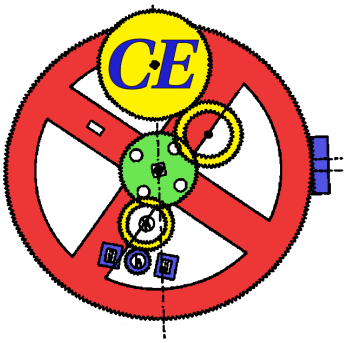

**Computer Engineering**
Mekelweg 4,
2628 CD Delft
The Netherlands
**http://ce.et.tudelft.nl/**

**2010**

# MSc THESIS

# Exploring suitable adder designs for biomedical implants

**Danny P. Riemens**

**CE-MS-2010-31**

## Abstract

Modern applications demand extremely low power budgets in computer architectures for battery-operated devices. In the particular case of implantable devices —the main focus of this thesis— the system must have a long life span and batteries may not be possible or easy to recharge. In addition to power, chip area is also of major concern in this specific scenario. Since implantable devices are sometimes placed at locations inside the body where limited space is available, the implant must be as small as possible. The vast amount of volume of an implant is typically occupied by the battery and its electrodes, so the affordable chip area is very limited. Another reason why we want very small processor cores, is because this approach leaves more space for cache memory and it statistically reduces the chance of hardware failures. In this thesis we focus on the arithmetic unit (AU) of such a core, which is typically the adder/subtracter. The goal is to explore existing fault-tolerant and low-power AUs which are suitable for implementation in biomedical implants. A second objective is to study our own idea for a resource-constrained AU, based on graceful degradation: the so-called scalable arithmetic unit (ScAU). When an error occurs, the ScAU is able to proceed with the computational work, but no longer at the normal throughput: instead of single-cycle we downgrade to double-cycle operations. The design of our ScAU as well as several reference designs are all implemented in VHDL, synthesized and analyzed using Synopsys Design Compiler/PrimeTime and ModelSim. A major part of this thesis is dedicated to fault-tolerant design. An extensive study among common and less frequently employed error-detection schemes is performed. Finally, an error-detection scheme is chosen, applied to the ScAU, as well as to the reference designs for providing fair comparisons. A simple error-correction scheme is implemented as well. The fault-tolerant ScAU proves to have some very interesting advantages over the current state of the art. The fault-tolerant ScAU saves 17% of area, with a speedup of 12% for a 7.3% increase in power consumption, compared to the conventional technique with the lowest costs. Because of these savings, the power-delay-area product reduces by almost 21%. Under specific circumstances, our fault-tolerant ScAU is even capable of saving both area and power.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

# Exploring suitable adder designs for biomedical implants

## A gracefully-degradable, fault-tolerant, and highly resource-constrained adder for SiMS

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Danny P. Riemens
born in Middelburg, The Netherlands

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

# Exploring suitable adder designs for biomedical implants

by Danny P. Riemens

**Abstract**

**Modern** applications demand extremely low power budgets in computer architectures for battery-operated devices. In the particular case of implantable devices —the main focus of this thesis— the system must have a long life span and batteries may not be possible or easy to recharge. In addition to power, chip area is also of major concern in this specific scenario. Since implantable devices are sometimes placed at locations inside the body where limited space is available, the implant must be as small as possible. The vast amount of volume of an implant is typically occupied by the battery and its electrodes, so the affordable chip area is very limited. Another reason why we want very small processor cores, is because this approach leaves more space for cache memory and it statistically reduces the chance of hardware failures. In this thesis we focus on the arithmetic unit (AU) of such a core, which is typically the adder/subtracter. The goal is to explore existing fault-tolerant and low-power AUs which are suitable for implementation in biomedical implants. A second objective is to study our own idea for a resource-constrained AU, based on graceful degradation: the so-called scalable arithmetic unit (ScAU). When an error occurs, the ScAU is able to proceed with the computational work, but no longer at the normal throughput: instead of single-cycle we downgrade to double-cycle operations. The design of our ScAU as well as several reference designs are all implemented in VHDL, synthesized and analyzed using Synopsys Design Compiler/PrimeTime and ModelSim. A major part of this thesis is dedicated to fault-tolerant design. An extensive study among common and less frequently employed error-detection schemes is performed. Finally, an error-detection scheme is chosen, applied to the ScAU, as well as to the reference designs for providing fair comparisons. A simple error-correction scheme is implemented as well. The fault-tolerant ScAU proves to have some very interesting advantages over the current state of the art. The fault-tolerant ScAU saves 17% of area, with a speedup of 12% for a 7.3% increase in power consumption, compared to the conventional technique with the lowest costs. Because of these savings, the power-delay-area product reduces by almost 21%. Under specific circumstances, our fault-tolerant ScAU is even capable of saving both area and power.

**Laboratory** : Computer Engineering
**Codenumber** : CE-MS-2010-31

**Committee Members** :

| | |
|---|---|
| **Advisor:** | Christos Strydis, CE, TU Delft |
| **Advisor:** | Georgi N. Gaydadjiev, CE, TU Delft |
| **Member:** | J. Stephan S.M. Wong, CE, TU Delft |
| **Member:** | Wouter A. Serdijn, ME, TU Delft |

*In memory of my beloved grandmother, Jannetje Riemens-Brik (1919-2002)*

# Contents

# List of Figures

# List of Tables

# Acknowledgements

First of all, I would like to thank my advisors, Christos Strydis and Georgi Gaydadjiev, for all that they helped me with throughout this thesis work. In particular I thank Christos: he has been very inspiring and motivating and I enjoyed our time puzzling out complex issues together. But above all, he is one of the nicest persons I ever had the opportunity to work with. All the way, both Christos and Georgi have been very understanding regarding my limitations as a result of my medical condition. I am very grateful for that. Not everyone would have been so understanding. Further, I thank Ph.D. student Daniele Ludovici. Daniele was of great help to get along with the Synopsys tools.

Special thanks to my mother Johanna. Her support was in particular valuable at those moments I was not so convinced in my own capabilities. Also special thanks to my uncle Adriaan who, together with my mother, provided the necessary financial aid to support my study. And finally, I deeply thank my grandmother for all the love she has given to me. It was a true blessing to have you as my grandmother. Your love and your faith in me, has always been and will always be a driving force in my life.

Danny P. Riemens
Middelburg
November 2, 2010

# 1 Introduction

## 1.1 Background

Nowadays, numerous electronic devices are battery-powered. Cell phones, MP3-players, and notebooks, are probably the most prominent examples. Although we welcome new advanced technologies and functionalities, we are very reluctant when it comes to sacrificing battery time. There has been done a lot of research in the field of low-power design, and designers are currently able to utilize energy much more efficiently than they did in the past. In addition, a lot of progress has been made in the design of batteries with enhanced energy capacities. Both these accomplishments have been crucial in the design of, for example, modern smart phones (such as the Apple iPhone) with relatively large TFT screens and numerous functions, very long stand-by times and only very small sized battery packs.

In this thesis we aim at a specific, very unique group of electronic devices: biomedical implants. These devices are utilized inside the human body to support or replace failing or distorted physiological functions of patients with a certain illness. The most common and well-known implants are pacemakers and implantable cardioverter defibrillators (ICDs). Pacemakers generate electric pulses, which are required to let the heart contract, in cases when one of the natural pacemakers (the sinoatrial node and atrioventricular node) does not fire pulses at all, fires them too slowly, or irregularly. ICDs are utilized for patients with certain types of heart arrhythmias (tachycardia) and high risk of ventricular fibrillation (which always leads to sudden death). ICDs deliver electrical shocks to the heart in order to bring the abnormal rhythm back to the normal rhythm. There are indeed more applications for implantable devices, such as cochlear implants, implantable neurostimulators (to treat neurological disorders as seizures and epileptic attacks), implantable drug-infusion pumps, and spinal-fusion stimulators (to enhance and facilitate the rate of bone healing) [1]. It can be expected that biomedical implants will be employed for a growing number of medical conditions in the future, such as for example, eye implants [2]. Further advances in implantable symbiotic brain-machine interfaces (BMIs) can be expected. For example motor BMIs, which will form the key in substituting lost motor functions. For patients who miss a limb, a neuroprosthetic device (e.g. a robotic arm) can then be employed, utilizing a BMI implanted in the brain to extract information from and deliver feedback to the brain in order to control the robotic arm. Another possibility is cognitive BMIs, which would enable implants to repair broken connections in the brain, e.g. between the long-term memory, the hippocampus, and the short-term memory [3, 4].

From the energy point of view, the most important differences between implants and all other electronic devices are that batteries should have a very long life span (up to ten years) and they are impossible or cumbersome to recharge. Patients with an implant will

visit their physician on a regular basis. During such visits the physician can read out information from the implant via a dedicated wireless communication link. For example, the physician is able to see how often the implant had to intervene, but also what the status of its battery is. Until recently, when the battery level of an implant became low, the the entire implant had to be replaced, requiring an invasive surgical procedure. Nowadays, some implants do have replaceable batteries, which allows replacing the battery without removing the entire implant out of the body. However, surgery is still required, although less invasive. This makes immediately clear why ultra-low-power design is of even greater importance in implants than anywhere else.

Another major difference between consumer electronics and implantable devices is the need of the latter for high system reliability. Human lives literally depend on it. The same holds true for, e.g., avionics and military defense systems, but these systems are typically located in power-rich environments, in contrast with biomedical implants. When power consumption is not an issue, it is much easier to build highly reliable, fail-safe systems, sometimes even with complete redundant backup systems. In case of implants, however, this is very difficult since every additional circuit utilized to increase the fault tolerance will contribute to the power consumption and will deplete the battery faster.

## 1.2 The SiMS Architecture

The SiMS project (Smart implantable Medical Systems, [5]) is a project targeting the design of highly reliable, ultra-low-power, and of miniature physical dimensions biomedical implants. SiMS is a truly multidisciplinary project, where many academic, industrial, and medical specialists work on different parts of the implant, such as the micro-architecture, the software (for example the compiler), the electrodes and battery, sensors and actuators, analogue electronics, (wireless) data communication etc. The main intention of the project is not to produce a single, specific implant, but to provide the building blocks and the methodology for a wide range of implantable systems. Basically, it provides biomedical researchers a toolbox of ready-to-use components, which enables them to design an implant for a specific application, without having to build all the sub-systems from scratch. This project will lead to significant design-time savings and higher design quality. It is also expected that the SiMS approach will shorten the time needed for medical approvals of new implantable devices.

At the computer engineering laboratory, a special minimalistic computer architecture for implantable microelectronic devices is currently being designed. The micro-architecture will have a data path of 16 bits wide. Since the implant is battery-powered and requires a relatively long lifetime, the architecture needs to be designed for ultra-low-power consumption. The system frequency has not been precisely determined yet. Most current architectures in implants employ clock frequencies up to 10 Mhz [6], but future developments in the biomedical field will probably demand higher clock frequencies, in order to enable higher throughputs. However, we need to be aware of the fact that power consumption grows linearly with the clock frequency, so the system frequency is envisioned to be limited (to approximately 20 MHz).

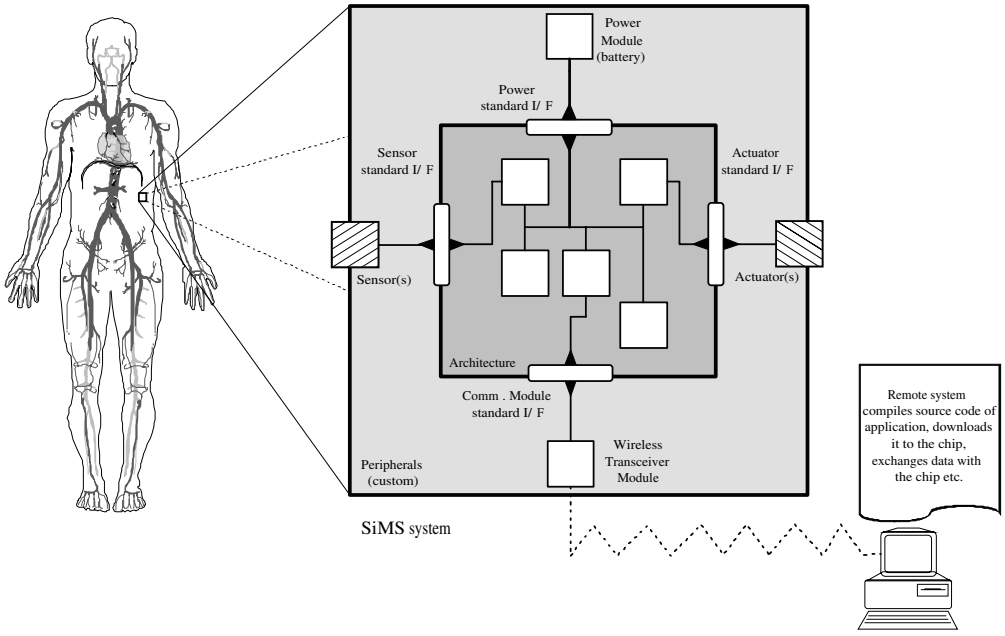


Figure 1.1: The SiMS concept [5]

## 1.3 Thesis motivation

Currently, the SiMS processor architecture is being specified and designed. No special ALU has been implemented yet. The ALU is the core processing component in any computer architecture and is heavily utilized, therefore significantly contributing to the total power consumption. In this thesis we focus on the **arithmetic unit** (addition and subtraction; including zero, overflow, and sign detection) only. Arithmetic operations are more difficult to perform than logical operations, and also the error detection of arithmetic operations is more complex. This reflects in the power consumption and area requirements of the arithmetic unit, which are significantly larger than that of the logic unit. Therefore, the design of the arithmetic unit is the most interesting from the low-power and low-area point of view, as well as from the reliability perspective.

The arithmetic unit that is eventually implemented in the architecture should be designed for high reliability, since it is a vital component in the architecture. On the other hand, increasing the reliability will always require additional (redundant) hardware, therefore increasing the area and power costs. As mentioned previously, the power budget is extremely small and increasing power will lead to a shorter life time of the battery and thus a shorter life time of the implant itself. The area budget is very tight as well. Since the physical dimensions of the implant are tightly bounded (implants tend to become smaller and smaller, enabling them to be implanted in parts inside the body where there is not much space), and the vast amount of area within the implant is used for the battery pack and the electrodes, the available chip area is very limited. Another reason to keep

the arithmetic unit small is: the smaller the circuit, the chance of hardware failures statistically reduces as well. In addition, we want to save as much "free space" on the chip, so it can be employed for other core structures which have been shown to benefit implant operation, such as caches [7]. Therefore, the design of a reliable arithmetic unit under these conditions is challenging.

At this moment, the SiMS architecture is still in development and we do not have any precise numbers about the maximum power consumption or acceptable chip area of the arithmetic unit. What we do know is that the slightest increase in power has always immediate effect on the battery lifetime. A slight increase in area does not necessarily have immediate negative consequences. Through the course of this thesis we will introduce a number of custom joint metrics, where different weights are assigned to power, in order to classify the different designs for different needs. The idea is to assign a higher weight to power, than we assign to area. What we also know is that in the SiMS-targeted biomedical implants we do not require high performances. Throughout the design process we will focus on low to medium performance.

One very common and effective technique for increasing reliability, is hardware replication. When one of the adders fails inside the arithmetic unit (we assume self-checking adders here, i.e. adders which are capable of detecting errors), the remaining functioning adder can proceed with the computational work. Clearly, replicating hardware blocks increases both power and area costs drastically. The idea was raised to explore alternative approaches. Instead of **replicating** adders, we want to implement only one **single** adder which is **scalable** in size. The basic idea is to shut down only a part of the adder, the part where the failure has occurred, and to continue with the computational work by employing the remaining part(s) of the adder. Preferably we do not want to compromise precision, so we will have to sacrifice latency. This approach would be an example of *graceful degradation*. We would like to know if this approach is viable for our purpose, and if it is advantageous over the common hardware replication technique, especially in terms of power consumption and area.

## 1.4 Thesis goals

A prerequisite for this particular thesis work is a good understanding of low-power design. Therefore, a comprehensive literature study within this field is mandatory. Another important objective of this thesis work is to set up an ASIC design tool flow and get familiar with all the different tools, from which the most important one is Synopsys Design Compiler.

A thorough study among plain adders should be performed to analyze their suitability for implementation in low-power, resource-constrained architectures. Apart from these general constraints, we also need to know which adders are suitable for implementation in a scalable structure. We will implement, synthesize, and analyze the results of these plain adders to make a comparison with studies from the literature. Existing literature studies all report their results based on outdated technologies, and we want to study if the power, area, and delay trends hold for relatively new technology nodes. This study is interesting for the current work within the SiMS framework, but also for other fields where low-power embedded design is applicable. We will also study the effects of

glitching in order to see if this phenomenon should have any influence on the chosen adder. Finally, we present the most suitable candidates.

As a next step, a new scalable arithmetic unit needs to be designed, implemented, and synthesized. Together with some reference designs, the results will be analyzed and compared. We will report if the scalable approach is viable or not. If it is, we will try to find the optimal design point of the scalable arithmetic unit, by studying its behavior for different frequencies and different technologies. If it is not, we will continue with the best conventional solution.

Then fault tolerance comes in the picture. A thorough literature study in the field of fault-tolerant design is required. Various error-detection schemes will be explored, focusing on fault coverage and costs (in terms of power consumption and area requirements). To determine the costs, some schemes might have to be implemented, synthesized, and analyzed in order to obtain exact numbers if the literature does not provide conclusive information. Also, the desired/acceptable fault coverage of the arithmetic unit has to be determined. Based on this study, the best error-detection scheme will be picked. Further, the implementation of a specific error-correction scheme has to be devised. The error detection and correction scheme will have to be implemented in our arithmetic unit. A number of standard, fault-tolerant arithmetic units will be implemented as well, which will serve as reference designs. Finally, our fault-tolerant arithmetic unit will be compared with these reference designs and conclusions will be drawn.

## 1.5 Thesis organization

Chapter 2 begins with an explanation of the fundamentals about power dissipation in CMOS technology. Subsequently, a survey through various common low-power/power-aware design methodologies and low-power optimizations is presented. This chapter is the result of a thorough literature study and provides the basic knowledge for the design of the scalable arithmetic unit.

Chapter 3 explains which tools are employed for the design, simulation, synthesis, and analysis of the scalable arithmetic unit (and other designs that have been implemented throughout this thesis work). It is explained how the tools are utilized, and in which setup.

In chapter 4, a number of well-known adder structures are investigated and compared to make a decision about which adder type is the most suitable for utilization in the scalable arithmetic unit and the SiMS architecture. Apart from these specific goals, we have generalized the adder study to provide up-to-date results regarding delay, area, and power trends of adders in modern technology.

Chapter 5 describes the design of the scalable arithmetic unit, including synthesis results and various optimizations. Here, some preliminary conclusions are drawn about the viability of our scalable design. Also, the design is studied for multiple system frequencies, technologies, adder types, and word widths to find an optimal design point. Further, some alternative applications of the scalable arithmetic unit are discussed. Finally, a number of approaches are presented to solve the problems which appear when the scalable arithmetic unit is gracefully degraded when an error has occurred.

Chapter 6 is dedicated to fault-tolerant design. The chapter starts with an extensive theoretical part which discusses the fundamentals about fault-tolerant design in general and error detection/correction in adders in particular. It is investigated which error detection/correction methods are the most suitable for implementation in the scalable arithmetic unit. We will focus on the error coverage and, obviously, on the power and area costs. The most suitable technique is implemented in the scalable arithmetic unit and the synthesis results are presented, along with the results of a number of reference designs, in order to make a good comparison.

Finally, chapter 7 presents the conclusions of the thesis work and lists a number of directions for future work.

# Designing for low-power consumption

# 2

## 2.1 Introduction

If we want a design for low-power consumption to be successful, it is important to have a thorough understanding of the sources of power dissipation, the factors that affect them, and the methodologies and techniques that are available to achieve optimal results. Therefore, this thesis starts with a literature study in low-power and power-aware design. We present —we believe— the most important low-power methodologies and power optimization techniques available. Low-power design can be applied on various different levels, such as the architectural level, the gate level, and the technology level. Apart from that, also a number of alternative logic-design styles are presented to report on their characteristics regarding power consumption. This chapter could as well be utilized by others as a quick study in the field of low-power/power-aware design.

## 2.2 Low-Power Design vs. Power-Aware Design

Nowadays, **low-power design** is a term that has become familiar to probably everyone in the engineering field, because of the simple fact that we want to do the same amount of work (or even more) with less power. Another design methodology exists, which employs low-power design techniques, called **power-aware design**. These methodologies are, however, not the same. Power-aware design is sometimes confused with low-power design, but there is an elementary difference in the point of view.

Low-power design is a design methodology which focuses on minimizing power of a certain circuit. Since low-power design techniques may severely affect the performance of a circuit (as will be explained later), a minimal performance constraint can be set. This means that the power consumption will be minimized without violating this performance constraint. Power-aware systems are typically systems that have limited power budgets but provide a respectable performance as well. When these circuits are designed, low-power consumption is of importance as well, but the point of view is different. In power-aware design the performance is maximized *subject to a power budget* [8].

Thus, to summarize, in low-power design we determine the absolute minimum performance we require and minimize power without violating this constraint, and in power-aware design we first determine our maximum power budget and we try to reach the optimal performance within this budget.

## 2.3 Sources of Power Consumption

There are a number of sources of power consumption in CMOS, which can be subdivided into **static** and **dynamic** power dissipation. Dynamic power dissipation is primarily caused by the switching of the CMOS devices (MOSFETs) when logic values are changed (known as **capacitive power** or **switching power**). The amount of power that is dissipated is directly related to the switching activity (which is the number of logic transitions per clock cycle in the entire circuit), the clock frequency, the supply voltage, and the capacitive load the circuit drives [8]. Another source of dynamic power dissipation is **short-circuit power**. CMOS is comprised of both PMOS and NMOS devices. During a logic transition, the PMOS and NMOS devices are simultaneously turned on for a very short period of time, allowing a short-circuit current to run from $V_{DD}$ to ground [9, 10]. This behavior is inherent to CMOS switching. Static power, which is a constant factor and has nothing to do with the switching activity, is caused by **leakage power**. In an ideal situation, a static CMOS circuit (i.e. one that does not switch) does not consume any power because there is no direct path from $V_{DD}$ to ground. In real life there will always be leakage currents, since CMOS devices are not perfect switches. The total power dissipation can be described by the following formula:

$$P_{total} = \alpha(C_L \cdot {V_{DD}}^2 \cdot f_{clk}) + I_{sc} \cdot V_{DD} + I_{leak} \cdot V_{DD} \quad (1)$$

The Greek letter $\alpha$ represents the **switching activity** in the circuit, expressed in a value between 0 (no switching activity at all) to 1 (maximum switching activity). $C_L$ is the capacitive load, driven by the circuit. As can been seen in the formula, the dynamic power consumption depends on $V_{DD}$ square, which makes the supply voltage a very important factor in low-power design, as will be explained later. $I_{sc}$ and $I_{leak}$ represent the total short circuit and leakage current, respectively. It is important to realize that that $I_{sc}$ is is a variable, while $I_{leak}$ is not. $I_{sc}$ depends on the charge carried by the short-circuit per transition, the cycle time, and the total number of transitions [11]:

$$I_{sc} = Q_{sc} \cdot f \cdot \alpha \quad (2)$$

A better way to represent the formula would be like this:

$$P_{total} = \alpha(C_L \cdot {V_{DD}}^2 \cdot f_{clk} + Q_{sc} \cdot f_{clk} \cdot V_{DD}) + I_{leak} \cdot V_{DD} \quad (3)$$

It is a misunderstanding that reducing power consumption will always lead to a reduction in energy as well. When we refer to power, we refer to the **momentary** electric energy that is dissipated, measured in Watts. Energy is measured in Joules, or Watts per second. Thus, the energy consumption depends on the power consumption and the time it takes to perform a task. For example, if we have two circuits A and B, and $P_A = 2P_B$, and $2D_A = D_B$ (where 'D' refers to the delay of the circuits), then $E_A = E_B$. Thus, even though the power consumption of circuit B is twice as low as circuit A's, no energy is saved. When we take another look at formula (3) in section 2.3, reducing $\alpha$, $V_{DD}$, and $C_L$ will always reduce power and energy consumption. Lowering $f_{clk}$ reduces only power, not energy [12] (assuming that $f_{clk}$ is not higher than $1/T_{critical\_path}$)[1].

[1]The system frequency is based on the demands of the system as a whole. That does not necessarily mean that every subsystem requires to operate at a frequency as high as the system frequency, e.g., when the subsystem does not reside in the critical path.

## 2.4 Basic Low-Power Design Methodologies

The following methodologies are the most powerful ones and applicable to virtually every system. They include static voltage scaling, frequency scaling, various other kinds of voltage scaling (sometimes combined with frequency scaling), clock gating and power gating. Finally, a section is dedicated to technology scaling.

### 2.4.1 Static voltage scaling

One way to decrease the power consumption significantly, is to decrease the supply voltage. As mentioned in section 2.3, dynamic power consumption depends quadratically on $V_{DD}$. **Voltage scaling** is therefore the most effective method to limit the power consumption. However, when $V_{DD}$ is lowered, it comes at a price: the delay of the logic increases. In systems where we desire a high throughput, and ask for the maximum performance of the technology being utilized, voltage scaling is not an option. If $V_{DD}$ would be lowered, we would not be able to meet the performance requirements. In many situations however, we do not ask for the maximum performance and we can safely lower $V_{DD}$ in order to save power. Even though delay is increasing, the *power-delay product is improving* when $V_{DD}$ is decreased, since power decreases quadratically while delay increases less fast. Delay scales with:

$$\frac{V_{DD}}{(V_{DD} - V_{TH})^2} \quad (4)$$

A graph of this function is depicted as an example in figure 2.1, where the **threshold voltage ($V_{TH}$)** is 1 Volt. The threshold voltage is the minimal voltage required between the gate and source in order to create a conductive channel between the drain and source of the MOSFET. Note that when $V_{DD}$ is lowered to such extent that it reaches the threshold voltage of the CMOS devices, the delay of the logic increases radically [13, 10]. In addition, the robustness of the transistors against noise is severely lowered and proper circuit behavior is compromised [12]. According to Rabaey [9], voltage scaling is only advantageous when $V_{DD} \geq 2 \cdot V_{TH}$, but a minimal value of $4V_{TH}$ is recommended. So, there is a limit in voltage scaling. It can provide massive power savings, but we have to be careful *not to jeopardize the reliability of the logic.*

There is no such thing as a maximum $V_{DD}$ limit, since the normal supply voltage is in fact the maximum $V_{DD}$. However, technologies are becoming smaller and smaller, and in an ideal situation the voltages, electric fields, and linear dimensions remain constant with the scaling factor. However, in reality this is not always the case. Sometimes constant electric field scaling is sacrificed by disproportional scaling of the supply voltage in order to achieve higher performances [14, 15]. The channels in modern technologies are already very short and the electric fields are very high. Careless scaling leads to even higher electric fields. Very high electric fields cause so-called **hot-carriers**: electrons gain so much kinetic energy that they can get injected and trapped in areas of the MOSFET where they are not supposed to be. These conditions do not contribute to the reliability of the technology since it leads to **early transistor aging** (which also negatively affects the delay of the devices). Proper scaling is therefore very important for the reliability of the technology. Obviously, also in carefully scaled technology device aging is inevitable.

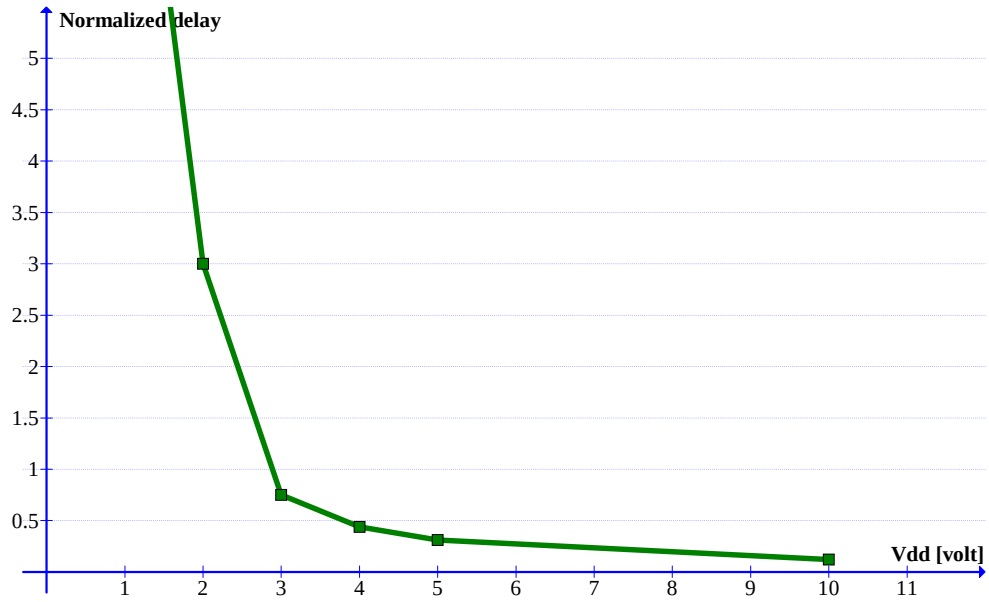


Figure 2.1: Graph displaying delay versus $V_{DD}$

However, in the next paragraph will be discussed how easily this process can be slowed down.

Operating at the normal supply voltage theoretically assures correct operation for a certain period of time, but if we require high reliability and need to depend on the technology for long periods of time (such as in biomedical implants), it is not only wise to reduce $V_{DD}$ for power-saving purposes but also for *increasing the reliability*. A small reduction in the supply voltage can already substantially diminish the device degradation over time, since device aging is nearly exponentially dependent on $V_{DD}$ [15]. According to Kakumu et al. [14], an optimum supply voltage exists for each technology which takes both performance and reliability into consideration. According to [15], a number of other factors can be altered to slow down the aging process, but this theory goes too deep to discuss in this survey. In conclusion we can say that even a small reduction in the supply voltage will always increase the reliability of the technology.

### 2.4.2 Frequency scaling

Apart from $V_{DD}$, there is another variable in the equation of section 2.3 that intuitively suggests possibilities for reducing power: frequency. Of course, the clock frequency is bound by the desired throughput of the system. However, a system does rarely operate at its maximum throughput all the time. Often, the desired throughput is much lower than its maximum performance. Then, it is possible to lower the operating frequency in order to save power (and thus also energy), which is called **frequency scaling**. Sometimes voltage scaling and frequency scaling are employed simultaneously, such as in cell phones, when they are in stand-by mode.

### 2.4.3 Multi-$V_{DD}$ and CVS

A system is often comprised of various subsystems or components. If one or multiple components require maximum performance and cannot permit voltage scaling, it does not mean that the entire system does not apply for voltage scaling. It is possible to subdivide the system in blocks, having their own different supply voltage: high-$V_{DD}$ (normal $V_{DD}$) or low-$V_{DD}$. These supply voltages are, however, fixed. Therefore this technique is still referred to as static voltage scaling. There appears, however, a problem when Multi-$V_{DD}$ is employed. Signals cross from low-$V_{DD}$ to high-$V_{DD}$ blocks and vice versa. Crossing from high-$V_{DD}$ to low-$V_{DD}$ does not result in any problems, but signals coming from a low-$V_{DD}$ block driving logic on a high-$V_{DD}$ block results in problems. A low-$V_{DD}$ output signal driving a high-$V_{DD}$ circuit may lead to *problems with cutting off the PMOS transistors* (as PMOS transistors may get stuck in triode mode)[2], which causes a static current flowing from $V_{DD}$ to ground in the high-$V_{DD}$ circuit [17]. And even when low-$V_{DD}$ is high enough to cut off the PMOS transistors, there will be an *increase in the rise and fall times* at the receiving inputs, leading to *higher switching currents* and *slower transition times*. Slower transition times will ultimately cause the short circuit current to last longer than necessary. The solution for these problems are **level shifters** (a.k.a. level converters). These are devices that can be placed in between the signals crossing from a low-$V_{DD}$ to a high-$V_{DD}$ block. The level shifter will lift the low-voltage signal to the level that is appropriate for the high-$V_{DD}$ powered block [12]. Note that these components have a certain cost, and when many level shifters are implemented, they may contribute significantly to the total area and dynamic power consumption. Therefore, the total number of blocks should be limited, since too many level shifters may cancel the power savings.

Another approach is CVS: **Clustered Voltage Scaling**. This technique avoids the implementation of many level shifters (and thus power) by clustering all critical and non-critical paths in *only two* separate clusters, one powered by low-$V_{DD}$, the other by high-$V_{DD}$. Before CVS is applied, a lot of level shifters can reside along a path from an input to an output of a circuit if a high-$V_{DD}$ circuit is connected to a low-$V_{DD}$ circuit, which is in turn connected to a high-$V_{DD}$ circuit, etc. The idea of CVS is that every combinational logic circuit can be rearranged in such a way that the structure is as follows: *primary inputs* $\rightarrow$ *high-*$V_{DD}$ *cells* $\rightarrow$ *low-*$V_{DD}$ *cells* $\rightarrow$ *level shifters* $\rightarrow$ *primary outputs*. On top of that, the level shifters can be combined with the output registers which saves both area and power. Level shifters in between the clusters are not required since high-$V_{DD}$ outputs drive low-$V_{DD}$ cells. An algorithm based on heuristics can be utilized to rearrange the circuit into clusters [17].

[2]Triode mode means that the MOSFET is not fully open or fully closed, but stuck somewhere in between. The MOSFET now functions basically as a variable resistor. Triode mode occurs when $V_{GS} > V_{TH}$ and $V_{DS} < (V_{GS} - V_{TH})$ The actual resistance is dependent on the gate voltage relative to both the source and drain voltages, which de facto determines the size of the static current. The conditions we have to meet in order to have the P-MOSFET fully closed are: $V_{GS} > V_{TH}$ and $V_{DS} > (V_{GS} - V_{TH})$. Note: G=gate, S=source, D=drain. For more background information, refer to [16].

### 2.4.4 Dynamic voltage scaling

Instead of a fixed supply voltage during circuit operation it is also possible to *dynamically adjust* the supply voltage based on the *current required performance* of the circuit. The required performance is often not always the same. When the required performance of the circuit is momentarily reduced, we can afford lower supply voltages. This is called **dynamic voltage scaling** (DVS). A **feedback control** is required to control the voltage based on the required performance. A **replica** of the critical path is implemented together with a **DC-DC converter**. Note that a critical path replica is not an exact functional copy of the critical path, but a virtual copy by creating a chain of gates that equals (or slightly exceeds) the delay of the real critical path [18]. The delay of the circuit is continuously monitored and the lowest possible supply voltage is generated by the DC-DC converter. Ideally, we want access to an infinite amount of supply voltages, such that the optimal voltage can be chosen. In reality, this is impossible, and we have to work with a large, but limited number of voltages. Apart from lowering the supply voltage, it is also possible to lower the clock frequency. A reduction in $V_{DD}$ will always increase the delay to some extent, but if the cycle time is still much higher than the delay of the circuit, energy is wasted. Therefore **dynamic voltage and frequency scaling** (DVFS) can be employed to save energy to the maximum.

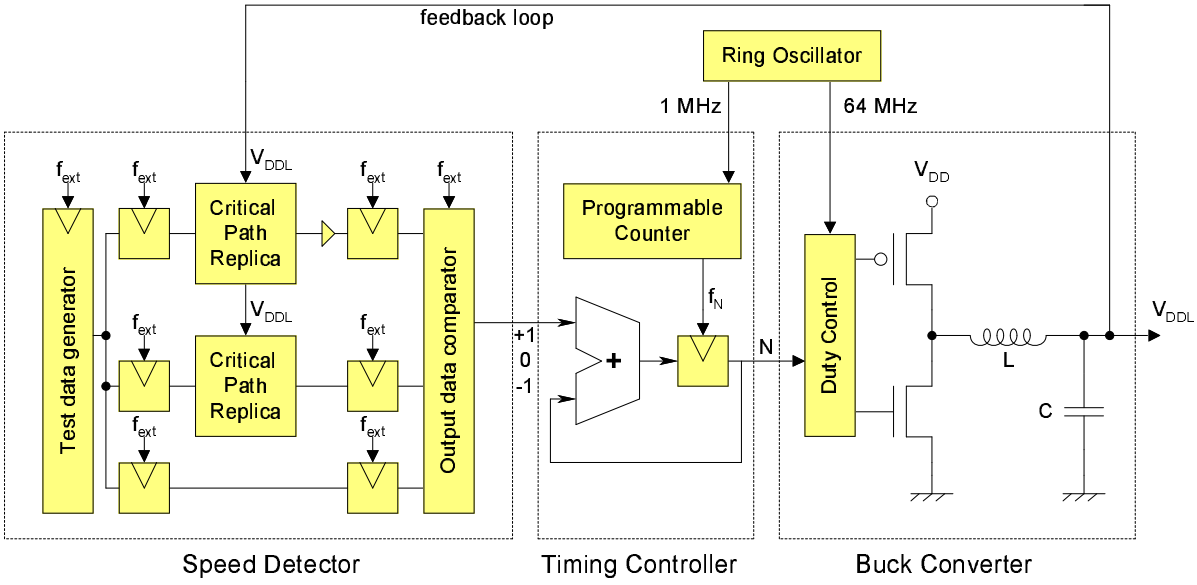


Figure 2.2: Variable supply voltage scheme [19]

Kuroda et al. [19] proposed a variable supply voltage scheme, divided into three elementary parts: the **speed detector**, the **timing controller** and the **buck converter**, as depicted in figure 2.2. The timing controller contains two critical path replicas (CPR and CPR+), both powered by the reduced supply voltage $V_{DDL}$. After one of the replicas a small delay element (a couple of inverters) is placed (CPR+), to increase the critical path by a slight amount. Further, a simple wire is utilized as a reference (path called REF). No matter how low $V_{DDL}$ is, the output register will always

be able to clock in the value of the input register, since there is virtually no delay in the REF path. If $V_{DDL}$ is too low, only the REF path is fast enough. The output data comparator will detect this and output '+1', indicating that $V_{DDL}$ must be increased. If $V_{DDL}$ is too high, all paths (REF, CPR, and CPR+) are fast enough, and the output is '-1', indicating that $V_{DDL}$ must be decreased. If the paths REF and CPR are fast enough, but CPR+ is not, then we have obtained the optimum supply voltage. The output data comparator outputs a '0', which makes sure that the value is maintained. All registers in the speed detector are clocked by the external system clock. The timing controller contains a counter, clocked by a **ring oscillator**. In this case the counter operates at 1 MHz, meaning that every 1 $\mu$s the counter can be increased/decreased. The counter's output is an integer N from 0 to 64, which is an indicator for the value of the supply voltage $V_{DDL}$. The buck converter is comprised of a **duty controller** (clocked by a 64 MHz ring oscillator), a **CMOS inverter**, and a **low-pass LC-filter**. The buck converter is the part that creates the $V_{DDL}$ from $V_{DD}$. The idea is as follows. Within every consecutive 1 $\mu$s time span, the duty control can turn $V_{DD}$ on for a period of time X (PMOS is on) and off (NMOS is on) for a period of time Y. The ratio X/Y is determined by the integer N. Since the duty controller runs at 64 MHz (which is 64 times faster than the timing controller), and N can represent 64 numbers, we are able to create 64 different values of $V_{DDL}$. For example, if N=32, then $V_{DD}$ is turned on for 0.5 $\mu$s and for the remaining 0.5 $\mu$s the value is zero. Then the **average** value of $V_{DDL}$ is $0.5 \cdot V_{DD}$. The low-pass filter[3] is placed off-chip. Finally, there is the feedback loop back to the speed detector. If $V_{DD}$ is 1 Volt, which is common in 90nm CMOS, this variable supply voltage scheme can provide a **resolution** of 15.6 mV (meaning that $V_{DDL}$ can be varied in steps as small as 15.6 mV). Obviously, the larger the frequency of the buck converter (and the range of the counter), the larger the resolution, and the closer $V_{DDL}$ will be to the optimal value. It appears that the *external* frequency $f_{ext}$ assumes some predefined values (based on different performance requirements), such that $V_{DDL}$ can be fine-tuned for the specific frequency. The reason why this scheme is presented in such close detail is mainly to provide a deeper insight in how dynamic voltage scaling exactly works, but also to show that a significant amount of overhead is required for this technique.

**Multi-level voltage scaling** is a form of dynamic voltage scaling and essentially an extension of static voltage scaling. Based on the required performance, the supply voltage can be scaled between a small number of fixed and discrete voltage levels [13]. The advantage of Multi-level voltage scaling is that it is a significantly less expensive power scheme than DVS with a virtually infinite number of supply voltages.

The **Dual Variable Supply-voltage scheme** (Dual-VS) is a combination between DVS and clustered voltage scaling (CVS). First, the circuit is clustered into a high-$V_{DD}$ and a low-$V_{DD}$ cluster. Both supply voltages are variable, and —in contrast with multi-level voltage scaling— non-discrete. The minimal voltage is controlled by the feedback loops for both the high and low supply voltages [18].

[3]We are referring to the LC-filter in figure 2.2. The purpose of this filter is to block the high frequency supply voltage (alternating between zero and $V_{DD}$) and pass the average value instead.

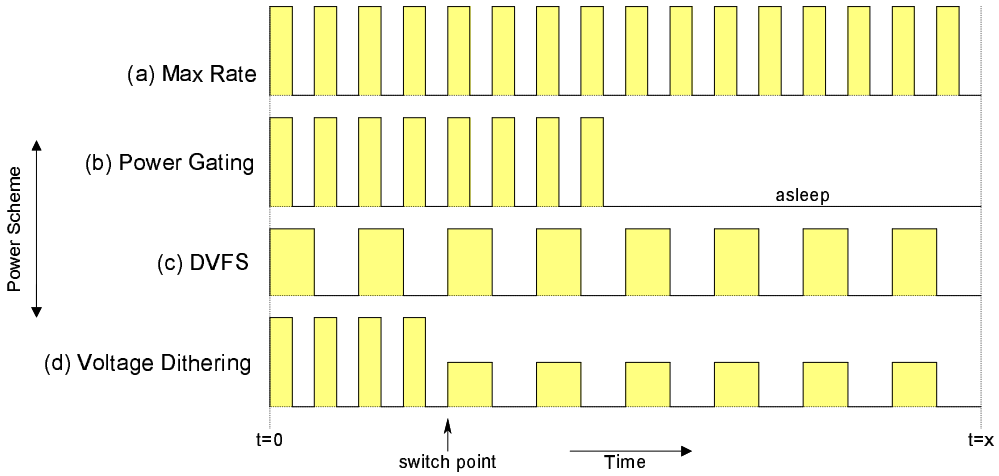


Figure 2.3: Voltage dithering compared to other approaches [20]

### 2.4.5 Voltage dithering

Assume that we, for example, require a rate (normalized frequency) of 0.5 for a certain period of time. If we do not implement a technique which allows us to dynamically adjust parameters as voltage and frequency, and the circuit always operates at the maximum rate of 1, a dramatic amount of the total dissipated energy is wasted energy. This situation is depicted in figure 2.3(a). If power gating is implemented, the circuit can operate at maximum rate and go to sleep when the work is done (2.3(b)). The energy savings are significant (theoretically 50%), but power gating has a number of serious drawbacks, as will be explained in section 2.4.7. When utilizing DVFS, $f_{clk}$ is lowered (in this case by 50%) and since a lower frequency is required also the supply voltage can be decreased (2.3(c)). However, also DVFS has an important drawback: access to a vast (ideally infinite) amount of different supply voltages requires a significant amount of hardware overhead. A solution to this problem has been presented by multi-level voltage scaling, utilizing a small number of discrete voltages, but this introduces a new problem. With not that many discrete voltages, the selected voltage will probably not be optimal, which results in a higher energy consumption than strictly necessary. A solution to both problems is **Voltage Dithering**. In voltage dithering, only *two* discrete voltage/frequency pairs are utilized: A high-frequency, high-voltage pair and a low-frequency, low-voltage pair. All possible rates can be achieved by utilizing *both* these two pairs in a time span $t(y) - t(0)$. The rate can be determined by the *switch point*: the point '$x$' in time ($t(y - x)$) where the high-frequency, high voltage pair is alternated with the low-frequency, low-voltage one. An example is depicted in 2.3(d) [20]. The costs for voltage dithering is lower than for DVS. This can be explained by the fact that only two discrete supply voltages (and frequencies) are required, which makes the power scheme less complex.

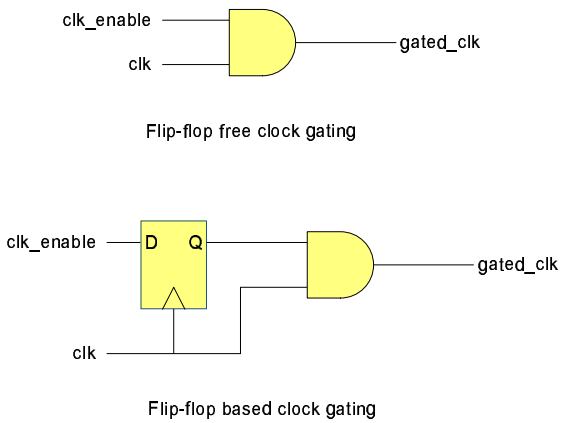


Figure 2.4: The clock gating mechanism

### 2.4.6 Clock gating

In the previous sections we have referred to (sub)systems that do not always operate at their maximum performance. It is also possible that parts of a system are *idle* for a period of time: then no useful computational work is performed. Still, there is power consumed. A subsystem being idle does not necessarily mean that the subsystem is not performing any computations. It only means that the results are *not being utilized*. This is possible when the subsystem is still fed with data, but the result is discarded, because it is not needed at that moment. But even when the subsystem is not performing any computational work, there is still the *static power dissipation* (leakage power) of the subsystem. What is more, flip-flops dissipate some dynamic power every single clock cycle, *even when the in- and outputs remain the same*. If there are large registers present in these subsystems, this power dissipation can become quite significant. And, finally, there is the power dissipation of the **clock network** in the subsystem. Clock networks are very expensive in terms of power. A major portion of the total power consumption of the system is dissipated in the clock network (mainly in the clock buffers/drivers) [13].

Considering the above, there is a lot of power that can be saved when a subsystem is idle. One way to achieve this goal is to apply **clock gating**. This essentially means that the clock signal of the subsystem is cut off. This will save the power dissipated in flip-flops and the clock network. If the combinational logic in the subsystem is fed by registers at the inputs, the logic will stop switching [21]. It will, however, *not* save the leakage power.

For clock gating an additional signal is required: a clock-enable signal. Clock gating is a very simple approach and can be automatically implemented by a synthesis tool. There are essentially two ways of implementing clock gating: **flip-flop-free** and **flip-flop-based** clock gating. Flip-flop-free clock gating is implemented by a simple AND-gate. This works fine, as long as the clock-enable signal is stable in between the rising edges of the clock. If it is not, additional clock pulses can be generated or the clock gating can be terminated prematurely. A better approach is to insert a flip-flop for the clock-enable signal to avoid these problems. Both methods of clock gating are depicted in figure 2.4. Thus, clock gating requires some additional logic, but the costs are low.

Clock gating is not only applicable to large subsystems, it can as well be applied to simple registers that do not need to be updated every clock cycle [11]. In some occasions, clock gating a few registers is sufficient to disable an entire subsystem. For example, when the input registers of an ALU are clock gated, the entire ALU can be "turned off" (desired in the case when the ALU is computing in vain, since the results are not utilized). However, clock gating a register *smaller than 3 bits is not efficient*, considering the overhead of the clock gating mechanism [13].

### 2.4.7 Power gating

**Power gating** provides basically a solution to the same problem mentioned in section 2.4.6. Power gating has however an important advantage over clock gating: it is capable to *save static power* of idle blocks as well, since it cuts of the power supply instead of the clock signal. In order to do that, blocks need to be placed onto separate "power islands", which can be powered on and off. In reality, **low-leakage PMOS transistors** (called switches) are placed in between every connection to $V_{DD}$ of the block, to create a virtual power network that can be turned on and off. These switches are controlled by power gating controllers. For these low-leakage PMOS transistors often **high-$V_{TH}$ transistors** (the higher $V_{TH}$, the lower the leakage current) are employed. Transistors with low-$V_{TH}$ are suitable for high performance, but not for low-leakage, and vice versa. Therefore, the transistors in the circuit have low threshold voltages and the switches have high voltage thresholds. This is called a **dual-threshold voltage technology** or MTCMOS (multi-threshold CMOS). These high-$V_{TH}$ transistors do, however, cause a problem. Since $V_{DD}$ is low and $V_{TH}$ is relatively high, these transistors will be *slow*. In order to speed them up, we would need to **resize** (upscale) them (see section 2.7.1) [22].

When a block is asleep it costs some time to wake it up again, the same as it costs some time to put a block to sleep. This introduces *additional delays*. Also during wake-up and going to sleep, still some leakage power is dissipated which makes power gating not perfect. The essential criteria for implementing power gating is the total leakage power component and how many and how often blocks are idle. The leakage power highly depends on the technology being utilized and the impact of the leakage power highly depends on the system frequency being utilized. If the leakage component is significant and many blocks are idle for longer periods of time, power gating may be efficient. One should however be aware of the fact that power gating is much more difficult to implement than clock gating and leads to significantly higher costs (mainly because of all the switches that are required). It is also important to realize that power gating is much more **invasive** than clock gating. While clock gating does not affect the functionality of the system, power gating does. It affects inter-block communication and, as mentioned before, adds time delays to safely enter and exit power gated modes [13].

### 2.4.8 Technology scaling

Another way to save energy is to improve and scale the technology. Over the last decades CMOS technology has improved and scaled from 10$\mu$m in the early 1970's to 32nm in 2010. Sizes as small as 11 nm are expected around the year 2015. Ideally, the voltages, electric fields, and linear dimensions remain constant with the scaling factor

$\varphi$ as explained in section 2.4.1. Therefore, the energy savings scale with $\varphi^3$ ($V_{DD}$ and the capacitance of the transistors scales with $\varphi$, where power/energy has a quadratic dependence on $V_{DD}$). In reality it is difficult to scale $V_{TH}$ along with $V_{DD}$ [23]. It will be explained in section 2.7.2 why this is true. In smaller technologies the leakage currents (and thus the leakage power) become a factor of significance (as they grow exponentially) and ultimately, as technologies continue to shrink in the future, we may very soon face a situation where the leakage power will be the dominating factor in the total power consumption at any frequency [22]. However, at this point in time (at least in most cases) smaller technologies still lead to significantly lower energy consumptions.

## 2.5 Methodologies at Architectural Level

Two methodologies at architectural level are presented, namely parallelization and pipelining. Note that in both cases voltage scaling is required as well, and these methodologies are not applicable to every circuit, since it depends on the implementation of the architecture and constraints of the system.

### 2.5.1 Parallelization

Parallelization is essentially a technique where we trade power consumption for area. If we take an ALU as an example, with a delay of $T$ seconds, operating at a clock speed of $1/T$ Hz, at its maximal (normal) voltage $V_{DD}$ ($V_{max}$). Now, if we would reduce $V_{DD}$ to a value $V_{low}$, such that the delay of the ALU is no longer $T$ but $2T$ (which allows a $V_{DD}$ reduction of roughly 40%), we are not able to run at $1/T$ Hz anymore. However, if we replicate the ALU and feed both outputs to a multiplexer, as depicted in figure 2.5, we are still able to deliver the same performance. Each ALU now produces a result twice as slow (the ALUs operate on $1/2T$ Hz), but they deliver a result every system cycle ($1/T$) in turns [10, 24]. The dynamic power consumption for the single ALU data path (reference path) is given by:

$$P_{ref} = C_{ref} \cdot V_{ref}^2 \cdot f_{ref} \quad (5)$$

And the same formula holds true for the parallelized data path:

$$P_{par} = C_{par} \cdot V_{par}^2 \cdot f_{par} \quad (6)$$

And if we take formula (6) and express the right side of the equation with terms from formula (5) only, we obtain:

$$P_{par} = (\phi C_{ref})(\gamma V_{ref})^2(\frac{f_{ref}}{2}) \quad (7)$$

The total effective capacitance being switched increases by more than two times (thus $\phi$ is slightly larger than 2), since apart from the duplicated ALU, we have to account for the additional wiring and the extra multiplexer as well. The system frequency inside the ALUs is divided by 2. For convenience, division by two and multiplication by $\phi$ can be cancelled out. This leads us to the following upper boundary for power saving by parallelization:

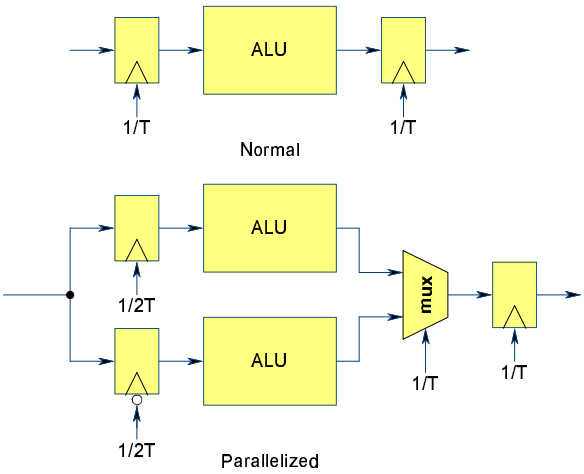


Figure 2.5: Parallelization principle

$$P_{par} = \mathrm{O}(\gamma^2 P_{ref}) \quad (8)$$

Note that $\gamma$ represents the supply voltage reduction; a number between 0 and 1. Since the cycle time doubles, we can decrease $V_{DD}$ until the delay of the circuit has (almost) doubled as well. According to [10], this happens when $\gamma = 0.58$. Since the ALU is replicated, the leakage power would theoretically double. However, since $V_{DD}$ is lowered by 42%, the increase in leakage power is only small. It may be obvious that parallelization is not suitable for area constrained designs (such as the arithmetic unit in biomedical implants). Also, if there is a **feedback loop** in the circuit, parallelization *cannot* be employed.

### 2.5.2 Pipelining

Another effective technique to reduce power is pipelining. Again we take an ALU as an example, with a delay of $T$ seconds, operating at a clock speed of $1/T$ Hz, at its maximal (normal) voltage $V_{DD}$ ($V_{max}$). If we can somehow subdivide the ALU in blocks and we place a pipeline register in between (depicted in figure 2.6), we decrease the latency of the ALU from $T$ to $2T$, but this does not have to be a problem since every cycle a result can be produced. Now the critical path in both blocks is much shorter (say, exactly half what it was before), we are allowed to decrease the supply voltage. This results in the following:

$$P_{pipe} = C_{pipe} \cdot V_{pipe}^2 \cdot f_{pipe} \quad (9)$$

$$P_{pipe} = (\phi C_{ref})(\gamma V_{ref})^2 f_{ref} \quad (10)$$

$$P_{pipe} = \mathrm{O}(\gamma^2 P_{ref}) \quad (11)$$

So, also here, the power savings are upper bounded by the supply voltage reduction $\gamma$. The total capacity being switched has been increased slightly because of the extra pipeline registers, so $\phi$ will be slightly larger than one. The supply voltage can be

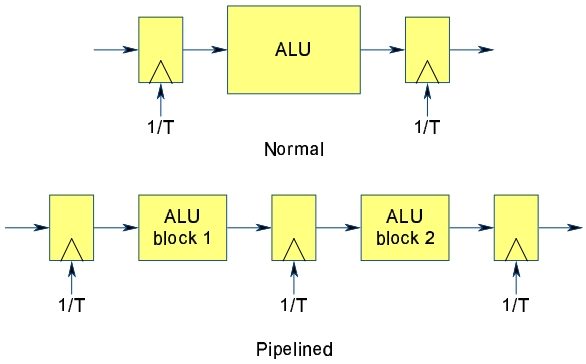


Figure 2.6: Pipelining principle

reduced to the same extent as in parallelization. This technique is, however, much more interesting for designs with *limited area budgets*. However, if there is a **feedback loop** in the circuit, pipelining *cannot* be employed.

Note that pipelining and parallelization can also be employed *simultaneously* to obtain even larger power savings. The same upper bound holds true for this method as mentioned in (8) and (11), but $\gamma$ can be even smaller, since the critical path has now been reduced to $4T$ instead of $2T$. A voltage reduction of approximately 60% ($\gamma = 0.4$, the point where the delay has quadrupled) is now possible [10]. Note that the relation between $V_{DD}$ and delay may vary between different types of technology, so the numbers of $\gamma$ we presented serve only as an indication.

## 2.6 Optimizations at Gate Level

At lower abstraction levels, techniques can also be applied to decrease the power consumption. At gate level, these techniques focus on optimization of the netlist in order to reduce power. We present three important optimization techniques: path balancing, high-activity net remapping, and fan-in decomposition.

### 2.6.1 Path balancing

**Spurious transitions** are a significant problem in combinational designs, since the portion of the total power consumption of a combinational circuit that is caused by spurious transitions can be as high as 10 to 40 percent [11]. Spurious transitions are useless transitions, since they do not contribute to the real computation and thus the power that is dissipated during these transitions is a waste of power. Their occurrence is caused by timing differences between different paths leading to the same logic element. For example, when a two-input XOR-gate does not receive both inputs simultaneously, but with a certain delay between input 1 and 2, a spurious transition may occur. This example is depicted in figure 2.7. On the left side of the figure (a), the paths 1 and 2 are assumed to have no delay. The signals on inputs A and B of the gate switch on the exact same time, such that the output signal Z remains unaltered. On the right side (b),

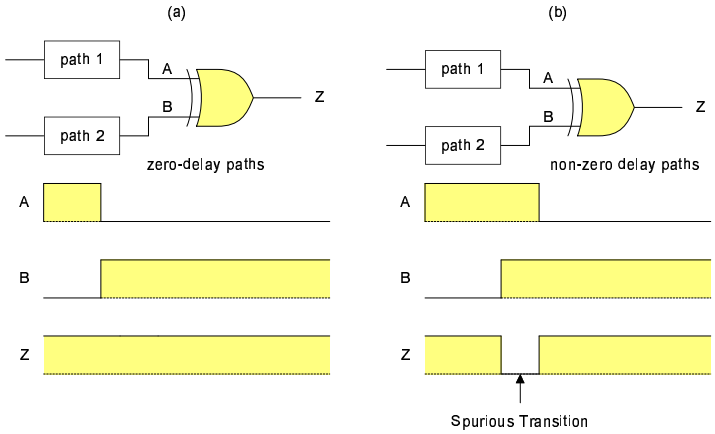


Figure 2.7: Example of a spurious transition due to unequal path delays

| Input | Capacitance |
|---|---|
| | [fF] |
| I1 | 1.628 |
| I2 | 1.898 |
| I3 | 1.667 |
| I4 | 1.995 |

Table 2.1: Input capacitance of a 4-input AND-gate

the paths do have a certain delay, as they will have in a real life situation. Path 1 has however a slightly longer delay than path 2, causing a spurious transition in output Z.

These undesired transitions can be eliminated by **path balancing** (a.k.a. path equalization), a technique that makes sure that the delay of all paths that converge at each gate is about equal [11, 21]. This can be done by inserting unit-delay buffers in the paths that are shorter than the others. The paths do not have to be exactly equal, since *if the width of a glitch is very small, it will not cause a spurious transition* (the glitch is then absorbed). A signal needs to be stable for a certain period of time in order to be propagated.

### 2.6.2 High-activity net remapping

Since an $n$-input gate (e.g. a 4-bit AND-gate) can have significant differences in input capacitance between its various inputs, it is wise to connect the net with the highest switching activity to the input pin with the lowest input capacitance and vice versa, to reduce power (the higher the capacitive load, the higher the power dissipation). For example, the input capacitance of a 4-bit AND-gate in UMC 90nm technology can be observed in table 2.1 (values are represented in Femtofarad; $10^{-15}$ F). In this particular situation, the high-activity net should be mapped to input 'I1'. An optimization tool (which obviously has to be aware of the switching characteristics of each net) can remap the nets in order to reduce dynamic power [13].

### 2.6.3 Fan-in decomposition

Gates with large fan-ins are undesired because these gates have typically high input capacitances. High input capacitances result in slow logic and high power consumption [25]. It is better to decompose a gate with a high fan-in into a network of multiple gates with a low fan-in, which significantly reduces the total capacitance. For example, in the UMC 90nm library, gates have a maximum fan-in of four inputs. Typically, the synthesis tool takes care of these optimizations. For example, a 16-bit AND-gate implemented in VHDL, will be decomposed by the synthesis tool and implemented by e.g. a tree of 4-bit AND-gates.

## 2.7 Optimizations at Technology Level

At the lowest abstraction level, the technology (or transistor) level, a number of optimizations can be performed in order to reduce power consumption. When no low-power methodologies have been applied to the circuit, optimizations at the technology level will not be necessary. But if the supply voltage is altered and the delay of the circuit is compromised by low-power design methodologies, optimizations at technology level may be desired. Optimizations at technology level includes adjusting the threshold voltage of the transistors, and/or altering their sizes.

### 2.7.1 Resizing transistors

One should know, that the standard size of the transistors (MOSFETs) is based on the normal, maximum supply voltage. When $V_{DD}$ is reduced, the size of the transistors is no longer optimal.

With the size of the transistor we mean the W/L-ratio, which is the **channel-width to channel-length** ratio, as depicted in figure 2.8. If we increase the W/L-ratio by a factor N, the delay of the transistors will decrease. At the same time, the energy consumption will increase linearly. If we want to consume the absolute minimal amount of energy, we should shrink the devices to their minimal size. The delay of the transistors then increases to their maximal values. The reason for this is inherent to the shrinking of the transistors. As the transistors become smaller, also the **current drive** of the transistors decreases, which effectively makes the transistors slower [23].

If we take the **parasitic contributions** into consideration, the theory becomes more complex and more interesting. With parasitic contributions (P) is meant the ratio between $C_p$ and $C_{ref}$. $C_p$ represents the parasitic capacitance which is the sum of the **junction** (substrate coupling) **capacitance** and **interconnect capacitance** of the transistor. $C_{ref}$ is the gate capacitance of a transistor with the smallest possible W/L-ratio (N=1). If P=0, then energy is indeed linearly dependent on N. If $P > 0$ this is no longer the case, as depicted in figure 2.9. For higher values of P, we can afford to increase the size of the transistors and thus decrease the delay of the transistors, *without* increasing the energy consumption. In fact, the energy consumption first decreases for increasing N, and then starts increasing again. This means that for every value of P, there is an *optimal* value for N. Note that for P=0.5 in figure 2.9, the optimal value

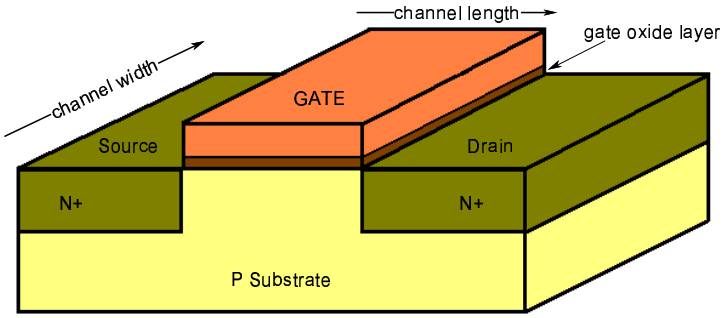


Figure 2.8: MOSFET (NMOS)

of N is still 1, since E does not decrease when N increases. The benefits of resizing is that we can compensate for the speed loss due to the $V_{DD}$ reduction. By increasing the transistor size, the speed of the transistors goes up again. For relatively small values of P, the energy consumption will increase, but E only increases sub-linearly with N. Still, the energy increase may cancel the energy savings by $V_{DD}$ reduction. In that case resizing is not useful. For higher values of P, there exists an optimal value for N, which enables a significant speed-up and additional energy savings. At least we can say that in *some cases* resizing and $V_{DD}$ reduction together enable significant energy savings without compromising the delay of the circuit [10].

It is obvious that not every transistor resides on the critical path. Resizing transistors (shrinking them) may violate the performance constraints of the design. Not resizing transistors may violate the power constraints. A solution is to shrink all transistors except those on the critical path. The transistors on the critical path can maintain their original size.

### 2.7.2 Optimizing the $V_{DD}/V_{TH}$ ratio

As mentioned before in section 2.4.1, voltage scaling is limited by the threshold voltage of the CMOS devices. The closer $V_{DD}$ gets to $V_{TH}$, the larger the delay penalty will be. If a significant reduction in $V_{DD}$ is required, but this results in unacceptable delays (when $V_{DD} < 4V_{TH}$), it is possible to decrease $V_{TH}$ as well in order to keep $V_{DD} > 4V_{TH}$, and thus keep the delay penalty acceptable. However, we have to be aware that the threshold voltage is also an important factor in the leakage energy. Even a small reduction in $V_{TH}$ leads to a significant increase in $E_{leak}$, since $E_{leak}$ has an exponential dependence on $V_{TH}$ [22]. However, even though $V_{DD}/V_{TH}$ scaling will increase leakage energy, the leakage energy component is generally a small fraction of the total energy consumption, so the impact will be limited. The dynamic energy will decrease quadratically as well, since it is quadratically dependent on $V_{DD}$. [9, 24]. Normally, $V_{DD}/V_{TH}$ scaling is therefore efficient. *Extreme* $V_{DD}/V_{TH}$ scaling leads however to *extreme* leakage currents. And even though dynamic power energy be very low, ultimately the increase in leakage energy will *cancel* the dynamic energy savings. So there is a limit to $V_{DD}/V_{TH}$ scaling. An optimum for a given $f_{clk}$ exists, and is reached when $E_{dynamic} \approx E_{leak}$ [24]. Others claim the optimum at $E_{dynamic} \approx 2E_{leak}$.

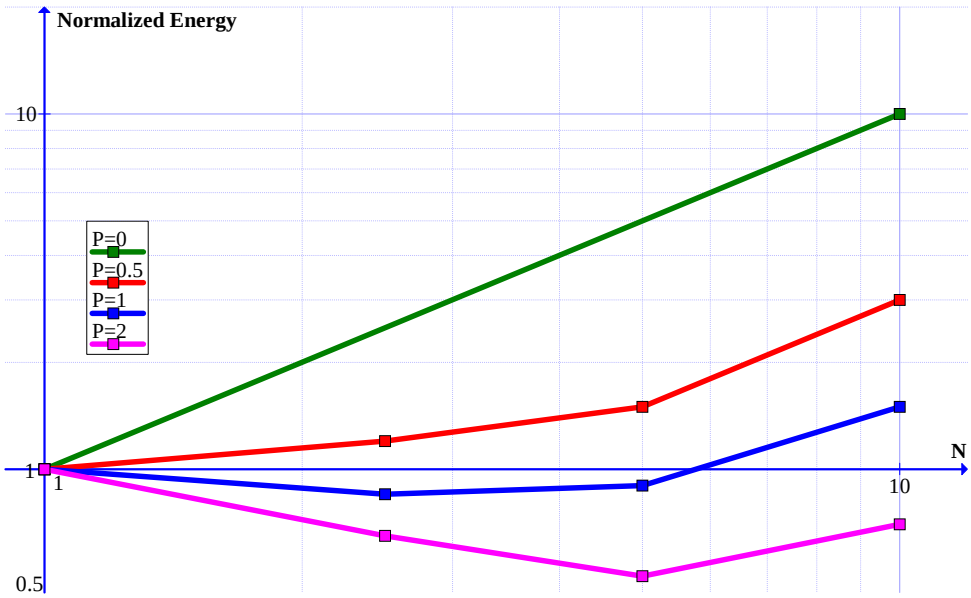


Figure 2.9: Energy consumption versus scaling factor N for various values of $P$ (logarithmic scale)

In conclusion, simultaneously optimizing $V_{DD}$, $V_{TH}$, *and* transistor size will lead to the optimal result. By not only reducing $V_{DD}$, but also optimizing $V_{TH}$ and transistor size it is possible to achieve significant energy savings *without* compromising the speed of the circuit.

## 2.8 Different Digital Logic Styles

Standard CMOS is the most common and widely utilized digital logic in almost any application field. Still, other digital logic styles exist and are utilized in the industry as well. Since we aim in this thesis work for special design characteristics, such as very low power consumption and very small sized designs, it is fair to have a look at other digital logic design styles as well. First the fundamental differences between static and dynamic logic will be explained. Then, the three most interesting alternative digital logic styles are discussed.

### 2.8.1 Static vs. Dynamic logic

In static logic circuits the clock signal is only utilized for memory cells (flip-flops). Pure combinational circuits do not need a clock signal at all. In dynamic logic, *all cells are clocked* (this is the reason why dynamic logic is also referred to as clocked logic), even if the circuit is purely combinational. This may seem odd, especially given the fact that the clock network is one of the largest energy consumers in almost any design, but dynamic logic provides a number of advantages. Dynamic logic is actually commonly utilized in computer memories nowadays. All types of DRAM is dynamic logic (Dynamic Random Access Memory). Another well-known application of dynamic logic is **domino logic**.

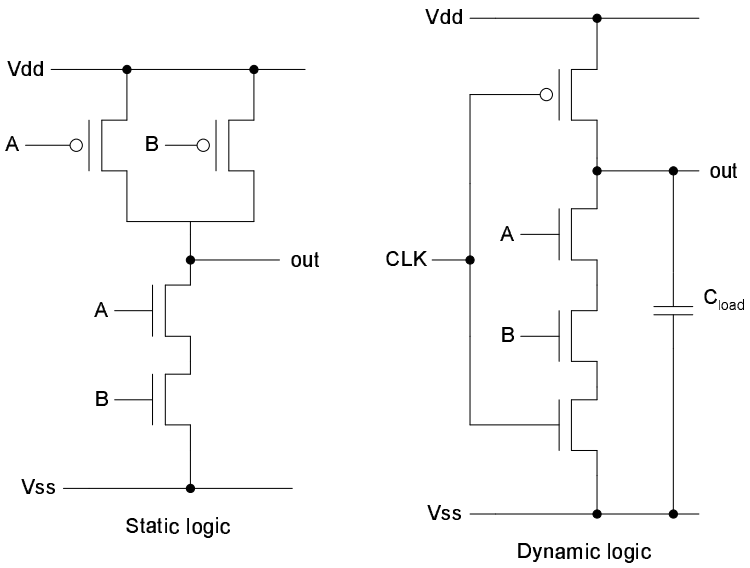


Figure 2.10: AND-gate implemented in static and dynamic logic

In dynamic logic we distinguish between two phases: the **precharge phase** (when the clock is low) and the **evaluation phase** (when the clock is high). During the precharge phase, the output of the logic circuit is driven to '1', regardless of the inputs. This way, the load (whatever is connected to the circuit) is charged. During the evaluation phase, the circuit is actually producing a meaningful result. However, when the output is '1', it is not because it is driven by $V_{DD}$. It is high because it is driven by the charged load. After a period of time this charge will leak away, so every clock cycle the charge process has to repeat. This is the reason why DRAM requires **'refreshing'** (recharching) to avoid losing data. To give an example of static versus dynamic logic circuits, a static and a dynamic logic implementation of an AND-gate is depicted in figure 2.10.

One of the primary advantages of dynamic logic is the *speed*. It is significantly faster than static CMOS, because dynamic does not utilize slow PMOS-transistors (PMOS transistors are slower than NMOS transistors) for the actual computations (the computations are performed by NMOS pull-down networks), and the capacitive load and the threshold voltage of the devices is lower in dynamic CMOS. The packing density of complex gates in dynamic CMOS is much higher than that of static CMOS, which ultimately leads to *smaller chip areas*. Further, in dynamic logic there is *no glitching* (which is obvious, since all cells are clocked). The major drawback of dynamic logic is its power consumption, which exceeds that of static CMOS by far (due to large clock loads and huge switching activities), even though we eradicate glitching power [26, 9, 10, 27].

There exist several techniques to lower the power consumption of dynamic logic (such as dual-threshold voltages in domino logic [22]), which makes it more attractive for low-power environments, but still, employing dynamic logic is only justified when very high performances are required. In general, *dynamic logic is not suitable for low-power design* [26].

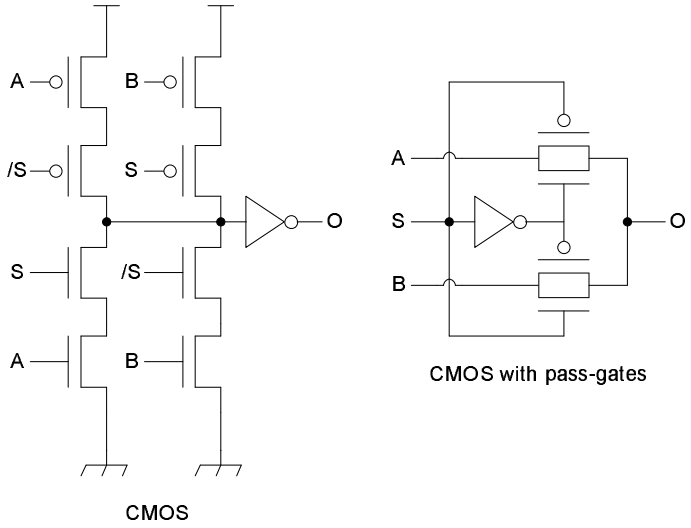


Figure 2.11: Multiplexer implemented in standard CMOS and CMOS with pass-gates [26]

### 2.8.2 Complementary Pass-Transistor Logic

Gates in standard CMOS are comprised of a PMOS pull-up and an NMOS pull-down network. The pull-up network is complementary to the pull down network. So called **pass-gates** can be employed for the efficient implementation of, e.g., multiplexers and flip-flops. Pass gates are *combinations* of a PMOS and an NMOS transistor, both in **pass transistor mode**. The fundamental difference of pass-transistors, is that the source is connected to a certain input signal, instead of $V_{DD}$. If pass-gates are employed as well, we refer to the logic style as CMOS with pass-gates (which is standard CMOS technology, with standard CMOS libraries with pass gates added to it). In figure 2.11 an implementation of a 2-input multiplexer is depicted, both in standard CMOS and CMOS with pass gates. It is clear that the implementation in CMOS with pass-gates is much more efficient, since it requires fewer transistors. In the case when pass-gates lower the amount of required transistors, the circuit area, power, and delay decreases. The number of logic functions that can be efficiently implemented by pass-gates is however limited. This has to do with the fact that each pass-transistor network must contain a multiplexer structure to avoid shorts between inputs. Other drawbacks of pass-transistor logic are:

- layout of pass-transistors is more difficult (irregular arrangements, high wiring)
- not very robust against voltage scaling
- very sensitive to transistor sizing (in fact, transistor sizing is crucial for correct operation)
- high short circuit currents

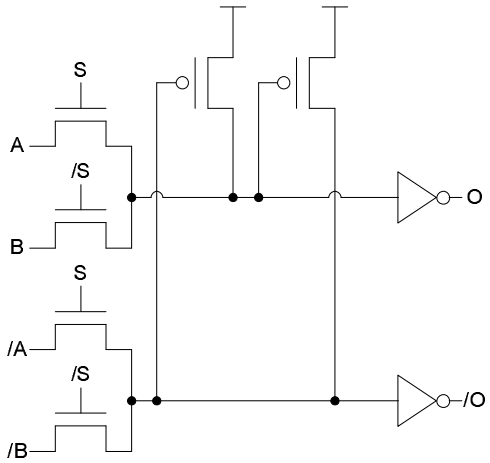


Figure 2.12: Multiplexer implemented in CPL [26]

A number of new pass-transistor logic styles have been developed since, which performs better than CMOS with pass-gates. One of them is CPL (**Complementary Pass-gate Logic**). It is one of the most common and well-know styles. CPL employs a double-rail structure. Every CPL gate consists of two NMOS networks, two PMOS transistors, and two inverters at the output (to provide the complementary outputs). The multiplexer we discussed earlier, now implemented in CPL, is depicted in 2.12 (this figure illustrates the function of the double-rail structure and complementary outputs perfectly as well). Note that also in CPL, every gate has to be based on a multiplexer structure, so this circuit is the *basis for every gate* in CPL. That means also for NAND and NOR gates, which results in rather inefficient implementations. CPL does have small capacitive input loads, very efficient implementations for, e.g., multiplexers and exclusive-or gates, and very good output drives (powered by the output inverters). On the other hand, double-rail structures involve many transistors and a lot of wiring, and as mentioned before, simple gates have inefficient implementations. And, CPL performs significantly worse at low supply voltages, which is a major drawback [9]. According to Zimmermann and Fichtner [26], *standard CMOS is the logic style of choice* for the implementation of *arbitrary* combinational circuits if low-voltage, low-power, and small power-delay products are of concern. Essentially the only component that appeared to be more efficient in CPL than in standard CMOS in the experiments of Zimmermann and Fichtner, was the full-adder. This is also reported by [10], since a full-adder contains XOR-gates, which can be implemented very efficiently. However, standard CMOS appears to be superior to CPL in almost all cases, in terms of power, but also in terms of area and delay.

### 2.8.3 Adiabatic CMOS logic

Adiabatic CMOS logic is a form of static CMOS with a couple of fundamental differences with respect to standard CMOS. As explained in section 2.3, dynamic energy is dissipated (as heat) because of switching capacitive loads and short circuit currents. Adiabatic

CMOS logic is also known as **energy-recovery CMOS**: techniques are applied to **conserve and recover** the energy that otherwise would have been dissipated as heat. Instead of a constant supply voltage, in adiabatic CMOS logic we focus on a **constant current**. What is necessary is a constant-current, variable-voltage source. In practice, the supply voltage is commonly combined with the clock signal for this purpose (**powered-clock**). The main idea is that when the powered-clock is rising, the circuit is evaluated and energy is transported to and stored in the circuit. When the powered-clock is falling, the recovery phase starts, where (most of the) energy is recovered from the circuit by the source. When a circuit capacitance is charged and the supply voltage decreases, the energy stored in the capacitance can be recovered by the source. Otherwise it would have discharged to ground. Only energy stored in capacitances can be recovered, energy dissipated as heat in resistances is lost forever. Adiabatic CMOS logic is rather complicated, and we will not explain the functionality of this logic style in detail, because it goes beyond the scope of the thesis work.

By definition, *adiabatic CMOS logic is applicable for (ultra) low-power design.* It however requires higher supply voltages than standard CMOS, and adiabatic circuits have longer delays. This makes adiabatic CMOS logic *less suitable for high-performance circuits.* Plus, adiabatic CMOS logic circuits have *higher area requirements* than in standard CMOS and adiabatic CMOS logic design requires extra design effort. Altogether, the primary question is whether the drawbacks are tolerable, and if adiabatic CMOS logic design actually leads to better results than *aggressive low-power design* for standard CMOS [9].

Blotti et al. [28] has implemented a carry-lookahead adder in PFAL (Positive Feedback Adiabatic Logic), which is one of the many different types of adiabatic CMOS logic. They report an energy recovery of 94%, which ultimately results in a power consumption that is about twenty times smaller than in standard CMOS. The area increase with respect to standard CMOS is limited to 20%. At first appearance these results seem formidable, however Blotti et al. have not applied any form of low-power design for the standard CMOS implementation. Therefore it remains unclear what the true power of adiabatic CMOS logic is in terms of energy savings.

Amirante et al. [29] has performed a similar experiment, with the implementation of an adiabatic logic ripple-carry adder in 0.13 $\mu$m CMOS. It is shown that PFAL is the most interesting adiabatic logic type, since it has the lowest energy consumption and the highest robustness against technology parameter variations. It is reported that the energy consumption of the adder is 6 times smaller than in standard CMOS (at 20 MHz). At frequencies above 100 MHz, the energy savings become significantly smaller. One important drawback can be found in the area requirements: the adder requires 50% more area than its standard CMOS counterpart. In resource constrained designs, this might lead to problems. Further, PFAL requires a four-phase powered-clock generator (off chip). Amirante et al. took this into account for the power comparison. It is possible to embed an adiabatic circuit (in this case an adder) in a standard CMOS environment (this is possible because it is based on the same *technology* (CMOS), except that it employs a different *style* (which is described in the adiabatic libraries)). Obviously, these interfaces will increase area and power overheads, but for larger adiabatic circuits, these overheads might be affordable. Therefore, PFAL to standard CMOS interfaces (and vice versa)

are required. Unfortunately, the energy results mentioned in this paper are insignificant, since no information is provided about the technology parameters of the standard CMOS circuit (e.g. if low-power design is applied by lowering $V_{DD}$, resizing transistors, etc.). That means we have no clear answer whether PFAL achieves larger energy savings than aggressive low-power design in standard CMOS. This topic is therefore an interesting direction for future research.

## 2.9 Conclusions

We have discussed the **sources of power consumption** in CMOS: static (leakage power) and dynamic power consumption. Dynamic power can be subdivided into capacitive power and short-cut power. Only dynamic power is dependent on the switching activity in the circuit. Further, the difference between low-power design and power aware-design is explained.

**Standard low-power methodologies** are voltage scaling, frequency scaling, clock gating, and power gating. Voltage scaling (lowering $V_{DD}$) is the most powerful technique, since power (and energy) have a quadratic dependency on the supply voltage. Voltage scaling can be static or dynamic. In dynamic voltage scaling, the supply voltage is adjusted based on the required performance of the system at that particular time. There exist several forms of dynamic voltage scaling, sometimes combined with frequency scaling. Voltage scaling does not only reduce energy, but also contributes to the reliability of the technology. Clock gating is a simple, low cost, but excellent way to shut down idle blocks and prevent any internal switching activity. Power gating can be utilized for the same purpose, but is more complex and invasive. The major advantage of power gating is that not only dynamic power is cut off, but also leakage power. Drawbacks of power gating are additional delays and significant area costs.

At **architectural level**, low-power methodologies can be applied as well, such as parallelization and pipelining. Parallelization increases area significantly (since the original design is replicated) but is a powerful method (by employing voltage scaling) to decrease power without compromising the throughput. About the same results can be achieved with pipelining. The original design must however be dividable into two parts. The area costs of pipelining is much lower than of parallelization, which makes it more attractive to resource constrained designs. But, both parallelization and pipelining cannot be employed if the is a feedback-loop in the design.

At **gate level**, we can balance unequal paths (try to make the delays of the paths involved equal), to avoid spurious transitions, which can account for 10-40% of the total power consumption. Another optimization technique is high-activity net remapping. Here, the nets with the highest switching activity are connected to the inputs of the gate with the lowest input capacitance, to keep the power consumption as low as possible. A third technique, performed by the synthesis tool, is fan-in decomposition. Gates with large fan-ins are slow and have high input capacitances. By decomposing these gates into a network of smaller gates, the total capacitance is reduced, and so is the power consumption.

At **technology level**, we are able to resize transistors and to optimize the $V_{DD}/V_{TH}$ ratio. Their sizes can also be increased, which is in particular interesting when the

parasitic capacitances of the transistors are not that small. It allows us to decrease the delay of the transistors, without increasing the energy consumption. This is a very interesting technique to compensate for delay penalties due to voltage scaling. If $V_{DD}$ is lowered significantly, and $V_{DD}$ gets close to the threshold voltage of the transistors, the delay penalty will be massive. To prevent this, $V_{TH}$ should be reduced as well. The problem is, however, that leakage power increases quadratically when $V_{TH}$ is lowered. Extreme $V_{TH}$ lowering will lead to extreme leakage energies, which might cancel the dynamic energy savings.

In conclusion we can say that low-power design always has a price. The easiest way is to trade performance for lower-power. If we cannot afford losing (much) performance, we will have to resort to more complicated methodologies, which will cost for example larger chip areas and/or longer design times.

Finally we had a look at *alternative digital design styles*. Dynamic logic has a number of very interesting properties such as higher speeds and smaller areas compared to standard CMOS, plus the property that it has no glitching, however the significantly higher power consumption makes dynamic logic not applicable for low-power design. Pass-transistor logic (like CPL) leads to lower power results for a small group of cells, such as multiplexers, exclusive-or gates, and full-adders, but in arbitrary designs, standard CMOS still is still the best choice (primarily because the implementation of other cells is often less efficient). Adiabatic CMOS logic is by definition suitable for low-power design, however the higher area requirements might be a problem in resource-constrained designs. The key in adiabatic design is that techniques are applied to conserve and recover energy that otherwise would have dissipated as heat. Unfortunately, based on the available literature, we were unable to tell whether adiabatic CMOS logic still performs significantly better than standard CMOS when aggressive low-power design is applied to standard CMOS design.

# 3 Tools, Libraries and Tool Flow

## 3.1 Introduction

Apart from an extensive literature survey of existing designs, in this thesis work we also introduce novel designs. These novel designs are implemented in synthesizable VHDL in order to analyze their characteristics. The VHDL language provides us the means to describe hardware and to test its functionality in a simulator. It is of great importance to be able to report on the power consumption, area requirements, and also the delay of a specific design in order to be able to make comparisons. For that purpose, we need synthesis and analysis tools. For very accurate results, we might even need to resort to place-and-route tools. Synthesis can also be utilized to back up theoretical assumptions. Before the final design is presented, a number of iterations is, generally, passed. For writing the VHDL code and obtaining the characteristics of the design, many different tools are involved. There is no ready-to-use software tool which can perform all the necessary tasks by simply pressing a button. Setting up the tool flow is a complicated task and getting familiar with the different tools is very time-consuming. This chapter explains which tools are utilized and why they are required. Above all, the overall tool flow is explained in detail. Our experience during this thesis work has been that, although the complexity and interaction details of the various tools needs are high, still no single tutorial text could be found to help us in our efforts. Given also that the utilized tools are established, top-of-the-line commercial products, we thought it would be very beneficial to include a chapter delving into the practical details of these tools. It should be noted that depending on the level of expertise with these tools, resulting designs can range from infeasible to highly attractive. This means that a deep understanding of the tools and their various quirks is a necessity for any serious design nowadays. This chapter is compiled based on information from various sources, as well as our own experience with the tools. This chapter may serve as a short but —we believe— very useful tutorial for future students who will have to work with ASIC synthesis tools as well.

## 3.2 Tools and Technology Libraries

All designs presented in this thesis are described in VHDL [30, 31]. ModelSim from Mentor Graphics [32] is utilized as VHDL compiler, simulation and debugging environment. Synthesis and chip area analysis is performed by utilizing Synopsys Design Compiler, and Synopsys PrimeTime for accurate timing and power analysis [33, 34]. Together with Cadence SoC Encounter, which is employed for place-and-routing and post-layout area analysis [35, 36], it provides an excellent hardware (ASIC) design environment. The tools, according functions, and general flow, are depicted in figure 3.1. Apart from that, the figure tells also something about the precision of the area,

timing, and power reports after each design phase, and which libraries are required for which tools. We will explain much more about this in the next sections. Please note that figure 3.1 is a simplified representation of the tool flow. Later in this chapter we will discuss and depict the tool flow in close detail.

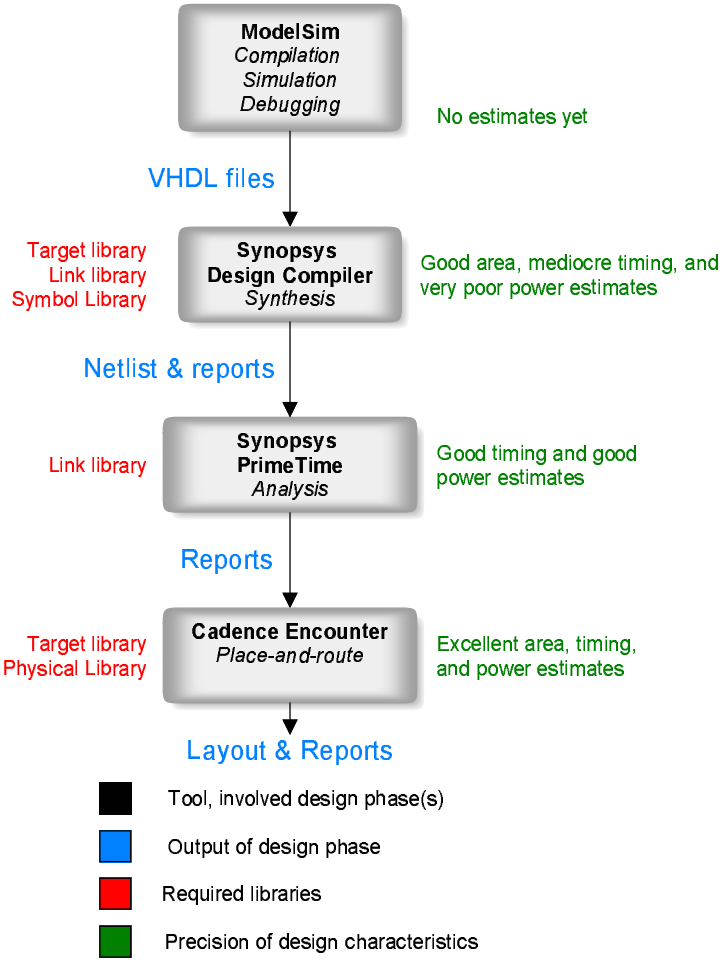


Figure 3.1: Tools, functionalities, and simplified tool flow

In this thesis work, we utilize four different technologies for synthesis from two different manufacturers (United Microelectronics Corporation and Taiwan Semiconductor Manufacturing Company):

- UMC 90nm SP (Standard Purpose)
- UMC 130nm SP
- TSMC 65nm GP (General Purpose)
- TSMC 65nm LL (Low Leakage)

The reason several technologies are employed is to investigate how certain designs scale with different technologies, which is one of the thesis's objectives.

## 3.3 Synthesis

Synthesis is the process where a hardware description on a high abstraction level is transformed into a design implementation in terms of logic gates. The input of the synthesis tool are VHDL (or Verilog) files, usually describing the hardware on RTL level. The primary output of the tool is a **netlist**, which is also a VHDL (or Verilog) file, but now on a much lower level of abstraction, namely on **gate level**.

Before the synthesis tool, Synopsys Design Compiler (DC), can be utilized, a **setup file** must be provided with proper links to the technology libraries. This involves the target library, link library and the symbol library. The **target library** contains all elementary cells, like logic gates, multiplexers, flip-flops, etc. which are utilized to build the design with. Each cell is annotated with information about the size, delay, power consumption and input and output capacitances, which is utilized by DC to compute the area, delay and power consumption of the design. Apart from cells, the target library also contains a number of **wire load models** and **operating conditions**. A wire load model is utilized to calculate the net delays (and net area) and can be chosen manually, but it is better to leave it up to DC to decide which model is appropriate. The criterion for choosing the appropriate wire load model is the total cell area. The idea behind this is that the average wire length increases for designs with a higher total cell area.

One should note that for small CMOS geometries, 350 nm or below, the net delay calculations based on wire-load-models are not very accurate, but provide a reasonable estimate. For accurate net delay calculations, a place and route tool (such as Cadence Encounter) is required. Different operating conditions (e.g. best, typical, and worst case) can be chosen, to compute the delay and power consumption subject to variations in supply voltage and operating temperature.

The **link library** refers to a library of cells used solely for reference. This means that the cells from this library are *not* inferred by DC. For example, the target library may contain all standard cells, while the link library contains macros such as memories and I/O pads, which we only want to **link** to, but for whatever reason not want to **infer** in the design. A technology library can be utilized as target library, as well as link library. Thus, there is no such thing as a target and a link library in the sense that they are different in nature. They are only utilized in a different manner. The link library is in particular useful to **remap** a design to a newer library version. In this case, the target library is set to the new library and the link library to a previous version. Using a special command in DC, a design mapped by older technology can easily be converted into one utilizing new technology. The target library is only utilized by DC, PT only works with the netlist produced by DC and, if necessary, with macros from the link library.

The **symbol library** assigns symbols to all target cells, utilized to translate the netlist into a schematic representation. This is very useful to see how the VHDL design is actually implemented and to search for bugs. Synopsys Design Vision is utilized for this purpose, which is the graphical front-end version of DC. For all other purposes, the command line interface of DC is the most convenient.

Apart from the libraries we have mentioned so far, the is another important library: the **physical library**. The physical library is utilized only by the place-and-route tool and contains the physical characteristics of each logical cell, like physical dimensions,

layer information, etc. The place-and-route tool performs the physical cell placement and routing (called layout) and provides the ability to place cells optimally, based on the area and timing constraints of the design. Post-layout results are the most accurate results up to this point in the design process. Which libraries are utilized in which phase of the design process, is also depicted in figure 3.1.

After providing the setup file, another important file is required: the **synthesis script file**. This file is to be created by the designer and should tell DC what to do. The script file generally contains the following sections:

- List of VHDL files to compile and the name of the top level design
- Setting the operating conditions (and wire load model)
- Setting **input drive strength** (characteristics of cells driving the inputs of the design)
- Setting **load capacitances** on output ports.
- Setting **design constrains** (maximum area, delay, dynamic power, static power)
- Compile command with necessary options (level of design effort)
- Report commands, to report on the results of the synthesis (preferably writing it to an output file)
- Write command to produce the netlist in VHDL format

By setting the drive strength we let go of DC's unrealistic assumption of zero drive resistance and thus infinite drive strength at the inputs of the design. The drive strength can be set by a number or by selecting a cell from one of the technology libraries which is used as a reference for the driving cell. A finite drive strength will affect the transition delay of the port and DC will buffer the net if the drive strength is too low. By setting a load capacitance on the output ports of the design, the design —and its characteristics— will be more realistic, since DC assumes zero capacitance by default (no load). In reality there will always be a capacitive load. Setting the correct output load helps DC selecting selecting the appropriate cell drive strength of an output pad. When we compile we can select a number of effort levels. Without extra parameters a minimum solution is produced. A medium or high effort (incremental) compile requires additional instructions. This is only the case for the standard compiler of DC. Nowadays, virtually only DC Ultra is utilized, which produces the optimal result by default. DC Ultra has more powerful algorithms to perform the synthesis than the standard version. One should note that the standard version of DC is still present in the latest Synopsys software. Therefore, it is important to invoke the command 'compile_ultra' and not 'compile', in order to obtain the optimal results. In the following sections and chapters, the utilized commands in the script files will have some further explanation wherever necessary. The DC and PT script files that were involved in the design of the fault-tolerant scalable arithmetic (introduced in chapter 1 and discussed in chapter 5 and 6) unit can be found in appendix A, in order to give an example of a complete script file we actually wrote and utilized.

## 3.4 Area and Timing Analysis

The area analysis is straightforward. After the synthesis step is completed successfully, the area can be reported by the

```
report_area
```

command (generally present in the script file). The total area of the design is subdivided in **cell area** and **net area**. The UMC and TSMC libraries are provided with zero-area wires only, which makes it impossible to report on the net area. However, in general the net area is just a small portion of the cell area. The area is reported in logic units. These logic units (in this thesis often simply abbreviated by 'units') are dimensionless quantities and provide only **relative** information about the required chip area. Only the place-and-route tool can calculate the actual chip area in square micrometers. To give an impression about the logic units: a simple NOR-gate has a size of 4, an AND-gate of 5, and a XOR-gate of 10 logic units (according to the UMC 90nm databook).

Reporting the delay of the critical path in purely combinational circuits is usually easy. By invoking the

```
report_timing
```

command DC computes the delay from all inputs via all possible paths to all outputs. The longest path is reported. One should, however, be aware of the possibility that the critical path reported, might never occur in real life. DC does not detect that, because the timing analysis is static. Only dynamic timing analysis would discover this, by applying all possible input vectors to the design. When this occurs, one needs to disable the false path manually. More about this topic can be found in the discussion of the Carry Skip Adder (which will be discussed in the next chapter, section 4.4.1). In sequential designs, the timing analysis is slightly more complex. Our topic of interest is the critical path, which may be:

- a purely combinational path from an input port to an output port
- a path from an input port to a register
- a path from a register to an output port
- a path between two registers

All four situations mentioned above must be analyzed individually by invoking the commands:

```
report_timing -from [all_inputs] -to [all_registers -data_pins]
report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins]
report_timing -from [all_registers -clock_pins] -to [all_outputs]
report_timing -from [all_inputs] -to [all_outputs]
```

These commands can be added to the DC script file. A faster and more powerful timing analyzer is Synopsys PrimeTime (PT). The estimations are more accurate than the ones produced by DC [37]. It requires a script file with instructions as well.

## 3.5 Power Analysis

### 3.5.1 Basic power analysis

Accurate power analysis is the most complex of all, because the actual power consumption highly depends on the circuit's switching activity, which is, in turn, highly dependent on the circuit's input data. It is possible to simply invoke the command:

```
report_power
```

after the synthesis has completed. Then, the power compiler of DC just assumes a static probability of 0.5 (i.e. the percentage of the time that a signal is high is assumed to be 50%) and the maximum possible toggle rate as the switching activity on all design nets and ports. DC then performs a power analysis that does provide a very rough estimate. Since the estimations are based on non-realistic switching activities, they are not of much use.

### 3.5.2 Power analysis based on simulated switching activity

The best method is to obtain accurate information about the circuits switching activity first. ModelSim can be utilized for this purpose. After the synthesis has completed, DC can output the netlist in VHDL format. The netlist, together with a testbench, can be imported in ModelSim. The netlist file contains numerous references to logic cells from the technology library. A new ModelSim library must be created to make ModelSim aware of these components. Fortunately, every technology library provides VHDL descriptions of the logic cells as well, which can be referenced once they are compiled and added to a library in ModelSim.

It is, however, very important that the stimuli in the testbench represent real data, i.e. data which the design eventually in real life has to process. If it is unknown at design time how the data looks like, it is wise to use worst-case-scenario stimuli or so-called 'optimistic worst-case' scenario stimuli, which are random inputs. For generating random test vectors, we utilized a generator which can be found online [38]. This generator produces numbers based on the atmospheric noise, which leads to better results than most pseudo-random computer algorithms. During simulation it is possible to back-annotate the switching activity of each net in the design. ModelSim is, like most other EDA logic simulation tools, capable of dumping all occurring value changes during the simulation in an ASCII-based VCD (Value-Change Dump) file. For this purpose, it is most convenient to create a ModelSim script file. It should contain the following commands:

```
vsim work.testbench
vcd file ./foo.vcd
add wave -r *
vcd add -r *
run -all
vcd flush
vcd off
quit -sim
```

What happens, is that the testbench (and all underlying designs in the hierarchy) is loaded by the simulator and a VCD file is created, called foo.vcd. All signals (nets) are added to both the wave window and the VCD file. Then the actual simulation is executed. Every time a signal changes value in the wave window, the switching event is written to the VCD file. Finally, the VCD file is closed and the simulation is terminated. Since the Synopsys software does not work with VCD files, the VCD file needs conversion into a SAIF (Switching Activity Interchange Format) file by utilizing the vcd2saif tool (part of the Synopsys software). A SAIF file is actually a hierarchical list of nets with annotated switching activity, e.g.:

```
(n168
   (T0 522) (T1 478) (TX 0)
   (TC 60) (IG 0)
)
```

means that a certain net 'n168' has value '0' during 522 time units (T0), value '1' during 478 time units (T1), and changes from value (1-0 or 0-1) 60 times (TC, which stands for transition count). TX stand for the total time the signal is undefined, and IG for the number of 1-X-1 and 0-X-0 transitions. At this point, the SAIF file can be utilized by Synopsys Power Compiler to calculate the power consumption. However, it appeared that there is a naming mismatch between the instance and net names in the netlist and the SAIF file. The VHDL compiler is case-insensitive, however, the power compiler that reads the VHDL netlist is not. This means that all instance and net names in the netlist are considered to be case sensitive and have to match with the names in the SAIF file. Unfortunately, ModelSim dumps all information to the VCD file in lower-case letters, regardless of the use of upper-case letters in the netlist. Since all names in the VCD file are lower-case, so are they in the SAIF file. When the power compiler finds a net 'N168' in the netlist and a net 'n168' in the SAIF file, they are considered to be two different nets. In reality they are the same. This leads to false power calculations. The easiest way to solve this issue, is to convert all upper-case letters in the netlist to lower-case, before we supply it to the power compiler. Perl is an excellent tool to do this:

```
> perl -pi.bak -e "tr/A-Z/a-z/" netlist.vhd
```

Having the synthesized netlist and the switching annotations, the actual power calculation can be performed. The most accurate power compiler is the power compiler in PrimeTime (called PrimeTime-PX) [37]. Once the calculation is done, the power compiler provides the following information:

- The cell internal power
- The cell leakage power
- The net switching power
- Total power consumption

As mentioned in chapter 2, power dissipation in CMOS has multiple sources. To recapitulate: the total power consumption can be divided into **static** and **dynamic** power consumption. DC utilizes specific definitions that need some explanation. The **cell internal power** is the dynamic power, dissipated inside all cells in the design, thus both the switching power and dissipated power due to shortcut currents. The **net switching power**, is the switching power dissipated outside the cells, as a result of the load capacitances at the outputs of the cells. Thus, the dynamic power is the sum of the cell internal power and the net switching power. Finally, the **cell leakage power** is what we call the static power component and needs no further explanation. The entire tool flow is depicted in detail in figure 3.2.

### 3.5.3 Power analysis including glitching power

The power analysis discussed in the previous section is much more accurate because of the utilization of simulated switching activity of nets and ports. We can however obtain even higher accuracies, if we account for glitching/spurious transitions as well. The glitching power component can be a significant portion of the total dynamic power consumption, as explained in the previous chapter (section 2.6.1). The reason why spurious transitions are not accounted for in the first place, is because ModelSim utilizes standard zero-delay transmission models. Obviously, in reality, each cell/gate has a certain delay. This implies that the outputs or internal nodes of a cell/gate can switch before the correct logical value is stable. However, glitching is often caused by signals arriving at different times at the inputs of a cell/gate. If we want to perform a power analysis based on a non-zero delay transmission model, we have to provide ModelSim with a SDF (Standard Delay Format) file, which is a file containing the delays of all cells utilized in the design. This file can be generated in DC, but generation in PT leads to more accurate results. Before the simulation in ModelSim can be started, the first line of the ModelSim script has to be modified:

```
vsim work.testbench -sdfmax /testbench/dut=timing-postsyn-PrimeTime.sdf
```

The option 'sdfmax' makes sure that the worst case delays are utilized from the SDF file. Now also the glitches are counted as transitions. If the width of a glitch is very small, it will in real CMOS never cause a transition since a signal needs to be stable for a certain period of time in order to be propagated. The glitch is then **absorbed**. ModelSim's VITAL Glitch[1] will detect this and disables the glitch generation preemptively. If the width of the glitch exceeds a predetermined threshold, such a glitch at the input of a gate may cause a transition at the output and then another transition when the output falls back to its original level (0-1-0 or 1-0-1). This glitch counts as two transitions and is commonly referred to as a spurious transition. This glitch is **propagated** [39]. When PrimeTime-PX is invoked, the netlist, SAIF file, and SDF file must be provided. The tool flow for power analysis based on non-zero delay transmission models is depicted in figure 3.3.

[1] VITAL Glitch is a feature of ModelSim to generate glitches based on the provided timing information of the circuit.

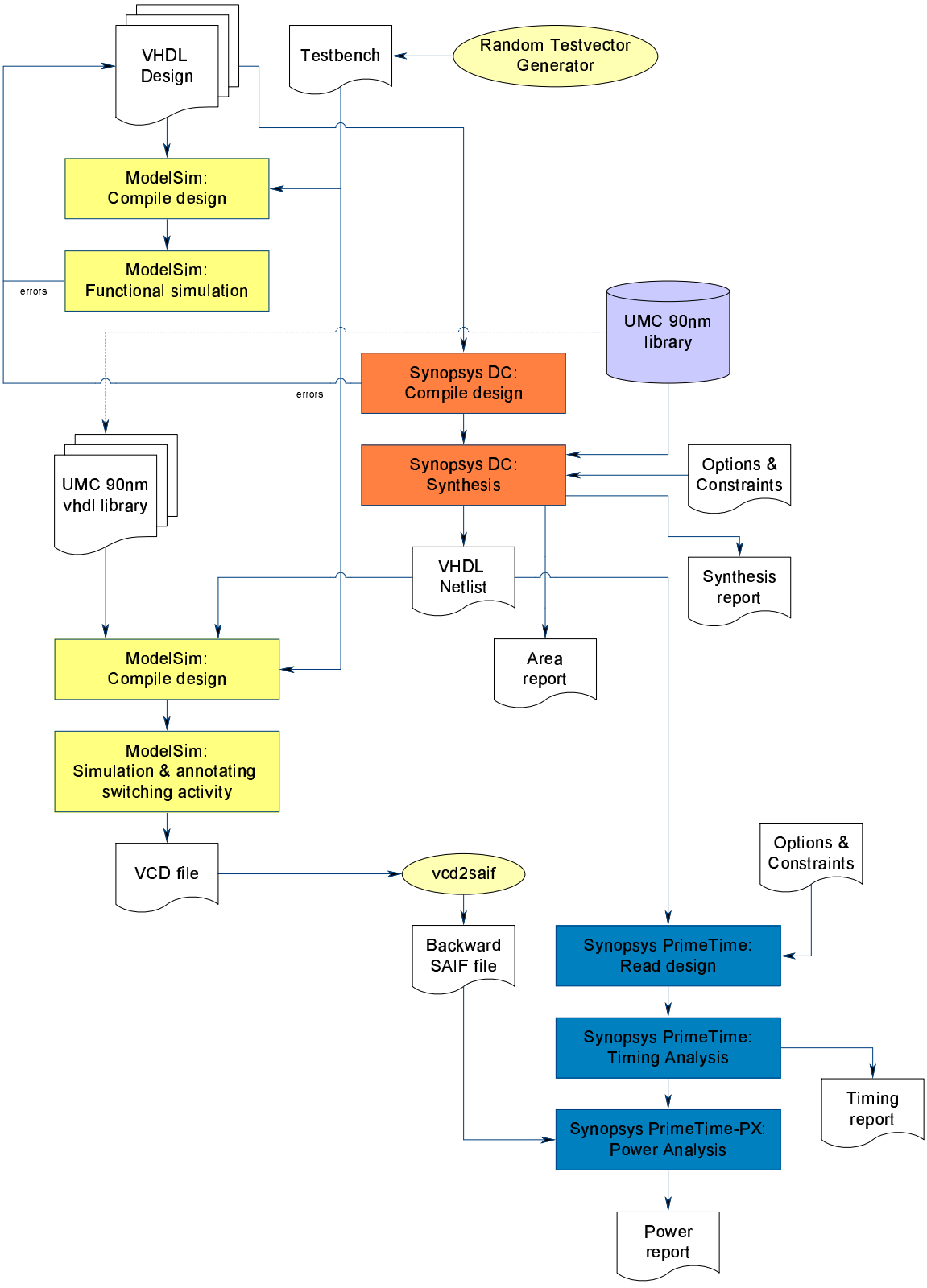


Figure 3.2: Tool flow for area, timing, and power analysis

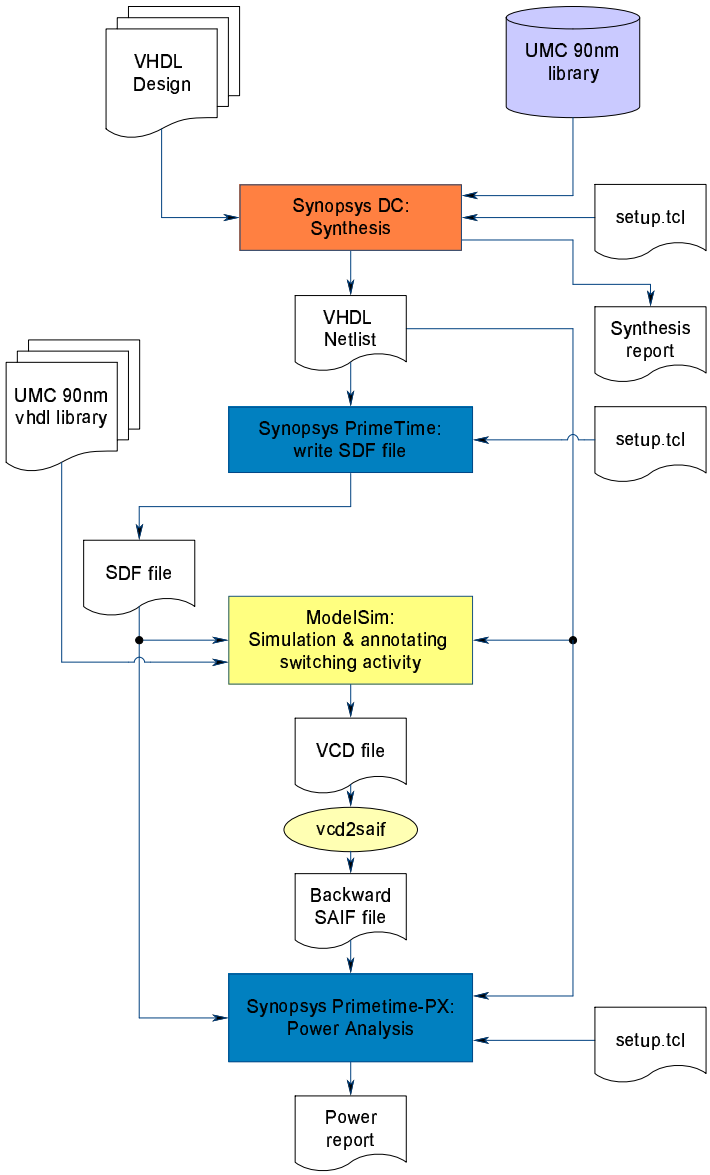


Figure 3.3: Tool flow for power analysis including glitching power

## 3.6 Place-and-Route

As mentioned earlier in the text, the place-and-route tool performs the physical cell placement and routing of nets. The tool has highly sophisticated algorithms at its disposal in order to place cells and nets optimally, trying to meet the demands of the designer. The output of the tool is the layout, which is the actual blueprint of the chip. The layout is the lowest level of abstraction in the hardware design process.

Even though DC/PT provides good estimates, they remain estimates. Estimated power, area, and timing numbers will always differ from the real values once the

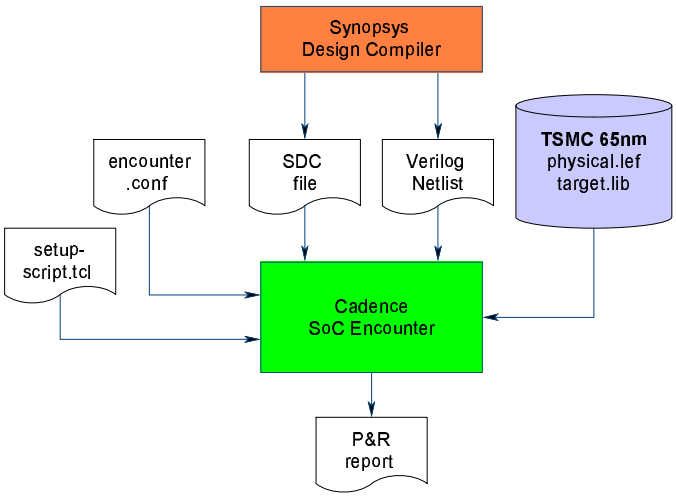


Figure 3.4: Tool flow for place-and-route

design is laid out and implemented in the chip. One could say that the numbers themselves only have a limited meaning: it is the relation between the results of different designs/implementations that provides us the real valuable information. For example, if the post-synthesis power consumption estimates of three designs X, Y, and Z are $a$, $b$, and $c$ (in $\mu$W), where $a > b > c$, it is not of great importance if the post-layout results turn out to be $\alpha = a \pm \Delta a$, $\beta = b \pm \Delta b$, and $\gamma = c \pm \Delta c$, as long as the trend holds: $\alpha > \beta > \gamma$. The chance that this is the case is actually very high: our designs are small, have a lot of common circuitry, and the wiring area is practically negligible for such small circuits. However, according to [40], there is always a chance that the layout changes the trend predicted by the synthesis results. For example, DC simply adds up the size of each cell in the design to compute the logic area. Two different designs with equal logic area might however, after layout, have substantial differences in chip area. This is the case when one of the designs is much more complicated for layout than the other. A larger chip area will in turn also affect delay and power.

Therefore, we decided to check some of the designs by a place-and-route tool. We employed Cadence SoC Encounter 7.0 for this purpose [35, 36]. We utilized an available script file with setup for TSMC 65nm GP technology [37]. This is the only technology we test for, since it would be too time-consuming to convert the setup to, for example, UMC 90nm technology. The setup of Encounter is much more complicated than the setup of the synthesis tools, and strictly speaking, place-and-route goes beyond the scope of the thesis. The only reason why we employ place-and-route is to obtain absolute certainty that the trends are correctly predicted by the synthesis tools. The tool flow for the place-and-route is depicted in figure 3.4. The synthesis files that are required are the netlist (in Verilog format) and the SDC file, which is the file containing the constraints of the design. Further, Encounter needs two important libraries: the target library (.lib file) and the physical library (.lef file).

## 3.7 Conclusion

Setting up a correct tool flow is a difficult task, since many tools are involved and each tool needs its own setup. The tool flow depends on the needs of the designer, which complicates it even further. For example, if we decide to analyze the glitching power as well, the tool flow becomes more complicated. Apart from the tool flow, it is important to have a thorough understanding of *at least* the basic functionalities of the tools as well. Without having any experience with synthesis whatsoever, this is a very time consuming task. It is however an essential part of the thesis work. Without synthesis it is obvious that we are unable to report on the characteristics of designs. In particular the experience one gains working with these tools provides an interesting insight in the transformation process of a hardware description in VHDL into a netlist of logic components and ultimately a layout. The reader should be aware that throughout this thesis pre-layout numbers are provided, unless specifically mentioned otherwise. Therefore the numbers have to be looked at for their relative, and not their absolute value.

# Exploratory Study Among Different Adder Types

# 4

## 4.1 Introduction

In this chapter, an extensive study among different adder structures is presented. A number of motivations have led to this study. First, one of the motivations for this study is to find the most suitable adder structure for implementation in the **scalable arithmetic unit**. As explained in chapter 1, instead of duplicating adders, we want to implement a single adder only, which is scalable in size. The basic idea is to shut down only a part of the adder, namely that specific part where the failure has occurred, and to continue with the computational work by employing the remaining part(s) of the adder. Not every adder structure will be suitable for implementation in the scalable arithmetic unit, since the adder must be divisible into (at least) two segments, capable of operating independently and together as a whole. A more extensive explanation of the scalable structure will be discussed in the next chapter (sections 5.2 and 5.3).

Second, a number of researchers have compared different adders in the past, such as [9, 41, 42, 43]. However, in all studies we found the adders are implemented in older technologies, such as 2$\mu$m CMOS [9] and 1.2$\mu$m CMOS [41], so it is difficult to predict what the delay and power consumption will be for much smaller, submicron technologies, such as 90nm CMOS. More importantly, it is important to know if the trends between the various adders in 90nm CMOS remain the same as in, e.g., 2$\mu$m CMOS. Therefore, the decision was made to implement, synthesize, and analyze the adders in order to make an accurate comparison between the different adder types, and between our results and the results in the literature.

This exploratory study has the intention to provide an up-to-date insight in the delay, area, and power characteristics of well-known adder structures implemented in modern technology. Please note that, at this point, we have intentionally decided to expand the scope of this thesis work. This part of the study is a **general study**, which does *not* focus on the employment of the adder in implants. The results of the study can be utilized by anyone who requires an adder in their design.

Apart from the general study, it is an important objective to find an adder which is suitable for implementation in low-power and low-area architectures, such as the one in SiMS. Therefore, we utilize standard metrics to classify the different adders. And, finally, we have a look at glitching power as well. Even though fast adders might not be required for the intended purpose, they might become interesting if they prove to reduce the unwanted glitching power to such extent, that the total power consumption of the fast adder is lower than the slower and simpler adder.

## 4.2 Various adder implementations

The most well-known and simplest adder is the **ripple-carry adder (RCA)**. The RCA design is regular, easy to implement and has the lowest area and power costs of all existing adders. Its primary limitation is the long delay, which grows linearly with the word width (for an $n$-bit adder, the delay is O($n$)).

Fast adders are designed for having shorter delays, which obviously comes at a certain price regarding the adders' area and power consumption. In this thesis, apart from the RCA, the following adders are investigated:

- Carry-lookahead adder
- Carry-select adder
- Carry-skip adder
- Ripple-carry/carry-lookahead adder (hybrid adder)

Fast adders have the advantage to scale better with increasing word widths (except for the ripple-carry/carry-lookahead adder, which is explained later on), since the delay of the previously mentioned n-bit fast adders is either O($\sqrt{n}$) (the carry-select and carry-skip adder), or O($log(n)$) (the carry-lookahead adder). Among the fast adder we find not only a variety in speedups, but also in area and power overheads. Although these adders are common and well-known, they will be discussed briefly for completeness. An in depth explanation can be found in [44].

The main idea behind the **carry-lookahead adder (CLA)** is to eliminate the slow carry propagation. In order to do that, each full-adder (FA) requires two additional outputs '$g$' and '$p$'. The '$g$' signal indicates that a carry is generated inside the FA, the '$p$' signal indicates that an incoming carry will be propagated (in other words, there is no 'room' to absorb an incoming carry). A fast **lookahead logic unit** (which utilizes the '$p$' and '$g$' signals to compute the carry input of each stage) enables the addition to be computed in O($log(n)$) time. The lookahead logic unit that is implemented is depicted in figure 4.1. The '*p-block*' and '*g-block*' signals are used to compute the carry output, or are fed to the next level of lookahead logic. This adder is the fastest one among the fast adders, so it is not surprising it also has a high cost. Although the full adders are simpler (partial adders can be utilized, since there is no need to compute the carry output), the carry-lookahead logic is the part of the design which makes it expensive.

The **carry-select adder** is based on *module replication*. The $n$-bit adder is divided into '$k$' ripple-carry adders of $n/k$ bits each, and all these adder blocks are replicated (except the lowest order part). The simplest $n$-bit carry-select adder is built using three $n/2$-bit ripple-carry adders. The first adder is utilized to compute the lower half of the $n$-bit sum, while the other two compute the higher half: one based on the assumption that the input carry is zero, the other based on the assumption that it is one. This way the computation of the higher half can start immediately; there is no need to wait for the lower half to complete. When the lower half of the sum is computed, and the carry input value for the next stage is available, the correct higher half of the sum is selected by a multiplexer. It is not difficult to see that the overhead of this adder (depicted in

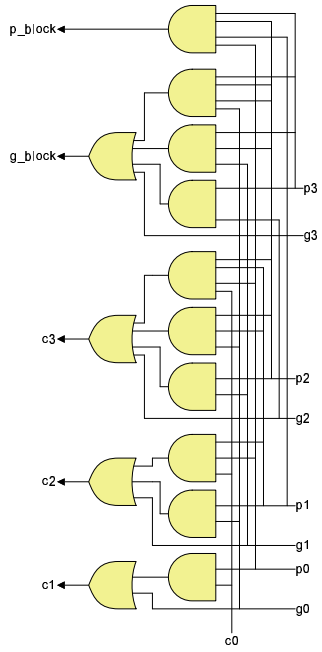


Figure 4.1: Implementation of the lookahead logic in the CLA [44]

figure 4.2) is quite dramatic. The required area and power consumption of this type of adder practically doubles with respect to the RCA (in particular when $k > 2$), because of the replication technique.

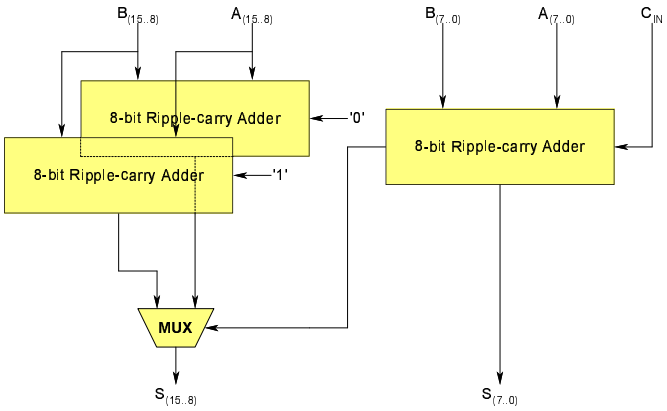


Figure 4.2: A 16-bit carry-select adder utilizing three 8-bit adders [44]

The **carry-skip adder (CSK)** is a very simple but ingenious adder, with a minimum of additional logic. The $n$-bit adder is divided into '$k$' ($n/k$-bit) ripple-carry adder blocks, where '$k$' is greater than, or equal to 4. Each adder block has a group propagate signal, which means that when this propagate signal is 1, an incoming carry cannot be absorbed and will propagate through the adder block instead. If that is the case, we could as well skip this adder segment. This is exactly the idea behind the skip adder: when it can be

determined, a priori, that a block cannot absorb a carry, it skips the block via the **skip logic**. In this study, the sizes of all blocks are fixed and identical. Variable block-size CSKs were omitted in this study, because of the irregularity of the design. In the next section is explained why this is such an important criterion in this study. However, for each adder width an optimal block size exists. The block size '$b$' can be easily computed by the formula: $b = \sqrt{n/2}$, where '$n$' is the adder width. E.g., an 8-bit CSK has an optimal block size of 2 bits, and a 32-bit CSK of 4 bits. For a 16-bit CSK the outcome of the formula is not an integer ($\sqrt{8}$), so in this study the 16-bit CSK is implemented with both 2- and 4-bit blocks, to investigate which block size is beneficial.

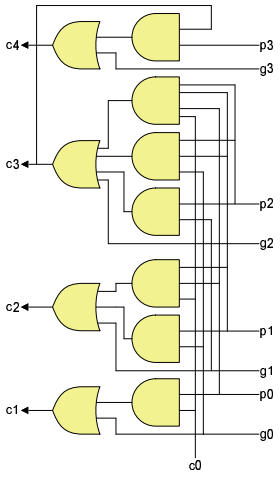


Figure 4.3: Implementation of the lookahead logic in the RCLA [44]

The **ripple-carry/carry-lookahead adder (RCLA)** is a *hybrid adder* which has a lower area and power overhead than a pure CLA, but is significantly faster than a RCA. The problem with CLAs is that the carry-lookahead logic cannot be extended to large word-widths. Not only because full lookahead logic increases the overhead dramatically for larger word-widths, but also because of the physical limitations regarding its implementation. The speedup of the CLA is based on the small number of logic stages to compute the carry signals. The larger the word-width, the higher the fan-in of the gates in the lookahead logic, which ultimately forces the synthesizer to implement more logic stages to cope with the large fan-ins. In practice, lookahead logic is implemented by standard 4-bit blocks. For example, a 16-bit CLA is implemented by four 4-bit CLAs, with an additional 4-bit lookahead block to compute the block carries. This is called a 2-level CLA and requires five lookahead blocks. The RCLA omits the lookahead block which computes the block carries, and connects the four 4-bit CLAs in a ripple-carry manner. This way it saves both area and power. Another advantage of the RCLA is its regular structure, in contrast with the 2-level CLA. In this study, all RCLAs (8, 16, and 32 bits) are implemented by a chain of 4-bit CLAs. The lookahead logic unit that is used here is depicted in fig 4.3. Since the block '$p$'and '$g$' signals are not required here, the carry output 'c4' is computed in a more relaxed manner, which decreases the overhead of the lookahead logic.

## 4.3 Suitable adders for employment in a scalable structure

For implementation in the scalable arithmetic unit, not only the delay, area, and power consumption of the adder are of great importance. The adder should also lend itself to segmentation (i.e. the ability to divide the adder up in segments, without compromising the functionality and benefits of the adder in question), and should therefore have a *regular structure*. Obviously, the RCA is always preferred in terms of power and area, when it appears to be fast enough for its intended purpose. It is also has the most regular structure of all known adder types. However, even when the RCA is fast enough, there might still be reasons to choose a faster adder. For example, a fast adder could buy the designer extra time for the implementation of additional hardware, such as error detection/correction. In this situation, where it is still unknown what the exact system frequency of the micro-architecture will be, it is important to investigate multiple adder types. Therefore, the CLA, and in particular the RCLA hybrid, as well as the carry-skip adder deserve some further investigation, mainly because of their regular structures and their high speed potentials. The carry-select adder will be omitted in the subsequent study. There are two reasons for that. First, this adder has no regular structure which makes the implementation in the scalable ALU difficult. Second, previous studies such as [9], show that the overheads of this type of adder are considerably higher than those of the CLA, while the delay appears to be longer.

## 4.4 Synthesis results of different adder types

### 4.4.1 Preface

The RCA, CLA, RCLA, and CSK are synthesized and analyzed for different adder widths, in order to have a good insight on how a particular adder type scales with word sizes. The adder widths that are utilized for the analysis are 4, 8, 16, and 32 bits. However, two exceptions exist. First, the RCLA is implemented as 8-, 16- and 32-bit versions only. Since 4-bit CLAs are utilized to build the adder with, the 8-bit RCLA is the smallest implementation possible. Secondly, the same is true for the CSK. The minimum adder size of the CSK is 8 bits. The CSK requires more than one full adder in each block to take advantage of the fast skip logic, and there must be a sufficient amount of blocks to actually be able to skip a block. A 2-block CSK is obviously useless, because since there are no blocks 'in between' the two there are no blocks that can be skipped. A 3-block CSK would be possible, but since we use word widths that are a power of two only, a 4-block CSK with a minimum of two full adders per block, makes an 8-bit CSK the smallest possible CSK.

Synthesis and analysis of the RCA is straightforward, however the analysis of the CSK and synthesis of the(R)CLA requires overcoming some obstacles. In general, the synthesis and analysis methods described in chapter 3 apply, except for the following cases.

Synthesizing the lookahead logic of the CLA requires some extra attention. The speed of the CLA is determined by the speed of the lookahead logic. In the literature, e.g. [44], the lookahead logic is implemented with a minimum number of logic stages,

which implies the use of logic gates with a higher fan-in. The lookahead function is described by the recurrence $c_{i+1} = g_i + p_i c_i$. In VHDL, the lookahead logic can be described in the same way:

```
c(1) := g(0) or (p(0) and c(0));
for i in 1 to (WIDTH-1) loop
     c(i+1) := g(i) or (p(i) and c(i));
end loop;
```

Even though this description is correct, Synopsys Design Compiler will *not* automatically unroll the loop to build the lookahead logic as depicted in figure 4.1. The implementation by DC is depicted in figure 4.4. It is obvious that this implementation is worthless, since it does not provide the desired speedup.

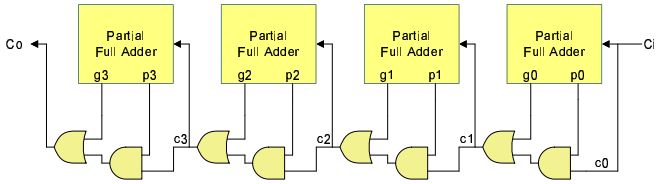


Figure 4.4: Incorrect implementation of the lookahead logic

Also, manually unrolling the loop in VHDL:

```
c(1)<=g(0) or (c_in and p(0));
c(2)<=g(1) or (g(0) and p(1)) or (c_in and p(0) and p(1));
c(3)<=g(2) or (g(1) and p(2)) or (g(0) and p(1) and p(2))
      or (c_in and p(0) and p(1) and p(2));
etc...
```

appears to have *no* effect on the actual implementation by DC. The reason, is that DC standard searches for small-size implementations, even *without* specific manual instructions by the designer regarding the area constraints. Therefore, the carry recurrence is not unrolled. And even when the VHDL description provides the unrolled recurrence already, DC will detect that the logic can be implemented as a chain of logic gates which is, in terms of area, beneficial. To *force* DC to unroll the lookahead logic loop, the command

```
set_max_delay 0.07 [all_outputs]
```

must be utilized (which can be added to the synthesis script file). This command forces DC to implement the lookahead logic in such a way, that all carry outputs have a maximum delay of, in this particular case, 0.07 ns. This way, DC has no other option than unroll the loop, utilizing gates with a higher fan-in and a higher drive strength, to meet the demands of the designer. The maximum delay command overrides all (manually set) area constrains in case conflicts occur.

The choice of the maximum delay value is the most difficult. Picking a value that is too low results in a CLA that does not have the speedup that it should have. Picking a

value that is too high, results in only minor, if any, further speedup. However, it does increase the area, since DC dauntlessly tries to meet the demands of the designer, by upscaling the drive strength of the logic gates and inferring driving buffers. The trick is to strike the golden mean: sufficient speedup, without massive increase in area costs.

Synthesis of the CSK is straightforward, but delay analysis is more difficult. A carry-skip adder is basically a ripple-carry adder. The important difference is that a the CSK is subdivided in RCA blocks and has additional skip logic attached to each block. In a RCA there is just one path for the carry to travel: through the RCA. In a CSK *two* paths exist: through the RCA-block, or bypass the RCA-block via the skip logic. Synopsys DC/PT just seeks for the longest path: which is through all RCA-blocks successively, and never by-passing them via the skip logic. However, in reality this is a *false path* (depicted in figure 4.5). The maximum path a carry will travel in a CSK is when a carry is generated in the first full adder of the first block, and absorbed in the last full adder of the last block. Thus, the maximum delay is determined by a carry rippling trough two adder blocks and skipping every other block in between. This path is depicted in figure 4.6.

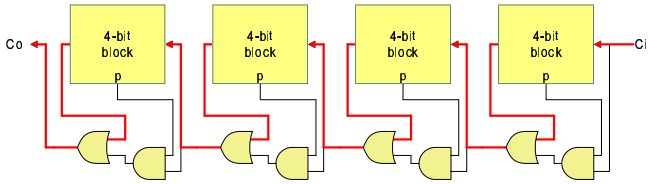


Figure 4.5: False path

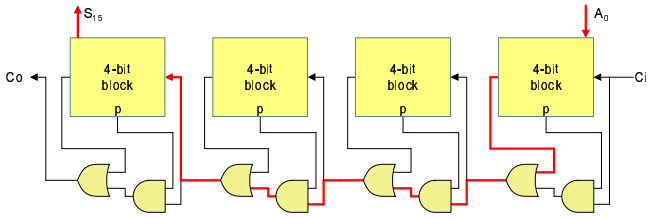


Figure 4.6: True critical path

Thus, there is a difference between the physical longest path and the true critical path. In this particular case, the physical longest path will never occur in reality. Synopsys is not able to detect that, because the timing analysis is static. Only **dynamic timing analysis** would discover this, by applying all possible input vectors to the design. To analyze the true critical path, the false path must be disabled manually. In Synopsys DC/PT, we can use the following construct:

```
set_false_path -through RCA_1/FA_0/c_in
set_false_path -through RCA_2/FA_0/c_in
```

An 8-bit CSK is composed of four RCA blocks (RCA0...RCA3) with carry-skip logic. By using the false path command like above, we declare the carry inputs of the middle

two RCAs as a no-go zone. In other words, we force the timing analyzer to follow the alternative path: the skip logic. This way we obtain the correct delay estimates.

### 4.4.2 Chosen frequency

The power analysis in this study is based on a frequency of 100 MHz (of course, the the adders are combinational circuits and do not have a clock, but we target on the frequency the operands are provided to the adder by the testbench). This frequency is not randomly chosen. As discussed in chapter 2, the power consumption is linearly dependent on the clock frequency. However, for relatively low frequencies this is not entirely true. We found that (for our designs, in the specific technology we utilize) for frequencies below approximately 20 MHz, the leakage component becomes a quite significant portion of the total power consumption, because the dynamic power component becomes smaller. When the frequency is reduced even further (below 10 MHz), the leakage component will ultimately dominate the total power consumption. Since leakage power is a *constant*, the power consumption no longer scales linearly with the frequency. Since we design for low-power and obviously cannot afford high system frequencies, this is an important issue. However, the primary objective of the study in this chapter is to make a good and fair comparison between various adder types, regardless of the system speed, and therefore we are primarily interested in the dynamic power consumption. The frequency of 100 MHz is chosen, since the leakage component is then just a relatively small fraction of the total power consumption. In the next chapter we will revisit this topic. For now the synthesis results of the various adder types provide sufficient information to make a judgment about which adder can or cannot be utilized for our purpose.

### 4.4.3 Synthesis results

The synthesis results of the various types of adders can be found in table 4.1. In this chapter, random input vectors are utilized for the measurements, unless specifically mentioned otherwise. The **input drive strength** of the adder circuits is considered to be high, and the **capacitive load** on the circuit's output ports is zero. The input drive strength is the reciprocal of the output driver resistance (of the cell which drives the input of the circuit involved), as explained in chapter 3 (section 3.3). Synopsys DC assumes by default zero drive resistance on inputs ports, which is an unrealistic assumption. In order to override this default setting one can use two commands to manually set the input drive strength by a number, or by selecting a driving cell from the technology libraries (which was also explained in section 3.3). We chose for the last option, and drive every input of the adders by a simple buffer. This means that the input drive strength will be high, but not unrealistic. A zero-capacitive load means that we assume open output ports, enabling us to observe the purest results of the adders, since the input capacitance of any successive circuitry (negatively) influences the delay and power consumption of the adder itself. Of course, open output ports are unrealistic, but it gives us an impression about he maximum capabilities of the adders. Later on we will repeat the measurements with realistic capacitive loads. The technology utilized for this study is UMC 90nm Standard Purpose.

| Component | Delay | Area | Power |
|---|---|---|---|
| | [ns] | [units] | [$\mu$W] |
| RCA 4-bit | 0.30 | 116 | 6.55 |
| RCA 8-bit | 0.57 | 232 | 12.04 |
| RCA 16-bit | 1.11 | 464 | 23.67 |
| RCA 32-bit | 2.22 | 928 | 47.40 |
| CLA 4-bit | 0.21 | 189 | 11.41 |
| CLA 8-bit | 0.35 | 378 | 22.31 |
| CLA 16-bit (2-level lookahead) | 0.44 | 817 | 45.79 |
| CLA 32-bit (2-level lookahead) | 0.61 | 1634 | 91.99 |
| RCLA 8-bit | 0.40 | 316 | 17.54 |
| RCLA 16-bit | 0.74 | 632 | 34.11 |
| RCLA 32-bit | 1.45 | 1264 | 68.44 |
| CSK 8-bit (4x2-bit blocks) | 0.53 | 268 | 14.42 |
| CSK 16-bit (8x2-bit blocks) | 0.83 | 536 | 28.80 |
| CSK 16-bit (4x4-bit blocks) | 0.90 | 528 | 27.48 |
| CSK 32-bit (8x4-bit blocks) | 1.25 | 1080 | 56.83 |

Table 4.1: Synthesis results of adders (with open output ports)

Table 4.1 reports the synthesis results. In table 4.2 well-known metrics such as the **power-delay product**, **area-delay product** and **power-delay-area product** can be found. Table 4.3 illustrates the speedup and power/area overheads of the fast adders compared to the ripple-carry adder more clearly. Finally, figures 4.7, 4.8, and 4.9 depict the delay, area, and power consumption respectively, as a function of the adder width. The numbers that are displayed in boldface in table 4.2 represent the 'winners' in each category, which is the adder with the lowest PD, AD, or PDA product for various adder-widths.

#### 4.4.3.1 Conclusions

As depicted in figure 4.7, the delay of the RCA and RCLA scales linearly with the adder width. However, there is a significant difference between the slopes of these linearities. As expected, the delay of the RCA is worst and results in relatively large delays for 16- and 32-bit adders. The RCLA and CSK are close competitors within the 8- to 16-bit spectrum: for larger adder widths (32 bits or more), the CSK is advantageous in terms of delay (which is obvious, since the CSK's delay is $O(\sqrt{n})$). Since the required area and power consumption of the CSK is significantly lower than that of the RCLA, the CSK is a very interesting alternative. The CLA is one of the adder types that shows a non-linear delay trend, since the delay is logarithmically dependent on the adder width. However, in practice this is not a continuous logarithmic curve, but a discrete one, or in other words: a curve with twists (decreased slope) at discrete points along the $x$-axis, with a linear progression in between these points. These twists coincide with the transition of the adder to a higher-level lookahead structure. For example, a 4-bit CLA has only one level of lookahead logic. An 8-bit CLA is simply built by placing two 4-bit CLAs in

| Component | PD | AD | PDA |
|---|---|---|---|
| RCA 4-bit | **1.97** | **34.80** | **227.94** |
| RCA 8-bit | **6.86** | 132.24 | **1,592.17** |
| RCA 16-bit | 26.27 | 515.04 | **12,191.00** |
| RCA 32-bit | 105.23 | 2060.16 | 97,651.58 |
| CLA 2-bit | 0.55 | 10.36 | 40.71 |
| CLA 4-bit | 2.40 | 39.69 | 452.86 |
| CLA 8-bit | 7.81 | 132.30 | 2,951.61 |
| CLA 16-bit (2-level lookahead) | 20.15 | **359.48** | 16,460.59 |
| CLA 32-bit (2-level lookahead) | 56.11 | **996.74** | 91,690.11 |
| RCLA 8-bit | 7.02 | **126.40** | 2,217.06 |
| RCLA 16-bit | 25.24 | 467.68 | 15,952.56 |
| RCLA 32-bit | 99.24 | 1832.80 | 125,436.83 |
| CSK 8-bit (4x2-bit blocks) | 7.64 | 142.04 | 2,048.22 |
| CSK 16-bit (8x2-bit blocks) | **23.90** | 444.88 | 12,812.54 |
| CSK 16-bit (4x4-bit blocks) | 24.73 | 475.20 | 13,058.50 |
| CSK 32-bit (8x4-bit blocks) | **71.04** | **1350.00** | **76,720.50** |

Table 4.2: Performance metrics (open outputs)

| Component | Speedup | Area overhead | Power overhead |
|---|---|---|---|
| | [%] | [%] | [%] |
| CLA 2-bit | 14.29 | 27.59 | 28.43 |
| CLA 4-bit | 42.86 | 62.93 | 74.20 |
| CLA 8-bit | 62.86 | 62.93 | 85.30 |
| CLA 16-bit (2-level lookahead) | 152.27 | 76.08 | 93.45 |
| CLA 32-bit (2-level lookahead) | 263.93 | 76.08 | 94.07 |
| RCLA 8-bit | 42.50 | 36.21 | 45.68 |
| RCLA 16-bit | 50.00 | 36.21 | 44.11 |
| RCLA 32-bit | 53.10 | 36.21 | 44.39 |
| CSK 8-bit (4x2-bit blocks) | 7.55 | 15.52 | 19.77 |
| CSK 16-bit (8x2-bit blocks) | 33.73 | 15.52 | 21.67 |
| CSK 16-bit (4x4-bit blocks) | 23.33 | 13.80 | 16.10 |
| CSK 32-bit (8x4-bit blocks) | 77.60 | 16.38 | 19.89 |

Table 4.3: Speedup and overheads with respect to the RCA (open outputs)

series and therefore the delay characteristics are not altered. A 16-bit CLA is built with a 2-level lookahead structure, which explains the twist point in figure 4.7 at a width of 8 bits. A 64-bit CLA would be implemented with a 3-level lookahead structure, which would cause another twist point at a width of 32 bits. There is no doubt that the CLA is superior in speed, especially for larger adder sizes. However, the required area and power consumption of the CLA increases dramatically with larger widths, and is therefore less interesting for low-power designs. On the other hand the CSK appears to be very applicable in low-power and low-area budget designs. The 32-bit implementation

actually has the lowest PDA product of all adders, including the RCA. And apart from the low overheads, the speedup of the CSK is very significant. The twist point in the delay trend of the CSK at a width of 16 bits is caused by the transition from 2- to 4-bit blocks for the 32-bit CSK. After all, the delay of the CSK is $O(\sqrt{n})$, thus the efficiency of the CSK increases as the adder width increases.

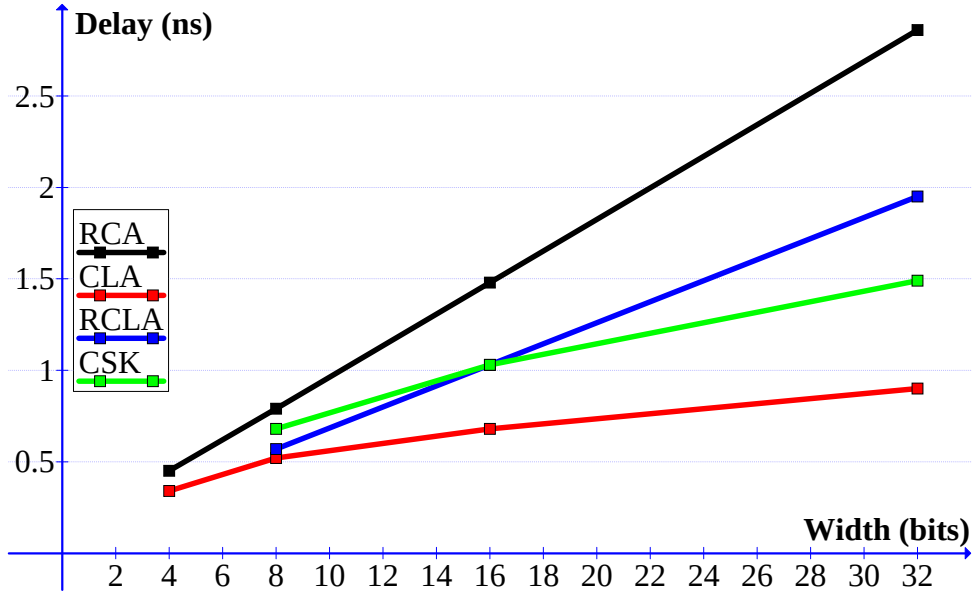


Figure 4.7: Delay as a function of the adder width

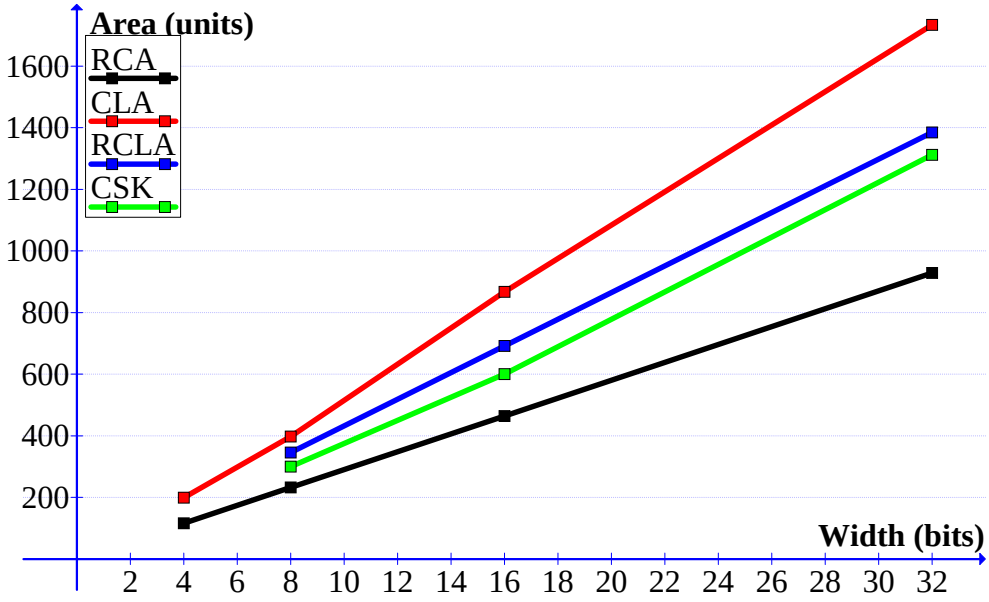


Figure 4.8: Area as a function of the adder width

Figures 4.8 and 4.9 depict the area requirements and power consumption of the four

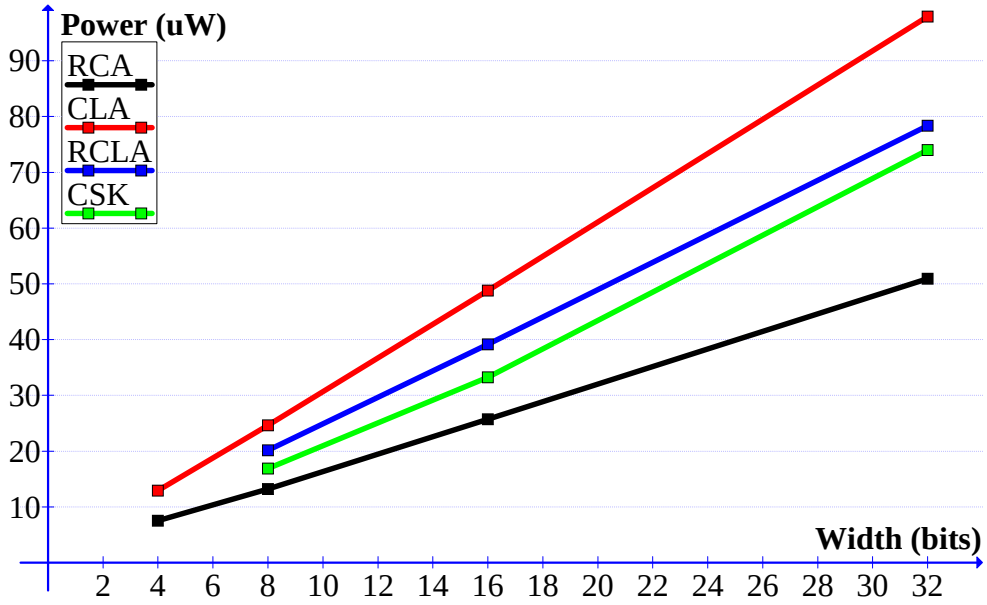


Figure 4.9: Power consumption as a function of the adder width

different adder types. The trend lines are linear, except for the cost trend of the CLA. The area requirements of the CLA increases with $O(n \cdot log(n))$. This is true because every time an extra level of lookahead logic is added, the costs increase even further (this explains the twist point in the area plot at a width of 8 bits). However, the delay grows less fast with the adder size than the costs. From that perspective, the CLA becomes more interesting for larger adder widths. The proof for that can be found in table 4.2, which shows that the PDA product grows less fast with increasing adder width than, for example the RCLA, an adder which scales linearly both in delay and costs. In both the area and power plot we see that the costs slightly increase when the transition is made from 2- to 4-bit blocks (twist point at a width of 16 bits). This is caused because the skip logic now receives four 'propagate' signals (from four full-adders), and since the skip logic has to be fast, the skip logic is implemented by gates with higher fan-ins, higher drive strengths, and thus higher costs. The RCA performs best in terms of area and power, followed by the CSK, the RCLA, and finally at significant distance, the CLA.

Based on the synthesis results, a number of other conclusions can be drawn. First, the 16-bit 4x4 CSK adder is less interesting than its 8x2 counterpart. The 8x2 version is faster, while requiring only little extra area and power. Therefore, the 16-bit 4x4 CSK adder is omitted in the remaining sections of this study. Secondly, the 16-bit CLA with 2-level lookahead logic turns out to be impractical. The primary reason for this is that, based on the previous results, the power consumption almost doubles when compared to the 16-bit RCA, and therefore is not suited to our low-power demands. Also, the 2-level CLA has an irregular structure and is therefore difficult to implement as a scalable adder.

| **Component** | **Delay** | **Area** | **Power** |
|---|---|---|---|
| Typical Case | [ns] | [units] | [$\mu$W] |
| RCA 4-bit | 0.45 | 116 | 7.54 |
| RCA 8-bit | 0.79 | 232 | 13.20 |
| RCA 16-bit | 1.48 | 464 | 25.71 |
| RCA 32-bit | 2.86 | 928 | 50.93 |
| CLA 4-bit | 0.34 | 199 | 12.94 |
| CLA 8-bit | 0.52 | 398 | 24.60 |
| CLA 16-bit | 0.68 | 867 | 48.77 |
| CLA 32-bit | 0.90 | 1734 | 97.92 |
| RCLA 8-bit | 0.57 | 346 | 20.16 |
| RCLA 16-bit | 1.03 | 692 | 39.17 |
| RCLA 32-bit | 1.95 | 1384 | 78.36 |
| CSK 8-bit (4x2-bit blocks) | 0.68 | 300 | 16.89 |
| CSK 16-bit (8x2-bit blocks) | 1.03 | 600 | 33.25 |
| CSK 32-bit (8x4-bit blocks) | 1.49 | 1312 | 74.02 |

Table 4.4: Synthesis results of adders (with $C_{out}$=0.01pF)

### 4.4.4 Impact of capacitive load

Now that we obtained the synthesis results of the different adder types, we investigated the impact on the delay and power consumption of the adders when they are implemented inside the surrounding circuitry: the ALU. However, at this point it is still unknown how the entire ALU is implemented, so it is impossible to come up with a *close estimate* of the capacitive load of the successive circuitry (which will be a bank of flip-flops, zero and overflow detection logic, error correction/detection logic, etc.). To be on the safe side, a *pessimistic rough estimate* is the best solution here. The capacitive load is set to 0.01 pF, which means in practice that each output drives at the most 5 to 7 logic cells (e.g. gates or flipflops). Tables 4.4, 4.5, and 4.6 show the results in the same way as in the previous (zero-load) situation. Also here, in table 4.5 the numbers that are displayed in boldface represent the 'winners' in each category, which is the adder with the lowest PD, AD, or PDA product for various adder-widths. It is clear from the results that the capacitive load has a serious impact on both the delay and the power consumption of the adders. E.g., the delay of the 32-bit RCA increases with almost 29% and the power consumption increases with 7.4% when we attach a load of 0.01 pF. Also area increases (slightly) for the CLA, RCLA, and CSK, since the synthesis tool optimizes the lookahead logic/skip logic to compensate for the speed loss, and therefore utilizes gates with higher drive strengths.

### 4.4.5 Comparison with literature

It is far from easy to make a fair comparison between the results of this adder study with other adder studies in the literature. There are a number of reasons for this. First, the exact implementations of the skip logic of the CSK and the lookahead logic of the CLA

| Component | PD-product | AD-product | PDA-product |
|---|---|---|---|
| RCA 4-bit | **2.71** | **38.28** | **314.66** |
| RCA 8-bit | 8.68 | 139.20 | **2,014.22** |
| RCA 16-bit | 32.22 | 528.96 | **14,948.41** |
| CLA 4-bit | 3.06 | 45.36 | 577.89 |
| CLA 8-bit | 9.03 | 139.86 | 3,413.98 |
| RCLA 8-bit | **8.07** | **129.56** | 18,444.29 |
| RCLA 16-bit | 29.18 | 480.32 | 18,444.29 |
| CSK 8-bit (4x2-bit blocks) | 9.35 | 150.08 | 2,504.84 |
| CSK 16-bit (8x2-bit blocks) | **29.02** | **466.32** | 15,556.44 |

Table 4.5: Performance metrics (with $C_{out}$=0.01pF)

| Component | Speedup | Area overhead | Power overhead |
|---|---|---|---|
| | [%] | [%] | [%] |
| CLA 4-bit | 37.50 | 62.93 | 54.99 |
| CLA 8-bit | 62.16 | 62.93 | 68.69 |
| RCLA 8-bit | 46.34 | 46.83 | 36.07 |
| RCLA 16-bit | 50.00 | 36.21 | 35.88 |
| CSK 8-bit (4x2-bit blocks) | 7.14 | 15.52 | 15.34 |
| CSK 16-bit (8x2-bit blocks) | 31.03 | 15.52 | 18.05 |

Table 4.6: Speedup and overheads with respect to the RCA (with $C_{out}$=0.01pF)

may differ, since different synthesis tools and different technology libraries are employed. Second, there are difficulties with comparing the area. Some papers report the area results only in *square micrometers*, while others report in *number of transistors*, *number of gates*, or *transition counts*. Reports in number of transistors or logic gates makes a comparison with our results possible, but has still no ideal one-to-one correspondence to our pre-layout logic units. Nevertheless, we have compared the results of three different 16-bit adders of the following studies:

- Our own study (Riemens): 90 nm CMOS, $V_{DD}$ = 1.0 Volt, $f$ = 100 MHz
- Nagendra's study [41]: 1.2 $\mu$m CMOS, $V_{DD}$ = 5.0 Volt, $f$ = 10 MHz
- Rabaey's study [9]: 2 $\mu$m CMOS, $V_{DD}$ = 10.0 Volt, $f$ = 2 MHz

Nagendra et al. and Rabaey et al. utilized, like we did, (pseudo) random input vectors for the power measurements. Our results are based on pre-layout results. The results of Nagendra's study and Rabaey's study are based on post-layout results (Nagendra et al. employed HSPICE for the circuit simulations, while Rabaey et al. employed CAzM). The comparison can be observed in table 4.7. The results are also represented in graphs for a more revealing view, depicted in figures 4.10, 4.11, and 4.12.

Note that area overheads and speedups, mentioned in the table, are calculated with respect to the RCA. Both the delay and area results of the CLA between our study and the study of Nagendra et al. show virtually no differences. The same holds true for

| | *Riemens* | | *Nagendra* | | *Rabaey* | |
|---|---|---|---|---|---|---|
| **Adder 16-bit** | **Area [units]** | **Overhead** | **Area [#tors[1]]** | **Overhead** | **Area [#gates[2]]** | **Overhead** |
| RCA | 464 | | 596 | | 144 | |
| CSK | 536 | 15.5% | 682 | 14.4% | 156 | 8.3% |
| CLA | 817 | 76.1% | 1038 | 74.2% | 200 | 38.9% |
| **Adder 16-bit** | **Delay [ns]** | **Speedup** | **Delay [ns]** | **Speedup** | **Delay [ns]** | **Speedup** |
| RCA | 1.11 | | 28.00 | | 54.27 | |
| CSK | 0.83 | 25.2% | 18.00 | 35.7% | 28.38 | 47.7% |
| CLA | 0.44 | 60.4% | 11.00 | 60.7% | 17.13 | 68.4% |
| **Adder 16-bit** | **Power [$\mu$W]** | **Overhead** | **Power [$\mu$W]** | **Overhead** | **Power [$\mu$W]** | **Overhead** |
| RCA | 23.67 | | 1.80 | | 117.00 | |
| CSK | 28.80 | 21.7% | 2000.00 | 11.1% | 109.00 | -6.8% |
| CLA | 45.79 | 93.5% | 2600.00 | 44.4% | 171.00 | 46.2% |

[1] Total number of transistors
[2] Total number of logic gates

Table 4.7: Comparison results with literature

the area results of the CSK. However, the speedup of the CSK is somewhat larger in Nagendra's study. Possibly this difference is caused by a more optimal implementation of the skip logic. The area results of the study of Rabaey et al. are hard to compare with our study. Here the number of gates is counted, but obviously not every gate has the same complexity and thus not the same size. We however do see the same trend: a small area increase for the CSK and a much larger one for the CLA. What is remarkable in this study is the delay of the CSK and CLA in the study of Rabaey et al. The speedup of both adders with respect to the RCA is significantly larger than in our study and the study of Nagendra et al. It is very difficult to draw a conclusion about the cause, as will be explained in the next paragraph.

The power consumption of the CSK and CLA in our study appears to be *considerably higher* than in the study of Nagendra et al. The overheads are about twice as high. It is extremely difficult to give an explanation for this. To begin with, Nagendra et al. employed a totally different technology and their results are acquired in a totally different manner. We have no idea which synthesis tool has been utilized (synthesis algorithms may vary significantly between different synthesis tools) and which technology library has been employed. Power results are acquired by a simulation tool, called HSPICE, *after* layout, while we presented only pre-layout results. Further, we have absolutely no insight in the different effort and optimization levels that have been set during synthesis by Nagendra et al. So, there are many variables that can influence the exact implementation of the design, and thus, the characteristics of the design. Although not impossible, it is, however, questionable if such a large difference in power overhead can be explained

by these implementation variables alone. A possible cause could be the size of the technology: maybe the skip logic and in particular the lookahead logic (which has high fan-ins and requires considerable drive strengths) can be implemented more efficiently in 1.2 $\mu$m than in 90 nm CMOS. One way to possibly find out more about the cause of the differences in power overhead is to perform synthesis and analysis of the adders again, by employing the same design tools and parameters we have utilized for our previous measurements, only now utilizing an older technology library (preferably UMC 1.2$\mu$m CMOS). This way we would be able to tell whether the size of the technology has any influence on the power overhead. Unfortunately, we do not have access to this technology library nor the actual designs available to Nagendra. In conclusion, we cannot provide a decisive answer explaining the large differences in power consumption. We should, however, at least be aware of the possibility that newer, smaller technologies alter —in particular— the power trend, when compared to older, larger technologies.

The power results of the adders in Rabaey's study show a remarkable phenomenon: the power consumption of the CSK is actually lower than that of the RCA. According to Rabaey et al., the power supply current of the CSK falls to zero faster than in the RCA, even though it is larger at the peak, causing the average power to be lower than that of the RCA. We have not been able to observe this phenomenon, and neither has Nagendra et al. Later on in the study of Rabaey et al., physical measurements are presented [9], which show a totally different picture. Therefore, we believe the results of Rabaey et al. are questionable.

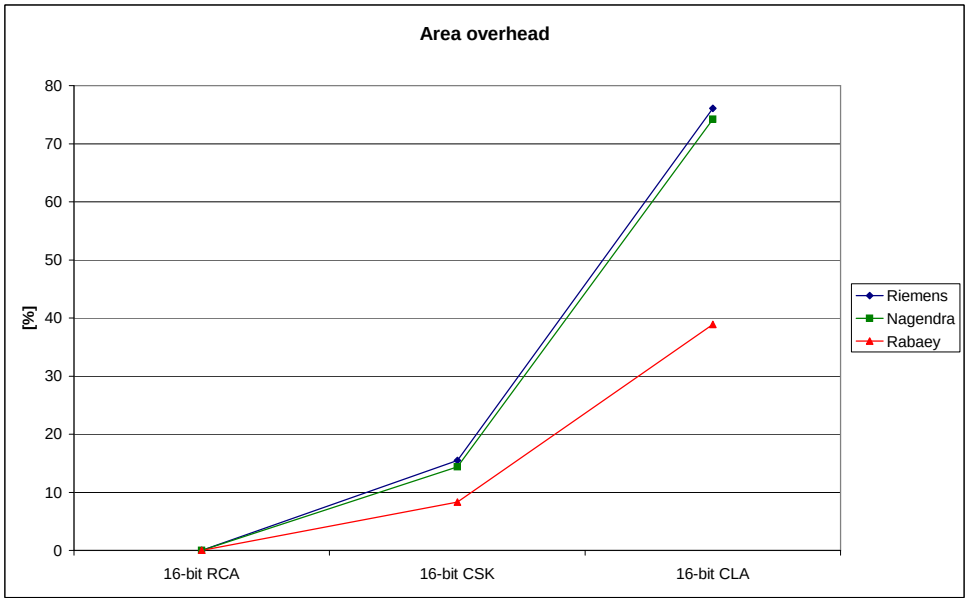


Figure 4.10: Area overheads of different adders reported in different studies

The difficulties we encountered during the comparison of our results with a couple of older studies from the literature emphasizes the value of our adder study. This adder study, gives every designer quickly an impression about the speed, area, and power consumption of the most common adders in relatively modern technology. Although

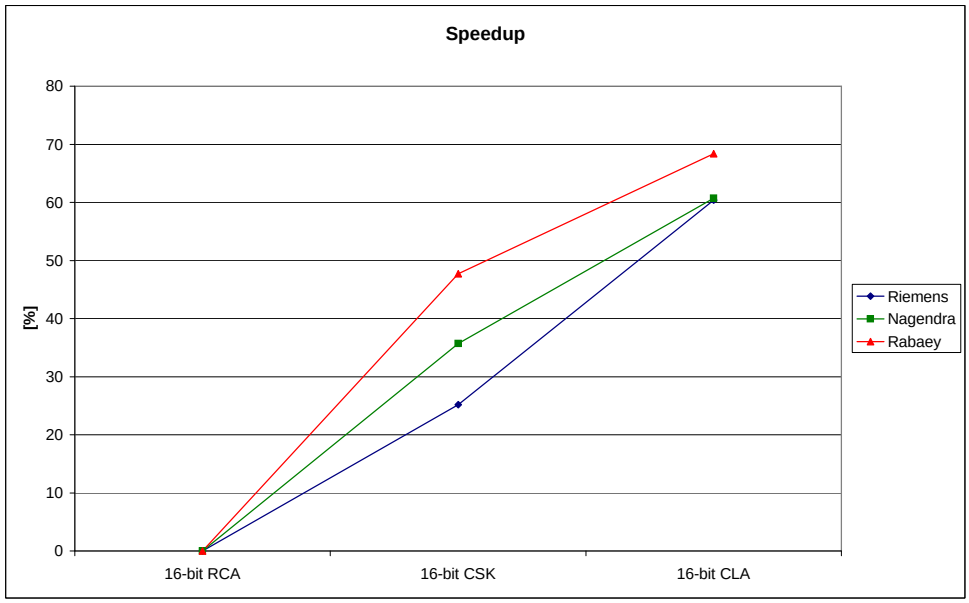


Figure 4.11: Speedups of different adders reported in different studies

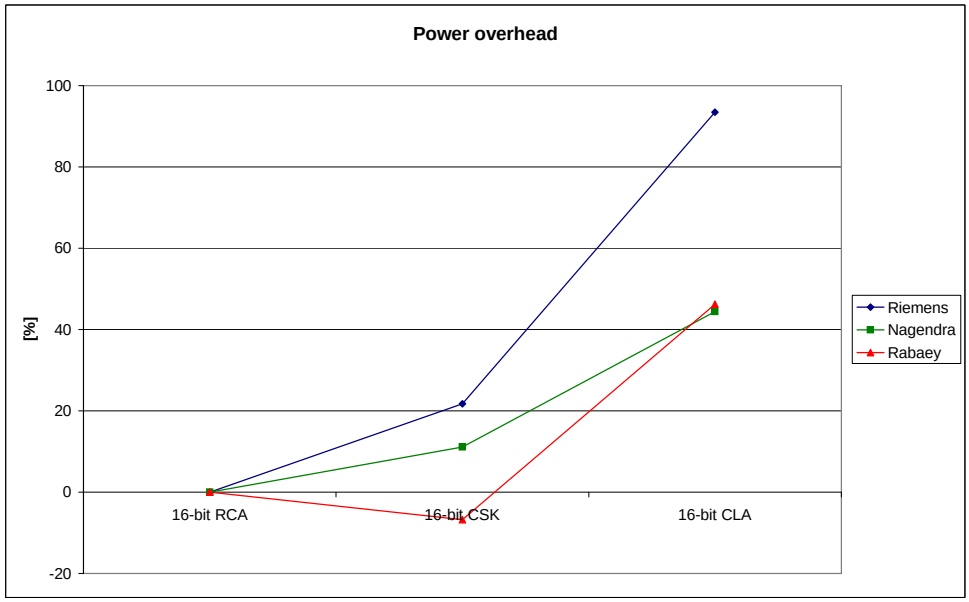


Figure 4.12: Power overheads of different adders reported in different studies

we have to look at pre-layout numbers for their relative value and not their absolute value, these results do provide a relatively good insight in the actual delay and power numbers in 90nm technology. Above all, we believe the pre-layout area estimates in logic units is actually a very good metric for depicting a trend, since all individual cells in the technology library have their own 'weight'. It is much more meaningful than reporting number of transitions or number of gates. Providing pre-layout trends has another advantage. Some papers report their post-layout results based on full-custom design,

while others employ semi-custom design. Then, a fair comparison cannot be made at all. We also showed that there are many variables involved in the design process which can lead to different characteristics of the ultimate design. A designer should always be aware of the fact that utilizing different tools, settings, constraints, libraries, and technologies will in most cases lead to deviations from the results that are reported in the literature. This is in fact a general remark, which holds true for all designs.

### 4.4.6 Impact of glitching

Since an RCA implemented in 90nm CMOS is most certainly fast enough for implementation in SiMS, and the RCA has the lowest power and area costs, the decision seems obvious. However, according to [40, 45], the RCA does not *necessarily* have the lowest power consumption. Since the RCA is the slowest adder of all, and in the worst-case scenario, the carry-ripples through all the full adders, it takes a relatively long time before the carry signal reaches the full adders further down the path, allowing glitches (spurious transitions) to occur. The reasoning is, if the carry signal reaches these full adders faster, at least some of the spurious transitions can be avoided. So far we only considered useful transitions. In the following power analysis of three different 8-bit adders we also account for glitching power (how this is done is explained in chapter 3). We use *random input vectors* and *worst-case input vectors* which cause the *most spurious transitions possible*. Normally, the worst-case input vectors for these adders would be like this:

```
Time Cin    Operand A           Operand B
0    0      0000000000000000    0000000000000000
1    1      1111111111111111    1111111111111111
2    0      0000000000000000    0000000000000000
3    1      1111111111111111    1111111111111111

pattern repeating...
```

With these inputs, every time (cycle) all inputs change value, and all output values change value (result and carry out bits), leading to the *maximum switching activity* inside the adder. However, if we look closely to what really happens inside the adders, we find that these inputs cause *zero* glitches, i.e. all transitions are useful transitions. This is easy to see, since when both operands (and $C_{in}$) are zero, no carry-rippling occurs, so no spurious transitions occur. The next cycle, all operand bits are ones, as well as the carry input. This addition leads to the maximum ripple-carry path. However, the rippling carry does not cause spurious transitions. The addition of operand A and B is performed bitwise in parallel, leading to the sum '00000000' and a carry out of '1'. Note that this is the same result as the previous addition. After the carry-rippling is complete, the result is '11111111'. Then, in the next cycle, the sum is immediately '00000000' again, but since no ripple-carry path exists, no spurious transitions exist. So, even though we achieve the highest possible switching activity inside the adder, these input vectors are definitely *not* suitable for testing the impact of glitching on the adder. Therefore we need the following input data:

```
Time Cin    Operand A           Operand B
0    1      1111111111111111    0000000000000000
1    0      0000000000000000    0000000000000000
2    1      0000000000000000    1111111111111111
3    0      0000000000000000    0000000000000000

pattern repeating...
```

These inputs all lead to the same result ('00000000'), so the useful switching power is zero, but the glitching power is now maximal. This means that *all transitions are spurious transitions* and no useful transitions exist (except for $C_{out}$). In the first transition, when A and B are added bitwise in parallel, the result is '11111111'. After the carry-ripple is complete, the result is '00000000'. That means that all intermediate values are changed by the rippling carry, and therefore the maximum amount of spurious transitions occur. The same holds true for the third addition. In between additions are performed with both operands (and $C_{in}$) being zero, in order to 'reset' the internal values. If we would not insert a zero-addition (addition 2 and 4), the switching activity would stop after the very first addition, since all additions lead to the same sum and carry values. The results of utilizing both random inputs and worst-case-glitching inputs can be observed in table 4.8.

| **Component** | **Random inputs** | **WC glitching** |
|---|---|---|
| Typical Case | [$\mu$W] | [$\mu$W] |
| RCA 8-bit | 17.20 (+30.3%) | 30.68 |
| CLA 8-bit | 31.03 (+26.1%) | 40.85 |
| CSK 8-bit | 23.04 (+36.4%) | 37.04 |

Table 4.8: Power results of 8-bit adders with random and worst-case inputs

It is immediately clear that the glitching power is a *significant* portion of the total power consumption, when we compare the results of the random inputs to the results in table 4.4. The column 'random inputs', shows the power results when we account for spurious transitions as well. The increase in power compared to our previous zero-delay measurements is also shown in percentages of the total power consumption. In the second column we shown the maximum glitching power. Note that this is pure glitching; there is no useful switching activity involved. These numbers make crystal clear that glitching power is *not* something we should ignore. We were, however, unable to prove that any fast adder performs better in terms of power than the RCA, when the worst case glitching scenario is simulated. According to [45], the CLA will in all situations consume more power than the RCA because of the high switching activity in the lookahead logic, which explains the result. However, also the CSK performs far worse than the RCA. Therefore, we hold on to the conclusion that the RCA is the adder with the *lowest* power consumption under *all* circumstances.

## 4.5 Conclusions

To determine which adder suits our needs best is obviously highly dependent on the environment it is intended for. Within the SiMS environment, high throughputs are not required (refer to chapter 1, section 1.3) and the power budget is extremely limited. The speed of the RCA in 90nm CMOS will most certainly be high enough, and since we were unable to prove that the CSK utilizes less power for worst-case inputs when considering the glitching power as well, the RCA appears to be the most suitable adder for our purpose. If the RCA proves to be not fast enough for the intended throughput (when it is utilized in other architectures), the CSK is a very good alternative. The area and power overheads of the CSK are small, while the speedup is (in particular for larger word widths) considerable. An interesting observation is that the PDA product of the CSK, for a 16-bit adder width, is virtually equal to that of the RCA (for the 32-bit version even better). Therefore the conclusion may be drawn that the CSK is a suitable adder type within low-power and resource-constrained architectures as well. The difficulties we encountered (reporting in different metrics, utilizing different design methodologies) during the comparison of our results with a couple of older studies from the literature emphasizes the value of our adder study. This adder study, gives every designer quickly an impression of the speed, area, and power consumption of the most common adders in relatively modern technology.

# 5 The Scalable Arithmetic Unit

## 5.1 Introduction

In this chapter we start with an explanation of the basic functionality of the arithmetic unit and a short survey of existing scalable arithmetic designs. Then, the general concept behind our idea of a gracefully-degradable arithmetic unit is presented. It is investigated if this approach is, in terms of power consumption and/or area requirements, advantageous over the most common method of increasing hardware reliability, which is hardware replication. After the scalable arithmetic unit is fully designed and optimized, the post-synthesis results of the design are compared to those of an arithmetic unit with a single adder and with duplicated adders. This way we can place the results of the scalable design between the two extremes: having no replicated hardware at all on one side, and having full hardware replication on the other side. Note that in this chapter we focus on the *potential* for reliability of the designs alone. We do not focus on error detection and correction yet (which is, of course, vital in the design of reliable systems). These topics will be discussed in detail in the next chapter. All three designs are tested for multiple clock frequencies and multiple technologies, in order to find the optimal design point of the ScAU. Further, the results of the ScAU are presented when other adder types than the RCA are utilized.

### 5.1.1 Basics of the arithmetic unit

The ALU consists of different parts: the logic unit, a comparator unit, a shift unit, and an arithmetic unit. The arithmetic unit contains the actual adder and is our topic of interest. A standard arithmetic unit consist of the following components:

- The adder
- A complementer (for subtraction)
- Zero and overflow computation circuitry

Thus, the arithmetic unit is in fact an adder/subtracter with a zero and an overflow output. However, in practice this arithmetic unit is often called an adder as well, to avoid confusion with multipliers and dividers. We also require the following registers:

- Input registers for the operands
- Output registers for the result
- Output registers for zero and overflow bit

Typically, the in- and output registers are not part of the arithmetic unit itself, but are pipeline registers in the micro-architecture. However, we specifically add them to the design of the arithmetic unit. It will be shown later, that for the scalable arithmetic unit, special output registers will be required. Apart from that, we need these registers anyhow in order to simulate the environment.

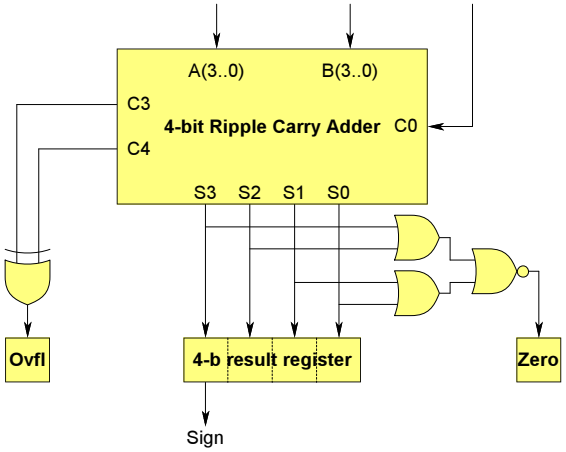


Figure 5.1: Zero, overflow, and sign detection

The zero and overflow detection can be implemented by a simple logic circuit [44], as depicted in figure 5.1. The figure shows the detection logic for a 4-bit adder. The labels C4 and C3 represent the adder's carry output as well as the last internal carry, respectively. The sign flag does not require a storage register, since the sign flag is basically the MSB of the result (since all numbers are in 2's complement representation), and is already stored in the result register. Modification of the adder to support subtractions is rather simple. A subtraction A-B requires an addition of the operand A and the 2's complement representation of the negated operand B. The operand B is first negated (inverted) by the complementer (which is implemented by a bank of XOR-gates, where one of the two inputs of each gate functions as the 'add/sub' control signal). The conversion from 1's complement to 2's complement representation requires the addition of '1', which is performed by providing a carry input to the adder.

### 5.1.2 Scalable structures

Scalable arithmetic units, ALUs and micro-architectures have been studied by a number of others. For example, Lee [46] presented a scalable booth-multiplier. The objective of this design is to *reduce the power consumption*, by scaling down the multiplier width, when both operands are small-sized (detected by a dynamic-range detection unit). A similar design is presented by Pfänder et al. [47] which also focuses on flexible word-length multiplication in order to save energy.

Kumar et al. [48] presented a design for an energy-efficient, high-performance architecture (including functional units, register files, caches, etc.), where the datapath is bit-sliced. The higher order bit-slices are only activated if required, which saves a significant amount of energy. Iyer and Marculescu [49] also worked on a scalable

micro-architecture. The resources are dynamically allocated, based on the needs of the running program. Here, each basic block of code (straight sequence of instructions without any jumps or branches) is analyzed (determining the parallelism and resource usage) and the processor is reconfigured for each basic block. Any FU that is not used is disabled by a clock-gating mechanism. Lin et al. [50] presented a low-power ALU cluster design, which is basically an ALU that is composed of multiple clusters (two arithmetic units, two multipliers and a divider), each placed onto a separate voltage island. That means, that any of the clusters can be power-gated when they are not utilized, which in this case results in higher power and energy savings than the regular clock-gating mechanism. Also Monteiro [51] reports about the so-called data-dependent power shut down, which is for example used in Intel Pentium processors. Even though these implementations do not have a flexible word-length, they can be considered as scalable designs in terms of power consumption.

Another form of power-scalable designs, exclusively used in arithmetic units, are mechanisms to *vary the precision* of the arithmetic operation. An example is given in [52] (using distributed arithmetic), however many different implementations can be found. The basic idea is that, whenever possible, the arithmetic unit does not produce exact results, but approximate ones.

But there are more motivations for scalable designs. Kursun et al. [53] describes the problems that occur, when the on-chip temperatures are increasing. It compromises the lifespan of the device and leads to slower operating speeds. **Dynamic thermal-management** techniques are developed to cope with these problems. These techniques limit the heat dissipation, to avoid critical temperatures. Scalable designs are employed for this purpose as well.

The primary objective of this thesis work, is to make the arithmetic unit scalable in order to increase the reliability and to *save power and/or area* compared to regular non-scalable fault-tolerant designs. In case of an error, we might be willing to degrade (downscale) the performance, but not the precision of the calculation. This of course, strongly depends on the specific application of the implant. In some applications precision may be of much greater importance than speed, while in other applications deadlines must be met at all costs, even if it means that the precision of the calculation must be downgraded. The literature provides little, if any, information about this specific purpose. However, the previously mentioned scalable-designs might be useful for this purpose as well. We will discuss both possibilities in this chapter: downgrading performance as well as downgrading precision. We focus, however, on downgrading performance.

## 5.2 Basic idea behind the scalable, gracefully-degradable arithmetic unit

Instead of **duplicating** the adder inside the arithmetic unit, we want to implement a single adder only, but now a version which is able to scale the width of the adder. For example, an $x$-bit adder can be **divided** into two $x/2$-bit *segments*. When only one of these segments fails, the arithmetic unit could proceed with the segment that

still functions correct (by this we create the necessary potential for reliable computing). In that case a one cycle $x$-bit addition or subtraction will be replaced by an $x/2$-bit operation, requiring two clock cycles. The word size of the arithmetic unit is downscaled, even though the word size of the architecture remains the same. Possibly, the adder can be divided into more than two segments, which increases the reliability even further (the arithmetic unit would be able to proceed with the computational work, even when multiple adder segments are damaged). However, this approach might result in serious problems regarding the required throughput (the more segments are shut down, the higher the latency will be). This scalable approach is an example of **graceful degradation**. We would like to know if this approach is viable for our purpose, and if it is advantageous over the common hardware duplication technique, especially in terms of power consumption and area.

## 5.3 Design of the Scalable Arithmetic Unit (ScAU)

The primary concerns in the design of the scalable arithmetic unit (or shortly: ScAU) are to design *an efficient multiplexing network to redirect the data* (operands and results) via alternative routes, to *store intermediate results* for multi-cycle operations, and to *keep track of the current cycle*. At the time the design of the ScAU was started, it was not sure yet if the word size of the SiMS architecture would be 8- or 16-bit. Apart from that, it is interesting to investigate how the ScAU scales with word size anyhow. Therefore, the ScAU is designed, synthesized, and analyzed for both 8- and 16-bit word sizes. In the following discussion an 8-bit version is assumed for the sake of convenience.

The basis of the ScAU are *two separate 4-bit adders* A and B. The lower 4-bit part of the operands are supplied to adder B, the upper four bits to adder A. The carry out signal of adder B is connected to the carry in of adder A. Each adder has its own 4-bit complementer and both the upper and lower half of the result are stored in a 8-bit output register. This provides the basis for a single-cycle 8-bit operation. When one of the adders gets damaged (which will be detected by the error-detection, as discussed in chapter 6), the data needs to be rerouted (by action of the error-correction hardware, also discussed in chapter 6). The adder which is still intact (adder A or B) needs to be provided with the lower half of the operands in the first cycle, and with the upper half of the operands in the second cycle (in downscaled mode). This is implemented by a **multiplexer network**. To keep track of the current cycle (cycle 1 or cycle 2), and to control the multiplexer network, a **controller** is implemented. Also, a multiplexer network is required for the output of the adders. The output of the functioning adder should provide its result to the lower half of the result register in the first cycle, and to the upper half of the result register. Finally, also flip-flops are required to store the value of the carry output of the functioning adder during the first cycle, to feed it to the carry input of the same adder in the second cycle. This means multiplexing at the carry inputs of both adders is required as well.

Besides all this, there is another important issue that requires attention. When the scalable AU is in downscaled mode, the AU will require **two clock cycles** per operation instead of one, and therefore the amount of energy per instruction increases dramatically. The most obvious way to limit the energy per instruction is to shut down the segment

that is not used (prevent switching activity). For sequential circuits, clock gating is the easiest way to disable a (sub)circuit. However, adders are pure combinational and therefore require another approach: **input-gating**. This means keeping all of the inputs of the circuit constant to prevent any switching activity and thus limiting dynamic power consumption inside the circuit. There are a number of methods to do this. In this study we investigate the utilization of latches, as well as tri-state buffers for this purpose.

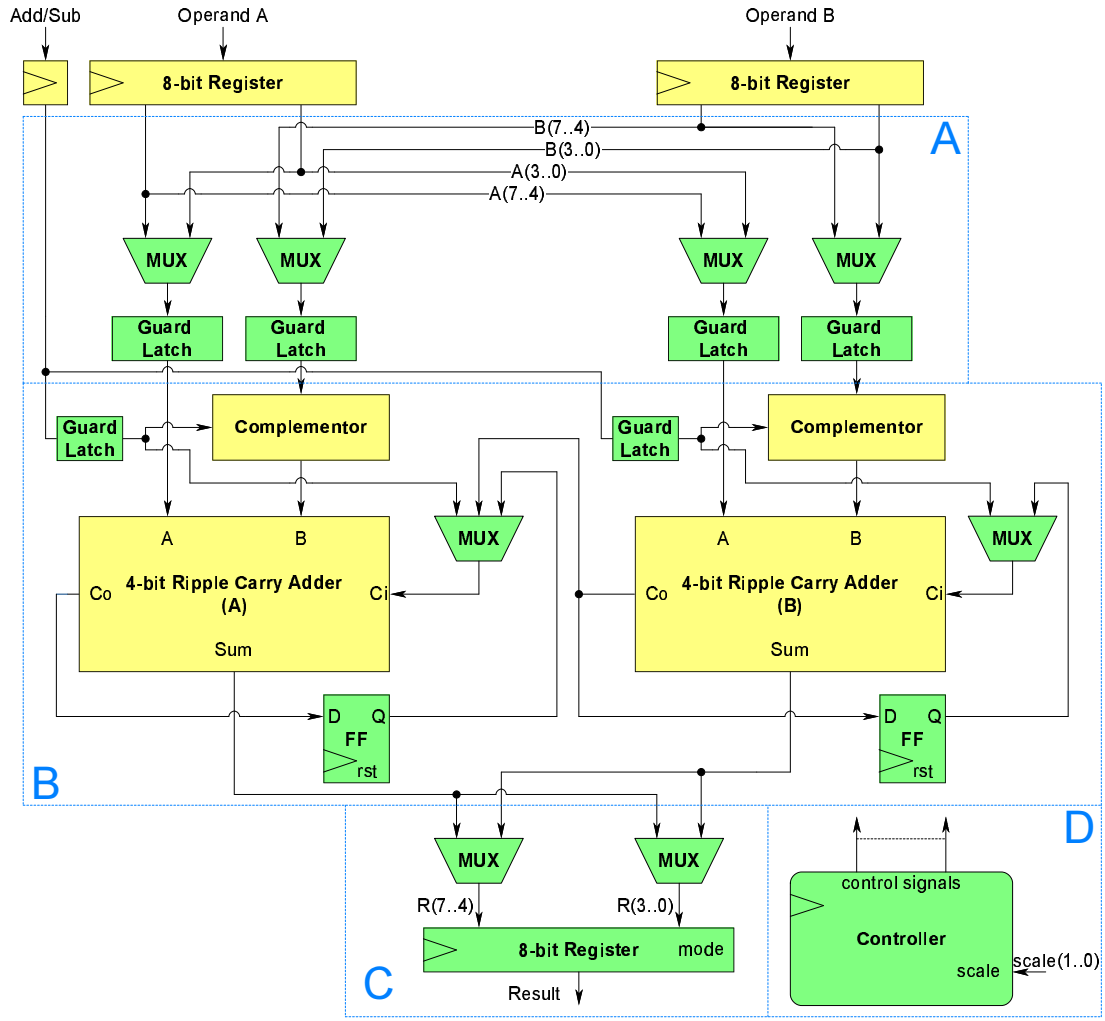


Figure 5.2: The scalable arithmetic unit (initial design)

The initial design of the (8-bit) scalable AU is depicted in figure 5.2. The design is roughly subdivided into four parts that already have been discussed: (A) the multiplexing and input-gating network, (B) the adder logic, (C) the output multiplexing logic, and (D) the control logic. It is easy to see that the overhead of the ScAU mainly lies in part A, C, and D. In this design, **guard latches** [54, 55] are used to disable unused combinational logic which, in this case, is any unused adder. Guard latches are one method to keep the inputs of an adder constant. Apart from the operand inputs, there is an input signal to select an addition or a subtraction as well as a 'scale' signal. This 2-bit signal is

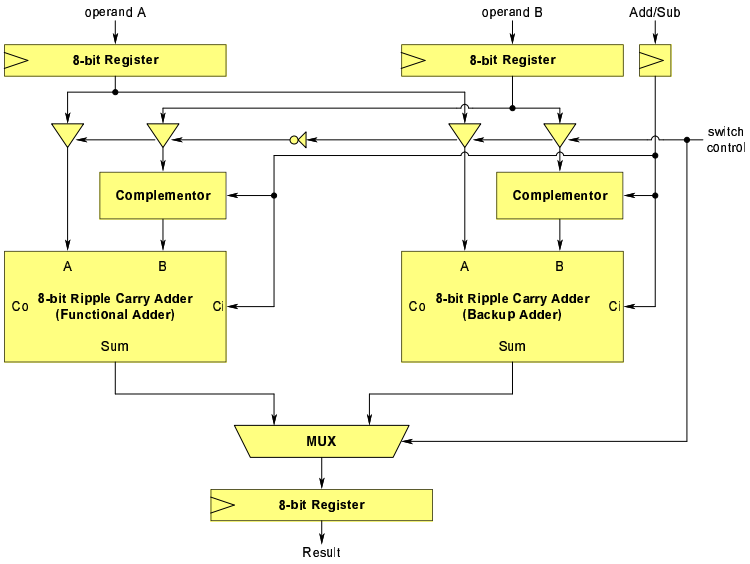


Figure 5.3: The duplicated adder structure (including input buffering)

used to control the scalability feature of the AU. When the signal is asserted to '11', the AU works in normal mode at full word width. When asserted to '10' or '01', the AU is downscaled and only the left (adder A) or right adder (adder B) is active, respectively. At this point we assume that the 'scale' signal is a properly externally controlled input of the design (controlled from the testbench). In the next chapter we will get to the generation of this signal. The controller keeps track of the current cycle (in downscaled mode) and controls all multiplexers and guard latches. The controller, along with all components that are controlled by it, are displayed in green. The exact functionality, including the FSM of the controller, will be discussed in section 5.5.2. Finally, a special result register is required. This register must be able to load all bits in parallel, as well as load the lower half and upper half of the result individually. When the scalable AU is in downscaled mode, the register should load the lower half of the result in the first cycle, store it, and load the upper half in the second cycle.

Obviously, there must be error-detection logic as well as error-correction logic present, in order to detect a failure and reconfigure the circuit. However, at this point in the process, the error detection/correction is omitted. This topic is covered in chapter 6. The synthesis, analysis, and optimization process of the ScAU is described in section 5.5.

## 5.4 The duplicated arithmetic unit (DAU)

Figure 5.3 depicts an example of the hardware replication technique. This arithmetic unit contains two adders. If the **primary adder** produces erroneous results, e.g. because of a hardware malfunction, the **backup adder** can be utilized to proceed with the computational work (this is basically the potential for reliable design, we referred to earlier). An output multiplexer can be utilized to select the output of the correct functioning adder. Not only the adders are replicated, but also the complementers and

zero and overflow detection logic (the latter is not depicted in the figure). In figure 5.3, both adder's inputs are input-gated. These input-gates are, however, optional. When these input-gates are omitted, both adders perform all calculations in parallel, and thus both dissipate the same amount of dynamic power. The result of one adder is utilized while the result of the other one is simply discarded. It might be beneficial to shut down the backup adder (which is essentially idle, since the result is not utilized). By gating the inputs, one of the adders can be disabled. This way, any switching activity inside this adder is prevented, which saves the dynamic power of the adder that produces no useful results (the power consumption in the backup adder is however still not zero, because of leakage currents). However, this approach has serious implications on the type of error-detection and -correction logic that can be used, and thus at this point it is not certain whether this approach is advantageous or not. However, here applies also that, at this point, error detection/correction is omitted and is covered in a later stage. Whether we should implement input-gating or not highly depends on the error-detection scheme which will be utilized.

## 5.5 Synthesis and comparison

### 5.5.1 Basics

For this study the following technology and operating conditions are utilized:

- Technology: UMC 90nm SP
- Operating voltage: 1.0 Volt
- Operating temperature: 25°C
- Clock frequency: 100 MHz
- Operating conditions: TCCOM (Typical-Case, Commercial use)
- Input test vectors: random ('optimistic worst-case scenario')

The UMC library contains three different operating conditions: best-case, typical-case, and worst-case. The operating condition that is set determines the operating voltage, temperature and the process scaling factor which accounts for variations in the outcome of the actual semiconductor manufacturing steps. The worst-case conditions are not utilized in this study since they assume an operating temperature of 125 degrees Celsius. In medical implants, such high temperatures are obviously unacceptable, since any heating of the surrounding tissue must be prevented at all times. The ScAU (as well as the AUs with the single and duplicated adder) is synthesized and analyzed for both 8- and 16-bit implementations. The synthesis script can be found in appendix A. Clock gating is employed to enable shutting down the entire ScAU during cycles where it is idle. But also the clock gating ensures that the input registers do not receive a clock cycle during the second cycle in downscaled mode, which prevents unnecessary switching activity.

### 5.5.2 Optimizations and final design

After synthesis and analysis it became clear that the overhead (in particular the power overhead) of the ScAU is, utilizing the implementation depicted in figure 5.2, quite dramatic. By *overhead* is meant the increase in area, delay, and power consumption *compared to the non-scalable single-adder AU*. Therefore, the scalable design has gone through a number of optimizations. The optimizations and their effect on power consumption and area can be found in table 5.1. Note that these results are based on the 8-bit implementation. The first optimization was replacing the guard latches by **tri-state buffers**. Instead of keeping the adder inputs constant by the guard latches (by storing the data), the adder inputs are now pulled to high-impedance state, which also prevents any switching activity in the adder itself, in case it should be disabled. The results show a *significant reduction* in both power and area. Therefore, the conclusion can be drawn that guard latching is too expensive for this purpose. This also means that tri-state buffers should also be used for the AU with the duplicated adder, in order to disable the adder that is currently unused.

| | **Optimization** | **Area** | **Power**[1] | **Power**[2] |
|---|---|---|---|---|
| 0 | Guard latching, standard multiplexing | 1751 | $> 100$ | $> 100$ |
| 1 | Adder inputs tri-state buffered | 1543 | 98.37 | 98.26 |
| 2 | Output register multiplexing optimized | 1546 | 94.47 | 94.66 |
| 3 | Input multiplexing by tri-state buffering | 1570 | 89.67 | 90.66 |
| 4 | Removing asynchronous resets | 1558 | 89.17 | 89.76 |
| 5 | Adding enable input to carry-store flip-flops | 1572 | 88.57 | 89.16 |
| 6 | Various controller optimizations | 1470 | 83.15 | 79.02 |
| 7 | Simpler control logic, carry-store latches | 1441 | 80.87 | 75.76 |

[1] *Normal (8-bit) mode*, [2] *Downscaled (4-bit) mode*

Table 5.1: Optimizations of the 8-bit ScAU

Another important optimization was the removal of the multiplexers at the inputs of the adders. Since the adder inputs now are buffered by a tri-state buffer, instead of a guard latch, the *multiplexing could as well be implemented by tri-state buffers alone*. Each of the adder's inputs, at this point, is fed by a 2-input multiplexer in series with a tri-state buffer (to disable the adder). Then, it is beneficial to remove the multiplexer and add another tri-state buffer, such that the two tri-state buffers operate as a multiplexer. Now, to disable the adder, both tri-state buffers must be in high-impedance state. Replacing the multiplexers by tri-state buffers decreases the power consumption significantly, however, it *increases the area*. The cause of this phenomenon will be explained later. Then, the asynchronous resets were removed from the carry-save flip-flops, since they are not truly necessary. This saves little area and power. Further, there is no need for the carry-store flip-flops to operate every single clock cycle. In fact, in normal full-width mode, there is no need to store the output carries at all. Only when the AU is in downscaled mode, the carry-store flip-flop of the adder that is still functioning is required. Thus, in all other situations the switching in the flip-flops is a waste of energy. Therefore, an *enable input* is added to the carry-store flip-flops, to

make sure they switch only when they are required to. This optimization increases the area slightly, because of the extra logic that is required, but saves some power.

The most effective optimization was the optimization of the controller (number 7 in the table). It appeared that the controller could be implemented with a much simpler FSM by more careful design (parts of the FSM turned out to be redundant), such that two of the flip-flops inside the controller could be omitted. This results in a substantial decrease in both power and area. This optimization includes the utilization of a *three bit* scale signal instead of two bits, reducing the number of outputs of the controller (and by that also the number of flip-flops) to only one, and replacing the carry-store flip-flops by **latches**. The 3-bit scale signal simplifies the controller, since this signal can now be used for most of the control signals of the tri-state buffers and multiplexers, and the only thing the controller is required to do is to keep track of the current cycle (i.e. in downscaled mode, when single-cycle operations change into multi-cycle operations). For example, the control of the tri-state buffers at the input of adder segment A (the left segment, normally computing the upper byte of the result) is coded as follows:

```
--normal mode mode and downscaled mode (cycle 2)
input1_adder_A <= operand_X_high_order_byte when
  (scale(2)='1' or (cycle='1' and scale(1)='1'))
            else (others => 'Z');
input2_adder_A <= operand_Y_high_order_byte when
  (scale(2)='1' or (cycle='1' and scale(1)='1'))
            else (others => 'Z');

--downscaled mode (cycle 1)
input1_adder_A <= operand_X_low_order_byte  when
  (cycle='0' and scale(1)='1')
            else (others => 'Z');
input2_adder_A <= operand_Y_low_order_byte  when
  (cycle='0' and scale(1)='1')
            else (others => 'Z');
```

This piece of code tells a lot about how the input multiplexing/gating is controlled, and what the exact function of the 'scale' and 'cycle' signals is. In normal mode, and in the second cycle in downscaled mode, the left adder segment has to add the upper bytes of the operands X and Y. In normal mode 'scale=100', thus 'scale(2)=1'. We only have to test for the value of this single signal, since all three signals 'scale(2..0)' are mutually exclusive. In downscaled mode we have to check if the left or the right adder is active. To test if the left adder is active, we should check if 'scale(1)=1'. If the right adder is active (then 'scale(0)=1'), the left adder segment must not receive any inputs, so all tri-state buffers should close ('HiZ' state, or high impedance state). Now we only have to check whether we are in the first cycle or in the second. This is signaled by the 'cycle' signal, coming from the controller. If 'cycle=0' we are in cycle 1, if 'cycle=1' we are in cycle 2. The 'scale' and 'cycle' signals are employed to control every tri-state buffer, multiplexer, and other components that need to be controlled in the ScAU in a similar way.

The FSM of the cycle-controller is depicted in figure 5.4. Finally, using latches instead of flip-flops for storing the output carries of the adders saves a considerable

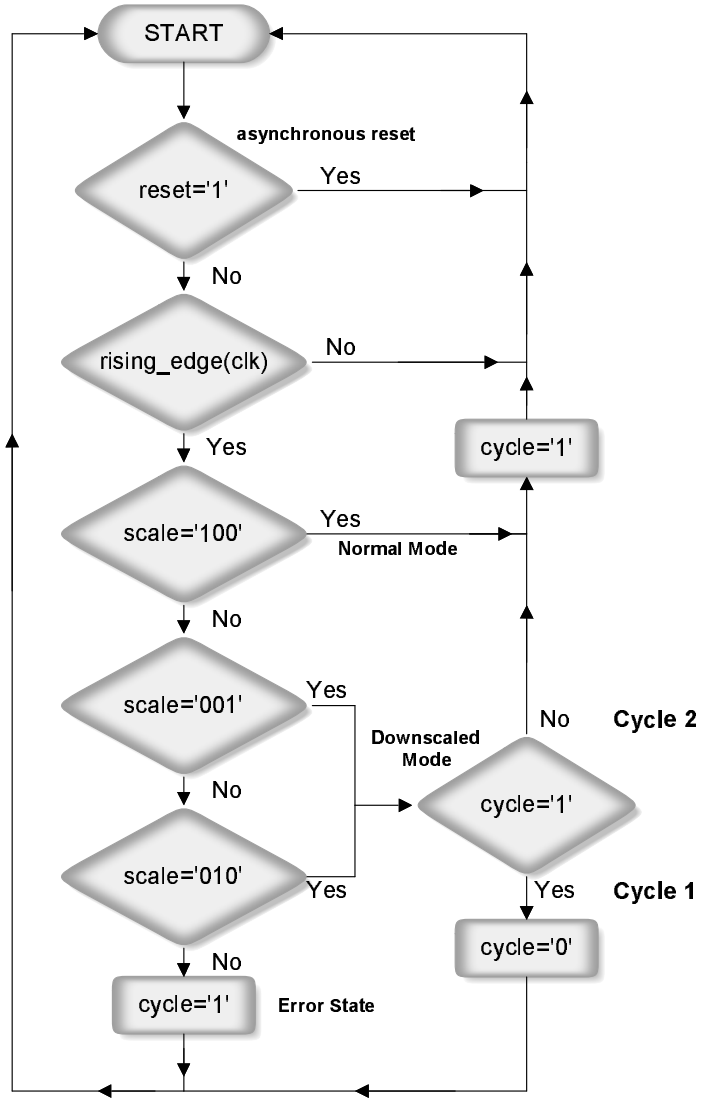


Figure 5.4: FSM of cycle-controller

amount of area and power as well, since latches are less complex devices than flip-flops. After synthesis of the initial design we passed several times through the steps of (1) closely analyzing the results and thinking about possible optimizations, (2) modifying the design, and (3) re-synthesizing, until we were unable to optimize the design any further.

The optimized ScAU is depicted in figure 5.5 (here also the zero and overflow detection circuitry is depicted). The figure shows the tri-state buffers that are utilized as multiplexing network for the inputs of the adders as well as input-gates for the individual adder segments. However, for the multiplexing network on the outputs of the adders, regular **gate-based** multiplexers are used. First we intended to replace all gate-based multiplexers by tri-state-based versions, since in the literature various references can be

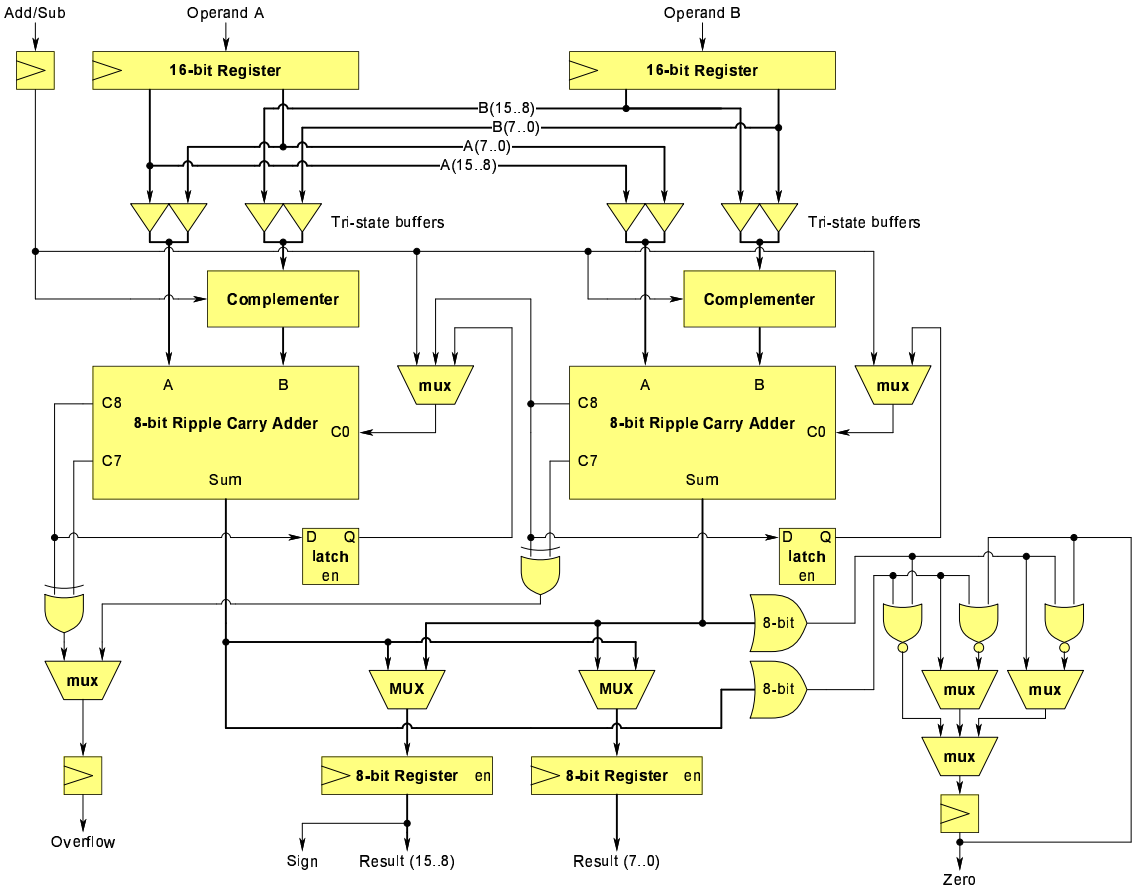


Figure 5.5: The optimized scalable arithmetic unit

found which claim that the tri-state-based multiplexer is an efficient implementation. Intuitively, this implementation seems indeed cheaper than the gate-based approach. However, we found that, at least in the case of 2-input multiplexers, a tri-state-based multiplexer is *not* advantageous over the regular gate-based multiplexer. Both area and power consumption are higher for a tri-state-based multiplexer, than it is for a gate-based one. The implementation of both versions is depicted in figure 5.6. On the left, the optimal implementation of the gate-based multiplexer is depicted. Every gate is annotated with a number, which represents the *number of transistors* which are required for that particular gate [56]. In total, the gate-based multiplexer consists of 14 transistors. On the right side, the tri-state-based multiplexer is depicted. A tri-state buffer does not exist as a library cell in UMC technology, it is built by placing a tri-state inverter in series with a static inverter. The tri-state inverter and the static inverter are depicted at transistor level in figure 5.7.

The tri-state inverter consists of 6 transistors, 4 are depicted, plus another two which are required for inverting the control signal 'C'. In total, the tri-state-based multiplexer consists of 18 transistors, which is more than the gate-based version. Especially when the inputs are e.g. 16 bits wide, a significant difference in area and power consumption can be observed. A tri-state-based multiplexer is most likely only efficient for multiplexers

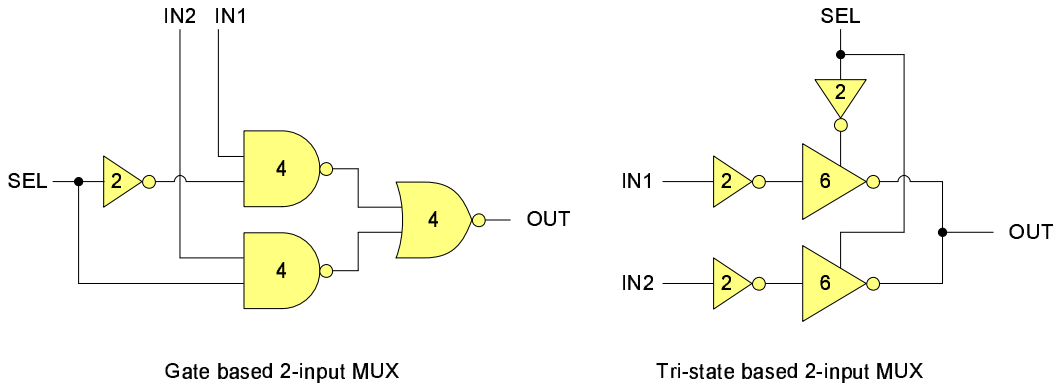


Figure 5.6: A 2-input gate- and a tri-state-based multiplexer

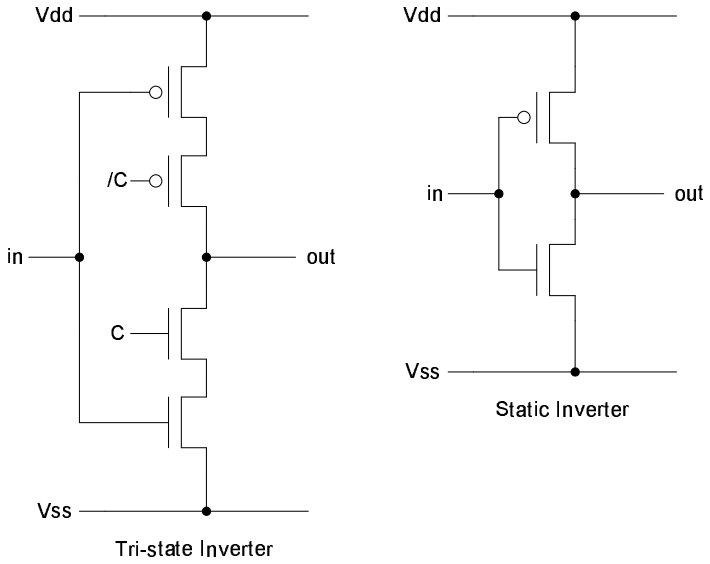


Figure 5.7: A tri-state inverter and static inverter implemented in CMOS

with a large number of inputs (and, of course in our specific case where multiplexing is *combined* with input-gating). In UMC 90nm technology, the area ratio between the gate-based and tri-state-based multiplexer is not 14:18, but close to 14:26. It appears that the tri-state buffer (in terms of area) is not implemented as efficiently as possible. However, the trend holds true. The large ratio has, however, the effect that the area increases, when the multiplexer in series with the tri-state buffer is replaced by two tri-state buffers, that handle both the multiplexing and input-gating of the adder inputs, as mentioned previously. The reason why we chose this approach nevertheless, is because the power consumption *does* decline significantly. The exact dataflow in the ScAU in normal mode is depicted in figure 5.8. The dataflow in downscaled mode is depicted in figures 5.9 (cycle 1) and 5.10 (cycle 2).

The synthesis results of the 8- and 16-bit ScAU, as well as the AUs with the single and duplicated adder, are depicted in figures 5.11, 5.12, and 5.13. The **AU** refers

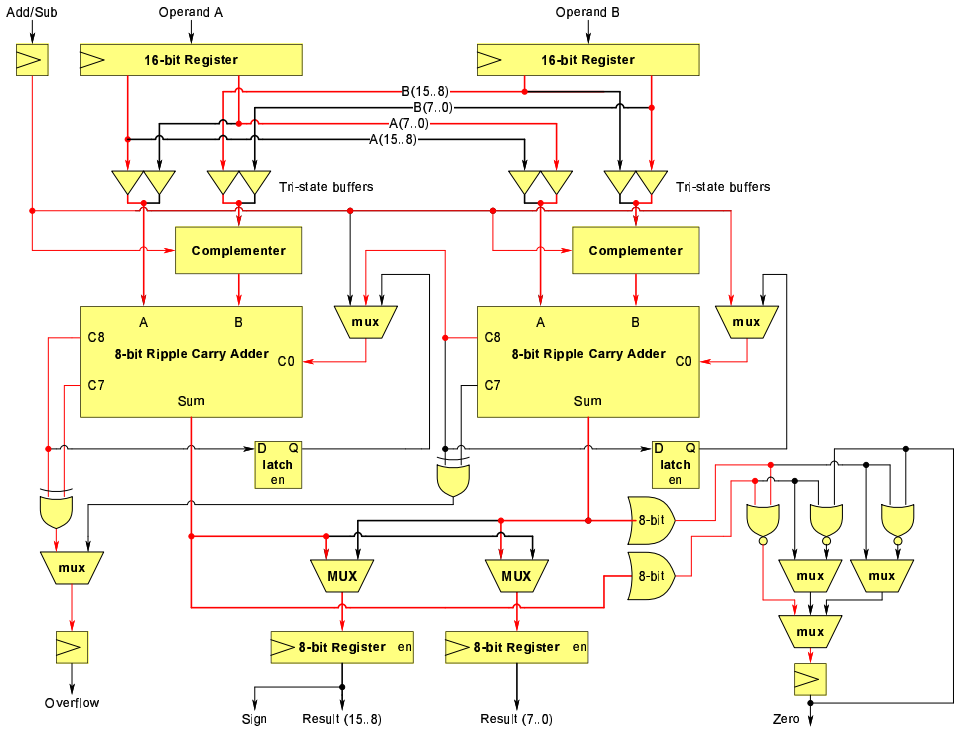


Figure 5.8: Dataflow in normal, full width operation

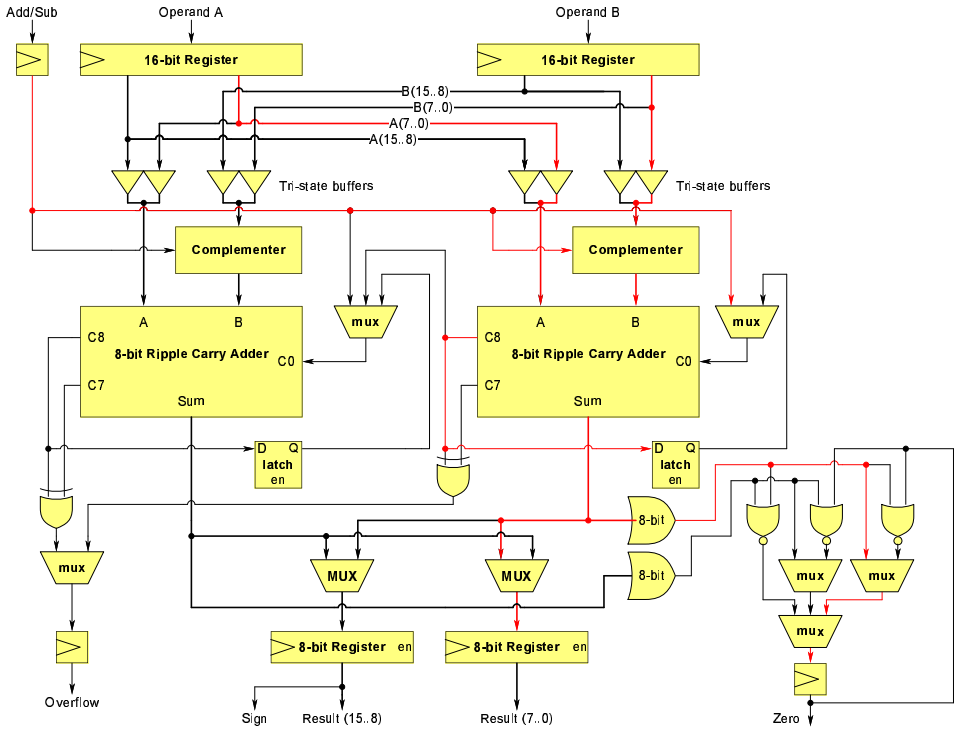


Figure 5.9: Dataflow in downscaled mode during first cycle

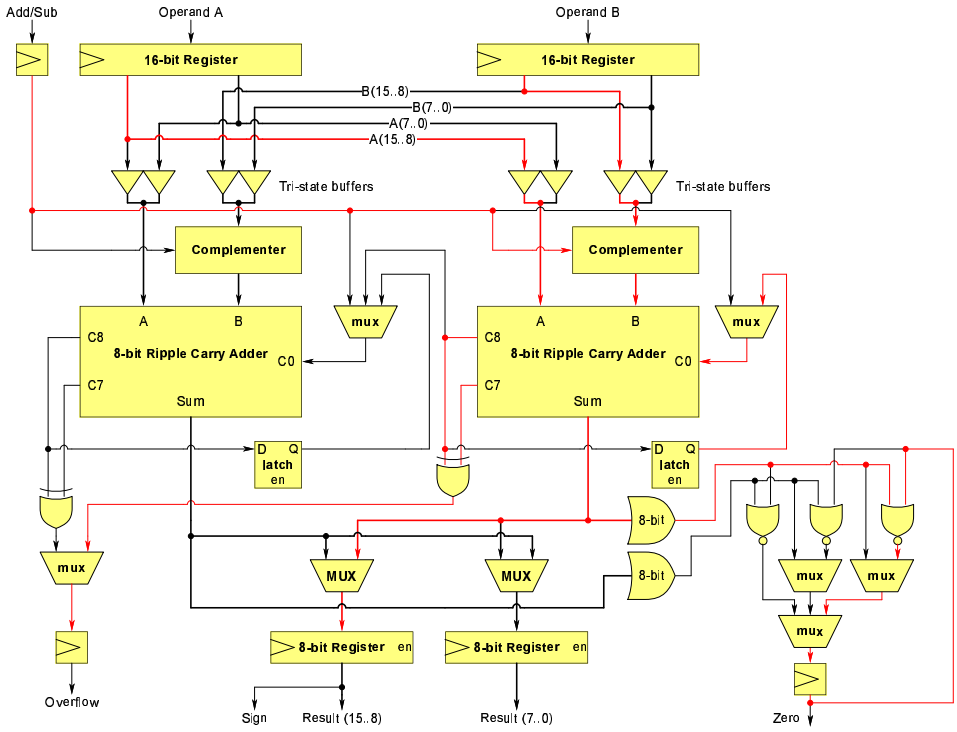


Figure 5.10: Dataflow in downscaled mode during second cycle

to the arithmetic unit implemented with a single adder, where **DAU** refers to the arithmetic unit implemented with a duplicated adder structure (where both adders are simultaneously active). The **DAU-RAS** is the arithmetic unit with duplicated adder where the backup adder is disabled by tri-state buffers (**RAS** stands for Redundant Adder Shut down). As mentioned before, the acronym **ScAU** refers to the scalable arithmetic unit, where '*nm*' stands for normal, full-width mode, and '*dm*' for downscaled mode. These acronyms are employed throughout the rest of the thesis.

From these figures, a number of conclusions can be drawn. First, the 16-bit versions are evaluated. Obviously, the DAU and DAU-RAS show an increase in area and power consumption, compared to the single adder. The area and power consumption does, however, not double, because the input and output registers are not replicated. Only the **combinational logic** is replicated. Actually, the difference in area and power overhead between the single-adder AU and DAU is relatively small, which means that the sequential logic dominates both the area and power consumption. When the unused adder is disabled, as is the case in the in DAU-RAS, the overall power consumption reduces significantly, but area increases to a much larger extent due to the required tri-state buffers.

The ScAU requires slightly more area than DAU, but *significantly less* than DAU-RAS. Even though the power consumption of the ScAU in normal mode is significantly less than that of DAU, it is slightly higher than it is of the DAU-RAS, which is optimized for power by disabling the backup adder. This is not what we hoped for, but also not very surprising, since when the 16-bit backup adder in the DAU-RAS

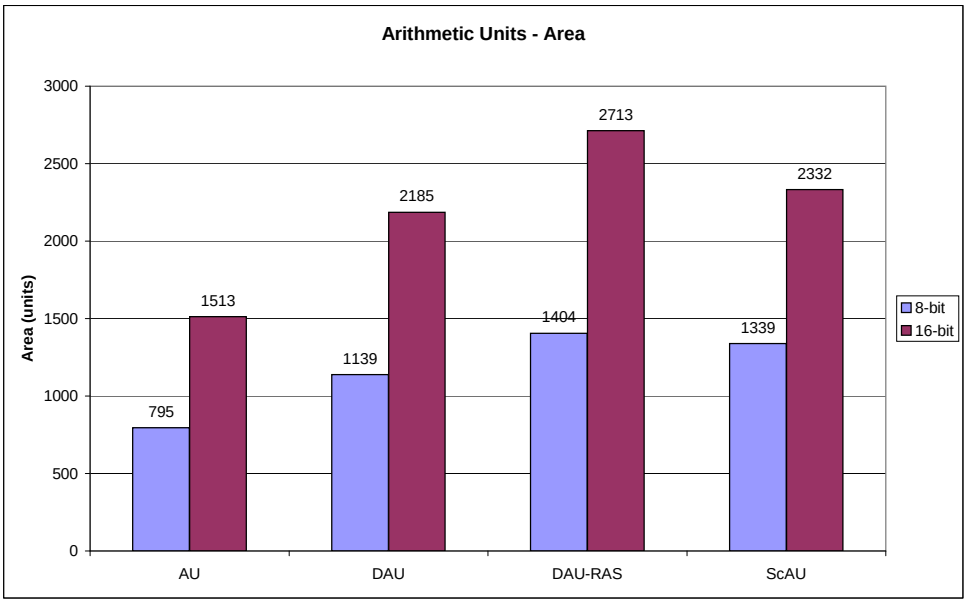


Figure 5.11: 8- and 16-bit Arithmetic Units - Area

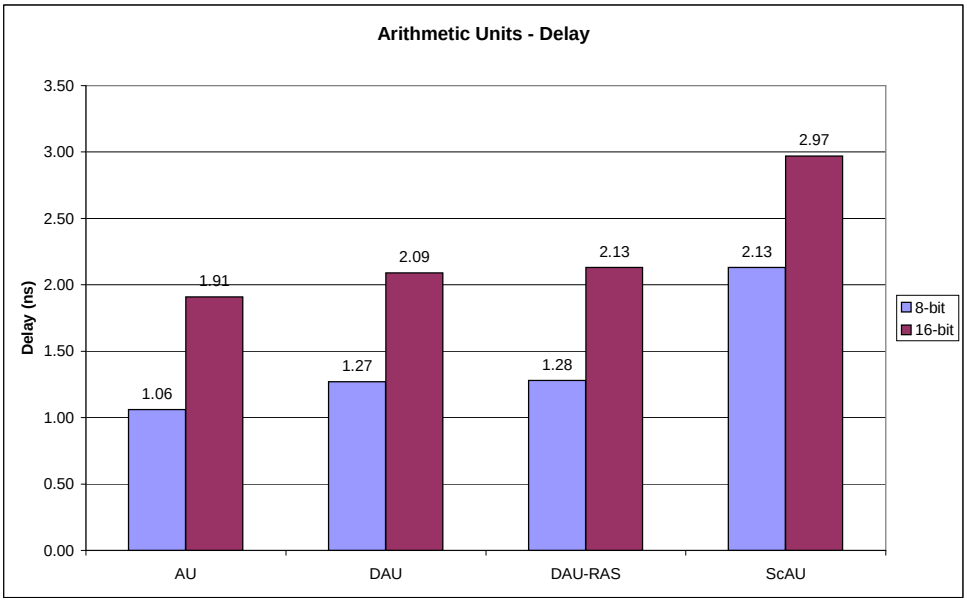


Figure 5.12: 8- and 16-bit Arithmetic Units - Delay

is disabled, there is no dynamic power consumed (only a small amount of static power is dissipated). Further, there is not much additional hardware which consumes power apart from the output multiplexer and the tri-state buffers (32 tri-states in total, 16 of them in 'pass' state, 16 of them in 'HiZ' state) at the inputs. The ScAU on the other hand, does have the same power consumption for the adder part (2x8-bit adder segments $\equiv$ 16-bit adder), the same power consumption of tri-state buffers (also 32 tri-states in total, 16 of them in 'pass' state, 16 of them in 'HiZ' state), but has on top of that power

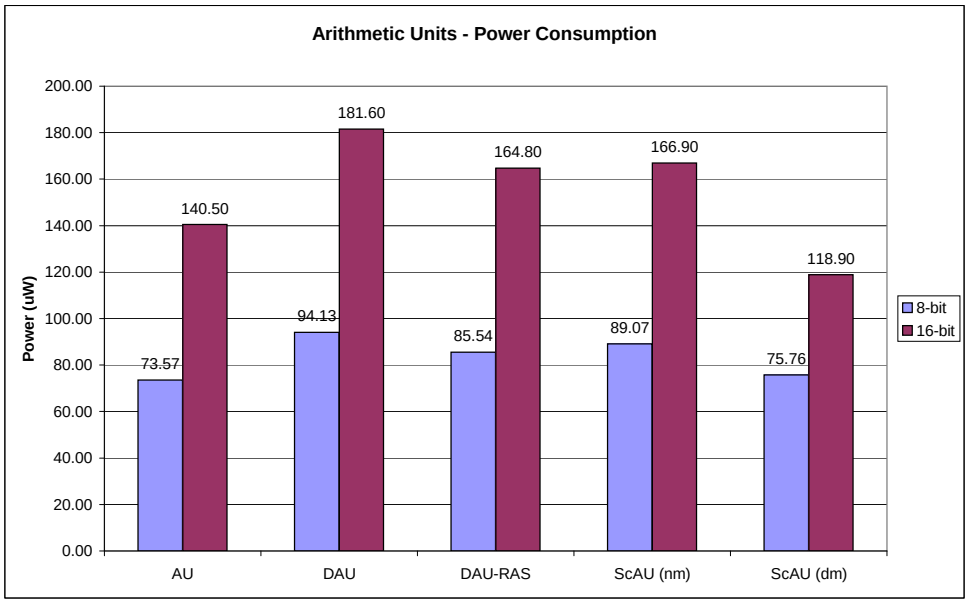


Figure 5.13: 8- and 16-bit Arithmetic Units - Power

dissipated by the additional logic (many multiplexers, a cycle-controller, control logic, and latches) which is necessary for the scalability feature. This is the reason why the ScAU in normal mode cannot dissipate less power than the DAU-RAS.

The power consumption of the ScAU in downscaled mode (dm) is significantly lower than in normal mode (nm), significantly lower than that of the DAUs, and only slightly higher than that of the single-adder AU. Unfortunately, the ScAU shows a dramatic *increase in delay*, because the additional logic resides in the adder's critical path. In chapter 6 we will show that this problem —which appears at this stage in the design process— is irrelevant, since in the next stage (adding error detection and correction) the critical path will be shortened again.

| **ScAU** | **8-bit** | **16-bit** |
|---|---|---|
| Delay | 100.9% | 55.5% |
| Area | 68.4% | 54.1% |
| Power (nm) | 23.2% | 18.8% |
| Power (dm) | -4.4% | -15.4% |

Table 5.2: Overheads of ScAU with respect to single-adder AU

The 8-bit ScAU is performing significantly worse than its 16-bit counterpart, as can be observed in table 5.2. The area as well as the power overheads (in percentages), with respect to the single-adder AU, are significantly higher. Therefore, the conclusion is drawn that the 8-bit ScAU is not efficient. If we cannot save large amounts of area without increasing the power consumption too much compared to the DAU-RAS, the ScAU is not of much use. The reason why the 16-bit version performs significantly better, is because a large portion of the additional hardware (enabling the scalability

feature) is *constant*, regardless of the adder width. Only the amount of tri-state buffers will increase linearly with the word width.

In table 5.3 the synthesis results (earlier depicted in figures 5.11, 5.12, and 5.13) of all 16-bit arithmetic unit designs are displayed. Table 5.4 shows the area, delay, and power overheads of the DAU, DAU-RAS, and ScAU, compared to the non-scalable, single-adder AU. What we see here is that the DAU creates a potential for reliable computing by implementing a second adder, however this comes at a very large price in terms of power overhead. The DAU-RAS does the same, only here the backup adder is disabled to save power. It is clear from the table that the power overhead then significantly reduces (from 73.8 to 57.7 percent, which is a difference of 16.1%). On the other hand, the DAU-RAS does further increase the area overhead by a large amount (from 44.4 to 79.3 percent, an increase of 35.0%). The ScAU has a much lower power overhead than the DAU (-14.1%), only slightly higher than the DAU-RAS (+2.0%), and has a significant lower area overhead than the DAU-RAS (79.3 minus 54.1 percent, which comes down to a reduction of 25.2%).

| **Design** | **Area** | **Delay** | **Power** |
|---|---|---|---|
| | [units] | [ns] | [$\mu$W] |
| AU | 1513 | 1.91 | 104.5 |
| DAU | 2185 | 2.09 | 181.6 |
| DAU-RAS | 2713 | 2.13 | 164.8 |
| ScAU(nm) | 2332 | 2.97 | 166.9 |

Table 5.3: Synthesis results of different 16-bits AUs

| **Design** | **Area OH** | **Delay OH** | **Power OH** |
|---|---|---|---|
| | [%] | [%] | [%] |
| DAU | 44.4 | 9.4 | 73.8 |
| DAU-RAS | 79.3 | 11.5 | 57.7 |
| ScAU(nm) | 54.1 | 55.5 | 59.7 |

Table 5.4: Overheads of different 16-bits AUs

If we compare real numbers (instead of comparing overheads), the power consumption of the ScAU is 8.1% lower than the DAU and 1.3% higher than the DAU-RAS. The area of the ScAU is 6.7% higher than the DAU and 14.0% lower than the DAU-RAS. The delay of the ScAU is 39.4% higher than the DAU-RAS.

Since the ScAU requires a significant amount of hardware to enable the scalability feature, we decided *not* to examine a ScAU with four segments. This would require even more additional hardware and would only be efficient when the adder width is large enough. For an 8- or 16-bit adder, where each of the four segments is no larger than 2 or 4 bits, this will most certainly not be the case.

| Frequency: | **100 MHz** [$\mu$W] | **20 MHz** [$\mu$W] | **10 MHz** [$\mu$W] |
|---|---|---|---|
| AU | 140.50 | 29.46 | 15.61 |
| DAU | 181.60 | 38.29 | 20.39 |
| DAU-RAS | 164.80 | 35.45 | 19.29 |
| ScAU (nm) | 166.60 | 35.41 | 18.99 |
| ScAU (dm) | 118.90 | 27.79 | 14.20 |

Table 5.5: Power consumption AUs for different frequencies

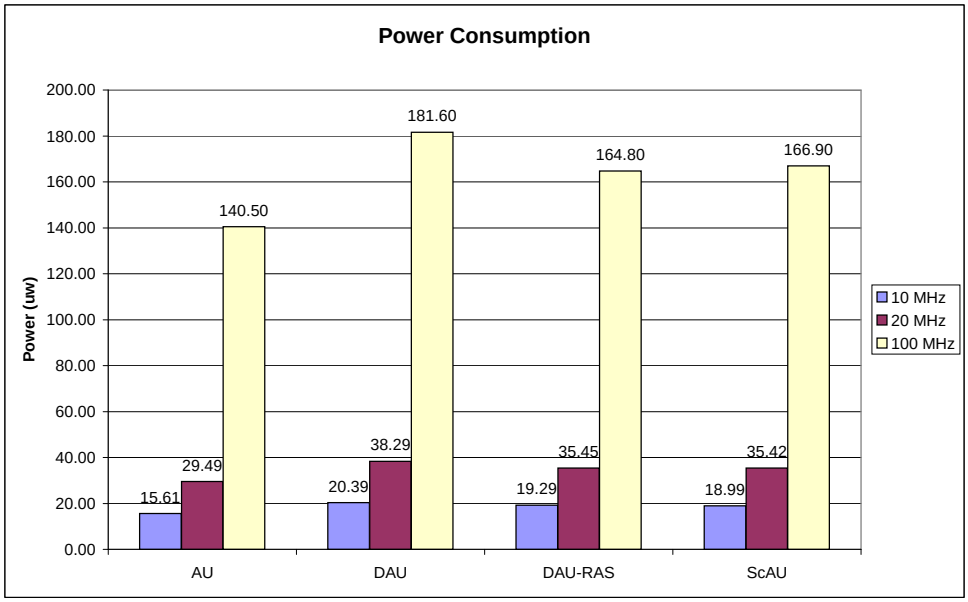


Figure 5.14: Power consumption of arithmetic units as a function of frequency

### 5.5.3 Power versus frequency

The power consumption of the different AUs has been tested for three different clock frequencies, as can be observed in table 5.5 and figure 5.14. At 100 MHz, the static power component is only a very small portion of the total power consumption and is negligible (depicted in figures 5.15 and 5.16). Even though static power is constant, for significantly lower frequencies, such as 10 or 20 MHz, the static-power component becomes a factor of importance since the dynamic power is linearly dependent on the frequency. The lower the frequency, the lower the dynamic power consumption, and the higher the impact is of the static power. As said before, static power does not depend on frequency, but it does depend highly on the size of the design. The measurements clearly show that the ScAU becomes more interesting for lower frequencies. The reason for this is that the ScAU is smaller in size than the DAU-RAS and thus has a smaller static power component. While the ScAU was at a slight disadvantage at 100 MHz, at 20 MHz the difference in power consumption between the ScAU and the DAU-RAS is close to zero. At 10 MHz, the ScAU performs actually better (1.6% less power). Although

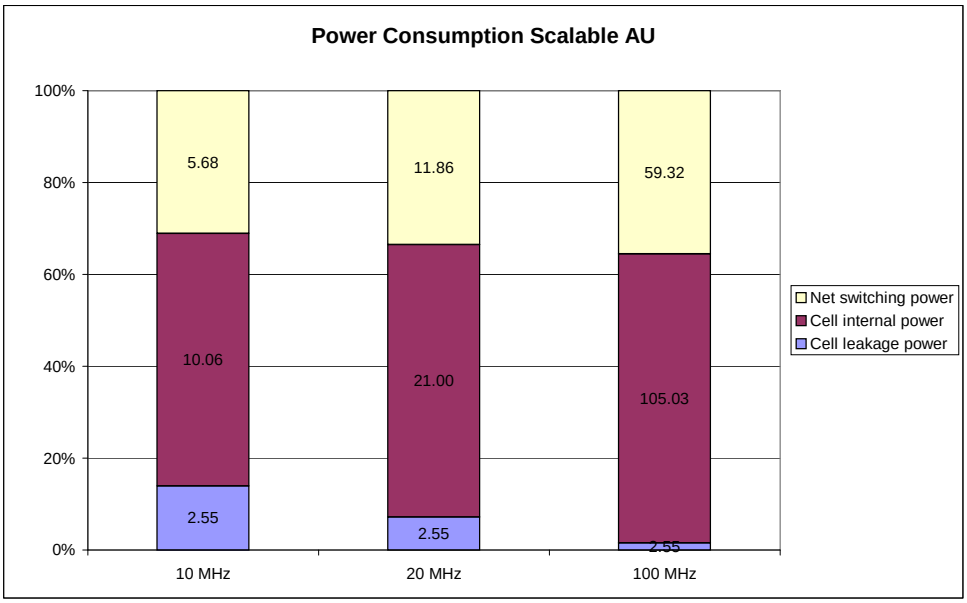


Figure 5.15: Power consumption of the ScAU as a function of frequency

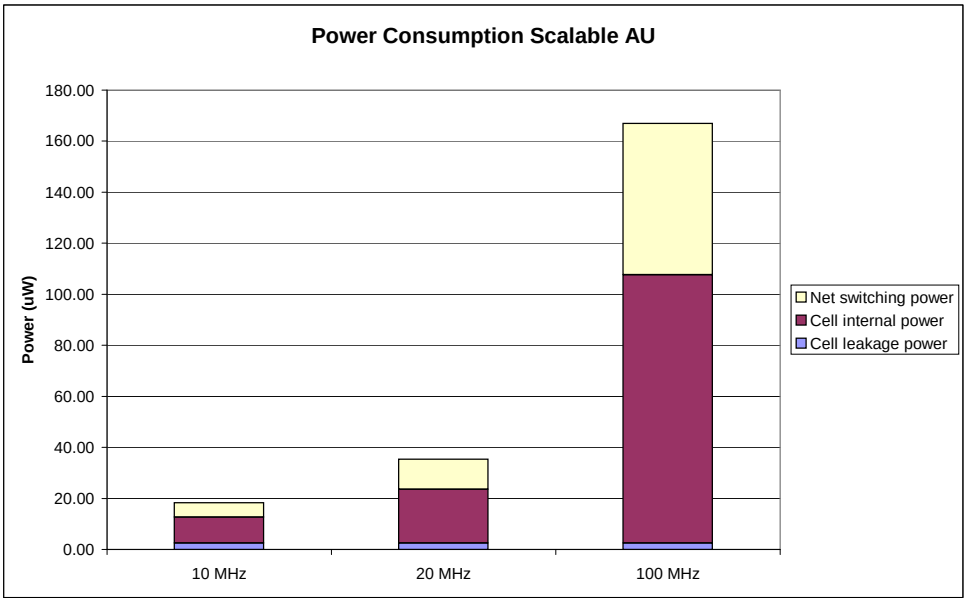


Figure 5.16: Power components for different frequencies

the ScAU becomes more efficient with respect to the DAU-RAS at lower frequencies, the power overheads of the ScAU, the DAU, and the DAU-RAS actually do increase. Thus, in general, the efficiency of the designs is decreasing. This is also due to the higher impact of the static power on the total power consumption. Therefore we cannot lower the clock frequency without limits, in order to favor the power characteristics of the ScAU with respect to the DAU-RAS. For very low frequencies (below 10MHz) we should resort to low leakage technology (refer to section 5.5.7).

### 5.5.4 Alternative employment of the ScAU

Within the scope of this thesis it is also interesting to explore whether the ScAU, unaltered or with some modifications, can be employed to serve other purposes as well. For example, downscaling the arithmetic unit, apart from the situation where an adder segment fails, might also be useful for limiting the overall heat dissipation of the ScAU. We would then de facto trade performance and higher energy consumption for less heat dissipation. Since the ALU is one of the most heavily used part in the micro-architecture, it probably is one of the primary sources of heat dissipation. Since this ScAU is intended for utilization inside a biomedical implant, high temperatures of the chip (compromising reliability) and casing (damage to surrounding live tissue) should be avoided at all times. Therefore, it might be desired to downscale the ScAU, even when no error has occurred. Whether such a mechanism to handle critical temperatures is really required is something that cannot be predicted at this stage. It is a topic for future research.

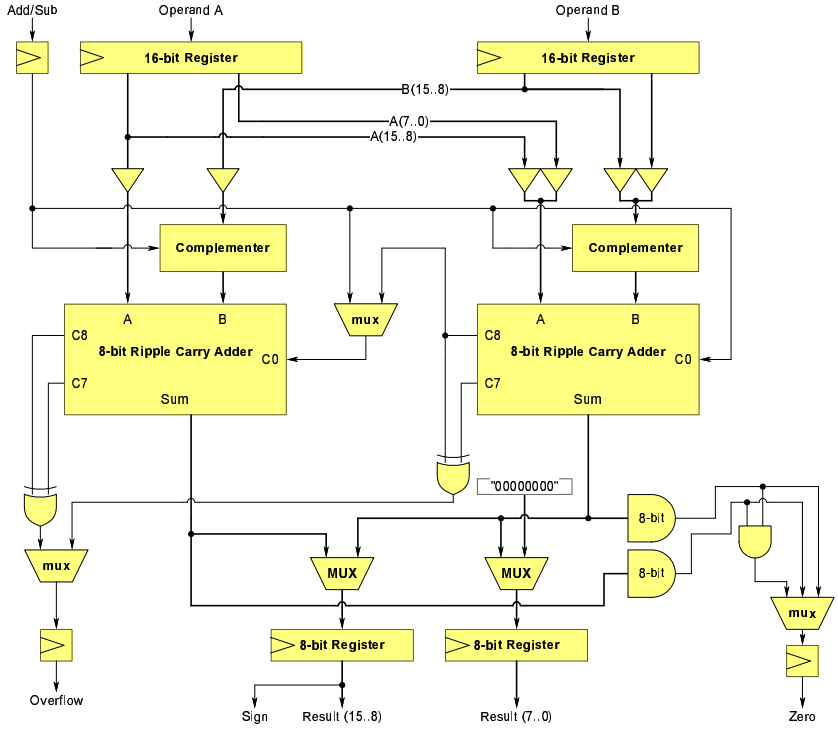


Figure 5.17: Precision scalable arithmetic unit

Another implementation of the ScAU, which will dramatically reduce the energy per instruction in downscaled mode, is the **precision-scalable arithmetic unit (P-ScAU)**. Instead of performing the addition in full precision in two cycles, we might want to discard the lower 8 bits of the operands. If an error occurs, we shut down the erroneous adder segment and continue with the other segment, but we compute only the higher order byte of the result, and pad the lower order byte with zeros. We are compromising the precision of the addition, but we now need only one cycle in downscaled mode instead of two. The P-ScAU is depicted in figure 5.17. Note that in

downscaled mode the lower order byte is filled with zeros (the multiplexer selects the input "00000000", which is generated by a direct connection to the 'scale' signal, which is zero during the first cycle). In table 5.6, the results of the ScAU and the P-ScAU are compared (UMC 90nm, at 100 MHz). In downscaled mode, the ScAU requires 42.5% more energy per cycle, while the P-ScAU requires 36.1% less. The P-ScAU is an option for architectures where double-cycle operations are not permitted (e.g. because of the implications it has on the throughput), but where precision-loss is acceptable.

| Design: | **ScAU** | **P-ScAU** |
|---|---|---|
| Delay [ns] | 2.97 | 2.14 |
| Area [units] | 2332 | 2050 |
| Power (nm) [$\mu$W] | 166.9 | 165.9 |
| Power (dm) [$\mu$W] | 118.9 | 106.40 |
| Energy/instruction (nm) [pJ] | 1.67 | 1.66 |
| Energy/instruction (dm) [pJ] | 2.38 | 1.06 |

Table 5.6: Comparison ScAU and P-ScAU

### 5.5.5 Scalable arithmetic unit with other adder types

Table 5.7 gives an impression about the impact of the overhead (with respect to the non-scalable, single-adder AU) of the 16-bit ScAU, when faster adder types are employed for the adder blocks inside the ScAU. We studied both the utilization of the CSK and RCLA. This study is performed, in the case the ScAU will be utilized in an architecture where the RCA is not fast enough. If a faster and thus larger adder is utilized, the impact of the ScAU's other logic will become less significant. As the table shows, the area overhead of the ScAU decreases, the larger the adder structure is we choose. Thus, in terms of area, the ScAU becomes more efficient for fast adders.

| **Metric** | **RCA-16-8** | **CSK-16-8** | **RCLA-16-8** |
|---|---|---|---|
| Area overhead | 54.1% | 49.7% | 47.0% |
| Delay overhead | 55.5% | 88.4% | 73.3% |
| Power overhead | 18.8% | 17.8% | 17.4% |

Table 5.7: Overheads of 16-bit ScAU with different adder types (normal operation)

For the delay, the opposite is true. The 16-bit ScAU is composed of two 8-bit adders in series, while the non-scalable AU employs one 16-bit adder. This is of no significance for RCA's, but it is for fast adders: two 8-bit CSKs in series can never be as fast as one 16-bit CSK (refer to chapter 3). Typically, fast adders become more efficient when the word width increases. Therefore, the delay overhead increases when fast adders are employed.

Finally, the power overhead decreases when fast adders are employed. This phenomena can be explained in the exact same way as the decrease in area overhead: fast adders utilize more power, which makes the power consumption of the other logic less significant. In conclusion we can say that when higher adder speeds are required,

fast adders make the ScAU more interesting with respect to the DAU-RAS, since they reduce both the area and power overhead. Obviously, the speed limit of the ScAU is significantly lower than that of the DAU-RAS (again, because one 16-bit fast adder is generally faster than two 8-bit fast adders in series).

### 5.5.6 Using different technologies

Apart from different adder types and different frequencies, it is also important to know how the performance and overheads of the design scales with different technologies. Up until now, only the UMC 90nm SP (Standard Purpose) technology is utilized. In this section it is investigated how the ScAU (as well as the AUs with single and duplicated adders) performs when UMC 130nm SP, and TSMC 65nm GP technologies are utilized. Note that all three technologies are targeting *Standard Purpose* designs, otherwise the comparison would not be fair. We are in particular interested in the power and area trends, and the goal of this study is to investigate in which technology the ScAU performs optimally. The results can be observed in tables 5.8 and 5.9, as well as figures 5.18 and 5.19.

| **Absolute area [units]** | | | |
|---|---|---|---|
| Technology | UMC 130nm SP | UMC 90nm SP | TSMC 65nm GP |
| AU | 1721 | 1513 | 584.6 |
| DAU | 2412 | 2185 | 804.2 |
| DAU-RAS | 2984 | 2713 | 1077.8 |
| ScAU | 2565 | 2332 | 949.3 |
| P-ScAU | 2244 | 2050 | 814.0 |
| **Area overheads** | | | |
| Technology | UMC 130nm GP | UMC 90nm GP | TSMC 65nm GP |
| DAU | 40.15% | 44.42% | 37.56% |
| DAU-RAS | 73.39% | 79.31% | 84.37% |
| ScAU | 49.04% | 54.13% | 62.39% |
| P-ScAU | 30.39% | 35.49% | 39.24% |

Table 5.8: Area of 16-bit AUs utilizing three different technologies

It is immediately clear that the choice of technology is of great importance for the performance of the ScAU. As depicted in figure 5.18, the trend lines between UMC 90nm and UMC 130nm are *equal*, only slightly shifted along the $y$-axis. The trend line of TSMC 65nm is different. The area overhead of the DAU is slightly lower, while the area overhead of all other designs are higher than when UMC technology is utilized. The largest difference in the trend we see at the point of the ScAU. One reason for this phenomenon is the larger size of tri-state buffers. Thus, in TSMC 65nm, the ScAU performs significantly worse in terms of area than in UMC 90/130nm.

Figure 5.19 shows remarkable differences in the power trends between all three technologies. When we consider the UMC 90nm and 130nm technologies, the power overheads for the DAU and DAU-RAS do not differ much. The contrary is true for the ScAU and the P-ScAU. In UMC 90nm, the power overhead is significantly lower

| **Absolute power [$\mu$W]** | | | |
|---|---|---|---|
| Technology | UMC 130nm SP | UMC 90nm SP | TSMC 65nm GP |
| AU | 224.60 | 140.50 | 73.48 |
| DAU | 294.10 | 181.60 | 86.29 |
| DAU-RAS | 262.00 | 164.80 | 81.40 |
| ScAU | 273.00 | 166.90 | 84.31 |
| P-ScAU | 270.20 | 165.90 | 82.29 |
| **Power overheads** | | | |
| Technology | UMC 130nm GP | UMC 90nm GP | TSMC 65nm GP |
| DAU | 30.94 % | 29.25 % | 17.43 % |
| DAU-RAS | 16.65 % | 17.30 % | 10.78 % |
| ScAU | 21.55 % | 18.79 % | 14.74 % |
| P-ScAU | 20.30 % | 18.08 % | 11.99 % |

Table 5.9: Power of 16-bit AUs utilizing three different technologies

for these two designs than in UMC 130nm. The TSMC 65nm technology trend makes perfectly clear that this technology is not very efficient for implementing the ScAU. The power overhead between the ScAU and the DAU-RAS is large, the power savings with respect to the DAU only small. The reason why the TSMC 65nm power trend is located so low in the figure with respect to the UMC trends, is because in TSMC the ratio between register and combinational power is different: in TSMC 65nm a larger fraction of the total power is consumed by registers than in UMC 90/130nm, which makes the impact of the combinational logic overhead less significant.

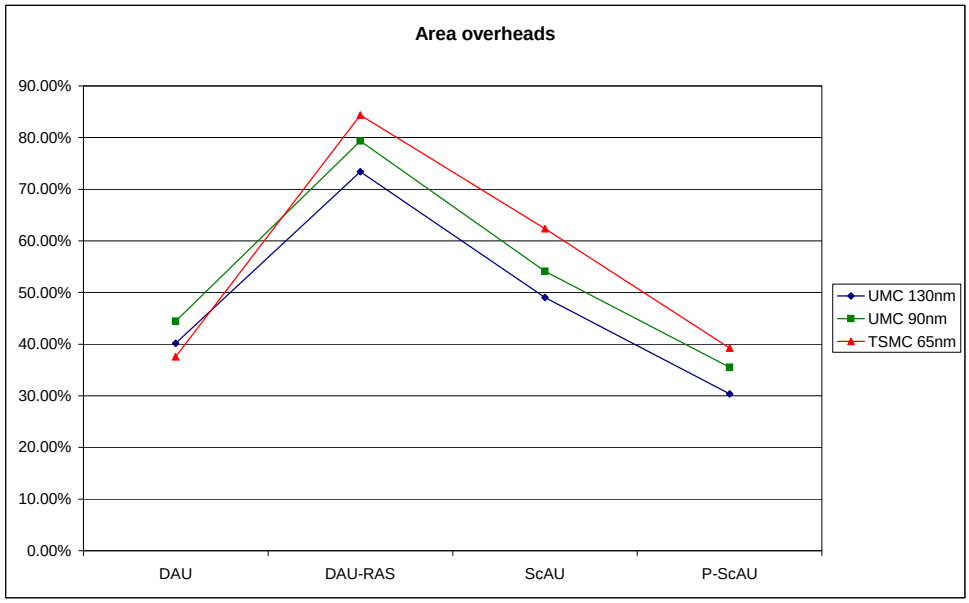


Figure 5.18: Area trends for different technologies

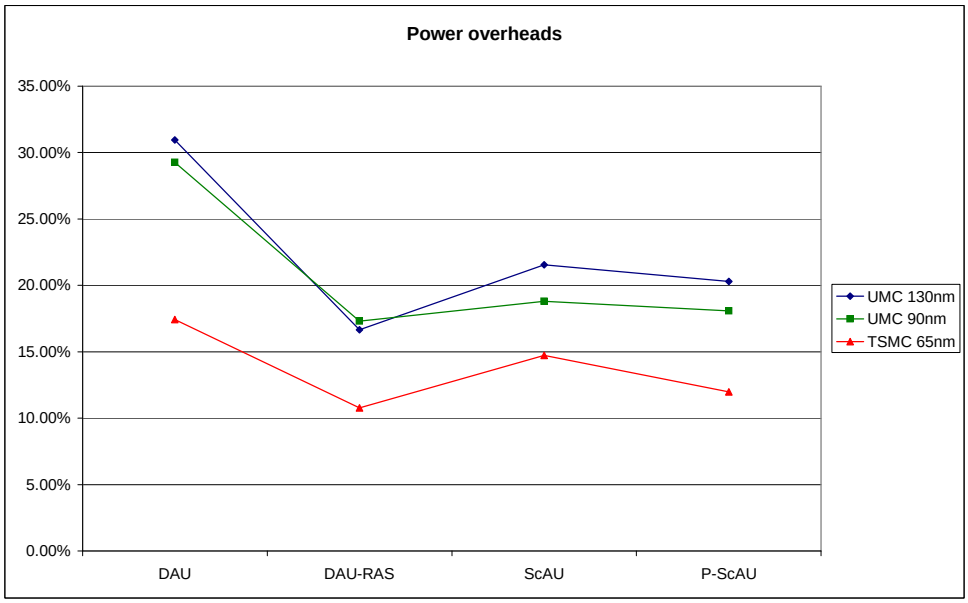


Figure 5.19: Power trends for different technologies (at 100 MHz)

Out of the three technologies we studied, we can say that UMC 90nm technology is the optimal technology for implementation of the ScAU. We proved that technology is a very important variable in the results of the entire design process. Even simple and tiny designs as our arithmetic units show considerable differences in their area and power trends when implemented in different technologies. The differences appear to be the largest when technologies from different *manufacturers* are employed. The source of the differences lies in the total set of available cells and the different ways cells are implemented in different technology libraries.

### 5.5.7 General purpose vs. Low leakage technology

Finally, we took a short look at the impact on the results when we employ a low leakage technology, instead of general purpose. We made a small comparison for the ScAU, based on the TSMC 65nm GP and LL technology. The results can be observed in table 5.10.

The delay of the ScAU significantly increases when we employ LL technology (about 58%), while the area requirements remain virtually equal. At 100 MHz, the LL technology does not do much good to the power consumption: the total power consumption even increases by 13.9%. Also at 20 MHz, the total power consumption is still higher than in GP, even though more than 20 percent of the total power consumption is wasted due to leakage power. At 10 MHz, LL finally proves to be advantageous over GP.

It appears that reducing the static power of the library cells has a downside, as it has a negative effect on both the delay and dynamic power consumption. Based on these observations we conclude that utilizing LL technology is only beneficial if the leakage component in GP is dramatically high (say, more than one third of the total power

| **General Purpose** | | | | |
|---|---|---|---|---|
| Area | [units] | 949.30 | | |
| Delay | [ns] | 2.02 | | |
| Frequency: | | 100 MHz | 20 MHz | 10 MHz |
| $P_{total}$ | [$\mu$W] | 84.31 | 19.25 | 11.27 |
| $P_{switching}$ | [% of $P_{total}$] | *19.16* | *15.66* | *12.31* |
| $P_{internal}$ | [% of $P_{total}$] | *76.19* | *63.95* | *52.85* |
| $P_{leakage}$ | [% of $P_{total}$] | *4.65* | *20.39* | *34.84* |
| **Low Leakage** | | | | |
| Area | [units] | 965.20 | | |
| Delay | [ns] | 3.20 | | |
| Frequency: | | 100 MHz | 20 MHz | 10 MHz |
| $P_{total}$ | [$\mu$W] | 96.00 | 18.40 | 8.90 |
| $P_{switching}$ | [% of $P_{total}$] | *24.00* | *22.90* | *22.75* |
| $P_{internal}$ | [% of $P_{total}$] | *75.96* | *76.90* | *76.86* |
| $P_{leakage}$ | [% of $P_{total}$] | *0.04* | *0.20* | *0.39* |

Table 5.10: Power consumption of ScAU: GP vs. LL technology

consumption). Obviously, the observations are solely based on the TSMC 65nm GP and LL technologies. Because of limited time, we have not investigated the low leakage technologies of UMC. We mark this for future research. Since we also aim at system frequencies above 10 MHz, and we also do not know if the entire SiMS architecture is even suitable for implementation in LL technology, we will not utilize LL technology any further in our experiments in this thesis.

### 5.5.8 Verification of area trends by place and route

The ScAU appears (at least at this point in the design process and under the current configuration) not as interesting for saving power as we hoped for (see figure 5.13), but on the other hand, the ScAU is capable of remarkable area savings (see figure 5.11). To be certain about the correctness of this observation we checked if the trend holds true after layout (please refer to chapter 3 for the details about the place and route tool). The results are depicted in figure 5.20. The post-synthesis (a.k.a. pre-layout) results appear to be astonishingly accurate: the maximum deviation between the pre- and post-layout results is only 0.8%. As explained in chapter 3, we had only access to layout scripts with links to the TSMC 65nm GP technology. The results in figure 5.20 are therefore based on this technology. We believe however that it is justified to assume that the trends will hold true for the UMC technologies as well.

## 5.6 Application of low-power design techniques

Since we proved that the ScAU is more than fast enough (when implemented in 90nm CMOS) to operate on frequencies between 10-100 MHz, we are able to lower $V_{DD}$ of the ScAU in order to save power (voltage scaling). Ideally, it would be best if $V_{DD}$ of

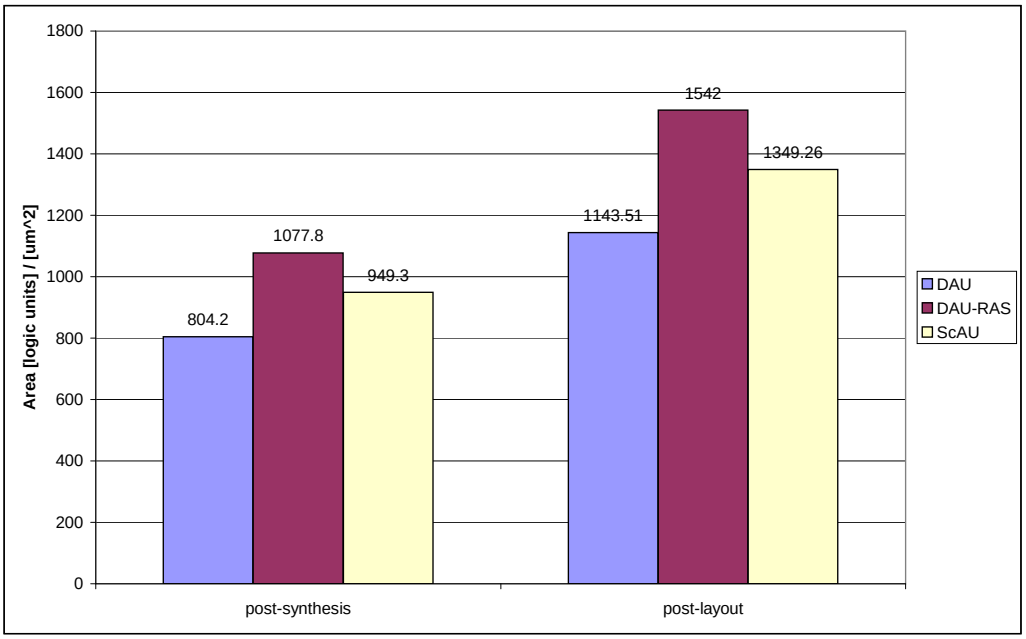


Figure 5.20: Comparison between pre- and post-layout area trends

the entire architecture could be reduced. If not, level converters and a separate power supply is required. It is difficult to predict if this would be efficient.

One of the downsides of the ScAU is the significant increase in energy per cycle in downscaled mode (+42.5%). One way to overcome this downside is to decrease $V_{DD}$ only (or even further) in downscaled mode. This is possible, since only one adder segment is utilized in downscaled mode, and thus the critical path has become shorter. There is, however, a limit to the extent we can reduce $V_{DD}$ (as explained in chapter 2, section 2.4.1). What is more, if we employ relatively low system frequencies, we will be able to reduce $V_{DD}$ to the maximum extent even when the ScAU is in normal mode. Thus, this approach is only applicable when the ScAU is employed in systems with higher frequencies.

For example, when we utilize a frequency of 333 MHz (which is the maximum frequency the ScAU can operate on, considering the delay of almost 3 ns). At this speed, lowering $V_{DD}$ is not an option, since it not only decreases power but also increases delay. When the ScAU is in downscaled mode (after an error has occurred), the critical path has become significantly shorter, since one adder segment and a multiplexer are by-passed. This reduces the critical path by approximately 30%. Ideally, we want the energy consumed per operation to be equal in normal mode and in downscaled mode. Now the critical path is shorter, we can accomplish this by reducing $V_{DD}$. We utilize the power results of the measurement at 100 MHz as an example:

$$P_{nm} = 166.9\mu\text{W},\ E_{nm} = 1.67p\text{J}$$
$$P_{dm} = 118.9\mu\text{W},\ E_{dm} = 2.38p\text{J}$$

Again, what we want is to make sure that the energy consumption in downscaled mode does not exceed the energy consumption in normal mode. We can do that if we reduce

$V_{DD}$ in downscaled mode, which leads to extra delay, but this does not have to be a problem since the critical path is significantly shorter in downscaled mode than in normal mode. In order to compute the necessary voltage reduction:

$$\begin{gathered}\text{if } E_{nm} = E_{dm} \text{ then} \\ 166.9 = 118.9x + 118.9x \\ x \cong 0.7\end{gathered}$$

we must make sure that the power consumption of the ScAU in normal mode during one cycle is equal to the power consumption of the ScAU in downscaled mode during two cycles. The variable $x$ represents the necessary reduction in power (in downscaled mode), in order to have $E_{nm} = E_{dm}$. Since $x$ is 0.7, this means that with 70% of the power we meet our goal. Thus, in downscaled mode we require a reduction in power consumption of 30% with respect to the power consumed in normal mode. Normally, $V_{DD} = 1$ Volt in 90nm CMOS, and since power depends quadratically on $V_{DD}$, $V_{DD}$ needs to be reduced to:

$$\sqrt{x} = \sqrt{0.7} = 0.837 \text{ Volt}$$

Which comes down to a voltage reduction of 16.3%. Since the critical path was reduced by 30%, this $V_{DD}$ reduction is certainly allowed. Again, this method will require level converters and an additional power supply, so it is unknown if this approach is cost-effective. This topic is therefore an interesting direction for future research.

Another option to avoid increased latency and design problems of the pipeline to support the double-cycle operations, is to double the clock frequency of the ScAU in downscaled mode. Obviously, the power consumption of the ScAU will double as well, but since power is quadratically dependent on $V_{DD}$, we can cancel this effect by lowering $V_{DD}$ by 25%. It is clear that this approach is only possible under certain conditions. What is more, since we need an extra power supply, a variable clock signal, and level converters, it is questionable if this approach is cost-effective.

For implementation in the SiMS architecture all of the above approaches are not applicable. In SiMS relatively low frequencies will be utilized, so $V_{DD}$ can already be reduced to the maximum extent (i.e. the maximum reduction that does not compromise reliability and does not increase leakage power out of proportion, as explained in chapter 2) in normal mode.

## 5.7 Conclusions

The ScAU is a newly proposed design for an arithmetic unit which is able to downscale its calculations when an error has occurred in one of the adder segments. This is a form of graceful degradation, where the **precision** of the calculation is preserved, but the **throughput** is compromised since the calculations now take two cycles per operation. Supporting **multi-cycle** operations will complicate the design of the pipeline. At this point it is unknown how the pipeline should be modified and what the costs will be. This topic is, however, beyond the scope of this thesis work, but it is an interesting topic for future research.

We have presented the initial design of the ScAU, several optimizations as well as the final design. After each synthesis step we closely analyzed the results and considered possible optimizations, modified the design, re-synthesized, and repeated this loop until we were unable to optimize the design any further. We proved that input-gating of combinational logic (adders) is much more efficient when we employ **tri-state buffers** instead of guard latches. It is shown that tri-state buffers can be utilized as a multiplexer network and input-gates at the same time to reach optimal efficiency. The adder-output multiplexing is implemented by regular gate-based multiplexers, since we proved that two-input tri-state-based multiplexers require more power and area than gate-based ones.

The ScAU's power consumption is significantly lower than that of the DAU (-8.1%), equal to that of the DAU-RAS at 20 MHz, and higher than the DAU-RAS at all frequencies above 20 MHz (+1.3% at 100 MHz). We can conclude that the ScAU performs better at relatively *low frequencies*. The ScAU appears not as interesting for saving power as we hoped for, but it certainly is capable of *significant area savings*. The area requirements for the ScAU are 14.0% lower than for the DAU-RAS. The reason why the ScAU does not perform very good in terms of power (in the sense that the power consumption is slightly higher than the DAU-RAS, even though the adder is not duplicated as in the DAU-RAS) is because the ScAU requires a significant amount of hardware to enable the scalability feature. The delay of the ScAU is high (39.4% higher than the DAU-RAS), since much of the additional logic resides in the critical path. However, in the next chapter this problem will be solved. A drawback of the ScAU is the **energy per instruction** which increases significantly when the ScAU is in *downscaled mode*. A number of approaches have been discussed to overcome this disadvantage.

We have showed that the 8-bit ScAU is not efficient, since the ratio between adder logic and control logic is too small. The 16-bit ScAU performs much better, since most of the control logic is constant and does not depend on the word size. As mentioned previously, the ScAU performs also better at lower frequencies. Based on the results in this study we decided to continue with the next study in chapter 6 with a frequency of 20 MHz, since a clock frequency of 20 MHz would be a good and realistic choice for the SiMS architecture as well (refer to chapter 1, section 1.2).

Further, we discussed some alternative applications of the ScAU, such as the **precision scalable AU** (P-ScAU). The P-ScAU could be employed when dual-cycle operations are not allowed. The precision of the calculation is, however, sacrificed. Another major advantage of the P-ScAU is the very low energy consumption per instruction (in particular in downscaled mode). If the ScAU is employed in architectures where high throughputs are required and the RCA is not fast enough, the ScAU has to be equipped with a fast adder. We proved that the ScAU becomes *more efficient* in terms of power and area overhead when fast adders are employed. Finally, we implemented the ScAU (as well as the reference designs) in different technologies as one of our many attempts to find the optimal design point. We showed that the ScAU performs best in UMC 90nm and worst in TSMC65nm technology and proved that technology is a very important variable in the results of the entire design process. Even simple and tiny designs as our arithmetic units show considerable differences in their area and power trends when implemented in different technologies. The source of the differences lies in

the total set of available cells and the different ways cells are implemented in different technology libraries. Utilizing low-leakage technology is only useful when low clock frequencies are utilized (up to 10 MHz). For higher frequencies, this technology is useless since it *increases dynamic power consumption*. In order to prove that the promising post-synthesis estimations regarding low-area are correct we utilized a place and route tool to confirm. The post-synthesis results appeared to be remarkably accurate.

A number of low-power design techniques are presented if the increased energy consumption per operation and implications of double-cycle operations in downscaled mode of the ScAU turn out to be problematic.

# 6 Fault-Tolerant Design

## 6.1 Introduction

In this thesis, as stated before, we are targeting highly mission-critical applications such as biomedical implants. Accordingly, in this chapter we investigate various fault-tolerance techniques previously reported in the literature. Then, we move to retrofit the adders discussed in the previous chapter with error-detecting capabilities. This chapter is essentially divided into three major parts. The first part (sections 6.2 to 6.11), starts with a theoretical and general introduction in fault-tolerant (FT) design. All important terminology that is employed in fault-tolerant design is explained. Since the design of arithmetic units is unique in the sense that some of its error types are different from that in logic structures, the types of errors specific to adders is discussed in detail. A selection of common and less frequently utilized error-detection techniques is discussed, as well as some methods for error correction. In part two (sections 6.12 to 6.16), a few error-detection techniques are singled out for a more detailed investigation. The main questions are: which error detection/correction (in this chapter often abbreviated by ED/EC) technique is suitable for utilization in the ScAU, what is the area/power cost, and what is the error coverage of each scheme. In addition, we also have to determine what the minimal fault coverage of the ScAU should be. At the end of this part we make the decision which ED/EC scheme we will utilize for the ScAU, and we assemble a number of reference schemes, in order to make a good comparison later. In part three (section 6.17), when the ED/EC scheme for the ScAU has been chosen, will be explained how the scheme is exactly implemented. Most importantly, the synthesis results (area, power, and delay) of the design are presented as well as the synthesis results of a number of reference designs, which enables a good comparison. After that a number of important conclusions will be drawn.

## 6.2 Logic error types and their sources

First, a number of terms will be introduced, which are frequently utilized in the field of fault-tolerant design. For example, it is important to realize that the terms **fault**, **failure**, and **error** do not have the same meaning. There are many reasons why a certain component in a circuit might show abnormal behavior. However, if this abnormal behavior occurs, we speak about a fault. An error is a situation where a fault changes the system's *logical state* that is different from the expected value. One should however note that *not all faults lead to errors* and therefore they are not identical. Technically, errors are a *subset* of the faults set. In the situation where an error occurs and the system is *not able to recover* from this erroneous state, we speak about a failure. Thus, if we can detect and correct the error, we can prevent a system failure. The primary goal of

a fault-tolerant design, is to improve the *dependability* by enabling a system to perform its intended task *in the presence* of a given number of faults [57]. So, fault-tolerant design involves detection of faults as well as correcting them once they occur. Since we are primarily interested in faults that cause errors, we speak generally about ED/EC. Again, here one should realize that two other terms that are frequently utilized, namely **dependability** and **reliability**, are not the same. Dependability can be quantified and it depends on the type of faults the system can detect and recover from. Reliability is the probability that the system can perform its designed function at time $t = x$. However, one can say that reliability *includes* dependability. In practice fault-tolerant computing is also referred to as **reliable computing**. It is also a misunderstanding that a fault-tolerant design is automatically highly dependable. It may be the case that the FT-design is able to detect and correct faults of type A, but if it does not detect faults of type B which occur more frequently, the system is still not highly dependable. Therefore it is very important to understand as much as possible about the types of faults that can occur and the chance that they occur.

A **permanent** hardware fault is the most simple fault to detect and handle. It occurs at a certain point in time, and the fault remains present forever (e.g. breaking of a wire or short circuiting of wires due to overheating). There exist many fault models to discriminate their locality of occurrence, like the *stuck-at fault model* (fixed value to a signal), the *bridging fault model* (a short between a group of signals), and the *stuck-open fault model* (i.e. one transistor is permanently open) [58]. The most frequently utilized model is the stuck-at model, primarily because it has a higher abstraction level. The stuck-at model is applicable at gate-level, while the other two models are only applicable at transistor level (which requires technology level knowledge, which not every computer engineer is very familiar with). There are however more types of faults that are much more complicated to detect and/or handle: **transient** and **intermittent** faults. A transient fault occurs only once and then disappears. In most cases these faults are caused by cross-talk, noise or some kind of external interference [44]. Errors caused by transient faults are known as **soft errors**, since the hardware itself is not actually damaged. Soft errors occur more frequently when technology scales, when voltage level reduction is applied, and at very high speeds [59]. Such soft errors may affect only one bit or a number of consecutive bits. In the first situation we call it a **random error**, in the latter a **burst error**. Intermittent faults manifest themselves repeatedly. One cause for intermittent faults are, e.g., bad connections. If the temperature of the chip varies, so does the resistance of the connection. At a certain temperature the connection may function properly, while at other temperatures the connection is broken. The varying temperatures of the chip are in most cases also the cause of the bad connection in the first place. Intermittent errors are sometimes treated as permanent faults, and sometimes as transient ones (often depending on the rate of occurrence) [60]. Since soft errors do not damage hardware, but do corrupt results, it makes it difficult to handle them. If we immediately shut down a part of the hardware when an error is detected and activate a redundant (backup) circuit to be able to proceed with the computational work, we waste valuable hardware (and power) if the error was only a transient error. In such situations it would be better to *mask* the error. In section 6.11 we will discuss this topic in more detail. However, in this thesis we do not distinguish between permanent and soft errors.

This decision is made based on two reasons. First, it certainly simplifies designing the fault-tolerant ScAU, so there is a possibility that it will also simplify the complexity of the design (and thus saves power and area). Second, this simplification allows us to explore the entire range of ED/EC schemes, and not just a subset.

All error-detection schemes discussed in this thesis are commonly referred to as **CED** schemes. CED stands for **Concurrent Error Detection**, and means that the error detection is continuously operational: every result that is produced is checked by the error-detection hardware. Another variant of error detection is non-concurrent error detection, i.e. checking for errors *periodically* [61].

## 6.3 Boolean difference and Hamming distance

Any combinational logic circuit can be described by one or more logic functions. Assume $X = x_1, x_2, ..., x_n$ is the input of function $F$ and $F(X)$ is its output. If we want to know if an error at a particular input has effect on the output, we need to compute the **Boolean difference** [62]. The Boolean difference can be computed by:

$$\frac{dF(X)}{dx_i} = F(x_1, ..., x_i, ..., x_n) \oplus F(x_1, ..., \bar{x_i}, ..., x_n)$$

Note that this is not a differential equation, but the Boolean difference between $F(X)$ with correct inputs, and $F(X)$ with an incorrect value at input $x_i$ (called a **single error**). This difference can be obtained by a bitwise exclusive-or of the two output values. There are three possible outcomes: the result is either zero, one, or a new function $G(X)$. The incorrect value at input $x_i$ leads to an erroneous result in $F(X)$ when the Boolean difference is one. Then the error is always detectable. When zero, $F(X)$ has not changed in the presence of the error at the input and therefore the error is undetectable. If the Boolean difference is a new function $G(X)$, then an error in $x_i$ will cause an error in $F(X)$ if and only if $G(X) = 1$, i.e. the error is detectable but only under specific circumstances. We can determine the Boolean difference as well, if two inputs are erroneous (two errors occur at the same time, a.k.a. **double errors**), called the double Boolean difference:

$$\frac{d^2F(X)}{dx_i dx_j} = F(x_1, ..., x_i, ..., x_j, ..., x_n) \oplus F(x_1, ..., \bar{x_i}, ..., \bar{x_j}, ..., x_n) = G(X)$$

Here $G(X)$ is either one (error always detectable), zero (error never detectable), or any logic function (error detectable under specific circumstances). Another important formula is:

$$\frac{d^2F(X)}{d(x_i + x_j)} = \frac{dF(X)}{dx_i} + \frac{dF(X)}{dx_j} = G(X)$$

Here we do not assume that both errors at the inputs occur at the same time (thus, this is not a double error), but we question what happens if $x_i$ *or* $x_j$ is erroneous. Obviously, we can expand the formula for all inputs of the function.

A convenient way to compute these Boolean differences for small circuits is to employ *Karnaugh maps*. One can simply make two Karnaugh maps of the function $F$, one with

correct inputs and one with one or multiple erroneous inputs. Exclusive-oring of all corresponding positions in the Karnaugh maps results in the Karnaugh map of the function $G(X)$ and this function can thus be easily read out. Obviously, for large circuit we will have to resort to special computer programs. In general, Boolean difference is an important tool to investigate and prove if certain errors at the inputs of a system will also result in erroneous outputs, and —when the answer is yes— under which specific circumstances.

Another important term in fault-tolerant design is **code distance**. For example, when we have three bit lines, we have eight possible binary settings. If we utilize these bit lines to send messages, and we have eight distinct messages, then every time an error occurs in either one of the three lines the original message is lost and a corrupted message is given. The problem here is that the corrupted message is also a legal massage so the error can *never* be detected. What we can do is create a situation where, if an error occurs, the corrupted message is no longer part of the set of legal messages (or values). Therefore the code distance between any two legal messages should be at least two: meaning that any two messages from the legal set should differ by at least two corresponding bit positions. Now, when the code distance is two, all **single errors** result in illegal messages and thus all single errors can be detected. Obviously, this means that for the same number of messages we need more bit lines. Code distance is very important in designing communication buses and controllers.

Since Richard Hamming (1915-1998) was the first one who formally defined code distance, code distance is generally referred to as **Hamming distance**. The Hamming distance between two pairs of bit vectors (messages) X and Y can be formally described by:

$$d(X,Y) = \sum_{i=1}^{n}(x_i \oplus x_j)$$

It represents the total number of corresponding bit positions where X and Y differ. When a code has a distance 'd', it can detect errors with a multiplicity of d-1 and lower. For example, when d=3, we can detect all double and all single errors. Much more information about Boolean difference and Hamming distance can be found in the book of Sellers et al. [62].

## 6.4 Errors in adders

The basic element in adders is the full-adder. A full-adder is build by two (partially shared) circuits, to implement the functions 'sum' and 'carry' [44, 62]:

$$s_n = a_n \oplus b_n \oplus c_{n-1} \text{ (I)}$$

$$c_n = a_n \cdot b_n + (a_n + b_n)c_{n-1} \text{ (II)}$$

Where (II) can be subdivided into a generate (III) and a transmit/propagate (IV) function:

$$g_n = a_n \cdot b_n \text{ (III)}$$

$$t_n = a_n + b_n \text{ (IV)}$$

Such that:

$$c_n = g_n + t_n \cdot c_{n-1} \text{ (V)}$$

A single fault in an adder circuit leads to an error which can fall into two different categories: a single sum digit error or a burst of sum and carry digit errors [62]. A single error occurs when the fault is caused by a fault in a sum circuit, a burst error by a fault in a carry circuit. This makes the error characteristics of adders special, since burst errors *normally* only occur as a result of soft errors. We will explain this special error type in more detail.

A single error in $a_n$, $b_n$, or $c_{n-1}$ will always cause an error in $s_n$ (I). An error in a carry digit $c_{n-1}$ will always propagate and cause an error in $s_n$ as well. So, a fault in a carry circuit will cause *two* errors at the *minimum*: one carry digit error and one sum digit error. Now, if the situation is such that due to the error the incoming carry cannot be absorbed in stage n, carry digit $c_n$ will be erroneous and so will be $s_{n+1}$. This way, the burst error can become arbitrarily long. Because of this phenomena, carry errors are of primary concern in adder error detection [62].

Instead of reasoning, we can utilize Boolean difference equations (commonly referred to as **error functions**) here to provide proof. For the sum circuitry:

$$\frac{ds_n}{da_n} = (a_n \oplus b_n \oplus c_{n-1}) \oplus (\overline{a_n} \oplus b_n \oplus c_{n-1})$$

$$\frac{ds_n}{da_n} = (a_n \oplus \overline{a_n}) \oplus (b_n \oplus b_n) \oplus (c_{n-1}) \oplus c_{n-1})$$

$$\frac{ds_n}{da_n} = 1 \oplus 0 \oplus 0 = 1$$

Which means that errors in $a$ always cause an error in $s$. The error functions for $b$ and $c_{n-1}$ are similar.

And for the carry circuitry:

$$\frac{dc_n}{dg_n} = \frac{d(g_n + t_n \cdot c_{n-1})}{dg_n} = \overline{t_n \cdot c_{n-1}}$$

$$\frac{dc_n}{dt_n} = \overline{g_n} \cdot c_{n-1}$$

$$\frac{dc_n}{dc_{n-1}} = \overline{g_n} \cdot t_n$$

Thus, in words, $c_n$ is in error if $c_{n-1}$ is in error, if and only if $\overline{g_n} \cdot t_n = 1$. So, it depends on the data if a carry error propagates or not. For the derivation of the error functions above, we refer to chapter 6 of the book of Sellers [62].

If an error in $c_n$ causes an error in $c_{n+q}$, then $c_n$ also causes an error in $c_{n+1}, c_{n+2}, ..., c_{n+q-1}$. Thus, one single fault in a carry circuitry can cause any number of successive carry errors. And since:

$$\frac{ds_n}{dc_{n-1}} = 1$$

all carry errors lead to (successive) erroneous sum results as well. Thus, the results $s_{n+1}, ..., s_{n+q+1}$ are in error too.

## 6.5 Important terminology employed in fault-tolerant design

A circuit is said to be **fault-secure** for a set of faults F, if for every fault in F, the circuit never produces an incorrect codeword at the outputs for correct codewords at the inputs. Alternatively, it produces either correct code-space outputs or outputs in the error space [62, 63]. Thus, for example, if an error-detection scheme covers 100% of all single errors in a circuit, the circuit is fault-secure for single-errors. **Self-testing** means that every fault from a prescribed set can be detected during normal operation. In other words, there exists at least one input pattern in normal operation that produces an erroneous result for each fault in the set. In fact, this property avoids the existence of redundant faults. Such faults remain undetectable and could be combined with new faults occurring later in the circuits, resulting in multiple faults that could destroy the fault secure property [64, 65]. The **totally self-checking (TSC)** property can be achieved if the circuit is both fault-secure and self-testing for the prescribed set of faults. Basically, the TCS property guarantees that a fault is detected the first time it manifests itself as an error at the outputs of the circuit. We speak about a **strongly fault-secure (SFS)** circuit when the circuit is (I) totally self-checking, or (II) fault-secure and after the occurrence of faults: guaranteed fault-secure or self-testing with respect to the accumulated faults [66]. Although not necessarily equal, in many cases the TCS and SFS properties are considered to be equivalent. One of the two properties is sufficient for designing a trustworthy self-checking circuit. A circuit is called **code-disjoint** if it maps codewords at the inputs to codewords at the outputs and non-codewords at the inputs to non-codewords at the outputs. This property is important for the design of checkers.

## 6.6 Achieving the fault-secure and self-testing property in adders

According to Nicolaidis [65], the following goals have to be met for a self-checking adder in order to be efficient: it should be totally self-checking or strongly fault secure for single errors, it should have a low hardware overhead and be checked by a compact checker, and it should be able to be combined with parity-checked data paths and memories without using code translators. Self-checking (error-detecting) memories often utilize **parity prediction** and fault-tolerant (error-detecting and -correcting) memory systems utilize, e.g., **Hamming SECDED** (single error correcting, double error detecting) which is compatible with parity prediction. Parity prediction is a type of error detection we will explain later. We will also study if it is true that this particular type of ED is the most efficient. However, Nicolaidis provides some very good guidelines here. The

most important guideline is that the choice of the ED should not only be based on the efficiency and applicability in the AU, but also in the entire architecture.

Since adders are well optimized circuits (ripple carry adders, but also carry-lookahead adder, carry-skip adders, etc.), no redundancies exist in the adder designs. Thus, by definition, the self-testing property holds true for all stuck-at errors in adders [65]. As discussed in section 6.4, the fault-secure property (for all single errors) can be achieved when we are able to detect single faults which manifest as single errors in the result and single faults which manifest as burst errors in the result. Achieving the TSC property in adders is not difficult. However, if we consider the error checker part of the circuit (which means that the checker itself has to achieve the TSC goal as well), achieving the TSC property is much harder as we will see in section 6.10.

## 6.7 Error detection by hardware duplication

The error-detection technique that appeals the most to one's imagination is without doubt **hardware duplication**. It is a technique that has been utilized for many years and is known for its high fault coverage. When we have a certain circuit, we simply duplicate the circuit and feed both outputs to a comparator. These systems are also referred to as **duplex** or **Duplication With Comparison (DWC) systems** [44, 62, 67, 68]. Almost all errors at the outputs can be detected this way, except **Common-Mode Failures (CMFs)**, where a single cause leads to multiple faults, namely faults in *both* circuits. The chance that both circuits produce identical errors at the same time is small, but certainly not impossible. One should realize that both circuits have the same layout, are connected to the same power supply and are subject to the same external disturbances. CMFs can be caused by, e.g., electromagnetic interference, power-supply disturbances, and of course design errors. A solution exists to make the hardware duplication technique less susceptible for CMFs: instead of adding an exact copy of the original circuit (identical duplication), it is better to redesign the circuit, with the same functionality, but a different implementation. This technique is called **diverse duplication** and this increases the fault coverage even further [67]. The major drawback of the duplex technique is the high costs. Since the circuit is fully duplicated and also a comparator is required, area and power costs increase by more than 100%. However, when reliability is crucial and power and area budgets are not that tight, this method is a very good choice.

## 6.8 Error-detection codes

All error-detection codes are able to at least detect single errors, some of them can also detect double errors, and other even triple errors, etc. In contrast with the hardware duplication technique (which covers basically errors of any multiplicity), it highly depends on the type of code what the fault coverage is. The general rule is, the higher the fault coverage, the higher the hardware overhead, and the more check bits are required. It is important to know that double errors occur *much less frequently* than double errors. The chance of an occurrence of a triple error is even lower. This makes

sense, since the chance that 2 or 3 errors occur in different parts of the circuit at exactly the same time is statistically much smaller than the chance that a single error occurs. So, for the majority of logic designs protection against (all) single errors is sufficient. The extra protection against double or even triple errors does *not* add significantly to the probability of detecting errors [62, 69]. Only in systems where reliability is of crucial importance these codes may be desirable. However, often it will remain a trade-off between reliability and cost.

Error-detection codes do not require full hardware duplication (sometimes partial duplication), but in any case the ED-hardware introduces inherent diversity in the system. This makes systems checked by error-detecting codes far less vulnerable to CMFs than systems checked based on full hardware replication [67].

When an error-detecting code is utilized, all messaged need to be **encoded**. This means that the original message is altered by adding additional (redundant) information. If something goes wrong during processing the message, the **decoding** process will notify this and signal an error. Since in this context we design an arithmetic unit, the messages represent the operands that need to be added/subtracted.

The most common error-detection codes utilized in computer arithmetic are **parity prediction** and **residue checking**. Less common ED codes are **Berger and Bose-Lin coding**. These are the four ED codes that will be discussed in this chapter.

### 6.8.1 Parity Prediction

When utilizing **parity prediction**, the $n$-bit operand is extended by a single check bit (the so-called **parity bit**). The parity bit of each operand needs to be *generated* (encoding process) before the actual arithmetic operation is executed. The parity is defined as (for binary systems):

$$P(N) = \sum_{i=0}^{n} x_i \bmod 2 = x_n \oplus x_{n-1} \oplus x_{n-2} \oplus ... \oplus x_0$$

where $N = x_n 2^n + x_{n-1} 2^{n-1} + ... + x_0 2^0$ is the $n$-bit operand represented in the conventional polynomial form. When there is an even number of one's in the operand, the sum of the operands is divisible by two, so the remainder will be zero. This is called even parity. When the number of one's is odd, the remainder is one, indicating odd parity. Thus, the parity bit contains information about the evenness/oddness of the number of one's in the operand. The parity bits of the operands can be generated inside the AU, or can be generated early in the pipeline when parity prediction is utilized for error detection for more components in the architecture than just the AU. The choice, whether to generate the parity bits inside the AU or not, has major consequences for the power consumption and area requirements of the AU.

When an addition is performed, two operands are added, generating a result. Adding two operands with both even parity, or both odd parity, leads to a result with even parity. If one of the operands has even parity and the other has odd parity, the result has odd parity. The scheme is called parity prediction, since the scheme contains a circuit which *predicts* what the parity of the newly generated result will be. At the same time, the parity of the result is generated in the same way as described for the operands. Finally, the predicted and the actual parity of the result are *compared*. If they do not match,

an error has occurred in the arithmetic operation. Parity prediction covers all errors, which affect only a single bit. If multiple bits are affected, the detection depends on the oddness/evenness of the number of affected bits. E.g. if two bits are erroneous, the parity-prediction scheme will not detect any error, since the parity will not change (the message is corrupted, but the message is still legal). Parity prediction has a wide range of applications: it allows error checking in telecommunications, in arithmetic and logical operations, and in memories.

With respect to the conclusions at the end of section 6.4, now it can be shown clearly, that standard RCAs are not fault-secure for *carry errors* when checked by parity-prediction schemes. If $c_n, ..., c_{n+q}$ are in error because of a carry circuit fault, and as a result of that $s_{n+1}, ..., s_{n+q+1}$ are in error as well, the total number errors in the carries and sum are *always even*, regardless of 'q'. This is a major problem for parity-prediction schemes, since they cannot detect an even number of errors, because of the simple fact that the parity of the result is unaffected. Therefore the RCA needs some modifications in order to be able to detect carry errors at all times. In section 6.16 three different parity-prediction schemes are discussed, which solve the problem of undetectable carry-errors.

### 6.8.2 Residue Checking

A more advanced type of an ED-code is **residue checking**. For each integer '$A$' there exists a unique **residue** (or remainder) '$R$' when the integer '$A$' is divided by another integer '$m$', where $R < m$. In the case that two integers '$A$' and '$B$', both divided by '$m$', result in the same remainder '$R$', the numbers '$A$' and '$B$' are said to be congruent modulo '$m$', which is denoted as $A \equiv B \bmod m$. The residue is formally defined as:

$$R(N) = N \bmod \mathrm{m} = \{\textstyle\sum_{i=0}^{n} x_i \cdot r^i\} \bmod \mathrm{m}$$

where $N = x_n 2^n + x_{n-1} 2^{n-1} + ... + x_0 2^0$ is the $n$-bit operand, '$r$' is the radix which is 2, and '$m$' represents the modulus. The modulus has to be chosen in such a way that $0 \leq R(N) < m$. Both operands need to be provided with check bits: the residue of the numbers need to be *generated* prior to the arithmetic operation. The functionality of the scheme is as follows: the residue of the result is predicted by adding the residues of both operands in a modulo-$m$ adder. The predicted residue is then compared with the actual residue, generated by a modulo-$m$ generator. If there is a mismatch, an error occurred during the arithmetic operation. The scheme is depicted in figure 6.1. As mentioned earlier, for each integer '$A$' there exists a unique remainder, when divided by the modulus. Only when errors occur in such a way that the number '$A$' is corrupted in a new number '$B$', and $A \equiv B \bmod m$, the errors remains undetected. In other words: a residue code will detect all error patterns, except those when $\sum_{i=0}^{n} e_i \cdot 2^i = 0 \bmod m$ [62]. As long as *a modulus is chosen, which is equal to the radix of the number system plus one, all single errors can be detected*. Thus, in our case with a modulus of three we can cover all single errors. Modulus-3 checking has an error coverage of 50% of all double errors (like parity prediction). Note however, that the set of double errors that is covered by this scheme is *not the same set* of double errors covered by the parity-prediction scheme. We took a closer look to see what really happens. Therefore

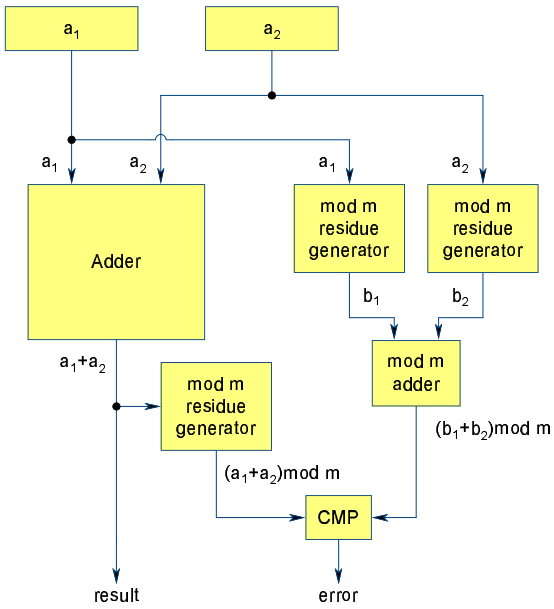


Figure 6.1: Residue Checking Scheme

we picked a number of 4-bit numbers and introduced all possible double errors in it. For parity prediction, the error coverage of double errors is always 50%, regardless of the actual number. Half of the total amount of double errors is detected, the other half is not. Modulo-3 checking performs better or worse depending on the actual number we chose. In some cases the error coverage is 33.3%, in others 66.7%. However, in a set of numbers $[0...n]$, the error coverage (of double errors) *on average* is 50%.

A major drawback of residue checking, is that it is only capable of checking *arithmetic operations* (it is designed for checking arithmetic circuits only). It is however possible with some modifications to enable checking logical operations as well, but this increases the overhead of the scheme significantly.

The true power of this scheme is the ability to modify the modulus. The higher the modulus, the higher the fault coverage of double errors, but also the higher the cost of the scheme. Note that modulo-3 checking already requires 2 additional bits per operand (parity prediction requires only one), and e.g. modulo-15 checking would require 4 extra bits.

The paper of Langdon and Tang [70] teaches that, under certain circumstances, modulo-3 checking can be cheaper than parity prediction. In section 6.15 a more detailed comparison is made between parity prediction and modulo-3 checking, based on further research. We will explain under which circumstances modulo-3 checking is advantageous over parity prediction, and when it is not.

### 6.8.3 Berger and Bose-Lin Codes

For the **Berger code**, a code word is formed by appending a binary string representing the *number of logic ones* (or zeros) in the given operand [67]. For an $n$-bit operand we

need $\lceil log_2 n \rceil$ check bits. Note, that for a 16-bit operand we already need four check bits. The Berger code is based on **unidirectional** error detection, which means that the code assumes that all errors are unidirectional: zeros are changed into ones or ones are changes into zeros, but never both at the same time. To ensure that a single fault causes an unidirectional error at the output, the logic circuit must be synthesized in such a way that the circuit itself is *free from inverters* (only at the primary inputs of the circuit inverters are allowed). This complicates the synthesis of such a circuit. Only then the Berger code is capable of detecting all unidirectional errors. Whenever the number of ones in the result differs from the predicted value, an error is detected. Thus, Berger code can detect any number of one-to-zero (or zero-to-one) bit-flip errors, as long as no zero-to-one (or one-to zero) errors occur in the same code word. This sentence summarizes both the strength and the weakness of the code. Berger codes can be used for error detection in telecommunications, arithmetic and logical operations [71].

When utilizing Berger code in adders, the ED-scheme is called **Berger Check Prediction (BCP)**. If we have two n-bit numbers $X$ and $Y$, to obtain the sum $S$ with internal carries $C$, and $N(X)$ denotes the number of binary ones in the number X, then:

$$N(X) + N(Y) + c_{in} = N(S) + c_{out} + N(C)$$

For an n-bit number $X$, $X_c = n - N(X)$ where $X_c$ is called the **Berger check symbol** (which is actually the number of zeros in the binary number $X$). Together with the formula above, we are able to predict the Berger check symbol of the sum of the addition:

$$S_c = X_c + Y_c - c_{in} - C_c + c_{out}$$

And for 2's complement subtractions:

$$N(X) + N(\bar{Y}) + \bar{c_{in}} = N(S) + c_{out} + N(C)$$

In the paper of Lo et al. [71] a BCP scheme for utilization in adders/ALUs is presented. The scheme is depicted in figure 6.2. The circles with the plus sign in it represent exclusive-or gates and the MCSA is a multi-operand carry-save adder (refer to [71] for the implementation of the MCSA). The scheme is an analogous implementation of the equations given above. According to this paper, the first important motivation for utilizing the BCP scheme, is the inability of other error code based schemes, such as parity prediction and residue checking, to check both arithmetic and logical operations. It is true that residue checking is not capable of checking logical operations, but parity prediction most certainly is, according to the proof given in the paper of Nicolaidis [65]. Thus, Lo et al. appears to be incorrect at this point. The second important motivation for BCP is the fact that neither parity prediction nor residue checking are able to achieve the TSC goal in their predictors (producing the predicted parity/residue). Note, however, that only recently a TCS parity-prediction scheme is presented by Nicolaidis [65]. This scheme will be discussed in section 6.16.3. BCP achieves the TCS goal in the predictor (producing the Berger check symbol), with respect to any *single fault* in the adder/subtracter. This makes BCP a very reliable ED-technique. BCP is capable

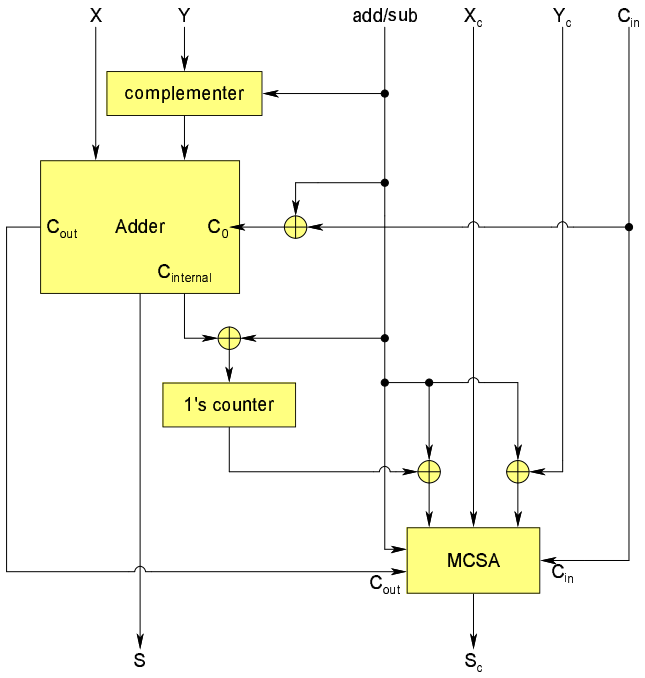


Figure 6.2: Berger Check Prediction [71]

of detecting both *single and double errors*. On top of that, it is capable of detecting *all unidirectional errors*. This in contrast with parity prediction, which is capable of detecting unidirectional errors of *odd multiplicity* only. BCP is therefore superior to parity checking, but still cannot measure itself with hardware duplication.

The **Bose-Lin code** is alike the Berger code but has a few fundamental differences. The Berger code is able to detect all unidirectional errors, but in many systems there is no need to cover all, but only a limited number of errors. The Bose-Lin code can detect '$t$' errors, and the value of '$t$' is basically up to the designer and is related to the number of check bits '$r$' being utilized. The most common values for '$r$' are 2 (covering double errors) or 3 (covering triple errors). Note that we are referring to unidirectional errors. The other fundamental difference with the Berger code is that the Bose-Lin code has a *fixed* number of check bits '$r$', independent of the number of information bits. This is possible because Bose-Lin check prediction is based on *modulo operations* (also the number of ones is represented as a residue; mod 4 and mod 8 are utilized for $t$=2 and $t$=3, respectively) [72, 67]. Altogether, this makes the Bose-Lin check prediction a lot *cheaper* than BCP (how much depends on the selected value of '$t$'). According to Gorshe et al. [73], **Bose-Lin check prediction** assures *fault-secureness for single errors* in adders. Also, Bose-Lin check prediction is capable of checking all *logical operations* as well.

Unfortunately, neither Lo et al. or Gorshe et al. provide a comparison between Berger/Bose-Lin check prediction and other ED-techniques based on hardware costs. The paper of Mitra and McCluskey [67] clearly shows that the area cost of Berger check prediction is very high compared to parity prediction, and even higher than full (identical/diverse) duplication. But that raises the question why the BCP scheme has

never been dismissed. A later published paper by Lo et al. [74] explains that is difficult to say whether the BCP scheme is doing better or worse than duplication in terms of hardware requirements. It depends on the exact implementation of the BCP scheme but also on the architecture. E.g., if we utilize Berger code to check both the ALU and the memory/data transfers, we might save code translators. According to Lo et al. [74] the application of BCP to an Intel 8080 processor proves to be an efficient method to design a strong fault-secure self-checking processor.

## 6.9 RESO

All previously discussed error-detection techniques are based on *hardware redundancy*. The most obvious one is full duplication, but also error-detection codes introduce a significant amount of redundant hardware. **RESO**, short for **Recomputed with Shifted Operands**, is based on **time redundancy** [68]. The operation requires two steps. In the first step, the operands $X$ and $Y$ are applied to the arithmetic unit in the normal way and the result is stored in the result register. If we would repeat this step and compare the result to the previous result, only a *transient* error occurring during either of two computation steps could be detected. Obviously, we want to detect all errors: *soft errors* and *permanent errors*. The way to do that is to *encode* the operands in the second step, before applying them to the AU, and after processing *decode* the result in order to compare is with the result from step one. Thus, if $f(x)$ represents the computation inside the AU, and the functions $d(x)$ and $c(x)$ represent the decoding and encoding respectively, then $d(c(f(x))) = f(x)$. The reason why we want this, is because the AU is now computing with different operands: instead of with $X$ and $Y$ the AU is now working with $c(X)$ and $c(Y)$. After decoding, normally the result will be the same, but any errors in the AU will now manifest at *different positions* in the result, compared to the result of step one. In most cases the RESO scheme is implemented as depicted in 6.3. The encoding is performed by a simple *shift left*, and the decoding by a *shift right*. More about the RESO ED-scheme can be found in the paper of Patel and Fung [75].

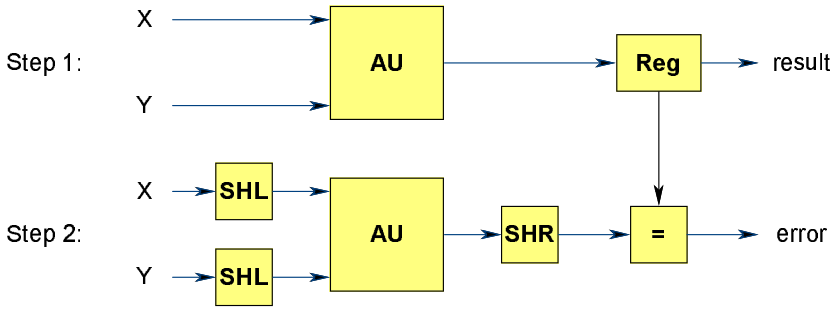


Figure 6.3: Error detection using RESO [68]

It is clear that this ED-scheme requires very little redundant hardware. The scheme requires two left- and one right-shifter, and the AU must be one bit wider than the $n$-bit operands. On top of that an $n$-bit comparator is necessary. On the other hand, the speed penalty is huge, since two computational steps actually means two clock cycles per

operation. Even when this does not result in problems with the throughput, there will arise problems with the energy consumption. RESO will most probably have one of the lowest power consumptions of all ED-schemes, since the required amount of hardware is low. This means that the energy consumption of the AU *per cycle* will not increase much when the RESO scheme is added. But, since two cycles are needed, the required amount of energy *per operation* is huge: well over 100% more than without ED (referring to power consumed in combinational logic only, not register power). Apart from checking arithmetic operations, RESO is also capable of checking *logical operations*.

## 6.10 Achieving the TSC goal for predictors and checkers

As mentioned in section 6.6, adders are by definition TSC circuits. This is not the case for added ED logic. As explained in section 6.8.3, predictors predicting the check bits of the result of the calculation may or may not be TSC circuits. But even when the Berger check predictor is said to achieve the TSC goal, is does not mean that the entire ED circuit achieves the TSC goal. This can be confusing. There is a difference between producing the check bits and producing the error signal. In the discussion about the BCP in the paper of Lo et al. an important component is missing: *the checker*. A checker should here contain a counter to count the number of ones in the *actual result* (real check symbol) and a comparator to compare the predicted check symbol with the real check symbol. If the checker does not meet the TSC goal, the entire ED scheme does not meet the TSC goal. Nevertheless, if the predictor meets the TSC goal, the reliability of the scheme is always enhanced.

Often, the checkers utilized in ED schemes, are simple comparators. There might occur a situation where, for example, the output of a comparator gets stuck-at-0. Then, in the case when an adder starts producing erroneous results, the error-correction scheme will not intervene, since it seems there is no error present. This makes the comparators crucial elements in the ED/EC scheme. To cope with this problem, a checker must preferably be *self-testing* [76]. Thus, for each modeled fault, there must be a code input for which the checker produces a non-code output. Obviously, this is impossible with a single-output circuit such as a comparator.

The first implication of the self-testing property is that a checker must have at least two outputs [77]. This means would have to implement the comparator by an alternative technique, if we want to meet the self-checking goal, in order to increase the reliability of the circuit. A common technique to do so is to employ **dual-rail checkers**, which have two outputs (a dual-rail checker cell is in terms of functionality identical to a XOR-gate and therefore dual-rail checkers are very applicable for performing comparisons). A dual-rail checker cell is depicted in figure 6.4. Then, e.g., the output of the checker "01" would indicate a correct result, and "10" an erroneous result (an error in the adder). The other two possible outcomes would indicate an error in the checker. Dual-rail checking however increases hardware overhead. If the self-testing checker also satisfies the *code-disjoint* property, the checker meets the TSC goal. A predictor which meets the TSC goal is presented in 6.16.3. Because of limited time, none of the predictors/checkers that are actually implemented (section 6.17) have been designed to meet to TSC goal. This would be a topic for future work.

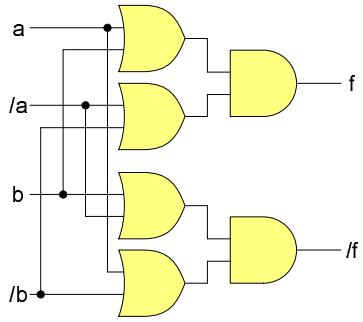


Figure 6.4: A dual-rail checker cell

## 6.11 Error correction

As discussed earlier, detection of errors is the first but not sufficient step to make a system fault-tolerant. We need to be able to *correct* errors as well. There are, basically, two methods of error correction: **masking** the error or (online) **repair/reconfiguration** [57]. By masking the error, the circuit itself is designed in such a way that it dynamically corrects a generated error, *without* having to reconfigure or resort to backup circuits (thus, the error itself is not fixed). In the case of repair/reconfiguration, the circuit is dynamically repaired when an error is detected. This means that faulty circuits are eliminated, replaced, or bypassed. For example, a particular erroneous circuit can be shut down, a backup circuit can be activated, and the data path reconfigured.

An example of an ED/EC-adder purely based on hardware replication is the **TMR (Triple Modular Redundancy)**. Here we have three adders, operating simultaneously, and feeding their results to a majority voter. This voter seeks for a match between any two of the three results, and forwards that result to the output. So, when one of the adders is producing erroneous results, the TMR still functions correctly. Note that the TMR is based on error masking. Even though an adder is producing erroneous results, the adder is not shut down, nor is another adder activated, nor is there a reconfiguration of any kind. Another ED/EC-adder based on pure hardware replication is the **QMR (Quadruple Modular Redundancy)**. Two adders and a comparator form together an **SCA**, a **Self-Checking Adder**. If an error is detected, we need to resort to another SCA. Note, that it is impossible to obtain information about which of the two adders inside the SCA is producing erroneous results and therefore, the QMR contains two SCAs, with both outputs attached to a multiplexer. In total, the QMR scheme contains four adders. Also the QMR is an example or error masking [44, 57].

Another example of error masking are **error-correction codes (ECCs)**. These codes are more advanced than error-detection codes (error correction is much more complicated than error detection) and have the ability to both *detect and correct* an error [57]. We, however, will not discuss ECCs in this thesis. In the general case, ECCs are (disproportionally) expensive, especially for tiny architectures as the ones we are interested in. For example, a very common ECC is based on the Hamming code, which is often employed for single-error correction. For this ECC we need $(log_2 N) + 1$ check

bits, where $N$ is the word-width [78]. So, in our case, where the word-width is 16 bits, we would already require 5 check bits. Also an ECC based on residue coding would require *multiple* residues [79]. This will have a major impact on the area and power costs of the arithmetic unit, as well as on the entire micro-architecture if these check bits/digits are generated early in the pipeline. According to the book of Chen (section IX, chapter 8.2) [79], in arithmetic circuits ECCs are incompatible with other methods. However, we are well-aware that there *may* be certain error-correcting schemes (with acceptable levels of error correction) that are cheaper in terms of power consumption and area requirements than some of our more expensive reference designs, such as the QMR. Without underestimating the importance of this topic, practical time limitations of this thesis work do not permit delving into such special issues. We consider this as an important direction for future work.

When utilizing *error-detection codes*, like parity prediction, we can build an SCA *without* replicating the adder inside. However, when the scheme detects an error, we still need to resort to duplicated hardware (backup hardware) in order to be able to *recover* from the error. The solution is then to implement two SCAs (as well as the required EC logic and a multiplexer, based on the scheme of the DAU-RAS which was explained in chapter 5, section 5.4). When the active SCA is erroneous, it is shut down, the backup SCA is activated, the data path is reconfigured, and the calculations proceed by employing the backup SCA. This is a scheme based on online repair/reconfiguration.

Note that the comparators in the DAU-RAS and QMR scheme, and the voter in the TMR scheme, are critical elements and the weak links in the design, since they are not checked for errors. It is possible to apply self-checking design to these components as well (checking the checkers). Another solution is self-testing design (as explained in section 6.5, a circuit is self-testing if there exists at least one input pattern in normal operation that produces an erroneous result for each fault in the faults set). Ideally the entire design, including the error-detection (checkers and predictors, as discussed in section 6.10) and -correction logic, should therefore meet the self-testing property. To give an example of self-testing design is to employ double-rail checking for the comparators in the QMR (double-rail checking was also discussed in section 6.10).

## 6.12 Minimal required error coverage in ScAU

As mentioned in section 6.8, single errors occur much more often than double errors. The fault coverage does not increase significantly if we decide to extend the ED-scheme for double errors or errors of even higher multiplicities. What does certainly increase quite dramatically, is the *overhead* of the ED scheme. What we must realize is that the AU is to be designed for application within the SiMS micro-architecture, and thus, for utilization within medical implantable devices. Then it is obvious that reliability is crucial. Even though the chance that a double error occurs is small, the chance is *not zero*, so ideally we should check for double errors as well.

On the other hand, as explained previously in chapter 1, there is a reason why the SiMS micro-architecture is so minimalistic and why the power and area budgets are so low. In biomedical implants, batteries should have a very long life span and since the physical dimensions of the implant are bounded (implants tend to become smaller and

smaller, enabling them to be implanted in parts inside the body where there is not much space), and the vast amount of area within the implant is used for the battery pack and the electrodes, the available chip area is very limited. So, increasing reliability might lead to intolerable power and area requirements (which would lead to significantly shorter battery lifetimes, or require larger batteries, which is not a preferred option) and larger circuit area (which leaves less space for other core structures which have been shown to benefit implant operation, such as caches [7]). So also for a highly mission-critical device such as a biomedical implant, the decision to be made is based on a trade-off between reliability and cost, like it is the case for virtually every application. We ultimately made the decision to have the design be at least *fault-secure for single errors*. So, we decided to choose for the minimum possible fault coverage. We believe this is justifiable because of the following. First, we established earlier that single errors occur much more frequently than errors of higher multiplicities. The extra protection against double or even triple errors does not add significantly to the probability of detecting errors. Second, since the AU will not be employed all the time and will be idle for longer periods of time (within the SiMS application), the intention is to periodically run online tests with a set of predetermined test vectors to check the full functionality of the AU. This way, some errors may not be covered by the ED-scheme, but ultimately they will be captured by the online testing.

## 6.13 Applicable ED/EC techniques for the ScAU

Whether an ED scheme is applicable for implementation in the ScAU depends on two factors: first, the cost of the scheme should be acceptable, and second, the scheme should be compatible with the scalability feature. Since the ScAU is subdivided into two adder segments (in normal operation mode placed in series with each other), we can apply the ED-scheme to each of the segments *independently*. However, achieving fault-secureness for single errors in the ScAU *as a whole* is difficult when two independent ED-schemes are employed, as we will see later on in section 6.17.7. Duplication to enable error correction is not necessary in the ScAU: if one segment fails, we shut it down, together with the ED logic, and continue with the other segment. The QMR and TMR do have a phenomenal fault coverage (capable of detecting all errors of any multiplicity, except common-mode errors), but the costs of these ED/EC schemes are very high. For the QMR scheme the power and area costs are at least 400% the costs of the unchecked AU, because of the simple fact that we need four adders. Therefore, the QMR scheme is not interesting for implementation in the ScAU. The TMR will probably be cheaper than the QMR (we will actually prove this in section 6.17.6), but still the costs will be very high (3 adders operating simultaneously), and therefore also the TMR is not applicable for implementation in the ScAU.

Both parity prediction and modulo-3 checking seem, based on the literature, interesting ED-techniques for implementation in the ScAU. The fault coverage of both schemes is identical and sufficient for our purpose: both schemes are fault-secure for single errors and cover 50% of all double errors. However, it is not easy to predict which technique is most suitable for the ScAU. In the literature, parity prediction is said to be cheaper than modulo-3 checking, but Langdon et al. [70] proves otherwise if we meet a

number of conditions. On top of that, comparison is complicated, since more than one parity-prediction scheme exists. Therefore we decided to make a detailed comparison between parity-prediction and modulo-3 checking, based on thorough research, before we decide if we implement modulo-3 checking or not (see section 6.15). In section 6.16, three different parity-prediction schemes are discussed and two of them are implemented in order to compare the costs. Since both parity prediction and modulo-3 checking are error codes, they work with check digits. Regardless of where we generate the check bits of the operands (inside the AU or early in the pipeline), check bits generated for 16-bit operands are *useless* when we decide to check the upper and lower byte, as we do when we check both segments independently. If we decide to generate the parity bits early in the pipeline we must **generate** *two check digits per operand* (lower and higher byte), or we must **derive** the lower and higher check digit from the 16-bit check digit inside the ScAU. Later will be shown which method is the most cost-effective.

We consider Berger check prediction (BCP) not suitable for implementation in the ScAU, since the hardware cost is at least close to that of full hardware duplication (but most probably more costly). Apart from that, we do not need the high fault coverage that BCP offers (covering single and double arithmetic errors, and covering all unidirectional errors). Also the area cost of Bose-Lin check prediction is reported to be (prohibitively) high, but Mitra and McCluskey [67] do not mention how many code-bits they utilized. However, since Gorsche et al. [73] reports that in a 16-bit adder, with 2 Bose-Lin code bits (capable of detecting single arithmetic errors, and double unidirectional errors), the area cost is significantly lower than when utilizing Berger check prediction, Bose-Lin check prediction (with a limited number of code bits) might be cheaper than hardware duplication. Implementation and analysis would be necessary to provide us an answer, since it cannot be found in the literature. It is, however, highly unlikely that Bose-Lin check prediction will ever be cheaper than parity prediction. After all, unlike parity prediction, Bose-Lin check prediction is designed for detecting multiple errors and requires at least two check bits. Because of the costs, Bose-Lin check prediction is most likely not applicable for implementation in the ScAU. However, Bose-Lin check prediction is without doubt an interesting code when we require a (slightly) higher error coverage than parity prediction or modulo-3 checking. Since there is not much literature available about Bose-Lin check prediction, and we only have limited time, future research is desired to study the exact costs of Bose-Lin coding and make an accurate comparison with other ED schemes (such as residue checking).

As discussed previously, RESO is superior to most (if not all) other ED-techniques when we consider area cost, but the energy per operation more than doubles since we need dual-cycle operations. Note, however, that only the AU, shifters, and comparator need to be active in the second cycle, so the operand and result registers which consume a significant portion of the total power consumption, can be clock-gated to prevent switching activity. But even then, the expectations are that the energy per instruction when utilizing RESO will exceed by far the energy per instruction when utilizing single-cycle techniques such as parity prediction. Since we assign a *higher weight to power than to area* savings (as discussed in chapter 1), the RESO technique is less suitable. There is another reason why RESO is not suitable for implementation in the ScAU. In normal mode, the ScAU performs one operation/cycle. When applying RESO,

this would become one operation per two cycles. But in downscaled mode, when a AU segment is damaged, we would have to switch to one operation per *four* cycles. So, RESO complicates the design of the pipeline dramatically and, in case of errors, leads to a throughput that is most likely unacceptably low.

## 6.14 Reference designs

Even though QMR and TMR schemes are too costly for implementation in a micro-architecture with very low-power and low-area budgets, like the SiMS architecture, the decision was made to implement these schemes anyway. The first reason for that is because these schemes are very common and provide the comparison of other ED/EC designs with more depth: by including these schemes we provide a much more extensive comparison between different ED/EC schemes and we can actually prove that the QMR and TMR schemes are not suitable when we have to cope with very small power and area budgets. Second, by including these schemes as well, we provide a more general comparison. The intentions are the same as with the adder study in chapter 4. The results of this ED/EC study can be utilized by anyone who requires a fault-tolerant arithmetic unit in their design. Another less obvious reason for the QMR being implemented is because the original design can be optimized for lower power consumption. As explained, the QMR contains two SCAs, both simultaneously active. There is, however, no need for both SCAs to be active simultaneously. What we did is modify the design in such a way that only one SCA is active at a time. When the active SCA detects an error, this affected SCA is shut down and the non-active SCA is activated. This modification will increase the area of the scheme, but we are interested in the power savings. Note that this QMR scheme (we call it QMR-RAS; Redundant Adder Shut down), is no longer an error-masking scheme. Since the failing component is shut down, a new component is activated and the data path is reconfigured, this scheme is now based on *repair/reconfiguration*. Further, the last and most important reference design is the DAU-RAS, or Duplicated Arithmetic Unit. This design is built by two AUs, each of them checked by a certain error-detection code. Also this scheme is power-optimized (based on repair/reconfiguration): only one AU is active at a time (therefore the naming RAS).

## 6.15 Parity Prediction vs. Modulo-3 checking

According to Mitra and McCluskey [67], modulo-3 is *never* economical unless the operands are already provided with the modulo-3 check bits. This has to do with the *generation* of the check bits: generating modulo-3 check bits (the *residue*) is considerably more expensive than generating parity bits [67, 70]. According to Langdon and Tang [70] the **modulo-3 predictor** is cheaper than the **parity predictor**, as long as the operands are *already* provided with the check bits. The reason why modulo-3 checking can be cheaper than parity prediction is because the **modulo-3 checker** itself is cheaper than the **parity checker**; a modulo-3 adder to predict the *residue* of the result is significantly cheaper than the circuit predicting the *parity bit* of the result [70].

A closer look shows that Langdon and Tang [70] utilize a carry-lookahead adder and a high-speed parity-prediction circuit. The high-speed parity-prediction circuit is employed to avoid unnecessary delays by the ED [62]. Since the predicted parity is available considerably later than the result, this affects the throughput of the adder. Ideally, the result and the predicted parity should be available at the same time. High-speed parity-prediction does, however, increase the costs of the ED scheme. Our primary concerns are low-power consumption and low-area. If some delay has to be traded for lower power and area usage, this will be tolerable (the SiMS architecture's operating frequency is relatively low, so we have enough time to perform relatively slow ED operations, especially given the speed of modern CMOS technology). What is more, high-speed parity prediction is not even applicable to RCAs, the type of adder we employ in our AU. By omitting the employment of high-speed parity prediction, the costs of modulo-3 checking will *no longer be lower* than those of parity prediction. Langdon and Tang [70] do not explicitly mention this in their paper.

As mentioned before, generating the residues inside the AU is not efficient. This implies that there must be use for the modulo-3 check bits for *other* error-checking purposes in the architecture as well, apart from the AU. When an architecture contains for example multiple adders/subtracters, a multiplier and a divider, modulo-3 checking can perfectly be employed to check all these units. Modulo-3 checking can also be employed to check data transfers and memories. Modulo-3 checking does, however, not seem to be particularly interesting for our purpose: in the SiMS architecture we have only one AU, and no multiplier or divider. Apart from the AU, the ALU contains also an LU (logical unit), but modulo-3 checking is an arithmetic error-detection code and *cannot* be employed for checking logical operations (as discussed in section 6.8.2). It is possible to modify the AU including modulo-3 checker to check logical operations as well, but this requires additional hardware. What is more, this approach makes it unable to *separate* the AU and LU, which is undesirable if long sequences of logical operations occur frequently (simulations of the SiMS architecture show that this is the case). On top of that, checking memories and data transfers by parity prediction is more efficient since we require only one parity bit per data word.

Thus, the following conclusion can be drawn: modulo-3 checking can be cheaper than parity checking for larger, fast adders (provided that the check bits are already present and that the architecture has use for the check bits in more cases than just the adder, e.g. when also a multiplier is present in the architecture). But still, in most situations parity checking is to be preferred over modulo-3 checking, when we consider the two major drawbacks of the scheme mentioned in the previous paragraphs. For this reason, a modulo-3 checked arithmetic unit is not implemented in this study. The decision is made to implement a parity-prediction scheme, to compare the results with the previously mentioned hardware replication techniques.

In general, if we desire a *higher degree* of error detection (higher than fault-secure for single errors and 50% coverage of double errors), residue checking becomes interesting. Note that area and power increases significantly when we increase the modulus (the overhead scales linearly with the amount of check bits utilized). Particularly, for our specific purpose (small-size adder/AU), the overhead of utilizing a larger modulus would be dramatic. The QMR(-RAS) scheme would probably be a better (cheaper) approach,

if a higher degree of error detection is desired.

## 6.16 Selection of parity-prediction scheme

There exist several parity-prediction schemes which are fault-secure for single errors. We will discuss three of them. These three schemes are chosen since they appeared the most promising in terms of low-power consumption, low-area, and reliability.

### 6.16.1 Duplicated-Carry Scheme

Any parity-prediction scheme for adders contains the following parts:

- Parity generators to produce the parities $P_A$ and $P_B$ of the operands (XOR-trees)
- Parity generator for the internal carries to produce $P_C$
- Parity generator to produce the parity of the sum $P_S$
- Predictor/comparator which requires $P_A$, $P_B$, and $P_C$ to predict $P_S$ and compare it with the real $P_S$, and signals an error when they do not match

As established earlier, carry errors *always* manifest as *several* errors of *even* multiplicity. Therefore, the parity-prediction scheme normally would not be able to detect these type of errors. Sellers et al. [62] proposed a parity-prediction scheme with *duplicated-carry circuits*, as one of the first parity-prediction schemes which is fault-secure for single errors (i.e. also capable of covering all carry errors).

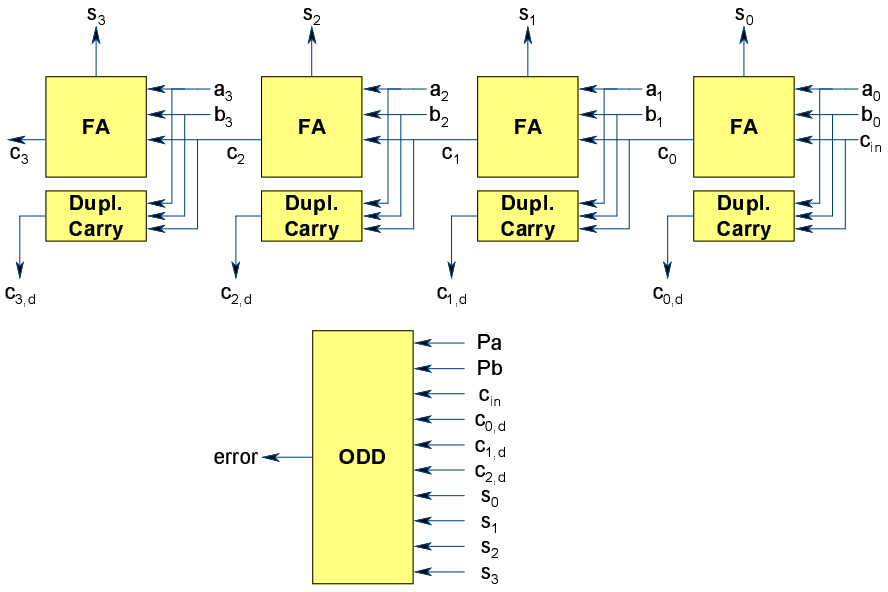


Figure 6.5: Parity checking with duplicated carries

When the carry circuit of every full-adder cell is duplicated, we are able to *compare* the carry output of the full-adder cell and the duplicated-carry circuit. Since we aim for

fault-secureness for single errors, only one carry circuit is assumed to be erroneous at a time. If we feed these compare signals to the checker, we are now able to detect this single carry error since the maximum number of detected carry circuit errors is one, and thus odd. Sellers called this scheme "Duplicate carry with parity check I". The actual comparison between the 'normal' carries and duplicated carries can be avoided to simplify the scheme. It is sufficient to generate $P_C$ based on the duplicated carries (instead of the normal ones) to achieve the fault-secure property, since the total number of errors in the sum plus the total number of errors in the duplicated carries will always be odd in case of a carry error. The reason for this is that the corresponding duplicated-carry circuit of the normal carry circuit, which contains the fault, does not produce an erroneous result. This means that the total amount of carry errors in the duplicated carries contains always one error less than in the normal carries. This scheme is called "Duplicate carry with parity check II". The duplicate carry with parity check II scheme is depicted in figure 6.5. Note that the $P_A/P_B$ parity generators are omitted in this figure. The *ODD circuit* is the checker, $P_S$, and $P_C$ generator all in one. The design is nothing more than an $n$-input XOR. If the number of ones at the inputs is even, the output is zero, otherwise it is one.

### 6.16.2 Carry-Dependent Sum Adder Scheme

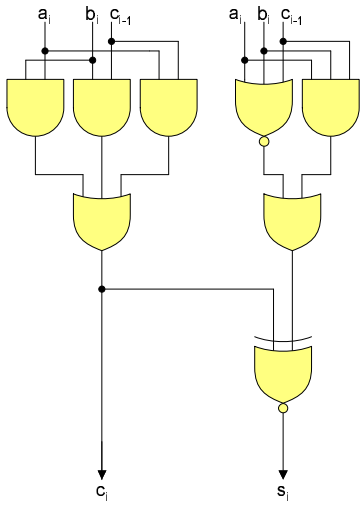


Figure 6.6: The carry-dependent sum (full-)adder [80]

The parity-prediction scheme based on the **carry-dependent sum adder (CDSA)** was proposed by Hsiao and Sellers [80, 62]. The most elementary distinction with the previous scheme is that the carry circuits are not duplicated. As explained in section 6.4, if an error in $c_n$ causes an error in $c_{n+q}$, then $c_n$ also causes an error in $c_{n+1}, c_{n+2}, ..., c_{n+q-1}$. All carry errors lead to (successive) erroneous sum results as well: the results $s_{n+1}, ..., s_{n+q+1}$ are in error too. Since carry errors always cause an even number of errors (which makes them undetectable), in the CDSA scheme the full-adder cells are modified in such a way that this is no longer the case. The idea is to make sure

that when the error occurs in $c_n$, also $s_n$ is in error. If that can be done, $s_n, ..., s_{n+q+1}$ is in error, having one error more than before, turning the total amount of sum and carry errors into an *odd* number. This is what the carry-dependent sum (full-)adder does. As the name suggests, the sum is dependent on the carry. If the carry is affected by a fault, so is the sum. A carry-dependent sum full-adder cell is depicted in figure 6.6. The checker can be implemented by a simple ODD circuit:

$$error = P_A \oplus P_B \oplus P_C \oplus P_S$$

### 6.16.3 Nicolaidis's Scheme

Nicolaidis [65] proposed a new carry-checking/parity-prediction scheme. It is based on the duplicated-carry scheme but has been optimized in several ways. This scheme is the only parity-prediction scheme discussed in this thesis where a large portion of the prediction/checking logic ($P_C$ generator and the the carry-checker) meets the TSC goal as well. The reliability of this scheme is, therefore, higher than the previous ones. The scheme is a combination of *double-rail checking* of the carries and *parity prediction* for the outputs. Normally, double-rail checking results in high overheads, but there are *three hardware reduction techniques* applied to limit the costs. First, the double-rail checker and the carry parity generator collapse into a single block. Second, the scheme avoids duplicating complex blocks, such as the carry-lookahead or carry-skip block required for performing carry checking. Third, some hardware can be saved by employing partial carry duplication, instead of duplicating the carry circuitries entirely (by sharing some logic between the normal and duplicated-carry circuits). The primary function of the double-rail checker is to check the normal carries with the (partially) duplicated carries. Both the (inverted) normal carry and the duplicated carry are provided to the double-rail checker. Note that the individual comparison of the normal and duplicated carry per bit-slice *can* be omitted, as explained in section 6.16.1. Nicolaidis decided to choose this approach because it enables the implementation of a double-rail checker which increases the reliability of the checking logic. The reason why the double-rail checker and the parity generator can be merged is because double-rail checkers have a *one-to-one correspondence with parity trees*: they can be seen as parity generators with double-rail in- and outputs. Thus, apart from checking the carries, the double-rail checker immediately provides Pc as well, with no need for extra hardware. Since double-rail checkers are *self-testing*, the $P_C$ predictor meets the TSC goal. Note that this is the only known parity-prediction scheme which meets this property. Even though the costs of this scheme are said to be low, we have sufficient reasons to believe that the costs of this scheme will not be any cheaper than the costs of the duplicated-carry and CDSA schemes. More likely, the costs will be higher. One of the hardware reduction techniques (the second one) is *not* applicable in our situation, since we work with simple RCAs, not with fast adders such as CLAs. It is true that the carry checker and carry generator are collapsed into one double-rail checker and that carry circuits are only partially duplicated, but still, double-rail checkers have much higher overheads than normal parity generators (area of a double-rail checker cell vs. area of a XOR-gate = 18 vs. 11 logic units, according to the UMC 90nm databook). Ultimately, implementation, synthesis and analysis of this scheme would be required to make an accurate comparison with the previously discussed

parity-prediction techniques and to draw a conclusion based on hard evidence. Because of limited time, we omitted this work and we mark it for future research.

### 6.16.4 Final choice

The final choice between the duplicated-carry and CDSA scheme will be based on costs, since the fault coverage of both schemes is *equal*. In the CDSA scheme, the full-adder cells are more complex, but the carry circuits are no longer duplicated. However, it is to be expected that the costs of the CDSA are lower but in the literature no comparison between the two parity schemes could be found. Therefore, in the next section both schemes are implemented, synthesized and analyzed. Our assumptions turn out to be correct. It is the *CDSA scheme* that is chosen for implementation in the ScAU.

The TSC property of the parity-prediction scheme of Nicolaidis provides, however, a significant advantage in reliability, so ultimately the scheme of Nicolaidis might be a better choice, provided that the costs of the scheme are acceptable. Since the exact costs of this scheme are unknown, this is a topic for future research.

Note that for implementations of the ScAU in other architectures than the SiMS micro-architecture, another error-detection scheme might be more suitable, for example with a higher fault coverage. We believe, however, that this study provided sufficient information about various error-detection techniques for any designer to make a well-considered choice.

## 6.17 Implementation of ED/EC in ScAU and reference designs

In this section will be explained how the ED/EC scheme we selected in the previous sections are implemented in the ScAU and in the most important reference design; the DAU-RAS. Also the implementation of the other reference designs (the TMR, QMR, and QMR-RAS) will be discussed. An important aspect of this chapter is that we no longer focus on plain adders, but on arithmetic units: units capable of both addition and subtraction. Also, the computation of the zero and overflow signals is part of the AU's task.

### 6.17.1 Subtraction

So far, the error detection of binary addition has been discussed. Obviously, an arithmetic unit is also capable of **subtracting** numbers so it is a legitimate question whether the error-detection methods described are capable of checking subtractions as well. First of all, subtraction is no different than addition, except from the fact that operand B needs to be *negated*. In our case all numbers are in *2's complement* representation. Negating an operand thus requires a 1's complementation and an addition of a binary '1'. Chapter 5 discussed subtraction in detail. What matters here, is what the impact of the complementer is on the error-detection scheme. When the operand has an even number of bits, the complementation will *not* change the parity of the operand. However, every single error in the complementer will change the parity. Then, the operand enters the

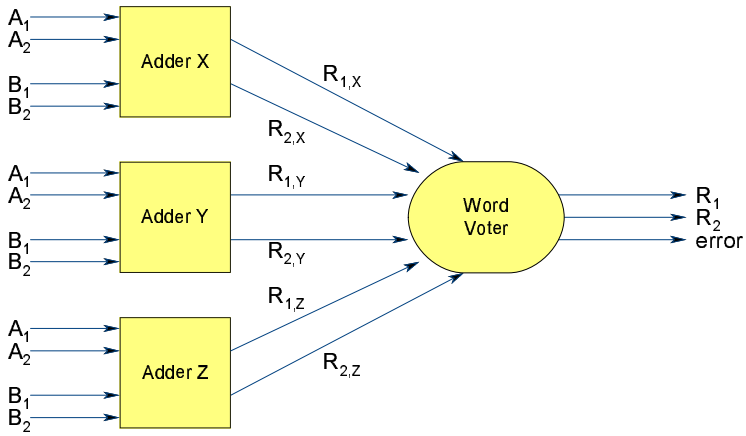


Figure 6.7: The TMR scheme with word-voter [81]

adder with a parity bit that does not match, thus results in a detected error. Obviously, a double error (in fact, all errors with an even multiplicity) in the complementer will remain undetected, but this is also true for 50% of all double errors in the adder itself. The conclusion is that none of the error-detection schemes designed for checking adders needs to be modified for handling subtractions as well.

### 6.17.2 The TMR

The TMR is implemented by the **word-voter scheme** proposed by Mitra and McCluskey [81], which is depicted in figure 6.7 (note that this is a two-bit TMR). The fundamental difference with regular TMR schemes is the *error output* of the voter. Regular voters mask erroneous outputs originating from one adder. But if two or all three adders produce errors (multiple errors/ common-mode errors), the TMR scheme does not signal this (it is simply unaware), creating a situation where erroneous computations pass the ED freely. The word-voter is an entirely different approach. Instead of voting the results *bit-by-bit*, the results are first —as a word— compared by three comparators as depicted in figure 6.8. All possible combinations (X,Y), (X,Z), and (Y,Z) are compared. If adder X, Y, and Z all produce different results, either two or all three adders are damaged. In that case, the error signal is '1'. As long as two adders produce identical results, error='0'. It is obvious that the addition of the comparators increases the hardware cost of the voter significantly. On the right side in the figure the voting logic is depicted. Thanks to the comparators, the voting logic can be simpler than in regular voters. In regular voters the result at bit position '$n$' is: $R_n = R_{n,x} \cdot R_{n,y} + R_{n,y} \cdot R_{n,z} + R_{n,x} \cdot R_{n,y}$. But since we already know whether the word $R_x$ is equal to the word $R_z$ ($\gamma = R_x \cdot R_z$), we can simplify the logic to $Rn = \gamma \cdot R_{n,x} + \overline{\gamma} \cdot R_{n,y}$.

### 6.17.3 The QMR and QMR-RAS

Implementation of the QMR is very simple. The figure of the DAU (Duplicated Arithmetic Unit) in chapter 5 (figure 5.3) is essentially the basis of the design. Instead of

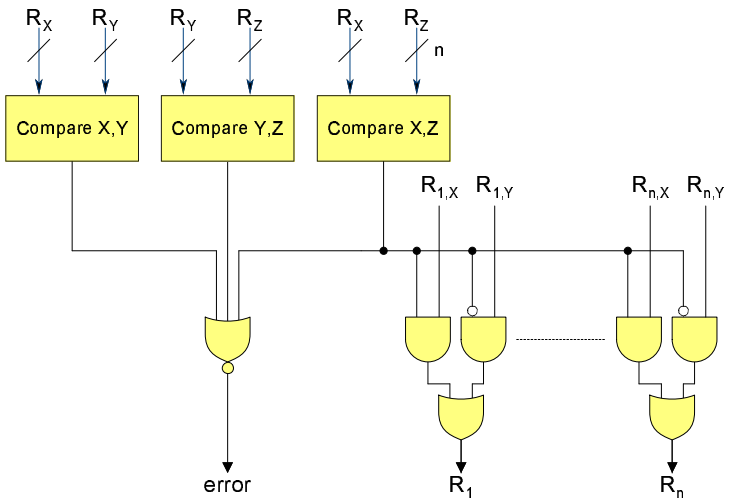


Figure 6.8: Implementation of the word-voter with error output [81]

two AUs we placed two SCAs: each containing 2 AUs and a comparator. In the standard case the left SCA is active. The moment the comparator signals an error, the input of the multiplexer selects the output of the right SCA. If the error is temporary, the left SCA takes over again. Finally, there is a fail signal which is asserted when both SCAs are in error.

The design of the **QMR-RAS** (the power-optimized version of the QMR, where only one self-checking adder is active at a time) is similar, with the difference that the inputs of the SCAs are *input-gated* by tri-state buffers. If an error occurs, the occurrence is stored in a register and at the next rising clock edge the erroneous SCA is disabled and the backup SCA is activated. So, in case of an error, the QMR-RAS produces no useful result for a single cycle, before the other SCA takes over. The pipeline will have to build in a **repetition cycle** to be able to obtain the correct result of the operation that was corrupted earlier (the same operands have to be applied again). Even if the error was temporary, the scheme will *never* switch back to the left SCA. The reason for this is as follows: if the error is an intermittent one, occurring frequently, the pipeline would have to build in repetition cycles continuously, compromising the throughput severely. Also, here a fail signal is present. If the right SCA reports an error as well, the fail signal is asserted. The right SCA remains always active, so in the case the error was temporary, we might be able to continue with the operations.

### 6.17.4 The PC-DAU-RAS

The **PC-DAU-RAS** is a version of the DAU-RAS (the DAU-RAS, or power-optimized duplicated AU was explained in chapter 5, sections 5.4 and 5.5.2), where both AUs are equipped with *parity prediction*. An error-correction scheme is implemented which transfers the computation to the other AU when an error is detected. Again we refer to the DAU in figure 5.3. All adder inputs are tri-state buffered, except for the add/sub signal. The reasoning was that the addition of extra tri-state buffers (which are expensive

devices as we established earlier), would do more harm than good. Without a tri-state buffer, some switching activity in the first full-adder of a disabled adder is then inevitable, but manual (pen-and-paper) calculations showed that the power consumption of two tri-state buffers (one pass, one closed) would exceed this switching power. Ultimately, we decided to implement the tri-states anyway and found that our initial assumption was wrong. A small reduction in power could be achieved, obviously at the expense of some additional area. However, the power and area differences are small and because of limited time we decided not to re-synthesize the all the results (for different adder widths, technologies, clock frequencies, etc.) in chapter 5. We assume that the parity bits of the operands are *generated at an early stage in the pipeline* (thus, not in the AU itself), since the parity bits can be utilized for several error-checking purposes of other components in the architecture as well. Therefore, the parity bit generation circuits are *not* part of the ED/EC scheme, which saves already a significant amount of power and area. The left AU is active as long as no error is detected by the parity-prediction scheme. If an error is detected, the occurrence is stored, and on the next rising clock edge the erroneous AU is disabled and the backup adder is enabled. Now, the calculations can proceed. Note that also here we require a *repetition cycle*. For the PC-DAU-RAS we both implemented the parity-prediction scheme based on the CDSA and the duplicated-carry scheme, in order to compare the two parity prediction schemes based on power consumption, area, and delay. Based on these results, the cheapest parity prediction scheme is chosen for implementation in the ScAU, as explained in section 6.16.4. The internal circuitry of the 16-bit parity-checked AUs can be compared with the 8-bit parity-checked AU-segments (CDSA scheme) depicted in figure 6.11. When the duplicated parity scheme is implemented, the implementation simply changes according to figure 6.5. Note that, in the latter case, *regular* full-adder cells are utilized.

### 6.17.5 The PC-ScAU

The **PC-ScAU** is a version of the ScAU (the implementation of the ScAU was explained in chapter 5, section 5.3 and 5.5.2), where both AU segments are equipped with *parity prediction*. An error-correction scheme is implemented which disables the damaged AU segment if an error is detected, downscales the structure, and proceeds with the computation with the remaining AU segment. Since in normal operation both adder segments are active, there is no need for tri-state buffering the add/sub signal. Adding tri-state buffers here would decrease the power consumption in downscaled mode, but slightly increase it in normal mode. That is certainly not what we want as it would make the the difference in power between the ScAU and the DAU-RAS slightly larger than presented in the results of chapter 5, in in favor of the DAU-RAS. We assume that the parity bits are *generated at an early stage in the pipeline*, like we did for the PC-DAU-RAS. In section 6.13 we already mentioned the problem with the parity bits in the PC-ScAU. Parity bits generated for 16-bit operands are useless when we decide to check the upper and lower byte *independently*. So, either we must *generate* two check digits per operand (one of the lower order byte and one of the higher order byte: $P_{low}$ and $P_{high}$), or we must *derive* the $P_{low}$ and $P_{high}$ from the 16-bit check digit $P$. For the first option (the so-called **split-parity**), we generate $P_{low}$ and $P_{high}$ early in the pipeline,

which means that each pipeline register must have room for two parity bits instead of one. Parity checkers in the architecture which need the parity of the full 16-bit operand $P$, can easily compute it by exclusive-oring $P_{low}$ and $P_{high}$. Obviously, this results in additional overhead. The second option, where we derive $P_{low}$ and $P_{high}$ from $P$, would require an eight-bit parity generator to generate $P_{low}$. Then, $P_{high}$ can be computed by exclusive-oring $P$ and $P_{low}$. Also here we have additional overhead, however we save three (two at the input, one at the output) flip-flops in the design (since the pipeline registers are incorporated in the PC-ScAU design). Implementation and synthesis of both options showed that option two is slightly more expensive in terms of power (0.8%) and area (3.7%), *from the arithmetic unit's point of view*. Therefore, we decided to implement option one. However, from the *point of view of the overall pipeline* option 1 is more expensive. Since the SiMS architecture is very minimalistic and has only a small number of pipeline stages, the overhead due to the extra flip-flop in the pipeline registers will most likely be marginally. We require, however, some future research in order to determine the exact impact on the costs of the pipeline.

The schematics of the design of the PC-ScAU are depicted in figure 6.9. All components related to ED/EC are displayed in green. The split-parity bits at the inputs are fed to multiplexers. In case a segment is shut down, the parity bits can be rerouted this way. $P_{r,h}$ and $P_{r,l}$ represent the split-parity bits of the result. The FSM of the **EC-controller** (error-correcting controller) is depicted in figure 6.10. The main task of the EC-controller is to reconfigure (downscale) the PC-ScAU when it receives an error notification from one of the parity checkers and (2) signal the **cycle-controller** (which keeps track of the current cycle) on *if and how* the ScAU is reconfigured. Both the EC-controller and the cycle-controller provide all the control signals for the various multiplexers, tri-state buffers, etc. in the design. The FSM of the cycle-controller is identical to the one depicted in chapter 5 (figure 5.4). The internal circuitry of the 8-bit parity-checked segments is depicted in figure 6.11. $P_x$ and $P_y$ represent the input parity bits, $P_z$ the output parity. Note that the checker is essentially no different than the parity generators (generating $P_z$ and $P_c$): those are all ODD-circuits (XOR-trees).

### 6.17.6 Synthesis results

After crafting all designs in VHDL, they are synthesized and analyzed for area, power, and delay at a clock frequency of 20MHz, which turned out to be the optimal frequency for the ScAU in chapter 5 and a realistic system frequency for the SiMS architecture. The results can be observed in table 6.1 and figures 6.12, 6.13, and 6.14. Also the power-area, area-delay, and power-delay-area products are depicted, in figure 6.15. The **PC'-DAU-RAS** represents the parity-checked duplicated adder based on *duplicated carries*. The PC-DAU-RAS represents the same structure, but now checked by parity prediction based on *the CDSA*. The PC-ScAU is also checked by parity prediction based on the CDSA. The abbreviations 'nm' and 'dm' represent the characteristics of the PC-ScAU in normal and downscaled mode, respectively.

It is immediately clear that the QMR and TMR schemes perform far worse than the others when we consider the costs. These schemes are, however, the fastest. The QMR-RAS, optimized for power, does indeed result in significant power savings (22%).

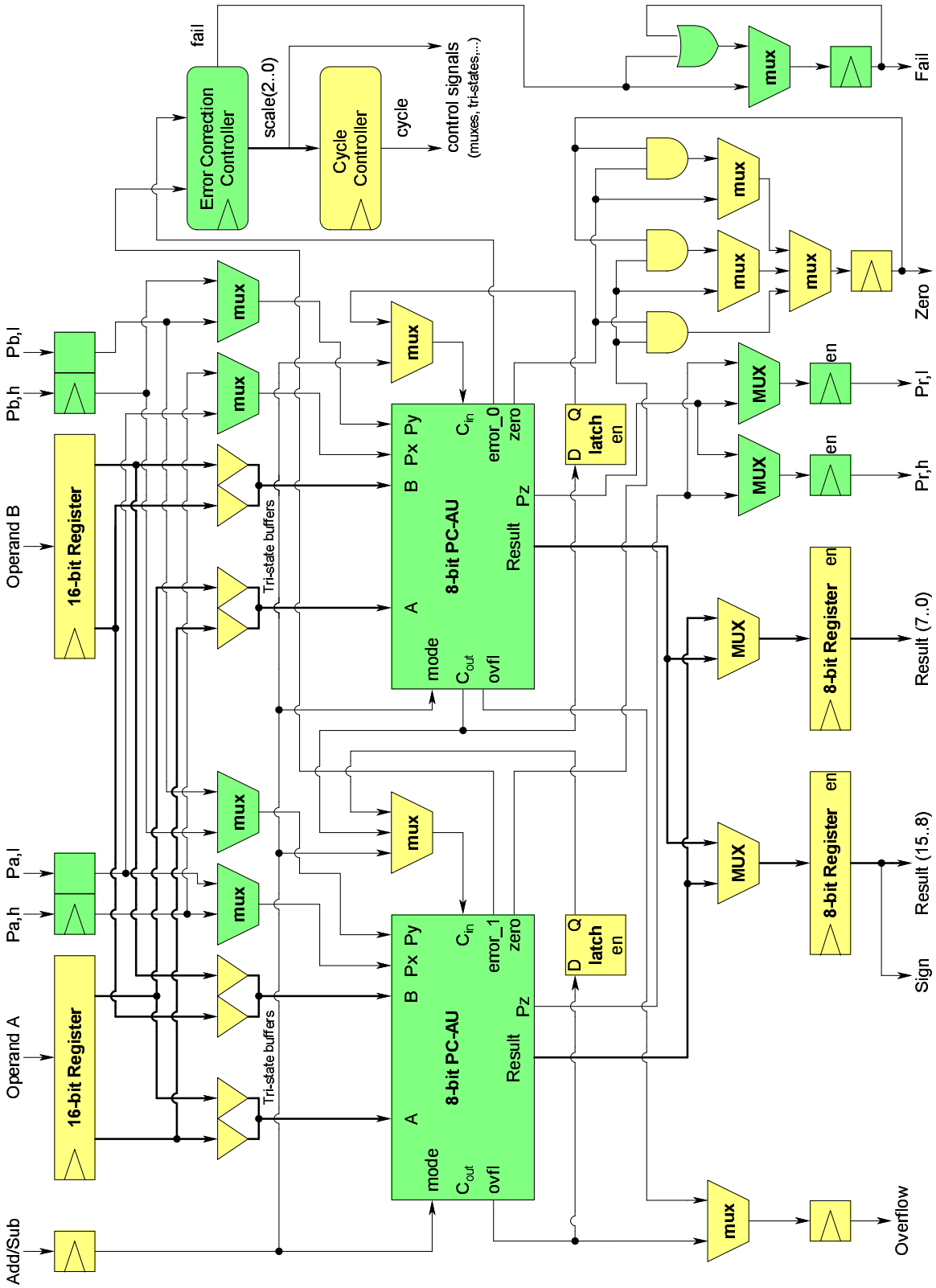


Figure 6.9: The PC-ScAU

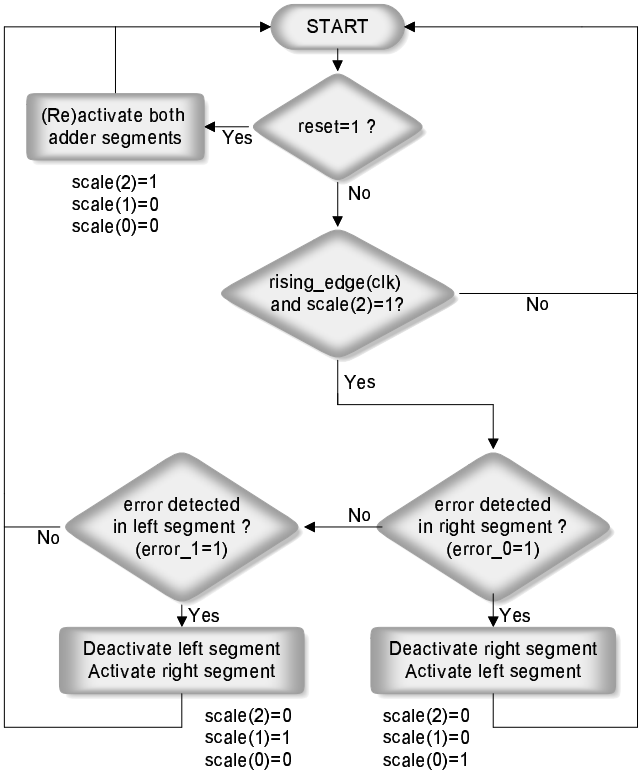


Figure 6.10: FSM of the EC-controller

| ED/EC scheme | Area | Total Power | Dynamic Power | Leakage Power | Delay |
|---|---|---|---|---|---|
| | [units] | [$\mu$W] | [$\mu$W] | [$\mu$W] | [ns] |
| TMR | 3380 | 51.48 | 47.88 | 3.60 | 3.67 |
| QMR | 3842 | 58.76 | 54.38 | 4.38 | 3.34 |
| QMR-RAS | 4308 | 45.88 | 40.97 | 4.91 | 3.73 |
| PC'-DAU-RAS | 3615 | 40.02 | 36.10 | 3.92 | 4.26 |
| PC-DAU-RAS | 3435 | 37.88 | 34.18 | 3.70 | 4.75 |
| PC-ScAU (nm) | 2884 | 40.64 | 34.47 | 3.17 | 4.18 |
| PC-ScAU (dm) | 2884 | 29.22 | 26.00 | 3.22 | <4.18 |

Table 6.1: Synthesis results of AU with different ED/EC schemes

This makes the scheme, in terms of power, even more attractive than the TMR. The area cost is, however, quite dramatic (12.1% higher than that of the standard QMR, which already had a very high area cost). But in an environment where there is a need for detecting multiple errors at a relatively low-power cost, the QMR-RAS is a suitable candidate, as long as the high area cost is not a problem. In architectures where we currently focus on, with both power and area budgets being very limited, the QMR-RAS is obviously not an option. Thus, we prove here that we have to resort to error-detection codes when area budgets are very tight, since all all ED-schemes based on full hardware replication result in high area costs.

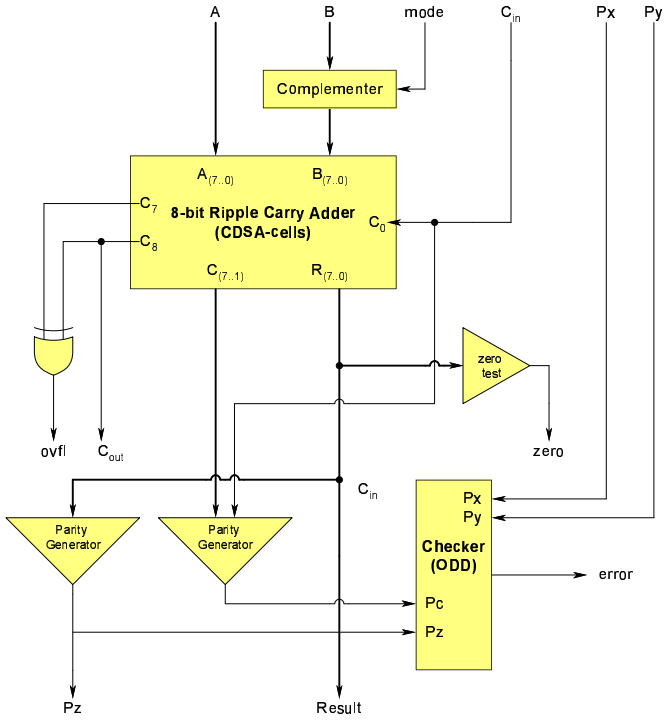


Figure 6.11: The parity-checked AU (CDSA scheme)

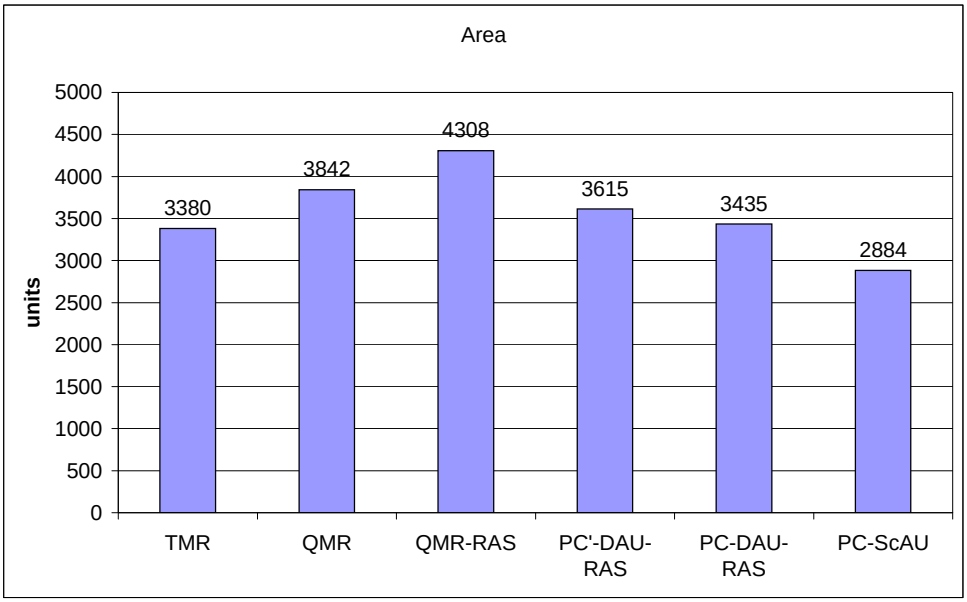


Figure 6.12: The area requirements of the AU with different ED/EC schemes

Synthesis provides us the information we expected: the parity-prediction scheme based on the CDSA is cheaper than that based on duplicated carries. The CDSA scheme requires 5.0% less area and 5.3% less power. This is the reason why we have

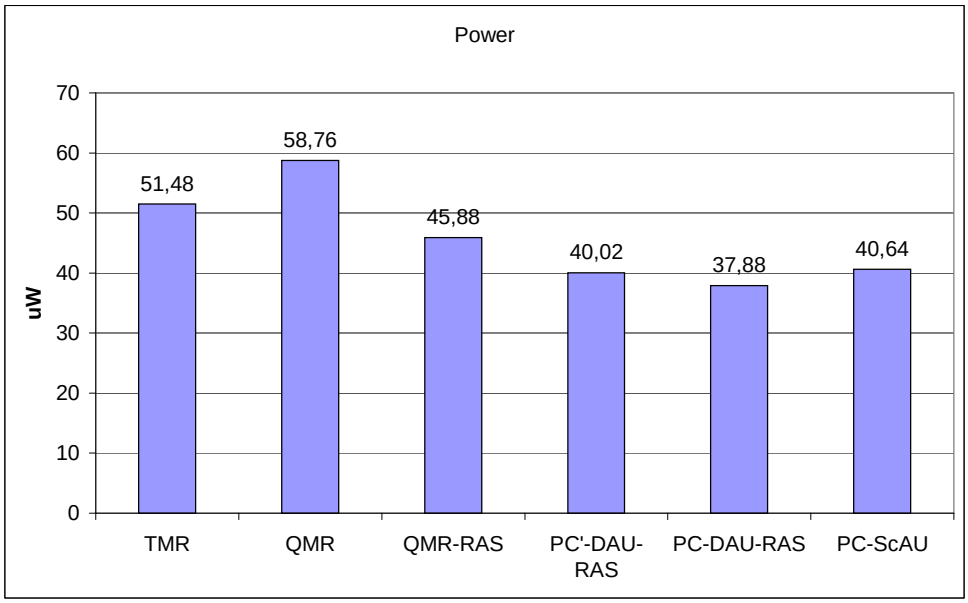


Figure 6.13: The power consumption of the AU with different ED/EC schemes

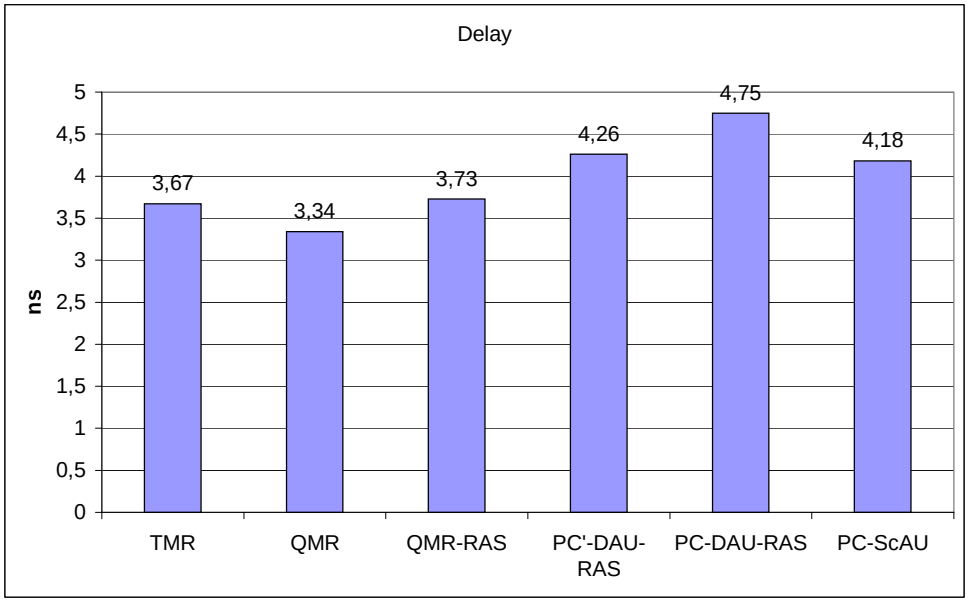


Figure 6.14: The delay of the AU with different ED/EC schemes

implemented the *CDSA scheme* in the ScAU. The PC-DAU-RAS (the implementation with the CDSA scheme) does, however, have a longer delay than the implementation with the duplicated-carry scheme (PC'-DAU-RAS). This is because the carry-dependent full-adders are more complex and, consequently, slightly slower.

What is remarkable, is the delay of the PC-ScAU. It is *lower* than that of the PC-DAU-RAS, while in chapter 5, this was (*without* ED/EC) exactly the other way

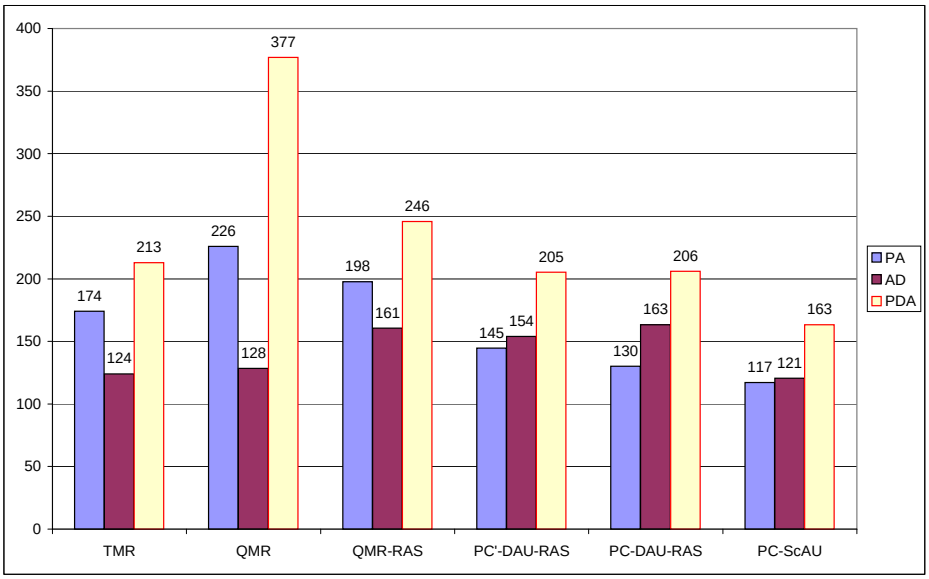


Figure 6.15: The PA, AD, and PDA product of the AU with different ED/EC schemes

around. There are two different causes that explain this. First, *the path through the ED logic is shorter in the PC-ScAU, than it is in the PC-DAU-RAS*. The critical path goes through the first 8-bit adder segment, then —via the carry-out, connecting logic, and carry-in— through the 8-bit second adder segment, and finally through the 8-bit parity tree of the sum of the second adder segment. In the PC-DAU-RAS, the critical path goes through the 16-bit adder and then through the 16-bit parity tree of the sum. Obviously, an 8-bit parity tree is faster than a 16-bit version. Second, *the critical path of the PC-DAU-RAS was altered by adding the tri-state buffers as input-gating for the add/sub signal*. The adder and complementer cannot perform any useful computations until the add/sub signal has arrived. Since there is a tri-state buffer present in this path that is controlled by the EC-controller, this path is somewhat *lengthened*. Thus to summarize, the critical path of the PC-ScAU was not lengthened to the same extend as the PC-DAU-RAS by the ED logic, and the critical path of the PC-DAU-RAS has become longer due to the input gating of the add/sub signal.

Unfortunately, we have to establish that the PC-ScAU performs *slightly worse* in terms of power than the PC-DAU-RAS. In chapter 5 we concluded that the power consumption of the ScAU and DAU (without ED/EC) was more or less the same at a clock frequency of 20 MHz. The reason why the PC-ScAU has a higher power consumption than the PC-DAU-RAS is because of the additional multiplexers (multiplexing the split-parity bits), the additional split-parity bit registers, and the added EC-controller in the PC-ScAU. The required area of the PC-ScAU is *remarkably low*. Of all ED/EC implementations it requires, by far, the *least* amount of area. So, even though the PC-ScAU performs slightly worse in terms of power than the PC-DAU-RAS, the significant area savings makes the PC-ScAU the design with the *lowest PDA and PA product*, as depicted in 6.15. It also has the *lowest AD product* of all AUs with EC/ED. We present a qualitative overview of the fault-tolerant AUs we have implemented and

studied in this chapter in table 6.2[1].

| ED/EC scheme | Area | Latency/ operation | FT | Momentary Power | Energy/ operation |
|---|---|---|---|---|---|
| TMR | □/+ | + | ++ | – | – |
| QMR | – | ++ | ++ | – – | – – |
| QMR-RAS | – – | + | ++ | □ | □ |
| PC-DAU-RAS | □/+ | – | □ | ++ | ++ |
| PC-ScAU (nm) | ++ | □ | –/□ | + | + |
| PC-ScAU (dm) | ++ | – – | –/□ | ++ | – – |

Table 6.2: Qualitative comparison between different ED/EC schemes

In figure 6.15, all objectives (power, area, and delay) have an equal weight. In chapter 1, we established that power savings are of greater significance than area savings. To which extend, we do not know yet. Therefore we assigned *various weights to the power consumption* in the PA product, as can be observed in table 6.3. For assigning additional weight to power, we utilize exponentiation. We explore the effect on the PA product if we assign additional weight, and try to find the **twist point** where the PC-ScAU is no longer advantageous over the PC-DAU-RAS. The table shows that the PC-ScAU is beneficial according to the PA metric if we utilize an exponent up to 2 (lowest PA products are displayed in boldface in the table). In table 6.4, the PDA product can be observed with different weights assigned to power *and* delay. Since we care much more about power and area than we do about delay, it is fair to assign *smaller weights to delay*. Again we seek the twist point, until where the PC-ScAU is beneficial. Only when power is raised to the fourth power, and delay is square rooted, the ScAU is no longer advantageous over the PC-DAU-RAS in terms of the PDA-product.

| AU | PA x$10^5$ | $P^2A$ x$10^6$ | $P^3A$ x$10^8$ | $P^4A$ x$10^{10}$ |
|---|---|---|---|---|
| TMR | 1.74 | 8.96 | 4.61 | 2.37 |
| QMR | 2.26 | 13.27 | 7.79 | 4.58 |
| QMR-RAS | 1.98 | 9.07 | 4.16 | 1.91 |
| PC'-DAU-RAS | 1.45 | 5.79 | 2.32 | 0.93 |
| PC-DAU-RAS | 1.30 | 4.93 | **1.87** | **0.71** |
| PC-ScAU | **1.17** | **4.76** | 1.94 | 0.79 |

Table 6.3: Custom metrics based on the PA product with different weights assigned to power

### 6.17.7 Scalable structure destroys fault-secureness property

As mentioned before, the parity scheme that is utilized for the ScAU is fault-secure for single errors. This means that the two separate 8-bit adders inside the ScAU are

[1]Meaning of the indicators: ++ represents the best, + a good, □ a moderate, – a relatively poor, and – – the least attractive implementation according to the criteria involved.

| **AU** | **PDA** $\mathbf{x10^5}$ | $\mathbf{P^2DA}$ $\mathbf{x10^7}$ | $\mathbf{P^4DA}$ $\mathbf{x10^{10}}$ | $\mathbf{PD^{\frac{1}{2}}A}$ $\mathbf{x10^5}$ | $\mathbf{P^2D^{\frac{1}{2}}A}$ $\mathbf{x10^7}$ | $\mathbf{P^4D^{\frac{1}{2}}A}$ $\mathbf{x10^{10}}$ |
|---|---|---|---|---|---|---|
| TMR | 6.39 | 3.29 | 8.71 | 3.33 | 1.72 | 4.55 |
| QMR | 7.54 | 4.43 | 15.13 | 4.13 | 2.42 | 8.37 |
| QMR-RAS | 7.37 | 3.38 | 7.12 | 3.82 | 1.75 | 3.69 |
| PC'-DAU-RAS | 6.16 | 2.47 | 3.95 | 2.99 | 1.19 | 1.91 |
| PC-DAU-RAS | 6.18 | 2.34 | 3.36 | 2.84 | 1.07 | **1.54** |
| PC-ScAU | **4.90** | **1.99** | **3.29** | **2.40** | **0.97** | 1.61 |

Table 6.4: Custom metrics based on the PDA product with different weights assigned to power and delay

fault-secure for single errors. Nevertheless, the two 8-bit adders connected together by the required logic for the scalability feature, which forms the 16-bit scalable adder, is *not* fault-secure for single errors as we found out during the design of the PC-ScAU. This is caused by a small number of so-called '*weak spots*' in the error detection: logic parts along the data path that are *not* checked for errors. The parity-prediction scheme assumes that the carry input of the adders is correct. However, the carry input (which is the add/subtract signal) is not connected directly to the carry inputs of both 8-bit adders. Because of the scalable nature of the scheme, the carry-inputs have some multiplexers and latches attached to these inputs. When, for example, a stuck-at fault occurs at the output of such a multiplexer or latch, the carry input is erroneous and also the computation is erroneous, which is not detected by the ED scheme. There are two methods to overcome this problem:

1. Since the self-testing property of the 16-bit adder is not compromised by the additional logic, an error is by definition still detectable. In the SiMS micro-architecture there is sufficient time to *self-test* the arithmetic unit on a regular basis with a predetermined set of test vectors to test the entire ScAU for errors.

2. The logic components that are not covered by the parity-prediction scheme can be *duplicated* and the outputs can, e.g. , be *compared* with a simple XOR gate. The outputs of the XOR gates can be considered as *additional error signals*, which should be *logically added* to the existing error signal. Obviously, this comes at a (small) price. It is estimated that area will increase by 60 units (2.1%) and power by 0.5 $\mu$W (1.2%), but accurate estimations are difficult to make. By implementing this method, we are able to achieve the fault-secure property for single errors of the 16-bit ScAU.

Which option is viable ultimately depends on the demands of the application and the exact power/area budgets. If an incorrect result is intolerable at all times, option two is the only legitimate option (if higher cost is allowed), since option one might pass incorrect results in between self-tests.

We decided to opt for method 1. The reason for this is that the unchecked logic that forms the 'weak spots' in the ED is only a very small fraction of the total amount

of logic. Statistically considered, the chance that an error occurs in these parts of the hardware is very small. Further, an additional increase in power consumption of the PC-ScAU would be highly undesirable: an additional 0.5 $\mu$W increases the gap in power consumption between the PC-DAU-RAS and the PC-ScAU even further.

### 6.17.8 Some notes regarding the power consumption of the PC-ScAU

As we have shown before, the PC-ScAU is capable of significant area (and delay) savings when compared to the PC-DAU-RAS. Even though the difference is very small, the higher power consumption of the PC-ScAU, however, is a disadvantage in comparison with the PC-DAU-RAS. There are, however, situations (some of them targeting *other architectures than SiMS*), where this disadvantage can possibly be eliminated, for example:

1. *When the adder is wider*, e.g. 32 bits. While the adder itself scales linearly with the word-size, the largest part of the control logic of the PC-ScAU (overhead) remains constant. Thus, the overhead of the PC-ScAU becomes a less significant portion of the total (as discussed in chapter 5).

2. *When the adder needs to operate at high frequencies/when a high throughput is desired*. Then, we would have to pick a faster adder (e.g. the CLA). Since fast adders are larger and consume more power, the overhead of the ScAU becomes a less significant portion of the total (as discussed in chapter 5). Note, that with modern technologies, we are talking about hundreds of MHz, more than a tenfold of the clock frequency we require in the SiMS architecture.

3. The power numbers we obtained of the PC-ScAU and the PC-DAU-RAS are close. It is very well possible that, even though the duplicated adder is the winner in terms of power in the technology we utilized (UMC 90nm SP), this *does not hold true for another technology*. One should note that the size of the PC-ScAU is considerably smaller than that of the PC-DAU-RAS. In 90nm technology, static power is only a relatively small portion of the total power consumption. However, if we decide to employ the newest technologies, like 45nm CMOS, it is possible that the impact of the static power becomes so large, that the PC-ScAU utilizes the least amount of power after all, simply because it is a smaller design and, therefore, has a smaller leakage power component.

4. Again, because the post-synthesis power numbers we obtained of the PC-ScAU and the PC-DAU-RAS are so close, *there might be some deviation in the final result after place and routing*. Synopsys PrimeTime-PX can give good estimates on power consumption, but ultimately, we will only know the real results after layout (utilizing e.g. Cadence Encounter). It is possible that the differences in power consumption become even smaller, with a small chance that in the end the ScAU might be the winner in power consumption after all. However, the difference in power could as well grow larger. There is no way to predict this. What we do know is that we do not expect large differences between pre-layout and post-layout results (based on the study in chapter 5).

### 6.17.9 Adding error detection for zero and overflow signals

As an optional part in this thesis work, we decided to investigate what the impact on the area and power results is, *if we implement ED/EC for the zero and overflow computation inside the arithmetic unit as well*. Error detection of zero/overflow signals in AUs is *crucial* according to Townsend et al. [68]. For example, the zero and overflow signals are utilized by the **Compare Unit** in the ALU, which enables so-called '**test-and-set**' instructions in order to compare operands $A$ and $B$, such as SEQ (set if equal), SNE (set if not equal), SGT (set if $A$ greater than $B$), SLT (set if $A$ less than $B$), etc. We have only studied the impact of error checking in zero/overflow signals for those ED/EC schemes which are the most interesting for implementation in the SiMS micro-architecture in terms of costs.

First, we analyze the PC'-DAU-RAS (duplicated-carry scheme). If we have another look at figure 6.5, we can see the leftmost duplicated-carry circuit is not utilized by the checker. Thus, the synthesizer will optimize this block, which means it will normally *remove* the block. However, if we want to check the overflow computation, this duplicated-carry signal is vital. The overflow signal $ovfl = c_n \oplus c_{n-1}$, where $c_n$ is the carry-out and $c_{n-1}$ is the last internal carry. Before we can check the overflow signal, we must be sure that the inputs are correct. The last internal carry is always correct: this is already checked by the parity-prediction scheme. The carry-out signal can easily be checked by comparing $c_n$, with $c_{n,dupl}$, with a simple XOR-gate. Then, we only have to add this error signal to the existing error signal by an OR-gate. *It is inefficient to actually check the overflow signal itself*, since it is computed by a single XOR-gate. So, to be correct, we check the carry-out signal for errors, not the overflow signal.

In the PC-DAU-RAS and PC-ScAU (CDSA scheme), the situation is different. We do *not* have duplicated-carry circuits, so we cannot compare the carry output with anything in order to see if the signal is correct. Fortunately, we do not have to. Since the carry is dependent on the sum, and since the parity-prediction scheme checks the last sum bit, then the carry-out bit is checked indirectly as well. In other words, *we do not need to check anything here*: $c_n$ and $c_{n-1}$ are both checked by the parity-prediction scheme. The zero signal can be checked for errors in a straightforward way in all three designs: the zero-generator is duplicated and the outputs are compared. The additional error signal is added to the existing error signal by an OR-gate.

In table 6.5 the hardware overhead and power/area overhead as a result of ED in zero/overflow signals can be observed. Note that two components of each type are required, since the PC-DAU-RAS/PC'-DAU-RAS and the PC-ScAU have two separate AUs or AU segments including error detection. The area and power costs of the PC'-DAU-RAS are higher than that of the PC-DAU-RAS, which is obvious since we check the carry-out signal for errors in the PC'-DAU-RAS and in the PC-DAU-RAS we do not. This is another reason why the CDSA scheme is superior to the duplicated-carry scheme. The area overhead of the PC-ScAU is significantly lower than the PC-DAU-RAS. This is because the ScAU has two 8-bit AU segments, while the PC-DAU-RAS has two 16-bit AUs. The power overhead is however significantly higher because both segments in the PC-ScAU are always active (in normal mode), while the PC-DAU-RAS has only one AU active at a time. If only one AU is active, then only one ED-scheme is also active,

| ED/EC scheme | PC'-DAU-RAS | PC-DAU-RAS | PC-ScAU |
|---|---|---|---|
| Required hardware zero checking | 2 x XOR2<br>2 x 16-bit zero-generator<br>2 x OR2 | 2 x XOR2<br>2 x OR2<br>2 x 16-bit zero-generator | 2 x XOR2<br>2 x OR2<br>2 x 8-bit zero-generator |
| Required hardware overflow checking | 2 x duplicated carry<br>2 x XOR2<br>2 x OR2 | | |
| **Total area cost [units]** | **138** | **108** | **61** |
| **Total power cost [$\mu$W]** | **0.72** | **0.47** | **0.73** |

Table 6.5: Area and power overheads when error checking zero/overflow signals

| ED/EC scheme | Area [units] | Total Power [$\mu$W] | Delay [ns] |
|---|---|---|---|
| PC'-DAU-RAS | 3753 | 40.74 | 4.35 |
| PC-DAU-RAS | 3543 | 38.35 | 4.79 |
| PC-ScAU (nm) | 2945 | 41.37 | 4.23 |
| PC-ScAU (dm) | 2945 | 29.56 | <4.23 |

Table 6.6: Synthesis results of AU with different ED/EC schemes, including ED of zero/overflow signal

and the other one is disabled. Table 6.6 is analogous to table 6.1, only here the results represent the version of the PC-ScAU including the ED of the zero/overflow signals.

## 6.18 Conclusions

In this chapter we focused on fault-tolerant design. Sources and different types of errors are discussed, as well as the most important terminology that is employed in fault-tolerant design. The concept of **Boolean difference** is introduced, which is an important tool to investigate and prove if, and under which circumstances, certain errors at the input of a system will result in erroneous outputs. Errors in adders appear to be different in nature than in any other type of logic structure, since single faults can result in arbitrarily long bursts of sum and carry errors. Several error-detection schemes are presented, such as **hardware duplication**, various **error-detection codes**, and a scheme based on **time redundancy** (RESO) instead of hardware redundancy. By definition, adders are TSC circuits. That does not hold true for prediction circuits and checkers. It requires special attention to design these components to meet the TSC property. But when they do, the reliability of the entire design (adder plus ED) is significantly enhanced.

*Error detection* is important, but does not make a circuit fault-tolerant on its own. Therefore, we need to be able to *recover* from errors as well. There are two types of *error correction*: **masking** and **online repair/reconfiguration**. The QMR and TMR

are examples of error masking, since an erroneous adder is not disabled or replaced. The DAU-RAS is an example of online repair. The erroneous adder is disabled and another adder is activated. In this thesis we have not explored **error-correcting codes**. In the general case, ECCs are (disproportionally) expensive, especially for tiny architectures as the ones we are interested in. However, we are well-aware that there may be certain error-correcting schemes that are cheaper in terms of power and area than some of our more expensive reference designs. We consider this as a direction for future work.

We have discussed what the **minimal fault coverage** of the ScAU should be in the SiMS micro-architecture. Even though we desire a very high reliability, we are forced to limit the fault coverage because of the low-power and low-area budgets. The decision has been made to choose an ED scheme which is **fault-secure for single errors**. We believe this is justifiable, if and only if, the ScAU (in fact, the entire ALU) is self-tested on a regular basis by a predetermined set of test vectors which ensures a higher fault coverage than single errors alone. The regular self-tests are to be planned by the designers of the SiMS architecture anyway, since the ALU will not be utilized continuously. The ALU can be employed for self-tests whenever it is idle (i.e. no meaningful computations are required from it).

Hardware duplication leads to excellent FT results, but incurs high costs. The area and power costs of RESO are low, but the energy consumption per instruction is quite dramatic. Therefore, we resort to ED codes. Berger check prediction has a high fault coverage, is designed for covering multiple errors and indications in the literature suggest very high costs. Bose-Lin check prediction is also designed for covering multiple errors but the error coverage can be adjusted by choosing the number of check bits. Even though Bose-Lin check prediction will have lower costs than Berger check prediction, indications can be found that also this ED scheme is not cheap in terms of power and area. This leaves only parity prediction and residue checking behind. After a thorough study we learned that residue checking is not efficient for RCAs with a modulus of 3.

The QMR and TMR schemes have high costs and are not suitable for implementation in the ScAU. However, they are implemented with regular AUs, in order to present a more *general comparison* which might be useful to a broad audience. Also, the QMR has been *optimized for power* by having only *one* SCA active at a time (QMR-RAS). The TMR is implemented with a **word-voter**, which enables the detection of multiple AUs being in error. Further, the DAU-RAS has been implemented with two *different types* of parity prediction, in order to find out which one is the cheapest in terms of power consumption and area. For the implementation of a parity-prediction scheme in the ScAU, we had to overcome an obstacle: since the adder segments have their own separate ED, the parity bits based on the 16-bit operands can not be utilized in a straightforward manner. The best way to cope with this is to provide the operands with two parity bits: one per byte, or so-called **split-parity**. All previously mentioned designs are synthesized and analyzed for power consumption, area, and delay, at a clock frequency of 20MHz.

The QMR has the highest power consumption, followed by the TMR. The QMR-RAS does display *significant power savings* in comparison with the conventional QMR (-22%), resulting in a power consumption that is even much lower than that of the TMR, but the area requirements are dramatic. However, the QMR-RAS is suitable for a low-power/power-aware architecture which demands a very high error-coverage and has

no tight area budget.

We have shown that the PC-DAU-RAS equipped with a parity-prediction scheme based on the carry-dependent sum adder is cheaper in terms of area (-5.0%) and power (-5.3%) with respect to the scheme based on duplicated carries (PC'-DAU-RAS). The delay of the latter is, however, shorter (-10.3%). The PC-ScAU is —due to these results— equipped with the parity-prediction scheme *based on the carry-dependent sum adder*. Unfortunately, the PC-ScAU appears to have a *slightly higher* power consumption (+7.3%) than the PC-DAU-RAS (in contrast with the results in chapter 5, which were about equal at 20 MHz). This is caused by the more complex ED/EC logic required in the scalable structure. The *area savings* of the PC-ScAU are *significant*: **-16%** compared to the PC-DAU-RAS. Also the delay is *shorter* (**-12%**). The PA product of the PC-ScAU is *lower* than that of the PC-DAU-RAS (**-9.9%**), and the PDA product is *much lower* (**-20.7%**). We believe that the PC-ScAU is a *very interesting alternative* for the PC-DAU-RAS. We have presented a qualitative overview of the fault-tolerant AUs we have implemented and studied in this chapter in table 6.2.

Ultimately, whether the PC-ScAU is suitable for the SiMS micro-architecture is not just a trade-off between power and area, but obviously depends on the absolute maximum power budget as well. The slightly higher power consumption of the PC-ScAU is the only disadvantage in comparison with the PC-DAU-RAS. There are, however, situations (some of them targeting other architectures than SiMS), where this disadvantage can possibly be eliminated, for example: when the adder is *wider* (e.g. 32 bits), when we would require a *faster* adder (e.g. the CLA), or both. Also, since the power numbers we obtained of the PC-ScAU and the PC-DAU-RAS are so close, it is very well possible that, even though the duplicated adder is the winner in terms of power in the technology we used (UMC 90nm SP), this does not hold true for *another technology*. Finally, there might be some *deviation* in the final result *after* layout. It is possible that the differences in power consumption become even smaller, with a small chance that in the end the ScAU might be the winner in power consumption after all. However, the difference in power could as well grow larger.

The power consumption of the PC-ScAU is significantly lower in downscaled mode than in normal mode (41.37 vs. 29.56 $\mu$W, which is a reduction of 28.5%). Obviously, the power consumption does *not* drop by 50%, because the scalable adder requires a significant amount of hardware overhead to make the two-cycle operations possible. In normal mode, most of this logic does not consume dynamic power, because there is no switching activity. In downscaled mode, these components become active. This means that the energy per instruction increases by 42.9% in downscaled mode (and of course, latency doubles). Essentially this is the price we pay for the smaller size of the PC-ScAU in comparison with the PC-DAU-RAS. This again emphasizes the *essential need* for a *very small-sized* arithmetic unit in a certain architecture, to consider the PC-ScAU as a viable option.

# 7 Conclusions and Future Work

## 7.1 Conclusions

This thesis work presented a hands-on study of the field of low-power design, power-aware design, fault-tolerant design, and their practical implementations in ASIC technology. Besides, a significant time was spent on setting up appropriate tool flow, understanding the fine details of the different tools, and improving skills in VHDL design and synthesis. We have presented a relatively short, but rather comprehensive tutorial on the required software and the setup details of a tool flow involved in ASIC design, with some basic information about all the tools involved. Based on these studies, a new arithmetic unit (AU), the **scalable arithmetic unit (ScAU)**, has been proposed and built as an attractive solution in implant design.

Prior to the design of the ScAU, we have performed an extensive study on **plain adders** and compared the results obtained by us with those found in the literature. There were several motivations for performing the adder study. One of them was to explore adder structures that are suitable for segmentation: a necessity for the scalability feature of the arithmetic unit we were about to design. Another motivation was that all previously published adder studies employ outdated technology. Even though the differences in speed and area between our study and older studies were only small, the differences in power consumption were significant. There are many variables that can influence the exact implementation of the design, and thus, the characteristics of the design. A couple of examples are the synthesis tool (synthesis algorithms may vary significantly between different synthesis tools), the technology library, and the effort and optimization levels that have been set during synthesis. We cannot provide a decisive answer explaining the large differences in power consumption, since we have no access to the actual designs employed in the literature studies. The RCA is the adder of choice for the ScAU for the investigated technology node, as long as it meets the system timing requirements. If not, the CSK is the best second choice. Both the RCA and CSK are applicable in low-power, resource-constrained systems on a chip.

Our ScAU is a newly proposed design for an AU, with the goal to increase the reliability and to save power and/or area compared to regular, non-scalable, fault-tolerant designs. The ScAU is able to downscale its calculations from single-cycle operations to double-cycle operations once an error has occurred in one of the adder segments inside the ScAU. It is in essence a design with **graceful degradation** support: the throughput is compromised, however, the precision of the calculation is preserved. We have presented the initial design of the ScAU, various optimizations, and its final design. We have shown that **input gating** of combinational logic is much more efficient by utilizing **tri-state buffers**, than by utilizing latches; at least, in the currently utilized CMOS libraries. Also, it has been shown that tri-state buffers can be employed simultaneously

as a *multiplexer network and input gates* to achieve optimal efficiency. When only multiplexing is required, gate-based multiplexers are to be preferred over tri-state-based ones, since the latter require additional area and power, if the number of inputs is small.

Unfortunately, the ScAU does not appear to be as interesting for power savings as we expected (since the ScAU requires a significant amount of additional hardware to enable the scalability feature), but is certainly capable of *significant area savings*, compared to the DAU (duplicated AU). We have also shown that an 8-bit ScAU is not as efficient as 16-bit version which performs much better: since most of the *control logic is constant* and does not depend on the word size, the impact of the control logic becomes smaller as the adder's data path width increases. The ScAU becomes also more efficient when *fast adders* are employed.

The ScAU has been implemented in *several technologies* in our attempt to find the optimal design point. We showed that the ScAU performs best in UMC 90nm and worst in TSMC 65nm technology and provided strong evidence that technology is a very important variable in the results of the entire design process. Even simple and tiny designs as our arithmetic units show considerable differences in their area and power trends when implemented in different technologies. The source of the differences lies in the total set of available cells and the different ways cells are implemented in different technology libraries. Low-leakage technology proves to *increase dynamic power* for frequencies above 10 MHz, so low-leakage technology does not seem useful for our purpose, considering the fact that the leakage power is only a small fraction of the total power consumption. In order to validate that the promising low-area characteristics of the ScAU will hold after layout, we utilized a **place-and-route** tool to verify these data. The post-synthesis results appeared to be remarkably accurate. One disadvantage of the ScAU is the *increased energy per operation* in downscaled mode. We have presented a number of possible approaches to diminish these effects. Also, we proposed a couple of **alternative applications** of the proposed ScAU, such as the precision-scalable AU and the ScAU employed for dynamic thermal management.

In respect to fault tolerance, first the **sources** of errors and the error **types** —in general, but in particular in adders— have been discussed. Errors in adders appear to be different in nature than in any other type of logic structures, since single faults can results in arbitrarily long **bursts** of errors in the sum and internal carry signals. Several **error-detection schemes** are discussed, such as full hardware duplication, error detection codes, and a scheme based on time redundancy: RESO. Hardware duplication leads to an excellent error coverage, but has high costs. RESO, on the other hand, has low area and power costs, but the increased energy per instruction is dramatic. Since we are targeting low-power/energy and low-area solutions, we had to resort to ED schemes based on ED codes. Residue checking is capable of covering single and multiple errors (depending on the residue), and parity prediction is a cheap scheme to detect single errors. Berger and Bose-Lin Check Prediction are designed for covering multiple errors, but Berger Check Prediction has very high costs. Available literature has been very scarce about the actual costs of Bose-Lin Check Prediction, but it is clear that the costs are significantly lower than Berger Check Prediction.

We had to make a decision about the minimum required fault coverage of the ED in the ScAU. Ideally, we want it to be as high as possible, but we are limited by extremely

tight power and area constraints. The decision was made to have the ScAU at least *fault-secure for single errors*. This is sufficient, provided that the ScAU is tested by a predetermined set of test vectors on a regular basis (whenever the ALU is idle), in order to cover multiple errors as well. The actual choice for the ED scheme was then narrowed down to residue checking (modulo-3 checking), and three different implementations of parity prediction: the duplicated carry, the carry-dependent sum adder(CDSA), and Nicolaidis's scheme.

Based on a thorough literature study, as well as implementation, synthesis, and analysis of two different parity-prediction schemes, we found that the CDSA scheme is the most cost-effective scheme. Modulo-3 checking is *never* efficient for RCAs, only under specific conditions can it be efficient for fast adders. The CDSA parity-prediction scheme is the ED scheme that is implemented in our ScAU (now called PC-ScAU). Implementation of a parity-prediction scheme in the ScAU was, however, not straight-forward, since the parity bits are based on 16-bit numbers, while the upper and lower 8-bit adder segments are checked for errors independently. The ED of both adder segments individually is fault-secure for single errors, but unfortunately, the 16-bit adder as a whole is not, due to a few 'weak spots' in the design. This is caused by some control logic which resides in the ripple-carry adder path. We presented two methods to overcome this problem: add additional error detection or leave the detection up to the periodical self-tests. Apart from error detection, error correction is required as well, in order to make a system fault-tolerant. We decided to employ **hardware duplication** for error correction, and not to utilize error-correcting codes because of their high costs.

The PC-ScAU has been compared to a number of other fault-tolerant arithmetic units, such as the TMR, the QMR, the QMR-RAS (optimized for low-power), and the DAU-RAS with the same ED scheme as the PC-ScAU, in order to provide a comprehensive and meaningful comparison. The QMR has the highest power consumption, followed by the TMR. The QMR-RAS we optimized for low-power consumption consumes significantly less power than both the QMR and TMR. It provides a relatively low-power and a very high fault coverage solution. The area increase is however dramatic, so this design is not applicable for resource-constrained designs. The two designs that are truly interesting for our purpose are the PC-DAU-RAS and the PC-ScAU. The PC-ScAU has a slightly higher power consumption than the PC-DAU-RAS (+7.3%), but *a significant lower area* (**-16%**) *and delay* (**-12%**). This means that the PA product of the PC-ScAU is almost **10% lower** and the PDA product almost **21% lower** than that of the PC-DAU-RAS. Considering these results, we believe the PC-ScAU is a very *interesting alternative* for the PC-DAU-RAS. Especially because area is of great importance in the SiMS architecture, the designer might want to trade some power for a significant area reduction.

The slightly higher power consumption of the PC-ScAU can be *further reduced* under certain circumstances. As mentioned previously, the efficiency of the ScAU increases when when a **wider adder** is utilized (e.g. 32 bits) and/or a **fast adder** is required. Possibly, the higher power consumption can even be avoided. In UMC 90nm technology the PC-DAU-RAS may be the winner in terms of power consumption, but that does not mean this is necessarily the case for all other technologies as well, especially considering the fact that the power values are so close together. Moreover, there may always be some

deviation between the post-synthesis and post-layout results. Again, since the results are so close, this is something to take into consideration as well. Note, however, that this argument can go both ways, the difference in power consumption can also grow larger after layout.

## 7.2 Future work

A number of directions for future research can be extracted from this thesis work. For example, we have still no clear answer whether **adiabatic CMOS logic** is really capable of achieving larger energy savings than **aggressive low-power design** in standard CMOS. If it is, it might be a very interesting logic style for the ScAU (provided that the reliability of adiabatic CMOS logic is comparable with standard CMOS logic), and possibly for the entire SiMS architecture, since we do not require high performances. The larger area requirements of adiabatic CMOS logic does not necessarily have to be a problem, since the ScAU is very small-sized. Trading area for significant energy savings may then be justifiable, considering the fact that battery lifetime is a number-one priority.

What needs a thorough investigation is the necessary *modification of the pipeline* in order to support multi-cycle operations (in cases where the ScAU works in downscaled mode). It is important to know how this can be done efficiently, what the overhead is, and what the system-wide effects will be. The so-called split-parity we require for the PC-ScAU, which means that each pipeline register must have room for two parity bits instead of one, requires some future research in order to determine the exact impact on the costs of the pipeline.

Further research into the **Bose-Lin error-detection scheme** for implementation in the ScAU is desired, since it appears to be a relatively cheap scheme for covering multiple (unidirectional) errors, but there is not much information available about this scheme in the literature. Since we only had a limited amount of time, we were not able to implement, synthesize, and analyze **Nicolaidis's parity prediction/carry checking scheme**. Since Nicolaidis does not provide any solid comparisons with other parity-prediction schemes, it is interesting to know whether this new scheme is efficient for implementation in the ScAU, since the carry checker/parity predictor meets the TSC (Totally Self Checking) property and thus enhances the reliability of the ED scheme.

An important topic for future work is exploring **error-correcting codes**. Because of limited time we were not able to delve into this. However, it would be very interesting to examine if error-correcting codes exist which have very limited costs. Considering the conditions we have set regarding the minimum error coverage of the ScAU, a simple single-error correcting scheme would suffice. Implementing an error-correcting code in the ScAU instead of an error-detecting code would have a major advantage. Single errors could be corrected by the code *without* having to downscale the ScAU (which saves valuable hardware and energy). Only when the error-correcting code is no longer able to correct the erroneous result, the ScAU can downscale and actually shut down an adder segment. This would also solve the problems with **soft errors**: the occurrence of a soft error would no longer lead to the immediate shutdown of the adder segment where the error occurred.

Regarding the **alternative applications** of the ScAU we have discussed, it should be

investigated whether a situation could occur where it would be necessary to downscale the ScAU, not only when a fault occurs, but also when a **critical temperature** is reached. By downscaling the ScAU, momentary power decreases, and heat dissipation diminishes. Although it is unknown how much heat the entire SiMS chip will generate, this is an interesting topic regardless, since the ScAU can be employed in other architectures as well, where excessive heat dissipation might be problematic. Another direction for future work is to equip the PC-ScAU with a parity prediction scheme whose predictor/checkers fully **meet the TSC goal**, to improve the self-testability of the ScAU. This will ensure that an error in the ED-hardware will be detected as well.

Voltage scaling is desired for the ScAU and for the SiMS architecture in general, but it is unknown if the supply voltage for the SiMS architecture can be reduced as aggressively as for the ScAU. If this is not the case, we might opt for **multi-Vdd**, where the ScAU operates on a lower supply voltage than its surrounding environment. This would require **level shifters** which add extra power and area costs. It is unknown if this approach is cost-effective in this particular case. It would be interesting to find out whether this is the case or not.

Finally, the arithmetic unit is, of course, not the only unit in the ALU. There is also the logical unit, the shift unit, and the comparator unit. These units have to be designed under the exact same design constraints: very low-power consuming, very resource-constrained, and as reliable as possible under the previously mentioned design constraints. It would be interesting to investigate whether the gracefully degradable scaling technique from the ScAU could be employed in these other units as well. At the very least one would have to think about what to do if the ScAU is downscaled because of a fault, and a fault-free other unit (say, the logical unit) is utilized. Is the entire ALU downscaled to double-cycle operations, or do we switch back to single-cycle operations when the logical unit is utilized? All these questions need answering when the entire ALU is designed.

# Bibliography


[1] A. Sawyer-Glover and F. Shellock, "Pre-mri procedure screening: recommendations and safety considerations for biomedical implants and devices," *J Magn Reson Imaging*, vol. 12, no. 1, pp. 92–106, 2000.

[2] N. Savage. (2008) Chip for future eye implants runs on picowatts, thanks to new deep-sleep tech. [Online]. Available: spectrum.ieee.org/biomedical/bionics/chip-for-future-eye-implants-runs-on-picowatts-thanks-to-new-deepsleep-tech

[3] J. C. Sanchez, B. Mahmoudi, J. DiGiovanna, and J. C. Principe, "2009 special issue: Exploiting co-adaptation for the design of symbiotic neuroprosthetic assistants," *Neural Netw.*, vol. 22, no. 3, pp. 305–315, 2009.

[4] T. W. Berger, M. Baudry, R. D. Brinton, J. S. Liaw, V. Z. Marmarelis, A. Y. Park, B. J. Sheu, and A. R. Tanguay, "Brain-implantable biomimetic electronics as the next era in neural prosthetics," *Proceedings of the IEEE*, vol. 89, no. 7, pp. 993–1012, 2001.

[5] C. Strydis, G. N. Gaydadjiev, and S. Vassiliadis, "A new digital architecture for reliable, ultra-low-power systems," in *Proceedings ProRISC 2006*, November 2006, pp. 350–355.

[6] C. Strydis *et al.*, "Implantable microelectronic devices: A comprehensive review," Computer Engineering, TU Delft, CE-TR-2006-01, Dec. 2006.

[7] C. Strydis, "Suitable cache organizations for a novel biomedical implant processor," in *The 26th IEEE International Conference on Computer Design*, October 2008, pp. 591–598.

[8] M. Pedram, *Power Aware Design Methodologies*, J. M. Rabaey, Ed. Kluwer Academic Publishers, 2002.

[9] J. M. Rabaey and M. Pedram, *Low power design methodologies*. Kluwer Academic Publishers, 1996.

[10] 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, Apr 1992.

[11] S. M. Srinivas Devadas, "A survey of optimization techniques targeting low power vlsi circuits," in *Design Automation, 1995. DAC '95. 32nd Conference on*, 1995, pp. 242 –247.

[12] M. Ditzel, R. H. J. M. Otten, and W. A. Serdijn, *Power-Aware Architecting: for data-dominated applications*. Springer Publishing Company, Incorporated, 2007.

[13] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company, Incorporated, 2007.

[14] M. Kakumu, M. Kinugawa, and K. Hashimoto, "Choice of power-supply voltage for half-micrometer and lower submicrometer cmos devices," *Electron Devices, IEEE Transactions on*, vol. 37, no. 5, pp. 1334 –1342, may 1990.

[15] Y. Leblebici, "Design considerations for cmos digital circuits with improved hot-carrier reliability," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 7, pp. 1014 –1024, jul 1996.

[16] J. Singh, *Semiconductor Devices: Basic Principles*. Wiley, 2000.

[17] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," in *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*. New York, NY, USA: ACM, 1995, pp. 3–8.

[18] T. Kuroda and M. Hamada, "Low-power cmos digital design with dual embedded adaptive power supplies," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 4, pp. 652 –655, apr 2000.

[19] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama, "Variable supply-voltage scheme for low-power high-speed cmos digital design," *Solid-State Circuits, IEEE Journal of*, vol. 33, no. 3, pp. 454 –462, mar 1998.

[20] B. Calhoun and A. Chandrakasan, "Ultra-dynamic voltage scaling using sub-threshold operation and local voltage dithering in 90nm cmos," feb. 2005, pp. 300 –599 Vol. 1.

[21] L. Benini, G. De Micheli, and E. Macii, "Designing low-power circuits: practical recipes," *Circuits and Systems Magazine, IEEE*, vol. 1, no. 1, pp. 6 –25, first 2001.

[22] J. Kao and A. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 7, pp. 1009 –1018, jul 2000.

[23] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, 10-12 1994, pp. 8 –11.

[24] D. Markovic, V. Stojanovic, B. Nikolic, M. Horowitz, and R. Brodersen, "Methods for true energy-performance optimization," *Solid-State Circuits, IEEE Journal of*, vol. 39, no. 8, pp. 1282 – 1293, aug. 2004.

[25] E. Vittoz, "Low-power design: ways to approach the limits," in *Solid-State Circuits Conference, 1994. Digest of Technical Papers. 41st ISSCC., 1994 IEEE International*, 16-18 1994, pp. 14 –18.

[26] R. Zimmermann and W. Fichtner, "Low-power logic styles: Cmos versus pass-transistor logic," *Solid-State Circuits, IEEE Journal of*, vol. 32, no. 7, pp. 1079 –1090, jul 1997.

[27] V. Friedman and S. Liu, "Dynamic logic cmos circuits," *Solid-State Circuits, IEEE Journal of*, vol. 19, no. 2, pp. 263 – 266, apr. 1984.

[28] A. Blotti, M. Castellucci, and R. Saletti, "Designing carry look-ahead adders with an adiabatic logic standard-cell library," in *PATMOS '02: Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation.* London, UK: Springer-Verlag, 2002, pp. 118–127.

[29] E. Amirante, J. Fischer, M. Lang, A. Bargagli-Stoffi, J. Berthold, C. Heer, and D. Schmitt-Landsiedel, "An ultra low-power adiabatic adder embedded in a standard 0.13 $\mu$m cmos environment," in *Solid-State Circuits Conference, 2003. ESSCIRC '03. Proceedings of the 29th European*, 16-18 2003, pp. 599 – 602.

[30] M. Zwolinski, *Digital System Design with VHDL.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[31] S. Yalamanchili, *Introductory VHDL: From Simulation To Synthesis.* Prentice-Hall, Inc., 2004.

[32] *ModelSim LE/PE Users Manual and Reference Manual*, 2009. [Online]. Available: www.model.com

[33] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys' Design Compiler and PrimeTime.* Kluwer Academic Publishers, 1999.

[34] *Design Compiler, Power Compiler, Prime Time: user guides and reference manuals*, Sept 2008. [Online]. Available: www.synopsys.com

[35] *Cadence Encounter User Guide*, 2005. [Online]. Available: www.eecs.tufts.edu/~jqiu02/doc/encounter/

[36] A. de Graaf, H. L. Arriens, and T. van Leuken, *Digital Design Flow for EDA Tools*, Delft University of Technology, March 2009.

[37] D. Ludovici, Delft University of Technology, Delft, The Netherlands, personal communication.

[38] M. Haahr. True random number service. [Online]. Available: www.random.org

[39] S. J. Abou-samra, A. Guyot, and B. Laurent, "Spurious transitions in adder circuits: Analytical modelling and simulations," 2007.

[40] T. van Leuken, Delft University of Technology, Delft, The Netherlands, personal communication.

[41] 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, no. 10, pp. 689–702, Oct 1996.

[42] M. Vratonjic, B. R. Zeydel, and V. G. Oklobdzija, "Low- and ultra low-power arithmetic units: Design and comparison," *Computer Design, International Conference on*, vol. 0, pp. 249–252, 2005.

[43] T. Callaway and E. Swartzlander, "Optimizing arithmetic elements for signal processing," in *VLSI Signal Processing, V, 1992., [Workshop on]*, Oct 1992, pp. 91–100.

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

[45] S. Cotofana, Delft University of Technology, Delft, The Netherlands, personal communication.

[46] H. Lee, "Power-aware scalable pipelined booth multiplier," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 88, no. 11, pp. 3230–3234, 2005.

[47] O. A. Pfänder, R. Nopper, H.-J. Pfleiderer, S. Zhou, and A. Bermak, "Comparison of reconfigurable structures for flexible word-length multiplication," *Advances in Radio Science*, vol. 6, pp. 113–118, 2008.

[48] S. Kumar, P. Pujara, and A. Aggarwal, "Bit-sliced datapath for energy-efficient high performance microprocessors," in *Power-aware computer systems: 4th international workshop, PACS (revised selected papers).* Springer, 2005, pp. 30–45.

[49] A. Iyer and D. Marculescu, "Power aware microarchitecture resource scaling," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe.* IEEE Press, 2001, pp. 190–196.

[50] T.-W. Lin, M.-C. Lee, F.-J. Lin, and H. Chiueh, "A low power alu cluster design for media streaming architecture," in *48th Midwest Symposium on Circuits and Systems*, vol. 1, Aug 2005, pp. 51–54.

[51] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar, "Scheduling techniques to enable power management," in *Design Automation Conference Proceedings 1996, 33rd*, Jun 1996, pp. 349–352.

[52] R. Amirtharajah, T. Xanthopoulos, and A. Chandrakasan, "Power scalable processing using distributed arithmetic," in *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design.* New York, NY, USA: ACM, 1999, pp. 170–175.

[53] E. Kursun, G. Reinman, S. Sair, A. Shayesteh, and T. Sherwood, "Low-overhead core swapping for thermal management," in *Power-aware computer systems: 4th international workshop, PACS (revised selected papers).* Springer, 2005, pp. 46–60.

[54] S. Kaxiras, *Computer Architecture Techniques for Power-Efficiency.* Morgan and Claypool Publishers, 2008.

[55] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: pushing power management to logic synthesis/design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 10, pp. 1051 –1060, oct. 1998.

[56] (2000) Cmos logic circuits. University of California, Berkely, USA. EECS150 classnotes. [Online]. Available: www.inst.eecs.berkeley.edu/~cs150/sp00/classnotes/u10/

[57] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[58] D. Soudris, C. Piguet, and C. Goutis, *Designing CMOS circuits for low power*. Springer, 2002.

[59] R. Ramanarayanan, N. Vijaykrishnan, Y. Xie, and M. J. Irwin, "Soft errors in adder circuits," 2004.

[60] D. J. Sorin, *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.

[61] C. N. Hadjicostis, "Non-concurrent error detection and correction in fault-tolerant linear finite-state machines," *IEEE Transactions on Automatic Control*, vol. 48, pp. 2133–2140, 2002.

[62] F. F. Sellers, M.-y. Hsiao, and L. W. Bearnson, *Error detecting logic for digital computers*. McGraw-Hill New York, 1968.

[63] N. Jha, "Totally self-checking checker designs for bose-lin, bose, and blaum codes," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 10, no. 1, pp. 136 –143, jan 1991.

[64] R. O. Duarte, M. Nicolaidis, H. Bederr, and Y. Zorian, "Efficient totally self-checking shifter design," *J. Electron. Test.*, vol. 12, no. 1-2, pp. 29–39, 1998.

[65] M. Nicolaidis, "Carry checking/parity prediction adders and alus," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 1, pp. 121–128, 2003.

[66] A. P. Kakaroudas, K. Papadomanolakis, V. Kokkinos, and C. E. Goutis, "Comparative study on self-checking carry-propagate adders in terms of area, power and performance," in *PATMOS '00: Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*. London, UK: Springer-Verlag, 2000, pp. 187–194.

[67] S. Mitra and E. J. Mccluskey, "Which concurrent error detection scheme to choose?" in *Proc. International Test Conf*, 2000, pp. 985–994.

[68] W. J. Townsend, J. A. Abraham, and J. Earl E. Swartzlander, "Quadruple time redundancy adders," *Defect and Fault-Tolerance in VLSI Systems, IEEE International Symposium on*, vol. 0, p. 250, 2003.

[69] R. A. Davis, "A checking arithmetic unit," in *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I.* New York, NY, USA: ACM, 1965, pp. 705–713.

[70] G. G. Langdon and C. K. Tang, "Concurrent error detection for group look-ahead binary adders," *IBM J. Res. Dev.*, vol. 14, no. 5, pp. 563–573, 1970.

[71] J.-C. Lo, S. Thanawastien, and T. Rao, "Concurrent error detection in arithmetic and logical operations using berger codes," in *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, 6-8 1989, pp. 233 –240.

[72] D. Das and N. Touba, "Synthesis of circuits with low-cost concurrent error detection based on bose-lin codes," *VLSI Test Symposium, IEEE*, vol. 0, p. 309, 1998.

[73] S. Gorshe and B. Bose, "A self-checking alu design with efficient codes," *VLSI Test Symposium, IEEE*, vol. 0, p. 157, 1996.

[74] J.-C. Lo, S. Thanawastien, T. Rao, and M. Nicolaidis, "An sfs berger check prediction alu and its application to self-checking processor designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, no. 4, pp. 525 –540, apr 1992.

[75] J. Patel and L. Fung, "Concurrent error detection in alu's by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. C.31, pp. 589–598, 1982.

[76] M. Nicolaidis, "On-line testing for vlsi: state of the art and trends," *Integr. VLSI J.*, vol. 26, no. 1-2, pp. 197–209, 1998.

[77] M. Nicolaidis, Y. Zorian, and D. K. Pradan, *On Line-Testing for VLSI.* Kluwer Academic Publishers, 1998.

[78] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[79] W.-K. Chen, *The Electrical Engineering Handbook.* Elsevier Academic Press, 2005.

[80] M. Y. Hsiao and F. F. Sellers, "The carry-dependent sum adder," *Electronic Computers, IEEE Transactions on*, vol. EC-12, no. 3, pp. 265 –268, june 1963.

[81] S. Mitra and E. J. McCluskey, "Word voter: A new voter design for triple modular redundant systems," in *VTS '00: Proceedings of the 18th IEEE VLSI Test Symposium.* IEEE Computer Society, 2000, p. 465.

# Appendix A - Synthesis Script Files

The following script is utilized for synthesis in Synopsys DC:

```
###  Synopsys Design Compiler script ###
###      Jun 2010, D.P. Riemens      ###
###  Delft University of Technology  ###

### DEFINE VARIABLES ###

set CLOCK_PERIOD  50.0
set INPUT_DELAY  1.00
set OUTPUT_DELAY  1.00
set OUTPUT_LOAD  0.01

### CLEAN UP FOR START ###

reset_design
remove_design -all

### SET CLOCK GATING STYLE ###

set_clock_gating_style -minimum_bitwidth 8

### DESIGN ENTRY ###

read_file -format vhdl {ArithUnit.vhd, ScalingController.vhd}
read_file -format vhdl {PCA.vhd, ParityTrees.vhd, ZeroDetector.vhd}
read_file -format vhdl {RCA.vhd, CDSFA.vhd, f.vhd, Carry.vhd}
current_design arith_unit
check_design

### SETUP OPERATING CONDITIONS ###

set_operating_conditions TCCOM

### SETUP CLOCK ###

create_clock clk -period \$CLOCK_PERIOD
set_dont_touch_network clk

### INPUT DRIVES ###

set INPUTPORTS [remove_from_collection [all_inputs] clk]
set_driving_cell -lib_cell BUFX1 \$INPUTPORTS
set_drive 0 clk
```

```
### OUTPUT LOADS ###

set_load \$OUTPUT_LOAD [all_outputs]

### INPUT & OUTPUT DELAYS ###

set_input_delay \$INPUT_DELAY -clock clk \$INPUTPORTS
set_output_delay \$OUTPUT_DELAY -clock clk [all_outputs]

### AREA AND POWER CONSTRAINTS ###

set_max_area 0
set_max_dynamic_power 0

### COMPILE AND WRITE NETLIST ###

compile_ultra -gate_clock -no_autoungroup

### GENERATE VHDL NETLIST ###

change_names -rule vhdl -hierarchy
write -format vhdl -hierarchy -output arith.vhd

### SAVE MAPPED DESIGN ###

write -hierarchy -format ddc -output arith.ddc

### Generate SDF data ###

write_sdf -version 3 arith.sdf

### GENERATE VERILOG NETLIST ###
# The design is reloaded from scratch to avoid potential naming problems
# when using the netlist for placement and routing

remove_design -all
read_file -format ddc arith.ddc
change_names -rule verilog -hierarchy
write -format verilog -hierarchy -output arith.v

### SAVE SYSTEM CONSTRAINTS

write_sdc -nosplit arith.sdc

### CREATE REPORTS ###

report_area > area.rpt
report_area -hierarchy >> area.rpt

report_timing -from [all_inputs] -to [all_registers -data_pins] > timing.rpt
```

```
report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] >> timing.rpt
report_timing -from [all_registers -clock_pins] -to [all_outputs] >> timing.rpt
report_timing -from [all_inputs] -to [all_outputs] >> timing.rpt

report_design > synthesis.rpt
report_constraint -all_violators -verbose >> synthesis.rpt
report_hierarchy >> synthesis.rpt
report_reference >> synthesis.rpt
report_cell >> synthesis.rpt
report_clock_gating -structure >> synthesis.rpt
```

The following script is utilized for timing and power analysis in Synopsys PrimeTime:

```
###    Synopsys PrimeTime script     ###
###      Jun 2010, D.P. Riemens      ###
###  Delft University of Technology  ###

### DEFINE VARIABLES ###

set CLOCK_PERIOD  50.0
set INPUT_DELAY  1.00
set OUTPUT_DELAY  1.00
set OUTPUT_LOAD  0.01

### PX SETTINGS ###

set power_enable_analysis TRUE
#set power_clock_network_include_register_clock_pin_power FALSE

### DEFINE DESIGN AND READ NETLIST ###

read_vhdl ./arith.vhd
current_design arith_unit

### ANNOTATE SWITCHING ACTIVITY ###

read_saif -strip_path "testbench/dut" ./arith.saif

### SETUP OPERATING CONDITIONS ###

set_operating_conditions TCCOM

### SETUP CLOCK ###

create_clock clk -period \$CLOCK_PERIOD

### INPUT DRIVES ###

set INPUTPORTS [remove_from_collection [all_inputs] clk]
```

```
set_driving_cell -lib_cell BUFX1 \$INPUTPORTS
set_drive 0 clk

### OUTPUT LOADS ###

set_load \$OUTPUT_LOAD [all_outputs]

### INPUT & OUTPUT DELAYS ###

set_input_delay \$INPUT_DELAY -clock clk \$INPUTPORTS
set_output_delay \$OUTPUT_DELAY -clock clk [all_outputs]

### DELAY CONSTRAINTS ###

set_max_delay \$CLOCK_PERIOD -from [all_inputs] -to [all_outputs]
set_max_delay \$CLOCK_PERIOD -from [all_inputs] -to [all_registers -data_pins]
set_max_delay \$CLOCK_PERIOD -from [all_registers -clock_pins] -to [all_registers -data_pins]
set_max_delay \$CLOCK_PERIOD -from [all_registers -clock_pins] -to [all_outputs]

### CREATE REPORTS ###

check_timing
update_timing
report_timing -from [all_inputs] -to [all_registers -data_pins] > ./pt_timing.rpt
report_timing -from [all_registers -clock_pins] -to [all_registers -data_pins] >> ./pt_timing.rpt
report_timing -from [all_registers -clock_pins] -to [all_outputs] >> ./pt_timing.rpt
report_timing -from [all_inputs] -to [all_outputs] >> ./pt_timing.rpt

check_power
update_power
report_power

report_constraint -all_violators
report_power -verbose > ./px_power.rpt
report_power -hierarchy >> ./px_power.rpt
report_power -leaf -clocks ClockGen_1 >> ./px_power.rpt
report_power -groups clock_network >> ./px_power.rpt
report_power -net_power >> ./px_power.rpt

create_power_waveforms -output "vcd"
```

# Curriculum Vitae

**Danny P. Riemens** was born in Middelburg, The Netherlands on the 25th of June, 1980. After completing intermediate vocational education in Electronics, he started with the bachelor program in Electronic System Engineering at HZ University of Applied Sciences in Vlissingen. He received the Bachelor's degree (B.Eng.) in the summer of 2003, with an average grade of 8.2 (out of 10.0).

Immediately after graduating Danny continued his studies at Delft University of Technology. He started with the pre-master program in Delft in September 2003. About a year later he was accepted in the Master's degree program with a major in Computer Engineering and a minor in Biomedical Engineering. Delayed due to ongoing health reasons, he finished all courses in 2009, with an average grade of 8.5.

His research interests are computer architecture, computer arithmetic, embedded systems, ASIC hardware design and biomedical engineering. The interface between computer engineering and biomedical engineering is a topic of special interest.

*Make a plan*
*Set a goal*
*Work toward it*
*But every now and then, look around*
*Drink it in*
*'Cause, this is it*
*It might all be gone tomorrow*

From: "Grey's Anatomy" (ABC)