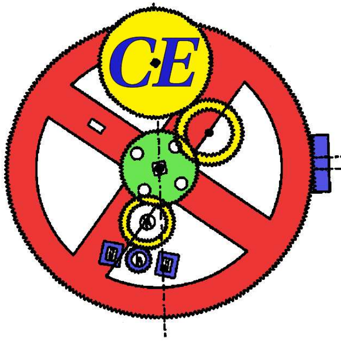


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2003

MSc THESIS

PROSA
Profiling-based State Assignment for Low Power Dissipation

Robbert Eggermont

CE-MS-2003-11

Abstract

In this thesis we address the problem of state assignment for finite state machines (FSMs). We target the reduction of power dissipation in FSM circuits by minimizing the switching activity in the state register. We introduce a novel method that utilizes dynamic loop information extracted from FSM profiling data. We propose three different loop-based state assignment algorithms, trading off quality for computational effort. The depth-first search (DFS) algorithm performs an exhaustive search of the FSM encoding space, using the loop information for intermediate cost estimates of an encoding. The loop-based DFS algorithm performs a similar search on a loop-by-loop basis, where the loops are ordered in descending order of weight. The heuristic algorithm encodes the states individually, on the same loop-by-loop basis. The algorithms have been implemented and evaluated on the standard FSM benchmark suite MCNC/LGSynth '89. Simulation results indicate an 8% average reduction of the switching activity in the state register for the heuristic algorithm when compared with POW3, a state of the art state assignment algorithms for low power dissipation. Additionally, our experiments suggest that no current state assignment algorithm that utilizes state register switching activity as metric for power minimization is able to achieve a consistent reduction in power consumption. Therefore, we conclude that the cost metric utilized for FSM state assignment algorithms for low power dissipation should be extended to also reflect the switching activity in the combinatorial circuit.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

PROSA

Profiling-based State Assignment for Low Power Dissipation

THESIS

submitted in partial fulfi llment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Robbert Eggermont
born in Amsterdam, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

PROSA
Profiling-based State Assignment for Low Power Dissipation

by Robbert Eggermont

Abstract

In this thesis we address the problem of state assignment for finite state machines (FSMs). We target the reduction of power dissipation in FSM circuits by minimizing the switching activity in the state register. We introduce a novel method that utilizes dynamic loop information extracted from FSM profiling data. We propose three different loop-based state assignment algorithms, trading off quality for computational effort. The depth-first search (DFS) algorithm performs an exhaustive search of the FSM encoding space, using the loop information for intermediate cost estimates of an encoding. The loop-based DFS algorithm performs a similar search on a loop-by-loop basis, where the loops are ordered in descending order of weight. The heuristic algorithm encodes the states individually, on the same loop-by-loop basis. The algorithms have been implemented and evaluated on the standard FSM benchmark suite MCNC/LGSynth '89. Simulation results indicate an 8% average reduction of the switching activity in the state register for the heuristic algorithm when compared with POW3, a state of the art state assignment algorithms for low power dissipation. Additionally, our experiments suggest that no current state assignment algorithm that utilizes state register switching activity as metric for power minimization is able to achieve a consistent reduction in power consumption. Therefore, we conclude that the cost metric utilized for FSM state assignment algorithms for low power dissipation should be extended to also reflect the switching activity in the combinatorial circuit.

Laboratory : Computer Engineering

Codenumber : CE-MS-2003-11

Committee Members :

Advisor: Sorin Cotofana, CE, TU Delft

Member: Ben Juurlink, CE, TU Delft

Member: Stephan Wong, CE, TU Delft

Contents

List of Figures v

List of Tables vii

Acknowledgments ix

1 Introduction 1
- 1.1 Research Questions . . . 1
- 1.2 Contributions . . . 2
- 1.3 Report Overview . . . 3

2 Background 5
- 2.1 Power Consumption in Digital CMOS . . . 5
- 2.2 Finite State Machines . . . 6
- 2.3 Design Approaches & Tools . . . 8
  - 2.3.1 Finite State Machine synthesis . . . 8
- 2.4 Terminology . . . 9
- 2.5 State of the Art in State Encoding . . . 10
  - 2.5.1 POW3 . . . 11
  - 2.5.2 Nöth-Kolla . . . 11

3 Profiling based FSM state assignment algorithms 13
- 3.1 Introduction . . . 13
- 3.2 General Approach . . . 13
- 3.3 FSM state profiling . . . 14
- 3.4 Loop detection . . . 15
  - 3.4.1 Examples . . . 16
- 3.5 Loop-based FSM state assignment algorithms . . . 20
  - 3.5.1 Basic FSM state assignment algorithm . . . 21
  - 3.5.2 Loop-based DFS state assignment algorithm . . . 24
  - 3.5.3 Loop-based heuristic state assignment algorithm . . . 26
  - 3.5.4 Optimized loop-based heuristic state assignment algorithm . . . 28
  - 3.5.5 Example . . . 30
- 3.6 Implementation . . . 32
  - 3.6.1 FSM data structures . . . 33
  - 3.6.2 Loop data structures . . . 33
  - 3.6.3 Functions . . . 33

4 Experimental Results **37**
4.1 Introduction 37
4.2 Method 37
4.2.1 Setup 38
4.2.2 FSM Profiling and State Assignment 40
4.2.3 Circuit Synthesis 40
4.2.4 Simulation 40
4.3 Benchmarks 41
4.4 Results 43
4.4.1 DFS 43
4.4.2 Loop-based DFS 45
4.4.3 Loop-based Heuristic 46
4.4.4 Profiling-based POW3 49
4.4.5 Nöth e.a. 50
4.4.6 Pow3 52
4.4.7 Jedi 53
4.4.8 Comparison 54

5 Conclusions **59**
5.1 Summary 59
5.2 Main contributions 61
5.3 Future work 62

Bibliography **63**

List of Figures

2.1 A CMOS inverter with current flows. 6
2.2 Finite State Machine 7

3.1 FSM state profiling and loop detection 14
3.2 FSM with sequential loops 17
3.3 FSM with nested loops 18
3.4 FSM with intersecting loops 19
3.5 FSM BBTAS: KISS description (left) and State Transition Graph 31
3.6 FSM data structures 33
3.7 Loop data structures 34

4.1 Experimental method 38
4.2 Setup step 39

List of Tables

3.1 Iterations of Algorithm 2 for sequential loops 17
3.2 Iterations of Algorithm 2 for nested loops 18
3.3 Iterations of Algorithm 2 for intersecting loops 19

4.1 Benchmarks Statistics 41
4.2 DFS average switching activity 43
4.3 Loop-based DFS average switching activity 45
4.4 Loop-based Heuristic average switching activity 46
4.5 Loop-based Dynamic Latch-allocation Heuristic average switching activity . . 48
4.6 Profiling-based Pow3 average switching activity 49
4.7 Nöth e.a. average switching activity 50
4.8 Pow3 average switching activity 52
4.9 Jedi average switching activity 53
4.10 Overall state register switching activity 55
4.11 Overall circuit switching activity 57

Acknowledgments

First of all, I would like to thank professor Stamatis Vassiliadis for providing me with the chance to graduate at the Computer Engineering department. Furthermore, I would like to thank my supervisor, professor Sorin Cotofana, for his patience, and acknowledge his efforts to successfully finish this project. Finally, I would like to acknowledge the understanding, support and advice I received from all CE members, my colleagues, friends, and last but not least, my family. Without you, this thesis would not have become what it is know.

Robbert Eggermont
Delft, The Netherlands
August 29, 2003

1 Introduction

With the increase in speed, mobility and miniaturization of current electronic products, the power consumption of these products has become a major design factor. Especially for mobile devices, the power consumption determines the battery life-time, the generated heat and the required heat dispersion measures. Therefore, the designers and consumers of electronic devices, as well as environmental considerations, demand a reduction in the power dissipation of digital circuits.

Digital circuits consists of a number of interconnected logic gates which together perform a function on one of more input signals. Every time an input signal changes, the change propagates via the gates through the circuit, causing signal switching activity in every place where the signal propagates to. This signal switching activity causes a current to charge or discharge the capacitive load of CMOS gates, which results in power dissipation. This power dissipation depends on the CMOS fabrication technology, operating frequency, but most of all on the switching activity per clock cycle within the digital circuit.

Current integrated circuits (ICs) are designed to perform a large number of complex functions. To ensure the correct (inter)operation of the functions, control logic is needed to manage the functions. This control logic is often implemented as a finite state machine (FSM), which keeps track of the "state" the IC is in using a state register. The (control) output of the FSM depends on the state it is in.

To implement an FSM in a digital circuit, the states of the FSM need to be assigned unique binary codes to represent the states in the state register. The encoding of the states determines the logic functions which operate on the state register, therefore the encoding determines the switching activity, and thus the power dissipation, in the FSM and the entire circuit.

This thesis addresses the state assignment of FSMs for low power dissipation. A novel state assignment approach is proposed and evaluated that uses FSM profiling data to assign states for lower switching activity.

1.1 Research Questions

With computer programs, in order to create faster programs, profiling is used to find the sections of code the program spends most of it's cycles in. Often this code will be one or more loops that gets executed repeatedly. The largest performance gain can be achieved by optimizing the instructions in the most executed loops.

Similarly, most FSMs are designed to run one or more fixed sequences of states over and over again, returning to a certain "default" state when one sequence is completed to wait for the next task. Therefore, these FSMs contain loops: one for each (sub)task. Along the lines of the program profiling above, one might theorize that the most effective way to optimize an FSM state assignment for low power dissipation is to perform an FSM profiling, and detect the loops that contribute the most to the FSM's power dissipation. Then, assign the states of those loops in a way that reduces the power dissipation of the FSM.

Current state assignment algorithms for low power dissipation use static FSM descriptions to perform the state assignment on. In this thesis we propose a novel method that utilizes dynamic information, extracted from FSM profiling data. Therefore, the first question we have to address is:

- What kind of loops are of interest, and how can we detect these loops?

As the main goal of our research is to find an FSM state assignment approach for low power dissipation, the main research question can be formulated as follows:

- How can we use the loop information to assign the states of an FSM most optimally in order to reduce the power dissipation?

And the final question to be answered is:

- How does the performance of our profiling, loop-based state assignment approach compare to that of current state of the art state assignment algorithms for low power dissipation?

1.2 Contributions

This report presents the results of our investigation related to the research questions stated in the previous section. In particular, the main contributions can be summarized as follows:

- We present a novel loop-based profiling FSM state assignment approach for low power dissipation.
- We propose a loop detection algorithm.
- We propose three loop-based FSM state assignment algorithms that minimizes the power dissipation of the FSM:
  - **DFS** performs an exhaustive search of all possible encodings of the FSM, and uses the loop data to estimate the cost of an encoding.
  - **Loop-based DFS** performs a similar search on a loop-by-loop basis, in descending-weight order of the loops.
  - **Heuristic** encodes the states indicidually, on a loop-by-loop basis in descending order of weight.

In order to evaluate the efficiency of our proposal we compared our approach with other state of the art FSM state assignment methods. Our experimental results indicate the following:

- For fixed width state registers, our heuristic state assignment approach shows an 8% reduction in average state register switching activity when compared to the power-based POW3 [1] algorithm, and a 41% reduction when compared to the area-based JEDI [4] algorithm.

- The variable state register width Nöth and Kolla algorithm [6], although it requires a larger state register, then more area, achieves a 6% reduction when compared with our fixed width heuristic. This suggests that state algorithms for low power dissipation should use a variable state register width approach to achieve the largest possible reduction in state register switching activity. Our preliminary **Dynamic Heuristic** is at a too early stage of development to be able to match the results of Nöth and Kolla's algorithm.
- Our experiments indicate that no current state assignment algorithm that utilizes state register switching activity as metric for power minimization is able to achieve a consistent reduction in power consumption. This clearly suggests that the switching activity in the state register only is not a suitable metric to reduce the power consumption in FSMs. Instead, a metric should be used that also reflects the switching activity in the combinatorial circuit.

1.3 Report Overview

This thesis is organized as follows:

- In Chapter 2 we present the general problem of state assignment for low power dissipation. We introduce the FSM and state assignment terminology and give a short overview of the state of the art state assignment algorithms.
- In Chapter 3 we present the profiling, loop detection and state assignment algorithms, as well as some examples.
- In Chapter 4 we describe the method used to compare the different algorithms, and compare our state assignment approaches to other current algorithms.
- In Chapter 5 we present the conclusions of our work described in this thesis.

2 Background

In the introduction we defined the main goal of our research: finite state machine (FSM) state assignment for low power dissipation. In this chapter, we first discuss the general considerations for power consumption in digital circuits. Next, we introduce the finite state machine, in theory and as a digital circuit, and the process of FSM state assignment. Following this, we describe the general FSM design approach from model to circuit, including the evaluation of an FSM's power dissipation. Then, we explain the terminology used in FSM state assignment. Finally, we discuss the history of state assignment algorithm, and we describe two state of the art state assignment algorithms for low power dissipation.

2.1 Power Consumption in Digital CMOS

Complimentary metal oxide semiconductor (CMOS) is at the moment the most widely used technology for the digital integrated circuits (ICs) that are present in all digital electronic equipment. One of the main reasons CMOS is the dominant logic style in use today is it's lack of static power consumption. CMOS logic cells consist of complementary NMOS and PMOS transistor pairs, which belong to the family of MOS field-effect transistors. These transistors have a very high input impedance ($> 10^{10}$ Ohm), which is capacitive. In steady state, CMOS has important advantages over other technologies:

- Because of the high input impedance, virtually no current is running from the output of one gate to the input of the next gate.
- Of the PMOS and NMOS transistor pairs between supply and ground, only one transistor of each pair is conducting. Thus, there is no conducting path between supply and ground, and there is no current flow.

Therefore, the static power consumption, caused by leakage currents related to the fabrication technology, is minimal ($\sim 1 - 2$nW/gate at 5V supply voltage).

In contrast to the minimal steady state power consumption, CMOS-technology can have a significant dynamic power consumption (as much as 100 Watt for a modern microprocessor with 42 million transistors). The dynamic power consumption of a CMOS-based digital IC is made up of three main components, as shown in the following equation:

$$P_{dynamic} = P_{switching} + P_{short-circuit} + P_{leakage} \tag{2.1}$$

Figure 2.1 shows a CMOS inverter with the current flows that result in these power consumption components:

Switching or capacitive power. $P_{switching}$ represents the signal-switching related component of the power consumption. This power consumption is caused by the current I_{sw} needed

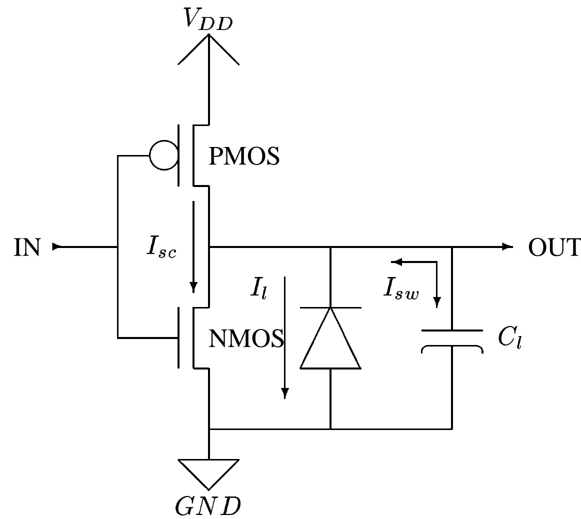


Figure 2.1: A CMOS inverter with current flows.

to charge or discharge the capacitive load C_l at the output of a cell whenever a signal transition (0 to 1 or 1 to 0) occurs. These signal transitions generate switching activity in the circuit. $P_{switching}$ is proportional to the switching activity in one clock period, the capacitance, the voltage swing, the supply voltage and the clock frequency. The capacitive load is made up of the parasitic gate, diffusion and interconnect capacitances related to the CMOS-technology. The switching power dissipation accounts for roughly 90% of the overall power consumption in most CMOS circuits.

Short-circuit power. $P_{short-circuit}$ is the result of the supply to ground short-circuit current I_{sc} flowing when PMOS and NMOS transistors are both shortly conducting during an output transition, and accounts for about 10% of the power consumption.

Leakage power. $P_{leakage}$ is the power dissipation due to leakage currents I_l, which consist of reverse bias diode currents and sub-threshold effects related to the fabrication technology. This represents less than one per cent of the power consumption.

In CMOS ICs, signal activity causes both switching and short-circuit power dissipation and is one of the main sources of power consumption. In digital circuit design, the signal activity in the circuit is often used as a measure for the power consumption. Conversely, the power dissipation of a circuit depends on the signal activity within the circuit. Therefore, in theory, a reduction of the switching activity will lead to a reduction in the power dissipation. This theory is often used in circuit design to reduce power consumption, and it is the method we will use to obtain lower power dissipation for finite state machines.

2.2 Finite State Machines

Digital circuits can generally be divided into two groups: combinatorial circuits and sequential circuits. In steady state, the output of combinatorial logic is defined completely by the current

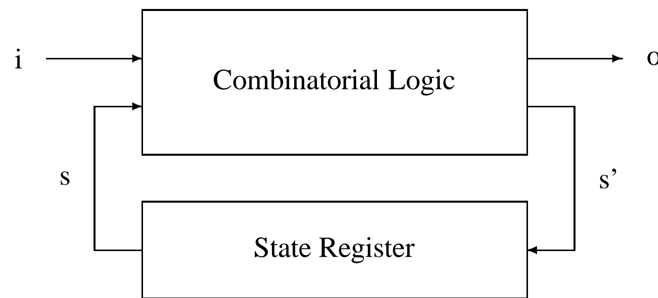


Figure 2.2: Finite State Machine

input. No combination of input signals can have more than one resulting combination of output signals. Sequential circuits, on the other hand, have at least one combination of input signals that has more than one possible combination of output signals. The output is not defined solely by the current input, but also by the "state" the circuit is in. The state of the sequential circuit is determined by the state of one or more latches, also known as a register. Sequential circuits are ideal for control functions, and are found in many digital circuits, from traffic light-controllers to microprocessors.

One way to describe sequential circuits is a Finite State Machine (FSM) model. An FSM is a computational model consisting of a (finite) set of states S, a set of input vectors I, a set of output vectors O, and two functions f_s and f_o. Each FSM state has zero or more transitions to itself or other states. The transition function $f_s : (S, I) \rightarrow (S)$ maps the current state s and input vector i to the next state s'. The output transition function maps the current state (Moore machine; $f_o : (S) \rightarrow (O)$) or the transition (Mealy machine; $f_o : (S, I) \rightarrow (O)$) to the output vector o. A special case of the Moore machine, where the output vector resembles the state vector, is called the Medvedev machine.

An FSM sequential circuit consists of two distinct, but strongly related, parts (see Figure 2.2):

- the state register, which defines the current state the circuit is in, and
- the combinatorial logic, that computes the next state and output vector based on the current state and input vector according to $f1$ and $f2$.

When the FSM changes state, or the input vector changes, the combinatorial logic computes the new state and output vector. On the next clock-cycle, the state register stores the new state value.

Each state of the FSM needs to have a unique representation in the state register in order for the combinatorial logic to be able to determine the correct state transition. For a binary circuit, the number of states $\#states$ in the FSM forces the width, or the number of bits $\#bits$, of the state register, to be at least $\#bits = \lceil log_2\ \#states \rceil$. The maximum width of the state register is only limited by the available chip area. The available number of unique codes $\#codes$ for the states is $\#codes = 2^{\#bits}$. A state can be assigned every code, as long as the assignment is unique. If $\#codes$ is larger than $\#states$, some codes will be left unused.

The binary codes of the states determine the structure of the combinatorial logic. Thus, the state assignment influences the area requirements (circuit size) and power consumption (switching activity) of the resulting circuit. Ideally, both area requirements and power consumption

should be minimized. However, the size of the logic and its switching activity are related, so a trade off is needed. For low-power designs, the state assignment algorithms minimize the switching activity.

In this thesis, we consider the minimization of the power consumption only. More in particular, we investigate FSM state assignment algorithms that utilize the switching activity within the state register as a measure of the entire FSM power consumption.

2.3 Design Approaches & Tools

Current integrated circuits consists of millions of transistors wired together to form digital function. These circuits are so complex, that it is no longer possible to design them solely on transistor, or even on (logic) gate level. Instead, these circuits are designed using Computer-Aided Design (CAD) tools, which synthesize the circuit from a high-level description. A typical design trajectory for an integrated circuit usually follows the following path:

High-level description The designer uses a hardware description language (HDL) to model the structure or behavior of a circuit:

- Combinatorial logic can be expressed as truth tables or logic equations of Boolean functions.
- Sequential circuits can be described as finite-state machines or state transition graphs.

Logic synthesis The HDL model is transformed into a gate-level implementation.

Logic simulation The gate-level implementation is simulated to verify the correctness of the digital model.

Layout The gates are placed onto a chip-layout, and connections are routed.

Layout simulation The circuit is extracted from the chip-layout, and simulated to verify the working of the actual implementation.

The purpose of this project is to devise an FSM state assignment algorithm, which takes an FSM description and assigns binary codes to every possible state of the FSM. This places our algorithm between the high-level description and the logic synthesis. However, to determine the power dissipation of the FSM encoding, we need to synthesize and simulate the gate-level (logic) implementation. For the logic synthesis of the FSMs we use the existing synthesis system SIS [2]. The implementation is then simulated using the MERCURY simulator from the Stanford Olympus Synthesis System [5] to determine the switching activity.

2.3.1 Finite State Machine synthesis

The synthesis system transforms a high level FSM description in several steps into a gate-level implementation. During this transformation the system applies optimizations according to a cost-function specified by the designer, such as chip area, latency or power consumption. The synthesis path for an FSM typically consists of the following steps:

- State assignment: the symbolic states of the FSM description are assigned binary codes.
- Logic synthesis: the FSM description is translated into logic functions for the state transitions and the output signals.
- Technology mapping: the logic functions are rewritten to use only the logic gates available to the target chip technology.

The SIS synthesis system incorporates the JEDI [4] and NOVA [8] state assignment programs, logic synthesis tools, technology mapping including several example libraries of gates, and a large number of optimization routines. For the experiments, our state assignment approach will perform the first step, and SIS finishes the synthesis.

2.4 Terminology

Before presenting the state assignment algorithms, we briefly describe the terminology utilized in the description of the state assignment algorithms.

State register The set of latches whose values determine the FSM state in a logic circuit. Each latch can have two possible values, high logic level (one) or low logic level (zero).

FSM state assignment The process which assigns to each state a unique binary code to represent that state in the state register. The code is the logic level equivalent of the state's name or number in the FSM model description. Each latch corresponds to one bit, so the number of bits in a state's code must match the number of latches in the state register.

Encoding A set of codes for all FSM states. An FSM has many different possible encodings.

Hamming distance The number of bits that differ between the codes c_i and c_j, corresponding to the states s_i and s_j: $H(s_i, s_j) = Count(c_i \oplus c_j)$.

Switching activity The level switching of signals within a logic circuit, triggered by a change in the input vector or state register. The state register switching activity reflects the bits that differ between successive states, and can therefore be calculated using the Hamming distance between the states.

Cost metric The measure utilized by the state assignment algorithm to determine the success of a state assignment. In our FSM state assignment approach the cost metric is (an estimate of) the state register switching activity based upon the FSM profiling data. Lower switching activity equals lower cost, thus a better solution.

FSM state profiling The process of collecting a state register trace from the FSM during a run with a relevant input vector data set, consisting of a sequence of input vectors.

State register trace The sequence of states that the FSM has been in during the FSM state profiling.

Loop A sequence of states within the state trace that forms a cycle, i.e., at the end of the sequence of states the FSM returns to the first state of the sequence. A loop has the following properties:

- it consists of multiple (different) states,
- it can be entered through any of the states,
- it has a specific order in which the states occur, and
- it is exited from the same state through which it was entered.

Simple loop A sequence of states containing only one loop, i.e., each state occurs only once within the sequence.

Nested loop A sequence of states featuring one or more inner loops within one or more outer loops. With nested loops, the FSM first enters the outer loop. At a certain state in the outer loop, the FSM enters an inner loop. When the inner loop is completed, the FSM returns to the same state in the outer loop, and completes the outer loop. The inner and outer loops are separate entities, i.e., when the inner loop is removed from the state trace, the outer loop does not change.

Frequency The number of occurrences of states, transitions, or loops in the trace.

Loop weight A value specifying the impact of the state assignment of the states in that loop on the cost of the complete FSM assignment. In our state assignment algorithms, the weight depends on the loop's frequency, and in some cases the number of states in the loop.

2.5 State of the Art in State Encoding

In the beginning of ICs, FSMs were small, i.e., they had few states. Computers where hard to come by, costly and very slow, therefore state assignment was done by hand. One of earliest algorithms used was the One Hot encoding, where each state code has exactly one high (hot) bit. Consequently, the size of the state register needs to be equal to the number of states. Because every two state encodings differ in exactly two places, the ones, One Hot encoding guarantees a fixed two bit switching activity in the state register for each transition. Although this method is favorable for an FSM's power consumption, the area required by the state register prohibits it use for the large real-life FSMs used nowadays.

As the FSMs grew larger and computers became available to synthesize the combinatorial logic, the main concern became the area that the FSM circuit occupied on chip. Algorithms like JEDI [4] and NOVA [8] assign codes with the least possible number of bits (and thus the smallest state register) in such a way that the combinatorial logic network is minimized. Furthermore, the algorithms are able to reduce the switching activity to one bit for some of the transitions.

Recently, IC power dissipation has become a concern, while logic area has become less of a problem due to smaller transistor sizes. Therefore, current state assignment algorithms attempt to minimize the switching activity, and thus reduce the power dissipation, in the circuit. The ideal result is an FSM encoding whereby, for each state transition to every possible state, only one bit switches in the state register. Such an encoding is called a Gray code. However, most times such a solution is not possible, and advanced algorithms try to find a solution with the most optimal encoding, in terms of switching activity, possible. Up to now, most sophisticated state assignment algorithms utilize static state transition probabilities to target switching activity. Two such algorithms have been proposed: POW3 [1] and Nöth and Kolla's Spanning Tree Based algorithm [6].

2.5.1 POW3

The POW3 [1] state assignment algorithm for low power dissipation, by Luca Benini and G. De Michelli, targets the reduction of switching activity in the state register during state transitions. The algorithm utilizes a probabilistic description of the FSMs, and minimizes the Boolean distance between the codes of states with a high transition probability. The Greedy heuristic algorithm assigns state codes bit by bit, and attempts to give states with a high transition probability the same bit value. If one of the states' bit is already assigned, the other state will receive the same bit value. The algorithm takes into account the constraints on bit assignments posed by the requirement for unique state codes. The heuristic uses a cost function based on the weighted sum of the Hamming distance between state codes.

2.5.2 Nöth-Kolla

Winfried Nöth and Reiner Kolla [6] propose a spanning tree based state encoding which also uses state transition possibilities. The FSM state transition graph is transformed into an undirected graph, and each edge is assigned a weight corresponding to the state transition probabilities. Using a modified version of Prim's algorithm [7], a maximum spanning tree is constructed from this graph. The state assignment problem is formulated as an embedding of the spanning tree into a Boolean hypercube. Two algorithms are proposed, a fast embedding algorithm, and a Greedy embedding algorithm.

The fast embedding algorithm chooses an edge which separates the tree into two evenly sized subtrees. The edge is mapped onto an edge of the hypercube, and the states that are connected through that edge are mapped to the corresponding nodes. Then, the algorithm recursively process the two subtrees. When all states are assigned to nodes, the nodes are assigned codes in such a way, that the codes of two nodes that are connected by an edge will differ in exactly one bit position.

The Greedy embedding algorithm expands the fast algorithm by a Greedy selection procedure for the hypercube edge to which a graph edge is mapped. For this, the algorithm takes into account the nodes that are already assigned. If one of the states of the edge is already assigned to a node, the algorithm calculates the cost for the assignment of the unassigned state to a node on the other side of a free edge. This cost is a function from the Hamming distance and transition probability of all assigned edges connected to the free node. The state is assigned to the node with the lowest cost.

The state assignment approach we propose resembles the current state assignment algorithms in the fact that it uses a cost function based upon the Hamming distance between connected states. However, the approach differs from the current state assignment algorithms in the fact that it utilizes profiling info and does not rely on state transition probabilities. The next chapter presents our proposed state assignment approach in detail.

Profi ling based FSM state assignment algorithms

3

3.1 Introduction

Until now, finite state machine (FSM) state assignment for low power dissipation was mostly based on the static FSM description (POW3[1], Nöth et.al.[6], JEDI[4], NOVA[8]). This does not take into consideration an important aspect of the behavior of an FSM, the interaction between the FSM and the outside world.

In this chapter we propose an FSM state assignment approach based on dynamic FSM state profiling. The profiling data allows us to identify the most frequently executed states or sequences of states within the FSM. More specifically, our approach targets frequently executed cycles of states, or loops. We introduce several state assignment algorithms that utilize this profiling data to minimize the power dissipation of the FSMs.

The outline of the chapter is as follows: We start by describing the general approach, and an explanation of the utilized terminology. Section 3.3 discusses the concept and implementation of FSM state profiling, followed by the proposed loop detection algorithm in Section 3.4. Finally, in Section 3.5 we present a number of state assignment heuristics.

3.2 General Approach

We present a general approach to FSM state assignment based on dynamic FSM state profiling data. The main goal of this method is to minimize the switching activity within the FSM state register in an attempt to lower power dissipation. This approach is based on the idea that the operation of many FSMs consists of recurring cycles (loops) of the same states, and the FSMs spend most of their time walking through these loops. Therefore, the state assignments of the states in these loops have the largest impact on the overall switching activity in the FSM state register, and by targeting these loops with our state assignment algorithm the largest reduction in switching activity can potentially be realized.

The approach can be divided into the following three steps:

1. **FSM state profiling** collects information about the dynamic behavior of the FSM. A (simulated) FSM run under a relevant input data set generates an FSM state register trace, and from this trace, state and transition statistics are collected.

2. **Loop detection** searches for loops in the state trace. Loops are identified by the repeated occurrence of the same state in the trace, and each discovered loop is stored and counted to obtain the frequency of the loops.

3. **FSM state assignment** assigns each state of the FSM a unique code (bit vector) to represent it in the state register. The data gathered in the first two steps are utilized to minimize the switching activity in the FSM state register.

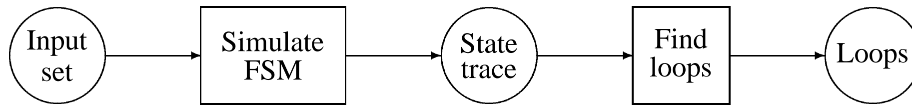


Figure 3.1: FSM state profiling and loop detection

A successful execution of these steps requires an FSM description and a relevant input vector data set and results in a valid encoding for all states of the FSM, which attempts to minimize the switching activity of the FSM *for the given input data set*. The rest of this chapter describes the different steps in detail.

3.3 FSM state profiling

For programs, code profiling means determining how often certain pieces of code are executed. We define FSM state profiling in the same way: state profiling determines how often a certain state is entered during an FSM run. FSM state profiling (Figure 3.1) records the state of the FSM during a (simulated) run of the FSM with a relevant input data set. From the resulting state register trace the following statistics can be derived:

- **state** frequencies, which are a measure for the impact of a state's assignment on the FSM encoding,
- **transition** frequencies, which specify the influence of a transition on the switching activity in the state register, and
- **loop** frequencies, which determine the importance of a loop for the switching activity.

State and state transition information is directly available in the state trace, therefore state and transition frequencies are obtained simply by counting the occurrences of states and transitions in the state trace. Loop information however lies hidden within the state trace and thus requires additional analysis (see Section 3.4).

Algorithm 1 FSM state profiling

```
state = ResetState(InputDataSet, FSM)
trace → Add(state)
for each vector in InputDataSet do
  transition = state → Transition(vector)
  if transition then
    state = transition → NextState()
    trace → Add(state)
  end if
end for
```

Algorithm 1 describes the FSM state profiler. It requires an FSM description and an Input Data Set containing a sequence of input vectors for the FSM. The initial state of the FSM is specified by the reset state of the FSM, but can be overridden to match the initial state for the Input Data Set.

The algorithm simulates the FSM run by repeatedly matching an input vector to the possible transitions of the state the FSM is in. If a match is found, the FSM state is changed to the destination state of the transition, and that state is added to the state trace. If no match is found, the FSM is assumed to remain in the same state.

For the cause of reducing the switching activity of the FSM state register, the operations that do not cause a state change are irrelevant, therefore repeated entries are omitted from the state trace.

3.4 Loop detection

FSMs consists of a finite number of states, between which the FSM switches during its operation. Unless the FSM enters a state from which no state transitions to other states are possible, it is very likely that the FSM at some time will enter a certain state for the second time. The FSM has no memory of its previous states, thus for the FSM there is no difference between the first and the second time the state was entered, and the FSM has in effect looped back to this state. We call the cycle, formed by the sequence of states from the first occurrence of a state (up) to the second occurrence of that same state, an FSM state loop.

FSM loops can be nested, i.e., an outer loop can contain one or more inner loops. Several strategies can be followed to count nested loops:

1. count the complete set of nested loops as one, or
2. count every subset of the outermost loop that is itself a loop, or
3. count each simple (inner or outer) loop separately.

Our FSM state assignment approach attempts to lower the FSM's power dissipation by reducing the state register switching activity. Conflicting state sequences between loops inhibit an optimal encoding of all loops, therefore the best results are obtained by assuring that at least the loops that contribute the most to the overall state register switching activity are encoded optimally. This contribution, or weight, is a function of the loop's frequency. Thus, our approach is most successful for FSMs that feature a small number of loops with a significantly higher frequency than the other loops.

The first strategy counts only the sets of nested loops. However, unless the inner loops are executed in exactly the same way every time the outermost loop is executed, most detected loops will not be duplicates. Therefore this strategy most likely finds only a large number of low frequency loops, which makes it unsuitable for our approach.

The second and the third strategy count all loops separately, whereby the second strategy counts loops both separately, and in all possible nested forms. The most probable highest frequency loop is a simple loop, because a simple loop occurs at least as often as any nested loop it is a part of. If a simple loop occurs in more than one nested loop, the simple loop will have the combined frequency of the nested loops. Both strategies will correctly find all simple loops, and are therefore functionally equivalent for simple loops. If the loop with the highest frequency is a nested loop, the third strategy will only find the simple loops it consists of. However, these simple loops will have the same frequency as the nested loop. During the state assignment, these simple loops will all be assigned before any lower frequency loops, just like the nested loop.

Given that the nested loops detection provides no advantage to the state assignment algorithms, we choose the less complex, third, strategy for loop detection.

Algorithm 2 Detect Loops

```
for each state in trace do
  if minimal_trace → Present(state) then
    loop = minimal_trace → RemoveLoop(state)
    loops → Add(loop)
  end if
  minimal_trace → Add(state)
end for
```

We propose the loop detection algorithm described in Algorithm 2. This algorithm finds loops by detecting the recurrence of states in the state trace. The loop detection algorithm takes a linear search approach, i.e., it performs a single analysis of the state trace, in one direction.

The algorithm utilizes its own internal memory, called the minimal trace, to store a list of the states it encounters in the state trace. Before a state is added to the list, the algorithm performs a simple check for the presence of the encountered state in the minimal trace, which indicates the presence of a loop. The order of the state in the minimal trace matches the order of the states in the state trace, therefore the algorithm can determine the states in the loop, and the order of those states, from the minimal trace. When a loop is detected, the states in the loop are removed from the minimal trace, and the loop is added to the set of detected loops. If there is already a loop present that matches the states and the order of the states in the detect loop, the frequency count for that loop is simply incremented instead.

The removal of the detected loop serves an important purpose: removing the detected loop in effect removes the innermost loop from a set of nested loops within the state trace. When the loop is removed, the last state in memory is the last state of the outer loop before entering the innermost loop. The algorithm continues to add states to the memory until the outer loop is detected. This loop does not contain an inner loop and is counted as a separate, simple loop (hence the name *minimal* trace). This process continues until all loops of the nested set are detected.

The last step of each iterations adds the encountered state to the minimal trace, even when the state is part of a detected loop, because the state that joins an inner and an outer loop is an indispensable part of both loops, and the previous occurrence of the state has been removed from the memory.

To clarify the way the proposed algorithm is working we present in the following subsection a number of examples.

3.4.1 Examples

Assume the FSM in Figure 3.2 produced the following state trace: **B** → **a1** → **a2** → **a3** → **a4** → **a1** → **a2** → **b1** → **b2** → **b3** → **b4** → **b1** → **b2** → **E**. The trace contains two separate state loops: **a1** → **a2** → **a3** → **a4** and **b1** → **b2** → **b3** → **b4**.

Table 3.1 demonstrates the iterations of Algorithm 2 for this state trace. For each step it shows the state being checked, the minimal trace after the step, and if present, the detected loop. A loop is detected when the current state is found present in the minimal trace (indicated in

bold). The loop is removed from the minimal trace, and added to the set of loops. After the check, the current state is added to the minimal trace.

Table 3.1: Iterations of Algorithm 2 for sequential loops

State	Minimal trace	Detected loop
B	B	
a1	B → a1	
a2	B → a1 → a2	
a3	B → a1 → a2 → a3	
a4	B → **a1** → a2 → a3 → a4	
a1	B → a1	a1 → a2 → a3 → a4
a2	B → a1 → a2	
b1	B → a1 → a2 → b1	
b2	B → a1 → a2 → b1 → b2	
b3	B → a1 → a2 → b1 → b2 → b3	
b4	B → a1 → a2 → **b1** → b2 → b3 → b4	
b1	B → a1 → a2 → b1	b1 → b2 → b3 → b4
b2	B → a1 → a2 → b1 → b2	
E	B → a1 → a2 → b1 → b2 → E	

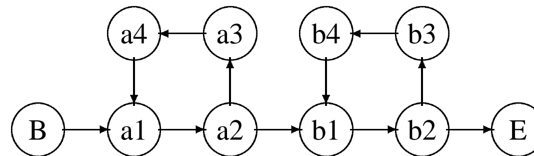


Figure 3.2: FSM with sequential loops

For the second example, consider this state trace from the FSM in Figure 3.3: **B** → **a1** → **b1** → **b2** → **b3** → **b4** → **b1** → **b2** → **b3** → **b4** → **b1** → **b2** → **a2** → **a3** → **a4** → **a1** → **b1** → **b2** → **a2** → **E**. It contains two nested loops: the inner loop **b1** → **b2** → **b3** → **b4** (twice) and the outer loop **a1** → **b1** → **b2** → **a2** → **a3** → **a4**. This example show that loops can be nested. In the state trace, the states of inner loops always lie between the begin and end state of the outer loops, thus the inner loops are detected first. After the inner loops are removed from the minimal trace, the algorithm correctly detects the outer loop. Nested loops must be detected separately to obtain accurate loop frequencies, as loops can also occur separately outside the nested construction.

Table 3.2: Iterations of Algorithm 2 for nested loops

State	Minimal trace	Detected loop
B	B	
a1	B → a1	
b1	B → a1 → b1	
b2	B → a1 → b1 → b2	
b3	B → a1 → b1 → b2 → b3	
b4	B → a1 → **b1** → b2 → b3 → b4	
b1	B → a1 → b1	b1 → b2 → b3 → b4
b2	B → a1 → b1 → b2	
b3	B → a1 → b1 → b2 → b3	
b4	B → a1 → **b1** → b2 → b3 → b4	
b1	B → a1 → b1	b1 → b2 → b3 → b4 (2nd)
b2	B → a1 → b1 → b2	
a2	B → a1 → b1 → b2 → a2	
a3	B → a1 → b1 → b2 → a2 → a3	
a4	B → **a1** → b1 → b2 → a2 → a3 → a4	
a1	B → a1	a1 → b1 → b2 → a2 → a3 → a4
b1	B → a1 → b1	
b2	B → a1 → b1 → b2	
a2	B → a1 → b1 → b2 → a2	
E	B → a1 → b1 → b2 → a2 → E	

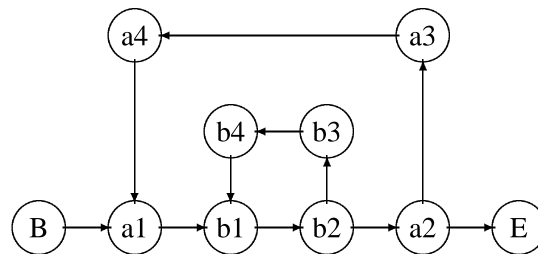


Figure 3.3: FSM with nested loops

Table 3.2 illustrates that Algorithm 2 first finds the second and third occurrence of state **b1**, indicating the inner loop. Then, the search detects the second occurrence of state **a1**, marking the outer loop.

The last example involves an FSM (Figure 3.4) state trace featuring intersecting loops: **B** → **a1** → **b4** → **b1** → **b2** → **b3** → **b4** → **b5**→ **b6** → **b1** → **b2** → **b3** → **b4** → **b5**→ **b6** → **b1** → **a2** → **a3** → **a4** → **a1** → **b4** → **b1** → **a2** → **E**. The inner loop **b1** → **b2** → **b3** → **b4** → **b5** → **b6** appears twice, the outer loop **a1** → **b4** → **b1** → **a2** → **a3** → **a4** once.

Table 3.3: Iterations of Algorithm 2 for intersecting loops

State	Minimal trace	Detected loop
B	B	
a1	B → a1	
b4	B → a1 → b4	
b1	B → a1 → b4 → b1	
b2	B → a1 → b4 → b1 → b2	
b3	B → a1 → **b4** → b1 → b2 → b3	
b4	B → a1 → b4	b4 → b1 → b2 → b3
b5	B → a1 → b4 → b5	
b6	B → a1 → b4 → b5 → b6	
b1	B → a1 → b4 → b5 → b6 → b1	
b2	B → a1 → b4 → b5 → b6 → b1 → b2	
b3	B → a1 → **b4** → b5 → b6 → b1 → b2 → b3	
b4	B → a1 → b4	b4 → b5 → b6 →
b5	B → a1 → b4 → b5	b1 → b2 → b3
b6	B → a1 → b4 → b5 → b6	
b1	B → a1 → b4 → b5 → b6 → b1	
a2	B → a1 → b4 → b5 → b6 → b1 → a2	
a3	B → a1 → b4 → b5 → b6 → b1 → a2 → a3	
a4	B → **a1** → b4 → b5 → b6 → b1 → a2 → a3 → a4	
a1	B → a1	a1 → b4 → b5 → b6 →
b4	B → a1 → b4	b1 → a2 → a3 → a4
b1	B → a1 → b4 → b1	
a2	B → a1 → b4 → b1 → a2	
E	B → a1 → b4 → b1 → a2 → E	

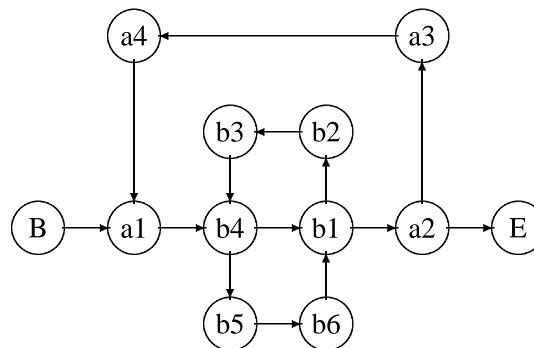


Figure 3.4: FSM with intersecting loops

Table 3.3 presents the iterations of the algorithm. It detects these three loops: **b4** → **b1** → **b2** → **b3**, **b4** → **b5** → **b6** → **b1** → **b2** → **b3** (which matches the inner loop) and **a1** → **b4** → **b5**→

b6 → **b1** → **a2** → **a3** → **a4**. Although they differ from the previously specified inner and outer loops, these are all valid loops, and no loops are left undetected in the minimal trace. Therefore, this example shows that for some state traces more than one result is possible.

3.5 Loop-based FSM state assignment algorithms

The object of a state assignment algorithm is to assign a unique code (state register bit vector) to every state in the FSM. There are many ways to perform an FSM state assignment, the easiest (and probably the fastest) one just assigns an increasing binary number to the states of the FSM (in no particular order). While this approach results in a valid FSM state assignment, the encoding will not be optimized for any target (except for ease of assignment possibly). We propose several loop-based state assignment algorithms that specifically target the reduction of FSM power dissipation.

A state assignment algorithm has a significant influence on the power dissipation of the resulting FSM circuit. Through the state codes, the encoding determines not only the switching activity in the state register, but also the structure of the combinatorial logic circuit of the FSM and its switching activity. Ideally, a state assignment algorithm for low power dissipation should consider both state register and combinatorial circuit switch activity when searching for a low power state assignment. The state register switching activity can be determined by evaluating the (known) FSM's state transitions, which renders it possible for the algorithm to evaluate the cost of each choice for the state register switching activity. However, to be able to evaluate the combinatorial circuit switching activity, the FSM needs to be synthesized. Synthesis is a complex, and time consuming, process, which makes it unfeasible to evaluate the cost of each choice for the circuit switching activity. Therefore, low power state assignment algorithms consider only the state register switching activity.

Under our assumptions, an optimal solution of the state assignment problem is a solution which results in the minimal amount of switching activity in the state register. A zero bit change in the state register leaves the FSM in the same state, therefore the minimal amount of switching activity for one transition is one bit change in the state register. This requires that the codes of two subsequent states differ in only one bit position, i.e., the states must have a Hamming distance of one. Therefore, the optimal state assignment solution is a state assignment for which the Hamming distance for all possible state transitions is one. However, it is clear that this result cannot be obtained if an FSM with more than two states contains a fully connected state, i.e., a state with a transition to each other state. The same is true for most FSMs with a normal level of connectivity.

The FSM state assignment problem has a computational complexity that is exponential in the number of states $\#s$ of the FSM: the best solution can only be determined by trying every combination of possible codes for the states of the FSM. The number of possible codes $\#c$ is determined by the width, or number of bits $\#b$, of the state register: $\#c = 2^{\#b}$. To minimize the complexity of the assignment, the number of possible codes should be chosen as the smallest power of two needed to assign each state a unique code: $\#c = 2^{\lceil \log_2 \#s \rceil}$. The number of possible FSM encodings $\#e$ then becomes:

$$\#e = \frac{\#c!}{(\#c - \#s)!}. \tag{3.1}$$

For FSMs with a large number of states, the exponential complexity makes it unfeasible to try every possible encoding even with a minimal number of possible encodings. Therefore, we present several algorithms that evolve from an exhaustive search exponential computational complexity algorithm to a linear complexity state assignment heuristic featuring a "best guess" approach. Although the heuristic cannot guarantee an optimal solution, the computational complexity of the heuristic allows large size FSMs to be assigned. Even more, the computational complexity allows several runs of the heuristics, for example to compare different loop weighting strategies, and choose the best solution.

The FSM state assignment algorithms we propose are based on the loops detected in the state trace. Each loop is assigned a weight that represents the significance of the loop's state assignments on the overall switching activity of the state register for the state trace. By assigning the states loop by loop in descending loop weight order, the loops which contribute most to the state register switching activity are most optimally assigned, and the state assignment algorithm can achieve the largest reduction in switching activity.

First, we describe the basic FSM state assignment approach. This algorithm tries every possible combination of state codes, and assigns the combination for which the cost is the lowest. The cost metric is the switching activity of the state register for the provided profiling data. This algorithm guarantees the best state assignment solution, and can thus be utilized to evaluate the efficiency of other state assignment algorithms for low power dissipation, including the other algorithms described here. This algorithm has an exponential complexity, therefore it is only suited for FSMs with very few states. To alleviate this problem we present several enhancements which expand the workable range.

A well-known solution to the exponential computation time problem is to divide the problem into smaller problems which can be solved much faster. The loops found during profiling provide a partitioning of the states of the FSM which lends itself naturally to the minimization of the switching activity of the state register. Therefore, we propose a loop-based state assignment algorithm which processes loops in a serial manner in the order of decreasing weight of the loops. During the assignment of a loop, the algorithm respects the codes of states that where already assigned in previous loops, but no backtracking will be performed. Therefore, only the first loop to be assigned is guaranteed to be assigned optimal.

In order to assign FSMs with a large number of states, a solution with a linear complexity is required. The last algorithm we present does not perform a search of the solution space, but instead chooses (for each state) a code that is optimal considering the previous assignments. The states are assigned in the order in which they occur in the loops, which are sorted by decreasing weight. As with the previous algorithm, the complete FSM's state assignment is not guaranteed to be optimally, only the first loop's state assignment is.

3.5.1 Basic FSM state assignment algorithm

The basic FSM state assignment algorithm finds the FSM state assignment solution with the lowest switching activity by trying every possible combination of state codes for the states. The algorithm performs a depth-first search (DFS) in a search tree, where each state is represented by a level, each node of a level corresponds to an unassigned code, and each incoming branch to a node symbolizes a state assignment. The top level (zero) represents the unassigned FSM. When the search reaches the bottom level, the resulting search path corresponds to a valid encoding for

the FSM.

To find the best state encoding for low power consumption, the algorithm uses the state register switching activity as its cost metric, and minimizes this. The cost of an encoding is an estimate of the state register switching activity defined as:

$$cost = \sum_{l \in \text{loops}} \left(\text{frequency}(l) \times \sum_{t \in \text{T}_l} \text{H}(t)\right), \tag{3.2}$$

where T_l is the set of transitions in loop l, and $H(t)$ is the Hamming distance between the codes of the begin and end state of transition t. Therefore, the complexity of the cost estimate is linearly related to the combined number of states for all loops. After the evaluation of all possible search paths, the encoding with the lowest cost is assigned to the FSM.

The algorithm consists of two parts, the initialization routine (Algorithm 3) and the (recursive) DFS function (Algorithm 4).

3.5.1.1 Initialization

The initialization routine, as displayed in Algorithm 3, is the top level of the search. This routine sets the parameters for the search, initializes the variables and starts the search. When the search is completed, the most optimal encoding is assigned to the FSM.

Algorithm 3 Initialization

```
#latches = Ceiling(Log(#states))
#codes = 2^#latches
minimum_cost = ∞
DFS
for each state in FSM do
   code = state → best_code
   Assign(state, code)
end for
```

The DFS function has one fixed parameter $\#codes$, which is the number of possible codes. To minimize the search space of the DFS tree, i.e., the number of possible encodings, the algorithm utilizes only the minimum number of possible codes for the state register. The minimum number of *unique* codes required for an FSM state assignment equals the number of states $\#states$ of the FSM. The minimum number of latches $\#latches$ required to uniquely represent these codes in the state register equals the logarithm of the *minimum* number of codes ($\#states$), rounded up: $\#latches = \lceil \log_2(\#states) \rceil$. The number of *possible* codes then follows as: $\#codes = 2^{\#latches}$.

The search algorithm also has one variable, $minimum_cost$, which keeps track of the (minimum) cost of the best encoding found so far. If an encoding with a cost lower than the current minimum is found, that encoding is more optimal. The search algorithm will store the encoding and update the $minimum_cost$ to reflect the new minimum cost. Before the search starts, the minimum cost is set to infinite (∞).

The actual search is performed by the DFS function (Algorithm 4). When the search is completed, the encoding with the minimum cost is assigned to the FSM, and the algorithm finishes.

3.5.1.2 DFS function

The initialization (Algorithm 3) starts the search for an optimal FSM encoding by calling the recursive DFS function (Algorithm 4). The DFS function recursively traverses the search tree of possible FSM encodings in a depth-first manner. Each level in the search tree corresponds to a state. A node on that level indicates a partial FSM encoding for the first "level" states. Each branch in the search tree corresponds to a code assignment to a state. Every time a leaf node (at the bottom) of the tree is reached, the serach path correspond to a possible FSM encoding, and the algorithm estimates the cost of that FSM encoding. If the cost of the encoding is lower than the minimum cost so far, the minimum cost is updated and the encoding is stored. When the entire tree has been traversed, the algorithm ends.

Algorithm 4 DFS function

```
state = UnassignedState(FSM)
if state then
   for each code in unassigned_codes do
      Assign(state, code)
      DFS
      Release(code)
   end for
else
   cost = Estimate(Loops)
   if cost < minimum_cost then
      minimum_cost = cost
      for each state in FSM do
         state → best_code = state → code
      end for
   end if
end if
```

The search algorithm arbitrarily selects an unassigned state for the next level of the search tree. The available (unassigned) codes correspond to the nodes of that level. The assignment of a code to the state symbolizes the traversal of a branch from the current level to a node on the next level. Then, this process is repeated for the next level(s). When a search path has been traversed, the algorithm returns to the previous level by releasing the code, i.e., going back up the branch.

When all states are assigned, the cost of the encoding is estimated using Equation 3.2. If the cost is less than the minimum cost found so far, the minimum cost is updated and the codes in the search path are stored as the minimum cost encoding.

The basic DFS function has an exponential computational complexity described by Equation 3.1, which makes it impossible to find the best solution for an FSM with a larger number of states. To find the best solution for larger FSMs, the search space for the best solution must be reduced.

3.5.1.3 Intermediate-cost DFS function

Algorithm 5 is a direct replacement for Algorithm 4 that implements a method to reduce the search space. This method reduces the search space by aborting search paths which can only lead to worse, i.e., higher than minimum cost, encodings. The viability of the search path is

determined by performing an intermediate, lower bound, estimate of the cost after each state assignment. For the intermediate estimate, the same cost function (Equation 3.2) is used. However, for all transitions from and to unassigned states, a minimal Hamming distance of one is assumed. The result is a lower bound cost estimate. If the estimate is equal or higher than the minimum cost, the search path cannot result in an encoding with a lower cost, and it is therefore aborted.

Algorithm 5 Intermediate-cost DFS function (DFS')

```
state = UnassignedState(Loops)
if state then
  for each code in unassigned_codes do
    Assign(state, code)
    cost = Estimate(Loops)
    if cost < minimum_cost then
      DFS'
    end if
    Release(code)
  end for
else
  minimum_cost = cost
  for each state in FSM do
    state → best_code = state → code
  end for
end if
```

For the largest reduction in search space, the states must be assigned loop by loop in the order of descending loop frequencies. Because the estimate is a function of the loop frequencies, a high-cost assignment contributes more to the cost for states from high frequency loops than from low frequency loops. And the faster the intermediate cost estimate increases, the quicker the minimum cost is exceeded, and the search path is aborted.

The lower the minimum cost becomes, the higher up in the tree a high cost search path will be aborted, and the fewer the estimates. Furthermore, for higher level estimates, when only a few states are assigned, only a few Hamming distances need to be calculated. Together with the reduced search space, this significantly reduces the execution time of the intermediate-cost DFS function as compared to the non-optimized DFS function (Algorithm 4).

3.5.2 Loop-based DFS state assignment algorithm

The basic state assignment algorithm has an exponential computational complexity to the number of states, which makes it unsuitable for FSMs with many states. A well-known solution is to divide the FSMs in several smaller groups, and assign each group of states separately. This way the computational complexity is only exponential to the number of unassigned states in the largest unassigned group. However, the algorithm is only able to find the best solution if all groups are completely unrelated. But in FSMs all states are connected, because all states can be reached from the starting state of the FSM. Therefore the solution cannot be guaranteed to be optimal.

The best way to partition an FSM is to group together strongly connected states, so that

at least the assignments within each group are optimal. The FSM is already partitioned into strongly connected groups in the form of the loops found during the FSM profiling, because the higher the frequency of a loop, the stronger the connection between it's states. We propose a loop-based state assignment algorithm that assigns the states in a loop by loop manner in descending order of loop frequencies. Thus the stronger connected groups are assigned first. Because several loops can contain the same state, loops do not divide the FSM in disjunct partitions, and the algorithm has to take into account the states that were encoded previously. Therefore, only the first loop is guaranteed to be assigned optimally, and other loops might not be assigned optimally.

The loop-based state assignment algorithm (Algorithm 6) starts to encode the loops with the highest weight. The encoding of these loops has the largest impact on the overall cost, and because the pool of free codes is still quite full, it is possible to reduce the cost of a state assignment to a minimum. By optimally assigning the highest weight loops that contribute most to the state register switching activity, the overall switching activity should be reduced.

The loop-based state assignment algorithm consists of two parts, the setup part (Algorithm 6) and the recursive DFS function (Algorithm 7). The setup sorts the loops in descending order according to the loop frequencies, and calls the DFS function for every loop. The DFS function optimizes on the loop level, therefore the assignment cost is only relevant within each loop. When all solutions for a loops have been tried, the best solution is assigned and the setup continues with the next loop.

Algorithm 6 Loop-based DFS setup

```
#latches = Ceiling(Log(#states))
#codes = 2^#latches
loops → AssignWeight(frequency)
loops → Sort(descending)
for each loop in loops do
   minimum_cost = ∞
   DFS(loop)
   for each state in loop do
      code = state → best_code
      Assign(state, code)
   end for
end for
for each state in FSM do
   if not (state → code) then
      code = FindFreeCode()
      Assign(state, code)
   end if
end for
```

The DFS function Algorithm 7 resembles Algorithm 4. The cost estimation function is

$$cost = \sum t \in \mathrm{T_l H}(t), \tag{3.3}$$

where T_l is the set of transitions int the loop, and H is the Hamming distance between the codes of the from and to states of transition t. The DFS only processes the unassigned states of a

loop, but the cost estimate takes into account all states, including the states that were assigned previously.

Algorithm 7 Loop-based DFS function

```
state = UnassignedState(loop)
if state then
  for each code in unassigned_codes do
    Assign(state, code)
    DFS
    Release(code)
  end for
else
  cost = Estimate(loop)
  if cost < minimum_cost then
    minimum_cost = cost
    for each state in loop do
      state → best_code = state → code
    end for
  end if
end if
```

3.5.3 Loop-based heuristic state assignment algorithm

The previous state assignment algorithms both have an exponential computational complexity to the number of (unassigned) states in the FSM or the loop. In practice, when the number of states, and thus the computational complexity, is large, these methods cannot be used. We propose a loop-based heuristic state assignment algorithm that uses a "best guess" approach to assign a free code to an unassigned state. The heuristic has a linear computational complexity to the number of states in the FSM, which is very well suited to large FSMs. Because the cost of the generated solution depends heavily on the quality of the guess, the heuristic chooses a code based upon a minimal Hamming distance to its previous and next states.

The first step of the Algorithm 8 sets the number of latches (and thus the number of possible codes) of the FSM to the minimum required for the number of states in the FSM ($\lceil \log_2(\#states) \rceil$).

The second preliminary step of the algorithm assigns a weight to each loop according to a specified function. Our approach assumes a difference in the frequency of occurrence of loops, therefore the loop frequency obtained by the profiling algorithm is the primary variable of the weight function. However, the number of states in a loop can also be a factor, because the chance of finding an optimal encoding for large loops is higher when less states of the loops are assigned previously and more free codes are still available. Therefore, several weighing functions were considered:

- $weight(l) = frequency(l)$,
  where $frequency(l)$ is the occurrence frequency of loop l,
- $weight = frequency(l) \times \#states(l)$,
  where $\#states(l)$ is the number of states in loop l, thus favoring loops with more states,

Algorithm 8 Heuristic EncodeLoops

```
#latches = Ceiling(Log(#states))
#codes = 2^#latches

loops → AssignWeight(function)
loops → Sort(descending)

for each loop in loops do
  if (state = loop → FindNextAssignedState()) then
    loop → Rotate(state)
  else
    state = loop → state(0)
    code = FindFreeCode()
    Assign(state, code)
  end if
  for each state in loop\state(0) do
    if not (state → code) then
      next state = loop → FindNextAssignedState()
      if (next state ≠ previous state) then
        if (code = FindFreeCode(previous state, next state)) then
          Assign(state, code)
          continue
        end if
      end if
      if (code = FindFreeCode(previous state)) then
        Assign(state, code)
        continue
      end if
      code = FindFreeCode(previous state, next state, Cost())
      Assign(state, code)
    end if
  end for
end for

for each state in FSM do
  if not (state → code) then
    code = FindFreeCode()
    Assign(state, code)
  end if
end for
```

- $weight = frequency(l) \times \#bits(l)$,
  where $\#bits(l) = \lceil \log_2 \#states(l) \rceil$, i.e., the size of the free partition of code space required for an optimal encoding of loop l. This function moderates the influence of the number of states in a loop on its weight.

When the weights are assigned the loops are sorted by order of descending weight.

The main loop of the algorithm targets each loop in weight order, assigning codes to each unassigned state of the loop while minimizing the cost of the assignment. The algorithm needs

to be aware of the states that were assigned earlier, as these states' codes limit the freedom of choice for the codes of the unassigned states. The algorithm features three successive methods to address this problem, falling back to the next method if a method fails.

The first and most advanced assignment method uses both backward and forward dependencies on assigned states' codes. This method requires the previous state to be assigned. To this end, for each new loop the algorithm starts by searching for a state that was already assigned. If such a state is present, the loop is rotated so that state becomes the first state in the loop. If no assigned state was present, the algorithm is free to assign an arbitrary code to the first state without cost penalties (for the assignments in this loop).

Then the algorithm searches forward in the loop to find the next state that is assigned (folding back to the first state in the loop if required). If the previous and next assigned state differ, the first method determines the bits differing between the previous state's code and the next assigned state's code. Each differing bit needs to be changed in some state assigned between the previous state and the next assigned state. Therefore, a code that only differs from the previous state's code by one of the differing bits does not increase the cost for the assignments of this loop. If such a code is found, it is assigned to the state and the algorithm continues with the next state.

The second method, which is utilized when the first method fails, determines the code based solely on the previous state's code. This method searches for a free code with a Hamming distance of 1 from the previous state's code. While this code is an optimal assignment for this state, it cannot be guaranteed that this code leads to an optimal assignment of the whole loop because this assignment could prevent an optimal assignment for some of the subsequent states.

The third and last method is the most expensive method, but it is fail-safe. This method searches for a free code with the smallest cost. The cost of an assignment is based on the Hamming distance to both the previous state's code and the next assigned state's code, taking into account the distance (in states) to the next assigned state. The minimum cost is $1+distance$, with a Hamming distance of 1 between the previous state and the current state, and a cost of $distance$ to reach the next assigned state. For example, if the next assigned state is separated from the current state by one state, or two states away, the minimal cost of the assignments up to the next assigned state is two: 1 for the minimal Hamming distance between the current state and the next state, and 1 for the minimal Hamming distance between the next state and the next *assigned* state. This is equal to the distance from the current state to the next assigned state. When all free codes have been evaluated, or a code with the (minimal) cost of $1 + distance$ was found, the code with the minimal cost is assigned to the state.

When the algorithm has assigned all states in all loops, it assigns arbitrary codes to any unassigned states to complete the FSM state assignment. As the unassigned states were not present in any loops, their assignments have little to no impact on the efficiency of the resulting state assignment.

3.5.4 Optimized loop-based heuristic state assignment algorithm

The final state assignment algorithm proposed here is based upon Algorithm 8. Two optimizations (*marked in italics*) are added to Algorithm 9. The preliminary and final parts of the algorithm remain the same, therefore only the actual state assignment loop is shown.

The first optimization is the use of a binary reflected Gray code for the assignment of the first loop (the loop with the highest weight). The Gray code assigns codes with a minimum

Algorithm 9 Optimized EncodeLoops

```
⋮
  AssignGrayCode(loops → loop(0))
  for each loop in loops\loop(0) do
    if (state = loop → FindNextAssignedState()) then
      loop → Rotate(state)
    else
      state = loop → state(0)
      code = FindFreeCode()
      Assign(state, code)
    end if
    for each state in loop\state(0) do
      if not (state → code) then
        next state = loop → FindNextAssignedState()
        if (next state ≠ previous state) then
          if (code = FindFreeCode(previous state, next state)) then
            Assign(state, code)
            continue
          end if
        end if
        if (code = FindFreeCode(previous state)) then
          Assign(state, code)
          continue
        end if
        if Optimize(dynamic latch allocation) then
          #latches = #latches + 1
          #codes = 2^#latches
          code = InvertHighBit(previous state → code)
          Assign(state, code)
          continue
        end if
        code = FindFreeCode(previous state, next state, Cost())
        Assign(state, code)
      end if
    end for
  end for
⋮
```

Hamming distance between two successive states, thus ensuring the minimum cost of the state assignments for this loop. The code is called reflected because of the way the code is generated: first, half of the necessary codes is generated, in ascending order, with one bit (the same bit for every code) fixed to a certain value. Next, these codes are mirrored, i.e., the mirrored codes are in descending order, and the fixed bit is inverted for all mirrored codes. It can be easily seen that the first code generated differs only in the fixed bit from the last code generated. The same is true for the last of the regular codes and the first of the mirrored codes. Therefore, a loop with an even number of states is guaranteed to have a Hamming distance of one between two successive states, even between the last state and the first state of the sequence. The same is true for a loop with an odd number of states, except for one transition which inevitable needs to have

a Hamming distance of two to obtain an even number of bit changes throughout the loop. This step eliminates the need to assign the first loop using the heuristics, and should thus require less computational effort.

The second, and optional, optimization improves Algorithm 8 by dynamically increasing the number of latches in the state register. By adding a latch to the state register, a new, unused, bit is added to the codes. This ensures that for every existing (assigned) code, a new (unassigned) code is created that differs by exactly one bit, namely the existing code with the new bit inverted. Using this technique, the algorithm will always find an optimal assignment, i.e., assign a code with the minimum Hamming distance of one from the previous state's code. This dynamic latch allocation is performed when the first state assignment method of the heuristic fails to find an optimal assignment.

A problem of the state assignment heuristic using dynamic bit allocation is that it optimizes only locally, i.e., the Hamming distance between the codes of the current and previous states, and does not consider the influence of an assignment on other state transitions. When a code is assigned using dynamic latch allocation, that code will have (at least) one bit that differs from all previously assigned codes. For a transition, in a lower weight loop, which states are both already assigned, this might increase the Hamming distance between the codes of those states.

The example in Section 3.5.5 demonstrates the operation of the optimized state assignment algorithm.

3.5.5 Example

This example demonstrates the working of the optimized state assignment algorithm (Algorithm 9) on the benchmark FSM bbtas.kiss2, which is shown in Figure 3.5.

The FSM consist of six states, therefore three latches are needed, resulting in eight possible codes (000, 001, 010, 011, 100, 101, 110, 111).

Now assume the following loop profiling data (weighted and sorted):

Weight	Loop
100	st0 → st1 → st0
50	st0 → st1 → st2 → st3 → st4 → st5 → st0
25	st1 → st2 → st1

The sequence of states st0 → st1 → st2 → st1 → st0 consists of the (nested) inner loop st1 → st2 → st1 and the outer loop st0 → st1 → st0, which are detected and counted separately.
The first step of the algorithm is the assignment of the first loop using a Gray Code:

State	Code
st0	000
st1	001

The next step is to assign the states of all other loops (in the sorted order). The first assigned state of the second loop is **st0**, therefore the second loop is rotated such that **st0** becomes the first state of the loop (in this case, nothing changes). Now, the algorithm loops through each state of the second loop. **st0** and **st1** are already assigned, so the algorithm skips these.

Next is **st2**. Its previous state is **st1**, with code 001, and its next assigned state is **st0**, with code 000. Their only differing bit is the rightmost bit, and no free codes starting with two zeros

```
.i 2
.o 2
.p 24
.s 6
00 st0 st0 00
01 st0 st1 00
10 st0 st1 00
11 st0 st1 00
00 st1 st0 00
01 st1 st2 00
10 st1 st2 00
11 st1 st2 00
00 st2 st1 00
01 st2 st3 00
10 st2 st3 00
11 st2 st3 00
00 st3 st4 00
01 st3 st3 01
10 st3 st3 10
11 st3 st3 11
00 st4 st5 00
01 st4 st4 00
10 st4 st4 00
11 st4 st4 00
00 st5 st0 00
01 st5 st5 00
10 st5 st5 00
11 st5 st5 00
```

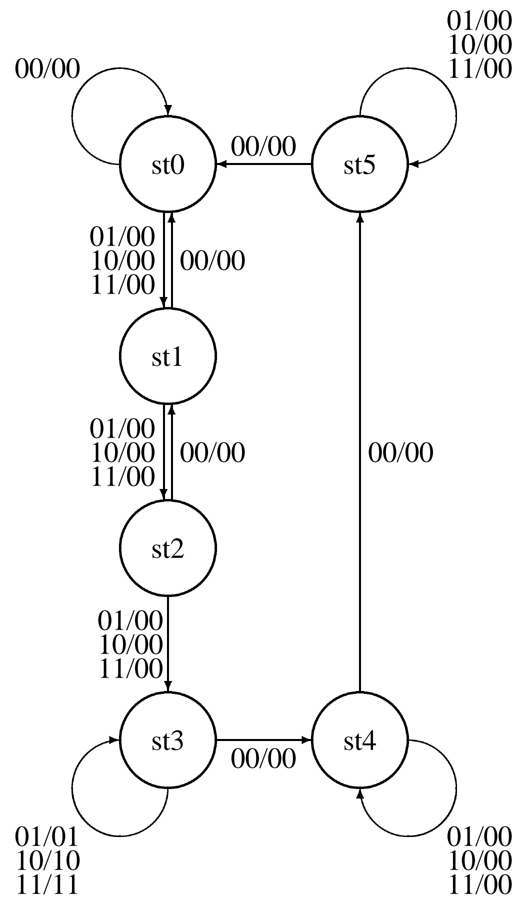


Figure 3.5: FSM BBTAS: KISS description (left) and State Transition Graph

can be found. Therefore, the algorithm has to find a free code only based on the previous state's code 001. The first free code differing only one bit from 001 is 011, so this code is assigned to **st2**:

State	Code
st0	000
st1	001
st2	011

The algorithm continues with **st3**. Its previous state (**st2**) code is 011, its next assigned state (**st0**) code is 000, so the codes differ in the two rightmost bits. The algorithm now searches for a free code differing one bit from the two rightmost bits of the previous state code 011, and finds 010:

State	Code
st0	000
st1	001
st2	011
st3	010

Now, the algorithm starts with **st4**. No free code can be found based on the the previous and next assigned states' codes only differing middle bit. The first free code differing one bit from the previous state's code is 110:

State	Code
st0	000
st1	001
st2	011
st3	010
st4	110

Last in this loop is **st5**. The algorithm searches for a code differing from 110 in one of the two leftmost bits, and finds 100:

State	Code
st0	000
st1	001
st2	011
st3	010
st4	110
st5	100

The algorithm continues with the last two loops, and finds their states are already assigned. Finally, the algorithm checks for states that were not assigned in any loops, but all six states of the FSM are assigned, so the algorithm is finishes.
The resulting state assignment for this FSM has a Hamming distance of one for each state transition in each loop, and is therefore (by definition) optimal. Of course, the possibility of this ideal result depends on the structure of the FSM, and can most probably not be obtained for more complex FSMs.

3.6 Implementation

We have created a C^{++} framework for the FSM data structures necessary to implement the proposed algorithms. This framework includes FSM input/output functions, a Hamming distance function, a random input vector set generator and an FSM simulator. The profiling, loop detection and state assignment algorithms which we proposed in this chapter are implemented on top of this framework.

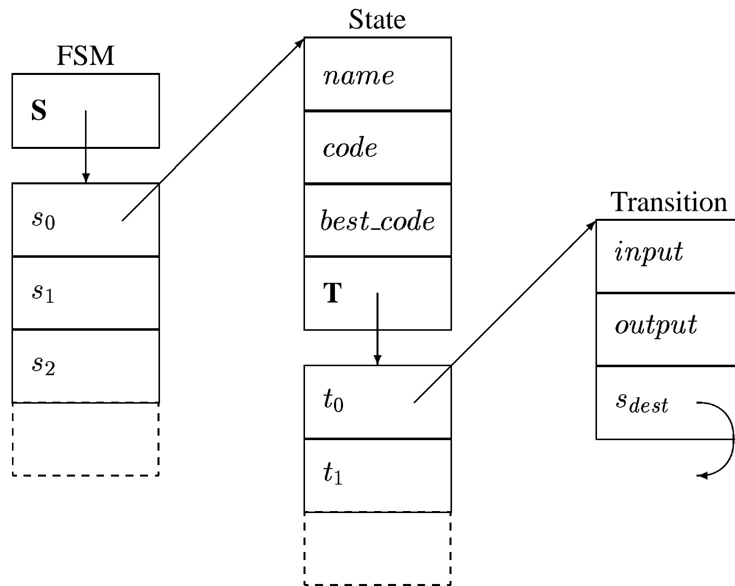


Figure 3.6: FSM data structures

3.6.1 FSM data structures

Our framework defines an FSM using a behavioral model, that is based on a State Transition Graph (STG). An example of an STG is shown in Figure 3.5.

In our implementation, an FSM is defined by a set of states **S**. Each state s_i has a unique identifier $name$ and code $code$. Furthermore, a state has a variable $best_code$ to store the best code so far during a state assignment, and a set of transitions **T** to other states. Every transition t_j contains for an input vector $input$ the destination state s_{dest} and the output vector $output$ of the FSM.

The three datastructures FSM, State and Transition and their relationships are shown in Figure 3.6. Each of the datastructures is implemented as a separate C^{++} class.

3.6.2 Loop data structures

For our loop-based FSM state assignment approach, the FSM framework contains two additional classes: Loops and Loop. Loops is a set of loops **L** used to store, sort and process the detected loops. Each loop l_i consists of an ordered set of states in the loop **S**, and an occurence frequency $frequency$. Furthermore, a loop has a weight $weight$ by which the set of loops are sorted.

Figure 3.7 shows the loop data structures, and their relationship. Both data structures are implemented as separate C^{++} object classes.

3.6.3 Functions

This section describes several of the functions provided by the framework, and the implementation of our proposed state assignment approach.

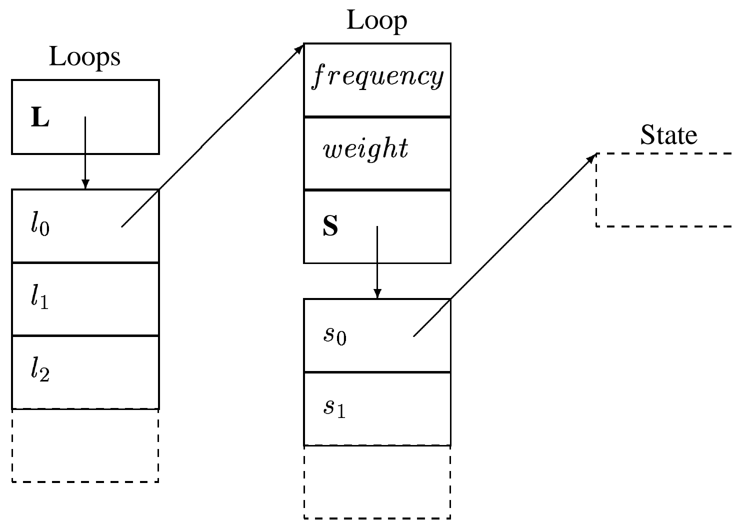


Figure 3.7: Loop data structures

First, the framework provides functions for reading and writing FSM descriptions in KISS and BLIF formats [2]. The BLIF format contains both the KISS FSM description and the state assignments. The framework can also read and write input vector data sets, or randomly generate input vector data sets for a specified FSM, seed and data set length.

Furthermore, the framework provides several functions to support the state assignment algorithms and the evalutation of encodings:

Eliminate
: This function eliminates states that are unreachable from the reset state. The resulting *minimal* FSM ensures that different state assignment implementations encode the same FSM.

HammingDistance
: This function returns the Hamming distance between two codes, which is used in the state assignment cost estimates.

Simulate
: This functions simulates an encoded FSM, and returns the state register switching activity for the provided input vector data set.

Our loop-based profiling state assignment approach is implemented using two main entry functions:

DetectLoops
: This function integrates an FSM state profiler with the proposed loop detection algorithm (Algorithm 2). The FSM is simulated using a provided input vector data set, and the result is a set of loops with their frequencies. By integrating the FSM profiling and the loop detection algorithm, the intermediate state trace does not have to be stored.

Encode

This functions encodes the FSM using the requested state assignment algorithm, the provided loops, and the selected weight function (if applicable).

In the following chapter we evaluate the proposed state assignment approaches using this implementation.

Experimental Results 4

4.1 Introduction

In this chapter, we evaluate the algorithms we proposed in Chapter 3 utilizing an finite state machine (FSM) benchmark suite, and compare the results with state of the art state assignment algorithms. The algorithms are evaluated based upon the switching activity both in the state register and the gate-level circuit generated from the synthesis of the assigned FSMs. We compare our new state assignment algorithms with two state of the art low-power FSM state assignment algorithms, POW3 [1] and the algorithm by Nöth and Kolla [6], as well as with the base for all state assignment algorithm comparisons, the area-oriented JEDI [4] state assignment algorithm. To evaluate the algorithms we use the industry-standard MCNC/LGSynth '89 FSM benchmark suite [3].

The sections of this chapter are outlined as follows. First, we describe the method, and the programs, we use to perform the experiments. Then, we present the results from our new algorithms, followed by the result from the existing algorithms. Finally, we compare the results of all algorithms.

4.2 Method

The aim of FSM state assignment algorithms for low power dissipation is to minimize the power consumption of the resulting real circuits. For our experiments, the switching activity of the gate-level FSM circuit provides a good estimate of the power dissipation of the circuit, with which to compare the algorithms. The experimental method we present consist of the following steps:

1. Setup,
2. FSM profiling and loop detection,
3. FSM state assignment,
4. Synthesis of a gate level FSM circuit, and
5. Simulation of the resulting circuit.

The setup prepares the FSM descriptions and the input vector data sets. The FSM profiling simulates the FSM under an input vector data set to obtain the loops in the state trace. This step is not required by the algorithms we compare with, as the existing algorithms all operate on the static FSM description.

Figure 4.1 shows the steps common to all state assignment algorithms. Each algorithm performs a state assignment, either based on the profiling data or based on the static FSM description. From the assigned FSM, a gate level FSM circuit is synthesized. During the simulation

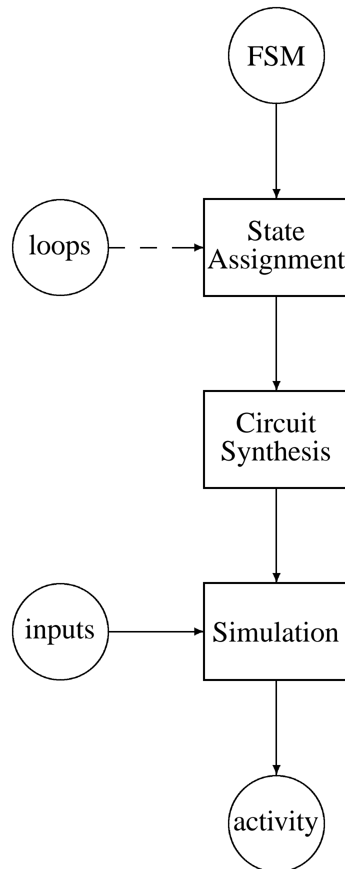


Figure 4.1: Experimental method

of the circuit, the switching activity of the FSM state register and the complete circuit is measured in order to evaluate the overall power consumption. The steps are described in detail in the following sections.

4.2.1 Setup

Before the experiments are run, we set up the FSM and input vector data sets for the experiments. The source for the experiments is the industry-standard MCNC/LGSynth '89 [3] FSM benchmark suite, which was also used by Benini et al [1] and Nöth et al [6]. We utilize the FSMs in the high level KISS description [2].

The profiling and simulation of the FSM requires a set of relevant input vectors which specify the interaction of the outside world with the FSM. Actual (recorded) input vector data sets are not available for the benchmark FSMs, therefore we use several randomly generated data sets to obtain an average result. The starting state of the FSM for the simulation needs to match the starting state for the profiling, therefore we specify the starting state for the data set to be the reset state of that FSM.

The first step of the setup, as shown in Figure 4.2, is to prepare the proper KISS format FSM

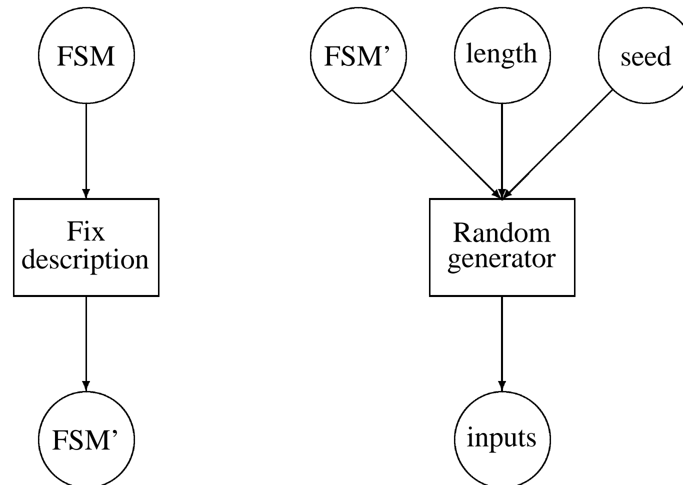


Figure 4.2: Setup step

descriptions. Although the KISS format description provides an optional mechanism to specify the reset state of the FSM, many FSM descriptions from the LGSynth '89 FSM benchmark suite do not use this mechanism. In these cases we need to add the reset state specification to the KISS description. If the reset state is indicated either by name, comment or functionality, we specify this state, otherwise we assume the first state in the description to be the reset state. If the description contains a separate input signal to reset the FSM, this signal is removed from the description to prevent accidental triggering by the randomly generated input vectors.

After the specification of the reset state, some FSM descriptions contain states that can never be reached by starting FSM execution from the reset state. In these cases either the FSM description is incomplete, or the FSM has more than one starting state. However, these states will never be reached within our further experiments, and will therefore be removed either by the state assignment implementation or during the generation of the gate level circuit description. We remove these unreachable states from the KISS FSM description to provide all state assignment implementations with a minimal FSM description.

The second step of the setup is the generation of the input vector data sets, as these sets need to be the same for all simulations of the circuits obtained from the different state assignments to allow a comparison of the results. The required width of the input vector is obtained from the modified FSM description that forms the input for the circuit synthesis. The input data sets are generated using a random generator. To be able to average the influence of the input data sets on the state encodings found, different seeds are used to obtain 10 different input data sets for each FSM. To determine the influence of the length of an input data set on profiling-based algorithms, two different, arbitrary, lengths of data sets (based on the same seeds) will be used, one with 1000 vectors and one with 10000 input vectors. The input vector sets are used during the FSM profiling and the circuit simulation.

4.2.2 FSM Profiling and State Assignment

Chapter 3 describes the method for FSM state assignment using FSM profiling that is evaluated during these experiments. The FSM profiler simulates the operation of the FSM under a certain input vector data set, and returns a set of state loops and a set of state transitions. Our state assignment algorithms utilize the loop data to produce a state encoding for the FSM optimized for that specific input vector data set. This results in a separate state encoding, and thus a separate circuit, for each input vector data set.

The state assignment algorithms that we compare with operate on the static FSM description, and therefore produce only one state encoding (and one circuit) for an FSM. However, POW3 [1] performs the state assignment based upon the state transition probabilities of the FSM. For comparison, we have run POW3 with both the static FSM state transitions probabilities, and the state transition probabilities determined during the FSM profiling. Like the other profiling-based state assignment algorithms, this results in a separate state encoding for each input set.

The resulting state assignment is added to the KISS FSM description to produce a BLIF FSM description [2] suitable for the circuit synthesis.

4.2.3 Circuit Synthesis

The gate level circuit description is generated by the sequential circuit synthesis system **SIS** [2]. The circuit simulator we use requires requires the code of the reset state to correspond to the reset value of the state register, for which all latches are zero. If this is not the case for the FSM encoding under evaluation, the bits that are one for the reset state's code are inverted for all state codes. Because the same bits are inverted for *all* state codes, the state register switching activity is unaffected. The standard SIS script **script.rugged** translates the high level FSM description with encoding into a set of logical functions, and minimizes these functions to reduce the number of gates required for a gate level implementation. This approach minimizes the required area, but it does not necessarily minimize the power dissipation of the circuit. Finally, the logical functions are mapped onto the standard SIS MCNC gate-library **mcnc.genlib** to obtain a gate level description. Because the output signal of certain gates such as NANDs and NORs are inverted, extra inverters are inserted to compensate for this. A last optimization evaluates for each interconnecting node if complementing that node will eliminate inverters, thus minimizing the required area. The gate level circuit is returned in SLIF format. The input and output signals are ordered as they appear in the high level FSM description. After specifying a clock signal for the latches, this description is suitable for simulation by the logic simulator.

4.2.4 Simulation

The simulation of the circuit is performed by the **Mercury** logic simulator, part of the Olympus Synthesis System [5]. Using the settings **set sim_view all** and **reset sim_view io only**, the output of the **simple_simulate_plot** command contains the changes of all internal signals of the circuit, including the state register latches. From this output, the state register and circuit switching activities are determined.

For the profiling-based state assignment methods, each input vector data set is input into its matching, unique, circuit. The reference state assignment approaches produce only a single circuit, which is simulated (separately) for all input sets.

4.3 Benchmarks

The FSMs we use as benchmarks to evaluate the performance of the state assignment algorithms come from the MCNC/LGSynth '89 [3] FSM benchmark suite. This is the industry-standard benchmark suite for FSM state assignment. The benchmarks use the table-based KISS format [2] to describe the FSM.

As explained in Section 4.2.1, the KISS descriptions of some examples need to be modified to allow proper operation of the different state assignment algorithms and the simulation. Where needed, we write out the wild-cards in the description in full, and add a reset state specification. This specification replaces explicitly specified reset functionality using a reset input signal, which is removed, as are any states which cannot be reached from the reset state.

Table 4.1 shows the benchmarks, the number of states each FSM consists of, and the number of codes available to the state assignment algorithms for the minimum size of the state register. Furthermore, the table lists the average number of loops detected in the state trace for both the 1000 input vector and the 10000 input vector data sets, and the average number of state transitions *to another state* that occurred. The last value is used to determine the percentage of bit changes in the state register for each state transition. The transitions, for which the FSM does not change its state, do not cause bit changes in the register, so these are excluded.

Table 4.1: Benchmarks Statistics

Benchmark	**States**	**Codes**	**Loops**		**Transitions**	
			1000	10000	1000	10000
bbara	10	16	20.3	35.7	228.1	2211.2
bbsse[1]	13	16	15.4	22.8	683.1	6748.4
bbtas	6	8	3.0	3.0	438.4	4459.7
beecount	7	8	8.0	10.0	417.8	4076.9
cse	16	16	8.5	13.9	229.7	2272.8
dk14	7	8	38.9	59.4	827.6	8236.8
dk15	4	4	7.0	7.0	706.6	7072.5
dk16	27	32	134.3	799.9	962.4	9622.0
dk17	8	8	17.9	23.0	837.1	8387.9
dk27	7	8	12.0	12.0	1000.0	10000.0
dk512[1]	14	16	26.8	29.0	1000.0	10000.0
donfi le[2]	24	32	89.0	427.7	754.5	7509.7
ex1	20	32	32.4	72.4	516.2	5182.7
ex2[3]	19	32	0.3	0.3	2.9	2.9
ex3[3]	10	16	0.2	0.2	2.9	2.9
ex4	14	16	3.0	3.0	439.6	4347.4
ex5[3]	9	16	0.4	0.4	3.5	3.5
ex6	8	8	30.9	43.5	806.7	8021.8
ex7[3]	10	16	0.0	0.0	1.2	1.2
keyb	19	32	5.9	9.1	542.2	5496.1
kirkman[45]	16	16	1.0	1.0	501.1	4995.0
lion	4	4	3.0	3.0	380.5	3734.0
lion9[6]	9	16	8.0	8.0	446.0	4442.1
mark1[146]	13	16	8.0	8.0	1000.0	10000.0
mc	4	4	1.0	1.0	423.3	4288.9

[1]Unreachable state(s) [3]Terminating state(s) [5]Wild-card(s)
[2]Constant output(s) [4]Reset functionality [6]Reset state unknown

Table 4.1: Benchmarks Statistics (Continued)

Benchmark	States	Codes	Loops		Transitions	
			1000	10000	1000	10000
modulo12[2]	12	16	1.0	1.0	507.8	5005.3
opus[4]	10	16	6.0	6.0	736.7	7340.1
planet	48	64	27.4	50.1	959.0	9602.3
planet1	48	64	27.4	50.1	959.0	9602.3
pma	24	32	28.3	62.0	442.8	4359.3
s1	20	32	91.9	341.8	733.0	7311.9
s1488	48	64	9.2	18.3	303.1	2894.8
s1494	48	64	9.2	18.3	303.1	2894.8
s1a[2]	20	32	91.9	341.8	733.0	7311.9
s208[1]	18	32	4.4	5.7	454.8	4387.5
s27	6	8	32.5	40.4	685.0	6817.5
s298	218	256	74.3	352.3	745.6	7525.8
s386	13	16	14.8	22.8	673.4	6709.5
s420[1]	18	32	4.4	5.7	454.8	4387.5
s510	47	64	6.0	6.0	650.7	6706.6
s8[2]	5	8	4.0	4.0	124.8	1300.9
s820	25	32	6.8	13.4	535.6	5371.0
s832	25	32	6.8	13.4	535.6	5371.0
sand	32	32	29.8	48.6	476.0	4843.9
scf[124]	115	128	32.6	132.5	1000.0	10000.0
shiftreg	8	8	16.9	17.0	877.0	8750.8
sse[1]	13	16	15.4	22.8	683.1	6748.4
styr	30	32	10.6	16.9	511.2	5073.2
tav	4	4	1.0	1.0	1000.0	10000.0
tbk	32	32	73.1	257.4	568.5	5716.5
tma[6]	20	32	7.8	17.0	164.6	1571.5
train11[6]	11	16	4.0	4.0	332.8	3335.8
train4[6]	4	4	1.0	1.0	391.0	4002.1

[1]Unreachable state(s) [3]Terminating state(s) [5]Wild-card(s)
[2]Constant output(s) [4]Reset functionality [6]Reset state unknown

Some benchmark FSMs contain states from which no transition is possible. If an FSM enters such a terminating state, it will remain trapped in that state for the remainder of the simulation. These benchmarks, **ex2**, **ex3**, **ex5** and **ex7**, do not generate valid results during simulation and are unsuited for our experiments. On the other hand, for **kirkman**, **mc**, **modulo12**, **tav** and **train4**, only one even-length loop containing all FSM states, is detected. These benchmarks ultimately benefit from our loop-based state assignment approach. Furthermore, these FSMs can be assigned optimally (one bit change for each state change) by utilizing some form of (binary reflected) Gray code. The same holds true for **bbtas** (Figure 3.5), for which in our experiments the most frequently detected loop always contains all states. The other loops of **bbtas** form transitions of the first loop, so they are always assigned optimally.

The states of the FSMs **lion** and **lion9** form an open-ended chain, i.e., each state only has transitions back to the previous state, or forward to a single next state. A state assignment based on a Gray code will result in an optimal solution.

The benchmarks **donfile**, **modulo12**, **s1a** and **s8** have only constant outputs. When a circuit

is generated from these descriptions, the simplification step reduces the complete circuit to only the constant outputs, so the total circuit switching activity can not be determined. Benchmark **s1a** is the constant output equivalent of benchmark **s1**, so it is excluded. The benchmarks **planet** and **planet1** are exact duplicates, so only the first one is included in the test set.

The following sections present the results from the experiments with the remaining benchmarks.

4.4 Results

We have applied our experimental method to all algorithms and for all FSM benchmarks, and here we present a summary of the results. FSM state assignment algorithms for low power dissipation do not target the circuit switching activity directly, but instead minimize only the state register switching activity. However, we compare not only the state register switching activity, but also the gate-level circuit switching activity.

The switching activity is expressed as the number of bit changes in the state register, or the number of signal changes in the circuit, that occurred during the simulation. During the simulation, each state transition will cause at least one bit to change in the state register. Therefore, the number of state transitions forms a lower bound for the state register switching activity. In the experimental results, we provide both the actual state register activity and the value normalized to the lower bound. The circuit switching activity can not be normalized to an ideal value, therefore those results need to be compared to the results of the other algorithms during the concluding algorithm comparison. For the comparison with other algorithms, any incomplete results will be ignored.

The experiments are run using two sets of 10 input data vector sets each. The first set has 1000 input vectors per data set, the second set has 10000 input vectors per data set. The results of the experiments for each of the 10 data sets are averaged, and listed separately for the two data set sizes.

First, we present the results from the new algorithms we propose in this report. Then, we present the results of the best, known, existing algorithms. Finally, we will compare the results of the algorithms to determine the efficiency of the algorithms we propose.

4.4.1 DFS

In this section, we discuss the results from the implementation of the intermediate-cost DFS state assignment algorithm, Algorithm 5. This algorithm guarantees to find the optimal state assignment solution for the FSM for a minimal width state register, at the cost of computing time. Table 4.2 displays the benchmark results.

Table 4.2: DFS average switching activity

Benchmark	**1000 Input Vectors Data Set**			**10000 Input Vectors Data Set**		
	State Register		Circuit	State Register		Circuit
bbara	283.2	124.2%	3893.1	2729.3	123.4%	41512.8
bbsse	786.2	115.1%	8358.2	7771.4	115.2%	89415.3
bbtas	438.4	100.0%	1955.3	4459.7	100.0%	20078.0
beecount	439.3	105.1%	3376.4	4296.0	105.4%	33554.4

Table 4.2: DFS average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
cse	240.5	104.7%	8922.4	2374.4	104.5%	88387.5
dk14	1108.6	133.9%	14533.2	11061.0	134.3%	156378.2
dk15	844.2	119.5%	9301.0	8497.0	120.1%	92876.1
dk16	-	-	-	-	-	-
dk17	1024.8	122.4%	9158.1	10327.4	123.1%	89671.4
dk27	1186.3	118.6%	4677.5	11887.7	118.9%	49641.2
dk512	1183.6	118.4%	9322.6	11834.1	118.3%	98681.2
donfi le	-	-	-	-	-	-
ex1	660.4	127.9%	9498.8	6738.3	130.0%	101464.8
ex4	485.5	110.4%	3000.7	4785.7	110.1%	29812.6
ex6	1017.3	126.1%	10408.0	10062.9	125.4%	98476.7
keyb	548.7	101.2%	16553.1	5570.2	101.3%	159576.3
kirkman	501.1	100.0%	9282.9	4995.0	100.0%	93045.1
lion	380.5	100.0%	1574.9	3734.0	100.0%	15551.1
lion9	446.0	100.0%	1516.0	4442.1	100.0%	14666.3
mark1	1300.5	130.1%	8558.4	13039.7	130.4%	87259.3
mc	423.3	100.0%	2312.2	4288.9	100.0%	23307.6
modulo12	507.8	100.0%	-	5005.3	100.0%	-
opus	937.2	127.2%	7023.2	9355.0	127.5%	71531.0
planet	-	-	-	-	-	-
pma	536.2	121.1%	8899.3	5309.5	121.8%	88108.0
s1	-	-	-	-	-	-
s1488	345.8	114.0%	26777.0	-	-	-
s1494	346.1	114.1%	27526.0	-	-	-
s208	495.0	108.8%	5518.0	4758.1	108.4%	57285.5
s27	890.0	129.9%	3675.3	8879.4	130.2%	34044.7
s298	-	-	-	-	-	-
s386	775.1	115.1%	9094.7	7730.9	115.2%	89046.7
s420	495.0	108.8%	5541.0	4758.1	108.4%	57296.1
s510	-	-	-	-	-	-
s8	143.9	115.4%	-	1478.5	113.7%	-
s820	545.2	101.8%	18436.0	5473.4	101.9%	170828.1
s832	545.2	101.8%	15968.9	5473.4	101.9%	148959.6
sand	-	-	-	-	-	-
scf	-	-	-	-	-	-
shiftreg	999.9	114.0%	4444.7	9998.7	114.3%	44393.0
sse	786.2	115.1%	8358.2	7771.4	115.2%	89415.3
styr	551.5	107.9%	25645.0	5483.1	108.1%	236871.6
tav	1000.0	100.0%	2499.7	10000.0	100.0%	25000.2
tbk	-	-	-	-	-	-
tma	188.3	114.4%	2553.9	1804.6	114.9%	21843.1
train11	403.1	121.2%	1994.7	4127.5	123.7%	22354.8
train4	391.0	100.0%	1126.6	4002.1	100.0%	11522.2

The special benchmarks **bbtas**, **kirkman**, **lion**, **lion9**, **mc**, **modulo12**, **tav** and **train4**, described in Section 4.3, are assigned optimally, resulting in an average state register switching activity of 100%, or only one bit change for each state transition. On the other hand, the **dk14**

benchmark requires on average 34% extra bit changes per state transition for an optimal state assignment. The reason for this is the relatively large number of loops (49 on average) for the number of states, seven.

As expected, for FSMs with a large number of states or a large number of loops, the algorithm fails to complete in acceptable time. This is the case for benchmarks **dk16**, **donfile**, **planet**, **s1**, **s1488**, **s1494**, **s298**, **s510**, **sand**, **scf** and **tbk**. As explained before, for benchmarks **modulo12** and **s8** the circuit can not be synthesized.

The difference in the results between the 1000 and the 10000 input vector data sets is negligible, with a maximum difference of 2.5% for the average state register switching activities of **train11**.

4.4.2 Loop-based DFS

The loop-based DFS state assignment implementation of Algorithm 7 improves the cost of computing time of the basic DFS state assignment algorithm, at the cost of a less optimal state assignment solution for some benchmarks.

Table 4.3: Loop-based DFS average switching activity

Benchmark	**1000 Input Vectors Data Set**			**10000 Input Vectors Data Set**		
	State Register		Circuit	State Register		Circuit
bbara	298.3	130.8%	4407.3	2812.1	127.2%	39162.1
bbsse	803.9	117.7%	8029.2	7847.8	116.3%	86158.5
bbtas	438.4	100.0%	2226.5	4459.7	100.0%	22821.3
beecount	446.3	106.8%	3433.3	4398.6	107.9%	34912.1
cse	243.7	106.2%	9061.0	2397.2	105.5%	92177.4
dk14	1155.4	139.6%	14821.2	11282.2	137.0%	141619.0
dk15	900.1	127.4%	9892.6	8761.7	123.9%	98208.0
dk16	1890.3	196.5%	33287.7	19386.8	201.5%	372877.9
dk17	1105.0	132.0%	8547.5	11332.3	135.1%	86895.4
dk27	1416.7	141.7%	5443.6	14091.1	140.9%	57053.1
dk512	1552.2	155.2%	11333.3	16145.0	161.5%	121703.6
donfi le	1454.0	192.7%	-	14623.8	194.8%	-
ex1	718.3	139.0%	9938.2	7651.0	147.6%	101161.1
ex4	501.8	114.1%	2705.4	4988.1	114.7%	27050.8
ex6	1200.2	148.8%	11763.8	11875.2	148.0%	126867.8
keyb	551.9	101.8%	17938.5	5617.8	102.2%	179321.0
kirkman	501.1	100.0%	9282.9	4995.0	100.0%	93045.1
lion	416.8	109.5%	1612.8	4097.9	109.8%	15811.6
lion9	526.4	117.8%	1894.0	5556.6	125.1%	19833.9
mark1	1344.1	134.4%	8572.6	13442.7	134.4%	85906.5
mc	423.3	100.0%	2312.2	4288.9	100.0%	23307.6
modulo12	507.8	100.0%	-	5005.3	100.0%	-
opus	1015.9	137.9%	8557.7	10121.0	137.9%	88388.3
planet	-	-	-	-	-	-
pma	584.7	132.0%	9907.5	5423.9	124.4%	86698.0
s1	1263.8	172.4%	31557.3	13222.1	180.8%	292258.7
s1488	389.7	128.5%	27365.0	3755.7	129.7%	263844.9*
s1494	389.9	128.6%	27331.5	3755.7	129.7%	256551.6

* Incomplete results

Table 4.3: Loop-based DFS average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
s208	519.9	114.3%	6603.5	5008.9	114.2%	54666.6
s27	980.9	143.1%	4653.0	10179.8	149.3%	50626.0
s298	1118.6*	149.8%*	88218.7*	11952.0	158.8%	1006787.2*
s386	802.3	119.1%	9397.7	7801.5	116.3%	86259.7
s420	519.9	114.3%	6048.3	5008.9	114.2%	49019.2
s510	-	-	-	-	-	-
s8	150.4	121.1%	-	1478.5	113.7%	-
s820	551.8	103.0%	19941.9	5546.8	103.3%	191724.0
s832	551.8	103.0%	16289.1	5546.8	103.3%	159608.7
sand	599.5*	127.8%*	36343.8*	6275.4*	128.7%*	355033.8*
scf	-	-	-	-	-	-
shiftreg	1296.3	147.8%	5051.3	12617.0	144.2%	47092.8
sse	803.9	117.7%	8029.2	7847.8	116.3%	86158.5
styr	555.7	108.7%	26180.8	5523.7	108.9%	235782.9*
tav	1000.0	100.0%	2499.7	10000.0	100.0%	25000.2
tbk	984.2	173.2%	31468.1	10556.0	184.7%	350766.9
tma	200.3	121.9%	2670.2	1811.4	115.3%	23701.3
train11	403.1	121.2%	1994.7	4127.5	123.7%	21634.0
train4	391.0	100.0%	1126.6	4002.1	100.0%	11522.2

* Incomplete results

This implementation is able to complete almost all state assignments, in acceptable time. Again, the single-loop benchmarks are assigned optimally, **lion** and **lion9** however are not. The state assignment solutions for **dk16**, **donfile** and **sand**, benchmarks that the DFS algorithm failed to assign, are the best of all our proposed state assignment algorithms. On average, the results are 7.2% worse than the solutions of the DFS algorithm.

For some input vector data sets, the algorithm fails to find a state assignment or circuit solution in acceptable time. The average results shown in the table only take into account the valid solutions, thus care must be taken when comparing the resulting average to averages from complete results.

4.4.3 Loop-based Heuristic

The loop-based state assignment heuristic (Algorithm 9) is tested with each of the three different loop sorting weight functions we propose. Table 4.4 shows the best average state register results obtained, and the weight function(s) that produced them. The circuit switching activity results shown correspond to that weight function.

Table 4.4: Loop-based Heuristic average switching activity

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
bbara	292.8[1]	128.4%[1]	4033.6[1]	2797.9[1]	126.6%[1]	41168.2[1]

* Incomplete results
[1] $weight = frequency$ [2] $weight = frequency \times \#states$ [3] $weight = frequency \times \#bits$

Table 4.4: Loop-based Heuristic average switching activity (Continued)

Benchmark	**1000 Input Vectors Data Set**			**10000 Input Vectors Data Set**		
	State Register		Circuit	State Register		Circuit
bbsse	792.8[1]	116.0%[1]	8900.2[1]	7831.0[2]	116.0%[2]	88028.9[2]
bbtas	438.4[123]	100.0%[123]	2001.6[123]	4459.7[123]	100.0%[123]	20527.0[123]
beecount	440.7[123]	105.5%[123]	3406.0[123]	4301.6[123]	105.5%[123]	33514.6[123]
cse	243.7[123]	106.2%[123]	9001.1[123]	2397.2[1]	105.5%[1]	84411.6[1]
dk14	1190.7[2]	143.9%[2]	15266.7[2]	11591.2[1]	140.7%[1]	154789.0[1]
dk15	898.1[1]	127.2%[1]	10135.4[1]	8637.0[1]	122.1%[1]	98384.4[1]
dk16	1932.2[2]	200.8%[2]	37196.3[2]	19464.9[3]	202.3%[3]	377403.2[3]
dk17	1071.6[2]	128.0%[2]	8061.0[2]	10377.5[2]	123.7%[2]	76003.0[2]
dk27	1305.8[1]	130.6%[1]	4658.9[1]	13369.8[1]	133.7%[1]	47163.1[1]
dk512	1471.3[3]	147.1%[3]	11041.5[3]	14672.3[2]	146.7%[2]	113401.3[2]
donfile	1516.7[2]	201.2%[2]	-	15843.2[3]	211.0%[3]	-
ex1	713.5[2]	138.1%[2]	10252.9[2]	7529.1[3]	145.3%[3]	95307.4[3]
ex4	498.7[2]	113.4%[2]	2990.6[2]	4995.6[23]	114.9%[23]	30136.9[23]
ex6	1049.6[1]	130.1%[1]	9497.2[1]	10542.3[123]	131.4%[123]	93326.6[123]
keyb	551.9[123]	101.8%[123]	17938.5[123]	5617.8[123]	102.2%[123]	179321.0[123]
kirkman	501.1[123]	100.0%[123]	8446.4[123]	4995.0[123]	100.0%[123]	83804.6[123]
lion	380.5[123]	100.0%[123]	1594.9[123]	3734.0[123]	100.0%[123]	15667.1[123]
lion9	591.0[1]	132.8%[1]	1749.5[1]*	6484.0[123]	146.1%[123]	27841.2[123]
mark1	1380.2[1]	138.0%[1]	7965.2[1]	13650.5[3]	136.5%[3]	86069.6[3]
mc	423.3[123]	100.0%[123]	2312.2[123]	4288.9[123]	100.0%[123]	23307.6[123]
modulo12	507.8[123]	100.0%[123]	-	5005.3[123]	100.0%[123]	-
opus	979.0[1]	132.9%[1]	7516.4[1]	9677.4[1]	131.8%[1]	74622.2[1]
planet	1231.3[3]	128.3%[3]	35670.6[3]	12219.9[3]	127.3%[3]	341095.4[3]*
pma	564.0[2]	127.4%[2]	9458.3[2]	5835.5[2]	133.9%[2]	88686.5[2]
s1	1250.9[3]	170.7%[3]	32909.7[3]	12951.9[3]	177.1%[3]	334074.1[3]
s1488	348.7[1]	115.0%[1]	27052.4[1]	3327.9[1]	115.0%[1]	276139.8[1]
s1494	348.9[2]	115.0%[2]	27633.1[2]	3327.9[3]	115.0%[3]	257408.0[3]
s208	494.8[123]	108.8%[123]	5904.1[123]	4758.3[123]	108.5%[123]	61029.3[123]
s27	990.3[1]	144.5%[1]	4381.8[1]	10325.2[12]	151.5%[12]	46268.0[12]
s298	1115.9[1]	149.6%[1]	131157.7[1]	12099.3[1]	160.8%[1]	1273708.7[1]*
s386	783.5[3]	116.3%[3]	9802.6[3]	7785.3[1]	116.0%[1]	92258.1[1]
s420	494.8[123]	108.8%[123]	5918.9[123]	4758.3[123]	108.5%[123]	61041.6[123]
s510	675.7[2]	103.9%[2]	12418.2[2]	6992.6[23]	104.3%[23]	126094.0[23]
s8	143.9[23]	115.4%[23]	-	1478.5[123]	113.7%[123]	-
s820	551.4[3]	102.9%[3]	16631.4[3]	5533.4[3]	103.0%[3]	188571.3[3]
s832	551.4[2]	102.9%[2]	18146.3[2]	5533.4[3]	103.0%[3]	186136.8[3]
sand	638.3[2]	134.0%[2]	38281.0[2]	6671.0[2]	137.8%[2]	365520.1[2]
scf	1216.6[2]	121.7%[2]	38774.4[2]	12192.5[2]	121.9%[2]	413467.8[2]*
shiftreg	1165.4[2]	132.9%[2]	4857.9[2]	11976.5[2]	136.9%[2]	47425.6[2]
sse	792.8[1]	116.0%[1]	8900.2[1]	7831.0[2]	116.0%[2]	88028.9[2]
styr	574.3[2]	112.3%[2]	25058.2[2]	5682.7[3]	112.0%[3]	253564.0[3]
tav	1000.0[123]	100.0%[123]	2499.7[123]	10000.0[123]	100.0%[123]	25000.2[123]
tbk	999.9[2]	175.9%[2]	32773.4[2]	10830.1[23]	189.4%[23]	349604.5[23]
tma	198.9[2]	120.8%[2]	2509.8[2]	1891.5[2]	120.4%[2]	25383.5[2]

* Incomplete results

[1] $weight = frequency$ [2] $weight = frequency \times \#states$ [3] $weight = frequency \times \#bits$

Table 4.4: Loop-based Heuristic average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
train11	406.8[123]	122.3%[123]	2272.0[123]	4150.0[123]	124.4%[123]	23615.0[123]
train4	391.0[123]	100.0%[123]	1126.6[123]	4002.1[123]	100.0%[123]	11522.2[123]

* Incomplete results
[1] $weight = frequency$ [2] $weight = frequency \times \#states$ [3] $weight = frequency \times \#bits$

The heuristic is able to determine a state assignment solution (nearly) instantly, and for all benchmarks. The algorithm performs optimally for the single-loop benchmarks and **lion**, however the result for **lion9** is not optimal. For **s1488**, **s1494** and **s298**, the results are better than the solutions of the other algorithms, and for many other benchmarks, the best solution is equaled. Overall, the results are 5.0% worse than the solutions of the DFS algorithm, but 5.3% better than the loop-based DFS algorithm solutions.

The second weight function has a small advantage over the weight functions: 48 times, the second function results in the best solution, versus 44 times for the first function, and 40 times for the third function. However, the heuristic is fast enough that all three functions can be compared to obtain the best solution, by utilizing the loops statistics to estimate the state register switching activity.

4.4.3.1 Dynamic Heuristic

The loop-based state assignment heuristic (Algorithm 9) offers the preliminary option to dynamically increase the number of latches of the state register. By expanding the state register, the dynamic heuristic can always find a code that differs in only one bit from the previous state's code. The results are obtained utilizing the first weight function, $weight = frequency$. If the heuristic does not require a wider register, the results equal the basic loop-based heuristic results. To highlight the differences between the basic results and the new results, only the benchmarks for which more than half of the encodings utilize a wider register are shown in Table 4.5.

Table 4.5: Loop-based Dynamic Latch-allocation Heuristic average switching activity

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
dk15	885.9	125.5%	10209.9	8451.0	119.5%	100641.7
dk16	1985.7	206.4%	36747.6	19561.9	203.3%	370429.4
dk17	1189.3	142.1%	9818.3	12131.3	144.7%	99971.9
dk512	1543.5	154.4%	10712.2	16813.6	168.1%	117584.4
donfi le	1622.9	215.2%	-	17342.4	230.9%	-
mark1	1357.1	135.7%	8420.3	13622.4	136.2%	88290.6
opus	975.2	132.3%	7303.2	9677.4	131.8%	77249.8
planet	1239.7	129.3%	36808.4	12161.2	126.6%	412652.4*
s1	1326.5	181.1%	36194.3	13101.7	179.2%	357042.2
sand	678.5	142.4%	39335.1	6814.8	140.7%	356968.9

* Incomplete results

Table 4.5: Loop-based Dynamic Latch-allocation Heuristic average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
scf	1223.1	122.3%	29546.0	12196.3	122.0%	264248.1*
tbk	771.0	135.7%	23916.3	7165.8	125.4%	189925.0
tma	230.5	140.1%	2787.1	2293.1	145.9%	22634.8

* Incomplete results

Overall, this preliminary stage dynamic heuristic achieves better results than the default loop-based state assignment heuristic in four instances, but worse in nine. Like the default heuristic, this approach optimizes only locally, or the current state's assignment, not globally. For certain profiling results, the increased number of latches causes extra switching activity in the sub-optimally assigned lower weight loops. This added switching activity exceeds any decrease in switching activity achieved by optimally assigning the high weight loops. Although this heuristic is only in it's preliminary stage, the **tbk** benchmark demonstrates the significant improvement possible for well-suited FSMs.

4.4.4 Profiling-based POW3

This section discusses the results from the POW3 state assignment heuristic when supplied with the actual state transition probabilities obtained from the FSM profiling. This approach gives POW3 the same advantage as the other profiling-based algorithms, so it enables us to compare our heuristics to the heuristics implemented in POW3.

Table 4.6: Profiling-based Pow3 average switching activity

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
bbara	291.8*	129.2%*	4271.8*	2816.7	127.4%	39792.1
bbsse	-	-	-	-	-	-
bbtas	461.8	105.0%	2215.4	4459.7	100.0%	22150.0
beecount	438.8*	105.6%*	3165.8*	4299.6	105.5%	31536.3
cse	-	-	-	-	-	-
dk14	1191.8	144.0%	13536.4	12053.9	146.3%	136018.1
dk15	844.2	119.5%	10004.5	8497.0	120.1%	100650.0
dk16	1879.3*	195.3%*	37204.6*	19053.1	198.0%	383426.9
dk17	1117.6	133.5%	8542.2	11183.9	133.3%	85083.8
dk27	1256.3	125.6%	4582.5	12565.0	125.6%	44115.2
dk512	1452.1	145.2%	10755.6	14889.4	148.9%	110390.4
donfile	1545.1	204.9%	-	15418.1	205.3%	-
ex1	-	-	-	-	-	-
ex4	502.3	114.3%	3241.1	4994.6	114.9%	32291.6
ex6	1039.7	128.9%	10114.0	10062.9	125.4%	100005.7
keyb	-	-	-	-	-	-
kirkman	562.9	112.3%	8201.7	5618.4	112.5%	81953.5
lion	416.8	109.5%	1613.7	4097.9	109.8%	15899.9

* Incomplete results

Table 4.6: Profiling-based Pow3 average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
lion9	518.9	116.3%	1785.6	5556.6	125.1%	20095.7
mark1	1376.2	137.6%	8587.3	13767.5	137.7%	85788.9
mc	423.3	100.0%	2312.2	4288.9	100.0%	23307.6
modulo12	507.8	100.0%	-	5005.3	100.0%	-
opus	1028.5	139.6%	9245.2	10272.6	139.9%	92128.8
planet	1428.8*	147.8%*	50327.5*	14852.1	154.7%	-
pma	-	-	-	-	-	-
s1	1210.0	165.0%	33356.1*	12145.1	166.1%	332168.2
s1488	-	-	-	-	-	-
s1494	-	-	-	-	-	-
s208	-	-	-	-	-	-
s27	915.5	133.7%	4175.8	9002.4	132.1%	37536.5
s298	-	-	-	-	-	-
s386	-	-	-	-	-	-
s420	-	-	-	-	-	-
s510	771.9	118.6%	17162.8	8052.5	120.1%	154454.9
s8	167.7	134.3%	-	1732.1	133.1%	-
s820	-	-	-	-	-	-
s832	-	-	-	-	-	-
sand	834.0*	178.2%*	41683.0*	8323.5	171.9%	397018.9*
scf	-	-	-	-	-	-
shiftreg	1283.4	146.3%	5234.1	13195.3	150.8%	58367.8
sse	-	-	-	-	-	-
styr	-	-	-	-	-	-
tav	1000.0	100.0%	2499.7	10000.0	100.0%	25000.2
tbk	961.9	169.2%	34208.6	10225.3	178.9%	335687.2
tma	-	-	-	2296.7*	144.1%*	27251.7*
train11	418.4	125.8%	2170.7	4182.4	125.4%	19856.2
train4	391.0	100.0%	1126.0	4002.1	100.0%	11521.6

* Incomplete results

Problems with the implementation of the POW3 algorithm result in a lot of missing results. For the remaining results, this approach is on average 7.6% worse than our heuristic approach.

4.4.5 Nöth e.a.

The spanning tree based state assignment algorithm by Nöth and Kolla [6] has been implemented utilizing four different heuristics. The best results are shown in Table 4.7.

Table 4.7: Nöth e.a. average switching activity

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
bbara	288.8[34]	126.6%[34]	4083.2[34]	2771.7[34]	125.4%[34]	40313.2[34]

[!] Wider register [1]fastk [2]fastp [3]greedyk [4]greedyp

Table 4.7: Nöth e.a. average switching activity (Continued)

Benchmark	**1000 Input Vectors Data Set**			**10000 Input Vectors Data Set**		
	State Register		Circuit	State Register		Circuit
bbsse	792.4[12!]	116.0%[12!]	8969.2[12!]	7832.4[12!]	116.1%[12!]	89086.0[12!]
bbtas	438.4[34]	100.0%[34]	1737.9[34]	4459.7[34]	100.0%[34]	17810.8[34]
beecount	440.9[34]	105.5%[34]	3059.9[34]	4301.0[34]	105.5%[34]	30237.2[34]
cse	241.1[12!]	105.0%[12!]	9868.2[12!]	2380.0[12!]	104.7%[12!]	97521.8[12!]
dk14	1108.6[34]	133.9%[34]	13660.8[34]	11061.0[34]	134.3%[34]	136527.9[34]
dk15	824.4[1234!]	116.7%[1234!]	9350.3[1234!]	8286.8[1234!]	117.2%[1234!]	93768.8[1234!]
dk16	1686.4[34!]	175.3%[34!]	37174.4[34!]	16565.7[34!]	172.2%[34!]	348515.0[34!]
dk17	1033.4[12!]	123.4%[12!]	7180.4[12!]	10341.3[12!]	123.3%[12!]	71507.5[12!]
dk27	1193.3[34]	119.3%[34]	4676.1[34]	11901.0[34]	119.0%[34]	46669.8[34]
dk512	1253.2[34!]	125.3%[34!]	10714.5[34!]	12510.5[34!]	125.1%[34!]	106848.9[34!]
donfi le	1344.1[3!]	178.1%[3!]	-	12869.9[4!]	171.4%[4!]	-
ex1	697.1[34!]	135.0%[34!]	10419.6[34!]	6971.3[34!]	134.5%[34!]	104297.7[34!]
ex4	-	-	-	-	-	-
ex6	-	-	-	-	-	-
keyb	548.7[2!]	101.2%[2!]	12906.8[2!]	5570.8[1!]	101.4%[1!]	164202.8[1!]
kirkman	501.1[34]	100.0%[34]	7526.8[34]	4995.0[34]	100.0%[34]	75237.3[34]
lion	380.5[1234]	100.0%[1234]	1586.2[1234]	3734.0[1234]	100.0%[1234]	15568.4[1234]
lion9	446.0[12]	100.0%[12]	1035.7[12]	4442.1[12]	100.0%[12]	10248.1[12]
mark1	-	-	-	-	-	-
mc	423.3[1234]	100.0%[1234]	2312.2[1234]	4288.9[1234]	100.0%[1234]	23307.6[1234]
modulo12	-	-	-	-	-	-
opus	-	-	-	-	-	-
planet	1124.1[34!]	117.2%[34!]	47717.5[34!]	11243.5[34!]	117.1%[34!]	473383.3[34!]
pma	541.5[34!]	122.3%[34!]	7761.1[34!]	5330.5[34!]	122.3%[34!]	76474.8[34!]
s1	1091.2[34]	148.9%[34]	25409.4[34]	10886.3[34]	148.9%[34]	253558.2[34]
s1488	366.9[34!]	121.0%[34!]	32367.9[34!]	3489.7[34!]	120.5%[34!]	316897.0[34!]
s1494	366.9[34!]	121.0%[34!]	32444.2[34!]	3489.7[34!]	120.5%[34!]	317915.2[34!]
s208	-	-	-	-	-	-
s27	906.4[34]	132.3%[34]	3402.0[34]	8999.0[34]	132.0%[34]	33839.4[34]
s298	1155.4[3!]	155.0%[3!]	149217.3[3!]	11708.6[3!]	155.6%[3!]	-
s386	783.5[34!]	116.3%[34!]	8296.9[34!]	7790.3[34!]	116.1%[34!]	82852.7[34!]
s420	-	-	-	-	-	-
s510	675.7[34]	103.9%[34]	15605.6[34]	6992.8[34]	104.3%[34]	158746.8[34]
s8	143.9[1234]	115.4%[1234]	-	1478.5[1234]	113.7%[1234]	-
s820	553.0[3!]	103.2%[3!]	14725.6[3!]	5544.4[3!]	103.2%[3!]	147984.6[3!]
s832	553.0[4!]	103.2%[4!]	14823.3[4!]	5544.4[4!]	103.2%[4!]	148720.0[4!]
sand	563.5[34!]	118.4%[34!]	36485.8[34!]	5702.9[34!]	117.7%[34!]	368743.5[34!]
scf	-	-	-	-	-	-
shiftreg	999.9[34!]	114.0%[34!]	5078.9[34!]	9998.7[4!]	114.3%[4!]	50680.9[4!]
sse	792.4[12!]	116.0%[12!]	8969.2[12!]	7832.4[12!]	116.1%[12!]	89086.0[12!]
styr	577.7[4!]	113.0%[4!]	16185.5[4!]	5732.7[4!]	113.0%[4!]	161929.4[4!]
tav	1000.0[1234]	100.0%[1234]	2499.7[1234]	10000.0[1234]	100.0%[1234]	25000.2[1234]
tbk	774.6[4!]	136.3%[4!]	21086.0[4!]	7707.0[4!]	134.8%[4!]	208908.6[4!]
tma	213.1[34]	129.5%[34]	2460.5[34]	2014.8[34]	128.2%[34]	23511.6[34]
train11	407.4[1!]	122.5%[1!]	2037.2[1!]	4150.4[1!]	124.4%[1!]	20854.4[1!]
train4	391.0[1234]	100.0%[1234]	2101.3[1234]	4002.1[1234]	100.0%[1234]	21524.3[1234]

[!] Wider register [1]fastk [2]fastp [3]greedyk [4]greedyp

This algorithm utilizes (where needed) wider state registers, which relieves one of the contraints of the other low power state assignment algorithms. This allows the algorithm to obtain lower state register switching activities, at the cost of a larger chip area for the state register. Therefore, this does not allow of a fair comparison to the other algorithms.

The results for the **dk15** becnhmark show that a wider state register can decrease the state register switching activity below that possible with a minimum width state register. Thus, this algorithm demonstrates the possibilities of dynamic latch allocation for a more advanced state assignment heuristic.

The Greedy approach produces better results than the fast approach for most of the benchmarks, as shown by Nöth and Kolla [6].

4.4.6 Pow3

This section discusses the results from the POW3 [1] algorithm utilizing static state transition possibilities, as presented in Table 4.8.

Table 4.8: Pow3 average switching activity

Benchmark	**1000 Input Vectors Data Set**			**10000 Input Vectors Data Set**		
	State Register		Circuit	State Register		Circuit
bbara	288.8	126.6%	4083.2	2771.7	125.4%	40313.2
bbsse	793.4	116.1%	7724.8	7834.2	116.1%	76377.2
bbtas	552.7	126.1%	2764.9	5636.0	126.4%	28348.7
beecount	439.9	105.3%	3357.5	4299.4	105.5%	33278.0
cse	248.5	108.2%	8260.9	2450.2	107.8%	81623.8
dk14	1197.2	144.6%	11650.1	11936.2	144.9%	116545.9
dk15	844.2	119.5%	10611.3	8497.0	120.1%	106056.2
dk16	1849.4	192.2%	36950.3	18363.2	190.8%	367778.2
dk17	1033.8	123.5%	7910.0	10361.2	123.5%	79467.4
dk27	1356.9	135.7%	4612.5	13559.4	135.6%	45947.4
dk512	1530.8	153.1%	-	15341.3	153.4%	95720.7
donfi le	1582.2	209.7%	-	15813.7	210.6%	-
ex1	806.1	156.2%	10927.7	8023.3	154.8%	108887.3
ex4	-	-	-	-	-	-
ex6	-	-	-	-	-	-
keyb	548.7	101.2%	18855.9	5571.4	101.4%	189771.9
kirkman	-	-	-	-	-	-
lion	380.5	100.0%	1571.0	3734.0	100.0%	15463.7
lion9	716.1	160.6%	3326.5	7242.9	163.0%	33951.2
mark1	-	-	-	-	-	-
mc	423.3	100.0%	2266.8	4288.9	100.0%	22761.7
modulo12	677.0	133.3%	-	6673.5	133.3%	-
opus	-	-	-	-	-	-
planet	1572.2	164.0%	53341.3	15771.2	164.2%	532587.9
pma	-	-	-	-	-	-
s1	1287.3	175.6%	30729.3	12774.6	174.7%	306846.3
s1488	-	-	-	3363.1	116.2%	275184.8
s1494	-	-	-	-	-	-
s208	-	-	-	-	-	-
s27	906.4	132.3%	3856.2	8999.0	132.0%	38367.5

Table 4.8: Pow3 average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
s298	-	-	-	-	-	-
s386	-	-	-	-	-	-
s420	-	-	-	-	-	-
s510	-	-	-	-	-	-
s8	167.7	134.3%	-	1732.1	133.1%	-
s820	-	-	-	5536.6	103.1%	195068.2
s832	-	-	-	5536.8	103.1%	176861.2
sand	743.7	156.3%	33327.3	7577.5	156.4%	336131.7
scf	-	-	-	-	-	-
shiftreg	1371.5	156.4%	4754.2	13742.3	157.0%	47510.3
sse	-	-	-	-	-	-
styr	573.7	112.2%	17873.7	5693.9	112.2%	176567.3
tav	1000.0	100.0%	2499.7	10000.0	100.0%	25000.2
tbk	1072.6	188.7%	31939.6	10997.7	192.4%	322405.9
tma	239.9	145.7%	2480.3	2283.4	145.3%	23689.7
train11	504.3	151.4%	2087.0	5005.1	150.0%	20607.6
train4	586.2	149.9%	1364.6	6002.7	150.0%	14003.1

The POW3 algorithm performs on average 9.6% worse than our heuristic, and worse than Nöth and Kolla's algorithm. The profiling-based POW3 results demonstrate that a profiling-based approach can obtain better results than a static approach.

4.4.7 Jedi

The JEDI [4] state assignment program, part of SIS [2], supports several area-oriented state assignment algorithms. We opted for the default **output dominant** algorithm. Because this approach is area-oriented, the circuit switching activity rather than the register switching activity is the result to compare.

Table 4.9: Jedi average switching activity

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
bbara	324.8	142.4%	4642.8	3153.2	142.6%	45483.3
bbsse	1045.2	153.0%	13509.4	10369.5	153.7%	133417.1
bbtas	592.5	135.1%	2822.2	6023.0	135.1%	28812.5
beecount	449.7	107.6%	3776.6	4396.4	107.8%	37497.3
cse	347.3	151.2%	10461.8	3463.8	152.4%	103458.0
dk14	1409.2	170.3%	15624.3	14070.1	170.8%	155916.6
dk15	983.3	139.1%	11397.5	9807.8	138.7%	113570.6
dk16	2427.9	252.3%	34679.2	24060.6	250.1%	345452.7
dk17	1272.0	151.9%	10688.4	12678.7	151.2%	106661.0
dk27	1784.0	178.4%	5101.9	17867.0	178.7%	50989.6
dk512	1930.9	193.1%	10270.8	19318.8	193.2%	102189.5
donfi le	1842.5	244.2%	-	18348.6	244.3%	-
ex1	1119.0	216.8%	14761.5	11196.9	216.0%	147934.4

Table 4.9: Jedi average switching activity (Continued)

Benchmark	1000 Input Vectors Data Set			10000 Input Vectors Data Set		
	State Register		Circuit	State Register		Circuit
ex4	905.4	206.0%	5023.3	8915.8	205.1%	49962.6
ex6	1267.1	157.1%	11381.4	12724.6	158.6%	113984.8
keyb	643.1	118.6%	25679.2	6505.2	118.4%	258140.5
kirkman	814.7	162.6%	9567.4	8117.0	162.5%	95420.5
lion	510.9	134.2%	1696.1	4981.5	133.4%	16420.6
lion9	547.3	122.8%	1031.2	5587.4	125.8%	10709.5
mark1	1797.7	179.8%	9624.2	17977.7	179.8%	96291.0
mc	423.3	100.0%	2266.8	4288.9	100.0%	22761.7
modulo12	507.8	100.0%	-	5005.3	100.0%	-
opus	1070.2	145.3%	7483.6	10690.6	145.6%	74764.7
planet	3249.4	338.9%	66576.3	32540.5	338.9%	665284.7
pma	899.8	203.2%	12614.4	8851.0	203.0%	123867.8
s1	1338.0	182.5%	23696.2	13316.3	182.1%	236549.6
s1488	638.8	210.7%	26050.6	6102.8	210.8%	254273.1
s1494	639.1	210.8%	31213.9	6099.4	210.7%	304712.5
s208	521.9	114.7%	7282.9	5010.6	114.2%	70658.1
s27	916.8	133.8%	2638.9	9104.6	133.6%	26294.3
s298	2059.2	276.1%	107735.8	20905.0	277.8%	-
s386	908.4	134.9%	8838.3	8962.6	133.6%	87722.3
s420	626.9	137.8%	8769.1	5999.6	136.7%	84705.8
s510	1510.0	232.1%	26310.1	15481.5	230.8%	268001.0
s8	143.9	115.4%	-	1478.5	113.7%	-
s820	1585.4	296.0%	24116.0	15902.2	296.1%	242430.0
s832	1579.8	295.0%	28877.1	15842.4	295.0%	289800.9
sand	750.7	157.7%	39191.8	7665.9	158.3%	394476.4
scf	3661.2	366.1%	80041.1	36654.0	366.5%	799571.7
shiftreg	1500.1	171.0%	2496.5	15004.7	171.5%	24999.8
sse	1045.2	153.0%	13509.4	10369.5	153.7%	133417.1
styr	613.7	120.0%	27833.6	6086.7	120.0%	-
tav	1500.0	150.0%	3498.7	15000.0	150.0%	34999.2
tbk	1013.2	178.3%	15418.6	10345.7	181.0%	155867.4
tma	376.5	228.9%	4129.1	3597.9	229.0%	39708.4
train11	510.5	153.3%	1985.3	5039.5	151.1%	19383.9
train4	391.0	100.0%	1959.5	4002.1	100.0%	20046.8

The state register switching activities obtained by this area-oriented approach are generally significantly worse than the results from the other algorithms. On average, the results from JEDI are 41% worse than the results for our heuristic. Only for the **s27** benchmark is JEDI's result better.

4.4.8 Comparison

FSM state assignment algorithms for low power target the power dissipation of FSMs through the switching activity in the state register, an approach shared by our algorithms as well as the Nöth and Kolla and POW3 algorithms. Therefore, a comparison of the resulting state register switching activity is a valid means to determine the effectiveness of our algorithms.

As the individual results show, the difference in the results for the 1000 and the 10000 input vector data sets is insignificant. Therefore, we compare the algorithms by the average of all 1000 and 10000 input vector state register switching activity results. Table 4.10 shows the resulting average state register switching activity for all algorithms. Incomplete results are excluded from the overall average and the number of best results.

Table 4.10: Overall state register switching activity

Benchmark	**DFS**	**Loop DFS**	**Heuristic**	**Dynamic Heuristic**	**Profiling POW3**	**Nöth**	**POW3**	**JEDI**
bbara	**123.8**%	129.0%	127.5%	←	127.9%*	126.0%	126.0%	142.5%
bbsse	**115.1**%	117.0%	116.0%	←	-	116.0%!	116.1%	153.3%
bbtas	**100.0**%	**100.0**%	**100.0**%	←	102.5%	**100.0**%	126.3%	135.1%
beecount	**105.3**%	107.4%	105.5%	←	105.5%*	105.5%	105.4%	107.7%
cse	**104.6**%	105.8%	105.8%	←	-	104.9%!	108.0%	151.8%
dk14	**134.1**%	138.3%	142.3%	←	145.2%	**134.1**%	144.8%	170.5%
dk15	119.8%	125.7%	124.6%	122.5%	119.8%	**116.9**%!	119.8%	138.9%
dk16	-	199.0%	201.5%	204.9%	196.7%*	**173.7**%!	191.5%	251.2%
dk17	**122.8**%	133.6%	125.9%	143.4%	133.4%	123.4%!	123.5%	151.6%
dk27	**118.8**%	141.3%	132.1%	←	125.6%	119.2%	135.6%	178.5%
dk512	**118.4**%	158.3%	146.9%	161.2%	147.1%	125.2%!	153.2%	193.1%
donfile	-	193.7%	206.1%	223.1%	205.1%	**174.8**%!	210.1%	244.3%
ex1	**129.0**%	143.3%	141.7%	←	-	134.8%!	155.5%	216.4%
ex4	**110.3**%	114.4%	114.2%	←	114.6%	-	-	205.5%
ex6	**125.8**%	148.4%	130.8%	←	127.2%	-	-	157.9%
keyb	**101.3**%	102.0%	102.0%	←	-	**101.3**%!	**101.3**%	118.5%
kirkman	**100.0**%	**100.0**%	**100.0**%	←	112.4%	**100.0**%	-	162.5%
lion	**100.0**%	109.7%	**100.0**%	←	109.7%	**100.0**%	**100.0**%	133.8%
lion9	**100.0**%	121.5%	139.5%	←	120.7%	**100.0**%	161.8%	124.3%
mark1	**130.2**%	134.4%	137.3%	136.0%	137.6%	-	-	179.8%
mc	**100.0**%	**100.0**%	**100.0**%	←	**100.0**%	**100.0**%	**100.0**%	**100.0**%
modulo12	**100.0**%	**100.0**%	**100.0**%	←	**100.0**%	-	133.3%	**100.0**%
opus	**127.3**%	137.9%	132.4%	132.1%	139.8%	-	-	145.5%
planet	-	-	127.8%	128.0%	152.4%*	**117.2**%!	164.1%	338.9%
pma	**121.4**%	128.2%	130.6%	←	-	122.3%!	-	203.1%
s1	-	176.6%	173.9%	180.2%	165.6%	**148.9**%	175.2%	182.3%
s1488	114.0%*	129.1%	**115.0**%	←	-	120.8%!	116.2%*	210.7%
s1494	114.1%*	129.1%	**115.0**%	←	-	120.8%!	-	210.7%
s208	**108.6**%	114.2%	**108.6**%	←	-	-	-	114.5%
s27	**130.1**%	146.2%	148.0%	←	132.9%	132.2%	132.2%	133.7%
s298	-	154.5%*	**155.2**%	←	-	155.3%!	-	277.0%
s386	**115.2**%	117.7%	116.2%	←	-	116.2%!	-	134.2%
s420	**108.6**%	114.2%	**108.6**%	←	-	-	-	137.3%
s510	-	-	**104.1**%	←	119.3%	**104.1**%	-	231.4%
s8	**114.5**%	117.4%	**114.5**%	←	133.7%	**114.5**%	133.7%	**114.5**%
s820	**101.8**%	103.1%	103.0%	←	-	103.2%!	103.1%*	296.1%
s832	**101.8**%	103.1%	103.0%	←	-	103.2%!	103.1%*	295.0%
sand	-	128.3%*	135.9%	141.6%	172.4%*	**118.1**%!	156.4%	158.0%
scf	-	-	**121.8**%	122.1%	-	-	-	366.3%
shiftreg	**114.1**%	146.0%	134.9%	←	148.6%	**114.1**%!	156.7%	171.2%

* Incomplete results ! Wider register

Table 4.10: Overall state register switching activity (Continued)

Benchmark	DFS	Loop DFS	Heuristic	Dynamic Heuristic	Profiling POW3	Nöth	POW3	JEDI
sse	**115.1**%	117.0%	116.0%	←	-	116.0%[!]	-	153.3%
styr	**108.0**%	108.8%	112.2%	←	-	113.0%[!]	112.2%	120.0%
tav	**100.0**%	**100.0**%	**100.0**%	←	**100.0**%	**100.0**%	**100.0**%	150.0%
tbk	-	178.9%	182.7%	**130.5**%	174.1%	135.5%[!]	190.6%	179.6%
tma	**114.6**%	118.6%	120.6%	143.0%	144.1%*	128.9%	145.5%	228.9%
train11	**122.4**%	**122.4**%	123.4%	←	125.6%	123.5%[!]	150.7%	152.2%
train4	**100.0**%	**100.0**%	**100.0**%	←	**100.0**%	**100.0**%	150.0%	**100.0**%
Average	112.9%	127.7%	125.6%	126.3%	129.6%	119.6%	139.3%	177.1%
Best results	35	7	15	15	4	18	4	4

* Incomplete results   ! Wider register

Except for our heuristic and JEDI, the implementations all fail for one or more benchmarks, therefore the overall average and the number of best results do not allow a direct comparison, but only provide an indication. Furthermore, the dynamic latch allocation heuristic and Nöth and Kolla's algorithm produce codes for wider state registers. When compared to the other algorithms, this allows for a lower state register switching activities at the cost of extra chip area. Therefore, the results of these algorithms are not suitable for a fair comparison.

The DFS state assignment algorithm is clearly superior to all other algorithms with respect to the lowest average register switching activity. When we only compare the complete results, the average state register switching activity result of the FSM-based DFS is 6.6% lower than the next best algorithm, our heuristic. However, the DFS algorithm is computationally time consuming, and even fails to produce (within reasonable time) results for the larger FSMs. The loop-based DFS state assignment algorithm suffers from much the same problem, and on average it's results are worse than the results of our heuristic.

Our loop-based heuristic state assignment approach is very fast, and with the exception of the DFS state assignment algorithm, the heuristic achieves better results than all the other algorithms with a fixed width state register. When we only compare the valid results, the average state register switching activity result of the heuristic is 8.5% lower than the state of the art POW3 algorithm, and 41% lower than the area-based JEDI approach.

When we compare our heuristic to Nöth and Kolla's approach width a variable state register width, our average state register switching activity result is 6.3% higher. Their lower average can be contributed largely to the benchmarks for which a wider register is used. Our preliminary stage dynamic heuristic fails to take advantage of variable state register width, and it's average result is worse than that of the standard heuristic. However, Nöth and Kolla's approach shows the improvements possible using a wider register.

The profiling-based POW3 results show the improvement profiling brings for some benchmarks when compared to the original static approach. However, the results for other benchmark indicate that the POW3 algorithm is not optimally suited to the profiling-based approach.

The actual objective of state assignment for low power dissipation is not to reduce the state register switching activity, but to lower the power consumption of the resulting circuit. The circuit switching activity is a good measure for the real circuit's power dissipation. To be able to average the results of the 1000 and 10000 input vector data sets, we divide the measured circuit

switching activity by the length of the data sets to obtain the average circuit switching per input vector. Table 4.11 shows the resulting average circuit switching activity for all algorithms.

Table 4.11: Overall circuit switching activity

Benchmark	DFS	Loop DFS	Heuristic	Dynamic Heuristic	Profiling POW3	Nöth	POW3	JEDI
bbara	**4.02**	4.16	4.08	←	4.06*	4.06	4.06	4.60
bbsse	8.65	8.32	8.85	←	-	8.94	**7.68**	13.43
bbtas	1.98	2.25	2.03	←	2.22	**1.76**	2.80	2.85
beecount	3.37	3.46	3.38	←	3.16*	**3.04**	3.34	3.76
cse	8.88	9.14	8.72	←	-	9.81	**8.21**	10.40
dk14	15.09	14.49	15.37	←	13.57	13.66	**11.65**	15.61
dk15	**9.29**	9.86	9.99	10.14	10.03	9.36	10.61	11.38
dk16	-	35.29	37.47	36.90	37.80*	36.01	36.86	**34.61**
dk17	9.06	8.62	7.83	9.91	8.53	**7.17**	7.93	10.68
dk27	4.82	5.57	4.69	←	**4.50**	4.67	4.60	5.10
dk512	**9.60**	11.75	11.19	11.24	10.90	10.70	9.57*	10.24
donfile	-	-	-	-	-	-	-	-
ex1	**9.82**	10.03	9.89	←	-	10.42	10.91	14.78
ex4	2.99	**2.71**	3.00	←	3.24	-	-	5.01
ex6	10.13	12.23	**9.41**	←	10.06	-	-	11.39
keyb	16.26	17.94	17.94	←	-	**14.66**	18.92	25.75
kirkman	9.29	9.29	8.41	←	8.20	**7.53**	-	9.55
lion	1.57	1.60	1.58	←	1.60	1.57	**1.56**	1.67
lion9	1.49	1.94	2.49*	←	1.90	**1.03**	3.36	1.05
mark1	8.64	8.58	**8.29**	8.62	8.58	-	-	9.63
mc	2.32	2.32	2.32	←	2.32	2.32	**2.27**	**2.27**
modulo12	-	-	-	-	-	-	-	-
opus	**7.09**	8.70	7.49	7.51	9.23	-	-	7.48
planet	-	-	35.15*	38.92*	50.33*	**47.53**	53.30	66.55
pma	8.86	9.29	9.16	←	-	**7.70**	-	12.50
s1	-	30.39	33.16	35.95	33.28*	25.38	30.71	**23.68**
s1488	26.78*	26.90*	27.33	←	-	32.03	27.52*	**25.74**
s1494	27.53*	**26.49**	26.69	←	-	32.12	-	30.84
s208	**5.62**	6.04	6.00	←	-	-	-	7.17
s27	3.54	4.86	4.50	←	3.96	3.39	3.85	**2.63**
s298	-	93.20*	129.74*	←	-	149.22*	-	107.74*
s386	9.00	9.01	9.51	←	-	**8.29**	-	8.81
s420	5.64	**5.48**	6.01	←	-	-	-	8.62
s510	-	-	**12.51**	←	16.30	15.74	-	26.56
s8	-	-	-	-	-	-	-	-
s820	17.76	19.56	17.74	←	-	**14.76**	19.51*	24.18
s832	15.43	16.12	18.38	←	-	**14.85**	17.69*	28.93
sand	-	35.88*	37.42	37.52	39.90*	36.68	**33.47**	39.32
scf	-	-	39.99*	28.16*	-	-	-	**80.00**
shiftreg	4.44	4.88	4.80	←	5.54	5.07	4.75	**2.50**
sse	8.65	**8.32**	8.85	←	-	8.94	-	13.43
styr	24.67	24.95*	25.21	←	-	**16.19**	17.77	27.83*
tav	**2.50**	**2.50**	**2.50**	←	**2.50**	**2.50**	**2.50**	3.50
tbk	-	33.27	33.87	21.45	33.89	20.99	32.09	**15.50**

* Incomplete results

Table 4.11: Overall circuit switching activity (Continued)

Benchmark	**DFS**	**Loop DFS**	**Heuristic**	**Dynamic Heuristic**	**Profiling POW3**	**Nöth**	**POW3**	**JEDI**
tma	**2.37**	2.52	2.52	2.53	2.73*	2.41	2.42	4.05
train11	2.12	2.08	2.32	←	2.08	2.06	2.07	**1.96**
train4	**1.14**	**1.14**	**1.14**	←	**1.14**	2.13	1.38	1.98
Average	7.53	10.01	11.79	11.60	7.63	12.37	12.27	15.23
Best results	9	6	5	5	3	13	7	9

* Incomplete results

Due to the missing and incomplete results, a direct comparison of the overall average and the number of best results is not possible, so these values only provide an indication. The low average switching activity of the FSM-based DFS and profiling-based POW3 approaches is mainly the result of a number of missing high switching activity results.

Our experiments indicate that no algorithm is able to achieve consistent low circuit switching activity. On average, the low-power approaches do not perform better than the area-oriented JEDI approach. Furthermore, there is no correlation between the best state register results and the best circuit switching activity results. Therefore, it seems that the state register switching activity is not a good measure for the circuit switching activity.

Overall, the results of the current state assignment algorithms are close together, and the results for the DFS approach show that there is not much room for further improvement when only the state register switching activity is considered as metric. This clearly suggest that switching activity within the combinatorial circuit should be also included in the cost function utilized in algorithms for low power FSM state assignment.

Conclusions 5

In this chapter we summarize the results of our work, highlight our main contributions to the field of finite state machine state encoding for low power dissipation, and discuss future work for our approach.

5.1 Summary

Chapter 2 discusses the background of finite state machine (FSM) state assignment for low power dissipation.

First, we presented several considerations related to power consumption in CMOS digital circuits. Next, we introduced FSMs, and we explained FSM state assignment, followed by a typical design approach for FSM circuits, in relation to FSM power consumption. Then we explained the terminology used in our thesis.

Finally, we presented an historical overview of state assignment algorithms, and we discussed two state of the art low power FSM state assignment algorithms.

In Chapter 3, we proposed a novel low power FSM state assignment approach that consists of three steps:

- **FSM state profiling**: collects dynamic data related to the FSM behavior in the form of a state trace.
- **Loop detection algorithm**: finds the loops in the state trace.
- **FSM state assignment**: generates a state encoding which minimizes the state register switching activity utilizing the data gathered in the first two steps.

First, we discussed three strategies for the detection of nested loops. Based on this discussion, we proposed a linear search loop detection algorithm that separately detects simple loops within nested loops. The algorithm recognizes duplicate loops, and counts the frequency of occurrences for each loop.

Second, we assumed that the state register switching activity is the cost metric for the low power FSM assignment, and that the width of the state register is fixed to the minimal width required for a valid encoding, and we proposed three loop-based state assignment algorithms as follows:

- **DFS** performs an exhaustive (depth-first) search of all possible encodings of the FSM, using the loop data to estimate the intermediate cost of an encoding.
- **Loop-based DFS** performs an exhaustive search of all possible encodings per loop, in descending-weight order of the loops, and with a loop's weight equal to it's frequency.

- **Heuristic** assigns the states in the order of occurrence in the loops while minimizing the cost of the state transitions for that state in the loop.

For **DFS**, we additionaly presented an optimization which utilizes an intermediate cost estimate to reduce the search space.

For **Heuristic**, we proposed three weight functions for the loops:

- $weight = frequency$,
- $weight = frequency \times \#states$, where $\#states$ is the number of states in the loop, and
- $weight = frequencybits$, where $\#bits = \lceil \log_2 \#states \rceil$.

Based on **Heuristic**, we also proposed the preliminary stage **Dynamic Heuristic**, that dynamically increases the state register width in an attempt to further reduce the state register switching activity.

Finally, we described the C^{++} framework that we created to implement the proposed state assignment approach.

In Chapter 4, we proposed an experimental method for the evaluation of FSM state assignment algorithm for low power. This method consist of the following steps:

- Setup of the FSM benchmarks from the MCNC/LGSynth '89 FSM benchmark suite [3], including the generation of random input vector data sets.
- FSM profiling and loop detection.
- FSM state assignment.
- Synthesis of a gate-level circuit using SIS [2].
- Simulation of the resulting circuit by Mercury [5].

We discussed the characteristics of the benchmark FSMs, followed by the results of the experiments. We compared **DFS**, **Loop-based DFS**, and **Heuristic** with the power-based POW3 [1] algorithm, as well as the area-based JEDI [4] algorithm.

DFS produced the lowest average state register switching activity results of all the algorithms with a restricted state register width, but the algorithm failed to produce results for large FSMs due to it's complexity. The same holds true for **Loop-based DFS**. **Heuristic**'s approach was very fast, and it achieved better results than the POW3 and JEDI algorithms with respectively 8% and 41% lower average state register switching activities.

The variable state register width Nöth and Kolla algorithm [6], although it requires a larger state register, then more area, achieved a 6% reduction when compared with our fixed width heuristic. This suggests that state algorithms for low power dissipation should use a variable state register width approach to achieve the largest possible reduction in state register switching activity. Our preliminary **Dynamic Heuristic** is at a too early stage of development to be able to match the results of Nöth and Kolla's algorithm.

Our experiments indicated that no current state assignment algorithm that utilizes state register switching activity as metric for power minimization is able to achieve a consistent reduction in power consumption. This clearly suggests that the switching activity in the state register only is not a suitable metric to reduce the power consumption in FSMs. Instead, a metric should be used that also reflects the switching activity in the combinatorial circuit.

5.2 Main contributions

The main contributions can be summarized as follows:

- We have presented a novel loop-based profiling FSM state assignment approach for low power dissipation consisting of three steps:
  - FSM profiling,
  - Loop detection, and
  - FSM state assignment.
- We have proposed a linear search loop detection algorithm that separately detects simple loops within nested or intersecting loops.
- We have proposed three loop-based FSM state assignment algorithms that minimize the switching activity in the state register:
  - **DFS** performs an exhaustive search of all possible encodings of the FSM, and uses the loop data to obtain an intermediate estimate of the cost of an encoding.
  - **Loop-based DFS** performs an exhaustive search of all possible encodings for a single loop, for all loops in descending-weight order.
  - **Heuristic** assigns the FSM states on-by-one, in the order of occurrence of the states within the loops, while the loops are sorted in descending-weight order. This heuristic minimizes the cost of a state assignment for the state transitions to and from that state in the current loop.
- We have developed a general C^{++} framework to implement FSM profiling, loop detection and FSM state assignment algorithms.
- We have proposed a method to compare different state assignment algorithms based upon state register and circuit switching activity.

In order to evaluate the efficiency of our proposal we compared our approach with other state of the art FSM state assignment methods. Our experimental results indicated the following:

- For fixed width state registers, our heuristic state assignment approach showed an 8% reduction in average state register switching activity when compared to the power-based POW3 [1] algorithm, and a 41% reduction when compared to the area-based JEDI [4] algorithm.
- The variable state register width Nöth and Kolla algorithm [6], although it requires a larger state register, then more area, achieved a 6% reduction when compared with our fixed width heuristic. This suggests that state algorithms for low power dissipation should use a variable state register width approach to achieve the largest possible reduction in state register switching activity. Our preliminary **Dynamic Heuristic** is at a too early stage of development to be able to match the results of Nöth and Kolla's algorithm.

- Our experiments indicated that no current state assignment algorithm that utilizes state register switching activity as metric for power minimization is able to achieve a consistent reduction in power consumption. This clearly suggests that the switching activity in the state register only is not a suitable metric to reduce the power consumption in FSMs. Instead, a metric should be used that also reflects the switching activity in the combinatorial circuit.

5.3 Future work

This study of loop-based profiling FSM state assignment methods leaves some questions unanswered, and some further possibilities to be explored.

The most important question is the suitability of the state register switching activity as a measure for the circuit switching activity, a metric that is used by all the current power-based state assignment algorithms. An alternative metric should be devised that is easy to be evaluated and reflects the combinatorical switching activity.

The experimental results show that wider state registers can provide an extra reduction of the state register switching activity. However, our preliminary stage **Dynamic Heuristic** approach is unable to take full advantage of this, therefore the heuristic should be developed further.

We have described several methods for the loop detection, and chosen one based on our assumptions. These assumptions should be verified, and based on the conclusions, another method for the loop detection might be devised.

Finally, we obtained the results from the experiments using randomly generated input sequences. However, we believe that realistic input data might improve the efficiency of the approach we proposed. Therefore, the experiments should be repeated using FSMs with actual input sequences.

Bibliography

[1] L. Benini and G. Micheli. State assignment for low power dissipation. *IEEE Journal of Solid-State Circuits*, 30:258–268, March 1995. Available from World Wide Web: `http://citeseer.nj.nec.com/benini95state.html`.

[2] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1992. Available from World Wide Web: `http://citeseer.nj.nec.com/sentovich92sis.html`.

[3] LGSynth89 Benchmark Suite [online]. Available from World Wide Web: `http://www.cbl.ncsu.edu/CBL_Docs/lgs89.html`.

[4] B. Lin and A.R. Newton. Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages. In *Proceedings of the International Conference on VLSI*, pages 187–196, Munich, August 16 - 18 1989.

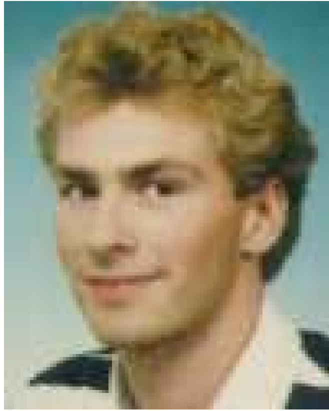
[5] G. De Micheli, D.C. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System for Digital Design. *IEEE Design and Test of Computers*, pages 37–53, 1990. Available from World Wide Web: `http://akebono.stanford.edu/users/cad/synthesis/olympus/doc/olympus.ps`.

[6] Winfried Nöth and Reiner Kolla. Spanning Tree Based State Encoding for Low Power Dissipation. Technical report, Department of Computer Science, University of Würzburg, 1998. Available from World Wide Web: `http://citeseer.nj.nec.com/25222.html`.

[7] R. C. Prim. Shortest connection networks and some generalizations. Technical report, Bell Systems Technical Journal, 1957.

[8] Tiziano Villa and Alberto L. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-level Logic Implementations. In *Proceedings of the 1989 26th ACM/IEEE conference on Design automation conference*, pages 327–332, 1989.

Curriculum Vitae



Robbert Eggermont was born in Amsterdam, the Netherlands, on November 12 1973. He attended the Alfrink College high school in Zoetermeer, from which he graduated in 1992. In the same year, he was admitted to Electrical Engineering faculty of the Delft University of Technology in the Netherlands. After receiving his Bachelor degree, he joined the Computer Engineering laboratory, led by professor Stamatis Vassiliadis, to start his MSc graduation project under the supervision of professor Sorin Cotofana. His thesis was titled **PROSA: Profiling-based State Assignment for Low Power Dissipation**. His research interests include computer architecture and networks.