDP 2	TDP 3	TDP 4	TDP 5	TDP 6	TDP 7	TDP 8	Average
FUB_1	89944	7672	7228	7310	7423	7326	7683	3826	3826	7.2%
FUB_2	142553	476	474	468	474	478	479	431	439	0.3%
FUB_3	139955	5072	3510	3072	3868	3818	4863	832	801	2.3%
FUB_4	55922	2115	2401	2283	2138	2149	2298	1326	1772	6.1%
FUB_5	26617	4920	4934	4410	4795	4764	4889	2594	2986	16.1%
FUB_6	133422	12065	12804	11975	11810	11663	12554	4752	5652	7.8%
FUB_7	391301	11290	10301	10181	8529	8053	11032	3324	4237	2.1%
FUB_8	5901	604	646	450	451	522	523	292	270	7.9%
FUB_9	89101	9675	9910	7245	12206	12256	12113	5103	2994	10.0%
FUB_{10}	486716	62793	64036	55528	61067	60046	60695	33074	31105	11.0%
Average	15614.32	7.4%	7.4%	6.6%	7.2%	7.1%	7.5%	3.5%	3.4%	5.3%

The table shows the number of total nets in each FUB and the number of toggling nets in each TDP test

4.2.2 Experiment II - Candidate generation convergence

Ten different FUBs were randomly chosen from different functional units of the design. In the same manner we chose randomly a register for which to learn gating condition. The results shown in Table 4.4 indicate that the number of candidates peaks at iterations 2-3, and begins to converge towards zero afterwards. Figure 4.5 illustrates the convergence graphically, when the values are the average of the experiments over the ten FUBs.

Table 4.4: Candidates convergence

FUB	Iteration 1	Iteration 2	Iteration 3	Iteration 4
FUB_1	304	6383	4353	1883
FUB_2	120	1396	636	41
FUB_3	132	3247	7341	2196
FUB_4	175	2479	6080	4652
FUB_5	114	1161	2128	513
FUB_6	548	5902	4664	857
FUB_7	316	1051	145	0
FUB_8	117	791	634	94
FUB_9	118	400	337	285
FUB_{10}	124	354	96	0

Each iteration column inducates the number of candidates remain after pruning for the specified FUB run.

4.2.3 Experiment III - cross-FUB gating

The experiment was conducted on a section of the design containing four FUBs, when we have applied Intelligate on one of the four FUBs. We focused upon the cross-FUB learning on the same register for which we have learned a gating condition in term of minterms. The results are summarized in Table 4.5. Signals a, b and c gate the same group of 114 registers over two FUBs. The disjunction of b and c does not add support to any of them; thus c can be omitted from the DNF expression, where the disjunction $F = a \vee b$ has an average support of 65%, deriving the gating function $F = \neg(\bar{a} \vee \bar{b}) = a \wedge b$.

Figure 4.5: Candidates convergence graph

Table 4.5: Minterms analysis

Minterm	Registers group size	Power reduction								
		TDP 1	TDP 2	TDP 3	TDP 4	TDP 5	TDP 6	TDP 7	TDP 8	Avarage
$\bar{a}$	114	36%	56%	18%	56%	40%	72%	0%	15%	37%
$\bar{b}$	114	32%	36%	18%	56%	38%	72%	0%	14%	33%
$\bar{c}$	114	34%	50%	18%	56%	38%	72%	0%	14%	23%

The mintems column lists the inferred minterms for each of them gates 114 registers, whereas the dynamic power reduction from the clock nets is listed in the power reduction columns.

4.2.4 Experiment IV - Unit level clock gating

The unit experimented on contains 2,664,269 signals, and as far as we know, there is no clock gating method able to cope with designs of this magnitude. Since only 178,778 signals toggled during the TDP benchmark, the minterms' learning was feasible by Intelligate, and the experimental result is displayed in Table 4.6. The inferred gating signal a was inferred in each hierarchy level wherein the algorithm was applied ; thus it is the same signal. With the knowledge of a human expert the inferred gat-

ing signal was identified as a control signal for a structural loop in the design. When the loop control is disabled the registers inside the loop are inactive and therefore can be gated. The signal a is gating 1603 clock nets in eleven different unit's FUBs. The gating distribution among the FUBs is shown in Table 4.7. A new gated clock network can be applied to FUBs with substantial amount of gated registers.

Table 4.6: Unit level inferred minterms

Minterm	Register group size	Number of FUBS covered	Power reduction								
			TDP 1	TDP 2	TDP 3	TDP 4	TDP 5	TDP 6	TDP 7	TDP 8	Average
a	1603	11	36%	56%	18%	56%	40%	72%	0%	15%	37%

The minterm found is in the left column, gates 1603 registers in 11 FUBs in the unit. The power reduction columns list the power reduction on the clock nets.

4.2.5 Experiment V - Random results

The learning algorithm was experimented upon randomly chosen FUBs and target set of registers. For almost every FUB and register pairs the algorithm yields effective gating minterms. The most meaningful minterms that were inferred are summarized in Table 4.8.

4.2.6 Summary

In this chapter we described in detail the source of data input for the learning algorithm and defined a dynamic power saving metric to test the inferred minterms' power efficiency. Five experiments were conducted to determine the average ratio between the total number of signals in the design and the number of toggling signals during the TDP benchmarks, the algorithms feasibility and efficiency over industrial-size state

Table 4.7: Unit level clock gating

FUB	Registers group size
FUB_1	204
FUB_2	628
FUB_3	127
FUB_4	51
FUB_5	27
FUB_6	55
FUB_7	380
FUB_8	8
FUB_9	7
FUB_{10}	89
FUB_{11}	27

A list of FUBs gated by the unit level gating signals, and the number of registers gated in each FUB.

Table 4.8: Random experiments results

FUB	Hold register	Minterm size	Power saving			Number of gated registers
			min	max	average	
FUB_1	reg_1	1	0%	89%	22%	104
	reg_2	2	0%	50%	19%	28
FUB_2	reg_1	1	48%	51%	49%	72
	reg_2	1	0%	72%	37%	114
FUB_3	reg_1	3	3%	32%	22%	32
	reg_2	2	13%	46%	29%	42
FUB_4	reg_1	2	0%	87%	30%	28
	reg_2	2	0%	51%	39%	35
FUB_5	reg_1	1	24%	67%	56%	128
	reg_2	2	8%	39%	35%	48

The FUBs and the registers chosen at random are listed in the left columns. Inferred minterms size is listed in the third column. The power reduction on the clock nets is summarized in the forth column, where the number of gated registers is in the fifth.

of the art designs and the ability to detect cross-FUB gating conditions. We have demonstrated conclusively that the algorithm can come with unit level hierarchy and also has the ability to detect cross-FUB gating.

Chapter 5

Conclusions and further work

In the previous chapters we introduced a novel method for extracting clock gating functions from simulation traces. A framework was defined and implemented in order to test the feasibility of learning power saving Boolean functions via machine learning and data mining techniques upon power benchmarks simulation traces of industrial size state of the art micro-processors designs.

We conclude upon the results, advantages and disadvantages of our work in Section 5.1 and assess what further work will be needed bearing in mind also possible future developments as may conceivably augment the current technological state of the art of Intelligate in Section 5.2.

5.1 Conclusions

The main hypothesis of this research was the ability to infer global clock gating conditions from power simulation benchmarks. This work has demonstrated conclusively that dynamic power saving functions can be extracted from the traces and then successfully applied for clock gating. The results indicate that the proposed method produces power beneficial clock gating conditions on large industrial designs, larger than any state of the art clock gating method can. We have also demonstrated the ability to reveal global correlations in the design, i.e., cross-FUB gating conditions. And cross-FUB analysis for clock gating is not feasible by current state of the art dynamic power management techniques.

The main advantages of the Intelligate method are 1) By using TDPs as our source of data and ignoring non-toggling signals, we can focus upon relevant parts of the design, look for clock gating opportunities and ignore irrelevant areas. 2) By filtering out signals which did not toggle during the simulation, we can apply our method on large designs which hither to were not in the scope of clock gating methods. 3) Global correlations like cross-FUB gating can be found. 4) By choosing different simulation traces, the user can focus upon different execution scenarios in order to save dynamic power there upon. For example if the user is interested in reducing the dynamic power of maximal power execution scenarios, then a simulation trace representing maximal power executions should be the data source for inference. 5) By manipulating the trace file the user can focus upon relations between specific design modules. For example, even only two FUBs in a simulation will be entirely adequate for the relation between them to be searched.

The most obvious disadvantage of the method is the need to apply formal verification

process on the inferred gating function, a task which is very complex when dealing with large designs. Another disadvantage is the lack of consideration in timing and placement constraints. Timing is a very important constraint to circuit design. Often, it will be crucial to proper function, for a given electrical signal to arrive in a specific time frame to its destination. And placement refers to the actual layout where the FUBs are positioned in the design; e.g an inferred gating function with signals too far apart is not power beneficial.

5.2 Further work

Although our method conclusively demonstrated to be feasible and power beneficial, there are still several areas to advance as future work. 1) Many signals and minterms can be pruned pursuant to timing and placement constraints. Adding these constraints will improve the algorithm's run time and increase the capacity of Intelligate. 2) In our work, all the design's signals have been taken into account. Applying the learning algorithm only in choosing the designs registers outputs can yield the same results and improve run time and capacity. 3) The proposed method can be applied to find correlations in different levels of design abstraction. The experiments reported herein were conducted on the RTL level only, whereas feasibility and efficiency as yet remain to be tested at the architectural and schematics levels as well. 4) By adding a clock gating function to a large number of clock nets, a new clock tree should be designed with less capacitance on its nets on order to reduce dynamic power, and therefore our power saving metric remains somewhat conservative and therefore actually inaccurate as to the full measure of power saving opportunity from the new methods.

Bibliography

[1] Agrawal R, Srikant R. *Fast Algorithms for Mining Association Rules.* VLDB. Sep 12-15 1994, Chile, 487-99.

[2] A. Michail. *Data Mining Library Reuse Patterns using Generalized Association Rules.* 22nd International Conference on Software Engineering (ICSE '00), pp.167.

[3] Anthony, Martin. *Learning boolean functions.* Centre for Discrete and Applicable Mathematics, LSE. CDAM-LSE-2005-24 (2005).

[4] A.P. Hurst, *Fast Synthesis of Clock Gates from Existing Logic.* Workshop on Logic Synthesis, 2007.

[5] Babighian, P. Kamhi, G. Vardi, M. *PowerQuest: trace driven data mining for power optimization.* Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07

[6] B.K. Natarajan. *On Learning Boolean Functions.* tech. report CMU-RI-TR-86-17, Robotics Institute, Carnegie Mellon University, December, 1986.

[7] B. Livshits, T. Zimmermann. *DynaMine: finding common error patterns by mining software revision histories.* Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005, pp. 296-305.

[8] Bshouty, N. H. and Burroughs, L. *Maximizing Agreements with One-Sided Error with Applications to Heuristic Learning.* Mach. Learn. 59, 1-2 (May. 2005), 99-123.

[9] C. Kern and M. Greenstreet. *Formal Verification in Hardware Design: A Survey.* ACM Transactions on Design Automation of E. Systems, Vol. 4, April 1999, pp. 123-193.

[10] Dasgupta, Pallab. *A Roadmap for Formal Property Verification.* 2006, XIII, 251 p., Hardcover,ISBN: 978-1-4020-4757-2.

[11] D. Larose. *Data Mining Methods and Models.* Wiley-IEEE Press, February 2006.

[12] F. Theeuven, E. Seelen. *Power reduction through clock gating by symbolic manipulation.* Symposium on Logic and Architecture Design, pp.184-191, Dec. 1996.

[13] G. Ammons, R. Bodik, and J. R. Larus. *Mining specifications.* POPL 2002: 4-16.

[14] Hangal, S., Chandra, N., Narayanan, S., and Chakravorty, S. *IODINE: a tool to automatically infer dynamic invariants for hardware designs.* In Proceedings of the 42nd Annual Conference on Design Automation (San Diego, California, USA, June 13 - 17, 2005). DAC '05. ACM Press, New York, NY.

[15] http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm

[16] http://en.wikipedia.org/wiki/Thermal_Design_Power

[17] http://en.wikipedia.org/wiki/De_Morgan%27s_laws.

[18] J. Kivinen. *Learning reliably and with one-sided error.* Theory of Computing Systems, vol 28, March 1995, 141-172.

[19] Kearns,M.J.,Vazirani,U.V. *An Introduction to Computational Learning Theory.* The MIT Press, 1994.

[20] K. Sadohara. *Learning of boolean functions using support vector machines.* In Proc. of the 12th International Conference on Algorithmic Learning Theory, pages 106-118. Springer, 2001. LNAI 2225.

[21] L. Benini, G. De Micheli, *Automatic synthesis of low-power gated-clock finite-state machines*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 15(6), Jun. 1996.

[22] L. Benini and G. De Michel, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1997.

[23] L.Benini, G. De Micheli, E.Macii, M. Poncino, R. Scarsi, *Symbolic Synthesis of clock-Gating Logic for Power Optimization of Control-Oriented Synchronous Networks*. Proceedings of the European Design & Test Conference, 1997, pp 514-520.

[24] L. G. Valiant. *A theory of the learnable.* Communications of the ACM, 27(11), November 1984.

[25] Maurer, P. M. *Dynamic Functional Testing for VLSI Circuits.* IEEE Des. Test 7, 6 (Nov. 1990), 42-49.

[26] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin. *Dynamically discovering likely program invariants to support program evolution.* IEEE Trans. On Soft. Eng. 27:2(2001), pp. 1-25.

[27] M.D. Ernst, A. Czeisler, W.G. Griswold, D. Notkin. *Quickly Detecting Relevant Program Invariants.* 22nd International Conference on Software Engineering (ICSE '00) pp. 449.

[28] Michael J. A. Berry, Gordon S. Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management, 2nd Edition.* Wiley-IEEE Press, February 2004.

[29] M. Ohnishi, A. Yamada, H. Noda, and T. Kambe. *A Method of Redundant Clocking Detection and Power Reduction at RT Level Design.* Proc. Int'l Symp. Low Power Electronics and Design (ISLPED '97), p. 131, Aug. 1997.

[30] P. Babighian, L. Benini, G. De Micheli. *A Scalable ODC-Based Algorithm for RTL Insertion of Gated Clocks.* In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (February 16 - 20, 2004). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 10500.

[31] T, Veerarajan. *Discrete Mathematics with Graph Theory and Combinatorics.* Tata McGraw-Hill, 2007.

[32] W. Qing, M. Pedram, W. Xunwei. *Clock-gating and its application to low power design of sequential circuits.* IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications, 47 (3), Mar. 2000.

[33] W. Wu, M.S. Hsiao. *Mining global constraints for improving bounded sequential equivalence checking.* Proceedings of the 43rd annual conference on Design automation, 2004, pp.743-748.

[34] Y. Mansour. *Learning Boolean Functions via the Fourier Transform.* Theoretical Advances in Neural Computation and Learning, Kluwer, 1994.

Appendix A

Candidates generation combinatorial analysis

This appendix describes the number of possible legal candidates to be generated in each iteration in process of the algorithm. Legal candidates are candidates such that every literal appears only once and literals and their negations can not appear together.

Theorem 5 *The number of possible candidates of size k out of $N = n + m$ literals, without literals repetition and literals with their negations in the same candidate is*

$$\sum_{i=0}^{min\{k,m\}} \frac{m!}{i!(m-i)!} \cdot \frac{(n-i)!}{(k-i)!(n-k)!} \tag{A-1}$$

Where m is the number of negated literals, which their complementary literals are in N.

Proof. the number of combinations of k objects you can choose from a set of N objects is:

$$\binom{N}{k} = \frac{N!}{k!(N-k)!} \tag{A-2}$$

When choosing k objects out of $n + m$ we must make sure a literal does not appear with its complementary literal in the same group. First we choose $i = 0$ to min{k,m} literals out of m; thus:

$$\binom{m}{i} = \frac{m!}{i!(m-i)!} \tag{A-3}$$

Then we need to choose $k - i$ literals out of the $n - i$ literals to reach a group of size k; thus:

$$\binom{n-i}{k-i} = \frac{(n-i)!}{(k-i)!((n-i)-(k-i))!} = \frac{(n-i)!}{(k-i)!(n-k)!} \tag{A-4}$$

Since the number of literals from m in each group can be min{k,m} we must sum all these possibilities:

$$\sum_{i=0}^{min\{k,m\}} \frac{m!}{i!(m-i)!} \cdot \frac{(n-i)!}{(k-i)!(n-k)!} \tag{A-5}$$

■

Appendix B

Gating logic dynamic power estimation

This appendix describes a gross estimation method to estimate the dynamic power consumption of the added gating logic.

The dynamic power consumption of a net is depicted by:

$$\frac{1}{2} \cdot C \cdot f \cdot Vdd^2 \cdot AF \tag{B-1}$$

Since the clock gating logic is composed of signals that were already present in the design, therefore these signals do not add any power consumption. The only dynamic power added is by the disjunctions and conjunctions implemented between the present signals. Each conjunction and disjunction between two signals adds a net to the design, therefore the added dynamic power by the gating logic is:

$$\sum_{i=1}^{N} \frac{1}{2} \cdot Ci \cdot f \cdot Vdd^2 \cdot AFi \tag{B-2}$$

where N is the number of new nets. The parameters frequency and voltage parameters are similar to these of the entire design. The activity factor of the new nets is computed from the trace according to its driven logic, where the capacitance is grossly and pessimistically estimated as the capacitance of the clock net.