Low power digital signal processing

Ph.D. thesis

by

Özgün Paker, M.Sc.

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark

June, 2002

This thesis has been submitted in partial fulfillment of the conditions for acquiring the Ph.D. degree at the Technical University of Denmark. The Ph.D. study has been carried out at the Section for Computer Science and Engineering at Informatics and Mathematical Modelling, supervised by Associate Professor Jens Sparsø.

Copenhagen, June 2002

Özgün Paker

Abstract

This thesis introduces a novel approach to programmable *and* low power platform design for audio signal processing, in particular hearing aids. The proposed programmable platform is a heterogeneous multi-processor architecture consisting of small and simple instruction set processors called *mini-cores* as well as standard DSP/CPU-cores that communicate using message passing.

The work has been based on a study of the algorithm suite covering the application domain. The observation of dominant tasks for certain algorithms (FIR, IIR, correlation, etc.) that require custom computational units and special data addressing capabilities lead to the design of low power mini-cores. The algorithm suite also consisted of less demanding and/or irregular algorithms (LMS, compression) that required sub-sample rate signal processing justifying the use of a DSP/CPU-core.

The thesis also contributes to the recent trend in the development of intellectual property based design methodologies. The actual mini-core designs are parameterized in word-size, memory-size, etc. and can be instantiated according to the needs of the application at hand. They are intended as low power programmable building blocks for a standard cell synthesis based design flow leading to a system-on-chip.

Two mini-cores targeting FIR and IIR type of algorithms have been designed to evaluate the concept. Results obtained from the design of a prototype chip demonstrate a power consumption that is only 1.5 – 1.6 times larger than commercial hardwired ASICs and more than 6 – 21 times lower than current state of the art low-power DSP processors.

An orthogonal but practical contribution of this thesis is the test bench implementation. A PCI-based FPGA board has been used to equip a standard desktop PC with tester facilities. The test bench proved to be a viable alternative to conventional expensive test equipment.

Finally, the work presented in this thesis has been published at several IEEE workshops and conferences [71, 70, 72], and in the Journal of VLSI Signal Processing [73].

Preface

This work has been carried out in collaboration with the Thomas B. Thrige Center for Microinstruments and it has been supported by the Thomas B. Thrige Foundation, the Danish Research Training Council and, Oticon A/S. I am grateful for this support.

Furthermore, during the 6 years I have stayed in Denmark, I am glad to say that I was lucky to meet many people who in some way had a positive effect on my life and career.

First of all, I am very grateful to the Garring Foundation (via TEV, Turkish Education Foundation) who financed the first 2 years of my study at the Technical University of Denmark as a MSc. student. I would like to thank both foundations for that matter.

A special thanks goes to my supervisor Jens Sparsø, not only for his technical contribution and thought provoking questions during my Ph.D, but also for encouraging me to look for the "big picture" always. I am also very grateful for his help regarding non-technical matters. I could not ask for more!

The list continues with great people I got to know at Oticon A/S. I would like to thank Lars S. Nielsen and Thomas E. Christensen for all the discussions we had. A special thanks goes to Thomas Gleerup who was very helpful during his time at DTU. Especially his input on CAD tool related issues has been invaluable. Morten Elo Pedersen should also get credit for spending quite some effort while setting up the ARC core evaluation.

During the design and test phase of the prototype, I had the chance to work with brilliant students such as Niels Handbæk [38], Mogens Isager [42], and Faisal Ali [80]. Thanks to all.

I also would like to thank Sune Nielsen, my office-mate for his feedback on the thesis and his cheerful mood.

Last, but not the least, I am grateful to my family and my fiance for their unlimited support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Semiconductor technology is still following the exponential integration trend i.e., doubling of the transistor density every 1.5 to 2 years as predicted by Gordon E. Moore in 1965 in his original paper [33], widely known as “Moore’s law”. This trend is expected to hit the “law of nature” around 2015, as fundamental barriers in physics will start to play a limiting factor in wafer fabrication technology. As the CMOS technology improved drastically over the last 3 decades in terms of die area, speed and power consumption, more and more sophisticated compute intensive applications involving heterogeneous components are becoming integrated into a single chip and finding their way into the portable electronics market [31]. The burden of designing these so called systems-on-chip solutions has lead the engineers and researchers all over the world to develop new architectures and design methodologies in order to meet extremely tight design constraints (low power, high speed, low cost, flexibility etc.). This thesis contributes to the area by presenting a new approach to programmable hearing aid design with low power being the most important design constraint.

This chapter will provide an introduction to the thesis. The chapter is organized as follows. Section 1.1 will describe the field of research that this thesis contributes to. Following this, section 1.2 will present the particular application domain of interest, and section 1.3 will describe the power consumption issues regarding programmable platforms. The proposed approach in this thesis is briefly summarized in the same section. Finally the organization of the thesis will be presented in section 1.4.

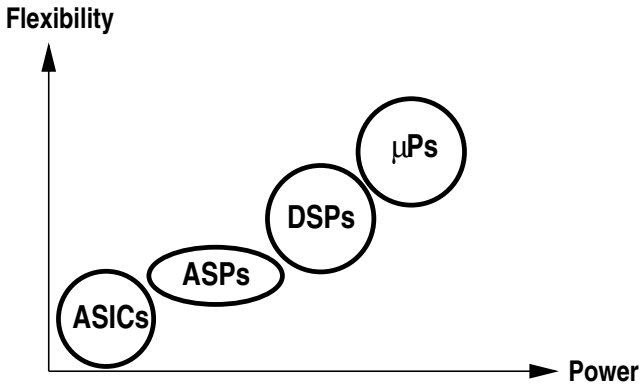

Figure 1.1: Power versus flexibility.

1.1 Application/Domain-specific processors

The ever-increasing functional complexity of sophisticated portable applications require carefully designed integrated circuits (systems-on-chip) that consume low power. Energy-efficiency is best achieved with dedicated hardwired circuits (ASICs) that are tailored to a single application. A closely related issue is time-to-market. These future single-chip, full-function devices need to accommodate rapid changes in algorithms and evolving standards with a fast turn-around time. This calls for programmable and/or reconfigurable designs. Unfortunately programmability and low-power are conflicting goals as illustrated in figure 1.1: dedicated hardwired circuits (ASICs) offer low-power consumption, high speed, and small area but they are not flexible. Even a small change in function calls for a redesign and refabrication of a new chip. At the other end of the spectrum are programmable digital signal processors (DSPs), and general-purpose microprocessors (μP). These general purpose machines have the ability to run a broad range of applications on a general purpose datapath, using a sequential control mechanism, leading to high power consumption, large die areas, and many execution clock cycles per task.

Ideally one would want the power efficiency of a hardwired ASIC solution while maintaining the flexibility of a programmable processor, and the design space between the hardwired ASICs and the general-purpose DSP's attracts a significant amount of research interest [85, 93, 77, 56, 78, 61, 57, 58, 63, 82, 69, 89, 48, 1, 52, 54]. A similar trend is identified in the SIA 2001 technology roadmap that predicts "flexibility-efficiency trade-off shifting away from general purpose processing" [12]. Some researchers address the problem from the DSP side and advocate so-called *ASPs – application/domain-specific processors;* i.e. special-

ized instruction set processors that are optimized for a given set of algorithms. Other researchers address the problem from the ASIC-side and provide the designer/programmer with a set of RTL-level components (register files, multipliers, adders etc.) and a (dynamically) reconfigurable network that allow arbitrary data-flow types of computing structures to be formed. This thesis explores an architecture that falls between the two, although closer to the application/domain-specific approach.

1.2 Motivation for this thesis

The application domain we are considering: audio signal processing – and more specifically digital hearing aids; has enjoyed the advances in integrated circuit technology like other portable equipments. The first transistor-based behind the ear (BTE) hearing aid was introduced in 1952 [2]. The first BTE hearing aid featuring an integrated circuit hit the market in 1964. Up until 1986, hearing aids were based on analog circuitry. The first commercial release of a digital IC to be integrated into an analog hearing aid occurred the same year [3].

Because hearing aids have extremely low power consumption requirements – typical total power consumption in the order of 0.5 - 1.0 mW (at 1.0 V supply) – many commercial hearing aids are based on hardwired ASIC solutions (including the recently published [62]). With the advances in audiology, and the development of more sophisticated algorithms such as noise reduction, feedback cancellation, adaptive filtering (directional amplification); the algorithmic complexity for hearing aids is increasing considerably. Added to this is the fact that design of a hardwired ASIC implementation is a tedious task that involves high non-recurrent engineering (NRE) costs and high risks. For this reason, there is a constant push from the industry to bring forward an ultra-low power programmable DSP that meets the target power consumption and area constraints. Such a programmable DSP is yet to exist, and it is unclear if or when such DSP technology will catch up with the design constraints implied by the increasingly sophisticated algorithms. This push for programmability recently started to give promising results. A domain-specific DSP processor [61, 4] developed by GN Resound and Audiologic was among the first fully programmable DSP architecture to be used in hearing aids. The instruction set and datapath of this architecture are optimized for a set of algorithms used in GN Resound hearing aids, hence the term *domain-specific*.

The aim of this thesis is to explore and contribute to the field of application/domain-specific processing by devising a programmable platform for audio signal processing, in particular hearing aids. A limited but representative set of DSP algorithms used in hearing aids are studied in chapter 4. The platform we

aim for will be fully programmable within the application domain, with an energy-efficiency approaching that of a dedicated ASIC implementation.

1.3 Programmable platforms

Even though programmable DSPs are specialized in digital signal processing, they offer a high degree of flexibility. The flexibility of a programmable DSP stems from a general-purpose datapath and control. The datapath of a programmable DSP typically includes general purpose storage such as register files, program and data memories often coupled with caches to minimize the processor-memory speed bandgap. Such a datapath also includes ALUs, multipliers that are fixed to a word length that has often larger precision than required, and highly capacitive global data, and program memory buses. The control circuitry is designed to handle a very large instruction set that covers all signal processing algorithms. Unfortunately such a general purpose datapath typically consumes an order of magnitude more power than a dedicated ASIC datapath.

An alternative programmable platform to programmable DSPs is reconfigurable architectures. The main focus on reconfigurable architectures has been to improve performance of DSP systems. This has been possible because, compared to sequential DSP processors parallel hardware provides a better match for the signal processing algorithms. Currently, there are some attempts to get low power consumption using such architectures [10, 20]. Reconfigurable architectures possess both software and hardware programmability. However, this comes at a price. A prominent drawback of these architectures is the high-energy consumption of flexible interconnect structures. Further research is needed in this field to come up with an overall low power system.

What is offered as a solution in this thesis is a heterogeneous multiprocessor architecture consisting of a low power DSP/CPU core as well as small and simple instruction set processors called *mini-cores* each tailored to a single class of algorithms within the application domain. For instance an FIR mini-core for FIR algorithms, an IIR mini-core for IIR algorithms etc. We overcome the issues related to general-purpose flexibility of a conventional DSP by providing a custom processor for each algorithm class. Furthermore the platform with its multitude of various mini-cores and the inclusion of a DSP/CPU core has more parallelism than that of a single programmable DSP. As it will be clear in chapter 4, the application domain we are investigating has modest communication requirements, thus a network optimized for mostly idle operation together with low power mini-cores will lead to an energy efficient overall architecture.

The idea is to provide a platform with energy-efficient mini-cores running com-

pute intensive parts of an application, and DSP/CPU-cores running less demanding irregular and/or control oriented parts. The mini-cores and DSP/CPU core will be wrapped with the same communication protocol leading to a modular, easy-to-build programmable platform. Furthermore, communication between processor nodes in the system will be provided by an interconnection network of any topology (Bus, Torus etc.) that supports message passing among the processors. The topology of the network depends on the application requirements.

1.4 Thesis organization

The thesis is organized as follows.

Chapter 2 "Low power design" provides background in low power design. The sources of power consumption, the design parameters to optimize are presented. Furthermore, techniques at different levels of design abstraction are discussed.

Chapter 3 "Related work" discusses related work, by presenting some alternatives for a low-power and programmable platform. These are (1) some commercial low power programmable DSPs (2) domain-specific DSP-cores (3) reconfigurable coarse-grained FPGA like architectures (4) methodologies and tools for synthesis of ASIPs – application specific instruction set processors.

Chapter 4 "Algorithm suite for hearing aids" presents the target application domain i.e., the algorithm suite used in hearing aids, and discusses possible implementations aiming for a programmable platform.

Chapter 5 "Overall architecture" describes the proposed template architecture, lists its advantages and discusses mapping of the hearing aid algorithms onto this architecture.

Chapter 6 "Implementing the idea" gives insight to the design of two mini-cores and an interconnect network, used in the prototype chip that has been fabricated and tested successfully.

Chapter 7 "Testing the chip" presents the prototype chip and the test environment.

Chapter 8 "Results" compares the prototype chip with some alternatives: (1) a low power off-the-shelf DSP processor by Texas Instruments (2) a low power

RISC/DSP-core intended for SoC-based designs by ARC International (3) Two hardwired ASICs designed by Oticon A/S. The goal is to identify where the mini-core platform is in the power vs. flexibility curve of figure 1.1.

Chapter 9 “Conclusion” finally concludes the thesis, and discusses future work.

Chapter 2

Low Power Design

The beginning of low power electronics can be traced to the invention of the bipolar transistor in 1947. Elimination of the requirements for several watts of filament power and several hundred volts of anode voltage in vacuum tubes in exchange for transistor operation in the tens of milliwats range was a breakthrough of unmatched importance in low power electronics. The capability to fully exploit the superb low power assets of the bipolar transistor was provided by a second breakthrough, the invention of the integrated circuit in 1958. Although far less widely acclaimed as such, a third breakthrough of indispensable importance to modern low power digital electronics was the complementary metal-oxide-semiconductor or CMOS integrated circuit announced in 1963 [44].

This chapter summarizes techniques for minimizing power consumption in CMOS circuits and can be skipped by the "expert" reader. The goal is to provide a background in low power design. Section 2.1 motivates the importance of low power consumption. Sources of power consumption are explained in section 2.2. Design parameters that effect power consumption is discussed in section 2.3. Finally, section 2.4 presents power minimization techniques at various levels of abstraction.

2.1 Motivation for low power

Historically, the task of the VLSI designer has been to explore the Area-Time implementation space, attempting to strike a reasonable balance between these often conflicting objectives. But area and time are not the only metrics by which we can measure implementation quality. *Power consumption* is yet another criterion [46].

The motivation for low power electronics has stemmed from three reasonably distinct classes of requirement [13]:

- the earliest and most demanding of these is for portable battery operated equipment that is sufficiently small in size and weight and long in operating life. The goal is to satisfy the user of hearing aids, implantable cardiac pacemakers, wristwatches, pocket calculators and pagers.
- the most recent need is for ever-increasing packing density in order to further enhance the speed of high performance systems, which imposes severe restrictions on power dissipation density.
- and the broadest need is for conservation of power in desk-top and desk-side systems where cost-to-performance ratio for a competitive product demands low power operation to reduce power supply and cooling costs.

Viewed together, these three classes of need appear to encompass a substantial majority of current applications of electronic equipment. Low power electronics has become the mainstream of the effort to achieve gigascale integration (GSI).

2.2 Sources of power consumption

In CMOS circuits, there are two major sources of power dissipation [64].

- **Static dissipation**, due to leakage current or other current drawn continuously from the power supply.
- **Dynamic dissipation**, due to
  - switching transient (short-circuit) current,
  - charging and discharging of load capacitances

Total power dissipation can be obtained from the sum of these components as summarized in equation (2.1).

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

2.2.1 Dynamic dissipation

The first two terms in equation (2.1) represent the dynamic source of power dissipation. The switching component, $P_{switching}$, arises when the capacitive load, C_L, of a CMOS circuit is charged through PMOS transistors to make a voltage transition from 0 to the high voltage level, which is usually the supply, V_{dd}.

For an inverter circuit as shown in figure 2.1, the power dissipated because of a 0 to 1 transition can be determined from the product $V_{dd} \cdot I_C$ where I_C is the transient

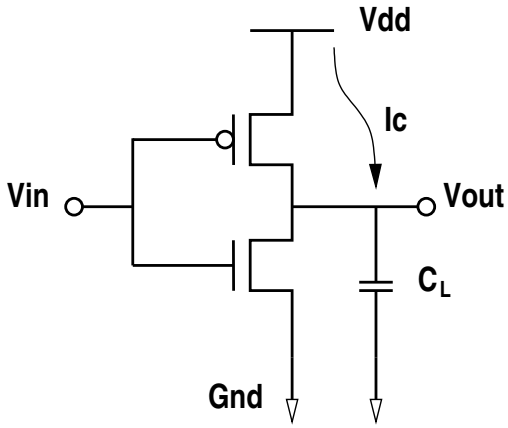

Figure 2.1: An inverter.

current drawn from the supply. The time duration for this current flow is T. It can be written in 2.2.

$$I_C = C_L \frac{dV_{out}}{dt} \tag{2.2}$$

The energy drawn from the power supply is given in 2.3.

$$E_{0\to1} = \int_0^T V_{dd} \cdot I_C(t)dt = V_{dd} \int_0^{V_{dd}} C_L \cdot dV_{out} = C_L \cdot V_{dd}^2 \tag{2.3}$$

Half of the energy given in (2.3) is stored in the output capacitor and half of it is dissipated in the PMOS transistor [14]. On the 1 to 0 transition at the output, no charge is drawn from the supply, however the energy stored in the output capacitor is consumed. If these transitions occur at a clock rate, f_{clk}, the power drawn from the supply is $C_L \cdot V_{dd}^2 \cdot f_{clk}$. However, in general the switching will not occur at the clock rate (except for clock buffers), but rather at some reduced rate, which is best described probabilistically. $\alpha_{0\to1}$ is defined as the average number of times in each clock cycle that a node with a capacitance C_L will make a power consuming transition (0 to 1), resulting in an average switching component of power for a CMOS gate to be,

$$P_{switching} = \alpha_{0\to1} \cdot C_L \cdot V_{dd}^2 \cdot f_{clk} \tag{2.4}$$

Another dynamic component of power dissipation is $P_{short-circuit}$. At some point during the switching transient, both the NMOS and PMOS devices in figure 2.1 will be turned on. This occurs for gate voltages between V_{tn} and $V_{dd} - |V_{tp}|$ where V_{tn} and $|V_{tp}|$ are threshold voltages of the NMOS and PMOS transistors, respectively. During this time, a short-circuit exists between V_{dd} and ground. Therefore currents are allowed to flow. If $V_{dd} - Gnd < V_{tn} + |V_{tp}|$ is satisfied then a short circuit path between the power supply and ground will never exist, meaning that this component of (2.1) can be eliminated. But even though $P_{short-circuit}$ can not always be ignored, it certainly is not the dominant component of power consumption. An analytical derivation for $P_{short-circuit}$ is given in [37].

2.2.2 Static dissipation

Ideally, CMOS circuits dissipate no static (DC) power since in the steady state there is no direct path from V_{dd} to ground. Of course, this scenario can never be realized in practice since in reality the MOS transistor is not a perfect switch. Static power dissipation, $P_{leakage}$, stems from the leakage current, $I_{leakage}$, which can arise from substrate injection and subthreshold effects and primarily determined by fabrication technology considerations. This current is typically in the nA region and contributes little to the overall power consumption. However, in future deep submicron technologies, leakage power will become a problem.

The most dominant component of power dissipation currently is $P_{switching}$ given in (2.4). Next section will introduce techniques in order to reduce $P_{switching}$.

2.3 Techniques for low power

The previous section revealed the parameters that the designer needs to change for low power design as shown in equation (2.4): voltage, physical capacitance, and activity. Unfortunately the difficulty for power optimization arises from the fact that these parameters are not completely orthogonal. Therefore they can not be optimized independently.

2.3.1 Supply voltage

With its quadratic relationship to power, voltage reduction offers the most direct means of minimizing power consumption. Without requiring any special circuits

or technologies, a factor of two reduction in supply voltage yields a factor of four decrease in energy. Because of this quadratic relationship, designers are willing to sacrifice increased physical capacitance and activity for reduced voltage. Unfortunately supply voltage can not be decreased without bound. In fact several other factors influence the selection of a system supply voltage. The primary determining factors are performance requirements and compatibility issues. Reducing the supply voltage degrades the speed of a CMOS circuit. There are architectural techniques that deal with this problem. They will be presented in section 2.4.3.

The other limiting criterion is the issue of compatibility. Most of the off-the-shelf components operate at either 5 V supply or, more recently, a 3.3 V supply. Unless an entire system is being designed completely from scratch, it is likely that some amount of communication between standard and non-standard components will be required. Highly efficient DC-DC level converters ease the severity of this problem, but still there is some cost involved in supporting several different supply voltages. This hints that it might be useful to support only a small number of distinct intra-system voltages.

2.3.2 Physical capacitance

Dynamic power consumption depends linearly on the physical capacitance being switched. In addition to operating at low voltages, minimizing capacitance offers another technique for minimizing power consumption.

The physical capacitance in CMOS circuits stems from two primary sources: devices and interconnect. As technologies continue to scale down, interconnect parasitics will start to dominate over device capacitances.

Capacitances can be kept at a minimum by using less logic, smaller devices, and fewer and shorter wires. Some techniques reducing the active area include resource sharing, logic minimization and gate sizing. Techniques for reducing the interconnect include register sharing, common sub-function extraction, placement and routing. However we are not free to optimize capacitance independently. For example reducing device sizes reduces physical capacitance, but it also reduces the current drive ability of the transistors making the circuit operate more slowly. This loss in performance might prevent us from lowering V_{dd} as much as we might otherwise be able to do. If the designer is free to scale voltage it does not make sense to minimize physical capacitance without considering the side effects. Likewise, if voltage and/or activity can be significantly reduced by allowing some increase in interconnect capacitance, then this may result in a net decrease in power.

2.3.3 Activity

A chip can contain a huge amount of physical capacitance, but if it does not switch then no dynamic power will be consumed. The activity determines how often this switching occurs. As given in (2.4) there are two components to switching activity. the first is the data rate, f_{clk}, which reflects how often on average, new data arrives at each node. This data might or might not be different from the previous data value. In this sense, the data rate f_{clk} describes how often on average, switching *could* occur. For example, in synchronous systems f_{clk} might correspond to the clock frequency.

The second component of activity is the data activity, $\alpha_{0\to1}$, corresponding to the expected number of energy consuming transitions that will be triggered by the arrival of each new piece of data. So while f_{clk} determines the average periodicity of data arrivals, $\alpha_{0\to1}$ determines how many transitions each arrival will spark. For circuits that do not experience *glitching* $\alpha_{0\to1}$ can be interpreted as the probability that an energy consuming (zero to one) transition will occur during a single clock period.

Calculation of $\alpha_{0\to1}$ is difficult as it depends not only on the switching activities of the circuit inputs and the logic function of the circuit, but also on the spatial and temporal correlations among the circuit inputs. The data activity inside a 16-bit multiplier may change by as much as one order of magnitude as a function of input correlations [46].

The data activity $\alpha_{0\to1}$ can be combined with the physical capacitance C_L to obtain an effective capacitance, $C_{eff} = \alpha_{0\to1} \cdot C_L$ which describes the average capacitance charged during each $1/f_{clk}$ period. This reflects the fact that neither the physical capacitance nor the activity alone determines dynamic power consumption. Evaluating the effective capacitance of a design is non-trivial, as it requires knowledge of both the physical aspects of the design (such as technology parameters, circuit structure, delay model) as well as the signal statistics (data activity and correlations). This explains why, when lacking proper tools, power analysis is often deferred to the latest stages of the design process.

2.4 Minimizing power consumption

We have seen the design variables that effect the dynamic power consumption of a CMOS circuit. Now we will investigate the power minimization problem from various design aspects that effect power dissipation: technology, circuit techniques, architectures and algorithms.

2.4.1 Technology

An optimization that could be done at this level is driven by voltage scaling. As seen in section 2.3.1, it is necessary to scale supply voltage for a quadratic improvement in energy per transition. Unfortunately, we pay a speed penalty for a V_{dd} reduction with delays increasing, as V_{dd} approaches the threshold voltage of the devices. The simple first order relationship between V_{dd} and gate delay, t_d for a CMOS gate is given in 2.5,

$$t_d = \frac{2 \cdot C_L \cdot V_{dd}}{\mu \cdot Cox \cdot (W/L) \cdot (V_{dd} - V_t)^2} \tag{2.5}$$

The objective is to reduce power consumption while keeping the throughput of the overall system fixed. Therefore compensation for these delays at low voltages is required. Section 2.4.3 will present architectural techniques for meeting throughput constraints.

At the technology level, an approach to reduce the supply voltage without loss in throughput is to lower the threshold voltage of the devices. However, lower threshold means higher stand-by power consumption, therefore only transistors that comprise delay-critical paths should be modified. These multi-threshold circuits attract significant research interest [76, 53, 79].

Since a significant power improvement can be gained by the use of low-threshold devices, another issue to address is how low the thresholds can be reduced. The limit is set by the requirement to retain adequate noise margins and the increase in subthreshold currents.

2.4.2 Circuit techniques

There are a number of options available in choosing the basic circuit approach and topology for implementing various logic and arithmetic functions. Choices between static vs. dynamic implementations, pass-transistor vs. conventional CMOS logic styles, and synchronous vs. asynchronous timing are just some of the options open to the system designer. At the RT level, there are also various architectural choices for implementing a given logic function; for example to implement an adder module one can utilize a ripple-carry, carry-select, or carry-lookahead topology.

Dynamic vs. static logic

Dynamic logic has some inherent advantages in a number of areas including (1) reduced switching activity due to hazards, (2) elimination of short-circuit dissipa-

tion, and (3) reduced parasitic node capacitances. These are explained briefly in the following.

(1) Static designs can exhibit spurious transitions (also called dynamic hazards [64]) due to finite propagation delays from one logic block to the next i.e., a node can have multiple transitions in a clock cycle before settling to the correct level. The number of these extra transitions is a function of input patterns, internal state assignment in the logic design, delay skew and logic depth. Though it is possible with careful logic design to eliminate these transitions, dynamic logic does not have this problem, since any node can undergo at most one power consuming transition per clock cycle.

(2) Short circuit currents caused by a direct path from power supply to ground are found in static CMOS circuits. However, by sizing transistors for equal rise and fall times, the short-circuit component of the total power can be kept to less than 20% of the dynamic switching component [37]. Dynamic logic does not exhibit this problem, except for those cases in which static pull-up devices are used to control charge sharing.

(3) Dynamic logic typically uses fewer transistors to implement a given logic function, which reduces the amount of capacitance being switched.

The one area dynamic logic has a distinct disadvantage is the requirement for a precharge operation and the "charge sharing" problem. In dynamic logic every node must be precharged every clock cycle. Even when the logic inputs do not change, output nodes with "low" voltages (logic zero) are precharged only to be immediately discharged again as the node is evaluated. The other drawback, "charge sharing" stems from turned on NMOS transistors that short-circuit the output node to internal nodes. Even if the gate should not evaluate to logic zero as there is no direct path to the ground, charge sharing may cause the output voltage level to drop significantly and cause the next logic stage to interpret a logic zero instead of logic one. Charge sharing can be solved by using a weak static pull-up device (PMOS transistor), unfortunately this means static power consumption.

Finally, power-down techniques achieved by disabling the clock signal have been used effectively in static circuits, but are not as well suited for dynamic techniques.

Pass-transistor vs. static logic

Complementary pass-transistor logic (CPL) family is one form of logic that is popular in NMOS-rich circuits [64, 51]. The gate design uses only NMOS transistors and requires the inverted input signals as well to implement Karnaugh maps for logic functions. As logic signals are only passed through NMOS transistors, the "high" output signal may deteriorate because of threshold voltage drops. This will

require the output signals to be regenerated by inverters/buffers.

Pass-transistor logic is attractive as fewer transistors are required to implement important logic functions, such as XOR's which only require two pass transistors in a CPL implementation. This particularly efficient implementation of an XOR is important since it is key to most arithmetic functions, permitting adders and multipliers to be created using a minimal number of devices. Likewise, multiplexers, registers, and other key building blocks are simplified using pass-gate designs.

However, a CPL implementation (explained in detail in [51]) has two basic problems: (1) the threshold drop across the pass transistors results in reduced current drive and hence slower operation at reduced supply voltages and (2) The "high" input voltage level at the regenerative inverters is not V_{dd}, therefore the PMOS device in the inverter is not fully turned off. This may cause significant static power dissipation.

Synchronous vs. asynchronous

In synchronous designs, the logic between registers is continuously computing every clock cycle based on its new inputs. To reduce the power consumption in synchronous designs, it is important to minimize switching activity by powering down execution units when they are not performing useful operations.

While the design of synchronous circuits requires special design effort and power-down circuitry to detect and shut down unused units (clock gating), asynchronous logic has inherent power-down of unused modules, since transitions occur only when necessary. However, asynchronous implementations require the generation of a completion signal indicating the validity of the output signals. This control logic represents an overhead in terms of silicon area, speed and power consumption. Therefore, one has to ask whether or not, the use of asynchronous techniques result in a substantial improvement over the synchronous counterpart [47].

Circuit topology

Independent of the logic style used, the topology to implement a given function can affect the capacitance switched. For instance, let's consider a ripple-carry vs. carry select adder. These designs are explained in detail in [64].

In order to do addition faster, a carry-select adder (CSA) incorporates dual carry path. One carry path assumes logic zero at the carry input signal, and the other assumes a logic one. Therefore, one of these paths is computing irrelevant outputs. Furthermore selecting the actual carry and sum requires extra circuitry. Obviously, the number of transitions per addition is bigger in the carry select adder

assuming both adders being implemented in static CMOS logic style. Ideally, it is always better to use a topology that consumes the least amount of energy per operation. Unfortunately, the choice of circuit approach is not independent of circuit speed. At large bit-widths, the CSA is faster than the ripple carry adder. This speed advantage can be used to lower the supply voltage while keeping the throughput of the system constant. Consequently a CSA could very well be the low power choice even though it switches more capacitance.

2.4.3 Architecture optimization

As seen in equation 2.5, gate delays increase drastically, when supply voltage approaches the threshold voltage of the MOS transistor. There are two architectural techniques that can improve the speed of the circuit under reduced supply voltage:

(1) Pipelining: It is a powerful transformation of the datapath to reduce the critical path of the system and improve the speed. It involves the insertion of delay elements/flip flops at specific points of a data flow graph of an algorithm/architecture. The speed gained by this transformation can be traded for low power by voltage scaling.

(2) Parallelism: It is similar to pipelining in that it exploits parallelism in a system, however here this is achieved by duplicating hardware in order to perform a number of similar tasks concurrently.

The authors of [19] show the advantages of both approaches through an adder-comparator example. The original design consists of an adder followed by a comparator with equal circuit delays. There are registers at the input of the adder and the comparator. The pipelined version is created by inserting registers in between the adder and comparator. The supply voltage could be scaled down as the pipeline register allows the delays to increase by a factor of two. This is due to the equal circuit delay assumption for both the adder and the comparator. The parallel version is created by using a pair of adder-comparator structures. Each adder-comparator unit runs two times slower than the original design. By overlapping the operation of each adder-comparator unit, this version selects the available output from the "finished" adder-comparator unit via a multiplexer. This parallel version still communicates data with the external world using the original clock rate even though the individual units work slower. This speed gain can be traded for low power by scaling the supply voltage. The gains for both approaches in terms of power consumption are similar. However pipelining has a smaller area overhead compared to hardware duplication. One could of course combine both approaches to gain even more improvements in speed.

2.4.4 Algorithm

Choosing the algorithm to implement the application at hand represent the most important decision in meeting the power constraints. From the previous section, we can deduce that in order to reap the greatest architectural gains, the ability to parallelize an algorithm will be critical, and the basic computation must be optimized, as the basic theme in low power design is *voltage reduction*.

Therefore, at the algorithmic level, transformations that can be used to increase speed and allow lower voltages are useful. Often these approaches translate into larger silicon area; hence the approach has been termed *trading area for power*. Design exploration at this level require methods and tools to guide the system-on-chip designer.

Another technique for low power design is to avoid wasteful activity. At the algorithm level, the size and complexity of a given algorithm i.e. operation counts, word lengths and so on determine the activity. If there are several algorithms for a given task, the one with the least number of operations (arithmetic operation, memory access etc.) is generally preferable. A study based on the vector quantization algorithm [60] supports the importance of optimizing at this level.

Algorithm optimization should also consider memory usage as memory access in digital systems is typically expensive in terms of power. At the architectural level, using memory hierarchy to reduce power consumption is a well-known idea. This is based on the fact that memory power consumption primarily depends on the access frequency and the size of the memory [28]. At the algorithmic level, optimizations that reduce memory access frequency (exploitation of temporal locality [84]), and HW/SW partitioning of a system based on minimizing memory requirements are important aspects of design that effect memory and hence overall system power consumption [22].

2.5 Summary

Present-day technologies possess computing capabilities that enable the design of powerful work stations, sophisticated computer graphics, and multi-media applications such as real-time audio and video signal processing. Furthermore, users of these applications have the desire to access this computation at any location. Thus, the requirement of portability has put severe restrictions on size, speed and power consumption. Improvements in battery technology are being made, but it is highly unlikely that a dramatic solution to power is forthcoming.

Interest in low power has urged the researchers to look at the problem from the designer's point of view. Techniques at various levels of design abstraction

are being investigated. This chapter introduced the source of the problem and presented some of the techniques involved.

Chapter 3

Related Work

This chapter presents a collection of state-of-the-art work within the application/domain specific programmable computing field. As power dissipation is becoming a major concern accompanied by time-to-market issues, we can identify mainly three research areas that focus on flexible and low-power platforms:

(1) Programmable DSPs are among the oldest domain-specific processors, their specific application domain being digital signal processing. Section 3.1 will present programmable DSPs, their assets and the architectural evolution they have gone through since their introduction.

(2) When flexibility is of concern, reconfigurable architectures have also been preferred design solutions for signal processing algorithms during the past couple of decades. Section 3.2 will focus on recent developments and trends within the field.

(3) Section 3.3 will present work regarding automated ASIP (application-specific instruction set processor) design methodologies and/or techniques that assist the system-on-chip designer in developing domain-specific computer architectures.

Finally, section 3.4 will summarize the chapter

3.1 Programmable DSPs

Programmable DSPs are specialized microprocessors for real-time number crunching [26, 27]. Because of their specialized applications, programmable DSPs have evolved architectures that are significantly different from conventional microprocessors. With special arithmetic capabilities and data addressing modes, DSPs have consistently outperformed microprocessors in signal processing applications. One could say that a programmable DSP is a *domain-specific* processor that targets

signal processing.

Moreover, the current trend in the electronics market indicates that wireless technologies for mobile applications are becoming a reality for the new millennium [31]. The vision of future telecommunications is "information at any time, any place, and in any form". In the core of these sophisticated applications lie intensive signal processing algorithms thus an increasing need for DSP processors in general. Realizing that DSP processors have already become a driving force in both multimedia and communications, conventional microprocessors have added increasingly more DSP extensions to their products over the past three years [91].

As these battery-powered constantly evolving/changing mobile applications push for flexible and low-power system-on-chip solutions, DSP vendors are putting more effort into architecture and process enhancements in order to obtain energy-efficient DSP processors. One such approach taken by DSP vendors is to optimize DSP architectures with an application domain in mind i.e., to design *domain-specific* DSPs. For instance, Texas Instruments' C54x family, is optimized for wireless applications [32]. This processor has a domain-specific compare, select, and store unit (CSSU) to accelerate the Viterbi butterfly operations that are part of many communications algorithms. Texas Instruments extended the basic architecture of c54x family further by adding one more MAC unit, thereby increasing instruction level parallelism. The end low power DSP product family is called the c55x family. Other DSPs on the market that target wireless applications are the Lucent 16000 series [11] and the ADI21xx series from Analog Devices.

A domain-specific approach has also been chosen to design the Lode DSP core [89]. It is a 16-bit DSP engine developed specifically for next generation wireless digital systems. It has a dual multiply-accumulate unit with two data buses, and an ALU unit. The internal bus network is designed such that all three units (2 MAC, ALU) are operating in parallel. With a smart organization of the dual MAC unit as shown in figure 3.1, the processor requires only half the number of memory accesses during an FIR filter computation compared to a conventional DSP processor.

The organization in figure 3.1 computes two outputs in parallel with 2N+1 memory accesses. Here N is the order of the FIR filter being computed. In a traditional single MAC DSP, each output sample is computed in sequence and requires 2N memory accesses. Notice the shift register that contributes this performance increase in figure 3.1. That local register will shift the input samples. Data bus 0 will be fetching the coefficients, whereas data bus 1 will be fetching input data. The first accumulator, a0, will store $y(n)$ output, and the second accumulator will store $y(n+1)$ output. This structure can be generalized to contain N MACs in parallel connected by a delay line, resulting in an N-fold increase of the performance. The performance increase of the architecture can be used to achieve low power by

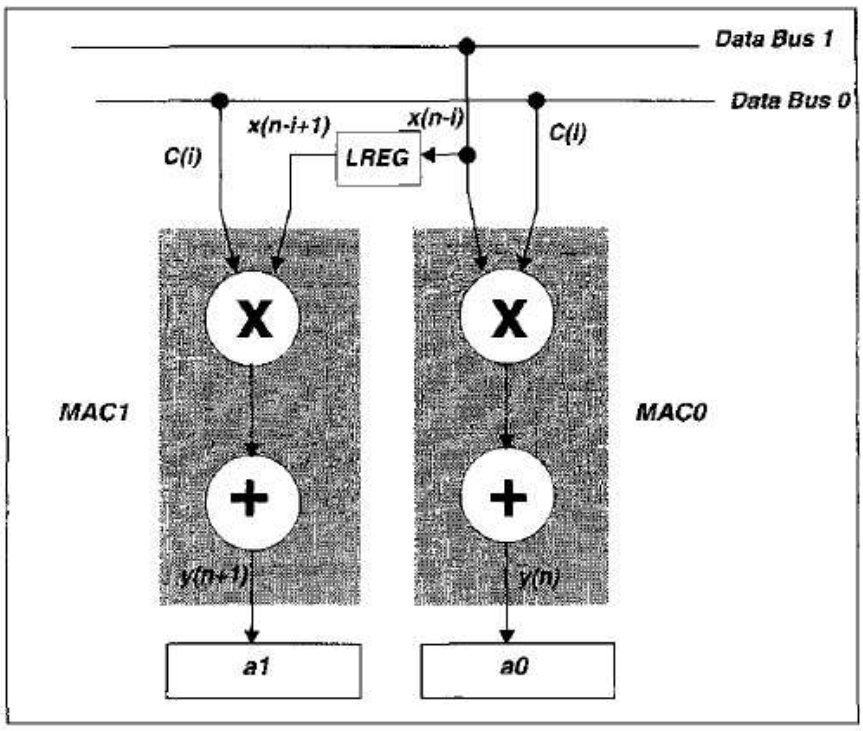

Figure 3.1: Dual MAC architecture of the Lode DSP core, Verbauwhede et al.

slowing the clock rate or to add more functionality in software.

DSP processor architectures are also evolving towards more instruction level parallelism [45]. This is achieved by VLIW (Very Long Instruction Word) instruction set processors that contain multiple execution units such as MAC units, ALUs and address generator units that are operating in parallel. The CARMEL core from Infineon is such a VLIW architecture that can do 6 simultaneous operations. It is a 16-bit, fixed point DSP core that targets advanced communications and consumer applications. Its modular architecture allows for complete SoC implementations. The datapath of the architecture consists of 2 ALUs, 2 MAC units, an exponent unit and a barrel shifter. The exponent unit is used for determining a shift value to normalize 16-, 32- or 40-bit input operands. The core has three distinct classes of instruction types corresponding to 24-, 48-, and 144-bits. The 144-bit block instruction is used to specify two ALU and two MAC operations together with two data moves.

In some designs, the performance improvements obtained through parallelism can be traded with low power consumption [89, 52] by using low voltage and slow

clock frequency. One such DSP architecture is from Kumura et al. [52]. It is a 4-way VLIW machine, with 2 MACs, 2 ALUs, 2 data address units (DAUs) and a system control unit (SCU). Up to four units among these can work during the same clock cycle. The MACs execute 16 x 16-bit multiply and 40 bit multiply-accumulate operations. The instructions of [52] are either 16 or 32-bit wide and can be grouped into 64-bit instruction packets. The functional block diagram of the processor is shown in figure 3.2. It has 8 general purpose registers and 16 data address registers.

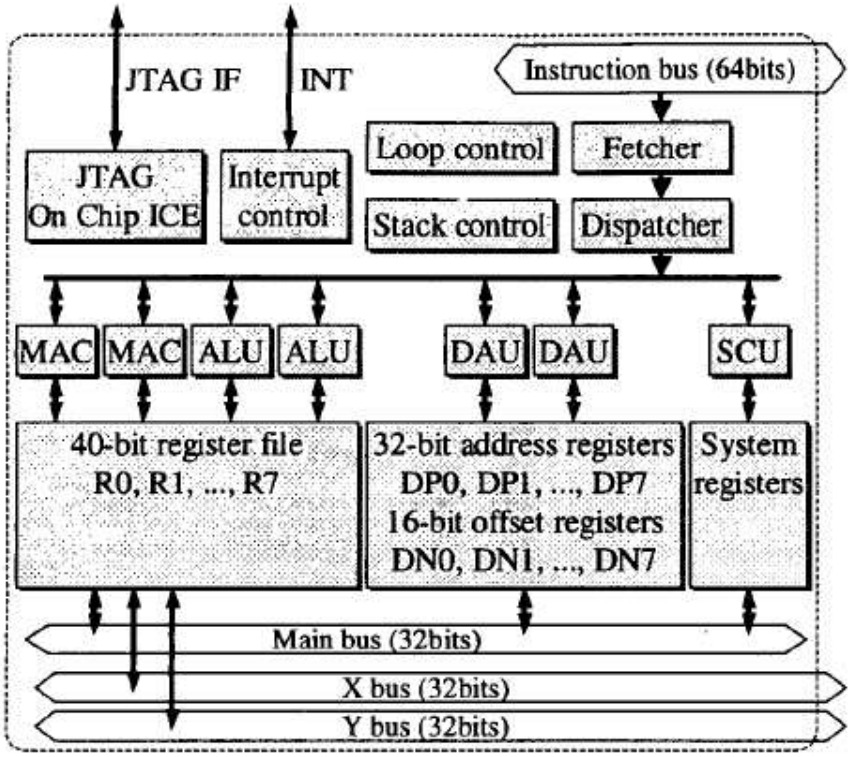

Figure 3.2: Functional block diagram of the DSP-core for 3G mobile terminals by Kumura et al.

The processor in [52] is realized in 0.13 μ process, and is able to perform both video and speech codec for 3G wireless communications at 384 kbit/sec with a power consumption of approximately 50 mW at 0.9 Volts while running 250 MHz system clock.

Lai et al. describes another domain-specific DSP core in [54]. The application domain of interest is the MP3 decoding algorithm. It is a 4-stage pipeline:

instruction fetch, instruction decode, operand fetch and instruction execution. The authors of [54] use instruction level clock gating i.e., clocking only the necessary pipe stages/modules during the execution of a single instruction. The design employs three power modes: (1) running mode, (2) idle, and (3) shutdown in order to reduce unnecessary switching activity. The instruction set has 92 instructions in total. The authors of [54] do not provide power figures but the techniques they present are interesting within the low power processor design context.

It is also relevant to mention a couple of state of the art low-power DSP's intended for audio applications. The designs presented in [63] and [58] all use a variety of full-custom circuit techniques, and some of them even use dual V_t processes to obtain high speed and low standby power consumption at the same time. The Coyote processor developed by GN Resound and Audiologic is among the most power efficient designs in existence today [61, 5]. This design significantly resembles a general-purpose DSP architecture with optimizations that emphasize audio signal processing. It has a specialized instruction set that displays high parallelism and a datapath with a special multiply accumulate unit called PMAC. Compared with our approach it is a much more coarse grained processor, and when it comes to power efficiency it benefits from a hand-crafted full-custom design methodology and (like any other traditional general-purpose DSP) it suffers from its size and from its highly flexible datapath that can accommodate all the algorithms within the application domain.

Another related work is [57] where an instruction set processor with a configurable datapath is presented. The application domain covers various wireless communication standards. The datapath basically consists of simple functional units: multipliers, ALUs and shifters. The instruction set of this architecture can be extended with macro-operations that can configure a compound computational unit using the basic functional units. These macro-operations are similar to the LMS and FIRS instructions found in the TMS320C54x DSP processor. The output of any functional unit can be input to another by a configurable feedback path. In our approach, we also have compound functional units to decrease the instruction count of sophisticated DSP algorithms, but we avoid the complexity of configurable structures. For instance, a *dedicated* dual-multiply-accumulate unit exists in the IIR mini-core (presented in chapter 6) in order to handle biquad filters efficiently.

It is also necessary to emphasize that the domain-specific programmable computing field is growing. And it is not only low power that drives the field, as we have encountered with some recent work in this area that focus on compute power i.e., the ability to compute more within a given amount of time. There is an interesting challenge facing multimedia and digital communication systems engineering. The algorithmic complexity in these systems is growing at a phenomenal pace that

the compute power delivered by DSP processors can not follow. Architectures with heterogeneous programmable units are evolving [82, 1] to fill the compute power gap to realize such systems.

Currently most programmable DSPs are inherently sequential machines, even though some parallel VLIW DSPs (such as the TMS320C6x family by Texas Instruments) have recently been developed.

3.2 Reconfigurable computing

Reconfigurable hardware has numerous advantages for many signal processing systems. For instance, customizing the datapath for irregular data widths is possible. Specific constant values can be directly mapped to hardware, reducing implementation area, power and improving data throughput of the system. For a given sampling rate, the algorithm complexity that a DSP processor can handle is limited by the clock cycles available, which is further decided by the maximum clock frequency. On the other hand, more parallelism is available on the reconfigurable hardware, and the application designer has more freedom to deal with sophisticated signal processing.

The inherent data parallelism found in many DSP functions has made DSP algorithms ideal candidates for hardware implementation. Before the introduction of Field Programmable Gate Arrays (FPGA) in mid 80ies, semi-custom approaches such as mask-programmed gate arrays (MPGAs) were often the choice of application designers for implementing DSP type of applications, mainly for speed, cost, and time-to-market concerns [17]. However as easy as it was to implement an application on an MPGA, the end product was not flexible. In the electronics industry, not only time-to-market is vital, but it is also very important that financial risk incurred in the development of the new product is limited so that more new ideas can be prototyped. FPGAs have emerged as the ultimate solution to these time-to-market and risk problems because they provide instant manufacturing and very low cost prototypes.

Conventional FPGAs contain an array of uncommitted elements (configurable logic blocks, CLBs) that can be interconnected in a general way. A typical CLB consists of a 4-input look-up table, a few multiplexers as well as flip-flops. The look-up table can be used to implement any 4-input combinational logic circuit by mapping the truth table of the desired function. These structures offer fine-grained parallelism i.e., logic functionality and interconnect connectivity is programmable at the bit level. Recently the trend in FPGA architectures has been shifting to the use of more complex CLBs. While fine-grained look-up table FPGAs are effective for bit-level computations, many DSP applications benefit from modular arithmetic

operations that suit coarse-grained configurable devices better. Some of the architectures of this nature are PADDI [23], Matrix [25], and ReMarc [83].

The PADDI [23] device is a DSP-optimized multiprocessor architecture that includes 8 coarse-grained configurable blocks, so-called EXUs (Execution Units). The architecture is shown in figure 3.3.

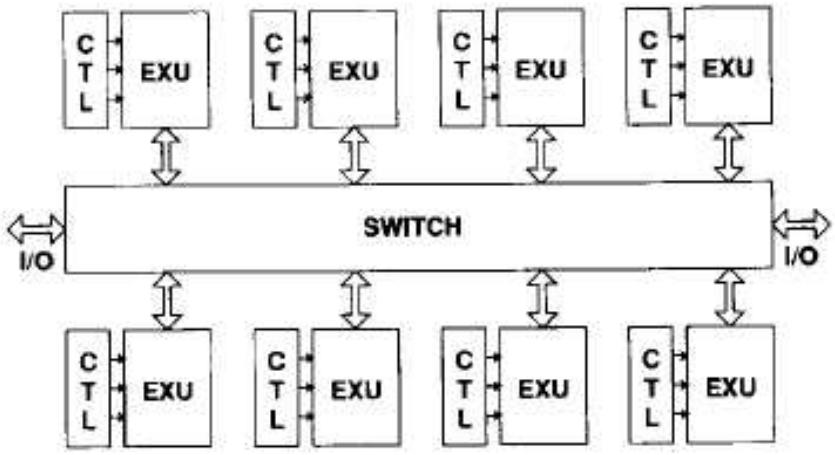

Figure 3.3: The PADDI architecture.

An EXU consist of a small local instruction store, and a configurable datapath with dual-ported register files that could be used to implement delay lines, multiplexers, registers and an ALU. Mapping an application onto the PADDI architecture involves partitioning the data flow graph onto several EXUs. The overall control is achieved by distributing a global address to all EXUs. This results in each EXU fetching and decoding an instruction from its local memory. Communication paths between processors are configured through a cross bar switch and can be changed on a per-cycle basis.

Compared to fine-grained FPGAs, the PADDI device enjoys a very fast ALU as it is a dedicated hard block. Furthermore, it supports flexible routing of large data buses and fast re-configuration of its EXUs through hardware multiplexing. All these advantages are related to performance metrics. Power consumption of this device has not been compared to other approaches in [23].

The Matrix [25] is composed of an array of identical 8-bit functional units called BFU (basic functional unit) overlayed with a configurable network. Each functional unit contains 256x8 bit memory, an ALU, multiply unit, and some control logic. While PADDI has a VLIW-like control word, which is distributed to

all EXUs, the Matrix exhibits more MIMD characteristics. The Matrix operation is pipelined at the BFU level, and furthermore each BFU can function as either instruction memory, data memory, or ALU. It has similar advantages to that of the PADDI compared to a fine-grained FPGA architecture.

The ReMarc [83] architecture targeted to multimedia applications exhibits SIMD-like characteristics with a control word distributed to all processors. It has a two-dimensional grid of 16-bit processors. The architecture is evaluated through a comparison with a conventional FPGA based co-processor. The speed-up of the application that can be achieved by both designs are similar, however the ReMarc architecture occupies a smaller area for the same speed-up factor.

Recently, a booming interest in reconfigurable logic originates from the multimedia and telecommunication community [55, 20]. The said application domain requires easily adaptable platforms for changing standards, and algorithms.

Lange et al. [55] proposes a hardware accelerator for future telecommunication systems based on a generic multiply-accumulate based configurable processing element (PE). The accelerator architecture as shown in figure 3.4 consists of a number of processing elements that are connected to a Read/Write memory for data I/O. The configuration of the PEs occur every clock cycle therefore the accelerator is reconfigurable during run time.

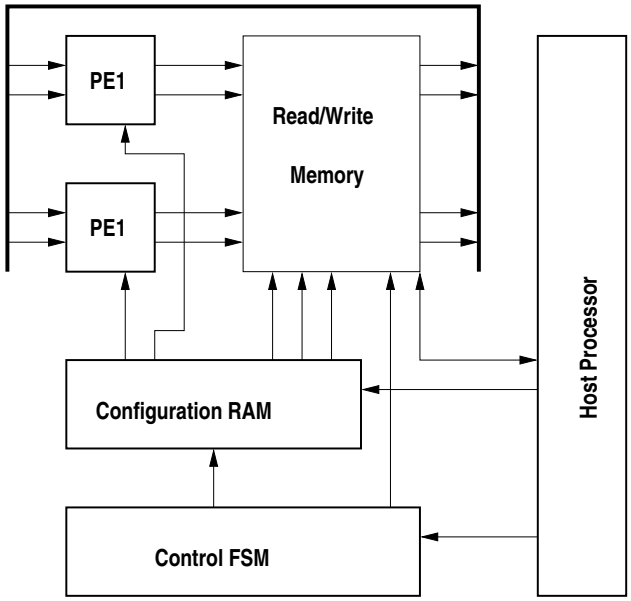

Figure 3.4: Hardware accelerator architecture.

The multiply-accumulate based PE described in [55] contains two multipliers, three adder/subtracter units, two accumulators and several data registers. The PE can efficiently perform multiply-accumulate or multiply-add based algorithms like FFT/IFFT (Finite Fourier Transform, Inverse FFT), real and complex valued FIR filtering, matrix-vector, or matrix-matrix multiplications as well as algorithms composed of these basic operations such as DCT/IDCT (Discrete Cosine Transform, Inverse DCT) or discrete wavelet transforms. The PE is shown in figure 3.5.

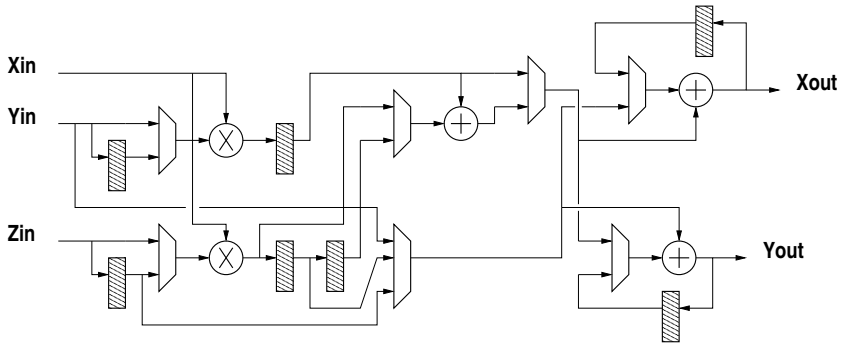

Figure 3.5: Reconfigurable multiply-accumulate based processing element.

The hardware accelerator described in [55] outperforms Infineon's CARMEL 10xx and TMS320C55x by Texas Instruments. Its main advantage compared to conventional hardware accelerators is its reconfigurability, hence its flexibility within the application domain.

In general, coarse-grained reconfigurable architectures have been investigated for performance improvements. Coudhary et al. at Philips Research investigate these architectures also from the energy consumption point of view as well [20]. Their reconfigurable coprocessor architecture, targeting audio codec algorithms consists of a grid of 32 coarse-grained processing and storage units (PSUs). Each PSU includes a multiplier, register files, and memory blocks. One such PSU is capable of performing a multiply, multiply-accumulate or multiply-subtract operation, while getting its data from the local memory blocks or the register files. The connections among the PSUs can be setup in an FPGA-like style as a homogeneous grid or as an irregular partially connected network. The coprocessor shares data memory with the host processor for data communication. A mapping of an application onto the coprocessor involves identifying time critical parts of an application called kernels. These kernels are replaced with remote procedure calls in the original source code. The simulations of the synthesized RTL level implementation of the coprocessor exhibits an order of magnitude lower energy consumption and

two orders of magnitude speed-up than the ARM7 RISC processor [20]. The co-processor architecture has the added advantage of flexibility within the application domain.

The FPGA industry is also following a similar trend towards coarse-grained architectures. For example the new Xilinx Virtex II Pro platform FPGAs contain dual-ported block RAMs, 18x18 multipliers, up to 4 IBM PowerPC processor cores [7] as coarse-grained CLBs.

Reconfigurable hardware has been capable of matching an inherently parallel application better than a general-purpose machine in terms of performance [35, 36, 90]. Currently some researchers are looking into the possibility of using reconfigurable hardware as a design alternative for low power and programmable platforms. We have already mentioned about [20]. Another related work within this context is the Pleiades project [10, 77, 93]. Here an on-chip general-purpose microprocessor (ARM8) is augmented with an array of heterogeneous programmable units called "satellite processors" that are connected by a reconfigurable network as shown in figure 3.6.

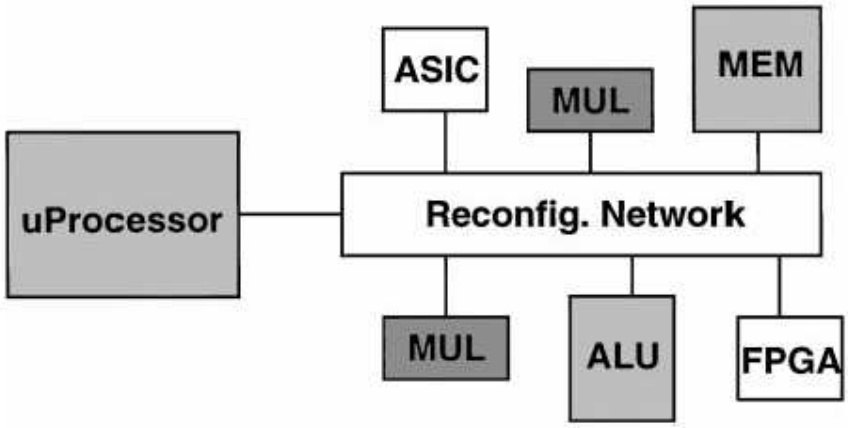

Figure 3.6: The Pleides architecture by Rabaey et al.

While the microprocessor supports the control-intensive components of the applications, repetitive and regular data-intensive loops are directly mapped onto the array of satellites by downloading some parameters and by configuring the interconnections between them. A typical satellite processor in this approach would be a multiply-accumulate unit, a memory or an address generator, and the configuration of the satellite processors corresponds to wiring up a dedicated data-flow circuit. To accommodate the need for non-numerical computations the chip also

has a block of traditional fine-grain FPGA logic. Because the communication rate between the satellite processors is rather high - typically close to the clock rate, the interconnection network is highly optimized, exploiting low-swing full-custom circuitry [94]. In this respect our approach is different (see chapter 5): our stored-program instruction set processors keep data structures and operator modules local, and the inter processor communication typically occur at a low rate close to the sampling rate.

3.3 HW/SW Co-design

Another line of research that relates to our work is the work on application specific instruction set processor (ASIP) synthesis. In the literature, two main approaches have been reported to solve the synthesis problem. Given a set of algorithms,

- Synthesize an ASIP from scratch [21]. The design process begins with a thorough examination of the application set to determine the hardware and the instruction set that will best implement the algorithms.
- Use a template DSP core architecture and remove instructions in an iterative process called *instruction subsetting* [24]. This approach aims to customize an existing processor to the application set.

Cousin et. al., [21] proposes a multi-algorithm synthesis technique as designing an ASIP from a customized ASIC. The input to their high-level synthesis system, *Breizh Synthesis System* in order to synthesize an ASIC, is a behavioral specification of a DSP algorithm with a timing constraint. BSS generates a register-register architecture model at the RT level. This structural model consists of a processing unit (PU), of a memory unit (MU) and of a finite state machine (FSM). The synthesis process of an ASIP is an incremental process that examines the behavioral descriptions of each algorithm within the application domain. The FSM of each synthesized algorithm is further studied to extract parallel elementary operations that could form complex instructions. The ASIP design eventually will have an instruction decoder instead of the single FSMs. The synthesis system is coupled with high-level power estimation to enable early design space exploration.

The authors of [24] propose instruction subsetting as a means of reducing power consumption. Instruction subsetting is defined as creating an ASIP from a more general DSP processor by removing unneeded instructions and resources. Unlike [24], we consider a bottom-up approach to our mini-core (specialized instruction set processor) designs where instruction set design and datapath customization is an integrated process. The end mini-core design may look significantly different from a general purpose DSP processor. For instance the AMAC

(add-multiply-accumulate), DMDA (dual-multiply-dual-add) units of the FIR and IIR mini-core designs that will be discussed later, as well as the special register file design for the IIR mini-core are custom made functional units that are not common in a general purpose DSP processor. We also aim at reducing the instruction count for a given task in order to reduce excess power related to instruction fetching and interpretation whereas the technique proposed by [24] may result with higher instruction counts for an ASIP design than a general purpose DSP. This is accepted in [24] provided that the improvements in the speed of the hardware by using only simple instructions compensate for any increase in the instruction count of programs. This is in contrast to our goal of providing a slow clock frequency for mini-cores in order to achieve low power.

Another way of determining an application specific instruction set is proposed by Despain et. al. [41]. Their design automation system (ASIA, Automatic Synthesis of Instruction-set Architectures) synthesizes instruction sets from application benchmarks. The benchmarks are represented as control/data flow graphs of micro-operations (MOP) such as addition, subtraction etc. The MOPs are scheduled into time steps subject to constraints of dependencies, hardware resources and instruction word length. Instructions are formed during the scheduling phase. An objective function of cycle counts and instruction set size is used to guide the design process. Simulated annealing algorithm is used to solve for the schedules. The target architecture is a pipelined datapath. However, an important limitation of the approach is for the designers to manually specify the number of hardware resources, which may take several iterations to find the best hardware allocation.

The desire for automating an ASIP design also invokes researchers to devise a retargetable framework that will enable software and hardware designers to concurrently perform design space exploration. A processor description that will cover the instruction-set, behavioral, and timing models of the hardware is needed to provide all essential information for the generation of software tools (compiler, assembler, linker, and simulator). Moreover the description should also cover micro-architectural details to enable generation of HDL code for the modeled processor. Such a design framework based on a machine description language (LISA) is proposed by Hoffmann et. al. [39].

3.4 Summary

As power has become an important design metric with the advent of portable computing, hardwired ASICs has been the designer's choice for most of these applications. However, the need to incorporate flexibility because of constantly evolving standards has urged the academia and the industry to find a decent compromise

between power and flexibility.

In this chapter we have looked at 3 different ways of designing flexible and low power/high performance systems.

First, we have examined domain-specific programmable DSP architectures. These architectures typically resemble a general purpose DSP architecture with some special hardware units and/or instructions that match a certain application domain efficiently in terms of instruction clock cycles and energy consumption.

Because of well-known capabilities in matching signal processing applications, reconfigurable computing is also heavily being investigated. Recently FPGA architectures have been shifting from fine-grained bit-level programmability to coarse-grained structures, where a configurable logic block (CLB) can be as complex as a multiplier, memory or even a processor core. As DSP applications are dominated by arithmetic operations, this shift in hardware proves to be advantageous in terms of algorithm performance. There has also been some reported work in literature that hints improved energy consumption figures compared to programmable DSPs.

Last, but not the least, automated design methodologies are also attracting some interest. Retargetable design frameworks, techniques/algorithms for creating application-specific instruction sets, stripping off a DSP architecture by removing unnecessary resources are among the several approaches we have studied. Work in this area is mostly about providing the necessary tools for early design exploration, and reducing time-to-market in creating ASIP based SoC designs.

Chapter 4

Algorithm Suite for Hearing Aids

This chapter starts with an introduction of the hearing aid algorithm studied previously [8]. Even though the algorithm is old, it provides a basic understanding of the hearing aid functionality. It also gives some insight into application domain characteristics, such as sampling rate, communication requirements, computationally demanding parts etc. This is explained in section 4.1.

Section 4.2 will present the aim of this algorithm study, whereas the rest of the chapter will describe the set of algorithms that are of interest for hearing aid applications. Theory for each algorithm will be given, and the emphasis will be mainly on implementation of these algorithms. For this purpose at the end of each algorithm description, ideas regarding its implementation will be presented.

4.1 An example application: DigiFocus algorithm

The DigiFocus algorithm is the backbone of the signal processing circuitry used in the DigiFocus hearing aid manufactured by Oticon A/S.

The human ear is able to perform sophisticated signal processing on incoming signals. For example, there are auditory systems for making sense of target signals despite noisy environments. The ear can measure fine frequency and intensity differences. It is commonly agreed that the auditory nerve processes sound tonotopically; that is by having different nerve bundles be sensitive to different frequencies [16]. This notion of the auditory system as a sophisticated filter bank is the basis of research in audiology.

The task of a hearing aid is to improve speech intelligibility for the user, while maintaining user comfort. The residual auditory area varies greatly among people with hearing losses. Therefore a need for an arbitrary and instantaneous adjustable frequency response exists.

This makes it necessary to customize the frequency response of each hearing aid to its user. The hearing aid algorithm consists of three main parts as shown in figure 4.1.

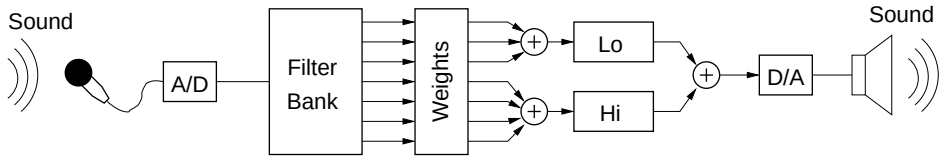

Figure 4.1: Overview of the DigiFocus algorithm

The filter bank splits the input signal into seven frequency bands. These are weighed individually in the attenuation block and merged into two frequency bands. Finally, these two signals go through additional signal processing in the compressors, namely HF and LF (high and low frequency). The filter bank constitutes about half of the signal processing circuitry in the hearing aid. As illustrated in figure 4.2, it consists of a tree structure of complementary interpolated linear phase FIR filters [59].

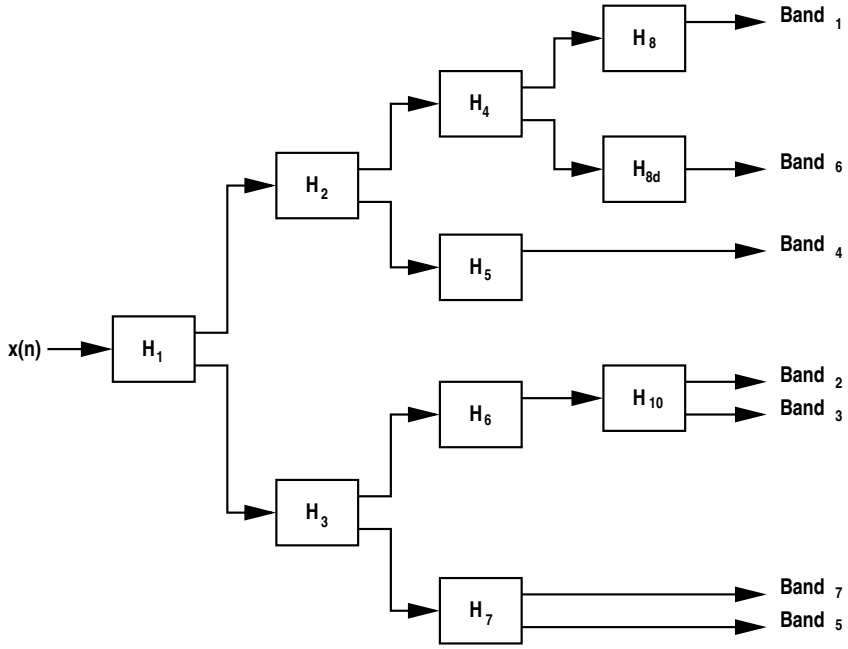

Figure 4.2: Filter bank

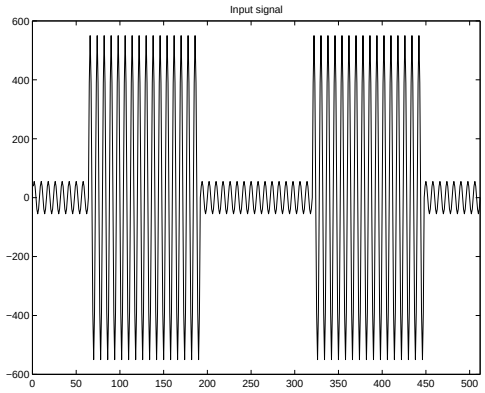

Figure 4.3: Input sine wave.

The tasks of the HF and LF compressors are to reduce the dynamic range of the signal. The outputs from the compressors are more comfortable for the user as large amplitudes are attenuated and small amplitudes are amplified.

The operation of the algorithm can be best explained with an I/O behaviour. For this purpose, the algorithm has been coded in matlab. Figures 4.3 and 4.4 illustrate how it works. Figure 4.3 shows the input signal that is a 2 KHz sine wave whose amplitude changes rapidly. We assume no attenuation of the sub-bands.

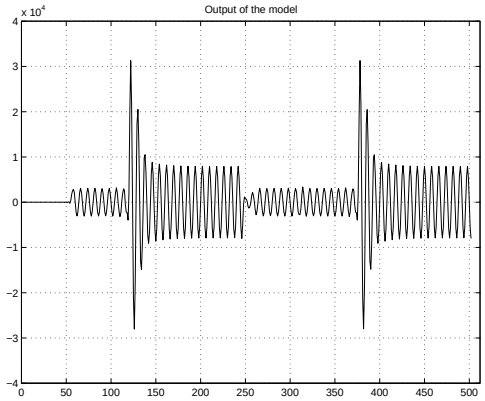

Figure 4.4: Output of the hearing aid.

Figure 4.4 shows the response of the algorithm to the input sine wave. When the level of the input signal increases suddenly, a time duration corresponding to the *attack time* elapses for the *compressors* to compress the dynamic range of the input signal by attenuating it. When this situation is reversed, i.e. when the input signal

decreases suddenly, a time duration corresponding to the *release time* elapses for the compressor to expand the signal into its normal shape.

4.2 Motivation for algorithm study

As our intention is to devise a programmable platform, during the algorithm study phase we will aim for a specialized stored instruction set processor for each algorithm. Therefore, we will analyze algorithms for features that could lead to efficient low power processor implementations. However, we will not discuss the overall system architecture, nor will we answer questions regarding how these processors can be united or how the actual processors should be implemented.

The basic idea behind low power processor design is to reduce the number of basic steps and clock cycles for the execution of a given task [74]. In a programmable processor this corresponds to reducing the number of instructions in a task, as well as the clock cycles per instruction (CPI).

A related issue in programmable architectures is the excess power consumption related to fetching and decoding instructions on a complex, general-purpose datapath. For example, the processing core of the TMS320C5x family of general-purpose DSPs from Texas Instruments draws a total current of 55mA from a 5V supply when executing a typical DSP application program with a 20% mix of multiply-accumulate operations [43]. Instruction fetching and decoding is responsible for 42 mA, i.e., 76%, of the total supply current.

Reducing the CPI is an architectural decision and largely depends on the implementation of the processor. Chapter 6 will talk about the implementation of these processors. However we can still look at how to reduce the instruction count for an algorithm within the application domain. This will help us to devise an ASIP for each algorithm suite.

There are two ways to reduce the instruction count of an application,

- Create powerful instructions that do more work in a single clock cycle than an ordinary add, sub instruction, which exist on any programmable DSP. An add-multiply-accumulate instruction is such a powerful instruction that will be presented in section 4.3.1.

- Incorporate a loop cache buffer that stores the decoded control signals of the inner loop instructions and avoids referring to the instruction memory for executing the remaining iterations of a loop. A similar approach to a loop cache buffer for reducing the number of instruction fetch and decodes is to use vector instructions that work on a stream of data. A vector instruction will be decoded for once and the computation on the operand vectors will

take as many clock cycles as required, preventing the need to fetch and decode the same scalar instruction/operation for each element of the source vector operands.

As it will be clear later in the chapter, most DSP algorithms within this application domain exhibit loop behaviour, which comes in two flavors:

- algorithms that consist of repeated computational patterns that do more work on input data. For example a symmetric FIR filter that consists of add-multiply-accumulate operations.
- algorithms that involve operations performed on a stream of data. These algorithms can be formulated as a collection of vector operations, such as addition, subtraction performed on vector operands.

Fortunately, the properties of the algorithms given above are willing to comply with the techniques for reducing the instruction count/basic steps of a DSP algorithm within the application domain. Furthermore these algorithms are loop intensive, meaning that it is quite likely that time spent in executing the inner loop computation will be dominant. Therefore we will be optimizing for the common case when we are focusing on the loop behaviour.

4.3 Filter algorithms

The term *filter* is often used to describe a device in the form of physical hardware or software that is applied to a noisy set of data in order to extract information about a prescribed quantity of interest.

In general linear time-invariant discrete-time filters are characterized by the general linear constant coefficient difference equation given in (4.1).

$$y(n) = -\sum_{k=1}^{N} a_k \cdot y(n-k) + \sum_{k=0}^{M} b_k \cdot x(n-k) \tag{4.1}$$

Filters are divided into two categories according to their impulse response, those that have a finite-duration impulse response (FIR) and those that have an infinite-duration impulse response (IIR).

4.3.1 Finite Impulse Response filters

In general an FIR filter is described by the difference equation given in (4.2).

$$y(n) = \sum_{k=0}^{M} b_k \cdot x(n-k) \tag{4.2}$$

Such a difference equation can be implemented using a *transversal* filter [81] as shown in figure 4.5. The transversal filter, which is also referred to as a *tapped-delay line* filter, consists of three basic elements: (1) *unit-delay element*, (2) *multiplier*, and (3) *adder*.

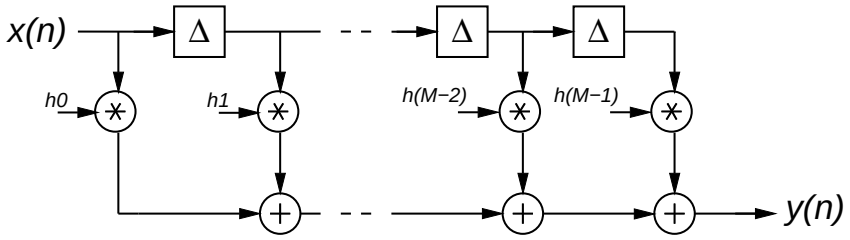

Figure 4.5: Transversal filter.

The number of delay elements used in the filter determines the finite duration of its impulse response. The number of delay elements, shown as M in figure 4.5 is commonly referred as the *filter order*. The role of each multiplier in the filter is to multiply the *tap input* by a filter coefficient called *tap weight*.

Linear Phase FIR filters

Because of its linear phase response, FIR filters are extensively used in existing DSP applications. For an FIR filter to have linear phase, its impulse response should satisfy the following symmetry (+), asymmetry (-) condition.

$$h(n) = \pm h(M-1-n) \quad n = 0, 1, ..., M-1 \tag{4.3}$$

M is the order of the FIR filter. The condition in (4.3) combined with interpolation leads to efficient implementation of narrowband linear phase FIR filters as shown in figure 4.6.

Interpolation requires filling zeros to the impulse response of the FIR filter, therefore only a small number of the taps exist in these filters. The condition in (4.3) can be exploited to reduce the number of multiplications by folding the FIR filter as in figure 4.6. Using the above condition, we can rewrite equation (4.2), as in equation (4.4) provided that M is even,

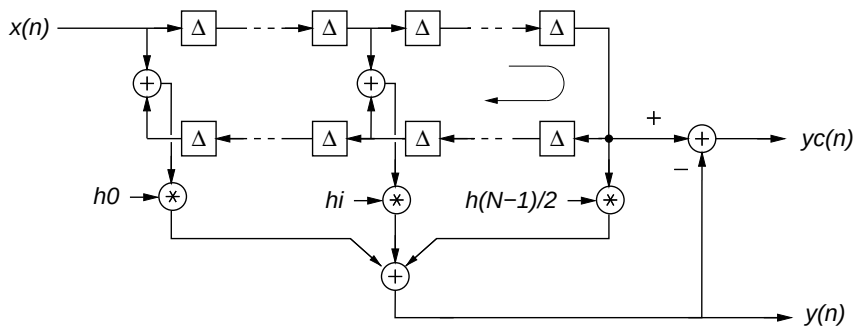

Figure 4.6: Interpolated symmetric FIR filters used in the hearing aids.

$$y(n) = \sum_{k=0}^{M/2-1} h(k)[x(n-k) \pm x(n-M+1-k)] \tag{4.4}$$

Ideas on implementing an FIR processor

A low power programmable processor core that will handle the FIR filters described above should have an add/subtract-multiply-accumulate (AMAC) unit as one big combinational unit coupled with an accumulator register. This requirement stems from the symmetry/asymmetry condition of folded FIR filters. Having an AMAC unit, provides several advantages in terms of reducing power consumption: The entire operation sequence $add \rightarrow multiply \rightarrow add$ is done in a single step and controlled by a single instruction. This reduces a significant amount of the effort required in instruction fetching , decoding, and controlling the datapath, if the processor core were to implement this operation sequence with separate instructions, such as add, multiply and add. The AMAC unit also avoids temporary values to be written/read to/from the register file, thus preventing excessive data movement. These temporary variables are mapped to the wires connecting the adder, multiplier and the accumulator of the AMAC unit.

In order to fully utilize the AMAC unit, we need to feed this unit with two data values and a coefficient simultaneously. This implies a dual-port data memory, which holds delay line elements of the FIR filter, and a coefficient memory.

Addressing the data memory is a challenge. We should have a data address generation unit that should support circular buffering technique. This technique is used to implement address pointer wraparound and therefore allows shifting the delay line of an FIR filter in a power efficient way. For instance when a new input is shifted into the delay line, it is replaced with the “oldest” delay element and

the address pointer is incremented by one, now pointing to the new “oldest” delay element. So instead of shifting all the delay elements, we modify the pointer to get the same effect. When the address pointer reaches the end of the delay line buffer, it is automatically wrapped around to the beginning of the same buffer. However, each modification to the address pointer should be checked if the address pointer is still within the bounds that specify the start and end of the delay line buffer. An alternative way to implement FIR delay line would be to cascade registers and form a shift register block with M registers, where M is the filter order. But this implementation will suffer from excessive switching activity while shifting all the delay elements. This method may only make sense if the order of the filter or equivalently the number of registers to be cascaded is relatively small. However FIR filters often tend to be deep and favor the former implementation.

In order to implement the general form of the FIR filtering equation (4.2), FIR processor should have a multiply-accumulate unit and a multiply-accumulate instruction. This instruction coupled with circular addressing will be able to support FIR filtering algorithms in general. If our FIR processor also supports addition, subtraction then we could basically handle various FIR filters with various topologies and hence map the filter bank of the hearing aid to this unit. A small, simple instruction set with specialized powerful instructions will require few instructions to encode FIR filters, and will be easy to implement.

4.3.2 Infinite Impulse Response filters

IIR systems are described by the general difference equation given in (4.1). By means of the z-transform, systems described by (4.1) are also characterized by the rational system function $H(z) = \frac{Y(z)}{X(z)}$ as in (4.5).

$$H(z) = \frac{\sum_{k=0}^{M} b_k \cdot z^{-k}}{1 + \sum_{k=1}^{N} a_k \cdot z^{-k}} \tag{4.5}$$

Direct Form Realization

Equation (4.5) can be seen as two filters in cascade, that is, $H(z) = H_1(z) \cdot H_2(z)$ where $H_1(z)$ and $H_2(z)$ are given in 4.6 and 4.7 respectively.

$$H_1(z) = \sum_{k=0}^{M} b_k \cdot z^{-k} \tag{4.6}$$

$$H_2(z) = \frac{1}{1 + \sum_{k=1}^{N} a_k \cdot z^{-k}} \tag{4.7}$$

$H_1(z)$ is an FIR filter and contains all zeros of $H(z)$. $H_2(z)$ is an IIR filter and contains all poles of $H(z)$. There are two different direct-form realizations, characterized by whether $H_1(z)$ precedes $H_2(z)$, or vice versa.

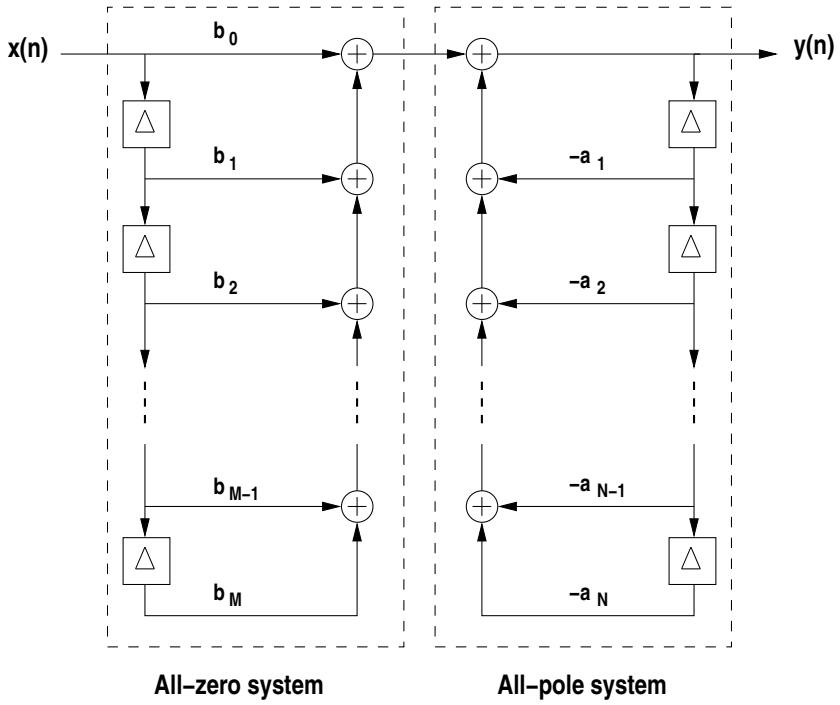

Figure 4.7: Direct form I realization.

Figure 4.7 shows the "direct form I" realization where $H_1(z)$ precedes $H_2(z)$. Another way of implementing $H(z)$ is to let $H_2(z)$ precede $H_1(z)$. This structure shown in figure 4.8 is more compact in terms of required memory locations. It is called "direct form II" realization.

Unfortunately both direct form realizations are extremely sensitive to parameter quantization, and are not recommended in practical applications. When the order of the IIR filter, N is large, a small change in the filter coefficient due to parameter quantization results in a large change in the locations of the poles and zeros of the IIR filter. Therefore this sensitivity becomes more prominent, as N

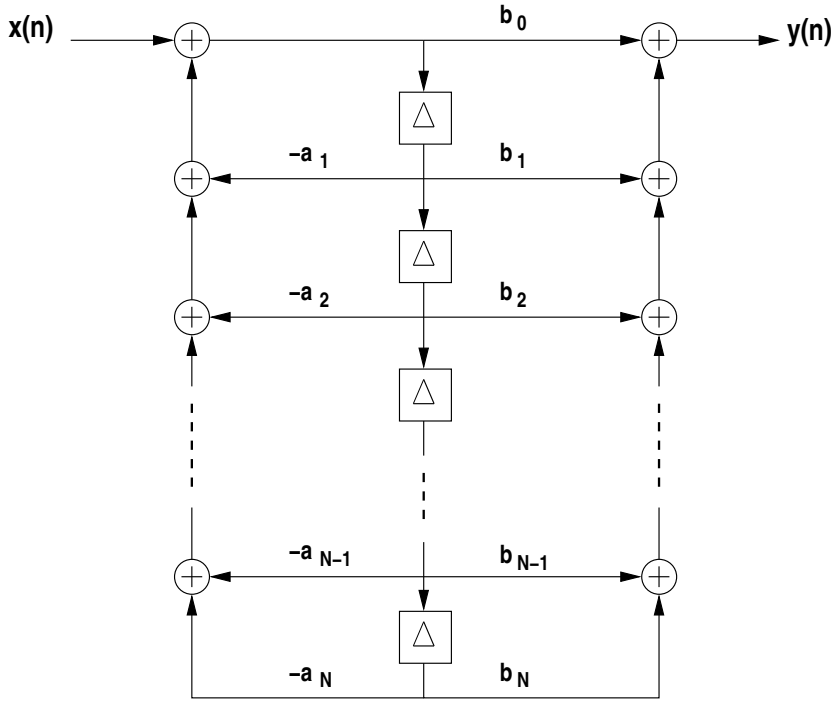

Figure 4.8: Direct form II realization (N=M).

increases. To alleviate this problem, higher order IIR filters are realized by a serial and/or parallel combination of low order IIR filters. A typical IIR filter used for this purpose is of second order and called a "biquad" [75]. If implemented in direct form II, a "biquad" can be expressed as in (4.8), and (4.9). As a biquad is realized in direct form II, one assumes $N = 2$ in figure 4.8.

$$w(n) = x(n) - a_1 \cdot w(n-1) - a_2 \cdot w(n-2) \tag{4.8}$$

$$y(n) = w(n) + b_1 \cdot w(n-1) + b_2 \cdot w(n-2) \tag{4.9}$$

In equations (4.8) and (4.9), $x(n), y(n), w(n)$ correspond to the filter input, output and the intermediate variables stored in the delay line of figure 4.8, respectively.

Advantages

IIR filters have some advantages over FIR filters:

- IIR filters require less memory and fewer instructions to implement a specified transfer function than FIR filters.
- IIR filters possess both poles and zeros whereas FIR filters are only made up of zeros. The poles give IIR filters an ability to realize transfer functions that FIR filters cannot.

The improved performance over FIR filters come at the expense of the following: (1) IIR filters are not necessarily stable. It is the designers' task to ensure stability. (2) Overflow must be considered. IIR filters are implemented with a sum of products operation that is based on an infinite sum. This construct can produce results that exceeds the maximum value represented by the processor.

Ideas on implementing an IIR processor

If we examine the biquad equations (4.8) and (4.9) the basic operation sequence is the same in both equations. The operations are two simultaneous multiplications followed by two additions. The first computed value $w(n)$, is fed back to the delay line and also used in the second equation to compute the output value, $y(n)$. If our IIR processor had a single multiplier and an adder, we need to store six temporary variables in order to compute the output. Furthermore, the entire computation would take 8 clock cycles if each addition and multiplication took a single cycle to execute. Obviously, data movement and control required to perform these steps are costly in terms of power consumption. Instead, a big combinational unit consisting of two multipliers and two adders could compute one of the equations of (4.8) and (4.9) in a single clock cycle. This unit would effectively eliminate the need to store the unnecessary temporary variables and the control required to perform this operation sequence in several steps if we were to use a single multiplier and an adder.

Circular buffers can also be used in addressing delay elements of an IIR filter. But each biquad section consists of only two delay elements. And if we recall the discussion about circular buffers versus cascaded registers to implement delay line of FIR filters, the smaller the number of registers, would it make more sense to cascade registers instead of using complex address calculations. Furthermore the typical number of biquad sections used in current hearing aids is in the range of 6 to 10. Therefore we may afford to have a specialized register file, which has two registers cascaded for each biquad section.

We need to feed two coefficients, two delay elements, and a single data value to our huge combinational block. This requires that our specialized register file should be able to provide two delay elements in a single clock cycle, i.e., two read ports and a single write port to shift in the new $w(n)$. Our IIR processor will have a

coefficient memory providing a coefficient pair at a time, namely a_1, a_2 and b_1, b_2 pairs. The proposed datapath of the IIR processor is shown in figure 4.9.

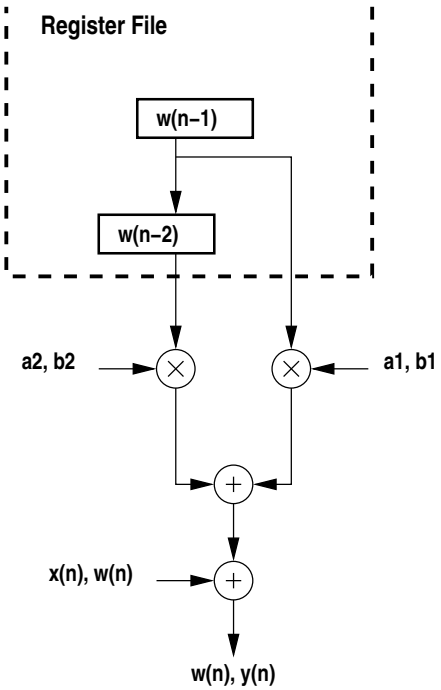

Figure 4.9: Datapath of the IIR processor. Two steps are required to perform a biquad section.

Fetching and decoding power can be reduced significantly by having a simple and small instruction set consisting of addition, subtraction and an instruction that would specify a biquad section computation. This instruction would need to specify, an input register from a general purpose register file, a biquad section delay line from the above proposed special register file and another register from the general purpose register file to specify the destination for biquad output. By addition and subtraction we could form various topologies of biquad sections, parallel or cascade, and by the help of such a biquad instruction we could actually implement a biquad section.

4.3.3 Lattice structures

Lattice filters are used extensively in digital speech processing and in the implementation of adaptive filters.

FIR Lattice Filters

The FIR lattice filter is generally described by the following set of recursive equations. K_m for $m = 1,2,...,M$ are called reflections coefficients of an M stage lattice filter. The output of the lattice filter is $f_M(n)$.

$$f_0(n) = g_0(n) = x(n) \tag{4.10}$$

$$f_m(n) = f_{m-1}(n) + K_m \cdot g_{m-1}(n-1) \quad m = 1,2,...,M \tag{4.11}$$

$$g_m(n) = K_m \cdot f_{m-1}(n) + g_{m-1}(n-1) \quad m = 1,2,...,M \tag{4.12}$$

Figure 4.10 shows FIR lattice structures. Each lattice stage in the structure implements a single step of the recursion given in (4.11) and (4.12).

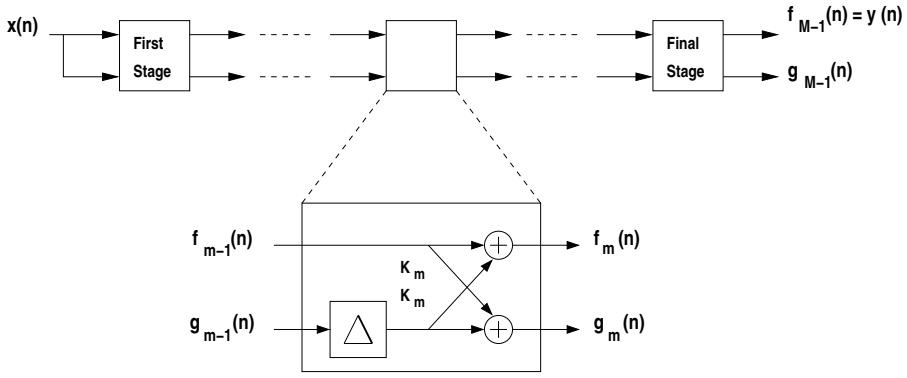

Figure 4.10: FIR lattice filters.

IIR Lattice filters

The IIR lattice filter is described by the following set of equations.

$$f_N(n) = x(n) \tag{4.13}$$

$$f_{m-1}(n) = f_m(n) - K_m \cdot g_{m-1}(n-1) \quad m = M, M-1, ..., 2, 1 \tag{4.14}$$

$$g_m(n) = K_m \cdot f_{m-1}(n) + g_{m-1}(n-1) \quad m = M, M-1, ..., 2, 1 \tag{4.15}$$

$$y(n) = f_0(n) = g_0(n) \tag{4.16}$$

Figure 4.11 shows IIR lattice structures. Each lattice stage in the structure implements a single step of the recursion given in (4.14), (4.15).

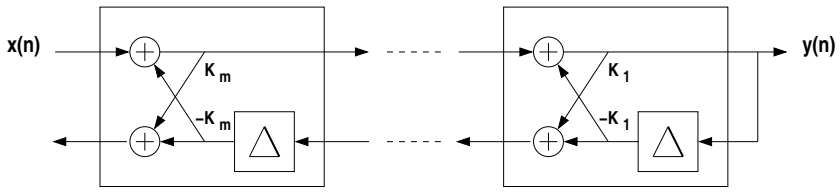

Figure 4.11: IIR lattice filters.

Ideas on implementing a Lattice processor

Once again we have a repetitive computation. This time it is a recursive computation. In FIR filtering, repetition was the add-multiply-accumulate, and multiply-accumulate operations that was performed on the delay elements of the FIR filters. In IIR filtering, a biquad section was repeated to form higher order IIR filters, so optimizing the recursive formulas given for FIR and IIR lattice structures will prove to be beneficial.

Let's start with lattice FIR structures. In order to implement the recursive relation in equations (4.11) and (4.12), we can use a combinatorial circuit consisting of two multipliers and two adders. The circuit is shown in figure 4.12 (a). This circuit will execute a single iteration of the recursion in a single step. The same arguments that were made for other combinatorial circuits in the previous sections can also be applied here. Inputs to this unit are $f_{m-1}(n)$, $g_{m-1}(n-1)$, and K_m. The outputs from this unit will be the inputs for the next lattice stage therefore they could be fed back to the same combinatorial unit again.

We need a coefficient memory to store the reflection coefficients and a data memory/register file to store delay line elements. Address generation unit do not have to be complex as the one in our FIR processor. It should only support post increment addressing. However each stage of the lattice FIR filter, involves reading its delay element and writing to the previous delay element the output from our combinational unit. Therefore we should have separate read and write address pointers and the delay line memory should support simultaneous reading and writing.

The operation sequence for a stage in a lattice IIR filter is different from than that of the FIR lattice stage even though the same number of multipliers and adders

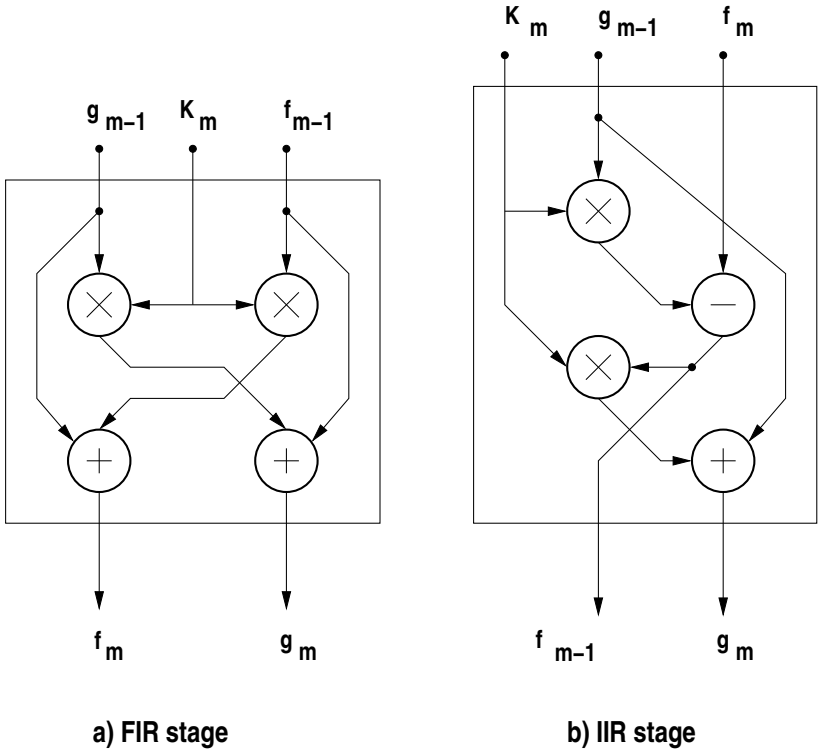

Figure 4.12: Proposed combinational circuit for: (a) a lattice FIR stage (b) for a lattice IIR stage.

can be used in both stages. A proposed combinatorial circuit that would compute equations (4.14) and (4.15) is given in figure 4.12 (b).

By using two multipliers and two adders, we can compose combinatorial circuits in figure 4.12 (a) and (b) to implement both lattice FIR and IIR structures.

The information for the Lattice processor to process either FIR or IIR lattice filters can be configurable by inserting multiplexers to figure 4.12.

4.4 Least Mean Square algorithm

The least-mean-square (LMS) algorithm is a linear adaptive filtering algorithm that consists of two basic processes:

1. A filtering process, which involves (a) computing the output of a transversal

filter produced by a set of tap inputs, and (b) generating an estimation error by comparing this output to a desired response

2. An adaptive process, which involves the automatic adjustment of the tap weights of the filter in accordance with the estimation error.

Thus the combination of these two processes working together constitutes a feedback loop around the LMS algorithm.

A significant feature of LMS algorithm is its simplicity. Indeed this feature has made LMS the standard against which other adaptive filtering algorithms are benchmarked [81].

The algorithm

Consider a transversal filter with tap inputs $u(n), u(n-1), ..., u(n-M+1)$ and a corresponding set of tap weights $w_0(n), w_1(n), ..., w_{M-1}(n)$ as shown in figure 4.13. From now on, we will use $\vec{u}(n)$ and $\vec{w}(n)$ to denote the tap input vector and weight vector corresponding to tap inputs, and tap weights as given above.

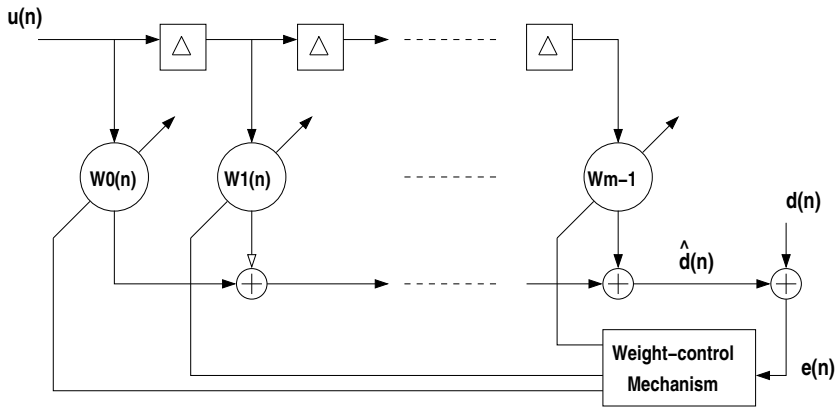

Figure 4.13: Adaptive transversal filter.

Note that coefficients of the filter are time-variant indicating the adaptive behaviour of the system. The least mean square algorithm consists of the following steps.

$$y(n) = \vec{w}^T(n) \cdot \vec{u}(n) \tag{4.17}$$

$$e(n) = d(n) - y(n) \tag{4.18}$$

$$\vec{w}(n+1) = \vec{w}(n) + \mu \cdot \vec{u}(n) \cdot e(n) \tag{4.19}$$

In (4.17) the transversal (FIR) filter output $y(n)$is computed. It is the well-known dot vector multiplication. In (4.18) the error signal $e(n)$ is computed by subtracting the actual filter output from the desired output, $d(n)$. Note that $e(n)$, $y(n)$, μ and $d(n)$ are representing scalar values unlike $\vec{w}(n)$, and $\vec{u}(n)$. These two steps constitute the first of the two processes described before. Finally the algorithm adapts the tap weight vector in equation (4.19). Here μ is called the step size parameter of the adaptation. The correction $\mu \cdot \vec{u}(n) \cdot e(n)$ applied to the tap-weight vector $\vec{w}(n)$ at iteration $n+1$ is directly proportional to the tap-input vector $\vec{u}(n)$. Therefore, when $u(n)$ is large, the LMS algorithm experiences a *gradient noise amplification* problem. To overcome this difficulty, we may use the *normalized LMS algorithm*. In particular, the correction applied to the tap-weight vector $\vec{w}(n)$ at iteration $n+1$ is "normalized" with respect to the squared Euclidean norm of the tap-input vector $u(n)$ at iteration n. Equation (4.19) should be replaced with equation (4.20) to get the normalized LMS algorithm.

$$\vec{w}(n+1) = \vec{w}(n) + \frac{\mu}{a + \|\vec{u}(n)\|} \cdot \vec{u}(n) \cdot e(n) \tag{4.20}$$

In (4.20), a is an auxiliary parameter that avoids division by zero when the input tap vector becomes zero. Computing $\|\vec{u}(n)\|$ looks complex but in practice there is a simple way of computing it using recursion. Normalized LMS introduces the requirement of division and therefore slightly increases the complexity of the LMS algorithm.

Ideas on implementing a processing unit for the LMS algorithm will be discussed after section 4.6 when we have covered both "Correlation" and "Levinson-Durbin" algorithm.

4.5 Correlation

A very important task in signal processing is to perform correlation on two signals in order to measure the degree to which these two signals are similar and thus to extract some information that depends to a large extent on the application.

Suppose that we have two real sequences $x(n)$ and $y(n)$ each of which has finite energy. The *cross-correlation* of $x(n)$ and $y(n)$ is a sequence $r_{xy}(l)$, which is defined as

$$r_{xy}(l) = \sum_{-\infty}^{\infty} x(n)y(n-l) \qquad l = 0, \pm 1, \pm 2, \ldots \tag{4.21}$$

However in practical cases, infinitely long input sequences may not be available or relevant, therefore we compute an estimate of the correlation sequence by using a finite length of input data. Let $\vec{C}_{xy}^{N}(n,m)$ denote cross correlation vector of size $m+1-$ by -1 consisting of lags $0\cdots m$. It is obtained by estimating the cross correlation of signals x and y at time n, using a finite length of N data samples from each signal. It is given in (4.22).

$$\vec{C}_{xy}^{N}(n,m) = \begin{bmatrix} \vec{C}_{xy}^{N}(n,0) \\ \vec{C}_{xy}^{N}(n,1) \\ \vec{C}_{xy}^{N}(n,2) \\ \vdots \\ \vec{C}_{xy}^{N}(n,m) \end{bmatrix} \tag{4.22}$$

We can form the vector given in (4.22) by using (4.21). This would require $m \times N$ multiplications. However, an efficient approach that requires fewer multiplications exist and is given in (4.23).

$$\vec{C}_{xy}^{N}(n,m) = \vec{C}_{xy}^{N}(n-1,m) + \frac{1}{N} \cdot \left(x(n) \cdot \begin{bmatrix} y(n) \\ y(n-1) \\ \vdots \\ y(n-m) \end{bmatrix} - x(n-N) \cdot \begin{bmatrix} y(n-N) \\ y(n-N-1) \\ \vdots \\ y(n-N-m) \end{bmatrix} \right) \tag{4.23}$$

Provided that we are given $\vec{C}_{xy}^{N}(n-1,m)$, we only need $2 \times m$ multiplications to compute $\vec{C}_{xy}^{N}(n,m)$ which is a dramatic improvement in terms of the number of multiplications required.

4.6 Levinson-Durbin algorithm

One of the most fascinating problems in time-series analysis is that of *predicting* a future value of a stationary discrete-time stochastic process, given a set of past samples of the process. In *linear prediction*, we express the predicted future value as a linear combination of the previous samples. Let's assume that $x(n)$ is predicted as a linear combination of $x(n-1), x(n-2), \ldots x(n-M)$. This operation corresponds to one-step prediction into the future, measured with respect to time $n-1$. Accordingly, we refer to this form of prediction as *one-step linear prediction in the forward direction* or simply *forward linear prediction*. In another form of prediction, we use the samples $x(n), x(n-1), \ldots x(n-M+1)$ to make a predic-

tion of the past sample $x(n-M)$. We refer to this second form of prediction as *backward linear prediction*.

Let's consider, in particular, the *one-step forward linear predictor*, which forms the prediction of the value $x(n)$ by a weighed linear combination of the past values $x(n-1), x(n-2), \ldots, x(n-p)$. Hence the linearly predicted value of $x(n)$ is

$$\hat{x}(n) = -\sum_{k=1}^{p} a_p(k) \cdot x(n-k) \tag{4.24}$$

where the $-a_p(k)$ represent the weights in the linear combination. These weights are called the *prediction coefficients* of the one-step forward linear predictor of *order p*. The difference between the value $x(n)$ and the predicted value $\hat{x}(n)$ is called the *forward prediction error*, denoted as $f_p(n)$:

$$f_p(n) = x(n) - \hat{x}(n) \tag{4.25}$$

We view linear prediction as equivalent to linear filtering where the predictor is embedded in the linear filter, as shown in figure 4.14.

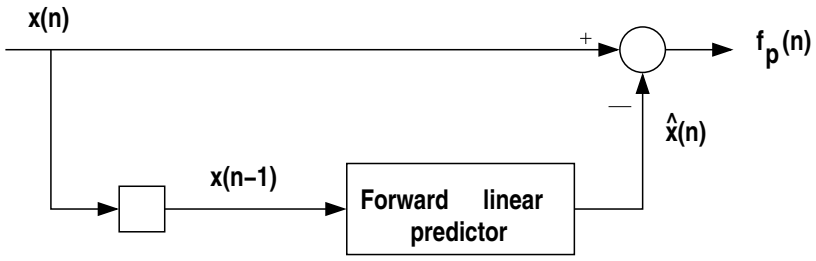

Figure 4.14: Forward linear prediction.

This is called a *prediction error filter* with input sequence $\vec{x}(n)$ and output sequence $\vec{f}_p(n)$.

The mean-square value of the forward linear prediction error $f_p(n)$ is a quadratic function of the predictor coefficients and its minimizations leads to a set of equations. The Levinson-Durbin algorithm is a computationally efficient algorithm for solving these equations. Here we will present the algorithm rather than deriving it. The interested reader is encouraged to read [81].

The algorithm

Let the $(m+1)-$by-1 vector $\vec{a}_m$ denote the tap-weight vector of a forward prediction error filter of order m. The $(m+1)-$by-1 tap-weight vector of the corre-

sponding backward prediction error filter is obtained by backward rearrangement of the elements of vector $\vec{a}_m$ and their complex conjugation. We denote the combined effect of these two operations by $\vec{a}_m^{B*}$. Let the $m-$ by -1 vectors $\vec{a}_{m-1}$ and $\vec{a}_{m-1}^{B*}$ denote the tap weight vectors of the corresponding forward and backward prediction error filters of order $m-1$, respectively. The Levinson-Durbin recursion may be stated as in (4.26) for updating tap weight vector of a forward prediction error filter.

$$\vec{a}_m = \begin{bmatrix} \vec{a}_{m-1} \\ 0 \end{bmatrix} + K_m \begin{bmatrix} 0 \\ \vec{a}_{m-1}^{B*} \end{bmatrix} \tag{4.26}$$

where K_m is a constant given in (4.27).

$$K_m = -\frac{\sum_{l=0}^{m-1} r(l-m) \cdot a_{m-1,l}}{P_{m-1}} \tag{4.27}$$

In (4.27), $r(l-m)$ and $a_{m-1,l}$ refer to the autocorrelation function of input process at a lag of $l-m$ and the lth tap weight of a forward prediction error filter of order $m-1$, respectively. P_{m-1} in the denominator represents the prediction error power of order $m-1$. A recursive relation for the order update of the prediction error power can also be derived as in (4.28).

$$P_m = P_{m-1} \cdot (1 - \mid K_m \mid^2) \tag{4.28}$$

Ideas on implementing LMS, Levinson-Durbin and Correlation

The three algorithms explained in sections 4.4, 4.5, and 4.6 have a common property: they can be written in vectorized form. This suggests designing a vector processor which is able to encode these algorithms in fewer instructions.

Such a vector processor will require a special register file consisting of vector registers. We will be able to exploit temporal and spatial locality that exists in vector operands. The datapath of our vector processor should be able to add, and subtract vectors and to perform specialized functions such as an inner product of two vectors as given in (4.17). The delay elements of an FIR filter used in the LMS algorithm could be mapped to one of the vector registers from the register file. This vector register should support circular buffering in order to implement delay line shifting efficiently. Our vector processor should also support scalar operations such as addition, subtraction, division and multiplication as well as a scalar register file.

Another specialized feature of the vector register file is to support reverse addressing the elements of a vector register. This requirement stems from Levinson-

Durbin recursion as given in (4.26). Figure 4.15 shows an interpretation of the reversed addressing of a vector register.

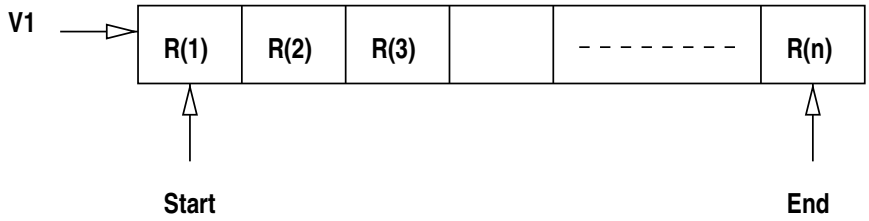

Figure 4.15: Addressing a vector register from both directions require two address registers, start and end.

During arithmetic operations on a vector register, one is able to specify whether the operand vector is to be reversed addressed or not. In the former case, the contents of the vector register are read starting from the "end" pointer. This feature requires maintaining two pointers in the register file depending on where to start reading the operand registers. Table 4.1 gives a list of the proposed instructions for a simple vector processor.

Cross-Correlation is the final algorithm that we would like to map on our vector processor. By examining (4.23), one can observe the requirement of storing $2N+m+2$ delay elements in order to compute cross-correlation coefficients of lags $0,1,\ldots,m$. This suggests a large storage medium since the number of data samples used for the estimate, N, can be large, typically in the range of 64-128. A data memory with a circular address generator could serve for this purpose.

4.7 Dynamic range control - Compression

Dynamic range control of audio signals is used in many applications to match the dynamic behaviour of the audio signal to differing requirements [87]. Figure 4.16 shows a block diagram of a system for dynamic range control.

After measuring the input level $X[dB]$, the output level $Y[dB]$ is affected by multiplying the delayed input signal $x(n)$ by a factor $g(n)$ according to (4.29).

$$y(n) = g(n) \cdot x(n-D) \tag{4.29}$$

The delay of the signal $x(n)$ compared with the control signal $g(n)$ allows predictive control of the output signal level.

Mnemonic	Explanation
addvv V1,V2,V3;	V3=V1+V2; V1,V2, and V3 are vector registers Source vector operands can be read in reversed form
subvv V1,V2,V3;	V3=V1-V2;
dvp V1,V2,R3;	R3=V1*V2: R3 is the result of the dot vector multiplication of V1 and V2. Typical multiply-accumulate loop.
mulsv R1,V2,V3;	V3=R1*V2; R1 is a scalar and multiplied by each element of the vector V2 and put into register V3.
addsv R1,V2(i),R3;	R3=R1+V2(i); V2(i) is the i'th element of vector V2 and added to a scalar R1. Result is another scalar value, R3.
movsv R1,V2(i);	V2(i)=R1; Copy a scalar to the i'th element of the vector V2;
div R1,R2,R3;	R3=R1/R2; Divide scalars.
mul R1,R2,R3;	R3=R1*R2; Multiply scalars.
add R1,R2,R3;	R3=R1*R2; Add scalars.
sub R1,R2,R3;	R3=R1*R2; Subtract scalars.

Table 4.1: The proposed instructions for a vector processor.

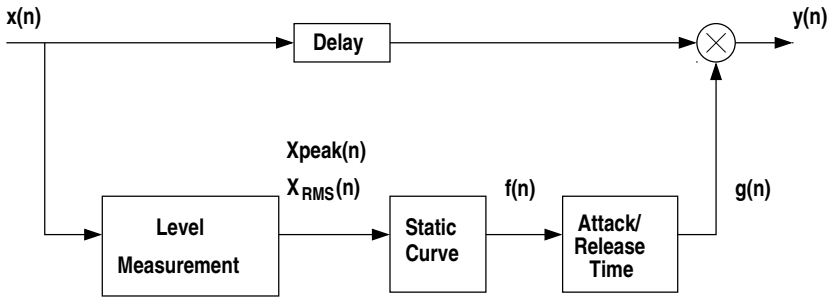

Figure 4.16: A system for dynamic range control.

Static Curve

The relationship between input level and weighting level is defined by a static level curve $G[dB] = f(X[dB])$. An example of such a static curve is given in figure 4.17

Here the output level and the weighting level are given as functions of the input level. With the help of a limiter, the output level is limited when the input level

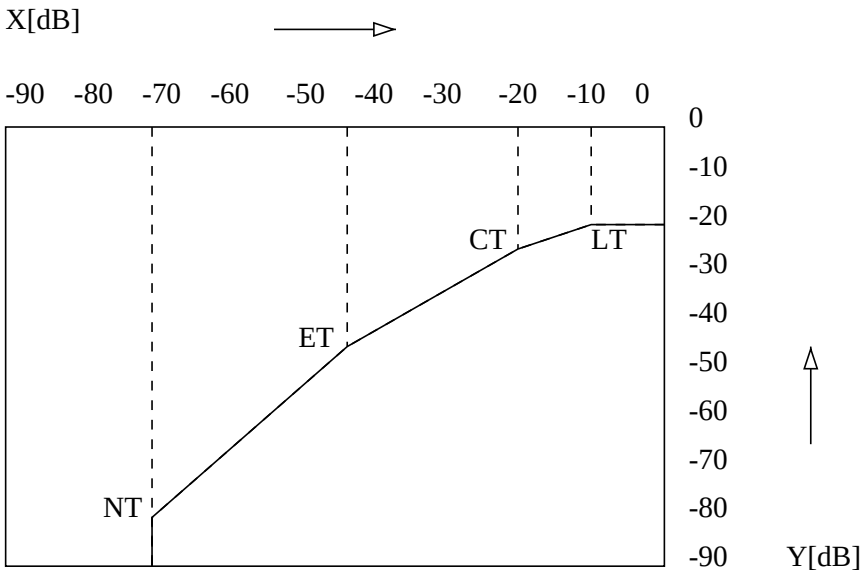

Figure 4.17: Static curve with parameters LT=Limiter threshold, CT=Compressor threshold, ET=Expander threshold and NT=Noise gate threshold.

exceeds the limiter threshold LT. All input levels above LT lead to a constant output level. The compressor maps a change of input level onto a certain smaller change of output level. In contrast to a limiter, the compressor increases the loudness of the audio signal. The expander increases changes in the input level to larger changes in the output level. With this, an increase in the dynamics for low levels is achieved. The noise gate is used to suppress low-level signals, for noise reduction. Every threshold used in particular parts of the static curve is defined as the lower limit for limiter and compressor and upper limit for expander and noise gate.

Level Measurement

Level measurements can be made with the systems shown in figures 4.18 and 4.19.

For peak measurement, the absolute value of the input is compared with the peak value $x_{peak}(n)$. If the absolute value is greater than the peak value, the difference is weighed with the coefficient AT (*attack time*) and added to $(1-RT)\cdot x_{peak}(n)$. Here RT is called *release time*. If the absolute value of the input is smaller than the peak value, the new peak value is equal to $(1-RT)\cdot x_{peak}(n)$. The difference equation for the block diagram in figure 4.18 is given by

$$x_{peak}(n) = (1-AT-RT)\cdot x_{peak}(n-1) + AT\cdot \mid x(n) \mid \qquad (4.30)$$

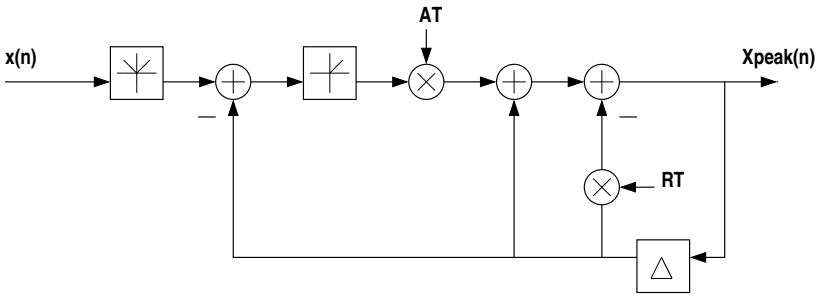

Figure 4.18: Peak measurement

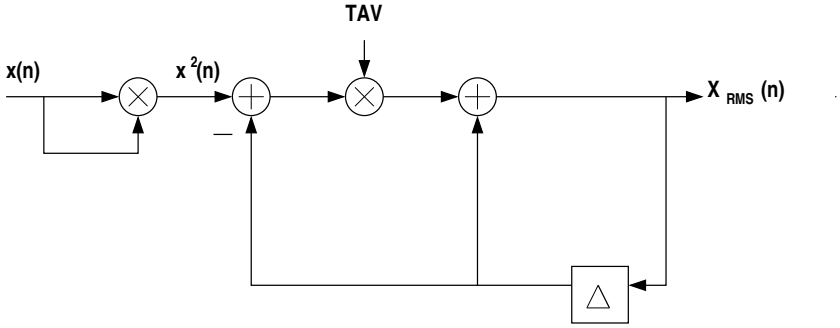

Figure 4.19: RMS measurement

The RMS measurement shown in figure 4.19, uses the square of the input and performs averaging with a first-order low-pass filter. TAV is a constant called the *averaging coefficient*. The difference equation is given by

$$x_{rms}(n) = (1 - TAV) \cdot x_{rms}(n-1) + TAV \cdot x^2(n) \tag{4.31}$$

Gain Factor Smoothing

Abrupt changes of the input typically create abrupt changes in the gain factor during compression. Therefore a mechanism that smoothens out this effect is needed. When the input signal increases abruptly, a time duration called *attack time* elapses until the actual increased gain factor is used. Likewise, when the input signal decreases suddenly, *release time* elapses until the actual gain factor is used. The effects of attack and release time can be seen in figure 4.4.

Attack and release times can be implemented by the system shown in figure 4.20. The attack coefficient AT or release coefficient RT is obtained by comparing

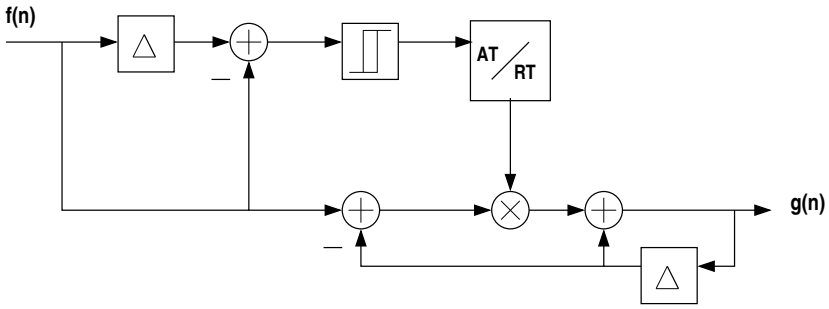

Figure 4.20: Implementing attack and release time.

the input control factor and the previous one. A small hysteresis curve determines whether the control factor is in the attack or release status and hence gives the coefficient AT or RT. The difference equation is given by

$$g(n) = (1-k) \cdot g(n-1) + k \cdot f(n) \tag{4.32}$$

Ideas on implementing Compressors

Compression involves two phases: (1) measuring the input signal level, (2) computing a gain factor to be used for shaping the output signal. A programmable compressor requires a gain table memory to store the gain factors and should support arithmetic operations such as addition, multiplication and subtraction. The only specialty encountered in the compression algorithm is the way to address gain factors. In [8], the compression algorithm uses the exponent of the input signal level to address the gain table. Therefore an exponent extraction instruction/operation should be supported.

4.8 Non-linear functions

Non-linear functions such as tanh, sine, cosine and square root are a part of the algorithm domain. Since they are not used so frequently, they could be mapped onto a general purpose DSP that is a part of the overall architecture.

4.9 Summary

In this chapter, we have analyzed all the algorithms and proposed a special low power processor implementation for each of them. However, with respect to com-

putational requirements of a hearing aid system, some of these algorithms (LMS, Levinson-Durbin etc.) in this algorithm suite are less demanding [6]. Therefore it is important to realize this distinction between these algorithms, as we could map the less-demanding algorithms onto a low power standard DSP core instead of actually spending design effort for a special processor implementation.

The basic idea behind low power processor design for demanding algorithms such as FIR, IIR, etc. is to reduce the number of basic steps and clock cycles for the execution of that particular task/algorithm. Therefore we tried to identify powerful instructions that would reduce the instruction count, hence execution time of an application comprised of any algorithm from the given set. Reducing instruction count and code size has two impacts on power consumption.

- Reduced instruction count means fewer instructions to fetch and decode and a smaller execution time for a given program.
- Reduced code size implies use of smaller instruction memory that in turn means low power dissipation.

In order to make effective use of such instructions, combinational circuits that perform the core computation of loops encountered in these algorithms are needed. The add-multiply-accumulate circuit introduced in section 4.3 that can compute one tap of a symmetric linear phase FIR filter in one clock cycle is one such example.

Chapter 5

A Heterogeneous Multiprocessor Architecture

Chapter 4 has shown that most DSP algorithms are compute-intensive and require various multiply-accumulate operations and special data fetching/addressing capabilities. This chapter builds on this observation and introduces a heterogeneous multiprocessor template architecture as a low power and programmable platform to be used in system-on-chip designs that target audio signal processing. Section 5.1 presents the idea and discusses advantages of this approach. Section 5.2 gives insight into the design of the small instruction-set-processors called mini-cores, which constitute the basic building blocks for a SoC based design flow. The inter-processor communication model used in the architecture is explained in section 5.3. Section 5.4 discusses the interconnect and section 5.5 deals with initialization and configuration of the system. Section 5.6 discusses a possible mapping of the hearing aid application to the mini-core platform. Finally 5.7 summarizes this chapter.

5.1 A heterogeneous multiprocessor

5.1.1 The idea

The design of an audio signal processing application (as for example a hearing aid) usually starts with a specification in Matlab – often in the form of a complex Simulink data-flow structure of filters and other signal processing blocks that communicate at the sampling rate: FIR, IIR, N-LMS, Viterbi, FFT, etc. The idea is to provide a platform composed of simple instruction set processors called mini-cores each optimized for one of these classes of algorithms as well as DSP- and/or mi-

croprocessor cores running less demanding irregular and/or control oriented parts of an application. Furthermore, communication is provided by an interconnection network of any topology depending on the application requirements (Bus, Torus etc.) that supports message passing among the processors as shown in figure 5.1.

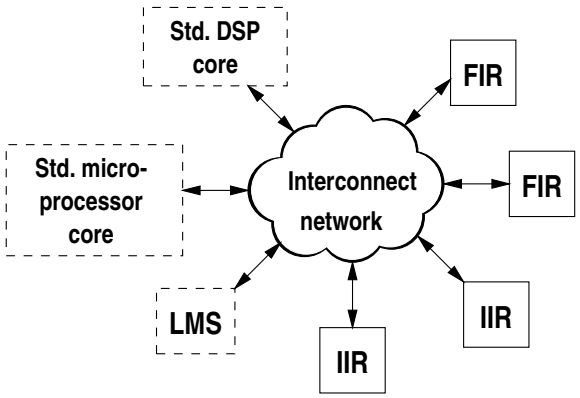

Figure 5.1: Example of a mini-core system architecture.

5.1.2 Flexibility and low-power

The programmability of each mini-core is confined to a single class of algorithm. This makes the single mini-core design, an energy-efficient compact processor compared to a general-purpose DSP for that particular algorithm. A mini-core can only execute programs within that particular algorithm – hence the term mini-core – whereas the general-purpose DSP can do all types of signal processing algorithms. Here programmability of a single mini-core is compromised for energy-efficiency. For instance an FIR mini-core for executing FIR algorithms (ordinary, symmetric, interpolated etc.), and an IIR mini-core for executing IIR algorithms. However the architecture will provide further flexibility through a multitude of different mini-cores as well as the addition of DSP and RISC cores.

Furthermore, within the audio signal-processing domain – especially hearing aids – the communication requirements between these mini-cores are moderate. The combination of such mini-cores with a simple network that accommodates low-rate data communication, hints to an energy-efficient and – at the same time – programmable architecture.

Moreover, such an architecture is inherently modular: It is a simple task to add new mini-cores, and the message passing approach to communication, makes it a simple task to fit in general purpose microprocessor- and/or DSP cores as well. To maintain the energy-efficiency of the architecture however, these general-purpose

processors are intended to run: (1) irregular tasks that do not require excessive compute power or (2) sub-sample rate signal processing algorithms that need to be executed at a slower pace. As there are advanced adaptive signal processing algorithms coming along, the need for sub-sample rate complex signal processing algorithms are increasing, justifying the use of a DSP-core for only those parts of an application.

5.1.3 Design methodology

Designing a mini-core based platform for a given application involves instantiating different mini-cores as well as different versions of some of the mini-cores. By introducing a well-defined communication protocol between a single mini-core and the interconnect network, we also enable concurrent design of different mini-cores and a variety of interconnect topologies. This means that the SoC designer can select desired mini-cores and the interconnect network from a library of components and reduce the design time-to-market by simply instantiating these components in a top level SoC based design.

To enable this, a traditional synthesis-based ASIC design flow can be used, where (parameterized) VHDL descriptions of the different mini-cores are mapped into netlists of standard cells. This soft-macro approach has further advantages: (1) it allows the integration of other proprietary circuits on the same chip, and (2) the implementation is foundry independent. This approach is part of the baseline for the project; therefore custom circuitry is not used.

The general trend in design of portable battery powered applications is that low power consumption is the main concern, while area and speed are less of a concern. The proposed architecture is in line with this. It consists of a multitude of relatively small dedicated mini-cores. The mini-cores may not be active all the time and if the same platform is used in different products there may even be unused mini-cores. For this reason, the mini-cores are designed to have zero dynamic power consumption when idle.

The leakage power consumption in the technology used to fabricate the test chip presented later in chapter 8 is less than 5 % of the dynamic power consumption and it can be neglected [6]. For future technologies static power consumption due to leakage is becoming more of a problem [88]. It is obviously important to minimize such power consumption when mini-cores are idle or not used at all. Here some of the many leakage reduction techniques based on V_{th} adjustments that have been published [50, 9, 92] may be applied.

Finally a word about the applicability of the mini-core approach beyond the specific application studied in this thesis. The mini core architecture obviously favors applications that consist of compute intensive processes with moderate

amounts of inter process communication. Many DSP dominated applications (other than hearing aids) have these characteristics. In addition to the characteristics of the application itself, the partitioning and mapping of processes onto processors (i.e. mini cores) also affects the amount of inter processor communication. These are tradeoffs that the system designer should consider when mapping an application to a mini-core based platform.

The mini-core architecture is obviously less suited for irregular control dominated applications and for applications that are dominated by (global) communication. Here other solutions are needed. These may well co-exist with mini-cores – the overall system architecture that is proposed here is a heterogeneous multiprocessor built from mini-cores and a range of other processor cores, figure 5.1.

5.2 Mini-core design philosophy

As the energy-efficiency of the architecture depends on mapping the dominant compute intensive parts of an application onto the mini-cores, the success of this approach very much depends on the existence of low power programmable mini-core designs.

Programmability can actually come at different levels of granularity, and each has its own preferred and optimal application domain. Figure 5.2 shows some architectures with different levels of programmability.

The first approach is a stored-instruction set processor. Instructions are stored in a memory, fetched and decoded by a decoder and finally executed on a customized datapath. The datapath may contain similarities to a programmable DSP datapath and yet be specialized in terms of arithmetic computational units and various data addressing modes. At this level of granularity, an instruction dynamically changes the behaviour of an otherwise statically connected datapath.

Going to a finer level granularity, we see programmability at the RT level. The datapath of the second approach can be *configured* to implement various algorithms within an application domain, by wiring up specific data flow architectures using datapath components ALUs, multipliers and multiplexers. Mapping a complex algorithm to such an architecture may require configuration of the circuit multiple times during one iteration of the algorithm. This is referred to as reconfiguration on the fly. This will typically be expensive in terms of power consumption and speed (waste of system clock cycles that is not related to actual computation). However, time multiplexing of such an architecture can be avoided by having enough configurable blocks in the system.

The final architecture is the most fine-grained approach. Here programmability is provided at the gate level inside the reconfigurable logic blocks (CLBs) as

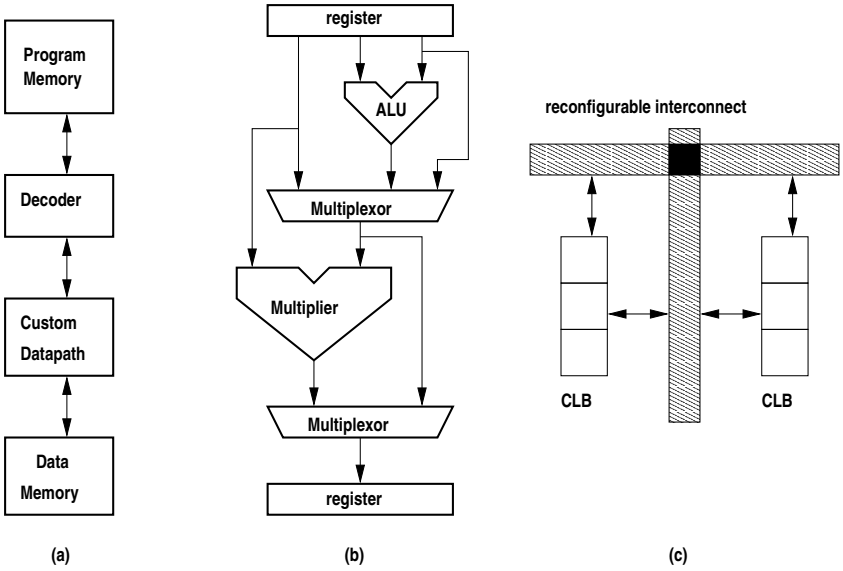

Figure 5.2: Architectures with different levels of programmability. (a) Stored-instruction processor (b) Reconfigurable datapath (c) Fine-grain reconfigurable logic found in conventional FPGAs. CLB:Configurable Logic block

well as the reconfigurable interconnect. *Programming* this architecture is basically configuring a dedicated data flow circuit for the target application. This approach is the most flexible one. Bit-level programmability gives it an advantage compared to stored instruction set processors as they can match irregular word lengths more efficiently. This added flexibility comes at the expense of very general interconnect structure that is costly in terms of area, power consumption and speed.

The selected mini-core design philosophy is similar to the first architecture of figure 5.2. With low power in mind, a simple small instruction set for each mini-core has been devised. This also meant customizing the datapath of each mini-core for a specific algorithm domain. The advantages of a specialized instruction set and customized datapath were: (1) a reduced instruction count for a signal processing task compared to a programmable DSP, and (2) low fetch/decode overhead for executing a single instruction on a mini-core compared to a programmable DSP.

RT-level coarse-grained granularity (second approach) is also becoming popular nowadays in the *domain-specific computing* field [55, 20] as discussed in chapter 3. Likewise, state-of-the-art FPGA technology is evolving towards heterogeneous coarse-grained logic blocks such as multipliers, memories, larger CLBs (configurable logic blocks), even processor cores [7]. Thus the distinction between

heterogeneous reconfigurable system-on-chip designs and FPGAs is getting blurry each day.

The mini-core designs that will be presented in the next chapter include generic-parameters. They are easy to mould according to requirements and are wrapped up with the same communication protocol. Hence each mini-core fits nicely into the modular architecture. Furthermore the effort that is required to design a mini-core is very small compared to a general purpose DSP and comparable to an ASIC design. The fact that they are re-usable means a library of mini-cores can be designed once-and-for-all to be used in future systems-on-chip.

5.3 Communication model

The mini-core system runs in two modes: configuration and normal operation. When the system is operating in the normal mode, the nodes of the system (mini-cores, DSP/RISC processors etc.) are executing programs independently from their local memories. At some points in time, intermediate results will be passed onto other nodes depending on the mapping of an application to the platform. Therefore a unified mechanism for data communication, as well as synchronization is needed. At the programming level, this is handled by message passing primitives (`send` and `receive`) that are common to all cores. With message passing, processes share *channels* instead of variables. Each channel provides a communication path between two processors and hence is an abstraction of a communication network that provides a physical path between nodes of the system.

For the mini-core system programmer, the overall application/program can be seen as a collection of independent threads that communicate through abstract channels similar to languages like CSP [18], Ada etc. Data values are never lost during transfer and the latency during a transfer should be considered arbitrary as it depends on the actual interconnect implementation.

5.3.1 Channels

Several different mechanisms for message passing have been proposed in the literature [30]. These vary in the way channels are used and the way communication is synchronized. For example, channels can provide one-way or two-way information flow, and communication can be asynchronous (non-blocking) or synchronous (blocking). Conceptually, a channel is a queue of messages that have been sent but not yet received. The effect of executing *send(channel, expression)* is evaluating the *expression* to a value, then to append a message containing the value to the end of the queue associated with the *channel*. Because this queue is unbounded (at least

conceptually), execution of *send* never causes delay; hence *send* is a non-blocking primitive. The effect of executing *receive(channel,variable)* is to delay the receiver until there is at least one message on the channel's queue. Then the message at the front of the queue is removed, and its fields are assigned to the *variable*.

The mini-core platform employs (synchronous for `receive`, asynchronous for `send`) one-way information flow over the channels. The channels of the mini-core platform are buffered. Hence, each mini-core has a number of input buffers, corresponding to the input channels, and an output buffer that is shared by the output channels.

When sending and receiving, the programmer must specify a channel identifier. From the programmer's point of view, each channel for each mini-core has a unique identifier, and is globally visible.

5.3.2 Send primitive

When a mini-core needs to transfer data to another processor core/node, this primitive is used. The usage of the send primitive involves specifying the destination – an input channel for the receiving mini-core/node – as well as the local register that stores the actual data to be transferred. The execution of the send primitive is asynchronous i.e., the sending mini-core does not wait for the data transfer to complete. (This of course depends on the amount of buffering available on the actual interconnection network).

5.3.3 Receive primitive

When a mini-core requires data from another core, or the environment, this primitive is used. The usage of this primitive involves specifying an input channel where the data is expected, as well as a local register as the destination address for the incoming data. The execution of the receive primitive is synchronous i.e., the receiving process sleeps until the expected data arrives.

5.4 Interconnection network

Peer-to-peer communication in the mini-core platform is done over an interconnection network. The topology of the interconnection network depends on the communication requirements, which is determined by the number of cores used in the system as well as the actual allocation and schedule of processes to the corresponding cores.

The send and receive channels described above have dedicated buffers that are part of an "interface" module as shown in figure 5.3. The interface module ab-

stracts the details of the interconnection network and provides a simple standardized interface to the mini-core designer. This enables concurrent design of different mini-cores and various network topologies as stated in section 5.1.3. The idea of having a standardized interface for system-level integration of intellectual property (IP) cores is also currently being pursued by the industry (Open Core Protocol Specification) [68].

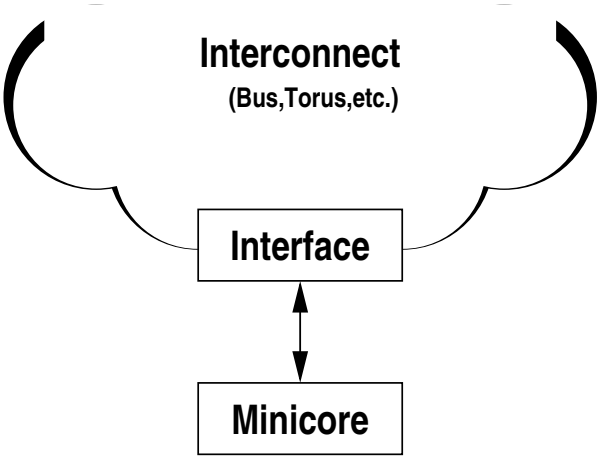

Figure 5.3: The mini-core is connected to the nodes of the interconnect structure via an interface module.

A detailed view of the interface module is illustrated in figure 5.4. The mini-core sends and receives data through the interface using two separate ports, *data_in* and *data_out*. As channel buffers are in the interface module, each mini-core specifies an input channel, and an output channel through *read_addr*, and *write_addr* ports. The mini-core initiates a read/write operation from/to the interface using a request signal and a read/write control signal denoted by *req* and *rw*, respectively.

The timing diagram for the protocol is shown in figure 5.5. The *iclk* signal shown in figure 5.5 corresponds to the generated clock for the mini-core. If the interface unit can not fulfill the request (read or write transactions from the mini-core, i.e. receive or send message requests) from the mini-core, it raises the hold signal and stops the clock for the mini-core. Obviously this will happen more frequently for receive operation where data is expected from another core. Otherwise, the effect of the request takes place one clock cycle later, by either reading data from the input channel or writing data to the output channel.

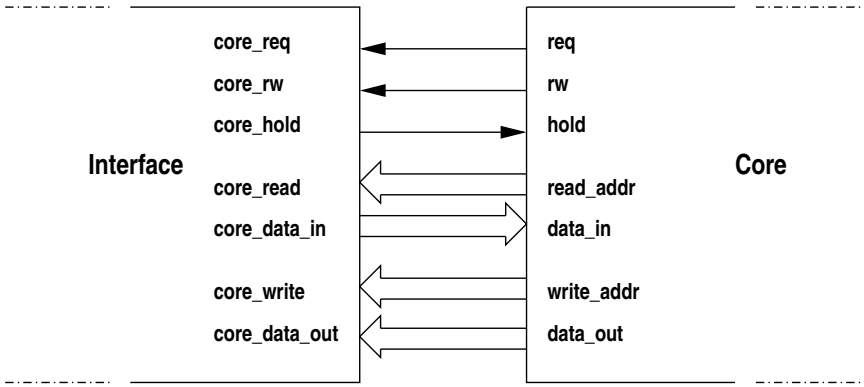

Figure 5.4: Signals connecting the interface module to a mini-core.

The choice of interconnect should be made based on the communication requirements of the system. For a few mini-cores that communicate few messages in a sample period, a simple bus structure is often sufficient. If the number of nodes increases the limited bandwidth and in particular the capacitive load of the shared

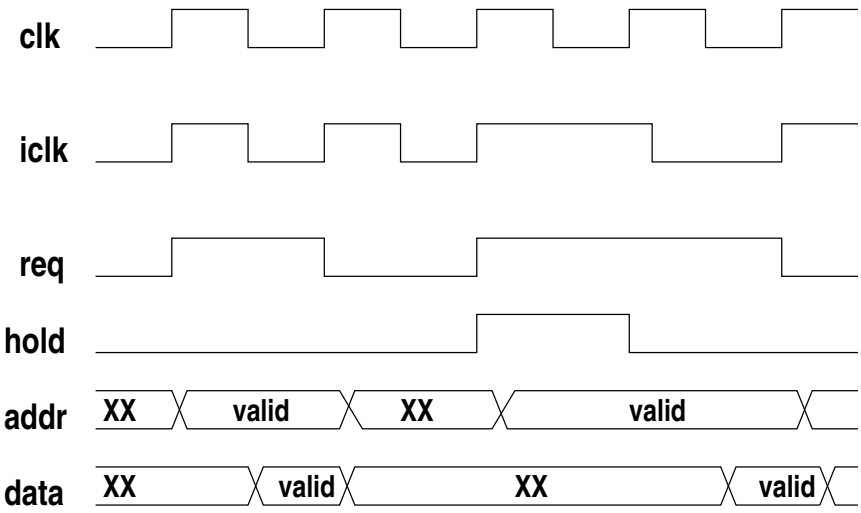

Figure 5.5: Timing diagram for the protocol.

bus may become a problem. To solve this, an interconnect structure that allows several simultaneous transfers such as a torus network may be used. In addition to the improved bandwidth, the torus also has a well-balanced capacitive load distribution to all nodes and therefore represents a good power efficient solution.

Another important design parameter accompanying scalability and bandwidth issues is the idle power consumption of the interconnect. As the nature of the algorithms combined with the overall architecture dictate that the interconnection network will be idle most of the time during program execution, the network designer should consider designs that offer low stand-by power consumption.

5.5 Configuration

What is not visible in figure 5.1 is the configuration bus that is connected to all the processors of the system. All the registers, data- and coefficient memories, network parameters are mapped onto a single address space.

During the configuration mode, the intention is to run only the configuration clock and download the binary image of the signal processing task. The control signals associated with configuration are:

- *cfg*: Enable/disable configuration mode.
- *cfgWe*: Enable/Disable write/read to/from a memory content, register file or a network parameter.
- *CfgData*: Configuration data value.
- *CfgAddress*: Target address for a write/read.

5.6 Mapping the DigiFocus algorithm

Mapping this application to a mini-core system is quite straightforward. Two types of mini-cores are needed – an FIR mini-core that could handle interpolated linear phase filters efficiently and a compressor mini-core. The compressor algorithm could also be run on a general purpose DSP, however chapter 8 will show that even a low power DSP-core may consume an order of magnitude higher power consumption than an optimized mini-core if the DSP is invoked at the nominal sampling rate. Algorithms that operate sub-sample rate (a decimation factor of 10 or more in this case) are more suitable for mapping onto a DSP.

5.7 Summary

This chapter described the proposed low power and programmable platform targeting audio signal processing, in particular hearing aids. The idea is to provide a platform composed of simple instruction set processors called mini-cores each optimized for one of these classes of algorithms as well as DSP- and/or microprocessor cores.

In order to achieve low power consumption, compute intensive parts of an application are mapped onto these mini-cores. Sub-sample rate signal processing, which is less demanding in terms of compute power and irregular control oriented tasks are mapped onto a DSP and/or RISC-core. Communication is provided over an interconnection network that meets bandwidth requirements of an application. Typically the inter-processor communication occurs at a low rate for this application domain and the overall architecture is inherently low power.

By selecting components from a library of mini-cores as well as various network topologies, the SoC designer has a relatively easy task of building up a low power and programmable systems-on-chip design for hearing aid applications. The existence of low power mini-cores poses an alternative to the use of hardware accelerators and coarse-grained reconfigurable logic in SoC design and this architecture concept is the main contribution of the thesis.

Chapter 6

Implementing the FIR and IIR Mini-cores

To evaluate the concept, and gain an in depth understanding of the architecture, we have designed two mini-cores (an FIR and an IIR mini-core), and a simple bus-based interconnection network. The design process has been a concurrent design effort involving two MSc. projects [42, 38]. This chapter will present highlights of the individual designs as well as the common design framework that has been targeted.

Sections 6.2, and 6.3 will explain the individual mini-core designs, whereas section 6.4 will talk about the bus based interconnect. The design flow and the clock gating strategy used throughout the design are explained in sections 6.5, and 6.6, respectively. The memory design common to all mini-cores are explained in section 6.7. Finally section 6.8 rounds off the chapter

6.1 Introduction

Each mini-core has a customized datapath for a particular class of algorithm. For instance the FIR mini-core employs an add-multiply-accumulate unit for linear phase filters. Likewise the IIR mini-core has a dual-multiply-accumulate unit to handle biquad sections in fewer clock cycles. On top of these customized datapaths, the instruction sets of both mini-cores are extremely simple and small.

Moreover, each mini-core has a separate program and data memory. The initializations of the mini-cores are done before run-time by downloading programs and data constants into the respective memories. For this purpose, a configuration bus that is not shown in figure 5.1, is used. The configuration bus effectively maps all register files and all data, constant, and program memories into random access

address space. The configuration bus is a separate communication medium than the interconnect network used during operation and can also be used for testing purposes. Currently the configuration bus consists of 12-bit address and 8-bit data buses, as well as a write enable signal for control.

6.2 The FIR mini-core

The FIR-mini-core is a stored instruction set processor that can be programmed to execute FIR filters studied in chapter 4. Transversal filter that is studied in figure 4.5 is also illustrated in figure 6.1 for convenience. Computing each tap of figure 6.1 requires: (1) fetching the instruction, (2) fetching two operands (data and coefficient) from memory, (3) multiplying, (4) accumulating, and (5) shifting of data in the delay line. Obviously, in order to compute one tap efficiently, some design decisions involving parallelism has to be made. For instance, separate memories for instruction and data, as well as coefficients (parallel operand access) are needed. Furthermore, in order to keep the multiply-accumulate units busy each clock cycle, efficient addressing modes for the delay lines should be provided.

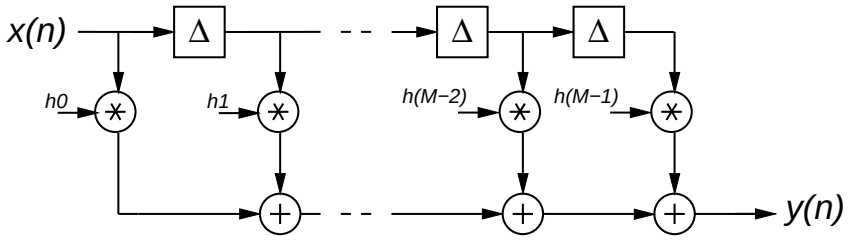

Figure 6.1: Transversal filter.

For audio applications it is often desirable to use linear-phase filters. The coefficients for such a filter are symmetric around the midpoint of the impulse response. A linear-phase filter can thus be implemented efficiently by a folded structure, where two samples from the delay-line are added before being multiplied with the corresponding coefficient. Interpolated filters are often used in filter banks. An interpolated filter has a large number of absent taps. Figure 6.2 shows an example of a linear phase interpolated FIR filter that was studied in previous work [65, 66]. The filter in figure 6.2 produces two outputs that have symmetry with respect to the half-band frequency $\pi/2$.

Computing a tap of the interpolated symmetric filter requires more complicated addressing as two symmetric data values are required at a time, and a different multiply-accumulate unit that can do add-multiply-accumulate in one clock cycle

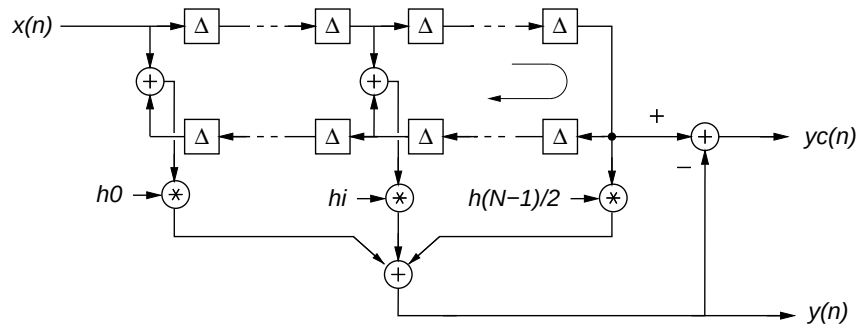

Figure 6.2: An interpolated FIR filter used in hearing aids.

as our goal is to execute the filter in as few number of clock cycles as possible in order to avoid power overhead required for controlling the datapath. The next section elaborates on the datapath.

6.2.1 Datapath

The datapath for the FIR mini-core is shown in figure 6.3. The architecture of the FIR mini-core consists of two pipeline stages. Fetching and decoding of instructions is carried out in the first stage whereas reading data operands, coefficients and computing multiply-accumulates is done in the second stage. The simple two-stage pipeline avoids overhead associated with data forwarding and control hazards.

FIR filters generally tend to have deep delay-lines in order to provide a better estimate of the *desired* frequency response [15]. For this reason, delay-line implementation of FIR filters is a critical design decision. Generic implementation

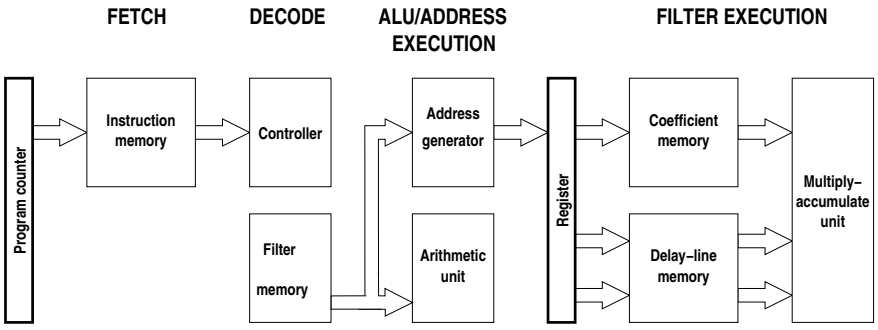

Figure 6.3: Block diagram of the FIR mini-core.

Abbreviation	Memory	Purpose:
IM	Instruction	Holds the program
CM	Coefficient	Holds the coefficients
DM	Delay-line	Holds the delay-line, and temporary variables
FM	Filter	Holds a set of filter parameters for each filter

Table 6.1: Memories in the FIR mini-core

of delay lines with strings of registers i.e., direct mapping of the delay line onto cascaded registers is difficult and very inefficient in terms of power consumption. All the energy consumed when moving data down the delay line is redundant since those transitions do not make any new computations.

A circular buffer [26] mapped into a random access memory was chosen as the data structure to implement the delay-lines of FIR filters. Instead of moving delay elements down the delay line, the pointers can be incremented or decremented. This also means that many delay lines can be mapped into a single memory as long as the arrays do not overlap.

The address calculations for the delay lines are basically addition/subtraction of an index/address pointer with a modification value (depending on where the next sample is) modulo the order of the filter. Modern DSPs can execute these address calculations concurrently with multiply-accumulates of the previous data, coefficient pair. If the updating of the index pointer occurs, after the memory is accessed, this addressing mode is called post-modification. Similarly, if the memory is accessed with the modified index pointers, this addressing mode is referred to as pre-modification.

As the FIR mini-core is required to execute symmetric interpolated FIR filters of figure 6.1 efficiently, two index pointers that address symmetric delay elements are required. Furthermore, significant memory bandwidth is required to fetch two data, one coefficient and one instruction As seen in the datapath, figure 6.3, the FIR mini-core has four different memories, each with a specific purpose. Table 6.1 summarizes the functionality of each memory.

The instruction memory holds the program for the filters and can only be written during configuration. The delay-line memory is used to map delay lines of FIR filters and it has two read ports, both of which are used in linear-phase filter implementations. The coefficient memory holds the coefficients for the filter computation and like the instruction memory, it can only be written during config-

uration. The filter memory holds the parameters i.e., pointers that specify an FIR filter. These are: the length of the filter; a pointer to the base of a coefficient array; an index into the coefficient array; a pointer to the base of the delay-line; and two pointers into the delay-line.

The FIR mini-core has several registers that the programmer can control. These are:

- **Program Counter (PC):** It is used to address the instruction memory and accessible through branch/jump instructions.
- **Filter pointer (CURR):** Current filter pointer. It is used to select which filter to operate on. This register serves as a pointer into the filter memory. It is accessible via a special instruction called *switch*.
- **Multiply-accumulate register (MACC):** It is used to store intermediate results during a filter computation. It is accessible by many instructions, involving computation and data transfer.
- **Coeffi cient pointer (CP):** Index into the coefficient memory.
- **Delay-line pointers (DP1,DP2):** These are used for addressing delay-line elements of a filter.
- **Complementary output register (COMP):** It is used to store complementary output of interpolated linear phase filters shown in figure 6.2.
- **General purpose registers (TMP1,TMP2,TMP3):** Three general purpose registers to be used for storing intermediate data. They are used as operands in ALU type of instructions.

6.2.2 Instruction Set

Filter programs covering the application domain were coded using simple RISC type of instructions like those of the DLX processor [49]. Recurring sequences of instructions were identified and replaced by a single complex instruction.

The FIR mini-core has instructions dedicated to ordinary filters (`macc`) and linear-phase filters (`asmacc`). These instructions automatically update the pointers into the delay-line and the list of coefficients and are thus very powerful for this particular application. The mini-core can also switch between filters using a single instruction (`switch`).

The instruction set of the FIR mini-core is quite small, namely 15 instructions in total. They can easily be implemented by a simple data path. The format of an

instruction is shown in figure 6.4. The mini-core has two types of instructions: (1) instructions with explicit register operands, (2) instructions with implicit register operands or without register operands.

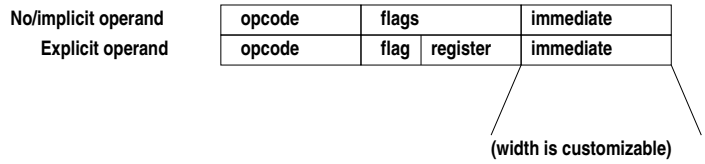

Figure 6.4: Instruction formats.

The width of the immediate field of the instruction can be customized The instruction mnemonics and a short description for each of the instructions are given in table 6.2. In the following, the instructions will be explained in greater detail such that the meaning of the flags will be clear. The possible registers for the instruction format as explained above are (unless otherwise specified) : MACC, COMP, CP, DP1, DP2, TMP1, TMP2, or TMP3. Due to the pipelined nature of the implementation, some of the instructions have delay-slots.

Mnemonic:	Function:	Flags:
MACC	Multiply-accumulate.	CLR,COMP,FIN
ASMACC	Add/Subtract-Multiply-Accumulate.	CLR,COMP,FIN,SUB
ADDI	Add immediate	MOD
SUBI	Subtract immediate	MOD
LSET	Set low part of register	-
HSET	Set high part of register	-
SWITCH	Switch the current filter.	CLR
MOVREG	Move between register and DM	REGDM,DMREG
LOAD	Load from CM or DM into register	DM,CM
STORE	Store from register into CM or DM	DM,CM
SEND	Send data to another mini-core.	-
RECEIVE	Receive data from another core.	-
JMP	Set program counter	-
BRA	Branch if register is zero/non-zero	ZERO,NZERO
NOP	No operation	-

Table 6.2: Instructions for the FIR mini-core.

Detailed descriptions

No operation:

Mnemonic:	NOP
Type:	no operands
Arguments:	none

Description: This instruction does nothing. It is used as a delay slot when no other instructions can be inserted.

Multiply-accumulate

Mnemonic:	MACC
Type:	Implicit operands
Arguments:	<flags>,<immediate>

Description: This instruction multiplies a value from the coefficient memory with a value in the delay-line and accumulates the result in the MACC register. The immediate is added to DP1 and thus used to skip over tabs where coefficient is zero. The coefficient pointer (CP) is incremented automatically. The final computation of a linear phase filter requires the adjustment of the delay line pointers. This can be done by setting the FIN flag. If the COMP flag is set then the COMP register will be set to the complementary output of the filter. If the CLR flag is set, then the MACC register will not accumulate, and the instruction will work as an ordinary multiply.

Add/subtract-multiply-accumulate

Mnemonic:	ASMACC
Type:	Implicit operands
Arguments:	<flags>,<immediate>

Description: This instruction is used for efficient implementation of linear phase filters. It adds or subtracts two values from the delay-line and multiplies the result with a coefficient from coefficient memory. The result is accumulated in the MACC register. The values taken from the delay line are indexed by DP1, and DP2 pointers. The FIN flag has the same effect as explained for the macc instruction. The immediate is added to DP1, and subtracted from DP2 (except when the FIN flag is set). The immediate helps skipping coefficients that are zero. The COMP, and CLR flags have the same meaning as in the MACC instruction. Likewise, the coefficient pointer is incremented automatically as in the macc instruction.

Add-immediate
Mnemonic: ADDI
Type: Explicit operand
Arguments: <flag>,<register>,<immediate>

Description: The immediate is added to the register. It can optionally do the addition modulo the length of the current filter. This instruction is useful to implement filters with pure delays, decimation filters and filters where the first tabs are zero.

Subtract-immediate
Mnemonic: SUBI
Type: Explicit operand
Arguments: <flag>,<register>,<immediate>

Description: The immediate is subtracted from the register. It can optionally do the subtraction modulo the length of the current filter. This instruction along with the branch-zero instruction, is useful to implement counters.

Set-low-part
Mnemonic: LSET
Type: Explicit operand
Arguments: <register>,<immediate>

Description: This instruction sets the lower part of the specified register and clears the upper part. The immediate will usually not be as wide as the general precision of the module and therefore this instruction can only be used to set the register to a small value. Large values can be set by using the HSET instruction followed by an ADDI instruction.

Set-high-part
Mnemonic: HSET
Type: Explicit operand
Arguments: <register>,<immediate>

Description: This instruction sets the high part of the specified register and clears the lower part. This instruction can be used, along with the ADDI instruction, to set the register to a large value.

Switch filter

Mnemonic: SWITCH
Type: Implicit operands
Arguments: <flag>,<immediate>

Description: This instruction is used to switch to another filter. It assigns the value of immediate to the register CURR which is a pointer to the filter memory. The flag (CLR) can be used to reset the coefficient pointer before starting a new computation.

Memory-register transfer
Mnemonic: MOVREG
Type: Explicit operand
Arguments: <flag>,<register>

Description: This instruction is used to transfer data between a register and the delay-line memory. It is useful for inserting the result of one filter operation into the delay-line of the next filter. It can also extract a value from a delay-line. It uses indirect addressing into the delay-line. The flag determines the direction of the transfer, REGDM flag for instance is a transfer from the register to the delay-line.

Load
Mnemonic: LOAD
Type: Explicit operand
Arguments: <flag>,<register>,<immediate>

Description: This instruction is used to load data from either the delay-line memory or the coefficient memory into one of the registers. It uses immediate addressing to the memory. If the value is loaded from the coefficient memory and that memory has less precision than the register, the value is extended with zeros to the right. The flag (CM or DM) determines the relevant memory for the load operation.

Store
Mnemonic: STORE
Type: Explicit operand
Arguments: <flag>,<register>,<immediate>

Description: This instruction is used to store data into either the delay-line memory or the coefficient memory into one of the registers. It uses immediate addressing to the memory. If the width of the register is greater than the memory, the least significant bits will be truncated. The flag (CM or DM) determines the corresponding

memory for the store operation.

Send

Mnemonic:	SEND
Type:	Explicit operand
Arguments:	<register>,<immediate>

Description: This instruction is used to transfer data between mini-cores. The register value is put on the channel specified by the immediate field.

Receive

Mnemonic:	RECEIVE
Type:	Explicit operand
Arguments:	<register>,<immediate>

Description: This instruction is similar to send except the direction of data transfer is reverse. These message passing primitives are explained in greater detail in chapter 5.

Jump

Mnemonic:	JMP
Type:	Implicit operand
Arguments:	<immediate>

Description: This instruction sets the program counter to the value of the immediate.

Branch

Mnemonic:	BRA
Type:	Explicit operand
Arguments:	<flag>,<register>,<immediate>

Description: Depending on the flag (ZERO/NZERO), this instruction will make a branch if the register is zero/non-zero. The branch is relative to PC+1. It is performed by adding the immediate to the program counter.

```
switch  clr, 0      ; Switch to filter 0
receive tmp1, 0     ; Receive sample
mov     regdm, tmp1 ; Move sample to delay-line
asmacc  clr, 2      ; First macc with
asmacc  2           ; clear of macc register.
asmacc  1
macc    comp, fin, 5; Final macc. Generate
                    ; complementary output and
                    ; adjust pointers.
nop                 ; Delay-slot.
send    macc, 10    ; Send output...
send    comp, 20    ; and complementary output.
```

Figure 6.5: A fragment of an interpolated symetric FIR filter program.

A sample program

Figure 6.5 is an FIR program fragment that implements a small interpolated complementary linear-phase filter of length 11 with seven non-zero coefficients. The first instruction clears the accumulator and selects the set of pointers for filter 0. The second instruction is a receive instruction that forces the mini-core to wait for a sample on channel 0. As soon as the data is available, it is stored in a local general-purpose register, tmp1. The input sample is inserted in the delay line of the said filter with the *mov* instruction. And the rest of the program is basically a sequence of multiply-accumulate instructions for computing the normal and complementary outputs of the filter. The numeric arguments to the macc and asmacc instructions are used to skip taps with zero coefficients.

Filters implemented on the FIR mini-core typically use significantly fewer instructions per sample as compared to a DSP processor. This is mainly due to the mini-core being able to skip coefficients with taps that are zero, but also because the overhead associated with switching from one filter to the next is only a single instruction. Therefore, a filter implemented on a FIR mini-core has a much lower instruction count than a traditional DSP implementation.

6.3 The IIR mini-core

The IIR mini-core is specialized in executing IIR filter programs. High order IIR filters are usually realized by a serial and/or parallel combination of low order IIR filters, to alleviate coefficient quantization sensitivity of the filter as discussed in chapter 4. The basic element for implementing a high order IIR filter is a second order IIR filter of direct form II implementation as shown in figure 6.6, known as a “biquad”.

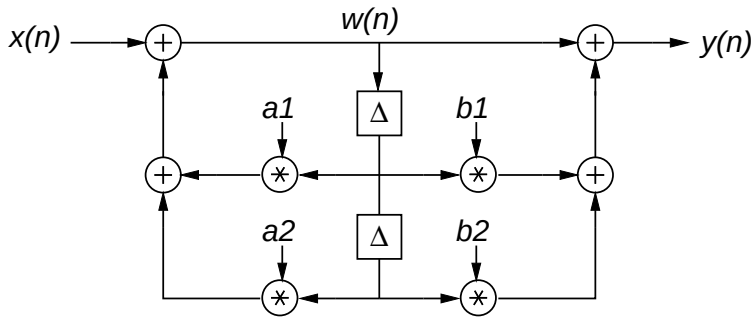

Figure 6.6: A biquad section.

6.3.1 Datapath

The IIR datapath is shown in figure 6.7. It is a simple three-stage pipeline. The design has separate memories for storing programs and coefficients. Furthermore it has a special register file to map delay elements of an IIR filter referred to as *Biquad register file*. Figure 6.7 shows a dual-multiply-accumulate unit that enables the computation of an entire biquad section in two clock cycles. It is a combinational unit that computes two multiplications and adds both results of the multiplications and the accumulator. The IIR mini-core also has a shift-add unit that is used for scaling input and/or output as well as implementing biquad sections that has a small number of '1's in their coefficients. The IIR mini-core contains the following registers that the programmer has access to through several instructions.

- **PC:** Program counter.
- **General purpose registers, r[0..n]:** They are used for storing intermediate data. The programmer has access to these registers via most instructions.
- **Biquad registers, w[0..n]:** Each biquad register is actually a pair of registers connected one after the other forming a shift-register pair. These registers are accessible via the biquad instruction.
- **DMDA accumulator:** Accumulator for the biquad instruction.
- **Shift-add accumulator, A:** Accumulator for shift-add type of instructions.
- **Auxiliary accumulator, C:** Used during shift-add type of instructions to store an intermediate value that is later used in the biquad instruction.

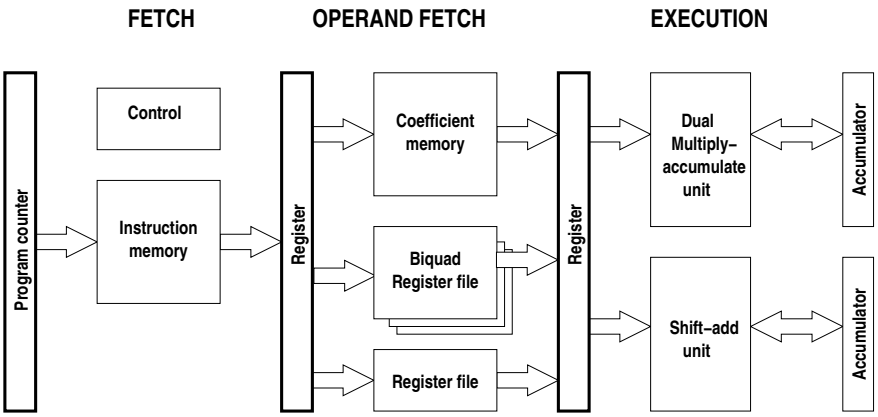

Figure 6.7: Block diagram of the IIR mini-core.

Another feature of the IIR mini-core that differentiates it from a DSP processor is the specialized register file used to store the delay elements of a biquad section. As shown in figure 6.8 the register file is implemented as a set of two-word shift-registers. The address input controls, which register pair to read. The shift-register pair implementation of a biquad stage is a direct mapping of the delay elements in figure 6.6 to hardware. The advantage of this approach is that complex addressing modes in the instruction set and the corresponding hardware can be avoided. As a biquad stage consists of only two delay elements, this way of implementing the delay line is preferable and energy-efficient.

6.3.2 Instruction Set

The instruction set of the IIR mini-core is simple and small. As all IIR filters can be constructed from biquad stages, a special instruction to execute a biquad stage would be appropriate. The IIR mini-core executes a biquad section, using a single instruction called `biq`. The biquad stage can also be computed using shift-add functionality instead of invoking the multiplier of figure 6.7.

Even though this section will present several instruction format types, one can easily observe that the instruction fields in various formats are matching, therefore there is almost no overhead circuitry that differentiates these types during decoding process. There are five types of instruction formats. All the fields of an instruction format is customizable i.e, these fields are provided as generics through the entity interface and can be set by the mini-core user. Therefore, an instruction word length is determined by the maximum word length of all types. Some fields for

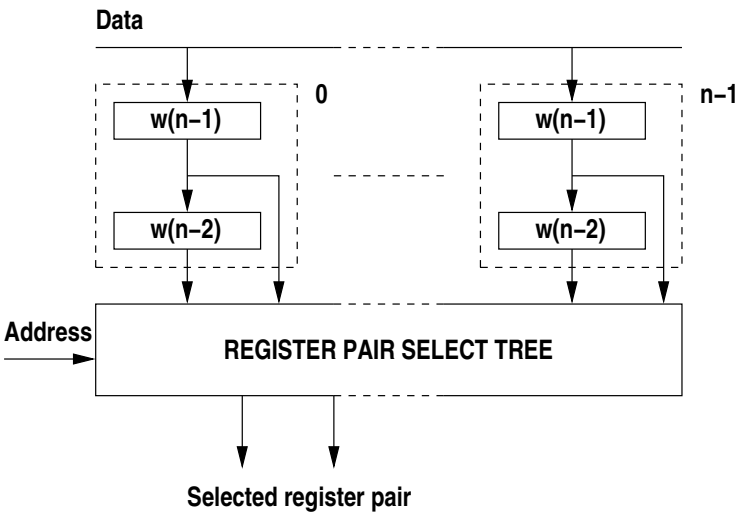

Figure 6.8: Register file implementation.

several instruction format types are shown with dotted lines. These fields for the corresponding instruction format are empty.

Type 1 Instruction format

Figure 6.9 shows type 1 instruction. The register field *regX*, and *regY* show the destination/source, and source registers, respectively. These registers belong to the general-purpose register file. The immediate field is shown by the *imm* symbol. The dotted field is empty. As all the fields are customizable in terms of word length, type 1 format simply means that the immediate field is right-aligned when forming the instruction, whereas other fields are left-aligned.

Opcode	Destination/Source (regX)	Source (regY)	_ _ _	imm

Figure 6.9: Instruction format, type 1.

The instructions of this type are:

No operation:

Mnemonic: NOP
Arguments: none
Function: none

Description: This instruction does nothing. It is used as a delay slot when no other instructions can be inserted.

Addition:
Mnemonic: ADD
Arguments: <regX><regY>
Function: regX := regX+regY

Description: This instruction adds both registers and puts the result into register regX.

Subtraction:
Mnemonic: SUB
Arguments: <regX><regY>
Function: regX := regX-regY

Description: This instruction subtracts regY from regX and puts the result into regX.

Move:
Mnemonic: MOV
Arguments: <regX><regY>
Function: regX := regY

Description: This instruction sets regX to the value of regY.

Shift-(right/left), destination accumulator, A:
Mnemonic: SHF(R/L)
Arguments: <regY><imm>
Function: A := regY » imm

Description: *SHFR* instruction shifts regY right by the amount specified in the immediate (imm) field and puts the result in the accumulator, A. It performs arithmetic shift on the operand. The same instruction for shifting left is denoted by the *SHFL* mnemonic.

Shift-(right/left), destination register file:
Mnemonic: SHF(R/L)R
Arguments: <regX><regY><imm>
Function: regX := regY » imm

Description: *SHFRR* instruction shifts regY right by the amount specified in the immediate (imm) field and puts the result in register, regX. The same instruction for shifting left is denoted by the *SHFLR* mnemonic.

Increment decimation counter, D:
Mnemonic: INC D
Arguments: implicit
Function: D := D+1

Description: This instruction coupled with a conditional branch is used in multi-rate signal processing algorithms.

Type 2 Instruction format

Figure 6.10 shows type 2 instruction format. These instructions use the shift-add unit of the IIR mini-core for computing IIR filters. If an IIR filter can be encoded with a few shift-add type of instructions, the end program may consume less power than an IIR filter implemented using biquad instructions.

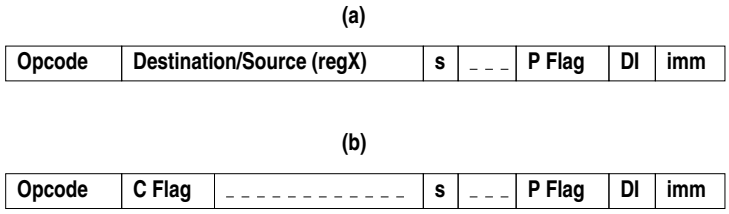

Figure 6.10: Instruction format, type 2.

The instructions of this type are:

Shift-add/subtract:
Mnemonic: SHF(R/L)A
Arguments: <flags><w(s)><imm>
Function: A := A+/-w(s)(n-DI)»imm || if C flag=1 then C:=A+/-w(s)(n-DI)»imm

Description: This instruction adds to or subtracts from the A accumulator the shifted value of delay line elements of the IIR filter specified by the biquad register, w(s)(n-DI). DI is a flag that chooses between w(n-1) or w(n-2). w(s) selects the corresponding biquad section. The C flag enables concurrent updating of the auxiliary C accumulator. The P flag shows if the shifted value will be added to or subtracted from the accumulator. The corresponding instruction for shifting left is denoted by *SHFLA*.

Shift-add/subtract, destination register file:
Mnemonic: SHF(R/L)AR
Arguments: <flags><regX><w(s)><imm>
Function: regX , A := A+/-w(s)(n-DI)»imm || w(s)(n) := C

Description: Similar to the above instruction. The added feature is the ability to specify a general purpose register for the result as well as updating the delay-line (special biquad register file) by shifting. Here the C auxiliary accumulator is used to update the delay line. This means that w(n) should be computed and saved to C, before this special instruction takes place.

Type 3 Instruction format

Figure 6.11 shows type 3 instruction format.

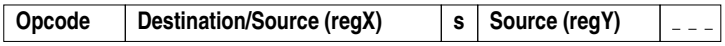

Figure 6.11: Instruction format, type 3.

The instruction of this type is:

Biquad:
Mnemonic: BIQ
Arguments: <regX><w(s)><regY>
Function: Implementing biquad equations

Description: This instruction computes a whole biquad section in two clock cycles. It computes the following difference equations. The coefficients a_1, a_2, b_1, and b_2

are indirectly addressed via an index pointer into the coefficient memory. The programmer sets a special register to the number of biquad stages during configuration. The circular buffering and updating of the index pointer is done automatically during the execution of the BIQ instruction.

- w(s)(n) := regY + a1 w(s)(n-1)+ a2 w(n-2)
- regX := w(s)(n) + b1 w(s)(n-1)+ b2 w(n-2)

Type 4 Instruction format

Figure 6.12 shows type 4 instruction format.

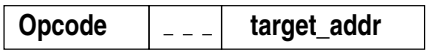

Figure 6.12: Instruction format, type 4.

The instructions of this type are:

Jump:
Mnemonic: JMP
Arguments: <target_addr>
Function: PC:= target_addr

Description: This instruction jumps the control of the instruction flow to a target address in the instruction memory.

Branch equal:
Mnemonic: BEQ
Arguments: <target_addr>
Function: PC:= target_addr if D=multi_rate_counter

Description: Branch if the decimation counter D is equal to the multi-rate counter. The multi-rate counter specifies the multi-rate coefficient and is set during configuration. This instruction along with INC D instruction is used in multi-rate signal processing algorithms where the sampling frequency of some parts of the system

is lower than the original sampling frequency (decimation). This requires the mini-core that is running slowly to output new data at its own "slow" sampling rate.

Type 5 Instruction format

Figure 6.13 shows type 5 instruction format.

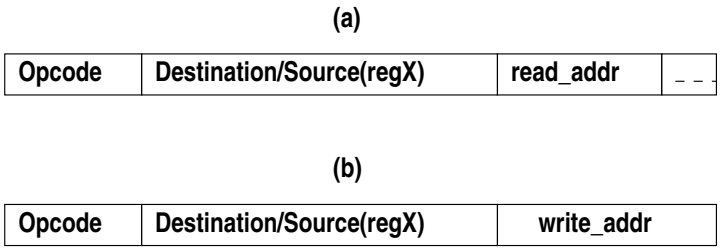

Figure 6.13: Instruction format, type 5.

The instructions of this type are SEND and RECEIVE. They have been explained in the FIR mini-core instruction set description as well. Naming of some of the fields of these two instructions are different, as they were designed by different individuals, but the basic functionality is the same. In figure 6.13, read address field represents the input channel, whereas the write address field specifies the input channel for the receiving mini-core. Therefore the write address field is potentially larger as the receiving mini-core should be specified as well.

Sample programs

Figure 6.14 shows an IIR program that implements a 4^{th} order cascaded IIR filter. The first instruction of the IIR program waits for the input sample to arrive at channel 0. Until the data arrives, the mini-core goes to sleep mode i.e., the clock signal to the mini-core is stopped. The input sample is stored in register r1. The second instruction computes a biquad stage. The biquad delay line consists of a shift-register pair, and is specified by *w0*. The output of this biquad stage is placed in register r2. The third instruction reads from register r2, and computes another biquad stage, putting the result in register r3. The output of the cascaded IIR filter is sent to channel 4 i.e., to another mini-core. The last instruction jumps to the start

of the program, and executes the receive instruction in the next clock cycle. The mini-core will go to sleep until the data for channel 0 shows up.

```
start:receive r1, 0     ; Receive sample to r1
      biq     w0,r2,r1  ; Compute biquad 1,
      biq     w1,r3,r2  ; Compute biquad 2,
      send    r3, 4     ; Send output...
      jump start        ; jump to start
```

Figure 6.14: An IIR filter with two biquad sections.

The IIR filter presented in figure 6.14 can also be implemented using shift-add type of instructions. The system programmer should be careful however, the number of instructions to implement a biquad stage will increase according to the number of "1"s in the coefficient representation. Even though shift-add instructions are cheap in terms of energy consumption, the overall program may not be power efficient. Figure 6.15 shows the IIR filter implementation using shift-add instructions.

```
start:receive r0,0
      shfr A,r0,1                 #A = x(n) >> 1
      shfr A,w0(n-1),2,add        #A = A + x(n-1) >> 2
      shfr A,w0(n-1),3,add        #A = A + x(n-1) >> 3
      shfr A,C,w0(n-2),2,sub      #C = A = A - x(n-2) >> 2
      shfr A,w0(n-2),0,add        #...
      shfl A,r1,w0(n-1),1,sub     #...
      shfr A,r1,2                 #1st biquad output
      shfr A,w1(n-1),2,add        #...
      shfr A,w1(n-1),0,add        #
      shfr A,w1(n-2),1,sub        #
      shfr A,C,w1(n-2),2,sub      #
      shfr A,w1(n-2),0,add        #
      shfl A,r1,w1(n-1),1,sub     #
      shfr r2,r1,2
      shfl r3,r1,1
      add  r2,r3
      send r2,24                  # send output...
      jump start
```

Figure 6.15: The same IIR filter with shift-add type of instructions.

Only the first four shift-add instructions will be explained here as the rest are similar. Input data sample is put into register r0 first. The next instruction shifts data right by 1 bit (i.e. divide by half) and puts the result in accumulator A. The third instruction is similar, only the shift amount is by three bits. The fourth instruction makes a copy of the A accumulator and puts the result of the computation

onto both the A, and C accumulators. The C accumulator holds $w(n)$, which is an intermediate result to be inserted to the delay line at the end of the biquad computation.

6.4 The Interconnect network

Because of its simplicity and the modest communication requirements, the test chip described in this thesis has a bus based interconnect network. The address space of this bus is the union of all input buffers in all interface units. A send instruction results in a write transaction to an input buffer. A receive instruction simply transfers data from the input buffer to the associated mini-core.

Since the bus is a shared medium, arbitration is required to ensure that only one node at a time is driving the bus. The test chip use a simple round robin arbitration scheme implemented in a distributed fashion using a circulating token.

A mini-core executing a `receive` instruction goes to “sleep” until the requested data item shows up at the specified channel. Likewise a mini-core executing a `send` instruction halts until the network consumes the data item in the output buffer. These sleeping modes are handled by clock gating at the module level. A mini-core is only clocked when necessary, and this results in significant power savings.

Currently the idle power consumption of the network is relatively high, compared to the energy consumption of actual data flowing through the network. Therefore, asynchronous solutions for the network that are showing promise in terms of idle and overall power consumption are being investigated [67].

6.5 Design flow

The design flow that we have used is based on synthesizing from an RT level VHDL description into a standard cell netlist. Synthesis and simulation was done using the Synopsys tool set (DC compiler, and VHDL debugger) and placement and routing was done using Cadence Silicon Ensemble. Total power consumption has been analyzed by using Synopsys Power Compiler and the detailed power breakdown reported in section 8.7 has been obtained using an "in house" tool capable of post processing the power report file generated by Synopsys.

We have deliberately avoided the use of full-custom layout and mask-level macro-cells like RAM-blocks; the design is implemented using standard cells only. Furthermore, we have used Synopsys designware multipliers and adders. In combination with the use of standard cells, this resulted in the lowest possible design

effort and maximum portability. Low power has been obtained by optimizing at architecture level, RT level and gate level by extensive use of clock gating (explicitly expressed in the VHDL code). Doing manual clock gating at the top level enabled us to be in more control of the clock network synthesis process.

The VHDL code is parameterized such that it is possible to instantiate mini-cores with different word-size and different size memories. For the interconnection network, the number of nodes is a parameter. Hence it is fairly straightforward to instantiate a complete platform with the appropriate amount of resources.

6.6 Clock gating strategy

Avoiding unnecessary switching in the clock network is a common technique used in synchronous designs to reduce power consumption. In general two levels of clock gating are employed in the mini-core based platform:

(1) At the highest level, the clock input to each mini-core is gated. The condition required for stopping the clock, depends on the availability of data to-and-from the interconnect network: while executing a “receive” instruction, a mini-core waits for a data item in an idle state until the corresponding item shows up at the associated channel input buffer. In the meantime, the clock input to the mini-core is stopped, avoiding energy waste that would otherwise occur during the idle state of the mini-core. Likewise, while sending a data item through the network, a mini-core waits for the availability of the network in an idle state. A mini-core executing a “send” instruction is only allowed to proceed when the network is available. The idle state is implemented by stopping the clock feeding the mini-core.

(2) Inside the individual mini-cores, we use clock gating at the RT level. The pipeline registers at the inputs of combinational units such as multipliers, ALUs, etc. are only clocked depending on the instruction being executed. Two well-known advantages of clock gating at the RT level are; it eliminates switching in the unused modules, it reduces switched capacitance in the clock tree inside the mini-cores.

Although quite aggressive, the above is all standard low power design practice, and it should be emphasized that the main power savings stem from the overall architecture.

6.7 Memory design

Due to the lack of a hard block SRAM, we used latch-based memories in all mini-cores. The energy penalty considering the small size memories used in the mini-

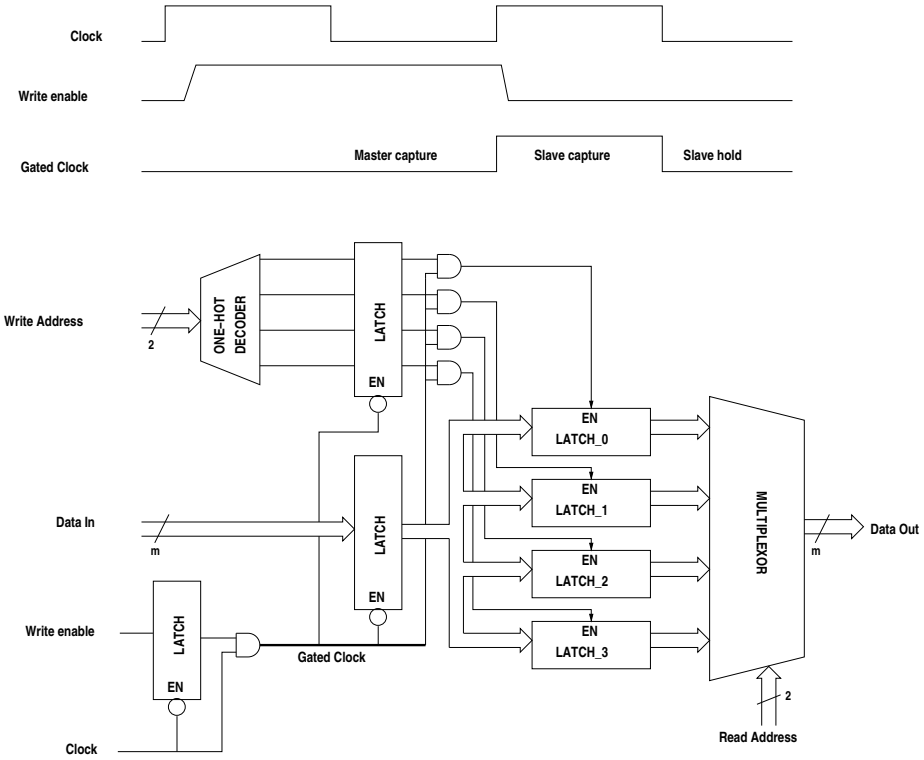

Figure 6.16: Implementation of the latch-based RAM.

cores is marginal, and furthermore layout generation became quite easy with all cells in the design being similar.

Conceptually the memory design used throughout the mini-core platform is an array of edge triggered flip-flops where only one is clocked depending on the address. The implementation uses a single shared master latch. The sample memory shown in figure 6.16 consists of $m-$bit words. An $n-$word memory is composed of 1 master latch and n slave latches.

Writing into the memory is controlled by the write enable signal that is used to produce a gated clock. When the clock is low, the master latch captures data from the “Data In” port. At the same time, depending on the destination address, the one-hot decoder selects a target slave latch. The select signal goes through an AND gate, and only the selected slave latch sees a gated clock at its enable input, as illustrated in figure 6.16. When this gated clock signal goes high the corresponding slave latch becomes transparent and when the clock signal goes low it closes.

The interesting part of this design is that during a write operation only the mas-

ter latch and the selected $m-$bit latch are clocked, thanks to clock gating circuitry being merged with the decoding unit as shown in figure 6.16. This results in very small energy consumption for a write.

A read operation is simple; the only active block in the memory during a read operation is the multiplexer tree at the output port. The clock feeding the memory is stopped during a read operation, hence saving unnecessary switching activity in the clock tree.

6.8 Summary

Two mini-core designs, and the interconnection network have been explained in this chapter. Both mini-cores are simple pipelined instruction set processors with specialized datapaths tailored to their respective application domains. The instruction set for each mini-core is explained through a set of filter application programs.

The mini-cores are parameterizable, and well suited for a synthesis based ASIC design flow. For this purpose, we have avoided the use of full-custom layout and macro-cells.

Unnecessary switching activity in the mini-cores is eliminated through a clock gating strategy that shuts-off entire and/or parts of unused mini-cores. This combined with custom datapaths with small instruction set designs, leads to energy-efficient mini-core designs.

Chapter 7

The Test Chip

To assess the feasibility of the mini-core approach, a test chip containing 6 mini-cores and an interconnect network has been designed and fabricated. It has been successfully tested and verified at 1.8 Volt.

The testing procedure has been carried out using a test bench that consists of (1) a Xilinx FPGA board (RC1000-PP) that is connected to a host PC via the PCI bus, (2) a test board that accommodates the test chip, a switch array for individual current measurements and some multiplexers, and (3) a logic analyzer.

This chapter will present these components and functionality of the test bench. Section 7.1 will give some information on the particular prototype implementation using the mini-core approach whereas section 7.2 will describe how the test bench works. Finally section 7.3 will summarize the chapter.

7.1 The chip

A layout of the test chip is shown in figure 7.1. The test chip is implemented using $0.25\mu m$ CMOS STMicroelectronics standard cell library. The core area is approximately $5mm^2$ and contains 520 K transistors.

The mini-cores on the test chip are instantiated with different memory sizes and can be programmed to any application consisting of FIR and IIR type of algorithms. Table 7.1 shows the main characteristics of the mini-cores on the test chip. Typical filter examples are used to determine these parameters. The Data Memory field for FIR type mini-cores shows the sizes of delay-line memories whereas for IIR type mini-cores, that field represents the sizes of biquad register files.

The mini-cores, and the network have separate power supply pins that allows us to measure current consumption of each block individually. The communication network is also available at the pins, allowing a possible extension with off-chip

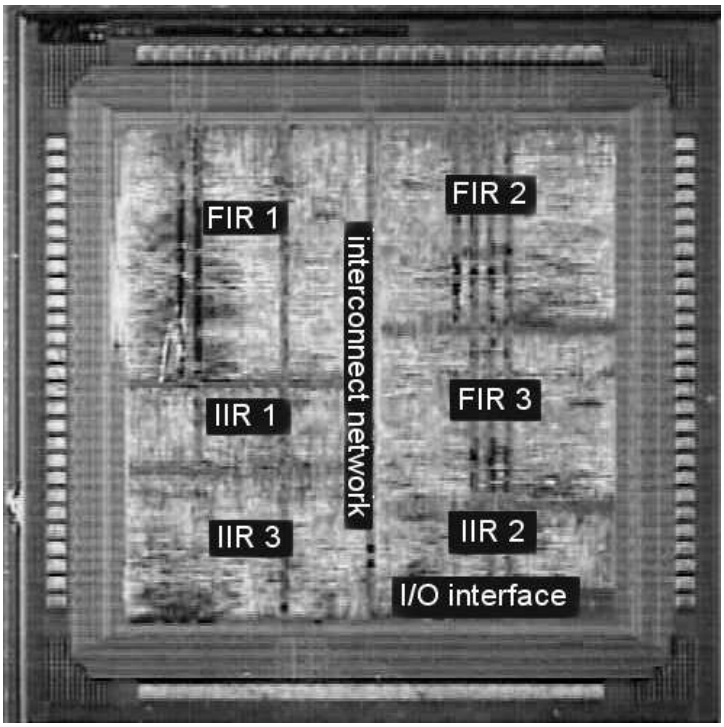

Figure 7.1: Die photo of the test chip.

mini-cores, DSP cores and micro-controller cores.

The test chip has successfully been tested and verified at 1.8 Volt power supply. As the FPGA board used for testing has a fixed 3.3 V power supply, scaling the power supply of the test chip further proved difficult. However, from a previous experience with the same process technology, the test chip is expected to operate correctly at 1 V.

7.2 Test bench

This section will present the architecture of the test bench in section 7.2.1, and the major components used in the test bench, in sections 7.2.2 and 7.2.3.

7.2.1 The idea

The test bench shown in figure 7.2 has two main functions:

Mini-core	Data Memory words x bits	Instruction Memory words x bits	Coefficient Memory words x bits
FIR1	118x16	41x16	16x16
FIR2	93x16	32x16	16x16
FIR3	64x16	32x16	25x16
IIR1	4x20	32x13	4x12
IIR2	8x20	32x14	8x12
IIR3	16x25	64x15	16x20

Table 7.1: Mini-core parameters.

- To verify the test chip functionality by exciting it with pre-defined input vectors, and comparing the outputs with the outputs of previous simulation runs. This is basically performing a crosscheck of the prototype with a "golden" model verified at the beginning of the design process.

- To measure current consumption of each individual block, i.e., mini-cores, and the interconnect network.

The following units make up the test environment: (1) the RC1000-PP FPGA board that is connected to a host PC via the PCI bus, (2) a custom designed test board that accommodates the test chip, a set of switches and, multiplexers as shown in figure 7.6, (3) the host PC that accommodates the FPGA and the software to control the test environment, (4) a logic analyzer for additional probing to the system.

Controlling the test bench is taken care by a test program implemented in C++ on the host PC [80]. The test program is a command line application that allows the test engineer to exercise several commands for verification and power measurement purposes. A simple finite state machine is designed as an interface between the C application and the test board. The basic task of the FSM is to input data to the test chip, and collect the corresponding outputs. For this purpose, the FSM communicates both with the test program and the test board. The test vectors for the test chip are stored in the host PC. Feeding the chip with input data occurs in two steps. (1) the test program asks for an ownership for the SRAM memories on the RC1000-PP board, and stores all the necessary data (audio samples as well as a command for the FSM) onto the memories via DMA transfer after the ownership of the memories has been granted. (2) The FSM on the FPGA is notified about the transfer and then granted with the ownership of the corresponding SRAM block. It then continues with the operation and executes the required task.

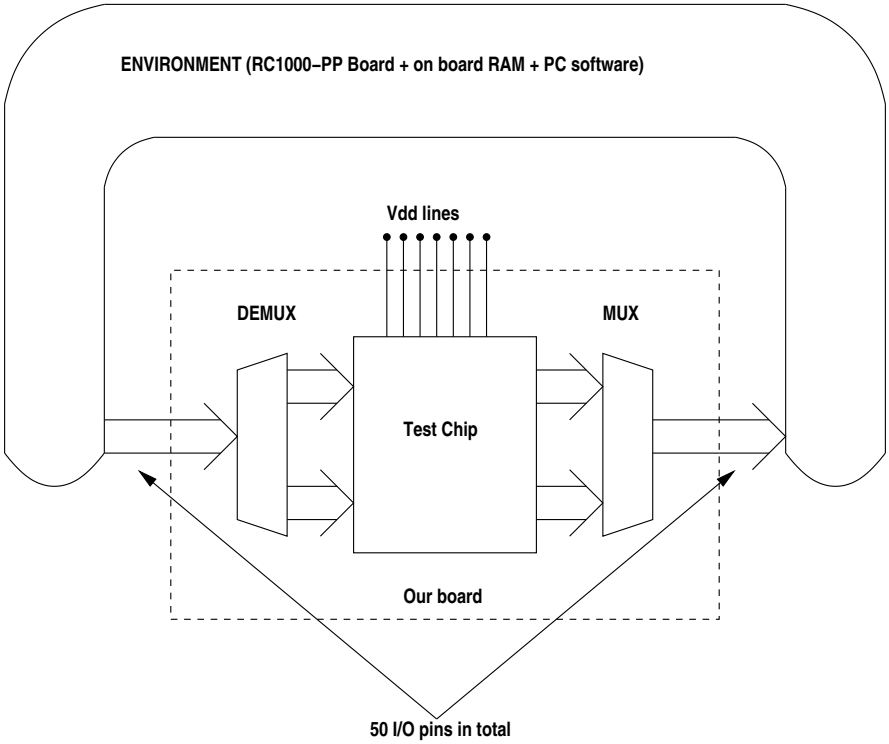

Figure 7.2: Functional block diagram of the test bench.

When the test program is started, the user first specifies the clock rate for the test bench, as well as the file that holds configuration data. The configuration data is a collection of test programs for all the mini-cores. After configuration, the program asks the user to select any of the commands below.

- **RunDataSet:** When this command is executed, a finite set of audio samples are fed to the test chip, and the corresponding output is stored onto the SRAM memories on the FPGA board. From here, they are written into an ASCII file, and compared to the golden model output. It is possible to test all mini-cores individually, as well as the network by executing various programs in succession.
- **RunForever:** When this command is executed, the FSM on the FPGA repeats the finite data set forever by looping to the start whenever the end of the data set is reached. This command is used for measuring current consumption of each block on the test chip.

- **RunBus:** This command is used to keep track of the data transfers over the interconnect network. As the communication bus is available on the pins, it can be observed with this command.

A photo of the actual test bench is given in figure 7.3

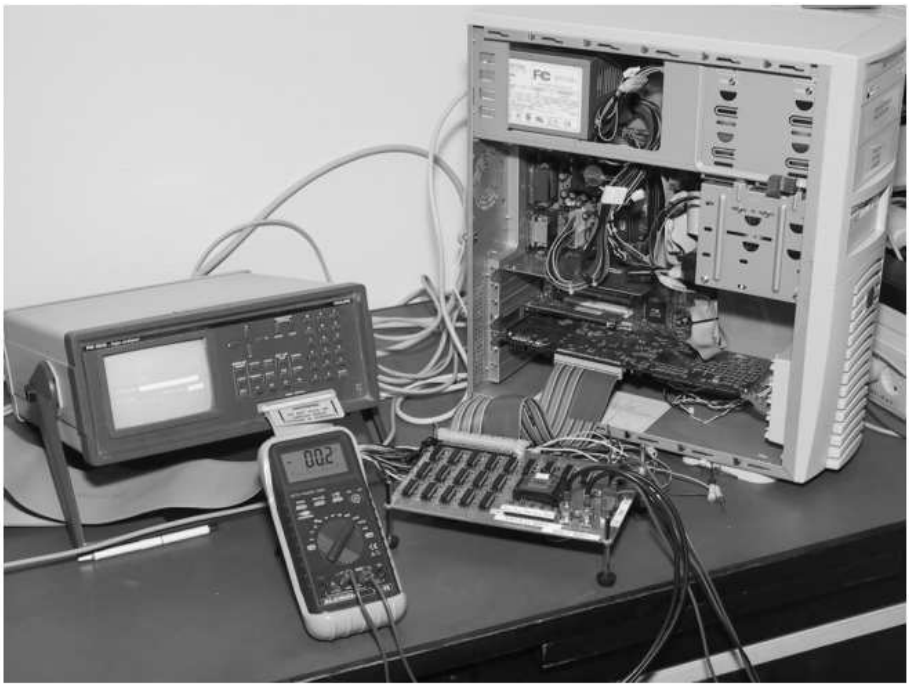

Figure 7.3: The test bench used for functional verification and power measurements.

7.2.2 RC1000-PP board

In order to apply test patterns to the test chip and collect the outputs on a PC for further comparison with simulated chip results, we have used an FPGA board (RC1000-PP) connected to a host PC via the PCI bus. The board is shown in figure 7.4.

The RC1000-PP hardware platform is a standard PCI bus card equipped with a Xilinx Virtex family BG560 FPGA with up to 1 million system gates. It has 8MB of SRAM directly connected to the FPGA in four 32-bit wide memory banks. The memory is also visible to the host CPU across the PCI bus as if it were normal memory. Each of the 4 banks may be granted to either the host CPU or the FPGA

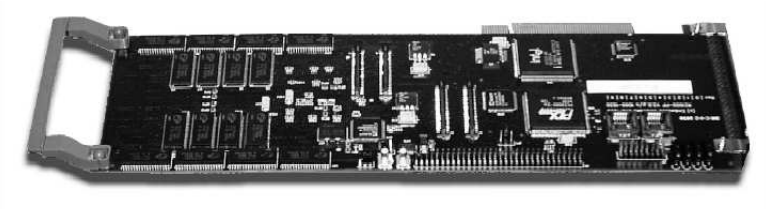

Figure 7.4: The RC1000-PP rapid prototyping development platform.

at any one time. Data can therefore be shared between the FPGA and the host CPU by placing it in the SRAM on the board. It is then accessible to the FPGA directly and to the host CPU either by DMA (Direct Memory Access) transfers across the PCI bus or simply as a virtual address.

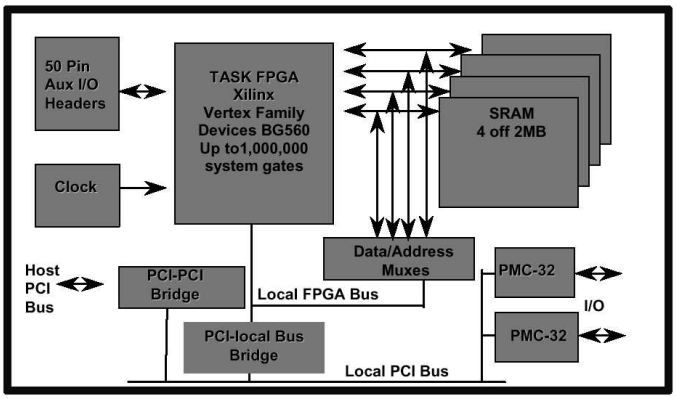

Figure 7.5: The RC1000-PP functional block diagram.

The board also includes two standard PMC connectors for directly connecting other processors and I/O devices to the FPGA; a PCI-PCI bridge chip also connects these interfaces to the host PCI bus, thereby protecting the available bandwidth from the PMC to the FPGA from host PCI bus traffic. A 50 pin unassigned header is provided for inter-board communication. Our custom designed test board that accommodates the test chip uses this 50 pin header to communicate with the FPGA. The functional block diagram of the FPGA board (RC1000-PP) is illustrated in figure 7.5.

7.2.3 Our test board

Because the number of I/O pins available on the RC1000-PP for inter-board communication were limited to 50, we have designed a test board that employed multiplexers accompanying the test chip as shown in figure 7.6. We have also implemented a switch array on the board to enable measuring individual current consumption of the mini-cores and the network.

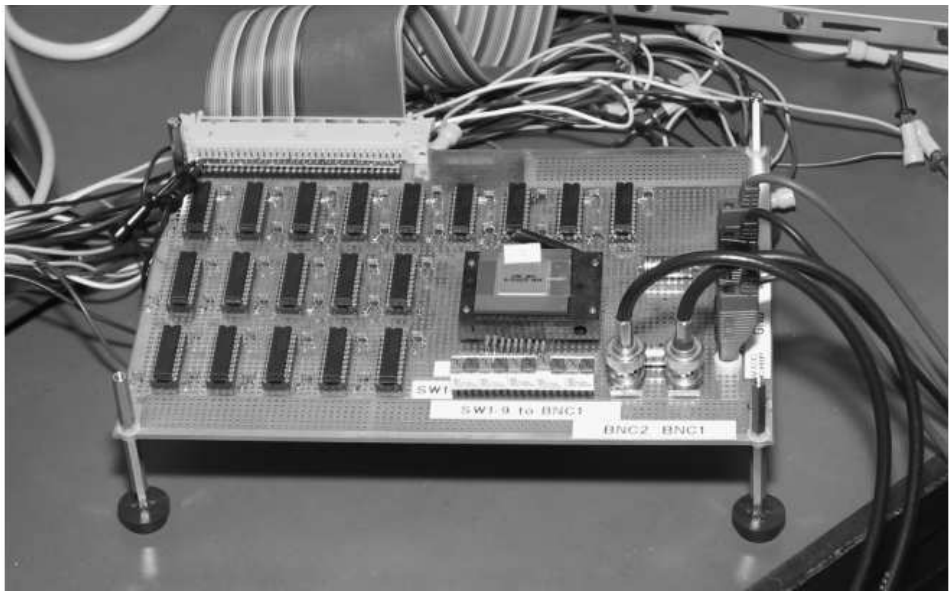

Figure 7.6: Photo of the test board.

7.3 Summary

The test bench described in this chapter proved quite useful and low cost as an alternative to the use of very sophisticated testing equipment. Basically the test bench served as a physical environment to perform crosschecks with the simulations of the design using conventional CAD tools.

Furthermore, we have successfully tested and verified the prototype at 1.8 Volt that enables us to gain more confidence in evaluating the mini-core approach.

As the FPGA board (RC1000-PP) used for testing has a fixed 3.3 V power supply, scaling the power supply of the test chip further proved difficult. However, from a previous experience with the same process technology, the test chip is expected to operate correctly at 1 V.

Chapter 8

Results

This chapter evaluates the mini-core approach by comparing it with two commercial low power DSP processors and hardwired ASICs that implement an FIR filter bank and IIR filters. The goal is to identify where the mini-core approach fits into power vs. flexibility trade-off curve of figure 1.1.

8.1 Introduction

Even though, we'll examine extensive comparisons with alternative implementations, a 100% apples-to-apples comparison is not always possible. Many articles on low power DSP architectures only state energy-per-instruction measures like: MIPS/W (mega instructions per second per Watt), or MOPS/W (mega operations per second per watt). These figures should be taken with some care as they completely ignore the instruction-count-per-task issue. A fair comparison requires one or more real benchmarks for which one can estimate the energy consumption. Other issues that may offset the numbers are technology, supply voltage, etc.

The mini-core designs are implemented in a 0.25 μm ST process and the hearing aid application is intended to operate at a supply voltage of 1.0 V. In the following we will scale power figures of alternative designs to this process and supply voltage whenever it is possible to do so.

We have used four benchmark programs to evaluate the mini-core designs: (1) a bandpass FIR filter with 25 coefficients, (2) a highpass IIR filter with two biquad stages, (3) a filterbank with interpolated FIR filters that divides the input signal into 7 frequency bands [65] and, (4) an equalizer, that is a combination of a filterbank with high and low-pass IIR filters. The filterbank is a non-trivial industrial application extensively used in hearing aids. The benchmark comparison tables 8.1, and 8.2 will consist of instruction count per sample, clock cycles per

sample, required clock frequency, and power per sample entries. We have assumed 16 KHz sampling rate for all benchmarks.

In sections 8.2 and 8.3 we will provide full benchmark comparisons between the mini-cores and two commercial general-purpose DSPs: the TMS320C54x DSP chip and the *ARCTangent* $-$ *A*4^{TM} synthesizable core (in the following referred to as "the ARC-core"). Following this, section 8.4 provides full benchmark comparisons between the mini-cores and dedicated hardwired ASIC implementations designed by our industrial partner, Oticon A/S. To put the mini-core approach into a broader perspective, section 8.5 reports W/MIPS figures for a collection of other designs reported in the literature. In section 8.6, we will present the power consumption of the current network and also report idle power of the mini-cores. Finally, in section 8.7, we will provide power consumption breakdown of a single mini-core in order to provide additional insight into where power is consumed.

8.2 Comparison with the TMS320C54x

We will first evaluate the mini-cores against a general-purpose DSP processor, the TMS320C54x produced by Texas Instruments. This processor is also implemented in a 0.25μm technology and it is a representative off-the-shelf DSP. It is thoroughly characterized in terms of its power consumption [86] and optimized assembly code for various applications is freely available [40].

Table 8.1 shows the power consumption for a couple of benchmarks. The mini-core power figures are based on measurements. The TMS320C5x figures are estimated using assembly programs published by Texas Instruments in [40]. We have removed initialization code and used instruction-level power measurements documented in [86]. The measurements in [86] are done at a supply voltage of 3 Volt, and we have scaled them to 1V to enable comparison, table 8.1. The programs for the TMS320C54x are assumed to be running from on-chip memories. Due to data dependencies and inter-instruction dependencies the power-per-instruction figures of the processor may vary and [86] provides best case and worst case figures. For this reason the TMS320C54x figures in table 8.2 show an estimation interval, rather than a single absolute power figure.

As seen from table 8.1 the FIR mini-core consumes only 15-21% of the power consumed by the TMS320C54x. This huge power saving is due to the mini-core being small and simple, thereby consuming less energy per instruction. For the IIR mini-core the picture is even more favorable. Its power consumption is only 3.3-4.1% of the corresponding figure for the TMS320C54x. Furthermore, the mini-core executes 2.25 times fewer instructions and has a 6.4 times lower energy/instruction figure.

FIR filter	FIR mini-core	TMS320C54x
Inst. per sample:	30	33
Clocks per sample:	30	33
Clock frequency:	500 KHz	532 kHz
Power for task @1V:	28 μW	133 - 183 μW
IIR filter	IIR mini-core	TMS320C54x
Instructions per sample:	10	27
Clocks per sample:	12	27
Clock frequency:	500 KHz	435 KHz
Power for task @1V:	4.8 μW	116 - 145 μW

Table 8.1: Power consumption of different filter implementations assuming a 16 KHz sampling rate. The figures for the FIR mini-core and the IIR mini-core can be compared with similar figures for a TMS320C54x DSP. All figures assume a supply voltage of 1.0V.

Please note that the IIR mini-core could be run at a slower clock rate, unfortunately the test bench could not create a slower clock than 500KHz. However, the mini-core does not consume power during the idle period before the next input sample. This is because the mini-core will execute a “receive” instruction and go to “sleep” mode where the clock that feeds the mini-core is stopped.

The instruction count figure for executing the FIR filter benchmark is similar for both designs. The reason is, this benchmark is an ordinary FIR filter that all DSP processors can implement with a tight loop. The FIR mini-core has been designed to allow efficient implementation of symmetric interpolated FIR filters that require complex dual-data addressing, and for such filters the mini-core has a significantly lower instruction count per task as seen in table 8.2 that is introduced in the next section.

One final remark about the figure in table 8.1 is that, our clock cycles per sample estimates could be fairly optimistic for the TMS320C54x, since data dependencies and pipeline behavior are not taken into account.

8.3 Comparison with the ARC-core.

Through our industrial partner, we had access to the ARC processor core developed by ARC International. It is a synthesizable 32-bit RISC-core intended for low-power, high performance SoC based designs. The basic CPU can be extended with a MAC unit and an XY data memory. Furthermore, it has an extendable

FIR filter	FIR mini-core	The ARC-core
Instructions per sample:	30	30
Clocks per sample:	30	38
Clock frequency:	500 KHz	607 KHz
Power for task @1V:	15 μW	$>$169 μW
IIR filter	IIR mini-core	The ARC-core
Instructions per sample:	10	20
Clocks per sample:	12	35
Clock frequency:	500 KHz	560 KHz
Power for task @1V:	6.8 μW	$>$148 μW
Filterbank	FIR mini-cores	The ARC-core
Instructions per sample:	73	153
Clocks per sample:	50	205
Clock frequency:	1 MHz	3.2 MHz
Power for task @1V:	71 μW	$>$423 μW
Equalizer	All mini-cores	The ARC-core
Instructions per sample:	101	200
Clocks per sample:	61	268
Clock frequency:	1 MHz	4.2 MHz
Power for task @1V:	92.5 μW	$>$554 μW

Table 8.2: Comparing the mini-cores with hardwired ASICs and a low-power DSP core, extrapolating to 16 KHz sampling rate, 1 V power supply and similar semiconductor process. The filterbank is partitioned and assigned to two mini-cores running in parallel, therefore clock cycles per sample figure is less than the total instruction count.

instruction set that can be customized based on the customer requirements. The specific instance that we have evaluated includes the basic CPU, 2x128x32 bits of XY-memory, and a 24-bit pipelined MAC unit.

Table 8.2 shows the power consumption for all four benchmarks introduced in section 8.1. The power data for the mini-cores are based on simulations except for the filter bank and equalizer applications that were based on actual measurements. Our experience is that power consumption estimates obtained through simulation is 15–20% higher. The power data for the ARC-core are based on power simulations of a synthesized netlist. The original power figures for the ARC-core are obtained for a 0.18μm UMC standard cell library. To enable comparison with 0.25μm ST library that was used to implement the mini-cores, we scaled the power figures by a

Filterbank	FIR mini-core(s)	ASIC
Power for task @1V:	71 μW	48 μW
IIR filter	IIR mini-core	ASIC
Power for task @1V:	6.8 μW	4.2 μW

Table 8.3: Evaluating flexibility vs. power trade-off between mini-core designs and dedicated circuitry. The IIR filter power numbers are based on power simulations, whereas the filterbank comparison is based on measurements. All figures assume a supply voltage of 1.0V and a sample rate of 16 KHz.

factor of 1.9. This scaling factor was obtained via a representative ASIC developed by our industrial partners who implemented the design in both technologies.

The input data applied for the first two benchmark programs involve noise, whereas the last two benchmarks are excited with a sine wave. The ARC-core data includes the power consumption of the XY memory but not the program memory as we used a behavioral model for the program memory in the simulations. The results presented therefore represent a lower bound, as indicated by the "$>$" symbol in the table. Another important thing to note, is that the ARC-core has a 24x24 multiplier, whereas the FIR mini-core, and the IIR mini-core contains 16x16, and 16x12 bit multipliers respectively. However, data and coefficient word lengths for the ARC-core programs and the mini-core programs are the same in all benchmarks.

Having said that, the mini-cores consume at least 6 – 21 times less energy per task while executing the benchmarks as can be seen from table 8.2. This is not a surprise, as the mini-cores are only programmable within their corresponding algorithm class, whereas the ARC-core can basically execute any DSP algorithm. It is interesting to note the power savings due to the reduced flexibility. The mini-cores also execute 2 times fewer instructions per task in general. The only exception benchmark is the ordinary FIR filter that can be implemented as a very efficient loop on the ARC-core.

8.4 Comparison with ASIC implementations

Another interesting question is how well a mini-core implementation compares with a hardwired synthesized ASIC implementation. For this purpose, another comparison is made between the mini-cores and hardwired ASICs designed in the same 0.25μm technology by our industrial partners as shown in table 8.3. While running the filter bank and the high-pass IIR filter, the mini-cores consume 1.5 and 1.6 times more power than the corresponding hardwired ASIC implementations.

Design	Technology	Methodology	Power metric
Coyote, [61]	0.25 μm	some full-custom	100 μW/MIPS
Lee et al., [58]	0.35 μm dual-V_t	some full-custom	210 μW/MHz
Mutoh et al., [63]	0.5 μm multi-V_t	some full-custom	1.1 mW/MHz
Pleiades, [93]	0.25 μm	some full-custom	10-100 μW/MOPS
Phonak IC,[62]	0.25 μm	standard cells only	14.4 μW/MOPS
Mini-cores	0.25 μm	standard-cells only	11-26 μW/MIPS

Table 8.4: Comparing the mini-core approach with other designs in literature.

8.5 Some additional comparisons

Based on the power figures and benchmark programs reported in the previous sections we can estimate an absolute power efficiency of mini-cores to be around 21-53 μW/MIPS (for relatively complex instructions), or 26-62 μW/MMACs (Mega Multiply-Accumulate per second). These results are obtained using normal standard cells and process parameters. The foundry also offers a special low-power process and standard cell library which exhibit half the power consumption. For comparison purposes it would be fair to claim a power efficiency of the mini-cores in the order of 11-26 μW/MIPS, and 13-31 μW/MMACs.

Table 8.4 shows a comparison with other designs reported in the literature. The Coyote DSP processor [61] is specifically designed for audio signal processing and low power consumption. It was originally implemented in a 0.50 μm CMOS process, but it has been re-implemented in 0.25 μm technology where it consumes 100 $\mu W/MIPS$ [5]. The authors of [58] achieve 210μW/MHz in a 0.35 μm dual V_t CMOS technology. The benchmark application consists of mainly MAC instructions. A 0.5 μm multi-threshold CMOS DSP by [63] offers 1.1 mW/MHz while running MACs. All these designs involve at least some full-custom layout, and can be characterized as "optimized" DSP's where an instruction typically involves one multiply-accumulate operation and some address pointer updating. Furthermore they all owe a great deal of their power efficiency to low-level full-custom circuit implementations.

To complete the picture we mention that an implementation of the Pleiades architecture achieves 10-100MOPS/mW [93], corresponding to 10-100 μW/MOPS, and that a hardwired fully synthesized hearing aid IC achieves 14.4 μW/MOPS [62]. For these designs it is rather unclear what is meant by an "instruction" or an "operation," and it is therefore unclear how to compare with our design. The mini-cores with a low power standard cell library consume approximately 11-26 μW/MIPS but they execute 2 times fewer instructions than a traditional DSP for

RTL level component	Power as %
Instruction memory	6.6
Instruction Decoder	5.5
Address generation unit	6.4
ALU	3.8
Coefficient memory	4.2
Data memory	18.7
Filter memory	5.1
Multiply-accumulate unit	49.7
Total	100

Table 8.5: Power breakdown figures for the FIR1 mini-core from the testchip.

the same task (table 8.2) hinting that 6-13 μW/MOPS is perhaps more realistic for comparison with [93, 62].

8.6 Interconnect network and idle power

Power consumption of the current bus-based interconnect network is approximately 8.1 μW while running the filter bank application. This corresponds to 9% of the total power consumption. The majority of the power consumed by the network is "idle power" and it stems from the distributed arbitration scheme where the token makes one round-trip through all interface units in every clock cycle. This is simple, but obviously not recommended in a real application. The idle power consumption of the interconnect network is 6.2 μW at 1 V at 1 MHz. As stated in section 6.4, for a reduced idle power consumption asynchronous solutions for the network have also been investigated [67].

We have also measured idle power consumption of the chip by running a test program that puts all the mini-cores in "sleep" mode. Mini-core "sleep" mode measurements report power consumption less than 1 μW. This supports the architecture concept as we envision even unused mini-cores in a SoC design, depending on the application. For this to work, idle power consumption of the mini-cores should be negligible.

8.7 Power consumption breakdown

Another way of evaluating the mini-core based platform is to look at the power breakdown figures of a single mini-core. The main idea behind customizing the

datapath for each mini-core was to reduce power consumed in fetching and decoding instructions by focusing on a specific class of algorithms.

Table 8.5 shows that only 12% of the total energy is being consumed by the fetching and decoding of instructions by the FIR mini-core design executing a typical filter program. This figure is determined by adding the *instruction memory* and *instruction decoder* entries of table 8.5. This is a promisingly low figure considering that power consumed during fetching and decoding instructions is a significant overhead associated with programmable architectures.

Another message is an expected one, that one should minimize the number of multiply-accumulates as well as the number of data memory write backs. These algorithmic level optimizations will result in huge power savings.

8.8 Summary

In this chapter, we tried to identify where the mini-core approach fits into power vs. flexibility trade-off curve of figure 1.1. For this purpose we provided full benchmark comparisons between the mini-cores and two commercial general-purpose DSPs: the TMS320C54x DSP chip and the $ARCTangent - A4^{TM}$ synthesizable core referred to as "the ARC core". We also provided full benchmark comparisons between the mini-cores and dedicated hardwired ASIC implementations designed by our industrial partner.

The results were encouraging. The prototype chip demonstrated a power consumption that is only 1.5 – 1.6 times larger than commercial hardwired ASICs and more than 6 – 21 times lower than current state of the art low power DSP processors.

Chapter 9

Conclusion

This thesis presented a novel approach to low-power *and* programmable DSP platform design for audio signal processing such as hearing aids. The proposed platform is a heterogeneous multiprocessor consisting of small and simple instruction set processors, *mini-cores* and DSP/CPU-cores that communicate using message passing.

Each mini-core is tailored to a particular class of algorithms from the application domain (FIR, IIR, LMS, etc.). The idea is to provide a platform in which energy-efficient mini-cores run the compute intensive parts of an application, while DSP/CPU-cores run less frequent, irregular, and control oriented parts.

This chapter will present the advantages of the mini-core approach, put the architecture in perspective and discuss future trends.

9.1 Advantages of the approach

9.1.1 Energy-efficient and programmable

The fact that the mini-cores are programmable within their corresponding algorithm suite gives complete programmability within the application domain. Differentiation of various products could be achieved by updating system software.

A prototype chip containing FIR, and IIR mini-cores has been designed to evaluate the platform. Results obtained from the prototype chip demonstrated that the power consumption of the mini-core based implementation is only 1.5 – 1.6 times larger than commercial hardwired ASICs and more than 6 – 21 times lower than current state of the art low-power DSP processors. This is due to: (1) the small size of the mini-cores and (2) a smaller instruction count for a given task.

Furthermore, the simple bus-based interconnect network used in the test chip

consumes 9% power of the total power while running the compute intensive filter bank application which is quite promising for an overall low power architecture.

In summary, the proposed heterogeneous multiprocessor platform consisting of our instruction set programmable mini-cores as well as one or more general purpose CPU and DSP cores offer both full programmability and a very low power consumption that approaches that of a hardwired ASIC.

9.1.2 Suitable for a SoC design flow

Another advantage of the mini-core approach is re-usability of the mini-cores. The mini-cores are parameterized in word-size, memory-size, etc. and can be instantiated according to the needs of the application at hand.

Furthermore, the introduction of a well-defined interface between the network and mini-cores has enabled “concurrent” engineering of a library of mini-cores and various network topologies. This approach fits nicely into a drop-and-use design flow where the SoC designer can freely select the required high-level intellectual property blocks (mini-cores, network topologies in this case). Using energy-efficient mini-cores as basic building blocks is the key to creating an overall low power system with reduced time-to-market.

9.2 Where does the mini-core approach fit in?

Chapter 3 presented a snapshot of the research both in industry and academia that focus on flexible and high-speed and/or low-power architectures. As the wireless communication market is growing immensely, DSP vendors are putting more "special" features onto the programmable DSPs that target this application domain. This means more domain-specific DSP processors are emerging. Among the examples are the C54x family from Texas Instruments, Lucent 16000 series, and the ADI21xx series from Analog Devices. Another trend within programmable DSP architectures is to provide more instruction level parallelism with multiple execution units (MACs, ALUs, etc.). The increased performance can be traded for low power by reducing supply voltage.

Another approach to domain-specific computing is the work that focuses on reconfigurable architectures. These architectures are evolving towards more FPGA-like structures consisting of coarse-grained heterogeneous configurable processing elements as basic building blocks. The interconnect network poses a serious problem to energy consumption in this approach [94] as general routability is required to handle various network configurations.

Finally, research in hardware/software co-design focus on early design exploration and providing guidance to the system-on-chip designer. Automated instruction set synthesis is among the hot topics [41, 24]. However, there is not yet a commercial tool available that can synthesize a low power programmable platform from scratch.

The mini-core platform is a multiprocessor architecture with several specialized mini DSP processors. It provides more parallelism than a single DSP processor. Furthermore, it is more efficient than a programmable DSP in terms of power consumption. The application domain covered in this thesis does not allow further voltage scaling. However, a mini-core system that has a balanced work-load on individual mini-cores provides potential for voltage scaling that can be applied in an appropriate application area.

Compared to reconfigurable architectures [10, 20], the mini-core approach shows similar and/or better energy-efficiency. However, reconfigurable architectures in general are more flexible as those architectures are "hardware" programmable (configurable) as well. But the highly flexible interconnect network that comes with reconfigurable hardware brings an overhead in terms of energy consumption. This may require full-custom design effort to get an acceptable energy-efficiency on the interconnect network [94]. A mini-core platform, on the other hand is synthesized with a suitable network that offers low power.

To conclude this section, the mini-core approach represents a viable alternative to platforms that use reconfigurable logic, and/or hardware accelerators. It also shows order of magnitude lower power consumption than programmable DSPs in general. Therefore it is a promising novel approach to programmable and low power platform design for application domains with moderate communication requirements.

9.3 Future trends

The hearing algorithm presented in chapter 4 is simple. Designing a system-on-chip using the mini-core approach for that particular application is quite straightforward.

However, more sophisticated algorithms are expected to appear in the future with more demanding computational and memory requirements, the question is then “Can this approach still be valid in designing hearing aid platforms?”. Several issues regarding the future perspectives will be discussed in this section.

9.3.1 Granularity of the mini-cores

Two issues regarding the granularity of the mini-cores will affect future systems: (1) memory size (2) data path.

Memory size of the mini-cores that are instantiated on the test chip currently match the hearing aid application requirements. However, memory requirement for future algorithms is likely to increase. The power breakdown of a typical FIR mini-core indicates 22% memory power consumption. This figure is achieved with a latch-based data memory that is 118×16 bits. A hard block RAM provided by Atmel for the same process technology is 1K byte i.e., it has slightly more than 4 times storage capacity than our latch-based memory [34]. It also consumes approximately 2.4 times higher power consumption for a similar read/write ratio.

The message here is that using hard block RAMs is advantageous in scaling the memory size of future mini-cores. Increasing the memory size of the FIR mini-core 4 times, would result in roughly 30% increase of the mini-core power consumption. This will still be well within the power consumption requirement limits for a mini-core.

A similar case exists for the IIR mini-core. The biquad register file for the largest IIR mini-core consists of 16 shift registers. Using that IIR mini-core to full capacity will require 16 clock cycles for 8 biquad section computation. This number of storage elements is enough for the benchmarks that were used for evaluation. However for a system clock rate of 1 MHz, the IIR mini-core will stay idle 70% of the sampling interval. For a better match between available system clock cycles and the functional capability of the IIR mini-core, the size of the register file should be increased. Because of the specialized memory structure for the biquad register file (a set of two-word shift-registers), a different solution other than using hard block RAMs should be investigated. A simple solution that comes to mind is for instance, to use a multi-bank memory architecture that consists of small and hence low power memory banks. Most significant bits of the address bus can be used to select the desired bank for data transfer.

The second issue is the granularity of the data path. The mini-core architecture suggests a custom data path for each algorithm domain. Another approach would be to use a unified data path that could be configured to behave as a desired mini-core. The unified data path would reduce design effort that has to be put in for each mini-core. This could be only done for a group of mini-cores that have common design requirements. However, this approach would lead to larger and less energy efficient mini-cores while reducing design effort. This would also be in contradiction with the idea of having specialized mini-cores at the first place. The overall architecture would be more flexible than the mini-core approach though, as this would mean flexibility in terms of both hardware configuration and software

programming.

9.3.2 Perspective regarding tools

Currently, the prototype chip is programmed using assembly language. An assembler that can handle both FIR and IIR mini-core programs has been implemented. However, future mini-core platforms will need tool assistance for handling more complex applications. There are two issues to investigate in the tool domain:

(1) Given a complex application, a problem would be to determine how many and which type of mini-cores are needed?

(2) Given a mini-core system i.e., fixed number and types of mini-cores how could one map a new application?

Both problems are hardware/software co-design related issues. Similar to those algorithms devised for high level synthesis and optimization problems such as resource allocation, and scheduling can be applied here [29].

9.3.3 Network implementation

As stated in section 8.6, the current bus-based network implementation suffers from a high idle power contribution. The arbitration scheme for the network is based on a token that circulates through all interface units in every clock cycle even when the network is idle. This inefficient distributed scheme could be replaced with a central arbiter that is only activated when more than one request occurs simultaneously.

Future network designs should optimize for low "idle" power consumption as the mini-core platform is envisioned to have moderate communication rate during operation.

9.4 Summary of the thesis

This thesis introduced a novel approach to programmable *and* low power platform design for audio signal processing, in particular hearing aids. The proposed programmable platform is a heterogeneous multi-processor architecture consisting of small and simple instruction set processors called *mini-cores* as well as standard DSP/CPU-cores that communicate using message passing.

The work has been based on a study of the algorithm suite covering the application domain. The observation of dominant tasks for certain algorithms (FIR, IIR, correlation, etc.) that require custom computational units and special data addressing capabilities lead to the design of low power mini-cores. The algorithm suite also consisted of less demanding and/or irregular algorithms (LMS, compression)

that required sub-sample rate signal processing justifying the use of a DSP/CPU-core.

Results obtained from the design of a prototype chip demonstrated the potential in creating low power systems consisting of reusable mini-core designs. To conclude, it can be said that this approach will play a key role in designing audio signal processing hardware in future.

A practical contribution of the thesis is the test bench implementation that equipped a standard desktop PC with sophisticated testing facilities. This was done by implementing a simple user interface program in C++ that communicates with the test bench hardware. A PCI-based FPGA board is used for this purpose.

Bibliography

[1] http://www.freehand-dsp.com.

[2] http://www.fishbeinhearingaids.com/hearing_aid_history.htm.

[3] http://www.hearingcenteronline.com/museum.shtml.

[4] http://www.gnresound.com/home.html.

[5] http://www.audiologic.com.

[6] L.S. Nielsen, Oticon A/S, personal communication.

[7] Xilinx Product Data Sheets. http://www.xilinx.com/partinfo/databook.htm.

[8] Özgün Paker. Low Power Audio Signal Processor. Master's thesis, Technical University of Denmark, June 1998.

[9] A. Dancy and A. Chandrakasan. Techniques for aggressive supply voltage scaling and efficient regulation. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 579 –586, May 1997.

[10] A. Abnous and J. Rabaey. "Ultra-Low-Power Domain-Specific Multimedia Processors". In *Proceedings of the IEEE VLSI Signal Processing Workshop*, pages 461–470, October 1996.

[11] M. Alidina, G. Burns, C. Holmqvist, E. Morgan, D. Rhodes, S. Simanapalli, and M. Thierbach. DSP16000: a high performance, low-power dual-MAC DSP core for communications applications. In *IEEE Custom Integrated Circuits Conference*, pages 119–122, 1998.

[12] A. Allan, D. Edenfeld, W. H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian. 2001 Technology Roadmap for Semiconductors. *IEEE Computer*, pages 42–53, January 2002.

[13] A.P. Chandrakasan and R.W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[14] A.P. Chandrakasan and R.W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. In *Proceedings of the IEEE*, 83(4), pages 498–523, April 1995.

[15] A.W.M. Van Den Enden and N.A.M Verhoeckx. *Discrete-Time Signal Processing*, chapter 8. Prentice Hall International, 1989.

[16] B. Gold and N. Morgan. *Speech and Audio Signal Processing*, chapter 14. John Wiley & Sons, 2000.

[17] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic. *Field-Programmable Gate Arrays*, chapter 1. Kluwer Academic Publishers, 1992.

[18] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, volume 21(8), pages 666–677, August 1978.

[19] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[20] V. Choudhary, A. van Wel, M. Bekooij, and J. Huisken. Reconfigurable Architecture for Multi Standard Audio Codecs. In *SoC2002*, April 2002.

[21] J.-G. Cousin, M. Denoual, D. Saille, and O. Sentieys. Fast ASIP synthesis and power estimation for DSP application. In *IEEE Workshop on Signal Processing Systems*, pages 591 –600, 2000.

[22] K. Danckaert, K. Masselos, F. Cathoor, H.J. De Man, and C. Goutis. Strategy for power-efficient design of parallel systems. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 7, pages 258 –265, June 1999.

[23] D.C. Chen and J.M. Rabaey. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. In *IEEE Journal of Solid-State Circuits*, volume 27, pages 1895–1904, December 1992.

[24] W. E. Dougherty, D. J. Pursley, and D. E. Thomas. Subsetting Behavioral Intellectual Property for Low Power ASIP Design. *Journal of VLSI Signal Processing*, 21(3):209–218, July 1999.

[25] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *IEEE*

Symposium on FPGAs for Custom Computing Machines, 1996., pages 157–166, 1996.

[26] E.A. Lee. Programmable DSP Architectures: Part I. In *IEEE ASSP Magazine*, volume 5, pages 4–19, October 1988.

[27] E.A. Lee. Programmable DSP Architectures: Part II. In *IEEE ASSP Magazine*, volume 6, pages 4–14, January 1989.

[28] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man. Global communication and memory optimizing transformations for low power signal processing systems. In *1994 Workshop on VLSI Signal Processing, VII.*, pages 178 –187, 1994.

[29] G. De Micheli. *Syhthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[30] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*, chapter 7. Addison-Wesley, 2000.

[31] G. Weinberger. The new millennium: Wireless Technologies for a Truly Mobile Society. In *IEEE International Solid State circuits Conference. Digest of Technical Papers*.

[32] A. Gatherer, T. Stetzler, M. McMahan, and E. Auslander. DSP-based Architectures for Mobile Communications: Past, Present and Future. *IEEE Communications Magazine*, 38(1):84–90, January 2000.

[33] G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, April 19 1965. Also available at http://www.intel.com/research/silicon/moorespaper.pdf.

[34] T. Gleerup. DSP2a in a 0.25-micron Technology - Results, Experiences, and Future Challanges. Technical report, Oticon A/S, 2002.

[35] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnuch, D. Sweely, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, 24(1):81–89, 1991.

[36] G.R. Goslin. A Guide to Using Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance. Technical report, Xilinx Inc., 1995. Xilinx Application Notes.

[37] H. J. M. Veendrick. Short-Circuit Dissipation of Static CMOS Circuitry and its Impact on the Design of Buffer Circuits. *IEEE Journal of Solid-State Circuits*, pages 468–473, August 1984.

[38] Niels Handbæk. Design and VLSI implementation of a dedicated low-power DSP circuit. Master's thesis, Technical University of Denmark, 2000.

[39] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1338 –1354, November 2001.

[40] Optimized DSP Library for C Programmers on the TMS320C54x. Application report, Texas Instruments, January 2000.

[41] Ing-Jer Huang and A.M. Despain. Synthesis of application specific instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):663 –675, June 1995.

[42] Mogens Isager. Block Level Interconnect Structures for Low-Power DSP Chips. Master's thesis, Technical University of Denmark, 2000.

[43] J. Bradley. Calculation of TMS320C5x Power Dissipation. Technical report, Texas Instruments, 1993.

[44] J. D. Meindl. A History of Low PowerElectronics: How It Began and Where It's Headed. In *International Symposium on Low Power Electronics and Design*, pages 149–151, August 1997.

[45] J. Eyre and J. Bier. The evolution of DSP processors. *IEEE Signal Processing Magazine*, 17(2):43–51, March 2000.

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

[47] J. Sparsø and S. Furber, editors. *Principles of asynchronous circuit design - A systems perspective.* Kluwer Academic Publishers, 2001.

[48] H.M. Jacobson and G. Gopalakrishnan. Application-specific programmable control for high-performance asynchronous circuits. *Proceedings of the IEEE*, 87(2):319–331, February 1999. Special issue on "Asynchronous Circuits and Systems" (Invited Paper).

[49] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[50] K. Roy. Leakage Power Reduction in Low-Voltage CMOS Designs. In *IEEE International Conference on Electronics, Circuits and Systems*, volume 2, pages 167–173, September 1998.

[51] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu. A 3.8-ns CMOS 16x16-b Multiplier Using Complementary Pass-transistor logic . *IEEE Journal of Solid-State Circuits*, 25(2):388 –395, April 1990.

[52] T. Kumura, D. Ishii, M. Ikekawa, I. Kuroda, and M. Yoshida. A low-power programmable DSP Core architecture for 3G Mobile terminals. In *IEEE InternationalConference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1017–1020, 2001.

[53] L. Wei, Z. Chen, K. Roy, M.C. Johnson, Y. Ye, and V.K. De. Design and optimization of dual-threshold circuits for low-voltage low-power applications . In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 7, pages 16–24, March 1999.

[54] D. Lai, Q. Lin, S. Chen, and M. Margala. A low-power DSP Core for an embedded MP3 Decoder. In *IECON '01. The 27th Annual Conference of the IEEE Industrial Electronics Society*, volume 3, pages 1892–1897, 2001.

[55] H. Lange, O. Franzen, H. Schröder, M. Bücker, and B. Oelkrug. Reconfigurable Multiply-Accumulate-based Processing Element. In *SoC2002*, April 2002.

[56] M. Lee, H. Singh, G. Lu, N. Bagherzadeh, F.J. Kurdahi, E. M. C. Filho, and V. C. Alves. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *Journal Of VLSI Signal Processing-Systems for Signal, Image Video Technology*, 24(2):147–164, March 2000.

[57] T. Andy Lee, Donald C. Cox, James Nichols, and Saf Asghar. "Low Power Reconfigurable Macro-Operation Signal Processing for Wireless Communications". In *48th IEEE Vehicular Technology Conference*, volume 3, pages 2560–2564, May 1998.

[58] W. Lee and et al. "A 1-V Programmable DSP for Wireless Communications". *IEEE Journal of Solid State Circuits*, 32(11):1766 –1776, November 1997.

[59] T. Lunner and J. Hellgren. A digital filterbank hearing aid – design, implementation and evaluation. In *Proceedings of ICASSP'91*, pages 3661–3664, Toronto, Canada, 1991.

[60] R. Mehra, D.B. Lidsky, A. Abnous, P.E. Landman, and J.M. Rabaey. *Low power design methodologies*, chapter Algorithm and Architectural Level Methodologies for low power. Kluwer Academic Publishers, 1996.

[61] F. Møller, N. Bisgaard, and J. Melanson. "Algorithm and Architecture of a 1V Low Power Hearing Instrument DSP". In *International Symposium on Low Power Electronics and Design*, pages 7–11, August 1999.

[62] P. Mosch, G. Van Oerle, S. Menzl, N. Rougnon-Glasson, K. Van Nieuwenhove, and M. Wezelenburg. "a 720 μW 50 MOPs 1V DSP for a Hearing Aid Chip Set". In *Proceedings ISSCC 2000*, pages 238–239, February 2000.

[63] S. Mutoh and et al. "A 1-V Multithreshold-Voltage CMOS Digital Signal Processor for Mobile Phone Application". *IEEE Journal of Solid State Circuits*, 31(11):1795–1802, November 1996.

[64] N. H. E. Weste and K. Eshraghian. *Principles Of CMOS VLSI Design, A systems perspective*. Addison-Wesley, Second edition, 1993.

[65] L.S. Nielsen and J. Sparsø. An 85 μW Asynchronous Filter-Bank for a Digital Hearing Aid. In *Proc. IEEE International Solid State circuits Conference*, pages 108–109, 1998.

[66] L.S. Nielsen and J. Sparsø. Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268–281, February 1999. Special issue on "Asynchronous Circuits and Systems" (Invited Paper).

[67] S.F. Nielsen and J. Sparsø. Analysis of low-power SoC interconnection networks. In *IEEE 19th Norchip Conference*, pages 77–86, November 2001.

[68] OCP International Partnership. Open Core Protocol Specification, 2001. Release 1.0.

[69] H. Okuhata, M.H. Miki, T. Onoye, and I. Shirakawa. A low-power DSP Core architecture for low bitrate speech codec. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 3121–3124, 1998.

[70] Ö. Paker and J. Sparsø. A heterogeneous multi-core platform for low power signal processing in systems-on-chip. In *IEEE Workshop on Heterogeneous reconfigurable Systems on Chip, Chances, Applications, Trends*, April 2002.

[71] Ö. Paker, J. Sparsø, N. Haandbæk, M. Isager, and L. S. Nielsen. A heterogenous multiprocessor architecture for low-power audio signal processing. In A. Smailagic and H. De Man, editors, *IEEE Computer Society Workshop on VLSI*, pages 47–53, April 2001.

[72] Ö. Paker, J. Sparsø, N. Haandbæk, M. Isager, and L. S. Nielsen. A heterogeneous multi-core platform for low power signal processing in systems-on-chip. In *European Solid-State Circuits Conference*, September 2002. To appear.

[73] Ö. Paker, J. Sparsø, N. Haandbæk, M. Isager, and L. S. Nielsen. A low-power heterogenous multiprocessor architecture for audio signal processing. *Journal of VLSI Signal Processing*, 2002. To appear in 2003.

[74] C. Piguet. *Low Power Design in Deep Submicron Electronics*, chapter 9, Microprocessor Design. NATO ASI series. Kluwer Academic Pubilishers, 1997.

[75] J.G. Proakis and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Prentice-Hall, 3 edition, 1996.

[76] Qi Wang and S.B.K. Vrudhula. Algorithms for minimizing standby power in deep submicrometer, dual-Vt CMOS circuits . In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, pages 306 –318, March 2002.

[77] J. Rabaey. "Reconfigurable Processing: The Solution to Low Power programmable dsp". In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 275–278, 1997.

[78] S. Ramanathan, S.K. Nandy, and V. Visvanathan. Reconfigurable Filter Coprocessor Architecture for DSP Applications. *Journal Of VLSI Signal Processing-Systems for Signal, Image Video Technology*, 26(3):333–359, November 2000.

[79] S. Thompson, I. Young, J. Greason, and M. Bohr. Dual Threshold Voltages And Substrate Bias: Keys To High Performance, Low Power, 0.1μm Logic Designs. *1997 Symposium on VLSI Technology, Digest of Technical Papers.*, pages 69–70, 1997.

[80] S.F. Ali. FPGA Based ASIC Chip test system. Technical report, Institute of Mathematical Modelling, 2001.

[81] Simon Haykin. *Adaptive Filter Theory*. Prentice-Hall International, Inc, 1996.

[82] R. Subramanian, U. Jha, J. Medlock, C. Woodthorpe, and K. Rieken. Novel Application-Specific Signal Processing Architectures for Wideband CDMA and TDMA Applications. In *IEEE 51st Vehicular Technology Conference Proceedings*, volume 2, pages 1311–1317, 2000.

[83] T. Miyamori and U. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1998*, pages 2–11, 1998.

[84] T. Van Achteren, G. Deconinck, F. Catthoor, and R. Lauwereins. Data reuse exploration techniques for loop-dominated applications. In *Design, Automation and Test in Europe Conference and Exhibition, 2002.*, pages 428 –435, 2002.

[85] R. Tessier and W. Burleson. Reconfigurable Computing for Digital Signal Processing. *Journal of VLSI Signal Processing*, 28(1-2):7–27, May-June 2001.

[86] C. Turner. "Calculation of TMS320LC54x Power Dissipation". Application report, Texas Instruments, 1997. http://www-s.ti.com/sc/psheets/spra164/spra164.pdf.

[87] Udo Zölzer. *Digital Audio Signal Processing*. Wiley, 1997.

[88] V. De and S. Borkar. Technology and design challenges for low power and high performance [microprocessors] . In *International Symposium on Low Power Electronics and Design*, pages 163–168, 1999.

[89] I. Verbauwhede and M. Touriguian. A low power DSP Engine for Wireless Communications. *Journal Of VLSI Signal Processing-Systems for Signal, Image Video Technology*, 18(2), February 1998.

[90] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[91] W. Strauss. Digital Signal Processing: The new semiconductor industry technology driver. *IEEE Signal Processing Magazine*, 17(2):52–56, March 2000.

[92] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high-performance circuits . In *Digest of Technical Papers. 1998 Symposium on VLSI Circuits*, pages 40–41, 1998.

[93] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. Rabaey. "A 1-V Heterogenous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing". *IEEE Journal of Solid State Circuits*, 35(11):1697–1704, November 2000.

[94] H. Zhang, M. Wan, V. George, and J. Rabaey. "Interconnect Architecture Exploration for Low Energy Reconfigurable Single-Chip DSPs". In *IEEE Computer Society Workshop On VLSI'99*, pages 2–8, April 1999.