Norwegian University of
Science and Technology

ULTRA LOW POWER APPLICATION SPECIFIC INSTRUCTION-SET PROCESSOR DESIGN

for a cardiac beat detector algorithm

Yahya H. Yassin

Master of Science in Electronics

Submission date: June 2009

Supervisor: Per Gunnar Kjeldsberg, IET

Co-supervisor: Jos Hulzink, IMEC-NL

Jos Huisken, IMEC-NL

Norwegian University of Science and Technology

Department of Electronics and Telecommunications

Problem Description

Topic: Application Specic Instruction-set Processor (ASIP) design and application mapping of a biomedical application.

Focus: Computer architecture/arithmetic, Processor design, Continuous wavelet transform (CWT)

IMEC-NL researches and develops wireless autonomous transducer solutions for Wireless Sensor Nodes (WSN) of which each component (radio, digital signal processors (DSPs), sensors, and actuators) is tailored to meet the tight power and performance constraints. The DSPs developed at IMEC-NL process the sensor node data locally, hence reducing power costly wireless data communication in order to minimize energy dissipation of the entire WSN. For this reason, the DSPs must be designed to run a specific application as efficient as possible.

The assignment targets the design of an ASIP with Target processor design tools, and the research and mapping of a biomedical application on this processor. This task includes conversion of Matlab and/or generic C-code to platform related code, mapping it on a generic DSP, and re-target the DSP for the biomedical application.

Assignment given: 12. January 2009
Supervisor: Per Gunnar Kjeldsberg, IET

Abstract

High efficiency and low power consumption are among the main topics in embedded systems today. For complex applications, off-the-shelf processor cores might not provide the desired goals in terms of power consumption.

By optimizing the processor for the application, or a set of applications, one could improve the computing power by introducing special purpose hardware units. The execution cycle count of the application would in this case be reduced significantly, and the resulting processor would consume less power.

In this thesis, some research is done in how to optimize a software and hardware development for ultra low power consumption. A cardiac beat detector algorithm is implemented in ANSI C, and optimized for low power consumption, by using several software power optimization techniques. The resulting application is mapped on a basic processor architecture provided by Target Compiler Technologies. This processor is optimized further for ultra low power consumption by applying application specific hardware, and by using several hardware power optimization techniques.

A general processor and the optimized processor has been mapped on a chip, using a 90 nm low power TSMC process. Information about power dissipation is extracted through netlist simulation, and the results of both processors have been compared. The optimized processor consume 55% less average power, and the duty cycle of the processor, i.e., the time in which the processor executes its task with respect to the time budget available, has been reduced from 14% to 2.8%. The reduction in the total execution cycle count is 81%.

The possibilities of applying power gating, or voltage and frequency scaling are discussed, and it is concluded that further reduction in power consumption is possible by applying these power optimization techniques. For a given case, the average leakage power dissipation is estimated to be reduced by 97.2%.

Preface

During the work in this project, I got the opportunity to work with IMEC-NL, where I met very talented engineers and researchers. They gave me the freedom to use state of the art development tools, and to be involved in one of their research projects.

First of all, I want to thank my professor, Per Gunnar Kjeldsberg, for recommending me to some of the master thesis projects from IMEC-NL. I would also like to thank him for an inspirational guidance throughout the project, which resulted in two articles submitted to different conferences.

Further, I would like to thank Jos Hulzink, MSc, and Serdar Yildirim, BSc, for giving me very good guidance throughout this project, and for providing me the necessary training in order to quickly start using the development tools. I also want to thank them for reviewing my thesis, and for giving me good feedbacks.

I would also like to thank

– Iñaki Romero Legarreta, PhD, for the helping me understand his algorithm, and for the help he provided during the Matlab to C conversion process.

– Bernard Grundlehner, MSc, for the help in understanding some specific details related to the provided Matlab model.

– Marjam Ashouei, PhD, for some help related to the software development.

– Filipa Duarte, PhD, for the help she gave me in the creation of low power memories.

– Michael De Nil, MSc, for providing me with a memory wrapper template that I used in order to create the memories in the ASIP design.

– Ben Busze, PhD, for all the help related to synthesis, place and route, and netlist generation.

– Ashoka Sathanur, PhD, and Frank Bouwens, MSc, for the help related to power gating, and voltage scaling analysis.

– Jos Huisken, MSc, for the ideas he shared with me and for reviewing my thesis.

I want to thank Target Compiler Technologies as well for providing superb support services, and for making development tools that makes our lives easier.

In general, I want to thank the DSP team and IMEC-NL for the warm welcome and all the efforts they made to make this master thesis assignment a pleasurable experience. They definitely inspired me to continue with research in the future.

Contents

Abstract **v**

Preface **vii**

List of abbreviations **xix**

1 Introduction **1**
1.1 The goal of ASIP design 2
1.2 Motivation and goal of work 4
1.3 Outline 5
1.4 Main contributions 5

2 The cardiac beat detection algorithm **7**
2.1 The heart and ECG 7
2.2 The wavelet transform 9
2.3 The beat detection algorithm 11
2.4 Optimization of the algorithm 13
2.5 The test databases 15
2.5.1 The MIT/BIH database 15
2.5.2 The database recorded by IMEC-NL 16

3 Energy-efficient SW and HW development **19**
3.1 Software code optimization techniques 20
3.2 Hardware optimization techniques 22
3.3 Power management techniques 24
3.3.1 Dynamic power management (DPM) 24
3.3.2 Component-level power management 24
3.4 Converting floating point to fixed point 25
3.5 Profiling 27
3.6 The energy model 27

4 Matlab to C conversion **31**
4.1 The flow of work 31
4.2 Matlab code analysis 32
4.2.1 The CWT function 34
4.2.2 The QRSDet3 function 34
4.2.3 The QRSDet function 34
4.3 The Matlab prototype 35
4.4 Simulation interface and validation 35
4.5 The floating point application 36

4.5.1 Implementation of the floating point application . . . 37
4.5.2 Optimization of the floating point application . . . 38
4.6 The fixed point application . . . 39
4.6.1 Changes to the floating point application . . . 40
4.6.2 Optimization of the fixed point application . . . 43
4.6.3 Validation of the fixed point application . . . 48
4.7 Results . . . 48
4.7.1 Simulation results . . . 48
4.7.2 Profiling results . . . 50
4.7.3 Application specifications for processor implementation . . . 53
4.8 Discussion of the results and conclusions . . . 54

5 The Target Development tools 57
5.1 Introduction to the HW/SW Co-design methodology . . . 57
5.2 ASIP design with Target . . . 58
5.2.1 About Target . . . 59
5.2.2 ASIP design with Target . . . 59
5.3 The nML language . . . 61

6 Application mapping and ASIP design 63
6.1 The embedded environment of the processor . . . 63
6.2 Platform architecture of the BASE core . . . 64
6.3 Implementation strategy . . . 67
6.4 Validation strategy . . . 67
6.4.1 Self-checking simulation . . . 67
6.4.2 Simulation interface with TCL . . . 68
6.5 Processor core modifications and optimizations . . . 69
6.6 Target profiling results . . . 75
6.7 VHDL generation . . . 77
6.8 Place and route . . . 80
6.9 Extraction of power numbers . . . 81
6.10 Simulation and power consumption results . . . 81
6.10.1 Register transfer level (RTL) and netlist simulation results . . . 81
6.10.2 Power numbers . . . 82
6.11 Discussion of the results and conclusions . . . 84

7 Conclusions 87

8 Future work 89

Appendices

A Analysis of the original Matlab model 91
A.1 The CWT function . . . 91
A.2 The QRSDet3 function . . . 93

B Detailed description of the Matlab prototype 101
B.1 Changes to the CWT step and modulus maxima step . . . 101
B.2 Changes to the step for finding clusters of beats . . . 101
B.3 Changes to the classification of beats . . . 102

C Calculation of convolution iteration times **105**

D Solving issues related to low threshold values **109**

Bibliography **111**

List of Tables

2.1 Chosen signals from the MIT/BIH database 16
2.2 IMEC ECG database, [RLG08a] 16

4.1 Memory usage and cycle count with different frequencies of the ECG signal. . 53
4.2 Application specifications for processor implementation. 54

6.1 Truth table for the memory enable signal of DM. 78

List of Figures

1.1 General ASIP design flow. 3

2.1 The heart 8
2.2 Beat demonstration 8
2.3 P,Q,R,S and T waves 9
2.4 QRS complex morphologies 9
2.5 The Mexican hat wavelet 11
2.6 The first algorithm, [RLAR+05]. 12
2.7 The overlap window 13
2.8 The optimized algorithm. 15

3.1 Dominant power sources for a processor. 28

4.1 An overview of the steps in the conversion process. 32
4.2 The Matlab model structure. 33
4.3 The C code structure. 36
4.4 Special case of an ECG sample. 38
4.5 Memory usage with respect to $FACTOR$ and $ECG_FS = 1000$. 46
4.6 Memory usage with respect to ECG_FS and $FACTOR = 15$. 47
4.7 The MIT/BIH test over all thresholds. 49
4.8 The IMEC test over all thresholds. 49
4.9 Noise tests for th=30%. 50
4.10 The sample of an ECG signal used for profiling. 50
4.11 The second beat detected by the sample in Figure 4.10. 51
4.12 The call graph with the focus on the convC function. 53

5.1 An outline of the Chess/Checkers environment. 60

6.1 Processing interval of the processor in a real-time environment. 64
6.2 The BASE processor core architecture. 65
6.3 Instruction pipeline. 66
6.4 ASIP implementation strategy. 68
6.5 Initial instruction-set of the 32-bit processor. 70
6.6 Initial simulation without the MAC unit. 71
6.7 Simulation with the resulting MAC unit. 71
6.8 The new parallel instruction assembly code. 72
6.9 The new parallel instruction. 73
6.10 The new parallel load and MAC instruction. 73
6.11 The new parallel load, equal and select instruction. 74
6.12 The optimized BASE architecture. 76
6.13 Improvement in execution cycles after every optimization step. 76

6.14 A memory block from VLC. 77
6.15 The memory wrapper of DM. 78
6.16 The top module. 79
6.17 How the memory blocks were placed on the chip. 80
6.18 How the delay of the circuit varies with different supply voltages. 81
6.19 Results from the netlist simulation with *ncsim*. 82
6.20 Total average power consumption of the processors. 82
6.21 Average leakage power consumption of the processors. 83
6.22 Average dynamic power consumption of the processors. 83
6.23 Total average power consumption of the processors compared together. 83
6.24 Total average power consumption of the processors. 84

A.1 Illustration of the cwt2 function. 92
A.2 Creation of the modulus maxima matrices. 94
A.3 Modulus maxima above the threshold. 94
A.4 Finding clusters of beats. 95
A.5 Matlab indexing of matrices. 96
A.6 Classification of beats. 97
A.7 The hanning window by definition. 98
A.8 The modified hanning window. 98

B.1 Finding clusters of beats in the prototype. 102
B.2 Finding valid clusters of beats in the prototype. 103

Source code

4.1 Code before optimization . . . 39
4.2 Code after optimization . . . 39
4.3 Code consideration of fixed point addition in a loop . . . 41
4.4 Code before fixed point conversion . . . 41
4.5 Code using solution 1 based on Code Snippet 4.7 . . . 41
4.6 Code using solution 2 based on Code Snippet 4.7 . . . 42
4.7 Code before fixed point conversion . . . 42
4.8 Fixed point round operations. . . . 42
4.9 Floating point elimination . . . 43
4.10 Floating point cosine calculation. . . . 43
4.11 Fixed point cosine calculation. . . . 43
4.12 Calculating the alternative lookup table in Matlab . . . 44
4.13 Optimization of the assignment of *__psi__scale.array*. . . . 44
4.14 Fixed point cosine calculation. . . . 45
4.15 Global define parameters in global.h. . . . 47
4.16 Pre-processor macro optimization of the *abs* function in C. . . . 47
4.17 Call graph results from *gprof*. . . . 52
5.1 AND-rule . . . 61
5.2 OR-rule . . . 61
5.3 A basic ALU instruction . . . 62
6.1 32-bit MAC unit . . . 71
6.2 32-bit MAC unit after modification. . . . 71
6.3 Direct assignment to CM. . . . 72
6.4 Parallel load instruction. . . . 72
6.5 Parallel load and MAC instruction. . . . 74
6.6 Parallel load, equal and select instruction. . . . 75
6.7 A *nop* and select instruction. . . . 75
C.1 Matlab code for convolution by the original definition. . . . 105
C.2 C code for convolution by the Matlab definition. . . . 107
D.1 The code before inserting the memory protection. . . . 109
D.2 Memory out of bounds protection . . . 109

List of abbreviations

ADC Analogue to Digital Converter

ADL Architecture Description Language

ALU Arithmetic-Logic Unit

ASA Application Specific Architectures

ASIC Application Specific Integrated Circuits

ASIP Application Specific Instruction-set Processor

AV Atrio-Ventricular

CCU Coronary Care Unit

CM Coefficient Memory

CPU Central Processing Unit

CWT Continuous Wavelet Transform

CWTMM Continuous Wavelet Transform Modulus Maxima

DM Data Memory

DPM Dynamic Power Management

DSP Digital Signal Processor

DTSE Data Transfer and Storage Exploration

DVFS Dynamic Voltage and Frequency Scaling

ECG Electrocardiogram

EMG Electromyography

GCC GNU C Compiler

GNU Gnu's Not Unix

HDL Hardware Description Language

HW/SW Co-design Hardware/Software Co-design

IC Integrated Circuit

IMEC-NL Interuniversity Microelectronics Centre Netherlands

ISA Instruction-Set Architecture

ISP Instruction-Set Processor

ISS Instruction-Set Simulator

MAC Multiply-Accumulate

MSB Most Significant Bit

PC Program Counter

PCU Program Control Unit

PDG Primitives Definition and Generation

PM Program Memory

RAM Random Access Memory

RPM Rounds Per Minute

RTL Register Transfer Level

SA Sino-Atrial

SNR Signal-to-Noise Ratio

SoC System on Chip

SP Stack Pointer

SPEF Standard Parasitic Exchange Format

TCL Tool Command Language

TSMC Taiwan Semiconductor Manufacturing Company Limited

ULP Ultra Low Power

VCD Value Change Dump

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLC Virage Logic Corporation

WSN Wireless Sensor Nodes

Chapter 1

Introduction

Cardiovascular disease is one of the most common cause of death in Western society, [LRW+03]. A Cardiovascular disease refers to the class of diseases that involve the heart or blood vessels (arteries and veins), [Wik09a] and [MHM+93]. In order to diagnose such diseases, an analysis of the heart rate is necessary, and the physiologic signal used to determine the heart rate is the electrocardiogram (ECG). By processing the ECG signal, it is possible to detect heart beats, and one can also analyze the ECG in order to classify cardiovascular diseases. Therefore, the algorithm processing the ECG signal must be as accurate as possible. Many approaches have been proposed for automatic beat detection algorithms, i.e., ECG beat detection using filter banks [ATNL99], zero-crossing [KHO03], signal derivatives [FN06], digital filters [KKA97], algorithms from the field of artificial neural networks [BFDVS+98], and algorithms using wavelet transforms [LZT95], [RLAR+05].

The requirements for an embedded device implementing applications with such algorithms, introduce several challenges that have to be solved. This is specially the case for ambulatory monitoring. Afonso et al. [ATNL99] points out that the ECG is prone to various types of noise, and it is important to reduce the noise without distorting the morphology of the ECG. However, the task is not easy, because the appearance of heartbeats varies considerably [BFDVS+98]. This variation is not only limited to different patients, but also appears as the consequences of movement, modifications of electrical characteristics of the body, etc. For such applications, embedded devices usually need processors with high performance.

According to Kucukcakar et al. [Kuc99], the use of off-the-shelf processors and memories is a common way to achieve the best time-to-market and the lowest cost. In addition, it is easy for designers to find the right processor with average performance and power consumption for an application. However, if the performance bottlenecks of the application can not be avoided through software re-organization, then a more complex processor is needed. This usually means upgrading to a higher performance processor, which most often result in higher cost and energy consumption. For applications implementing advanced heart beat detectors, this is usually the case. Instead of upgrading to a higher performance processor, one could design an Application Specific Architecture (ASA), specialized for the application. ASA refers to instruction-set processors (ISP), digital signal processors (DSPs) and integrated circuits (ICs) designed for a specific application. What kind of architecture is needed depends on the application. The use of an Application Specific Integrated Circuit (ASIC) gives the best solution for an application. However, if a newer version of the application is designed, then a complete new ASIC must be made. A more flexible solution is the use of Application Specific Instruction-set Processors (ASIP).

In embedded systems design today, there is need for high efficiency, as well as low

power consumption [KAFK+05], [GM01]. In recent years, ASIPs have been popular because they offer efficiency and short design cycles at the same time. In contrast to off-the-shelf processor cores, ASIPs show dedicated functional units and machine instructions that speed up execution of the "hot spots" in a given application [KAFK+05].

In this thesis, an Ultra Low Power (ULP) ASIP will be designed for a biomedical application, using an algorithm developed by Romero et al., [RLAR+05]. In this algorithm, the problem of automatic beat recognition in the ECG is tackled using a mathematical method called Continuous Wavelet Transform Modulus Maxima (CWTMM), [RLAR+05].

In a simulation environment, this algorithm produced very good results with world-wide known databases such as the MIT/BIH database. The algorithm has been further optimized to work in real time for implementation in an ambulatory ECG monitor by the Interuniversity Microelectronics Centre Netherlands (IMEC-NL), [RLG08a]. IMEC-NL is an independent micro- and nanoelectronics research center located in Eindhoven, Netherlands. This research center focuses on next generation electronics research, three to ten years ahead of industrial needs.

The purpose of this chapter is to provide the reader with the necessary background information for ASIP design in order to understand the work done in this master thesis.

1.1 The goal of ASIP design

According to [RAP08], an ASIP is a microprocessor specialized for an application. Its instruction set is designed from scratch or extended from a known microprocessor, in order to optimize the performance of the application. Application Specific Architectures (ASA) like ASIPs are able to reach a better cost and performance ratio by exploiting the characteristics of the application they are dedicated to [DGF90], using design methodologies like hardware/-software co-design (HW/SW Co-design). The computing power is improved by introducing special purpose hardware units, exploiting the parallelism available in the application. They can also be cost effective by saving hardware resources. This is done by only implementing the resources needed by the application.

The concept of ASIPs is to fill the gap between Application Specific Integrated Circuits (ASIC) and Digital Signal Processors (DSP), [RAP08]. ASICs are highly efficient, but they lack the flexibility. On the other hand, DSPs provide software reusability with less performance and energy efficiency as compared to ASICs, [RAP08].

In general, the design of an ASA is not an easy task, and requires the scanning of the architectural space to discover the most useful choices which could efficiently support the applications features. In addition, the development requires complex tools to evaluate and provide analysis of the ASA development, and hence guide the designer towards correct design choices.

Power and performance optimization could be applied at the logic level, register transfer level (RTL), in the ASIP software, and also at system level [GM01]. De Gloria et al. [DGF90] defines some main requirements for the design of an ASA, and in [JBK01], five general steps in ASIP synthesis are presented. Different design flows for ASIP design exists, such as the methodology presented in [Kuc99], and the ASSIST methodology [KJBK04]. Those methodologies differ from one another on a detailed level. However, they are all similar to the design flow presented in [JBK01]. The rest of this section will present this design flow.

The application is first provided with its design constraints. In order to design an ASIP, it is necessary to understand the behavior of the application. The application must be analyzed statically and dynamically to get the desired characteristics and requirements which can guide the hardware synthesis as well as instruction-set generation.

The next step is to identify possible architectures that meet the application constraints. In addition, the performance of possible architectures is estimated, and the architectures satisfying the performance and power constraints are chosen.

In the third step, the instruction set is generated for that particular application, and for the architectures selected. This is needed for the synthesis steps. A compiler generator or a re-targetable code generator is used in the fourth step to synthesize code for the particular application, or for a set of applications. The re-targetable compiler includes the steps of instruction mapping, resource allocation, and scheduling of the application.

In the final step, the hardware is synthesized using the ASIP architectural template and instruction set architecture starting from a description in VHDL or Verilog. Figure 1.1 summarizes the general design flow presented in [JBK01].

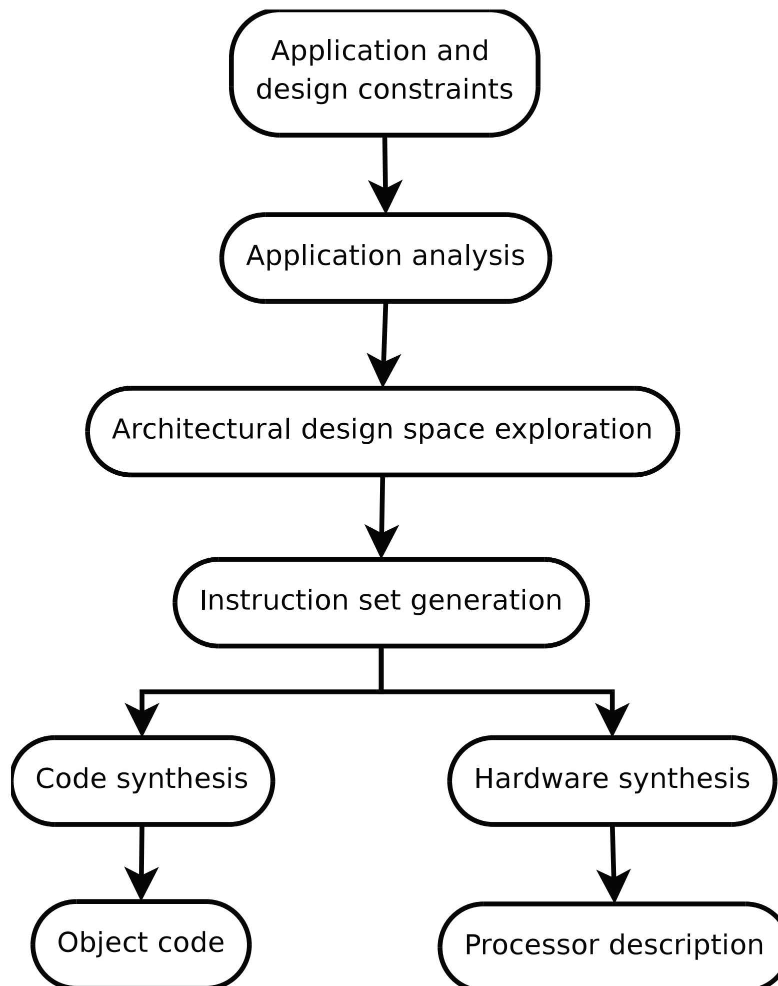

Figure 1.1: General ASIP design flow based on the one presented in [JBK01].

All these steps are more or less guidelines for the development of an ASIP. If all time constraints are fulfilled, then additional optimizations can be introduced to increase power efficiency. Glokler et al. [GM01] distinguish the energy required to execute a given algorithm in hardware in three parts; intrinsic energy, routing energy, and overhead energy.

The intrinsic energy refers to the minimum required processing energy to execute a computational part, i.e., an arithmetic operation. This energy only depends on the operations

of an algorithm and the process technology. The energy is constant as long as the algorithm remains the same.

Routing energy is the energy needed to move data in the chip. For instance, memory accesses requires energy to transfer the data to local registers. As long as changes to the architecture have a limited impact on data routing, the routing energy does not change significantly.

The remaining energy is referred to as the overhead energy, i.e., caused by wasteful logic activity, control activity, etc. Performance optimization for Instruction-Set Architectures (ISA) typically also reduce overhead energy due to the decrease of the processor runtime, because the overhead power remains nearly constant.

The ASIP design flow presented in this section will be used as a guideline for the ASIP design done in this thesis.

1.2 Motivation and goal of work

The purpose of this project is to develop knowledge and experience on how to design an ASIP for ultra low power (ULP), and map a specific application into it. In this case the application is a heart beat detector designed by Romero et al. [RLAR+05], based on the CWTMM. This application process the ECG signal by analyzing every 2 seconds at a time. In a real-time environment, every 2 seconds of an ECG signal must be sampled, and buffered before the signals are processed. The real-time requirement for this application is to finish processing this 2 second buffer before the next buffer is ready. This gives a processing deadline of 2 seconds.

Before an algorithm can be mapped into a processor one has to go through several design steps, as presented in Section 1.1. In this case, the algorithm is modeled in Matlab, using built-in Matlab functions which do not exist in programming languages like C. Therefore, the first step would be to convert the code into an embedded software. In this thesis the program language C will be used, because it is a common language used by many processor developers. In addition, C is supported among various development tools for processor design. During the conversion, the algorithm must be tested using good test benches, i.e., with test signals from known databases. In order to increase efficiency, all floating point signals should be converted into fixed point, and heavy computational parts of the code like square roots and divisions must be optimized away if possible. By doing so, one can eliminate the need for a floating point emulator on the processor, in addition to complex logic. This would result in less power consumption. After the embedded software is optimized, the memory resources can be determined. This will be done by profiling.

After finishing the software optimizations, the work on an energy efficient processor architecture will start. Based on the constraints and optimizations made from the embeddable software application, an ASIP will be designed. This step involves architecture exploration and mapping of the software application through an iterative process. In this thesis the development tools from Target will be used. An existing basic processor architecture will be used as a base, and this architecture will be modified in order to optimize the biomedical application for ultra low power. The work on this project will include research on how to design the software and an ASIP for ULP. Design methodologies like hardware/software co-design (HW/SW Co-design) will also be used.

1.3 Outline

Previous work associated with the biomedical application is presented in Chapter 2, in order to understand the given Matlab model. Theory related to the development of energy-efficient software and hardware is presented in Chapter 3, as well as some hardware and software optimization techniques that could be applied to reduce the power consumption of a processor. In the Chapter 4, the analysis of this Matlab model will be presented, including the work done to convert and optimize the application for an embedded environment. The results from the profiling and from the databases will be presented and discussed. In Chapter 5, the development tools used in this project will be introduced, before the work on the ASIP is presented in Chapter 6. The conclusions of this project is given in Chapter 7, before presenting future work objectives in Chapter 8.

1.4 Main contributions

- Converted the Matlab model of the algorithm, designed by Romero et al. [RLAR+05], into a floating point C application.
- Converted the floating point C application to a fixed point application.
- Optimized the application further by using lookup tables, inline functions, reducing read and write operations to global memories, eliminating divisions with constant values where it is possible, combining loops, implementing software division for usage outside the critical loop, implementing a special cosine function using a small lookup table, insertion of pre-processor macros, and re-usage of variables.
- Reduced the run-time memory usage of the application significantly, to only 150 KB for an ECG sample frequency of 198 Hz.
- Converted a 16-bit general purpose instruction-set processor into a 32-bit application specific instruction-set processor (ASIP).
- Designed a customized MAC instruction in the processor.
- Designed parallel custom instructions in the processor, i.e., parallel load and store instructions, and parallel load, equal/greater than, and select instructions.
- Implemented additional functional units, and divided the data memory of the processor in two smaller memories, in order to access two data memories within the same clock cycle, in parallel.
- Added instruction level parallelism with two load and one multiply-accumulate operation, reducing the main critical loop to one single clock cycle.
- Changed the factor registers of the multiplier unit, such that they could be loaded by the load operations directly. Similarly was done to the product register of the multiplier, in order to read from that register directly. Otherwise, the data would have been copied to other registers before the information could be used.
- Reduced the total execution cycle count by 81%.
- Generated VHDL code for the processor using the development tools from Target Compiler Technologies.

- Synthesized the design, using 90 nm TSMC low power memories, from Virage Logic Corporation.
- Extracted timing information and power numbers after a place and route step on a chip, using 90 nm TSMC.
- Reduced the total average power consumption by 55%.
- Suggested future work objectives for further reduction in total average power consumption.

Chapter 2

The cardiac beat detection algorithm

In order to understand how the application works, and in order to optimize it further for ultra low power (ULP), it is necessary to have an understanding of the theory behind the application itself and how the application works. Since the biomedical application is a heart beat detector, a general introduction of the heart, how it works, and ECG is given in Section 2.1. The theory related to the continuous wavelet transform is presented in Section 2.2, before presenting previous work in Section 2.3 and 2.4. Section 2.5 presents the test databases used in this thesis.

2.1 The heart and ECG

The heart of a human being is a muscle that could be viewed as a pumping machine. It has four different blood-filled areas called chambers. The two upper chambers, the atria (left and right atrium), returns the blood from the body and lungs, and the two lower chambers, the ventricles (left and right ventricle), squirts the blood back to the body and lungs. The circulation is referred to as the movement of blood through the heart and around the body. By this circulation, the blood transports important supplies for the body like oxygen and carbon dioxide. This movement is caused by a pumping action, which is triggered by electric impulses generated from within the heart. All the information regarded to the basic theory of the heart and how a heartbeat is created was taken from [HRS] and [Abd], and the reader is referred there for further detail about the heart.

In order to create a pulse, the heart has a mechanism that generates electrical signals. These electrical signals goes through the heart, causing the muscle cells to contract. This mechanism tells the heart when to beat, and it works similarly as a spark plug of an automobile, where each "spark" travels across a specialized electrical pathway, and stimulates the muscle wall of the four chambers of the heart to contract in a certain sequence or pattern. A picture of the electrical system of the heart can be viewed in Figure 2.1. There are mainly one electrical node that generates those electrical impulses. This node sits in the upper portion of the right atrium, and is a collection of specialized electrical cells known as the sinus or sino-atrial (SA) node. The upper chambers, the atria, are first stimulated. Between the ventricles and the atria, a node known as the atrio-ventricular (AV) node, holds the electrical impulse for a brief period. This delay allows the right and left atrium to continue emptying its blood contents into the two ventricles. Therefore, the AV node acts like a "relay station", re-generating and delaying stimulation of the ventricles long enough to allow the

two atria to finish emptying. A structure called the bundle of His emerges from the AV node and divides into thin, wire-like structures called bundle branches that extend into the right and left ventricles. The electrical signal travels down the bundle branches to thin filaments known as Purkinje fibers. These fibers distribute the electrical impulse to the muscles of the ventricles, causing them to contract and pump blood into the arteries. This whole process is referred to as a heart beat, and those beats are generated several times per minute.

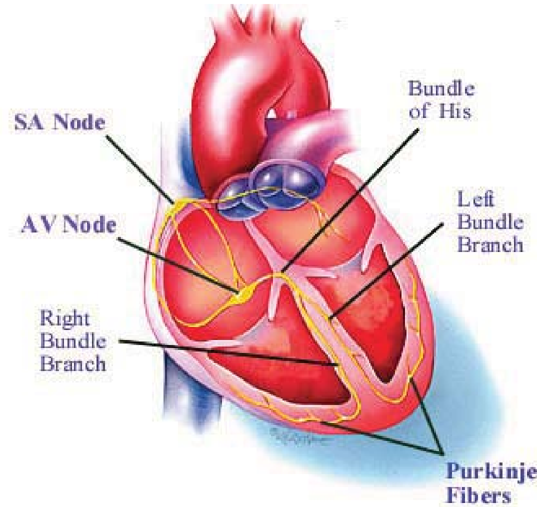

Figure 2.1: The heart, showing the electrical system of the heart [HRS].

Figure 2.2 shows how the electrical signal is generated, as well as the electrical wave which can be detected from the surface of the body by an ECG monitor. When the SA node fires a signal and causes the upper chambers to contract, the wave recorded on an ECG is known as the P-wave. The delay of the electrical impulse generated when the electrical impulse moves to the AV node, is recorded as the PQ-segment. When the electrical impulse travels through the ventricles and stimulates them to contract and pump blood into the body and lungs, it generates a wave known as the QRS complex. Then, the ventricles recover from this electrical stimulation and generates the T-wave, which is recorded by the ECG.

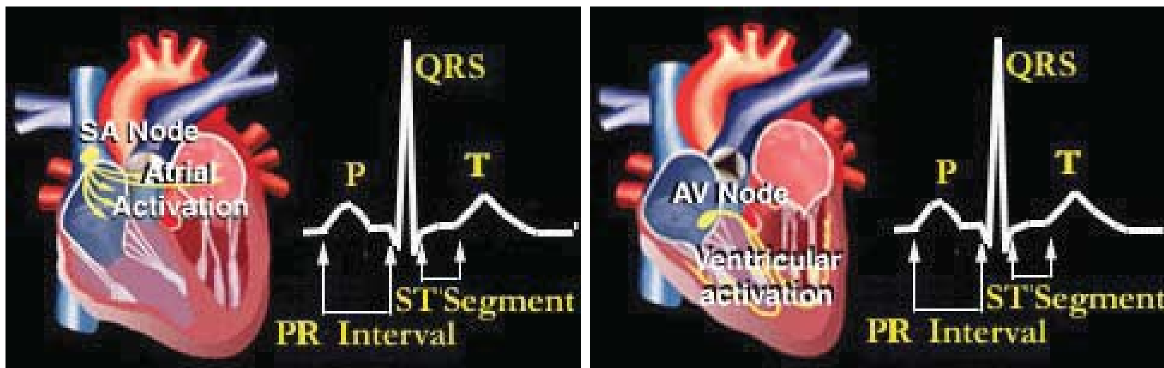

Figure 2.2: A demonstration of a heart beat [Abd].

Heart beat detection algorithms usually try to detect the QRS complex, because of its high amplitude. Figure 2.3 shows how the P,Q,R,S and T waves are displayed on an ECG monitor, using one standard electrode position. There exist different morphologies (different shapes

and amplitude) of the QRS complex, and Figure 2.4 demonstrates various QRS complexes.

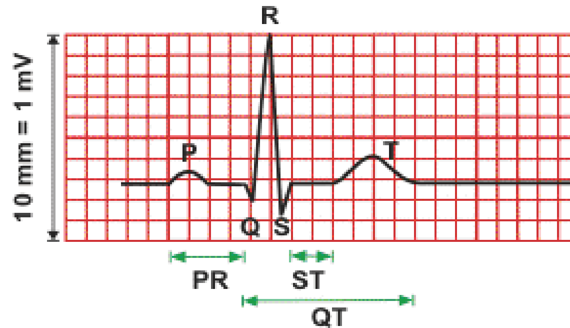

Figure 2.3: P,Q,R,S and T waves on an ECG monitor [RLG08a].

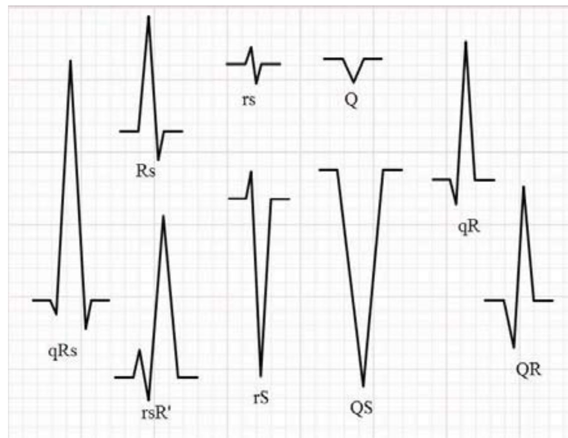

Figure 2.4: Various QRS complex morphologies [RLG08a].

The algorithm developed by Romero et al. [RLAR+05] has a high performance and is robust against sensible levels of noise or different morphologies of the QRS complex, according to [RLG08a] and [RLAR+05]. Since this algorithm is based on the continuous wavelet transform (CWT), the next section will describe the CWT in detail as well as the modulus maxima and the Mexican hat wavelet (which is used in the CWT as a reference wavelet), before describing the algorithm in Section 2.3.

2.2 Continuous and discrete wavelet transform, modulus maxima and the Mexican hat wavelet

In this section, the theories presented are taken from [Sch05] , [RLAR+05] and [RLG08a].

The wavelet transform of a continuous time signal, $x(t)$, is defined as

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} x(t)\psi^*(\frac{t-b}{a})\delta t, \tag{2.1}$$

where $\psi^*(t)$ is the complex conjugate of a wavelet function $\psi(t)$, a is the dilation parameter of the wavelet (also called the scale), and b is the location parameter of the wavelet.

Certain mathematical criteria has to be satisfied in order to be classified as a wavelet:

1 A wavelet must have a finite energy: $E = \int_{-\infty}^{+\infty} |\psi(t)|^2 \, \delta t < \infty$

2 If $\hat{\phi}(f)$ is the Fourier transform of $\psi(t)$, i.e., $\hat{\phi}(f) = \int_{-\infty}^{+\infty} \psi(t) e^{-i\omega t} \delta t$, then the following condition must hold: the admissibility condition.

$$C_g = \int_0^{+\infty} \frac{\left|\hat{\phi}(\omega)\right|^2}{\omega} \delta\omega < \infty, \tag{2.2}$$

where the wavelet has no zero frequency component. C_g is called the admissibility constant, and the value depends on the chosen wavelet.

3 For complex (or analytic) wavelets, the Fourier transform must both be real and vanish for negative frequencies.

The contribution to the signal energy at the specific scale a and location b is given by the two-dimensional wavelet energy density function known as the scalogram:

$$E(a, b) = |T(a, b)|^2 \tag{2.3}$$

The contribution to the total energy distribution contained within the signal at a specific scale a is given by the integral of $E(a, b)$ with respect to b. This is known as the wavelet variance, and can be used to find dominant scales associated with the signals. The total energy in the signal may be found from its wavelet transform as follows:

$$E = \frac{1}{C_g} \int_{-\infty}^{+\infty} \int_0^{\infty} \frac{1}{a^2} |T(a, b)|^2 \, \delta a \delta b \tag{2.4}$$

The Fourier transform is based on a continuous sinus signal as the reference signal. On the other hand, the wavelet transform implies, from the conditions above, that the reference signal (the mother wavelet) has a finite size. In addition, it is possible to use different reference signals as mother wavelets, i.e., the Mexican hat wavelet. The frequency parameter of the Fourier transform determines how the sinus signal is stretched over different frequencies, and in the wavelet transform, the stretching of the mother wavelet is done by changing the dilation parameter (the scale).

Wavelet modulus maxima is defined as:

$$\frac{\delta |T(a, b)|^2}{\delta b} = 0, \tag{2.5}$$

and it is used for locating and characterizing singularities in the signal. The CWT allows arbitrary high resolution in the time frequency plane, allowing each modulus maxima line to be followed with certainty across different scales.

The Mexican hat wavelet is used as the mother wavelet ($\psi(t)$ related to Equation 2.1) in the algorithm developed by Romero et al., and this wavelet is defined as the normalized (where $b = 0$ and $a, c = 0$ in Equation 2.7) negative of the second derivative of the Gaussian function. The equation is shown in Equation 2.6, and the shape is illustrated in Figure 2.5.

$$\psi(t) = (1 - t^2) e^{\frac{-t^2}{2}}, \tag{2.6}$$

where the Gaussian function is the following:

$$f(t) = a \, e^{\frac{-(t-b)^2}{2C^2}} \tag{2.7}$$

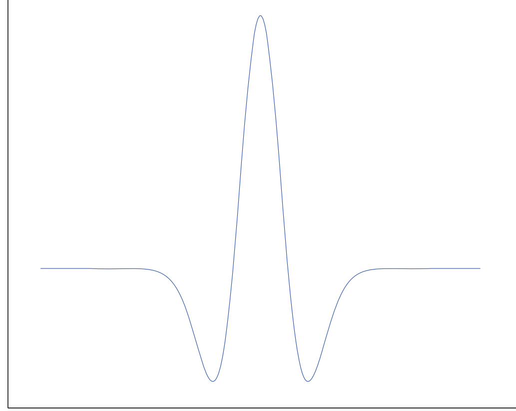

Figure 2.5: The Mexican hat wavelet.

Another reason for using the Mexican hat wavelet is that the Mexican hat maxima lines are guarantied continuous across scales for singularities in the signal. The conversion from wavelet scale to wavelet frequency is given by:

$$f = \frac{f_c}{a \cdot \triangle t}, \tag{2.8}$$

where a is the wavelet scale, f_c is a characteristic frequency of the wavelet, $\triangle t$ is the sampling period ($\triangle t = \frac{1}{fs}$, where fs is the sample frequency).

For the Mexican hat wavelet, the characteristic frequency employed is the standard deviation of the power spectrum, given by $f_c = \frac{\sqrt{\frac{5}{2}}}{2\pi} \approx 0{,}251$.

2.3 The beat detection algorithm

The CWT, as described in 2.2, is one of the building blocks in the algorithm developed by Romero et al. According to [RLAR$^+$05], the QRS complex is part of the ECG signal that represents the greatest deflection from the baseline of the signal. Within the QRS complex, the R-wave ideally represents the positive peak. Therefore, many QRS detection algorithms try to detect the R-peak within the QRS complex. And the use of continuous maxima lines, from the modulus maxima described in Section 2.2, allows the R-wave to be followed much more accurately across frequencies in the time-frequency plane. The algorithm which will shortly be described, does not require pre-filtering, and is robust against interference signals such as electromyography (EMG) noise or movement artifacts. EMG is a technique for evaluating and recording the activation signal of muscles, and is performed using an instrument called an electromyograph, [Wik09b] and [RHY06]. The technique used in this algorithm could also be used to detect P and T waves within the ECG signal.

The schematic diagram of the algorithm is shown in Figure 2.6, and is described in [RLAR$^+$05] as follows:

1 The ECG signal is analyzed within a two second window, as longer segments did not give satisfactory results. First, the CWT is calculated over this two second interval. It was then proved to work with the best performance when it operates within the frequency interval of 15-18 Hz. This frequency interval relates to the wavelet scales domain in Equation 2.8. Right after the computation of the CWT, a mask is applied

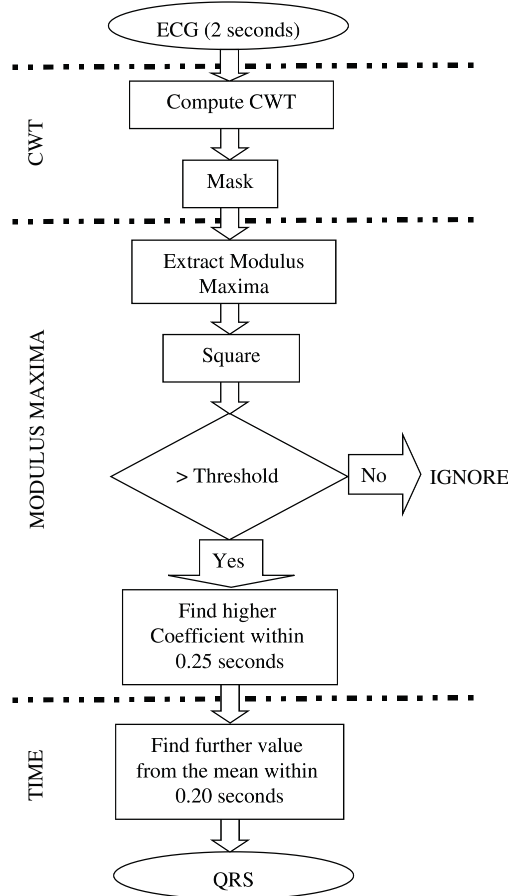

Figure 2.6: The algorithm developed by Romero et al., [RLAR+05].

to remove edge components of the result, which is set to be four times scale a of the wavelet. This mask has been found through trial and error.

2 In the second step, the modulus maxima of the CWT are extracted. The square of the the modulus is taken in order to emphasize the differences between coefficients. In addition, the maxima lines below a chosen threshold are ignored. The best threshold was found to be 30%.

3 The result from the previous step are taken to be possible R wave peaks. In order to separate the different peaks from each other, all modulus maxima points within an interval of 0.25 seconds of each other are analyzed in turn as search intervals. In every search interval, the point with the maximum coefficient value is selected as the R wave peak. The assumption in this step is that all the coefficients within each search interval of 0.25 seconds are due to the same QRS complex. Therefore, only the maximum point is chosen.

4 Finally, the algorithm finds the exact location of the peak in time-domain, based on the results from previous step. This is done by calculating the mean value of 0.10 seconds before and after the point detected in the wavelet domain. Then the point within the same interval of 0.2 seconds is located, where the signal is furthest from the mean. This step is necessary, because the modulus maxima line does not necessarily point to the exact location of the R wave peak in time domain.

This algorithm was tested for both positive and negative QRS complexes. In addition, the algorithm could cope with significant signal noise. Two comprehensive databases were used to test this algorithm. The MIT/BIH database and an in-house database recorded by Romero et al. within the Coronary Care Unit (CCU) at the Royal Infirmary of Edinburgh. How the sensitivity and the positive predictivity are calculated, as well as a description of the databases used in this thesis are given in Section 2.5. The results from the MIT/BIH database showed that the algorithm had a sensitivity 99.7% and a positive predictivity of 99.68%. The in-house database from the CCU resulted in 99.53% and 99.73%, respectively. However, these results were achieved from tests based on recordings done in hospitals. According to Romero et al. [RLG08a], the levels of noise and artefacts are therefore significantly lower than the levels recorded during ambulatory monitoring. For this reason, the algorithm is further optimized following the scheme as described above, but taking into account that the final application will be an ambulatory device. During the optimization of the algorithm, some changes were introduced, as described in [RLG08a]. These changes are presented in the next section.

2.4 Optimization of the algorithm

In [RLAR+05], the time window of input data to the algorithm is two seconds. During the implementation mentioned in [RLG08a], some preliminary tests resulted in some changes. It was determined that using windows of three seconds, and shifting the windows two seconds gave better results. In this case, the shifting would give a 0.5 second overlap with the previous window and again 0.5 second overlap with the next window. In order to avoid mismatches in the overlap sections, the detection in the first 0.5 seconds and the last 0.5 seconds within the window are ignored. Figure 2.7 illustrates this clearly. This was done in order to avoid boundary effects in the CWT computation. The optimization was mainly done in order to

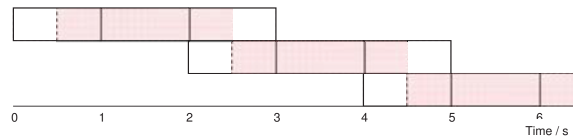

Figure 2.7: Illustration of the overlap solution of the input to the algorithm, [RLG08a].

improve robustness to noise which is encountered under ambulatory monitoring conditions. This resulted in four optimizations of the original algorithm.

The first optimization was done in order to achieve better refinement in time domain, when searching for a specific point within the QRS complex. In theory, the general idea was to search around the detected beat to the nearest maximum. However, the unfiltered signal in time domain contains the full range of noise and artifacts. Five possible solutions were proposed in [RLG08a] to improve this issue. The chosen optimal solution was to search for the absolute maximum, and chose peaks close to the detected beat. This is done by multiplying the signal within the search interval with a Hanning window, given by Equation 2.9, [Wik08] and [CL93].

$$\omega(n) = 0.5\,(1 - cos(\frac{2\pi n}{N-1})), \tag{2.9}$$

where N is the total length of the signal and n is an integer with values from 0 to $N-1$. With this solution, the algorithm is able to detect the right peak, even if the negative component of the QRS complex is bigger than the positive peak. If a person suffers from premature

ventricular contractions, the QRS complex might have opposite polarization. This is tackled well by the chosen solution.

The second optimization to the algorithm was done in the calculation of the threshold. Since the modulus maxima relates the peak detection to the maximum value within a search interval, it would be very sensitive to peaks in the CWT domain. From diseases or noise, peaks can occur easily. Therefore, a more robust threshold detection is desired. In [RLG08a], this is done by running average threshold, where the new threshold is partly determined as a percentage of the current maximum and partly on previous thresholds. This is identical to implementing a threshold feedback to the algorithm, in order to adjust the threshold. From [RLG08a], the formula for the threshold is given in Equation 2.10.

$$T = (1 - \mu)T_{old} + \mu T_{new}, \tag{2.10}$$

where T_{new} is defined as

$$T_{new} = C arg \max_{w,s} |\mathbf{w_s}| \tag{2.11}$$

Here, μ represents the "forget rate", a value which could, according to Romero et al. [RLG08a], be equal to the constant 1/8 or 1/4. The vector $\mathbf{w_s}$ represents the wavelet coefficients on a chosen scale s, and C is a constant fraction that relates the absolute maximum coefficient value to a threshold value. T_{old} is the previous threshold value. Different values for μ was tested in [RLG08a], and resulted in a trade-off between two considerations. For high values of μ, the algorithm is sensitive to impulsive noise, and too low values of μ will lead to smearing of artifacts over a long period of time. Based on these considerations, the value of μ was decided to be 0.5.

The third optimization was the removal of the squaring of the modulus maxima of the CWT. The squaring turned out not to give any actual improvements in the signals tested.

According to Romero et al. [RLG08a], if the threshold is exceeded in one scale in the initial algorithm, then this beat will be analyzed further, and there is only one threshold for all scales (see Equation 2.11). This solution will increase the probability of false detections if the number of scales is increased. The solution to this problem was to require the threshold to be exceeded in at least N out of K scales. In addition, the threshold is determined individually in each scale, giving a new equation for calculating T_{new}. This is given in Equation 2.12, [RLG08a].

$$T(s)_{new} = C arg \max_{w} |\mathbf{w_s}| \tag{2.12}$$

The implementation of this new threshold is described as follows (from [RLG08a]):

1 From all detected beats in all scales, clusters of beats are defined. This is done by clustering all beats within a 0.02 second time frame.

2 For each cluster, it is verified whether or not the beats have been detected in N out of K scales. If not, then the cluster is not analyzed any further.

From the conditions, the best results were given with $N = 2$ and $K = 4$, at a threshold of 30%.

The testing of this optimized algorithm resulted in a sensitivity of 99.68% and a positive predictivity of 99.75% on the MIT/BIH database. The results from the database created by IMEC-NL gave a sensitivity and a positive predictivity over 99.5% in most of the signals. The noise tests from the IMEC database gave satisfactory results for SNRs from 0 dB and above. Romero et al. [RLG08a] concluded from the noise tests that a key system requirement

for robust ECG beat detection is that monitoring hardware (mainly the amplification and signal conditioning stages of the system) shall provide a signal characterized by a SNR of 0 dB or above. As long as this condition is met, the optimized algorithm would achieve reliable beat detection.

The updated schematic diagram of the algorithm after the optimizations is shown in Figure 2.8.

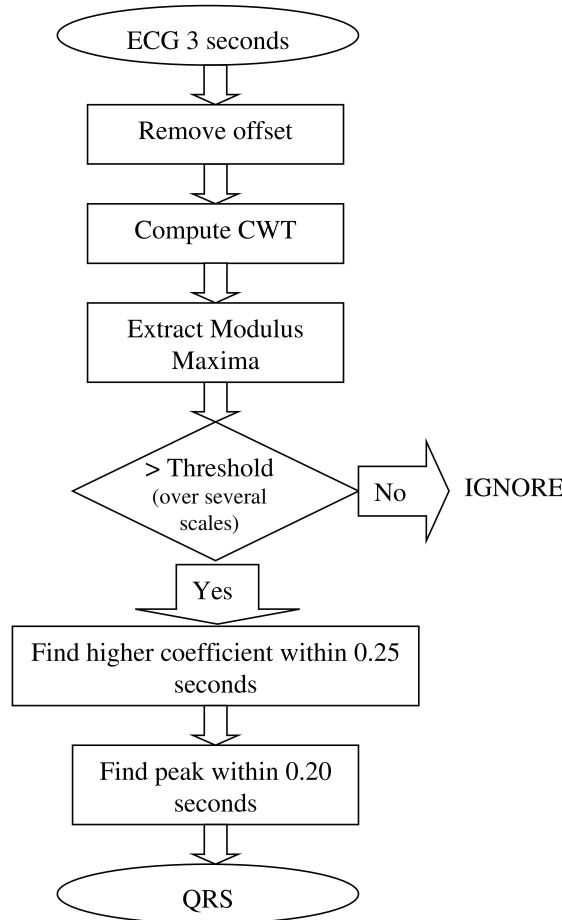

Figure 2.8: The algorithm after the optimization done in [RLG08a].

2.5 The test databases

In this section, the test databases used for the validation of the application in this project are described. The databases used are the MIT/BIH database and a recorded database done by IMEC-NL. Section 2.5.1 describes the MIT/BIH database, and in Section 2.5.2, the database recorded by IMEC-NL is presented.

2.5.1 The MIT/BIH database

One of the most common databases used for testing beat detection algorithms is the MIT/BIH database. According to Goldberger et al. [GAG+13] and Romero et al. [RLG08a], the MIT/BIH database contains recordings of ECG signals done in laboratories at Boston's Beth

Israel Hospital with the support from MIT since 1975. These recordings include several datasets for different test purposes, e.g., the Arrhythmia database which is among the test signals used in this thesis. As mentioned in [GAG+13] and [RLG08a], the MIT/BIH Arrhythmia Database contains several half-hour excepts of two-channel ambulatory ECG recordings, obtained from 47 subjects studied by the BIH Arrhythmia Laboratory between 1975 and 1979. 23 recordings were chosen at random from a set of 4000 24-hour ambulatory ECG recordings collected from patients at Boston's Beth Israel Hospital. 25 other signals were selected to include a variety of clinically relevant phenomena. All signals are numbered and the signals from the ambulatory recordings are labeled with numbers below 125, while the rest of the signals are labeled with the number 200 and higher. All signals used from this database are listed in Table 2.1. All signals have been digitalized at 360 Hz. For more information about the database, the reader is referred to [pyn], where the database is publicly available.

MIT/BIH database	
100	202
101	203
102	205
103	208
104	209
105	210
106	212
107	213
118	214
119	215
200	217
201	219

Table 2.1: Chosen signals from the MIT/BIH database.

2.5.2 The database recorded by IMEC-NL

IMEC-NL also recorded a set of ECG signals in order to test the algorithm under different levels of activity that produce noise and motion artifacts. The database consisted of recordings from ten volunteers, with three levels of activity; low, medium and high level of activity. The low level consisted of work, sitting at a desk, the medium level was performed by riding a static bike at 70 RPM and 100 Watts. The high level of activity was recorded by running on a treadmill at 7.5 Km/h. For these recordings, a belt and a textile patch was used. The number of recordings used in each case of activity are listed in Table 2.2. These signals were sampled with 198 samples per second.

	Belt	*Textile Patch*
Work	10	10
Bike	10	4
Run	8	3

Table 2.2: Number of recordings in the IMEC ECG database, [RLG08a].

In addition, signal 100 from the MIT/BIH database was selected as the base signal for the noise test. It has a clean ECG signal, and noise was added gradually to that signal for different SNRs, from 10 dB to -10 dB (in steps of one). This test is similar to the MIT-BIH Noise Stress Test Database, which could also be found in [pyn].

In order to test accuracy, an exact peak test was also created. This test contained three ECG signals, with two different sampling frequencies (198 Hz and 1000 Hz). The output from this test is the percentage of correctly detected beats at its peak, the average distance from the accurate beat, standard deviation from the average distance, and total number of beats which was wrongly detected.

The MIT/BIH and IMEC databases were embedded in a Matlab script using Matlab binaries. Usually, the parameters used to compare beat detection algorithms are sensitivity and positive predictivity of the result for each test signal. The sensitivity is defined as shown in Equation 2.13.

$$Se = \frac{TP}{TP + FN} \quad , \tag{2.13}$$

where TP stands for true positives (correctly detected beats), and FN stands for false negatives (missed beats).

The positive predictivity is shown in Equation 2.14.

$$+P = \frac{TP}{TP + FP} \quad , \tag{2.14}$$

where FP are false positives (wrongly detected beats). All database signals are annotated, compared to the result of the algorithm. A beat is in this case defined to be correctly detected if there is an annotated beat within a certain time window around the detected beat. This time window is set to be +/- 100 ms. These values are percentages in the range between 0% and 100%. An ideal algorithm would maximize these two parameters. Therefore the sensitivity and positive predictivity are summed together in a unique parameter to be maximized (Se + +P), where the range is from 0% to 200%. The results of the evaluation tests are stored in a text file produced by the matlab script.

Chapter 3

Energy-efficient software and hardware development

This chapter will provide some background theory related to energy-efficient software and hardware development in embedded systems.

In recent years, power consumption has been an important topic in embedded systems design. According to Russell et al. [RJ98], the power consumption places requirements on heat dissipation, which directly affect unit cost. Furthermore, in portable systems, the minimization of energy consumption, and the time integral of power, is critical to extending battery lifetime.

Since battery power density is increasing only at a rate of approximately 5% per year, any significant extension of battery lifetime must come from a thorough improvement of energy efficiency for each power-hungry component in a system, [CHC06].

Ultra low power or energy-efficient design refers to, in this thesis, the minimization/reduction of power and/or energy dissipation in the design, resulting in an optimal solution for a specific application.

An electronic system usually consist of a hardware platform, executing system, and a software application. In energy-efficient design, the reduction of power dissipation must be done in all parts of the design. Three major constituents consuming significant energy in hardware platforms are mentioned in [BDM00]. These are computational units, communication units, and storage units. The choices for software implementation also affect the energy consumption in those hardware components. Compilation of the software affects the instructions used by computing elements. Moreover, software storage and data accesses in memory affect energy balance, and data representation (i.e., special variables, encoding) affects power dissipation of communication resources (e.g., buses). Tackling power dissipation in every design stage is therefore important since both software and hardware platforms affect the power consumption.

In Chapter 4 and 6, several design techniques are used to optimize both the software application and the hardware platform, in order to achieve lower power consumption. In Section 3.1 and 3.2, the focus will be on presenting software and hardware design techniques, where some of them are used in this thesis. These techniques are described in [BDM00], and are only summarized here. In addition, some power management techniques will be presented in Section 3.3. Since floating point to fixed point conversion is used in Chapter 4, the theory behind fixed point conversion is given in Section 3.4. A general introduction to profiling and tools used to extract profiling information is given in Section 3.5. Section 3.6 presents an energy model used to calculate the average power consumption of a system. This model is based on the model presented in [RJ98] and [DNYB$^+$07].

3.1 Software code optimization techniques

Application software is typically written in high level programming languages (e.g., C, C++, Java) and then compiled into machine code for specific instruction-set micro-architectures.

Metrics for power consumption are the energy required by a program to perform a batch (one-time) job, in addition to average power consumption for interactive applications.

It is the execution and storage of software that requires energy consumption by the underlying hardware. Performing operations on hardware, as well as reading and writing data to memory, corresponds to the software execution. The software execution therefore involves power dissipation for computation, storage, and communication.

The energy budget for storing programs is typically small (with the choice of appropriate components) and predictable at design time. It is important to remember that reducing the size of programs, which is the usual goal in compilation, correlates with reducing their energy storage costs. Additional reduction of code size can be achieved by means of compression techniques, described in [BDM00].

It is often the case that programs that properly emulate the system functionality display poor performance and/or low energy efficiency. Therefore, such programs need to be re-written. The difficulty comes from the fact that procedural languages allow designers to represent functions in many different ways, and software programs are sometimes written (or ported) without the knowledge of the target processor or of its specifications.

The use of complex operations such as square roots, exponential functions, and divisions usually rely on a floating point unit in hardware, and floating point variables in software. Optimizing away such functions with the use of lookup tables, code simplification and alternative implementations result in a more energy-efficient software. In the rest of this section, some techniques for code optimization are briefly described. Several other optimization techniques are described in [BDM00].

Usage of lookup tables

If a complex function could be pre-calculated and stored in a lookup table, the computational energy would be reduced to only memory accesses. Even though memory accesses are expensive, by generating a lookup table, some complex operations could be removed and thus save the requirement for floating point hardware units on a processor.

Variable types

Usually, the most energy-efficient variable type in processors are their native datatype of the datapath. Some compilers by default use 32 bits in each assignment, so when either 16-bit or 8-bit variables are used, sign or zero extension is needed, costing at most two extra instructions as compared to their native datatype. Some architectures, like the x86 intel architecture, support multiple datatypes. However, most DSPs and ASIPs do not have that support for various reasons, i.e., power, area, etc.

Floating point to fixed point conversion

A widely known optimization technique is floating point to fixed point conversion. This optimization involves the process of eliminating all floating point variables by the cost of loss in accuracy. The theory behind fixed point conversion and fixed point numbers is described in Section 3.4.

Register allocation

If global control variables are used, it is beneficial to make them local instead so that they can be assigned to registers instead of placing them in memory. This saves load and store operations, and allows the compiler to generate more efficient loops in terms of cycle counts and memory accesses.

Conditional execution

Conditionalizing in a processor is usually done in two steps. First a few compare instructions set the compare codes. These instructions are then followed by the standard instructions with their flag fields set, so that their execution proceeds only if the preset condition is true. A more energy-efficient way to check if a variable is within some range is to use the ability of the compiler to conditionalize the arithmetic function.

Function design

By far the largest savings are possible with good function design, according to Benini et al. [BDM00]. In a processor, function arguments are usually placed on the stack, but when there are fewer arguments than the function overhead of the processor, they can be placed in registers instead. Upon return from a function, structures can be passed through registers to the caller. When the return from one function calls another, the compiler can convert that call to branch to another function.

Re-usage of variables

If an array or variable is only used during a part of the execution, the re-use of the memory for other purposes would reduce the memory footprint.

Loop elimination

If some loops could be combined, the execution time of the application could be optimized further by saving unnecessary loop iterations.

Memory optimization

Memory performance and power are now the key challenges in system design. Since a simple algorithm may require large storage, the efficiency of realization can not simply be related to computational energy. This is especially the case when the energy cost of storage and data transfer is significant. Lack of locality in a highly sequential control flow implies that results computed very early are needed much later in time. Data variables that have a long lifetime, would increase the storage requirements.

According to Lidsky et al. [LR94] and Catthoor et al. [CDGS98], memory power and access time usually dominate total power and performance for computations with large storage requirements in simple memories like the flat memory. The memory could in these cases become the main bottleneck for the application.

One obvious solution to the memory bottleneck problem is to reduce the storage requirements of the target applications. During algorithmic design, the principle of temporal locality [EM89] can be very helpful in reducing memory requirements. If the value of an array element, stored in memory, is accumulated inside a loop, then a temporary local variable could be applied in order to reduce memory accesses in the loop. The array element could then be updated by the temporary variable after the loop.

Other memory-reduction techniques strive to find efficient data representations that reduce the amount of inessential information stored in memory. Data compression is probably the best-known example of this approach. Unfortunately, storage reduction techniques cannot completely remove memory bottlenecks because they also try to optimize power and performance indirectly by reducing memory size.

3.2 Hardware optimization techniques

Three main limitations for processors, in an energy-efficient viewpoint, are described in [BDM00]. First, they have an intrinsic power overhead for instruction fetch and decoding, which is not a major issue for computation units with hardwired control. Second, they tend to perform computations as a sequence of instruction executions, and cannot take full advantage of algorithmic parallelism. Finally, they can perform only a limited number of elementary operations, as specified by their instruction-set architecture (ISA). Therefore, they must reduce any complex computation to a sequence of elementary operations. These limitations do not only increase power consumption, but they also decrease performance. The optimization of all complex computations in the software application is vital in this case.

When designing a processor for a specific application, it is possible to obtain a highly optimized specialized processor if energy (and performance) optimizations are made based on the knowledge of the target application. According to Goossens et al. [GVPL+97], this observation has lead to the development of a number of techniques for application-specific instruction-set processor (ASIP) synthesis.

The rest of this section will briefly summarize some hardware optimization techniques described in [BDM00].

Insertion of application-specific units

By inserting application-specific units, power efficiency could be achieved at the price of reduced flexibility. Typically, the design starts from an initial specification that is executable and given in a high-level language, with constraints on performance, power, and area. The outcome is a circuit that can perform only the specified function.

Exploiting profiling information

Profiling information could be exploited to tailor a parameterized super scalar processor architecture to a single application. Parameters such as number and type of functional units and register file size can be explored and optimized. The optimization loop is based on iterative simulation of a microarchitecture model under the expected workload. In the profiling reports, the critical parts of the code are identified, and hence the designer could find the corresponding critical part in the assembly code of the application. By studying the different instructions performed, it is possible to optimize the critical parts, e.g., by combining some instructions together in order to reduce the cycle count. Alternatively, one could create new custom instructions. It is worth noting that the software can be optimized by hardware profiling, since hardware profiling usually give a more useful information, i.e. execution cycle counts. It is also possible to do software profiling, but the information achieved is usually related to the number of function calls, cache hits and misses, etc. Some software profiling tools are presented in Section 3.5.

A linear relationship between power and frequency is shown in [RJ98] for the behavior of power consumption versus frequency for a few representative instructions of a processor.

Russel et al. [RJ98] conclude that, for the general class of embedded systems, the energy consumption would be minimized by minimizing the software execution time.

Instruction subsetting

Reducing instruction decoding and micro-architectural complexity by reducing the number of instructions supported by an ASIP, is done by instruction subsetting. The basic procedure for instruction subsetting, proposed by Dougherty et al. [DPT98], can be summarized as follows: The target application is first compiled for a complete ISA, then the executable code is profiled. Instructions that are never used or instructions that can be substituted by others with a small performance impact are dropped from the application-specific ISA. Then, the processor with a reduced ISA is synthesized.

Custom instructions

Specialized instructions are often proposed as a power and performance enhancement technique. The basic idea is to provide a few specialized instructions and the required architectural supports that allow the processor to execute very efficiently under some specific workload. Domain-specific instructions can reduce power and increase performance when the processor is executing data-dominated applications. Examples of such instructions are sub-word parallel instructions [IY98], special addressing modes [KI97], and multiply-accumulate instructions [LR02].

Pipelining and parallel instructions

By exploiting the pipeline stages of the processor, one could reduce the cycle count of the application by combining two or more instructions in the critical loop, to run within the same clock cycle. For instance, if two memory loads are done after each other, one could initially store the data of the load operations in two different memories. Extra functional units could then be applied in order to perform these load operations in parallel. If those load operations are done before, e.g., a multiply operation, then they could be combined by creating a custom instruction that does these operations in different steps of the pipeline. However, the clock cycle would be longer, making the processor slower. In most cases, this trade-off usually pays off on power efficiency, because making the processor slower reduces the power consumption, according to Russel et al. [RJ98].

Memory optimization

Some memory optimization techniques are discussed in [BDM00], where synergistic memory and computation optimization is concluded as the technique that gives the best memory optimization results. This technique consist of synergistically optimizing the structure of a computation and the memory architecture that supports it.

A methodology called data transfer and storage exploration (DTSE), developed by Catthoor et al. [CDGS98], [CFW+94], is based upon the fact that in data-dominated applications, the power consumption related to memory transfer and storage dominates the overall system power. This methodology consists of optimizing the memory in every step of the design flow. The main purpose of that methodology is to provide a specification that is very energy efficient, when synthesized into silicon.

3.3 Power management techniques

The total power in system on chip (SoC) design consist of dynamic power and static power [KFA+07]. Dynamic power is the power consumed when the device is active, while static power is the power consumed when the device is powered up and idle. In CMOS devices, the primary source of dynamic power consumption is switching power, which is the power required to charge and discharge the output capacitance on a gate. The static power is mainly due to leakage.

In the following subsections, a methodology called dynamic power management (DPM) is described and some circuit-level power management techniques are briefly presented. These management techniques contribute to lower the dynamic and leakage power of the system.

3.3.1 Dynamic power management (DPM)

Dynamic power management (DPM) refers to a selective, shut-off or slow-down of system components that are idle or underutilized [Ped02]. DPM has proven to be a particularly effective technique for reducing power dissipation in portable electronic systems. It is however a difficult process that may require many design iterations and careful debugging and validation.

For DPM, it is required that the system has at least two power states, ACTIVE and STANDBY. In the ACTIVE state, the system runs normally. When the system is in the STANDBY state, it does not perform any useful computation or service. Events or interrupts are used to wake the system up from STANDBY to ACTIVE state. Since the ACTIVE state typically consumes much more power than the STANDBY state, putting components of a system in different power states, could save power.

Pedram et al. [Ped02] presents several techniques for system-level power management, based on DPM.

3.3.2 Component-level power management

Component-level management techniques can be divided in two categories: dynamic power minimization techniques, and leakage power minimization techniques. This section will briefly present some of those techniques that could be applied in order to reduce the dynamic and leakage power of an embedded system.

Dynamic power minimization

Dynamic power is consumed every time the output of a transistor gate is changed, and by using Equation 3.1, its average value can be computed.

$$P = \frac{1}{2} C V^2 f \alpha \;, \tag{3.1}$$

where C is the capacitive load of the gate, V is the voltage supply, f is the clock frequency, and α is the switching activity. In order to reduce the dynamic power, any of the parameters in Equation 3.1 may be reduced.

In [Ped02], some techniques that reduces the dynamic power are described. Among them are clock gating, and dynamic voltage and frequency scaling (DVFS).

Clock gating involves disabling the clock of a system component whenever its output values are not used.

In DVFS the key idea is to vary the voltage supply and the clock frequency of the system to provide only the necessary circuit speed to process the workload while meeting the total computation time and/or throughput constraints, and thereby reduce the energy dissipation.

If the delay related to the change of the voltage supply is known, in addition to the voltage, and frequency, then it is possible to estimate the power ratio, $P_{without_DVFS}/P_{with_DVFS}$, that could be reduced in the power consumption by applying DVFS, assuming that the other parameters in Equation 3.1 are constant.

Leakage power minimization

Leakage power dissipation grows with every generation of CMOS process technology [KFA+07]. In current CMOS technologies, the cause of leakage power is due to the leakage current that appears in transistors, independent of switching activities.

To reduce overall leakage power of an embedded system, it is desirable to shut down components that are not being used. Power gating is a technique that uses two power modes; low power mode and active mode. When the system does not use a specific component, it switches the component from the active mode to a low power mode. The leakage power could then be neglected when the processor is in low power mode. The only leakage present would come from the switch transistor of the power gating. More information about power gating, and a description of other techniques that reduce leakage power, can be found in [Ped02].

3.4 Converting floating point to fixed point

For embedded systems, the use of floating point hardware is usually not preferred, because it increases power consumption and complexity. Using a fixed point version of the algorithm is more attractive, in order to satisfy the cost and power consumption constraints of embedded systems, [CC99] and [MCCS02]. There is always a trade-off between power efficiency and accuracy. Designers often have to consider if a little loss of accuracy is worth the reduction in power consumption.

According to [Obe07], a significant improvement in execution speed could be observed with the use of fixed point conversion. However, this speed improvement comes at the cost of reduced range and accuracy of the algorithms variables.

The idea behind fixed point conversion is to represent all floating point variables as numbers encoded as integer values, with minimum loss of accuracy.

A floating point number could be represented as shown in Equation 3.2.

$$f = g \cdot 2^n,\ -1 \leq g < 1, \tag{3.2}$$

where f is the floating point value represented by g, multiplied by 2 in the power of n. In this case, n represents the number of bits needed to represent the range of the integer part of the number. This could be found by counting how many bits one has to shift f in order to put the most significant bit (MSB) of f in the first fractional position. [Obe07] presents the Q-point notion for a fixed-point number. This notation is shown in Equation 3.3

$$Q[QI].[QF], \tag{3.3}$$

where QI represents the number of integer bits, and QF represents the number of fractional bits, which is used to represent the floating point number. For instance, Q3.5 would represent a 8-bit number with 3 bits to represent the integer part and 5 bits to represent the fractional part. If this number is signed, then only 2 of the QI bits are used to represent the number.

The integer range of a floating point variable in an algorithm determines the number of bits required to represent the integer portion of the number. If the value is unsigned, according to Equation 3.2, n equals to 3 because the Q3.5 value must only be shifted three times to the right in order to place the MSB in the first fractional point. However, if the value is signed, then n would be equal to 2.

For instance, if a 16-bit signed integer number is used to represent a floating point number g, where $-1 \leq g \leq 1$, then the fixed point representation is

$$i = g \cdot 2^{16-1}, \tag{3.4}$$

where g is taken from Equation 3.2, and i is the fixed point representation, shifted 15 times to the left. It is only shifted 15 times because the MSB bit of the 16-bit integer value is reserved as a sign bit. From Equation 3.2 and 3.4, the fixed point 16-bit integer could be calculated using Equation 3.5.

$$i = f \cdot 2^{-n} \cdot 2^{15} = f \cdot 2^{15-n}, \tag{3.5}$$

where n is the number of bits needed to represent the integer part of the number. QI is in this case $n+1$ because n bits are used for the integer range, and a sign bit is added using 2's complement notation. This leaves $15 - n$ bits for the fractional (QF) range.

The range of the integer and fractional parts are discussed in [Obe07], and the full range of the fixed-point number was found to be as shown in Equation 3.6.

$$-2^{QI-1} \leq \alpha \leq (2^{QI-1} - 2^{-QF}), \tag{3.6}$$

where α is the range of a signed fixed-point number (the signed bit is included in QI). The resolution of this number is given by Equation 3.7.

$$\epsilon = \frac{1}{2^{QF}}, \tag{3.7}$$

where ϵ is the resolution of the fixed-point variable. From this equation, the required number of bits for a given resolution could be found by calculating QF from Equation 3.8.

$$QF = ceiling(log_2(\frac{1}{\epsilon})) \tag{3.8}$$

When working with mathematical operations using fixed point numbers, one has to be careful not to forget the number of bits the numbers are shifted at all times. When an addition operation is performed between two numbers, the number of bits needed to represent the answer and handle overflows is determined by the QI of the largest number and a possible carry bit. If a multiplication is performed, then the number of bits in the answer is the sum of the number of bits of the operands. This means if two 8-bit numbers are multiplied, then the result requires 16-bit. The 16-bit result could then be scaled back to 8-bit.

When an addition is performed in a loop, then $ceil(log_2(N))$ number of extra bits are needed in the resulting answer in order to handle overflows from possible carry bits. This could be easily seen by splitting up a loop of additions and assume one carry is achieved from each sub-addition.

One should also note that the property shown in Equation 3.9 holds for multiplications and not for additions.

$$s \cdot (c << b) \Longleftrightarrow (s \cdot c) << b, \tag{3.9}$$

3.5 Profiling

Profiling is a method of gathering performance data at the execution phase of an application, [Bra]. For performance measurement, there exist different metrics, i.e., CPU time per routine, number of times a function executes, cache hits and misses, etc. The most known profiler for C and C++ development is the open source profiler *gprof*, using the GNU compiler. The output of this profiler is a file containing two tables, a call graph and the flat profile. The call graph shows for each function, how many times it was called, how many times it called other functions, and which functions it was called from. The flat profile on the other hand contains information on how many cycles the processor spent in each function of the program, and how many times each function was called.

There are two ways of analyzing code through profiling, static and dynamic. In static profiling, the code is analyzed without executing the program. Dynamic profiling is done while executing the program, and by doing so, one could derive runtime performance information of the program. Since the results are depending on the input signals, one could risk the fact that the input signal used during profiling did not give the worst case information.

Valgrind is another well known tool where one could get memory usage information, cache usage and other useful information. It uses a graphical representation of data, which is useful for identifying the most used functions. This tool could be used together with a graphical tool called *kcachegrind* in order to graphically present the profiling results. *Valgrind* is known for being the most complete profiling tool under UNIX, with several sub-tools such as *Memcheck*, and *Callgrind*. *Memcheck* provides information about the memory usage like uninitialized memory, reading or writing to memory, memory leaks and mismatched use of memory allocations. *Callgrind* performs detailed simulation of the I1, D1 and L2 caches in the CPU, and can accurately pinpoint the sources of cache misses in the code. In addition, it produces call graph information which could be viewed graphically using *kcachegrind*.

3.6 The energy model

In this section, the energy model used to estimate the power consumption of the processor designed in Chapter 6 is presented.

According to Russel et al. [RJ98], the energy, E, consumed by a processor running a program is

$$E = \int_{t_0}^{t_0+T} P(t)dt, \tag{3.10}$$

where T is the software execution time, and $P(t)$ is the instantaneous power. Average power, P_{ave}, is defined as

$$P_{ave} = \frac{1}{T}\int_{t_0}^{t_0+T} P(t)dt \tag{3.11}$$

By combining Equation 3.10 and 3.11, the energy can be written as

$$E = T \cdot P_{ave} \tag{3.12}$$

The instantaneous power mainly consist of dynamic power, and leakage power, as described in Section 3.3. The dynamic power can be split in two parts, active power, and idle power. The active power is consumed when the processor is in an active state, processing the set of input samples received. After the processor is finished with the processing of data, it goes to

an idle state, where it waits for new input samples to arrive (when it is clocked). Therefore, the power, $P(t)$, mainly consist of those three power sources, as shown in Equation 3.13.

$$P(t) = P_{active}(t) + P_{idle}(t) + P_{leak}(t) \ , \tag{3.13}$$

where P_{active} is the active power, P_{idle} is the idle power, and P_{leak} is the leakage power.

As Figure 3.1 illustrates, the processor is usually idle until it receives the input samples. While the processor is processing the information, it uses more power, as indicated by the active power. As soon as the processor is finished with processing the data, it waits for the next set of input samples by going to the idle state. The average time in which the processor consumes the active power is indicated by T_{active} in the figure. The idle time, T_{idle} is the average time that the processor waits until the next set of samples arrive. The sample time, T_{sample}, is defined to be the average time period in which a set of input samples are received, until the next set of input samples arrive.

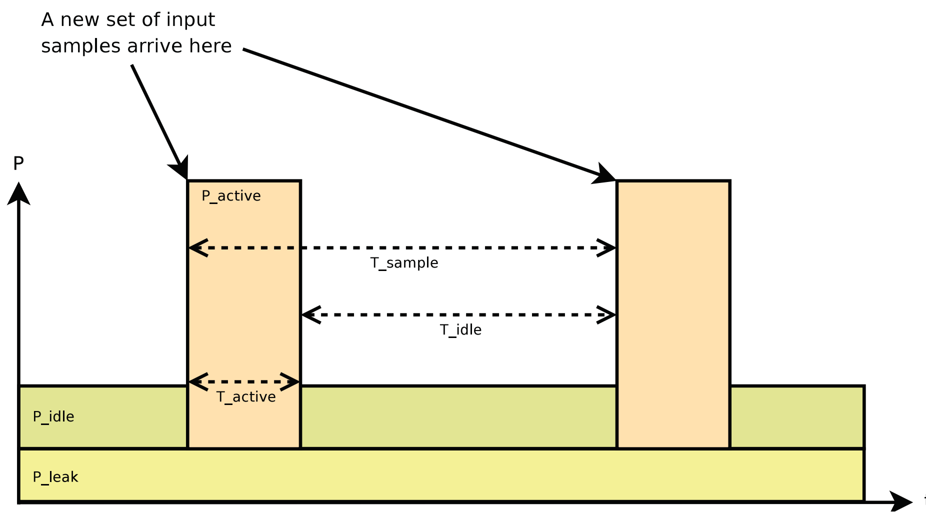

Figure 3.1: Dominant power sources for a processor.

The total average power, according to De Nil et al. [DNYB+07], is calculated by Equation 3.14.

$$P_{ave} = P_{ave_leak} + \frac{1}{T_{sample}}(P_{active} \cdot T_{active} + P_{idle} \cdot T_{idle}) \ , \tag{3.14}$$

where P_{ave_leak} is the average leakage power calculated by Equation 3.11, P_{active} is the total active power, and P_{idle} is the total idle power.

In order to estimate the software energy, there exist a more convenient way of calculating the average power. The total time, T, is related to N, the total number of execution cycles, and τ, the clock period, by $T = N\tau$. The average power, P_{ave} , is the sum of the average power during each clock period, $P_{\tau,ave}(i)$, where $i = 1...N$. This is given by Equation 3.15.

$$P_{ave} = \frac{1}{N}\sum_{i=1}^{N} P_{\tau,ave}(i) \tag{3.15}$$

The value $P_{\tau,ave}(i)$ is the average power consumed during one cycle of the instruction. As described in [RJ98], each $P_{\tau,ave}(i)$ is dependent on the present and past states of the processor, and thus measuring an instruction in isolation is a biased estimate of the power consumed by an instruction in a typical application.

Execution time depends significantly on runtime conditions such as cache misses, data dependent pipeline stalls, and operand dependent instruction execution times. Several accurate ways to measure execution time are also mentioned in [RJ98]: cycle accurate simulation, instrumentation of the actual system, or software based timers inside the processor. The instruction-set simulator (ISS) used in Chapter 6, simulates the ASIP through cycle accurate simulation.

If clock gating, as described in Section 3.3, is applied, then it is possible to neglect the idle power from Equation 3.14. The resulting equation is given in Equation 3.16.

$$P_{ave} = P_{ave_leak} + \frac{T_{active}}{T_{sample}} \cdot P_{active} \tag{3.16}$$

If power gating is applied in addition, the average leakage power would be reduced significantly.

In this thesis, the duty cycle, α, which refers to the time the processor is active, is given by the relation of T_{active} and T_{sample}, as shown in Equation 3.17.

$$\alpha = \frac{T_{active}}{T_{sample}} \tag{3.17}$$

For energy efficiency in ASIPs, the software and hardware are usually optimized in order to make the active power time, T_{active}, as short as possible. When clock gating and power gating are applied in addition, the processor would be switched off for a longer time. The average active power per cycle might increase by adding application specific hardware, but compared to the saved execution time, the total average power is reduced significantly.

Chapter 4

Matlab to C conversion

In this chapter, the conversion process of the algorithm developed by Romero et al., [RLAR+05], is described, analyzed and discussed. This process is divided into four activities. In each step, the code is validated using a Matlab testbench. This testbench integrates the test databases presented in Chapter 2.5. Some implementation challenges are discussed, and the results of the C application are presented. In addition, the information gathered from profiling are shown. How these results were found is also discussed.

The flow of work, describing the activities in the conversion process, is presented in Section 4.1. The following sections describe each step in the conversion process, before presenting the results. Finally, the results are discussed and some conclusions are made in Section 4.8.

4.1 The flow of work

As discussed in Chapter 1.2, a fixed point C version of the application is desired. Conversion from Matlab code to C code is not a simple task, and one could not simply convert a Matlab code optimized with built in Matlab functions without changing the algorithm. One possibility is to create a prototype version of the Matlab code, and for every built-in function, one could find the optimal algorithm that would do the same thing using only for-loops, while-loops and if-statements. This could be just as time consuming as starting from scratch in C. However, the advantage of this prototype is that it would be easier to simulate the test signals within the same environment to verify the correctness of the prototype. Because of the Matlab prototype, the conversion should be straight forward. In theory, the only thing one has to consider is the conversion of Matlab's one-indexing to zero-indexing in C. Instead of starting by scratch in C, and try to design the algorithm by the description in Chapter 2, a Matlab prototype is created in this project. In this prototype, the functionality of the Matlab code is re-written without the use of built-in Matlab functions. By making a prototype, it was possible to split the code in several parts, and convert one part at a time. Every part could then be validated by interfacing each part directly into the original Matlab code. Before that was done, the Matlab code had to be analyzed.

The conversion process was divided into four steps, where the application was validated in each step. An overview of the conversion process is shown in Figure 4.1.

The first step consist of analyzing the original Matlab code, in addition to the initial simulation of the model.

In the second step, a prototype is made, based on the considerations taken during the analysis. All built-in Matlab functions are interpreted by studying their Matlab documenta-

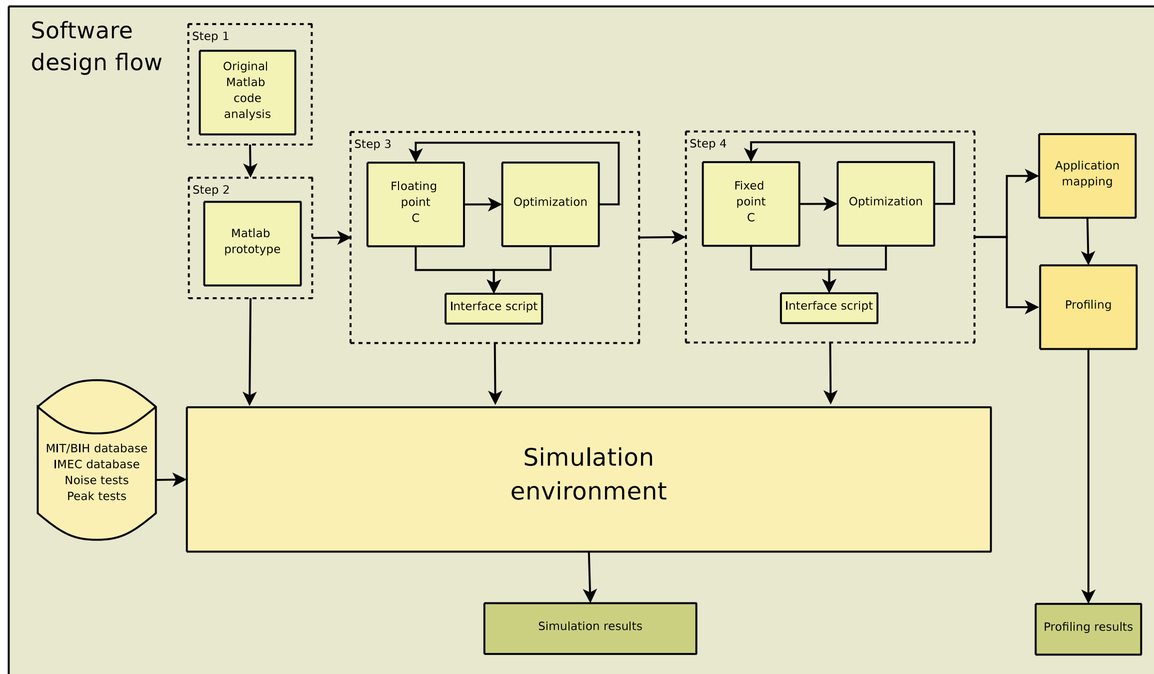

Figure 4.1: An overview of the steps in the conversion process.

tion, and they are implemented using only for-loops, while-loops and if-statements. In the case where a direct conversion is very inefficient, a more optimized solution is chosen.

In the third step, the Matlab prototype is translated into a floating point C application.

After creating the first running version of the C code, the optimization process can start. All memory usage must be optimized, and unnecessary code must be removed. In addition all complex functions must be optimized away.

This optimization step is followed by a fixed point conversion, where all floating point values are converted to integer values. If possible, some further optimizations are done.

After the conversion steps, the resulting application is run through profiling in order to find possible bottlenecks and memory leaks. Further, the application is simulated using an instruction-set simulator from Target, using a 32 bit processor, in order to determine the memory usage and estimate the cycle counts for the execution of the application.

The following sections describe how the conversion was done.

4.2 Matlab code analysis

Matlab is a development tool provided from MathWorks, optimized for using vector and matrix operations. It is possible to program algorithms fast and test functionality easier than standard programming languages like ANSI C. However, there is no need to take memory usage into account, and the programmer can focus on making the program model compact and fast using the built-in vector and matrix functions. Therefore, the provided model of the algorithm developed by Romero et al. [RLAR+05] was designed using the built-in Matlab functions when possible in order to keep the program as compact as possible.

The original Matlab model of the algorithm was done precisely as described in Section 2.3, with the optimizations described in 2.4.

This Matlab model could be viewed like a hierarchy, consisting of three functions (3 separate files). The functions are called QRSDet, QRSDet3, and cwt2. An illustration on how those functions are connected together is shown in Figure 4.2.

In Appendix A, the CWT function and the QRSDet3 function of the original Matlab model are described, analyzed and discussed, step by step. This is done in a bottom-up fashion, starting with the CWT function. In some of the steps, alternative implementations are suggested.

In the rest of this section, a summary of the changes suggested, based on the analysis, is given.

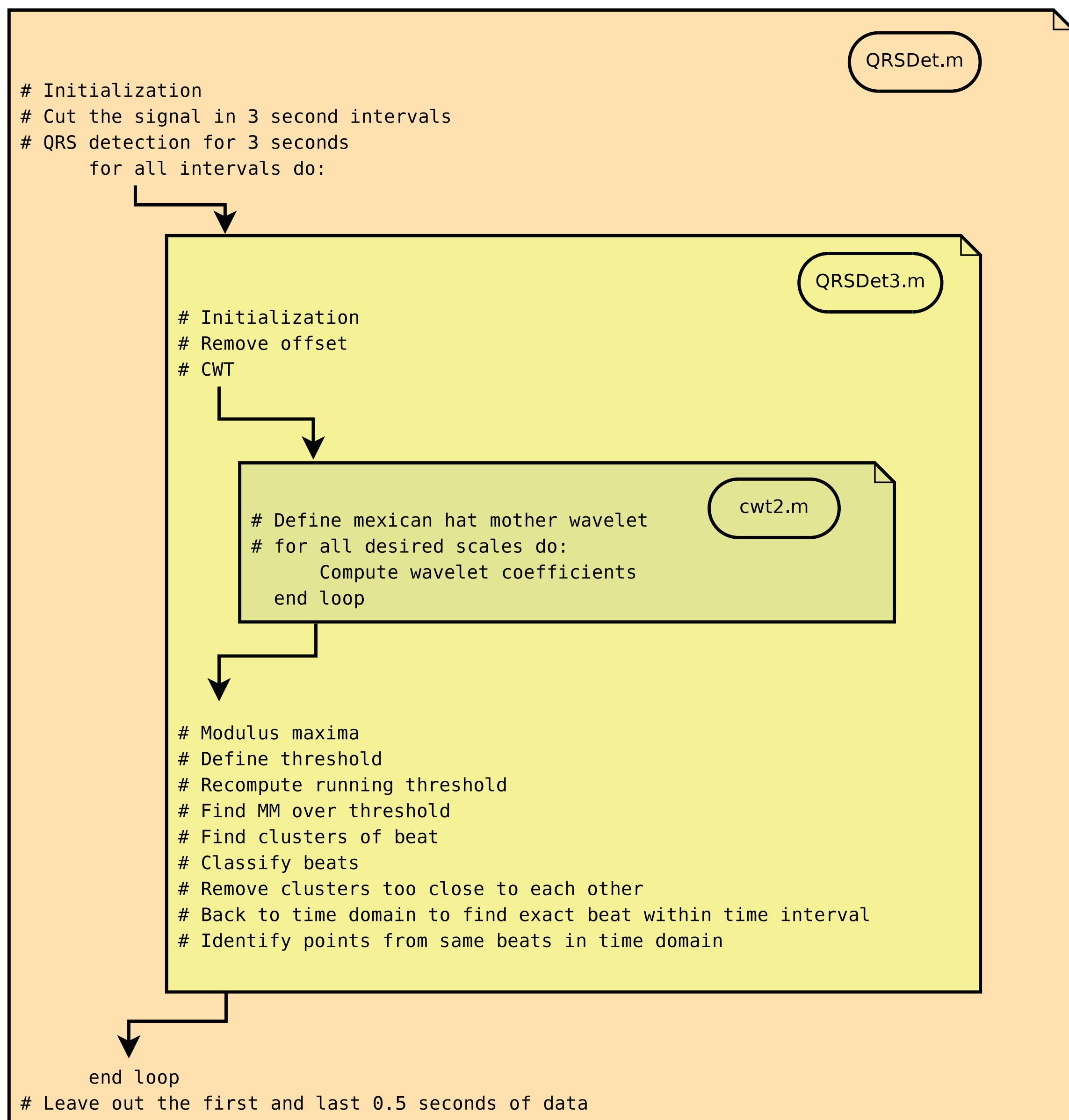

Figure 4.2: The Matlab model structure.

4.2.1 The CWT function

This function basically calculates the reference mother wavelet for all desired scales, and performs a convolution between the input ECG samples and a scaled mother wavelet. Then some calculations on the result of each convolution is done to obtain the desired coefficients, before the coefficients for all scales are returned in a huge matrix.

A detailed analysis of this function is done in Appendix A, and the analysis resulted in suggesting some lookup tables for the reference wavelet used in the wavelet transform, instead of calculating the values at runtime. This would reduce complex computations such as exponential functions and square roots. Further, the convolution is assumed to be in the main critical loop of the application. Two possibilities for the implementation of the convolution was discussed. One could either implement the convolution by its definition, or by converting the built in Matlab convolution function. An analysis of the running time is performed on both solutions in Appendix C. The conclusion is to convert the Matlab convolution function directly, because its running time is significantly faster than the convolution by the definition.

It was also observed that the maximum ECG sample frequency used in the test databases is 1000 Hz, and therefore the matrix containing all coefficients for all the different scales has to be big enough to contain the coefficients calculated using the largest scale.

4.2.2 The QRSDet3 function

This function is the largest function, containing all steps in Figure 2.8. It is called for one 3-second ECG input signal at a time.

In Appendix A, every step in this function is analyzed in detail, and therefore this section will only summarize the suggestions made for the conversion process.

In the step where the modulus maxima is calculated, some matrices were duplicated. Re-usage of matrices is suggested in this step, resulting in less memory requirements.

The step where the clusters of beats are identified, and the step related to the classification of beats, used matrix indexing in Matlab with only one parameter. Matrix indexing in ANSI C requires two parameters, unless an array is made out of the matrix. Several calculations were performed on index positions where the one-value Matlab indexing was used. The decision based on the analysis, is to use matrix indexing in C normally, but use the Matlab way of indexing in the calculations performed. This is the easiest solution.

The other steps in the QRSDet3 function required some minor changes, and the reader is referred to Appendix A for the analysis on those steps.

4.2.3 The QRSDet function

Since the QRSDet3 function, as described in Section 4.2.2, only analyzes 3 seconds at a time, the QRSDet is the function that takes an array of ECG signals of one length, divides it into several 3 second intervals and computes them with the QRSDet3 function for every interval. The answers from the QRSDet3 function are then shifted into the right time slot in the QRSDet function, and then stored in a big array. This resulting array contains the times in seconds where every QRS-beats are detected for the complete test signal, which is returned at the end. It is also in this function each 3-second interval is overlapped with the next 3-second sample, as described in Section 2.4.

The application must be able to detect QRS complexes and classify beats in real-time. Therefore, only the QRSDet3 and the cwt2 functions are chosen to be converted.

4.3 The Matlab prototype

The Matlab prototype was re-written step by step. Mainly all steps are modeled with slight changes, e.g., reading elements one at a time instead of a vector at a time. Other changes include dividing some calculations into several sub steps in order to simplify the code. For every built-in function, the Matlab documentation was studied, and the functionality was emulated using only for-loops, while-loops and if-statements. However, there were some steps that required alternative implementations of the code in the original Matlab model. These steps are implemented as suggested in Section 4.2 and Appendix A. The changes were mainly done in the cwt2 function, in the step where clusters of beat are created, and in the step where the clusters of beats are classified.

The resulting prototype contained several nested loops, making the application slower. This could be explained by looking into the simulation environment in Matlab. In general, Matlab is an interpreted language, and looping through a vector using for-loops is very slow. Since this code is meant to be implemented on an embedded platform, the built-in Matlab functions can not be used. It is worth noting that the alternative implementation in the Matlab prototype could be optimized further. Since the aim of this project is ASIP design, no further optimizations are done in this prototype.

This new Matlab model was simulated by interfacing the new QRSDet3 function to the QRSDet function. The prototype generated the exact same results as the Matlab code, but the simulation time was significantly longer.

All steps that required alternative implementations of the original Matlab code, i.e., the suggestions made in the previous section, are described in detail in Appendix B.

Before introducing the C application, the simulation interface between the test databases in Matlab and the C application is described.

4.4 Simulation interface and validation

In order to simulate the compiled executable of the C program, an interface must be made with the databases and test scripts in Matlab. This would save time since no test benches in C has to be created. There are two possible ways of creating this interface. For instance, one could implement the C code directly in Matlab and use its built-in GCC compiler, or one could use the method of reading and writing to a text file. One advantage of doing the latter alternative is that the C executable would be independent of any Matlab interference. However, the simulation time would be time consuming, since for every 3-second interval, the code reads from and writes to file many times. Since this program is going to be implemented on a processor at the end, the file I/O operations in C did not matter, only for simulation purposes. Therefore, the easiest solution is chosen, which is to interface the C code to the Matlab code by reading the inputs from file and write the results to a new file. Three files are needed for this interface. One file is used where the names of the files to read from and write to are written, and another file is used for some parameters like the size of the ECG signal, thresholds, frequencies, etc. The last file is meant for the ECG signal itself. First, the interfacing Matlab script writes all parameters to file before issuing a command to start execution of the C program. In the C program, the C interface reads those parameters from file, and generates the input arrays and variables. This C interface is the main function of the C code. When the C code has finished execution, it writes the outputs to file. The Matlab script interface then reads the answers from the output file of the C program, before continuing the execution of the test.

4.5 The floating point application

In the design of the C application, the cwt2 function was first converted, and validated, before the conversion of the QRSDet3 function. Therefore, these functions were coded in two separate files, cwt2.c and QRSDet3.c. Some input/output (I/O) libraries, and read and write functions in C are also included in order to interface the C application to the Matlab test environment, as described in Section 4.4. In the embedded platform architecture this application is mapped into in Chapter 6, these I/O libraries and functions are not supported. Therefore, they are placed in a separate file, global.c, such that they could easily be removed without major changes to the code. In addition, all define parameters and structure definitions are placed in the global header file, global.h. An illustration of how the C code is organized through the different files is shown in Figure 4.3. The global header file is included in cwt2.h and QRSDet3.h as indicated by the arrows in the figure. In addition, the cwt2 header file is included in the QRSDet3 header file in order to include the cwt2 function calls into the QRSDet3 function. Different define parameters in the global.h header file are created to easily disable or enable the I/O libraries and operations, the cwt2 test interface, and/or the QRSDet3 test interface. In a processor implementation, the global variables would be placed in the data memory, and the local variables would be assigned to the local registers. Therefore, large arrays and matrices are placed globally, and intermediate control variables for each function calls are allocated locally within their function.

The conversion of the Matlab prototype to a floating point application was done very quick, as expected. However, the implementation required some changes before the application worked correctly. When the floating point application produced the exact same results as the original Matlab code, some optimizations were done. The changes applied to make the Matlab prototype work in a C environment are presented in Section 4.5.1. In Section 4.5.2, the optimizations of the floating point application are described.

4.5.1 Implementation of the floating point application

The difference between the floating point application and the Matlab prototype, is that the C application required the creation of some structures containing information about the actual used sizes of large arrays and matrices. Without this information, the loops in the application would iterate through all elements of an array, even if the whole array is not used at run-time. Since the C language uses 0-indexing of arrays, and Matlab uses 1-indexing, a conversion of the accessing of elements had to be done as well. In addition, some issues related to the run-time memory usage occurred. These issues did not occur when simulating the Matlab prototype, because Matlab automatically adds more elements to an array or matrix if needed. This is not possible in C. If there is an attempt to write to, or access an array or matrix location that is larger than what is pre-allocated, the application will fail. Three changes were made to the code before the floating point application produced the exact same results as the original Matlab model. These changes are described in the rest of this section.

Creation of special complex data structures

In order to represent several arrays in the code and keep track of the needed length, a special complex data structure is created. This C structure contains a size parameter and a list of a fixed size of the maximum input interval, times the maximum sample frequency. The list is made big enough to contain the result of the convolution step for an ECG sample frequency of 1000 Hz. A similar complex data structure is also created for the coefficient

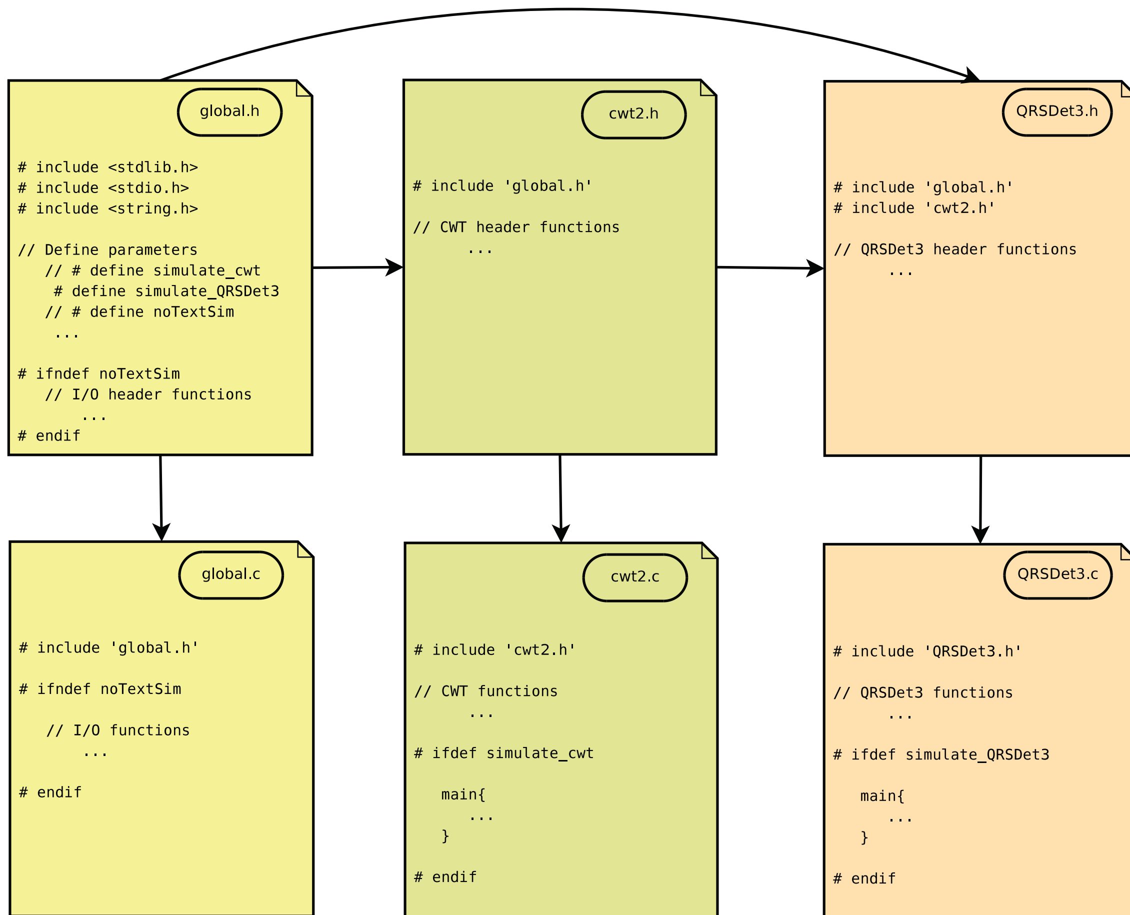

Figure 4.3: The C code structure.

matrix, containing one integer for the number of rows and one for the number of columns that are used in that matrix.

Run-time memory challenges

The simulation of the QRSDet3 C code showed that for some special ECG samples, i.e., the one displayed in Figure 4.4, there were many clusters detected around the beats. This required much more memory in specific matrices than what was pre-allocated in the Matlab prototype, resulting in a failure of the application. This problem was also studied in the Matlab prototype, and it turned out that Matlab automatically increased the size of the matrices and arrays without giving any warnings.

In addition, some column sizes assigned to some matrices were not big enough to handle all the elements of the clusters. To correct this issue, the these column sizes are set to the same size as the column size of the clusters. A new define parameter, called $FACTOR$, is also introduced. This parameter is multiplied with all array sizes of all arrays which are dependent on the calculation of clusters. This parameter is set to the value of 50, giving more than enough memory for the arrays and matrices. This over-dimension is done to make sure that special cases of the ECG signal, i.e., the one in Figure 4.4, are covered.

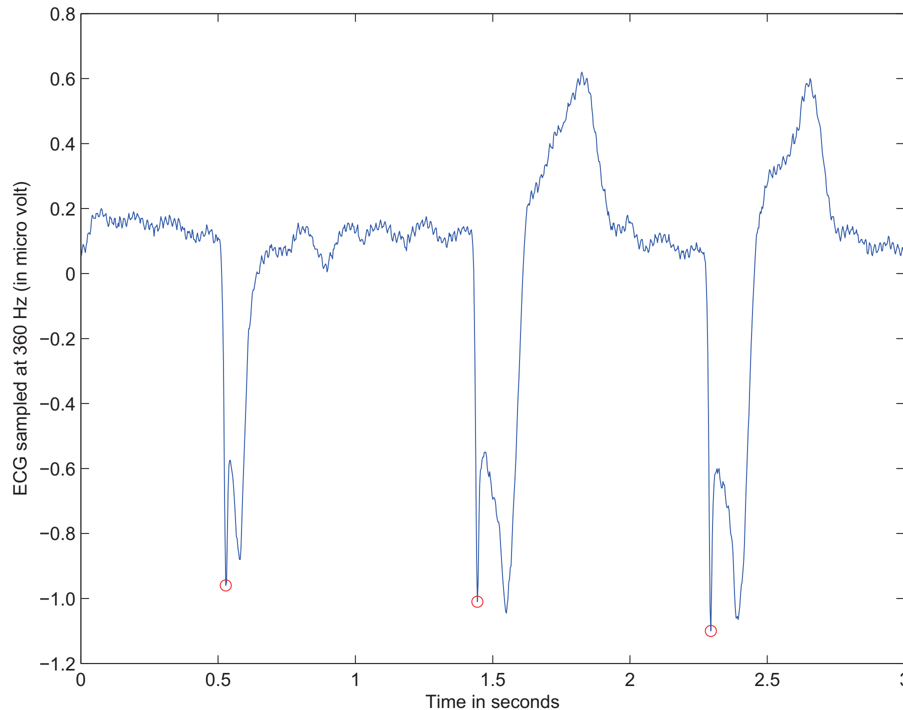

Figure 4.4: Special case of an ECG sample, where the red circles indicate the beats detected by the original Matlab model.

Initialization of arrays and matrices

Some minor issues related to the assignment of some temporary arrays and matrices were also detected in the validation process. Those issues occurred because these arrays and matrices were not reset before they were re-used from previous iteration rounds in some of the for-loops.

Finally, after correcting all issues, the simulation gave the exact same results as the original Matlab model.

4.5.2 Optimization of the floating point application

After running the initial version of the C application, and the correctness with regards to the Matlab model was verified, the optimization work could start. Optimizations like the ones described in Chapter 3 were applied, and the application was validated again. In the rest of this section, the applied optimizations are described.

Insertion of lookup tables

In the cwt2 function, the calculation of the reference mother wavelet is pre-calculated and stored in a lookup table. This lookup table is loaded into memory at the startup of the program. A lookup table for a square root calculation, containing 20 elements is also made. Some other lookup tables are created as well for some intermediate steps.

An index array which is used in order to select the elements for the scaled reference wavelet is also stored in a lookup table. This lookup table was created by making a matrix with the longest scale the default row size, and for every scale under this size, zeros are added at the end. This matrix is simplified into a long array, so that the first row is placed at the beginning of the array, while the last row is placed at the end of the long array. However,

this table is very large since it is different for every scale. Therefore, this lookup table has 6420 elements, where most of them are zeros because of the difference of the scale sizes.

These optimizations resulted in a CWT function without any complex computational parts. The only complex things to consider here (except from the cosine function in the Hanning window step, which is optimized in Section 4.6) are a few multiplications and some memory accesses.

Inline function definition

In the QRSDet3 file, some functions are declared *inline* in order to force the compiler to optimize them away.

Reduction of read and write operations to global memories

Code portions like the one in Code Snippet 4.1 are re-designed to code like the one shown in Code Snippet 4.2. This is done based on the principle of temporal locality, as described in Chapter 3, in order to optimize the program for processor implementation.

```
1  for(m = m_min; m ≤ m_max; m++){
2   _convrs.array[n-1] = _convrs.array[n-1] + (_x.array[n-m]*_h.array[m-1]);
3  }
4
5  // Divide result with the root of s
6  _convrs.array[n-1] = sqrt_lookup[_s-1] * _convrs.array[n-1];
```

Code Snippet 4.1: Code before optimization

```
1  temp = 0;
2  for(m = m_min; m ≤ m_max; m++){
3   temp = temp + (_x.array[n-m]*_h.array[m-1]);
4  }
5
6  // Divide result with the root of s
7  _convrs.array[n-1] = sqrt_lookup[_s-1] * temp;
```

Code Snippet 4.2: Code after optimization

In Code Snippet 4.1, the *_convrs.array* list is located in memory, and in the for-loop, this array is both read from and written to. A better alternative is to create a local variable, which in a processor could be a local register, and use it as shown in Code Snippet 4.2. This would reduce the amount of memory accesses to *_convrs.array* significantly.

The resulting floating point application is converted to fixed point, based on the theory of fixed point conversion, as presented in Chapter 3. This fixed point application is presented in the next section.

4.6 The fixed point application

In this section, the fixed point conversion of the C application is presented, and most of the techniques used are based on the theory discussed in Section 3.4. First, the changes to the floating point application are presented in Section 4.6.1. The optimizations done are then presented in Section 4.6.2, before the validation of the application is described in Section 4.6.3.

4.6.1 Changes to the floating point application

When converting a floating point application to fixed point, it is important to first analyze the requirement for the accuracy, i.e., the Q-notation for the fixed point numbers, as defined in Section 3.4. The input values to the algorithm from the IMEC database are integers, where signals between -3.2 and 3.2 μV was sampled into 12 bit unsigned numbers, [RLG08b]. The MIT/BIH database used 13 bit signed numbers. This gives the input signal an integer range of 11 or 12 bits, according to which database is used. Assuming that not more than 12 bits are needed for the input values, 16 bit signed numbers is decided to be used for the fixed point representation.

After deciding how many bits is going to be used to represent floating point numbers, the Q-notations for all floating point variables could be determined. All floating point variables and lookup tables are then adjusted according to their Q-notation. Addition and multiplication operations in loops are adjusted by shifting the numbers in every iteration, in order to preserve the desired accuracy. Rounding operations and conditional statements with floating point numbers are modified using shift operations and alternative representations. All divisions with constant values are eliminated by using shift operations, and a cosine lookup table is created with pre-calculated values. In the rest of this section, the implementation of all these changes are discussed separately, before introducing the optimizations done to the resulting application, in Section 4.6.2.

Determining the Q-notation for the floating point variables

By analyzing the integer range of all variables in the floating point application, including the fractional accuracy required, mainly 2 different Q-notations are determined, Q2.14 and Q10.6.

For most variables, including the coefficients after the convolution step, the Q2.14 signed integer format is chosen. This decision is based on the largest absolute value of the coefficients, calculated from the square root lookup table in the CWT function. All other coefficients had decimal values between minus one and one, with small values close to zero. From Equation 3.5, the square root coefficients have maximum values with n equal to one. Most signals represented in floating point are therefore shifted 14 times to the left, following Equation 3.5. However, many of those variables are used in additions and multiplications inside loops. Therefore, they had to be analyzed in every step in order to assure that all variables are shifted correctly.

For the threshold parameters, the Q10.6 signed format is chosen because of large integer values. The threshold parameters do not affect any other signals in the code. Therefore, assigning more bits for the integer representation in this case do not introduce any challenges.

Adjustment of addition and multiplication operations in loops

Within the convolution step in the cwt2 function, *_h.array* is the input ECG signal, as shown in Code Snippet 4.3. This array is assumed to have values with a 13 bit signed range. *_x.array* contains values shifted 14 bits to the left, from the reference mother wavelet lookup table. Therefore, the expected number of bits after the loop could be theoretically calculated as shown in Equation 4.1.

$$RB = 12 + 15 + s + ceil[log2(m_max - m_min + 1)], \tag{4.1}$$

where RB is the number of bits in the resulting *temp* variable, 12 is from $_h$, $15 + s$ is from $_x$, s is the signed bit, and $m_max - m_min + 1$ is the number of iterations this loop

is performed. This calculation allows approximately 4 bits for overflow if a 32 bit result is desired.

```
1   for(m = m_min; m ≤ m_max; m++){
2    temp = temp + (_x.array[n-m] * _h.array[m-1]);
3   }
```

Code Snippet 4.3: Code consideration of fixed point addition in a loop

$m_max - m_min + 1$ has a maximum value of $|1078|$. According to Equation 4.1, the ceiling statement is the requirement for overflow. In this case, 11 bits are needed. This means in theory that a 39 bit register is needed to handle the worst case accurately. The loss in accuracy here is assumed to be acceptable. Otherwise, the use of a 64 bit result would be necessary, and that is expensive in terms of area. If the accuracy turned out to be unacceptable, then a 39 bit specialized result register could be created in an ASIP, but a 64 bit standard register would have to be used in a general purpose register. Based on these considerations, the solution is to accept 4 bits for overflow, in order to eliminate the need for large registers.

Multiplication operations as shown in Code Snippet 4.4 also needed to be adjusted properly, in order to keep the result variable size as compact as possible. It is in a for-loop, and the values of the lookup table *sqrt_lookup* are signed values, shifted 14 bits to the left.

```
1   _convrs.array[n-1] = sqrt_lookup[_s-1] * temp;
```

Code Snippet 4.4: Code before fixed point conversion

The range of the *temp* variable is in this case 28 bits, with 4 bits for overflow in a 32 bit register. In addition, the values of *temp* are already shifted 14 bits to the left from Code Snippet 4.3. The result of the multiplication is therefore shifted 28 bits to the left. Since *_convrs.array* is an array assumed to have values shifted only 14 bits to the left, the result needs to be shifted back 14 bits. As shown in Equation 4.2, the size of the multiplication result requires 42 bits.

$$RBS = 15 + 27 + s, \tag{4.2}$$

where RBS is the number of bits in the resulting variable, 15 is from *sqrt_lookup*, 27 is from *temp*, and s is the signed bit.

The result is required to be a number that could fit in a 32 bit register. In this case, two possible solutions could be applied. Solution one is to do as shown in Code Snippet 4.5.

```
1   _convrs.array[n-1] = (sqrt_lookup[_s] * temp) >> 14;
```

Code Snippet 4.5: Code using solution 1 based on Code Snippet 4.7

Here, the the multiplication is performed, before the result is shifted. The end result fits within a 32 bit register, with minimum loss of accuracy, but 42 bits for the temporary result after the multiplication is still needed. The array *_convrs.array* must then be defined as *long int*, but then a 64 bit representation would be used.

Solution two is more optimal with respect to area, but comes with the cost of less accuracy. As shown in Code Snippet 4.6, only 32 bit registers are needed for the intermediate step and the answer. However, some accuracy is already lost before the multiplication takes place.

```
_convrs.array[n-1] = sqrt_lookup[_s] * (temp >> 14);
```

Code Snippet 4.6: Code using solution 2 based on Code Snippet 4.7

The differences are best shown through an example. The alternative solutions are shown using binary numbers in Code Snippet 4.7. Under the assumption that there is enough bits for overflow, solution 2 was chosen.

```
// Solution 1:
(101 * 110) >> 2 = 111 = 7 // 111|10 (maximum precision)

// Solution 2:
// ( 101 >> 2) = 1|01 --> Accuracy of the 2 LSB is thrown out
( 101 >> 2) * 110 = 110 = 6 // (precision lost)
```

Code Snippet 4.7: Code before fixed point conversion

Changes to the rounding operations

Some variables are rounded at run-time. A rounding function is performed by adding a half to the value and truncate the fractional part. In order to round one variable that is shifted 14 times to the left, one could simply add the value one, shifted 13 times to the left. The binary value of 0.5 is 0.1, and by shifting this value 14 times to the left, one would get the same result as taking the binary value 1 and shift it 13 times to the left instead. Similarly, a floor operation is performed by truncating the fractional part. A ceiling operation is performed by taking the value of one and shift it 14 times, before subtracting the value of one from the result. This result is then added to the number, and the fractional part is truncated.

An example of how this looks like in the C code is given in Code Snippet 4.8.

```
/*
 * FIXED POINT notice:
 *  round --> (int) (x+0,5)
 *  shifted 0 times (rounded using fixed point)
 */
x1 = (int) (((x1<<14)+(1<<13))>>14);

/*
 * FIXED POINT notice:
 *  x2 is shifted 14 times, to take the
 *  ceil the following is added: ((1<<14)-1)
 *
 *  x3 is shifted 14 times, to take the
 *  floor the value is truncated
 */
x2 = (int) ((x2 + ((1<<14)-1))>>14);
x3 = (int) ((x3)>>14);
```

Code Snippet 4.8: Fixed point round operations.

Elimination of divisions with constant values in conditional statements

Some conditions in if-statements are changed, as shown in Code Snippet 4.9. Alternative 1 was changed to alternative 2 in this example.

```
1  // Alternative 1:
2  x < 0.02 * fs
3
4  // Alternative 2:
5  50 * x < fs
```

Code Snippet 4.9: Floating point elimination

Implementation of a cosine lookup table

A cosine is used to calculate the Hanning window, and a part of that code is shown in Code Snippet 4.10. The fixed point conversion of this code is shown in Code Snippet 4.11.

```
1  nleft.array[i-1] = 0.5*(1-cos(2*((double) M_PI)*nleft.array[i-1]/(((double)
       _N)-1)));
```

Code Snippet 4.10: Floating point cosine calculation.

```
1  nleft.array[j-1] =
       8192-((8192*cos_func[((nleft.array[j-1]*n_lookup[_N-20])>>21)])>>14);
```

Code Snippet 4.11: Fixed point cosine calculation.

As shown in Code Snippet 4.11, the number 8192 is the same as 0.5 shifted 14 times to the left and rounded. Equation 4.3 was pre-calculated for every possible $_N$ values and placed in a lookup table, where the values are shifted 14 times to the left. This table has all values for $_N$ in the range from 20 to 200. By accessing the lookup table using the index $_N - 20$, one could achieve the desired value for the corresponding $_N$.

$$\frac{2\pi}{_N - 1}, \; 20 \le _N \le 200 \tag{4.3}$$

All values of $nleft.array$ are shifted 14 times to the left. Since all n_lookup values are shifted 14 times to the left, the cosine input must be corrected to a number in the Q2.14 format. This is done by first shifting $nleft.array[i-1]$ to the left 7 times, and then shift the answer of the multiplication back 28 times. This is similar as shifting the result of the multiplication 21 times to the right. All these considerations resulted in the code shown in Code Snippet 4.11. The cos_func lookup table consisted at this point of one period of a cosine wave, spread over 804 samples. This cosine lookup table is changed into a function call and further optimized, along with other code optimizations. All optimizations are described in Section 4.6.2.

4.6.2 Optimization of the fixed point application

After converting the floating point application into fixed point, the application was optimized further. Some lookup tables are optimized by the usage of alternative representations, resulting in smaller lookup tables. A cosine function based on the CORDIC method [Vol59] is also implemented, using a smaller lookup table than the previous one. All divisions are replaced by a software division function, made to support divisions in calculations located outside the critical loop of the application. The total memory usage is also further optimized, by inserting some constraints based on an analysis of the worst possible heart rate of a human being. A preprocessor macro is also made in order to optimize the code further. All optimizations are described in the rest of this section.

Lookup table size reduction

In the calculation of the scaled mother wavelet in the cwt2 function, the large lookup table in the floating point application consisted of 6420 elements. Since this table is quite big, an alternative lookup table is created from Code Snippet 4.12 in Matlab.

```
1 for i=1:20
2  x(i) =  floor((1/i)*2^9);
3 end
```

Code Snippet 4.12: Calculating the alternative lookup table in Matlab

All values in this lookup table are represented with a Q7.9 signed number (shifted 9 bits to the left). In addition, this lookup table is never used in other calculations than the ones involving the selection of index positions of the reference mother wavelet lookup table, as described in Section A.1. The index for a particular value in the reference mother wavelet is selected by the calculation shown in Code Snippet 4.13. All parameters in the divisions are changed with values in the power of two. This was done in order to optimize away the divisions and insert shift operations instead. In addition, this scaling solution depends on both the scale value and the current iteration. Instead of putting many coefficients in a large matrix, a scaling of one parameter is done while the values of the other parameter are selected from the new lookup table (*x_lookup*). The resulting *_psi_scale.array* is in this case similar to the one calculated in the floating point application (using the large lookup table). This optimization has reduced the memory size required for the *x_lookup* significantly, from 6420 down to 20 elements.

```
1   for (j = 1; j ≤ scl+1; j++){
2    _psi_scale.array[scl+1-j+1-1] =
3     psi_lookup[((int) (((( ((int) (((j-1)* x_lookup[s-1]))) *
        255+(1<<12))>>13))))];
4   }
```

Code Snippet 4.13: Optimization of the assignment of *_psi_scale.array*.

Combining loops

Some of the calculations in the Hanning window step, in the QRSDet3 function, were done in separate loops. The Hanning window step is therefore optimized by combining these calculations together in the same loop, giving a faster execution time.

Implementation of a cosine function based on the CORDIC method

In order to optimize the cosine function with a smaller lookup table, a special lookup table consisting of 202 samples of the first quadrant of a cosine is made (with steps of 1/128). A cosine function is created in addition, in order to handle the cosine input and return the right value dependent on which quadrant the input value is placed. This solution is based on the method of CORDIC cosine, as described in [Vol59]. The basic idea here is to exploit the fact that the points in the first quadrant of a circle, created by the cosine function, is the same in all other quadrants. The only difference is that the points are rotated differently, dependent on which quadrant the desired value is located. The input to this function is the angle in radians, shifted 14 bits to the left, and the output is the cosine of the input value, shifted 14 bits to the left. The implemented cosine function is shown in Code Snippet 4.14.

```
1 /*
2  * FIXED POINT notice:
3  *  All the values in the cosine table are shifted 14 times to the left.
4  *  First and second quadrant: M_PI*128/2 = 201 (approx.)
5  *  Second and third quadrant: M_PI*128 = 402 (approx.)
6  *  Third and fourth quadrant: 3*M_PI*128/2 = 603 (approx.)
7  *  Fourth quadrant: 2*M_PI*128 = 804 (approx.)
8  */
9 int cos_func(int beta){
10  // If the angle is in the first quadrant
11  if(beta ≥0 && beta < 201 ){
12   return cos_tab[beta-1];
13  }
14  // If the angle is in the second quadrant
15  else if(beta ≥201 && beta < 402){
16   return (-1*cos_tab[(sizeof(cos_tab)/sizeof(*cos_tab) - beta + 201 - 1)]) ;
17  }
18  // If the angle is in the third quadrant
19  else if(beta ≥ 402 && beta < 603){
20   return (-1*cos_tab[beta - 402 - 1]);
21  }
22  // If the angle is in the fourth quadrant
23  else if(beta ≥ 603 && beta < 804){
24   return cos_tab[(sizeof(cos_tab)/sizeof(*cos_tab)) - beta + 603 - 1];
25  }
26 }
```

Code Snippet 4.14: Fixed point cosine calculation.

The software division

Not all divisions are removed, because there are some mean values calculated at run-time in the QRSDet3 function. These mean values are dependent on the input signal, and can not be predicted. Therefore, a lookup table could not be used. A software division function is therefore implemented, using only additions and shift operations. This function is time consuming, and would make the application significantly slower if the division is located in a critical loop of the program. In the case of the algorithm, the division operations are not performed in a critical loop. A better solution is to perform the division operation in hardware, but if a software division function is already made, then it could easily be changed with a hardware division when mapped to the processor later, without major modifications to the code. In addition, the application would be embeddable without using any more complex operations than multiplications, additions and shift operations.

Memory optimization of the fixed point application

The memory required for the floating point numbers, compared to their fixed point representation, is twice as much in the floating point application than in the fixed point application. 32 bits are usually allocated to represent an integer number, whereas a floating point variable with double precision, like the ones used in the floating point application, requires 64 bits. Therefore, the conversion to fixed point alone already reduced the memory required for all floating point values by half.

A final analysis of the memory usage was performed statically to the code, and the memory usage, in terms of total number of integers used, turned out to be approximately 441 883 integers. Since four bytes is required for one integer, the memory usage was estimated to be approximately 1.8 MB for the integers in the application. This memory usage is unacceptable, because it is too much for this embedded application. Therefore, an analysis was performed on all the large arrays in order to determine the maximum memory size necessary.

According to [Ham80], a pulse for a human being can not be more than 300 heart beats per minute. A human being with this heart rate would have maximum 5 beats per second. For a 3-second interval, maximum 15 beats could be detected. This would generate clusters of 5625 (75x75) elements in the C code. This size includes the assumption that each cluster could have maximum 5 samples. By scaling down the cluster sizes, all other arrays and matrices directly dependent on the clusters could also be scaled down. Therefore, it is sufficient to multiply all sizes of the arrays and matrices related to the clusters with the total number of beats that could be detected in a 3 second interval. The $FACTOR$ parameter defined in the floating point application was used in the scaling of all arrays related to the clusters. Therefore, by reducing it from 50 to 15, the memory usage was reduced significantly because all arrays and matrices related to the clusters are dependent on this parameter.

All these memory optimizations were done first in the floating point application and simulated for different values of $FACTOR$, before finalizing the changes in the fixed point version. The simulation of the floating point application produced the exact same results as the original Matlab model for all values of $FACTOR$ larger than 10. In order to cover the extreme case of a heart rate of 300, the $FACTOR$ parameter is decided to be 15. After these optimizations, the required memory usage was reduced to 97 463 integers, and this corresponds to approximately 400 KB of memory space for the integers. Figure 4.5 gives an illustration on how the number of kilo bytes needed for integers in the application changes for different values of $FACTOR$. From this figure, one could see that the memory usage is reduced significantly.

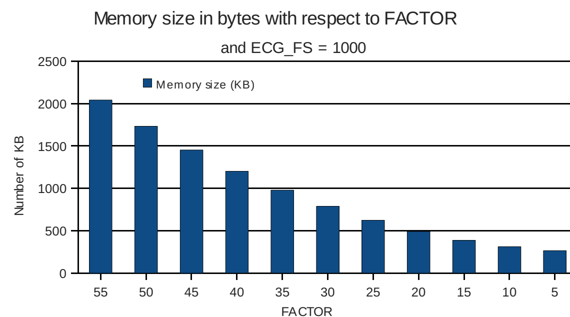

Figure 4.5: Memory usage with respect to $FACTOR$ and $ECG_FS = 1000$.

Despite the results, it is worth noting that when $FACTOR$ equals to 5, only two test signals from the databases fail the simulation due to the lack of memory. For $FACTOR$ equal to 10, the floating point simulation results are the exact same as the Matlab model. This simply means that there are none or to few test signals that produce the extreme case of a heart rate close to 300 per minute. Maybe some artificial signals should be produced in order to test the limits of this algorithm, but this is in any case a subject to future work.

In addition to the scaling of $FACTOR$, the sample frequency of the ECG signal will be fixed to one specific value after implementation on a processor. Since the ECG array size depends on the number of samples, one could reduce the memory size by using a frequency with a smaller value. The scaling parameters are shown in Code Snippet 4.15, and Figure 4.6 shows the reduction in memory size by using ECG signals sampled at shorter frequencies. ECG_FS is in Code Snippet 4.15 the sample frequency of the ECG signal.

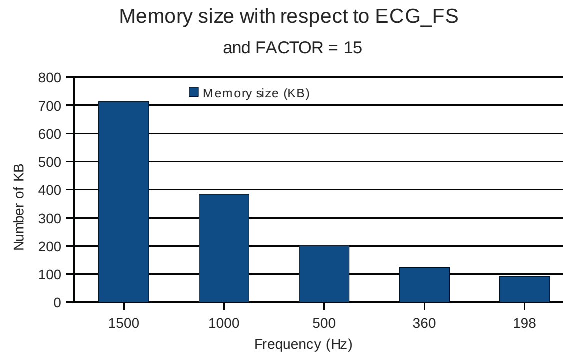

Figure 4.6: Memory usage with respect to ECG_FS and $FACTOR = 15$.

```
1 // Global parameters
2 #define ECG_FS 1000
3 #define FB0 15
4 #define FB1 18
5 #define SCALE0 ((int) ((((4112/FB1)*ECG_FS))>>14))
6 #define SCALE1 ((int) (((4112/FB0)*ECG_FS + ((1<<14)-1))>>14))
7 #define PSI_SCALE_SIZE (16*SCALE1)
8 #define ECG_INPUT_INTERVAL 3
9 #define MEXICAN_HAT_SIZE 255
10 // Number of scales and fs must be known, and fs
11 // should be big enough to cover the size of the
12 // convolution result. The convolution result is
13 // the number of samples of the ECG signal plus
14 // the size of psi_scale.
15 #define MAX_NUM_OF_SCALES (SCALE1-SCALE0+1)
16 #define MAX_FS (ECG_FS+((int) (PSI_SCALE_SIZE/3 + 1)))
17 // Factor can not be under 15
18 //(the matrices are then too small to cover a heart rate of 300)
19 #define FACTOR 15
```

Code Snippet 4.15: Global define parameters in global.h.

It is worth noting that the C application is not designed to be used for frequencies of the ECG signals above 1000 Hz, because the databases do not have any signals sampled at higher frequencies. The lowest frequency used in the databases is 198 Hz, which is a reasonable frequency for processor implementation, giving the best optimal memory usage. For this frequency, the application needs approximately 100 KB of memory just for the integers in the program.

Insertion of preprocessor macros

In addition to all memory optimizations, the built-in absolute function *abs* in C is replaced with the preprocessor macro shown in code snippet 4.16. This is done in order to prepare the fixed point application for processor implementation without the use of the standard GNU C libraries.

```
1 #define abs(x) ((x)>=0?(x):(-(x)))
```

Code Snippet 4.16: Pre-processor macro optimization of the *abs* function in C.

Re-usage of the coefficients matrix

It was observed that the largest matrix in the application, the matrix containing the coefficients after the CWT step, is never used in the steps related to the clusters of beat. In the step of finding the clusters of beat, some temporary values are stored in a temporary matrix.Although this matrix was smaller than the coefficients matrix, it is optimized away by re-using the coefficients matrix for the temporary storage.

4.6.3 Validation of the fixed point application

After converting all floating point values to fixed point integers, the program was simulated. The simulation of the fixed point application was done in a similar manner as the floating point application. However, some modifications are done in the Matlab interface scripts. All floating point input values to the algorithm are shifted 14 bits to the left and rounded before they are written to file, except from the thresholds, which are shifted 6 bits. The precision of the fixed point conversion is by Equation 3.7 calculated to be 0.000061 for all signals that are shifted 14 bits, and 0.0156 for the threshold parameters. The accuracy is assumed to be good enough for the thresholds. In addition, the detected beats read from file are shifted back 14 bits, and the new calculated thresholds are shifted back 6 bits before they are sent through the rest of the test bench. These modifications in the Matlab interface scripts are also needed in a receiver system which is going to read the resulting data from the application. The results from this simulation are good enough, with less than 0.5% differences from the results produced by the original Matlab code. These results are shown in Section 4.7.

It is worth noting that if the fixed point representation for the threshold values get out of their specified fixed point range in accuracy, the thresholds get to low. This might cause some issues related to the memory sizes of the cluster. The solution used to solve these issues, is explained in Appendix D.

4.7 Results

In this section the results are presented, and they are discussed in Section 4.8.

4.7.1 Simulation results

The simulation of the floating point application produced the exact same results as the original Matlab code. Therefore, only the results from the fixed point application are shown here.

When simulating the fixed point application, the CWT function was first interfaced to the original Matlab code and simulated, before the QRSDet3 application was simulated. By doing it this way, one could verify the functionality of the CWT function before simulating the whole program.

Figure 4.7 displays the simulation results from the MIT/BIH database, simulated over the thresholds from 10% to 90% with intervals of 5%. The same simulation was done for the IMEC database, and the results are shown in Figure 4.8. In these figures, the sensitivity (Se) in percent is plotted with respect to the positive predictivity (+P) in percent. The optimal threshold is achieved when the sum of Se and +P are maximized. From the figures one could locate that threshold by looking at the point on the graph closest to the upper left corner. The optimal threshold is found to be 30%.

Figure 4.9 illustrates the differences in the noise test. From this one could see that the fixed point application has more noise than the original Matlab code, but the differences are less than 0.5%.

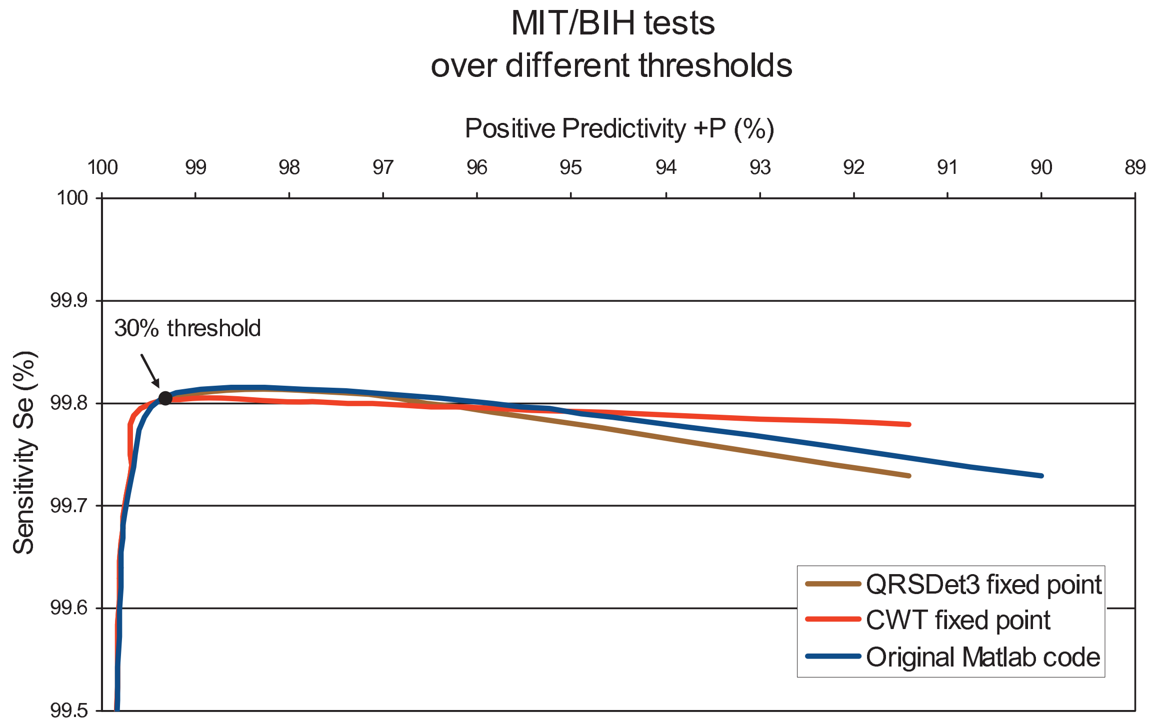

Figure 4.7: The MIT/BIH test over all thresholds.

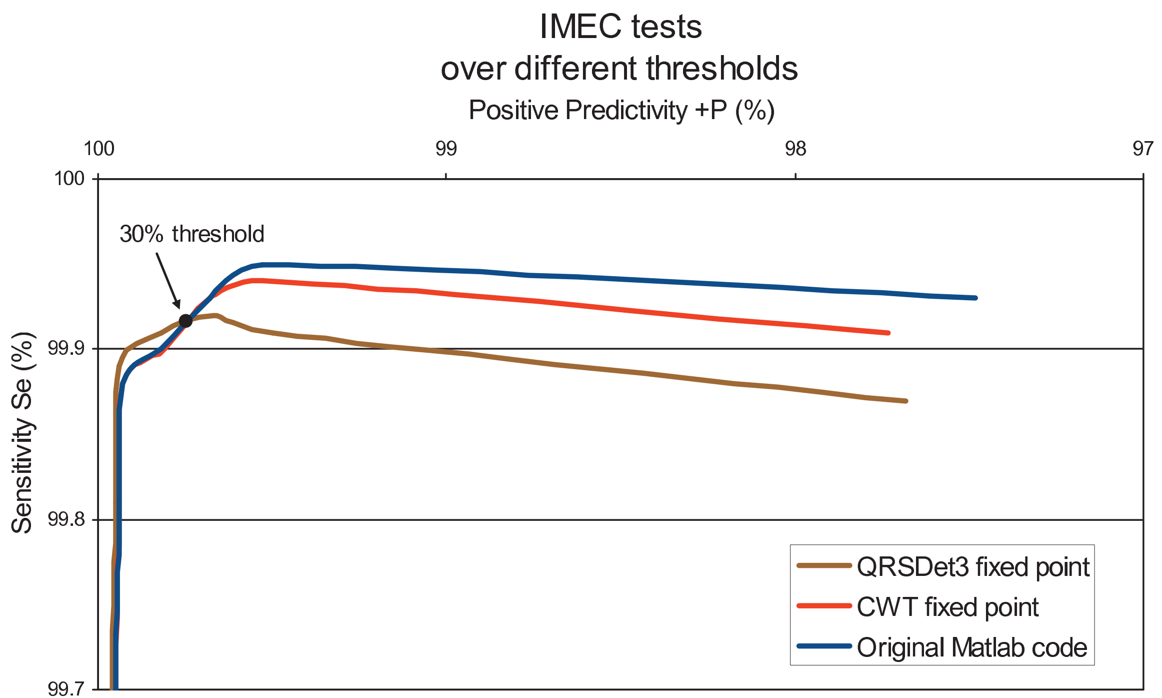

Figure 4.8: The IMEC test over all thresholds.

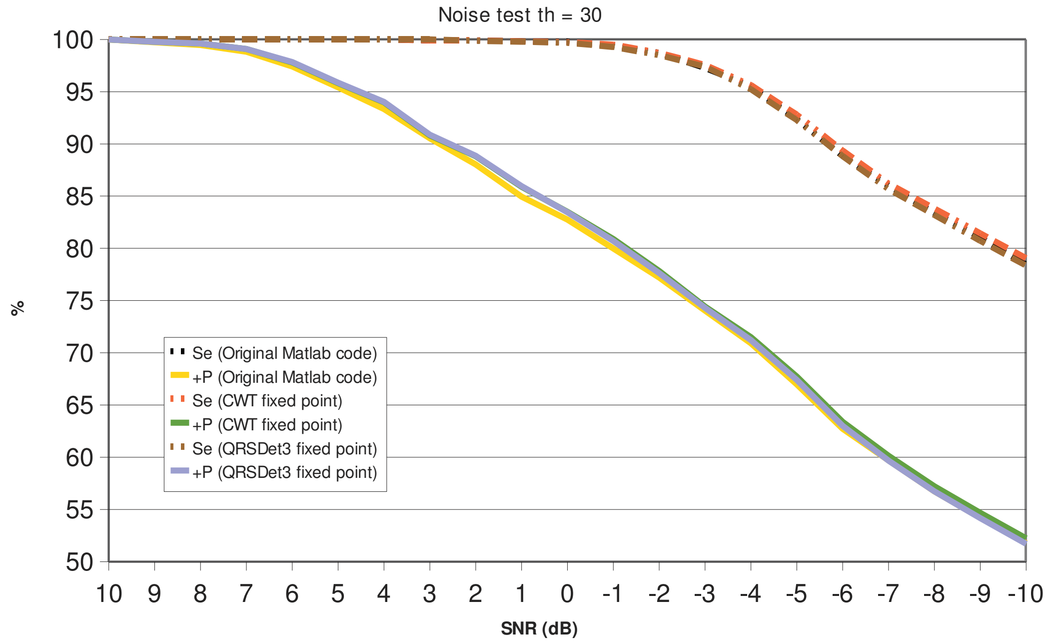

Figure 4.9: Noise tests for th=30%.

4.7.2 Profiling results

Both *gprof* and *Valgrind* was used to get profile information. For the profiling, a 3-second part from the peak test, sampled at 1000 Hz, was used from the IMEC database. With this frequency, the application uses all allocated memory at run-time. This 3-second signal is graphically displayed in Figure 4.10.

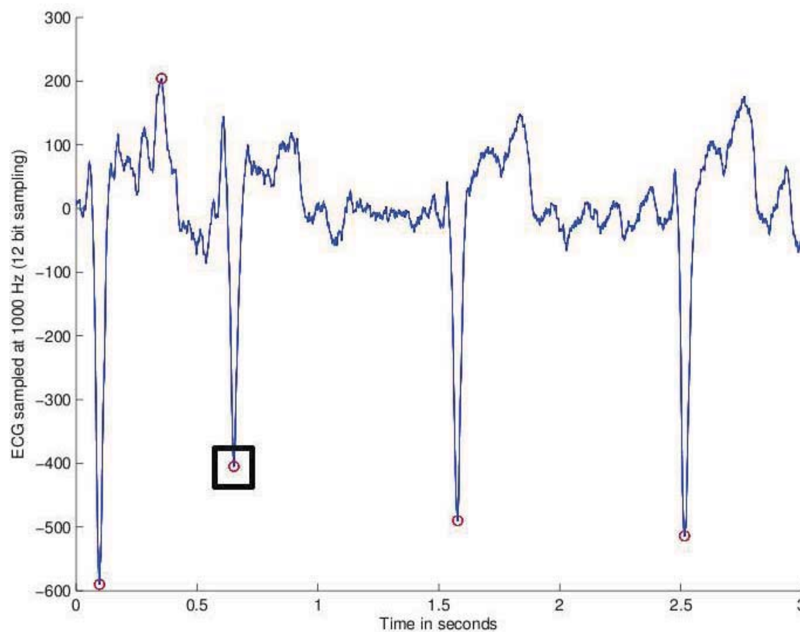

Figure 4.10: The sample of an ECG signal used for profiling.

Figure 4.10 illustrates a 3-second sample of the ECG signal taken from a patient diagnosed

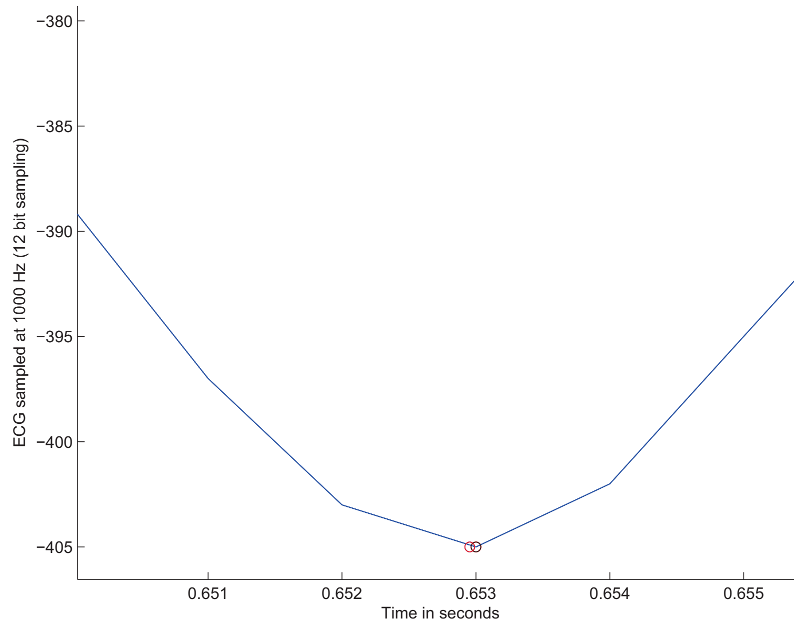

Figure 4.11: The third beat detected by the sample in Figure 4.10.

with Arrhythmia. In this figure, the detected beats from the fixed point C application are shown as red circles on the graph. The plot of the detected beats from the original Matlab code, are not seen in this figure. Figure 4.11 displays a close up of the third beat (the marked area) in Figure 4.10. If one takes a closer look at the same graph, one could see that the differences between the fixed point C application (red circles) and the original Matlab code (black circles) are very small. The other beats have similar differences between the fixed point C application and the original Matlab code. In order to give the profiling tools enough time to calculate the usage at run-time, the application was executed in a loop where the same calculation of the 3-second interval was done one hundred times.

The *gprof* profiler indicated in which function calls most of the execution time was spent. From the output file, the convC function call is executed the most, and this function call was assumed earlier to be in the critical loop of the application. A part of the *gprof* results is shown in Code Snippet 4.17.

From Code Snippet 4.17, one could see that, relative to the main function and QRSDet3 function, the convolution function call was executing in approximately 87.7 % of the execution time. It is also called 500 times, which means five times for every call to the QRSDet3 function. This function call consist of two nested loops, where min and max functions are called in the outer loop, while the inner loop performs the multiplication and addition operation of the convolution. The min and max function calls are executed approximately 1.6 million times. This means that for one iteration of the whole application, these function calls are called approximately 16 000 times. The number of iterations performed in the inner loop depends on the values of conditions of the for-loop in Code Snippet 4.3. However, the profiling tools used in Chapter 6 showed that the inner loop iterates approximately 3.6 million times in this case. This loop consist of the multiplication and addition operation of the convolution, and by optimizing these operations, one could reduce the execution time.

```
                Call graph
index % time    self   children    called      name
                0.21   1.58      100/100          main [1]
[2]    100.0    0.21   1.58      100          QRSDet3C(intlist&, int, int*, int,
     int,
                                                        int, int, int, int, int*,
                                                        intlist&, intlist&,
                                                             intlist&) [2]
                0.01   1.57      100/100          cwtC(intlist&, int*,
                     matrixIntList&,
                                                        intlist&, intlist&) [3]
                0.00   0.00      500/1620000      min(int, int) [5]
                0.00   0.00      500/1620000      max(int, int) [6]
                0.00   0.00     3800/4200         soft_div(int, int) [16]
                0.00   0.00      500/500          hanning3C(int, int, int, int,
                                                               intlist&) [18]
                0.00   0.00      100/100          mean(intlist, int) [19]
-----------------------------------------------
                0.01   1.57      100/100          QRSDet3C(intlist&, int, int*, int,
                                                             int, int, int, int, int,
                                                             int*, intlist&, intlist&,
                                                             intlist&) [2]
[3]     88.3    0.01   1.57      100          cwtC(intlist&, int*, matrixIntList&,
                                                   intlist&, intlist&) [3]
                1.53   0.04      500/500          convC(int, int, int, intlist&,
                                                         intlist&, intlist&) [4]
-----------------------------------------------
                1.53   0.04      500/500          cwtC(intlist&, int*,
                     matrixIntList&,
                                                        intlist&, intlist&) [3]
[4]     87.7    1.53   0.04      500          convC(int, int, int, intlist&,
     intlist&,
                                                    intlist&) [4]
                0.03   0.00  1619500/1620000      min(int, int) [5]
                0.01   0.00  1619500/1620000      max(int, int) [6]
-----------------------------------------------
Index by function name
   [2] QRSDet3C(intlist&, int, int*, int, int, int, int, int, int, int*, intlist&,
                intlist&, intlist&)
   [3] cwtC(intlist&, int*, matrixIntList&, intlist&, intlist&)
   [4] convC(int, int, int, intlist&, intlist&, intlist&)
```

Code Snippet 4.17: Call graph results from *gprof*.

Valgrind showed similar results, and these results are in *kcachegrind* shown graphically, as shown in Figure 4.12. In this figure, a call graph is shown with the execution time relative to the the cwtC function. The convC function, which is called within the cwtC function, is executed at 99.39% of the total execution time of the cwtC function. 86.84% of the total execution time of the QRSDet3 function consisted of calls to the cwtC function.

In addition to the call graph, the results from the *memcheck* tools reported no memory leaks and no errors.

The memory needed for data memory (DM) and program memory (PM), in addition to execution cycles and stack usage, was found using the Target instruction-set simulator (ISS). In order to achieve these profiling informations, a general 16-bit base processor from the Target templates was modified into a 32-bit processor. The fixed point C application was then mapped to the new platform and simulated with the ISS. The Target tools are introduced in Chapter 5. The results from the ISS report are given in Table 4.1 for different ECG_FS frequencies, between 198 Hz and 1000 Hz. All test samples from the IMEC and MIT/BIH databases are sampled at frequencies between 198 Hz and 1000 Hz. The processor clock frequency required to process the ECG signal must be high enough to execute the application before two seconds has elapsed because of the overlap described in Chapter 2.4.

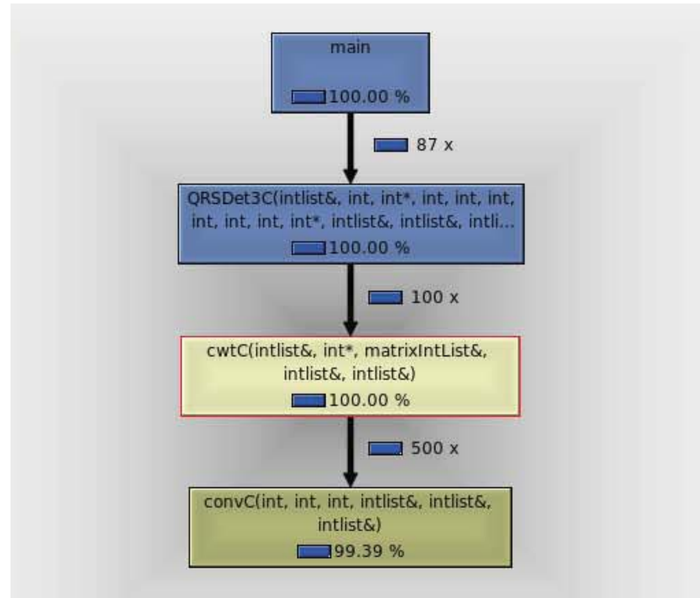

Figure 4.12: The call graph with the focus on the convC function.

This means that if the application needs 27 million cycles, then the processor must execute those cycles using a frequency of at least 13.5 MHz.

Profiling information from the Target tools.		
Frequency (Hz) (Hz)	*Data memory usage (KB)*	*Cycle count*
1000	512	27 761 417
500	300	6 341 634
360	180	2 913 374
198	150	1 589 178

Table 4.1: Memory usage and cycle count with different frequencies of the ECG signal. The program memory usage was constant, where 8 KB of memory was needed. 3320 instruction words was used in this case.

4.7.3 Application specifications for processor implementation

Based on the profiling and simulation results of the fixed point C application, the application specifications for a general processor implementation are summarized in Table 4.2. These specifications are based on the ECG sample frequency that gave the optimal solution in the analysis done for memory usage. In Chapter 6, it is concluded that a 32-bit processor is needed in order to be able to calculate all coefficients with acceptable accuracy. The limitations of the processor clock frequency is also discussed in Chapter 6. For an ECG signal which is sampled with a sample frequency of 198 Hz, the minimum clock frequency required for a processor is then 800 KHz. With this frequency, the program finishes just in time before the next set of samples arrive, as described in Chapter 6.1.

Application specification for processor implementation.	
Sample frequency (Hz) of the ECG signal	198 Hz
Data memory	150 KB
Program memory	8 KB
Stack memory	4 KB
Processor clock frequency (f_{clk})	800 KHz < f_{clk} < 100 MHz

Table 4.2: Application requirements for processor implementation.

4.8 Discussion of the results and conclusions

Even though the results of the fixed point application are satisfactory, there is still room for improvement. Due to the complexity of the Matlab model, there was not enough time to optimize the alternative code in the prototype for every built-in function. However, the execution time of the code is fast enough with the resulting fixed point application for real-time beat detection.

The data memory usage on the other hand, is far to much for a low power general processor. For instance, the MSP430F1611 microcontroller from Texas Instruments [Tex09] has reserved 48 KB for the program memory, while for data memory, only 10 KB is available. The fixed point C application will therefore not fit in the MSP430F1611 because it requires 150 KB data memory for beat detection of ECG signals sampled at 198 Hz.

There is not much one could do for memory optimization in software without changing the algorithm itself. For instance, the step for classifications of beats in the QRSDet3 function is based on Matlab indexing of matrices, as described in Appendix B. This is not easily converted to C. In order to change this, one has to figure out a new way to classify detected beats and to chose the clusters. In the fixed point application, an extra array is used for remembering the positions using C indexing of matrices, while the calculations are done in the same fashion as Matlab. If this step is to be optimized, one could reduce the memory more, but not significant enough to be considered. The MSP430F1611 would still not have enough space for the data memory.

Another alternative is to use smaller intervals of the ECG signal, i.e., 2 second intervals. The length of most arrays in the algorithm are dependent on the ECG input interval and the sample frequency of the ECG signal. Even though the interval is reduced to 2 seconds, one would only save up around 10 KB of data memory for a sample frequency of 198 Hz. One could reduce the sample frequency even further, but the requirements for the data memory would still be too much for the MSP430F1611.

In order to run the application, a DSP like TMS320C6413GTS500 from Texas Instruments [Tex06] could be used. This processor has enough memory, but it is more power consuming than the MSP430F1611.

In order to achieve better power efficiency, an ASIP is needed. Even if there existed a low power processor that had enough data memory for this application, the chance of having low power consumption for this application is very small due to the many iterations of the convolution function.

Another limitation of this algorithm is that it requires 32-bit integers for the desired accuracy, thus a 32-bit processor. It is possible to use a 16-bit processor to execute the application, as discussed in Chapter 6, but that would result in a more power consuming solution since most of the arrays and matrices in the code would have to be converted into long integer types. Then every assignment would have to use at least twice as many instructions, compared to a 32-bit processor, in order to do the same task. By adjusting the Q-factor

in the fixed point application, one could assign the numbers with 8-bit precision instead of 16-bit precision, eliminating the need for 32-bit integers. In that case, the algorithm would not produce good enough results for beat detection.

By using an ASIP, one could exploit the profiling information, and optimize the hardware for the application. In addition, the critical loop could be optimized using custom operations, multiply-accumulate (MAC) operations, etc. Instructions could also be pipelined in order to be executed in parallel within the same clock cycle. By optimizing the hardware for this application, it is possible to achieve a solution much more optimized in terms of power consumption than using a low power general purpose processor.

The next chapter will introduce the Target development tools, before the ASIP implementation of the fixed point C application is presented in Chapter 6.

Chapter 5

The Target Development tools

This chapter presents the Hardware/Software Co-design (HW/SW Co-design) methodology, as well as the Target development tools, which are the development tools used for the ASIP design in this thesis. Since an architecture description language called nML is used in the ASIP design, an introduction to this modeling language is also given.

Section 5.1 briefly introduces the HW/SW Co-design methodology, and in Section 5.2, the Target development tools are presented. The nML processor description language is introduced in Section 5.3.

5.1 Introduction to the HW/SW Co-design methodology

In some applications, i.e., software applications, it is desired to speed up the execution time. If the software containing data processing functions is already optimized in software, and a faster application is needed, one could tend to HW/SW Co-design in order to achieve specific goals. Similar, if a hardware application needs optimization with respect to area and/or time, HW/SW Co-design could solve the problem. The methodology consist of partitioning a given application in a software module and one or more hardware modules. The part of an application which executes best on hardware is placed in the hardware module(s), i.e., data processing elements which require a large amount of clock cycles to execute on software. Similar, the part of the application that is best suited to be in software, is placed in software. Control functions and monitoring are examples of good suited software applications. HW/SW Co-design is defined as follows:

> "Hardware/Software co-design means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design" [WS97].

In other words, HW/SW Co-design includes modeling, partitioning, scheduling and designing a system, where one exploits the similarities in hardware and software, in order to achieve specific requirements. In general, splitting a system or application into hardware and software introduces several design challenges that varies with the application domain, implementation technology and design methodology. The HW/SW Co-design methodology has two primary advantages; more time to evaluate trade-offs, and it creates better hardware/software interfaces.

According to [KR05], any design methodology should do the following: provide a checklist for the design process, facilitate the communication of design team members, help to predict costs, aid in the creation of a working prototype, aid in the creation of a time line for the

development cycle, help with the identification of metrics, aid with requirements specification, and assist with the development of test procedures.

The goal of HW/SW Co-design is to do all of these things as well as allow designers to "predict" implementation, "incrementally refine" a design over "multiple levels of abstraction", and create a "working first implementation" [WK05].

There exist several platforms where the HW/SW Co-design methodology could be applied. What kind of architecture is needed, depends on the target application. Most digital systems use components with an instruction set architecture (ISA). Among the most common architectures available, one usually find platforms including Central Processing Units (CPU's), Application Specific Integrated Circuits (ASIC's), Instruction-set Processors (ISP's), Application Specific Instruction-set Processors (ASIP's), and/or Digital Signal Processors (DSP's).

Many papers have been published on the HW/SW Co-design issues, including modeling and simulation, generic integration and interfaces, custom HW/SW synthesis, and especially partitioning. The latter category includes manual approaches: coarse-grain or fine-grain, and automatic approaches: starting from hardware allocation first or from software allocation [DCDM97].

Different design environments exists, i.e. homogeneous (one design language, i.e., SpecC) and heterogeneous (multi-language environment like C and VHDL combined).

In general, HW/SW Co-design methodology consist of four steps, according to [NdMM07]. First, a system specification is made. It can either be a software model (like in C++), or a hardware description using a hardware description language (like VHDL). A software specification indicates that the design follows a software-oriented approach, and if the system specification is written with a hardware description language, then the design is following a hardware-oriented approach. In a homogeneous design environment, the same description language is used for both hardware and software. The next step consists of hardware and software partitioning where the designer needs to determine which part to be implemented in software and which part to be implemented in hardware. In the third step, both hardware and software are designed and synthesized. In addition, an interface between hardware and software is designed and synthesized. This involves adding latches, FIFOs or address decoders in hardware, and inserting code for I/O operations and semaphore synchronization in software. The final step of the design is to integrate hardware and software components into the target architecture.

The verification of the design could be done in every step of the design process, but during system integration, co-simulation is needed to verify that both hardware and software work together as expected. This could either be done directly online on the platform, or through dedicated software tools that allow simulation of hardware and software executing together.

Several development tools exist today which makes the design easier for a designer to create and implement embedded systems using HW/SW Co-design.

For ASIP design, dedicated development tools, such as Silicon Hive, or IP designer from Target Compiler Technologies, could be used. In this thesis, the Target IP designer tools were used, and an overview of the tools will be given in the next section.

5.2 ASIP design with Target

In this section, the Target development tools are introduced, and the design environment is briefly described. First, a general introduction to Target Compiler technologies is given in Section 5.2.1, before the Target design environment is presented in Section 5.2.2.

5.2.1 About Target

Target Compiler Technologies is the leading provider of re-targetable software tools for the design, programming, and verification of Application Specific Instruction-set Processors (ASIPs), [abo]. The processor is designed with a processor description language called nML, which is a hierarchical and highly structured architecture description language, that is used to represent ASIP designs at the abstraction level of a programmer's manual [MD08]. Essentially, re-targetable tools behave as if they had been specifically created for the processor defined by the nML processor model, without any customization required. A re-targetable tool simply allows one to immediately determine the improvement in algorithmic performance for any given architectural modification. Therefore, with nML, architectural changes are easy to make, and with re-targetability, the affect is immediately apparent.

In the next section, the design environment is presented, where the focus will be on the tools used in Chapter 6.

5.2.2 ASIP design with Target

The re-targetable tool-suite from target is called Chess/Checkers, and it supports both the design and use of embedded processors in a heterogeneous HW/SW Co-design environment. In this design environment, C is used to model the software application, and nML is used to model the processor. The Chess/Checkers environment consists of six major packages; a C compiler, a linker, an assembler and disassembler, an instruction-set simulator, a hardware description language generator, and a test program generator. These tools are re-targetable, and can be applied to different instruction-set architectures. An outline of these tools is shown in Figure 5.1.

As shown in Figure 5.1, the application in C is compiled using the Chess compiler. This compiler generates machine code in an Elf or Dwarf format. The Bridge linker generates an executable Elf or Dwarf file from multiple C-files. It is also possible to translate assembly code into machine code, or the other way around, using the Darts assembler/disassembler tool. The instruction-set and datapath of the processor is described using the key element in the Chess/Checkers environment, the nML language. This modeling language is described in Section 5.3. Co-simulation of the software and hardware application is possible with the instruction-set simulator (ISS) tool called Checkers. This ISS simulates the execution of machine code on the target processor, using bit- or cycle-accurate simulation. It is also possible to extract profiling information from the Checkers tool, along with information about memory usage, stack usage, etc. When the correct behavior of the processor is verified, the Go tool could be used to generate a synthesizable VHDL and/or Verilog model of the processor. There is two possible ways of HDL generation; using the Primitives Definition and Generation (PDG) language, or interface generation of functional units and code them manually in VHDL or Verilog.

The term "primitives" refer to the primtive data types and operations of the processor architecture, modeled by means of C++ classes, which are declared in the processor header file. For the ISS, an implementation of these primitives in C++ is required, and for the synthesizable HDL model, an implementation in Verilog or VHDL is required.

Both of these implementations can be written by hand, but this requires twice the effort and raises the issue of consistency across these implementations. In addition, some things are missing from C++ to conveniently describe hardware, e.g., operations to do bit and slice selection, and types with a different number of bits than the standard C integer types.

To avoid these problems, the PDG language can be used to describe the primitives once. This language is based on C with a few extensions to address the short-comings described

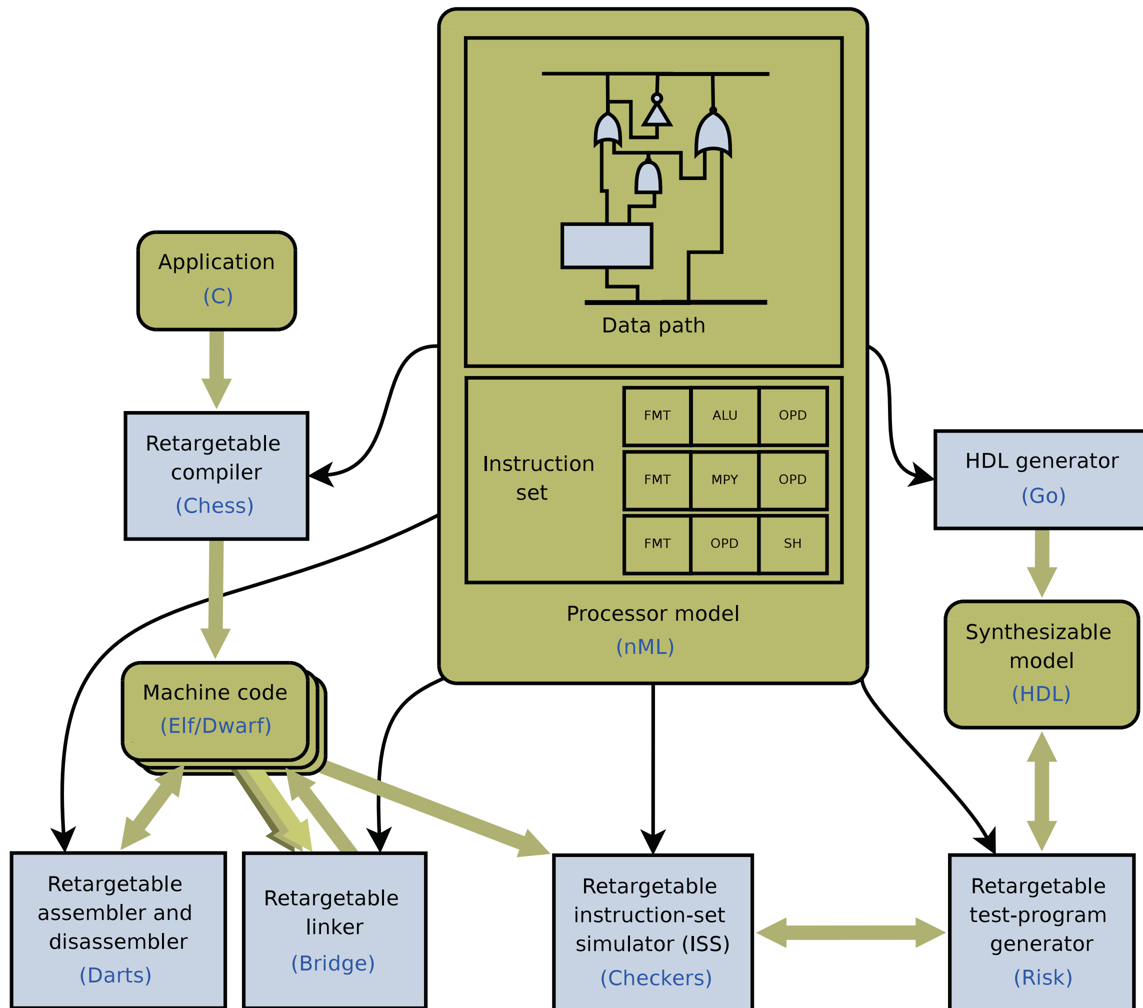

Figure 5.1: An outline of the Chess/Checkers environment.

above. From the PDG description, C++, Verilog and/or VHDL implementations of the primitives can automatically be generated by the PDG tool. The generated C++ file can be used when compiling the ISS. The generated Verilog or VHDL file can be used in the model generated by the Go HDL generator. For more information about the PDG language, the reader is referred to [Tar08].

The Risk test program generator quickly generates a large number of assembly-level test programs. These test-programs can be executed in the ISS and in the HDL model of the processor, in order to check the consistency of both models. According to [Tar03], the results of a number of ASIPs has shown that the timing, power, and area performance obtained from the generated VHDL code is comparable to manual designs. If necessary, the user could go back to the nML description to modify the instruction-set architecture, after having analyzed the performance.

It is possible to do the development of the application and the processor in parallel, but the application development must finish before the processor could be fully optimized for the application. For example, the design of the application is done in parallel with the design of a basic processor. When the application design is complete, it is compiled using the Chess

compiler, and then simulated with the Checkers ISS. By analyzing the profiling information achieved from the Checkers tool, and the assembly code generated by the Darts tool, it is possible to optimize the processor further. This could for instance be done by forcing the application to use special operations, like multiply-accumulate (MAC) operations, or to combine two or more operations to be performed in parallel, thus saving cycle counts.

In the next section, the nML language is briefly introduced.

5.3 The nML language

nML is a hierarchical and highly structured architecture description language (ADL), [MD08]. It models a processor in a concise way for a re-targetable processor design and software development tool suite. As described in [Tar03], the nML language captures the specification of the processor's instruction-set, in addition to the structural information to enable efficient compilation. In this section, the nMl language is described briefly, and more detailed information about the nML language can be found in [MD08].

An nML description is best written using a top down approach. The structural skeleton of the target processor is first specified, where all the storage elements are declared. This is done globally, in the first part of the nML description. This structural skeleton defines the connection points for primitive operations that are defined on the processor. In addition, the processor description contains a grammar that defines the machine language of the processor, and thus models its instruction-set.

After the structural skeleton for the datapath is defined, the instruction-set is analyzed.

The structure is then captured by writing down production rules for the underlying grammar. Every sentence in this language corresponds to an instruction in the instruction-set. The composition of the orthogonal instruction parts are described with so called *AND*-rules, and the alternatives of an instruction are described with *OR*-rules. An example of an AND-rule and an OR-rule is shown in Code Snippet 5.1 and 5.2, respectively.

In Code Snippet 5.1, the rule is called *arith_mem_ind_instr*, where the two instructions *ar* and *mi* are required. *ar* is an arithmetic instruction (called *arith_instr*), and *mi* is a memory indirect load/store instruction (called *mem_ind_instr*). The AND-rule is denoted by the comma which separates the two instructions *ar* and *mi*.

```
1 opn arith_mem_ind_instr(ar: arith_instr, mi : mem_ind_instr)
```

Code Snippet 5.1: AND-rule

Similarly, in Code Snippet 5.2, the direct load/store instruction called *mem_dir_instr* could either perform a direct load or a direct store operation. This is denoted by the "|" sign.

```
1 opn mem_dir_instr(load_direct | store_direct)
```

Code Snippet 5.2: OR-rule

If an AND-rule instruction is to be defined, three attributes are needed; action, syntax and image attributes. OR-rules do not use action or syntax attributes, but could have an image attribute for the different inputs, although it is not needed. Usually, OR-rules are used for a class of instructions related to each other, i.e., to separate between load and store instructions, while AND-rules are used to define the composition of, for instance, a load instruction. An example of a basic arithmetic-logic unit ALU instruction using the three attributes is shown in Code Snippet 5.3.

```
1  enum alu{add "+", sub "-", and "&", or "|"};
2
3  opn alu_instr (op : alu, al : alu_left_opd, ar : alu_right_opd)
4  {
5      action {
6        stage EX1:
7        A = al.value;
8        B = ar.value;
9        switch (op) {
10          case add : C = add(A, B, AS) @alu;
11          case sub : C = sub(A, B, AS) @alu;
12          case and : C = A & B @alu;
13          case or : C  = A | B @alu;
14       }
15       S = C @sh;
16       AR = S;
17    }
18    syntax : AR " = " al op ar;
19    image : op::ar::al;
20 }
```

Code Snippet 5.3: A basic ALU instruction

For the action attribute, a switch statement is used to select the primitive operation to be executed. The selector of the switch statement is declared by the "*op* : *alu*" parameter which refers to the enumeration type alu. This enumerate type generates a set of named binary constants. For *alu*, the *add*, *sub*, *and*, and *or* constants get the values 00, 01, 10, and 11, respectively. Each entry in a switch statement contains a case label, and one or more actions, which may be of any kind. That means they could refer to primitive operations, or references to actions of other rules, or even other switch statements. The @*alu* denotes that the instructions must be executed at the *alu* functional unit in the processor. *stage* $EX1$: indicates at which pipeline stage (in this case $EX1$) all operations are supposed to be executed. *al* and *ar* are mode rules, which are special types of rules. These types of rules are described in [MD08]. Basically, if special registers must be used as inputs, or a special selection of values are needed, mode rules could be created in order to meet such requirements.

The syntax attribute shows the assembly syntax of the instruction parts. Everything between the double quotes must literally appear in the assembly language statement for the corresponding rule.

The image attribute specifies the binary encoding of the instruction parts. All instantiation names in the parameter list occur in the instructions image attribute. The parameters are separated by "::", which represent a concatenation.

In the next chapter, the ASIP design for the fixed point C application, as described in Chapter 4, is presented.

Chapter 6

Application mapping and ASIP design

In this chapter, the ASIP design of the application described in Chapter 4 is presented. The processor is designed by modifying an existing 16-bit processor core with a basic architecture, from Target. The embedded environment around the processor is described in Section 6.1, before the basic core architecture is described in Section 6.2. Section 6.3 then presents the implementation strategy used in order to optimize the processor for the application, and the validation strategy of the processor is presented in Section 6.4. All optimizations are presented in Section 6.5, and the profiling results are presented in Section 6.6. VHDL generation is discussed in Section 6.7. The place and route process is described in Section 6.8 before the extraction of the power numbers is described in Section 6.9. The results for the processor after the VHDL generation and place and route are presented in Section 6.10. All results are discussed and some conclusions are made in Section 6.11.

6.1 The embedded environment of the processor

The embedded environment of the processor basically consist of an analogue to digital converter (ADC), a buffering system, and the processor. In the hardware development, it is assumed that the embedded environment will continuously receive input ECG samples from electrodes attached to a patient. The ECG will arrive at the sample frequency f_s, as illustrated in Figure 6.1.

Based on the overlap described in Chapter 2.4, the embedded system must pre-buffer 2 seconds of ECG samples before they are sent to the processor for processing. The design of this pre-buffering system is out of the scope of this thesis, but there exist several ways of implementing a buffering system, like ping pong buffering [JM98], and static buffering [LM87]. As soon as the buffer is filled up with 2 seconds of sampled ECG data, the samples are sent directly to the processor along with updated threshold values, calculated from the previous round. The next set of input data comes 2 seconds later. This gives the processor a processing deadline of 2 seconds. Therefore, the application has to be fast enough to finish in time, and send all detected beats to the receiver system before the next set of input data arrives.

The new threshold values must arrive together with every new set of ECG samples. Since the processor must finish before the deadline, the new threshold values have to be stored, and fed back as inputs along with the next set of ECG samples. The new thresholds should therefore be stored within the buffering system in order to be able to shut down the processor while it is idle.

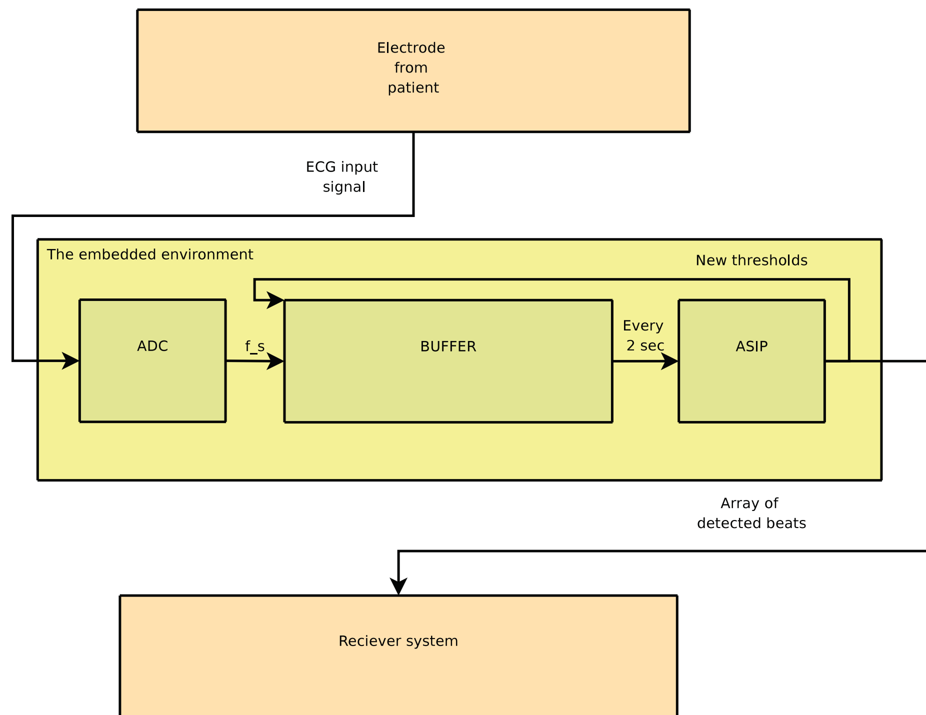

Figure 6.1: Processing interval of the processor in a real-time environment.

6.2 Platform architecture of the BASE core

Target provides a basic 16-bit processor core architecture, intended to illustrate various concepts related to processor modeling, as well as tool capabilities such as on chip debugging and processor verification [Tar09]. This processor core, called BASE, is in the scope of this thesis used as a starting point for the development of an ASIP.

The BASE processor contains a collection of general purpose instructions, in addition to other features such as:

- 16 bit integer arithmetic, bitwise logical and compare instructions
- Integer multiplications with 16 bit operands and 32 bit results
- 16 bit shift instructions
- Load and store instructions to and from a 16 bit data memory with an address space of 64K words (indirect addressing)
- 16 bit program memory
- An 8 field, 16 bit wide general purpose register file.
- Various control instructions such as jumps and subroutine call and return
- Zero overhead loops
- Support for interrupts
- Support for on-chip debugging

- Possibility for application specific custom instructions

Figure 6.2 basically shows the architecture of the BASE processor. The rest of this section will briefly describe the BASE architecture.

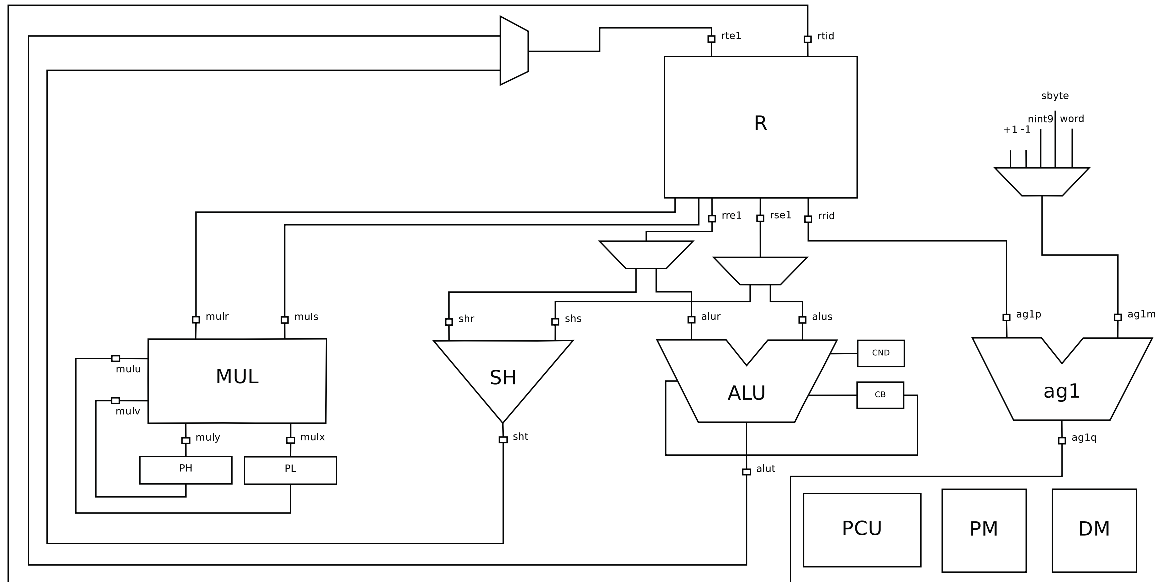

Figure 6.2: The BASE processor core architecture.

Memories and registers

The BASE core has two memories, one data memory (DM) and one program memory (PM). Both memories have a 16 bit word length, and a size of 64K words. In addition, it has a 16-bit wide general purpose register file with 8 fields. The register file accesses its units through ports, and it has one port for each pipeline stage, except from the instruction fetch stage. Other registers exist for the program counter (PC) and stack pointer (SP). In addition, there are loop control and loop flag registers, link and interrupt link registers, status and interrupt status registers, and product registers for the multiplier unit.

Instruction pipeline

The BASE processor has a 16-bit wide instruction word, and it is capable of executing single and double word instructions. in the case of a double instruction word, the second word contains a 16-bit immediate value, while the first word contains the encoding bits.

In order to meet the timing requirements for a given clock frequency, the processor executes instructions using a pipeline with 3 stages; IF, ID and E1. In stage IF, the instruction code is being fetched from the program memory, and in stage ID, the instruction is being decoded. Most operations execute in stage E1. Figure 6.3 depicts the operation of the pipeline. The time progresses from left to right, and the program instructions progresses from top to bottom. In cycle 4, instruction A is executing the actions of its second stage (if any), while instruction D is being fetched from the program memory.

	cycle	1	2	3	4	5	6
	next PC	1	2	3	4	5	6
PC	Instruction						
0	A	IF	ID	E1			
1	B		IF	ID	E1		
2	C			IF	ID	E1	
3	D				IF	ID	E1

Figure 6.3: Instruction pipeline of the BASE processor.

Arithmetic logic unit (ALU)

The instructions of the arithmetic logic unit (ALU) are intended to execute ANSI C arithmetic, logical, and relational operations. This is a 16-bit ALU, and all operations execute in pipeline stage E1.

Shift unit

The 16 bit shift unit executes its operations in pipeline stage E1, and the instructions are intended for ANSI C shift operations.

Multiply unit

The multiplier performs 16 by 16 bit multiply operations and stores the 32 bit result in the multiply registers (PH and PL). It operates directly on the general purpose register file through ports *mulr* and *muls*, as shown in Figure 6.2. This multiplier performs both signed and unsigned multiplications, and it has a multiply-accumulate (MAC) unit.

Load and store unit

The load and store instructions operate on the data memory (DM). The load and store unit contains one address generator unit ($ag1$), and indirect addresses are taken from the general purpose register file. The address generation unit can increment or decrement the address using post modification. As shown in Figure 6.2, the *nint*9 type is a 9-bit type with negative range (-1 to -256). This type is used for stack pointer (SP) indexed load and store operations.

In case a SP addition with an immediate value is needed, the *word* or *sbyte* type are used. *sbyte* is a 8-bit signed type, and *word* is a 16-bit signed type. These inputs to the address generator ($ag1$) unit are handled by the multiplexer shown in Figure 6.2. This address generator is used to communicate with the memories.

Control instructions

The control instructions are performed by the program control unit (PCU). They include jumps, subroutine call and return, hardware loops and interrupts. The PCU unit is also capable of executing multi-cycle instructions, multi word instructions and delay slot instructions. The purpose of this unit is to advance the flow of the program.

Hazards

The base controller solves all hazards (artifacts) of the pipeline execution of instructions, by organizing the software in such a way that the hazard does not occur. This is done through software stalling. For more information about this, the reader is referred to [Tar09].

6.3 Implementation strategy

Before the application from Chapter 4 can be mapped onto the platform architecture, the BASE processor has to be modified.

First, the processor has to be re-targeted to the application by using the Chess compiler from Target. Before the processor is re-targeted for this application, it has to be modified in such a way that the application can execute and produce the correct results. In addition, the main function of the application has to be modified to exclude the read and write operations to file, since these operations are not supported by the BASE processor.

The second step is to simulate the processor with the instruction-set simulator (ISS), and optimize the execution of the application through an iterative process. By reading the assembly code produced by the Chess compiler, and by exploiting the profiling information produced by the ISS, it is possible to identify the instructions in the critical loop, and optimize them by using some of the techniques described in Chapter 3.2. The main purpose of this optimization is to reduce the cycle count of the application.

When the critical loops of the application have been optimized, then VHDL files of the optimized processor are produced by using the Go tool. The memories must be added to the generated VHDL files, because the Go tool generates interfaces for the memories in the design. In this project, 90 nm TSMC low power memories from Virage Logic Corporation are used. The processor and memories have to be interfaced properly to each other through a top module, before simulating the design through RTL simulation. The *ncsim* tool from *Synopsis* is used for this simulation.

In the fourth step, the design is synthesized, and a place and route is performed on a chip using a 90 nm TSMC process. For the place and route, a tool called *Encounter* from *Cadence* is used. After a successful place and route, some information about the load resistance and load capacitance are extracted from the layout of the design, and a netlist simulation is performed. During the netlist simulation, several value change dump (VCD) files are created in order to extract information about the power dissipation (power numbers). A VCD file is an ASCII file which contains header information, variable definition and the value changes for specific values, or all variables, in a given design.

In the final step, the power numbers are extracted by using a development tool called *Primetime* from *Synopsis*. The VCD files created in the previous step are analyzed through this tool, where information related to the power consumption are reported in a text file.

Figure 6.4 illustrate the hardware development flow based on the implementation strategy just discussed. The validation strategy is presented in the next section, before the work done in each implementation step in Figure 6.4 are described.

6.4 Validation strategy

Two types of simulation environments are used; self-checking simulation and simulation through text files. These environments are described in the following subsections.

6.4.1 Self-checking simulation

In order to simulate the application fast and efficiently, the main function is modified to be self-checking. By having a self-checking application, it is possible to run RTL and netlist simulations, and validate the result of the simulation by reading the result registers. The same input ECG signal as the one used for profiling in Chapter 4.7.2 is used, with an ECG sample frequency of 1000 Hz.

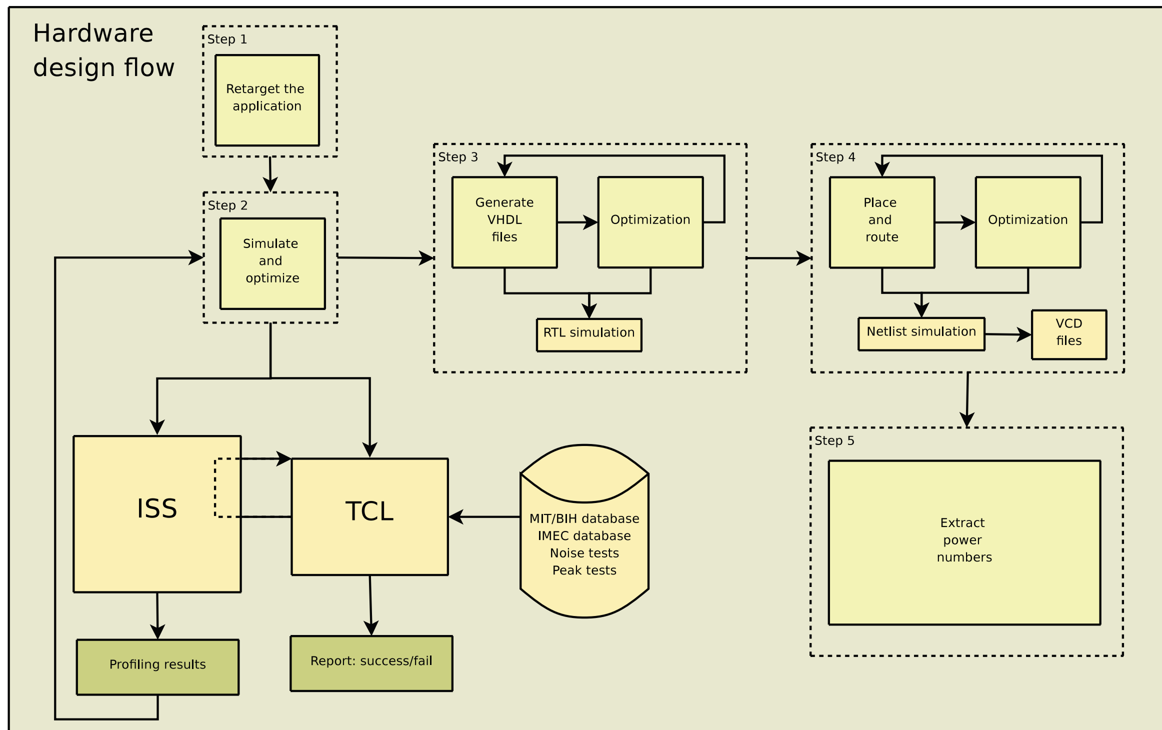

Figure 6.4: ASIP implementation strategy.

The ECG input sample array is defined globally, along with a parameter array, containing information like number of ECG samples, threshold parameters, etc. The answer arrays containing the detected beats and the new threshold values are stored separately, both defined globally. By defining the arrays globally, it is possible to read the allocated memory address in the data memory from within the ISS, and confirm the correct results.

6.4.2 Simulation interface with TCL

In order to simulate the processor with all other signals in the test databases used in Chapter 4, an interface through text files is used. A scripting language called Tool Command Language (TCL) is used to interface the processor simulation in the ISS by taking input values from a text file, and write the results to file. TCL is a textual language intended for issuing commands to interactive programs such as text editors, debuggers, and shells. It has a simple syntax and it is also programmable.

In this TCL interface, 4 files are read, one for the input ECG samples, one for the parameters, one for the expected beats, and one for the expected new thresholds. In a similar manner as the simulation interface for the C-files in Matlab (as described in Chapter 4.4), this script first initializes the ISS, loads both PM and DM, before the input parameters and arrays are updated by the TCL script. The application then executes normally through the ISS, and stores the results to their allocated registers. These registers are then read by the TCL script, and compared to the expected results. A text file is then produced, indicating if the test signal failed or succeeded. All input files, containing the input data and the expected results, are extracted by the Matlab databases. The TCL environment is also illustrated in Figure 6.4.

6.5 Processor core modifications and optimizations

This section presents the modifications and optimizations done to the processor architecture. The same test scenario as the one described in Section 6.4.1 and Chapter 4.7.2 has been used during the optimization process.

In order to handle ECG signals sampled at 1000 Hz, 512 KB data memory is required if 32-bit integers are used, or 256 KB if 16-bit integers are used. Therefore, the 128 KB data memory had to be extended.

Initially, the memory extension was done using two 16-bit wide data memories with a size of 128K words. This 16-bit BASE processor was then simulated using the self-checking test described in Section 6.4. However, this simulation failed because the results were wrong. An analysis of the application showed that shifting the coefficients 14-bits was too much for some input values. This required the integers to be defined as *long* instead of *int*. Since these coefficients were calculated in the critical loop of the application, the change to *long* variables would result in a longer execution time because the processor must execute additional instructions to handle 32-bit words. Therefore, a 32-bit processor would be faster and more energy friendly, since it would in this case execute less instructions. In Chapter 4.6 it was assumed that the number of bits reserved in the calculation within the critical loop was sufficient to handle overflow, based on the use of 16-bit integers. The 16-bit processor proved that more bits are needed for overflow. The software simulation did not indicate this issue because the standard GCC compiler assigned 32-bit integers for the software.

Alternatively, one could have shifted the coefficients less than 14 bit, but this would result in inaccurate values. Therefore, the decision was to convert the 16-bit processor to 32-bit.

The modification of the processor required an extension of the addressing mode, from 16-bit addressing to 32-bit addressing. The word data type of the processor is also extended, from 16-bit to 32-bit. This change required a change in all the instructions in the instruction-set that were related to immediate values. All 16-bit immediate words became 32-bits words, and appropriate bits needed to be assigned to each instruction that processed an immediate word value. In addition, the immediate bytes were now changed from 8-bit to 16-bit, and they had to be defined separately.

The program memory is still 16-bit, but the PCU is modified to handle triple instruction words (for 32-bit immediate values). The resulting instruction-set is at most 48 bit wide, and Figure 6.5 illustrates the 32-bit processors instruction-set, as seen from the nML viewer in the Target IP designer tools. In this figure, all instructions consist of (one or more) 16-bit words. They are all encoded in the first instruction word.

After the conversion, the resulting 32-bit processor was simulated using the simulation environments described in Section 6.4. The results are presented in Section 6.6. The processor was then ready for optimizations, and all optimizations done are described in the rest of this section.

Optimization with MAC operations

The existing multiply-accumulate unit (MAC) did not perform the desired MAC operation. The MAC unit multiplied two unsigned numbers and shifted the result of the multiplication 16 bit to the left, before it accumulated a 32-bit value formed by two input registers. This MAC was intended for other purposes than the one needed in the case of this application, and therefore it had to be removed. By studying the assembly code for the main critical loop of the application (the convolution step within the continuous wavelet transform), shown in Figure 6.6, one could see that a signed multiplication was performed, before the multiplication result was accumulated with the *temp* variable. Figure 6.6 is taken from the ISS window.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47

Group / Encoding			
base			
alu_instr			
000	alu_rrr		
000	compare_rr		
000	equal_rr		
000	alu_rr		
000	minmax_rrr		
001	select_rrr		
shift_instr			
000	shift_rrr		
mult_instr			
000	mul_rr		
move_instr			
001	00101	mv_wreg	
001	001000001	mvi_wreg_word	
001	1	mvi_wreg_byte	
001	00111	mvi_im	
ei			
001	0011000100000		
di			
001	0011000100001		
001	001100000	mv_dbg_wreg	
001	001100001	mv_wreg_dbg	
load_store_instr			
011	load_store_wreg_sp_indexed		
010	000	load_store_wreg_indirect	
000	add_sp_word		
010	1	add_sp_byte	0000
control_instr			
001	0110	0	jrd
001	0110	1	jr
001	0111	1	jcr
001	0111	010000	000 j
001	0111	010010	000 jc
001	0111	010011	000 cl
001	0111	010100	clid
rt			
001	0111	010111	000
rtd			
001	0111	011000	000
001	0111	011011	do
001	0111	011100	000 doi
001	0111	00111	swi
rti			
001	0111	011001	000
halt			
001	0111	011010	000
nop			
001	0111	000000	000

Figure 6.5: Initial instruction-set of the 32-bit processor.

The highlighted part of Figure 6.6 shows all instructions involved to execute the corresponding C code. All 6 assembly instructions were executed 3.6 million times each according to the profiling information generated by the ISS (in addition to the move operation between

the highlighted lines). Since all assembly instructions for this C-code are executed the same amount of time, it is easy to see that by reducing these instructions it would result in a shorter execution time for the application.

Therefore, the MAC unit is modified to multiply two signed integers, and accumulate without shifting the result of the multiplication. This was done in the primitives definition and generation (PDG) language of the processor. Code Snippet 6.1 shows the PDG code for the MAC unit before the modification, and in Code Snippet 6.2, the resulting MAC unit is shown. The processor was then simulated again in the ISS, and the resulting assembly code is shown in Figure 6.7. This modification alone saved approximately 14.4 million cycles of execution.

```
   469 2536          // mv r3,r6
>> 470 4160          // ld r0,dm(r5--)
 > 471 4057          // ld r7,dm(r1++)
 > 472 1807          // mulss r0,r7
   473 2508          // mv r0,pl
 > 474 00d8          // add r3,r3,r0
 > 475 1608          // xs r0,r0
 > 476 03b0          // addc r6,r6,r0
   477 6ff0          // ld r0,dm(sp-1)
```

```
 184        */
>185        for(m = m_min; m <= m_max; m++){
 186            //temp = temp + _x->array[n-m] * _h.array[m-1];
>187            temp = temp + _x.array[n-m] * _h.array[m-1];
 188        }
 189
 190    // Divide result with the root of s
 191        /*
 192         * FIXED POINT notice:
```

Figure 6.6: Initial simulation without the MAC unit.

```
1 void macl(word a, word b, word cl, word ch, word& rl, word& rh)
2 {
3     int64_t p = (uint32_t)a * (uint32_t)b;
4     int64_t q = (p << 16) + (ch::cl);
5     rl = q[31:0];
6     rh = q[63:32];
7 }
```

Code Snippet 6.1: 32-bit MAC unit

```
1 void macl(word a, word b, word cl, word ch, word& rl, word& rh)
2 {
3     int64_t p = a * b;
4     int64_t q = p + (ch::cl);
5     rl = q[31:0];
6     rh = q[63:32];
7 }
```

Code Snippet 6.2: 32-bit MAC unit after modification.

```
   438 3008          // mvib pl,0
>> 439 4025          // ld r5,dm(r0--)
 > 440 4056          // ld r6,dm(r1++)
 > 441 18ee          // macl r5,r6
   442 2558          // mv r5,pl
   443 2569          // mv r6,ph
>> 444 30e1          // mvib r1,14
 > 445 3022          // mvib r2,2
 > 446 1269          // lsr r1,r5,r1
```

```
 181       *     mmax - mmin + 1 is maximum 1-_h_size + 1 = 1-1
 182       *     is taken, and the ceiling of log2(|1078|), the
 183       *     that a 38 bit register is needed to handle the
 184       */
>185       for(m = m_min; m <= m_max; m++){
>186           temp = temp + _x.array[n-m] * _h.array[m-1];
 187 //        temp = macl(temp,_x.array[n-m],_h.array[m-1]);
 188
 189       }
```

Figure 6.7: Simulation with the resulting MAC unit.

Parallel load and store operations

Based on Figure 6.7, the load operations related to the MAC operation are combined by a customized instruction where they are performed in parallel. This optimization resulted in saving additional 3.6 million execution cycles. However, since it is only possible to load

and store from the same memory once per pipeline stage, the DM memory is split in two identical data memories, DM and CM. CM stands for coefficient memory, and half of the data elements are assigned to CM. The assignment of variables to a specific memory is done by specifying this in the C code directly, as shown in Code Snippet 6.3.

```
1 intlist chess_storage(CM) psi_scale;
2 intlist ecgsig3;
```

Code Snippet 6.3: Direct assignment to CM.

The *ecgsig*3 array is assigned to the standard data memory (which is assigned to DM by default), and the *psi_scale* array is assigned to the CM memory. Both those arrays are used in the MAC operation, and could now be performed in parallel.

In order to access two memories in parallel, another address generator, *ag*2, is created such that the load and store operations from the DM and CM memories could be performed within the same pipeline stage. This is done through separate ports to the register file, called *rr*1 and *rr*2, as shown in Figure 6.12.

Based on the load and store instructions for the data memory, the indirect load and store instruction is copied and modified to operate on CM. Since the CM does not have any stack, only the indirect instruction is needed. The nML code for the parallel load instruction is shown in Code Snippet 6.4. As shown, the address generator fetches the addresses in the ID stage of the pipeline for both DM and CM, and the load is performed in parallel in stage E1. The parallel store instruction is done similarly.

```
1 // Define the behavior of the parallel load instr operation:
2 opn par_load_pload(ag1: ag_opn, rr1: r_reg, ag2: ag2_opn, rr2: r_reg2)
3 {
4  action {
5   stage ID:
6    ag1;
7    ag2;
8   stage ID..E1:
9    dm_addr`ID` = ag1p`ID`;
10   rr1`E1` = dm_read`E1` = DM[dm_addr`ID`]`E1`;
11   cm_addr`ID` = ag2p`ID`;
12   rr2`E1` = cm_read`E1` = CM[cm_addr`ID`]`E1`;
13  }
14  syntax : "ld " rr1 ",dm(" ag1 ") | ld " rr2 ",cm(" ag2 ")";
15  image  : ag1::rr1::ag2::rr2;
16 }
```

Code Snippet 6.4: Parallel load instruction.

The processor was simulated after this optimization, resulting in the assembly code shown in Figure 6.8.

```
 > 436 2edd 01b8      // do r5,440
   438 3008           // mvib pl,0
>> 439 842e           // ld r0,dm(r1++) | ld r6,cm(r2--)
 > 440 18f0           // macl r6,r0
   441 2508           // mv r0,pl
   442 2569           // mv r6,ph
>> 443 30e5           // mvib r5,14
 > 444 3021           // mvib r1,2
 > 445 1471           // lsl r1,r6,r1
 > 446 1345           // lsr r5,r0,r5
```

```
 180      *
 181      *     mmax - mmin + 1 is maximum 1-_h_size + 1 = 1-1080+1 = -10
 182      *     is taken, and the ceiling of log2(|1078|), the result is
 183      *     that a 38 bit register is needed to handle the maximum va
 184      */
>185    for(m = m_min; m <= m_max; m++){
>186          temp = temp + _x->array[n-m] * _h.array[m-1];
 187 //       temp = macl(temp,_x.array[n-m],_h.array[m-1]);
 188
 189    }
```

Figure 6.8: The new parallel instruction assembly code, as shown in the ISS.

This new instruction could now be viewed by the nML viewer in the Target IP designer tools, as shown in Figure 6.9. In this figure, the parallel load instruction is called

par_load_pload, and the parallel store instruction is called *par_store_pload*. The third instruction is exactly the same as the indirect load and store instruction for DM, modified to access CM.

Figure 6.9: The new parallel instruction, as shown in the nML viewer.

Combining MAC and load instructions

By studying the assembly code in Figure 6.8, one could see that there is a possibility to load the input registers of the MAC unit directly. However, doing this requires another pipeline stage. The parallel load and MAC instruction are therefore combined by first adding another pipeline stage, E2, and creating a custom instruction that integrates both MAC and parallel load instructions. The MAC instruction is executed in pipeline stage E2, while the parallel load instruction executes in pipeline stage E1. Code Snippet 6.5 presents this new instruction.

Instead of assigning the two multiplicands to the multiplier from registers, additional pipes (delay registers for the pipeline) were created, called $mulr2$ and $muls2$. In this case, a pipe is used to delay a signal one pipeline stage. This is done because the values from the data memory in pipeline stage E1 would in this case be fed directly into the input of the multiplier in pipeline stage E2.

After this modification, the processor simulation resulted in 3.6 million cycles less than the previous round, optimizing this critical loop optimally. As shown in Figure 6.10, the critical loop, performs the MAC operation using only one instruction, which is executed 3.6 million cycles.

In order to eliminate pipeline hazards, the compiler adds one *nop* operation after the end of the inner loop of the convolution step. The convolution step is described in Chapter 4.2.1

```
 > 434 2edd 01b5      // do r5,437
   436 3008           // mvib pl,0
>> 437 c017           // macl(cm(r3--),dm(r1++))
   438 2e00           // nop
   439 2578           // mv r7,pl
   440 2559           // mv r5,ph
```

```
 184          */
>185         for(m = m_min; m <= m_max; m++){
>186                 temp = temp + _x->array[n-m] * _h.array[m-1];
 187 //              temp = macl(temp,_x.array[n-m],_h.array[m-1]);
 188
 189         }
```

Figure 6.10: The new parallel load and MAC instruction.

and Appendix A. This *nop* operation executes only 16 000 cycles, compared to the 3.6 million cycles that was saved by this parallelism.

```
1 opn par_load_macl_pload(ag1: ag_opn, ag2: ag2_opn)
2 {
3  action {
4   stage ID:
5    ag1;
6    ag2;
7    dm_addr = ag1p;
8    cm_addr = ag2p;
9   stage ID..E1:
10   mulr2`E1` = cm_read`E1` = CM[cm_addr`ID`]`E1`;
11   muls2`E1` = dm_read`E1` = DM[dm_addr`ID`]`E1`;
12  stage E2:
13   macl(mulr2,muls2,mulu2=PL,mulv2=PH,mulx2,muly2) @mul;
14   PL = mulx2;
15   PH = muly2;
16 }
17 syntax : "macl(cm(" ag2 "),dm(" ag1 "))";
18 image  : ag1::ag2;
19 }
```

Code Snippet 6.5: Parallel load and MAC instruction.

Combining load, equal, greater than and select instructions

In a similar way as the MAC operation, another critical loop is optimized by combining a load, select and equal instruction to execute in parallel. This optimization resulted in reducing three instructions, executing approximately 200 000 cycles each into one custom instruction, as shown in Figure 6.11. This optimization saved around 400 000 cycles. The nML code for this instruction is given in Code Snippet 6.6.

```
  2074 2e00           // nop
 >2075 4053           // ld r3,dm(r1++)
 >2076 1a1a           // eq r3,r2
>>2077 2edc 0820      // do r4,2080
 >2079 4053           // ld r3,dm(r1++)
>>2080 e238           // ld r3,dm(r1++) | eq r3,r2 | sel r0,1,r0
  2081 01b5           // add r6,r6,r5
 >2082 3003           // mvib r3,0
```

```
  795         found = 0;
> 796         for (i = 1; i <= clustrow; i++){
> 797                 for (j = 1; j <= clustcol; j++){
> 798                         if(clust[i-1][j-1] == s+(k-1)*(posrow)){
  799                                 found = 1;
  800                         }
  801                 }
  802         }
```

Figure 6.11: The new parallel load, equal and select instruction.

This instruction was created by copying the functionality from the equal and select instruction and combining them together with a normal load operation. The greater than instruction was added because it also optimized another part of the application similarly, by a parallel load, greater than, and select operation. As seen in Code Snippet 6.6, there are two ALUs executing in parallel, *alu* and *alu*2. The second ALU is created for the select operation.

If an independent select statement is assigned to *alu*2 by the compiler, then a delay (*nop*) must be inserted. One custom instruction that handles this situation is made, by assigning a delay (the *nop* instruction). This is shown in Code Snippet 6.7.

```
1 opn par_ld_eq_sel(ag1: ag_opn, op: leq_op, r: c2u, s: c2u, t: c2u)
2 {
3  action {
4   // Load:
5   stage ID:
6    ag1;
7   stage ID..E1:
8    dm_addr`ID` = ag1p`ID`;
9    RT[r]`E1` = rte1 = dm_read`E1` = DM[dm_addr`ID`]`E1`;
10  stage E1:
11  // eq or gt:
12   alur = rre1 = RT[r];
13   alus = rse1 = RT[s];
14   switch(op){
15    case eq: CND = eq(alur, alus) @alu;
16    case gt: CND = gts(alur, alus) @alu;
17   }
18  // Select:
19   alur2 = 1;
20   alus2 = rre12 = RT[t];
21   alut2 = select(CND, alur2, alus2) @alu2;
22   RT[t] = rte12 = alut2;
23 }
24 syntax : "ld r" r ",dm(" ag1 ") | " op " r"r ",r"s " | sel r"t ",1,r"t;
25 image  : ag1::op::r::s::t;
26 }
```

Code Snippet 6.6: Parallel load, equal and select instruction.

```
1 opn par_nop_sel(t: c3u)
2 {
3     action {
4         stage E1:
5         // NOP:
6             nop();
7         // Select:
8             alur2 = 1;
9             alus2 = rre12 = R[t];
10            alut2 = select(CND, alur2, alus2) @alu2;
11            R[t] = rte12 = alut2;
12    }
13    syntax : "nop | sel r"t ",1,r"t;
14    image  : t;
15 }
```

Code Snippet 6.7: A *nop* and select instruction.

Architecture modification

All optimizations and modifications resulted in a new BASE architecture, as shown in Figure 6.12. Basically, an address generator and a second ALU are added, in addition to some pipes and some ports. This figure shows Figure 6.2 with the additional hardware marked in red.

6.6 Target profiling results

Based on the profiling reports generated by the Target tools, the reduction of program execution cycles are shown in Figure 6.13, for each optimization step. The same signal was simulated using ECG sample frequencies of 500, 360 and 198, in order to illustrate how the different execution times vary with different input sizes (number of samples). From this

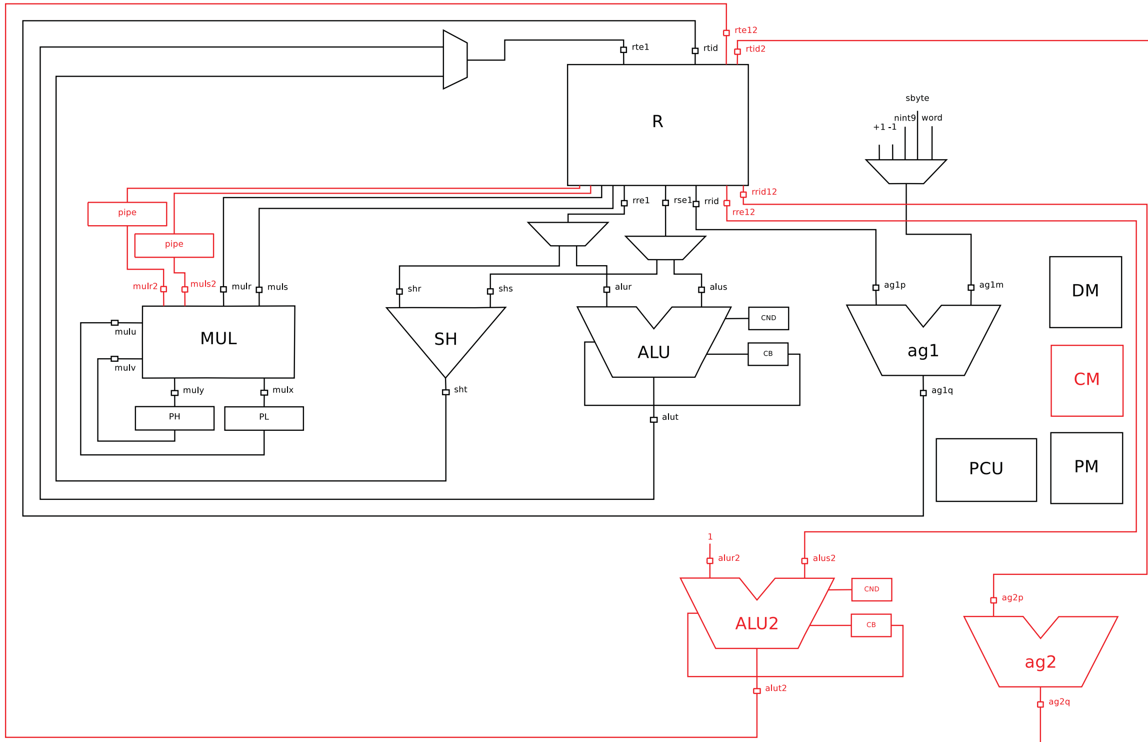

Figure 6.12: The optimized BASE architecture.

figure, it is clear that the execution cycle counts are highly dependent on the size of the input signal. In the figure, the relative ratio is the same for all frequencies.

In Figure 6.13, *unoptproc* is the unoptimized processor, *Cust MAC* is the improvement after the customized MAC operation has been added. Similarly, *Par ld, st* shows the optimization after the parallel load and store have been added. *Par MAC, ld* is after the parallel MAC and load instruction was applied, and *Par load, eq/gt, sel* shows the result after all custom instructions were created. *Ctrl sig in local reg* is after the final optimization, where some control signals were defined locally instead of globally. After all optimizations, the execution cycle count was reduced by 81%.

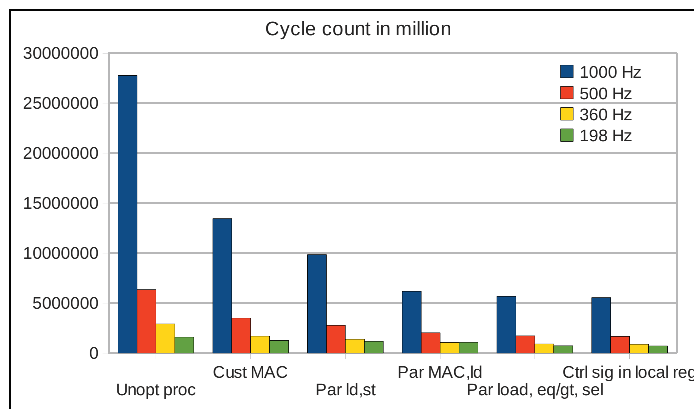

Figure 6.13: Improvement in execution cycles after every optimization step.

6.7 VHDL generation

After the optimizations mentioned in the previous section, the remaining code still had some critical loops. However, these loops did not execute more than 16 000 cycles each, and therefore the optimization process was stopped in order to proceed with the VHDL generation.

The VHDL files of the processor are produced by using the Go tool. In addition, the PDG language made sure all functional units were generated. Although the process was automated, the BASE controller VHDL file needed two modifications before it could function properly. The first modification was the killing of the next instruction when relative conditional jumps are executed. The second modification was to increment the program counter when returning from a subroutine call.

After these modifications, the optimized BASE processor was interfaced with three memory blocks consisting of 90 nm TSMC low power memories from Virage Logic Corporation (VLC). Two of the memories have a data word size of 32 bit, while the last memory has a word size of 16 bit. It was not possible to create one large memory of 256 KB for the data memories, because the largest possible memory available from VLC libraries used in this project, was a high performance ultra low power SRAM, with a memory size of 16K words. This memory could operate in a clock frequency up to 100 MHz. Therefore, 4 of these memory blocks had to be connected together through a memory wrapper. The entity of a VLC memory block is shown in Figure 6.14.

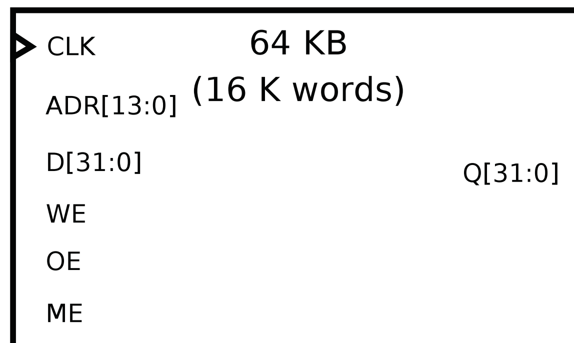

Figure 6.14: A 90 nm TSMC low power memory block from VLC.

Each memory block can be addressed through the ADR 14 bit address input. D is the data input, and WE is the write enable signal. The memory could only be written to when WE is logic high. OE is the output enable signal, and this signal indicates the contents of the memory location given by ADR to Q when it is logic high. When this signal is logic low, the data output, Q is in a high impedance state. ME is the memory enable input. When this signal is logic low, the memory is deactivated. Therefore, it must be active in order to perform a read or write operation to the memory. A similar block is made for the program memory, with a 16 bit word size. Figure 6.15 illustrates how the memories are wrapped together for the data memory DM, in order to achieve the desired memory size. CM and PM are done similarly.

In order to write to a 64K word memory, a 16 bit address is needed. But since each memory block only has 14 bit for the address memory, the 2 MSB of the input address signal is used to determine which memory is enabled when a read or write operation takes place. As seen in Figure 6.15, WEN is used to assign all WE signals. Similarly, D is connected to all data inputs. The 14 first bits of the A input are connected to the ADR input of all memory blocks. However, the CEN is connected to a demultiplexer, which uses the 2 MSBs

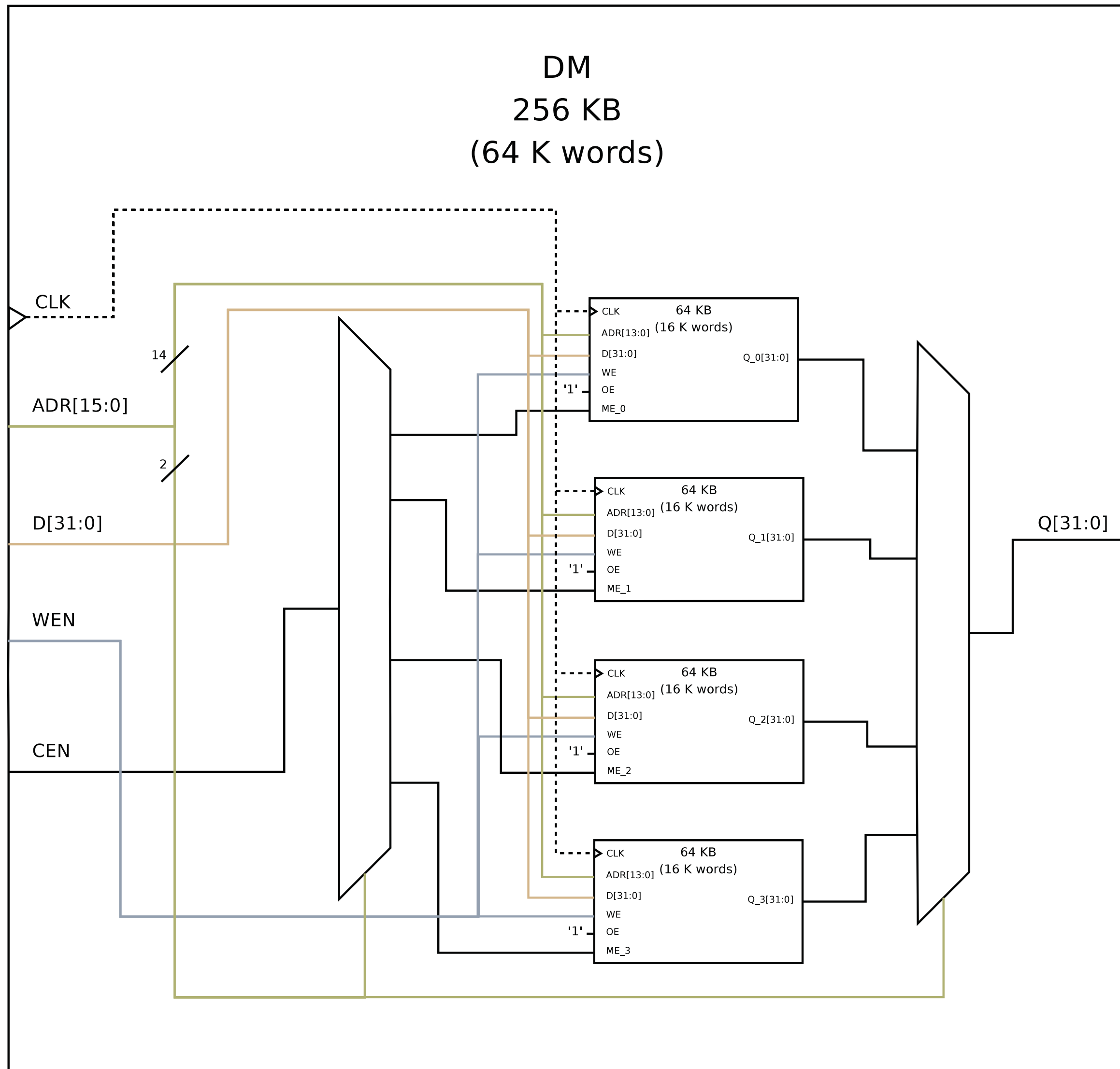

Figure 6.15: The memory wrapper of DM.

of the input address A to activate the desired memory. Each output of this demultiplexer is connected to the ME input of a specific memory. Table 6.1 shows the truth table of how the CEN signal is forwarded to these memories by the 2 MSBs of the address input signal.

Truth table of how the memories are enabled						
CEN	*A[15]*	*A[14]*	*ME_3*	*ME_2*	*ME_1*	*ME_0*
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Table 6.1: Truth table for the memory enable signal of DM.

The output, Q, is assigned by a multiplexer of all output signals from the memory blocks.

The 2 MSBs of the address input determine which output signal is forwarded to Q. Basically, only one internal memory block is enabled at a time.

The top module, connecting all memories with the BASE processor is shown in Figure 6.16. The *clk* signal is the processor clock, *mem_add* is the memory address location, *mem_data_in* is the data to be assigned to a memory selected by the *mem_select* signal. *mem_wen* is the write enable signal for the memories, and *mem_data_out* is the output signal of the memory selected by the *mem_select* signal. The *reset* signal is used to reset the base processor.

All input signals, except from the *reset*, are connected to the memories, in order to be able to program and read them externally. As long as the *reset* signal is logic high, the processor is deactivated, and an external testbench can program the memories with the desired information. After the memories are programmed by the testbench, the *reset* signal is set to logic low. The processor then starts the program counter and executes the program specified by the information in the program memory. The processor will continue executing the program memory in a loop unless the *reset* signal is set to logic high.

All memories have dedicated connections to the BASE processor, and an external testbench can program the memories, and initialize or reset the BASE processor through the inputs of the top module. Similarly, an external testbench can also read the values in a memory through the output signal of the top module.

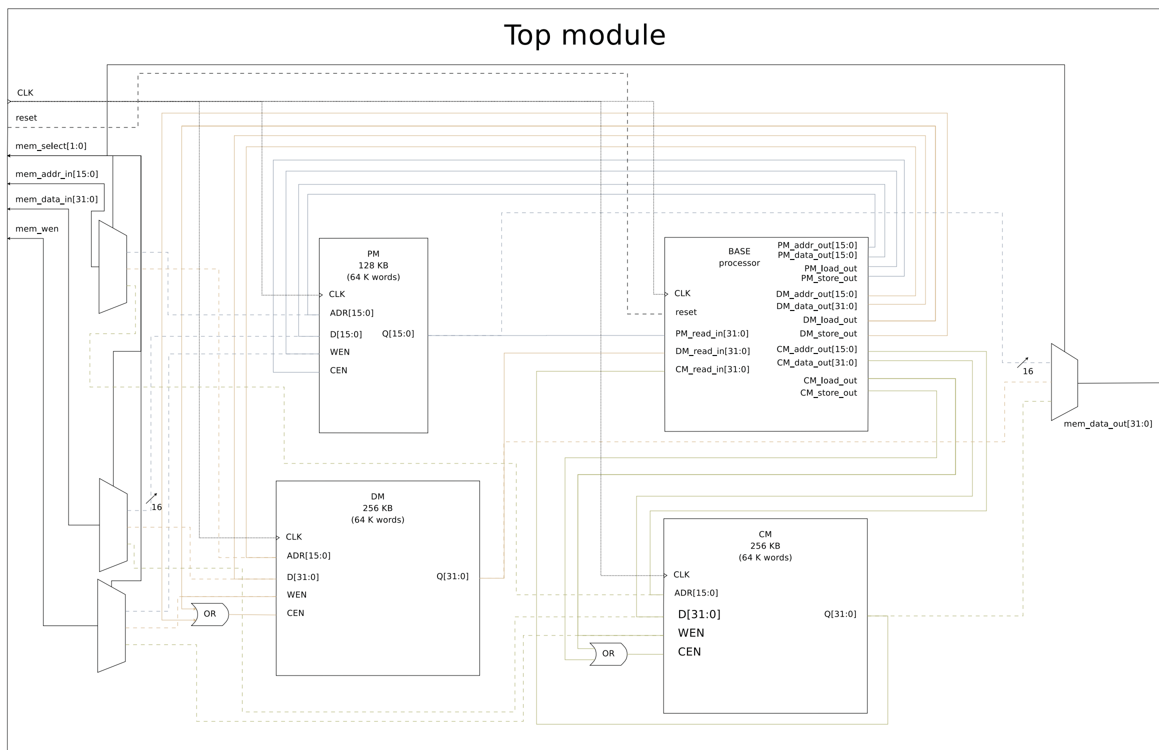

Figure 6.16: The top module, connecting the memories and the BASE processor.

The self-checking test described in Section 6.4 was used to verify the functionality of this processor, and the correct beats were detected by RTL simulation in *ncsim*.

6.8 Place and route

In the place and route step, the design was synthesized for 100 MHz, where the design was optimized for low power consumption. During synthesis, a library of standard cells was specified. In this thesis, standard cells from a low power TSMC library are used.

The memory blocks are placed and routed on a chip as shown in Figure 6.17. This chip is created by a 90 nm TSMC process, and the processor core is placed between the DM and CM. Two of the PM blocks (the smallest blocks) are placed next to the DM memory and the other two blocks were placed next to the CM memory. The processor core is centered between DM and CM because most of the communication is related to those memories.

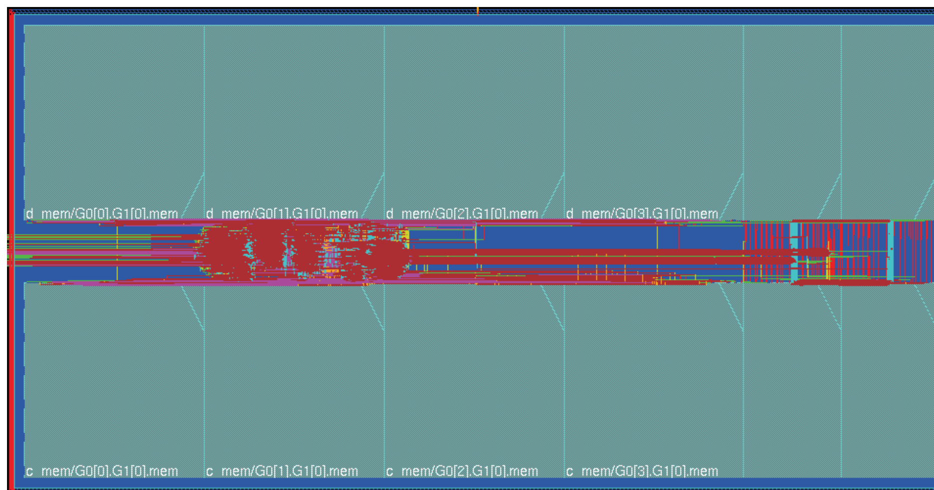

Figure 6.17: How the memory blocks were placed on the chip.

After this successful place and route, the location of the individual standard cells (in the design) were known. The capacitance on the output of each standard cell could then be calculated by adding both the input capacitance of the connected standard cells, and the interconnect (wire) capacitance. The wire capacitance thereby were calculated by using a technology file provided by TSMC. In this technology file, the capacitance for the different metal layers are listed. The capacitance of the complete design is calculated in this way, and the information was written into a so called standard parasitic exchange format (SPEF) file. Also, the resistance of the interconnect (wires) was extracted from the layout of the design.

After the capacitance and resistance of each node was known, and the drive strength of the cells was given, the timing of the design was calculated with *Primetime*. The drive strength refers to how much current the cells can source (flow into the output) or sink (flow out of the output). The timing is calculated by adding the interconnect delay, and the delay from the standard cells (from input to output) in all paths.

The netlist of this chip is generated through *Encounter*, and timing reports were generated. These reports indicated no violating paths, negative fan outs, or other problems. The timing information was used in the netlist simulation in *ncsim*. In addition, VCD files of the signals were created for the whole execution phase of the application in the processor. These dump files were used to extract the power numbers described in the next section. Figure 6.19 in Section 6.10.1 shows the correctly detected beats from the netlist simulation.

6.9 Extraction of power numbers

The final step in the design process was to extract information about the power dissipation of the optimized processor. All steps involved in the extraction of the power dissipation was also done for the unoptimized 32-bit BASE processor (using the same standard cell libraries) in order to compare the results.

These power numbers were extracted by *Primetime* from the VCD files created during the netlist simulation. The power numbers were only extracted during the execution phase of the application. No information about the power dissipation were extracted when the processor was idle, because it is in this thesis assumed that clock gating is applied. The idle power consumption could then be neglected, except from the leakage power. By using Equation 3.16, the average power consumption for the optimized processor was calculated. The duty cycle of each processor was calculated from Equation 3.17.

In addition, a graph containing the values of the circuit delay for a given supply voltage in a 90 nm TSMC process, was studied in order to explore the possibilities of reducing the power consumption even more by some of the techniques presented in Section 3.3. This graph is shown in Figure 6.18, and it was created using a ring oscillator for a different range of supply voltages, beginning from 1.2 V down to 0.6 V. In this thesis, standard voltage libraries are used, with a supply voltage of 1.08 V.

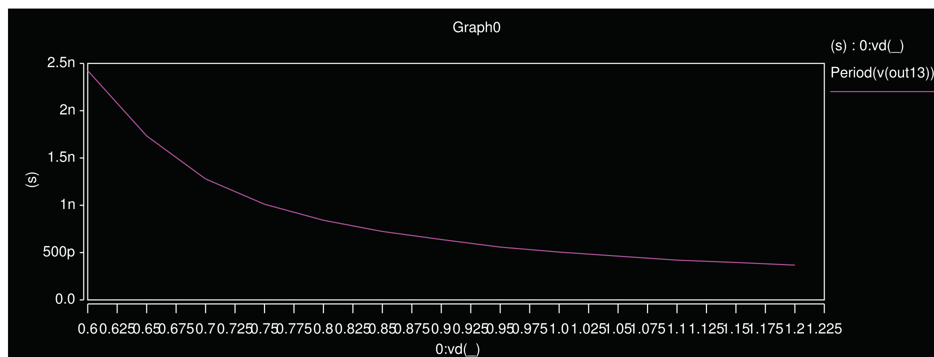

Figure 6.18: How the delay of the circuit varies with different supply voltages in a 90 nm low power TSMC process.

How this information could be used to reduce the power consumption is discussed in Section 6.11, based on the results presented in Section 6.10.

6.10 Simulation and power consumption results

In this section, all simulation results are presented. In addition, the power consumption of the optimized and the unoptimized processor are presented. Both power and simulation results are achieved by using the self-checking test described in Section 6.4.1. Section 6.10.1 shows how the detected beats were identified in the waveform simulation window by reading the memory address of the global result arrays location, for the RTL simulation and the netlist simulation. Finally, Section 6.10.2 presents the power dissipation of both the optimized and the unoptimized BASE processor.

6.10.1 Register transfer level (RTL) and netlist simulation results

The expected beats to be detected were pre-calculated, and stored in a lookup table in the data memory DM. Similarly, a lookup table was made for the results of the application. All

values in this lookup table were set to zero, and the detected beats are then copied to that lookup table in the main function of the program. Similarly, the expected new threshold values and the actual results were compared.

The netlist results were confirmed in a similar way as the RTL simulation, by observing the results on the Q-port of the data memory, at the addresses allocated for the result lookup tables. Figure 6.19 shows the waveform of the netlist simulation where the right beats were detected correctly.

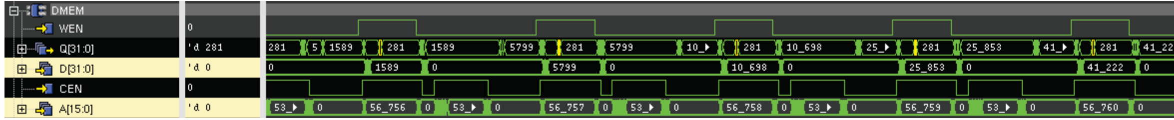

Figure 6.19: Results from the netlist simulation with *ncsim*.

6.10.2 Power numbers

From the equations in Section 6.9, the duty cycle of the optimized processor is found to be 2.8%. Similarly, the duty cycle of the unoptimized processor is found to be 14%. For the optimized processor, the total average power consumed within the base processor and the memories DM, CM and PM, were extracted from *Primetime*. The same was done for the unoptimized processor.

As expected, the unoptimized processor consumed less average dynamic power while it was running than the optimized processor. The total dynamic power of the unoptimized processor consumed is much greater than the optimized processor within the 2-second interval. This is illustrated in Figure 6.24. Figure 6.20 illustrates the portion of the total power consumption each module in the optimized processor (on the left) and the unoptimized processor (on the right). Figure 6.21 and 6.22 illustrates the same for the leakage power and dynamic power, respectively. Figure 6.23 and 6.24 display the total average power consumption of the processors compared to each other.

The total average power dissipation of the optimized processor is reduced by 55%. Similarly, the total average dynamic power dissipation is reduced by 78%. The total average leakage power on the other hand has increased by 13%. It is worth noting that the difference in the leakage power consumed in the optimized processor is not shown in Figure 6.24.

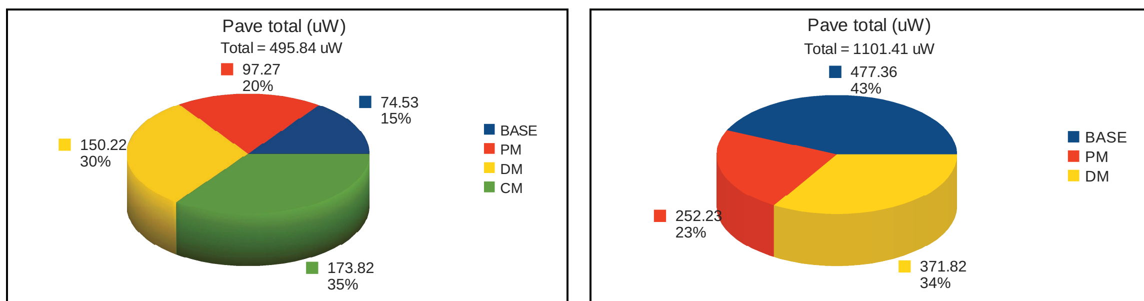

Figure 6.20: Total average power consumption of the processors. On the left are the results for the optimized processor, and the unoptimized processor results are displayed on the right.

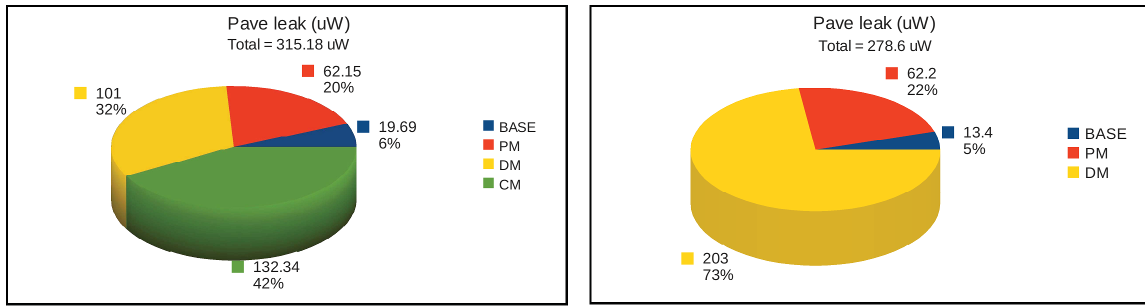

Figure 6.21: Average leakage power consumption of the processors. On the left are the results for the optimized processor, and the unoptimized processor results are displayed on the right.

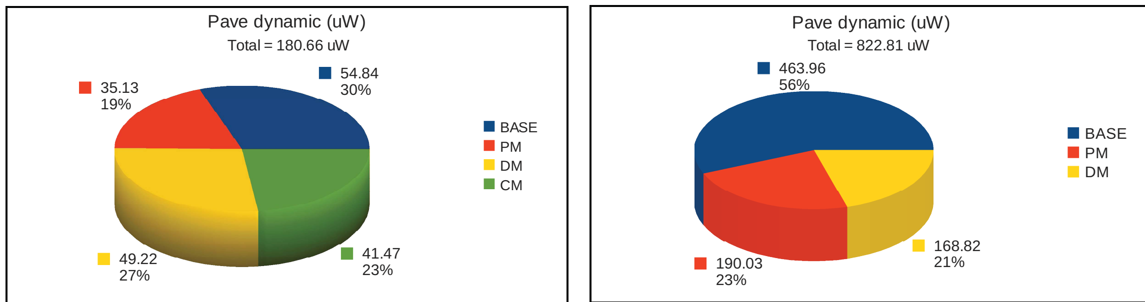

Figure 6.22: Average dynamic power consumption of the processors. On the left are the results for the optimized processor, and the unoptimized processor results are displayed on the right.

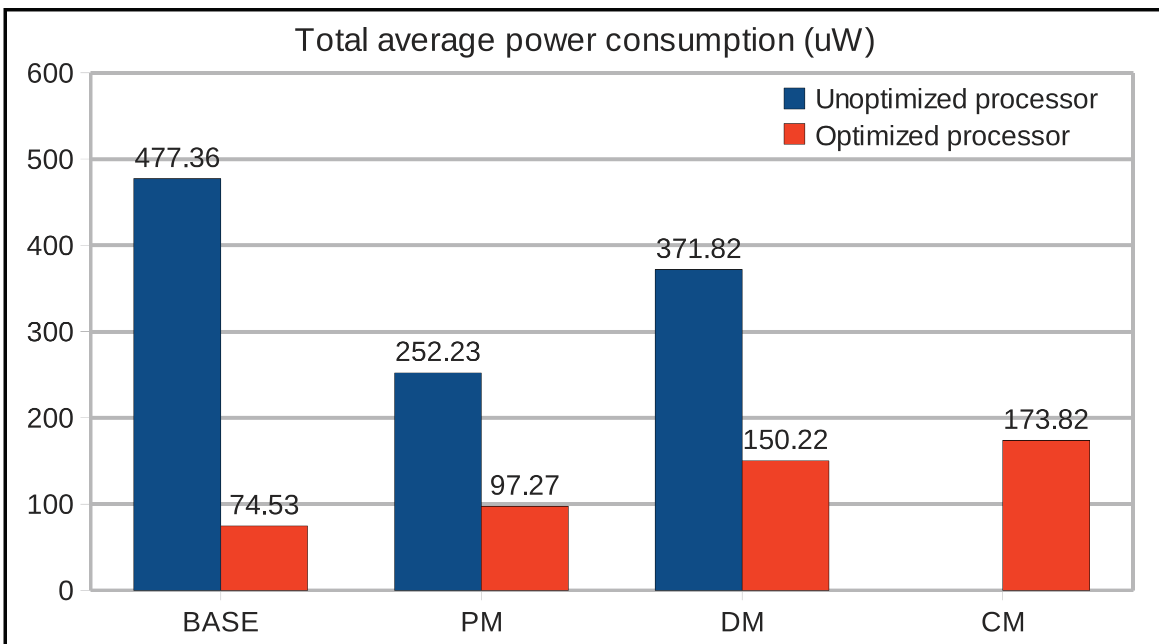

Figure 6.23: Total average power consumption of the processors compared together.

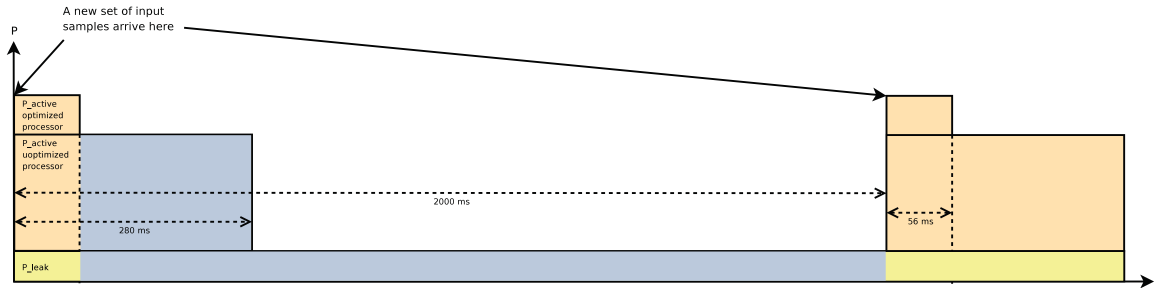

Figure 6.24: Total average power consumption of the processors.

6.11 Discussion of the results and conclusions

In Figure 6.13, it is shown that the amount of cycles is reduced in every optimization step. What is also observed from this figure, is that the processing time is highly dependent on the input ECG sample frequency, f_s. At a sample frequency equal to 198 Hz, the execution time of the processor is at a minimum.

The total average power consumption shown in Figure 6.20 illustrates clearly that the memories have a great impact on the power consumption. Furthermore, the percentage of the power consumed by the optimized processor is improved by 55%. The dynamic power shown in Figure 6.21 is reduced by 78% in the optimized processor. The leakage power has increased because of the additional application specific hardware, and the additional memory, CM.

Based on the results shown in Figure 6.23, the conclusion is that the optimized processor is more energy efficient than the general purpose processor. However, there are more optimizations that could be done in order to make the processor more energy efficient. There is no doubt that a change in the application that would result in less memory usage, would automatically reduce the memory sizes, and hence the power consumption.

In the place and route step, the ASIP was assigned a 128 KB program memory. According to the usage of instruction words, not more than 8 KB is needed to execute the application. However, ASIPs are meant to be used for several other applications, and a bigger size of the program memory is therefore preferred. In any case, more power could be saved by reducing the size of the program memory, because the processor reads from it during the whole execution time.

Since the BASE core supported several features, as described in Section 6.2, not all of them are used for the target application. Switching them off or removing the modules which are not used would result in optimizing the architecture of the processor.

In the case of the unoptimized processor, the execution time resulted in approximately 27 million cycles, while the optimized processor resulted in approximately 5 million cycles. This is a reduction of 81%. For both processors to finish processing before the deadline of 2 seconds, the unoptimized processor can operate in the frequency range 14 MHz to 100 MHz, when the ECG sample frequency is 1000 Hz. Timing issues could easily occur if the processor is synthesized for higher frequencies, because the memories applied operate at maximum 100 MHz. The optimized processor clock frequency range is on the other hand between 3 Mhz and 100 MHz.

The penalty of larger area is small, because the improvements were gained by adding a few application specific units. The same chip area shown in Figure 6.17 was used for the unoptimized processor.

Assuming that the application is optimal, and that the cycle count has been optimized fully, it would be possible to make the processor even more power efficient by applying voltage scaling and power gating, as described in Chapter 3.3. From Figure 6.18, the voltage supply of a transistor in a 90 nm TSMC process could be scaled down to 0.6 V. For lower voltages, the delay is to high. However, the corresponding delay would result in a 6 times slower circuit. In the case of a clock frequency of 100 Mhz, the operating clock frequency would have to be scaled down to approximately 17 Mhz, without violating the deadline of the processing time. The processor would in this case consume dynamic power for a longer time period. If the leakage power is dominant, then the total average power consumption might increase.

The supply voltage in the voltage libraries used in this project is 1.08 V. By using Equation 3.1, the dynamic power is estimated to be reduced with a ratio of 19 times (or 94.7% improvement) by applying voltage and frequency scaling.

Alternatively, by applying power gating, one could shut down the processor after it has finished processing, and wake it up again just before the next set of samples are about to arrive. By doing so, the leakage power could be reduced significantly. The leakage power consumed when the processor is idle (in Figure 6.24), would be reduced to approximately zero. This means that the reduction in leakage power would be 97.2% for the optimized processor.

For 90 nm TSMC, the improvement gained by using voltage scaling or power gating is more or less the same. However, the gain by using power gating is slightly better than the gain achieved by voltage scaling. When it comes to newer technologies than 90 nm TSMC, the leakage power starts to be more dominant compared to the dynamic power consumption. Therefore, one could reduce the leakage power significantly by reducing the duty cycle of the processor as much as possible, and by applying power gating.

Chapter 7

Conclusions

Compared to of-the-shelf general purpose processors, it is possible to reduce the power consumption of a processor significantly for a given set of applications, by designing application specific instruction-set processors (ASIPs). However, in this thesis, it is concluded that the memory usage of the application is dominant in the power consumption. It is only possible to fully optimize the processor if the application is fully optimized in terms of memory usage and power consumption.

In the software development activities presented in Chapter 4, the provided Matlab model was converted into a fixed point C application. This application was then optimized further by using lookup tables, inline functions, reducing read and write operations to global memories, eliminating divisions with constant values where it is possible, combining loops, implementing software division for usage outside the critical loop, implementing a special cosine function using a small lookup table, insertion of pre-processor macros, and re-usage of variables. The run-time memory usage of the application was reduced significantly, to only 150 KB for an ECG sample frequency of 198 Hz.

It is concluded that the data memory usage in the resulting fixed point C application is far to much for a low power general purpose processor, or microcontroller. In order to achieve higher power efficiency, the processor architecture needs to be optimized for the specific application, resulting in an ASIP. In addition, the application requires the use of 32-bit integers, and therefore a 32-bit processor is needed. The accuracy is not satisfactory with the use of 16-bit integers.

In the hardware development presented in Chapter 6, a 16-bit general purpose processor architecture was converted into a 32-bit architecture. The resulting architecture was then optimized by applying several hardware optimizations, i.e., a customized multiply-accumulate (MAC) operation, and by adding instruction level parallelism with two load and one MAC operation. Other optimizations include the creation of customized instructions that combined two or more assembly instructions to be executed in parallel within the same clock cycle.

Due to the added functionality in the optimized processor, and because the data memory has been divided into two independent memories, the leakage power has increased by 13%. For the technology used in this thesis, the leakage is not so dominant compared to the dynamic power. The number of execution cycles of the application has been reduced by 81%, resulting in a reduction of the total average power consumption by approximately 55%.

It is concluded that the processing time is highly dependent on the sample frequency of the input ECG signal. In addition, it is shown that the memories have a significant impact on the total average power consumption, and by reducing the memory usage, the power consumption would be reduced significantly. The processor is optimized for this application by adding

dedicated hardware, resulting in a reduction of the average dynamic power consumption by 78%.

The power consumption could be reduced even further by applying voltage and frequency scaling. A rough estimate shows that by reducing the supply voltage from 1.08 V to 0.6 V using 90 nm low power TSMC process, the dynamic power consumption could be reduced approximately 19 times. This comes at the cost of a slower circuit.

Alternatively, if power gating is applied, then one could shut down the processor when it is idle. Since the idle time is quite big in the case of this application when a clock frequency of 100 MHz is used, then the leakage power consumption is calculated to be reduced by 97.2% in the case of the optimized processor.

The optimized processor can operate in the frequency the range 3 to 100 MHz, without violating the deadline constraints of the application. However, if the clock frequency is 3 MHz, then power gating would not benefit in the reduction of the power consumption, because the processor would finish just in time before the next set of input data arrives. In this case, voltage scaling is a more attractive solution.

Since the leakage power is more dominant for newer technologies, it is concluded that power gating might be a better solution than voltage scaling.

Chapter 8

Future work

In future work, the application should be re-designed, with the focus of reducing the data memory usage. Alternatively, the memories could be forced to operate using slower frequencies than 100 MHz. This would reduce the power consumption by making the processor slower. Other software and hardware power optimization techniques, as described in [BDM00], could be applied as well. The memory size of the program memory could be reduced to 16 KB, which would result in a more power efficient processor.

In the case of this project, the leakage power consumption is not so dominant compared to the dynamic power consumption. However, when the voltage is scaled down by using ultra low voltage libraries, the leakage starts to grow compared to the dynamic power.

If the processor is switched off when it is in the idle state, then the leakage would be reduced. The only leakage present would come from the switch transistor of the power gating. In addition to the reduced dynamic power in the optimized processor, the leakage power consumed when the processor is idle, as shown in Figure 6.24, could be reduced by applying power gating.

It would be interesting to see how the power dissipation changes by applying power gating, and compare the results to the same processor with voltage scaling applied.

Instead of using a random access memory (RAM) for the program memory, one could use flash memory to see how it would effect the power consumption. With RAM, the data is lost when the power is switched off. With flash on the other hand, it would be possible to avoid programming the memory every time the power is shut down.

Appendix A

Analysis of the original Matlab model

In this appendix chapter, the detailed analysis of the CWT function and the QRSDet3 function of the original Matlab model is performed, and in some cases, alternative solutions are suggested.

A.1 The CWT function

The cwt2 function performs the advanced computational parts of the algorithm. It is used in the function called QRSDet3 as one of the steps for the QRS detection method, as described in Section 2.3 and 2.4. The wavelet transform is performed as described in Section 2.2. The cwt2 function is illustrated in Figure A.1.

Defining the Mexican hat mother wavelet

Initially, the Mexican hat mother wavelet is generated using Equation 2.6, and it is stored in an array of 256 elements (the psi reference model in Figure A.1). The scaled mother wavelet, which is an array called *psi_scale* in Figure A.1, is then created by selecting values from the reference model. These values are selected by first by making a temporary array from the lowest bound to the highest bound of the reference model, using steps of one divided by the scale. The elements of this array are then divided by the step size of the reference model, before they are rounded. The resulting temporary array contains the indexes to the desired values of the reference model array. Before the *psi_scale* array is sent to the input of the convolution function, it is flipped using the built-in Matlab function *fliplr*, which flips all elements of a vector horizontally.

If the mexican hat reference model is calculated on a processor every time the cwt2 function is called, it would require a lot of computing power, because of the computational mechanisms like square root functions and divisions. Those functions are extremely expensive in terms of power if implemented in hardware, and cycle counts if software emulated. Since it is only sampled over 256 elements, this model could be stored in a lookup table, and hence save all the calculation costs.

The temporary array, which contains the indexes of the desired elements of the reference model, could introduce scaling errors to the scaled mexican hat wavelet from the rounding. As a consequence, the *psi_scale* is not necessarily symmetrical. That is why this array is flipped before it is sent to the convolution step.

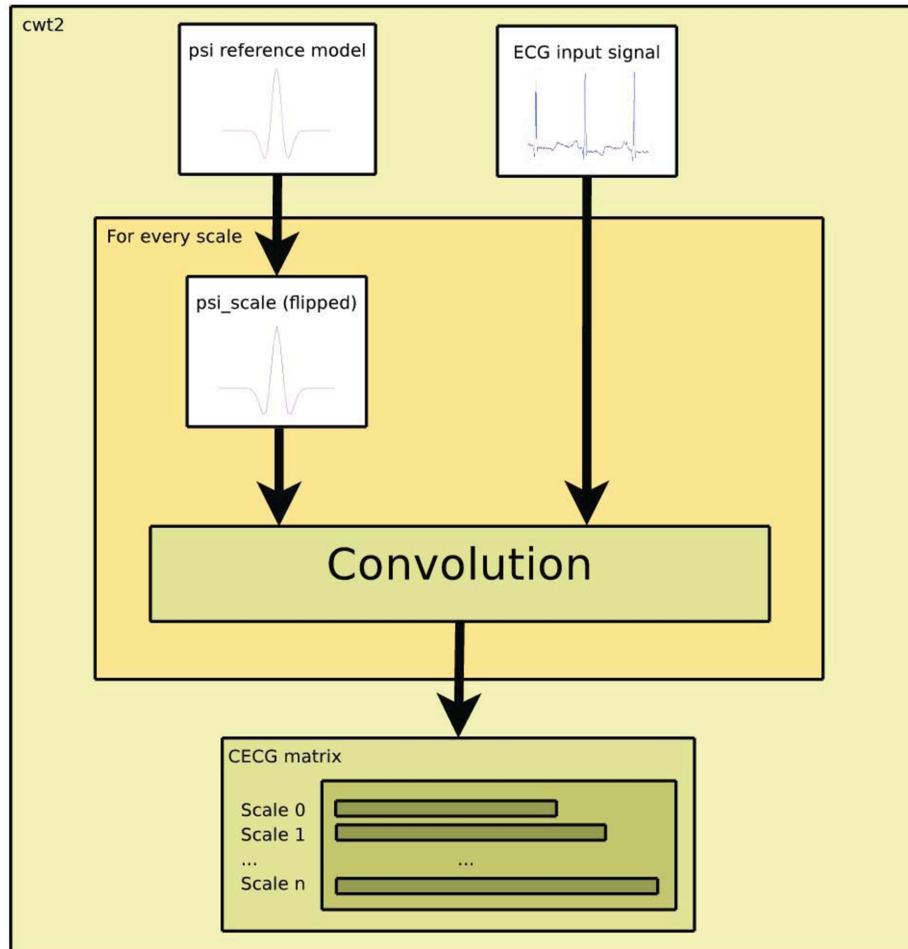

Figure A.1: Illustration of the cwt2 function.

Furthermore, by studying the code carefully, it was observed that if the sample frequency was 1000 Hz, then the scaled mother wavelet would be oversampled because the scaling would give this array a size of 272 elements. That simply means that some values of the reference model would be duplicated. One alternative to solve this oversampling is to increase the sampled Mexican hat wavelet to 512 elements. Another alternative is to make the wavelet dynamically scalable according some factor of the sample frequency of the input signal. Both methods was tested, and the results did not give any significant improvement of the algorithm itself. Therefore, the decision is to keep the fixed Mexican hat wavelet with the size of 256 elements in order to save the additional memory space.

Computation of the wavelet coefficients

The convolution step is then performed using the optimized Matlab convolution function *conv*, [mat]. In addition, the convolution is done for every scale separately. After the computation of the wavelet coefficients, the results for each scale are divided by the square root of the scale value, before they are stored in a matrix called *cecg*. In this matrix, coefficients from each scale are stored in different rows, as shown in Figure A.1. The output size of the convolution result (for every scale) is equal to the size of the scaled mother wavelet and the ECG signal minus one. Therefore, there are less coefficients for the lowest scale

than the highest scale. This implies that the *cecg* matrix must be big enough to cover the coefficients of the largest scale. The *cecg* matrix is the output of the cwt2 function.

The maximum frequency used in the test databases is 1000 Hz, giving at most five different scales. Considering the large sizes of the input ECG signal and the scaled mother wavelet, the cwt2 function was assumed to be within the critical loop of the algorithm, because the convolution step would iterate through most elements many times.

Since the convolution function does not exist in C, it must be created. This could be done by converting the Matlab function, or implement the convolution by its definition [Weia]. The Matlab function *conv* is therefore analyzed and compared to the standard definition of the discrete convolution function. This analysis is described in Appendix C. When the input signal is sampled with frequency 1000 Hz, it has 3000 elements. The scaled mother wavelet then contain 272 elements for this frequency. Based on these input sizes, the analysis concluded that the number of iterations is a around 11.7 million times in the worst case when using the convolution function by the definition. However, the number of iterations is dependent on the number of scales, and the size of the scaled wavelet. Instead of iterating through all elements in the inner loop every time, the optimized Matlab function uses a fast convolution algorithm. In the worst case, the Matlab *conv* function iterates 5.4 million times. This is significantly better than the convolution function by the definition. Therefore, a direct conversion of the Matlab *conv* function is suggested.

A.2 The QRSDet3 function

The QRSDet3 function is the largest function, containing all the steps in Figure 2.8. However, this function only performs the QRS-detection method on a 3-second input interval, as described in Section 2.4.

In the first step of this function, the ECG signal is normalized by subtracting the mean value from every element, and the desired scales are calculated before entering the CWT step. The continuous wavelet transform step is performed using the cwt2 function as described in Section A.1.

Calculation of the modulus maxima

The output of the cwt2 function (the *cecg* matrix) is used in the calculation of the modulus maxima. First, three new matrices are created, called $rightc$, $centc$ and $leftc$. $rightc$ contains column 1 to N-2 of the *cecg* matrix, where N is the total number of columns. Similarly, $leftc$ contains column 3 to N, and *centc* contains column 2 to N-1. This is illustrated in Figure A.2 for a 3x5 matrix. By doing so, the modulus maxima could be calculated by extracting the indexes of all elements that fulfill the constraints in Equation A.1. In Matlab, those elements are found by the built-in $find$ function, with the constraints as the input to that function.

$$(centc > leftc)\&(centc > rightc))\|((centc < leftc)\&(centc < rightc), \tag{A.1}$$

Copying the *cecg* matrix, which is the largest matrix in this algorithm, is extremely inefficient. Instead of creating three matrices, one could just modify the *cecg* matrix itself. For instance, by using a temporary list, the indexes of all values that do not meet the constraints could be copied for each row. By iterating through this array, the elements in *cecg* matrix could be reset, given by the index values. The resulting *cecg* matrix would then contain only the parameters that meet the constraints in Equation A.1. All elements that do not exceed the threshold are set to zero. By doing so, one would eliminate the need for three extra matrices, and thus save the memory space. This alternative solution is suggested because the *cecg*

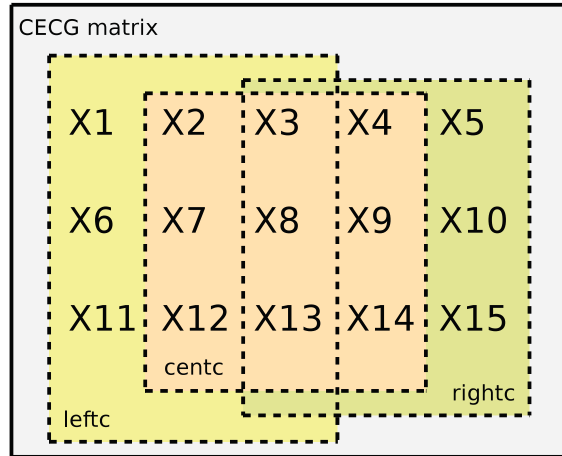

Figure A.2: Creation of the modulus maxima matrices.

matrix is the biggest matrix in the application, and saving the extra memory space would reduce the memory requirement significantly.

Finding the modulus maxima above the running threshold

After the modulus maxima are found, the algorithm defines the running threshold based on the old threshold, and extracts the values of all modulus maxima above this threshold. This is done for each scale. The Matlab code stores the indexes of all these values in a matrix called *posqrs*, and the absolute value of all modulus maxima above the threshold are stored in a matrix called *posqrsval*. In these matrices, each row represents a different scale. This is illustrated in Figure A.3.

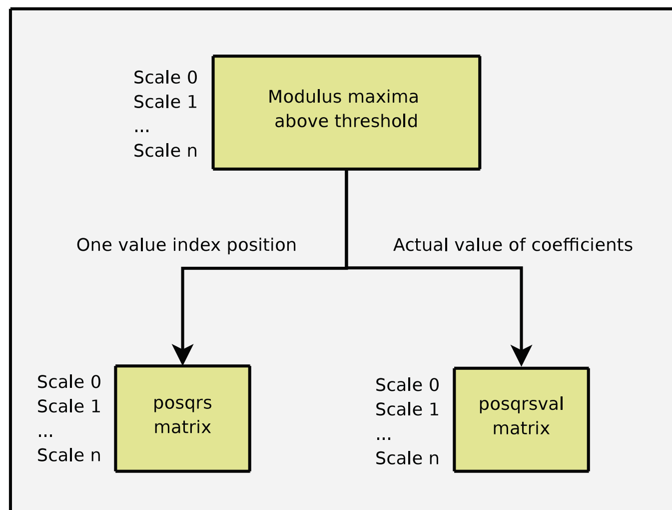

Figure A.3: Modulus maxima above the threshold.

The creation of the *posqrsval* matrix is a waste of space, because the values from the *cecg* matrix are just copied. However, the *posqrsval* matrix is much smaller than the *cecg* matrix, which means that there is a possibility of re-using the *cecg* matrix in a later step. Therefore, no alternative implementation suggestions are given for this step.

Finding clusters of beats

The clusters of beats are small intervals surrounding the actual peak of each beat. These clusters are determined by iterating through all index elements in *posqrs*. For each element in *posqrs*, the algorithm checks if the current element is already added to a matrix called *clust*. If not, the value of this element is subtracted from the *posqrs* matrix, and only the index of the remaining elements with values within 0.02 times the sample frequency are stored in a temporary array. The values in this temporary array are then added to a new row in the *clust* matrix. In addition, the row position for every element in the *clust* matrix is calculated using the built-in Matlab function *ind2sub*, and the results are stored in a matrix called *row*. This step is illustrated in Figure A.4.

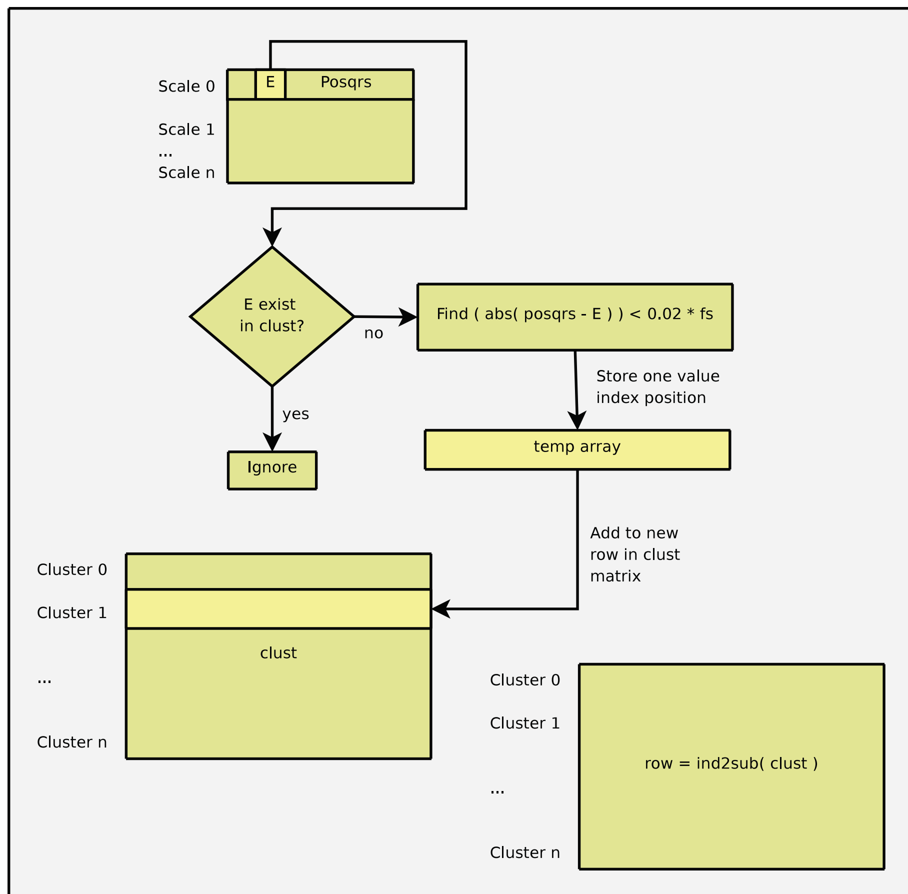

Figure A.4: Finding clusters of beats.

In C, the elements of a matrix are accessed by selecting the row and column position. This requires two index parameters. In Matlab, the elements of a matrix could be accessed using only one index parameter. The elements are then accessed column wise, i.e., in a 3x3 matrix M, the matrix element $M(1,1)$ could be accessed in Matlab by typing $M(1)$, and $M(1,2)$ by typing $M(4)$. Figure A.5 illustrates this.

This method of accessing elements is used throughout the Matlab model, and also in calculations during later steps. To do this in C, the easiest solution would be to copy this functionality. Finding another way of doing the calculations by using C indexing would be more optimal. However, this would result in changing the Matlab model completely.

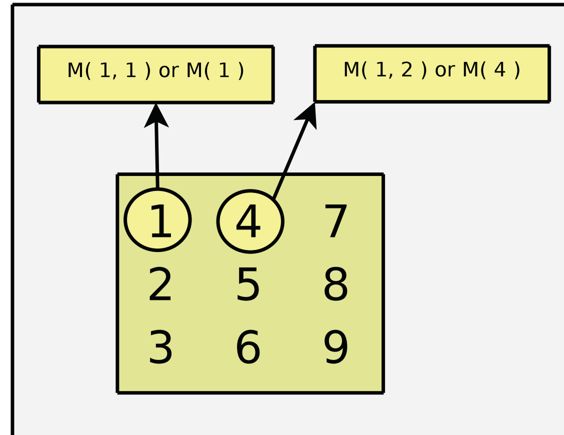

Figure A.5: Matlab indexing of matrices.

Classification of beats

All beats are then classified, according to whether or not they meet the criterion of being present in 2 out of 4 wavelet scales (as discussed in Section 2.4). For every cluster the representative time corresponding to the R-peak is selected. This is done by finding all unique values in the *clust* matrix for every row, and checking if the number of unique values are more than two. If so, then all elements of this row in the *clust* matrix are copied over to a matrix called *validclust*. Selecting all unique values is done using the built-in Matlab function *unique*. In addition, the mean value of this row in *posqrs*, which contain the indexes of the coefficients, is calculated for every valid cluster, and stored in an array called *meantime*. If the *validclust* matrix is empty after this step, i.e., no beats was detected, then all beats in *clust* are defined to be valid and copied over to *validclust*. The *meantime* array is then sorted, before the *validclust* matrix is sorted so that the first element in the *meantime* array corresponds to the first row in *validclust*, and so on. This step is illustrated in Figure A.6.

In this step, the matrices are accessed using the Matlab one-value indexing. As discussed in the previous step, the easiest way to implement this step is to copy the functionality. In addition, the *unique* function does not exist in C, and an alternative solution must be implemented. The easiest way to implement the *unique* function, is done by iterating through every element, and compare them with each other. This might not be the optimal solution, since comparing every element with each other would require multiple iterations. However, this step is not in the most critical loop of the algorithm. Therefore, the easy approach is suggested, where a further optimization of this step could be applied in future work.

Removing clusters too close to each other

Clusters that are too close to each other are then removed before the beats are stored in an array called *beats*. The values of the beats are picked out from the *posqrsval* matrix. Another array, $qrs2$, is then created and the beats divided by the sample frequency are stored in this array. The values in $qrs2$ are now represented in seconds of when a beat is detected in the 3-second interval.

In this step, the beats are divided by the sample frequency, resulting in a floating point representation. The possibility of using the detected beats as output of this function, instead of the time in seconds where they were detected, could make the conversion to fixed point easier. However, this re-arrangement would require several changes to the test environment.

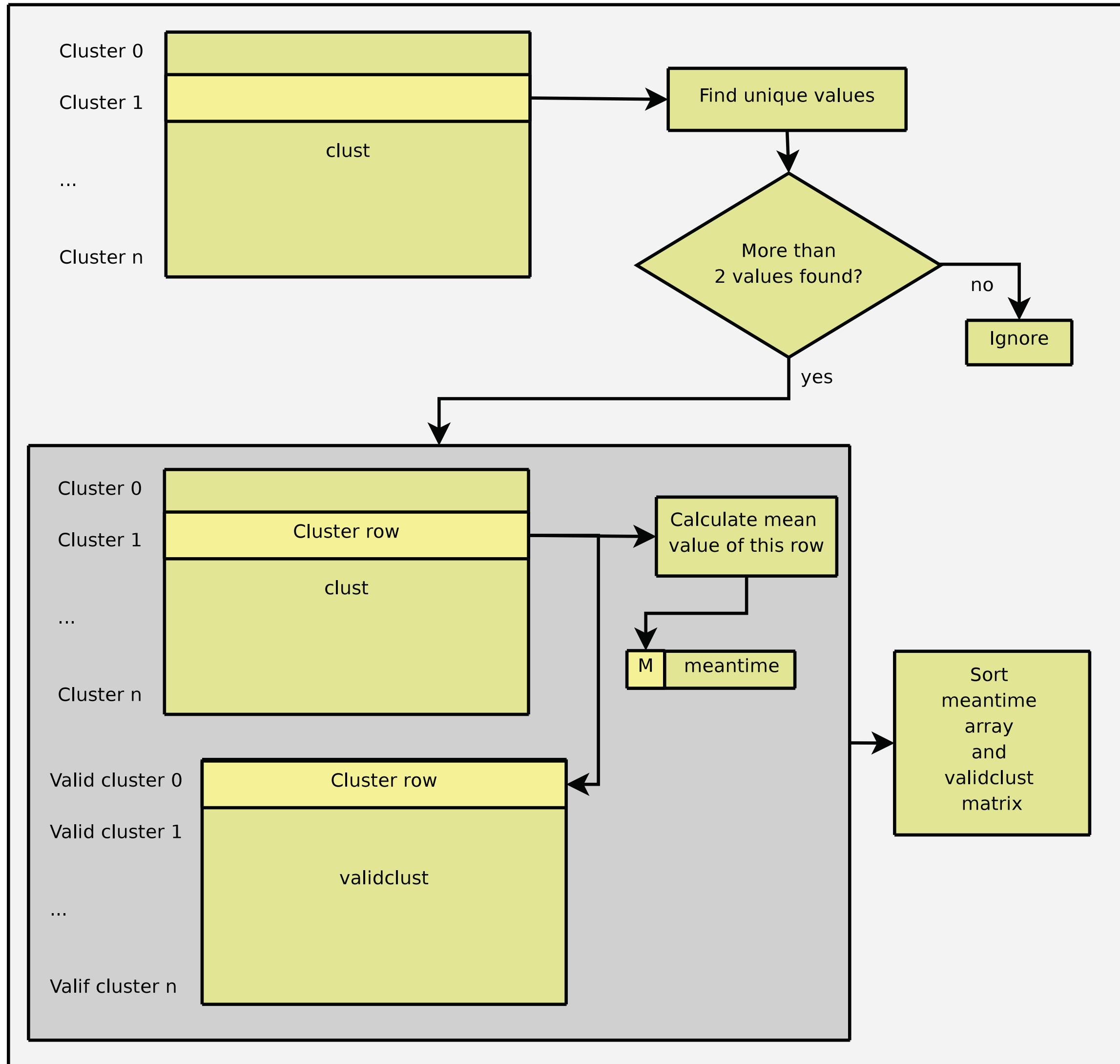

Figure A.6: Classification of beats.

The decision is not to do these changes, and consider this alternative in a future optimization if the fixed point C application does not give good enough accuracy.

Back to time domain

Before entering the final step of the algorithm, the beat from every valid cluster within the time interval is determined using a Hanning window. This Hanning window is implemented by using Equation A.2. The Hanning window is used as a window function to remove any unwanted noise and peaks near the R-peak of the QRS complex. It is usually used to reduce aliasing in Fourier transforms, [Weib], and according to [han], the Hanning window gives the best overall filter characteristics. The Hanning function is in [Weib] defined as shown in Equation A.2.

$$w(n) = 0{,}5 \cdot (1 - cos(\frac{2\pi n}{N-1})), \tag{A.2}$$

where N is the total number of elements, and n is an element from $-N/2$ to $N/2-1$. This function is illustrated in Figure A.7.

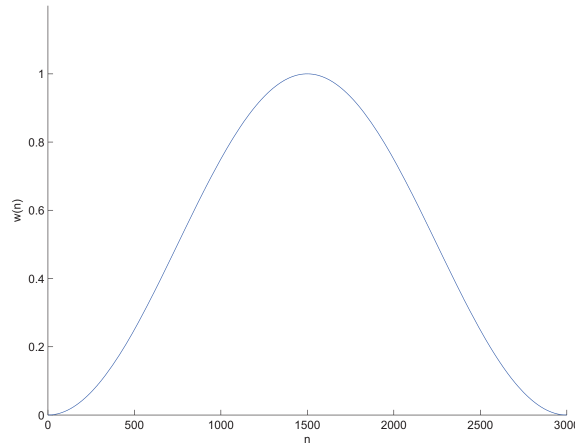

Figure A.7: The hanning window by definition.

This general function is modified in order to reduce the amplitude of the ECG signal except from a small interval around the R-peak. Since the R-peak is not always located at the highest point of the cosine function, the reduction of the R-peak amplitude is avoided. In Matlab, this is implemented by creating three arrays, $right$, $center$ and $left$. In $right$ and $left$, the corresponding side of the cosine window is calculated and stored. In $center$, the array is filled with ones in order to keep a small search interval around the R-peak unchanged. The size of the center array depends on the search interval calculated at runtime. An example of the modified Hanning window is shown in Figure A.8.

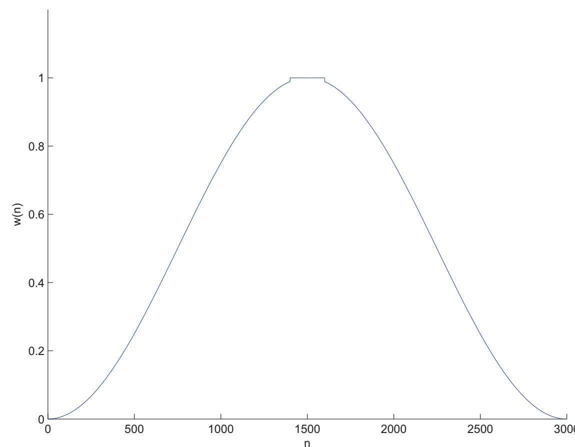

Figure A.8: The modified hanning window.

For all beats in the $qrs2$ array, this search interval is selected around the beat position. The Hanning window is then multiplied with the ECG signal input of the algorithm around this interval, and the maximum value of this interval is stored in the $qrs2$ array as the actual detected beat.

In the final stage of the QRSDet3 function, those beats are identified in time domain as described in Section 2.3. The output of this function is a time array (the $qrs2$ array) of detected beats (i.e., the ECG of the heart is measured for 3 seconds, and this function takes

the measured signal as input and returns an array, consisting of the times where it detected a heart beat).

This step is modeled in a straight forward manner, and there is no need to do an alternative implementation of this step.

Appendix B

Detailed description of the Matlab prototype

In this appendix chapter, all steps that required alternative implementations of the original Matlab code, i.e., the suggestions made in Chapter 4.3 and Appendix A, are described in detail.

B.1 Changes to the CWT step and modulus maxima step

Starting with the cwt2 function, the computation of the scaled Mexican hat wavelet is simplified, by dividing the compact Matlab code into several steps. First, the indexes are created and stored in an array. The Mexican hat mother wavelet is then accessed by the rounded values of the array. The chosen elements are then stored in the opposite end of the scaled mother wavelet list. This is done in order to emulate the $fliplr$ Matlab function. This scaled array is then sent to the convolution step along with the ECG signal.

Since the analysis in Appendix C proved that the Matlab convolution function was more optimal, this function is converted according to the Matlab documentation.

The square root calculation after the transform is integrated with the convolution step in order avoid iterating through the convolution array more than necessary.

In addition, the *cecg* matrix is defined with a fixed size which was large enough to include the worst case. The biggest size of this matrix is approximated to be five rows with 3300 elements when the ECG sample frequency was 1000 Hz. The actual used size of the matrix, in columns and rows, is updated during the calculation of coefficients. The rows in this case are the number of scales, and the columns are calculated at the end of the transform, taking into account the scale with most columns. By doing it this way, iterating through the whole allocated matrix would not be necessary, but only to the boundaries defined by the columns and rows. The calculation of the modulus maxima is changed as proposed in Section 4.2, in order to save memory usage.

B.2 Changes to the step for finding clusters of beats

Major changes are done in the step where the clusters of beats are found. In addition to the *clust* matrix, another clust matrix is introduced. This is called $clust2$, and the difference between them are how the indexes are stored in them. In the original Matlab function, all calculations and manipulations of matrices were done using only one index variable. Therefore, the *clust* matrix is used to store all indexes using the Matlab method of accessing

elements in a matrix, while *clust*2 stores the column indexes of the accessed elements, as normally done in C. The first attempt was to find a function that could calculate the right column by looking at the row matrix and clust matrix, but this resulted in a more complex solution. Therefore the solution using *clust*2 is chosen due to the limited time frame of the project.

The size of the cluster columns are assumed to be at maximum the *posqrs* column size, and since there could be several clusters in each scale, the assumed cluster row size is set to be twice the number of scales. The arrays *coltmp*, *rowtmp* and the matrix *abstmp* were created. These arrays are used as temporary arrays in order to calculate the intermediate steps illustrated in Figure A.4. *abstmp* has the same size as *posqrs*, and *rowtmp* and *coltmp* has the same length as the cluster column. Figure B.1 shows the changes from Figure A.4.

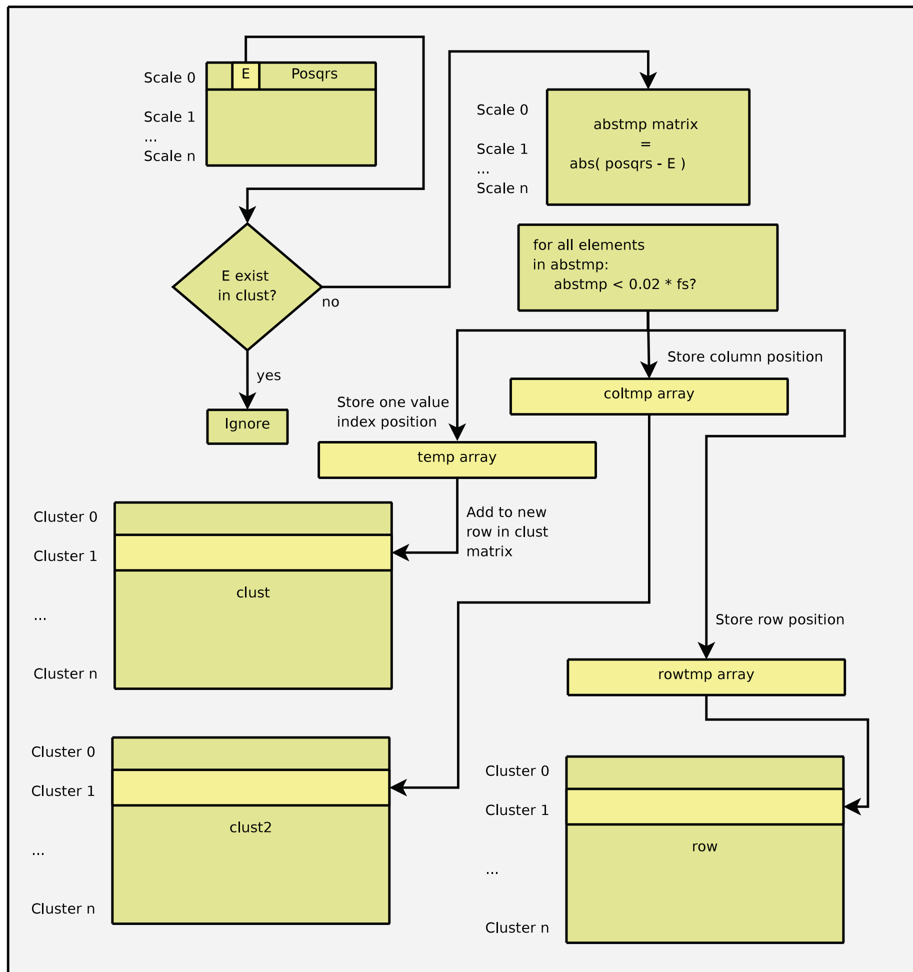

Figure B.1: Finding clusters of beats in the prototype.

B.3 Changes to the classification of beats

The classifications of beats (for *validcluster*) are done using the same Matlab indexing technique as the one described in the step of finding the clusters. Within this step, the original Matlab code used a *unique* function to find all unique elements in the *row* matrix. This function has been implemented by going through all elements in the *row* matrix, and

every index value which has not been discovered before is stored in a temporary array. By checking if an element exist in this temporary array, the unique element could be found. The new model is illustrated in Figure B.2.

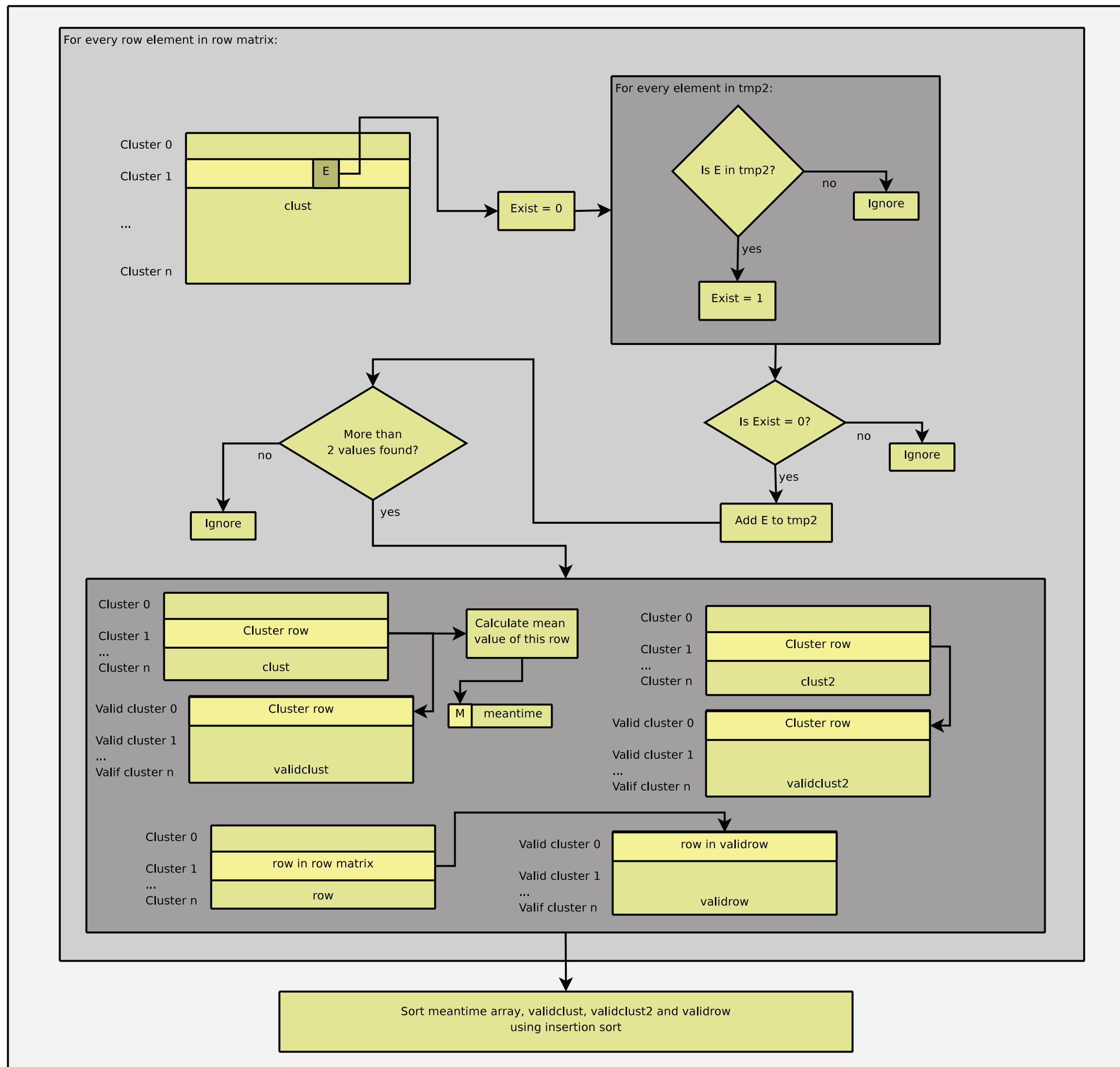

Figure B.2: Finding valid clusters of beats in the prototype.

After the classification of the valid clusters, the original code sorts the *validcluster* and *validrow* matrix. Here, this is done by using insertion sort as described in [CLRS03]. The reason for choosing the insertion sort algorithm is because it is not recursive and it is easy to implement.

Appendix C

Calculation of convolution iteration times

The convolution function in Matlab is defined in [mat] as shown in Equation C.1.

$$(u * v)[k] = w(k) = \sum_{j} u(j) \cdot v(k-j), \tag{C.1}$$

where the length of w is $m + n - 1$, m is the length of u, n is the length of v, and the range of j is defined in Equation C.2.

$$j = max(1, k + 1 - n)..min(k, m) \tag{C.2}$$

The discrete convolution is defined as shown in Equation C.3, [Weia].

$$(f * g)[n] = w(n) = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m] \cdot g[m] \tag{C.3}$$

In the case the discrete list of elements is finite, the length of w would be $N = L_f + L_g - 1$, where L_f is the length of f, and L_g is the length of g. This yields the following equation:

$$w(n) = \sum_{m=0}^{N-1} f[m] \cdot g[n-m]. \tag{C.4}$$

The convolution code which was implemented using the original definition of the convolution function was implemented using Code Snippet C.1.

```
1 for n=1:x_size+h_size-1
2  for m=1:h_size
3   if( ((n-m) ≥ 0) && ((n-m) < x_size) )
4    convrs(n)=(convrs(n)+(x(n-m+1)*h(m)));
5   end
6  end
7 end
```

Code Snippet C.1: Matlab code for convolution by the original definition.

In Code Snippet C.1, x is the *psi_scale*, h is the ECG signal, and *h_size* and *x_size* are their respective array length. By analyzing the number of loop iterations for the ECG sample frequency of 1000 Hz, one could estimate how many times the convolution function loops through its code (how many times the if-sentence is checked). In this case, the *psi_scale* has a length of 272 elements, and the ECG signal has approximately 3300 elements. The

inner loop of the code is visited 3300 times in this case. Since the outer loop is executed $3300 + 272 - 1$ times, the total number of times the inner loop is visited is approximately 11.7 million times.

The matlab function was analyzed in a similar manner, but here the inner loop was iterated the same number of times as j. In addition, the length of j is changed every time the code in the outer loop is executed, dependent on the value of k in Equation C.2. In order to estimate j, three cases has to be considered:

1 When $k < 272$, the the total number of iterations is by Equation C.5 calculated to be 37 128 times.

2 For the interval $272 < k \leq 3300$, the number of iterations is approximately 5.4 million times according to Equation C.6 ($a = 272$ and $b = 3300$).

3 When $k > 3300$, Equation C.6 results in approximately 863 772 times($a = 3300 - 272$ and $b = 3300$).

The sum of all three cases gives the total number of iterations this convolution function perform in the worst case. The number of iterations is in this case approximately 6.3 million iterations. This means that the Matlab convolution function is almost twice as fast as the convolution function implemented using the original definition. Another observation is that the *psi_scale* array is much smaller than the ECG signal. By changing the x array with the h array in the original convolution function (as done in the second case of Equation C.3), one would get a maximum of approximately 9 million iterations.

The Matlab convolution is in any case significantly faster than the original convolution function, and therefore this function was implemented in the C code. In Code Snippet C.2, the C code for the implemented floating point convolution function is shown, based on the Matlab convolution function.

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \tag{C.5}$$

where k and n are integers.

$$\sum_{k=a}^{b} k = \sum_{k=1}^{b} k - \sum_{k=1}^{a-1} k = \frac{b^2 + b - a^2 + a}{2}, \tag{C.6}$$

where k, a and b are integers.

```
1  /*
2   * Function: convC
3   * Input:
4   *  int _s      : Current scale
5   *  int _x_size    : Actual size needed of the allocated
6   *            signal x (Mexican hat wavelet)
7   *  int _h_size    : Actual size needed of the allocated
8   *            signal h (ECG signal)
9   *  doublelist _x    : Mexican hat wavelet as the mother wavelet
10  *  doublelist _h    : ECG signal
11  *  doublelist _convrs  : Allocated array for the result of this algorithm
12  *
13  * Result:
14  *  A modified _convrs with the convolution
15  *  of the signals _x and _h.
16  *
17  * Description:
18  *  This Algorithm calculates the convolution of two signals, _x and _h,
19  *  and returns the result divided by the root of the current scale.
20  */
21 void convC(int _s, int _x_size, int _h_size, doublelist &_x, doublelist &_h,
      doublelist &_convrs){
22  // Define parameters
23  int n=0;
24  int m=0;
25  int m_min=0;
26  int m_max=0;
27  double temp = 0;
28
29  for(n = 1; n ≤ (_h_size + _x_size − 1); n++){
30   // Do the convolution
31   // According to matlab specification (faster)
32   m_min = max(1,n+1−_x_size);
33   m_max = min(n,_h_size);
34
35   temp = 0;
36   for(m = m_min; m ≤ m_max; m++){
37    temp = temp + (_x.array[n−m]*_h.array[m−1]);
38   }
39
40   // Divide result with the root of s
41   _convrs.array[n−1] = sqrt_lookup[_s−1] * temp;
42  }
43  _convrs.size = (_h_size + _x_size − 1);
44 }
```

Code Snippet C.2: C code for convolution by the Matlab definition.

Appendix D

Solving issues related to low threshold values

If the fixed point representation for the threshold values get out of their specified fixed point range in accuracy, the thresholds get to low. This means that to many coefficients are assigned for the clusters. In that case, a variable called *poscol* is assigned a value much higher than the pre-allocated array sizes for the clusters. Some memory reads are dependent on this variable, and if this variable is too high, then it is possible that the application in later stages might try to access the clusters with an index outside their memory boundaries.

In order to protect the clusters from running out of memory, an upper limit is defined for *poscol*. This limit is set to be the column size of the *posqrs* matrix. If more elements are attempted to be accessed, then they are simply ignored. This modification was done by changing Code Snippet D.1 to Code Snippet D.2.

```
1   if ((cecg.matrix[i-1][j-1]>>8) ≥ _thm_new.array[i-1] ){
2    posqrs[i-1][pos-1] = j;
3    posqrsval[i-1][pos-1] = abs(cecg.matrix[i-1][j-1]);
4    iszero = 0;
5
6    // To make sure the largest row is known
7    if (pos > poscol){
8     poscol = poscol + 1;
9    }
10   pos = pos + 1;
11  }
```

Code Snippet D.1: The code before inserting the memory protection.

```
1  // To assure poscol is never higher than its limit
2  if(i ≤ posrow && pos ≤ posqrs_size){
3   if ((cecg.matrix[i-1][j-1]>>8) ≥ _thm_new.array[i-1] ){
4    posqrs[i-1][pos-1] = j;
5    posqrsval[i-1][pos-1] = abs(cecg.matrix[i-1][j-1]);
6    iszero = 0;
7
8    // To make sure the largest row is known
9    if (pos > poscol){
10    poscol = poscol + 1;
11   }
12   pos = pos + 1;
13  }
14 }
```

Code Snippet D.2: Memory out of bounds protection

Bibliography

[Abd] Dr. A. M. Abdulla. Heart electrical activity. http://www.heartsite.com. http://www.heartsite.com/html/electrical_activity.html on April 07, 2009.

[abo] About target. http://retarget.com/about.php on May 16, 2009.

[ATNL99] V.X. Afonso, W.J. Tompkins, T.Q. Nguyen, and Shen Luo. Ecg beat detection using filter banks. *Biomedical Engineering, IEEE Transactions on*, 46(2):192–202, Feb. 1999.

[BDM00] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2):115–192, 2000.

[BFDVS+98] S. Barro, M. Fernandez-Delgado, J.A. Vila-Sobrino, C.V. Regueiro, and E. Sanchez. Classifying multichannel ecg patterns with an adaptive neural network. *Engineering in Medicine and Biology Magazine, IEEE*, 17(1):45–55, Jan/Feb 1998.

[Bra] D. Brand. Integration of runtime profiling and static code analysis in linux. Computing Science Department, University of British Columbia, Vancuver, BC, Canada. danube@cs.ubc.ca.

[CC99] A.G.M. Cilio and H. Corporaal. Floating point to fixed point conversion of c code, 1999.

[CC03] S.W. Chen and H.C. Chen. Development of a real-time qrs beat classifier using a nonlinear trimmed moving averaging-based system. pages 577–580, Sept. 2003.

[CDGS98] F. Catthoor, E. De Greef, and S. Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[CFW+94] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man. Global communication and memory optimizing transformations for low power signal processing systems. pages 178–187, 1994.

[CHC06] A. Chao-Hung Cheng. *Application-specific architecture framework for high-performance low-power embedded computing*. PhD thesis, Ann Arbor, MI, USA, 2006. Adviser-Tyson, G. S. and Adviser-Mudge, T. N.

[CL93] B. Cabral and L.C. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 263–270, New York, NY, USA, 1993. ACM.

[CLRS03] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, 2nd edition, December 2003.

[DCDM97] K. Danckaert, K. Catthoor, and H. De Man. System level memory optimization for hardware-software co-design. In *CODES '97: Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, page 55, Washington, DC, USA, 1997. IEEE Computer Society.

[DGF90] A. De Gloria and P. Faraboschi. An evaluation system for application specific architectures. pages 80–89, Nov 1990.

[DNYB+07] M. De Nil, L. Yseboodt, F. Bouwens, J. Hulzink, M. Berekovic, J. Huisken, and J. van Meerbergen. Ultra low power asip design for wireless sensor nodes. pages 1352–1355, Dec. 2007.

[DPT98] W.E. Dougherty, D.J. Pursley, and D.E. Thomas. Instruction subsetting: Trading power for programmability. pages 42–47, Apr 1998.

[EM89] J. Eliot and B. Moss. Addressing large distributed collections of persistent objects: The mneme project's approach. In *DBPL*, pages 358–374, 1989.

[FN06] J. Fraden and M. R. Neuman. Qrs wave detection. *Medical and Biological Engineering and Computing*, 2006.

[GAG+13] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000 (June 13). Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215.

[GM01] T. Glokler and H. Meyr. Power reduction for asips: a case study. pages 235–246, 2001.

[GVPL+97] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P.G. Paulin. Embedded software in real-time signal processing systems: design technologies. *Proceedings of the IEEE*, 85(3):436–454, Mar 1997.

[Ham80] J.R. Hampton. *The E.C.G. Made Easy*. Churchill Livingstone, 2 edition, 1980.

[han] Hanning window. From http://www.diracdelta.co.uk. http://www.diracdelta.co.uk/science/source/h/a/hanning%20window/source.html on April 26, 2009.

[HRS] HRSonline. The normal heart: the electrical system. http://www.hrspatients.org. http://www.hrspatients.org/patients/the_normal_heart/electrical_system.asp on April 07, 2009.

[IY98] T. Ishihara and H. Yasuura. Power-pro: programmable power management architecture. pages 321–322, Feb 1998.

[JBK01] M.K. Jain, M. Balakrishnan, and A. Kumar. Asip design methodologies: survey and issues. pages 76–81, 2001.

[JM98] Y.-M. Joo and N. McKeown. Doubling memory bandwidth for network buffers. volume 2, pages 808–815 vol.2, Mar-2 Apr 1998.

[KAFK+05] K. Karuri, M.A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained application source code profiling for asip design. pages 329–334, June 2005.

[KFA+07] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual: For System-on-Chip Design.* Springer Publishing Company, Incorporated, 2007.

[KHO02] B.-U. Kohler, C. Hennig, and R. Orglmeister. The principles of software qrs detection. *Engineering in Medicine and Biology Magazine, IEEE*, 21(1):42–57, Jan.-Feb. 2002.

[KHO03] B.U. Kohler, C. Henning, and R. Orglmeister. *QRS Detection Using Zero Crossing Counts.*, pages 138–145. Erlangen, 2003.

[KI97] A. Kalambur and M.J. Irwin. An extended addressing mode for low power. pages 208–213, Aug 1997.

[KJBK04] M. Kumar Jain, M. Balakrishnan, and A. Kumar. An efficient technique for exploring register file size in asip design, 2004.

[KKA97] L. Keselbrener, M. Keselbrener, and S. Akselrod. Nonlinear high pass filter for r-wave detection in ecg signal. *Medical engineering & physics*, 19(5):481–484, 1 June 1997.

[KR05] K. C. Kassner and K. G. Ricks. Hardware/software co-design of embedded real-time systems from an undergraduate perspective. In *WCAE '05: Proceedings of the 2005 workshop on Computer architecture education*, page 9, New York, NY, USA, 2005. ACM.

[Kuc99] K. Kucukcakar. An asip design methodology for embedded systems. pages 17–21, 1999.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[LR94] D.B. Lidsky and J.M. Rabaey. Low-power design of memory intensive functions. pages 16–17, Oct 1994.

[LR02] Y. Liao and D.B. Roberts. A high-performance and low-power 32-bit multiply-accumulate unit with single-instruction-multiple-data (simd) feature. *Solid-State Circuits, IEEE Journal of*, 37(7):926–931, Jul 2002.

[LRW+03] I.R. Legarreta, M.J. Reed, J.N. Watson, P.S. Addison, N. Grubb, G.R. Clegg, C.E. Robertson, and K.A.A. Fox. Long term continuous collection of high resolution ecg signals from a coronary care unit. pages 407–409, Sept. 2003.

[LZT95] C. Li, C. Zheng, and C. Tai. Detection of ecg characteristic points using wavelet transforms. *Biomedical Engineering, IEEE Transactions on*, 42(1):21–28, Jan. 1995.

[Mal99] S. Mallat. *A Wavelet Tour of Signal Processing, Second Edition (Wavelet Analysis & Its Applications)*. Academic Press, September 1999.

[mat] conv. www.mathworks.com. Site last visited 14 April, 2009.

[MCCS02] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 270–276, New York, NY, USA, 2002. ACM.

[MD08] P. Mishra and N. Dutt. *Processor Description Languages, Volume 1*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[MHM+93] A. Maton, J. Hopkins, C. W. McLaughlin, S. Johnson, M. Q. Warner, D. LaHart, and J. D. Wright. *Human Biology and Health*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993.

[NdMM07] N. Nedjah and L. de Macedo Mourelle. *Co-design for System Acceleration: A Quantitative Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Obe07] E. L. Oberstar. Fixed-point representation & fractional math. August 30 2007. http://www.superkits.net/whitepapers/Fixed%20Point %20Representation%20&%20Fractional%20Math.pdf.

[Ped02] M. Pedram. *Power Aware Design Methodologies*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[pyn] http://www.physionet.org. Visited on April 08, 2009.

[RAP08] M. Rashid, L. Apvrille, and R. Pacalet. Application specific processors for multimedia applications. pages 109–116, July 2008.

[RHY06] M. B. Raez, M. S. Hussain, and Mohd F. Yasin. Techniques of emg signal analysis: detection, processing, classification and applications. *Biological procedures online*, 8:11–35, 2006.

[RJ98] J.T. Russell and M.F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. pages 328–333, Oct 1998.

[RLAR+05] I. Romero Legarreta, P.S. Addison, M.J. Reed, N. Grubb, G.R. Clegg, and C.E. Robertson. Continuous wavelet transform modulus maxima analysis of the electrocardiogram: beat characterisation and beat-to-beat measurement. *International Journal of Wavelets, Multiresolution and Information Processing*, 2005.

[RLG08a] I. Romero Legarreta and B. Grundlehner. Development and evaluation of an ecg beat detection algorithm for ambulatory monitoring applications. Technical report, IMEC-NL, 2008.

[RLG08b] I. Romero Legarreta and B. Grundlehner. Ecg datafile format. Technical report, IMEC-NL, October 15 2008.

[RLS01] I. Romero Legarreta and L. Serrano. Ecg frequency domain features extraction: a new characteristic for arrhythmias classification. volume 2, pages 2006–2008 vol.2, 2001.

[Sch05] P. Schniter. Continuous wavelet transform. Connexions, June 9 2005. http://cnx.org/content/m10418/2.13/ on April 06, 2009.

[Tar03] Target Compiler Technologies n.v., Haasrode Research Park, Technologielaan 11-0002, B-3001 Leuven, Belgium. *CHESS/CHECKERS: A retargetable tool-suite for embedded processors*, 3.3 edition, June 2003.

[Tar08] Target Compiler Technologies n.v., Haasrode Research Park, Technologielaan 11-0002, B-3001 Leuven, Belgium. *Primitives Definition and Generation*, November 2008. Release 08R1.

[Tar09] Target Compiler Technologies n.v., Haasrode Research Park, Technologielaan 11-0002, B-3001 Leuven, Belgium. *Target base core processor manual*, March 2009. v4.0.

[Tex06] Texas Instruments. *TMS320C6000 DSP Platform: The Highest-Performance DSPs*, 2006. Updated Q1 2006.

[Tex09] Texas Instruments. *MSP430 Ultra-Low-Power Microcontrollers*, 2009.

[Vol59] J. Volder. The cordic computing technique. In *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 257–261, New York, NY, USA, 1959. ACM.

[Weia] E.W. Weisstein. Convolution. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Convolution.html on April 26, 2009.

[Weib] E.W. Weisstein. Hanning function. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HanningFunction.html on April 26, 2009.

[Wik08] Wikipedia. Hann function, December 09 2008. http://en.wikipedia.org/wiki/Hann_function on April 10, 2009.

[Wik09a] Wikipedia. Cardiovascular disease, April 06 2009. http://en.wikipedia.org/wiki/Cardiovascular_disease on April 06, 2009.

[Wik09b] Wikipedia. Electromyography, April 08 2009. http://en.wikipedia.org/wiki/Electromyography on April 10, 2009.

[WJH03] B.B. Welch, K. Jones, and J. Hobbs. *Practical Programming in Tcl & Tk*. Prentice Hall Professional Technical Reference, 2003.

[WK05] W. Wolf and M. Kaufmann. *Computers as Components: Principles of Embedded Computer System Design*. Morgan Kaufmann, 2005.

[WS97] W. Wolf and J. Staunstrup. *Hardware/Software CO-Design: Principles and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.