TU Delft

Master Thesis

Implementation and benchmarking of processor architectures for application-specific instruction set processors for implantable medical devices

Author:
Jasper Smit (4217527)

April 26, 2021

Abstract

An application-specific instruction set processor (ASIP) can provide for higher power and computational efficiency compared to general-purpose processors. These attributes are essential for implantable medical devices which often run computationally intensive tasks on a strict power budget. This thesis compiles a collection of benchmarks by porting the existing benchmark suites ImpBench and CoreMark, and by implementing a novel benchmark for artificial neural networks. Four architectures are selected in the comparison; RISC, DSP, VLIW, and TTA. Implementations of these architectures are produced by the ASIP Designer and OpenASIP toolsets. The benchmarks are simulated on these implementations and the power consumption is measured on an FPGA. The thesis concludes that the implementations of the DSP and VLIW architectures do not deliver enough performance for their heavier use of resources, and recommends a follow-up research by extending the TTA PeLoTTA and RISC-V Tzscale processors with application-specific instructions and running simulations for ASIC power and area numbers.

Acknowledgements

First and foremost, I would like to thank my supervisors Zaid Al-Ars and Joost Hoozemans from the TU Delft Accelerated Big Data Systems group, who granted me the opportunity to start this research and supported me throughout. Their insightful periodical discussions contributed significantly to this thesis. I would also like to show my gratitude to Christos Strydis from Erasmus Medical Centre and Pekka Jääskeläinen from Tampere University for their continued support and interest, and especially their constructive feedback on the thesis.

Besides all the academic support, I would also like to thank my family and friends, who provided me with abundant emotional support. I received unlimited inspiration and motivation from my dear mother, father, and brother, without whom this work would never have been completed.

Glossary

ADF architecture definition file. 13, 31

ALU arithmetic logic unit. 21–23, 25, 52, 58, 59, 64

ANN artificial neural network. 15, 18, 26, 28, 66, 67

ASIC application-specific integrated circuit. 7, 9, 11, 12, 20, 67

ASIP application-specific instruction set processor. 7–9, 11–13, 16–21, 25, 28, 44, 65–67

CISC complex instruction set computer. 10, 16, 19, 21, 67

CLI command-line interface. 13

CNN convolutional neural network. 15–18, 28, 43, 44, 47, 54–56, 65–67

COTS commercial off-the-shelf. 17

CPLD complex programmable logic device. 20

CRC cyclic redundancy check. 49, 54, 55, 63, 65

DSP digital signal processor. 7, 10, 11, 17, 19, 21, 23–25, 66, 67

ECG electrocardiograph. 18, 21

EDP energy-delay product. 61, 65

EEG electroencephalograph. 17

EEMBC embedded microprocessor benchmark consortium. 15, 26, 27

ESN echo state network. 18

FPGA field programmable gate array. 12, 20, 58, 59, 65–67

FPU floating point unit. 42

FSM finite state machine. 20

FU functional unit. 7, 10–13, 21–23, 25, 29, 32, 48

GCN global clock network. 29

GPP general-purpose processor. 7, 11, 20

GUI graphical user interface. 13

HDB hardware database. 13

HDL hardware description language. 12–14, 20

IDF implementation definition file. 13

ILP instruction-level parallelism. 10, 21, 64, 67

IMD implantable medical device. 7–10, 15–21, 24–28, 43, 63, 65–67

IoT internet of things. 15

IPC instructions per cycle. 49, 55, 58, 63–65

IRF instruction register file. 25, 59

ISA instruction set architecture. 10, 11, 18, 21, 22, 25, 66

LE logic element. 29, 64, 67

LSU load-store unit. 21, 25, 58, 64

LUT lookup table. 44, 64

MCU microcontroller unit. 7, 20, 25

MOSFET metal-oxide-semiconductor field-effect transistor. 7

NOP no operation. 32, 47–49, 51, 55, 64

PCB printed circuit board. 17

ProDe Processor Designer. 13, 25

ProGe Processor Generator. 13, 14, 31

RBFNN radial basis feedback neural network. 18

RISC reduced instruction set computer. 10, 16–22, 25, 67

RNN recurrent neural network. 15, 16, 18, 28

SDK software development kit. 12

SNN spiking neural network. 18

SoC system-on-chip. 11, 15, 17, 26

TAP test access port. 40

TCE TTA-based Co-Design Environment. 7, 12–14, 21, 24, 25, 31, 35, 40, 42

TTA transport-triggered architecture. 7, 9–13, 18, 19, 21, 24, 25, 66, 67

ULP ultra-low power. 8, 15, 26

VHDL very high speed integrated circuit hardware description language. 12, 13, 20

VLIW very long instruction word. 10, 19, 21–23, 25, 66, 67

Contents

Chapter 1

Introduction

Ever since the success of the first MOSFET-based computers in the late 1950's, they reduced in both size and power following the famous Moore's Law. This development has enabled computers to transition from the initially massive, immovable machines to the portable devices we now use in our daily lives. One branch of computing has flourished exceptionally due these developments; embedded systems.

Embedded systems have started out simple control systems like in the Apollo project. As their performance increased and their size and power consumption diminished, embedded systems have become less constrained and are now closer resembling traditional computers. Application fields which at first were exclusive to application-specific integrated circuits (ASICs) or specialised digital signal processors (DSPs) are now largely dominated by general-purpose processors (GPPs) running Linux or Android. One field which is not as quickly to adapt modern advancements is medical implants. This is not necessarily because the medical sector is slower to adapt new technologies compared to other sectors, but more due to the fact that implantable devices have exceptionally strict constraints. The factor that constrains implantable devices the most is power usage. As the implant must be small in size, there is limited volume available for the battery to store energy in, and charging or replacing batteries is an activity that is desired to be infrequent. Although current solutions in implantable devices do employ the use of general-purpose microcontroller units (MCUs), there is much to gain with a more specialised approach.

1.1 Context

In 1991 the department of Electrical Engineering of the TU Delft used transport-triggered architectures (TTAs) to design application-specific processors as part of the MOVE framework [1]. In contrast to common architectures like RISC, TTA processors expose the data path of the processor to the programmer. This enables TTA processors to minimise the amount of data transfers and thus improve the energy efficiency of the processor. This property makes TTAs interesting for implantable medical devices (IMDs). The MOVE framework was the inspiration for TCE [2, 3], developed by the Customized Parallel Computing group of Tampere University. TCE is a toolset which both the hardware and software for specialised TTA processors can be developed. The instruction set, data path, and functional units (FUs) of the processors can be designed with the toolset, which makes TCE a toolset for application-specific instruction set processor (ASIP) design [4].

Meanwhile, Erasmus Medical Centre investigates low-power processor architectures for deployment in medical devices. They are looking for new processor architectures that fulfill their computational and power requirements, which creates an opportunity for an application of ASIPs. To assess performance of implantable processors, ImpBench [5, 6] was developed. ImpBench is a benchmark suite designed specifically to assess a processor's ability to handle implant-type workloads.

This thesis will assess the performance of computer architectures for ASIPs for use in IMDs. One of the architectures assessed is TTA. The aim of this research is to provide insight on the choice of base architecture for ASIPs in IMDs.

1.2 Challenges

The problem in IMDs is the constant need for processors cores with higher power efficiency, in order to decrease the frequency of surgical operations. The use of ASIPs may provide a solution to this problem.

In order to assess the performance of ASIP architectures for medical implantable processors, a proper experimental setup needs to be created which can accommodate for the different ASIP architectures. The implementations for the selected ASIP architectures should be suitable for small ultra-low power (ULP) processors. With the implementations selected, the performance of the processors can be compared using benchmarks to resemble the workload of IMDs. A comparison such as this is highly prone to pitfalls that may heavily influence the results of the research. The choice of architectures, their implementations, the simulated workloads running on these implementations, and the extraction of power and performance figures is non-trivial and requires insight in multiple levels of computer architecture.

1.3 Problem statement

The core question that arises is: *How do different ASIP computer architectures compare for medical implantable devices?* The following sub-questions arise from the problem statement:

- *Which benchmarks need to be run to represent the workloads of IMDs?*
  The workloads of IMDs need to be identified, and suitable benchmarks that represent these workloads need to be selected.
- *Which processor architectures are currently prevalent in IMDs?*
  The state of the art in processor architectures of processor-based IMDs will be reviewed in to provide insight in the selection of ASIP architectures.
- *Which low-power implementations of computer architectures are suitable for ASIPs?*
  A selection of ASIP implementations of the relevant architectures will be made. The benchmarks that represent the IMD workloads will run on these implementations, of which the performance can be measured.

In order to answer the question that lies at the core of the problem, the sub-questions listed above will first be answered in the next chapter.

1.4 Thesis outline

To start off, some background information and related work (Chapter 2) will be provided about IMDs, the competing processor architectures, and design and development of these architectures. The state of the art of IMDs will be assessed, which will provide insight on the architectures used and the workloads which they run. This background knowledge will provide the reasoning behind the selection of architectures, their implementations, and benchmarks used to simulate the workloads in Chapter 3. The selected processor implementations and benchmarks will be implemented in Chapter 4, providing insight in the design choices made in this process.

In the Results chapter (Chapter 5) the setup will be explained for simulations and power measurements, the resulting figures will be presented and discussed. The thesis will conclude in Chapter 6, which will summarise the findings in this thesis, formulate a concise answer on the research question, and provide recommendations for future work.

Chapter 2

Background

To get a better understanding of the problem, relevant topics for this thesis will be discussed first. The application domain of IMDs will be described, as well as their characterising workloads. Some required background for the implementation regarding ASIPs, ASIP development, and TTAs will follow, concluding with information about benchmarks and related work.

2.1 Processor-based implantable medical devices

An IMD is an active medical device used on humans or other animals for diagnostic, drug administration, stimulation, actuation, or other medical purposes. In the past, the majority of IMDs have been based on ASICs to comply with strict reliability and power requirements. Advancements in silicon process technology have relaxed the power and size constraints and enabled the use of a more generalised, structured design approach [5] using off-the-shelf microcontrollers.

IMDs come in many shapes and applications. Implantable devices which are or could be equipped with an embedded processor can be grouped into two categories [7]:

1. Monitoring of physiological data, for example:
    - Blood monitoring [8]
    - Tissue bio-impedance [9]
    - Neuron recording [10]
    - Brain imaging [11]
    - Brain interfacing [12]
    - Seizure detection [13]
2. Stimulation or actuation of tissue, for example:
    - Neurostimulation [14]
    - Neuromodulation [15]
    - Vestibular prosthesis [16]
    - Cortical control [17, 18]
    - Pacemakers [19]
    - Implantable defibrillators [20]
    - Corneal stimulation [21]
    - Bladder control [22]
    - Cochlear implants [23, 24]

These IMDs typically acquire data in some form, and have an optional signal-processing task for actuation.

The batteries for the IMD are usually limited to 1-3Ah [25]. For IMDs without actuators, the battery capacity in combination with a minimal lifetime of the IMD of 5 years forces the average working current of the device down to $23 - 65\mu A$ [26]. Thus generally all IMDs are optimised for low power. Many of these devices could be based on the same low power processor core, with the only variable being the actuator or sensor. For devices which do sink large currents in actuators like neurostimulators, the power consumption is less important as the battery has to be charged frequently.

2.2 Relevant processor architectures

In order to provide a better understanding for processor architectures prevalent in IMDs and their alternatives, relevant processor architectures will be discussed in this section.

2.2.1 RISC

In contrast to a complex instruction set computer (CISC), where the focus lies on reducing instruction count, a reduced instruction set computer (RISC) is focused on reducing the complexity of the instruction set. Reducing complexity in the instruction set is not only about reducing the number of instructions in the set, but also reducing the complexity of the hardware required to decode and execute the instructions. Although this does automatically lead to more instructions being executed, the reduced hardware complexity allows for higher clock frequencies and less area usage than a CISC.

RISC architectures are often load-store architectures, where the load and store instructions copy data between memory and general-purpose registers, and all arithmetic is executed on the registers. Another trait to achieve simplicity is fixing the length of the instructions and the locations of the operators and data in the instructions.

Because of the small area, RISC is popular amongst microcontrollers, and thus is also prevalent in IMDs. Popular modern implementations of RISC instruction set architectures (ISAs) are ARMv6, ARMv7, and RISC-V.

2.2.2 DSP

DSPs are microprocessors with specialised architectures for signal-processing workloads, like filters. A typical DSP instruction is the multiply-accumulate instruction, which can perform a multiplication and addition in one instruction. This operation is heavily used in typical DSP workloads, like Fourier transforms. As signal-processing workloads often require processing large amounts of data, a DSP typically allows for loading multiple data from memory in one instruction. Some implantable devices that focus on a heavy signal-processing task, like hearing aids or neurological analysis, feature a DSP [27, 28].

2.2.3 VLIW

VLIW processors are characterised by their long instruction word which can issue instructions to multiple FUs in the processor at once. The instruction word contains multiple issue slots in which RISC-like instructions are placed, which allows a VLIW core to exploit instruction-level parallelism (ILP) by design. This makes them also suitable for DSP applications. An example of a VLIW DSP is the C6000 series of Texas Instruments [29].

2.2.4 Transport-triggered architectures

A TTA is a one-instruction set VLIW architecture. As opposed to traditional architectures where the instruction specifies the operation, a TTA instruction defines the movement of data over the transport buses in the processor. The FUs do not have direct access to the register file, as it is also only accessed via the buses. This reduces the complexity of the register file, and reduces the traffic through the register file [30].

Operations are executed as a consequence of the movement of data over the buses, and are triggered by writing to a specific port for each FU. As opposed to DSP, VLIW, and RISC architectures, the result of a FU is not written to the register file, but instead remains in the register at the output of that FU until it is moved. The exposed internal data path allows for explicitly bypassing the register file, but may also require more instructions per operation. For example, `r3 = r0 + (r1 << r2)` in a RISC ISA could be:

```
slli r1, r1, r2
add r3, r0, r1
```

Whereas for a TTA (with a shifter and adder) it would become:

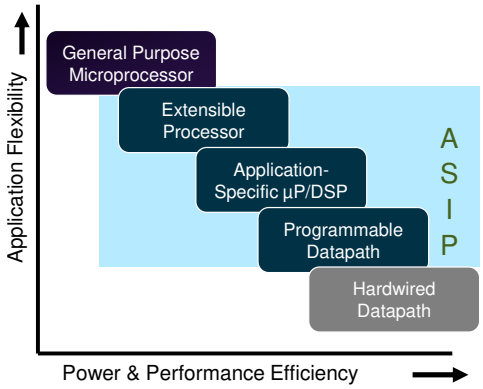

Figure 2.1: A visualisation of the trade-off of ASIPs in terms of flexibility and performance (Source [39]).

```
move r1, sll.in0
move r2, sll.in1t
move r0, add.in0
move sll.out, add.in1t
move add.out, r3
```

Here `sll` is the shifter and `add` is the adder, and the `-t` postfix indicates the port that triggers the operation.

A TTA can contain multiple buses, each of which can be connected to a subset of all FUs. Each bus is then typically controlled by their own slot in the instruction word. This also implies that multiple moves can be executed in a single instruction, thus providing instruction-level parallelism. Because the data path is so exposed and every movement of data is specified by the instructions, the compiler has more control over the processor and is able to optimise more compared to compilers for traditional architectures at the cost of added complexity.

The downside of a TTA is that context switching (and thus interrupts) are very hard to implement. This is because the state of the processor is not only described by the register file but also by the current state of each FU. The lack of context switching renders the TTA unsuitable for multithreaded or interrupt-driven applications, but it remains suited for always-on applications [31].

TTAs have mostly been implemented as solutions for low-power systems with accelerators for neural networks [32], Fourier transforms [33], audio processing [34], but also for low-power signal-processing [33] and always-on [35] applications.

2.3 ASIP

An ASIP is the trade-off between ASICs and general-purpose processors [36]. ASIPs are usually small processors with a tailored instruction set for the application and are usually deployed in an environment that forces such requirements [37]. This tailored ISA provides greater computational efficiency and requires less power for the specified workload.

ASIPs may be implemented by means of extending an existing/basic ISA with specialised instructions, or by adding reconfigurable hardware in the processor [38].

ASIPs can be deployed as standalone application-specific processors like DSPs, but are also prevalent in system-on-chips (SoCs), where they fulfill roles as programmable accelerators.

ASIP development

Because the compiler for the ASIP is dependent on the ISA and hardware design of the ASIP, ASIPs are developed in a so-called co-design environment in which both the hardware and software for the ASIPs are developed. This is the most significant difference in development for ASIPs compared to GPPs, as the ISA and its implementation are not part of the designs space when developing for a GPP.

The “leading tool solution” [40] in the industry is Synopsys ASIP Designer, and provides a highly

automated workflow. An open-source alternative is the TTA-based co-design environment (TCE), developed by the customized parallel computing research group of Tampere University.

ASIP Designer

ASIP Designer is a tool suite developed by Synopsys [40]. ASIP Designer focusses on a wide range of ASIPs, from extendible processors to programmable data path ASIPs.

Processors are described using ASIP Designer's proprietary language nML. nML is a C++ style language in which both the structure (storages, FUs, connections) of the processor and the instruction set are described. An example of nML is shown in Listing 2.2. Another proprietary language PDG, which is based on C, then describes the behaviour of the FUs, IO interfaces, and processor control unit. An example of PDG is shown in Listing 2.1.

With nML and PDG, the entire ASIP is described, and ASIP Designer can generate an SDK including C/C++ compiler (called Chess), and debugger/simulator (Checkers) based on that model. The hardware description language (HDL) description of the hardware can also be directly generated from the processor model using Go in both Verilog and VHDL. The Chess compiler supports the LLVM front-end Clang, which can provide more aggressive optimisation. Along with the program written in C or C++, the generated simulator can produce the memory contents for all defined memories in the processor.

As shown in Figure 2.2, the development cycle in ASIP Designer is first focused on designing the algorithm and the processor model. The processor model, which includes the description of the instruction set, is used to generate a semi-custom SDK for the processor. The SDK is then used to implement and test the algorithm, and the simulations in the SDK provide insight for refinement for the processor model. To assist in this process, Checkers provides an overview of all registers and memories in the processor and allows for stepping through the source code and single instructions. The placement of breakpoints and watchpoints in both the assembly and C is also supported. After simulation is successful and the hardware/software co-design is completed, the HDL can be generated and exported to other Synopsys tools for ASIC or field programmable gate array (FPGA) synthesis. Finally, automatically generated test programs can verify the integrity of the processor.

For further understanding of the workings of the tools of ASIP Designer, it is recommended to reference the online training program [40] and the ASIP Designer manuals, which come with the software.

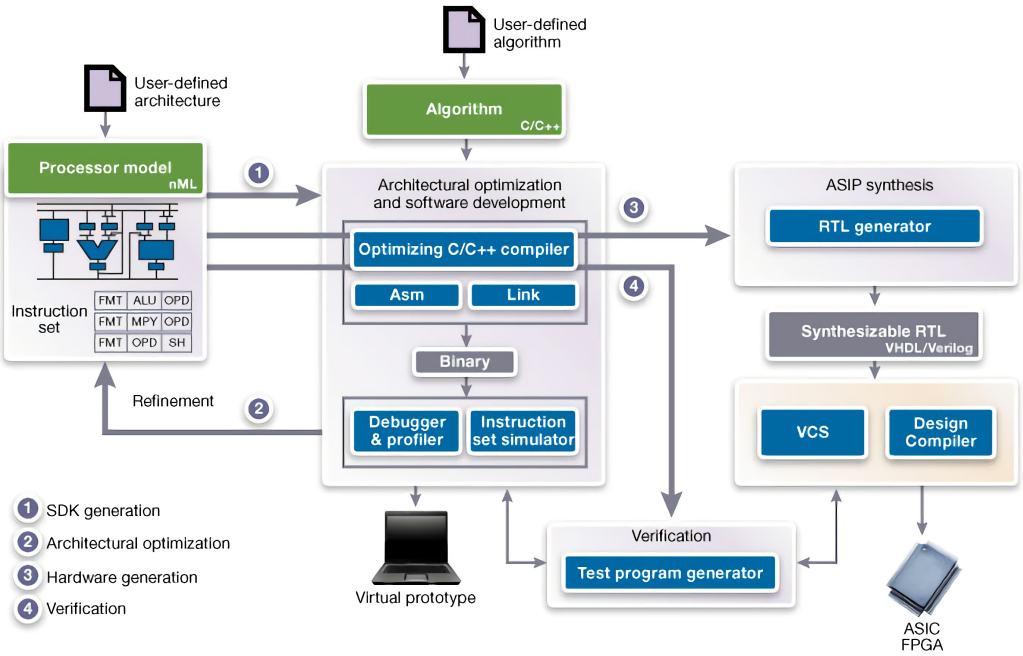

Figure 2.2: Workflow in Synopsys ASIP Designer (source [40])

```
word add(word a, word b) {
    return a + b;
}
```

Listing 2.1: An example of the behaviour `add` instruction as described with PDG.

```
enum stage_names {IF,   // Instruction Fetch
                  DE,   // Instruction Decode and Execute
                  WB};  // Writeback
mem PMb[2**14] <uint8,addr> access {};

enum eR { x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15};
reg R[16]<w32,uint4> syntax (eR) read(r1 r2) write(w1 wd);
```

Listing 2.2: A modified excerpt of an nML processor model of a Z-scale RISC-V processor. The pipeline stage names, program memory, and register file is described.

TCE

TCE [4] provides an open-source ASIP development toolset for developing ASIPs with transport triggered architectures. TCE is the successor of the MOVE project [41].

Figure 2.4 shows the workflow for designing an ASIP with TCE. The designer creates the processor model in the TTA Processor Designer (ProDe), in which FUs are created and connected by transport buses. The operations of the FUs are selected from the operation set abstraction layer database, which contains the static properties and behaviours of the operations. The behavioural models of the FUs are then described by a hardware database (HDB) file, which contains HDL descriptions of the blocks. The links between the architecture definition file (ADF) and HDB file are described in the implementation definition file (IDF) file. Figure 2.3 shows ProDe with a minimal TTA using one bus connected to some FUs. ProDe describes the processor model in an XML file called the ADF.

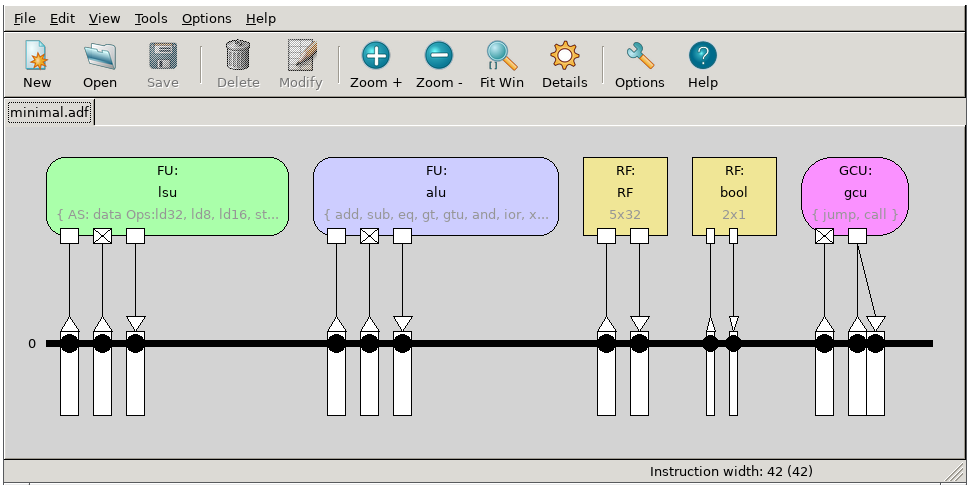

Figure 2.3: A minimal TTA architecture (provided by TCE) as shown in ProDe.

The compiler, `tcecc`, uses the generated ADF to compile programs written in a high-level language (C, C++, or OpenCL) for the specified architecture using the LLVM front-end Clang. The designer can then use `ttasim` (CLI) or `proxim` (GUI) to simulate the program on the processor and generate profiling information. This information will provide insight for new design iterations on the hardware and software.

Finally, the Processor Generator (ProGe) will be used to generate the hardware description of the processor in Verilog or VHDL. The ADF, IDF, and compiled binary, provide all necessary information

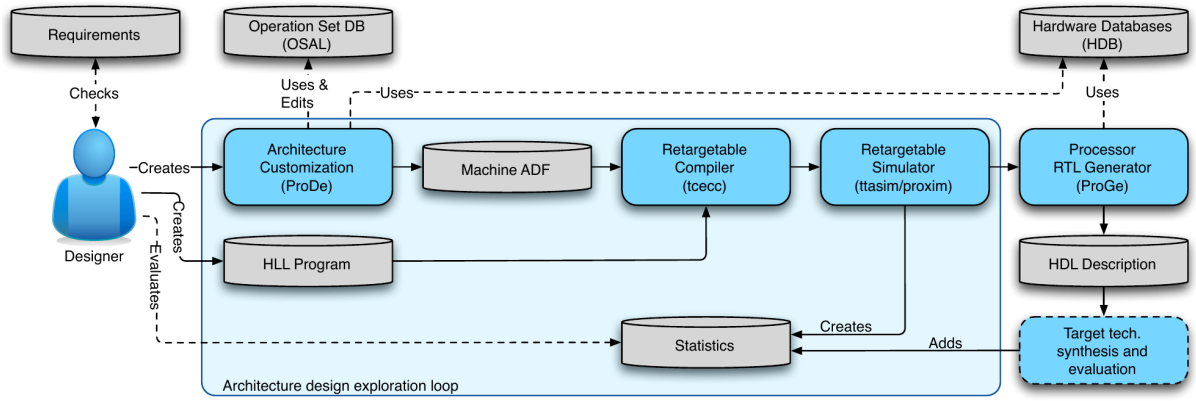

Figure 2.4: Workflow in TCE (source [4])

for ProGe to generate the HDL.

ProGe also supports generating HDL directly for specific platforms like the Altera Stratix II device family. To generate the memory contents of the processor, the command-line utility `generatebits` can be used, which supports multiple output formats.

For more information on TCE it is recommended to refer to the TCE homepage [4] and the book *Computing Platforms for Software-Defined Radio* [2].

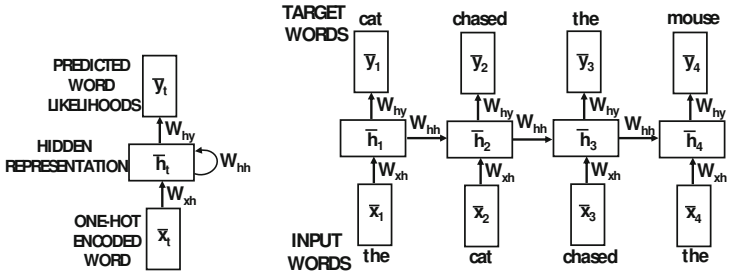

Figure 2.5: A simple recurrent neural network architecture for predictions of words, and its time layered representation (Source: [42]). W_{xh}, W_{hy}, and W_{hh} are the weight matrices. $\bar{x}$, $\bar{h}$, and $\bar{y}$ are the input, hidden state, and output vectors.

2.4 Benchmarks

The performance[1] of a single-threaded processor can be evaluated by using the *Iron Law of Performance* [43]:

$$T = N \cdot CPI \cdot \frac{1}{f} \tag{2.1}$$

Here T is the execution time, N is the number of total instructions for the program, CPI is the average clocks per instruction, and f is the clock frequency.

The maximum clock frequency is only dependent on the processor architecture and its implementation, but the CPI and number of instructions is dependent on both the architecture and the program. Simulations of programs on processors in ASIP Designer and TCE can provide these figures, but the results may vary wildly per supplied program.

Benchmarks are programs designed for assessing the performance of a processor for a specific workload by running algorithms typical for that workload. Benchmarks typically consist of multiple algorithms that characterise the workload. Many general-purpose benchmarks exist to compare general-purpose processors, even for embedded systems. However, when designing a system for a single application

[1] Performance is defined here as total execution time of a defined program on the processor.

(which is the case for IMDs and embedded systems in general), such general-purpose benchmarks may provide a distorted view of the performance for the application the system is designed for. It follows that the characterisation of the workload and the identification of benchmarks that represent that workload is important in comparing performance of processors in IMDs.

2.4.1 EEMBC

The embedded microprocessor benchmark consortium (EEMBC) is a consortium led by many market leaders in the field of embedded processors, like Arm, Texas Instruments, ST Microelectronics, and Silicon Labs. The EEMBC aims to develop "industry-standard benchmarks for the hardware and software used in autonomous driving, mobile imaging, the Internet of Things, mobile devices, and many other applications" []. Their most famous benchmark CoreMark is indeed the industry standard benchmark for embedded microcontrollers, and thereby the successor of Dhrystone. As ULP SoCs with integrated wireless radios are becoming more relevant for IoT applications, EEMBC has also released benchmarks for those application fields, such as IoTMark and ULPMark.

2.4.2 ImpBench

ImpBench is a benchmark suite developed specifically to benchmark biomedical implantable processors. It is the only benchmark specifically developed for IMDs. ImpBench was proposed in 2008 [5], with a revision in 2010 [6]. The benchmark suite contains eight benchmarks for four workloads; compression, error detection, encryption, and synthetic benchmarks which simulate real-life applications. The following benchmarks are included:

- MiniLZO, a light-weight variant of the LZO high-performance lossless compression library. LZO is in itself a variant of the LZ77 data compression algorithm. It implements zlib (RFC 1950 [44]) and Deflate (RFC 1951 [45]) for compression and decompression. In the benchmark, only compression is used.
- Finnish, the Finnish submission for the Dr. Dobbs compression contest in 1991 [46]. Also based on the LZ77 algorithm.
- MISTY1, a block cipher developed in 1995 (RFC 2994 [47]) which was recommended by CRYPTREC in 2003. The cipher has been broken in 2015 using integral cryptanalysis.
- RC6, a block cipher developed in 1998 with a small code size.
- CRC32, a 32-bit cyclic redundancy check algorithm for error detection.
- Checksum, a simple checksum algorithm with 32-bit summation and inversion. The algorithm folds the 32-bit sum over into a resulting 16-bit checksum.
- Motion, a synthetic benchmark which simulates a motion sensor. The benchmark simulates the functionality of a motion sensor by generating a sensor sample, comparing it against a threshold, and then sleeping the CPU by running a while-loop.
- DMU, a synthetic benchmark which simulates a drug delivery and monitoring unit based on the work of Cross et al [48].

2.5 Artificial neural networks

Artificial neural networks (ANNs) are a form of machine learning based on the learning mechanism of biological nervous systems. The similarities lie in the fact that the connections between biological neurons can be strengthened by external stimuli, which causes the system to learn [42]. Two types of ANNs will be reviewed; convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

2.5.1 Convolutional neural networks

CNNs are layered feed-forward networks used frequently in computer vision for image classification and object detection. CNNs have been the most successful of all types of neural networks and outperform humans in image classification [42]. As the name implies, a CNN makes use of convolution layers to map the outputs of the previous layer onto the input of the next layer. The output of every node in the layer is then given by convolution of the connected outputs of the previous layer and the weights of these connections. The convolution layers are interleaved with pooling and activation layers. The activation layer maps each output of the convolution layer on its activation function. The purpose of the activation

function is to map the values from the convolution layer into a binary class label. The aim of the pooling layer is to reduce the size of the network by sub-sampling the outputs of the previous layer.

A simple CNN architecture can be seen in Figure 2.6, which is designed to classify handwritten digits in images.

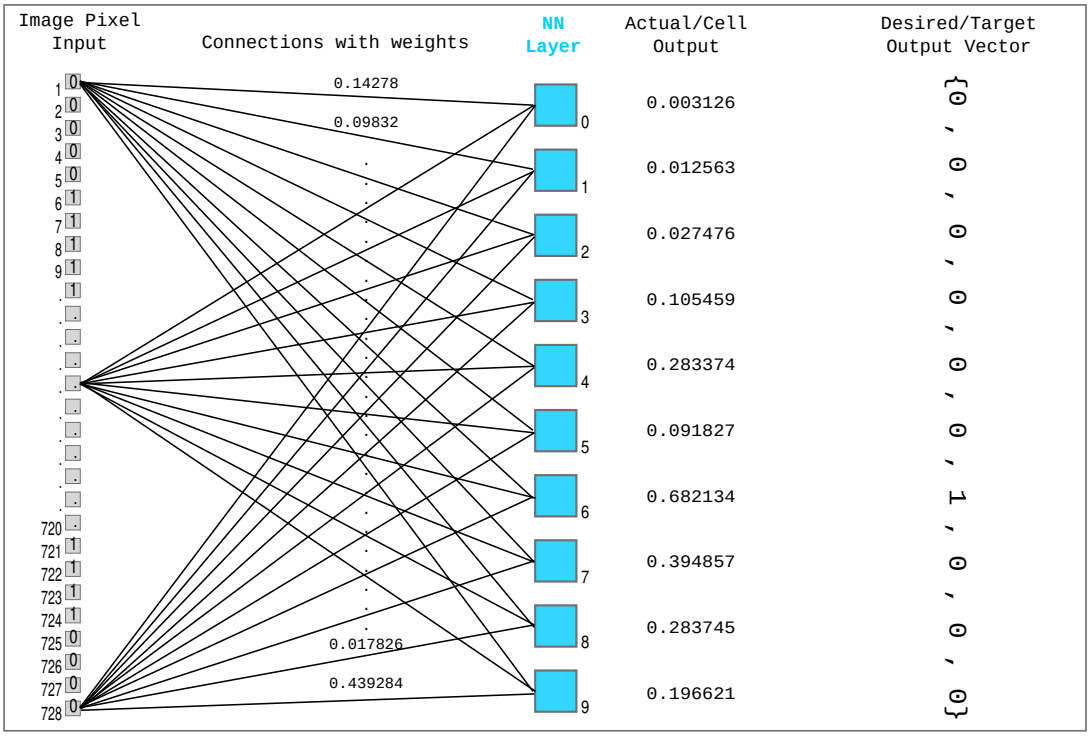

Figure 2.6: A simple convolutional neural network with one convolution layer for classification of handwritten images in the MNIST dataset (Source: [49]).

2.5.2 Recurrent neural networks

RNNs are used on sequential data like time series and text. To act on the temporal dimension, an RNN stores previously received samples in a hidden state in its network. New samples are then processed using the current state, and the current state is updated with the latest sample. This way the nodes in the network have two variables; the new sample and the internal state of the node. When the RNN acts on a fixed time window, the RNN can be unfolded in time. This representation is shown in Figure 2.5.

2.6 Related work

This section presents references to existing background work related to processor-based IMDs, their workloads, and ASIPs.

2.6.1 Processor-based IMDs

Many IMDs using processor cores have been described in academic work, of which some will be highlighted. [50] and [51] describe implantable devices for stimulation of denervated muscles, both controlled by a 16-bit PIC with a RISC architecture [52, 53]. The muscles are periodically stimulated with short electrical pulses. An intravaginal drug delivery unit is described in [48], which is controlled by a 16-bit Mitsubishi[2] M16C series microcontroller with a CISC architecture [54]. It features wireless communication and a gascell for actuation of the pump. A system for biological signal recording is described in [55] which features bi-directional wireless data transmission. The module is implanted on a rabbit to measure electroneurogram signals from the sciatic nerve and uses a PIC18F452 (16-bit RISC) as controller. The

[2]Now called Renesas

work in [19] describes software-related energy estimation of a pacemaker controlled by a Texas Instruments MSP430F1611. The MSP430F16 is a microcontroller with a 16-bit RISC architecture featuring analog-to-digital and digital-to-analog converters [56].

An electrocardiograph monitoring SoC is described in [27], which contains hardware signal processing blocks and a 32-bit ARM Cortex-M0+ (RISC) core for the arrhythmia detection algorithm. [57] implements a Texas Instruments MSP430FR5949 (16-bit RISC [58]) for data logging of temperature measurements in animal brains. A wireless bladder pressure monitor IMD is described in [59], which uses a Texas Instruments MSP430FR5994 (16-bit RISC [60]) and a MicroSemi ZL70123 transceiver module. Another MSP430FR5994 is used to run a seizure detection algorithm on electroencephalograph (EEG) signals based on a CNN in [61, 62]. [63] uses two ARM Cortex-M0+ cores for brain-machine interfacing; one to run the brain-machine interface algorithms and one to execute firmware updates. [28] implements a custom 16-bit DSP for speech comprehension in a cochlear implant.

Table 2.1 shows the referenced publications above with their categories, processors, and power consumption.

Table 2.1: An overview of publications of processor-based IMDs showing their application, processor, and power consumption. Note that the lifetime contains a safety margin for most listed IMDs.

Paper	Year	Application	Category	Processor	Avg. power consumption	Battery lifetime
[50]	2002	Electrical stimulation of denervated muscles	Neurostimulation	8-bit Microchip PIC16C54C-04	0.92mW	5 weeks
[48]	2004	Intravaginal drug delivery	Drug administration	16-bit Mitsubishi M16C	0.24mW	1 month
[51]	2005	Electrical stimulation of denervated muscles	Neurostimulation	8-bit Microchip PIC16F874	1.0mW	12 weeks
[55]	2005	Electroneurogram signal recording	Neuron recording	16-bit Microchip PIC18F452	90mW	-
[28]	2006	Speech comprehension enhancement	Cochlear implants	Custom 16-bit DSP	1.8mW	-
[19]	2008	Pacemaker	Neurostimulation	16-bit TI MSP430F1611	12µW	24 years
[27]	2014	Arrhythmia diagnosis	Neuron recording	32-bit ARM Cortex-M0+	64nW	5 days
[59]	2016	Wireless bladder pressure monitoring	Bladder control	16-bit TI MSP430F1611	3.8mW	25 days
[61, 62]	2018	Seizure detection	Neuron recording	16-bit TI MSP430FR5994	850µW	-
[63]	2019	Brain machine interface	Brain interfacing	2x 32-bit ARM Cortex-M0+	150mW	2 days

Regarding power consumption and battery life, it is heavily dependent on the frequency of use of wireless transmitters and actuators. When the ratio of the time either of these functions are on is very low, the average power consumption is dominated by the power consumption of the microcontroller [59]. If this is not the case, the use of an ASIP would not significantly benefit the battery life of the IMD.

The IMDs that have space for a battery and PCB can use a commercial off-the-shelf (COTS) processor in a standard package. Most works listed use a processor with a RISC architecture, which is common for COTS microcontrollers. The IMD ([27]) that can not accommodate a PCB uses wireless charging of a tiny battery and a fabricated SoC with all functionality, using a Cortex-M0+ core. This IMD uses a fixed hardware signal-processing front-end, which is not programmable, and thus makes the device less flexible. An implementation with an ASIP with DSP functionalities would improve the programmability of the device.

The benchmarks in ImpBench cover most of the workloads for the IMDs listed in Table 2.1, except for the neural network used in [61, 62].

2.6.2 ASIP

ASIPs can be developed by developing an application-specific instruction set from the ground up, or by extending an existing instruction set with application-specific instructions. For the latter method, extending available RISC instruction sets like RISC-V and MIPS are popular choices, and improvements can be measured compared to the original instruction set. Depending on the application, the ASIPs can achieve significant improvements in computational and energy efficiency. [64] and [65] show that algorithms in IMDs can also benefit significantly from specialised instructions.

Table 2.2 shows an overview of implementations of ASIPs, their architecture, and the performance gained.

Table 2.2: An overview of ASIP based on existing architectures, showing the computational and/or energy efficiency improvements of the extensions.

Paper	Year	Application	ISA	Exe. cycles improvement	Energy efficiency improvement
[64]	2005	Motion estimation in video	VLIW	-	2-2.5x
[66]	2005	Networking applications	MIPS extension	2x	2x
[65]	2009	Cardiac beat detection	16-bit RISC	5.3x	2.2x
[67]	2012	Compression of sparse signals (ECG)	16-bit RISC	62x	11.6x
[68]	2018	H.265/HEVC deblocking filter	RISC-V extension	1.07-1.11x	-
[69]	2020	RNN-based 5G radio resource management	RISC-V extension	10x	15x
[34]	2016	Binaural speaker localisation	TTA	151x[3]	-

2.6.3 Neural networks in IMDs

ANNs can be used to extract patterns or trends with less computational complexity than traditional algorithms [70]. Table 2.3 shows an overview of publications that employ ANNs in IMDs. The table shows that ANNs are mostly used in IMDs for detection or prediction of events. Most applications use either a CNN or an RNN.

Table 2.3: An overview of neural networks employed in IMDs. The number of layers includes the input and output layers.

Paper	Year	Application	NN type	Layers	Activation function
[71, 72]	2006	Seizure prediction	RNN	4	Sigmoid
[73]	2010	Prediction of tremors	RBFNN	3	Gaussian
[74]	2012	Brain-machine interfacing	RNN ESN	Unknown	Unknown
[75]	2013	Brain-machine interfacing	SNN	-	-
[76]	2016	Speech enhancement	CNN	4	ReLU
[61, 62]	2018	Seizure detection	CNN	10	Dropout, sigmoid
[77]	2018	Security of an insulin pump	RNN	Unknown	Tanh, softmax
[78]	2019	Detection of attack patterns	RNN	4	Linear
[79]	2020	Abnormal ECG beat detection	CNN	8	ReLU

Chapter 3

Alternative solutions

There are many viable solutions for implementing the functionality of an IMD. This chapter aims to provide an overview of these solutions and compare them. The comparison will form the basis for selecting the most promising solutions, which will be implemented in the next chapter.

To acquire performance metrics of the implemented cores, benchmarks will have to be run on the selected cores. Suitable benchmark suites will be presented, of which some will be selected to run on the cores. Finally, techniques will be discussed to obtain area and power numbers.

3.1 IMD platforms

An classification of solutions that will be reviewed can be found in Figure 3.1.

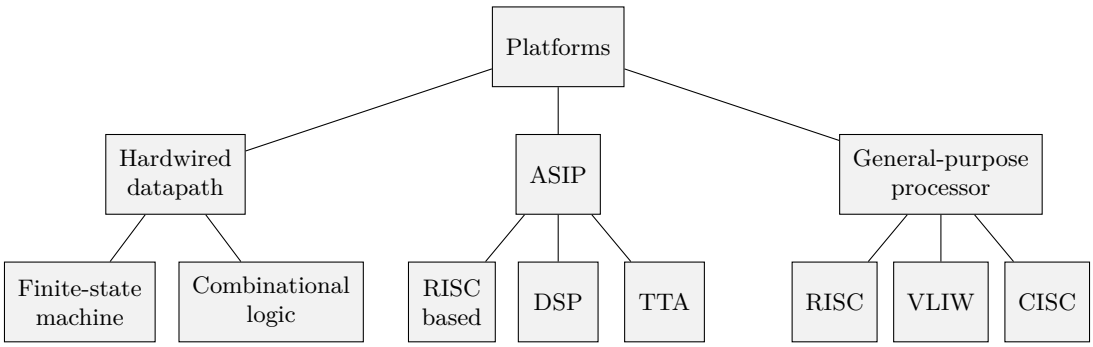

Figure 3.1: A tree representing a subset of possible solutions to use as compute platform in implantable devices.

3.1.1 Requirements

The background in Chapter 2 produces the following requirements for IMDs:

- High reliability
- Real-time performance (no deadline misses)
- Low peak and average power consumption
- Low area use
- Low design effort

For IMDs the most important metric is reliability. The requirement of reliability in combination with real-time performance implies that he IMD can not miss any deadlines. Context switching degrades the reliability of processors, as it is harder to provide guarantees on meeting deadlines and thus is something to avoid in IMDs.

The increasing efficiency of silicon technology benefits most of the solutions in terms of area and power consumption, thus there are multiple viable options that will be discussed.

3.1.2 Hardwired datapaths

The entire system could be described in a finite state machine (FSM) or combinational logic with a HDL such as Verilog or VHDL. The hardware description could be synthesized directly on an ASIC, complex programmable logic device (CPLD), or on an FPGA . This approach scores very high in terms of reliability, real-time performance and power consumption, but the design process can be very tedious compared to traditional programming of general-purpose processors. Due to the system being designed for one specific purpose, the solution is very inflexible.

When combinational logic or an FSM is implemented in an ASICs, it usually outperforms any other solution. However, updates or changes to the design requires new expensive research and development, and old implementations can not be updated as the ASIC is not programmable.

Although CPLDs and FPGAs can be reprogrammed, they still lack flexibility in terms of design effort. A small change to a complex algorithm in software is likely to cost less time than it would in a hardwired datapath.

3.1.3 General-purpose processors

GPPs are useful for their quick time-to-market compared to custom solutions. Especially well-known architectures like x86 or MIPS are supported by multiple compilers and debuggers that support high-level object-oriented languages, and reduce the design effort for the programmer. The inherent downside of the GPP is the higher power consumption compared to application-specific solutions.

GPP are available in MCUs, which is a popular choice for IMDs. For commercial MCUs, development boards and example programs are readily available and they are usually well documented. On top of that they are often in stock at large distributors or the manufacturer themselves, which enables large production volumes at short notice.

3.1.4 ASIP

The middle ground between hardwired datapaths and GPPs is owned by programmable application-specific processors; the ASIPs. ASIPs have the benefits and drawbacks of both worlds; development is lengthy and costly, but only has to be done once for a range of solutions, and they can be reprogrammed. ASIPs are defined by their unique instruction set, which usually contains instructions developed for the application. The tailored instruction set makes ASIPs more power efficient than GPPs, while maintaining programmability.

3.1.5 Platform comparison

The assessed platforms are qualitatively analysed for each requirement in Table 3.1. The design effort for implementations with hardwired datapaths is too high, which is the primary reason for the prevalence of GPPs in IMD. The ASIP do perform better in both power consumption and real-time performance than GPPs, but sacrifice design effort. Especially the increased power efficiency makes the ASIP an interesting choice for IMDs.

Table 3.1: Qualitative comparison of the different solution categories.

Solution	Reliability	Real-time performance	Power consumption	Area	Design effort
Hardwired datapath	++	++	++	++	−−
ASIP	+	++	+	−	+
GPP	+	−	−	−	++

3.2 ASIP architectures and implementations

The following architectures will be compared in this thesis:

- RISC
  As seen in Section 2.6.1, GPP in ASIPs are mostly RISC. RISC is not only popular amongst processor architectures in MCUs in IMDs, but also as a base for ASIPs. For this reason, a RISC architecture will be selected.

- DSP
  Some IMDs employ DSPs for audio processing in cochlear implants or signal processing of ECGs, thus it would be interesting to see how a DSP would fare in this comparison.
- TTA
  TTAs show promise as low-power ASIPs due to their inherent ILP and reduced complexity compared to a VLIW.
- VLIW
  To bridge the architectural gap between RISC and TTA, a VLIWs architecture will also be selected.

Although there are also some cases of CISC architectures to be found in IMDs [48], a CISC typically uses a lot of area and is already extended with complex instructions, so it does not form a good base architecture for an ASIP.

Popular design tools for ASIPs are Synopsys ASIP Designer and the open-source toolset TCE for TTA. From these toolsets some implementations will be presented.

3.2.1 ASIP Designer

ASIP Designer provides some example projects that implement some popular architectures that could be used to compare the TTA core against. Noteworthy examples are:

- Tmcu, 32-bit microcontroller with a Harvard architecture and 3-stage pipeline (IF/DE/EX). Supports variable length and parallel instructions. The Tmcu has 16 32-bit registers and byte addressable data memory.
- Tzscale, implementation of the RISC-V ISA with a 3-stage pipeline (IF/DE/EX). Also supports variable length instructions, and has 16 or 32 32-bit registers and byte addressable data memory.
- Tvliw, implementation of a 4-slot VLIW (2 arithmetic logic units (ALUs), 2 load-store units (LSUs)). Optional variable length instructions, predicated execution, and two stage program memory fetch.
- Tdsp, 16/32-bit DSP with three-way ILP (1 ALU, 2 LSUs). The ALU supports 16/32-bit MAC of integers and fractional numbers[1].

The Tmcu and Tzscale both feature RISC architectures, and achieve similar scores in CoreMark. However, the Tzscale consumes less area and uses a popular open-source instruction set, so the Tzscale is preferred to the Tmcu.

Tzscale

The Tzscale is an implementation of the RV32I base integer instruction set. As the RISC-V ISA does not define the pipeline architecture, Synopsys modelled the processor with a three-stage pipeline similar to the Z-scale [80]:

- IF, instruction fetch stage. Instruction is fetched from the instruction memory.
- DE, instruction decode and execution stage. Operands are loaded from the register file, the ALU, shifter, and multiplier execute their operations. Addresses and data are sent to the data memory for load/store operations. Control flow operations are also executed in this stage.
- WB, write back stage. Results from FUs from the DE stage and data from load instructions are written back to the register file.

The results from the operations in the DE stage are written back one stage later, which introduces data hazards. To prevent this data dependency hazard, the operands for the operations in the DE stage can bypass the register and fetch directly from the WB stage. One hazard that can not be circumvented by bypasses is introduced by the load instruction, of which the data is only available in the WB stage. For this hazard stalls will need to be used.

The only register file used in the Tzscale is the general-purpose register (`R` in Figure 3.2), which is 32x32-bits, with the first register hardwired to 0. The other user-visible register holds the program counter.

The data path of the Tzscale is shown in Figure 3.2. The registers and memories related to instruction fetching/decoding and control are not shown.

[1] For fractional numbers one additional left shift of one bit is necessary.

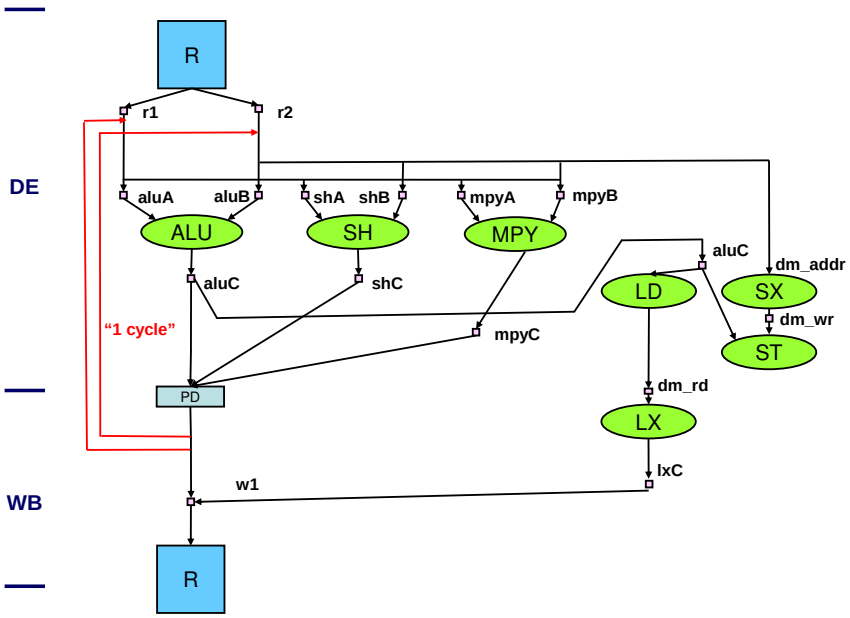

Figure 3.2: The data path of the Tzscale core (source [81]).

As can be seen in Figure 3.3, the RV32I instruction set defines four instruction types (R, I, S, and U), which are all 32-bits wide. The source (`rs1` and `rs2`) and destination (`rd`) are always in the same place in each instruction type to simplify decoding. The encoding of immediate values has the same consistency, as they are always sign-extended and packed to the leftmost bit in the field. The design focus of the ISA is hardware simplicity, as is typical for RISC.

The Tzscale also supports the use of the "C" extension for instruction compression, which can reduce a RV32I instruction to 16-bits under certain conditions.

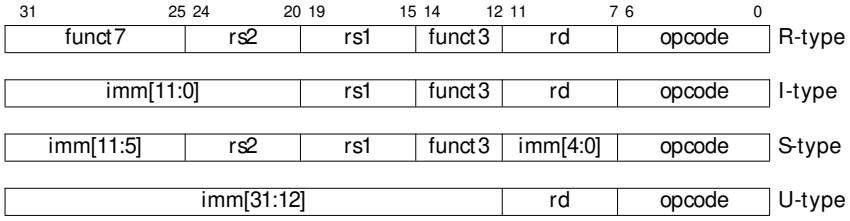

Figure 3.3: The instruction types in the RV32I RISC-V instruction set (source [80]).

Tvliw

The Tvliw is a VLIW processor with two ALUs and a three stage pipeline, with the following stages:

- IF, instruction fetch stage. The instruction is fetched from the instruction memory.
- ID, instruction decode stage. The instruction is decoded and any addresses for load or store operations are sent to the data memory.
- EX, execution stage. All arithmetic and move operations are executed.

The FUs in the VLIW are:

- `DU0` and `DU1`, two ALUs for arithmetic on data
- `AU0` and `AU1`, two units to execute operations on addresses

The ALUs, `DU0` and `DU1`, support addition, substraction, multiplication, bitwise AND, OR, and exclusive-OR, shift operations, and compare operations.

Next to the program counter, stack pointer, link register, and single bit condition register, the Tvliw contains three register files:

- `R`, general-purpose data register which can be accessed by both ALUs and both ports on the data memory.
- `P`, which holds addresses
- `M`, which holds register address modifiers

All registers and FUs have a data width of 32-bit.

The Tvliw has a 64-bit wide instruction memory (`PM`) with a 32-bit address bus and a separate 32-bit data memory (`DM`) with two read and write ports.

The data path of the Tvliw is shown in Figure 3.4. The registers and memories related to instruction fetching/decoding and control are not shown.

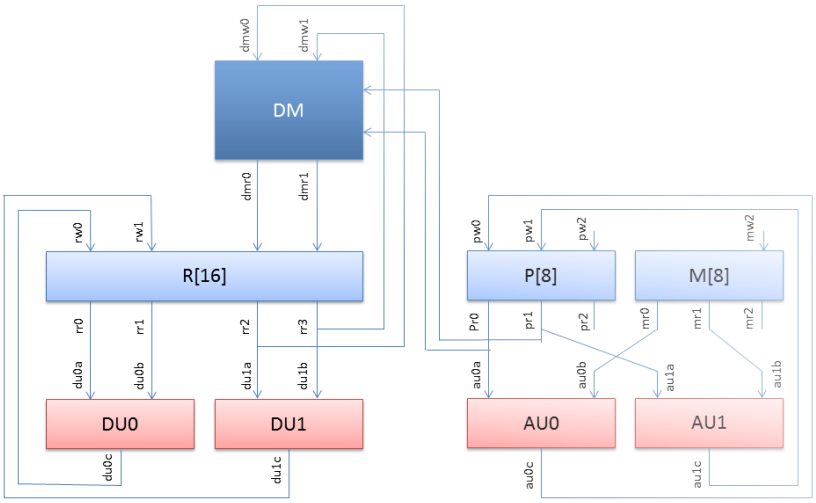

Figure 3.4: The data path of the Tvliw core (source [82]).

The Tvliw has two instruction formats, where it issues either two or four instructions. The first bit of the instruction indicates the format. For the four-slot VLIW instruction, two arithmetic operations on the `DU0` and `DU1` ALUs can be executed, and two load, store, or register moves can be executed at the same time. For the load and store instructions the addresses will be calculated by `AU0` and `AU1`.

Tdsp

The Tdsp is a DSP with a three stage pipeline, with the following stages:

- IF, instruction fetch
- ID, instruction decode
- EX, execution

The FUs of the Tdsp consist of:

- `alu`, a 32-bit general-purpose ALU
- `mpy`, a 16x16 multiplier
- `sh`, a 32-bit shift unit
- `norm`, a 32-bit norm unit, which is used to compute the shift factor for the shift unit to normalise the input
- `agu1`, `agu2`, address generation units
- `xa`, `xr`, `xs`, `xt`, sign extension or half-word extraction units
- `cnd`, the implementation of the calculation of the conditional jump bit
- `lp_incr`, the hardware loop control index update unit

The Tdsp uses the following registers:

- `R`, 8x16-bit general purpose register file
- `L`, 4x32-bit long word storage
- `A`, 8x20-bit for holding addresses
- `M`, 4x16-bit for holding address modifiers
- `SP`, stack pointer register, to store the address of the top of the stack
- `LR`, link register to store the return address of the caller function
- `BS`, `BL`, registers for holding the start address and loop count for circular buffers
- `LS`, `LE`, `LC`, `LP`, registers for loop control
- `PC`, program counter
- `SR`, status register which holds the `CND` conditional bit, `IE` interrupt enable bit, and `IMSK` interrupt mask
- `ISR`, `ILR`, interrupt registers

Finally, the memories Tdsp uses are `DM` and `PM`, the data and instruction (program) memory. `DM` can be accessed for words and long words (aliases `DMw` and `DMl`).

The data path of the Tdsp processor is shown in Figure 3.5. The registers and memories related to instruction fetching/decoding and control are not shown.

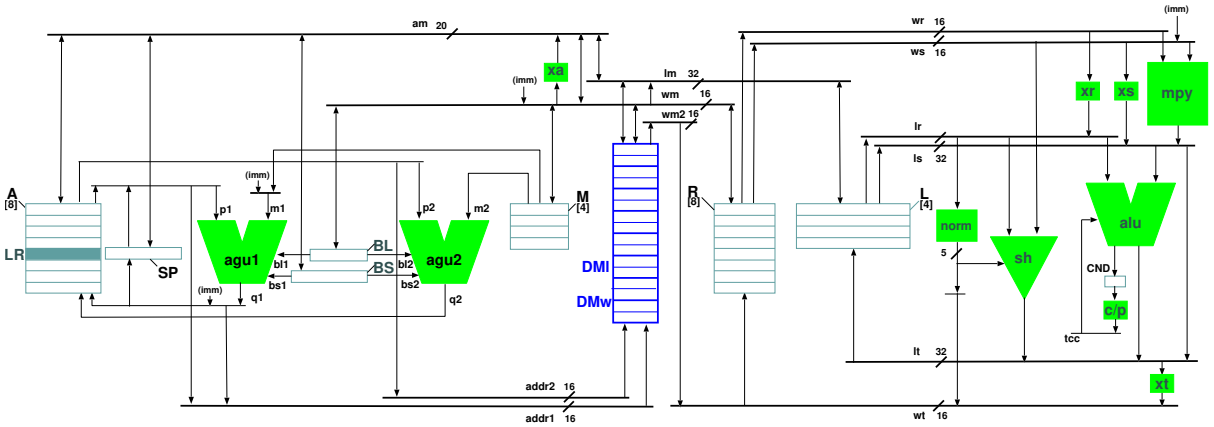

Figure 3.5: The data path of the Tdsp core (source [83]).

The Tdsp's unique data path is accompanied by its unique instruction set. The Tdsp supports 16 and 32-bit variable-length instructions. The 16-bit instructions can be one of the following:

- Arithmetic on the `R` or `L` register, or multiply-accumulate arithmetic
- Load/store instructions
- Move instructions
- Control instructions

32-bit instructions consist of the following categories:

- A combination of a 16-bit arithmetic and a control instruction
- A combination of a 16-bit arithmetic and a move instruction
- A combination of a control and a move instruction
- Long control or load/store instructions, which use direct addressing
- Long immediate instructions
- A DSP instruction combining a multiply-accumulate instruction with a load/store instruction and a second load instruction

The combination of the hardware loop control, circular buffer registers, dedicated shifter, and multiply-accumulate functionality is advantageous for signal-processing applications and makes the Tdsp a good example of a DSP.

3.2.2 TCE

With TCE, there are unlimited possibilities in designing a suitable TTA core for IMDs. However, designing a new TTA processor would be out of scope for this thesis. Therefore, existing implementations will be assessed.

Modern TTA cores designed in TCE are the LoTTA [35] and its derivatives the PeLoTTA and Super LoTTA, which are designed for always-on applications. Where the PeLoTTA still maintains the fast branching functionality and the energy-efficient operation, the Super LoTTA was designed for maximum clock frequency. The PeLoTTA also has the overall smallest code size and features an improved instruction register file (IRF). Because of the energy efficient design target of the PeLoTTA, this will be the only evaluated core for TCE.

The architecture of the PeLoTTA core can be seen in Figure 3.6. The PeLoTTA contains three transport buses which connect the following FUs:

- an LSU for moving data between the buses and the memory
- an ALU for basic arithmetic, binary operations, shifting, min/max operations, and conditional operations
- a 32x32-bit register file
- an immediate unit, for loading and storing immediates
- a control unit for executing jumps and branches
- a dedicated IO unit for standard output

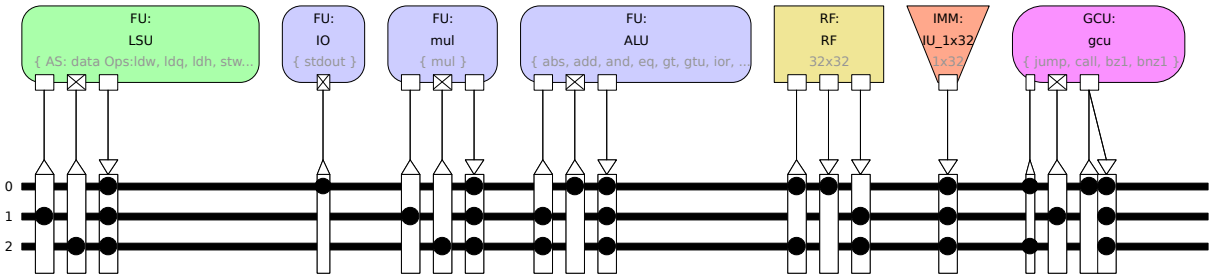

Figure 3.6: The PeLoTTA core as shown in ProDe with its buses and FUs.

Implementation comparison

The selection of ASIP implementations is dependent on the available models from the used toolsets and the architectures of interest. From the available TTA cores only the PeLoTTA is selected, providing improvements over the LoTTA and retaining energy-efficient operation. Representing conventional general-purpose RISC processors, ASIP Designer provides both the Tmcu and the Tzscale. Because of the prevalence of signal-processing tasks in IMDs and the relatively low effort of implementing another core in ASIP Designer, the Tdsp will also be implemented. To bridge the architectural gap between TTAs and the MCUs, a VLIW core will also be selected. The only VLIW core available from ASIP Designer is the Tvliw. An overview of all selected ASIPs with their properties can be found in Table 3.2.

Table 3.2: An overview of the selected cores and their properties

	Type/ISA	Instr. word size	Synthesis tool	Compiler	CoreMark/MHz
PeLoTTA	TTA	43 bit	OpenASIP	TCE	Unknown
Tdsp	DSP	16/32 bit	ASIP Designer	Chess + LLVM	Unknown
Tvliw	4-slot VLIW	64 bit	ASIP Designer	Chess	Unknown
Tzscale	RISC-V RV32I	16/32 bit	ASIP Designer	Chess + LLVM	2.38

For the PeLoTTA, Tzscale, and Tdsp synthesis results are also available, and are listed in Table 3.3. Table 3.3 shows that the PeLoTTA and Tzscale are comparable in terms of core area and clock frequency. The Tdsp reaches only half of the maximum clock frequency of the Tzscale and PeLoTTA and uses $20 - 32\%$ more area. The Tvliw likely uses the most area due to its high issue width, but no synthesis results are available for that core.

Table 3.3: Comparison of area and performance numbers of relevant ASIP Designer example projects that have it available, and the PeLoTTA core. Note that the Tvliw is not listed, as no synthesis results are available.

Core	Process	Max f_{clk} [MHz]	Gates	Core area [µm^2]
PeLoTTA (64 IRF)	FD-SOI 28nm	1351	Unknown	22036
Tzscale	TSMC 28nm	1370	36.6k	∼24000
Tdsp	TSMC 28nm	650	42k	29087

3.3 Benchmarks

Multiple benchmarks exist for evaluating the performance of an embedded system. Some benchmark suites will be presented in order to select the benchmarks that will represent the workloads of IMDs.

3.3.1 Requirements

The benchmarks should be representative of typical workloads for IMDs. Typical workloads can be broken down into six categories:

- Control systems; state machines, signal processing.
- Encryption; communication should be encrypted
- Data compression; if the IMD uses wireless communication to transfer data, every bit that can be compressed reduces the energy necessary for that transfer.
- Error detection/correction; transferred data should be able to be verified .
- ANN; a modern solution to pattern recognition and signal processing.

Next to a representative collection of benchmarks, there are some requirements related to the implementation of these benchmarks:

- Free and open-source license
- Able to run without any OS (bare bones)
- Architecture-agnostic
- Small memory footprint

3.3.2 EEMBC benchmarks

- CoreMark [84]: The industry-standard benchmark for microcontrollers is EEMBC's CoreMark. It focusses on list processing, matrix manipulation, state machines, and error detection (CRC16). The source code for CoreMark can be found on GitHub [85].
- ULPMark [86]: ULPMark is also developed by EEMBC, based on CoreMark, with a focus on ULP devices. ULPMark comes in three flavours:
  - ULPMark-CoreProfile [87], which sleeps after executing a workload, combining power measurements for sleep and active modes.
  - ULPMark-PeripheralProfile [88], which assesses the performance of peripherals.
  - ULPMark-CoreMark [89], which uses CoreMark as the active workload and an external power measurement board to determine the CoreMark executions per Joule.

  ULPMark-PeripheralProfile and ULPMark-CoreProfile will not be discussed because of the requirement of external modules (either peripherals or power measurement boards).
- SecureMark-TLS [90]: SecureMark-TLS implements TLS using ECDSA, SHA256, and SHA128.

EEMBC also provides benchmarks for machine learning; MLMark [91] and ULPMark-ML [92]. However, MLMark only runs on architectures with a GPU, and ULPMark-ML is still in development at the time of writing.

Another honourable mention is IoTMark, which focusses on wireless communication and reading out sensors. This benchmark is not listed because it requires a SoC with a Bluetooth radio.

3.3.3 MiBench

MiBench [93] is a large collection of 35 benchmarks. MiBench mainly focusses on consumer PC workloads and provides benchmarks chosen for six categories:

- Automotive/industrial control systems
- Consumer
- Office
- Network
- Security
- Telecommunications

MiBench was presented in 2001 as a free, open-source alternative to EEMBC's benchmarks (although CoreMark and other EEMBC benchmarks are now open-source as well).

3.3.4 SPEC2017

The Standard Performance Evaluation Corporation released their first benchmark in 1999, the SPEC2000. SPEC2017 is the third and latest release of the SPEC CPU benchmark suite, and contains 43 benchmarks. SPEC2017 is designed for heavy integer and floating point workloads such as benchmarks for fluid dynamics and weather forecasts, and thus is unsuited for embedded systems.

3.3.5 ImpBench

ImpBench [5] is a benchmark suite specifically designed for IMDs. The original release of ImpBench provides eight benchmarks for four categories:

- Compression
- Encryption
- Data integrity
- Real applications

In the revised version of ImpBench [6] introduces "stressmark" versions of the two benchmarks for real applications in order to improve simulation time.

3.3.6 Solutions comparison

Table 3.4 shows which benchmarks from the benchmark suites are suited for the required categories. Table 3.5 shows the licence and architectural requirements. Some notes about the tables:

- In Table 3.5 the requirement for machine learning is missing, as none of the benchmarks listed in that table have a focus on machine learning.
- The compression algorithm that MiBench possesses (`jpeg`) is lossy and designed for images, and thus is not representative for a compression algorithm for IMDs. Thus it is not listed in Table 3.4.

Table 3.4: Benchmark suites broken down in categories with workloads for IMDs.

	Control	Encryption	Compression	Error detection
CoreMark	State machines, find, sort, matrix manipulation			CRC16
ULPMark-CP	Bit permutation, bubble-sort, filters			
MiBench	qsort, bitcount, basicmath	Rijndael, PGP, blowfish		CRC32
ImpBench	DMU	RC6, MISTY1	Finnish, MiniLZO	CRC32, CSUM

Table 3.5: Analysis of licensing and architecture requirements for selected benchmarks.

	Free, open-source	Bare bones	Architecture independent	Small memory footprint
CoreMark	Yes	Yes	Yes	Yes
ULPMark-CP	No	Yes	Yes	Yes
MiBench	Yes	No	No	No
ImpBench	Yes	No	No	No

It comes as no surprise that the benchmark suite designed for IMDs has the best match for the requirements, although some work will be required to make the benchmarks run on the different architectures. ImpBench lacks a good benchmark for assessing typical control systems, other than the DMU benchmark. ULPMark-CoreMark is not an option; even though it can be used with a university licence, the benchmark relies heavily on letting the processor sleep, which is not possible for any of the selected implementations. Therefore, CoreMark will also be selected to run on the ASIP implementations.

As for a benchmark to assess performance for ANNs, no portable, lightweight benchmarks are available. Therefore, a custom benchmark will be designed and implemented in the next chapter.

Section 2.6.3 shows that most ANNs in IMDs are either CNNs or RNNs. The main difference between these networks is that a CNN is a feed-forward network (output of a neuron is only dependent on the input), whereas a neuron in an RNN has local memory to which data can be fed back from the outputs of other neurons.

For a benchmark, a CNN for digit classification of the MNIST dataset [94] will be implemented. Even though the data does not represent biological signals, the dataset is very well-known and CNNs have been used extensively to classify the digits in the set [94].

Chapter 4

Implementation

4.1 Development platform

The processor cores will be synthesized on an FPGA for power measurements. The FPGA selected is the Cyclone IV EP4CE15F23C8N from Altera. This FPGA was chosen simply because it is cheap and readily available. As the target is an Intel/Altera FPGA, the synthesis and fitting is done by Quartus Prime.

The FPGA provides 15k logic elements (LEs) and 504kbit memory, and thus can provide for 32kB of data memory, and 16kB of instruction memory [95]. The EP4CE15F also boasts 56 18x18 multipliers, 4 PLLs, and 20 global clock networks (GCNs). The units primarily used by the processor cores are LEs, M9K memory, and the multipliers.

4.2 Processor core implementation

The top-level entity of each core contains only a few components:

- The processor core
- Program memory
- Data memory
- PLL
- Standard output

For each core it will be explained how the HDL for the core is generated, how the program and data memory are attached to the core, how the PLL is implemented, and how the standard output module is implemented.

4.2.1 Top-level

The top-level entities of the Tzscale, Tvliw, and PeLoTTA are very much alike. Figure 4.1 shows a diagram of the layout of the top-level entity of the Tzscale. The top-level entity of the Tvliw and PeLoTTA are the same, but only have one DM bank, and the PeLoTTA has a standard output module as FU in the processor core.

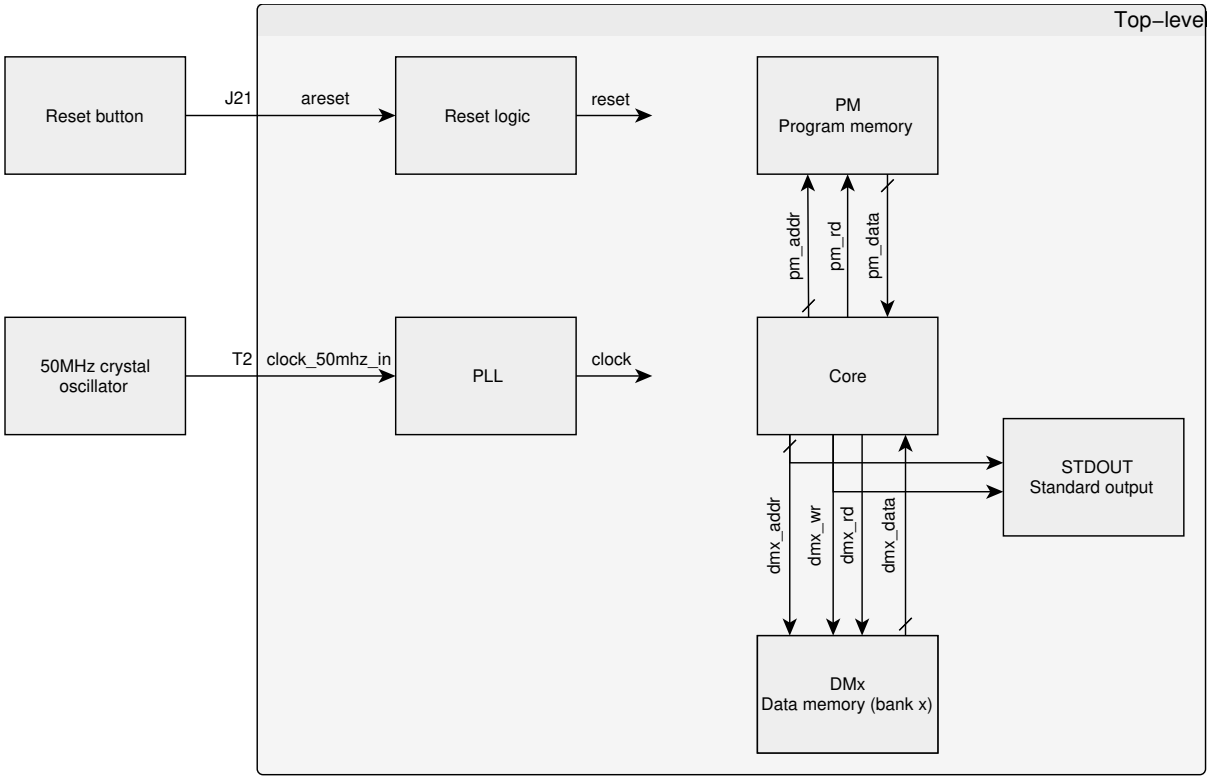

Figure 4.1: A diagram of the top-level entity of the Tzscale. The diagram for the Tvliw is the same, but with only one data memory bank, and for the PeLoTTA the standard output module is integrated in the core.

PLL

For all cores, the Quartus PLL IP is used to generate the clock at the required frequency. To generate the bitstream for cores running at a range of different frequencies, the adjustments to the PLL will have to be automated. The period of the generated clock from the PLL is determined by a division and multiplication factor (Equation 4.2.1), which are defined in the `pll.vhd` IP file. These factors can then be calculated and replaced in the IP file to attain the desired frequency.

To simplify the calculation of the factors, the division factor is set at 400. The multiplication factor is then the only missing factor determining the frequency, and can be set to bring the clock frequency closest to the desired frequency (see Equation 4.2.1).

$$f_{clk} = f_{in} \cdot \frac{c_{mul}}{c_{div}} \tag{4.1}$$

$$c_{mul} = f_{tgt} \cdot \frac{c_{div}}{f_{in}} = f_{tgt} \cdot \frac{400}{50MHz} \tag{4.2}$$

This approach is implemented in a bash script (`pll_adj`), and the replacement is done using `sed`. After the adjustment of the PLL, Quartus recommends the memory IPs are regenerated. This is done using the `quartus_sh -ip_upgrade -mode all <project>` command.

Configuration

Most configuration options of Quartus are left as-is, except for auto-recognition of DSP modules. Using the built-in multiplication and division module is inevitable however, as the alternative is designing and implementing one. Also the strategy is set to aggressive optimization for power. These options are set via the command line with

```
quartus_sh -set OPTIMIZATION_MODE="AGGRESSIVE POWER" <project>
```

and

```
quartus_sh -set AUTO_DSP_RECOGNITION="OFF" <project>.
```

After successful configuration and setting of the desired clock frequency, `quartus_sh -flow compile <project>` can be run to synthesize and fit the project, and generate the bitstream.

4.2.2 PeLoTTA

The Quartus project for the synthesis of the PeLoTTA core can easily be generated using ProGe from TCE. ProGe does not support targeting the Cyclone IV directly, but does support the Altera Stratix 2 platform. The generated processor for Stratix 2 uses Quartus IP components that are also available for the Cyclone IV, so re-targeting the project is done by just specifying a different target in Quartus Prime.

Memory

The memory size used by the PeLoTTA is set in the ADF file. The ADF file defines the memory size by setting the address range. Listing 4.1 shows the instruction and data memory definitions of the PeLoTTA.

```
839   <address-space name="data">
840     <width>8</width>
841     <min-address>0</min-address>
842     <max-address>32767</max-address>
843   </address-space>
844
845   <address-space name="instructions">
846     <width>8</width>
847     <min-address>0</min-address>
848     <max-address>4095</max-address>
849   </address-space>
```

Listing 4.1: The definition of the instruction and data memory in the ADF file for the PeLoTTA.

Standard output

One significant change to the generated RTL is in the standard output module. The standard output implementation uses UART over the Altera JTAG interface (which is connected to the USB Blaster). The module has a `lock_req` output (see Listing 4.2) which is connected to the `lock_req` input of the `pelotta_decoder` entity (the instruction fetcher/decoder). When the `lock_req` input of the `pelotta_decoder` is asserted, the decoder stalls until it is deasserted again when the buffer is cleared. This effectively stalls the entire processor. The standard output module requests the lock while the write buffer is not empty, and the buffer will not be emptied when there is no terminal (`nios2-terminal`) running on the PC host that acknowledges the data.

Although this feature makes sure the data is successfully transferred, the module can stall for tens of milliseconds while writing strings via standard output. During the lock, the processors energy consumption is very low, and thus the power measurements are lower as well.

To circumvent this error, the `lock_req` input of the `pelotta_decoder` is simply set to 0. To make sure that the write buffer doesn't overflow, care is taken in the ImpBench source code to minimise the amount of data transferred.

```
5  entity altera_jtag_uart_stdout_always_1 is
6    generic (
7      dataw : integer := 8);
8    port (
9      t1data   : in  std_logic_vector(dataw-1 downto 0);
10     t1load   : in  std_logic;
11     clk      : in  std_logic;
12     rstx     : in  std_logic;
13     glock    : in  std_logic;
14     lock_req : out std_logic
15     );
16 end altera_jtag_uart_stdout_always_1;
```

Listing 4.2: The definition of the `altera_jtag_uart_stdout_always_1` entity in `altera_jtag_uart_stdout_always_1.vhd`, which is used as the standard output module for PeLoTTA.

4.2.3 ASIP Designer processors

There are a few steps in the process of generating the RTL description for the target from the nML processor description.

HDL generation

To generate the HDL from the processor model, Go[1] is used. The `tzscale_vhdl.prx` HDL generator project file contains the options to specify the language (VHDL) and the Go configuration file. The configuration file contains options for on-chip debugging, simulation options, controller/decoder, low power, timing, and transformation of the data-path, among others. Some important options are explained below.

- These are the recommended options for optimising for low power [96]:
  - `extra_enable_bit;`
    This option adds an extra enable bit from the decoder, which controls if a FU is in an active state or not (NOP).
  - `split_opcode_registers;`
    Opcode registers in the decoder are split into smaller registers, combined with `extra_enable_bit` it makes the registers (which are split for each FU) update only when the enable bit is set, saving unnecessary bit toggles.
  - `default_primitive_operations : 1;`
    `default_register_reads : 1;`
    `default_memory_operations : 1;`
    The `default_*` options cause the affected units to always execute an operation, even when the control signals are all zero. This might save an extra bit in the control signals if the number of operations is a power of two, and saves some logic because the decoder doesn't need to generate a non-zero opcode for that operation. In combination with `extra_enable_bit` and `split_opcode_registers` this also saves power because the reduced bit toggling is propagated to these units.
  - `register_vector_write_enable;`
    `pipe_write_enable;`
    Generates a write enable signal for each (pipeline) register, allowing the synthesis tool to use clock gating for the registers.
  - `operand_isolation_functional_units;`
    `operand_isolation_multiplexers;`
    The `operand_isolation_*` options ensure that the control signals of the affected unit remain constant when it is in a passive state, again reducing unnecessary bit toggling.
  - `register_addresses_from_decoder;`
    Enabled by default. Ensures register addresses are kept constant, reducing power. Also reduces critical path by only fetching the necessary register addresses the stage before they are needed.

[1] The tool from ASIP Designer, not to be confused with the programming language.

- Data-path transformation:
  - `remove_false_paths;`
    Removes false paths between memories and false loops.
  - `print_false_paths;`
    Writes all false paths to a TCL script for the synthesis tool, so that no timing analysis will be done for those paths.
- Reducing critical path:
  - `direct_write_disable_on_stall : 2;`
    In addition to zeroing the opcodes when stalling, register writes and memory accesses in the modules that are stalled are also disabled.
- VHDL generation:
  - `merge_identical_entities;`
    `merge_package_body_files : 1;`
    `merge_entity_architecture_files : 1;`
    `configuration_files : 0;`
    These options reduce the number of files and also increase compatibility with ModelSim.

Memory

To modify the memory size of a processor in ASIP Designer, the processor nML model has to be modified so that the compiler and simulator know what memory is available. At the beginning of the processor model in `tzscale.n` the memories are defined (see Listing 4.3). Likewise, for the Tdsp and Tvliw the memory size is adjusted in `tdsp.n` and `tvliw.n`. The memory sizes are easily adjusted by changing the `pm_size` and `dm_size`, which specify their respective memory sizes in bytes. At the definition of the data memory, it can also be seen that the Tzscale has byte-aligned data memory (DMb), with aliases for half-words (DMh) and words (DMw).

```
22  // Program memory
23
24  def pm_size=2**14;
25
26  mem PMb[pm_size] <uint8,addr> access {};
27
28  mem PM[0..pm_size-4,2] <iword,addr> alias PMb align 2 access  {
29      ifetch : pm_rd '1' = PM[pm_addr]'1';
30  #ifdef HAS_OCD
31      istore : PM[pm_addr] = pm_wr;
32  #endif
33  };
34
35  properties {
36      program_memory: PMb;
37      unconnected : PM; // accessed in PCU
38  }
39
40  // Data memory
41
42  def dm_size=2**DM_SIZE_NBIT;
43
44  trn dmh_wr_hi<w08>;
45  trn dmw_wr_hi<w16>;
46
47  trn dmb_wr<w08>;
48  trn dmh_wr<w16> { dmb_wr; dmh_wr_hi; };
49  trn dmw_wr<w32> { dmh_wr; dmw_wr_hi; };
50
51  mem DMb [dm_size,1]<w08,addr> access {
52      ld_dmb: dmb_rd '1' = DMb[dm_addr '0' ] '1';
53      st_dmb: DMb[dm_addr] = dmb_wr;
54  };
55
56  mem DMh [dm_size-1,1]<w16,addr> alias DMb align 1 access {
57      ld_dmh: dmh_rd '1' = DMh[dm_addr '0'] '1';
58      st_dmh: DMh[dm_addr] = dmh_wr;
59  };
60
61  mem DMw [dm_size-3,1]<w32,addr> alias DMb align 1 access {
62      ld_dmw: dmw_rd '1' = DMw[dm_addr '0'] '1';
63      st_dmw: DMw[dm_addr] = dmw_wr;
64  };
```

Listing 4.3: The memory definitions for the Tzscale in `tzscale.n`

The data memory is divided in two equally sized interleaved banks; DM0 and DM1. This way, both banks of data memory can be accessed in the same cycle with word-aligned addresses. The 64-bit result can then be reduced to 32-bit with a byte-aligned offset. This is convenient for the implementation of the data memory, which is done using M9K memory blocks. The DM0 and DM1 memory entities, which are generated using the Quartus Megawizard Plug-in Manager and the 1-port RAM IP, plugs in directly to the interface of the Tzscale core.

The program memory for the Tzscale is also byte-aligned, but only has one bank. The memory implemented is actually half-word aligned, because the Tzscale only supports 32 and 16-bit instructions. This leads to an implementation where there are actually two interleaved memory banks (`pm0` and `pm1`) with a width of 16 bits. The last bit of the requested address remains unused, and the second-to-last bit determines the order and the actual read addresses of the memories. The calculation of the addresses and data order can be found in Table 4.1.

Table 4.1: The translation of the address and data to handle half-word aligned memory accesses. `pm_addr_in` is the requested addres from the Tzscale. The positions indicate if the resulting 16-bit value from `pmX` is placed at the lower 16 bits or the higher 16 bits of the 32-bit result. PM_SZ equals 14 for a program memory of 16kB.

Alignment	word aligned	half-word aligned
`pm_addr_in`	`XXXXXXXXXXXX0X`	`XXXXXXXXXXXX1X`
`pm0` address	`pm_addr_in(PM_SZ - 1 downto 2)`	`pm_addr_in(PM_SZ - 1 downto 2)`
`pm1` address	`pm_addr_in(PM_SZ - 1 downto 2) + 1`	`pm_addr_in(PM_SZ - 1 downto 2)`
`pm0` position	Low	High
`pm1` position	High	Low

The Tvliw uses 32-bit data memory and 64-bit program memory, and is only word-addressable for both memories. This makes the connection of the memory for the Tvliw very straightforward as no translation on the data and address are required.

The Tdsp uses two 16-bit data memory banks with two read ports and one write port. For the program memory it uses one 16-bit bank with two read ports, in order to load the 32-bit long instructions. The Quartus 2-port RAM and ROM IP is used to implement the memories for the Tdsp, with one write port disabled for the data memory. These memories plug in directly to the Tdsp processor core.

Standard output

The same standard output module as used in the PeLoTTA is implemented on the Tdsp, Tvliw, and Tzscale (`altera_jtag_uart_stdout_always_1`). Using the same standard output module ensures the same area and power consumption for debugging. To attach this module to the ASIP Designer processor cores, a word in memory is mapped to the input of the module. When the reserved memory is written, the value is forwarded to the standard output module and the `t1load` flag is set (which indicates new valid data is available).

The memory region is reserved in the default linker configuration file (`tzscale.bcf` for Tzscale, see Listing 4.5). The syntax for the directive is `_reserved <memory> <offset> <width>`.

```
145     process(clock)
146         constant data_addr : t_addr := x"00000000";
147     begin
148         if(rising_edge(clock)) then
149             if(nareset = '0') then
150                 if(st_dm1 = x"F" and dm1_addr = x"0000") then
151                     uart_data_new <= '1';
152                     uart_data <= std_logic_vector(dm1_wr(7 downto 0));
153                 else
154                     uart_data_new <= '0';
155                 end if;
156             else
157                 uart_data_new <= '0';
158                 uart_data <= (others => '0');
159             end if;
160         end if;
161     end process;
162
163     inst_uart : altera_jtag_uart_stdout_always_1
164     port map (
165                  t1data   => uart_data,
166                  t1load   => uart_data_new,
167                  clk      => clock,
168                  rstx     => areset,
169                  glock    => '0');
```

Listing 4.4: The implementation of the TCE `altera_jtag_uart_stdout_always_1` standard output module in the Tzscale top-level entity in `top.vhd`

```
16 _reserved DMb 0 4 // reserve location used by debug client (see  debug_client/pdc_data_address.h)
17 _reserved DMb 4 4 // reserve location used by stdout
18
19 _stack DMb 0x4000 0x04000
```

Listing 4.5: The reservation of memory for the standard out module in the Tzscale linker `tzscale.bcf`

Compiler options

The Tzscale and Tdsp support the LLVM compiler front-end Clang for the ASIP Designer Chess compiler. The LLVM front-end provides a significant performance increase compared to the Chess front-end. Using LLVM, the Tzscale achieves a CoreMark score of 2.38/MHz, whereas the Chess front-end only achieves 1.94/MHz [81]. Thus, to get the best performance out of the Tdsp and Tzscale, the LLVM compiler front-end is used. The only compiler option used for LLVM is the optimization level, which is set to aggressive (`-O3`).

The Tvliw does not support this front-end, and as the Chess compiler has no options for optimization, there is no configuration to be discussed.

4.3 ImpBench adaptations

ImpBench is not able to run on the selected processor implementations out-of-the-box because of unsupported OS calls, floating point instructions, and a high memory footprint. This section will cover all changes made to ImpBench in order to run it on the cores.

Developing for four (excluding x86 for testing) different architectures can create unforeseen bugs. Two major issues during debugging were found that caused bugs:

- Tvliw has word-aligned data memory, and thus uses an `unsigned char` and `uint8_t` size of 4 bytes.
- PeLoTTA has big endian memory, as opposed to the little endian memories of the Tdsp, Tvliw, and Tzscale

4.3.1 File I/O

ImpBench uses file I/O to load datasets and verify and store results. However, none of the processors run an OS with a file system, so this functionality will have to be edited to use only supported operations.

The only reason to use a pre-determined dataset is to be able to verify the outputs of the program and thus verify the correct functioning of the core it's running on. In order to obtain correct results, it is important to verify the execution of the benchmarks on the core, so the verification part can not just be removed. This also means that the file I/O can not be replaced by function calls to `rand()`. This leaves only one option, placing the data in the data region in RAM.

As only one input file used for all benchmarks (i.e. `AEP_10.ascii`), it would be efficient to include the contents from this file only once in memory. However, in ImpBench the data from that input file is read in a number of different ways:

- FIN: Reading the file one character at a time and casting it to an `int` (using `fgetc()`)
- CRC32, CSUM: Interpreting the ASCII bytes from the file directly as a different datatype (such as `double` and `short int`, using `fread()`)
- Motion, DMU: Reading one line at a time and converting the ASCII to a float (using `fgets()` and `atof()`)
- MLZO, MISTY1, RC6: Using the raw bytes from the file as raw data to compress or encrypt

Although the binary representation of the entire file could be loaded in RAM, it would be inefficient to translate it to all different data types and interpretations.

Compression

The only available form of memory implemented in the cores (aside from registers) is the instruction and data memory, so all data needs to be placed in those memories. The memories are severely limited by

the FPGA they are synthesized on. The FPGA (an Altera Cyclone IV E EP4CE15F23C8) has 504Kb of M9K memory blocks. The memory needs a fully addressable space of a power of 2 (restriction by Quartus), the maximum size for the data memory is 32KB. This leaves 16KB for the instruction memory.

Although the input data file (`AEP_10.ascii`) that is used by the benchmarks is 12KB and can fit inside the data memory, it takes up more than 35% of the RAM. As some benchmarks require more than 20KB RAM, it is beneficial to reduce the memory footprint of the input data. Fortunately, the data inside `AEP_10.ascii` is very repetitive (see Listing 4.6).

```
90.3014
90.3014
90.3014
90.3014
90.3014
90.3014
90.3014
90.3014
90.3014
90.3014
```

Listing 4.6: An exerpt from `AEP_10.ascii` showing the repetition of data.

In fact, `AEP_10.ascii` only has 80 unique lines, of which every line starts with "90.", followed by three or four numbers, and ends with a tab, line feed, and carriage return (`\t \r \n`) A very simple and specific compression algorithm can be used in which for every unique line it is specified how many times it is repeated, saving only the unique decimal numbers in RAM. The function only has to store the position in the line and at what line in `AEP_10.ascii` data is requested through function calls. The total data memory footprint in is 625 bytes, consisting of:

- 412 bytes for the storage of the decimal numbers of each different line
- 206 bytes for the line numbers at which these data transitions occur
- 7 bytes for counters and storage of endianness

This reduces the memory footprint of `AEP_10.ascii` by 95% (not counting the batch size when decompressing). The function implementing this routine can be found in Listing 4.7.

As all benchmarks read `AEP_10.ascii` from the beginning (no calls to `fseek()` to jump at a specific part in the file), there can be one simple function to implement the reset of the counters that store the line and the position within the line. This function is `data_gen_reset()` and is called at the start of every benchmark to ensure the data generation starts at the beginning.

```
174 static uint8_t data_gen_get8(void) {
175     // all lines start with "90."
176     uint8_t common_start[] = {0x39, 0x30, 0x2E};
177     // all lines end with "\t\r\n"
178     uint8_t common_end[] = {0x09, 0x0D, 0x0A};
179     uint8_t res;
180     // if we reached the last line of occurrence of the last line, go to the
181     // next
182     if (line_idx == data_lines[data_lines_idx]) {
183         data_mid_idx++;
184         data_lines_idx++;
185     }
186     // if this line only has 3 decimal characters, go to the common_end (by
187     // increasing the column_idx by one)
188     if (column_idx == 6 && data_mid[data_mid_idx][3] == 0x09)
189         column_idx++;
190     // now get the data
191     if (column_idx <= 2)
192         // common 3 start bytes (i.e. "90.")
193         res = common_start[column_idx];
194     else if (column_idx >= 7)
195         // common 3 end bytes (i.e. \t\r\n)
196         res = common_end[column_idx - 7];
197     else
198         // unique 3-4 mid bytes
199         res = data_mid[data_mid_idx][column_idx - 3];
200     if (data_mid_idx < sizeof(data_lines) / sizeof(uint16_t))
201         column_idx++;
202     if (column_idx > 9) {
203         column_idx = 0;
204         line_idx++;
205     }
206     return res;
207 }
```

Listing 4.7: `data_gen_get8()` from `data_gen.c`, which restores the compressed data in RAM. The function is delcared static as it is only used by `data_gen_buf()`, which fills arrays.

Batches

Although the data from `AEP_10.ascii` is now compressed in the RAM, the complete decompression of the data would again require the full 12KB (plus the 612 bytes for the compressed data). However, now that data from can be generated on the fly, the benchmarks can be executed on smaller portions of the data at a time. I.e., a single call to `data_gen_buf()` can be replaced by multiple calls until all the data is processed. The batch size should be increased as much as possible for each benchmark to keep the overhead of the decompression as small as possible. This routine can be seen in Listing 4.8 for the CSUM benchmark.

An incidental benefit of using batches is that the batch size is known beforehand and thus no dynamic memory allocation is necessary. This is beneficial because dynamic memory allocation in embedded systems may be unsafe when the memory can not be allocated (which is more prevalent in systems with extremely limited memory).

```
54     data_gen_reset();
55     for (i = 0; i < CSUM_DATA_SIZE - CSUM_BLK_SIZE + 1; i += CSUM_BLK_SIZE) {
56         data_gen_buf((uint32_t*)data, CSUM_BLK_SIZE);
57         sum = checksum(sum, (uint32_t*)data, CSUM_BLK_SIZE);
58     }
59     if (CSUM_DATA_SIZE - i > 0) {
60         data_gen_buf((uint32_t*)data, CSUM_DATA_SIZE - i);
61         sum = checksum(sum, (uint32_t*)data, CSUM_DATA_SIZE - i);
62     }
```

Listing 4.8: An exerpt from `csum.c` showing the use of data generation and handling in batches.

Endianness

The endianness of a processor defines the byte order in memory. The Tdsp, Tvliw, and Tzscale use little endian memory (least-significant byte at byte 0) and the PeLoTTA uses big endian (most-significant byte at byte 0). This can cause data that consists of multiple bytes to be wrongly interpreted when it is hard-coded into the memory. This only forms a problem at two points in the ImpBench code; in `data_gen.c` and somewhere in the MISTY1 benchmark (which will be discussed in Section 4.3.5).

For the data decompression/generation, the data is generated byte by byte. In some benchmarks (like in CSUM) the data is re-interpreted as data types with sizes larger than one byte. The `data_gen_buf()` solves this by taking three arguments; a pointer to some allocated memory `buf` of size `len` bytes, to be filled with data of size `datasize`. The function then reverses the order of every `datasize` bytes if the processor is big endian.

Endianness can be detected at run-time without hard coding by writing to a data type that consists of multiple bytes, casting it to a smaller data type, and reading it back out. This function is implemented in `endian.c` can be seen in Listing 4.9.

```
15 int endian_is_little(void) {
16     // set lower byte of a 16-bit int to 1
17     uint16_t i = 1;
18     // read out lower byte of i
19     uint8_t* j = (uint8_t*)&i;
20     // if the lower byte is 1, it is little endian
21     return *j == 1;
22 }
```

Listing 4.9: `endian_is_little()` from `endian.c`, which returns the endianness of the processor.

4.3.2 Random data

It is necessary to verify that the core is working before doing measurements. However, verification is unnecessary during the measurement assuming the measurement setup is a time-invariant system. This way, a new program can be loaded in the core which doesn't generate ImpBench-like data but just generates random data to use for the benchmarks.

Pseudo-random number generation

The easiest way to get random data is a call to `rand()`, which returns a random integer. Because the numbers are randomly generated in software whithout any seed from hardware, they are not truly random. I.e.; after compilation, every time the program runs it uses the exact same data in the same order. For benchmarking, this is favourable, as the benchmark would produce the same results every time it is called. Although there is one caveat; each compiler may use their own implementation of `rand()`. If different cores use different data, and benchmarks can branch based on that data, it may cause a difference in performance. To fix this, the implementation from ASIP Designer is copied to the modified ImpBench source to make sure every compiled version uses the exact same data. The ASIP Designer implementation is changed slightly to provide 32-bit random numbers instead of 16-bit. The resulting function is shown in Listing 4.10.

```
109 uint32_t rand(void) {
110     next = next * 1103515245 + 12345;
111     return next;
112 }
```

Listing 4.10: The `rand()` implementation from ASIP Designer used in `data_gen.c` in the modified version of ImpBench. `next` is a static volatile variable declared in the same file.

Compression benchmarks

Unlike most benchmarks from ImpBench, the compression benchmarks (FIN and MLZO) suffer greatly from using random data. This is the case because compression benchmarks are not designed to be used with truly random data (as that is impossible [97]). The unpredictability of the pseudo-random numbers cause the dictionary of FIN to fill up, and use large amounts of RAM.

To make the pseudo-random data more predictable, the set of possible random data to return can be reduced. This is implemented by just masking only the lower four bits of the random data, so there are only 16 possible outcomes instead of 256. The implementation can be found in Listing 4.11.

```
222 void data_gen_buf(uint32_t* buf, size_t len) {
223     size_t i;
224 #if DATA_GEN_RANDOM
225     uint32_t r;
226     uint32_t mask = 0;
227     // fill up buf 8 bytes at a time
228     for (i = 0; i < len / 8; i++) {
229         r = rand();
230         // only take last 4 bits per byte, so there are only 16 possible
231         // outcomes. this reduces memory footprint for compression benchmarks
232         buf[i] = r & 0x0F0F0F0F;
233         i++;
234         buf[i] = (r >> 4) & 0x0F0F0F0F;
235     }
236     // fill-up left-over bytes
237     r = rand();
238     if(len % 8 >= 4) {
239         buf[i] = r & 0x0F0F0F0F;
240         i++;
241         r >>= 4;
```

Listing 4.11: The usage of `rand()` in `data_gen_buf()` in `data_gen.c`, which uses only the lower 4 bits. The `DATA_GEN_RANDOM` macro determines if `data_gen_buf()` fills `buf` with pseudo-random data or with data from `AEP_10.ascii`.

4.3.3 Standard output

The functioning of the processor can be verified by using a debugger, GPIOs, and writing to standard output. Simulation is not enough as timing issues and other external factors can not be seen in simulation. A debugger would be interfacing via JTAG over the test access port (TAP). This requires the core to also implement a TAP controller. Although ASIP Designer does have the option to implement the TAP controller and interface to the processor, the attempt to implement is was unsuccessful. The JTAG debugger (a J-Link EDU) complained about not being able to access the TAP controller and analysis with a logic analyzer showed that the TAP controller was not responding. At that point further development for the JTAG interface was stopped, and the focus shifted to standard output.

Writing to standard output is an easy way of debugging and verifying the correct execution of the program. This is usually done using calls to `printf()`, like in ImpBench, but the `printf()` implementation is quite large and most functions are unused by ImpBench. TCE actually provides an alternative to `printf()` with a smaller memory footprint for this purpose, but as this is not availabe for ASIP Designer, an implementation is added to ImpBench.

All that would be necessary to provide a notification of successful execution is the printing of a single byte. For conveniency however, it is also nice to print strings and integers in both hexadecimal and decimal representation. These features require little code and little time to implement, and thus are implemented in `print_int()` and `print_str()` in `print.c`.

4.3.4 Verification

To verify that the benchmarks are executed successfully during measurement, the cores log the successful execution of the benchmark via standard output. The correct execution can be determined by two factors:

- The benchmark which is executed

- The benchmark produced the expected result

These two variables are printed over standard output. First the number corresponding to the benchmark is printed, then after the benchmark is finished, the check is printed. The resulting character printed that represents the success of the function can be found in Table 4.2. By minimising the characters printed over standard output, the overhead in power consumption is minimised while maintaining the knowledge of the validity of the measurements.

Table 4.2: Possible log results when running a benchmark.

	ImpBench data	Random data
Benchmark returns 0	V	v
Benchmark returns other than 0	X	x

For CSUM and CRC32, verifying correct execution can easily be done by comparing the resulting checksum to the correct checksum. For the other benchmarks however, this is not so trivial. For the compression and encryption benchmarks, the first byte of each compressed block is added to a resulting checksum. For the motion benchmark, the amount of active measurements is used as a checksum.

The checks are performed at the end of each function, comparing the sum to the expected result (see Listing 4.12 for the implementation in the CRC32 benchmark).

```
187     // check against correct answer
188     return res != 0x8201357A;
189 }
```

Listing 4.12: The calculation of the return value for the `crc()` benchmark, which is used to verify the result and determine if the execution was successful.

4.3.5 Benchmark specific changes

This section covers the changes made to individual benchmarks to port them to the selected processors.

CRC32

In the original CRC32 benchmark, the CRC32 table is generated at the start of the program. This is not very representative for the workload, as this is only done once after reset. Given that the table generation is done within a few milliseconds, and the processor runs for days or months, the effect is negligible in a normal application, but significant in the benchmark which only runs briefly. Therefore, the CRC32 table is instead hard coded in the source file.

CSUM

The original implementation of CSUM in ImpBench doesn't negate the left-over byte (if it exists, which it does in the used dataset). Although this seems like a small bug that was overlooked, it has no real implications in the performance or the results in the measurements, as it is only one operation.

In the original source the 32-bit checksum is folded over to a 16-bit checksum at the end of the `checksum()` function. This operation is now moved to the end of the `csum()` function, so that it is only done once and not after every block of data.

FIN

FIN works with a large prediction table to compress the data. This table has a size of 32kB in the original source. This is too large for 32kB of data memory and has been reduced to 4kB.

The `compress()` and `decompress()` functions have also undergone some slight changes. The functions now take pointers to arrays as arguments instead of file pointers. It also checks for buffer overflows when using those arrays.

MISTY1

In the initialization of the MISTY1 benchmark, the encryption key is initialized. The key is originally read from the arguments passed from the command line, and should represent the string `0123456789abcdef`. The key is then copied using `memcpy()` to a buffer of type `u4[]` (which is defined as a `long int`). This again is a problem for cores using big endian; it will reverse every group of four characters.

Next to fixing the bug caused by endianness, dynamic memory allocation calls for the key have been substituted by fixed size declarations.

MLZO

The MLZO benchmark uses 64-bit integer types (`long long`), which are not supported by the PeLoTTA, Tdsp, and Tvliw. It also uses more than 32kB memory, which can not be reduced without understanding and rewriting large portions of the tightly-coupled code. Thus it has been decided not to port the MZLO benchmark and rely solely on the FIN benchmark for compression performance metrics.

RC6

Other than the common changes for all benchmarks and hard coding the encryption key, the RC6 benchmark works out of the box and is left untouched.

Motion

The MOTION benchmark boils down to only counting the amount of 'samples' exceeding a certain threshold. The 'samples' from the input file used in the MOTION benchmark are ASCII representations of floats, and are interpreted as such. The issue here is that none of the selected cores have an floating point unit (FPU). This could be circumvented by using fixed point representations of the floating point numbers or software floating point libraries, but as the only operation here is a simple comparison, it was chosen to just use the pseudo-random number generator.

DMU

The DMU benchmark relies very heavily on floating point numbers and is very tightly coupled. An attempt was made to replace the floating point operations with a fixed-point arithmetic library, but the original execution of the benchmark could not be replicated. Therefore, this benchmark is also not ported.

4.3.6 Managing optimisation

The LLVM front-end of TCE uses aggressive optimization. ASIP Designer can also use the LLVM front-end, if the processor supports it. This optimization level is so aggressive that it will optimize functions away if their output values are fixed. To counter this, TCE implements the `-k <var>` flag, which keeps the compiler from optimising the indicated global variable away. For Chess, functions are available to instruct the Chess compiler to protect certain variables, functions, or operations from optimisations.

The benchmarks can also be protected against optimization by making the standard output depend on the input (pseudo-random data). This can be implemented by just declaring the `next` variable (which determines the next pseudo-random variable) as `volatile`. The `volatile` declaration forces the compiler to read the variable from memory each time and thus prevents optimizations.

The compiler-specific approach is chosen to provide the compilers with more flexibility and room for optimisations.

4.4 CoreMark

CoreMark is designed as a highly portable benchmark and thus requires less effort to implement than ImpBench. Porting CoreMark to a new platform can be done by modifying the `core_portme.c` and `ee_printf.c` files, which contain functions for standard output and calculating the elapsed time. The elapsed time is used to check if the benchmark has been running for long enough for the result to be validated. This feature is disabled in the code, as the execution time can be extracted from the simulations.

Although CoreMark is a benchmark on its own, it is called from `impbench.c`, so it integrates nicely with all scripts for ImpBench. This also implies that the standard output functions from CoreMark are not used, and it only returns the amount of errors it encountered. This way the same output is expected from CoreMark as from individual benchmarks from ImpBench, which can be parsed automatically.

Even though CoreMark is designed as a portable benchmark, it does not function correctly with all processor implementations. The Tdsp and Tvliw do not have byte-addressable memory, and their implementation of the `sizeof()` function returns the size of the datatype in words instead of bytes. This is a violation of the ANSI C standard [98], and makes the CoreMark validation fail for these cores.

Modifications to the CoreMark benchmark would make the scores lose their meaning in comparison to processors that are not evaluated in this thesis, so CoreMark is left untouched. Modifying the `sizeof()` implementation is also not an option, as the underlying problem of differences in expected data type size would still exist. The only solution therefore would be implementing byte-addressable memory in these cores, but that is outside of the scope of this thesis. Thus CoreMark is not to be run on the Tdsp and Tvliw.

4.5 Artificial neural network benchmark

The benchmark is based on a 3-layer CNN by Matt Lind [49, 99] to recognise digits in the MNIST dataset [94]. The CNN is built up as follows:

1. An input layer containing the values of the pixels of the image, with the same amount of nodes as pixels in the image.
2. A hidden layer, which executes the convolution. This layer has 20 nodes.
3. An output layer of 10 nodes, which classifies the digit in the image.

All layers in the network are fully connected, and all connections have weights. The activation function used is the Sigmoid function, which delivers the best results in this CNN [49].

Although CNNs used in IMDs often consist of more layers, the workload that the algorithm imposes is comparable because the difference would only reside in the amount of nodes the algorithm has to loop over and possibly the use of a different activation function. See Figure 4.2 for the overview of the network.

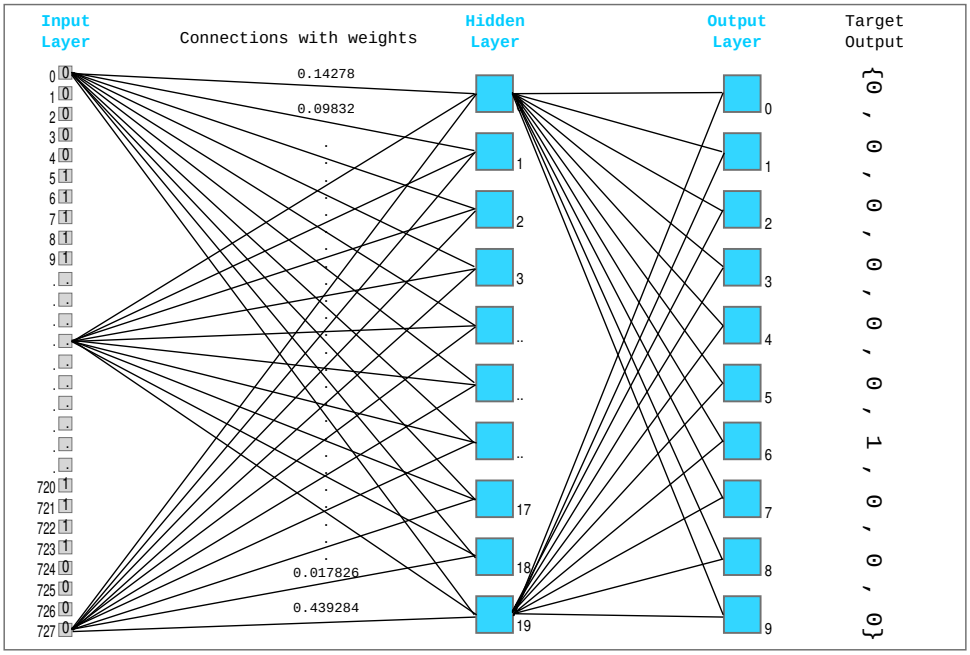

Figure 4.2: The architecture of the 3-layer CNN by Matt Lind (Source: [49]).

4.5.1 Data representation

The original source [99] uses double-precision floating point numbers to represent the weights and the input and output values of the nodes. As this is not supported by the selected ASIP implementations, this data type has been converted to fixed-point arithmetic. To facilitate this, a minimalistic library is constructed from scratch that supports addition, subtraction, multiplication, division, and the Sigmoid function.

Because the multiplication operation produces a integer twice the size of the input operands and the largest data type available on all platforms is a 32-bit integer, the fixed-point number can only be 16 bits long. Excluding the sign bit, this leaves only 15 bits for both the magnitude and the fraction. The best results were achieved by using $Q3.12$, which can represent numbers in the interval $[-8; 8)$ with a step size of 0.000244.

The Sigmoid function is implemented using a lookup table (LUT). Even though [62] states that the frequent memory accesses would be slower than a linear approximation, the combination of the use of single-cycle data memory in the ASIP implementations and fewer branches makes a solid case for a LUT. [62] also states that the size of the LUT would be too large. As a LUT for each possible value of the 16-bit $Q3.12$ number would require 128kB, the last eight bits of the input are discarded to reduce the size of the LUT to 512 bytes. This results in both a fast and small implementation of the Sigmoid function.

Another advantage of the LUT is the fact that other complex activation functions like *tanh* would impose the same workload on the ASIPs, as no changes are introduced in the lookup process.

4.5.2 Reduction of memory footprint

The original MNIST dataset has an image size of 28 by 28 pixels of 1 byte each, with 10000 images in the test set. Each image is accompanied by a label of one byte that identifies the digit in the image. Besides the images and the labels themselves, the weights from the input layer to the hidden layer also contribute significantly to the memory used, as each node in the hidden layer is fully connected and thus there are initially 15680 weights.

To reduce the size of the images, labels, and number of weights in memory the following changes were made to the dataset:

- Only the first 50 images in the test set are used
- The images are scaled down to 7x7 pixels
- The 8-bit greyscale pixels are converted to 1-bit black and white and grouped into 32-bit integers
- The labels are squashed into one byte in pairs of two

Furthermore, the memory organisation of the layers, nodes, and weights is changed so no memory has to be dynamically allocated for the creation of the network.

4.5.3 Training

The weights and biases of the nodes in the network are generated on a desktop PC, using the full MNIST training data set of 60000 images and three epochs. The training program also uses the smaller images, $Q3.12$ number representation, and the reduced Sigmoid LUT. After training, the program writes the `structs` describing the network to a header file, which can be directly included in the benchmark.

4.5.4 Performance

Even though the images only use 0.78% of the data, and the precision of the calculation of the convolution has been reduced from a 64-bit `double` to a 16-bit fixed-point number, the CNN is able to classify 62% of the supplied images correctly.

Chapter 5

Experimental results

5.1 Simulations

Most architectural differences can be found by compiling and simulating the benchmarks for the selected processor cores. The compilations and simulations can be done completely by tools provided by the toolsets.

5.1.1 Simulation setup

Both ASIP Designer and TCE provide cycle-accurate simulators. From these simulators profiling information will be extracted that will provide insight on the performance and bottlenecks for the benchmarks.

Profiling

The cycle count and instruction count can be retrieved from both simulators. Both simulators also support scripting in TCL to automate the simulations. For ASIP Designer, the simulators are generated based on the processor description, and profiling information can be generated with the script in Listing 5.1.

For TCE, the simulator is fixed and loads the processor description in runtime. The commands passed to `ttasim` can be found in Listing 5.2.

Both compilers can be instructed to use maximum optimization and still produce profiling output. There is one caveat for TCE however. As TCE inlines most functions at the optimization level of 1 and higher, inlining has to be explicitly disabled. The generally has a negative impact on the cycle count, which can be observed in Table 5.2 in Section 5.1.2.

```
iss::create %PROCESSORNAME% iss
set n [incr argc -1]
set proc [lindex $argv $n]
# Enable profiling
iss profile set_active 1
iss profile storages_set_active 1
iss profile reset
iss profile storages_reset
# Load program
iss program load ./Release/impbench -disassemble -dwarf -nmlpath
↪  /home/jrhrsmit/impbench_tce/thesis-asip-projects/$proc/lib -extradisassembleopts +Mdec
↪  -do_not_set_entry_pc 1 -do_not_load_sp 1 -pm_check first -load_offsets {}
↪  -software_breakpoints_allowed on -hardware_breakpoints_allowed on
# Execute until finished
iss step -1
# Save profiling info
iss profile save profile_inst -type instruction_level
iss profile save profile_func -type function_level
iss profile storage_access_save -file profile_stor
iss close
exit
```

Listing 5.1: The TCL script used to simulate TCL cores and extract profiling information. First all profiling is enabled, after which the machine (i.e. processor) and program are loaded and executed. Finally, the cycle count and bus statistics are printed before exiting.

```
# Enable profiling
setting bus-trace 1
setting execution_trace 1
setting history_filename profile_hist
setting rf_tracking 1
setting utilization_data_saving 1
setting profile_data_saving 1
setting procedure_transfer_tracking 1
setting profile_data_saving 1
# Set machine and program
mach MACHINE
prog PROGRAM
run
info proc cycles
info proc stats
exit
```

Listing 5.2: The TCL script used to simulate TCE cores and extract profiling information. First all profiling is enabled in line 5-8, after which the program is loaded and executed. The program is then executed with `iss step -1`.

Grabbing standard output

Because the cores designed in ASIP Designer use memory-mapped standard output, a watchpoint will need to be configured in the simulator. This configuration of the watchpoint and the grabbing of the written data can be found in Listing 5.3.

As the PeLoTTA core has a standard output module which is supported by the simulator (`ttasim`), no special instructions to the simulator are necessary.

```
14 puts "Project: $prj DM: $dm hits: $hits"
15 iss program load ./Release/impbench -disassemble -dwarf -nmlpath
   ↪ /home/jrhrsmit/impbench_tce/thesis-asip-projects/$prj/lib -extradisassembleopts +Mdec
   ↪ -do_not_set_entry_pc 1 -do_not_load_sp 1 -pm_check first -load_offsets {}
   ↪ -software_breakpoints_allowed on -hardware_breakpoints_allowed on
16 iss watchpoint add $dm 4 -write true
17 set fp [open "sim_output" w]
18 for {set i 0} {$i < $hits} {incr i} {
19     iss step -1
20     set x [iss get $dm 4]
21     set x [format %c $x]
22     puts -nonewline $fp $x
23 }
24 iss close
```

Listing 5.3: A snippet from the TCL script used to simulate ASIP Designer cores and grab the standard output. The program is loaded on line 15, after which the watchpoint on address 0x04 can be configured. The program then runs until the watchpoint is hit with `iss step -1`, after which the data memory can be read out on address 4. The simulator will run until it has reached the specified amount of hits on the configured watchpoint.

5.1.2 Profiling

In order to get a better understanding of the simulation and measurement results, the assembly and instruction trace of the processors can be analysed. This will provide insight in the cause for differences in program sizes and execution cycle count.

The profiling information has been parsed and can be found in the appendix (chapter 7.1). The listings in the appendix show the program counter, assembly, execution count, and cycle count or inserted NOPs. For the PeLoTTA, the cycles used for an instruction is always the same as the execution count, and number of NOPs are shown instead. The profiling information has been parsed to show only the most executed part of the algorithms, without the functions used for data generation. Some listings are cut-off before the end when the loop of the algorithm is heavily unrolled.

Overhead

The percentage of cycles used for data generation can be found in Table 5.1. These percentages represent the percentage of clock cycles spent running instructions for data generation (i.e. `data_gen_buf()`, `data_gen_buf_low_entropy()`, or `rand()`). The Tdsp requires the most time for data generation, even though it has two load-store units. All cores spend a lot of time for data generation in CSUM and Motion, to the point where it dominates the total cycle count. Other benchmarks are performing better with most having a share for data generation of under 10%. The CNN benchmark does not use data generation, and thus has no overhead in Table 5.1.

The cycle counts listed in the rest of this chapter subtract the amount of cycles dedicated to data generation.

Table 5.1: Percentage of cycles used for data generation, which consist of calls to `data_gen_buf()` or `rand()` (for Motion). Note that PeLoTTA has 0% on Motion because it has inlined the `data_gen_buf()` function in that benchmark, even though `--disable-inlining` was specified. Also note that the Tdsp and Tvliw did not pass validation for CoreMark, thus no profiling is done for those cores.

Core	CRC32	CSUM	Finnish	MISTY1	RC6	Motion	CoreMark	CNN
PeLoTTA	13.64%	44.28%	3.37%	8.63%	8.03%	0%	0.12%	0%
Tdsp	10.56%	56.99%	3.53%	9.62%	7.10%	72.07%	-	0%
Tvliw	6.51%	32.32%	3.15%	4.10%	10.10%	32.41%	-	0%
Tzscale	6.53%	46.11%	3.27%	7.29%	6.36%	67.53%	0.06%	0%

Cycles used for the main function, writing to standard output, checking the endianness, and resetting the seed for data generation have virtually no impact on the cycle count, with a share of $< 0.01\%$ of the

total execution cycles for all benchmarks.

The difference in disabling inlining for TCE for the PeLoTTA core is shown in Table 5.2. For CSUM and Motion the overhead is very large, up to 313.6%. For Finnish however, it actually improves performance by about 3.6%.

Table 5.2: Difference in execution cycle count with inlining implicitly enabled and explicitly disabled for the PeLoTTA core.

Benchmark	Exe. cycles `-O2`	Exe. cycles `-O2 --disable-inlining`	Overhead
CRC32	105487	114485	8.5%
CSUM	13643	56434	313.6%
MISTY1	574044	604819	5.3%
Motion	602147	1161275	92.8%
RC6	312409	325836	4.2%
Finnish	478903	462523	-3.5%
CoreMark	50743586	51072092	0.6%

CRC32

Most cycles in the CRC32 benchmark consist primarily (89%-94%) of the for-loop in `update_crc()`, where the CRC is calculated. This loop is executed 11648 times. This for-loop is shown in Listing 5.4. Most instructions in this loop are either shifts, and-operations, xor-operations, and loading words from the CRC table. Note that although the source code contains integer divisions, these are either replaced by right-shifts or optimised away by the compiler.

```
164     for (i = 0; i < len; i++) {
165         b = buf[i / 4] & 0xFF;
166         buf[i / 4] >>= 8;
167         pos = (c >> 24) ^ b;
168         c = crc_table[pos] ^ (c << 8);
169     }
```

Listing 5.4: The most executed part of the CRC32 benchmark, the for-loop in `update_crc()`.

As can be seen by looking at the execution count for the instructions in the loop for PeLoTTA in Listing 7.1, the TCE compiler partially unrolls the loop. One iteration of the loop in the PeLoTTA is equivalent to eight iterations in the C source code. Although this does effect the code size, the PeLoTTA is able to complete one iteration of the original for-loop in 8.25 cycles on average. The instructions consist mostly of moves to and from the ALU. The PeLoTTA maintains a bus usage of 80% in this loop.

The left-shift that can be seen at PC=40 is for calculating the address for reading the CRC table. Finally, at PC=98, the conditional jump is calculated, and the following two instructions consist of increasing the loop iterator by 8.

As can be seen in Listing 7.2, the Tdsp completes one iteration in only 12 cycles, issuing 21 operations. As the Tdsp has four register files, many move operations are necessary to get the data to the registers accessed by the relevant FUs. Due to the high amount of shifting operations in the loop (`lsl`) and the dedicated shifter, the Tdsp is able to execute an add-operation in parallel with shift-operation twice. The Tdsp also uses two load-instructions (`ll` for loading 32-bit data) and one save-instruction (`sl`). As the Tdsp has hardware loop control, it does not require any use of branching instructions. Note that the execution count differs from Tzscale by 88 cycles, because the last call to `update_crc()` makes the for-loop iterate less. This, in combination with the hardware loop control, causes the compiler to separate the last iteration.

Listing 7.3 shows the profiling and assembly of the loop for the Tvliw. The four-slot VLIW core completes an iteration in 14 cycles, where it issues 37 NOP-instructions. Only four times in the loop it exploits its instruction-level parallelism and issues more than one operation in the same cycle.

Listing 7.4 shows the profiling information of the loop in `update_crc()` for the Tzscale. The Tzscale can perform one iteration of this loop in 16 cycles, executing 15 instructions. The only instruction using two cycles is the `bne` instruction to leave the for-loop. The loop only contains two load-word instructions,

and one save-word instruction, used for the read-modify-write on `buf[i / 4]` and the lookup in the CRC table. This leaves all other instructions for the ALU.

Table 5.3 summarises the profiling results for the for-loop in the CRC32 benchmark.

Table 5.3: Analysis of the for-loop in `update_checksum()`, in the CRC32 benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	8.12	19.62	2.42	8
Tdsp	12.00	21.00	1.75	1
Tvliw	15.90	16.88	1.06	1
Tzscale	15.99	15.00	0.94	1

CSUM

The CSUM benchmark workload can also be summarised in a short for-loop. Listing 5.5 shows the loop, which contains two additions, two negations, an AND-operation, and a shift operation.

```
33      for(i = 0; i < len / 4; i++) {
34          sum += ~(data[i] & 0xFFFF);
35          sum += ~(data[i] >> 16);
36      }
```

Listing 5.5: The most executed part of the CSUM benchmark, the for-loop in `update_checksum()`.

Listings 7.5, 7.6, 7.7, and 7.8 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

The PeLoTTA unrolls this loop completely for the batch-size of 8 iterations, and even combines the next function call with it, resulting in 16 unrolled loop iterations. Because there are so few operations done in this loop, the PeLoTTA uses 171 instructions for 16 iterations. The Tdsp and Tzscale also completely unroll the loop for the batch-size of 8 iterations. The Tvliw, lacking the LLVM front-end, does not unroll the loop and executes it in 7 cycles. The Tvliw issues 17 NOPs and achieves an instructions per cycle (IPC) in this loop of 1.43. The average amount of cycles and operations for one loop iteration can be seen in Table 5.4.

Table 5.4: Analysis of the for-loop in `checksum()`, in the CSUM benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	10.62	24.12	2.27	16
Tdsp	7.65	10.62	1.39	8
Tvliw	8.34	9.11	1.09	1
Tzscale	8.77	8.50	0.97	8

MISTY1

The function to encrypt the data in the MISTY1 benchmark is `misty1_encrypt_block()`. `misty1_encrypt_block()` mostly consists of calling two other functions; `fl()` is called ten times, and `fo()` is called eight times. Inside `fo()` another function `fi()` is used and is called three times. `fl()`, `fo()`, and `fi()` are heavy on arithmetic with many data dependencies, as can be seen in Listing 5.6.

```
111 static u4 fi(u4 fi_in, u4 fi_key) {
112     u4 d9, d7;
113 
114     d9 = (fi_in >> 7) & 0x1ff;
115     d7 = fi_in & 0x7f;
116     d9 = s9[d9] ^ d7;
117     d7 = (s7[d7] ^ d9) & 0x7f;
118 
119     d7 = d7 ^ ((fi_key >> 9) & 0x7f);
120     d9 = d9 ^ (fi_key & 0x1ff);
121     d9 = s9[d9] ^ d7;
122     return ((d7 << 9) | d9);
123 }
124 
125 static u4 fo(u4* ek, u4 fo_in, byte k) {
126     u4 t0, t1;
127     t0 = (fo_in >> 16);
128     t1 = fo_in & 0xFFFF;
129     t0 ^= ek[k];
130     t0 = fi(t0, ek[((k + 5) % 8) + 8]);
131     t0 ^= t1;
132     t1 ^= ek[(k + 2) % 8];
133     t1 = fi(t1, ek[((k + 1) % 8) + 8]);
134     t1 ^= t0;
135     t0 ^= ek[(k + 7) % 8];
136     t0 = fi(t0, ek[((k + 3) % 8) + 8]);
137     t0 ^= t1;
138     t1 ^= ek[(k + 4) % 8];
139     return ((t1 << 16) | t0);
140 }
141 
142 static u4 fl(u4* ek, u4 fl_in, byte k) {
143     u4 d0, d1;
144     byte t;
145 
146     d0 = (fl_in >> 16);
147     d1 = fl_in & 0xffff;
148     if (k % 2) {
149         t = (k - 1) / 2;
150         d1 = d1 ^ (d0 & ek[((t + 2) % 8) + 8]);
151         d0 = d0 ^ (d1 | ek[(t + 4) % 8]);
152     } else {
153         t = k / 2;
154         d1 = d1 ^ (d0 & ek[t]);
155         d0 = d0 ^ (d1 | ek[((t + 6) % 8) + 8]);
156     }
157     return ((d0 << 16) | d1);
158 }
```

Listing 5.6: `fl()` and `fo()` in `misty1.c`, which are the functions in which most CPU cycles are spent.

Listings 7.9, 7.10, 7.11, and 7.12 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

The LLVM front-end inlines the `fi()`, `fo()` and `fl()` functions inside the `Compress()` function for optimisation. For the Tvliw this is not the case, and the instruction execution count gives away which instructions belong to the `fi()` function, with 24k executions. The other processors execute the instructions 1025 times (1024 calls to `Compress()` to encrypt the data plus one execution for the key generation). The impact of function inlining can also be seen in the amount of cycles necessary for one call to `Compress()` in Table 5.5.

Table 5.5: Analysis of the `fl()` and `fo()` functions called from `misty1_encrypt_block()` in the MISTY1 benchmark. The data is averaged for one call.

Core	Cycle count	Operations	Avg. ops/cycle
PeLoTTA	538.00	1245.00	2.31
Tdsp	601.59	940.00	1.56
Tvliw	1043.17	1707.23	1.64
Tzscale	585.57	581.00	0.99

Motion

In the Motion benchmark the generated pseudo-random data is compared to a threshold, as seen in Listing 5.7.

```
65      for (sample =0; sample < MONITORING_SAMPLE_PERIODS; sample++) {
66          // enter new monitoring mode
67          while (active_counter < MONITORING_SAMPLES) {
68              // read new sensory data
69              readout = rand();
70              // check if new readout is above the threshold value
71              if (readout >= threshold)
72                  motion_counter++;
73              active_counter++;
74          }
75          // add "active measurements" to result
76          res += motion_counter;
77          // reset counters for next monitoring session
78          active_counter = 0;
79          motion_counter = 0;
80          // idle mode is removed
81      }
```

Listing 5.7: The for-loop in `motion.c`, which simulates the readout of a sensor and detecting motion by means of a threshold. `MONITORING_SAMPLES` is set to 20, and `MONITORING_SAMPLE_PERIOD` to 4096. The generation of the pseudo-random sensor data costs most processors more cycles than the operations in this loop.

Each time the threshold is exceeded, the `motion_counter` is incremented. This comes down to one or two operations per random word; a combined comparison/branch instruction and an optional addition. The nested while-loop in the for-loop can be seen as a for-loop with `MONITORING_SAMPLES` iterations, which is defined as 20. Combined with the first for-loop which iterates `MONITORING_SAMPLE_PERIOD` = 4096 times, the threshold is evaluated 81920 times. Note that Listing 5.7 could be rewritten to:

```
for (int i = 0; i < 81920; i++) {
    if (rand() >= threshold)
        res++;
}
```

Listings 7.13, 7.14, 7.15, and 7.16 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

The Tdsp and Tzscale unroll the inner while-loop with 20 iterations completely. The PeLoTTA also unrolls the outer for-loop once, resulting in only 2048 executions of the unfolded instructions. The resulting average cycles per iteration can be found in Table 5.6.

Due to the lack of optimisation and the jump instruction that requires three cycles to complete, the Tvliw requires the most cycles per iteration. The Tdsp inserts a NOP when jumping, which happens every 5 cycles when calling the `rand()` function. The PeLoTTA only requires three instructions to execute threshold comparison and the call to `rand()`. The Tzscale only requires 5 instructions (plus one cycle for the jump) for the iteration.

Note that because the threshold is defined as `0x80000000`, the comparison is simply compiled as a right-shift of 31 bits of the random number. The result is 1 if the number is equal to or greater than the

threshold, and 0 otherwise. The result can then directly be used in the addition, preventing the use of a branch instruction. The Tvliw is the only processor that does not use this optimisation.

Table 5.6: Analysis of the for-loop in the Motion benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	3.15	6.25	1.98	40
Tdsp	6.20	6.30	1.02	20
Tvliw	13.95	12.49	0.90	1
Tzscale	6.25	5.20	0.83	20

RC6

The RC6 benchmark's encryption algorithm is mostly contained in one for-loop in `RC6_32_encrypt()`, where it calls the `ROTATE_l32()` preprocessor macro to operate on the data. The `RC6_32_encrypt()` function itself is called 513 times. The loop runs for 20 iterations (`key->rounds` = 20), and the . The loop can be seen in Listing 5.8. The macro is defined as follows:

```
#define ROTATE_l32(a, n) \
    (((a) << (n & 0x1f)) | (((a)&0xffffffff) >> (32 - (n & 0x1f))))
```

```
261     for (i = 1; i <= key->rounds; i++) {
262         t = ROTATE_l32((((B * B) << 1) + B), LOGW);
263         u = ROTATE_l32((((D * D) << 1) + D), LOGW);
264         A = ROTATE_l32((A ^ t), u) + key->S[i << 1];
265         C = ROTATE_l32((C ^ u), t) + key->S[(i << 1) + 1];
266
267         t = A;
268         A = B;
269         B = C;
270         C = D;
271         D = t;  // permute
272     }
```

Listing 5.8: The for-loop in `RC6_32_encrypt()` in `rc6.c`, in which four words of data are encrypted. `LOGW` is set to 5.

The 32-bit mask (the `a & 0xffffffff`) in the macro is actually useless as all variables in the loop are declared as 32-bit unsigned integers. The 5-bit mask in `ROTATE_l32` in lines 262 and 263 also contribute nothing to the result as the operand `n` is set to 5 (`LOGW` is set to 5). The first two uses for the macro can therefore be reduced to:

```
#define ROTATE_l32(a, n) \
    ((a << 5) | (a >> 27))
```

Listings 7.17, 7.18, 7.19, and 7.20 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

The resulting average cycles per iteration can be found in Table 5.7.

The LLVM front-end indeed reduces the `ROTATE_l32` macro for the Tzscale, as can be seen at PC=790 and PC=798. Apparently, the masking of the `n` operand is also unnecessary for the later two uses of the macro, as the loop does not contain any and-operations. Note that the loop also contains two multiplication instructions to calculate the square of `B` and `D` (line 262-263 in Listing 5.8).

`tcecc` partially unrolls the loop in order to skip the explicit permutation step at the end of the loop, resulting in 2565 loop executions. The other processors execute the loop 10260 times. Because of the few data dependencies in the loop, the Tvliw, Tdsp, and PeLoTTA are able to schedule more operations per cycle than other benchmarks. Especially the ALUs of the Tvliw are highly occupied, scheduling two arithmetic operations for 14 of the 17 instructions.

Table 5.7: Analysis of the for-loop in `RC6_32_encrypt()` in the RC6 benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	26.50	66.75	2.52	4
Tdsp	40.00	55.00	1.37	1
Tvliw	18.24	32.48	1.78	1
Tzscale	31.95	31.00	0.97	1

Finnish

Using the `Compress()` function in `fin.c`, 11740 bytes of data in blocks of 128 bytes is compressed. The `Compress()` function iterates over each byte of the block of data, and creates a prediction table for compression. The loop is shown in Listing 5.9. The loop contains two if-statements where the prediction is evaluated and where the mask (which indicates the correctness of the prediction) and characters are written to the output on every eighth character.

```
51      for (j = 0; j < a_len; j++) {
52          c = a[j / 4] & 0xFF;
53          a[j / 4] >>= 8;
54          // try to predict the next character
55          if (pcTable[INDEX(p1, p2)] == (char)c) {
56              // correct prediction, mark bit for correct prediction
57              mask = mask ^ (1 << ctr);
58          } else {
59              // wrong prediction, but next time ...
60              pcTable[INDEX(p1, p2)] = (char)c;
61              // buf keeps character temporarily in buffer
62              buf[bctr++] = (char)c;
63          }
64          // test if mask is full (8 characters read)
65          if (++ctr == 8) {
66              // write mask
67              b[k++] = mask;
68              // write kept characters
69              for (i = 0; i < bctr; i++) {
70                  if (k > b_len) {
71                      print_str("out of bounds write on b\n");
72                      break;
73                  }
74                  b[k++] = buf[i];
75              }
76              // reset variables
77              ctr = 0;
78              bctr = 0;
79              mask = 0;
80          }
81          // shift characters
82          p1 = p2;
83          p2 = (char)c;
84      }
```

Listing 5.9: The for-loop in the `Compress()` function in `fin.c`.

Listings 7.17, 7.18, 7.19, and 7.20 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

The profiling information reveals that the correct prediction in the if-statement on line 55 is made at 5963 of 11740 characters.

In contrast to the other benchmarks, the LLVM compiler chooses not to unroll the loop, even for the PeLoTTA. The Tdsp and Tzscale do separate the calls to `Compress()` with 128 bytes and the last call with the 92 remaining bytes.

The resulting average cycles per iteration can be found in Table 5.7.

Table 5.8: Analysis of the for-loop in `Compress()`, in the Finnish benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	35.40	71.32	2.01	1
Tdsp	36.43	44.43	1.22	1
Tvliw	30.45	29.45	0.97	1
Tzscale	30.21	26.43	0.87	1

CoreMark

CoreMark consists of four functions which contribute to > 95% of the total execution cycle count. These functions represent the three benchmarks CoreMark consists of, and the CRC for validation of the results. Table 5.9 shows the amount of clock cycles used for each function and the percentage of the total amount of execution cycles. The Tdsp and Tvliw are not listed because their validation fails, and thus the profiling information is irrelevant.

The CRC is relatively heavy for the PeLoTTA, using twice the amount of cycles the Tzscale needs. At the list processing benchmark the PeLoTTA is almost as fast as the Tzscale, but it falls behind with 10% and 20% extra cycles for the state machine and matrix benchmarks.

Table 5.9: Share of cycles used for functions in CoreMark for the cores that passed validation.

Core	State machine	List processing	Matrix	CRC	Others
PeLoTTA	170128 (33.3%)	122991 (24.1%)	128904 (25.2%)	64093 (12.6%)	24605 (4.8%)
Tzscale	155794 (36.8%)	118377 (25.9%)	102940 (28.0%)	26082 (6.3%)	8214 (3.0%)

CNN

The `calcNodeOutput()` function is used to calculate the outputs of each node in a layer of the CNN. The function is called 1500 times in total, and about 70% of all execution cycles are spent on this function. The function is called 20 times for each of the 50 images for the nodes in the hidden layer, and 10 times for the nodes in the output layer. The loops iterate over the outputs of the nodes of the previous layer, so the loop iterating over the hidden layer costs more cycles as the input layer has 49 outputs, and the hidden layer has 20 outputs. This loop can be seen in Listing 5.10.

The activation function (`q_sigmoid()`) is only responsible for >1% of the total cycle count.

```
242         for (int i = 0; i < nn->input_size; i++) {
243             // get the pixel value, i.e. the previous layer (input layer)'s
244             // output.
245             uint32_t out = nn->input_img[it0] & ((uint32_t)1 << it1);
246             it1++;
247             if (it1 > mnist_data_width) {
248                 it1 = 0;
249                 it0++;
250             }
251             // we reduced the value to 1 bit, so the entire addition to weights
252             // is discarded if the value is 0. Also we don't have to multiply
253             // with the value anymore as 1*weight = weight.
254             if (out) {
255                 calcNode->output =
256                     q_add(calcNode->output, calcNode->weights[i]);
257             }
258         }
```

Listing 5.10: The for-loop for the hidden layer in the `calcNodeOutput()` function in `cnn.c`.

Listings 7.25, 7.26, 7.27, and 7.28 show the profiling information for the PeLoTTA, Tdsp, Tvliw, and Tzscale, respectively.

For all processors except the Tvliw, the `calcNodeOutput()` function and the `calcLayer()` function that calls it are inlined, but the loops are not unrolled. The PeLoTTA contains two instructions for the loop where it inserts no move operations on the buses. This is necessary for the branch and load instructions on line 254 and 256 in Listing 5.10. Despite the inserted NOPs, the PeLoTTA completes the loop in about the same cycles as the other processors. The resulting average IPC and cycles per iteration for the loop can be found in Table 5.7.

Table 5.10: Analysis of the for-loops in `calcNodeOutput()`, in the CNN benchmark. The data is averaged for one loop iteration.

Core	Cycle count	Operations	Avg. ops/cycle	Unroll factor
PeLoTTA	19.00	39.00	2.05	1
Tdsp	19.00	26.00	1.37	1
Tvliw	22.55	17.44	0.77	1
Tzscale	19.88	17.96	0.90	1

5.1.3 Simulation results

Besides profiling, the simulations of the cycle-accurate simulators can provide insight on the following subjects, which will be presented in this section:

- Correct execution of the benchmark
- Program size in instruction memory
- Execution cycle count
- IPC

Validation

All processor implementations running benchmarks in ImpBench pass the validation checks for both ImpBench-like data and pseudo-random data, at all evaluated clock frequencies.

For CoreMark however, only the PeLoTTA and Tzscale pass validation. Tvliw and Tdsp do not pass the validation as CoreMark fails when the size of an integer does not equal 4. For Tvliw and Tdsp, this both equals 1, as their data memory is only addressable in words. The Tzscale and PeLoTTA do pass all checks, and their results are verified with a CRC.

Program size

The instruction set, compiler, and compiler configuration have impact on the compiled program size. A larger program size will require more program memory, and thus more area for an implementation on an ASIC. Figure 5.1 shows the occupied memory for each benchmark in ImpBench and for CoreMark.

The PeLoTTA is the least economical with the program memory, using the most space in every benchmark except for CSUM. The other VLIW processor with 64-bit instruction words, Tvliw, comes in at third place. The Tdsp and Tzscale occupy the least amount of program memory, using half the space for CoreMark.

Note that the memory size is often a power of two so that it is fully addressable with the size of the program counter. This is not the case for the PeLoTTA however, as the instruction word itself is 45 bits and thus not a power of 2. The memory sizes for fully-addressable program memory are displayed on the right axis of Figure 5.1. This implies that the PeLoTTA would require 2880B of program memory for the CRC32 benchmark, where the Tzscale only needs 512B.

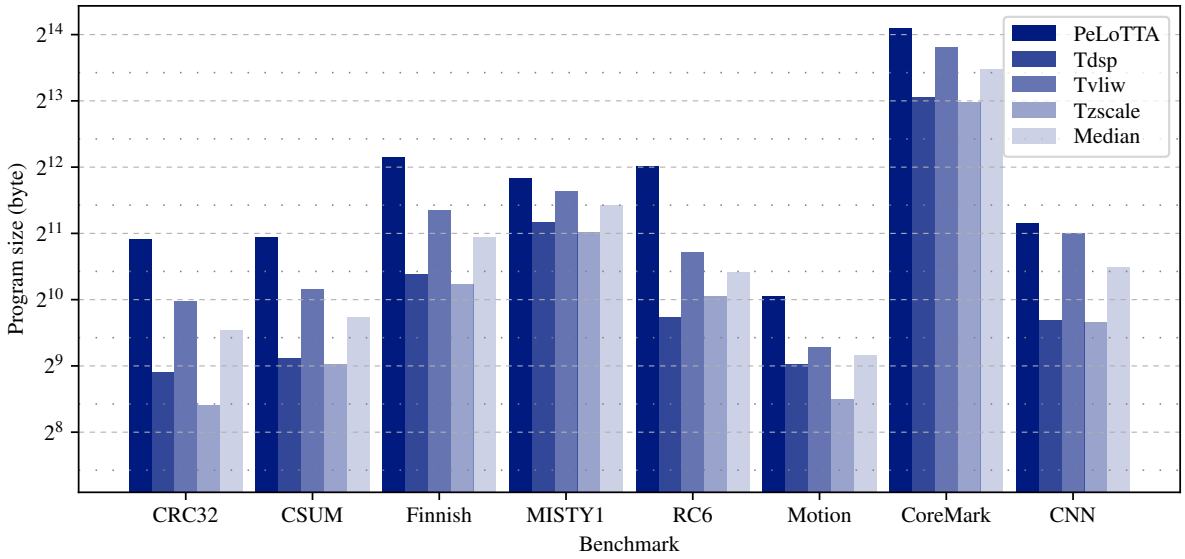

Figure 5.1: Program sizes in the instruction memories for the benchmarks in ImpBench and CoreMark for each core. The labels on the left axis indicates program memory sizes for fully addressable memory on the Tdsp, Tvliw, and Tzscale. The labels on the right axis indicate the program memory sizes for fully addressable memory for the PeLoTTA. Note that the program size is plotted logarithmically.

Execution cycles

The amount of execution cycles for the benchmarks can be found in Figure 5.2 and Figure 5.3 for the cycle count relative to the median. The cycle counts have been adjusted to exclude the cycles related to data generation (see Table 5.1).

The results vary greatly with each benchmark, with no single processor being the fastest or slowest for all benchmarks.

PeLoTTA requires the fewest amount of cycles in CRC32, MISTY1, and CNN. The Tzscale takes the cake in Finnish and CoreMark. The Tzscale and PeLoTTA require on average the fewest cycles to execute the benchmarks. The Tvliw is the slowest in CRC32, CSUM, MISTY1, and Motion, where in MISTY1 it needs more than twice the cycle count as the other processors. Remarkably the Tvliw outperforms the other cores by more than 20% in RC6. The Tdsp the fastest in both CSUM and Motion, but needs the most cycles for Finnish and RC6.

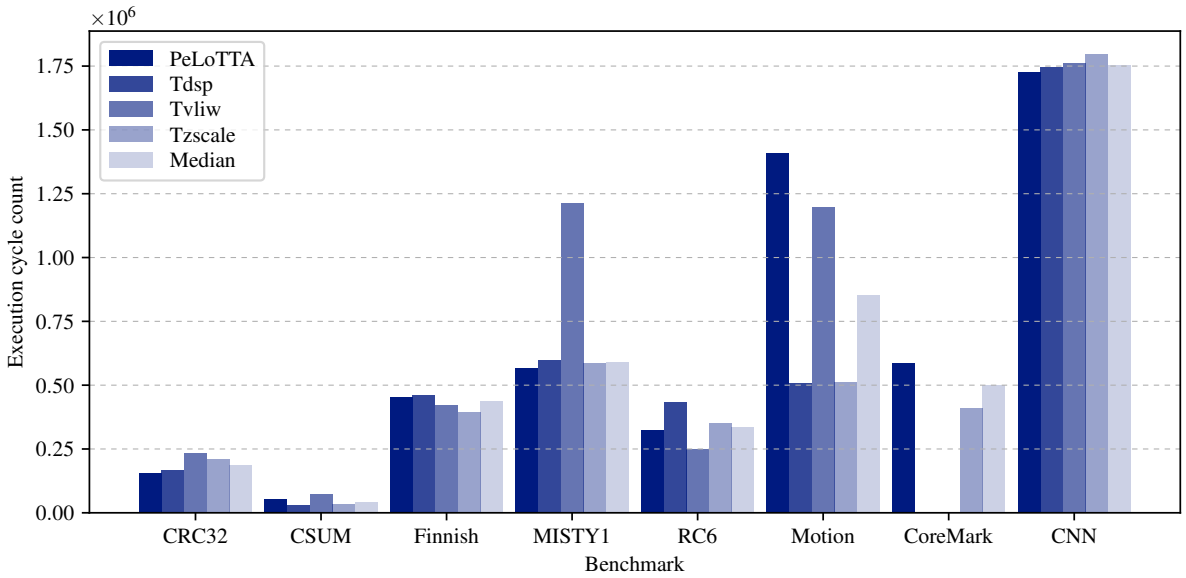

Figure 5.2: Execution cycle count for all benchmarks. Cycles dedicated to data generation are excluded. Note that the execution cycles for the CoreMark benchmark for the Tvliw and Tdsp are not displayed as they failed the validation.

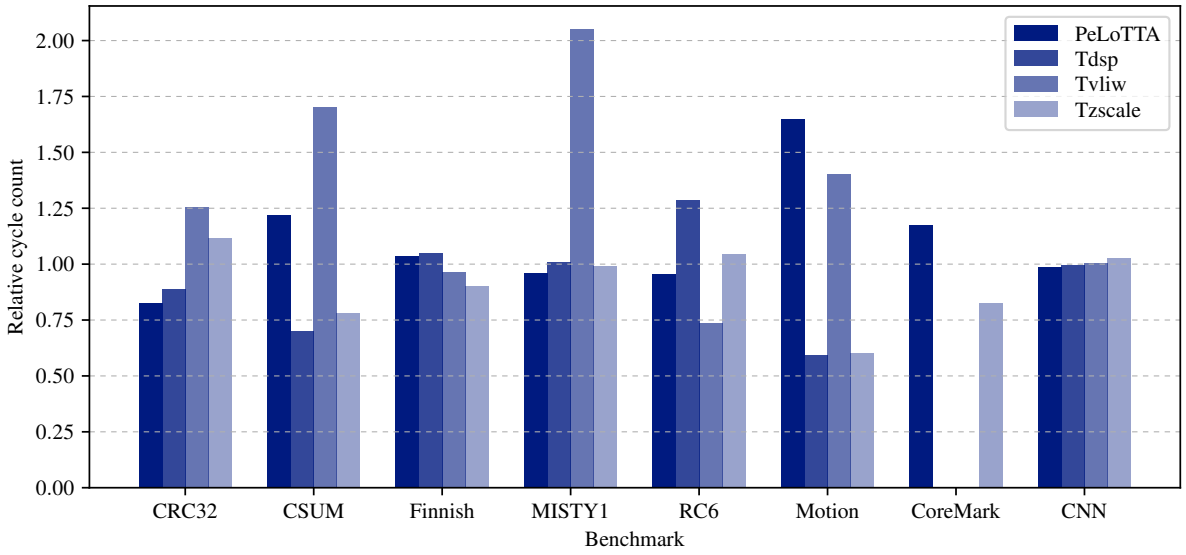

Figure 5.3: Execution cycle count for all benchmarks, relative to the median. Cycles dedicated to data generation are excluded. Note that the execution cycles for the CoreMark benchmark for the Tvliw and Tdsp are not displayed as they failed the validation.

With the execution cycle count the CoreMark score per MHz can be calculated. The CoreMark score is defined as number of CoreMark iterations per second, but as this is dependent on the clock frequency, only the CoreMark score per MHz is used. The execution cycle count, CoreMark scores per MHz, and validation results can be found in Table 5.11. The Tvliw and Tdsp are included for completeness, but do not pass validation. Note that the CoreMark cycles in Table 5.11 is for 100 iterations to ensure the benchmark is running for more than 10 seconds, as EEMBC requires.

The calculated CoreMark score for the Tzscale is just slightly higher than the CoreMark score provided by ASIP Designer, which is 2.39/MHz.

Table 5.11: CoreMark scores for the selected cores. These scores are unofficial and remain unsubmitted to the EEMBC. Note that the Tvliw and Tdsp did not pass the validation check, and thus their scores are unusable.

Core	CoreMark cycles (100 it.)	CoreMark score/MHz	Validation check
PeLoTTA	51072092	1.96	Pass
Tzscale	41430350	2.43	Pass
Tvliw	67176721	1.49	Fail
Tdsp	213530479	0.47	Fail

Instructions per cycle

The IPC for each processor for the benchmarks in ImpBench and CoreMark can be found in Figure 5.4.

Even though PeLoTTA has a relatively low IPC for the Motion benchmark, it requires the least amount of cycles. The Tdsp core always maintains an IPC of 1, as it only stalls the pipeline when jumping to an unaligned address. The Tzscale also maintains a very high IPC, with the lowest IPC for Finnish, even though it uses the least amount of execution cycles for that benchmark. The same disassociation goes for Tvliw, which has low IPC for CSUM and Finnish, but performs well in Finnish and substandard in CSUM.

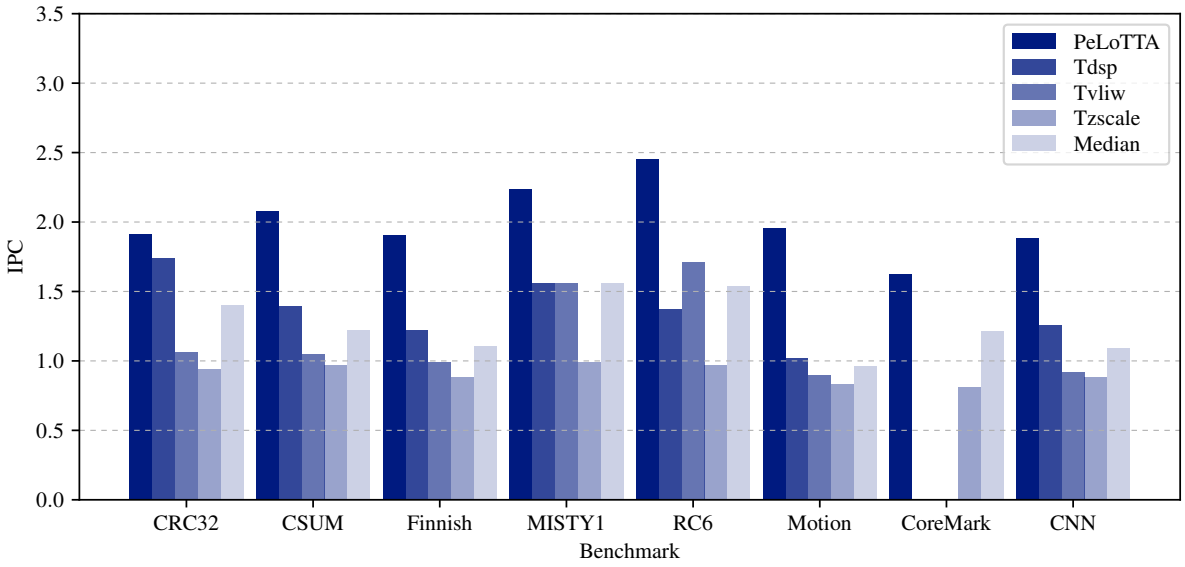

Figure 5.4: Instructions per cycle for the benchmarks in ImpBench and CoreMark. Cycles dedicated to data generation are excluded. Note that the IPC for the CoreMark benchmark for the Tvliw and Tdsp are not displayed as they failed the validation.

5.2 Power measurements

In order to get an indication of the power consumption of the cores for the different workloads, the power of the processors is measured after synthesis on an FPGA.

5.2.1 Fitting results

Table 5.12 shows the resource usage of the Cyclone IV FPGA, extracted from the fitting report produced by Quartus. The Tvliw core requires the most area, which comes as no surprises as it boasts two ALUs and two LSUs. The PeLoTTA is functioning with roughly one third of the logic elements used by the Tvliw.

The result of two 32-bit integer multiplication is 64-bits long, and thus requires eight 9-bit multipliers. The Tzscale is the only processor where the top 32-bits of the result can be used, the other processors

only generate the bottom 32-bits. Note how the Tvliw has two ALUs with each their own multiplier, and thus uses many multiplier elements.

Table 5.12: Reported resource usage by Quartus after fitting the design on the target.

	Logic elements	Combinational functions	Dedicated logic registers	Total Registers	Memory bits	9-bit multipliers
PeLoTTA	3262	3139	1555	1555	438784	6
Tdsp	7261	6019	1477	1242	393728	4
Tvliw	9223	9057	1564	1564	393728	12
Tzscale	4718	4648	1477	1477	393728	8

5.2.2 Timing results

Quartus Prime can do multi-corner timing analysis on the design. The corner which constrains the clock frequency the most is the Slow 0C 1200mV model, which estimates the timing for a process with slow silicon, a core voltage of 1200mV, and a core temperature of 0°C. These estimations are shown in Table 5.13.

Table 5.13: Maximum clock frequencies reported by the timing analysis for the Slow 0C model.

Core	Fmax [MHz] for the Slow 0C model	Critical path
PeLoTTA	45.9	Inside the IRF
Tdsp	37.5	Decoder to register file (reg R)
Tvliw	49.5	Decoder to register file (reg R)
Tzscale	32.9	Decoder to instruction memory address

5.2.3 Development board

The development board used for the FPGA is the QMTech "ep4ce15f23-sdram-v2" [100]. The board is very basic, and only contains:

- 3 Buck converters (5V to 3.3V, and 3.3 to 2.5V and 1.2V)
- 50MHz Crystal
- 64Mbit EEPROM
- 256Mbit SDRAM
- Three buttons (one of which is wired to the `nCONFIG` input)
- Two LEDs

The EEPROM and SDRAM are not be used. The EEPROM is not necessary for evaluating power consumption as it is not necessary to retain the FPGA configuration through power cycles, and the SDRAM would unnecessarily complicate the design for extra memory. The LEDs and buttons also remain unused, as all necessary interfacing with the implemented processors is done with standard output over the debugger.

5.2.4 Power measurement setup

The measurement setup for the FPGA consists of six items:

- The FPGA development board (QMTech ep4ce15f23-sdram-v2)
- A USB programmer/debugger (Altera USB blaster II)
- A linear triple-output power supply (ITECH IT6322A)
- A bench-top 5.5 digit digital multimeter (Keysight 34450A)
- A logic analyzer for debugging purposes (DSLogic Plus)
- A server running Linux

The triple-output power supply is not only used to power the board, but also control the reset input of the synthesized processor on the FPGA. The power supply is directly connected to the 3.3V rail on

the board, as 5V is not required. Because only one multimeter is available, the 1.2V rail is powered by the on-board buck converter from 3.3V. Although the power supply can measure the supplied current to the FPGA, the digital multimeter can provide more accurate measurement of 1µA at a range of 100mA. The power supply supports external sensing of the regulated output voltage, so the voltage drop over the multimeter does not influence the results. An overview of the test and measurement setup is shown in Figure 5.5. The logic analyzer is used for extracting instruction traces to debug incorrect executions.

The devices are controlled over USB by a Python script using PyVISA. This enables the use of more Python scripts which automate the measurement flow of synthesis, programming, measurements, and validation. The validation is performed by reading the standard output over the debugger, the instruction trace is not automatically verified. All scripts can be found on Gitlab [101].

All measurements are done with the full range of 5.5 digits on 100mA, after the multimeter has been powered on for 90 minutes. This provides a maximum error of about 57µA. As the measurements are just an indication of the power consumption of the circuit implemented on an ASIC, this setup provides enough accuracy.

The total power consumption can be divided in two categories; static and dynamic power consumption (Equation 5.1). For each core and benchmark combination, a null measurement is done to determine the static power consumption by asserting the reset of the core. The dynamic power scales with capacitance, clock frequency, switching activity, and voltage squared (Equation 5.2), so when the voltage and frequency are kept constant the measured dynamic power consumption directly correlates to the capacitance of the traces and transistor gates. Note that this is a very simple model, which does not include switching activities.

$$P_{total} = P_{static} + P_{dynamic} \tag{5.1}$$

$$P_{dynamic} = \alpha \cdot C \cdot f \cdot V^2 \tag{5.2}$$

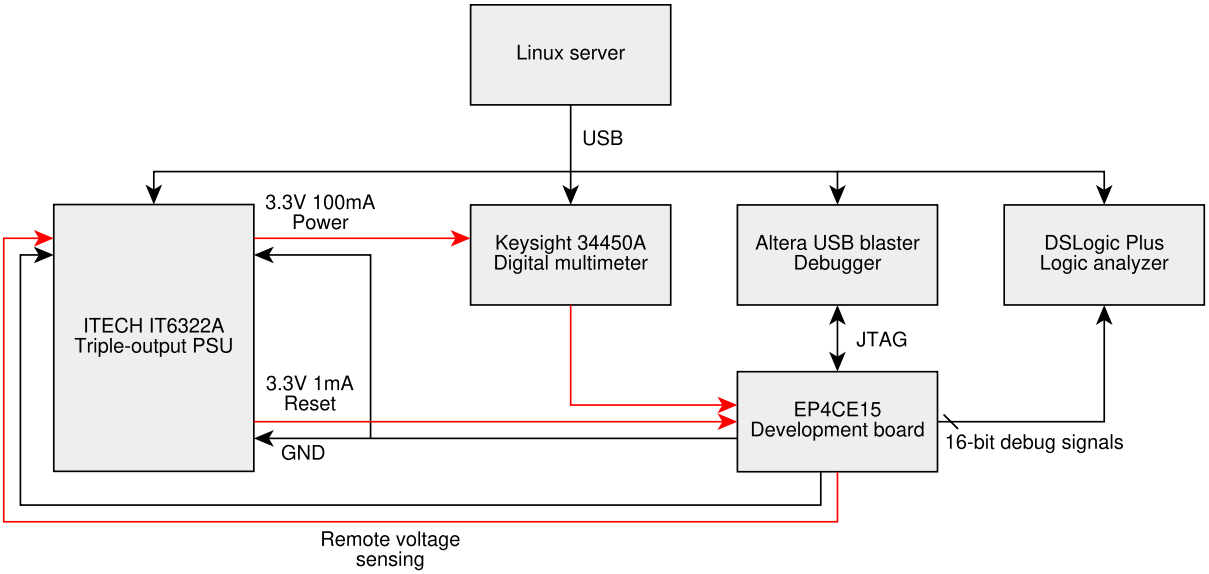

Figure 5.5: The test and measurement setup for acquiring power measurement results for the processor cores.

5.2.5 Power measurement results

Table 5.14 shows the measured static power consumption. It can be seen that the static power consumption is very similar, and that larger cores use slightly more power. The dynamic power consumption is then calculated by subtracting the static power, and is shown in Figure 5.6. Standard deviations in the graphs are not shown, as they are too insignificant.

Table 5.14: Static power consumption measurements with standard deviations for 210 measurements per core.

Core	Static power consumption [mW]	Std. dev. [mW]
PeLoTTA	154.2	0.027
Tdsp	154.4	0.045
Tvliw	154.6	0.041
Tzscale	154.1	0.060

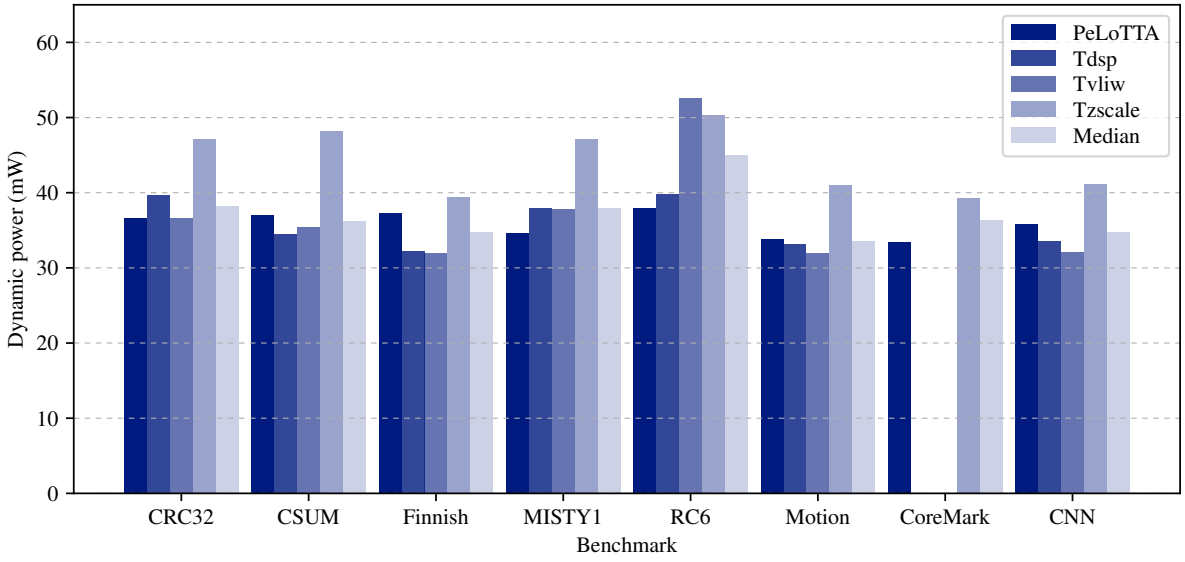

Figure 5.6: Average instantaneous dynamic power consumption of the processors for each benchmark.

In Figure 5.7 the relationship between power consumption and clock frequency can be seen for the CRC32 benchmark and CoreMark. The other benchmarks follow a similar pattern. Figure 5.9 shows the energy consumed, and Figure 5.10 shows the energy-delay product (EDP) relative to the median for the benchmarks. Figure 5.8 shows the CoreMark scores per MHz for the PeLoTTA and the Tzscale, and the CoreMark scores per MHz per Watt. Figure

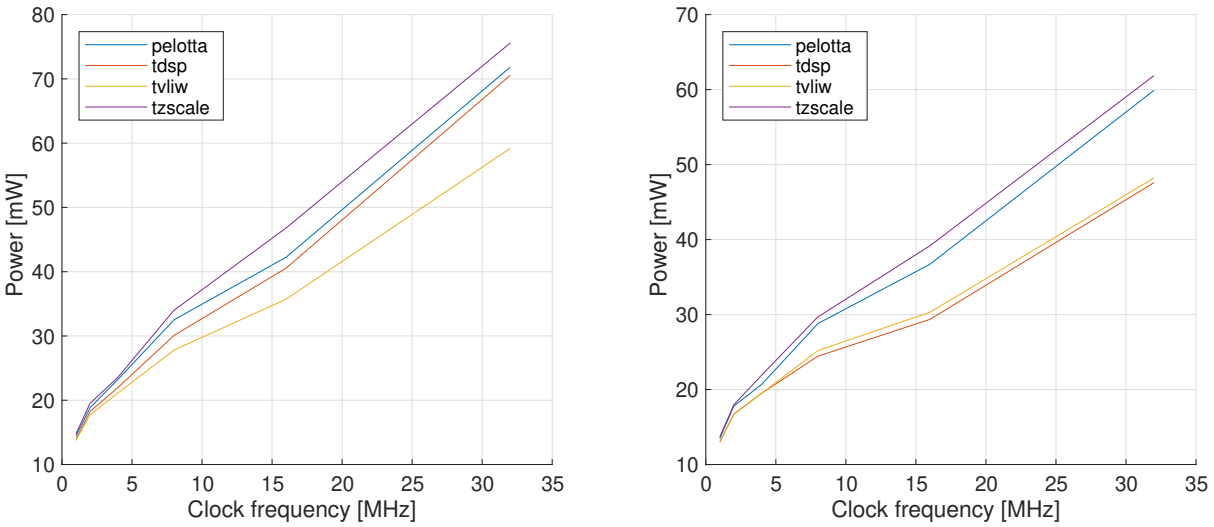

Figure 5.7: Dynamic power consumption for the CRC32 benchmark in ImpBench (left) and CoreMark (right), measured from 1 to 32MHz with intervals in powers of 2.

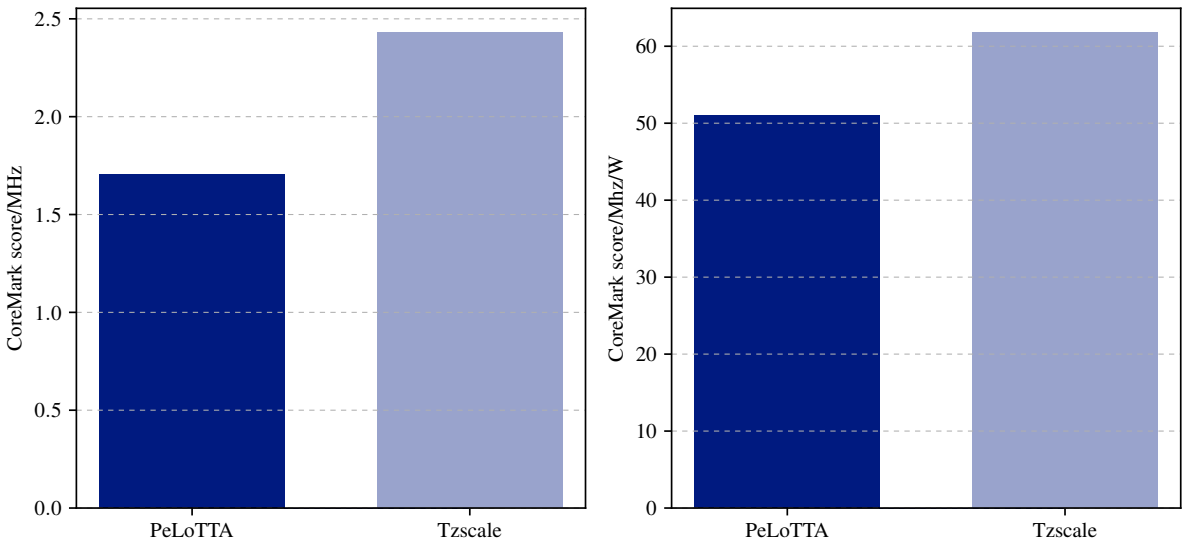

Figure 5.8: CoreMark scores per MHz and CoreMark score per MHz per Watt. Note that the results for the Tvliw and Tdsp are not shown as they failed the validation.

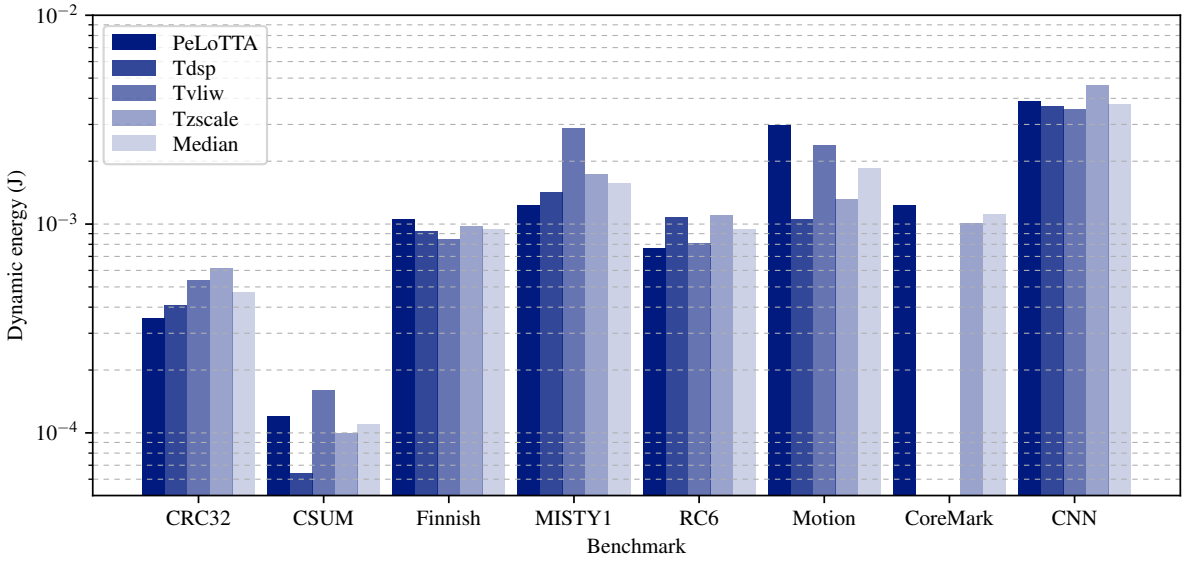

Figure 5.9: Energy consumption of the execution of the benchmarks. Note that the results for the Tvliw and Tdsp are not shown as they failed the validation.

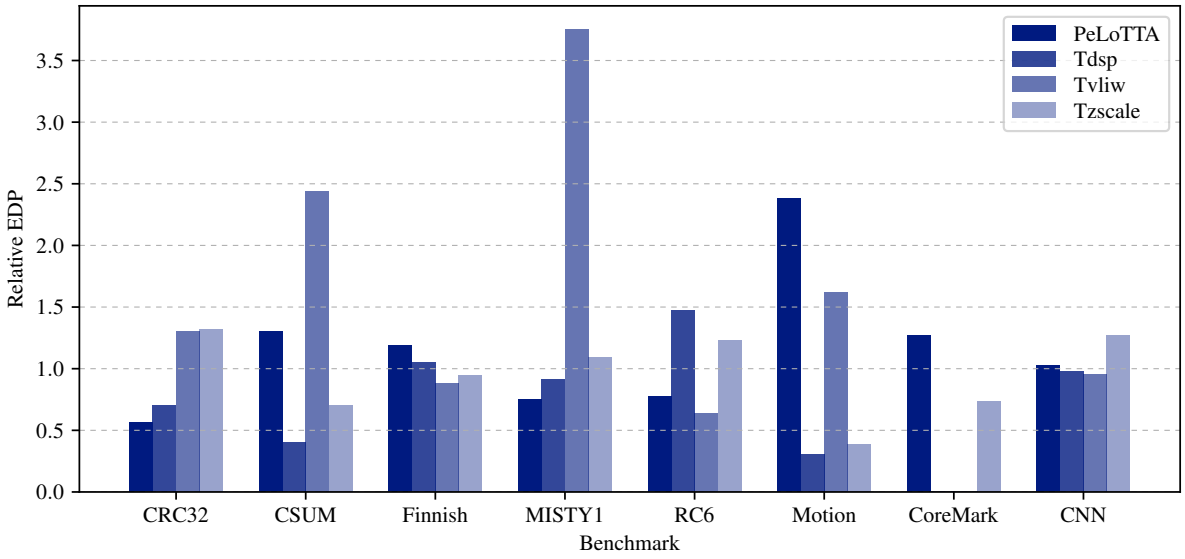

Figure 5.10: Energy-delay product of the processor implementations relative to the median for each benchmark. Note that the results for the Tvliw and Tdsp are not shown as they failed the validation.

5.3 Discussion

This section focusses on the interpretation and validity of the results from sections 5.1 and 5.2.

5.3.1 Benchmarks

Some of the benchmarks in ImpBench were left unimplemented, as they either contained too many floating point instructions (DMU) or required to much memory (MLZO). This makes Motion the only benchmark for real applications, and Finnish the only benchmark representing compression workloads.

Looking at the percentage of cycles spent for data generation in Table 5.1, it appears total execution cycles for both Motion and CSUM are dominated by data generation. This is because there are very few operations performed on the generated data in these benchmarks. Although the cycles dedicated to data generation are ignored in Figure 5.2 and 5.3, they do have impact on the power measurements. This shifts the representation of these benchmarks from the relevant workload to generation of pseudo-random data which makes the power results of these benchmarks inaccurate. The disqualification of these benchmarks leads to the lack of any benchmarks in ImpBench representing real workloads, and leaving CRC32 and Finnish the only benchmarks for their workload.

CoreMark could be seen as a partial substitution for Motion and DMU, as it contains classical workloads such as traversing state machines and data verification. Although CoreMark also uses a CRC, it is a different implementation of the algorithm (16-bit vs 32-bit). The other benchmarks in CoreMark, list processing and matrix manipulation, are not representative of IMD workloads.

Overall the targeted workloads are represented by the combination of ImpBench and CoreMark, and the performance of the processors for these benchmarks should provide an indication of the performance in an IMD.

5.3.2 Compiler optimisation

The profiling of the benchmarks in ImpBench reveals that the `tcecc` uses more aggressive optimisation than Chess, even with inlining disabled. `tcecc` is keen to unroll any loop to optimise some instructions away, and this gives the PeLoTTA an edge in some benchmarks. The optimisation is also visible in the IPC of the PeLoTTA, which shows that it maintains a high bus utilisation (except in CSUM and Motion) of 60-80%.

The profiling also reveals the lack of optimisation for the Tvliw, which is the only processor that does not support the LLVM front end. The Tvliw appears to schedule on average 1.3 instructions per cycle, which is low considering the Tvliw can issue up to four instructions per cycle. As the four possible

instructions of the Tvliw consist of two ALUs and two LSUs instructions, the low IPC could be explained by the fact that most benchmarks are focused on many operations per loaded data (except for Motion and CSUM). This reduces the effective issue width to two, which in combination with a lot of data dependencies even further reduces the IPC. It also seems that Chess without LLVM, does not reschedule instructions aggressive enough to optimise for ILP. The Tvliw is also the only processor for which no loops are unrolled.

For the Tdsp, the profiling shows that it is not able to really exploit the hardware loop control, as many benchmarks in ImpBench contain too many instructions in one iteration to see the effect. Although it is able to exploit the use of achieving a high IPC, most of the operations that are executed in parallel consist (partly) of move instructions to move data between its many registers.

The Tzscale is the only processor that does not support some form of ILP, but Chess is able to compile the benchmarks requiring very few execution cycles. This shows the efficiency of the RISC-V instruction set.

5.3.3 Program size

As seen by the profiling analysis in section 5.1.1, `tcecc` is very keen to unroll the loops for the PeLoTTA. ASIP Designer's compiler only does this when used with the LLVM front-end, and unrolls less than `tcecc`. Although the aggressive loop unrolling benefits the execution time of the PeLoTTA, it also increases the program size.

The large code size for the Tvliw is mostly caused due to the large amount of NOPs issued in the instructions, effectively increasing the instruction word size per useful instruction.

The Tdsp and Tzscale achieve the smallest program sizes, which is likely caused by their use of both 16 and 32-bit instructions.

Solutions for reducing the code size of the PeLoTTA could be adding instruction compression, or instructing the compiler to optimise for code size. Although `tcecc` does support features for specifying thresholds for function inlining and loop unrolling, it seems to ignore those options. `tcecc` does have a specific option for reducing instruction memory, `--lowmem-mode-threshold`, but this also had little effect. A compiler flag which instructs the compiler to optimise for size (like GCC's `-Os`), is not available.

5.3.4 Execution cycles

The execution cycle count for the benchmarks in Figure 5.3 mirrors the conclusions from the profiling. The high cycle count for the PeLoTTA in CSUM is explained by the inlined data generation (see Table 5.1). The bad compiler optimisation for the Tvliw are reflected in the amount of cycles for CSUM and MISTY1. Only RC6, where the LLVM front-end was not required to exploit the Tvliw's ILP, the Tvliw is able to execute the benchmark in the fewest amount of cycles. This is supported by the relatively high IPC of the Tvliw in RC6 which can be seen in Figure 5.4.

The Tdsp is slow in the compression (Finnish) and encryption (MISTY1, RC6) benchmarks, and therefore has an overall bad performance in ImpBench. For both the data verification and encryption workloads the PeLoTTA is on average the fastest. The Tzscale the fastest for the compression benchmark, and a close second in encryption. For error detection however, the Tzscale is almost 1.6 times slower than the PeLoTTA.

5.3.5 Power consumption

The PeLoTTA consumes more power than expected, this might be because the IRF is not so effective in an FPGA compared to an ASIC. It also has been found that reducing the program memory by 50% for the PeLoTTA only reduces the power consumption by 10%.

FPGA versus ASIC power consumption

In general, the FPGA has some functions that are implemented on a near-ASIC performance, such as dedicated multipliers, adders, and memory blocks. Other functions that do not have a direct implementation in the FPGA will be compiled from the LUTs and registers in the LEs. These latter functionalities are not only slower, but also consume more power. This discrepancy provides a skewed image when using the FPGA power measurements as reference for ASIC power consumption, and thus the resulting power figures should only be used as indication.

ImpBench

The results of the CSUM and Motion benchmarks will be disregarded, as the CPU time spent in these benchmarks consist largely of generating pseudo-random data.

Looking at Figure 5.6, it is apparent that the Tdsp and Tvliw use on average less power than the PeLoTTA and the Tzscale. This is unexpected when referencing Table 5.12, which shows that the Tdsp and especially the Tvliw use the most resources on the FPGA. This discrepancy indicates that large parts of these processors have low switching activity, or are just better suitable for FPGA implementation than the PeLoTTA and Tzscale. Figure 5.6 also shows that the composition of instructions has a noticeable impact on the power consumption, which implies that the selection of benchmarks is indeed relevant.

For data verification (the CRC32 benchmark), the PeLoTTA requires by far the least energy of all processors. This is mostly due to the loop unrolling and high bus utilisation, which results in a low execution cycle count of the PeLoTTA for this benchmark. The Tzscale scores the worst, using almost 60% more cycles than the PeLoTTA and using twice the energy. The Tvliw, reaching an IPC of only 1.06,

In encryption, the PeLoTTA also consumes the least amount of energy. Even though the Tvliw needs the least amount of cycles for RC6, the PeLoTTA burns just a little less energy than the Tvliw to execute the benchmark. The Tzscale here requires 10-20% extra energy compared to the PeLoTTA for encryption. The Tdsp is surprisingly just as efficient as the PeLoTTA in the MISTY1 benchmark, but can not reach the same efficiency as the PeLoTTA in RC6.

For the compression benchmark (Finnish) the largest cores, Tvliw and Tdsp, score the best. The PeLoTTA scores the worst in this benchmark, requiring 33% more energy than the Tvliw.

CoreMark

The CoreMark scores in Figure 5.8 show that for the traditional control systems workload that CoreMark represents, the Tzscale with the RISC-V RV32I instruction set performs better than the transport-triggered PeLoTTA. The PeLoTTA does compensate a little in power efficiency, where the Tzscale and Tvliw perform almost the same, with the Tzscale reaching 5% more CoreMark executions per Watt.

The Tzscale achieves a similar score as the ARM Cortex-M0+, which scores 2.47 [102].

CNN

Figure 5.3 shows that the Tdsp, Tvliw, and Tzscale perform nearly the same in terms of cycle count. The PeLoTTA completes the benchmark in about 12% fewer cycles, but uses more power than the Tdsp and Tvliw. This results in the Tvliw and Tdsp scoring the best for energy consumption, as seen in Figure 5.9. The low power consumption of the Tvliw might be caused by the low IPC, as it achieves the lowest IPC for CNN compared to the other benchmarks. Figure 5.10 shows that the PeLoTTA does achieve the lowest EDP, with the Tdsp and Tvliw achieving nearly the same EDP. The Tzscale, despite not dissipating the most power, has the worst EDP.

5.3.6 Customisation

Even though all implemented processors are ASIPs, none of the processors are adapted for the target workload. The instruction sets of the processors could have been extended with specific instructions for the benchmarks. Examples for implementing optimised instructions for ImpBench would be:

- The rotation of a 32-bit integer, which is heavily used in RC6
- The entire execution of a single iteration of the checksum algorithm in CSUM, which only consists of a few operations
- A hardware accelerator for the 32-bit CRC, which is included in many modern microcontrollers

Such optimisations would benefit the performance and power consumption of the processors at the cost of area. When designing a platform for IMDs, these optimisations may be implemented to accelerate those workloads which would be typical for IMDs. Such changes would produce different results than presented in this thesis, and may also lead to different conclusions.

Chapter 6

Conclusions and recommendations

This chapter concludes the thesis by providing an overview of the work in this thesis, stating the contributions, reflecting on the problem statement in Chapter 1, and suggesting recommendations.

6.1 Thesis overview

Chapter 1 presents the motivation and context of this thesis, and introduces the problem statement and its sub-questions. These sub-questions are answered in Chapter 2 as follows:

Chapter 3 reviews alternatives to ASIPs and selects implementations of the desired architectures. To represent the current state of processors in IMDs the Tzscale with its RISC-V RV32I instruction set and the Tdsp with its DSP ISA are selected. As a viable alternative the PeLoTTA is selected, which is a low-power TTA core. The Tvliw is added to the comparison to see the performance of another VLIW processor and to bridge the architectural gap in the comparison. The chapter concludes with an overview of benchmark suites and the selection thereof.

In Chapter 4 the selected processor implementations are implemented to run on an FPGA. The benchmarks in ImpBench are ported to be able to run without OS on a processor with a very small memory footprint. A benchmark for neural networks is also implemented, using a CNN on the MNIST dataset. Chapter 5 presents the experimental setup and the simulation and power measurement results, and are discussed at the end of the chapter.

6.2 Contributions

In this thesis a novel collection of benchmarks is used to assess implementations of a set of different ASIP architectures. This benchmark collection consists of the benchmarks from ImpBench, CoreMark, and a custom benchmark for CNNs. Benchmarks from ImpBench were ported to be completely architecture-agnostic. The new CNN benchmark represents the well-known workload of using a CNN for pattern recognition. The benchmark uses fixed-point arithmetic and is also developed to be architecture-agnostic and have a small memory footprint.

This thesis also presents the first use of this collection of benchmarks on implementations of a set of different ASIPs architectures. Next to simulations, power measurements are also performed by running the benchmarks on the ASIP implementations on FPGAs This provides insight on the performance of ASIP architectures for IMDs, which is the goal of this thesis.

6.3 Conclusions

To answer the research question stated in the problem statement, the sub-questions will first be answered:

Which benchmarks need to be run to represent the workloads of IMDs?

Workloads of IMDs are categorised in six categories; control, encryption, data compression, error detection/correction, and ANNs. The benchmark suites selected that represent these workloads are ImpBench and CoreMark. A design and implementation of an additional benchmark for ANNs is also presented in this thesis.

Which processor architectures are currently prevalent in IMDs?
Most processor-based IMDs use microcontrollers containing processors with RISC architectures. Besides RISC, there are also few occurrences of CISC and DSP architectures.

Which low-power implementations of computer architectures are available for ASIPs?
ASIP Designer provides many standard processor models for ASIPs based on architectures like VLIW, RISC, DSP, and has the option to optimise the processor implementation for low-power. OpenASIP provides TTA ASIP implementations on top of these architectures.

Finally, the core question of the thesis is: *How do different ASIP computer architectures compare for medical implantable devices?* For systems running simple control applications or compression workloads, the RISC-V ISA of the Tzscale processor performs well. It combines low area requirements and good performance, and delivers this on a low energy budget. The Tzscale also achieves the highest CoreMark score, with equivalent performance to the ARM Cortex-M0+. The PeLoTTA surpasses the Tzscale in terms of area numbers with roughly two-thirds of the LEs used, and in terms of power efficiency in the data validation and encryption benchmarks. For CoreMark, the PeLoTTA achieves comparable power efficiency compared to the Tzscale. The PeLoTTA does have some downsides, as it is does not support interrupts and requires more program memory.

The Tvliw and Tdsp would not be recommended for any implementations in IMDs, as they use more than twice the LEs than the PeLoTTA, but deliver only mediocre performance for the extra resources in most benchmarks. It must be noted however, that the Tvliw lacks a good compiler front-end, and is not fully able to exploit its ILP because of this.

This conclusion is largely based on measurements of the implementations on an FPGA. This approach does have its shortcomings in that it may not provide an accurate representation for implementations on ASICs, but does provide an indication of the relative area and performance of the implementations.

6.4 Recommendations

The power measurements of the benchmarks in ImpBench running on the implementations on the FPGA also measure the clock cycles dedicated to data generation. A better approach would be to generate the data completely and let the benchmark iterate several times over the data, to lower the overhead of data generation.

During the porting of ImpBench, two benchmarks were left unimplemented: MLZO, which claimed too much memory, and DMU, which contained too many floating-point operations. Although other benchmarks are included which cover their targeted workload, it would be good to have multiple benchmarks cover the same workload. For the same reason, it would be good to run a second ANN benchmark with a different architecture than the CNN benchmark. The CNN benchmark could also be improved to better match the true workload of CNNs in IMDs by replicating real applications and using real samples.

For the architectural decision for ASIPs for IMDs, it is recommended to extend the PeLoTTA and Tzscale with application-specific instructions. The performance of the processors can then be re-evaluated to see the effect of the extended instruction set and the relative performance gain of the processors. After this stage, implementation and simulations of ASICs would provide more accurate power and area numbers, and play a decisive role in the choice of ASIP architecture for IMDs.

Chapter 7

Appendix

7.1 Profiling

This section contains automatically parsed profiling information from profiling data generated by `ttasim` or ASIP Designer. The profiling information is categorised by benchmark, and listed in the order of PeLoTTA, Tdsp, Tvliw, Tzscale for each benchmark.

7.1.1 CRC32

```
 PC                                                                    Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
 35  ALU.out1 -> RF.5, ..., ALU.out1 -> LSU.in1t.ldw ;                              1456     1
 36  255 -> ALU.in1t.and, 0 -> LSU.in2, LSU.out1 -> ALU.in2 ;                       1456     1
 37  4 -> ALU.in1t.add, RF.5 -> ALU.in2, ALU.out1 -> RF.7 ;                         1456     0
 38  24 -> ALU.in1t.shru, RF.2 -> ALU.in2, ALU.out1 -> RF.12 ;                      1456     0
 39  RF.7 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, LSU.out1 -> RF.6 ;                  1456     0
 40  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                  1456     0
 41  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 42  8 -> ALU.in1t.shl, RF.2 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 43  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                           1456     0
 44  8 -> ALU.in1t.shru, RF.6 -> ALU.in2, ALU.out1 -> RF.7 ;                        1456     1
 45  255 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1456     0
 46  24 -> ALU.in1t.shru, RF.7 -> ALU.in2, ALU.out1 -> RF.8 ;                       1456     1
 47  RF.8 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                               1456     0
 48  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                  1456     1
 49  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 50  8 -> ALU.in1t.shl, RF.7 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 51  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, RF.5 -> LSU.in1t.stw ;          1456     0
 52  16 -> ALU.in1t.shru, RF.6 -> ALU.in2, ALU.out1 -> RF.5 ;                       1456     0
 53  255 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1456     0
 54  24 -> ALU.in1t.shru, RF.5 -> ALU.in2, ALU.out1 -> RF.7 ;                       1456     1
 55  RF.7 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                               1456     0
 56  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                  1456     1
 57  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 58  8 -> ALU.in1t.shl, RF.5 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 59  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                           1456     0
 60  RF.6 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ALU.out1 -> RF.5 ;                  1456     1
 61  22 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ... [IU_1x32.0=1020] ;               1456     0
 62  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                          1456     1
 63  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 64  8 -> ALU.in1t.shl, RF.5 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 65  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, RF.12 -> LSU.in1t.ldw ;         1456     0
 66  255 -> ALU.in1t.and, LSU.out1 -> ALU.in2, ALU.out1 -> RF.5 ;                   1456     0
 67  24 -> ALU.in1t.shru, RF.5 -> ALU.in2, ALU.out1 -> RF.6 ;                       1456     0
 68  RF.6 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, LSU.out1 -> RF.9 ;                  1456     0
 69  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                  1456     0
 70  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 71  8 -> ALU.in1t.shl, RF.5 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 72  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                           1456     0
 73  8 -> ALU.in1t.shru, RF.9 -> ALU.in2, ALU.out1 -> RF.5 ;                        1456     1
 74  255 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1456     0
 75  24 -> ALU.in1t.shru, RF.5 -> ALU.in2, ALU.out1 -> RF.6 ;                       1456     1
 76  RF.6 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                               1456     0
 77  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                  1456     1
 78  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                              1456     1
 79  8 -> ALU.in1t.shl, RF.5 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 1456     1
 80  LSU.out1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                           1456     0
 81  16 -> ALU.in1t.shru, RF.9 -> ALU.in2, ALU.out1 -> RF.5 ;                       1456     1
 82  255 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1456     0
 83  24 -> ALU.in1t.shru, RF.5 -> ALU.in2, ALU.out1 -> RF.6 ;                       1456     1
 84  RF.6 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                               1456     0
...
```

Listing 7.1: A modified excerpt from the profiling information generated by `tcecc` for the CRC32 benchmark running on the PeLoTTA. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                    Assembly  Exe-count  Cycles
---  --------------------------  ---------  ------
 82  lsl r6,r6, 1 . mv a2 ,a0        11557   11557
 84  add r0,r0,r2 . mv m1 ,r6         11557   11557
 86  lsl r6,r0, -1 . add a3,m1        11557   11557
 88  and r6,r4,r6 . ll l3, (a3)       11557   11557
 90  xor l3,l0,l3 . mv m0 ,r6         11557   11557
 92  lsl l0,l3,r1 . add a2,m0         11557   11557
 94  lsl l1,l3,r3 . ll l2, (a2)       11557   11557
 96  li l3,255                        11557   11557
 98  and l3,l2,l3                     11557   11557
 99  lsl l2,l2,r5                     11557   11557
100  xor l1,l1,l3 . sl l2, (a2)       11557   11557
102  lo r6,l1 . mv a3 ,a1             11557   11557
```

Listing 7.2: A modified excerpt from the profiling information generated by ASIP Designer for the CRC32 benchmark running on the Tdsp.

```
 PC                                        Assembly  Exe-count  Cycles
---  ---------------------------------------------  ---------  ------
 31  lsr r9, r0, r8 : lsr r10, r3, r5 : nop : nop       11736   11736
 32  lsl r0, r0, r7 : nop : nop : mv m0, r10            11736   11736
 33  nop : add r3, r3, r6 : nop : nop                   11736   11736
 34  nop : nop : nop : p1=p0+m0                         11736   11736
 35  nop : nop : nop : ld r10, [p1]                     11736   11736
 36  and r10, r4, r10 : lsr r11, r10, r7: nop : nop     11736   11736
 37  xor r9, r9, r10 : nop : nop : st r11, [p1]         11736   11736
 38  nop : add r9, r1, r9 : nop : nop                   11736   11736
 39  nop : nop : nop : mv p1, r9                        11736   11736
 40  nop : nop : nop : nop                              11736   11736
 41  nop : nop : nop : ld r9, [p1]                      11736   11736
 42  nop : xor r0, r0, r9 : nop : nop                   11736   11736
 43  nop : ult r3, r2 : nop : nop                       11828   11828
 44  nop : cjmp 31                                      11828   35392
```

Listing 7.3: A modified excerpt from the profiling information generated by ASIP Designer for the CRC32 benchmark running on the Tvliw.

```
 PC          Assembly  Exe-count  Cycles
---  ----------------  ---------  ------
124  andi x9,x10,-4        11648   11648
128  c.add x9, x6          11648   11648
130  c.lw x13, 0(x9)       11648   11648
132  srli x14,x12,24       11648   11648
136  and x15,x8,x13        11648   11648
140  c.xor x14, x15        11648   11648
142  slli x7,x14,2         11648   11648
146  add x14,x3,x7         11648   11648
150  c.lw x14, 0(x14)      11648   11648
152  c.slli x12, 8         11648   11648
154  c.srli x13, 8         11648   11648
156  c.addi x10, 1         11648   11648
158  c.sw x13, 0(x9)       11648   11648
160  c.xor x12, x14        11648   11648
162  bne x5,x10,-38        11648   23205
```

Listing 7.4: A modified excerpt from the profiling information generated by ASIP Designer for the CRC32 benchmark running on the Tzscale.

7.1.2 CSUM

```
 PC                                                                    Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
 43  ..., RF.6 -> LSU.in2, ALU.out1 -> LSU.in1t.stw ;                                183     1
 44  76 -> ALU.in1t.add, ..., ... ;                                                  183     1
 45  ALU.out1 -> RF.2, ..., ... ;                                                    183     2
 46  ALU.out1 -> RF.1, ..., ... [IU_1x32.0=326] ;                                    183     2
 47  4 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ... [IU_1x32.0=32] ;               183     2
 48  ..., IU_1x32.0 -> LSU.in2, ALU.out1 -> LSU.in1t.stw ;                           183     1
 49  72 -> ALU.in1t.add, RF.0 -> ALU.in2, ... ;                                      183     1
 50  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     1
 51  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                            183     2
 52  40 -> ALU.in1t.add, LSU.out1 -> LSU.in2, RF.0 -> ALU.in2 ;                      183     2
 53  60 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stw ;                             183     0
 54  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     1
 55  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                            183     2
 56  36 -> ALU.in1t.add, LSU.out1 -> LSU.in2, RF.0 -> ALU.in2 ;                      183     2
 57  68 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stw ;                             183     0
 58  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     1
 59  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                            183     2
 60  32 -> ALU.in1t.add, LSU.out1 -> LSU.in2, RF.0 -> ALU.in2 ;                      183     2
 61  64 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stw ;                             183     0
 62  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     1
 63  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                            183     2
 64  28 -> ALU.in1t.add, LSU.out1 -> LSU.in2, RF.0 -> ALU.in2 ;                      183     2
 65  48 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stw ;                             183     0
 66  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     1
 67  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                            183     2
 68  24 -> ALU.in1t.add, LSU.out1 -> LSU.in2, RF.0 -> ALU.in2 ;                      183     2
 69  56 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stw ;                             183     0
 70  12 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.ldw ;                             183     1
 71  ALU.out1 -> RF.2, ..., LSU.out1 -> LSU.in1t.ldw ;                               183     1
 72  52 -> ALU.in1t.add, ..., ... ;                                                  183     1
 73  LSU.out1 -> RF.4, ..., ALU.out1 -> LSU.in1t.ldw ;                               183     2
 74  76 -> ALU.in1t.add, ..., LSU.out1 -> LSU.in1t.ldw ;                             183     1
 75  ALU.out1 -> RF.2, LSU.out1 -> LSU.in2, RF.2 -> LSU.in1t.stw ;                   183     1
 76  4 -> ALU.in1t.add, ..., ... [IU_1x32.0=32] ;                                    183     0
 77  76 -> ALU.in1t.add, IU_1x32.0 -> LSU.in2, ALU.out1 -> LSU.in1t.stw ;            183     2
 78  ALU.out1 -> RF.1, RF.0 -> ALU.in2, ... [IU_1x32.0=326] ;                        183     0
 79  20 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, RF.2 -> LSU.in1t.ldw ;            183     1
 80  LSU.out1 -> RF.2, RF.4 -> LSU.in2, ALU.out1 -> LSU.in1t.stw ;                   183     0
 81  16 -> ALU.in1t.add, RF.0 -> ALU.in2, ... ;                                      183     0
 82  8 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.ldw ;                              183     1
 83  ..., ..., LSU.out1 -> RF.5 ;                                                    183     1
 84  -1 -> ALU.in1t.xor, RF.2 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 183     2
 85  ALU.out1 -> ALU.in1t.ior, LSU.out1 -> ALU.in2, LSU.out1 -> RF.26 ;              183     0
 86  RF.5 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                                183     0
 87  16 -> ALU.in1t.shru, RF.2 -> ALU.in2, ALU.out1 -> RF.4 ;                        183     1
 88  -1 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                                  183     0
 89  RF.4 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                                183     1
 90  12 -> ALU.in1t.add, RF.0 -> ALU.in2, ALU.out1 -> RF.4 ;                         183     1
 91  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                            183     0
 92  -1 -> ALU.in1t.xor, LSU.out1 -> ALU.in2, LSU.out1 -> RF.6 ;                     183     2
...
```

Listing 7.5: A modified excerpt from the profiling information generated by `tcecc` for the CSUM benchmark running on the PeLoTTA. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                        Assembly  Exe-count  Cycles
---  ------------------------------  ---------  ------
 44  cl 238                                366     366
 46  li r0,32                              366     366
 48  ll l1, (sp-6)                         366     366
 49  li l0,32                              366     366
 51  add l0,l0,l1 . ll a0, (sp-8)          366     366
 53  li l1,11708                           366     366
 55  sltu l0,l1 . sl l0, (sp-6)            366     366
 57  li m0,2                               366     366
 58  ll l1, (a0+m0)                        366     366
 59  li l0,-1                              366     366
 60  xor l2,l0,l1 . ll l3, (sp-4)          366     366
 62  ll l0, (2)                            366     366
 64  or l2,l0,l2 . li r0,-16               366     366
 66  add l3,l2,l3 . li l2,-1               366     366
 68  lsl l1,l1,r0                          366     366
 69  xor l1,l1,l2                          366     366
 70  add l3,l1,l3 . ll l1, (a0+m0)         366     366
 72  xor l2,l1,l2                          366     366
 73  or l2,l0,l2                           366     366
 74  add l3,l2,l3 . li l2,-1               366     366
 76  lsl l1,l1,r0                          366     366
 77  xor l1,l1,l2                          366     366
 78  add l3,l1,l3 . ll l1, (a0+m0)         366     366
 80  xor l2,l1,l2                          366     366
 81  or l2,l0,l2                           366     366
 82  add l3,l2,l3 . li l2,-1               366     366
 84  lsl l1,l1,r0                          366     366
 85  xor l1,l1,l2                          366     366
 86  add l3,l1,l3 . ll l1, (a0+m0)         366     366
 88  xor l2,l1,l2                          366     366
 89  or l2,l0,l2                           366     366
 90  add l3,l2,l3 . li l2,-1               366     366
 92  lsl l1,l1,r0                          366     366
 93  xor l1,l1,l2                          366     366
 94  add l3,l1,l3 . ll l1, (a0+m0)         366     366
 96  xor l2,l1,l2                          366     366
 97  or l2,l0,l2                           366     366
 98  add l3,l2,l3 . li l2,-1               366     366
100  lsl l1,l1,r0                          366     366
101  xor l1,l1,l2                          366     366
102  add l3,l1,l3 . ll l1, (a0+m0)         366     366
104  xor l2,l1,l2                          366     366
105  or l2,l0,l2                           366     366
106  add l2,l2,l3 . li l3,-1               366     366
108  lsl l1,l1,r0                          366     366
109  xor l1,l1,l3                          366     366
110  add l1,l1,l2 . ll l2, (a0+m0)         366     366
112  xor l2,l2,l3 . sl l2, (sp-4)          366     366
114  or l2,l0,l2                           366     366
115  li m0,14                              366     366
...
```

Listing 7.6: A modified excerpt from the profiling information generated by ASIP Designer for the CSUM benchmark running on the Tdsp. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                                          Assembly  Exe-count  Cycles
---  ---------------------------------------------  ---------  ------
 32  nop : add r9, r3, r9 : nop : ld r10, [p0+=m0]        2934    2934
 33  and r10, r1, r10 : lsr r11, r10, r7: nop : nop       2934    2934
 34  xor r10, r5, r11 : xor r12, r5, r10: nop : nop       2934    2934
 35  nop : add r0, r0, r12 : nop : nop                    2934    2934
 36  nop : add r0, r0, r10 : nop : nop                    2934    2934
 37  nop : slt r9, r8 : nop : nop                         3301    3301
 38  nop : cjmp 32                                        3301    9536
```

Listing 7.7: A modified excerpt from the profiling information generated by ASIP Designer for the CSUM benchmark running on the Tvliw.

```
 PC              Assembly  Exe-count  Cycles
---  --------------------  ---------  ------
 80  c.swsp x4, 12(x2)           366     366
 82  c.addi4spn x10,x2,16        366     366
 84  c.jal 384                   366     732
 86  c.lwsp x8, 16(x2)           366     366
 88  c.li x11, -1                366     366
 90  xor x12,x8,x11              366     366
 94  c.lui x10, -16              366     366
 96  c.lwsp x1, 4(x2)            366     366
 98  c.or x12, x10               366     366
100  c.lwsp x9, 20(x2)           366     366
102  c.add x1, x12               366     366
104  c.srli x8, 16               366     366
106  xor x13,x8,x11              366     366
110  xor x12,x9,x11              366     366
114  c.add x1, x13               366     366
116  c.or x12, x10               366     366
118  c.lwsp x8, 24(x2)           366     366
120  c.add x1, x12               366     366
122  c.srli x9, 16               366     366
124  xor x13,x9,x11              366     366
128  xor x12,x8,x11              366     366
132  c.add x1, x13               366     366
134  c.or x12, x10               366     366
136  c.lwsp x9, 28(x2)           366     366
138  c.add x1, x12               366     366
140  c.srli x8, 16               366     366
142  xor x13,x8,x11              366     366
146  xor x12,x9,x11              366     366
150  c.add x1, x13               366     366
152  c.lwsp x8, 32(x2)           366     366
154  c.or x12, x10               366     366
156  c.srli x9, 16               366     366
158  c.add x1, x12               366     366
160  xor x13,x9,x11              366     366
164  c.add x1, x13               366     366
166  xor x12,x8,x11              366     366
170  srli x13,x8,16              366     366
174  c.or x12, x10               366     366
176  c.lwsp x9, 36(x2)           366     366
178  c.add x1, x12               366     366
180  c.xor x13, x11              366     366
182  xor x8,x9,x11               366     366
186  c.add x1, x13               366     366
188  or x13,x8,x10               366     366
192  c.lwsp x12, 40(x2)          366     366
194  srli x8,x9,16               366     366
198  c.add x1, x13               366     366
200  xor x9,x12,x11              366     366
204  c.lwsp x13, 44(x2)          366     366
206  c.xor x8, x11               366     366
...
```

Listing 7.8: A modified excerpt from the profiling information generated by ASIP Designer for the CSUM benchmark running on the Tzscale. Only a part of the compiled loop is shown, as the loop is partly unrolled.

7.1.3 MISTY1

```
 PC                                                                          Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------------  ---------  ----
 50  ALU.out1 -> RF.1, ..., ... [IU_1x32.0=632] ;                                         1025     0
 51  48 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, RF.0 -> ALU.in2 ;                      1025     2
 52  ..., RF.7 -> LSU.in2, ALU.out1 -> LSU.in1t.stw ;                                     1025     0
 53  36 -> ALU.in1t.add, RF.0 -> ALU.in2, ... ;                                           1025     1
 54  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                                 1025     1
 55  ..., ..., LSU.out1 -> LSU.in1t.ldw ;                                                 1025     2
 56  ..., ..., ... [IU_1x32.0=65535] ;                                                    1025     2
 57  IU_1x32.0 -> ALU.in1t.and, LSU.out1 -> ALU.in2, LSU.out1 -> RF.4 ;                   1025     3
 58  ALU.out1 -> RF.5, ..., ... ;                                                         1025     0
 59  16 -> ALU.in1t.shru, RF.4 -> ALU.in2, ... [IU_1x32.0=687] ;                          1025     2
 60  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ALU.out1 -> RF.4 ;                   1025     1
 61  ALU.out1 -> ALU.in1t.xor, RF.5 -> ALU.in2, ... [IU_1x32.0=25185] ;                   1025     0
 62  IU_1x32.0 -> ALU.in1t.ior, ALU.out1 -> ALU.in2, ALU.out1 -> RF.5 ;                   1025     1
 63  56 -> ALU.in1t.add, RF.0 -> ALU.in2, ALU.out1 -> RF.6 ;                              1025     0
 64  RF.6 -> ALU.in1t.xor, RF.4 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                    1025     0
 65  ALU.out1 -> RF.4, ..., ... [IU_1x32.0=65535] ;                                       1025     0
 66  IU_1x32.0 -> ALU.in1t.and, LSU.out1 -> ALU.in2, LSU.out1 -> RF.6 ;                   1025     2
 67  16 -> ALU.in1t.shl, RF.4 -> ALU.in2, ALU.out1 -> RF.7 ;                              1025     0
 68  ALU.out1 -> ALU.in1t.ior, RF.5 -> ALU.in2, ... ;                                     1025     0
 69  ALU.out1 -> RF.5, ..., ... ;                                                         1025     1
 70  16 -> ALU.in1t.shru, RF.6 -> ALU.in2, ... [IU_1x32.0=13106] ;                        1025     2
 71  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ALU.out1 -> RF.6 ;                   1025     1
 72  ALU.out1 -> ALU.in1t.xor, RF.7 -> ALU.in2, ... [IU_1x32.0=56447] ;                   1025     0
 73  IU_1x32.0 -> ALU.in1t.ior, ALU.out1 -> ALU.in2, ALU.out1 -> RF.7 ;                   1025     1
 74  ALU.out1 -> ALU.in1t.xor, RF.6 -> ALU.in2, ... [IU_1x32.0=14134] ;                   1025     0
 75  IU_1x32.0 -> ALU.in1t.xor, RF.7 -> ALU.in2, ALU.out1 -> RF.6 ;                       1025     1
 76  127 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ALU.out1 -> RF.4 ;                         1025     0
 77  ALU.out1 -> RF.8, ..., ... ;                                                         1025     0
 78  5 -> ALU.in1t.shru, RF.4 -> ALU.in2, ... [IU_1x32.0=2044] ;                          1025     2
 79  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1025     1
 80  RF.2 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                                     1025     1
 81  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                                 1025     1
 82  LSU.out1 -> ALU.in1t.xor, RF.8 -> ALU.in2, ... [IU_1x32.0=3088] ;                    1025     2
 83  IU_1x32.0 -> ALU.in1t.add, ..., ALU.out1 -> RF.4 ;                                   1025     1
 84  ..., ..., ALU.out1 -> LSU.in1t.ldq ;                                                 1025     1
 85  RF.4 -> ALU.in1t.xor, LSU.out1 -> ALU.in2, ... [IU_1x32.0=287] ;                      1025     2
 86  IU_1x32.0 -> ALU.in1t.xor, RF.4 -> ALU.in2, ALU.out1 -> RF.8 ;                       1025     1
 87  2 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ... ;                                        1025     0
 88  RF.2 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... ;                                     1025     1
 89  90 -> ALU.in1t.xor, RF.8 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                      1025     1
 90  127 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                      1025     0
 91  ALU.out1 -> ALU.in1t.xor, LSU.out1 -> ALU.in2, ALU.out1 -> RF.8 ;                    1025     1
 92  9 -> ALU.in1t.shl, RF.8 -> ALU.in2, ALU.out1 -> RF.4 ;                               1025     0
 93  ALU.out1 -> ALU.in1t.ior, RF.4 -> ALU.in2, ... [IU_1x32.0=13106] ;                   1025     0
 94  IU_1x32.0 -> ALU.in1t.xor, RF.6 -> ALU.in2, ALU.out1 -> RF.8 ;                       1025     1
 95  ALU.out1 -> RF.4, ..., ... ;                                                         1025     0
 96  5 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ... [IU_1x32.0=2044] ;                       1025     2
 97  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                1025     1
 98  RF.2 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... [IU_1x32.0=3088] ;                     1025     1
 99  127 -> ALU.in1t.and, RF.4 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                     1025     1
...
```

Listing 7.9: A modified excerpt from the profiling information generated by `tcecc` for the MISTY1 benchmark running on the PeLoTTA. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                        Assembly  Exe-count  Cycles
---  ------------------------------  ---------  ------
 48  sub l3,l0,l1 . sl l2, (sp-10)       1025    1025
 50  sle l3,-1                           1025    1025
 51  jrcn 9                              1025    1025
 52  sl a0, (sp-6)                       1025    1025
 53  sl l3, (sp-8)                       1025    1025
...
 62  cl 1102                             1025    1025
 64  lo r0,l1                            1025    1025
 65  ll a0, (sp-6)                       1025    1025
 66  li m0,2                             1025    1025
 67  li a1,65535                         1025    1025
 69  ll l1, (a0+m0)                      1025    1025
 70  li r1,-16                           1025    1025
 71  lsl l0,l1,r1 . xz l2 ,a1            1025    1025
 73  and l2,l1,l2 . li r2,-6             1025    1025
 75  li l1,13106                         1025    1025
 77  and l3,l0,l1 . li r3,-7             1025    1025
 79  li a2,56447                         1025    1025
 81  xor l2,l2,l3 . xz l3 ,a2            1025    1025
 83  or l3,l2,l3 . sl l2, (sp-6)         1025    1025
 85  xor l0,l0,l3 . li r6,9              1025    1025
 87  xor l3,l0,l1 . sl l0, (sp-12)       1025    1025
 89  lo r0,l3                            1025    1025
 90  lsl r4,r0,r2                        1025    1025
 91  li r0,1022                          1025    1025
 93  li l1,14134                         1025    1025
 95  li a3,9216                          1025    1025
 97  and r4,r0,r4 . mv a6 ,a3            1025    1025
 99  xor l1,l1,l2 . mv m1 ,r4            1025    1025
101  lsl l2,l1,r3 . add a3,m1            1025    1025
103  li l0,127                           1025    1025
105  lo r3,l2 . ll l2, (a3)              1025    1025
107  and l3,l0,l3 . mv a7 ,a6            1025    1025
109  lo r4,l3                            1025    1025
110  li a3,5                             1025    1025
112  xor l2,l2,l3 . mv m1 ,r4            1025    1025
114  lsl r5,r3, 1 . mv a4 ,a3            1025    1025
116  lo r4,l2 . add a3,m1                1025    1025
118  li r7,311                           1025    1025
120  xor r5,r4,r7 . mv m1 ,r5            1025    1025
122  lsl r5,r5, 1 . lw r4, (a3)          1025    1025
124  li r3,70                            1025    1025
126  xor r3,r3,r4 . add a6,m1            1025    1025
128  xz l3,r3 . mv m1 ,r5                1025    1025
130  xor l2,l2,l3 . mv a3 ,a7            1025    1025
132  and l2,l0,l2 . add a7,m1            1025    1025
134  and l1,l0,l1 . ll l3, (a7)          1025    1025
136  xor l3,l2,l3 . mv a7 ,a4            1025    1025
138  lsl l2,l2,r6                        1025    1025
139  or l2,l2,l3 . ll l3, (sp-6)         1025    1025
141  xor l2,l2,l3                        1025    1025
...
```

Listing 7.10: A modified excerpt from the profiling information generated by ASIP Designer for the MISTY1 benchmark running on the Tdsp. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                                              Assembly  Exe-count  Cycles
---  ------------------------------------------------------  ---------  ------
 22  nop : mv r0, 7                                             24608   24608
 23  lsr r0, r1, r0 : mv r3, 511                                24608   24608
 24  and r0, r0, r3 : mv p0, 2437                               24608   24608
 25  mv m0, r0 : mv r0, 127                                     24608   24608
 26  and r4, r0, r1 : mv p1, 2309                               24608   24608
 27  and r1, r2, r3 : nop : mv m0, r4 : p2=p0+m0                24608   24608
 28  nop : nop : nop : ld r3, [p2]                              24608   24608
 29  nop : xor r3, r3, r4 : nop : p1=p1+m0                      24608   24608
 30  nop : xor r1, r1, r3 : nop : ld r4, [p1]                   24608   24608
 31  mv m0, r1 : mv r1, 9                                       24608   24608
 32  xor r2, r3, r4 : lsr r5, r2, r1 : nop : nop                24608   24608
 33  nop : xor r2, r2, r5 : nop : p0=p0+m0                      24608   24608
 34  nop : and r2, r0, r2 : nop : ld r0, [p0]                   24608   24608
 35  xor r2, r0, r2 : lsl r1, r2, r1 : nop : nop                24608   24608
 36  or r0, r1, r2 : ijmp lr                                    24608   49216
...
 86  nop : mv r0, 16                                            10250   10250
 87  lsr r3, r1, r0 : mv r6, 65535                              10250   10250
 88  and r1, r1, r6 : mv r4, 1                                  10250   10250
 89  and r5, r2, r4 : mv r6, 0                                  10250   10250
 90  nop : neq r5, r6 : nop : nop                               10250   10250
 91  nop : cjmp 103                                             10250   25625
...
117  nop : lsl r1, r1, r0 : nop : nop                          10250   10250
118  or r0, r1, r2 : ijmp lr                                   10250   20500
```

Listing 7.11: A modified excerpt from the profiling information generated by ASIP Designer for the MISTY1 benchmark running on the Tvliw.

```
 PC              Assembly  Exe-count  Cycles
---  ---------------------  ---------  ------
256  sub x9,x8,x11               1025    1025
260  c.addi4spn x10,x2,96        1025    1025
262  blt x3,x9,16                1025    2049
...
278  c.swsp x9, 84(x2)           1025    1025
280  c.jal 1744                  1025    2050
282  c.lwsp x10, 96(x2)          1025    1025
284  c.lwsp x5, 72(x2)           1025    1025
286  c.lwsp x22, 76(x2)          1025    1025
288  srli x11,x10,16             1025    1025
292  and x10,x10,x22             1025    1025
296  and x8,x5,x11               1025    1025
300  c.lwsp x9, 68(x2)           1025    1025
302  c.xor x10, x8               1025    1025
304  or x8,x10,x9                1025    1025
308  c.xor x8, x11               1025    1025
310  xor x9,x8,x5                1025    1025
314  srli x11,x9,5               1025    1025
318  addi x16,x0,2044            1025    1025
322  c.lwsp x1, 12(x2)           1025    1025
324  and x3,x16,x11              1025    1025
328  add x12,x1,x3               1025    1025
332  addi x11,x0,127             1025    1025
336  c.lw x12, 0(x12)            1025    1025
338  c.lwsp x3, 8(x2)            1025    1025
340  c.and x9, x11               1025    1025
342  c.lwsp x30, 60(x2)          1025    1025
344  c.xor x12, x9               1025    1025
346  add x6,x9,x3                1025    1025
350  lbu x13,0(x6)               1025    1025
354  xor x9,x10,x30              1025    1025
358  addi x29,x0,311             1025    1025
362  srli x14,x9,5               1025    1025
366  c.xor x13, x12              1025    1025
368  xor x4,x12,x29              1025    1025
372  and x6,x16,x14              1025    1025
376  add x12,x6,x1               1025    1025
380  c.lw x14, 0(x12)            1025    1025
382  and x12,x9,x11              1025    1025
386  xor x9,x12,x14              1025    1025
390  c.slli x4, 2                1025    1025
392  add x14,x4,x1               1025    1025
396  add x4,x12,x3               1025    1025
400  lbu x12,0(x4)               1025    1025
404  addi x7,x0,287              1025    1025
408  addi x25,x0,70              1025    1025
412  xor x6,x9,x7                1025    1025
416  xor x13,x25,x13             1025    1025
420  c.lw x14, 0(x14)            1025    1025
422  slli x17,x6,2               1025    1025
426  c.xor x9, x12               1025    1025
...
```

Listing 7.12: A modified excerpt from the profiling information generated by ASIP Designer for the MISTY1 benchmark running on the Tzscale. Only a part of the compiled loop is shown, as the loop is partly unrolled.

7.1.4 Motion

```
 PC                                                                    Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
 25  ..., IU_1x32.0 -> gcu.pc.call, ... ;                                          2048     2
 26  ..., ..., RF.0 -> LSU.in1t.stw ;                                              2048     2
 27  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 28  ALU.out1 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, RF.2 -> ALU.in2 ;         2048     1
 29  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 30  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 31  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 32  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 33  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 34  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 35  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 36  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 37  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 38  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 39  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 40  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 41  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 42  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 43  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 44  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 45  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 46  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 47  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 48  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 49  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 50  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 51  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 52  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 53  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 54  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 55  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 56  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 57  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 58  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 59  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 60  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 61  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 62  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 63  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 64  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 65  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 66  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 67  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 68  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 69  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 70  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 71  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
 72  31 -> ALU.in1t.shru, RF.1 -> ALU.in2, ... [IU_1x32.0=186] ;                   2048     2
 73  RF.2 -> ALU.in1t.add, IU_1x32.0 -> gcu.pc.call, ALU.out1 -> ALU.in2 ;         2048     1
 74  ALU.out1 -> RF.2, ..., ... ;                                                  2048     0
...
```

Listing 7.13: A modified excerpt from the profiling information generated by `tcecc` for the Motion benchmark running on the PeLoTTA. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                       Assembly  Exe-count  Cycles
---  ----------------------------  ---------  ------
 78  cl 36                              4096    4096
 80  nop                                4096    4096
 81  li r0,-31                          4096    4096
 82  lsl l0,l0,r0                       4096    4096
 83  sl l0, (sp-8)                      4096    4096
 84  cl 36                              4096    4096
 86  nop                                4096    4096
 87  li r0,-31                          4096    4096
 88  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
 90  add l0,l0,l1                       4096    4096
 91  sl l0, (sp-8)                      4096    4096
 92  cl 36                              4096    4096
 94  nop                                4096    4096
 95  li r0,-31                          4096    4096
 96  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
 98  add l0,l0,l1                       4096    4096
 99  sl l0, (sp-8)                      4096    4096
100  cl 36                              4096    4096
102  nop                                4096    4096
103  li r0,-31                          4096    4096
104  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
106  add l0,l0,l1                       4096    4096
107  sl l0, (sp-8)                      4096    4096
108  cl 36                              4096    4096
110  nop                                4096    4096
111  li r0,-31                          4096    4096
112  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
114  add l0,l0,l1                       4096    4096
115  sl l0, (sp-8)                      4096    4096
116  cl 36                              4096    4096
118  nop                                4096    4096
119  li r0,-31                          4096    4096
120  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
122  add l0,l0,l1                       4096    4096
123  sl l0, (sp-8)                      4096    4096
124  cl 36                              4096    4096
126  nop                                4096    4096
127  li r0,-31                          4096    4096
128  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
130  add l0,l0,l1                       4096    4096
131  sl l0, (sp-8)                      4096    4096
132  cl 36                              4096    4096
134  nop                                4096    4096
135  li r0,-31                          4096    4096
136  lsl l1,l0,r0 . ll l0, (sp-8)       4096    4096
138  add l0,l0,l1                       4096    4096
139  sl l0, (sp-8)                      4096    4096
140  cl 36                              4096    4096
142  nop                                4096    4096
143  li r0,-31                          4096    4096
...
```

Listing 7.14: A modified excerpt from the profiling information generated by ASIP Designer for the Motion benchmark running on the Tdsp. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                            Assembly  Exe-count  Cycles
---  --------------------------------  ---------  ------
 51  nop : bsr 24                           81920  163840
 52  nop : mv r1, -2147483648               81920   81920
 53  ult r0, r1 : ld r0, [sp-4]             81920   81920
 54  nop : mv r1, 1                         81920   81920
 55  nop : cjmp 59                          81920  204971
 56  nop : ld r2, [sp-3]                    40789   40789
 57  nop : add r2, r1, r2 : nop : nop       40789   40789
 58  nop : st r2, [sp-3]                    40789   40789
 59  add r0, r0, r1 : mv r1, 20             81920   81920
 60  ult r0, r1 : st r0, [sp-4]             81920   81920
 61  nop : cjmp 51                          81920  241664
```

Listing 7.15: A modified excerpt from the profiling information generated by ASIP Designer for the Motion benchmark running on the Tvliw.

```
 PC            Assembly  Exe-count  Cycles
---  ------------------  ---------  ------
128  c.swsp x10, 8(x2)        4096    4096
130  c.jal -70                4096    8192
132  c.srli x10, 31           4096    4096
134  c.swsp x10, 12(x2)       4096    4096
136  c.jal -76                4096    8192
138  c.lwsp x1, 12(x2)        4096    4096
140  c.srli x10, 31           4096    4096
142  c.add x1, x10            4096    4096
144  c.swsp x1, 12(x2)        4096    4096
146  c.jal -86                4096    8192
148  c.lwsp x1, 12(x2)        4096    4096
150  c.srli x10, 31           4096    4096
152  c.add x1, x10            4096    4096
154  c.swsp x1, 12(x2)        4096    4096
156  c.jal -96                4096    8192
158  c.lwsp x1, 12(x2)        4096    4096
160  c.srli x10, 31           4096    4096
162  c.add x1, x10            4096    4096
164  c.swsp x1, 12(x2)        4096    4096
166  c.jal -106               4096    8192
168  c.lwsp x1, 12(x2)        4096    4096
170  c.srli x10, 31           4096    4096
172  c.add x1, x10            4096    4096
174  c.swsp x1, 12(x2)        4096    4096
176  c.jal -116               4096    8192
178  c.lwsp x1, 12(x2)        4096    4096
180  c.srli x10, 31           4096    4096
182  c.add x1, x10            4096    4096
184  c.swsp x1, 12(x2)        4096    4096
186  c.jal -126               4096    8192
188  c.lwsp x1, 12(x2)        4096    4096
190  c.srli x10, 31           4096    4096
192  c.add x1, x10            4096    4096
194  c.swsp x1, 12(x2)        4096    4096
196  c.jal -136               4096    8192
198  c.lwsp x1, 12(x2)        4096    4096
200  c.srli x10, 31           4096    4096
202  c.add x1, x10            4096    4096
204  c.swsp x1, 12(x2)        4096    4096
206  c.jal -146               4096    8192
208  c.lwsp x1, 12(x2)        4096    4096
210  c.srli x10, 31           4096    4096
212  c.add x1, x10            4096    4096
214  c.swsp x1, 12(x2)        4096    4096
216  c.jal -156               4096    8192
218  c.lwsp x1, 12(x2)        4096    4096
220  c.srli x10, 31           4096    4096
222  c.add x1, x10            4096    4096
224  c.swsp x1, 12(x2)        4096    4096
226  c.jal -166               4096    8192
...
```

Listing 7.16: A modified excerpt from the profiling information generated by ASIP Designer for the Motion benchmark running on the Tzscale. Only a part of the compiled loop is shown, as the loop is partly unrolled.

7.1.5 RC6

```
 PC                                                                  Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
400  27 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ALU.out1 -> RF.5 ;                    2565     1
401  ALU.out1 -> RF.10, ..., ... ;                                                  2565     0
402  5 -> ALU.in1t.shl, RF.8 -> mul.in2, RF.8 -> mul.in1t.mul ;                     2565     2
403  ALU.out1 -> ALU.in1t.ior, RF.10 -> ALU.in2, ... ;                              2565     0
404  ALU.out1 -> ALU.in1t.xor, RF.9 -> ALU.in2, ... ;                               2565     1
405  mul.out1 -> ALU.in1t.add, mul.out1 -> ALU.in2, ALU.out1 -> RF.5 ;              2565     1
406  ALU.out1 -> ALU.in1t.add, RF.8 -> ALU.in2, ... ;                               2565     0
407  27 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ALU.out1 -> RF.9 ;                   2565     1
408  5 -> ALU.in1t.shl, ..., ALU.out1 -> RF.11 ;                                    2565     0
409  ALU.out1 -> ALU.in1t.ior, RF.11 -> ALU.in2, ... ;                              2565     1
410  ALU.out1 -> ALU.in1t.xor, RF.6 -> ALU.in2, ... ;                               2565     1
411  RF.10 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ALU.out1 -> RF.6 ;                  2565     1
412  32 -> ALU.in1t.sub, RF.10 -> ALU.in2, ALU.out1 -> RF.9 ;                       2565     0
413  RF.11 -> ALU.in1t.shl, RF.5 -> ALU.in2, ALU.out1 -> RF.10 ;                    2565     0
414  32 -> ALU.in1t.sub, RF.11 -> ALU.in2, ALU.out1 -> RF.16 ;                      2565     0
415  ALU.out1 -> ALU.in1t.shru, RF.5 -> ALU.in2, ... ;                              2565     0
416  RF.2 -> ALU.in1t.add, RF.4 -> ALU.in2, ALU.out1 -> RF.12 ;                     2565     1
417  -12 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ALU.out1 -> RF.5 ;                   2565     0
418  RF.12 -> ALU.in1t.ior, RF.16 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;            2565     0
419  ALU.out1 -> ALU.in1t.add, LSU.out1 -> ALU.in2, ... ;                           2565     0
420  ALU.out1 -> RF.12, ALU.out1 -> mul.in2, ALU.out1 -> mul.in1t.mul ;             2565     1
421  RF.10 -> ALU.in1t.shru, RF.6 -> ALU.in2, ... ;                                 2565     0
422  ALU.out1 -> ALU.in1t.ior, RF.9 -> ALU.in2, ... ;                               2565     1
423  mul.out1 -> ALU.in1t.add, mul.out1 -> ALU.in2, ALU.out1 -> RF.6 ;              2565     1
424  ALU.out1 -> ALU.in1t.add, RF.12 -> ALU.in2, ... ;                              2565     0
425  27 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ALU.out1 -> RF.9 ;                   2565     1
426  -8 -> ALU.in1t.add, RF.5 -> ALU.in2, ALU.out1 -> RF.10 ;                       2565     0
427  5 -> ALU.in1t.shl, RF.9 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;                 2565     0
428  RF.6 -> ALU.in1t.add, LSU.out1 -> ALU.in2, ALU.out1 -> RF.9 ;                  2565     0
429  ALU.out1 -> RF.6, ALU.out1 -> mul.in2, ALU.out1 -> mul.in1t.mul ;              2565     0
430  RF.9 -> ALU.in1t.ior, RF.10 -> ALU.in2, ... ;                                  2565     0
431  ALU.out1 -> ALU.in1t.xor, RF.8 -> ALU.in2, ... ;                               2565     1
432  mul.out1 -> ALU.in1t.add, mul.out1 -> ALU.in2, ALU.out1 -> RF.8 ;              2565     1
433  ALU.out1 -> ALU.in1t.add, RF.6 -> ALU.in2, ... ;                               2565     0
434  27 -> ALU.in1t.shru, ALU.out1 -> ALU.in2, ALU.out1 -> RF.9 ;                   2565     1
435  5 -> ALU.in1t.shl, ..., ALU.out1 -> RF.11 ;                                    2565     0
436  ALU.out1 -> ALU.in1t.ior, RF.11 -> ALU.in2, ... ;                              2565     1
437  ALU.out1 -> ALU.in1t.xor, RF.7 -> ALU.in2, ... ;                               2565     1
438  RF.10 -> ALU.in1t.shl, ALU.out1 -> ALU.in2, ALU.out1 -> RF.7 ;                  2565     1
439  32 -> ALU.in1t.sub, RF.10 -> ALU.in2, ALU.out1 -> RF.9 ;                       2565     0
440  32 -> ALU.in1t.sub, RF.11 -> ALU.in2, ALU.out1 -> RF.10 ;                      2565     0
441  ALU.out1 -> ALU.in1t.shru, RF.8 -> ALU.in2, ... ;                              2565     0
442  RF.11 -> ALU.in1t.shl, ..., ALU.out1 -> RF.13 ;                                2565     1
443  RF.13 -> ALU.in1t.ior, ALU.out1 -> ALU.in2, RF.5 -> LSU.in1t.ldw ;             2565     1
444  ALU.out1 -> ALU.in1t.add, LSU.out1 -> ALU.in2, ... ;                           2565     0
445  ALU.out1 -> RF.8, ALU.out1 -> mul.in2, ALU.out1 -> mul.in1t.mul ;              2565     1
446  RF.10 -> ALU.in1t.shru, RF.7 -> ALU.in2, ... ;                                 2565     0
447  ALU.out1 -> ALU.in1t.ior, RF.9 -> ALU.in2, ... ;                               2565     1
448  mul.out1 -> ALU.in1t.add, mul.out1 -> ALU.in2, ALU.out1 -> RF.7 ;              2565     1
449  ALU.out1 -> ALU.in1t.add, RF.8 -> ALU.in2, ... ;                               2565     0
...
```

Listing 7.17: A modified excerpt from the profiling information generated by `tcecc` for the RC6 benchmark running on the PeLoTTA. Only a part of the compiled loop is shown, as the loop is partly unrolled.

```
 PC                        Assembly  Exe-count  Cycles
---  ----------------------------  ---------  ------
302  lo r2,l0 . ll l3, (sp-18)          10260   10260
304  hi r7,l0 . sl l3, (sp-20)          10260   10260
306  mulss l2,r2,r7 . mv r6 ,r2         10260   10260
308  add l3,l2,l2 . sl l0, (sp-18)      10260   10260
310  muluu l2,r2,r6                     10260   10260
311  asl l3,l3,r3                       10260   10260
312  add l3,l2,l3 . ll l2, (sp-16)      10260   10260
314  lsl l3,l3, 1 . sl l1, (sp-16)      10260   10260
316  add l3,l0,l3 . sl l2, (sp-22)      10260   10260
318  lo r2,l1                           10260   10260
319  lsl l0,l3,r0 . mv r6 ,r2           10260   10260
321  hi r7,l1 . xz l1 ,a1               10260   10260
323  sub l1,l1,l0                       10260   10260
324  mulss l2,r2,r7                     10260   10260
325  lo r5,l1                           10260   10260
326  add l2,l2,l2                       10260   10260
327  muluu l1,r2,r6                     10260   10260
328  asl l2,l2,r3                       10260   10260
329  add l2,l1,l2 . ll l1, (sp-22)      10260   10260
331  lo r4,l0                           10260   10260
332  lsl l3,l3,r1                       10260   10260
333  or l0,l0,l3 . ll l3, (sp-16)       10260   10260
335  lsl l2,l2, 1                       10260   10260
336  add l3,l2,l3                       10260   10260
337  lsl l2,l3,r1                       10260   10260
338  lsl l3,l3,r0                       10260   10260
339  or l2,l2,l3                        10260   10260
340  xor l1,l1,l2 . xz l2 ,a1           10260   10260
342  sub l2,l2,l3                       10260   10260
343  lo r6,l2 . ll l2, (sp-20)          10260   10260
345  xor l2,l0,l2                       10260   10260
346  lo r7,l3                           10260   10260
347  lsr l3,l2,r6                       10260   10260
348  lsl l2,l2,r7                       10260   10260
349  or l2,l2,l3                        10260   10260
350  lsl l3,l1,r4                       10260   10260
351  lsr l0,l1,r5                       10260   10260
352  or l0,l0,l3 . ll l3, (a2+m0)       10260   10260
354  add l1,l2,l3 . ll l2, (a2+m0)      10260   10260
356  add l0,l0,l2                       10260   10260
```

Listing 7.18: A modified excerpt from the profiling information generated by ASIP Designer for the RC6 benchmark running on the Tdsp.

PC	Assembly	Exe-count	Cycles
80	mul r11, r2, r2 : mul r12, r0, r0 : nop : nop	10260	10260
81	lsl r11, r11, r4 : lsl r12, r12, r4: nop : nop	10260	10260
82	add r11, r2, r11 : add r12, r0, r12: nop : nop	10260	10260
83	lsl r13, r11, r6 : lsl r14, r12, r6: nop : nop	10260	10260
84	lsr r11, r11, r9 : lsr r12, r12, r9: nop : nop	10260	10260
85	or r11, r11, r13 : or r14, r12, r14: nop : nop	10260	10260
86	and r13, r8, r14 : and r12, r8, r11: nop : nop	10260	10260
87	xor r11, r1, r11 : xor r3, r3, r14 : nop : ld r1,	10260	10260
88	sub r15, r7, r13 : sub r14, r7, r12: nop : nop	10260	10260
89	lsl r15, r11, r13 : lsr r13, r11, r15: nop : ld r1	10260	10260
90	lsr r3, r3, r14 : lsl r12, r3, r12: nop : nop	10260	10260
91	or r12, r13, r15 : or r3, r3, r12 : nop : nop	10260	10260
92	add r0, r1, r12 : nop : mv r1, r2 : mv r13, r0	10260	10260
93	add r10, r4, r10 : add r2, r3, r11 : nop : nop	10260	10260
94	nop : nop : nop : mv r3, r13	10260	10260
95	nop : sle r10, r5 : nop : nop	10773	10773
96	nop : cjmp 80	10773	31806

Listing 7.19: A modified excerpt from the profiling information generated by ASIP Designer for the RC6 benchmark running on the Tvliw.

PC	Assembly	Exe-count	Cycles
780	mul x6,x10,x10	10260	10260
784	c.slli x6, 1	10260	10260
786	add x12,x10,x6	10260	10260
790	srli x6,x12,27	10260	10260
794	mul x7,x11,x11	10260	10260
798	c.slli x12, 5	10260	10260
800	c.slli x7, 1	10260	10260
802	or x12,x12,x6	10260	10260
806	c.xor x9, x12	10260	10260
808	add x14,x11,x7	10260	10260
812	srli x12,x14,27	10260	10260
816	c.slli x14, 5	10260	10260
818	c.or x14, x12	10260	10260
820	c.xor x13, x14	10260	10260
822	sub x14,x5,x6	10260	10260
826	sub x15,x5,x12	10260	10260
830	srl x14,x13,x14	10260	10260
834	srl x15,x9,x15	10260	10260
838	sll x9,x9,x12	10260	10260
842	sll x13,x13,x6	10260	10260
846	c.or x14, x13	10260	10260
848	lw x12,4(x1)	10260	10260
852	c.or x9, x15	10260	10260
854	c.mv x13, x11	10260	10260
856	add x11,x12,x9	10260	10260
860	lw x12,0(x1)	10260	10260
864	c.mv x9, x10	10260	10260
866	c.addi x8, -1	10260	10260
868	c.addi x1, 8	10260	10260
870	add x10,x12,x14	10260	10260
874	c.bnez x8,-94	10260	20007

Listing 7.20: A modified excerpt from the profiling information generated by ASIP Designer for the RC6 benchmark running on the Tzscale.

7.1.6 Finnish

```
 PC                                                                    Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
371  RF.12 -> ALU.in1t.ne, ALU.out1 -> ALU.in2, ... [IU_1x32.0=458] ;             11740     2
372  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bz1, ... ;                         11740     1
373  ..., ..., ... ;                                                              11740     1
374  20 -> ALU.in1t.add, RF.0 -> ALU.in2, ... ;                                   11740     3
375  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                         11740     1
376  6 -> ALU.in1t.shl, LSU.out1 -> ALU.in2, RF.18 -> RF.17 ;                     11740     2
377  -4 -> ALU.in1t.and, RF.16 -> ALU.in2, ALU.out1 -> RF.6 ;                     11740     0
378  RF.11 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... [IU_1x32.0=4032] ;           11740     0
379  ALU.out1 -> RF.19, ..., ALU.out1 -> LSU.in1t.ldw ;                           11740     1
380  8 -> ALU.in1t.shru, LSU.out1 -> ALU.in2, LSU.out1 -> RF.18 ;                 11740     1
381  IU_1x32.0 -> ALU.in1t.and, ALU.out1 -> LSU.in2, RF.6 -> ALU.in2 ;            11740     0
382  20 -> ALU.in1t.add, RF.0 -> ALU.in2, ALU.out1 -> RF.6 ;                      11740     0
383  RF.18 -> ALU.in1t.sxqw, ..., ALU.out1 -> RF.7 ;                              11740     0
384  255 -> ALU.in1t.and, RF.17 -> ALU.in2, ALU.out1 -> RF.8 ;                    11740     1
385  RF.6 -> ALU.in1t.xor, ALU.out1 -> ALU.in2, ... ;                             11740     0
386  RF.13 -> ALU.in1t.add, ALU.out1 -> ALU.in2, RF.19 -> LSU.in1t.stw ;           11740     1
387  ALU.out1 -> RF.6, RF.17 -> LSU.in2, ALU.out1 -> LSU.in1t.ldq ;                11740     0
388  LSU.out1 -> ALU.in1t.ne, RF.8 -> ALU.in2, ... [IU_1x32.0=394] ;              11740     0
389  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bnz1, ... ;                        11740     1
390  ..., ..., RF.7 -> LSU.in1t.stw ;                                             11740     1
391  RF.5 -> ALU.in1t.shl, 1 -> ALU.in2, ... [IU_1x32.0=399] ;                    11740     2
392  RF.4 -> ALU.in1t.xor, IU_1x32.0 -> gcu.pc.jump, ALU.out1 -> ALU.in2 ;         5963     1
393  ALU.out1 -> RF.4, ..., ... ;                                                 5963     0
394  ..., RF.18 -> LSU.in2, ... ;                                                 5963     2
395  RF.2 -> ALU.in1t.add, RF.15 -> ALU.in2, ... ;                                5777     2
396  ..., ..., RF.6 -> LSU.in1t.stq ;                                             5777     1
397  1 -> ALU.in1t.add, ..., ALU.out1 -> LSU.in1t.stq ;                           5777     2
398  ALU.out1 -> RF.15, ..., ... ;                                                5777     1
399  1 -> ALU.in1t.add, RF.5 -> ALU.in2, ... ;                                    5777     2
400  ALU.out1 -> RF.5, ..., ... ;                                                11740     1
401  8 -> ALU.in1t.ne, ALU.out1 -> ALU.in2, ... [IU_1x32.0=370] ;                 11740     2
402  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bnz1, ... ;                        11740     1
403  1 -> ALU.in1t.add, RF.16 -> ALU.in2, ... ;                                   11740     1
404  RF.9 -> ALU.in1t.add, RF.10 -> ALU.in2, ... ;                                11740     1
...
416  RF.9 -> ALU.in1t.add, RF.10 -> ALU.in2, ... ;                                5777     2
417  1 -> ALU.in1t.add, ..., ALU.out1 -> RF.4 ;                                   5777     1
418  1 -> ALU.in1t.add, RF.2 -> ALU.in2, ALU.out1 -> RF.10 ;                      5777     1
419  -1 -> ALU.in1t.add, RF.15 -> ALU.in2, ALU.out1 -> RF.2 ;                     5777     0
420  ALU.out1 -> RF.15, ..., ... ;                                                5777     0
421  0 -> ALU.in1t.ne, ALU.out1 -> ALU.in2, ... [IU_1x32.0=361] ;                 5777     2
422  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bz1, ... ;                         5777     1
423  ..., LSU.out1 -> LSU.in2, RF.4 -> LSU.in1t.stq ;                             5777     1
424  RF.14 -> ALU.in1t.gtu, RF.10 -> ALU.in2, ... [IU_1x32.0=415] ;               5777     1
425  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bnz1, ... ;                        5777     1
426  ..., ..., ... ;                                                             5777     1
```

Listing 7.21: A modified excerpt from the profiling information generated by `tcecc` for the Finnish benchmark running on the PeLoTTA.

```
 PC                        Assembly  Exe-count  Cycles
---  ------------------------------  ---------  ------
136  lsl r1,r1, -1 . mv a4 ,a3          11648   11648
138  li r7,32766                        11648   11648
140  and r7,r1,r7 . mv m1 ,r0           11648   11648
142  mv m2 ,r7                          11648   11648
143  li r7,6                            11648   11648
144  lsl r3,r3,r7 . add a4,m2           11648   11648
146  and r3,r2,r3 . ll l0, (a4)         11648   11648
148  xor r3,r3,r5 . li r7,-8            11648   11648
150  lo r3,l0 . mv m2 ,r3               11648   11648
152  li a6,9216                         11648   11648
154  lsl l0,l0,r7 . add a6,m2           11648   11648
156  lw r7, (a6)                        11648   11648
157  and r5,r3,r6 . mv r3 ,r5           11648   11648
159  seq r5,r7                          11648   11648
160  jrc 7                              11648   11648
161  li r1,1                            11648   11648
162  sl l0, (a4)                        11648   11648
163  mv a4 ,a2                           5732    5732
164  sw r5, (a6)                         5732    5732
165  jr 7 . add a4,m1                    5732    5732
167  add r0,r0,r1 . sw r5, (a4)          5732    5732
169  lsl r7,r1,r4                        5916    5916
170  lw r1, (sp-8)                       5916    5916
171  xor r7,r1,r7 . li r1,1              5916    5916
173  sw r7, (sp-8)                       5916    5916
174  add r4,r1,r4 . li r7,8             11648   11648
176  seq r4,r7                          11648   11648
177  jrcn 75                            11648   11648
178  lw r7, (sp-4)                      11648   11648
179  nop                                11648   11648
...
192  sltu r2,r3                          5732    5732
193  jrc 34                              5732    5732
194  nop                                 5732    5732
195  nop                                 5732    5732
...
229  add r0,r0,r4                        5732    5732
230  seq r0,0                            5732    5732
231  jrcn -41                            5732    5732
232  lw r5, (a2+m0)                      5732    5732
233  add r2,r1,r2 . sw r5, (a0+m0)       5732    5732
...
254  add r1,r1,r7                       11648   11648
255  li r7,128                          11648   11648
257  seq r1,r7                          11648   11648
258  jrcn -124                          11648   11648
259  sw r1, (sp-4)                      11648   11648
260  nop                                11648   11648
```

Listing 7.22: A modified excerpt from the profiling information generated by ASIP Designer for the Finnish benchmark running on the Tdsp.

```
 PC                                              Assembly  Exe-count  Cycles
---  ------------------------------------------------------  ---------  ------
 79  lsr r13, r7, r6 : lsl r14, r13, r8: nop : nop              11740   11740
 80  xor r13, r4, r14 : nop : nop : mv m3, r13                  11740   11740
 81  nop : and r13, r10, r13: nop : nop                         11740   11740
 82  nop : nop : mv p1, r13 : p3=p0+m3                          11740   11740
 83  nop : nop : mv r13, r4 : ld r14, [p3]                      11740   11740
 84  lsr r14, r14, r5 : and r4, r12, r14: nop : p1=p1+m         11740   11740
 85  nop : nop : nop : st r14, [p3]                             11740   11740
 86  nop : nop : nop : nop                                      11740   11740
 87  nop : nop : nop : ld r14, [p1]                             11740   11740
 88  nop : eq r4, r14 : nop : nop                               11740   11740
 89  nop : cjmp 93                                              11740   29443
 90  nop : nop : p2=p2+m0 : p3=p2+m1                             5777    5777
 91  nop : nop : st r4, [p1] : st r4, [p3]                       5777    5777
 92  nop : ujmp 95                                               5777   11554
 93  nop : lsl r14, r3, r9 : nop : nop                           5963    5963
 94  nop : xor r11, r11, r14: nop : nop                          5963    5963
 95  nop : add r9, r3, r9 : nop : nop                           11740   11740
 96  nop : neq r5, r9 : nop : nop                               11740   11740
 97  nop : cjmp 147                                             11740   33753
...
106  nop : ule r0, r2 : nop : nop                                5777    5777
107  nop : cjmp 129                                              5777   17331
...
129  add r0, r0, r3 : add r4, r3, r4 : nop : ld lr, [p0          5777    5777
130  nop : nop : nop : st lr, [p1+=m0]                           5777    5777
131  nop : slt r4, r1 : nop : nop                                7244    7244
132  nop : cjmp 106                                              7244   20265
...
147  nop : add r7, r3, r7 : nop : nop                           11740   11740
148  nop : ult r7, r1 : nop : nop                               11832   11832
149  nop : cjmp 79                                              11832   35404
```

Listing 7.23: A modified excerpt from the profiling information generated by ASIP Designer for the Finnish benchmark running on the Tvliw.

```
 PC            Assembly  Exe-count  Cycles
---  ------------------  ---------  ------
288  slli x14,x9,6           11648   11648
292  and x9,x12,x11          11648   11648
296  c.and x14, x13          11648   11648
298  c.xor x14, x9           11648   11648
300  andi x8,x10,-4          11648   11648
304  c.add x8, x3            11648   11648
306  c.mv x9, x12            11648   11648
308  c.lw x12, 0(x8)         11648   11648
310  add x21,x14,x1          11648   11648
314  srli x15,x12,8          11648   11648
318  slli x14,x12,24         11648   11648
322  c.sw x15, 0(x8)         11648   11648
324  lb x22,0(x21)           11648   11648
328  srai x8,x14,24          11648   11648
332  bne x22,x8,14           11648   17380
336  sll x21,x16,x20          5916    5916
340  xor x5,x5,x21            5916    5916
344  c.j 16                   5916   11832
346  add x22,x4,x31           5732    5732
350  c.addi x31, 1            5732    5732
352  sb x12,0(x22)            5732    5732
356  sb x12,0(x21)            5732    5732
360  c.addi x20, 1           11648   11648
362  bne x17,x20,134         11648   21840
...
390  bltu x7,x18,46           5732   11464
...
436  lbu x3,0(x1)             5732    5732
440  c.addi x11, -1           5732    5732
442  sb x3,0(x6)              5732    5732
446  c.addi x7, 1             5732    5732
448  c.addi x1, 1             5732    5732
450  c.addi x6, 1             5732    5732
452  c.bnez x11,-62           5732   10645
...
496  c.addi x10, 1           11648   11648
498  bne x19,x10,-210        11648   23205
```

Listing 7.24: A modified excerpt from the profiling information generated by ASIP Designer for the Finnish benchmark running on the Tzscale.

7.1.7 CNN

```
 PC                                                                  Assembly  Exe-count  NOPs
---  ------------------------------------------------------------------------  ---------  ----
107  RF.12 -> ALU.in1t.and, RF.10 -> ALU.in2, ALU.out1 -> RF.2 ;                   49000     0
108  -1 -> ALU.in1t.add, RF.9 -> ALU.in2, ALU.out1 -> RF.5 ;                      49000     0
109  ALU.out1 -> RF.9, ..., ... ;                                                 49000     0
110  0 -> ALU.in1t.ne, ALU.out1 -> ALU.in2, ... [IU_1x32.0=76] ;                   49000     2
111  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bz1, ... ;                          49000     1
112  ..., ..., ... ;                                                              49000     1
113  32 -> ALU.in1t.add, RF.0 -> ALU.in2, ... ;                                    49000     3
114  ..., ..., ALU.out1 -> LSU.in1t.ldw ;                                          49000     1
115  ..., ..., ... ;                                                              49000     2
116  2 -> ALU.in1t.shl, RF.2 -> ALU.in2, ... ;                                     49000     3
117  LSU.out1 -> ALU.in1t.add, ALU.out1 -> ALU.in2, ... [IU_1x32.0=32] ;           49000     1
118  1 -> ALU.in1t.add, RF.5 -> ALU.in2, ALU.out1 -> LSU.in1t.ldw ;               49000     1
119  ALU.out1 -> ALU.in1t.gtu, IU_1x32.0 -> ALU.in2, ALU.out1 -> RF.10 ;          49000     0
120  ALU.out1 -> ALU.in1t.sub, 1 -> ALU.in2, ALU.out1 -> RF.11 ;                  49000     0
121  RF.5 -> ALU.in1t.shru, LSU.out1 -> ALU.in2, ALU.out1 -> RF.12 ;              49000     0
122  1 -> ALU.in1t.and, ALU.out1 -> ALU.in2, ... ;                                49000     0
123  0 -> ALU.in1t.eq, ALU.out1 -> ALU.in2, ... [IU_1x32.0=106] ;                  49000     1
124  ALU.out1 -> gcu.cond, IU_1x32.0 -> gcu.pc.bnz1, ... ;                         49000     1
125  2 -> ALU.in1t.add, RF.8 -> ALU.in2, ... ;                                     49000     1
```

Listing 7.25: A modified excerpt from the profiling information generated by `tcecc` for the CNN benchmark running on the PeLoTTA.

```
 PC                    Assembly  Exe-count  Cycles
---  --------------------------  ---------  ------
 78  lo r5,l1 . ll a3, (a1)          49000   49000
 80  lsl r5,r5, 1                    49000   49000
 81  lo r5,l2 . mv m1 ,r5            49000   49000
 83  lsl l1,l0,r5 . add a3,m1        49000   49000
 85  add l2,l0,l2 . ll l3, (a3)      49000   49000
 87  and l1,l1,l3 . xz l3 ,a2        49000   49000
 89  sleu l2,l3 . mv l3 ,l2          49000   49000
 91  li l2,0                         49000   49000
 92  mvcn l2,l3 . li r5,1            49000   49000
 94  mvcn r5,r0                      49000   49000
 95  seq l1,0                        49000   49000
 96  jrc 28                          49000   49000
 97  xz l1,r5 . ll l3, (sp-16)       49000   49000
 99  add l1,l1,l3                    49000   49000
...
127  add r4,r3,r4                    49000   49000
128  slt r4,r2                       49000   49000
129  jrc -53                         49000   49000
130  sl l1, (sp-16)                  49000   49000
131  nop                             49000   49000
```

Listing 7.26: A modified excerpt from the profiling information generated by ASIP Designer for the CNN benchmark running on the Tdsp.

```
 PC                                                Assembly  Exe-count  Cycles
---  ------------------------------------------------------  ---------  ------
112  add r4, r0, r4 : lsl r5, r0, r4 : nop : ld m1, [p1         49000   49000
113  nop : ule r4, r1 : nop : nop                               49000   49000
114  nop : nop : nop : p4=p2+m1                                 49000   49000
115  nop : nop : nop : ld r6, [p4]                              49000   49000
116  and r5, r5, r6 : cjmp 118                                  49000  146000
...
118  nop : eq r3, r5 : nop : nop                                49000   49000
119  nop : cjmp 135                                             49000  142340
...
135  nop : add r2, r0, r2 : nop : nop                           49000   49000
136  nop : nop : nop : ld r5, [p3]                              50000   50000
137  nop : slt r2, r5 : nop : nop                               50000   50000
138  nop : cjmp 112                                             50000  149000
...
236  nop : mv r2, 32768                                         24660   24660
237  nop : slt r1, r2 : nop : nop                               24660   24660
238  nop : cjmp 240                                             24660   63496
...
240  nop : mv r0, 32767                                         24660   24660
241  nop : sgt r2, r0 : nop : nop                               24660   24660
242  nop : cjmp 244                                             24660   73980
...
244  nop : ijmp lr                                              24660   49320
```

Listing 7.27: A modified excerpt from the profiling information generated by ASIP Designer for the CNN benchmark running on the Tvliw.

```
 PC            Assembly  Exe-count  Cycles
---  ------------------  ---------  ------
134  c.lw x11, 0(x12)        49000   49000
136  slli x18,x5,2           49000   49000
140  c.add x11, x18          49000   49000
142  c.lw x11, 0(x11)        49000   49000
144  sll x8,x1,x17           49000   49000
148  addi x18,x17,1          49000   49000
152  c.and x11, x8           49000   49000
154  c.li x17, 1             49000   49000
156  bltu x6,x18,6           49000   50000
160  c.li x17, 0             48000   48000
162  c.add x5, x17           49000   49000
164  c.li x17, 0             49000   49000
166  bltu x6,x18,6           49000   50000
170  c.mv x17, x18           48000   48000
172  c.beqz x11,56           49000   93340
...
228  c.addi x4, 1            49000   49000
230  c.addi x7, 2            49000   49000
232  blt x4,x3,-98           49000   97000
```

Listing 7.28: A modified excerpt from the profiling information generated by ASIP Designer for the CNN benchmark running on the Tzscale.

Bibliography

[1] H. Corporaal and H. Mulder, "Move: A framework for high-performance processor design," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 692–701.

[2] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors.* Springer International Publishing, 2017, pp. 147–164. [Online]. Available: https://doi.org/10.1007/978-3-319-49679-5_8

[3] 2020. [Online]. Available: http://openasip.org/move/DelftMoveSite/MOVE/

[4] "Tta-based co-design environment (tce) tools," 2020. [Online]. Available: http://openasip.org/

[5] C. Strydis, C. Kachris, and G. Gaydadjiev, "Impbench -a novel benchmark suite for biomedical, microelectronic implants," in *IC - SAMOS 2008*, H. W. Najjar, Ed. IEEE Society, 2008, pp. 82–91, null ; Conference date: 21-07-2008 Through 24-07-2008.

[6] C. Strydis, D. Dave, and G. Gaydadjiev, "Impbench revisited: an extended characterization of implant-processor benchmarks," in *2010 Intl. conf. on embedded computer systems: architectures, modeling and simulation*, s.n., Ed. IEEE Society, 2010, pp. 126–135, iC-SAMOS 2010 ; Conference date: 19-07-2010 Through 22-07-2010.

[7] C. Strydis, "Universal processor architecture for biomedical implants: The sims project," 2011.

[8] L. Shepherd and C. Toumazou, "Towards an implantable ultra-low power biochemical signal processor for blood and tissue monitoring," in *2005 IEEE International Symposium on Circuits and Systems*, 2005, pp. 5226–5229 Vol. 5.

[9] K. Kim, J. H. Kim, S. Gweon, M. Kim, and H. J. Yoo, "A 0.5-v sub-10-µw 15.28-µ$\Omega\sqrt{Hz}$ bio-impedance sensor ic with sub-1° phase error," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 8, pp. 2161–2173, 2020.

[10] J. Ji and K. D. Wise, "An implantable cmos analog signal processor for multiplexed microelectrode recording arrays," in *IEEE 4th Technical Digest on Solid-State Sensor and Actuator Workshop*, 1990, pp. 107–110.

[11] M. Haruta, T. Kobayashi, C. Kitsumoto, T. Noda, K. Sasagawa, T. Tokuda, and J. Ohta, "Development of a cmos-based implantable device for wide-area brain functional imaging," in *2012 IEEE International Meeting for Future of Electron Devices, Kansai*, 2012, pp. 1–2.

[12] L. B. Leene and T. G. Constandinou, "A 2.7µw/mips, 0.88gops/mm2 distributed processor for implantable brain machine interfaces," in *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2016, pp. 360–363.

[13] S. Omar, H. Mostafa, T. Ismail, and S. Gabran, "Low-power implantable seizure detection processor," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2015, pp. 496–497.

[14] T. L. Yearwood and L. T. Perryman, "Peripheral neurostimulation with a microsize wireless stimulator," *Stimulation of the Peripheral Nervous System*, vol. 29, pp. 168–191, 2016.

[15] B. L. Bemelmans, A. R. Mundy, and M. D. Craggs, "Neuromodulation by implant for treating lower urinary tract symptoms and dysfunction," *European urology*, vol. 36, no. 2, pp. 81–91, 1999.

[16] T. G. Constandinou, J. Georgiou, and C. Toumazou, "A fully-integrated semicircular canal processor for an implantable vestibular prosthesis," in *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, 2008, pp. 81–84.

[17] M. D. Linderman and T. H. Meng, "A low power merge cell processor for real-time spike sorting in implantable neural prostheses," in *2006 IEEE International Symposium on Circuits and Systems*, 2006, pp. 4 pp.–4109.

[18] A. M. Sodagar, K. D. Wise, and K. Najafi, "A fully integrated mixed-signal neural processor for implantable multichannel cortical recording," *IEEE Transactions on Biomedical Engineering*, vol. 54, no. 6, pp. 1075–1088, 2007.

[19] S. Chede and K. Kulat, "Software related current/energy estimation in processor based implantable pacemaker," in *2008 IEEE Region 10 and the Third international Conference on Industrial and Information Systems*, 2008, pp. 1–6.

[20] Z. Gorup, D. Stajer, and M. Noc, "Signal analysis systems for optimal timing of electrical defibrillation," in *2000 10th Mediterranean Electrotechnical Conference. Information Technology and Electrotechnology for the Mediterranean Countries. Proceedings. MeleCon 2000 (Cat. No.00CH37099)*, vol. 2, 2000, pp. 694–697 vol.2.

[21] F. Le, J. C. C. Lo, X. Qiu, S. R. Lee, X. Li, C. Tsui, and W. Ki, "An implantable medical device for transcorneal electrical stimulation: Packaging structure, process flow, and toxicology test," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 6, no. 8, pp. 1174–1180, 2016.

[22] M. Sawan, F. Duval, S. Pourmehdi, and J. Mouine, "A new multichannel bladder stimulator," in *[1990] Proceedings. Third Annual IEEE Symposium on Computer-Based Medical Systems*, 1990, pp. 190–196.

[23] H. Bernhard, C. Stieger, and Y. Perriard, "Micro-actuator for new implantable hearing device," in *Conference Record of the 2006 IEEE Industry Applications Conference Forty-First IAS Annual Meeting*, vol. 1, 2006, pp. 369–372.

[24] C. Zhao, K. E. Knisely, and K. Grosh, "Modeling, fabrication, and testing of a mems multichannel aln transducer for a completely implantable cochelar implant," in *2017 19th International Conference on Solid-State Sensors, Actuators and Microsystems (TRANSDUCERS)*, 2017, pp. 16–19.

[25] M. A. Siddiqi and C. Strydis, "Imd security vs. energy: are we tilting at windmills? poster," in *Proceedings of the 16th ACM international conference on computing frontiers*, 2019, pp. 283–285.

[26] G. McLaughlin and M. Imran, "Set it and forget it: Innovations in implantable medical technology," *IEEE Solid-State Circuits Magazine*, vol. 4, no. 2, pp. 30–33, 2012.

[27] D. Jeon, Y. Chen, Y. Lee, Y. Kim, Z. Foo, G. Kruger, H. Oral, O. Berenfeld, Z. Zhang, D. Blaauw, and D. Sylvester, "An implantable 64nw ecg-monitoring mixed-signal soc for arrhythmia diagnosis," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 416–417.

[28] E. D. Marsman, R. M. Senger, G. A. Carichner, S. Kubba, M. S. McCorquodale, and R. B. Brown, "Dsp architecture for cochlear implants," in *2006 IEEE International Symposium on Circuits and Systems*. IEEE, 2006, pp. 4–pp.

[29] T. Instruments, "Tms320c6000programmer's guide," pp. 1–2, 7 2011. [Online]. Available: https://www.ti.com/lit/ug/spru198k/spru198k.pdf

[30] H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons, Inc., 1997.

[31] "Tta-based co-design environment (tce) tools | about transport-triggered architectures," 2020. [Online]. Available: http://openasip.org/tta.html

[32] J. IJzerman, T. Viitanen, P. Jääskeläinen, H. Kultala, L. Lehtonen, M. Peemen, H. Corporaal, and J. Takala, "Aivotta: an energy efficient programmable accelerator for cnn-based object recognition," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018, pp. 28–37.

[33] J. Žádník and J. Takala, "Low-power programmable processor for fast fourier transform based on transport triggered architecture," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 1423–1427.

[34] N. Behmann, C. Seifert, G. Paya-Vaya, H. Blume, P. Jääskeläinen, J. Multanen, H. Kultala, J. Takala, J. Thiemann, and S. van de Par, "Customized high performance low power processor for binaural speaker localization," in *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2016, pp. 392–395.

[35] J. Multanen, H. Kultala, P. Jääskeläinen, T. Viitanen, K. Tervo, and J. Takala, "Lotta: Energy-efficient processor for always-on applications," in *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*, 10 2018.

[36] C. Liem, T. May, and P. Paulin, "Instruction-set matching and selection for dsp and asip code generation," in *Proceedings of European Design and Test Conference EDAC-ETC-EUROASIC*. IEEE, 1994, pp. 31–37.

[37] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann, 2005.

[38] K. Keutzer, S. Malik, and A. R. Newton, "From asic to asip: The next design discontinuity," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 2002, pp. 84–90.

[39] "Modeling of application specific processors for the asip designer environment version 2019.09," Synopsys, p. 6.

[40] Synopsys, "Asip designer," 2021. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[41] "The move project," 2020. [Online]. Available: http://openasip.org/move.html

[42] C. C. Aggarwal *et al.*, "Neural networks and deep learning," *Springer*, vol. 10, pp. 978–3, 2018.

[43] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[44] A. Enterprises, "Zlib compressed data format specification version 3.3," 1996. [Online]. Available: https://www.ietf.org/rfc/rfc1950.txt

[45] ——, "Deflate compressed data format specification version 1.3," 1996. [Online]. Available: https://www.ietf.org/rfc/rfc1951.txt

[46] D. D. Journal, "Dr. dobbs data compression contest," 1991. [Online]. Available: https://www.drdobbs.com/the-ddj-data-compression-contest/184408494

[47] M. Ohta and M. Matsui, "A description of the misty1 encryption algorithm," 2000. [Online]. Available: https://www.ietf.org/rfc/rfc2994.txt

[48] P. S. Cross, R. Künnemeyer, C. R. Bunt, D. A. Carnegie, and M. J. Rathbone, "Control, communication and monitoring of intravaginal drug delivery in dairy cows," *International Journal of Pharmaceutics*, vol. 282, no. 1, pp. 35 – 44, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378517304002911

[49] M. Lind, "Simple 3-layer neural network for mnist handwriting recognition." [Online]. Available: https://mmlind.github.io/posts/simple_3-layer_neural_network_for_mnist_handwriting_recognition/

[50] H. Lanmueller, Z. Ashley, E. Unger, H. Sutherland, M. Reichel, M. Russold, J. Jarvis, W. Mayr, and S. Salmons, "Implantable device for long-term electrical stimulation of denervated muscles in rabbits," *Medical & biological engineering & computing*, vol. 43, pp. 535–40, 08 2005.

[51] R. G. Dennis, D. E. Dow, and J. A. Faulkner, "An implantable device for stimulation of denervated muscles in rats," *Medical Engineering & Physics*, vol. 25, no. 3, pp. 239 – 253, 2003, 3-D Computer Visualization and Animation of Medical Images. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1350453302001935

[52] Microchip, "Pic16c5x eprom/rom-based 8-bit cmos microcontroller series," 2013. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/30292D.pdf

[53] ——, "Pic16f87x 28/40-pin 8-bit cmos flash microcontrollers," 2013. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/30292D.pdf

[54] Renesas, "M16c product guide," 2006. [Online]. Available: http://olympus.dementix.org/misc/DiCE/Renesas%20CPU/M16C_Product_Guide.pdf

[55] C.-K. Liang, J.-J. Chen, C.-L. Chung, C.-L. Cheng, and C.-C. Wang, "An implantable bi-directional wireless transmission system for transcutaneous biological signal recording," *Physiological measurement*, vol. 26, pp. 83–97, 03 2005.

[56] T. Instruments, "Msp430f15x, msp430f16x, msp430f161x mixed signal microcontroller," 2011. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430f1611.pdf

[57] S. Heller, K. Allinger, U. Pelz, and P. Woias, "Implantable wireless ultra-low power data logger for temperature measurements in animal brains," in *MikroSystemTechnik 2017; Congress*, 2017, pp. 1–4.

[58] T. Instruments, "Msp430fr596x,msp430fr594xmixed-signalmicrocontrollers," 2018. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430fr5949.pdf

[59] A. Tantin, A. Letourneau, M. Zgaren, S. Hached, I. Clausen, and M. Sawan, "Implantable mics-based wireless solution for bladder pressure monitoring," in *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2017, pp. 1–4.

[60] T. Instruments, "Msp430fr599x, msp430fr596x mixed-signal microcontrollers," 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[61] M. Hügle, S. Heller, M. Watter, M. Blum, F. Manzouri, M. Dumpelmann, A. Schulze-Bonhage, P. Woias, and J. Boedecker, "Early seizure detection with an energy-efficient convolutional neural network on an implantable microcontroller," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–7.

[62] S. Heller, M. Hügle, I. Nematollahi, F. Manzouri, M. Dümpelmann, A. Schulze-Bonhage, J. Boedecker, and P. Woias, "Hardware implementation of a performance and energy-optimized convolutional neural network for seizure detection," in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2018, pp. 2268–2271.

[63] P. T. Wang, E. Camacho, M. Wang, Y. Li, S. J. Shaw, M. Armacost, H. Gong, D. Kramer, B. Lee, R. A. Andersen, C. Y. Liu, P. Heydari, Z. Nenadic, and A. H. Do, "A benchtop system to assess the feasibility of a fully independent and implantable brain-machine interface," *Journal of Neural Engineering*, vol. 16, no. 6, p. 066043, nov 2019. [Online]. Available: https://doi.org/10.1088/1741-2552/ab4b0c

[64] H. Peters, R. Sethuraman, A. Beric, P. Meuwissen, S. Balakrishnan, C. A. A. Pinto, W. Kruijtzer, F. Ernst, G. Alkadi, J. Van Meerbergen *et al.*, "Application specific instruction-set processor template for motion estimation in video applications," *IEEE transactions on circuits and systems for video technology*, vol. 15, no. 4, pp. 508–527, 2005.

[65] Y. H. Yassin, P. G. Kjeldsberg, J. Hulzink, I. Romero, and J. Huisken, "Ultra low power application specific instruction-set processor design for a cardiac beat detector algorithm," in *2009 NORCHIP*. IEEE, 2009, pp. 1–4.

[66] A. Chormoviti, N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "Enhancing embedded processors with specific instruction set extensions for network applications," in *2005 IEEE Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.* IEEE, 2005, pp. 199–203.

[67] J. Constantin, A. Dogan, O. Andersson, P. Meinerzhagen, J. N. Rodrigues, D. Atienza, and A. Burg, "Tamarisc-cs: An ultra-low-power application-specific processor for compressed sensing," in *2012 IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC).* IEEE, 2012, pp. 159–164.

[68] M. Alizadeh and M. Sharifkhani, "Extending risc-v isa for accelerating the h. 265/hevc deblocking filter," in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2018, pp. 126–129.

[69] R. Andri, T. Henriksson, and L. Benini, "Extending the risc-v isa for efficient rnn-based 5g radio resource management," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[70] A. Alarifi and A. A. AlZubi, "Memetic search optimization along with genetic scale recurrent neural network for predictive rate of implant treatment," *Journal of medical systems*, vol. 42, no. 11, pp. 1–7, 2018.

[71] J. N. Y. Aziz, R. Karakiewicz, R. Genov, B. L. Bardakjian, M. Derchansky, and P. L. Carlen, "Towards real-time in-implant epileptic seizure prediction," in *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, 2006, pp. 5476–5479.

[72] A. W. Chiu, S. Daniel, H. Khosravani, P. L. Carlen, and B. L. Bardakjian, "Prediction of seizure onset in an in-vitro hippocampal slice model of epilepsy using gaussian-based and wavelet-based artificial neural networks," *Annals of biomedical engineering*, vol. 33, no. 6, pp. 798–810, 2005.

[73] D. Wu, K. Warwick, Z. Ma, M. N. Gasson, J. G. Burgess, S. Pan, and T. Z. Aziz, "Prediction of parkinson's disease tremor onset using a radial basis function neural network based on particle swarm optimization," *International journal of neural systems*, vol. 20, no. 02, pp. 109–116, 2010.

[74] D. Sussillo, P. Nuyujukian, J. M. Fan, J. C. Kao, S. D. Stavisky, S. Ryu, and K. Shenoy, "A recurrent neural network for closed-loop intracortical brain–machine interface decoders," *Journal of neural engineering*, vol. 9, no. 2, p. 026027, 2012.

[75] J. Dethier, P. Nuyujukian, S. I. Ryu, K. V. Shenoy, and K. Boahen, "Design and validation of a real-time spiking-neural-network decoder for brain–machine interfaces," *Journal of neural engineering*, vol. 10, no. 3, p. 036008, 2013.

[76] N. Mamun, S. Khorram, and J. H. Hansen, "Convolutional neural network-based speech enhancement for cochlear implant recipients," *arXiv preprint arXiv:1907.02526*, 2019.

[77] U. Ahmad, H. Song, A. Bilal, S. Saleem, and A. Ullah, "Securing insulin pump system using deep learning and gesture recognition," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, pp. 1716–1719.

[78] H. Rathore, A. K. Al-Ali, A. Mohamed, X. Du, and M. Guizani, "A novel deep learning strategy for classifying different attack patterns for deep brain implants," *IEEE Access*, vol. 7, pp. 24 154–24 164, 2019.

[79] M. A. Ozdemir, O. Guren, O. K. Cura, A. Akan, and A. Onan, "Abnormal ecg beat detection based on convolutional neural networks," in *2020 Medical Technologies Congress (TIPTEKNO)*. IEEE, 2020, pp. 1–4.

[80] A. Waterman and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, document version 2.2," RISC-V Foundation, 5 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[81] Synopsys, "Asip designer tzscale processor manual p-2019.09," 2019.

[82] ——, "Asip designer tvliw core processor manual p-2019.09," 2019.

[83] ——, "Asip designer tdsp core processor manual p-2019.09," 2019.

[84] EEMBC, "Coremark." [Online]. Available: https://www.eembc.org/coremark/

[85] ——, "Github - coremark," 3 2021. [Online]. Available: https://github.com/eembc/coremark

[86] ——, "Ulpmark." [Online]. Available: https://www.eembc.org/ulpmark/

[87] ——, "Ulpmark-coreprofile." [Online]. Available: https://www.eembc.org/ulpmark/ulp-cp/

[88] ——, "Ulpmark-peripheralprofile." [Online]. Available: https://www.eembc.org/ulpmark/ulp-pp/

[89] ——, "Ulpmark-coremark." [Online]. Available: https://www.eembc.org/ulpmark/ulp-cm/

[90] ——, "Securemark." [Online]. Available: https://www.eembc.org/securemark/

[91] ——, "Mlmark." [Online]. Available: https://www.eembc.org/mlmark/

[92] ——, "Ulpmark-ml." [Online]. Available: https://www.eembc.org/ulpmark/ulp-ml/

[93] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[94] Y. LeCun, C. Cortes, and C. J.C. Burges, "The mnist database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[95] Altera, "Cyclone iv fpga device family overview," 2009. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyiv-51001.pdf

[96] Synopsys, "Asip designer go user manual p-2019.09," 2019.

[97] G. et al., "Compression of random data," 2020. [Online]. Available: https://biit.cs.ut.ee/~vilo/edu/2002-03/Tekstialgoritmid_I/Loengud/Loeng7_Compression/www.faqs.org/faqs/compression-faq/part1/section-8.html

[98] "Iso/iec 9899:1999 specification, tc3," p. 80.

[99] M. Lind, "Github - mnist-3lnn." [Online]. Available: https://github.com/mmlind/mnist-3lnn

[100] ChinaQMTECH, "China qmtech cyclone iv ep4ce15," 2019.

[101] J. Smit, "Gitlab - automation of synthesis and simulations," 4 2021. [Online]. Available: https://gitlab.com/jrhrsmit/thesis-automation

[102] Arm, "Cortex m0+ - arm developer," 2021. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0-plus