NTNU

Norwegian University of
Science and Technology

Power optimized multipliers

Stian Mathiassen

Master of Science in Electronics
Submission date: April 2010
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Johnny Pihl, Atmel Norge

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

In earlier work at NTNU, high speed multipliers have been studied[1]. The result of that work is available as a netlist generator for optimized high speed multiplier structures at http://modgen.dnsalias.com. The cost function here involves minimizing the number of carries fed forward between columns in the multiplier tree structure. This project assignment will investigate how this cost function can be extended to include power cost. It involves working with an existing optimization algorithm coded in C, and extend it with power estimation functionality based on[2].

[1] E. Sand, VLSI architectures for speech recognition, 1994
[2] S. T. Oskuii, Design of Low-Power Reduction-Trees in Parallel Multipliers, 2008

Assignment given: 18. November 2009
Supervisor: Per Gunnar Kjeldsberg, IET

Abstract

Power consumption becomes more important as more devices becomes embedded or battery dependant. Multipliers are generally complex circuits, consuming a lot of energy. This thesis uses Sand's [1] multiplier generator, made for his master thesis, as a basis. It uses tree structures to perform the multiplication, but does not take power consumption into account when generating a multiplier.

By adding power optimization to the generator, multipliers with low energy consumption could be made automatically. This thesis adds different reduction tree algorithms (Wallace [2], Dadda [3] and Reduced Area [4]) to the program, and an optimal algorithm might be found. After the multiplier tree generation, an optimization step is performed, trying to exploit the delay and activity characteristics of the generated multiplier. A simplified version of Oskuii's [5] algorithm is used. To be able to compare the different algorithms with each other, a pre-layout power estimation routine was implemented. The estimator is also used by the post-generation optimization. Since accuracy is important in an estimator, the delay through a multiplier was also investigated.

Taking the previous mentioned steps into account, we are able to get a 10% decrease in overall power reduction in a 0,18/0,15 μm CMOS technology, reported by "IC Compiler". Delay characteristics of a multiplier is also supplied, and can be used by other power estimators.

This thesis shows how to achieve less power consumption in multipliers. It also shows that the delay model is important for estimation purposes, and how an estimator is used to optimize a multiplier. The findings in this thesis can be used as is, or be used as a basis for further study.

Preface

This thesis is written as the final assignment for my Master's degree. The assignment was first given to me as a project assignment, corresponding to half a semester's work, and I chose the subject it involved C programming and optimization routines, which i find exciting.

Most of the research for this thesis was done by reading the work of Oskuii, and later further investigating other books and articles about the subject. The supplied code was also read, but some of the code was hard to understand. A lot of work was put in coding the estimator, and the tools trying to extract delay information from post-layout multiplier designs. An optimization routine was also build, using the theory by Oskuii.

Since the thesis uses a previous assignment as basis, some of the content in this thesis are reused from this assignment. The code and proposed algorithms generating multiplier tree are from the previous project. Parts of Chapter 2 are also from that report. The estimator was built by expanding the estimator coded for the previous assignment, but most of the code has been rewritten. Section 6.1.1 and 6.1.2 are also reused from the project assignment.

I want to thank Johnny Pihl and Per Gunnar Kjeldsberg for their advice while I was working on this thesis. They always had ideas and opinions on how I should proceed. Their interest for the field also seemed obvious, when I had to remind them that my time was running out, and I was not able to do everything they suggested. This made the assignment fun to work with. I would also want to thank Karoline Hovstad and Torgeir Thoresen for trying to understand and proof-read my thesis.

Stian Mathiassen

Contents

List of Tables

List of Figures

Source code

List of abbreviations

BDD Binary Decision Diagram [11]

FA Full-adder

HA Half-adder

PP Partial product

PVT Process, voltage and temperature

LFSR Linear feedback shift register [12, 13]

SDF Standard Delay Format [14]

VMA Vector Merging Adder

Chapter 1

Introduction

Multipliers are used in a wide range of devices, from large scale processors to small embedded DSP chips. As multipliers are large, slow and complex components, a lot of research has been done to make the components smaller and faster [2, 3, 4, 15, 16]. As more of the electronic devices becomes embedded or handheld, they become more dependant on battery as a power source. To improve battery life-time, the research focus has shifted to improve power consumption [17, 18, 19, 20, 5]. By reducing power-usage, it is possible to drastically improve the battery-life of handheld devices.

Since tree multipliers, like Wallace [2] and Dadda [3], are faster and use less power than traditional array multipliers [21] (though have larger area), this thesis concentrates its focus on tree multipliers. The wires inside the tree multipliers have very different length, because of the irregular layout of such multipliers, and signal delay and power impacts the design more than in array multipliers. Over half of the used power is because of excess switching, which produces nothing to the end result [22], and should therefore have lot of optimization potential [5].

By employing different kind of tree multiplier generating algorithms, we can find what kind of algorithm performs best. Since there also are a lot of spurious switching activity in multipliers, another way to power optimize multipliers are to reduce these glitches. By altering how the adders inside the multiplier are interconnected, it should be possible to get decreased power usage [5].

This thesis uses six different algorithms to design tree multipliers: Wallace [2], Dadda [3], Reduced area [4], algorithm used by Sand's multiplier generator [1] and algorithms proposed here and in a project prior to this master thesis work [23]. To compare these algorithm before layout, a Monte Carlo approach for power estimation is used [10]. The result of the power estimation is used to power a post-generation optimization of each multiplier using a simplified optimization algorithm proposed by Oskuii [5].

The background theory for the different kind of algorithms that generates the tree multipliers are presented in this thesis, together with theory on power estimation for combinational CMOS circuits. An implementation of five algorithms is added to the existing VHDL-netlister program written in C by Sand [1], to produce a wide range of tree multipliers, will be presented. An implementation of an estimator is also discussed, in addition to an implementation of a simpler optimization routine based in Oskuii's model [5]. The estimator is fed real-world timing delay from an SDF-file[14] of a post-layout multiplier. The delay data is extracted using a parser implemented for this thesis. Using netlister with the implementations from this thesis, multiple 8×8, 16×16 and 32×32 multipliers are generated and compared using the said estimator. Power estimates for two algorithms from post-layout analysis are also presented, to verify if an improvement are found.

1.1 Power Usage in CMOS

The power used in CMOS-circuits consists of two parts, dynamic and static power dissipation [18]:

$$P = P_{dynamic} + P_{static} \tag{1.1}$$

The dynamic power consumption is power used as a function of activity. The static component is power consumed as a function of time.

1.1.1 Static Power Consumption

The part describes power used even though there is no activity in the circuit. Ideally CMOS components should not have any static power consumption, since there are no direct paths from V_{dd} to ground. In practical applications this is not the case, since MOS transistors are not perfect switches. There will always be leakage currents in MOS transistors [18].

Reverse biased currents flows through the source or drain and the substrate, because parasitic diodes in the MOS transistors are one of the static leakage currents. The sub threshold leakages current run through the transistors (from source to drain), because the gate of the transistor is close to the threshold voltage, and therefore lets some current flow through. These currents used to be negligible, however it seems to become more prominent as transistors become smaller [19, 24] and really starts to emerge at 0,13 μm [25]. The static power dissipation is primarily determined by fabrication technology [26].

1.1.2 Dynamic Power Consumption

The dynamic part of the power consumption in CMOS can be divided into two parts [20].

$$P_{dynamic} = P_{short-circuit} + P_{switching} \tag{1.2}$$

The short circuit happens when both the PMOS and the NMOS transistor is open at the same time. This happens during a switch, because the PMOS and NMOS does not switch instantly, but has a switching delay. This makes a short circuit line from V_{dd} to ground through the CMOS component. As we can see in figure 1.1, if the NMOS and PMOS transistors in the inverter are both open at the same time, a short-circuit path i available from V_{dd} to ground. The phenomena is described in equation 1.3, where V_{dd} is the supply voltage and I_{sc} is the current flowing through during the short circuit period of the switch. As long as the inputs of the NMOS and PMOS transistors are properly balanced, this power dissipation should be less than 20% of the dynamic power dissipation [27].

$$P_{short-circuit} = V_{dd} I_{sc} \tag{1.3}$$

The power used in switching the CMOS from one state to another is largely used to charge parasitic capacitance in lines between the CMOS-cells [18]. When the output of a gate is turned from $0 \rightarrow 1$, the NMOS part of the CMOS cuts off the connection to ground, and the PMOS part of the CMOS enables a connection from V_{dd} to the output. This causes the capacitance on the output port and line to be charged, with the energy equal to:

$$\text{Energy}_{\text{transition}} = CV_{dd}^2 \tag{1.4}$$

Where V_{dd} is the power source. Half of this power is dissipated at once in the PMOS transistors, while the other half is stored in the capacitance [18]. When the port is turned

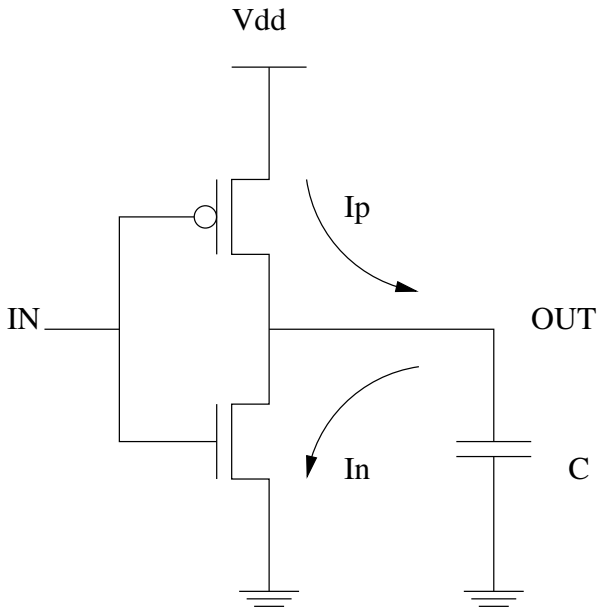

Figure 1.1: Switching power usage in CMOS

from '1' to '0', the line is connected to ground, and the energy stored in the capacitance is also dissipated (see figure 1.1).

Since an equal amount of energy is used to charge the circuit for each $0 \rightarrow 1$ transitions, it is possible to get an equation for power used in switching. Considering the frequency f of the circuit, and the probability for a $0 \rightarrow 1$ switch at the gate α, we get the equation[18]:

$$P_{switching} = \alpha f C V_{dd}^2 \tag{1.5}$$

Although the other sources of power dissipation have increased their share, switching power consumption is still by far the largest source for power usage in CMOS today [27, 18, 22], and is therefore a prime candidate for optimization.

As we can see from the equation, there are three elements to improve power usage: Voltage, physical capacitance and activity. Over the years, lower voltage has been employed in CMOS, causing a reduction in switching power usage [19, 25]. Physical capacitance is strongly correlated to the line length between transistors and the kind of technology being used (size of transistors and lines) [18]. Activity is maybe the most system-dependant factor in the equation. By reducing the activity in the design, it is possible to reduce the amount of power used in the design.

1.1.3 Glitching

So far we have looked at where the power is dissipated. As we have seen, switching activity dictates some of the power usage in CMOS. A problem arises when the inputs on an element do not change at the same time. This might cause the element to use energy two times instead of one. This leads to the problem that some circuits switch more than they need, to reach their final state. This effect is called glitching. This happens when the inputs on an element are not balanced, and the inputs enter the element at different times, as we can see in figure 1.2. Here we can see the output `OUTA` first become '1', then '0' and finally '1', which is its final state. The adder's `OUTA` uses double amount of energy (i.e. $\alpha > 1$ in equation

1.5) to get to the final state. If we have even more combinatorial elements following this FA, these spurious switching activities will spread down the whole design, until a buffer or register halt the propagation. According to Kalis [22], tree multipliers uses 30% to 75% of its power in this kind of spurious switching. This shows there is a lot of potential of reducing energy consumption by reducing glitching.

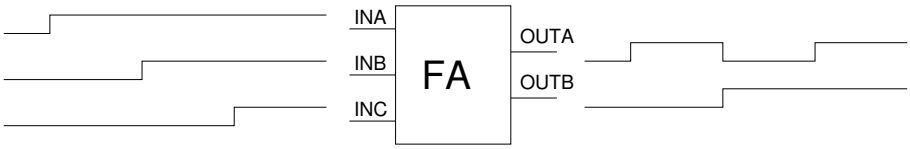

Figure 1.2: Glitching on an adder

1.2 VHDL netlister

The netlister this thesis uses as a foundation for the implementation of the multiplier generator algorithms, the estimator and the optimization routine is originally written by Espen Sand for his master thesis [1]. The code generates a multiplier, and write out the resulting multiplier using VHDL. The application takes arguments for the size of the multiplier, as well as different options to add Booth-recoding [15], pipelining and a vector merging adder. The program is used without these functions in this thesis, and is only used to generate VHDL for a multiplier tree.

The program is divided into several parts, and generates a list of partial products that it passes to a reduction tree generator module of the program. The netlister already has a tree generator build in, but several new tree generators can easily be added to the program, as long as the same data structures are used. Since the generation is sequential, it is easy to add step between the tree generator, and the writing of VHDL-file, making it possible to add an optimization step before writing the multiplier to VHDL.

1.3 Outline of this Thesis

Chapter 1 contains a small introduction to what power consumption in electronics are, and a brief explanation of the netlister program that is used as a basis for this thesis. Chapter 2 explains how multipliers in electronic circuits work, and how the partial products are generated. Since multipliers contains two schools of generation, the chapter gives a brief explanation of array multipliers. A more in-depth study of different tree multipliers is given, containing the theory behind the tree multiplier generation algorithms used in this thesis.

Chapter 3 reviews both probalistic approaches, as well as simulation approaches to estimate energy consumption. It also contains theory about the chosen simulation estimation, using the Monte Carlo approach [10].

Chapter 4 contains a comparison of the different tree multiplier generator algorithms, and some insight on the advantage and disadvantage of each algorithm. Oskuii's [5] algorithm for reducing power through changing the interconnection between the adders in the multiplier is also studied, together with a simplified version of said algorithm. The Optimization alternatives for the vector merging adder are discussed last.

Chapter 5 examines the implementation of the estimator used in this thesis. The estimator is implemented as a simulator at gate level, counting activity at each gate. It explains how

the simulator uses an event-based scheme to keep the run-time as low as possible. It also discusses what kind of timing-data is available to use in the estimator, and what data is beneficial to use. An SDF-parser was written for this thesis, and the implementation and choices made during the implementation are showed to the reader.

Chapter 6 contains details on how the optimization routine are implemented. Choices of implementation are also discussed.

Chapter 7 have results and discussions about the extraction of the delay model from existing multipliers. It also contains graphs over the performance of the estimator, and results on how good the line optimization are and discussions about the topics.

Chapter 8 has the conclusion of this thesis, and a suggestion of future work. It is follows by appendices and a bibliography

1.4 Main contributions

This thesis has made the following contributions to the field, and this are outlined here:

- An implementation of a gate-level estimator, that estimates power-usage before synthesis.
- Comparison of different algorithms power usage, using implemented estimator.
- Evaluation of the effect of line-optimization
- Timing data from post-layout multipliers.
- Implementation of a simplified version of Oskuii's [5] optimization routine.
- Added five algorithms for generating multiplier trees to the existing netlister [1].
- A tool to read SDF-files [14], to extract timing data to use in a timing model.

The SDF-parser is written in Perl. The netlister build upon during this thesis is written in C , and the added algorithms for generating multiplier trees, the estimator and the optimization routine is therefore also written in C.

Chapter 2

Multiplication

Multiplication is a very common task in modern digital electronics. The two most important methods used is to either perform shift and add operations and use existing components in a CPU, or add a multiplier unit. Further information about shift and add operation can be found in Parhami[9].

Multipliers in digital design are often divided into two subgroups: Array multipliers and tree multipliers[9, 28]. Array multipliers use a rigid pattern to construct their multipliers. This leads to compact designs and an evenly distributed delay. Tree multipliers on the other hand reduces the number of bits in each level in the tree until the calculation is done. Since this produces a complex tree structure, the delay is not evenly distributed. This may cause glitches that uses power. And the tree structure uses a lot of interconnection, end therefor uses a lot more area. Despite the larger area and not so evenly distributed delay, the tree multipliers use less power than array multipliers[21].

Another advantage tree multiplier have, is that they are a lot faster. The depth of an array multiplier is $O(n) = n$ while it is $O(n) = \log_2 n$ for multiplier trees[28] Even though the wiring cause more delay for multiplier trees, it still perform faster than array multipliers[21, 29].

2.1 Terminology

This thesis contains a lot of discussion around how multipliers are generated and different parts of the multiplier. To make the discussion understandable to the reader, it is important to be on the same terms when using different words. The three words: column, row and stage, will be used to describe different parts of a tree multiplier in this thesis. Partial products (see section 2.2) are organized into rows and columns. This is shown in figure 2.1, where all the partial products containing the bit b_0 share the same row. An example of a column is $[a_1b_0, a_0b_1]$. Each column contains partial products width different weight or value. By this we mean that each PP (partial product) in the rightmost column has the value of $2^0 = 1$, and in the next column $2^1 = 2$ and so forth. This is show in figure 2.7. To describe the value or weight of a partial product, the word bitweight or columnweight is used in this thesis.

During the reduction of partial products, full- and half-adders are added to the design, and a new set of partial products emerges (since adders reduce the number of partial products). During the thesis, each of these sets are called stages. The first stage is the initial set of partial products, the second stage is the set of partial products after the first reduction. Figure 2.8 show five stages, where the topmost tree contains the original partial products (denoted as $\bullet$ in the tree), and the next tree contains the tree at the second stage, after the first reduction.

2.2 Partial products

				a_3	a_2	a_1	a_0	Input
			$\times$	b_3	b_2	b_1	b_0	
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		Partial products
		a_3b_2	a_2b_2	a_1b_2	a_0b_2			
	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	Result

Figure 2.1: Basic bit-level multiplication

Multiplication is often done by dividing the problem into smaller multiplications, calculating the smaller multiplications and accumulate the result. In hardware this is often done by dividing the problem down to multiplying one bit with another, as this can be done with a regular AND operation. Each of these one bit multiplications are called a partial product (PP). The partial products are then added together to form the result of the multiplication. Figure 2.1 show how a is multiplied with b using unsigned integers as input, where value a_i is the bit in position i. Each bit from a is multiplied with each bit in b. This does however not account for signed numbers, and several other methods of generating partial products has been proposed [9].

When using two's complement form for signed integers, the corresponding partial products are given in Figure 2.2.(a). This PP generation does however require signed arithmetic to sum the partial products. This was improved by the Baugh-Wooley [8], and is shown in Figure 2.2.(b). This partial product generator uses NOT ports in addition to AND ports, but does not require signed arithmetic. The method does however require more additions, but this is usually outweighed by only requiring addition. The Baug-Wooley generator has been modified to require less additions, as shown in Figure 2.2.(c). This generator requires a minimal amount of extra additions to perform signed multiplication.

2.3 Array multipliers

Array multipliers use the fact that multiplications form a recurring pattern. In Figure 2.1 we see a basic setup for a bitwise multiplication. We multiply each bit in one multiplicand, with every bit in the other multiplicand. This is done for every bit in the first multiplicand, and then shifted position of the bit. Then all the bits are summed together, and produce the result of the multiplication.

2.3.1 Ripple-Carry Array

Using Ripple-Carry adders, we can exploit the recurring operation, and make an adder array. In Figure 2.3 we see the each column represents the first multiplicand that gets multiplied with each bit in the second multiplicand. The elements are actually an adder and an AND-gate that performs the actual multiplication. $Ain \cdot Bin$ is put into the adder together with Cin from the previous column and Sin. The adder then produces $Sout$ (Sum out) and $Cout$ (Carry out). $Aout$ and $Bout$ is just an extension of Ain and Bin respectively[28].

A problem with the Ripple-Carry approach is that it is very slow for larger implementations. Since the carry have to propagate through every row in the column, we get a very long critical path with this implementation.

					a_4	a_3	a_2	a_1	a_0
				$\times$	b_4	b_3	b_2	b_1	b_0
					$-a_4b_0$	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				$-a_4b_1$	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			$-a_4b_2$	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
		$-a_4b_3$	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	a_4b_4	$-a_3b_4$	$-a_2b_4$	$-a_1b_4$	$-a_0b_4$				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

(a)

					a_4	a_3	a_2	a_1	a_0
				$\times$	b_4	b_3	b_2	b_1	b_0
					$a_4\overline{b_0}$	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				$a_4\overline{b_1}$	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			$a_4\overline{b_2}$	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
		$a_4\overline{b_3}$	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	$a_4\overline{b_4}$	$a_3\overline{b_4}$	$a_2\overline{b_4}$	$a_1\overline{b_4}$	$a_0\overline{b_4}$				
	$\overline{a_4}$			a_4					
1	$\overline{b_4}$			b_4					
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

(b)

					a_4	a_3	a_2	a_1	a_0
				$\times$	b_4	b_3	b_2	b_1	b_0
				1	$\overline{a_4b_0}$	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				$\overline{a_4b_1}$	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			$\overline{a_4b_2}$	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
		$\overline{a_4b_3}$	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
1	a_4b_4	$\overline{a_3b_4}$	$\overline{a_2b_4}$	$\overline{a_1b_4}$	$\overline{a_0b_4}$				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

(c)

Figure 2.2: (a) 2's complement (b) Baugh-Wooley [8] (c) Modified Baugh-Wooley [9]

2.3.2 Carry-Save Array

A solution to the slow Ripple-Carry problem with a long critical path, is to let the carry travel down the column. This way the columns are not dependent on the column to the right, like they are with Ripple-Carry. In Figure 2.4 a Carry-Save Array is implemented. The element in the figure is an adder and an AND-gate that performs the actual multiplication. $Ain \cdot Bin$ is put into the adder together with Cin from the previous column and Sin. The adder then produces $Sout$ (sum out) and $Cout$ (carry out). $Aout$ and $Bout$ is just an extension of Ain and Bin respectively[28].

Since the carry only propagates down in each column, the result from the array is not completely finished. Some of the lines now consists of two outputs for a given bit level. Because of this, the result needs to be put into a Vector Merging Adder (VMA) to get only one output per bit level. This unit can be designed using different types for adder techniques, eg. Ripple-Carry or Carry-Look-ahead. Even though the array needs this extra calculation,

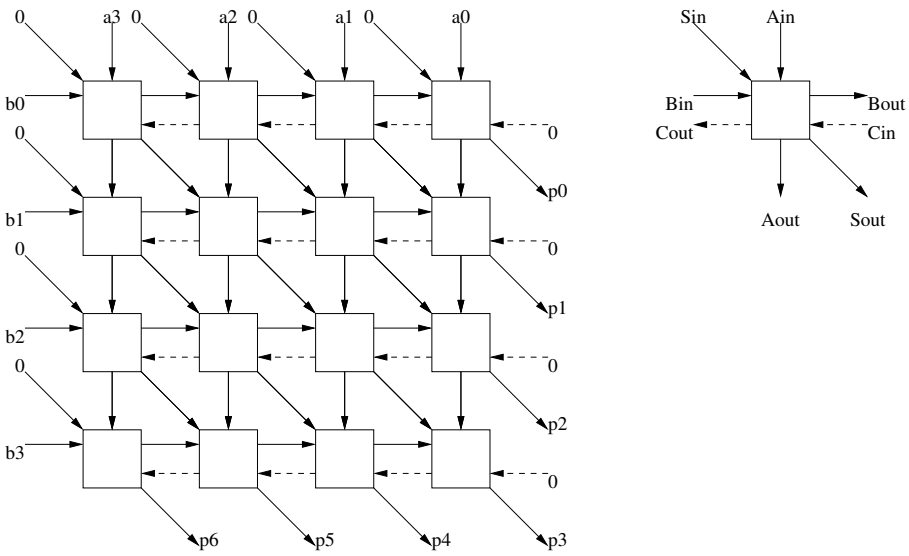

Figure 2.3: Ripple-carry array multiplier

the Carry-Save multiplier is faster than the Ripple-Carry Array in most cases[28]

2.4 Tree multipliers

Tree multipliers use different approach. They still use the same scheme for carry propagation, as Carry-Save adders used in previous section. But in addition to regular adders (hereby referred to as a full adder), they also use an element called a half adder. The full adder is a regular adder, with three inputs and two outputs (a 3,2 counter), while a half adder is an adder with two inputs and two outputs (a 2,2 counter). The output Sum is the same bitweight as the input, while $Cout$ is one value higher. As showed in the equation below ($C_{in} = 0$ for HA)[9]:

$$A + B + C_{in} = S + 2C_{out} \tag{2.1}$$

The block schematic of the full adder (FA) and the half adder (HA) is given i Figure 2.6, and standard gate-level designs for those two components is given in Figure 2.5. They have the following algorithmic output[9]:
HA:

$$S = A \oplus B \tag{2.2}$$

$$C_{out} = A \cdot B \tag{2.3}$$

FA:

$$S = (A \oplus B) \oplus C_{in} \tag{2.4}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B)) = (A \cdot B) + (C_{in} \cdot A) + (C_{in} \cdot B) \tag{2.5}$$

The result may have up to two outputs per bitweight (or column). By using a vector merging adder, one can reduce the output to a valid binary result. As said in Section 2.3.2, there are a lot of different ways to design a VMA. More discussion of the impact of VMA is done in section 4.3.

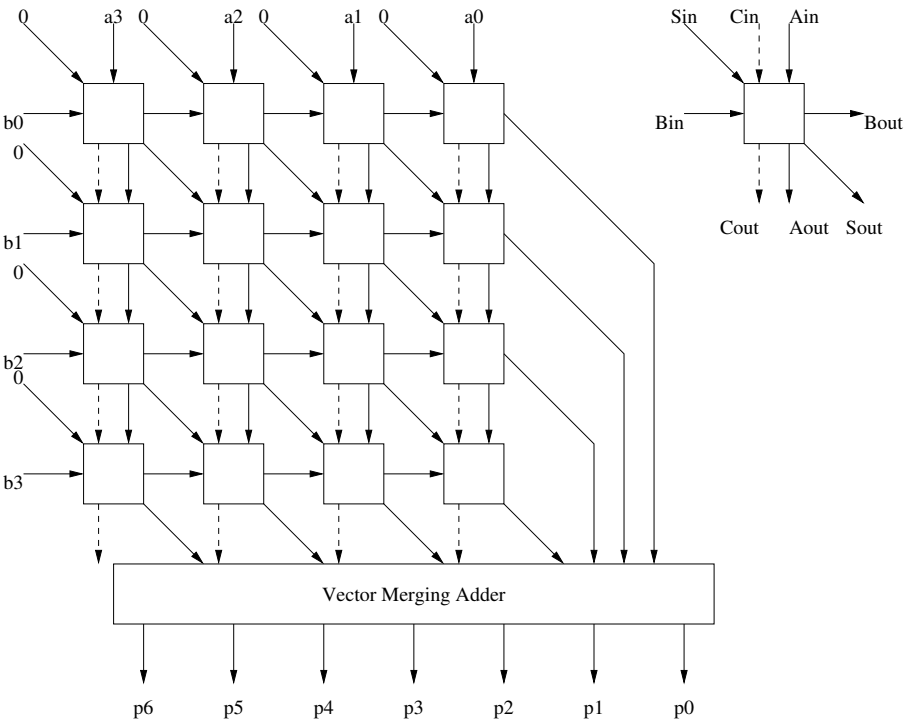

Figure 2.4: Carry-Save array multiplier

All of the algorithms use a matrix where the columns represents each bitweight and the number of rows represent how many partial products that bitweight has. Figure 2.7 shows an example of such a matrix, and how it is calculated. Each partial product is also often only represented by a dot (•) in the graph. The algorithm then reduces the matrix by adding FAs and HAs to it, which produces an output matrix, which represents the partial products for the next stage of the algorithm. The output matrix contains the partial products that still needs reduction. This task is repeated several times, until the output matrix contains only columns with one or two partial products. Each bitweight will then only have two outputs, and the result from the tree can be put into a vector merging adder (VMA). By connecting the FAs and HAs from each reduction stage, we will get a structure that looks very similar to a tree, hence the name: tree multiplier.

Because tree multipliers are faster and use less power than array multipliers[9, 21], the

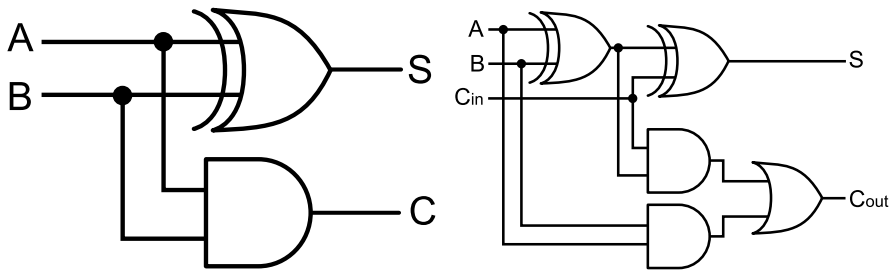

Figure 2.5: Full adder (FA) and half adder (HA)

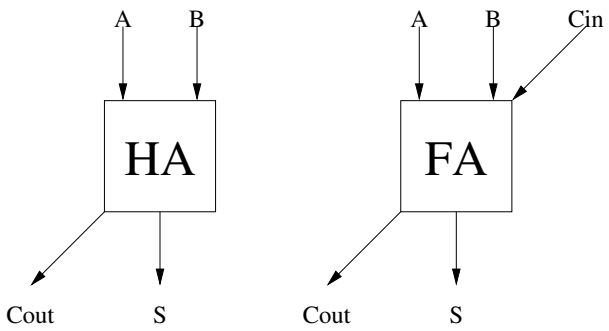

Figure 2.6: Half-adder and Full-adder elements used in Carry-Save adders

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
	a_3b_3	a_3b_2	a_3b_1	a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_2b_3	a_2b_2	a_2b_1	a_1b_1	a_0b_1	
			a_1b_3	a_1b_2	a_0b_2		
				a_0b_3			
Inputs:	1	2	3	4	3	2	1
Bitweight:	6 (2^6)	5 (2^5)	4 (2^4)	3 (2^3)	2 (2^2)	1 (2^1)	0 (2^0)

Figure 2.7: Setup of partial products in a tree multiplier

power optimizing done in this project is to decrease the power used in well-known tree multiplier schemes. The rest of this section contains explanation on how three tree algorithms function.

2.4.1 Wallace-tree

The Wallace algorithm[2] is the oldest of the algorithms presented here. It reduces the input matrix by grouping the rows together, and performs reductions on each group. Rows that are not part of any group is just transferred to the next stage of the algorithm. The algorithm is as following:

1 Group the rows into sets of three (see Figure 2.8)

2 Add FA for each group with three wires in, and a HA for each group with two wires in. This produces a new set of partial products, which represent the next stage.

3 If one or more columns contains more than two bits/rows, repeat the process.

In Figure 2.8 we see a 8×8 multiplier, that uses 39 FAs and 14 HAs. If we compare the resulting multiplier with the Carry-Save array, it uses less adder elements (8×8 Carry-Save array uses 64 adders). And the delay is a lot smaller. The Wallace tree needs for reduction stages, and therefore have a critical path of four elements, but the 8×8 Carry-Save array have a critical path consisting of eight elements. Tree structure does not have such a nice repetitive structure as the array, so it uses a lot more area on wiring[21].

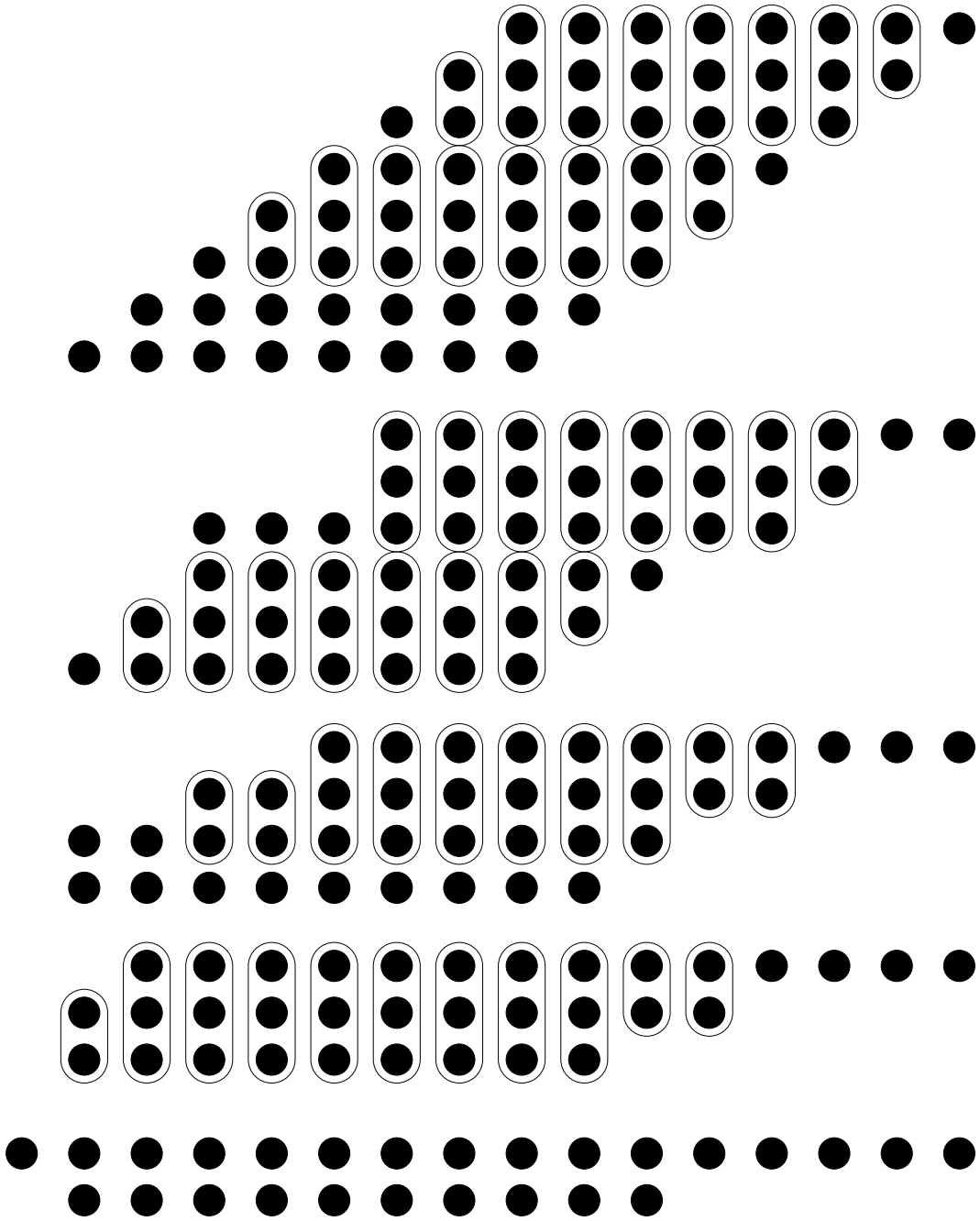

Figure 2.8: Wallace tree for a 8×8 multiplier

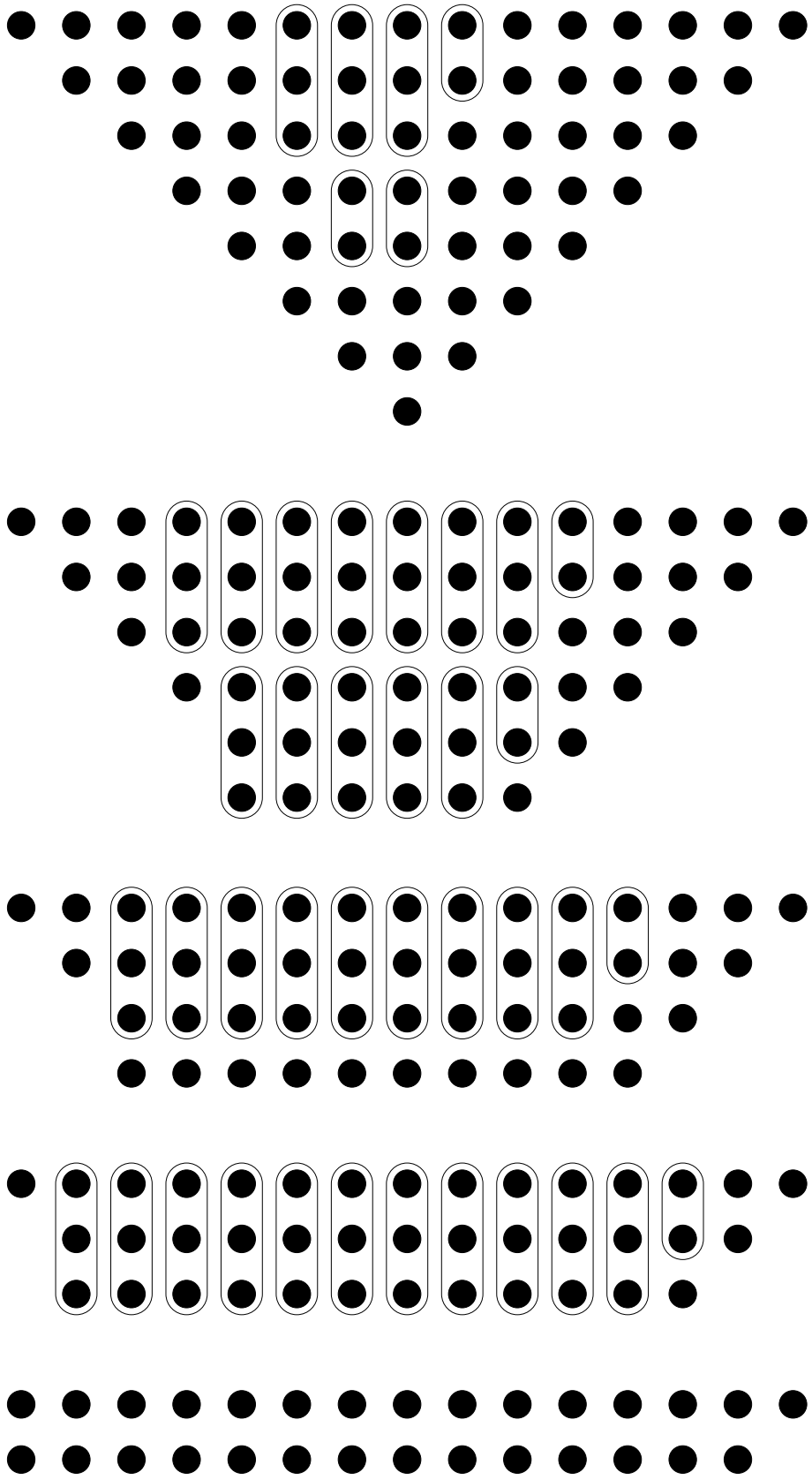

Figure 2.9: Dadda tree for a 8×8 multiplier

2.4.2 Dadda-trees

The Dadda algorithm reduces the tree by reducing columns instead of rows. The goal of the algorithm is to use the least amount of elements as possible. To accomplish this, the algorithm adds elements as late as possible. The algorithm is as follows[3, 30]:

1 Let $d_1 = 2$ and $d_{j+1} = \lfloor 3 \cdot d_j/2 \rfloor$, where d_j is the maximum height of the tree at the j-the reduction stage. Find the largest j, so that at least one of the columns has more bits than d_j.

2 Use FAs and HAs to reduce the partial products, so that no column has more than d_j bits left (see Figure 2.9)

3 If one or more columns contains more than two bits/rows, let $j = j - 1$ and repeat step 2.

The Dadda algorithm uses less FAs and HAs than Wallace. According to Habibbi and Witz[16] it uses the optimum amount of FAs. It is possible to make algorithms that use less HAs, but they would require more FAs.

And since it allocates elements as late as possible, it requires a larger VMA than Wallace. This is because the least significant bit would not get reduces until the last stage, and we will always start at bit two for the VMA. Therefore the VMA would always need $(n \cdot m) - 2$ bits for a Dadda multiplier. But the Dadda tree uses a lot less HAs, and is in studies found to be faster and smaller than the Wallace tree[31], despite the larger VMA.

2.4.3 Reduced Area multiplier

Since Dadda uses a larger VMA and Wallace uses HAs extensively, Bickerstaff et. al.[4] proposes an algorithm that tries to improve those drawbacks. Their "Reduced Area Multiplier" uses few HAs (about the same as Dadda) and needs a smaller VMA than Wallace. Since this algorithm also tries to reduce the number of wires as early as possible, this algorithm should produce less interconnection and smaller area than both Dadda and Wallace[4]. The Reduced Area multiplier uses this algorithm[4]:

1 Add $\lfloor b_i/3 \rfloor$ FAs in each column, where b_i is the number of bits in column i.

2 HAs are used only when

  2.1 When required to reduce the number of bits in a column to the number of bits specified in the Dadda series (see Chapter 2.4.2).

  2.2 To reduce the rightmost column containing only two bits

As we can see from step 2.2, this algorithm always tries to reduce the least significant two-wire output. Because of this, it will reduce the VMA with at least one for every row. This is the reason it gets smaller VMA-sizes. And since it has the least interconnection[4] it should dissipate the least amount of power through interconnections.

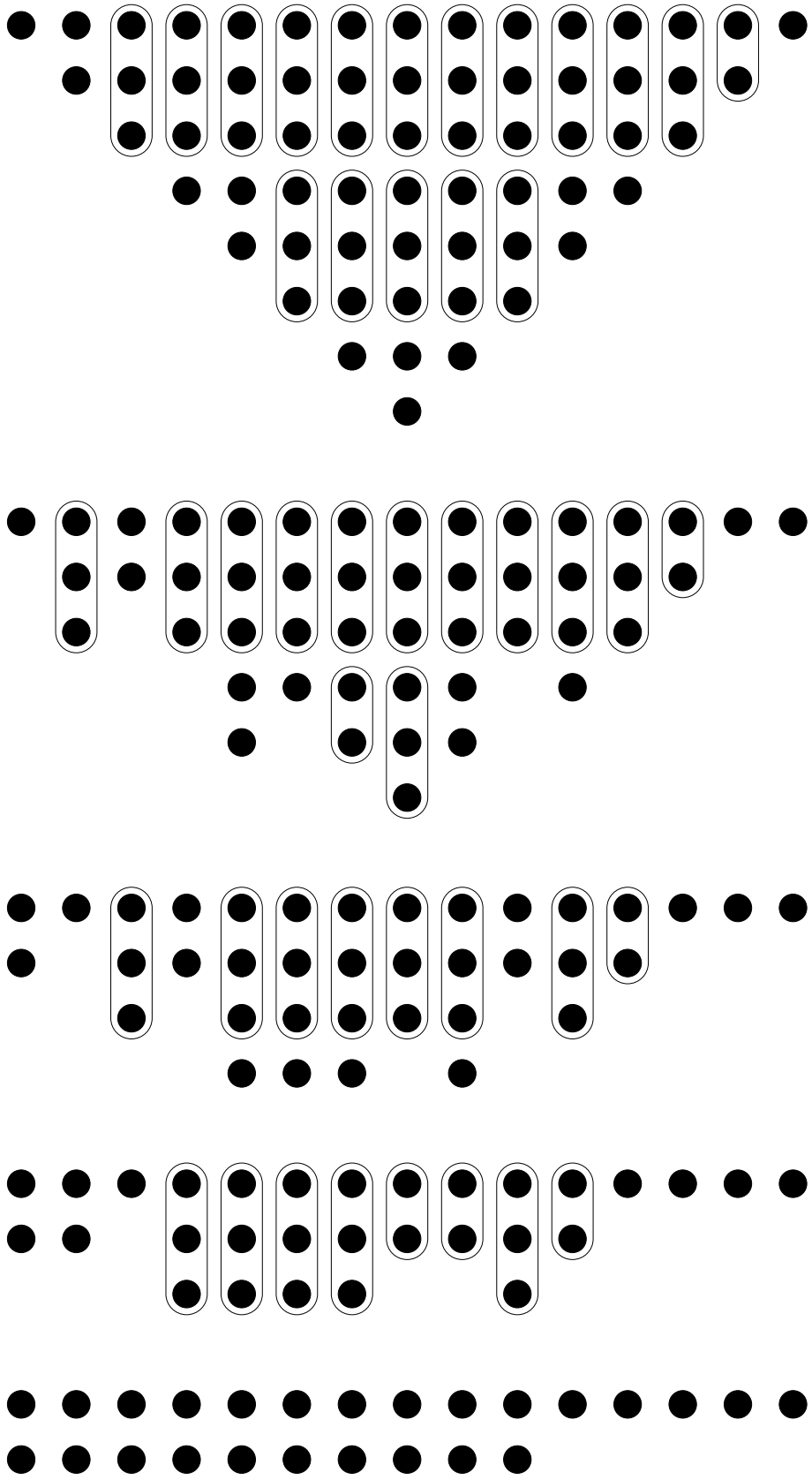

Figure 2.10: Reduced Area tree for a 8×8 multiplier

Chapter 3

Power estimation

Power estimation is the technique to find the power usage of a design, without having to do the actual implementation on silicon. Several different types of power can be measured, but finding the average or worst case power consumption is widely the most used application. By doing this before implementing, developers are able to cut the cost, since computer based estimations are much cheaper than actual silicon implementations. As the requirement for low power increases, the research and need for high level power estimation also increases [32].

Numerous methods for estimating power exists, often divided into two categories [33]: probabilistic and statistic. Inside each sub category, there are also different techniques based whether the estimation is done on system-, block-, gate- or transistor-level, and if the circuit is strongly combinatorial or not. In section 1.1.2 we show that the power usage is strongly correlated with the activity in the circuit. This is therefore the most used variable in the estimation. Several techniques considers the other power elements too small to be of any value to estimate [33]. The power consumed is also dependant on chip heating and temperature, but these variables are often set to a constant value when doing estimations [34]. The effect estimate of the chip might be higher or lower than an real world application This since corner values often are chosen, instead of typical operation temperatures. The estimate relative to other estimates using the same parameters will still be accurate.

The importance of accurate timing data are also important, since a lot of the switching activity comes from glitches (explained in section 1.1.3) in multipliers [22]. Without proper delay values, the power estimation will be considerably less accurate. This is especially true in circuits with high probability of glitches, such as multipliers.

3.1 Probabilistic based methods

Probabilistic approaches are based on calculating the probability of a change in a gate, and using that information to determine the probable power consumption. Several different techniques exists, but a common foundation is that it is easy to calculate a gate probable

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

Table 3.1: Truth table for NOR-gate

power consumption. Given that a gate consumes power during $0 \rightarrow 1$ transitions (see Equation 1.5 on page 3), it is only necessary to calculate the probability for this scenario to happen. Using the truth table of a NOR-gate, given in table 3.1, in a zero delay environment produces this equation:

$$p(0 \rightarrow 1) = p(0) \cdot p(1) = \frac{3}{4} \cdot \frac{1}{4} = 0{,}1875 \tag{3.1}$$

This calculation does, however, not take glitching into account, or consider that gates have delays between them. This kind of delay is shown to have a large impact on larger combinatorial circuits, such as multipliers [22]. This model also considers the probability for the inputs to be uniformly distributed. It is often assumed that signals are independent of each other, but that is not often the case. Two signals may never be high at the same time. This is called spatial correlation [34].

Another assumption is that a signals value over two clock cycles are independent of each other. This is often also not the case. This is called temporal correlation. The signals in equation 3.1 are considered both spatial and temporal independent [34].

To overcome the limitations of the simple signal probability model, several other methods have been proposed [34]. A proposition to use waveforms to solve the temporal dependency has been used, and by this changing the probability for each gate inputs based on time. This method might look similar to an event-driven simulation approach. A similar method is also used by Oskuii [5] in his optimization work. Another approach is to calculate the average number of transitions in each node in a circuit, using a single pass algorithm using the concept of Boolean difference. A third method tries to handle both spatial (though only internally) and temporal by using binary decision diagram [11]. The method defines the boolean function for each node, and uses this information to generate BDD-diagrams for the node, and using this information to calculate power usage. The disadvantage with BDD is that it is slow. A overview over these techniques is found in Najm [34].

All but one (BDD) of the techniques ignores spatial dependency. They are fast to compute, according to Najm [34]. BDD does take spatial dependency into account, but are also a slow algorithm. They are also less pattern dependent than their statistic counterpart. This is because the designer can specify the probability of the inputs, which is often more available to designers than specific input patterns. Another solution is to calculate the probabilities by surveying a large set of input patterns. Since the only calculation that needs to be done to the input patterns, is the calculation of probability for, large data sets can be used. They are however quite complex, and can be more difficult to implement than their counterpart. The accuracy is also slightly lower than statistical approaches, but they can be faster [34].

3.2 Statistic based estimations

Statistic approaches are simpler. The basic idea is to mimic a system, and do simulation with different input patterns, and then sum up the power used during the simulation. We only look at switching power, which is common in simulation techniques. This is because the majority of power dissipates from switching. Naming each input x_i, and combining this with Equation 1.4, the energy consumption of transition in each signal is [10]:

$$\text{Energy}_{\text{transition of x}_\text{i}} = \frac{1}{2} C_i V_{dd}^2 \tag{3.2}$$

By counting transitions in each node, it it possible to calculate the energy used by summing the energy used in each gate [10]:

$$\text{Energy}_{\text{total}} = \frac{1}{2}\sum_{i=1}^{N} n_{x_i} C_i V_{dd}^2 \tag{3.3}$$

Where N is the number of gates and n_{x_i} is the number of transitions in signal x_i. If we introduce $n_{x_i}(T)$ instead of n_{x_i}, which is the number of transitions happening during time T, one can compute the power effect of the system [10]:

$$P_T = \frac{1}{2}\sum_{i=1}^{N} \frac{n_{x_i}(T)}{T} C_i V_{dd}^2 \tag{3.4}$$

Since we usually need an average power dissipation over time, the average power dissipation is therefore given by the following equation [10]:

$$P_{switching} = \lim_{T\to\infty} P_T = \lim_{T\to\infty} \frac{1}{2}\sum_{i=1}^{N} \frac{n_{x_i}(T)}{T} C_i V_{dd}^2 \tag{3.5}$$

Simulation techniques use this as a basis to calculate the power used. The time T is in the equation set to be infinite, but that is not very feasible. In practical application simulation is done over a finite time interval, and the measured power is then believed to converge close to the actual power usage of the system [34].

These techniques are however very pattern dependent. Different input patterns could result in very different results of the power optimization, and is therefore hard to determine the accuracy of the result. And if we were to test all possible input variables, it would be too time consuming.

3.3 A Monte Carlo approach

To address the problems outline in the previous section, Burch [10] proposed a Monte Carlo approach for power estimation. By defining it $x_i(t)$ as a stochastic process, it is possible to calculate the error of the estimation. When the error is known, it is possible to know when to stop simulating, by stopping when the desired level of error is reached.

By defining $x_i(t)$ a stochastic process, $\mathbf{x}_i(t)$, one can define $\mathbf{P}_T$ as the random power of $\mathbf{x}_i(t)$ over the interval $(-\frac{T}{2}, +\frac{T}{2}]$:

$$\mathbf{P}_T = \frac{1}{2}\sum_{i=1}^{N} \frac{\mathbf{n}_{x_i}(T)}{T} C_i V_{dd}^2 \tag{3.6}$$

Burch [10] show that the expected value of $\mathbf{P}_T$ is the same for any T, and thus showing:

$$P_{switching} = \text{expected value}[\mathbf{P}_T] \tag{3.7}$$

The problem is now reduced to a mean estimation problem, which is common in statistics. The whole deduction is available in Burch [10].

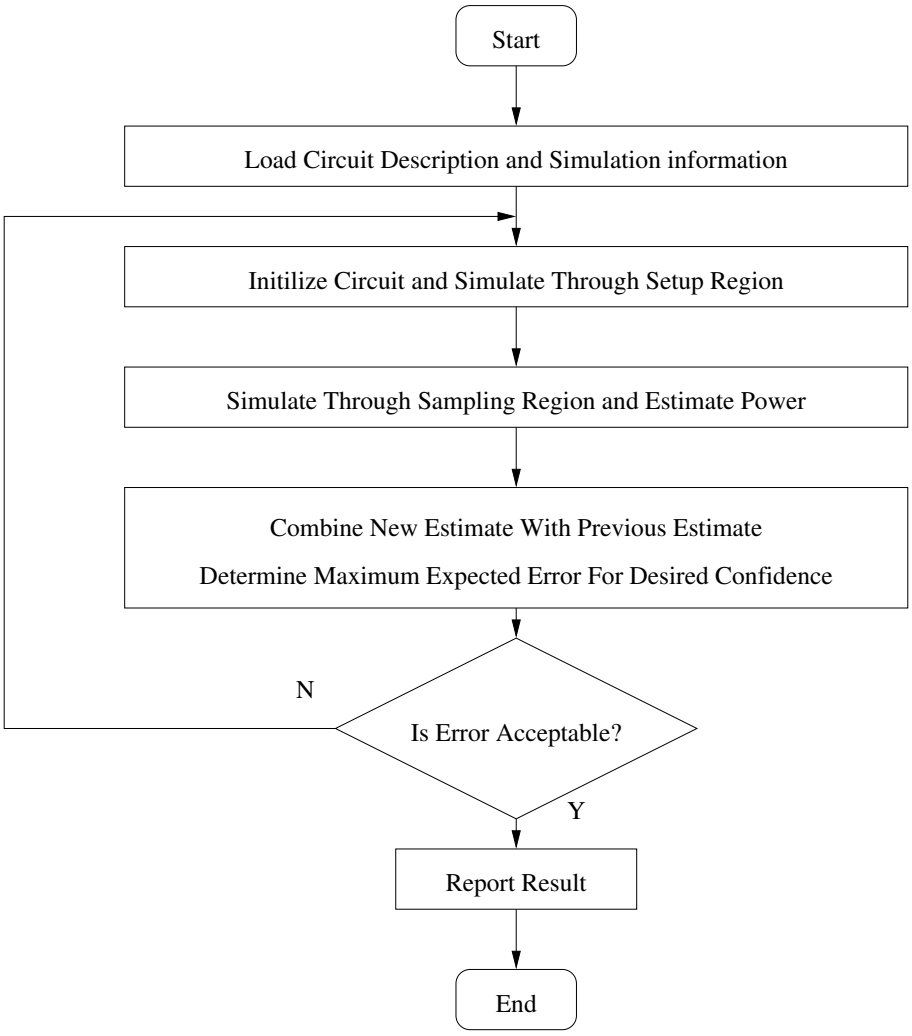

Figure 3.1: Flow diagram of Monte Carlo. From Burch [10]

3.3.1 Flow of the method

The Monte Carlo approach defines the flow-chart in figure 3.1 as its method. The first thing to do is to load the circuit and simulation information. The next step is the setup phase. The point of this is to put the circuit in a working state, so the power information extracted from the circuit is as accurate as possible. The length of the setup is determined by the length of the critical path. It is important that all elements in the design switch at stable rates before the recording of power information starts.

When the setup is done, it is possible to extract power information from the circuit. This is very similar to the setup step. By restarting the simulator, it is possible to record power usage in the circuit during this stage of the simulation. A problem of the approach, is that there are not easy to compute the length of this part of the simulation. The variable is most feasible to find through experiments, or by setting it larger enough by a good margin.

The power result is then analyzed, and the maximum expected error is calculated. If the maximum expected error is larger than the desired error, one has to run the simulation for another round. By doing the iterations until the expected maximum error is below the desired error, one has a power estimate that is sufficient.

3.3.2 Calculating Maximum expected error

An important aspect in the Monte Carlo approach is to calculate the, since the is the stop criteria for the algorithm. By computing the error, the algorithm becomes easier to use, since the user can specify if one wants very fast but not very accurate or a very accurate estimate of power.

By assuming $\mathbf{P}_T$ is normally distributed, Burch [10] gives the following equation for a stop criteria:

$$\frac{t_{\alpha/2} s_T}{\eta_T \sqrt{N}} < \epsilon \tag{3.8}$$

Where ϵ is the desired percentage error, η_T is the sample average and s_T is the sample standard deviation over the N different P_T-values found through simulation. The variable $t_{\alpha/2}$ is obtained through the t distribution [35]. Burch [10] also claims the number of simulations (i.e. the number of different P_T values) need to be done remains almost constant in proportion to the circuit size. The number of iterations should infant decrease slightly when used on larger circuits. This makes the Monte Carlo technique independent of circuit size, and places the run-time of the estimation in the hands of the simulator.

This stop criteria determines when to stop to get the power estimation of the whole circuit under the desired percentage error. A problem in optimization problems is that one often need information about where in the circuit is the power dissipated.

Xakellis [36] proposes a slightly different stop criteria, that simulates the circuit longer, to get accurate power results on gate level. Dividing the gates into regular density nodes and low density nodes, the algorithm uses a different stop criteria for each group. The regular density nodes have an average amount of transitions during the simulation, and can therefore with few iterations give power estimates within the acceptable error level. The low density nodes however have far less transitions during the simulation, and would require a lot more stimuli to get power estimates within the accepted level of error. Xekallis [36] therefore sets another stop criteria for those nodes. Since the low density nodes have few transition, they have the least impact on power usage, it should be acceptable with higher margin of error in these nodes. By dividing the nodes like this, the run time of the algorithm gets significantly reduces, compared to treating all nodes as regular nodes, with almost no decrease in the overall accepted level of error.

3.4 Random number generator

Since simulation approaches are in need of input patterns to perform power estimation, it is common to use a random or pseudo random number generator. By using a pseudo random number generator with a seed, it is possible to get the same random numbers for each set of numbers the random number generator delivers. This means the the generator can deliver the same input-patterns for several circuits that are under test, or even several different programs, without having to save the input-patterns.

A linear feedback shift register [12, 13] (LFSR) can be used as a pseudo random number generator, and produces the same sequence of random numbers, given the same seed. A LFSR is very easy to implement in both hardware and software, as it only uses a shift register and a set of XOR-ports. It delivers good random numbers, almost equal to the statistical expectation value of true random events. The LFRS has a sequence of $2^n - 1$ states, where n is the size of the shift register.

The LFSR can be implemented with either XOR or XNOR ports. This section will show an example of an XOR implementation. The LFRS has all 0's as an illegal state

Listing 3.1: 32-bit LFSR implemented in software (C code)

```
static uint32_t lfsr_seed = $0 \times 01$;

uint8_t lfsr_rand()
{
        uint32_t bit;
        bit = ((lfsr_seed >> 31) ^ (lfsr_seed >> 21) ^
                (lfsr_seed >> 1) ^ (lfsr_seed)) & 1;
        lfsr_seed = (lfsr_seed << 1) | (bit);
        return bit;
}
```

when implementing with XOR, since the register would then be locked in the same state indefinitely. When using XNOR, the illegal state is all 1's. The LFSR uses XOR/XNOR ports on selected bits in the register as a feedback into the register. This way it produces a new unique sequence for each shift, if the XOR/XNOR ports are placed to get maximum sequence length [12, 13].

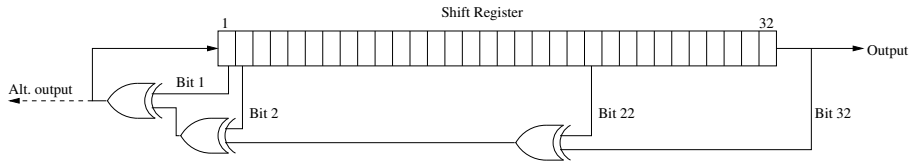

Figure 3.2: A 32-bit LFSR, implemented in hardware

Figure 3.2 show an example of an implementation in hardware. Here bits 1, 2, 22 and 32 are XOR'ed, and used as a feedback for the register. The placement of the XOR elements ensures a maximum length of the sequence. A list of placement of XOR elements for maximum length can be found in [12, 13]. The bit shifted out of the register is in most cases used as the output. This means that one has to do a shift operation for each pseudo random bit, to ensure statistical properties needed for a random number generator. Listing 3.1 show an implementation in software, although using the alternative output from Figure 3.2 instead of the regular output, and a startup seed of 1.

Chapter 4

Power optimization

Optimization may be applied in most stages of the generation of multiplier. Most optimizations are done at either block-level or at transistor-level. This chapter focuses on three types of optimizations done at block-level. Algorithm optimization concentrates on how different reduction schemes differ from each other, and that some might have better power-characteristics than others.

Interconnect optimization focuses on connections between adders inside the multiplier tree. There are several different ways of connecting the adders together, and some mutations might be better than others. Since the multiplier trees use Carry-Save scheme, there is a need for a VMA as last stage of the multiplier. The last section look at sizes of VMAs, and their impact on the overall power consumption.

4.1 Algorithm optimization

There are several algorithms used to generate multiplier trees, as explained in Section 2.4. Since they use different reduction schemes, they will also use a different amount of adders, as seen in Table 4.1. This gives each scheme different delays and power consumption properties. The difference in the adders also makes the bit width of the VMA different, which in turn influences how much power the VMA will use. The smaller the VMA is, the less power consumption it will have.

Research effort has been made to compare array multipliers with tree multipliers, and most sources agree that tree multipliers have less delay and use equal or less power, at the

Generation algorithms	Size	FA	HA	Adder size
Wallace [2]	8×8	38	15	11
Dadda [3]	8×8	35	7	14
Reduced Area [4]	8×8	39	7	10
Wallace [2]	12×12	102	34	18
Dadda [3]	12×12	99	11	22
Reduced Area [4]	12×12	104	11	17
Wallace [2]	16×16	200	54	25
Dadda [3]	16×16	195	15	30
Wallace [2]	32×32	906	164	55
Dadda [3]	32×32	899	31	62

Table 4.1: Comparison of well-known tree generation algorithms. Numbers from [4] and [6]

cost of area on the chip [21, 29]. There are however less research about how power usage of different tree multipliers is compared to each other.

Several studies have investigated the delay and size differences of the different multipliers [16, 31, 4, 6]. The conclusion of all of them is that Dadda multipliers use less area then a Wallace type multiplier, while having about equal delay. Since the Dadda multipliers use less area and adder elements, there is reason to believe they also use less power. This is however not confirmed by any of the studies.

The Reduced Area (RA) multiplier by Bickerstaff [4] uses less area than the Dadda multiplier, but uses a few more adder elements. The main reason RA multipliers are smaller, are because they use less area on wiring. Since Dadda reduces as late as possible, it should be more prone to glitching problems than RA. These two properties (less area and earlier reduction) of the RA multiplier should lead to less power consumption than Dadda, and therefore also Wallace. In addition to using less power, the RA multiplier needs the smallest VMA of all three algorithms, and should therefore save energy in that part of the multiplier as well.

This comparison leads us to believe that the Wallace uses the most power of the three algorithms. Which algorithms, the Dadda or the RA, that uses the least amount of power is not easily determined. RA uses slightly more adders, while using less wires than Dadda. It would seem as if the RA algorithm would perform better than Dadda.

4.2 Interconnect optimization

Each algorithm produces a tree structure of adders. However, the algorithms do not describe how the interconnections between the adders are supposed to be. Changing the interconnections inside a column does not change the functionality. A column containing four partial products might get reduced with one FA. Three of the PPs are then connected to the adder, and the fourth PP is just a feed-through line to the next stage in the reduction process. This causes different delays for different partial products, and raises the question: How should the lines be connected to consume the least amount of power? An exhaustive search of all possible mutations would be too time consuming to compute.

4.2.1 Reduction of search space

Oskuii [5] suggests that the lines with the highest activity should be connected to the least amount of adders. This will lead to less transitions in the circuit. He also extends the assumptions, and proposes that lines with the most activity should be connected to gates with the the least amount of delay. Since glitches are caused by unbalanced inputs, it should be energy efficient to let the lines with highest probability for transition traverse as fast as possible in the tree. This will let the high transition lines less likely to cause glitches.

Najm [34] introduced the notation of transition density, which Oskuii [5] also adopts. The density describes the average amount of transitions in a gate:

$$D_i = \lim_{T \to \infty} \frac{n_{x_i}(T)}{T} \tag{4.1}$$

Since each column adds PPs to the the same column on the next reduction stage, or the column with a larger bit weight, optimizing should be done from the rightmost columns (least significant bits) to the rightmost columns (most significant bits), and from the first stage to the last stage. This should prevent the optimization to change the power consumption of previous optimizations. Oskuii [5] therefore assumes it is safe to optimize each column

separately, if done in this order. By doing this, one might end up at a local minima, but this is a good trade-off compared to doing a brute force search of all possibilities. Since the problem is NP hard [37], local solution is an acceptable solution, since a global minima would require a full test of all possibilities.

Oskuii [5] proposes to do a post-optimization, after the multiplier tree is generated. Iterating through every column in every stage, each partial product is sorted by transition density (like in Figure 4.1). In each column, a set of interconnect mutations believed to give low power consumption is made. This reduces the problem dramatically, since it discards most mutations in this step. Each low power mutation runs through a power estimator, and the mutation yielding the lowest power consumption is used for the column. This is done for all columns in the multiplier.

The reduction of interconnect mutation is done with the knowledge that FAs and HAs have different delay. An HA only has one gate in its critical path for both inputs, which is less then the FA. The HA should therefore have higher density partial products as inputs, than an FA. A feed-through (a partial product not connected to a adder in the stage) has no gates, and should therefore have a very small delay. The partial products with the highest density should therefore not be connected to an adder, but rather get redirected to the next stage of the reduction. We now have two rules, which should reduce the amount of mutations in each column.

Looking at Figure 2.5 on page 11, we can see that an FA does not have balanced inputs. The C input has less gates to both outputs than input A and B. This means that input C has less delay through the adder than A and B. Adding the partial products with least density to input A and B should give less power consumption. This makes three rules to reduce the amount of interconnect mutations. By employing these rules when construction a set of possible interconnections, we get a much smaller set then by using all possible mutations.

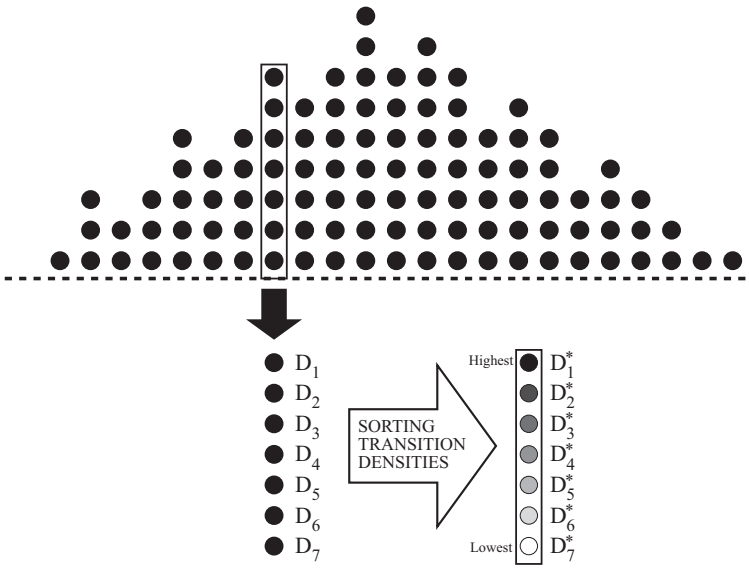

Figure 4.1: Sorting partial products based on activity. From [5]

The algorithm can be rather computing exhaustive, since it is mandatory to do power estimates for each mutation, and then choose the mutation that uses the least amount of energy. It is therefore critical to have a fast power estimator. The optimization routine can

also be used to optimize against highly correlated input data. By estimating power usage using correlated data, the optimization will optimize towards the least amount of power usage for that kind of input. This shows that the optimization routine proposed is very dependent on the estimation for accurate results.

4.3 Vector Merging adder

ASYMPTOTIC TIME AND AREA REQUIREMENTS OF n-bit ADDERS

Adder Type	Abbreviation	Time	Area
Ripple Carry Adder	RCA	$O(n)$	$O(n)$
Manchester Carry Chain adder	MCC	$O(n)$	$O(n)$
Constant width carry SKip adder	CSK	$O(\sqrt{n})$	$O(n)$
Variable width carry SKip adder	VSK	$O(\sqrt{n})$	$O(n)$
Carry SeLect adder	CSL	$O(\sqrt{n})$	$O(n)$
Carry Lookahead Adder	CLA	$O(\log n)$	$O(n \log n)$
Brent and Kung adder	B&K	$O(\log n)$	$O(n \log n)$
ELM adder	ELM	$O(\log n)$	$O(n \log n)$
Signed Digit adder (base-r)	SD-r	$O(b)^\dagger$	$O(n)$
Carry Save Adder	CSA	$O(1)$	$O(n)$

† b is the number of bits per digit.

Table 4.2: List of surveyed adder types in Nagendra [7]

There are a lot of different types of adders that can be used as a Vector Merging Adder, to convert the result to a regular binary number. Nagendra [7] has an excellent overview over commonly used adders, and Table 4.2, Table 4.3 and Figure 4.2 are from that survey.

ADDER CIRCUIT DESCRIPTION

Adder	*Area* ($\times 10^6$ λ^2)			*No. of transistors*			*Max transistor size (n/p)*		
Type	*16-bit*	*32-bit*	*64-bit*	*16-bit*	*32-bit*	*64-bit*	*16-bit*	*32-bit*	*64-bit*
RCA	0.40	0.80	1.60	596	1204	2420	3/3	3/3	3/3
MCC	0.48	0.96	1.90	642	1298	2610	4/4	4/4	4/4
CSK	0.82	1.62	3.22	682	1410	2866	4/4	4/4	4/4
VSK	0.81	1.76	3.44	706	1440	2900	3/3	3/3	3/3
CSL	0.76	1.45	2.75	914	1982	4128	5/7	6/10	8/13
CLA	1.14	2.27	4.55	1038	2132	4348	4/4	4/4	4/4
B&K	1.25	3.00	6.76	1072	2442	5444	3/3	3/3	3/3
ELM	1.08	2.36	5.38	892	2078	4752	3/5	5/7	8/12
CSA	1.05	2.03	3.90	1176	2360	4728	3/3	3/3	3/3
SD-4	1.36	2.71	5.41	1550	3166	6398	3/5	3/5	3/5
SD-8	1.11	2.42	4.61	1228	2812	5452	3/5	3/5	3/5
SD-16	1.09	2.17	4.32	1186	2490	5098	3/5	3/5	3/5
SD-32	0.97	2.25	4.16	1020	2572	4900	3/5	3/5	3/5
SD-64	1.12	2.23	4.07	1180	2530	4780	3/5	3/5	3/5
SD-128	1.27	2.11	3.78	1340	2364	4412	3/5	3/5	3/5

Table 4.3: Area and number of transistors in adders from Nagendra [7] survey

Table 4.2 contains the different adders surveyed, and their delay and area functions of size. Table 4.3 contains area and transistor usage information about the adder structures. Since the survey counts transistors instead of adder elements, one has to convert the tree structures in this thesis to transistors to compare the size of the VMA to the size of the multiplier tree. Nagendra uses 24 transistors for each FA and 14 transistor for each HA, and uses six transistors around each adder as buffer and driver. The total transistors for each adder element can be found in table 4.4.

Since it is beneficial to know how much impact the VMA has on the overall power usage, we will combine the information from Nagendra [7] with known sizes for multiplier trees [4].

Element	Transistors
FA	30
HA	20

Table 4.4: Transistor count for each element used in Nagandra [7]

			Transistors		
Generation algorithms	Size	Adder size	Tree	RCA adder	CSA adder
Wallace [2]	8×8	11	1440	836	1628
Dadda [3]	8×8	14	1190	1064	2072
Reduced Area [4]	8×8	10	1320	760	1480
Wallace [2]	12×12	18	3740	1368	2664
Dadda [3]	12×12	22	3190	1672	3256
Reduced Area [4]	12×12	17	3340	1292	2516
Wallace [2]	16×16	25	7080	1900	3700
Dadda [3]	16×16	30	6150	2280	4440
Wallace [2]	32×32	55	30460	4180	8140
Dadda [3]	32×32	62	27590	4712	9176

Table 4.5: Comparison of reduction trees and VMA

By using the tree information found in table 4.1, and converting the FA and HA numbers to a transistor count, it is possible to make comparison of the multiplier tree and the corresponding vector merging adder. This estimation will not be correct, since we assume the transistor count would be the same for each adder element as the transistor count used in Nagendra [7], but it should give a good approximation. The result is displayed in table 4.5.

The size of the VMA is also presented, and the size of two possible adder structures are approximated using linear regression. It should be accurate since both algorithms have $O(n)$ size increase (according to Table 4.2). The adders chosen are one small (Ripple Carry Adder) and one large (Carry Save Adder) structure, to show the extreme points.

As we can see in Table 4.5, the 8×8 multiplier tree as about the same size as the VMA. This is true for 8×8 multipliers, but since the tree structure grows $O(n^2)$ [4], this is not true for larger multipliers. The 12×12 are as we can see larger than the corresponding VMA. Width even larger multipliers, the power significance of the VMA gets smaller.

Figure 4.2 presents the power usage of the different adder structures. The power values used are computed using HSPICE at 5V. Nagendra [7] does not recommend using the values for external comparison, only to compare the adders relative to each other. Using the survey done by Nagendra [7], it would be easy to choose a VMA based on the desired delay (information found in the Nagendra [7]) and power (from Figure 4.2) based on what the design needs.

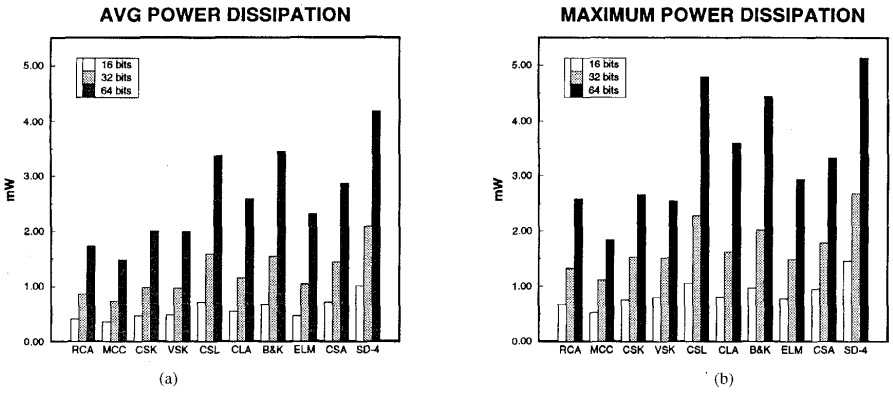

Figure 4.2: Power usage of adders in Nagendra [7] survey

Chapter 5

Implementing the power estimator

The estimator in this thesis is implemented as a simulator that simulates the propagation of signals through the multiplier. The power is then calculated based on $0 \rightarrow 1$ transitions in the different gates. To get a more accurate result, the simulator runs several simulations, using the Monte Carlo[10] approach, explained in section 3.3. The simulation is done at gate-level, and the multiplier therefore needs to be flattened before the simulation starts, because the generated multiplier have generated at block-level.

A simulation method was chosen since it gives very accurate results to the problem, as it is possible to choose the margin of error by tuning the length of the simulation. The simulator also records the activity in each gate, to use as a profile for the optimization to be done later. This is similar to the technique used in Xakellis [36] and explained in section 3.3. This is because we want to use the activity information in each gate as a parameter for the optimization. To get the simulation to be accurate, it is important to supply accurate delay data to the model. This aspect of the simulation is discussed in section 5.4.

An alternative to this method is to choose one of the probabilistic approaches from section 3.1. It was not chosen, due to the simulation being more accurate and taking spatial correlation into account. The method was also believed to be easier to implement. A simulation for power estimation is, according to Najm [34], very effective in practice.

5.1 The input data-structure

The netlister generates a multiplier tree using a specific data structure to store the result, before it is transformed into VHDL. This data structure is also used by the tree generators added to the program for this thesis. This section gives a quick explanation of the data structures used. A more thorough explanation can be found in Sand [1].

In figure 5.1 we can see a representation of the data structure used in the netlister. The data structures use linked lists [37] to tie the structures together. The structure `ADDERTREE` represents a collection of columns, each with the same column weight. The list of `FAGROUP` represent a column at a given stage in the multiplier generation process. The first `FAGROUP` in the first `ADDERTREE` represent the first column in the first stage of the process. See section 2.1 for a summary of terminology used for multipliers.

The `FA` structure represents the individual adder. It can represent a regular full-adder, but also an half-adder or a feed-through element. Each `FA` is connected to the `FA`-structures that is its input and outputs (show by the S and C block in figure 5.1). `FAGROUP` also holds a list of `FA`-elements, representing the adders at a given stage and column in the multiplier tree.

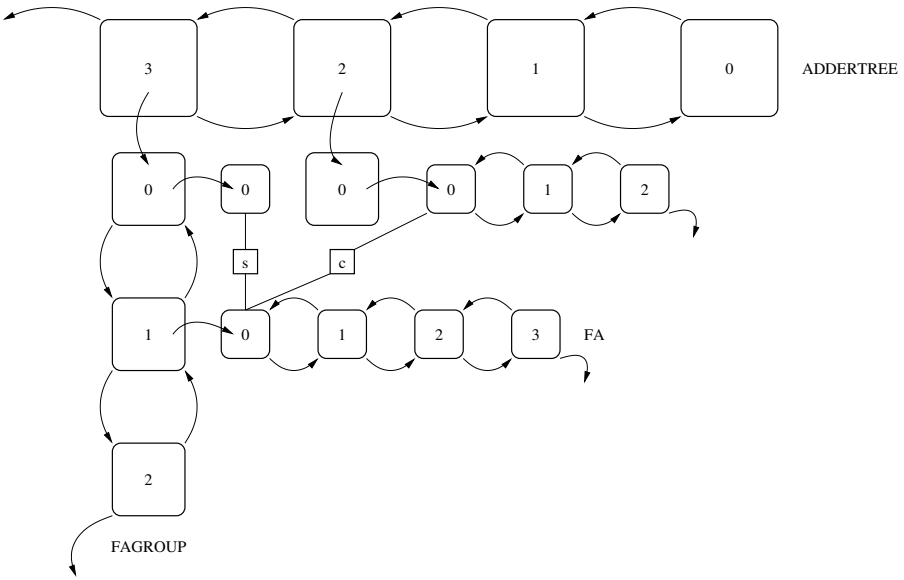

Figure 5.1: Multiplier trees as they are represented in the netlister. Figure taken from Sand [1].

5.2 Preprocessing of the multiplier

Since the simulator runs several times on each multiplier, an easy way to improve performance, is to do as all the calculations independent of the simulation, before the simulation starts. The multiplier generated by the netlister uses the structure explained in the previous section, and is flattened to gate-level by transforming each FA-block and HA-block to their respective gates (as shown in figure 5.2).

The gates is then put into a data structure that makes it easy to transverse back and forth in the structure, and makes calculations of outputs easy. This approach uses more memory, but makes the calculations done during the simulation easier. The reason for this design is that we want the simulation to be fast. The amount memory for this approach is minimal, since each adder in the tree only uses 288 bytes. This corresponds to 338Kb memory for a 32×32 multiplier (approximately 1200 gates). This is a small memory footprint consider that today's computers have between 1Gb and 4Gb RAM. A possibility for increased run speed is superior to this little memory increase.

Listing 5.1 show the data structure used to store each gate (SIMGATE). The structure contains information on what type of gate it is, and which gates it is connected to (both input and output connections). The amount of $0 \rightarrow 1$ transitions is stored in `activity`. The variables `next` and `first` are used to traverse through the gates, and is primarily used to unallocate memory when the simulation is finished.

Since the delay through a gate is stored in the data structure representing the gate, and not as a fixed number, it is possible to apply a timing model to the multiplier tree prior to the actual simulation. In this implementation, the delay is set to a fixed number, based on the type of port. This makes the design flexible, since it is possible to write a routine at a later date that analyzes the tree, and assigns timing data to each gate. This step would be naturally fit between this step and the simulation.

Since the amount of transitions is stored in each gate, this information can be used to make a power-profile of the multiplier. This profile can then be used in the optimization process.

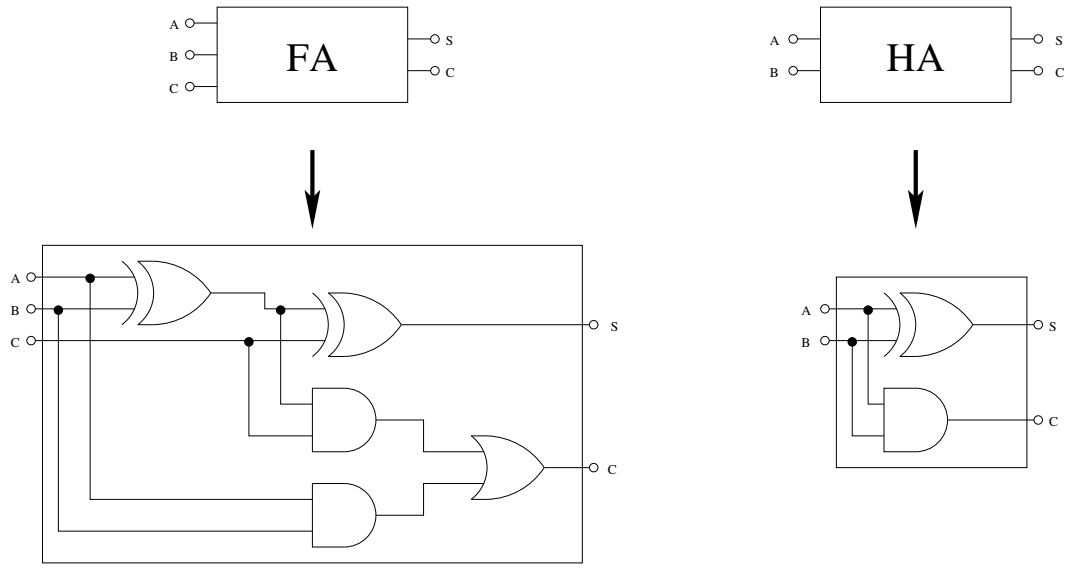

Figure 5.2: Shows how FA-blocks and HA-blocks get converted to their respective gates

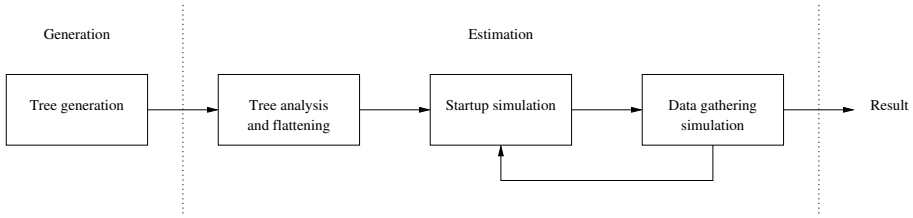

Figure 5.3: Model of simulator operation

The structure `SIMBLOCK` is used as a bridge between the block-level and the gate-level. Each block element (`FA`-element) in the pre-generated multiplier has a corresponding `SIMBLOCK` element. This way the generated tree has a 1:1 mapping of the estimated power used.

5.3 Simulation

After the generation of the gate-level data structures, the model is ready to be simulated. The simulator uses an event-queue to determine what to do. The input-gates is set to pseudo-random numbers, and the gates are added to the event queue. The model of the circuit is then simulated, by calculation outputs from the gates, and then adding the new input to the event-queue, until the circuit has reached its steady-state.

The pseudo-random number generator is a linear feedback shift register (LFSR)[13, 12]. It is wise to use a commonly known algorithm that produces the same result every time, since that would make the simulations reproducible. The reason to choose this algorithm is that it is very easy to implement in hardware, and therefore also in any VHDL test bench. The algorithm also produces very uncorrelated and very uniformly distributed numbers, and that makes the result from the estimation more accurate. The LFSR used in this simulator uses a 32 bit wide register to produce the pseudo-random numbers, and always start with the seed 0×01 at the start of each estimation-cycle (when the multiplier is flattened).

Listing 5.1: C definitions of data structures used in the simulator

```
enum simgatetype_t {
        SIMGATE_NULL, SIMGATE_AND, SIMGATE_XOR, SIMGATE_OR,
SIMGATE_OUT, SIMGATE_IN, SIMGATE_PIPE, SIMGATE_NO
};

struct SIMBLOCK
{
        struct FA* element;
        struct SIMBLOCK* next;
        struct SIMGATE* gates[GATESIZE];
        struct SIMGATE* output[2]; /* 0 = SUM, 1 = CARRY */
        struct SIMGATE* input[3]; /* 0 = InA, 1 = InB, 2 = InC */
};
typedef struct SIMBLOCK SIMBLOCK;

struct SIMGATE
{
        struct SIMGATE* input[2];
        struct SIMGATE* output[2];
        enum simgatetype_t type;
        uint8_t value;
        struct SIMBLOCK* parent;
        struct SIMGATE* next;
        struct SIMGATE* first;
        uint32_t delay;
        uint32_t activity;
        uint32_t timestamp;
};
```

The standard C library is delivered with a function that delivers a random number. The reason for not using this function is that the implementation of the function is left to the writer of the C library. It is therefore hard to make the result reproducible. We also want a uniform distribution between zero and one. This is not guaranteed with the C library call. Other random number generators were not considered, since LFSR fitted the requirement very well.

The event queue is actually implemented using several queues. Each queue corresponds to a specific time in the simulation. The current queue holds what signals that needs to be set at the current time, the next queue whats needs to be set at the next time interval and so forth. When the queue that represents the current time is empty, next queue is set to be the current one (time goes on), and the empty queue is reused for a different time of the simulation.

The simulation continues until all the queues are empty. When this occurs, it means that the multiplier has come to a state where everything is stable, and the correct result is on the output pins of the multiplier. No more energy will be used to change the values of the gates before a new set of inputs are used.

This simulation is done several times. First without counting the number of transitions, and then several more times by counting the number of transitions. This is done to set the multiplier in an active state, by letting the gate get a to an unknown state. This is more realistic than letting all the gates start at the value of zero, and should therefore provide better accuracy. The simulator has a pre-run of ten simulations before starting the actual simulation. This is a bit more than suggested in Burch [10], but is done since the multiplier tree is quite complex. It is important that every gate have a chance to change its value before the actual simulation begins.

After the setup simulation is done, a set of simulations that also records the switching in the tree is done. Through experiments, this number is set to 100. Section 7.2 shows that this number seem to be too high. But since we need an accurate power profile for optimization, we have to do more simulations to get higher accuracy at gate level. Remembering Section 3.3, it states that a higher number of simulation rounds were needed for accurate power estimation per gate.

The simulator does not contain a stop criterion. This makes the validity of the estimator small. This should have been improved during the thesis, but due to time constraints, it was not possible. The estimator is instead configured to do more iterations than should be necessary, to compensate for the lack of stop criterion and error calculation.

Using this method[10], the mean value of the power used would statistically converge to the real world power usage of the multiplier.

5.4 Timing model

It is important that the timing model used by the estimator is accurate. Since glitches happen due to timing between gates, a poor timing model will give inaccurate or erroneous glitching compared to a real world implementation.

Since timing is such an important aspect of the estimator, some time was allocated to investigate how timing in post-layout multipliers are. This information would then be used to configure the timing parameters of the simulator, to give a more accurate estimate.

5.4.1 Available timing data

Timing data is supplied by Johnny Pihl at Atmel Norway, originally used in Kalis [22]. They are supplied as SDF-files [14], which contain both the netlist, and the timing data

inside elements and between elements in the design. Data from four post-layout designs were supplied, with a list of functional behavior of the elements in the netlist.

The multipliers supplied were three multipliers designed by Oskuii for his thesis [5] and one designed by the netlister used in this thesis [1]. All of the multipliers are mapped into 0,18/0,15 μm CMOS process by Atmel Norway. Timing data for each multiplier is also supplied under two different process voltage and temperature (PVT) conditions, since CMOS elements behave differently under different conditions. The first PVT condition, called PVT-MIN, is using a working temperature of -40°C and a supply voltage of 1.95V. The second PVT condition, called PVT-MAX, is using a working temperature of 100°C and a supply voltage of 1.60V.

The two multipliers generated by Oskuii [5] are using a optimization routine from his thesis. One is optimized for minimum power usage, and the other for maximum power usage. The third multiplier from Oskuii [5] is generated through random interconnection. The last multiplier is generated by the netlister created by Sand [1] (also called Modgen) for his thesis. It was generated by the netlister before modification made in this thesis, and his therefore using the original reduction tree generator. All four multipliers are 32×32 in size.

To improve the estimation, it is possible to use the timing data from these multipliers as delay for the block-level multipliers we want to simulate. By examining the delay through and between full- and half-adders in supplied multipliers, the estimation could become more accurate. This would also make the estimator more dependent on the synthesis tools and technology library used. The reason the delay between the gates is different from each other, is because the wire they have to charge to make a transition have variable length. When the synthesis tool places the adder elements on the silicon during layout, the placement will be in an irregular pattern. This causes different length between adder elements inside the tree, which causes the wires between them to have variable length.

The irregularities is due to the way the adders are connected to each other. The adders have two outputs, and those outputs are connected to adders in different columns. Another reason the lines have different length is that some PP are not connected to any adder in that stage, but rather fed through to the next stage in the reduction process. There is reason to believe these lines are longer than the rest. Figure 5.4 show a column in a tree multiplier that have seven PP inputs. Six of them are connected to two adders, while one of them is not connected any adder in this stage. We assume that this would lead to a longer wire for this PP, as it will be harder for the synthesis tool to place the input and output adder close together. The figure also shows how the Carry and Sum lines are going to be connected to adders in different columns later in the reduction tree.

Since the multipliers are synthesized and technology mapped, they are also optimized in regards to what kind of blocks the technology has available. This means the synthesis and mapping tool will have to flatten the multiplier tree to perform the optimization, and therefore might use different gates than those we assume (from section 2.6) in our estimation.

5.4.2 Implementation of data extraction

A problem when the netlist is flattened by the synthesis tool, is that the data of the internal structure is lost. This makes it a lot harder to determine why the delay is the way it is, because it is hard to determine which element gives which delay. The netlist only contains the name of the technology block, input and outputs to the block, interconnections and delay. The name of the block has been is lost, and instead of grouping the full- and half-adder together, they are split up to improve optimization. The multiplier tree is just a list of elements named $U < number >$.

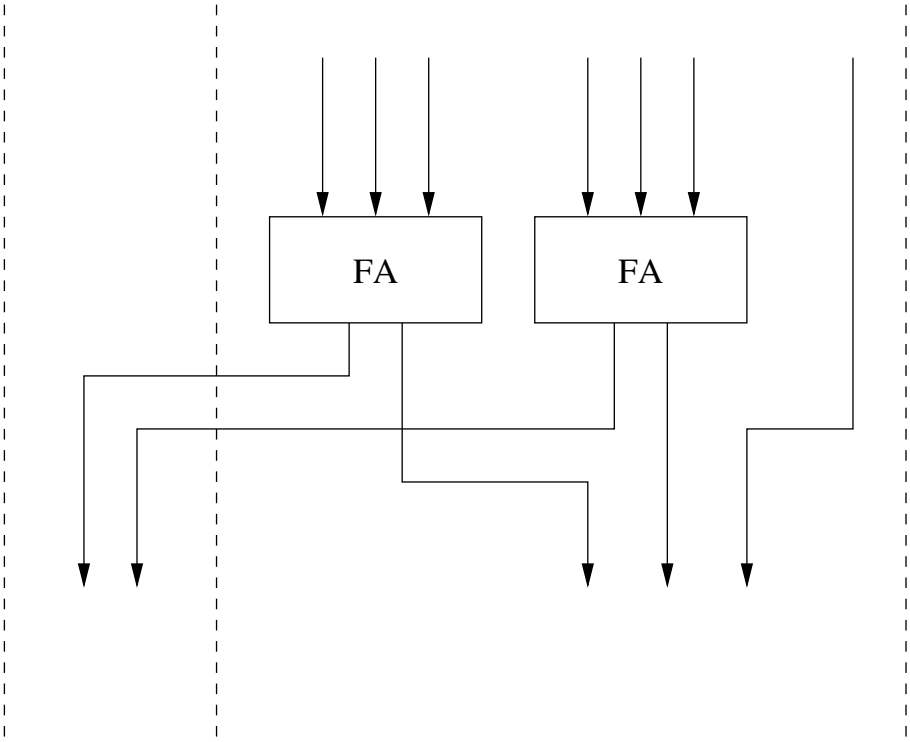

Figure 5.4: Example of reduction of a seven PP inputs

To determine where delays are introduced, a SDF-parser was written in Perl (see Appendix B.1). Since most of the structure information is gone in the SDF-file, a simple program counting each element in the netlist was written (code in Appendix B.2). Since the multiplier tree is the main component in the multiplier, the elements used the most should be a part of the multiplier tree. The program also print the truth table of each element, so it easy determines what the function of the element is. A text file containing the boolean function of each element in the technology library was supplied by Johnny Pihl at Atmel Norway, and was used to produce the truth table of each element.

Some of the elements were identified as parts in a full-adder. The technology library has a triple XOR element, which is assumed used to calculate the Sum-bit in a full adder (see Figure 2.6 on page 12). An element with three inputs was identified as a Carry-generator. These two technology blocks are assumed to be a full adder. It is interesting to see that the synthesis and mapping tool inserts two separate elements for generating the full-adder. As we see in Figure 2.6, one of the XOR-gates in this configuration is shared between the Sum and Carry output. This might mean the synthesis tool adds more elements to the design than necessary, and the multiplier might use more energy.

There are around 900 FAs in a 32×32 bit multiplier, but there were only around 250 of each these two elements. This means the synthesis tool designs full-adders differently, depending on unknown circumstances. The list over counted elements provided some information on the distribution of other elements, and NAND and XNOR-ports were the most used ports in the design. These ports are however also the building blocks for half-adders.

To recognize the full-adders from half-adders, a tool searching for a specific type of element (code in Appendix B.3), and reporting the connections this element has, was written in Perl. A small portion of the elements were manually examined. The connections the element has is

Element	Name	Function
Triple XOR	3XOR	Sum element of FA
Carry	3CARRY	Carry element of FA
XNOR	XNOR	Sum element of HA (inverted)
NAND	NAND	Carry element of HA (inverted)

Table 5.1: Technology mapping for adders

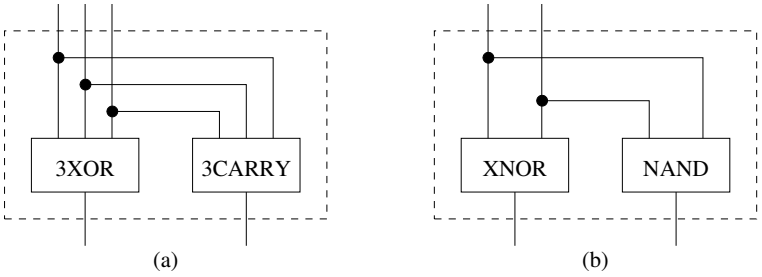

Figure 5.5: (a) Assumed technology mapping of some FAs (b) Assumed technology mapping of some HAs

drawn on paper, trying to determine a pattern of how the FAs are build by the synthesis tool. This research did not come to a conclusion. It seems as if the synthesis tool optimizes the adders together, so two full-adders or a full-adder and a half-adder might share technology blocks. This makes it hard to determine the delay through a single adder. Instead we have to calculate the delay through some of the adders in the design, and only those using the triple XOR and carry element. This gives us less elements to examine, but enough elements to give us an indication of the delay.

The delay through the half-adder suffers similar drawbacks. It is hard to determine which elements are used to generate half-adders. By assuming the XNOR and NAND correspond to the XOR and AND port in Figure 2.6, it should be possible to extract delay for HA adders. The function is the same, if the outputs are inverted.

An extraction tool was written in Perl (code in Appendix B.4). This extraction tool searches through the netlist, and extracts the elements described in this section (shown in Table 5.1). To verify that the extracted element is in fact part of an adder, the tool checks the connections to the inputs of the extracted element. This is illustrated in Figure 5.5. The triple XOR (3XOR) has to be connected to a carry block (3CARRY) to be assumed a FA. The extraction tool only reports delay information for elements verified this way.

The delay from the extracted elements from the netlist are written to datafiles, and can be put into Octave or Matlab to create graphs. Histograms of the delay are presented in Chapter 7 and in Appendix A.

Chapter 6

Implementation of power optimization

6.1 Algorithm optimization

Since the netlister is very modular, two new algorithms were added. The first one was added as an extreme point, and tries to use as few half-adders as possible. The second algorithm was added due to a wrongful implementation of a Dadda reduction scheme.

The reason why as many algorithms as possible is added to the implementation, is because very little research comparing power consumptions in different multipliers were found. By implementing three well known algorithms together with the original algorithm used in the netlister [1] and two new algorithms proposed here, it is possible to compare all algorithms. This might lead to a definite answer of which algorithm is the most power effective.

6.1.1 Conservative

This algorithm tries to use the minimum amount of half-adders, without creating a too deep tree. The algorithm is very simple, and is based on three simple rules.

1 Add $\lfloor b_i/3 \rfloor$ FAs in each column, where b_i is the number of bits in column i.

2 Start at the rightmost column. If the column has two bits left ($b_i \bmod 3 = 2$) and no column to the right of the current column has more than two output bits ($o_{i-n} <= 2\,,\ \text{for all}\ n \in [1, i]$) and o_i is one (where o_i is the number of outputs from column i already assigned), add a HA. If not, then transfer the remaining bits to the next stage. Perform the same test on the next column, until all columns are checked.

3 If any column has more than two partial products, another stage is needed. Run the algorithm again with the new set of partial products.

The algorithm is a greedy algorithm that adds as many FA as possible. The reason for the second rule is to prevent the tree to get too deep. HAs are mostly added at the last stage to prevent the use of FAs to propagate an extra carry bit to the leftmost column. For each column a carry bit needs to be pushed to the left, an extra reduction stage is needed when only using FAs. With the help of HAs, this can be done in one stage.

In the left part of Figure 6.1 we have an example of only adding FAs. The figure shows that we need three stages to reduce the relatively small collections of wires. If we, on the other hand, use rule 2, as we see on the right part of the figure, it can be done in one stage,

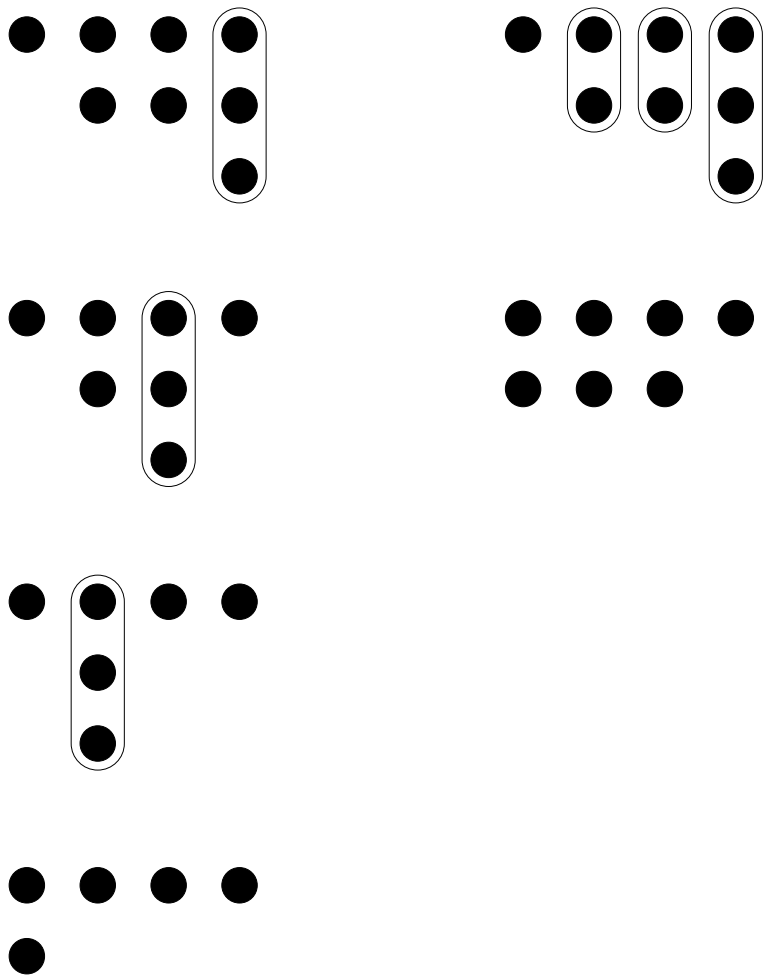

Figure 6.1: Left: Propagation of an extra carry bit using FAs only. Right: Propagation of an extra carry bit using both FAs and HAs

and therefore is much faster. This might cause the need for a larger VMA, but only if this occurs to the rightmost bits. It is still probably worth because of the speed gain. Worst case is that we get extremely long chains similar to the one on the left side.

A problem with this algorithm is that is does not guarantee a shortest possible tree. In some cases you might end up with a deeper tree than by using Wallace, Dadda or RA. Testing indicates that the algorithm in some cases needs an extra reduction stage. But since this tree generator optimizes with regards to power, it should not be written off before we see if it uses less power than the other alternatives.

This algorithm uses the least amount of HAs of all the algorithms tested, and uses about the same amount of FAs as the Reduced Area multiplier algorithm. However it uses a VMA as large as Dadda.

6.1.2 Almost Dadda-trees

The concept behinds this algorithm is to try to balance the tree to get a multiple of three in each column after each stage. It is in most cases ineffective to add an HA, if that prevents adding an FA in the next reduction stage, since FAs actually removes a bit, but HAs only

send one of the bits to the next column. It is called almost Dadda since the Dadda algorithm also tries to utilize the FAs as good as possible. But this algorithm tries to add FAs as soon as possible instead of as late as possible, to make the tree more delay balanced.

1 Add $\lfloor b_i/3 \rfloor$ FAs in each column, where b_i is the number of bits in column i.

2 Start at the rightmost column.

   2.1 If the column has two bits left ($b_i \bmod 3 = 2$) and no column to the right of the current column has more than two output bits ($o_{i-n} <= 2\,,\ \text{for all}\ n \in [1, i]$) and o_i is one (where o_i is the number of outputs from column i already assigned), add an HA.

   2.2 Or if the column has two bits left ($b_i \bmod 3 = 2$), then add an HA if it enables the creation of an extra FA at the next stage ($o_i \bmod 3 = 2$).

   2.3 If not, then transfer the remaining bits to the next stage.

   Perform the same test on the next column, until all columns are checked.

3 If any of the columns has more than two partial products, another stage is needed. Run the algorithm again with the new set of partial products.

The algorithm is equal to the Conservative algorithm, but it has an extra rule. Point 2(c) is there to add HAs to the design if this makes it possible to add another FA at the next reduction stage. This way we get as many FAs as possible in the design as soon as possible, and thus reduces the number of lines through the multiplier.

This algorithm uses the most FAs of all the algorithms. This algorithm might not be the best with regards to area and speed, but since it is delay balanced, it proves to be quite energy effective.

6.2 Interconnect optimization

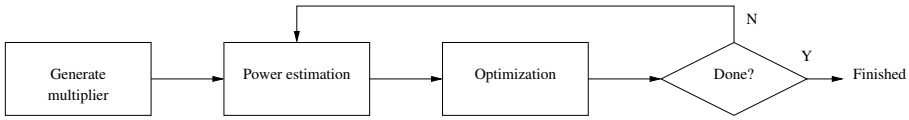

Figure 6.2: Flow diagram of optimization

The implementation of the optimization is a simplified version of Oskuii's [5] post optimization algorithm. The reason it was simplified for this implementation is that the simulator uses a couple of seconds to achieve a power estimate. In the original approach, each adder required several power estimations, and would therefore use a long time optimizing. The simplification is done to reduce the runtime of the optimization considerably.

6.2.1 Implementation

The method used in this implementation uses the estimator to calculate the activity in each gate, which provides a power or activity profile for the multiplier tree. The optimization routine then uses this profile to rearrange the interconnections. This implementation rearranges the whole multiplier based on one estimation, and chooses one mutation, based on

Port	Priority	Paired with
Port A and Port B, on FA	1	Low transition partial products
Port C, on FA	2	
HA	3	
Feed-through	4	High transitions partial products

Table 6.1: Priority used by the optimizer. High priority means high power consumption. Port names from Figure2.5 on page 11

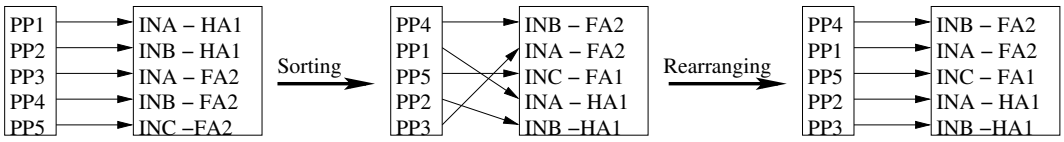

Figure 6.3: An example of optimization of a column

the criteria set by Oskuii [5] (see Section 4.2). The implementation chooses the mutation of interconnections it sees as the best fit, opposed to trying out different mutations. This is done by sorting the partial products and ports of each column in each stage, and then pairing the highest activity partial product with the port causing the least power consumption. Figure 2.5 on page 11 contains the assumed design of the adders used in the multiplier tree.

Feed-through ports are not really ports, but internally used elements to indicate that the partial product should be forwarded to the next stage without any action. These lines therefore do not contain any gates, and should be considered to have very low or no power consumption. Half adders (HA) have only one gate as their critical path for both input ports, and is considered to have low power consumption on transitions. The full-adder (FA) has an unbalanced layout, and the input ports are therefore considered separately. Input C has only a single XOR gate in its critical path to output S, but input A and B have two XOR gates. This should imply that input C has lower dynamic power consumption than input A and B. These are the same assumptions made by Oskuii [5]. The priorities used by the optimizer is therefore as in Table 6.1.

The optimization routine used in this thesis iterates through every stage in the multiplier tree, and in each stage iterates through every column. In each column every partial product is put into a list, which is sorted using insertion sort [37]. Every port which the partial products can connect to, is put into another list, and sorted using the criteria in Table 6.1. We now have two lists, where the top of the partial product list is the partial product with the most activity, and the top of the port list is the port which consumes the least amount of energy per transition. By connecting the top two entries in each list, and doing this kind of pairing for the rest of the list as well, one should get the optimal power usage for his column.

Figure 6.3 contains an example of how a column is optimized. The first stage shows the column before optimization, when the partial products are connected to adder ports at random. The partial products are then sorted, based on the amount of transitions during the power estimation. The ports are also sorted according to Table 6.1. This is the first step in the optimization of a column. As we can see, `PP4` is the partial product with the most transitions, and `PP3` has the least transitions. In the port list, port `INB-HA2` has the highest priority, and port `INB-FA1` has the lowest priority. Ports with equal priority are placed in arbitrary order. This state is shown in the middle part of the figure. The interconnections between the partial products are then removed and reapplied, pairing the partial product

Oskuii [5]	Thesis implementation
Several different interconnections in each column	Tries one solution for each column
Estimates power for each solution	Estimates power for whole design
One iteration for the whole design	Several iterations for the whole design

Table 6.2: Comparison of Oskuii's [5] optimization algorithm and the one used in the thesis

with their corresponding adder port. The is the second step of the optimization of a column, and is showed in the right part of the figure.

After all the interconnections in every column in the multiplier are rearranged, the multiplier should use less power. The optimization routine assumes independency among the gates, but the activity in one gate is dependent on the activity of the connected gates. To compensate for this, another round of power estimation is done. This should update the dependencies, and let the optimization routine base its optimization on more accurate data. An optimization is done based the new power profile, which could further lessen the power consumption. A flowchart of this approach is shown in Figure 6.2. The optimizer will eventually converge to a local minima. This implementation uses ten iterations to be sure the optimization has reached a steady state. This number was found through experimentation.

6.2.2 Comparison

The main difference with this implementation compared to Oskuii's [5] is shown in table 6.2. This implementation only tries one solution of interconnections, while Oskuii's tries several solution. The reason this was simplified, is because of run time issues. The optimization would take a very long time to compute. This will probably make this optimization routine optimize less than Oskuii's method. This disadvantage is however improved upon by running the optimization routine several times. Each run of the optimization routine will improve power, until a steady state is found.

Both Oskuii's [5] method and this implementation will only find a local minima, since it does not try every solution possible. This is an accepted limitation to NP hard [37] problems, since computing trying every solution would be practically impossible to calculate due to the amount of possible solutions.

Chapter 7

Results and discussion

This chapter contains the results of the experiments done for this thesis. The first experiment tries to calculate the delay in a multiplier. This information is then used as delay-parameters for the estimator designed in this thesis. The estimator run-time and accuracy is analyzed. Six multipliers are then generated and optimized, and their optimization are discussed.

7.1 Timing model

Since one of the multipliers supplied is generated with the netlister modified in this thesis, it is natural to use that design for the basis of delay calculation. We have chosen to look at the MAX-PVT condition, as this is closer to everyday temperature in processors. Similar graphs for some of the other multipliers can be found in Appendix A.

Each collection of histograms contains information about one output pin of a adder. The separate histograms describes the delay from each input to the output. The different bars in each graph represents possible events on the input and output. $0 \rightarrow 1$ represents change from zero to one on the output, and $1 \rightarrow 0$ the opposite. The label 'posedge' and 'negedge' explains if a positive or a negative edge on the input drives the transition.

Figure 7.1 shows the delay from the input of a FA to the Sum output bit. It is very interesting to see that the delays varies a lot. Some adders might have double amount of delay compared to others. It would be very interesting to know why the delay varies as much as it does. Due to the time constraints for this thesis, no exploration of the cause to this was done. Speculation suggests the partial products not going to a adder in a stage (feed-through) might add significant delay to that wire. Other reasons might be because of the irregular pattern, the synthesis tool can not put the adders on the silicon in a balanced order, and the delay is therefore very unbalanced.

The load on each output (apart of the wire load) should be the same. Every adder output on each adder should drive exactly two other gates (see Figure 2.6 on page 12). This result shows that delay is very dependent on adder placement.

Another result from the same figure, is that one of the inputs has significantly less delay than the other two. Is confirms the assumption made in Section 4.2, that C has less delay than input A and B. The names of the ports and their order are different though. The synthesis tool might therefore connect the adders wrongfully together, since we assumes input C (the last input) is the fastest, but input A1 (the first input) is in fact the fastest. Some of the post-optimization done in this thesis might therefore not work as intended after synthesis, even though the full adder is explicitly defined as Figure 2.6 in the generated VHDL-file. This is hard to verify, since the structure is flattened, and the wire and port names are lost.

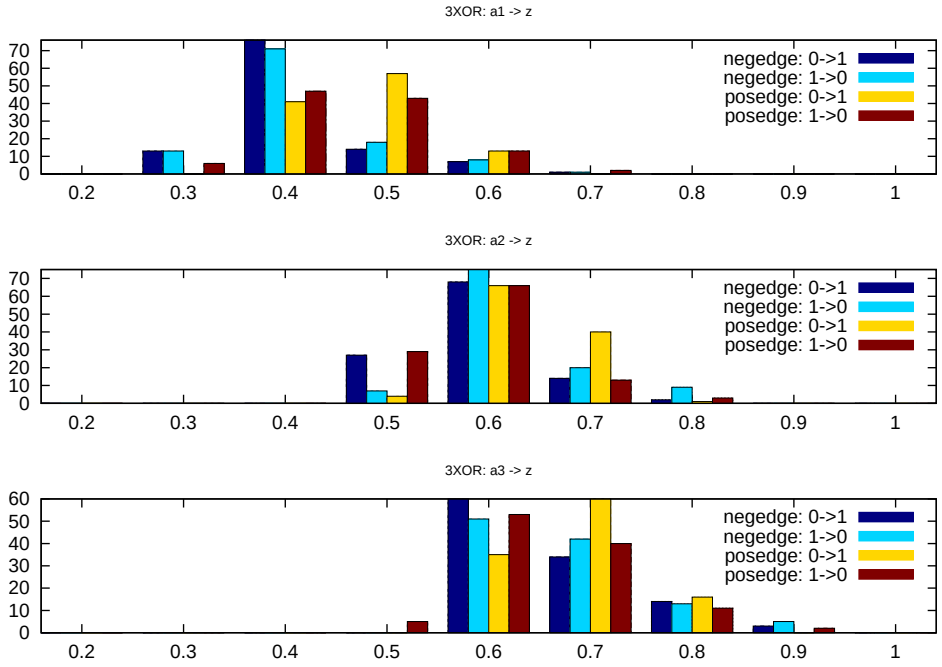

Figure 7.1: Timing through full-adder, from input to sum output

Figure 7.2 shows the delay from the inputs of a FA to the carry output bit. The delays for each input is quite equal to each other. There are minor differences in the delay, but it probably just small variations in the transistors in the element.

The delay is also very spread for this element, probably for the same reasons as the Sum element. Variations from 0.3 μs to 0.6 μs , and some ports have as much as 0.9 μs delay. The delay is about the same as input A1 on the Sum bit, but considerably less than input A2 and A3. This shows a reason why the transitions through the multiplier tree are uneven.

Figure 7.3 contains information about line delay from the SDF-file. The line delay is accumulated for both elements in the FA, and thus for both output lines. As we can see, the line delay is almost zero. The reason for this is that the delay from charging the wires and transistors are baked into the delay of the elements. The estimator therefore sets the line delay to zero, and uses the delay from the elements as configuration instead. The net result should be the same.

The delay of Sum line in the HA is shown in Figure 7.4. The reason XNOR ports are examined, and not XOR ports is because it is assumed that the synthesis tool optimizes the XOR ports into XNOR ports. The output of the XNOR is inverted, and it is assumed the synthesis tool also converts some of the adders to accept inverted inputs. This might explain why it was very hard to find other adder structures in the netlist (as explained in Section 5.4). The delay from the XNOR is used as is, and not with the additional delay of an inverter on the output port. Since it adder structures that accept inverted inputs is possible and does not invert the input is possible, the delay of an inverter would probably create more inaccurate delay data. The sole delay for the XNOR port was therefore chosen as configuration of the simulator.

This element also has varied delay. A interesting note is that the delay varies after what kind of transition is done. This is something the simulator implemented does not take into account. The simulator might therefore be less accurate.

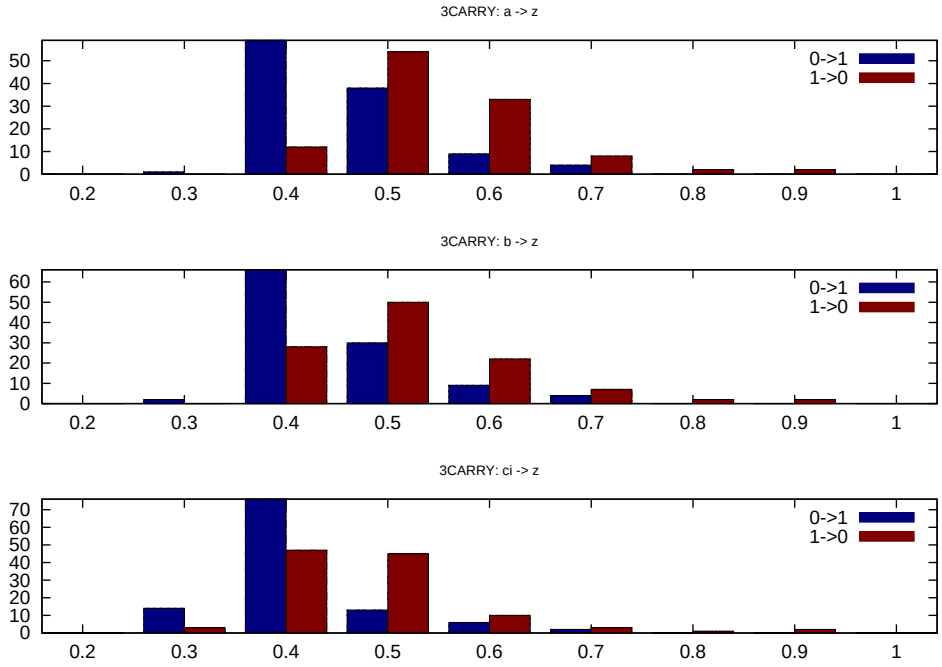

Figure 7.2: Timing through full-adder, from input to carry output

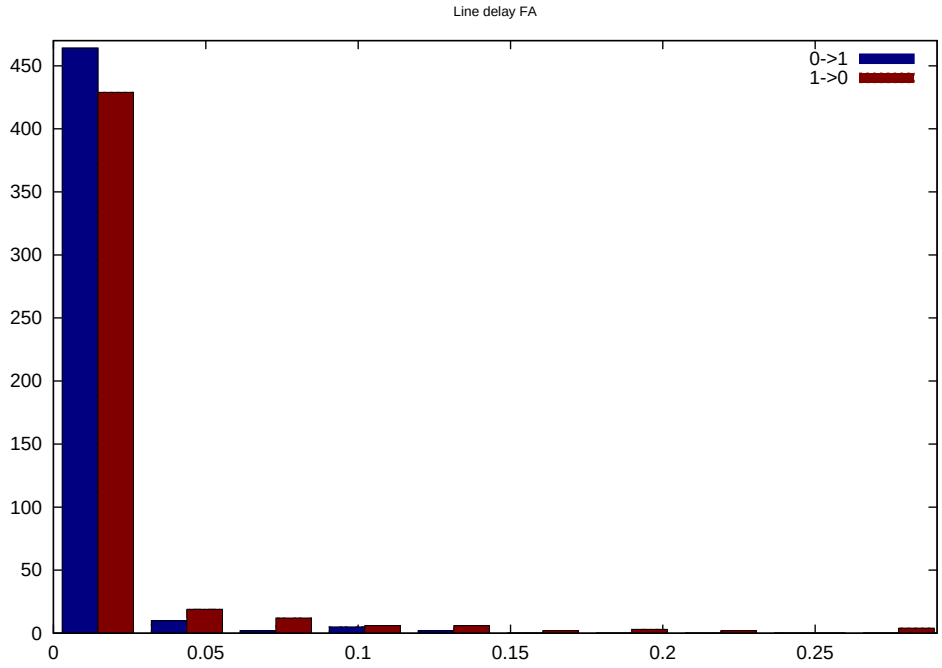

Figure 7.3: Line timing from full-adder to next element

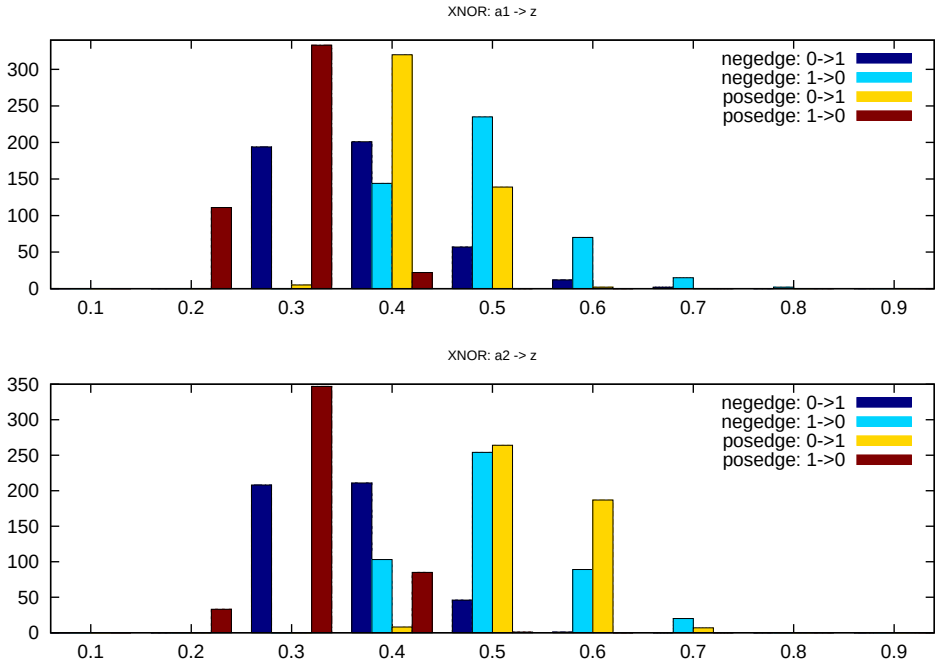

Figure 7.4: Timing through half-adder, from input to sum output

Figure 7.5 contains the delay information for the Carry bit of the HA. This element has the most varied delay of all the elements examined. From 0.1 μs to 0.9 μs . This is quite puzzling, and might indicate that the extraction tool listed NAND ports not part of a HA. The results for the NAND and XNOR ports as a indication of the delay through a HA should therefore be used carefully. NAND ports are also examined for the same reason as XNOR ports were examined instead of XOR ports.

The line delay is shown in Figure 7.6. The delay is almost zero for HA as well, and probably for the same reason as the line delay of the FA.

A mean value for all the delays is calculated, and used to configure the simulator. The mean delay is shown in Table 7.1. Since the delay through the FA is irregular, the low value for the SUM is chosen for one input, and the two other values mean value chosen for the two other inputs. The delay to the carry bit is considered equal for all inputs, and therefore the mean value is used as input to the simulator. Since the simulator uses a gate level approach, the delay is distributed over the gates in the element.

Since we assume that feed-through elements make longer wires, an extra delay has been added to wires going through a feed-through element. This delay has been set to 150 μs . Since we were not able to find correlation about longer delays through feed-through wires in the netlist, this value is just a guess based on the timing variance of the histograms.

7.2 Estimator

The simulator can be configured to sample run a fixed amount sample. Each sample contains the number of transitions made by the multiplier tree to do one calculation. By increasing the amount of samples collected, the accuracy of the estimation is increased at the cost of runtime. This section contains results for the runtime and accuracy of the estimator.

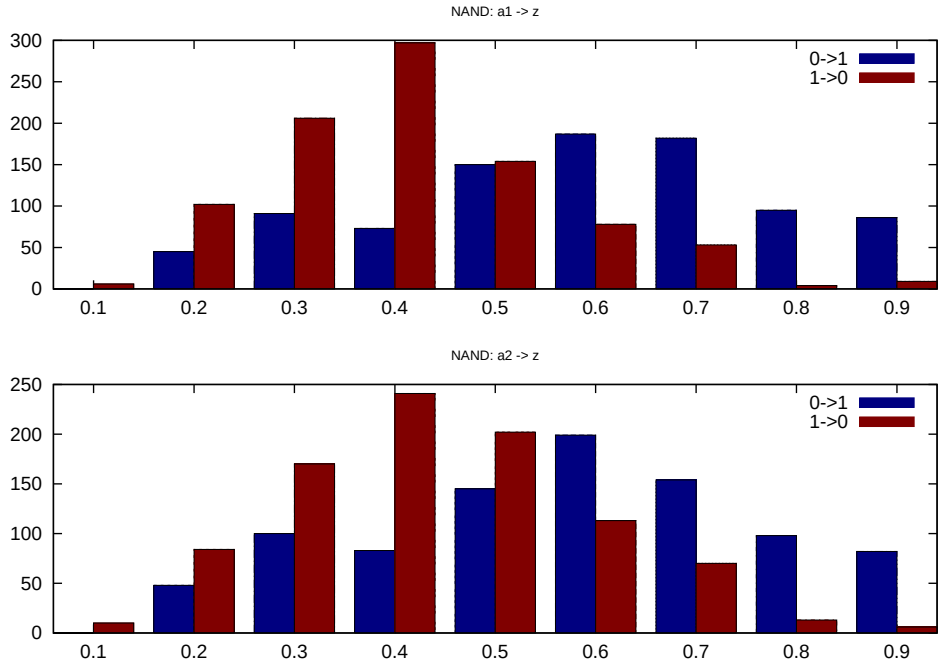

Figure 7.5: Timing through half-adder, from input to carry output

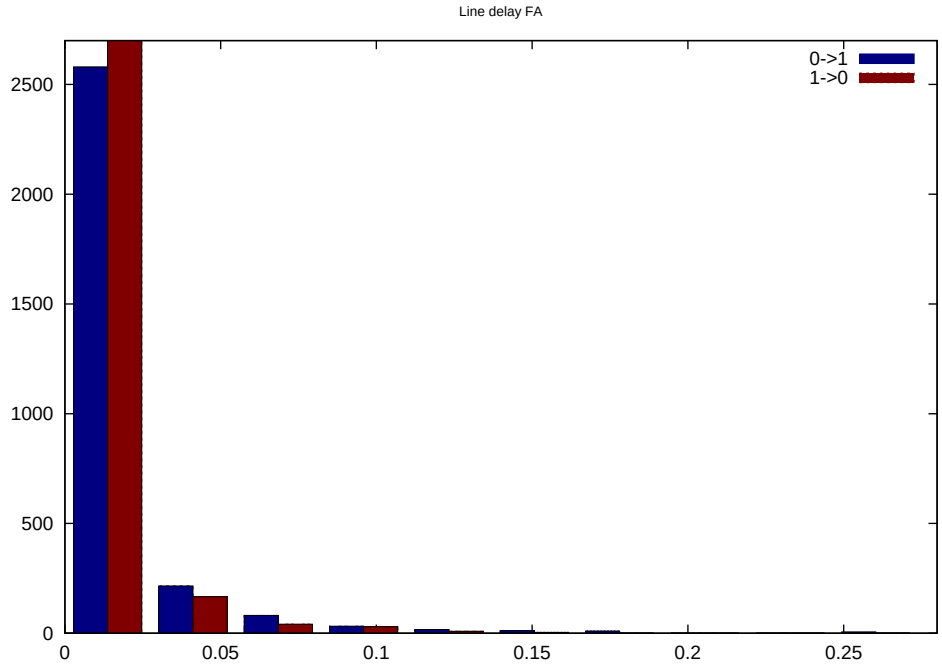

Figure 7.6: Line timing from half-adder to next element

FA	SUM	A1	0.443
		A2	0.613
		A3	0.671
	CARRY	A	0.5015
		B	0.4865
		CI	0.446
HA	SUM		0.508
	CARRY		0.413

Table 7.1: Mean delay through elements

Figure 7.7 and 7.8 contains the runtime of the estimator. As we can see from the figure, one estimation with 100 samples takes around 2.5 seconds for a 32×32 multiplier. This is the reason the optimization on the whole multiplier at once. If the optimization routine had estimated the power usage for several mutations in each column, runtime of the optimization would dramatically increase. The runtime data was gathered using a Intel(R) Xeon(R) CPU X5550 2.67GHz CPU.

The graph also shows that the runtime increases linearly with sample size. Size is expected, as each sample takes a finite amount of time to compute. By comparing the run time across sizes, we can see the estimator's run speed increase linearly as well.

The runtime for other multiplier sizes are given in Appendix A.3.

To determine how many samples are needed, we need to investigate how accurate we want the estimation to be. Figure 7.9 and 7.10 show the average amount of transitions after n samples. For the 8×8 multiplier, the average transitions starts to converge after around 30 samples, and the 32×32 multiplier after around 50.

The estimator should calculate the error of the estimation, as described in Section 3.3, but due to time constraints, this was not implemented. To get the error of the estimation to be as small as possible, a larger sample size where chosen. The sample size also have to be larger to get a accurate power profile of the multiplier, and not just an overall estimate. A sample size of 100 samples seemed through experiments as a good value, considering speed and accuracy.

The graphs also shows how the different multipliers react equally on the same input. By using the same input patterns for all the multipliers, and through the optimization step, the power estimate should be considered very accurate for comparing. This makes the comparison done before and after optimization quite accurate.

The different algorithms use different amounts of power. These graphs represent the number of transitions used by the multiplier, before interconnect optimization is performed on the multiplier. As we can see, the choice of algorithm greatly impacts the power performance of the multiplier. The Wallace algorithm is by far the worst reduction tree scheme to use, considering power consumption. More discussion of power usage of the different algorithms is given in the next section.

7.3 Optimization

Figure 7.11 and 7.12 contains amount of transitions used after 100 samples, shown after n number of optimization runs. All multipliers reduce their power consumption by between 5% and 28%, according to Table 7.2. The algorithm benefiting the most from interconnect

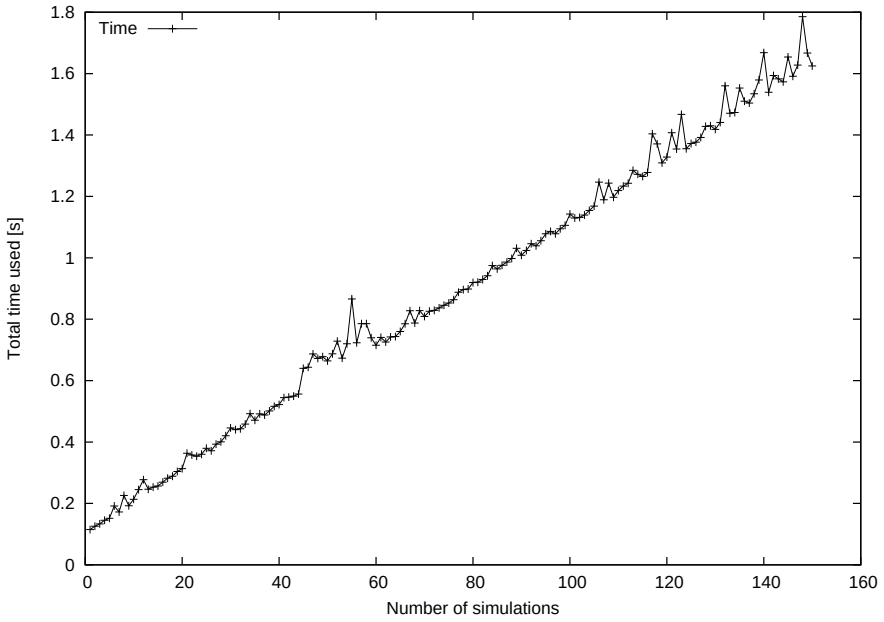

Figure 7.7: Time usage of the estimator, 8×8 multiplier

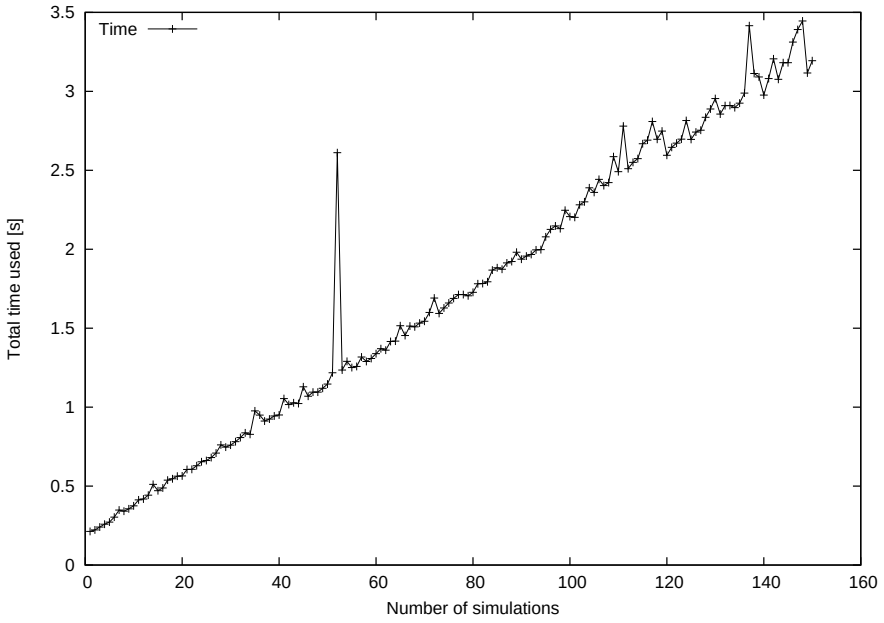

Figure 7.8: Time usage of the estimator, 32×32 multiplier

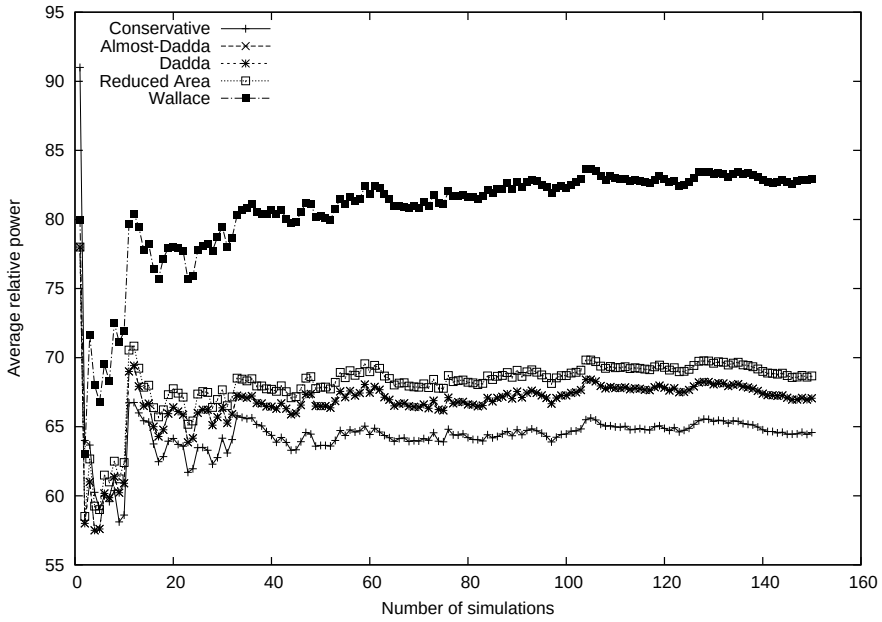

Figure 7.9: Power estimation accuracy of simulations size, 8×8 multiplier

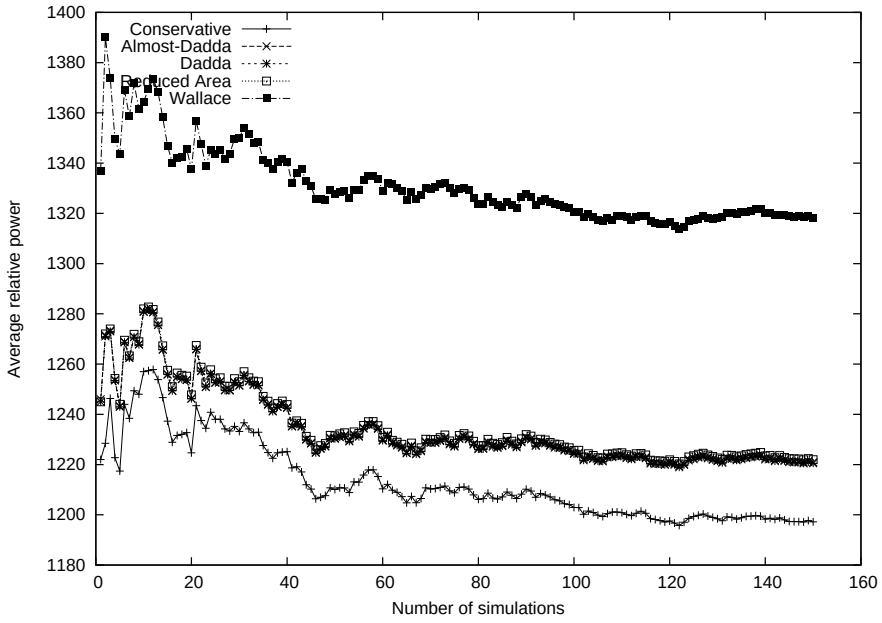

Figure 7.10: Power estimation accuracy of simulations size, 32×32 multiplier

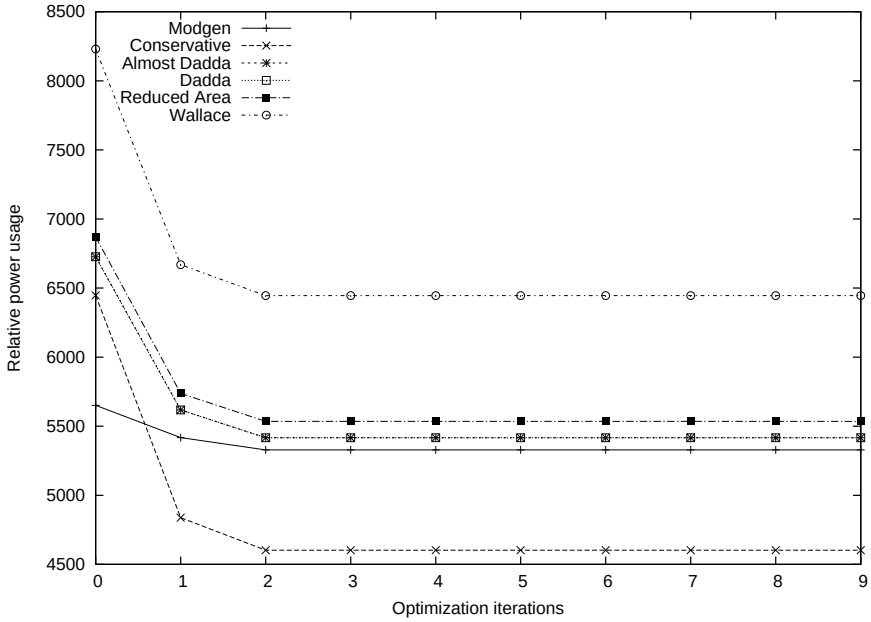

Figure 7.11: Power usage after each optimization step, 8×8 multiplier

optimization is the conservative algorithm. The interconnect optimization seems to reduce the power consumption by a fair deal.

It might seem that the Modgen algorithm benefits the least from the interconnect algorithm. The reason for this might be because the estimator does not provide a power profile that is as good as the one for the other multipliers. Another reason is that the Modgen tree is more difficult to balance.

The other algorithms show a large reduction in power consumption. This reduction might be larger than it is in reality. By using the same input vector for all of the estimations, the multiplier gets optimized based on this input. Another set of input vectors might optimize the multiplier differently, and there is no guarantee this optimization is the best for all input vectors. The random number generator used in this thesis should be good enough to get a very random set of data, but might not be.

The Monte Carlo method should on the other hand give very accurate results for the design as a whole, but lower accuracy about the power consumption of each element. This means that the power profile used by the optimization might not be very accurate, but it seems to be accurate enough to optimize the design. The estimation of the whole design should be accurate, given the delay assumptions made earlier. For this reason, the estimation should be quite pattern independent, but an increase of samples per estimation or a calculation of the error margin could shed further light on the topic. This was not possible to do, due to the time limit of the thesis.

Figure 7.13 shows a estimated gate count for different multipliers in different sizes. The estimation uses five gates for FA and two gates for HA as a basis for the calculation, because this configuration is a very common adder structure (see Figure 2.6). Note the logarithmic scale in the graph. This shows that the different reduction algorithms use about the same

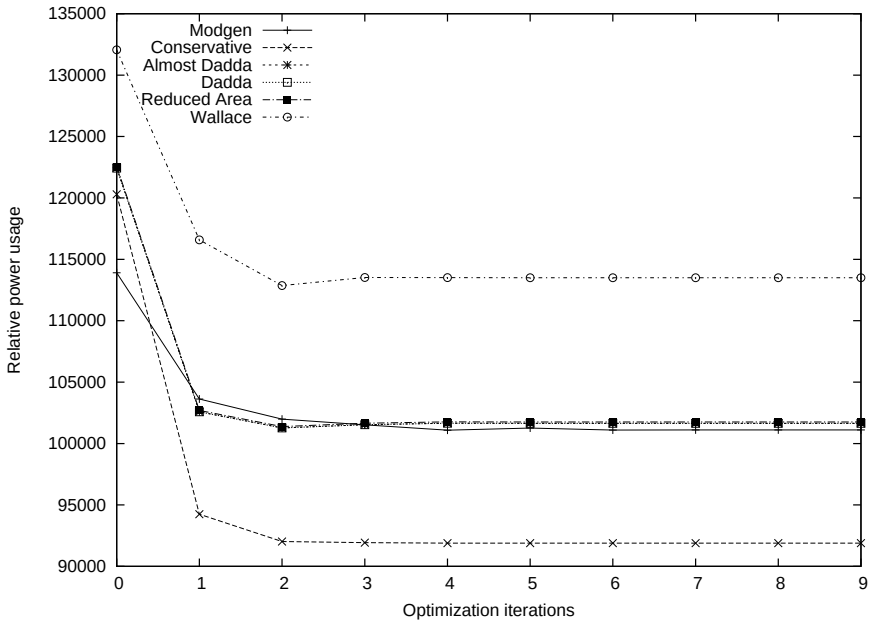

Figure 7.12: Power usage after each optimization step, 32×32 multiplier

	8×8 multiplier			32×32 multiplier		
Name	Pre opt	Post opt	Change	Pre opt	Post opt	Change
Modgen	5651	5328	-5.51%	113909	101105	-11.2%
Conservative	6447	4602	-28.6%	120291	91895	-23.6%
Almost Dadda	6726	5416	-19.5%	122465	101645	-17.0%
Dadda	6726	5416	-19.5%	122423	101645	-16.9%
Reduced Area	6870	5535	-19.4%	122559	101758	-16.9%
Wallace	8230	6445	-21.7%	132054	113501	-14.1%

Table 7.2: Improvement by optimization

amount of gates, with the exception of Wallace. The Wallace algorithm uses slightly more gates than the rest. The graph also shows that the size of the multiplier grows exponentially.

To compare the different algorithms, their transition count after interconnect optimization has been placed in Figure 7.14. Note the logarithmic scale of the Y axis. Since the graph does not show clearly the differences between the algorithms, a percentage difference between the algorithms is shown in Figure 7.15. The algorithms are compared again the Modgen algorithm, and this algorithm is therefore represented as 0% in the graph. A value over 0 means the algorithm consumes more power than the Modgen algorithm, and under 0 means less power.

As we can see the power consumption clusters into three groups, Wallace, Dadda-like and the Conservative. The Wallace algorithm performs the worst of all the algorithms, and it is also the algorithm using the most adder elements. This might explain why it is the least performing candidate.

It seems the Dadda algorithm should be used as the preferred algorithm when comparing power consumption with other techniques, which is nearly the same age as Wallace, and performs a better and use less area [6]. Wallace is more famous than Dadda, despite not performing equally well.

The cluster of algorithms called Dadda-like earlier contains the original Dadda, Almost Dadda, Reduced Area and the original Modgen algorithm. Both the Reduced Area and Almost Dadda algorithms are modifications to the original Dadda algorithm. The original Modgen algorithm uses a greedy approach, and tries to reduce the number of HA elements, which also is a Dadda-like behavior. The similarities of these algorithms might be the cause of their very equal power consumption. They also use very similarly amount of adders in the tree.

The Conservative algorithm uses the least amount of power after interconnect optimization. It also has the least amount of HAs, while still having fewer FAs than both Dadda and Reduced Area. The amount of full- and half-adders used by the different algorithms are found in Appendix A.6. The Conservative algorithm performs consistently better for multiplier sizes from 8×8 to 32×32. Graphs showing transition counts for other multiplier sizes are given in Appendix A.5.

The reason the Conservative consumes less power than the other algorithm might be because it has fewer adder elements then the other algorithms. Less transistors to switch might transfer to less power used.

7.4 Post-layout analysis

A few of the designs were chosen and sent to Johnny Pihl at Atmel Norway, and synthesis and technology mapping was done by him. The designs were sent through "IC Compiler", to get an estimated power usage after layout. This is a coarse power estimate, put should give an realistic indication of how much power consumption is actually reduced. The results are presented in Table 7.3.

Since the assignment says the work done in this thesis should reduce power consumption of the generated by the netlister, a multiplier generated by the original netlister was used as a basis for the comparison. The other algorithm used for synthesis is the Conservative. It was chosen because it seems to be the algorithm with the least power consumption. Both algorithms were synthesized before and after interconnect optimization.

All of the designs were implemented as full multipliers, containing a booth recoding step, the reduction tree and a VMA step. This was done to examine the overall power reduction,

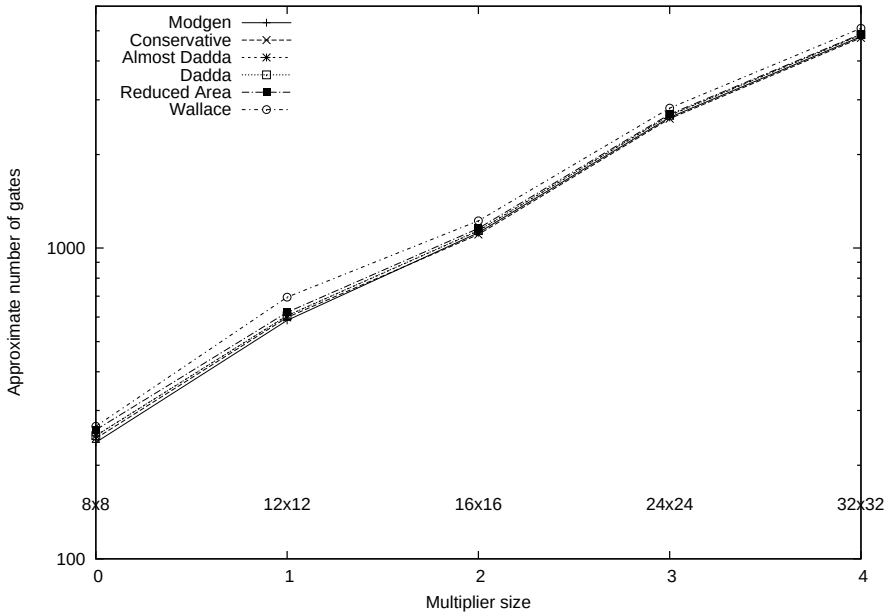

Figure 7.13: Approximate of transistors for each multiplier tree

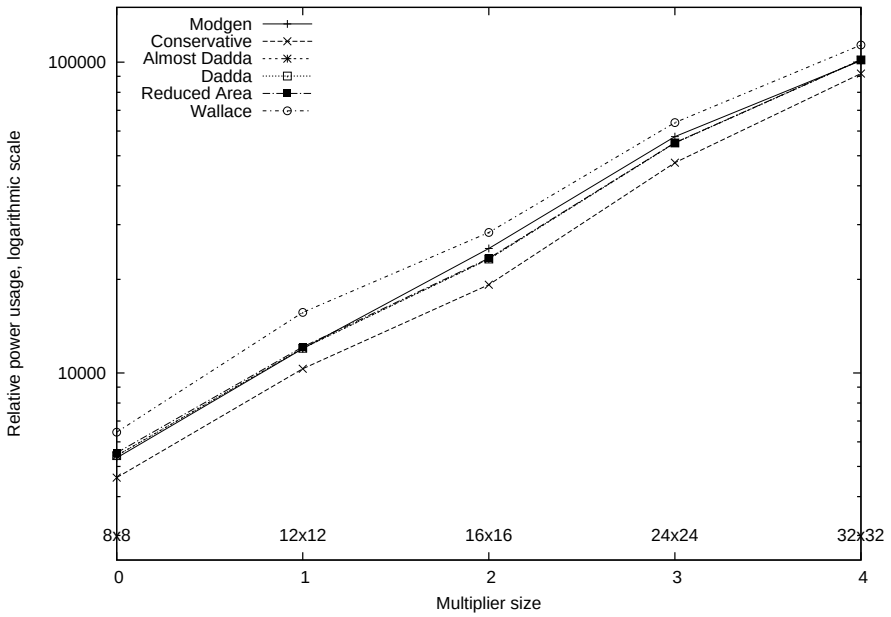

Figure 7.14: Power estimation after optimization for different multiplier trees

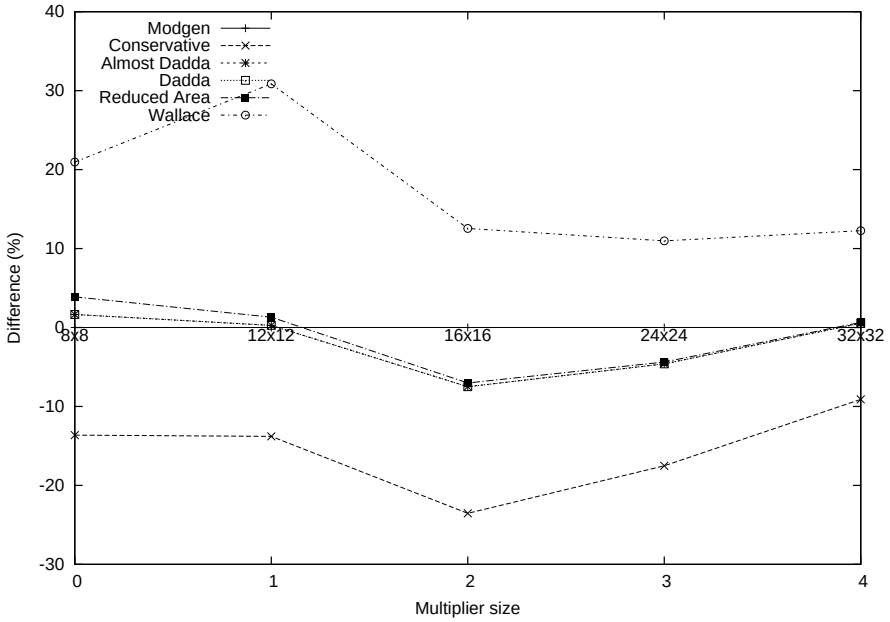

Figure 7.15: Power comparison after optimization for different multiplier trees

and not just from the reduction tree, since the size of the VMA varies among the different algorithms.

It would have been beneficial to compare all of the algorithms used in this thesis, but this was not possible given the time constraints of the thesis.

The results in Table 7.3 shows a 11% decrease in power when choosing the conservative over the original Modgen algorithm. This is inconsistent with the results of the simulator used in this thesis, which show the Modgen algorithm to be more energy effective before optimization. The reason for this might be inaccuracy in the simulator used in this thesis. The power calculation done in IC Compiler is also using the standard settings, and this might not put the multiplier under standard multiplication operation. The result is also done with a booth recoder and a VMA, which power usage is not estimated by the simulator in this thesis. These are probably the main reasons for the deviation.

The effect of interconnect optimization is low in the post-layout design. It is between 2.5% and 1.0%. This might be because the synthesis tool does not interconnect the multiplier tree the way we assume. Since we are exploiting the fact that one of the inputs in a FA is faster than the others, it is important that this assumption gets transferred to the synthesis tool. The synthesis tool is doing optimization as well, and this might interfere with the optimization done in this thesis.

		Change from	
Design	Effect	Original	Conservative
Modgen Original	54.6909 mW	0 %	13.17 %
Modgen Original Optimized	53.2783 mW	-2.58 %	10.25 %
Conservative	48.3245 mW	-11.64 %	0 %
Conservative Optimized	47.8362 mW	-12.53 %	-1.01 %

Table 7.3: Effect used by complete multipliers, before and after interconnect optimization

Chapter 8

Conclusion

8.1 Delay and estimating

The investigation of how the delay varies between elements inside the multiplier is important because it shows us that the delay is very varied. By using a too coarse delay model, the estimated power consumption might not be as accurate as intended. The main reason for the varying delay is possibly the different length of the wires, so more research about why the wires are longer could possibly lead to better ways of optimizing multiplier trees. The delay in the multiplier is the decisive reason for the extra power consumption through glitching, and is therefore also important for the optimization. The delay data found in this thesis can be used as a starting point for a better delay model used by an estimator or optimization routine.

A power estimator is very important when comparing different multipliers. It is used as the basis for the comparison. The estimator might also be used in the optimization, to verify that one solution is better than another. This thesis contains an implementation of a power estimator that can be reused and improved to help further research in the field of low power circuits.

By using the delay data it is possible to train the estimator to make estimations based on which synthesis and layout tool that is being used. Since the delay data can be changed within the estimator, it is possible to analyze a SDF-file made from another synthesis tool with a different technology library, and use that delay information instead. The current implementation is dependent on the synthesis tools and technology library used, but it is possible to change the estimator to be dependant on another tool and library. This, however, must be done manually. An improvement of the SDF-reading tools would be able to produce delay information from SDF-files automatically.

8.2 Multiplier generation

Choosing the best algorithm when generating a multiplier tree can reduce power consumption with at least 10%. This emphasizes the importance of using different kinds of generation methods when comparing multipliers. The different algorithms have different properties. This thesis also shows that more research should be put in the subject of which reduction tree method that is preferred for low power usage, and not only research concerning size and delay.

The interconnect optimization shows a reduction in power consumption in the multiplier tree. How much the gain the optimization gives is not verified in this thesis. The synthesis tool tries to optimize the circuit by changing the type of gates used inside the adders. This

might negate some of the power improvements shown in the pre-layout estimation. By trying different synthesis and technology mapping setting for the tools, it might be possible to force the synthesis tool to make less optimizations and transformations during the process. This can improve the usefulness of the interconnect optimization, but might not give an overall power improvement. Synthesis tools are mature software, and generally good at optimizing, but they usually optimize in regards to size and delay, and not power.

The algorithm proposed by Oskuii [5] should be investigated further, as this thesis shows that a simplification of the algorithm produce power reductions. The original algorithm should improve power usage better than the implementation used in this thesis. It is very important to use a good estimator when employing the algorithm, and an accurate timing model is important to model the glitches through the multiplier properly.

The netlister improved by this thesis should produce power effective multipliers. It also gives researchers a tool to fast and easy generate a lot of different multipliers. The tool can be used to research a broad range of different multipliers with different configurations. This thesis does not contain a survey of how the multipliers perform after post-synthesis and layout. The netlister should provide a tool that enables fast and easy generations of a lot of different multipliers, and would be useful for anyone trying to make such a survey.

8.3 Directions for further work

The timing model should be improved. Now it only uses a mean value through the elements, and a delay penalty when being feed-through. The delay penalty for the lines routed through stages is only assumed in this thesis, and suggests that further work should therefore investigate if this is indeed the The reason for the different delays are known to be different wire length between adders, and it would be very interesting to be able to predict wire length. Since the workings of the synthesis tool is in most cases proprietary, it is apparently hard to get the wire length estimation 100%, but there should be a possibility to be fairly accurate.

By estimating wire lengths, one can use the data in the optimization. The long and slow wire should be used sparingly, and thus be connected to low density gate outputs. By having this information it might be possible to do optimization without doing estimation for each step.

The estimator implemented in this thesis should be extended to calculating the error from the Monte Carlo method. This way it would be possible to stop after a dynamic number of samples instead of a fixed amount. This might decrease running time, and make the netlister faster. If the estimator gets further improved, it might be able to use it to estimate all the different mutations in the tree, instead of the simplified optimization routine used now.

Gate-level estimation is probably not as accurate as post-layout power estimation. A survey of the power consumption of different configurations should enlighten how well the estimator works, and which solutions will reduce power consumption in real implementations. The netlister includes possibilities to add Booth recoding [15, 17] and pipelining. These additions could also be surveyed, coupled with different VMA configurations. The netlister currently only supports one type of VMA, but several variations could be implemented.

Appendix A

Timing models

A.1 Modgen multiplier, min PVT

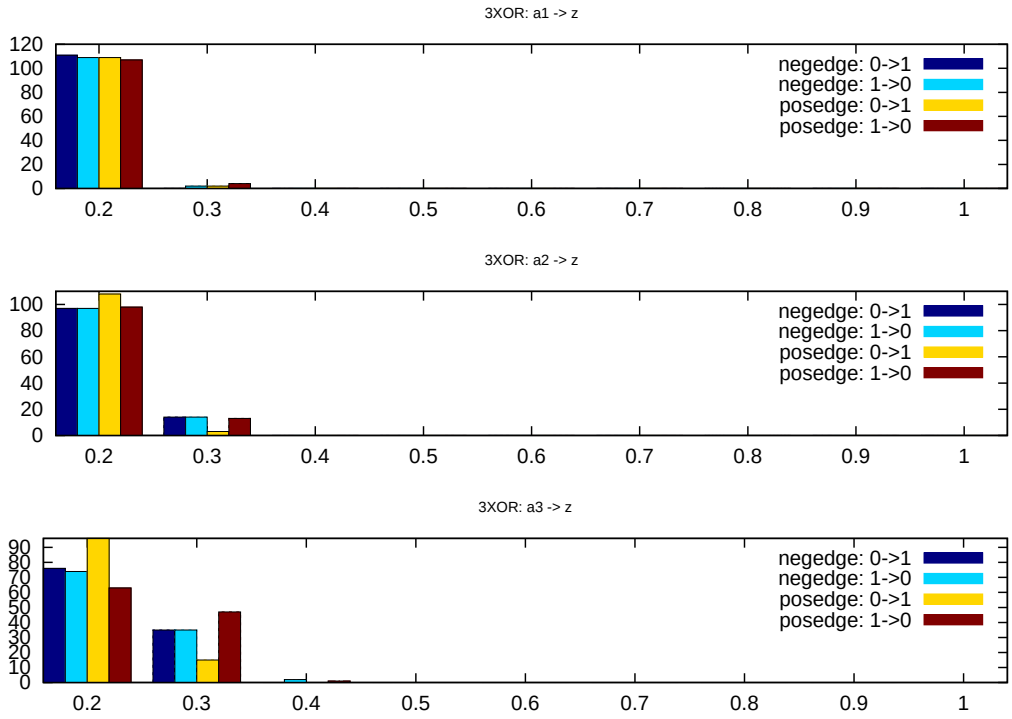

Figure A.1: Timing through full-adder, from input to sum output

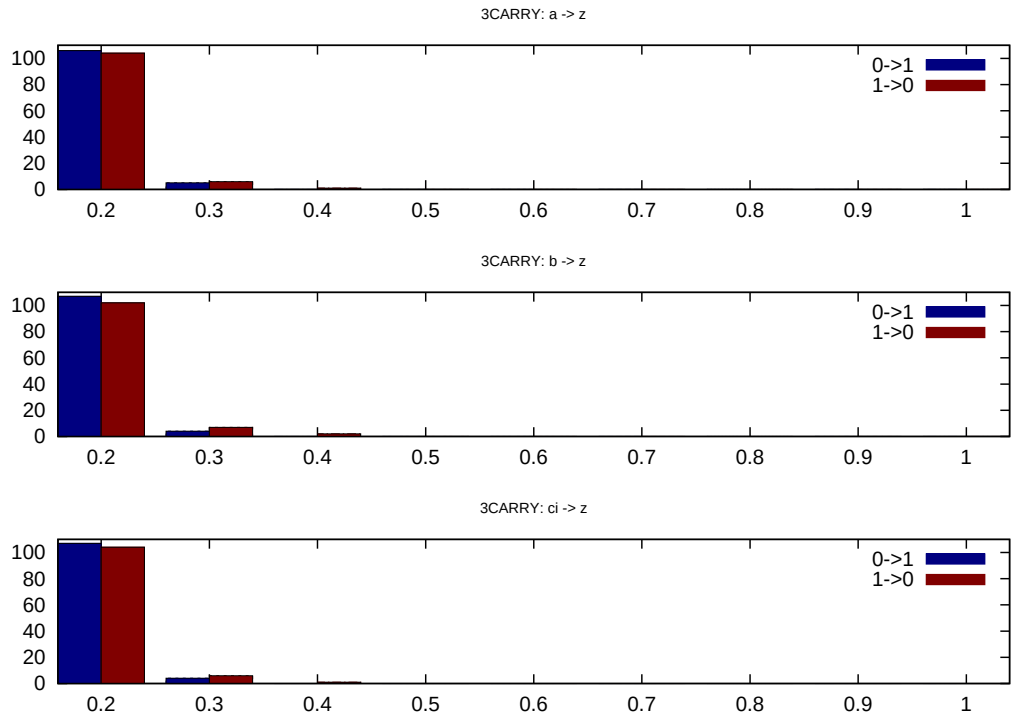

Figure A.2: Timing through full-adder, from input to carry output

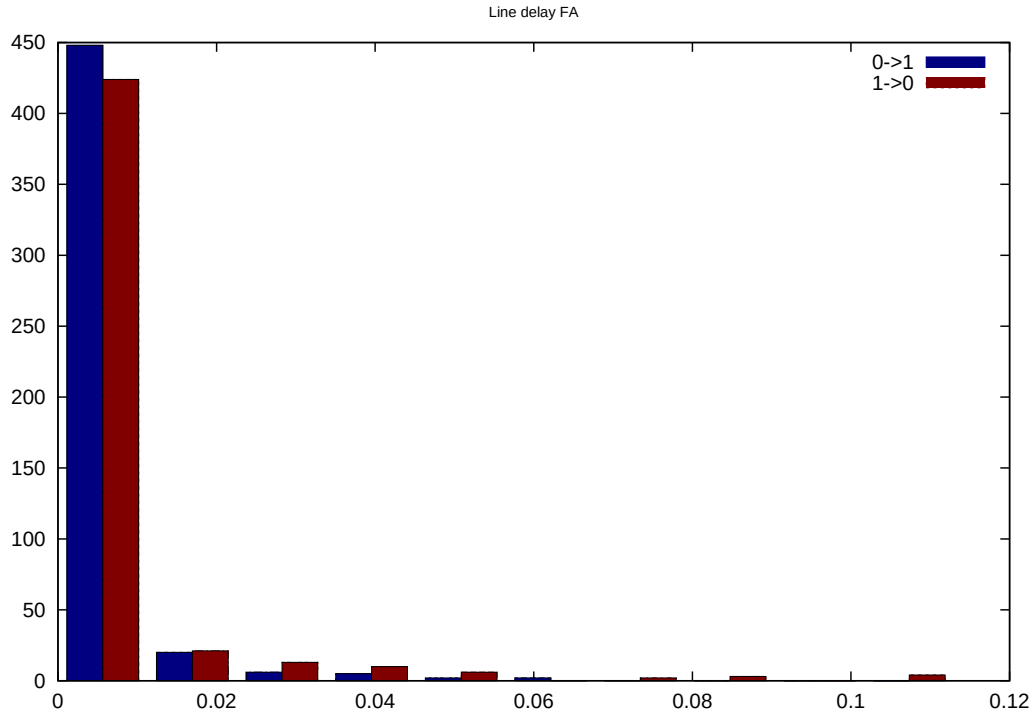

Figure A.3: Line timing from full-adder to next element

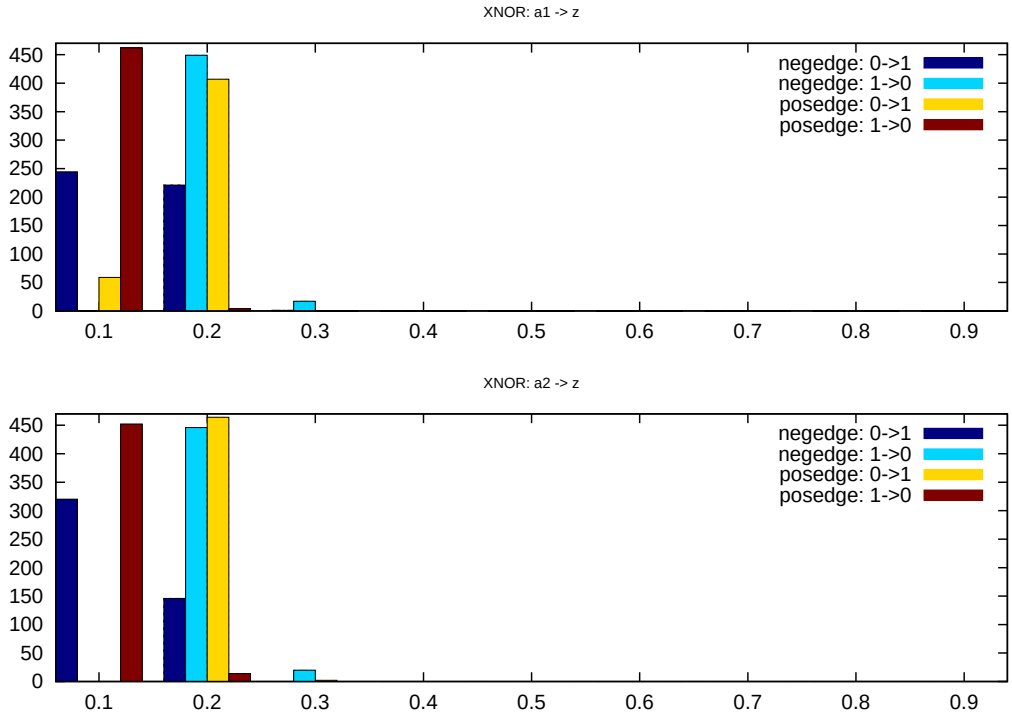

Figure A.4: Timing through half-adder, from input to sum output

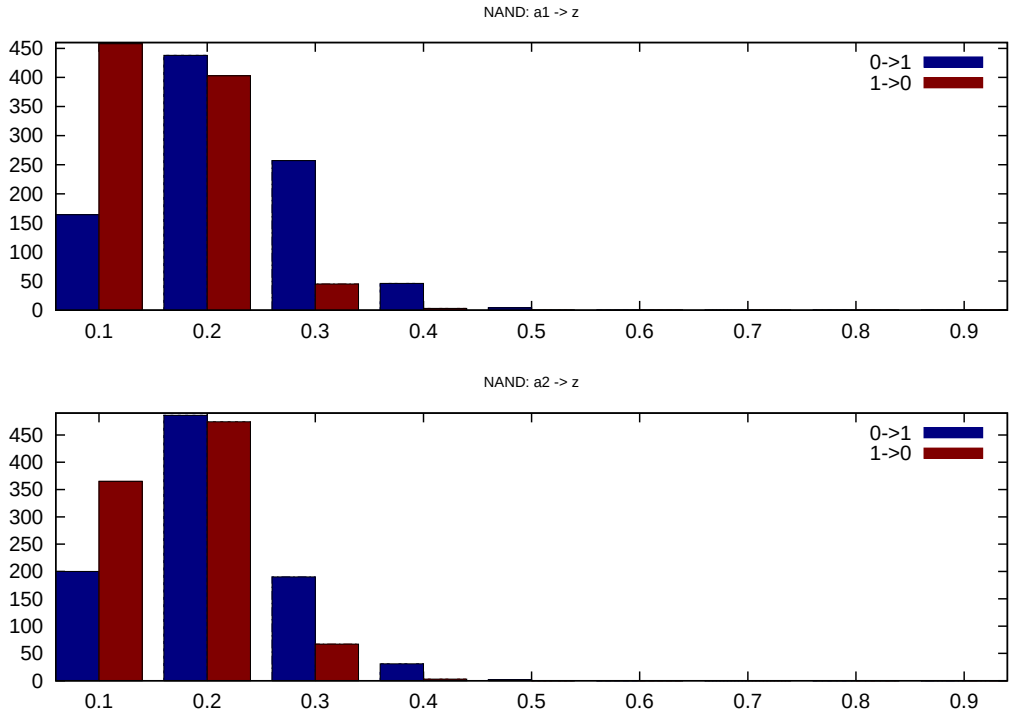

Figure A.5: Timing through half-adder, from input to sum output

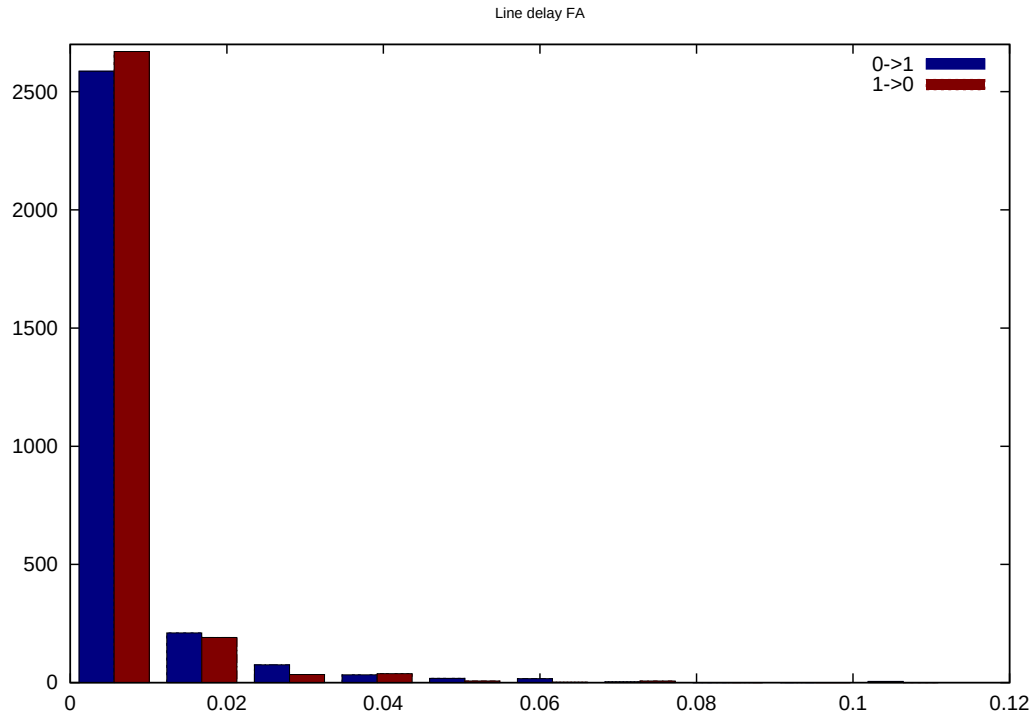

Figure A.6: Line timing from half-adder to next element

A.2 Generated multipliers

Multiplier generated in [5]

A.2.1 Optimized for minimum power, max PVT

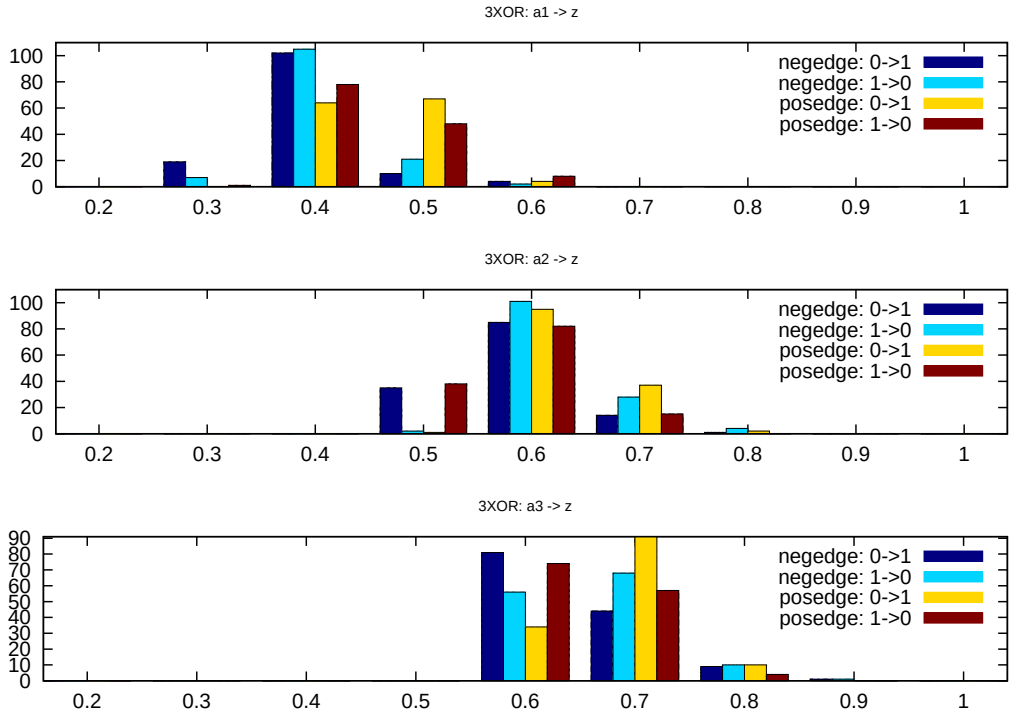

Figure A.7: Timing through full-adder, from input to sum output

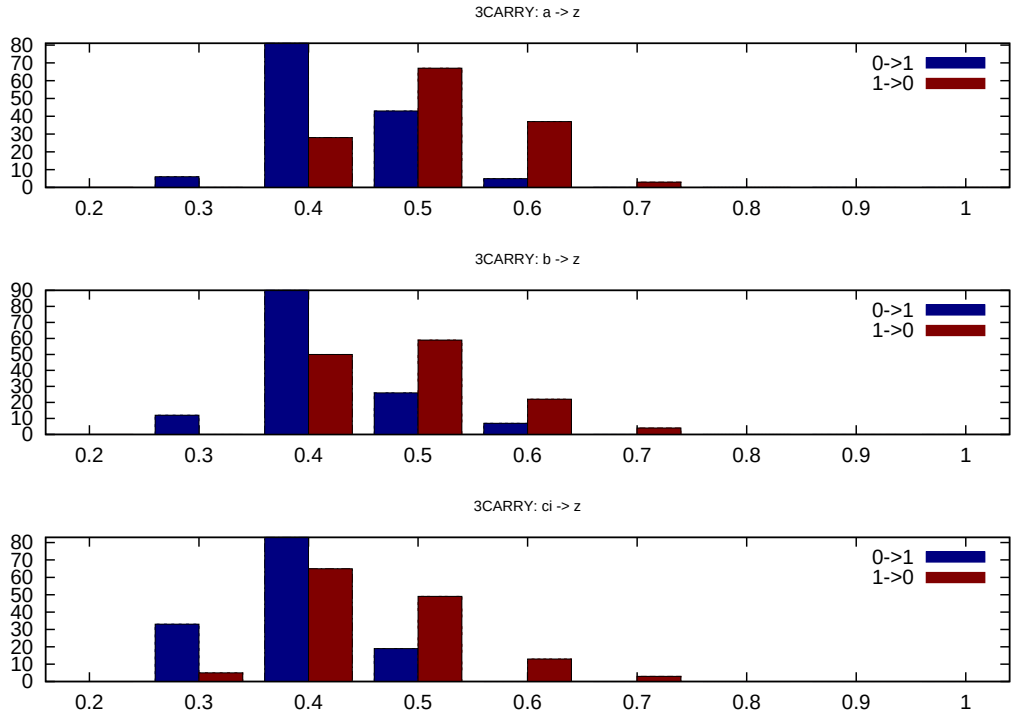

Figure A.8: Timing through full-adder, from input to carry output

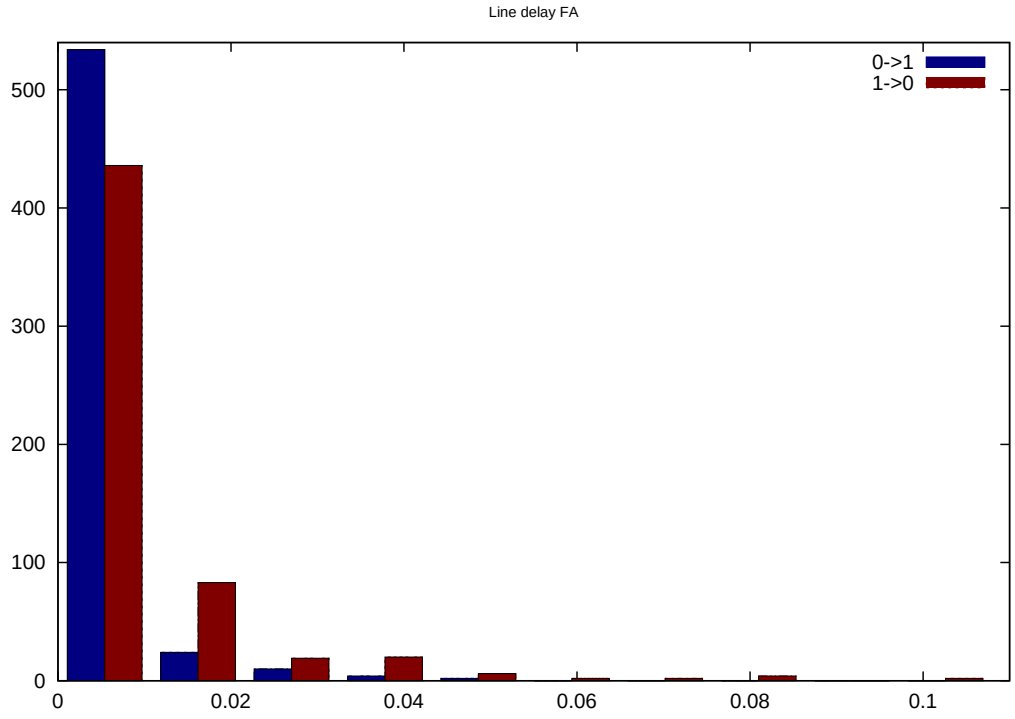

Figure A.9: Line timing from full-adder to next element

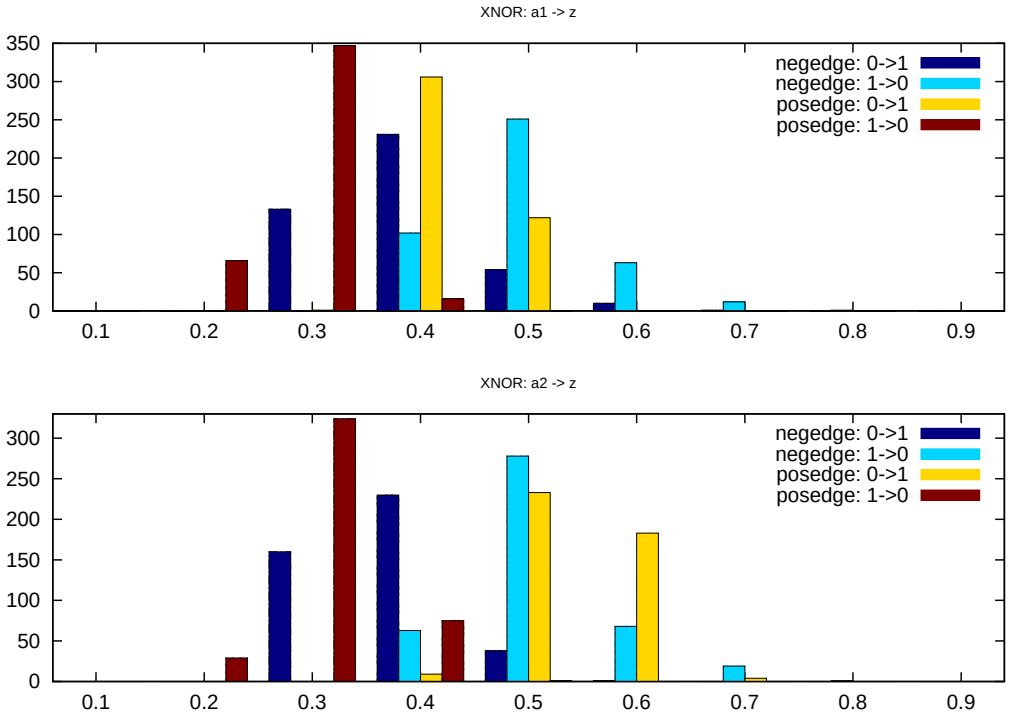

Figure A.10: Timing through half-adder, from input to sum output

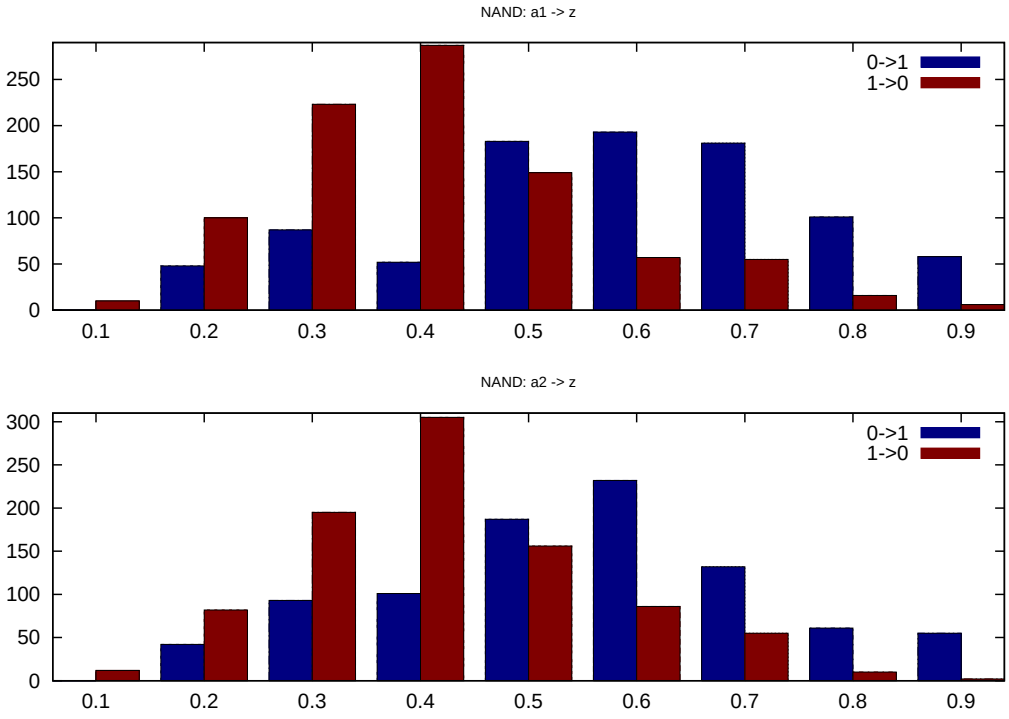

Figure A.11: Timing through half-adder, from input to sum output

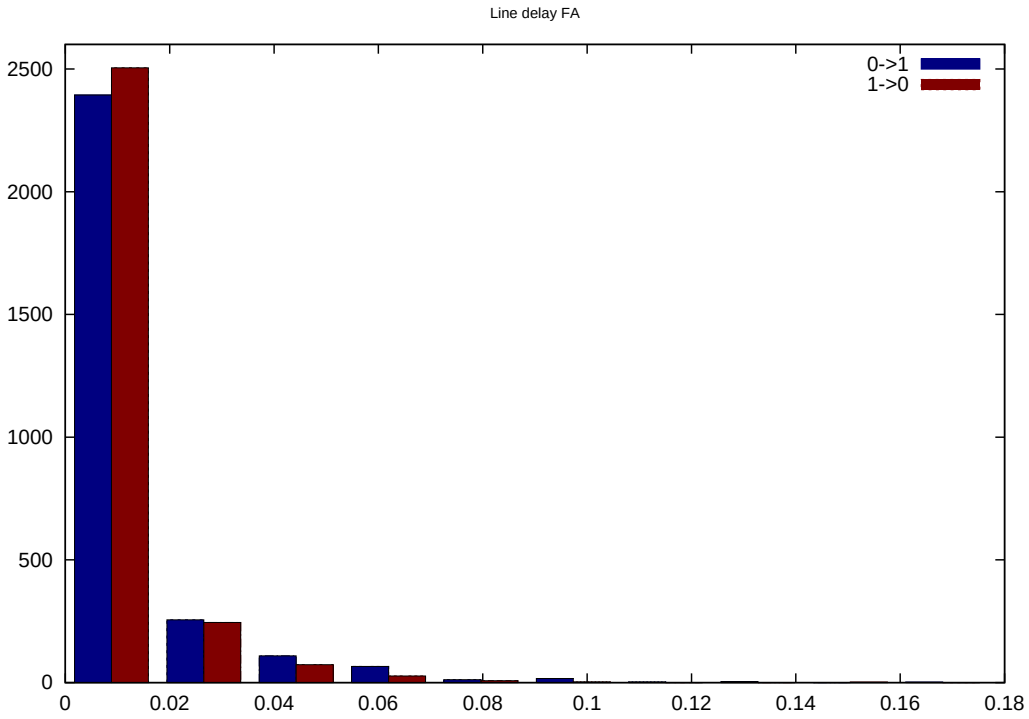

Figure A.12: Line timing from half-adder to next element

A.2.2 Optimized for maximum power, max PVT

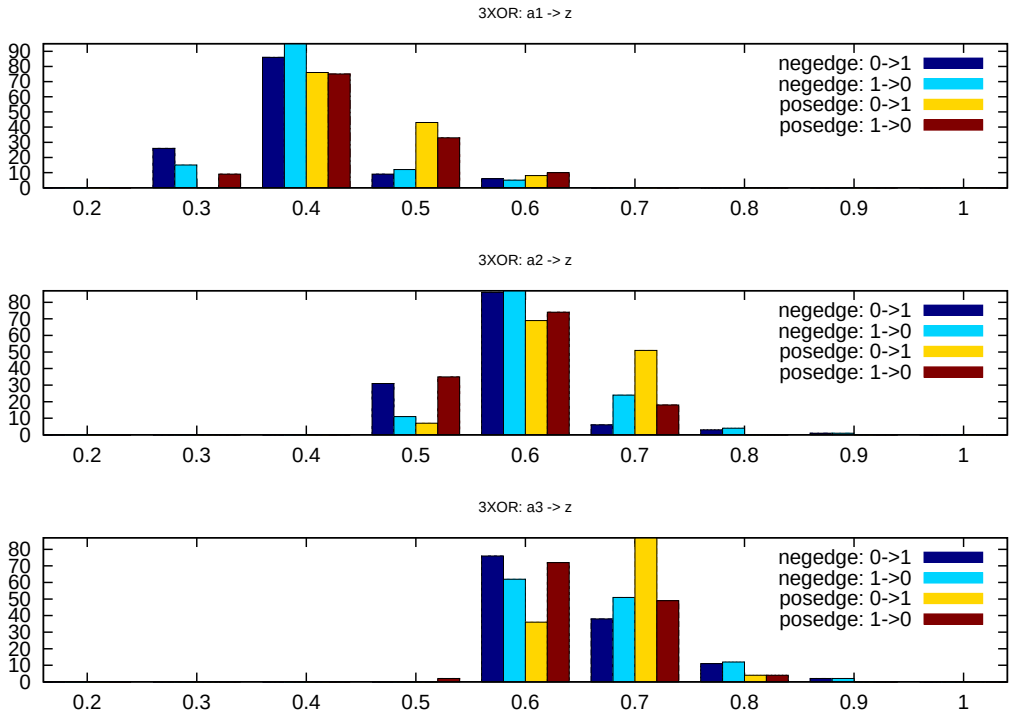

Figure A.13: Timing through full-adder, from input to sum output

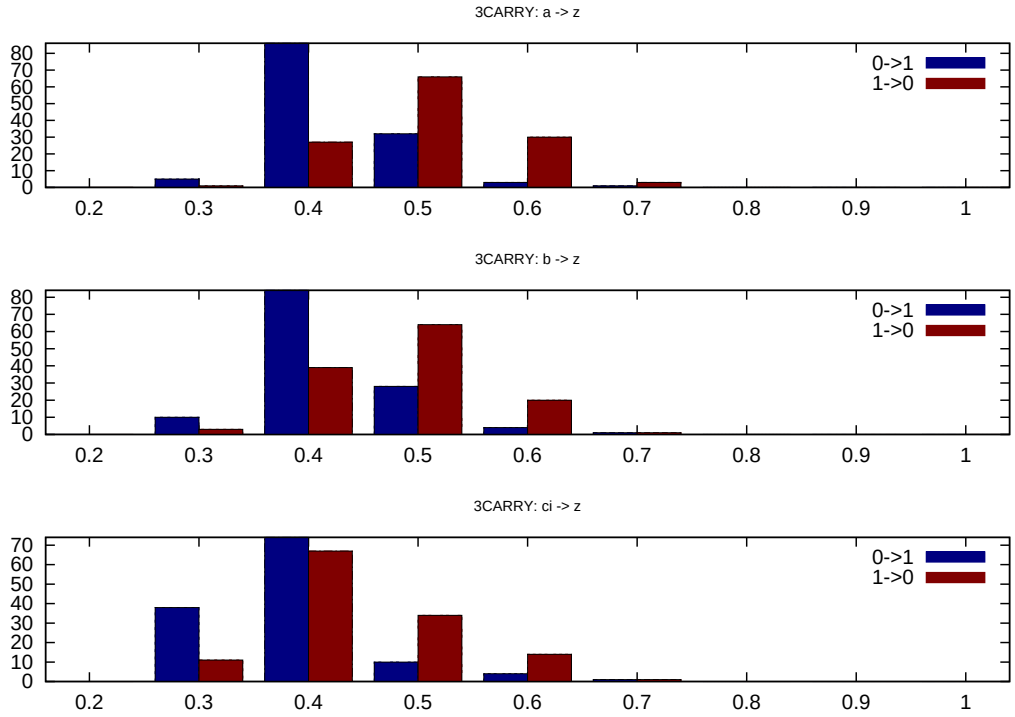

Figure A.14: Timing through full-adder, from input to carry output

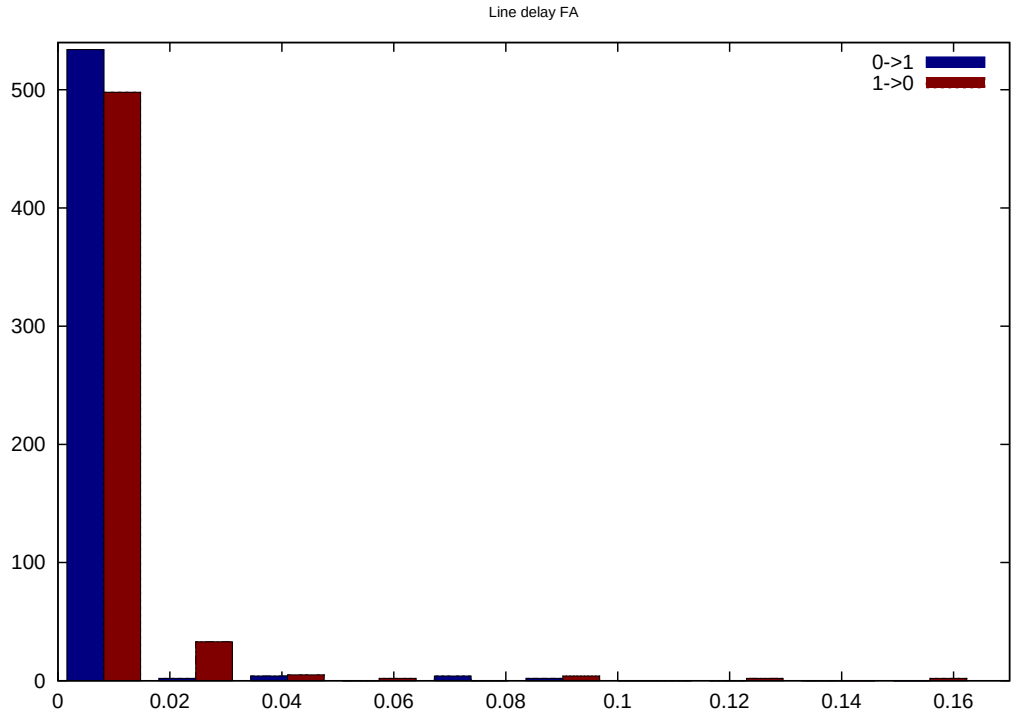

Figure A.15: Line timing from full-adder to next element

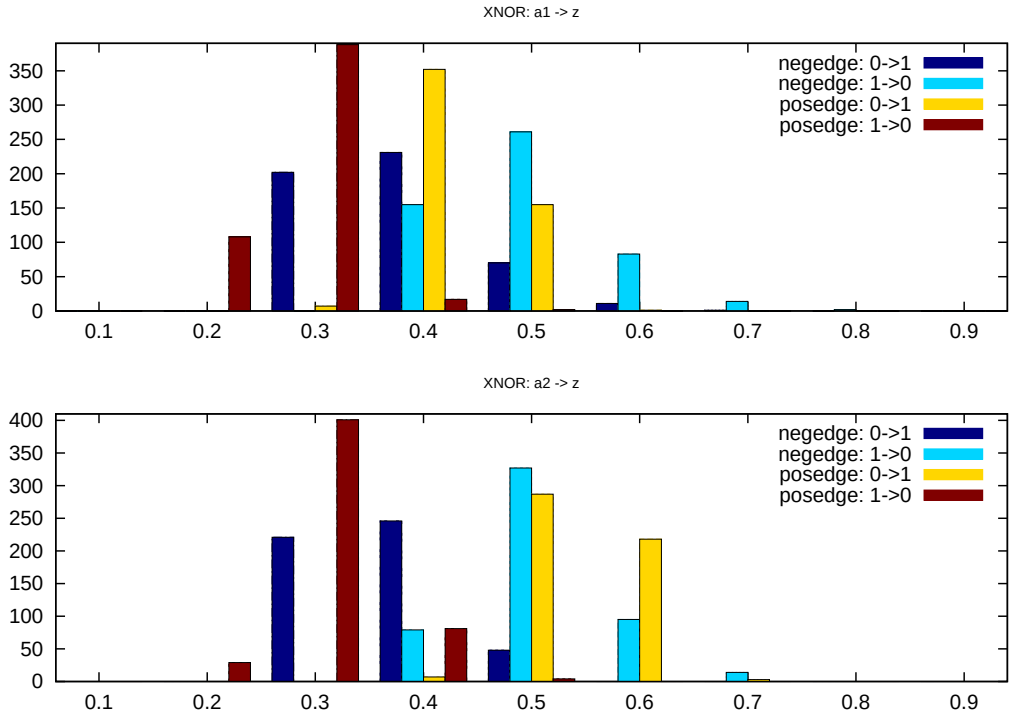

Figure A.16: Timing through half-adder, from input to sum output

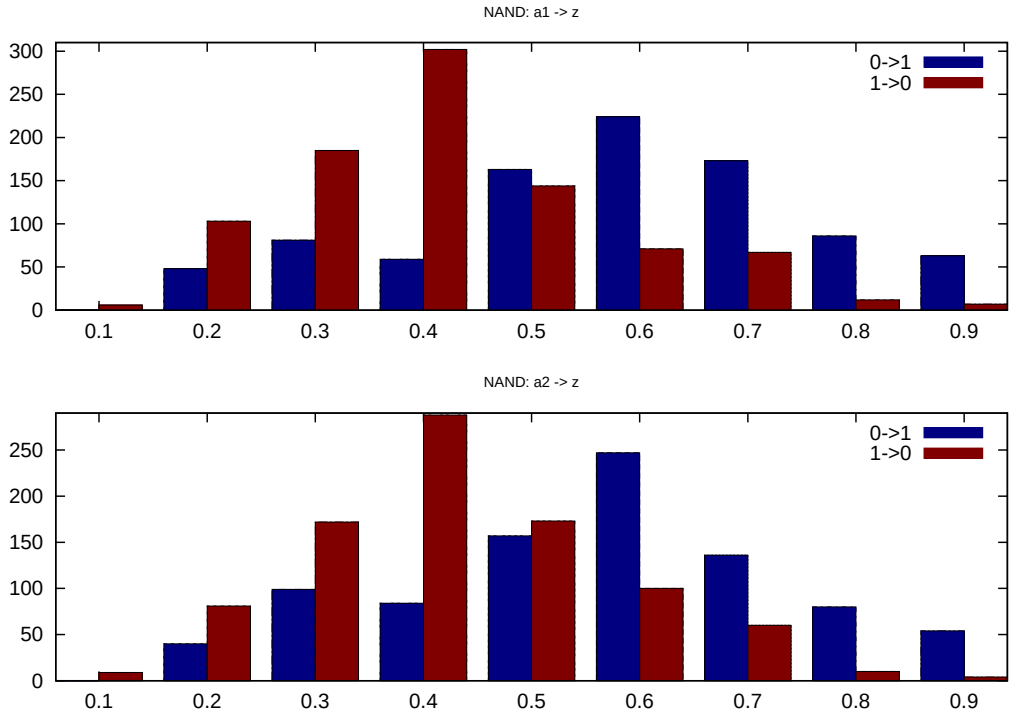

Figure A.17: Timing through half-adder, from input to sum output

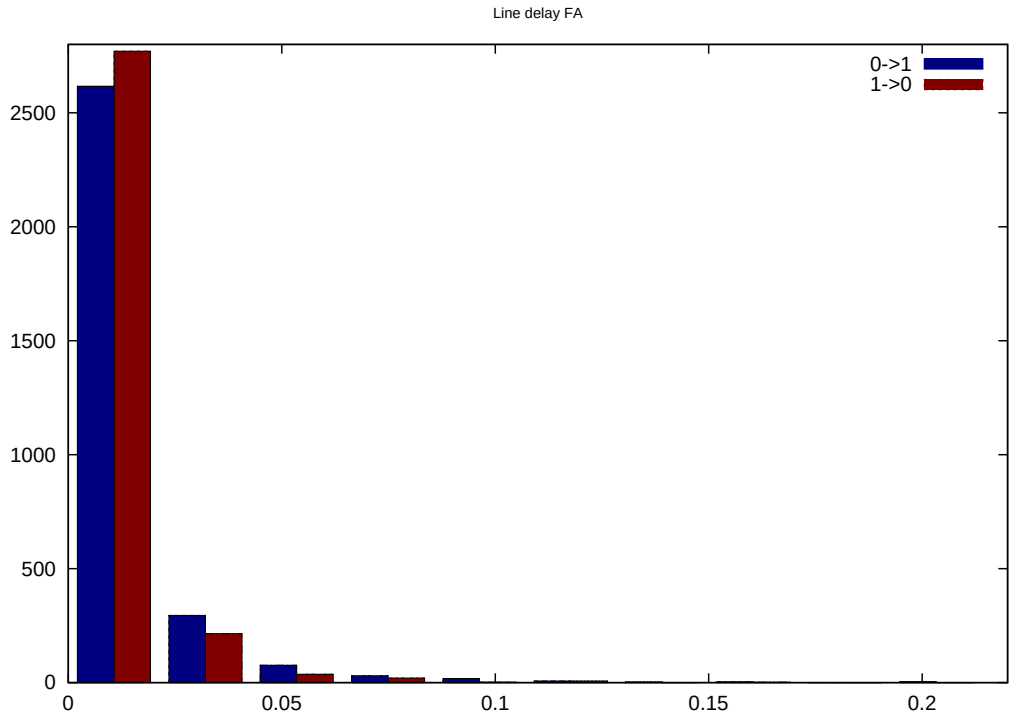

Figure A.18: Line timing from half-adder to next element

A.3 Estimator time usage

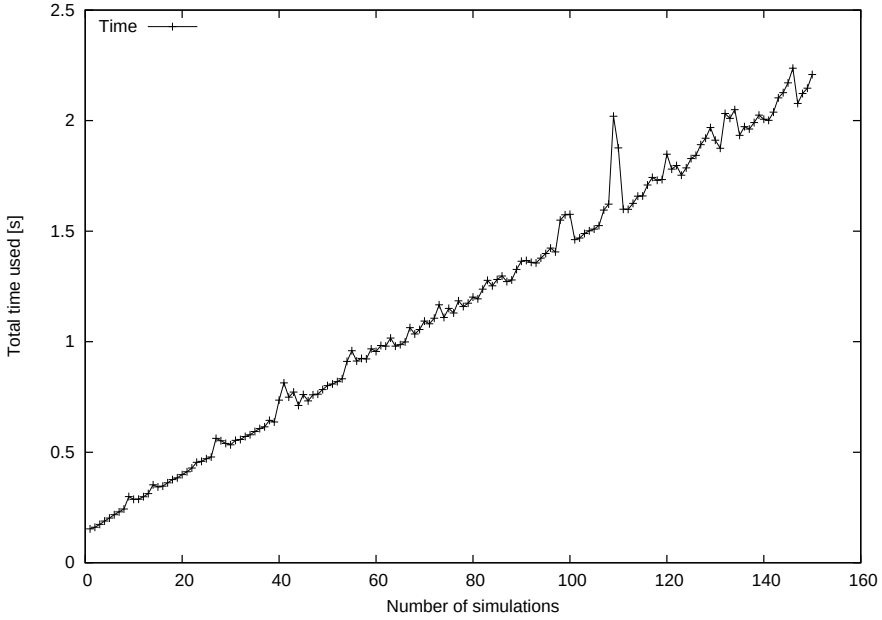

Figure A.19: Time usage of the estimator, 12x12 multiplier

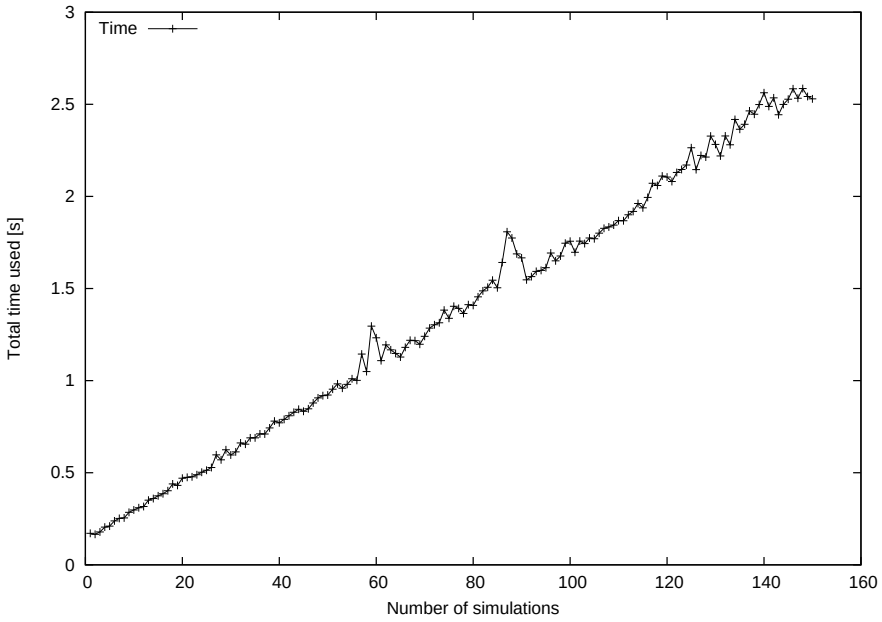

Figure A.20: Time usage of the estimator, 16x16 multiplier

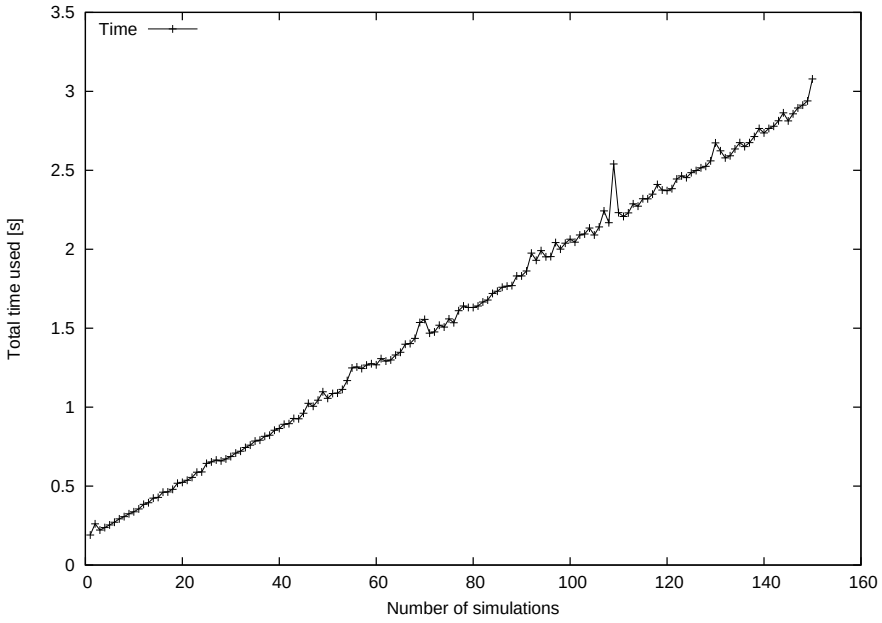

Figure A.21: Time usage of the estimator, 24x24 multiplier

A.4 Estimator accuracy of sample size

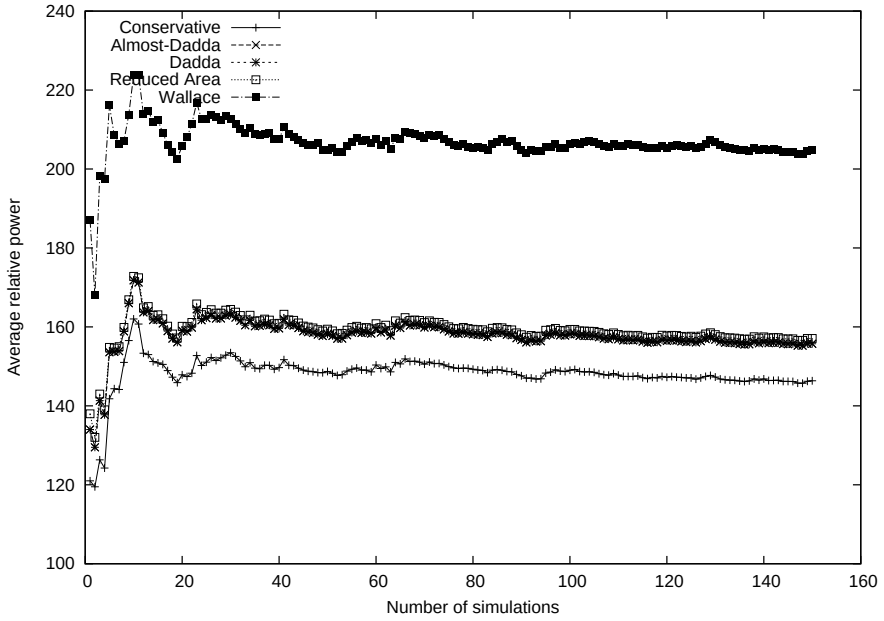

Figure A.22: Power estimation accuracy of simulations size, 12x12 multiplier

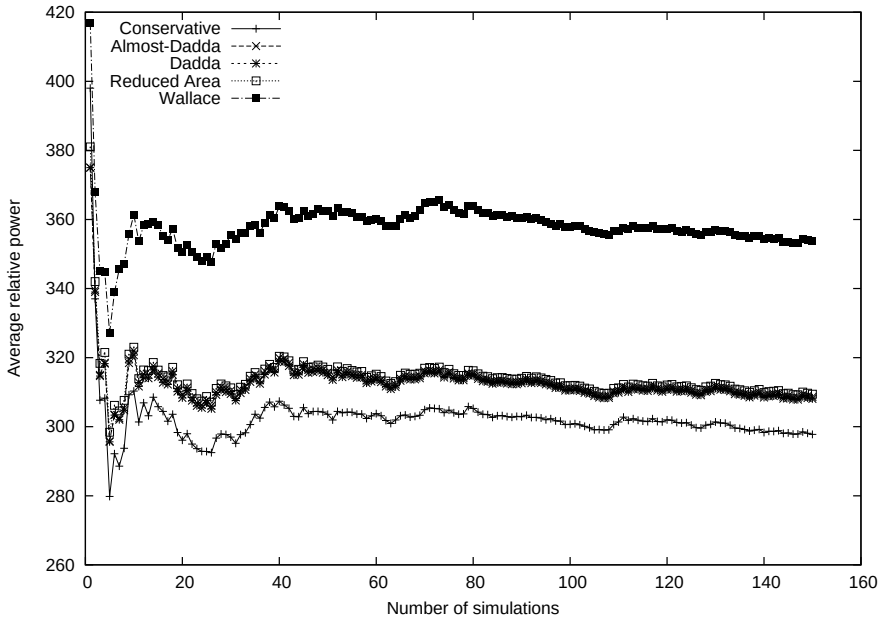

Figure A.23: Power estimation accuracy of simulations size, 16x16 multiplier

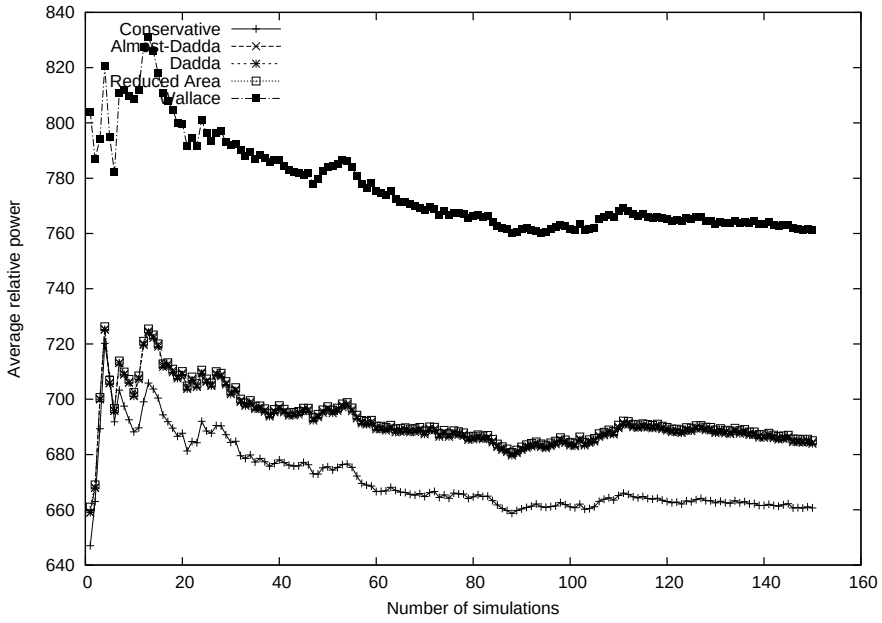

Figure A.24: Power estimation accuracy of simulations size, 24x24 multiplier

A.5 Power usage after optimization

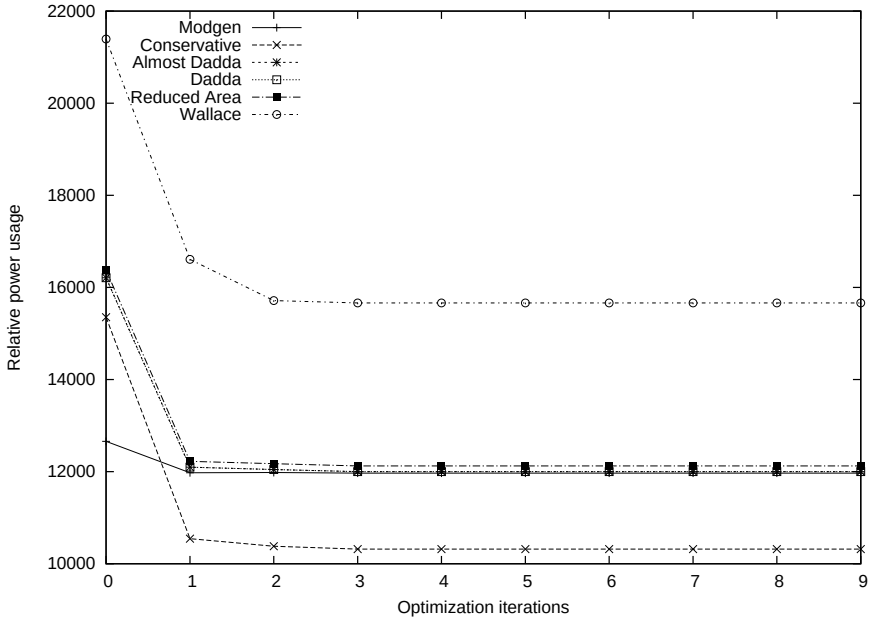

Figure A.25: Power usage after each optimization step, 12x12 multiplier

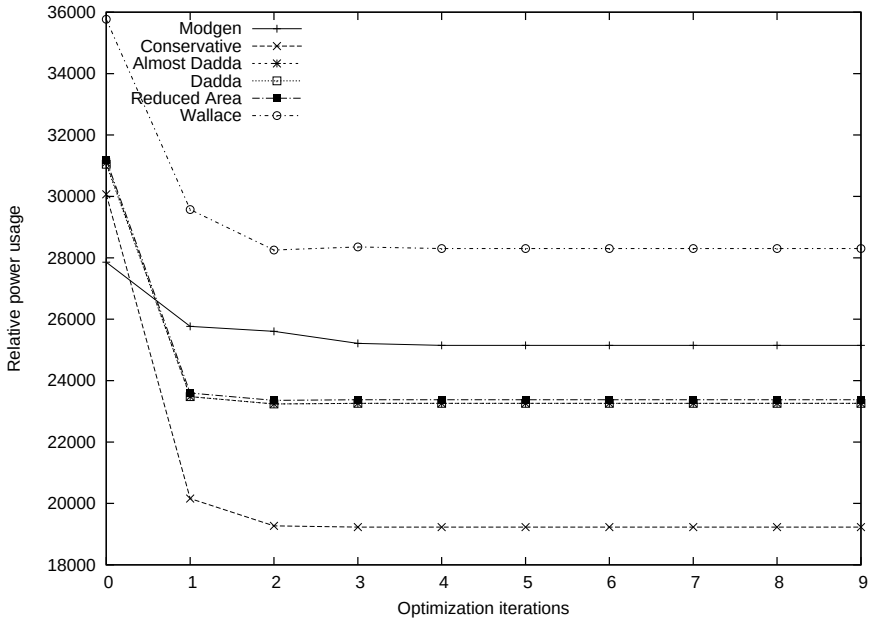

Figure A.26: Power usage after each optimization step, 16x16 multiplier

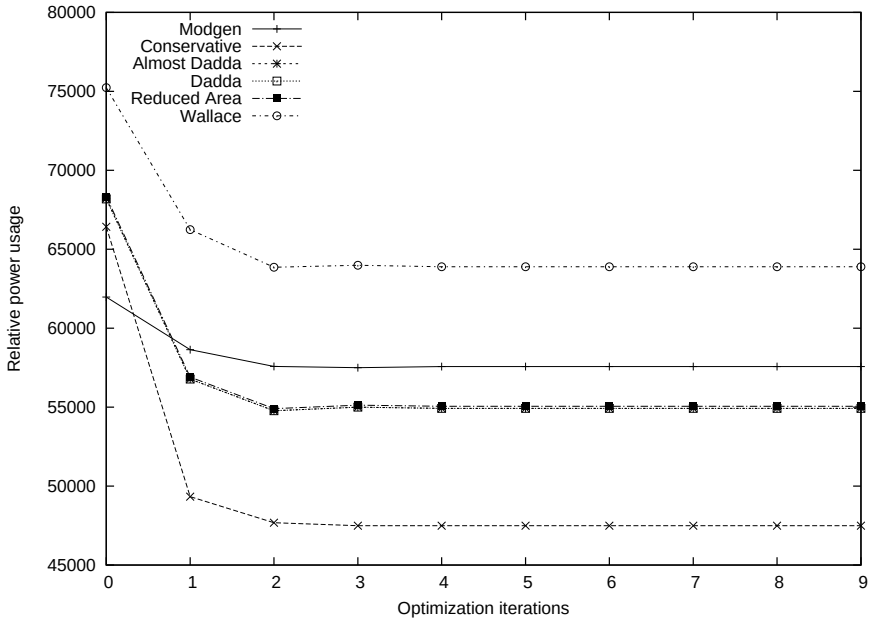

Figure A.27: Power usage after each optimization step, 24x24 multiplier

A.6 Multipler adder usage

Table A.1: Number of adders, depth and size of VMA for a 8x8 multiplier

Type of tree	No FA	No HA	Depth	Output lines	VMA size
Conservative	48	2	4	26	14
Almost Dadda	47	7	4	27	14
Dadda	43	14	5	31	14
Reduced Area	49	6	5	25	10
Wallace	47	23	5	27	10
Old algorithm	45	6	4	29	13

Table A.2: Number of adders, depth and size of VMA for a 16x16 multiplier

Type of tree	No FA	No HA	Depth	Output lines	VMA size
Conservative	218	8	6	56	30
Almost Dadda	218	23	6	56	30
Dadda	212	14	6	62	29
Reduced Area	218	14	6	56	25
Wallace	215	62	6	59	26
Old algorithm	212	30	6	61	29

Table A.3: Number of adders, depth and size of VMA for a 32x32 multiplier

Type of tree	No FA	No HA	Depth	Output lines	VMA size
Conservative	940	22	8	118	62
Almost Dadda	945	64	8	113	62
Dadda	932	30	8	126	61
Reduced Area	940	30	8	118	55
Wallace	938	163	8	120	55
Old algorithm	932	56	8	125	61

Appendix B

Perl-code for reading SDF-files

B.1 SDF-reader library

Listing B.1: SDF-reader library

```
#!/usr/bin/perl

use strict;
package sdf;

# Internal functions
sub split_delay
{
  my $temp = $_[0];
  $temp =~ s/^\(//;
  $temp =~ s/\)$//;
  my @retval = split(":", $temp);
  return @retval;
}

sub get_delay
{
  my @temp = @_;
  my %retval;
  @{$retval{'01'}} = split_delay(pop(@temp));
  if ($#temp >= 0)
  {
    @{$retval{'10'}} = split_delay(pop(@temp));
  }
  else
  {
    @{$retval{'10'}} = @{$retval{'01'}};
  }

  return %retval;
}

# Variables for LLSClib-parsing
my $pin;
our %celldefinition;

# Variables for paring the SDF-file
our %config;
our %tree;
our %cellprofile;
our %conn;
our %connrev;

my $cell, my $instance;

my @lastcommand;
```

```
47 my $nesting = 0;
48
49 sub split_port
50 {
51   my $temp = $_[0];
52   my @temp = split(/$config{'divider'}/, $temp);
53   my $port = pop(@temp);
54   my $inst = join($config{'divider'}, @temp);
55   return ($port, $inst);
56 }
57
58
59
60
61 open(FILE, "../LLSClib_functions.txt") || die("Can␣not␣open␣LLSClib");
62
63 while(<FILE>)
64 {
65   my $command, my $parameter;
66
67   chomp();
68   if (/^\s*(\w+)\s*\("(\w+)"\)\s*{\s*$/)
69 ␣␣{
70 ␣␣␣␣$command␣=␣$1;
71 ␣␣␣␣$parameter␣=␣$2;
72 ␣␣␣␣if␣($command␣eq␣"cell")
73 ␣␣␣␣{
74 ␣␣␣␣␣␣$cell␣=␣$parameter;
75 ␣␣␣␣}
76 ␣␣␣␣elsif␣($command␣eq␣"pin")
77 ␣␣␣␣{
78 ␣␣␣␣␣␣$pin␣=␣$parameter;
79 ␣␣␣␣}
80
81 ␣␣}
82 ␣␣elsif␣(/^\s*(\w+)\s*\:\s(.+)$/)
83 ␣␣{
84 ␣␣␣␣$command␣=␣$1;
85 ␣␣␣␣$parameter␣=␣$2;
86 ␣␣␣␣if␣($command␣eq␣'function')
87 ␣␣␣␣{
88 ␣␣␣␣␣␣$celldefinition{$cell}{$pin}{'oldfunc'}␣=␣$parameter;
89 ␣␣␣␣␣␣$parameter␣=~␣s/^\"//;
90 ␣␣␣␣␣␣$parameter␣=~␣s/\"$//;
91 ␣␣␣␣␣␣$parameter␣=~␣s/(\w+)/\$inputs\{\'$1\'\}/g;
92 ␣␣␣␣␣␣$parameter␣=~␣s/\*/␣\&␣/g;
93 ␣␣␣␣␣␣$parameter␣=~␣s/\+/␣\|␣/g;
94 ␣␣␣␣␣␣$parameter␣=~␣s/\!/\~/g
95 ␣␣␣␣}
96 ␣␣␣␣$celldefinition{$cell}{$pin}{$command}␣=␣$parameter;
97 ␣␣}
98 ␣␣elsif␣(/test_cell/)
99 ␣␣{
100 ␣␣␣␣$cell␣=␣"test_cell_".$cell;
101 ␣␣}
102 ␣␣else
103 ␣␣{
104 ␣␣␣␣print␣STDERR␣"Error paring cell-definitions: ".$_."\n";
105 ␣␣}
106
107 }
108
109 close(FILE);
110
111
112 while(<STDIN>)
113 {
114 ␣␣#␣Mathes␣each␣line
115 ␣␣#␣$1␣Command
116 ␣␣#␣$2␣Parameters
```

```
117 ␣␣#␣$3␣End␣of␣node/not␣end␣of␣node
118 ␣␣if␣(/^\s*\(([^\s]*)\s*((?:\s*[^\(\)\s]+\s*|\s*\([^\)]+\)\s*)*)(\)?)$/)
119 ␣␣{
120 ␣␣␣␣my␣$command␣=␣$1;
121 ␣␣␣␣my␣$parameters_string␣=␣$2;
122 ␣␣␣␣my␣$endofnode␣=␣$3;
123 ␣␣␣␣my␣@parameters;
124 ␣␣␣␣while␣($parameters_string␣=~␣m/\s*([^\(\)\s]+|\([^\)]+\))\s*/g)
125 ␣␣␣␣{
126 ␣␣␣␣␣␣push(@parameters,␣$1);
127 ␣␣␣␣}
128
129 ␣␣␣␣#␣Remember␣where␣in␣the␣tree␣we␣are.␣Push␣commands␣on␣the␣array
130 ␣␣␣␣#␣to␣create␣an␣array␣with␣all␣the␣above␣nodes
131 ␣␣␣␣if␣($endofnode␣ne␣")")
132 ␣␣␣␣{
133 ␣␣␣␣␣␣$nesting++;
134 ␣␣␣␣␣␣push(@lastcommand,␣$command);
135 ␣␣␣␣}
136
137 ␣␣␣␣if␣($command␣eq␣"DIVIDER")
138 ␣␣␣␣{
139 ␣␣␣␣␣␣$config{'divider'}␣=␣$parameters_string;
140 ␣␣␣␣}
141 ␣␣␣␣elsif␣($command␣eq␣"CELLTYPE")
142 ␣␣␣␣{
143 ␣␣␣␣␣␣$cell␣=␣$parameters[0];
144 ␣␣␣␣␣␣$cell␣=~␣s/^\"//;
145 ␣␣␣␣␣␣$cell␣=~␣s/\"$//;
146 ␣␣␣␣}
147 ␣␣␣␣elsif␣($command␣eq␣"INSTANCE")
148 ␣␣␣␣{
149 ␣␣␣␣␣␣$instance␣=␣$parameters[0];
150 ␣␣␣␣␣␣if␣($instance␣eq␣"")
151 ␣␣␣␣␣␣{
152 ␣␣␣␣␣␣␣␣$instance␣=␣"TOP";
153 ␣␣␣␣␣␣}
154 ␣␣␣␣}
155 ␣␣␣␣elsif␣($command␣eq␣"INTERCONNECT"␣&&␣$lastcommand[$#lastcommand]␣eq␣"ABSOLUTE")
156 ␣␣␣␣{
157 ␣␣␣␣␣␣my␣$inst␣=␣shift(@parameters);
158 ␣␣␣␣␣␣my␣$longin␣=␣$inst;
159 ␣␣␣␣␣␣my␣$in␣=␣"default";
160 ␣␣␣␣␣␣my␣$out␣=␣shift(@parameters);
161
162 ␣␣␣␣␣␣my␣$regex␣=␣$config{'divider'}.'([^'.$config{'divider'}.']+)$';
163 ␣␣␣␣␣␣if␣($inst␣=~␣s/$regex//)
164 ␣␣␣␣␣␣{
165 ␣␣␣␣␣␣␣␣$in␣=␣$1;
166 ␣␣␣␣␣␣}
167 ␣␣␣␣␣␣else
168 ␣␣␣␣␣␣{
169 ␣␣␣␣␣␣␣␣$in␣=␣$inst;
170 ␣␣␣␣␣␣␣␣$inst␣=␣"TOP";
171 ␣␣␣␣␣␣}
172 ␣␣␣␣␣␣%{$tree{$inst}{'io'}{$in}{$out}}␣=␣get_delay(@parameters);
173 ␣␣␣␣␣␣if␣(!␣grep($longin,␣@{$tree{$inst}{'output'}}))
174 ␣␣␣␣␣␣{
175 ␣␣␣␣␣␣␣␣push(@{$tree{$inst}{'output'}},␣$longin);
176 ␣␣␣␣␣␣}
177 ␣␣␣␣␣␣push(@{$conn{$longin}},␣$out);
178 ␣␣␣␣␣␣$connrev{$out}␣=␣$longin;
179 ␣␣␣␣}
180 ␣␣␣␣elsif␣($command␣eq␣"IOPATH"␣&&␣$lastcommand[$#lastcommand]␣eq␣"ABSOLUTE")
181 ␣␣␣␣{
182 ␣␣␣␣␣␣my␣$in␣=␣shift(@parameters);
183 ␣␣␣␣␣␣my␣$out␣=␣shift(@parameters);
184 ␣␣␣␣␣␣%{$tree{$instance}{'io'}{$in}{$out}}␣=␣get_delay(@parameters);
185 ␣␣␣␣␣␣%{$tree{$instance}{'iopath'}{$in}{$out}}␣=␣get_delay(@parameters);
186 ␣␣␣␣␣␣$in␣=~␣s/^\(\w+␣(\w+)\)$/$1/;
```

```
187 ␣␣␣␣␣␣$tree{$instance}{'route'}{$in}␣=␣$out;
188 ␣␣␣␣␣␣$tree{$instance}{'input'}{$instance.$config{'divider'}.$in}␣=␣1;
189 ␣␣␣␣␣␣#$connrev{$instance.$config{'divider'}.$out}␣=␣$instance.$config{'divider'}.$in;
190 ␣␣␣␣␣␣#push(@{$conn{$instance.$config{'divider'}.$in}},␣
        $instance.$config{'divider'}.$out);
191 ␣␣␣␣␣␣#push(@{$connrev{$instance.$config{'divider'}.$out}},␣
        $instance.$config{'divider'}.$in);
192 ␣␣␣␣}
193 ␣␣}
194 ␣␣elsif␣(/^\s*\)\s*$/)
195 ␣␣{
196 ␣␣␣␣#␣Remove␣a␣node␣from␣the␣array,␣since␣the␣node␣is␣closed
197 ␣␣␣␣$nesting--;
198 ␣␣␣␣my␣$command␣=␣pop(@lastcommand);
199 ␣␣␣␣if␣($command␣eq␣"CELL")
200 ␣␣␣␣{
201 ␣␣␣␣␣␣if␣($cell␣eq␣"")␣{␣$cell␣=␣"UNKNOWN";␣}
202 ␣␣␣␣␣␣push(@{$cellprofile{$cell}},␣$instance);
203 ␣␣␣␣␣␣$tree{$instance}{'celltype'}␣=␣$cell;
204 ␣␣␣␣␣␣$cell␣=␣"";
205 ␣␣␣␣}
206 ␣␣}
207 ␣␣else
208 ␣␣{
209 ␣␣␣␣print␣STDERR␣"Error reading line\n";
210 ␣␣}
211 }
212
213 1;
```

Listing B.1: SDF-reader library

B.2 Gate counter

Count the number of each element, so one can guess what kind of elements is used in the multiplicator

Listing B.2: Code to count each element in multiplier tree

```
1  #!/usr/bin/perl
2
3  my $inc = __FILE__;
4  $inc =~ s/\/[^\/]*$//;
5  push(@INC, $inc);
6  require 'readsdf.pl';
7
8  foreach my $k (sort {$#{$sdf::cellprofile{$a}} <=> $#{$sdf::cellprofile{$b}}} (keys
       %sdf::cellprofile))
9  {
10   # Print info about cell
11   print "\n";
12   print "-------------------------\n";
13   print $k." occurs ".($#{$sdf::cellprofile{$k}}+1)."\n";
14   # Gather input and output info about cell
15   my @inputs, my @outputs, my @outputfunc, my %inputs;
16   foreach my $i (keys %{$sdf::celldefinition{$k}})
17   {
18     if ($sdf::celldefinition{$k}{$i}{'direction'} eq "output")
19     {
20       push(@outputs, $i);
21       push(@outputfunc, $sdf::celldefinition{$k}{$i}{'function'});
22       print $i." has function ".$sdf::celldefinition{$k}{$i}{'oldfunc'}."\n";
23     }
24     else
25     {
26       push(@inputs, $i);
27       $inputs{$i} = 0;
```

```
28      }
29    }
30    # Gather statistics
31    my %stat;
32    foreach my $instance (@{$sdf::cellprofile{$k}})
33    {
34      foreach my $i (keys %{$sdf::tree{$instance}{'iopath'}})
35      {
36        if (1)
37        {
38          foreach my $j (keys %{$sdf::tree{$instance}{'iopath'}{$i}})
39          {
40            $stat{$i}{$j}{'count'}++;
41            $stat{$i}{$j}{'01'} += $sdf::tree{$instance}{'iopath'}{$i}{$j}{'01'}[1];
42            $stat{$i}{$j}{'10'} += $sdf::tree{$instance}{'iopath'}{$i}{$j}{'10'}[1];
43          }
44        }
45      }
46    }
47    # Print stats
48    foreach my $i (sort keys %stat)
49    {
50      foreach my $j (keys %{$stat{$i}})
51      {
52        print $i."=>".$j.": ";
53        print "  0->1: ".($stat{$i}{$j}{'01'}/$stat{$i}{$j}{'count'});
54        print "  1->0: ".($stat{$i}{$j}{'10'}/$stat{$i}{$j}{'count'});
55        print " (".$stat{$i}{$j}{'count'}.")\n";
56      }
57    }
58
59    # Make truth-table
60    print "| ".join(" | ", @inputs)." | ".join(" | ", @outputs)." |\n";
61
62    my $size;
63    if ($#inputs < 0)
64    {
65      $size = 0;
66    }
67    else
68    {
69      $size = (1 << ($#inputs+1));
70    }
71    for (my $i = 0; $i < $size; $i++)
72    {
73      for (my $j = 0; $j <= $#inputs; $j++)
74      {
75        $inputs{$inputs[$j]} = ($i >> $j) & 1;
76        print "| ".$inputs{$inputs[$j]}." ";
77      }
78      for (my $j = 0; $j <= $#outputs; $j++)
79      {
80        # Eval is dangerous, but saves a lot of time
81        my $res = eval($outputfunc[$j]) & 1;
82        print "| ".$res." ";
83      }
84      print "\n";
85    }
86    print "-------------------------\n";
87  }
```

Listing B.2: Code to count each element in multiplier tree

B.3 Gate printer

Listing B.3: Prints of part of the multiplier tree

```
#!/usr/bin/perl

use strict;

my $inc = __FILE__;
$inc =~ s/\/[^\/]*$//;
push(@INC, $inc);
require 'readsdf.pl';

foreach my $i (keys %sdf::tree)
{
  if ($sdf::tree{$i}{'celltype'} eq 'xn02d1ll')
  {
    my %printlist;
    print
        "-------------------------------------------------------------------------------\n";
    print " FOUND CELL!!!\n";
    print
        "-------------------------------------------------------------------------------\n";
    foreach my $base_input (keys %{$sdf::tree{$i}{'input'}})
    {
      print "INPUT: ".$sdf::connrev{$base_input}." 
          (".($#{$sdf::conn{$sdf::connrev{$base_input}}}+1)." outputs)\n";
      my @temp = split(/$sdf::config{'divider'}/, $sdf::connrev{$base_input});
      my $port = pop(@temp);
      my $inst = join($sdf::config{'divider'}, @temp);

      $printlist{$inst} = 1;

      foreach my $nextport (@{$sdf::conn{$sdf::connrev{$base_input}}})
      {
        my @temp = split(/$sdf::config{'divider'}/, $nextport);
        my $port = pop(@temp);
        my $inst = join($sdf::config{'divider'}, @temp);
        $printlist{$inst} = 2;
      }
    }

    foreach my $inst (sort keys %printlist)
    {
      print "INSTANCE: ".$inst."  CELLTYPE: ".$sdf::tree{$inst}{'celltype'};
      print " COLUMN: ".$printlist{$inst}."\n";
      foreach my $input (keys %{$sdf::tree{$inst}{'input'}})
      {
        print "IO: ".$sdf::connrev{$input}." -> ".$input."\n";
      }
    }

    foreach my $inst (sort keys %printlist)
    {
      foreach my $output (@{$sdf::tree{$inst}{'output'}})
      {
        foreach my $input (@{$sdf::conn{$output}})
        {
          my @temp = split(/$sdf::config{'divider'}/, $input);
          my $port = pop(@temp);
          my $inst = join($sdf::config{'divider'}, @temp);
          print "INSTANCE: ".$inst."  CELLTYPE: ".$sdf::tree{$inst}{'celltype'};
          print " COLUMN: 3\n";
          foreach my $input (keys %{$sdf::tree{$inst}{'input'}})
          {
            print "IO: ".$sdf::connrev{$input}." -> ".$input."\n";
          }
        }
      }
    }
  }
}
```

Listing B.3: Prints of part of the multiplier tree

B.4 SDF to datafile generator

Used to make datafiles for generating the histograms of the SDF-files

Listing B.4: Generates datafiles for delay histograms

```
#!/usr/bin/perl

use strict;

my $inc = __FILE__;
$inc =~ s/\/[^\/]*$//;
push(@INC, $inc);
require 'readsdf.pl';

my %statlist;

foreach my $i (keys %sdf::tree)
{
  # Find elements used in FA1 (Triple-XOR and a 3-in-carry element)
  if ($sdf::tree{$i}{'celltype'} eq 'xr03d1ll')
  {
    my $sum = $i;
    my $carry;
    my $count = 0;
    foreach my $pp (@{$sdf::conn{$sdf::connrev{$i.$sdf::config{'divider'}.'a1'}}})
    {
      (my $port, my $inst) = &sdf::split_port($pp);
      if ($sdf::tree{$inst}{'celltype'} eq 'cg01d0ll')
      {
        $carry = $inst;
        $count++;
      }
    }
    foreach my $pp (@{$sdf::conn{$sdf::connrev{$i.$sdf::config{'divider'}.'a2'}}})
    {
      (my $port, my $inst) = &sdf::split_port($pp);
      if ($carry eq $inst) { $count++; }
    }
    foreach my $pp (@{$sdf::conn{$sdf::connrev{$i.$sdf::config{'divider'}.'a3'}}})
    {
      (my $port, my $inst) = &sdf::split_port($pp);
      if ($carry eq $inst) { $count++; }
    }
    if ($count == 3)
    {
      push(@{$statlist{'3sum'}}, $sum);
      push(@{$statlist{'3carry'}}, $carry);
    }
  }
  # Find elements used in HA (just take all nand and xnor-ports)
  elsif ($sdf::tree{$i}{'celltype'} eq 'nd02d0ll')
  {
    push(@{$statlist{'ha_nand'}}, $i);
  }
  elsif ($sdf::tree{$i}{'celltype'} eq 'xn02d1ll')
  {
    push(@{$statlist{'ha_xnor'}}, $i);
  }
}

foreach my $type (keys %statlist)
{
```

```
58  open(GATE, ">".$type."_gate.dat");
59  open(LINE, ">".$type."_line.dat");
60
61  my @inputs;
62  my @outputs;
63
64  foreach my $input (keys %{$sdf::tree{$statlist{$type}[0]}{'iopath'}})
65  {
66    push(@inputs, $input);
67    foreach my $output (keys %{$sdf::tree{$statlist{$type}[0]}{'iopath'}{$input}})
68    {
69      if (!(grep($output, @outputs)))
70      {
71        push(@outputs, $output);
72      }
73    }
74  }
75  @inputs = sort @inputs;
76  @outputs = sort @outputs;
77  my $line_rows = 0;
78
79  # Calculate rows for line-delay
80  foreach my $entry (@{$statlist{$type}})
81  {
82    foreach my $output (@outputs)
83    {
84      foreach my $input (keys %{$sdf::tree{$entry}{'io'}{$output}})
85      {
86        $line_rows++;
87      }
88    }
89  }
90
91  print GATE "# ".join("\t\t", @inputs)."\n";
92  print GATE "# name: gate\n";
93  print GATE "# type: matrix\n";
94  print GATE "# rows: ".($#{$statlist{$type}}+1)."\n";
95  print GATE "# columns: ".(($#inputs+1)*2)."\n";
96
97  print LINE "# Comment - ". join("\t", @outputs)."\n";
98  print LINE "# name: linedelay\n";
99  print LINE "# type: matrix\n";
100 print LINE "# rows: ".$line_rows."\n";
101 print LINE "# columns: ".(($#outputs+1)*2)."\n";
102
103 # Generate avarage
104 my @gateavg;
105 my @lineavg;
106
107 foreach my $entry (@{$statlist{$type}})
108 {
109   my $incrementor = 0;
110   foreach my $input (@inputs)
111   {
112     print GATE " ".$sdf::tree{$entry}{'io'}{$input}{$outputs[0]}{'01'}[1];
113     print GATE " ".$sdf::tree{$entry}{'io'}{$input}{$outputs[0]}{'10'}[1];
114     $gateavg[$incrementor++] +=
          $sdf::tree{$entry}{'io'}{$input}{$outputs[0]}{'01'}[1];
115     $gateavg[$incrementor++] +=
          $sdf::tree{$entry}{'io'}{$input}{$outputs[0]}{'10'}[1];
116   }
117   print GATE "\n";
118
119   foreach my $output (@outputs)
120   {
121     my $longoutput = $entry.$sdf::config{'divider'}.$output;
122     foreach my $input (keys %{$sdf::tree{$entry}{'io'}{$output}})
123     {
124       $incrementor = 0;
125       print LINE " ".$sdf::tree{$entry}{'io'}{$output}{$input}{'01'}[1];
```

```
126             print LINE " ".$sdf::tree{$entry}{'io'}{$output}{$input}{'10'}[1];
127             $lineavg[$incrementor++] +=
                    $sdf::tree{$entry}{'io'}{$output}{$input}{'01'}[1];
128             $lineavg[$incrementor++] +=
                    $sdf::tree{$entry}{'io'}{$output}{$input}{'10'}[1];
129             print LINE "\n";
130           }
131         }
132       }
133
134       # AVG
135       print GATE "# NAMES: ".join(" ", @inputs)."\n";
136       print GATE "# AVG:  ";
137       foreach my $avg (@gateavg)
138       {
139         print GATE " ".($avg/($#{$statlist{$type}}+1));
140       }
141       print GATE "\n";
142
143       print LINE "# NAMES: ".join(" ", @outputs)."\n";
144       print LINE "# AVG:  ";
145       foreach my $avg (@lineavg)
146       {
147         print LINE " ".($avg/$line_rows);
148       }
149       print LINE "\n";
150
151
152       close(GATE);
153       close(LINE);
154   }
```

Listing B.4: Generates datafiles for delay histograms

Appendix C

C-code

C.1 Estimation

C.1.1 estimation.h

C-code C.1: Header file for estimator

```
1  #ifndef ESTIMATE_H
2  #define ESTIAMTE_H
3
4  #include "modgen.h"
5
6  #define PRINT_ELEMENT(x) ((x & FA_ELEMENT)?"FA":((x & HA_ELEMENT)?"HA":"NO"))
7
8  enum simgatetype_t {
9      SIMGATE_NULL, SIMGATE_AND, SIMGATE_XOR, SIMGATE_OR, SIMGATE_OUT, SIMGATE_IN,
           SIMGATE_PIPE, SIMGATE_NO
10 };
11
12 #define DELAY_MAX 510
13 #define NO_DELAY 0
14 #define LINE_DELAY 0
15
16 #define AND_DELAY 12  /* 2.4 delay - 12 */
17 #define OR_DELAY  12  /* 2.4 delay - 12 */
18 #define XOR_DELAY 21  /* 4.2 delay - 21 */
19
20 #define INIT_STACK_SIZE 1024
21
22 #define GATESIZE 5 /* Max number of gates in each block */
23
24 /* From modgen.c */
25 extern FA *ExternalInput;
26 extern FA *ExternalOutput;
27 extern FA *RoundBitInput;
28
29 struct STACKENTRY;
30 typedef struct STACKENTRY STACKENTRY;
31 struct SIMGATE;
32 typedef struct SIMGATE SIMGATE;
33
34 /* Datatypes and variables for the hashmap */
35 struct SIMBLOCK
36 {
37     struct FA* element;
38     struct SIMBLOCK* next;
39     struct SIMGATE* gates[GATESIZE];
40     struct SIMGATE* output[2]; /* 0 = SUM, 1 = CARRY */
41     struct SIMGATE* input[3]; /* 0 = InA, 1 = InB, 2 = InC */
42 };
43 typedef struct SIMBLOCK SIMBLOCK;
```

```
44
45 struct SIMGATE
46 {
47     struct SIMGATE* input[2];
48     struct SIMGATE* output[2];
49     enum simgatetype_t type;
50     uint8_t value;
51     struct SIMBLOCK* parent;
52     struct SIMGATE* next;
53     struct SIMGATE* first;
54     uint32_t delay;
55     uint32_t activity;
56     uint32_t timestamp;
57 };
58
59 struct SIMHASHMAP {
60     uint32_t mask;
61     uint32_t size;
62     uint32_t logsize;
63     struct SIMBLOCK **map;
64 };
65 typedef struct SIMHASHMAP SIMHASHMAP;
66
67 /* Datatypes and variables for the simulation stack */
68 struct STACKENTRY
69 {
70     uint8_t value;
71     struct SIMGATE* gate;
72     int16_t delay;
73 };
74
75 struct STACKPAGE
76 {
77     struct STACKENTRY* stack;
78     uint32_t size;
79     uint32_t current;
80     uint32_t last;
81 };
82 typedef struct STACKPAGE STACKPAGE;
83
84 struct FASTACK
85 {
86     struct STACKPAGE* page;
87     uint32_t size;
88     uint32_t current;
89 };
90 typedef struct FASTACK FASTACK;
91
92 struct SIMSTRUCTURES
93 {
94     struct SIMHASHMAP hashmap;
95     FASTACK *stack;
96     FASTACK *outstack;
97     ADDERTREE* tree;
98     uint32_t activity;
99     uint32_t maxdepths;
100    uint32_t outputs;
101    uint32_t count;
102 };
103 typedef struct SIMSTRUCTURES SIMSTRUCTURES;
104
105 /* Functions */
106 /* Structurefunctions */
107 SIMBLOCK* getFA(SIMHASHMAP* hashmap, FA* element);
108
109 /* Simulatorfunctions */
110 double RunTestSimulationOnStructures(struct SIMSTRUCTURES sim, uint32_t iterations,
        uint32_t preruns);
111 struct SIMSTRUCTURES InitEstimationStructures(ADDERTREE *WTree);
112 void deallocEstimationStructures(struct SIMSTRUCTURES sim);
```

```
113
114 #endif /* ESTIMATE_H */
```

C-code C.1: Header file for estimator

C.1.2 estimation.c

C-code C.2: Source file for estimator

```
1
2  #include "modgen.h"
3  #include <assert.h>
4  #include "estimate.h"
5
6  /* The outputgate. Put this as output, to indicate that this is the last element */
7  static SIMGATE OutputGate;
8  /* List of inputgates. Used to start the simulation with a set of input values */
9  static SIMGATE* InputGate = 0;
10
11 /* Print-functions */
12
13 void print_stackentry(STACKENTRY s)
14 {
15     char *constants[] = {
16             "SIMGATE_NULL",
17             "SIMGATE_AND",
18             "SIMGATE_XOR",
19             "SIMGATE_OR",
20             "SIMGATE_OUT",
21             "SIMGATE_IN",
22             "SIMGATE_PIPE",
23             "SIMGATE_NO"
24     };
25     printf("Gate: 0x%x  Type: %s  Gate value: %i  Change value: %i  Delay: %i",
           (uint32_t) s.gate,
26             constants[s.gate->type], s.gate->value, s.value, s.delay);
27 }
28
29 /* Linear feedback shift register pseudo-random number generator */
30 /* Xilinx XAPP210 and XAPP052 */
31 static uint32_t lfsr_seed = 0x01;
32
33 uint8_t lfsr_rand()
34 {
35     uint32_t bit;
36     bit = ((lfsr_seed >> 31) ^ (lfsr_seed >> 21) ^ (lfsr_seed >> 1) ^ (lfsr_seed)) &
           1;
37     lfsr_seed = (lfsr_seed << 1) | (bit);
38     return bit;
39 }
40
41 void lfsr_reset()
42 {
43     lfsr_seed = 0x1;
44 }
45
46
47 /* Functions for the hashmap */
48
49 /* Calculates a hash for the FA */
50 static uint32_t gethash(SIMHASHMAP* hashmap, FA* element)
51 {
52     return (((uint32_t)element ^ ((uint32_t)element>>3) ^ ((uint32_t)element<<5)) &
           hashmap->mask);
53 }
54
55 static void inithash(SIMHASHMAP* hashmap, uint32_t size)
56 {
```

```
57      double logtemp;
58      hashmap->size = size;
59      logtemp = log((double) size)/log(2.0); /* log2 of size */
60      if (logtemp > (uint32_t) logtemp)
61          hashmap->logsize = (uint32_t) (logtemp+1);
62      else
63          hashmap->logsize = (uint32_t) logtemp;
64      hashmap->size = 1<<hashmap->logsize;
65      hashmap->mask = (hashmap->size)-1;
66      hashmap->map = (SIMBLOCK**) calloc(hashmap->size, sizeof(SIMBLOCK*));
67      assert(hashmap->map != NULL);
68  }
69
70  static void deletehash(SIMHASHMAP* hashmap)
71  {
72      uint32_t i = 0;
73      SIMBLOCK *ptr,*next;
74      SIMGATE *gate, *nextgate;
75      for (i = 0; i < hashmap->size; i++)
76      {
77          while (hashmap->map[i] != NULL)
78          {
79              ptr = hashmap->map[i];
80              while (ptr != 0)
81              {
82                  next = ptr->next;
83                  free(ptr->gates[0]);
84                  free(ptr);
85                  ptr = next;
86              }
87              hashmap->map[i] = NULL;
88          }
89      }
90
91      gate = InputGate;
92      while (gate != 0)
93      {
94          nextgate = gate->next;
95          free(gate);
96          gate = nextgate;
97      }
98      InputGate = 0;
99      free(hashmap->map);
100 }
101
102 static SIMBLOCK* addFA(SIMHASHMAP* hashmap, FA *element)
103 {
104     uint32_t hash;
105     SIMBLOCK *hashelement, *ptr;
106     hash = gethash(hashmap, element);
107     if (hashmap == NULL)
108     {
109         return NULL;
110     }
111     hashelement = (SIMBLOCK*) malloc(sizeof(SIMBLOCK));
112     assert(hashelement != NULL);
113     if (hashelement == NULL)
114     {
115         return NULL;
116     }
117
118     hashelement->element = element;
119     hashelement->next = 0;
120     /*
121     hashelement->inputA = 0;
122     hashelement->inputB = 0;
123     hashelement->inputC = 0;
124     hashelement->outA = 0;
125     hashelement->outB = 0;
126     */
```

```
127
128     if (hashmap->map[hash] == 0)
129     {
130         hashmap->map[hash] = hashelement;
131     }
132     else
133     {
134         ptr = hashmap->map[hash];
135         while (ptr->next != 0) ptr = ptr->next;
136         ptr->next = hashelement;
137     }
138     return hashelement;
139 }
140
141 SIMBLOCK* getFA(SIMHASHMAP* hashmap, FA* element)
142 {
143     SIMBLOCK *ptr;
144     if (element == NULL)
145     {
146         return NULL;
147     }
148     ptr = hashmap->map[gethash(hashmap, element)];
149     while (ptr != NULL)
150     {
151         if (ptr->element == element)
152             return ptr;
153         ptr = ptr->next;
154     }
155     return NULL;
156 }
157
158 static void printhashmap(SIMHASHMAP* hashmap)
159 {
160     SIMBLOCK *ptr;
161     int i;
162     uint32_t counter;
163     for (i=0; i<hashmap->size; i++)
164     {
165         counter = 0;
166         ptr = hashmap->map[i];
167         if (ptr != NULL)
168         {
169             counter++;
170             ptr = ptr->next;
171             while (ptr != NULL)
172             {
173                 counter++;
174                 ptr = ptr->next;
175             }
176         }
177         printf("Hashmap %.6i: %i\n", i, counter);
178     }
179 }
180
181 static void printbighashmap(SIMHASHMAP* hashmap)
182 {
183     SIMBLOCK *ptr;
184     int i;
185     uint32_t counter;
186     for (i=0; i<hashmap->size; i++)
187     {
188         counter = 0;
189         ptr = hashmap->map[i];
190         if (ptr != NULL)
191         {
192             printf("Hashmap(%.4i)(0): %x\n", i, (unsigned int) ptr->element);
193             counter++;
194             ptr = ptr->next;
195             while (ptr != NULL)
196             {
```

```
197                 printf("Hashmap(%.4i)(%i): %x\n", i, counter, (unsigned int)
                        ptr->element);
198                 counter++;
199                 ptr = ptr->next;
200             }
201         }
202     }
203 }
204 /* Stack functions */
205 static FASTACK* initstack(uint32_t size)
206 {
207     int i = 0;
208     FASTACK* stack = malloc(sizeof(FASTACK));
209     assert(stack != NULL);
210     if (stack == NULL)
211     {
212         printf("Out of memory to make stack for simulation\n");
213         return NULL;
214     }
215     stack->page = calloc(size, sizeof(STACKPAGE));
216     assert(stack->page != NULL);
217     stack->size = size;
218     stack->current = 0;
219     if (stack->page == NULL)
220     {
221         printf("Out of memory to make stack for simulation\n");
222         return NULL;
223     }
224     for (i=0; i < size; i++)
225     {
226         stack->page[i].stack = calloc(INIT_STACK_SIZE, sizeof(STACKENTRY));
227         assert(stack->page[i].stack != NULL);
228         if (stack->page[i].stack == NULL)
229         {
230             printf("Out of memory to make stack for simulation\n");
231         }
232         stack->page[i].size = INIT_STACK_SIZE;
233         stack->page[i].current = (INIT_STACK_SIZE-1);
234         stack->page[i].last = 0;
235     }
236     return stack;
237 }
238
239 static void deletesimstack(FASTACK* stack)
240 {
241     int i = 0;
242     for (i = 0; i < stack->size; i++)
243     {
244         free(stack->page[i].stack);
245     }
246     free(stack->page);
247     free(stack);
248     stack = NULL;
249
250 }
251
252 static void pushsimstack(FASTACK* stack, STACKENTRY element)
253 {
254     uint32_t crnt = stack->current+element.delay;
255     assert(stack->size >= element.delay);
256     if (crnt >= stack->size)
257         crnt -= stack->size;
258     stack->page[crnt].current++;
259     if (stack->page[crnt].current >= stack->page[crnt].size)
260         stack->page[crnt].current = 0;
261     /* Create a larger stack if the stack fills up
262      * In what order is the stack is in is not important. Everything
263      * happens at the same time anyway.
264      * We just double the stack, since thats most effective
265      */
```

```
266     if (stack->page[crnt].stack[stack->page[crnt].current].gate != NULL)
267     {
268         #ifdef DEBUG
269         printf("Expanding stack(%i) %i->%i\n", crnt, stack->page[crnt].size,
                stack->page[crnt].size<<1);
270         #endif
271         STACKENTRY *temp = calloc(stack->page[crnt].size<<1, sizeof(STACKENTRY));
272         if (temp == NULL)
273         {
274             printf("Could not create a larger stack, simultation result will be
                    wrong\n");
275             return;
276         }
277         memcpy(temp,
278                 &stack->page[crnt].stack[stack->page[crnt].current],
279                 (stack->page[crnt].size -
                        stack->page[crnt].current)*sizeof(STACKENTRY));
280         memcpy(&temp[stack->page[crnt].size - stack->page[crnt].current],
281                 &stack->page[crnt].stack[0],
282                 (stack->page[crnt].current)*sizeof(STACKENTRY));
283         free((stack->page[crnt]).stack);
284         stack->page[crnt].stack = temp;
285         stack->page[crnt].current = stack->page[crnt].size;
286         stack->page[crnt].last = 0;
287         stack->page[crnt].size <<= 1; /* Multiply with 2 */
288     }
289     stack->page[crnt].stack[stack->page[crnt].current] = element;
290
291 }
292
293 static STACKENTRY popsimstack(FASTACK* stack)
294 {
295     uint32_t crnt = stack->current;
296     STACKENTRY retval;
297     retval = stack->page[crnt].stack[stack->page[crnt].last];
298     if (retval.gate == NULL)
299         return retval;
300     stack->page[crnt].stack[stack->page[crnt].last].gate = NULL;
301     stack->page[crnt].last++;
302     if(stack->page[crnt].last >= stack->page[crnt].size)
303     {
304         stack->page[crnt].last = 0;
305     }
306     return retval;
307 }
308
309 static void nextsimstack(FASTACK* stack)
310 {
311     stack->current = stack->current+1>=stack->size?0:stack->current+1;
312 }
313
314 static int simstackisempty(FASTACK* stack)
315 {
316     uint8_t isempty = 1;
317     uint32_t i;
318     for (i = 0; i < stack->size; i++)
319     {
320         if (stack->page[i].stack[stack->page[i].last].gate != NULL)
321         {
322             isempty = 0;
323         }
324     }
325     return isempty;
326 }
327
328 static int simstackiscurrentempty(FASTACK* stack)
329 {
330     uint32_t crnt = stack->current;
331     if (stack->page[crnt].stack[stack->page[crnt].last].gate == NULL)
332     {
```

```
333         return 1;
334     }
335     else
336     {
337         return 0;
338     }
339 }
340 
341 
342 /* Utility functions */
343 uint8_t FindFAInput(const FA* out, const FA* in)
344 {
345     return 0;
346 }
347 
348 struct SIMGATE* findGateInput(const SIMBLOCK* current, const SIMBLOCK* input)
349 {
350     return 0;
351 }
352 
353 /* Simulator */
354 
355 struct SIMGATE* newInputGate (struct SIMGATE* last)
356 {
357     last->type = SIMGATE_IN;
358     last->input[0] = 0;
359     last->input[1] = 0;
360     last->output[0] = 0;
361     last->output[1] = 0;
362     last->next = malloc(sizeof(SIMGATE));
363     assert(last->next != NULL);
364     last->next->first = last->first;
365     last->next->next = 0;
366     return last;
367 }
368 
369 uint32_t InitTree(ADDERTREE *WTree, FASTACK* stack, SIMHASHMAP* hashmap)
370 {
371     uint32_t Outputs = 0;
372     ADDERTREE *CurrentTree = WTree;
373     FAGROUP *CurrentGroup, *TopGroup;
374     FA *CurrentFA, *TopFA;
375 
376     SIMBLOCK *ptr;
377     SIMGATE *inputlist;
378 
379     inputlist = malloc(sizeof(SIMGATE));
380     assert(inputlist != NULL);
381     inputlist->first = inputlist;
382     inputlist->next = 0;
383     InputGate = inputlist;
384 
385     OutputGate.type = SIMGATE_OUT;
386 
387     /* Make SIMBLOCK for each FA, and add FAs into the hashmap */
388     CurrentTree = WTree;
389     while(CurrentTree != NULL)
390     {
391         CurrentGroup = CurrentTree->FaGrp;
392         TopGroup = CurrentTree->FaGrp;
393         while(CurrentGroup != NULL)
394         {
395             CurrentFA = CurrentGroup->Grp;
396             TopFA = CurrentGroup->Grp;
397             while(CurrentFA != NULL)
398             {
399                 ptr = addFA(hashmap, CurrentFA);
400                 if (ptr == NULL)
401                 {
402                     printf("Error adding FA to HASH\n");
```

```
403                 }
404
405                 if (CurrentFA->OutA == ExternalOutput)
406                     Outputs++;
407                 if (CurrentFA->OutB == ExternalOutput)
408                     Outputs++;
409                 // NEXT FA
410
411                 /* Allocate memory for gates */
412                 ptr->gates[0] = (SIMGATE*) calloc(5, sizeof(SIMGATE));
413                 assert(ptr->gates[0] != NULL);
414                 ptr->gates[0]->value = 0;
415                 ptr->gates[0]->timestamp = 0;
416                 uint32_t i;
417                 for (i = 1; i<GATESIZE; i++)
418                 {
419                     ptr->gates[i] = ptr->gates[0] + i;
420                     ptr->gates[i]->value = 0;
421                     ptr->gates[i]->timestamp = 0;
422                 }
423
424                 /* Define which internal gates represent the output of the block */
425                 if (CurrentFA->Status & FA_ELEMENT)
426                 {
427                     ptr->output[0] = ptr->gates[3];
428                     ptr->output[1] = ptr->gates[4];
429                 }
430                 else if (CurrentFA->Status & HA_ELEMENT)
431                 {
432                     ptr->output[0] = ptr->gates[0];
433                     ptr->output[1] = ptr->gates[1];
434                 }
435                 else if (CurrentFA->Status & NO_ELEMENT)
436                 {
437                     ptr->output[0] = ptr->gates[0];
438                     if (CurrentFA->InB != NULL)
439                     {
440                         ptr->output[1] = ptr->gates[1];
441                     }
442                 }
443
444                 CurrentFA = CurrentFA->Next;
445             }
446             CurrentGroup = CurrentGroup->Next;
447         }
448         CurrentTree = CurrentTree->Next;
449     }
450
451     /* Populate predefined actions for each SIMFA */
452     CurrentTree = WTree;
453     while(CurrentTree != NULL)
454     {
455         CurrentGroup = CurrentTree->FaGrp;
456         TopGroup = CurrentTree->FaGrp;
457         while(CurrentGroup != NULL)
458         {
459             CurrentFA = CurrentGroup->Grp;
460             TopFA = CurrentGroup->Grp;
461             while(CurrentFA != NULL)
462             {
463                 ptr = getFA(hashmap, CurrentFA);
464                 if (ptr == NULL)
465                 {
466                     printf("Error getting FA from HASH\n");
467                 }
468
469                 SIMGATE *inputA, *inputB, *inputC;
470
471                 /* Find inputA */
472                 if (ptr->element->InA != 0 && ptr->element->InA->OutA == ptr->element)
```

```
473                 {
474                     SIMBLOCK* in = getFA(hashmap, ptr->element->InA);
475                     if (in != 0)
476                     {
477                         inputA = in->output[0];
478                     }
479                 }
480                 else if (ptr->element->InA != 0 && ptr->element->InA->OutB ==
                        ptr->element)
481                 {
482                     SIMBLOCK* in = getFA(hashmap, ptr->element->InA);
483                     if (in != 0)
484                     {
485                         inputA = in->output[1];
486                     }
487                 }
488                 else if (ptr->element->InA == ExternalInput)
489                 {
490                     /* Add a dummy gate, used to set the input value */
491                     inputA = newInputGate(inputlist);
492                     inputlist = inputlist->next;
493                 }
494                 else
495                 {
496                     inputA = 0;
497                 }
498                 /* Find inputB */
499                 if (ptr->element->InB != 0 && ptr->element->InB->OutA == ptr->element)
500                 {
501                     SIMBLOCK* in = getFA(hashmap, ptr->element->InB);
502                     if (in != 0)
503                     {
504                         inputB = in->output[0];
505                     }
506                 }
507                 else if (ptr->element->InB != 0 && ptr->element->InB->OutB ==
                        ptr->element)
508                 {
509                     SIMBLOCK* in = getFA(hashmap, ptr->element->InB);
510                     if (in != 0)
511                     {
512                         inputB = in->output[1];
513                     }
514                 }
515                 else if (ptr->element->InB == ExternalInput)
516                 {
517                     /* Add a dummy gate, used to set the input value */
518                     inputB = newInputGate(inputlist);
519                     inputlist = inputlist->next;
520                 }
521                 else
522                 {
523                     inputB = 0;
524                 }
525                 /* Find inputC */
526                 if (ptr->element->InC != 0 && ptr->element->InC->OutA == ptr->element)
527                 {
528                     SIMBLOCK* in = getFA(hashmap, ptr->element->InC);
529                     assert(in != 0);
530                     if (in != 0)
531                     {
532                         assert(in->output[0] != 0);
533                         inputC = in->output[0];
534                     }
535                 }
536                 else if (ptr->element->InC != 0 && ptr->element->InC->OutB ==
                        ptr->element)
537                 {
538                     SIMBLOCK* in = getFA(hashmap, ptr->element->InC);
539                     assert(in != 0);
```

```
540                     if (in != 0)
541                     {
542                         assert(in->output[1] != 0);
543                         inputC = in->output[1];
544                     }
545                 }
546                 else if (ptr->element->InC == ExternalInput)
547                 {
548                     /* Add a dummy gate, used to set the input value */
549                     inputC = newInputGate(inputlist);
550                     inputlist = inputlist->next;
551                 }
552                 else
553                 {
554                     inputC = 0;
555                 }
556
557                 /* Add inputs to block */
558                 ptr->input[0] = inputA;
559                 ptr->input[1] = inputB;
560                 ptr->input[2] = inputC;
561                 if (ptr->element->Status & FA_ELEMENT)
562                 {
563                     assert(ptr->input[2] != NULL);
564                 }
565
566
567
568                 if (ptr->element->Status & FA_ELEMENT)
569                 {
570                     /* Reverse connection */
571                     if (inputA != 0)
572                     {
573                         inputA->output[0] = ptr->gates[0]; /* Connected to X */
574                         inputA->output[1] = ptr->gates[2]; /* Connected to Z */
575                     }
576                     if (inputB != 0)
577                     {
578                         inputB->output[0] = ptr->gates[0]; /* Connected to X */
579                         inputB->output[1] = ptr->gates[2]; /* Connected to Z */
580                     }
581                     if (inputC !=0)
582                     {
583                         inputC->output[0] = ptr->gates[1]; /* Connected to Y */
584                         inputC->output[1] = ptr->gates[3]; /* Connected to S */
585                     }
586
587                     /* XOR-gate - Output X */
588                     ptr->gates[0]->input[0] = inputA; /* input A */
589                     ptr->gates[0]->input[1] = inputB; /* input B */
590                     ptr->gates[0]->output[0] = ptr->gates[1]; /* Connected to */
591                     ptr->gates[0]->output[1] = ptr->gates[3]; /* Connected to */
592                     ptr->gates[0]->parent = ptr;
593                     ptr->gates[0]->delay = 200;
594                     ptr->gates[0]->type = SIMGATE_XOR;
595                     /* AND-gate - Output Y */
596                     ptr->gates[1]->input[0] = inputC; /* input C */
597                     ptr->gates[1]->input[1] = ptr->gates[0]; /* line X */
598                     ptr->gates[1]->output[0] = ptr->gates[4]; /* Connected to */
599                     ptr->gates[1]->output[1] = 0; /* Connected to */
600                     ptr->gates[1]->parent = ptr;
601                     ptr->gates[1]->delay = 46;
602                     ptr->gates[1]->type = SIMGATE_AND;
603                     /* AND-gate - Output Z */
604                     ptr->gates[2]->input[0] = inputA; /* input A */
605                     ptr->gates[2]->input[1] = inputB; /* input B */
606                     ptr->gates[2]->output[0] = ptr->gates[4]; /* Connected to */
607                     ptr->gates[2]->output[1] = 0; /* Connected to */
608                     ptr->gates[2]->parent = ptr;
609                     ptr->gates[2]->delay = 294;
```

```
610                     ptr->gates[2]->type = SIMGATE_AND;
611                     /* XOR-gate - Output S */
612                     ptr->gates[3]->input[0] = ptr->gates[0]; /* line X */
613                     ptr->gates[3]->input[1] = inputC; /* input C */
614                     ptr->gates[3]->output[0] = 0; /* Connected to output */
615                     ptr->gates[3]->output[1] = 0; /* Connected to output */
616                     ptr->gates[3]->parent = ptr;
617                     ptr->gates[3]->delay = 443;
618                     ptr->gates[3]->type = SIMGATE_AND;
619                     //ptr->output[0] = ptr->gates[3];
620                     /* OR-gate - Output C */
621                     ptr->gates[4]->input[0] = ptr->gates[1]; /* line Y */
622                     ptr->gates[4]->input[1] = ptr->gates[2]; /* line Z */
623                     ptr->gates[4]->output[0] = 0; /* Connected to output */
624                     ptr->gates[4]->output[1] = 0; /* Connected to output */
625                     ptr->gates[4]->parent = ptr;
626                     ptr->gates[4]->delay = 200;
627                     ptr->gates[4]->type = SIMGATE_OR;
628                     //ptr->output[1] = ptr->gates[4];
629
630                 }
631                 else if (ptr->element->Status & HA_ELEMENT)
632                 {
633                     /* Reverse connection */
634                     if (inputA != 0)
635                     {
636                         inputA->output[0] = ptr->gates[0]; /* Connected to S */
637                         inputA->output[1] = ptr->gates[1]; /* Connected to C */
638                     }
639                     if (inputB != 0)
640                     {
641                         inputB->output[0] = ptr->gates[0]; /* Connected to S */
642                         inputB->output[1] = ptr->gates[1]; /* Connected to C */
643                     }
644                     /* XOR-gate - Output S */
645                     ptr->gates[0]->input[0] = inputA;
646                     ptr->gates[0]->input[1] = inputB;
647                     ptr->gates[0]->parent = ptr;
648                     ptr->gates[0]->delay = 413;
649                     ptr->gates[0]->type = SIMGATE_XOR;
650                     //ptr->output[0] = ptr->gates[0];
651                     /* AND-gate - Output C */
652                     ptr->gates[1]->input[0] = inputA;
653                     ptr->gates[1]->input[1] = inputB;
654                     ptr->gates[1]->parent = ptr;
655                     ptr->gates[1]->delay = 508;
656                     ptr->gates[1]->type = SIMGATE_AND;
657                     //ptr->output[1] = ptr->gates[1];
658                 }
659                 else if (ptr->element->Status & NO_ELEMENT)
660                 {
661                     /* Feed-through - Port A */
662                     if (inputA != 0)
663                     {
664                         ptr->gates[0]->input[0] = inputA;
665                         ptr->gates[0]->parent = ptr;
666                         ptr->gates[0]->delay = 150;
667                         ptr->gates[0]->type = SIMGATE_NO;
668                         //ptr->output[0] = ptr->gates[0];
669                     }
670                     /* Feed-through - Port B */
671                     if (inputB != 0)
672                     {
673                         ptr->gates[1]->input[0] = inputA;
674                         ptr->gates[1]->parent = ptr;
675                         ptr->gates[1]->delay = 150;
676                         ptr->gates[1]->type = SIMGATE_NO;
677                         //ptr->output[1] = ptr->gates[1];
678                     }
679                 }
```

```
680                 /* Check if we have reached the end of the chain */
681                 if (ptr->element->OutA == ExternalOutput)
682                 {
683                     ptr->output[0]->output[0] = &OutputGate;
684                 }
685                 if (ptr->element->OutB == ExternalOutput)
686                 {
687                     ptr->output[1]->output[0] = &OutputGate;
688                 }
689                 CurrentFA = CurrentFA->Next;
690             }
691             CurrentGroup = CurrentGroup->Next;
692         }
693         CurrentTree = CurrentTree->Next;
694     }
695
696
697     //printbighashmap();
698     return Outputs;
699 }
700
701 static void SetInputs(ADDERTREE* WTree, FASTACK* stack, SIMHASHMAP* hashmap)
702 {
703     ADDERTREE *CurrentTree = WTree;
704     FAGROUP *CurrentGroup, *TopGroup;
705     FA *CurrentFA, *TopFA;
706
707     // SIMFA *ptr;
708     STACKENTRY entry;
709     entry.delay = 0;
710
711     CurrentTree = WTree;
712     while(CurrentTree != NULL)
713     {
714         CurrentGroup = CurrentTree->FaGrp;
715         TopGroup = CurrentTree->FaGrp;
716         while(CurrentGroup != NULL)
717         {
718             CurrentFA = CurrentGroup->Grp;
719             TopFA = CurrentGroup->Grp;
720             while(CurrentFA != NULL)
721             {
722                 SIMBLOCK *block = getFA(hashmap, CurrentFA);
723                 /* Add inputvectors to the stack */
724                 if (CurrentFA->InA == ExternalInput)
725                 {
726                     entry.gate = block->input[0];
727                     entry.value = lfsr_rand();
728                     block->input[0]->value = entry.value;
729                     pushsimstack(stack, entry);
730                 }
731                 if (CurrentFA->InB == ExternalInput)
732                 {
733                     entry.gate = block->input[1];
734                     entry.value = lfsr_rand();
735                     block->input[1]->value = entry.value;
736                     pushsimstack(stack, entry);
737                 }
738                 if (CurrentFA->InC == ExternalInput)
739                 {
740                     entry.gate = block->input[2];
741                     entry.value = lfsr_rand();
742                     block->input[2]->value = entry.value;
743                     pushsimstack(stack, entry);
744                 }
745
746                 // NEXT FA
747                 CurrentFA = CurrentFA->Next;
748             }
749             CurrentGroup = CurrentGroup->Next;
```

```
750         }
751         CurrentTree = CurrentTree->Next;
752     }
753 }
754
755 struct SIMSTRUCTURES InitEstimationStructures(ADDERTREE *WTree)
756 {
757     struct SIMSTRUCTURES retval;
758     ADDERTREE *CurrentTree = WTree;
759     FAGROUP *CurrentGroup, *TopGroup;
760     FA *CurrentFA, *TopFA;
761
762     /* Make sure the activity starts at 0 */
763     retval.activity = 0;
764
765     /* Count the number of FA/HA/NO-elements */
766     retval.count = 0;
767     retval.maxdepths = 0;
768     while(CurrentTree != NULL)
769     {
770         if (CurrentTree->Depth > retval.maxdepths)
771         {
772             retval.maxdepths = CurrentTree->Depth;
773         }
774         CurrentGroup = CurrentTree->FaGrp;
775         TopGroup = CurrentTree->FaGrp;
776         while(CurrentGroup != NULL)
777         {
778             CurrentFA = CurrentGroup->Grp;
779             TopFA = CurrentGroup->Grp;
780             while(CurrentFA != NULL)
781             {
782                 retval.count++;
783                 // NEXT FA
784                 CurrentFA = CurrentFA->Next;
785             }
786             CurrentGroup = CurrentGroup->Next;
787         }
788         CurrentTree = CurrentTree->Next;
789     }
790
791     /* Init the hashtable */
792     inithash(&(retval.hashmap), retval.count);
793     /* Init the different event-stacks */
794     //stack = initstack(FA_DELAY>HA_DELAY?FA_DELAY+1:HA_DELAY+1);
795     retval.stack = initstack(DELAY_MAX+1);
796     retval.outstack = initstack((LINE_DELAY*2)+1);
797     /* Fill the hashtable */
798     retval.outputs = InitTree(WTree, retval.stack, &(retval.hashmap));
799     /* Remember what addertree these stacks and hashmaps belongs to */
800     retval.tree = WTree;
801
802     return retval;
803 }
804
805 void deallocEstimationStructures(struct SIMSTRUCTURES sim)
806 {
807     deletesimstack(sim.stack);
808     deletesimstack(sim.outstack);
809     deletehash(&(sim.hashmap));
810 }
811
812 double RunTestSimulationOnTree(ADDERTREE *WTree, uint32_t iterations, uint32_t
        preruns)
813 {
814     double retval;
815     struct SIMSTRUCTURES sim = InitEstimationStructures(WTree);
816     retval = RunTestSimulationOnStructures(sim, iterations, preruns);
817     deallocEstimationStructures(sim);
818
```

```
819     return retval;
820 }
821
822 double RunTestSimulationOnStructures(struct SIMSTRUCTURES sim, uint32_t iterations,
        uint32_t preruns)
823 {
824     int i = 0;
825     double estimate = 0.0;
826     uint32_t zerotoone = 0, onetozero = 0;
827     uint32_t timestamp = 0;
828
829     SIMGATE *ptr;
830     STACKENTRY entry, nextentry;
831     nextentry.value = 0;
832     nextentry.gate = 0;
833
834     struct timeval starttime, stoptime;
835
836     gettimeofday(&starttime, NULL);
837
838     /* Reset the pseudo-random number generator */
839     lfsr_reset();
840
841     /* Start simulation */
842     for (i = 0; i < (iterations+preruns); i++)
843     {
844         /* Fill the tree with input */
845         SetInputs(sim.tree, sim.stack, &(sim.hashmap));
846         /* Run until all stacks are empty */
847         while (!simstackisempty(sim.stack) || !simstackisempty(sim.outstack))
848         {
849             /* Run until this spesific time is done */
850             while (!simstackiscurrentempty(sim.stack) ||
                    !simstackiscurrentempty(sim.outstack))
851             {
852                 entry = popsimstack(sim.stack);
853                 /* Change value of lines */
854                 while (entry.gate != NULL)
855                 {
856                     ptr = entry.gate;
857                     if (ptr->value != entry.value || ptr->type == SIMGATE_IN)
858                     {
859                         /* Monte Carlo - Do not count activity during prerun-period */
860                         if (i >= preruns)
861                         {
862                             if (ptr->value == 0 && entry.value == 1)
863                             {
864                                 ptr->activity++;
865                                 zerotoone++;
866                             }
867                             if (ptr->value == 1 && entry.value == 0)
868                             {
869                                 onetozero++;
870                             }
871                         }
872
873                         ptr->value = entry.value;
874                         /* Calculate line delay */
875                         if (ptr->output[0] != 0 && ptr->output[1] != 0)
876                         {
877                             nextentry.delay = 2 * LINE_DELAY;
878                         }
879                         else
880                         {
881                             nextentry.delay = LINE_DELAY;
882                         }
883                         if (ptr->type == SIMGATE_NO)
884                         {
885                             nextentry.delay = ptr->delay;
886                         }
```

```
887                         /* Send value to next gate */
888                         if (ptr->output[0] != 0)
889                         {
890                             nextentry.gate = ptr->output[0];
891                             pushsimstack(sim.outstack, nextentry);
892                         }
893                         if (ptr->output[1] != 0)
894                         {
895                             nextentry.gate = ptr->output[1];
896                             pushsimstack(sim.outstack, nextentry);
897                         }
898                     }
899                     /* Input calculation */
900                     entry = popsimstack(sim.stack);
901                 }
902
903                 /* Change timestamp */
904                 timestamp++;
905                 entry = popsimstack(sim.outstack);
906                 while (entry.gate != NULL)
907                 {
908                     ptr = entry.gate;
909                     nextentry.gate = entry.gate;
910                     /* Output calculation */
911                     switch (ptr->type) {
912                         case SIMGATE_AND:
913                             nextentry.value = ptr->input[0]->value &
                                    ptr->input[1]->value;
914                             nextentry.delay = ptr->delay;
915                             break;
916                         case SIMGATE_XOR:
917                             nextentry.value = ptr->input[0]->value ^
                                    ptr->input[1]->value;
918                             nextentry.delay = ptr->delay;
919                             break;
920                         case SIMGATE_OR:
921                             nextentry.value = ptr->input[0]->value |
                                    ptr->input[1]->value;
922                             nextentry.delay = ptr->delay;
923                             break;
924                         case SIMGATE_NO:
925                             if (ptr->input[1] != 0)
926                             {
927                                 nextentry.value = ptr->input[1]->value;
928                                 nextentry.delay = ptr->delay;
929                                 pushsimstack(sim.stack, nextentry);
930                             }
931                             nextentry.value = ptr->input[0]->value;
932                             nextentry.delay = ptr->delay;
933                             break;
934                         default:
935                             break;
936                     }
937                     if (entry.gate->type != SIMGATE_OUT)
938                     {
939                         pushsimstack(sim.stack, nextentry);
940                     }
941                     /* NEXT */
942                     entry = popsimstack(sim.outstack);
943                 }
944             }
945             /* Next cycle */
946             nextsimstack(sim.stack);
947             nextsimstack(sim.outstack);
948         }
949     } /* End for loop */
950
951     gettimeofday(&stoptime, NULL);
952     starttime.tv_sec = stoptime.tv_sec - starttime.tv_sec;
953     starttime.tv_usec = stoptime.tv_usec - starttime.tv_usec;
```

```
954     if (starttime.tv_usec < 0)
955     {
956         starttime.tv_sec--;
957         starttime.tv_usec += 1000000;
958     }
959
960     estimate = zerotoone;
961
962     printf("␣␣␣␣Powerestimate:␣%f␣␣␣␣0->1:␣%i␣␣␣1->0:␣%i␣(Used␣%i,%.6i␣sec)\n",
            estimate, zerotoone, onetozero,
963             starttime.tv_sec, starttime.tv_usec);
964     //printf("  Outputs: %i   MaxDepth: %i\n", Outputs, MaxDepth);
965
966     return estimate;
967 }
```

C-code C.2: Source file for estimator

C.2 Optimaliztion

C.2.1 optimize.c

C-code C.3: Source file for optimalization routine

```
1
2  #include "modgen.h"
3  #include "estimate.h"
4  #include <assert.h>
5
6  enum faport_t
7  {
8      FAOUTPUT_OUTS, FAOUTPUT_OUTC, FAINPUT_INA, FAINPUT_INB, FAINPUT_INC
9  };
10
11 struct FALISTELEMENT
12 {
13     FA *element;
14     enum faport_t outputport;
15     struct FALISTELEMENT *next;
16     uint32_t priority;
17 };
18
19 struct FALIST
20 {
21     struct FALISTELEMENT ***inputs;
22     struct FALISTELEMENT ***outputs;
23     uint32_t stages;
24     uint32_t columns;
25     uint32_t *startlevel;
26 };
27
28 void initFAList(struct FALIST *list, struct ADDERTREE *WTree)
29 {
30     ADDERTREE *CurrentTree = WTree;
31     FAGROUP *CurrentGroup, *TopGroup;
32     FA *CurrentFA, *TopFA;
33     uint32_t i;
34     uint32_t stage = 0, columns = 0;
35     static const uint32_t extra_stages = 2;
36
37     /* Calculate size of the multiplicator */
38     CurrentTree = WTree;
39     while(CurrentTree != NULL)
40     {
41         CurrentGroup = CurrentTree->FaGrp;
42         TopGroup = CurrentTree->FaGrp;
```

```
43          columns++;
44          stage = 0;
45          while(CurrentGroup != NULL)
46          {
47              CurrentFA = CurrentGroup->Grp;
48              TopFA = CurrentGroup->Grp;
49              stage++;
50              while(CurrentFA != NULL)
51              {
52                  CurrentFA = CurrentFA->Next;
53              }
54              CurrentGroup = CurrentGroup->Next;
55          }
56          if (stage+extra_stages > list->stages)
57          {
58              list->stages = stage+extra_stages;
59          }
60          if (CurrentTree->StartLevel+stage+extra_stages > list->stages)
61          {
62              list->stages = CurrentTree->StartLevel+stage+extra_stages;
63          }
64          CurrentTree = CurrentTree->Next;
65      }
66
67      list->columns = columns+1;
68
69      list->inputs = calloc(list->stages, sizeof(struct FALISTELEMENT**));
70      list->outputs = calloc(list->stages, sizeof(struct FALISTELEMENT**));
71      assert(list->inputs != NULL);
72
73      /* Calculate starting stages */
74      list->startlevel = calloc(list->columns, sizeof(uint32_t));
75      CurrentTree = WTree;
76      columns = 0;
77      while(CurrentTree != NULL)
78      {
79          list->startlevel[columns] = CurrentTree->StartLevel;
80          columns++;
81          CurrentTree = CurrentTree->Next;
82      }
83
84      for (i = 0; i < list->stages; i++)
85      {
86          list->inputs[i] = calloc(list->columns, sizeof(struct FALISTELEMENT*));
87          assert(list->inputs[i] != NULL);
88          list->outputs[i] = calloc(list->columns, sizeof(struct FALISTELEMENT*));
89          assert(list->outputs[i] != NULL);
90      }
91  }
92
93  void destroyFAList(struct FALIST *list)
94  {
95      uint32_t i, j;
96      struct FALISTELEMENT *entry, *nextentry;
97
98      for (i = 0; i < list->stages; i++)
99      {
100         for (j = 0; j < list->columns; j++)
101         {
102             entry = list->inputs[i][j];
103             while (entry != NULL)
104             {
105                 nextentry = entry->next;
106                 free(entry);
107                 entry = nextentry;
108             }
109
110             entry = list->outputs[i][j];
111             while (entry != NULL)
112             {
```

```
113                 nextentry = entry->next;
114                 free(entry);
115                 entry = nextentry;
116             }
117         }
118         free(list->inputs[i]);
119         free(list->outputs[i]);
120     }
121
122     free(list->inputs);
123     free(list->outputs);
124 }
125
126 void setFAListPriorities(struct FALIST* list, struct SIMSTRUCTURES sim)
127 {
128     uint32_t stage, column;
129     struct FALISTELEMENT *entry;
130     struct SIMBLOCK* block;
131
132     for (stage = 0; stage < list->stages; stage++)
133     {
134         for (column = 0; column < list->columns; column++)
135         {
136             /* Set activity on outputports */
137             entry = list->outputs[stage][column];
138             while (entry != NULL)
139             {
140                 block = getFA(&(sim.hashmap), entry->element);
141                 if (entry->outputport == FAOUTPUT_OUTS)
142                 {
143                     /* Get activity from SUM-output, and use it to during the sort */
144                     entry->priority = block->output[0]->activity;
145                 }
146                 if (entry->outputport == FAOUTPUT_OUTC)
147                 {
148                     /* Get activity from CARRY-output, and use it during the sort */
149                     entry->priority = block->output[1]->activity;
150                 }
151
152                 entry = entry->next;
153             }
154         }
155     }
156 }
157
158 /* Insertion-sort of FAListElements */
159 struct FALISTELEMENT* sortFAListElement(struct FALISTELEMENT* first)
160 {
161     struct FALISTELEMENT *entry, *sortentry, *lastsortentry, *nextentry;
162     if (first == NULL)
163     {
164         return first;
165     }
166     entry = first;
167     nextentry = first->next;
168     first->next = NULL;
169     while (nextentry != NULL)
170     {
171         /* Get next element in the list */
172         entry = nextentry;
173         nextentry = entry->next;
174         entry->next = NULL;
175         sortentry = first;
176         lastsortentry = NULL;
177         /* Find placement */
178         while (sortentry != NULL && entry->priority < sortentry->priority)
179         {
180             lastsortentry = sortentry;
181             sortentry = sortentry->next;
182         }
```

```
183         /* Insert */
184         /* If last element in the list */
185         if (sortentry == NULL)
186         {
187             lastsortentry->next = entry;
188         }
189         /* If first element in the list */
190         else if (sortentry == first)
191         {
192             entry->next = sortentry;
193             first = entry;
194         }
195         /* If middle element */
196         else
197         {
198             entry->next = sortentry;
199             lastsortentry->next = entry;
200         }
201     }
202     return first;
203 }
204
205 void sortFAList(struct FALIST* list, struct SIMSTRUCTURES sim)
206 {
207     uint32_t stage, column;
208     //struct FALISTELEMENT *entry, *nextentry;
209
210     setFAListPriorities(list, sim);
211
212     for (stage = 0; stage < list->stages; stage++)
213     {
214         for (column = 0; column < list->columns; column++)
215         {
216             /* Sort inputs */
217             list->inputs[stage][column] =
                    sortFAListElement(list->inputs[stage][column]);
218
219             /* Sort outputs */
220             list->outputs[stage][column] =
                    sortFAListElement(list->outputs[stage][column]);
221         }
222     }
223 }
224
225 void rearrangeFAList(struct FALIST* list)
226 {
227     uint32_t stage, column;
228     struct FALISTELEMENT *input, *output, *iterator;
229     uint32_t external = 0;
230
231     for (stage = 0; stage < list->stages; stage++)
232     {
233         for (column = 0; column < list->columns; column++)
234         {
235             input = list->inputs[stage][column];
236             output = list->outputs[stage][column];
237             while (input != NULL)
238             {
239                 external = 0;
240
241                 /* Connect outputelement to input */
242                 if (input->outputport == FAINPUT_INA)
243                 {
244                     if (input->element->InA == ExternalInput)
245                     {
246                         external = 1;
247                     }
248                     else
249                     {
250                         input->element->InA = output->element;
```

```
251                     }
252                 }
253                 else if (input->outputport == FAINPUT_INB)
254                 {
255                     if (input->element->InB == ExternalInput)
256                     {
257                         external = 1;
258                     }
259                     else
260                     {
261                         input->element->InB = output->element;
262                     }
263                 }
264                 else if (input->outputport == FAINPUT_INC)
265                 {
266                     if (input->element->InC == ExternalInput)
267                     {
268                         external = 1;
269                     }
270                     else
271                     {
272                         input->element->InC = output->element;
273                     }
274                 }
275                 else
276                 {
277                     assert(0); /* This should not happen */
278                 }
279
280                 /* Connect inputelement to output */
281                 if (external != 1)
282                 {
283                     assert(output != NULL);
284                     if (output->outputport == FAOUTPUT_OUTS)
285                     {
286                         output->element->OutA = input->element;
287                     }
288                     else if (output->outputport == FAOUTPUT_OUTC)
289                     {
290                         output->element->OutB = input->element;
291                     }
292                     else
293                     {
294                         assert(0); /* This should not happen */
295                     }
296                     output = output->next;
297                 }
298
299                 input = input->next;
300             }
301             /* Move outputs to the next stage, if they havent been connected yet */
302             if (output != NULL)
303             {
304                 iterator = list->outputs[stage][column];
305                 /* Is this the first in the list? */
306                 if (output == iterator)
307                 {
308                     list->outputs[stage][column] = 0x0;
309                 }
310                 else
311                 {
312                     while (iterator->next != output)
313                     {
314                         iterator = iterator->next;
315                         assert(iterator == NULL);
316                     }
317                     /* We have found the last element that was connect */
318                     iterator->next = NULL;
319                 }
320
```

```
321                 /* Lets shorten this this list, and append the unconnected ones on the
                        next stage */
322                 iterator = list->outputs[stage+1][column];
323                 if (iterator == NULL)
324                 {
325                     list->outputs[stage+1][column] = output;
326                 }
327                 else
328                 {
329                     while (iterator->next != NULL)
330                     {
331                         iterator = iterator->next;
332                     }
333                     iterator->next = output;
334                     list->outputs[stage+1][column] =
                            sortFAListElement(list->outputs[stage+1][column]);
335                 }
336             }
337         }
338     }
339 }
340
341
342 struct FALISTELEMENT* addPort(struct FALIST *list, uint32_t stage, uint32_t column,
        FA* element, enum faport_t port)
343 {
344     /* Do not change the finished outputs of the multiplication */
345     if (port == FAOUTPUT_OUTS && element->OutA == ExternalOutput)
346     {
347         return NULL;
348     }
349     else if (port == FAOUTPUT_OUTC && element->OutB == ExternalOutput)
350     {
351         return NULL;
352     }
353
354
355     struct FALISTELEMENT *entry;
356     struct FALISTELEMENT *new_entry = malloc(sizeof(struct FALISTELEMENT));
357     new_entry->outputport = port;
358     new_entry->element = element;
359     new_entry->next = NULL;
360
361     /* Check if this stage is an too early stage */
362     if (stage < list->startlevel[column])
363     {
364         stage = list->startlevel[column];
365     }
366
367     if (port == FAOUTPUT_OUTC)
368     {
369         assert(element->OutB != 0x0);
370     }
371
372     if (port == FAOUTPUT_OUTS || port == FAOUTPUT_OUTC)
373     {
374         entry = list->outputs[stage][column];
375         if (entry == NULL)
376         {
377             list->outputs[stage][column] = new_entry;
378         }
379         else
380         {
381             while (entry->next != NULL)
382             {
383                 entry = entry->next;
384             }
385             entry->next = new_entry;
386         }
387     }
```

```
388     else if (port == FAINPUT_INA || port == FAINPUT_INB || port == FAINPUT_INC)
389     {
390         /* Classify input port */
391         if (new_entry->element->Status & FA_ELEMENT)
392         {
393             if (port == FAINPUT_INC)
394             {
395                 new_entry->priority = 1;
396             }
397             else
398             {
399                 new_entry->priority = 2;
400             }
401         }
402         else if (new_entry->element->Status & HA_ELEMENT)
403         {
404             new_entry->priority = 3;
405         }
406         else if (new_entry->element->Status & NO_ELEMENT)
407         {
408             new_entry->priority = 4;
409         }
410 
411         entry = list->inputs[stage][column];
412         if (entry == NULL)
413         {
414             list->inputs[stage][column] = new_entry;
415         }
416         else
417         {
418             while (entry->next != NULL)
419             {
420                 entry = entry->next;
421             }
422             entry->next = new_entry;
423         }
424     }
425 
426     return new_entry;
427 }
428 
429 void printFAList(struct FALIST* list)
430 {
431     uint32_t stage, column;
432     struct FALISTELEMENT *entry;
433     uint32_t outputs, inputs;
434 
435     for (stage = 0; stage < list->stages; stage++)
436     {
437         for (column = 0; column < list->columns; column++)
438         {
439             printf("\nCOL: %.2d (%.2d)", column, list->startlevel[column]);
440             entry = list->outputs[stage][column];
441             outputs = 0;
442             while (entry != NULL)
443             {
444                 //printf(" %d: 0x%x", i, (uint32_t) entry);
445                 //printf(" %d", entry->priority);
446                 printf(" %d", stage);
447                 outputs++;
448                 if (entry->outputport == FAOUTPUT_OUTS && entry->element->OutA ==
                        ExternalOutput)
449                 {
450                     printf(" OUTA");
451                     outputs--;
452                 }
453                 if (entry->outputport == FAOUTPUT_OUTC && entry->element->OutB ==
                        ExternalOutput)
454                 {
455                     printf(" OUTB");
```

```
456                     outputs--;
457                 }
458                 entry = entry->next;
459             }
460 
461             entry = list->inputs[stage][column];
462             inputs = 0;
463             while (entry != NULL)
464             {
465                 inputs++;
466                 if (entry->outputport == FAINPUT_INA && entry->element->InA ==
                        ExternalInput)
467                 {
468                     printf("␣INA");
469                     inputs--;
470                 }
471                 if (entry->outputport == FAINPUT_INB && entry->element->InB ==
                        ExternalInput)
472                 {
473                     printf("␣INB");
474                     inputs--;
475                 }
476                 if (entry->outputport == FAINPUT_INC && entry->element->InC ==
                        ExternalInput)
477                 {
478                     printf("␣INC");
479                     inputs--;
480                 }
481                 entry = entry->next;
482             }
483 
484             printf("␣IN:␣%d␣␣OUT:␣%d", inputs, outputs);
485         }
486     }
487     printf("\n");
488 }
489 
490 /**
491  *
492  * Takes a multiplicator and rearranges the connections between the block
493  * to decrease the power usage
494  */
495 void PowerOptimize(ADDERTREE* WTree, int iterations)
496 {
497     struct SIMSTRUCTURES sim;
498     double powerusage;
499     ADDERTREE *CurrentTree = WTree;
500     FAGROUP *CurrentGroup, *TopGroup;
501     FA *CurrentFA, *TopFA;
502     uint32_t columns = 0, stage = 0, i = 0;
503     struct FALIST falist;
504 
505     for(i = 0; i<iterations; i++)
506     {
507 
508         /* Initilize test-structures */
509         sim = InitEstimationStructures(WTree);
510         /* Create power-profile */
511         powerusage = RunTestSimulationOnStructures(sim, 100, 10);
512 
513         initFAList(&falist, WTree);
514 
515         /* Add all of the outputs and inputs info the netlist */
516         columns = 0;
517         CurrentTree = WTree;
518         while(CurrentTree != NULL)
519         {
520             CurrentGroup = CurrentTree->FaGrp;
521             TopGroup = CurrentTree->FaGrp;
522             columns++;
```

```
523             stage = 0;
524             while(CurrentGroup != NULL)
525             {
526                 CurrentFA = CurrentGroup->Grp;
527                 TopFA = CurrentGroup->Grp;
528                 stage++;
529                 while(CurrentFA != NULL)
530                 {
531                     struct FALISTELEMENT *entry;
532                     uint32_t effective_stage = stage + CurrentTree->StartLevel;
533                     entry = addPort(&falist, effective_stage+1, columns, CurrentFA,
                            FAOUTPUT_OUTS);
534                     /* Dont add empty line elements, and route line elements through
                            */
535                     if (CurrentFA->Status & NO_ELEMENT)
536                     {
537                         if (CurrentFA->OutB != NULL)
538                         {
539                             entry = addPort(&falist, effective_stage+1, columns,
                                    CurrentFA, FAOUTPUT_OUTC);
540                             assert(CurrentFA->OutB != 0x0);
541                         }
542                     }
543                     else
544                     {
545                         if (CurrentFA->OutB != NULL)
546                         {
547                             /* Add the carry bit over to the next column */
548                             entry = addPort(&falist, effective_stage+1, columns+1,
                                    CurrentFA, FAOUTPUT_OUTC);
549                             assert(CurrentFA->OutB != 0x0);
550                         }
551                     }
552
553                     /* Add inputports */
554                     addPort(&falist, effective_stage, columns, CurrentFA,
                            FAINPUT_INA);
555                     if (CurrentFA->InB != NULL)
556                     {
557                         addPort(&falist, effective_stage, columns, CurrentFA,
                                FAINPUT_INB);
558                     }
559                     if (CurrentFA->Status & FA_ELEMENT)
560                     {
561                         addPort(&falist, effective_stage, columns, CurrentFA,
                                FAINPUT_INC);
562                         assert(CurrentFA->InC != NULL);
563                     }
564
565                     CurrentFA = CurrentFA->Next;
566                 }
567                 CurrentGroup = CurrentGroup->Next;
568             }
569             CurrentTree = CurrentTree->Next;
570         }
571
572         sortFAList(&falist, sim);
573         rearrangeFAList(&falist);
574         //printFAList(&falist);
575
576         destroyFAList(&falist);
577         deallocEstimationStructures(sim);
578     }
579     //printf("Power optimization ended\n");
580 }
```

C-code C.3: Source file for optimalization routine

Bibliography

[1] E. Sand, "Vlsi architectures for speech recognition," Master's thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, 1994.

[2] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, pp. 14–17, Feb 1964.

[3] L. Dadda, "Some scheme for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, Mar 1965.

[4] K. Bickerstaff, M. Schulte, and J. Swartzlander, E.E., "Reduced area multipliers," *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pp. 478–489, Oct 1993.

[5] S. T. Oskuii, *Design of Low-Power Reduction-Trees in Parallel Multipliers*. PhD thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, 2008.

[6] K. Bickerstaff, J. Swartzlander, E.E., and M. Schulte, "Analysis of column compression multipliers," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 33 –39, 2001.

[7] C. Nagendra, M. Irwin, and R. Owens, "Area-time-power tradeoffs in parallel adders," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 43, pp. 689 –702, oct 1996.

[8] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *Computers, IEEE Transactions on*, vol. C-22, pp. 1045 – 1047, dec. 1973.

[9] B. Parhami, *Computer arithmetic: algorithms and hardware designs*. Oxford, UK: Oxford University Press, 2000.

[10] R. Burch, F. Najm, P. Yang, and T. Trick, "A monte carlo approach for power estimation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 1, pp. 63–71, Mar 1993.

[11] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on computers*, vol. 100, no. 35, pp. 677–691, 1986.

[12] P. Alfke, *Efficient Shift Registers, LFSR Counters, and Long Pseudo- Random Sequence Generators*. Xilinx Inc. Application Note: XAPP052.

[13] M. George and P. Alfke, *Linear Feedback Shift Registers in Virtex Devices*. Xilinx Inc. Application Note: XAPP210.

[14] Open Verilog International, *Standard Delay Format Specification, Version 2.1*, 1994. Versjon 2.1.

[15] A. D. BOOTH, "A SIGNED BINARY MULTIPLICATION TECHNIQUE," *Q J Mechanics Appl Math*, vol. 4, no. 2, pp. 236–240, 1951.

[16] A. Habibi and P. Wintz, "Fast multipliers," *IEEE Transactions on Computers*, vol. 19, no. 2, pp. 153–157, 1970.

[17] K.-Y. Khoo, Z. Yu, and J. Willson, A.N., "Improved-booth encoding for low-power multipliers," *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, vol. 1, pp. 62–65 vol.1, Jul 1999.

[18] J. M. Rabaey and M. Pedram, eds., *Low Power Design Methodologies*. Kluwer Academic Publisher, 1996.

[19] D. Liu and C. Svensson, "Trading speed for low power by choice of supply and thresholdvoltages," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 1, pp. 10–17, 1993.

[20] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992.

[21] P. Meier, R. Rutenbar, and L. Carley, "Exploring multiplier architecture and layout for low power," *Custom Integrated Circuits Conference, 1996., Proceedings of the IEEE 1996*, pp. 513–516, May 1996.

[22] J. J. Kalis, "Switching multipliers," Master's thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, 2009.

[23] S. Mathiassen, "Power optimized multipliers," 2008. Project assignment at NTNU.

[24] S. Narendra and A. Chandrakasan, *Leakage in nanometer CMOS technologies*. Springer-Verlag New York Inc, 2006.

[25] C. Piguet, "Low power design in deep submicron 65 and 45 nm technologies," pp. 915–918, dec. 2007.

[26] R. Watts, *Submicron integrated circuits*. Wiley, 1989.

[27] H. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 468–473, 1984.

[28] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, ch. 13. John Wiley & Sons, 1999.

[29] T. Callaway and J. Swartzlander, E.E., "Power-delay characteristics of cmos multipliers," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pp. 26 –32, jul 1997.

[30] L. Dadda, "On parallel digital multipliers," *Alta Frequenza*, vol. 45, pp. 574–580, Oct 1976.

[31] W. J. Townsend, J. Earl E. Swartzlander, and J. A. Abraham, "A comparison of dadda and wallace multiplier delays," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII* (F. T. Luk, ed.), vol. 5205, pp. 552–560, SPIE, 2003.

[32] P. Landman, "High-level power estimation," in *ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design*, (Piscataway, NJ, USA), pp. 29–35, IEEE Press, 1996.

[33] F. N. Najm, "Power estimation techniques for integrated circuits," in *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, (Washington, DC, USA), pp. 492–499, IEEE Computer Society, 1995.

[34] F. N. Najm, "A survey of power estimation techniques in vlsi circuits," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 446–455, 1994.

[35] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists (International Edition)*. Pearson Education, 7th ed., 2001.

[36] M. G. Xakellis and F. N. Najm, "Statistical estimation of the switching activity in digital circuits," in *DAC '94: Proceedings of the 31st annual Design Automation Conference*, (New York, NY, USA), pp. 728–733, ACM, 1994.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, Mass.: The MIT Press, 2nd ed., 2001.