Master's Thesis

# of Simon de Vegt

Prodrive Technologies

Compiling Motion Control Algorithms for the PD-CPU Instruction Set Architecture.

Reference:	DSD6001206302R04
Reference date:	2021-03-15
Author(s):	Simon de Vegt
TUe Supervisor:	Kees Goossens
Prodrive Supervisor:	Jasper Kuijsten
Status:	Accepted

Public

Document history

Rel.	Date	Changes
R01	2021-02-15	Initial release, for review purposes
R02	2021-02-22	Restructured document, for review purposes
R03	2021-03-01	Fixed review comments, thesis submission
R04	2021-03-15	Changed document confidentiality status to Public.

Name	R01	R02	R03	R04
TUe				
Kees Goossens	x	x	x	
Kanishkan Vadivel	x	x	x	
Prodrive Technologies				
Maik van Kranenburg	x	x	x	
Jasper Kuijsten	x	x	x	
Simon de Vegt	A	A	A	A

Dank jullie wel
Thank you
Danke schön
Baguette

SIMON DE VEGT

Compiling Motion Control Algorithms for the PD-CPU Instruction Set Architecture

Abstract Prodrive Technologies has automated the process of creating FPGA implementations from a Simulink model to a great extend. A missing piece of the puzzle is currently a solution for automated implementation of a medium latency & high resource consuming floating point control algorithm. This thesis proposes a novel instruction set architecture which consists entirely of floating point instructions. The ISA is specifically designed for computing control algorithms on a soft-core. A new soft-core has been proposed, which implements this novel ISA. Finally, a simulator has been developed to obtain performance statistics and validate the output of the toolchain.

By evaluating the developed toolchain with a set of reference models it used up to 6 times less FPGA resources than an HLS solution while staying within the latency requirements. But even though the code complexity of the sampled control algorithms was low, to maintain and/or contribute to the developed LLVM-backend still has a huge (knowledge) barrier to entry. The existing solutions can be improved by knowledge obtained during this graduation project. Therefore, the advice to Prodrive Technologies is to improve the already existing solutions without integrating the developed toolchain as a whole. This will result in a more maintainable and therefore sustainable solution.

Contents

List of Figures

List of Tables

List of Listings

Internal References (Available Upon Request)

[1.1]	Title:	SPD of Simulink to RTL - Design Process Optimisation
	Author (Company):	Moş, Paul (Prodrive Technologies)
	ID, Version, Date:	SPD6001202672, R01, 2020-04-20
	File:	SPD6001202672R01
[1.2]	Title:	Simulink to RTL - Design Process Optimisation
	Author (Company):	Moş, Paul (Prodrive Technologies)
	ID, Version, Date:	DSD6001202673, R01, 2020-04-20
	File:	DSD6001202673R01
[1.3]	Title:	SPD of PD-CPU Instruction Set Architecture
	Author (Company):	de Vegt, Simon (Prodrive Technologies)
	ID, Version, Date:	SPD6001203473, R03, 2021-03-15
	File:	SPD6001203473R03
[1.4]	Title:	Pre-Study Document of PD-CPU Compiler
	Author (Company):	de Vegt, Simon (Prodrive Technologies)
	ID, Version, Date:	PSD6001204959, R02, 2020-08-22
	File:	PSD6001204959R02

External References

[2.1]	Title:	AArch64: ARM's 64-bit architecture
	Author (Company):	Northover, Tim (ARM)
	Date:	2012-11-08
	File:	https://llvm.org/devmtg/2012-11/Northover-AArch64.pdf
[2.2]	Title:	Comparing Fixed- and Floating-Point DSPs
	Author (Company):	Frantz, Gene; Simar, Ray (Texas Instruments)
	Date:	2004
	File:	https://www.ti.com/lit/wp/spry061/spry061.pdf?ts=15952388181
[2.3]	Title:	7 Series DSP48E1 Slice
	Company:	Xilinx
	Date:	Published March 2018
	File:	https://www.xilinx.com/
[2.4]	Title:	Life of an instruction in LLVM
	Author:	Eli Bendersky
	Date:	2012-11-28
	File:	http://blog.llvm.org/2012/11/life-of-instruction-in-llvm.html
[2.5]	Title:	Performance and Resource Utilization for Floating-point v7.1
	Company:	Xilinx
	Date:	Visited at 2021-02-05
	File:	https://www.xilinx.com/
[2.6]	Title:	TableGen BackEnds
	Company:	LLVM
	Date:	2021-02-05
	File:	https://llvm.org/docs/TableGen/BackEnds.html
[2.7]	Title:	Writing an LLVM Pass - The MachineFunctionPass class
	Company:	LLVM
	Date:	2020-10-12
	File:	https://releases.llvm.org/11.0.0/docs/WritingAnLLVMPass.html
[2.8]	Title:	Intel® 64 and IA-32 Architectures Software Developer's Manual
	Company:	Intel
	Date:	Published October 2019
	File:	https://www.felixcloutier.com/x86/cmovcc
[2.9]	Title:	LLVM Alias Analysis Infrastructure
	Company:	LLVM
	Date:	2021-02-05
	File:	https://llvm.org/docs/AliasAnalysis.html#aliasanalysis-class-overview
[2.10]	Title:	Computer Organization and Design - The Hardware / Software Interface
	Authors:	Patterson, D. A.; Hennesy, J. L.
	Date:	2011
	ISBN-13:	978-0123747501
[2.11]	Title:	MULTI: A game changer for the elevator industry
	Company:	Prodrive Technologies
	Date:	2019
	File:	https://prodrive-technologies.com/news/

[2.12]	Title:	ISO/IEC 9899:TC3
	Company:	ISO and IEC
	Date:	09-07-2007
	File:	http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

Referenced Papers

[3.1]	Title:	On Computable Numbers, with an Application to the Entscheidungsproblem
	Author:	Turing, A. M.
	Date:	Published November 2012
	File:	https://www.cs.virginia.edu/ robins/Turing_Paper_1936.pdf
[3.2]	Title:	LLVM: An Infrastructure for Multi-Stage Optimization
	Author (Company):	Lattner, Chris (University of Illinois at Urbana-Champaign)
	Date:	Published December 2002
	File:	http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf
[3.3]	Title:	The Pitfalls of Verifying Floating-Point Computations
	Author (Company):	Monniaux, David (CNRS/École normale supérieure, Paris)
	Date:	Published May 2008
	File:	https://dl.acm.org/doi/pdf/10.1145/1353445.1353446
[3.4]	Title:	Transport Triggered Architectures: Design and Evaluation
	Author (Company):	Corporaal, Hendrik (Technische Universiteit Delft)
	Date:	Published October 1995
	File:	https://repository.tudelft.nl/
[3.5]	Title:	Code generation for transport triggered architectures
	Author (Company):	Hoogerbrugge, Jan (Technische Universiteit Delft)
	Date:	Published February 1996
	File:	http://resolver.tudelft.nl/uuid:a5d923e0-1f1c-4404-b489-6de92baef5b7
[3.6]	Title:	Automatic code generation from Matlab/Simulink for critical applications
	Author (Company):	Krizan, J; Ertl, L; Bradac, M; Jasansky, M; Andreev, A (UNIS corporation)
	Date:	Published May 2004
	File:	https://ieeexplore.ieee.org/document/6901058
[3.7]	Title:	Benchmarking and optimisation of Simulink code using Real-Time Workshop and Embedded Coder for inverter and microgrid control applications.
	Author (Company):	Roscoe, A.J.; Blair, S.M.; Burt, G.M. (University ofStrathclyde)
	Date:	Published September 2009
	File:	https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5429517
[3.8]	Title:	Vector Control of PMSM Using TI's Launchpad F28069 and MATLAB Embedded Coder with Incremental Build Approach
	Author (Company):	Mehta, Hrishikesh; Apte, Aishwarya; Pawar, Swapnil; Joshi, Vrunda (PVG's College of Engineering And Technology)
	Date:	Published December 2017
	File:	https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8387393
[3.10]	Title:	Code generation for reconfigurable explicit datapath architectures with LLVM
	Author (Company):	Adriaansen, Michaël; Wijtvliet, Mark; Jordans, Roel; Waeijen, Luc; Corporaal, Henk (Eindhoven University of Technology)
	Date:	Published August 2016
	File:	http://www.es.ele.tue.nl/ rjordans/downloads/adriaansen2016dsd.pdf

[3.11]	Title:	CANAL: A Cache Timing Analysis Framework via LLVM Transformation
	Author (Company):	Sung, Chungha; Paulsen, Brandon; Wang, Chao (University of Southern California)
	Date:	Published July 2018
	File:	https://arxiv.org/pdf/1807.03329.pdf
[3.12]	Title:	Skink: Static Analysis of Programs in LLVM Intermediate Representation (Competition contribution)
	Author (Company):	Cassez, Franck; Sloane, Anthony, M; Roberts, Matthew; Pigram, Matthew; Suvanpong, Pongsak; Gonzalez de Aledo, Pablo (Macquarie University)
	Date:	Published April 2017
	File:	http://science.mq.edu.au/ fcassez/bib/papers/sv-comp-2017.pdf
[3.13]	Title:	Alive-FP: Automated Verification of Floating-point Based Peephole Optimizations in LLVM
	Author (Company):	Menendez, David; Nagarakatte, Santosh; Gupta, Aarti (Rugers University)
	Date:	Published September 2016
	File:	https://rucore.libraries.rutgers.edu/rutgers-lib/58517/PDF/1/play/
[3.14]	Title:	8-Bit softcore microprocessor with dual accumulator designed to be used in FPGA.
	Author (Company):	Sáenz Rodríguez, William; Rivera Sánchez, Fernando; Martínez Santa, Fernando (Tecnura)
	Date:	Published April 2018
	File:	http://www.scielo.org.co/pdf/tecn/v22n56/0123-921X-tecn-22-56-00040.pdf
[3.15]	Title:	Multi-Softcore Architecture on FPGA
	Author (Company):	Baklouti, M; Abid, M (University of Sfax)
	Date:	Published November 2014
	File:	http://downloads.hindawi.com/journals/ijrc/2014/979327.pdf
[3.16]	Title:	Toward understanding compiler bugs in GCC and LLVM
	Author (Company):	Sun, Chengnian; Le, Vu; Zhang, Qirun; Su, Zhendong (University of California at Davis)
	Date:	Published July 2016
	File:	https://web.cs.ucdavis.edu/ su/publications/issta16-compiler-bug-study.pdf

1. Introduction

This document has been written as a master's thesis. The thesis describes the co-development of a soft-core (a CPU implemented in configurable hardware), it's instruction set, and an LLVM-based compiler. The soft-core is based on previous work while the compiler-backend is entirely the result of this graduation project.

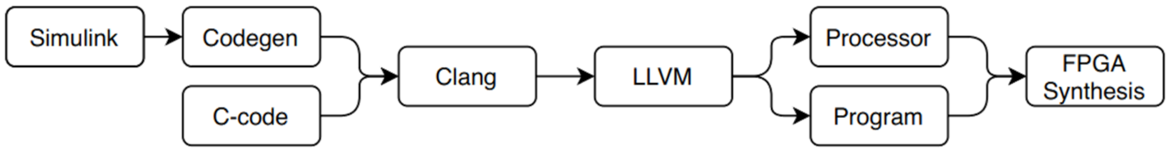

Figure 1.1: Schematic overview of the proposed workflow.

Figure 1.1 shows the toolchain as it is proposed in this thesis, the different elements are described in different sections of this thesis. This entire toolchain is focussed on working with motion control algorithms.

This introduction chapter will first give some context and information about the company. After this, an introduction is given to the current development-flow and the problem statement is presented. In the end, Section 1.3 provides a textual overview of the content of this thesis.

1.1. Context

Prodrive Technologies' projects range across a lot of different sectors and technologies. The developed products require multidisciplinary teams, using a combination of software, electronics, mechanics, and magnetics. A lot of these products require some form of control algorithm.

This project's first (internal) customer and primary stakeholder is the Prodrive Motion Platform (PMP). This motion platform is an internally developed, real-time, motion control platform in the technology program 'Motion & Mechatronics'. Motion & Mechatronics is the program responsible for the software, electronics, mechanics, magnetics and control algorithms used in motion applications. PMP is the basis for most products within this program, ranging from wafer stages with nanometer positioning accuracy to a rope-less elevator with thousands of power amplifiers distributed across kilometers of elevator shaft. [2.11]

The control algorithms required for these projects can be implemented and run on a simple microcontroller, a more complex CPU (ARM or x86), or in programmable logic (e.g. FPGA). Some algorithms have to run on an FPGA for example because the FPGA is already present in the product for other purposes, like current control or high performance peripheral interfaces. This option, implementation in an FPGA can be subdivided into two situations. One where High Level Synthesis (HLS) is applicable, and one where a manual implementation by an FPGA engineer is required.

Instead of doing such a manual implementation, some of these projects can also use the aforementioned soft-core. At Prodrive Technologies such a soft-core has been developed already specifically for floating point control algorithms. This CPU is called 'FP-CPU', Floating Point CPU, because it is only able to do floating point operations. In this thesis a new soft-core is proposed, which is named 'PD-CPU', to provide a clear distinction between the two.

1.1.1. Development Using the FP-CPU

Based on earlier work by Paul Moş [1.1], it has been concluded that for performance constrained implementations, Matlab Embedded Coder which creates C-code from Simulink models, followed by HLS tools, can be used to generate performance efficient FPGA implementations. This HLS-approach works well for integer and fixed point algorithms; however, for floating-point algorithms the generated implementation can become too large.

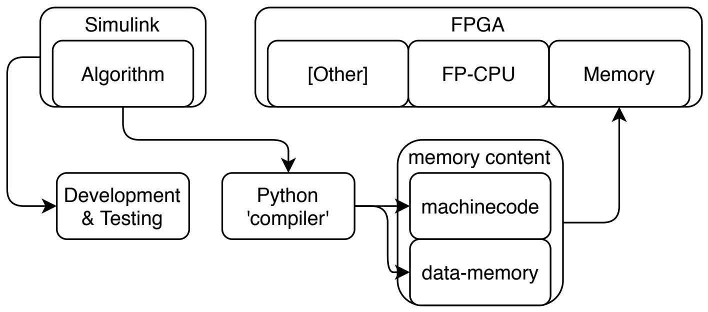

Figure 1.2: Toolchain after addition of the Python compiler script.

An existing soft-core, like the MicroBlaze, was not a viable solution for the reference projects used in the thesis of Paul Moş [1.1], because the latency did not meet the requirements. However, both the new custom CPU suggested by Paul Moş and the analysed FP-CPU performed sufficient on latency and resources. Therefore, the FP-CPU solution has been improved by developing the Python script which generates assembly for the FP-CPU when given a Matlab function formatted according to very specific rules. Figure 1.2 shows the state of the toolchain after this addition.

Adding technologies like the FP-CPU changes the way product development happens. The following paragraphs show the changes that happened because of the introduction of the FP-CPU.

Without FP-CPU In situations where HLS was not a sufficient implementation method a Control Engineer needed to wait until an FPGA Engineer had finished the manual implementation of the control algorithm. After the implementation was done the Control Engineer could then verify that the algorithm indeed performed as intended in the lab. If additional changes or tuning was required the FPGA Engineer again needed to invest some time. This dependency on another engineer is not efficient and increases the development lead time.

With FP-CPU When using the FP-CPU, an algorithm has to be implemented in a very specific way in Matlab code. Embedding this code in a Simulink block allows it to be used for testing in Simulink. The algorithm gets translated into the memory content for the FP-CPU soft-core by a Python script. This memory content, together with the implementation of the FP-CPU get synthesised with the other parts of the FPGA implementation into the final bitfile used to program the FPGA.

```
if(discriminant > zero)
  % d2Target = (-b1 + sqrt( discriminant )) / (2*a1);
  temp = two*a1;
  result = sqrt(discriminant);
  result = result - b1;
  d2Target = result / temp;
else
  d2Target = I2Control / b1;
end
```

Listing 1.1: Example Matlab-code written for the FP-CPU

The FP-CPU in its current form is already in use for several projects. The implementation is functional, however the quality of the generated code is mainly dependent on the way the algorithm is coded by the control engineer. As mentioned, the control engineer has to write this control algorithm in a very specific way in a Matlab '.m-file'. Only trivial statements with an assignment and a binary operator can be used, extended with if-else, an example is shown in Listing 1.1. This can be parsed by a Python script and converted directly into instructions for the FP-CPU.

Using either an HLS-tool or a compiler and soft-core will both solve the same problems. Quick development iterations become possible because changes to the model can be directly tested using the generated implementation. This prevents human-errors due to typo's. The mentioned iterative process between the Control Engineer and FPGA Engineer disappears because now the Control Engineer has full control over his implementation as well as his algorithm. This prevents communication errors and reduces the lead-time of the development.

Both solutions are very similar in terms of design flow. The reason why the soft-core sometimes trumps the HLS solution is that for floating-point algorithms the FPGA footprint of the soft-core is significantly smaller than the corresponding HLS solution.

1.2. Problem Statement

Paul Moş concluded in his research [1.2] that within the design space for FPGA implementations, there was a gap for floating point, low resource solutions. A floating point soft-core would fit in the gap with medium latency and low LUT(Look-up Table) usage.

The existing implementation of the FP-CPU and the corresponding Python script have already improved the workflow. Previously, the control engineer and the FPGA engineer both had to implement the same algorithm, each in their own respective set of tools. This is double the work and can lead to double the mistakes. This problem was solved by using HLS; however, HLS was not feasible for floating-point algorithms due to constraints on the FPGA footprint. When the FP-CPU is used, the implementation of the control algorithm is done by the control engineer, hence the communication issues with, and workload of the FPGA engineer disappear. Furthermore, the FPGA engineer is no longer a main dependency for the control engineer so there is more scheduling freedom within a project. However, multiple issues are still present in the current situation (described in the background section (Section 1.1.1)) and more importantly, the FP-CPU is currently only used within a couple of projects. This leads us to the first problem.

> The FP-CPU is not yet applicable to a lot of projects within the control domain.

The second problem is that the control engineer, in order to target the FP-CPU, has to write his/her algorithm in a very specific way in Matlab code. Because currently no real compiler exists to generate the machine code for this FP-CPU. The Python ‘translator’ script only supports basic operation i.e. assignment, binary operator and if/else statements. This increases the workload, it basically shifted work from the FPGA engineer towards the control engineer.

> The control engineer has more, and more complex, work because he/she is very constrained in the way he/she can implement the algorithm.

The currently available Python script only supports the constrained Matlab source. Therefore, single source, multiple target solutions (where algorithms are implemented either in Simulink or C/C++ and used for HLS, ARM (STM32), or the FP-CPU) are not feasible right now. Because no C/C++ to FP-CPU workflow exists. Keeping the option to choose where to implement the design, allows the engineer to first focus on the actual algorithm and implementation before focussing on the specifics as which target to choose. It might even be possible to quickly iterate the design space and see how each of the targets perform, to finally pick the best performing option. However, this only works when automated toolchains are available for all options.

> The lack of a proper compiler prevents the support of single source, multiple target solutions.

Having a compiler create the program also creates consistency in the program quality. Human-errors won’t be a source of bugs and performance might be even better with a lot less effort from the engineer. The HLS workflow has integrated testing when the model contains a Simulink TestBench. If testing can be integrated just as with the HLS solution, the output will be qualified without additional effort from the engineers.

These problems, figuring out a way to reduce the downsides of using the FP-CPU target, are the basis for this thesis.

1.2.1. Research Questions

From the prestudy [1.4] and literature the conclusion is, that 'novel elements' within this project are found mainly in the lacking instruction-set and therefore limited capabilities of the soft-core. The instruction set is different because of the lack of integer instructions. Often the floating point operations are an extension to an integer-based base instruction set. Other different elements are the lacking memory interface (no load/store instructions) and the lack of support for immediate values. Furthermore, stack-supporting instructions like jump-register or jump-and-link are not present in the architecture as well as dedicated registers for the stack pointer.

Because of these limitations, challenges arise in the mapping from the control algorithm onto the soft-core. The main research question corresponding with these issues is presented below.

RQ$_M$ ***How can motion-control algorithms be compiled for the, domain specific, PD-CPU instruction set architecture?***

This question limits the toolchain to motion control algorithms but allows freedom in the compiler design and the new instruction set architecture. The resulting results instruction set and compiler will be presented in the remainder of this thesis.

1.2.2. Reference Models

To do a proper performance evaluation a set of reference models has been defined. The set consists of a couple of internally developed control models, all of which are currently in use. These models are chosen to provide a decent overview and to represent the entire control domain within Prodrive Technologies. The models are specified in Table 1.1.

Table 1.1: The reference models used to evaluate the performance of the developed toolchain.

Identifier	Frequency [kHz]	Latency [ns]	Distinctive Properties
Model A	800	300	Very easy model, very low latency possible
Model C	48	12000	The perfectly 'average' model
Model D	60	16667	The first FP-CPU model
Model T	800	-	Upperbound model, probably too complex
Model X	16	-	Large model, best case

Most control models have several available implementations. Most important distinction is that some implementations use fixed point and others use floating point. The chosen reference models all have a floating point implementation which is the implementation used throughout this thesis.

1.3. Outline

This document has been written as a master's thesis, based on the assignment description provided by Prodrive Technologies, with the research question mentioned above. After these introduction a chapter, Chapter 2, on soft-cores will describe the existing FP-CPU as well as the proposed PD-CPU. After this, Chapter 3 will go in depth on the code generation process from Simulink models. In Chapter 4, first LLVM will be introduced and then the LLVM-based compiler. Chapter 5 evaluates the presented solutions on resources and performance by means of a newly developed simulator. A conclusion and advice to Prodrive Technologies is presented in Chapter 6. Finally Chapter 7 presents improvements which came up during the research and development stage of this thesis but were deferred for later investigation and or implementation.

1.4. Definitions & Abbreviations

1.4.1. Definitions

Marked text	Text needs to be changed or completed.
Marked text	Text has changed compared to the previous release.
Marked section	Section headers that are intended for review.

1.4.2. Abbreviations

ALU	Arithmetic Logic Unit
BRAM	Block RAM
CI/CD	Continuous Integration, Continuous Delivery
CLB	Configurable Logic Block
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSD	Design Specification Document
ERT	Embedded Real Time
GRT	General Real Time
FP-CPU	Floating Point CPU
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
IEEE	Institute of Electrical and Electronics Engineers
IO	Input Output
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
LLVM	Low Level Virtual Machine
LUT	Look-Up-Table
NOP	No OPeration
PD-CPU	Prodrive CPU
PMP	Prodrive Motion Platform
PSD	Pre-Study Document
RAM	Random Access Memory
SPD	SPecification Document
SSA	Static Single Assignment
VHDL	Very High speed integrated circuit - Hardware Description Language

2. PD-CPU

This chapter will introduce the PD-CPU Instruction Set Architecture as well as an implementation, the PD-CPU. However, first some general knowledge on the topic of soft-cores will be presented. The existing FP-CPU will be presented in Section 2.2. After this, the issues and improvement goals will be explained, to be followed by the proposed solution, the PD-CPU.

2.1. Softcores

As already mentioned before, a soft-core is a processor implemented in configurable hardware. Benefits are in the configurability, the flexibility, and the lower cost compared to developing an implementation in actual silicon. Other benefits are the possibility of quickly changing the algorithm or other control parameters. Because swapping memory content instead of having to re-synthesise an entire FPGA implementation and reprogramming the FPGA is quicker. This is not only useful during development, but even in-the-field upgrades of this core are possible. This reduces the inherent risks of development and future deprecation because updates and hot-fixes can be applied after deployment.

There are also downsides. Using a soft-core increases the complexity because it adds an additional layer of abstraction to the design. There is a more limited design space as everything is constrained to the capabilities of the configurable hardware. Lastly, the performance is lower compared to using dedicated silicon.

2.1.1. Existing Softcores

A lot of soft-cores have been developed, both proprietary like MicroBlaze or NIOS II and open source like the cores on opencores.org. They all have their own design goals. For the purpose of this research the goals were, reduced LUT usage, single precision floating point support and within those bounds, as performant as possible. As mentioned in the introduction (Section 1.1.1), previous work showed that the existing soft-cores did not fit the requirements for most of the tested projects.

2.1.2. FPGA

FPGA's are a convenient development target for domain specific processors. However, use is not only limited to development as FPGA's are a common compute platform in electro-mechanical products. The configurability of an FPGA allows the implementation of all kinds of soft-cores, ranging from an 8-bit microprocessor as presented by Rodríguez et al. [3.14] to a multicore system consisting of multiple NIOS II cores as presented by Baklouti and Abid [3.15].

Modern FPGA's like the Xilinx 7 series, have all kinds of build-in accelerators. Fast Fourier transforms, floating-point operations, computational instructions like multiplication, addition, subtraction, and division which are grouped in DSP slices. [2.3]. These elements can be used to implement fast operations for use in a soft-core. Furthermore, FPGA's contain embedded memory, both in their CLBs (Configurable Logic Block) as well as separate, often larger, memories called Block RAM. Both of these can be used to fulfil storage needs of the implemented design.

When a soft-core is implemented in an FPGA this also imposes some constraints on the design. The total freedom which exists in custom silicon is not there. The first constraint is that the design is bound to the in the FPGA available building blocks. The following paragraphs will introduce some other issues that arise when implementing cores in configurable hardware.

Block RAM In the case of the FP-CPU, Block RAM (BRAM) is being used to implement the register-file. The BRAM is configured in Dual Port mode. This means that a single read and write operation, on different addresses, can happen in the same cycle. Three-operand instructions like 'FADD' perform operations on two register operands and write the result of the previous operation in the same cycle. This means Dual Port BRAM is not sufficient.

To solve this, the register file is entirely duplicated. The first operand is read from the first register file and the second operand from the second register file. All writes happen to both registers, keeping the contents in sync. Using this solution, a single result value can be written back and two operands can be read in the same cycle. This way two two-port memories can serve as a three-port memory.

Another issue with the specific Xilinx Block RAM used for the FP-CPU is that updates to memory addresses don't immediately propagate. In the worst case, the new data is available 2 cycles later.

Operating Frequency Ideally all modules in an FPGA run at the same frequency, this prevents clock domain crossings. As will be shown in Section 5.1.1 the FP-CPU (and probably PD-CPU) can run at frequencies between the 50MHz and 200MHz, which also happens to be the range of the 'normal' FPGA operating frequencies. When other parts of the implementation require a different operating frequency, these clock domain crossings have to be resolved.

2.2. FP-CPU

The mentioned domain specific soft-core, the FP-CPU, is a floating-point processor designed from a resource conservative standpoint (mainly reduced LUT usage). This processor supports a small set of floating-point instructions specifically tailored to running control algorithms, including dedicated sine, cosine and square root operations. The next section will quickly introduce the floating point number format after which the FP-CPU architecture will be introduced.

2.2.1. Floating-point Numbers

Floating-point numbers are used to both express a large range of values and be very precise about others. As opposed to integer (and other fixed point formats), the numbers are not linearly spread across the range. The maximum value for a 32 bit float is a little over $3.4 \cdot 10^{38}$, but at the same time, around half of the floating-point numbers are in the interval [-1, 1]. Floating-point numbers are standardised in the IEEE-754 standard, originating from 1985. The standard not only specifies the format but also how to round or how floating-point expressions should be evaluated. The format for a 32-bit IEEE-754 float is shown in Figure 2.1.

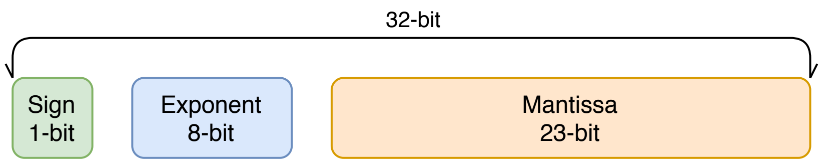

Figure 2.1: Single Precision Float (IEEE-754 format).

Floating-point computations are hard and might lead to unexpected behaviour, a lot of examples are given by David Monniaux in 'The Pitfalls of Verifying Floating-Point Computations' [3.3]; for example, how adding a *printf* can alter the outcome of your program. For a single precision floating-point number there are only 2^{32}, which is a little over 4 billion, different single precision floating numbers. This number is small enough to simply iterate all values and test a given implementation. However, as Monniaux pointed out, testing floating-point implementations is hard because even on the same IEEE-754-compliant platform a program can behave differently based on compiler settings. An implementation can break in certain cases because different register allocation caused a value to spill to memory in the middle of a set of operations, forcing an intermediate rounding. In addition to that, the IEEE Floating Point standard not only specifies numerical meaning to numbers but there are also two kinds of NaN (Not-a-Number) values and both a positive and negative infinity.

All these values need to be accounted for when doing floating point computations. For the purposes of this thesis all floating point numbers are expected to be 'normal', that is not NaN or infinity.

2.2.2. FP-CPU Architecture

As described in the introduction of this chapter, the goal of this CPU was to be as small as possible while still being able to compute control algorithms. Further limitations on the architecture were imposed by the fact that it should be implemented on an FPGA and that, in order for it to be maintainable, it should be as simple as possible. The original architecture features a 4 stage pipeline. However due to the presence of unresolved possible data dependencies this pipeline is not used and instructions are executed sequentially. A shortcut is possible from the ALU output to the ALU input using the reserved 'result' register.

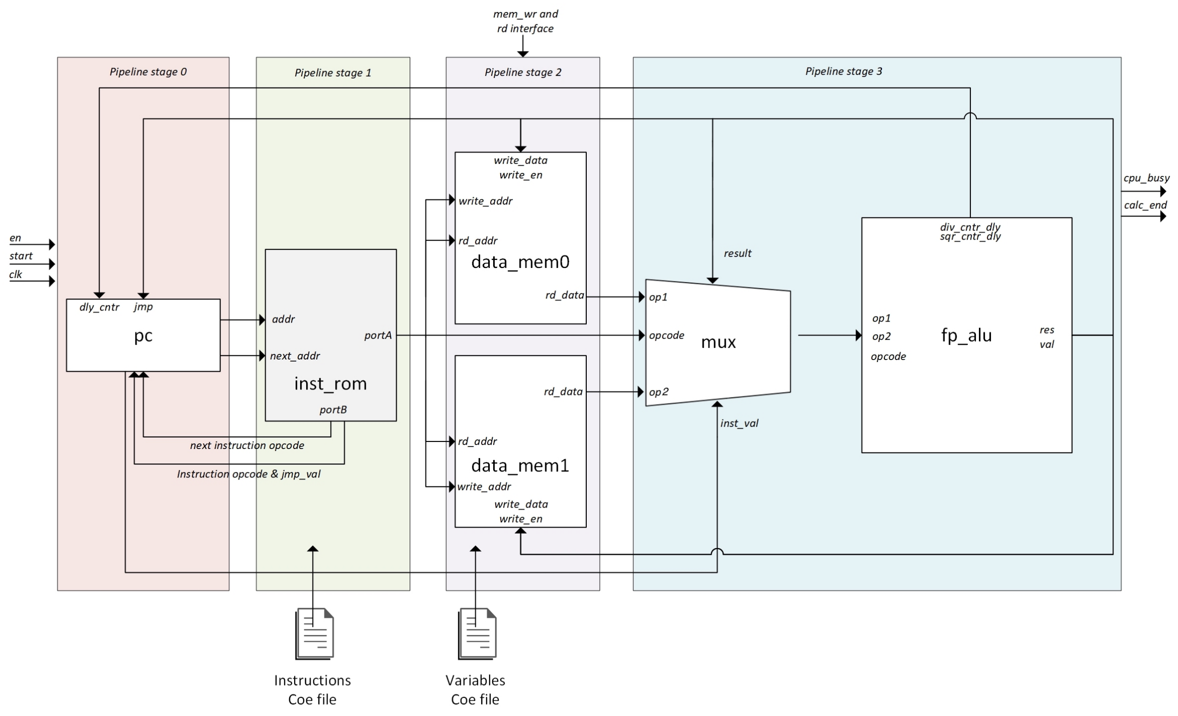

Figure 2.2: FP-CPU architecture block diagram, taken from the FP-CPU repository.

The architecture of this FP-CPU is shown in Figure 2.2. The CPU has a couple of unique features. For instance, there is no memory interface, which is why there is also no memory pipeline-stage. The two visible memory blocks are BRAM and they function like one big register file as explained in Section 2.1.2. All these registers are 32-bit floating point registers and all instructions only operate on floating point values. This is also a reason why there are no instructions with immediate support, as a floating point immediate is not common.

The processor is a combination of custom logic for the pipelining, decoding, and other control logic and Xilinx IP for the floating point operations. Most of the FPGA resources are spend on the implementation of the IP blocks used to create the ALU. The architecture of the soft-core is mostly fixed but there are some details that can be changed. For example, the latency of the Xilinx IP blocks can be modified to tweak performance, this requires only minor changes to the implementation.

2.2.3. Input / Output

A control algorithm has outputs which try to influence the state of the controlled system. Usually there are inputs as well to observe said system. The FP-CPU uses register-mapped IO (Input/Output). This means that for the control algorithm these input values simply 'appear' in a register. Actually, there is another process/module in the FPGA providing the input values to the control algorithm by placing them in register. Similarly, the outputs are read from registers as well.

Table 2.1: FP-CPU instruction format.

	31	30 27	26 18	17 9	8 0
R-type	-	opc	res	op1	op2

2.2.4. FP-CPU Instructions

The FP-CPU has 32-bit instructions. Because the latest version of the FP-CPU supports up to 512 registers, this requires 9 bit addressing. Therefore, for instructions with three operands only 5 bits remain for other purposes. This instruction format is shown in Table 2.1. The FP-CPU has 11 instructions which are shown in Table 2.2, this means another 4 bits are required for encoding the opcode. One bit is left for possible use in the future.

Table 2.2: The instructions present in the FP-CPU Instruction Set Architecture.

Opcode	Mnemonic	Operation	Functioning
0	ADD	Addition	\$res = \$op1 + \$op2
1	SUB	Subtraction	\$res = \$op1 - \$op2
2	MUL	Multiplication	\$res = \$op1 * \$op2
3	DIV	Division	\$res = \$op1 / \$op2
4	SQR	Square root	\$res = sqrt(\$op1)
5	GRT	Branch if greater than	\$pc = (\$op1 > \$op2)? pc + 1 : pc + 2
6	JMP	Jump	\$pc = \$pc + \$op1
7	CPY	Copy	\$res = \$op1
8	END	End	halt
9	SIN	Sine	\$res = sin(\$op1)
10	COS	Cosine	\$res = cos(\$op1)

2.3. Improvement Goals

The main usability-issue for the FP-CPU is the missing compiler. The currently used Python script is only able to translate Matlab code with a very strict syntax into FP-CPU assembly. Having an automated workflow from either generated or manually written C-code provides a lot of opportunities like single-source multiple-target implementations. This way the same algorithm implementation can be used to generate code for an FPGA, a microcontroller like STM32 or processors (x86 or ARM).

When creating an LLVM-based compiler it is easier if the CPU architecture is alike other by-LLVM-targeted architectures. That is why the PD-CPU has an offset on the conditional branch instruction instead of a fixed jump. Lastly, some poor choices have been made due to time constraints in past development which can now be resolved.

The final goal is to make the solution more configurable. Having the choice of doing a latency & resource tradeoff is valuable because this way the PD-CPU can be used for more projects than the fixed implementation of the FP-CPU. The new PD-CPU is presented in the following sections.

2.4. PD-CPU Instruction Set Architecture

Between a compiler and the corresponding hardware is an interface, a layer of abstraction where the underlying implementation no longer matters. A CPU-compiler interface is described in the ISA, Instruction Set Architecture. This interface describes all the possible instructions, how they interfere with each other and how memory is accessed. Because of this layer of abstraction, a compiler can be created for a certain ISA. This compiler can then be used to target all different implementations and configurations of a CPU as long as it adheres to the ISA specifications.

The PD-CPU ISA is largely based on the existing design of the FP-CPU. The goals are similar and the FP-CPU is already used so it is a proven concept. However there are some updates to the available instructions, which are described in the corresponding Specification Document (SPD) document [1.3], attached as Appendix C. The PD-CPU Architecture is explained in the upcoming Section.

2.5. PD-CPU Architecture

The PD-CPU implements the instruction set architecture as described in its SPD [1.3]. The PD-CPU is, just as the FP-CPU, a 4-stage pipelined soft-core. However, the PD-CPU resolves all data hazards by forwarding operands (explained below). Therefore, the PD-CPU is able to make proper use of the existing pipeline as opposed to the sequential operation of the FP-CPU. As follows from the instruction set [1.3] not having memory operations, external memory is not supported by the ISA and therefore not implemented by the PD-CPU. All operations operate on a big register and therefore program execution is limited by the size of that register. An overview of the architecture is depicted by Figure 2.3. Relevant differences with the FP-CPU are mentioned down below.

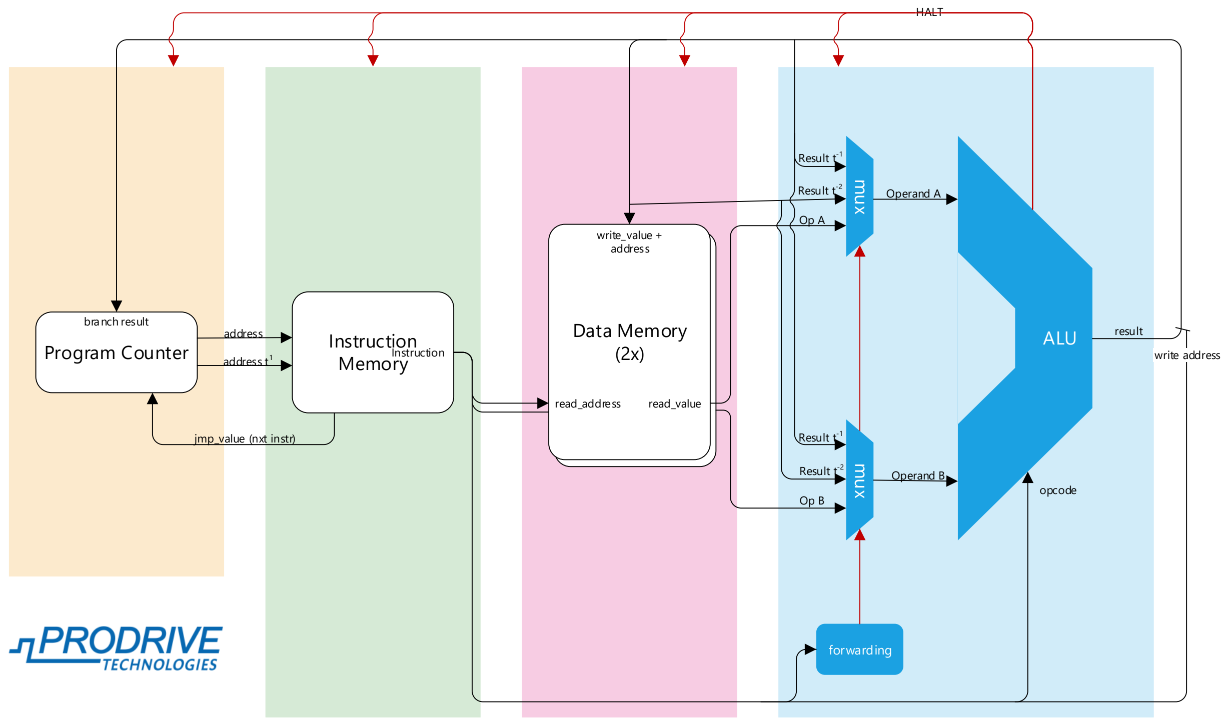

Figure 2.3: PD-CPU pipeline diagram.

Dynamic jump offset The FBGT instruction in the FP-CPU has a fixed jump offset. Either jumping to the next instruction or the one after that. Following the FBGT with a JMP instruction was possible but the FBGT also has one operand left because there is no destination operand address required. Therefore it is possible to use the remaining operand field as jump-offset, which is the case for the PD-CPU.

Signed Jump The offset in the JMP instruction in the FP-CPU is treated as an unsigned offset. This limits the possibility of jumping backwards and therefore implementing loops in a trivial way. A workaround would have been to jump past the last register address in order to 'wrap around' back to the beginning, but this complexity becomes unnecessary by simply allowing a signed offset. The PD-CPU allows a signed jump, both for the JMP instruction as well as the FBGT instruction.

Forwarding The FP-CPU could not make use of its pipeline because the possibility of data dependencies. The PD-CPU contains forwarding logic (the muxes, implicitly cached values on the 'wires' marked $Result\ t^{-1}$ and $Result\ t^{-2}$, and a controller (forwarding unit) visible in the fourth pipeline stage in Figure 2.3). The forwarding unit checks the write addresses of previous instructions and compares them with the read addresses of the current instruction to see if data should be forwarded, i.e. if the cached results should be injected instead of one (or both) of the operands. The forwarding unit caches the two previous results to also resolve the issue where the register file does not return the most recent value as explained in Section 2.1.2. This logic allows insertion of the result of the previous instruction (and the one before that) directly into the fourth pipeline stage. Inserting these previously computed values directly into the ALU resolves these data-hazards.

Single-cycle assumption Because the PD-CPU makes use of the pipeline, it is assumed that every cycle a new instruction can be started. This was not the case at the FP-CPU where had to be waited for the previous instruction to finish. If an instruction takes multiple cycles, the entire pipeline has to be halted. Once the multi-cycle instruction is finished in the ALU stage, the pipeline can 'move' again and a new instruction will be decoded. If a division takes for example 11 cycles, the pipeline will halt for 10 cycles. Because the PD-CPU halts the entire pipeline, the compiler can assume that every cycle a new instruction is fetched. This way the compiler is also not dependent on the instruction delay which varies, depending on the exact configuration of the used IP blocks. There is only one exception, which is explained in the following paragraph.

Mandatory delay after FBGT Because it is not immediately known which instruction to execute after an FBGT instruction, NOPs have to be inserted to force the PD-CPU to wait on the result being present. Three mandatory NOPs have to be inserted after every FBGT instruction to wait on the result of the comparison. These NOPs can be replaced by branch invariant instructions. Modern CPUs implement speculative execution and pipeline flushes, but that is too complex and unnecessary for this simple soft-core.

2.6. Latency & Resource Trade-offs

Different projects within the control domain also have different goals and requirements. A fixed implementation of the PD-CPU is therefore not the best solution. It is also not necessary, as long as the CPU keeps conforming to the ISA it will be able to run the compiled code. Obvious possibilities for customisation are the exclusion (pruning) of certain accelerators, like hardware sine/cosine operations if they are not required for executing a specific control algorithm. This is the topic of the next section. It is also possible to choose specific implementations of used IP-blocks. For example, the multiplier can be generated with different settings, either focussing on optimising for area, or latency. This is the topic of the last section of this chapter.

2.6.1. Partial Generation

Partial generation of the PD-CPU, e.g. generating the PD-CPU without support for some of the instructions. The PD-CPU ISA [1.3], specifies that the SIN, COS and FSQRT operations are all not necessary to be implemented. For example, leaving out the sin/cos IP saves around 600 LUTs. Worst case this is a 30% reduction in area and that might very well be relevant. This can be quite easily achieved by using the 'generate' keyword in VHDL.

The downside is that, once a PD-CPU without one of these optional features has been synthesised and programmed into the FPGA, the control algorithm cannot simple be swapped out for one which uses that instruction. In practice, optimising the control algorithm will often not drastically change the algorithm. How this is realised by the compiler will be discussed in Section 4.5

2.6.2. Optimising IP-Blocks

In some cases there are multiple IP blocks to choose from, for example, a pipelined division IP or just a regular, or a Xilinx or a custom Prodrive block. It is also possible to choose for example an IP which can do both floating point addition and subtraction instead of requiring two separate IPs. Combining these operations can save up to 400 LUTs. Most of the blocks are configurable meaning that they have an inherent capability of changing the area latency tradeoff itself by for example changing the latency setting of the IP.

Prodrive IP-Blocks Within Prodrive a lot of FPGA development is happening, including the development of custom IP blocks. The reasoning being that a dependency on third party IP might limit or impede the use of other vendors tools. The developed Prodrive IP blocks are not yet as optimised, both in speed and area, as their Xilinx equivalents, nor are they as customisable. If in the future performance of these custom IP blocks becomes sufficient they can be used instead, thereby removing the third party dependencies of this PD-CPU.

3. Control Model Mapping

This chapter describes the mapping from a control model onto the proposed toolchain. It first provides background information on Simulink and control algorithms in general. After that the focus will shift to code generation and how this can be configured for the specific usecase of this thesis.

3.1. Simulink

Simulink is an in this industry well-known toolbox for model-based design from Mathworks. Within Prodrive it is extensively used by the control department.

Simulink offers a graphical, block-based interface to design both the to-be-controlled system and the controller itself. It is able to simulate outputs based on a set of inputs (Simulink testbench). There are also lots of plugins available like the circuit simulator PLECS or an add-on by Xilinx for HLS. The designed controllers can be exported immediately into Verilog/VHDL or C/C++. These workflows have been integrated in the development processes as well as in the automated build pipelines.

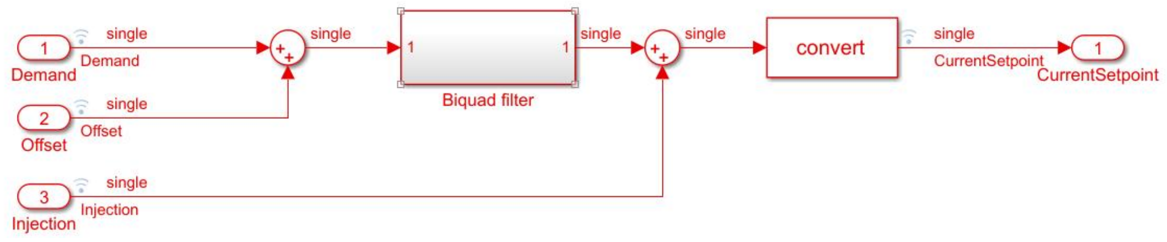

Figure 3.1: Example Simulink control model.

3.1.1. Control Algorithms

A 'Control Algorithm' is observing a series of inputs to steer the controlled system into a specific state by changing outputs of the controller. An example for this would be a motor drive which controls the terminal voltages such that the motor moves as required. The relation between inputs and outputs can be very simple and linear or any complexity necessary. A more complex but very common example would be a PID (Proportional Integral Derivative) controller.

A control algorithm is designed to operate at a certain frequency. Every cycle of this algorithm new inputs are obtained and new outputs are produced to manipulate the state of the controlled system to react as intended. The time between the sampling of the input till the output updates is called latency. There can also be requirements on this latency in addition to the operating frequency. Furthermore, the maximum latency, can be shorter than the time between two cycles as enforced by the operating frequency.

The computations of a control algorithm can be done with any data-type, however sometimes there are constraints on the numeric values which have to be expressed. If for example more precision or range than a regular integer can provide is required, fixed point can be use. The last resort is to use floating point numbers as they are more compute heavy.

3.1.2. Numerical Correctness

Floating point numbers are not only used to represent literal floating point values but there are also representations of numbers like plus and minus infinity as explained in Section 2.2.1. While these exceptional cases can technically occur in a control model, it can often be prevented by having preventive checks. This is however still an issue with the Matlab generated code. Matlab can only check for the occurrence of for example minus infinity as the result of an operation, instead of inserting preventive checks like if a divisor is going to be zero.

IEEE-754 Compliance The PD-CPU toolchain is not built to deliver IEEE-754 spec compliant floating point results. Most projects don't require the full 32 bit precision. The *-ffast-math* flag which is enabled does 'illegal' operations already but also the custom transformations in the compiler don't account for possible 'NaN' or 'Infinite' values. What would be an issue is bias, as a control loop might accumulate error if bias in a certain direction, e.g. due to constantly rounding up, is present. Until now, no indication of bias has been found but there are no hard guarantees that bias cannot occur with the current implementation.

Model validation When doing validation on the developed model there somehow needs to be a consensus on when the output is 'correct'. As already mentioned, the output won't be bitwise perfect as this toolchain is not providing IEEE-754 compliant results. The way to solve this is by defining an 'epsilon', which is proportional to the number value and defines a small range (plus and minus) in which the output is considered correct.

3.2. Model Mapping

In the Simulink block model there is a part which represents the controller and blocks representing the to-be-controlled system. This controller-block is the block which needs to be implemented. The other blocks can assist in simulating system behaviour but are not mandatory. The controller block will have a certain set of inputs and outputs. These inputs and outputs in the final implementation will need to be wired up with the actual inputs and outputs, that is, be connected to the corresponding ports in the FPGA implementation. Section 4.4 describes how these input/outputs in the generated C code are mapped to registers.

The goal is to eventually automate the mapping from the Simulink inputs and outputs to the correct FPGA register. This can be done using a configuration file as additional input to the toolchain. This file is an agreement between the control engineer and FPGA engineer on which inputs and outputs are available, how they should be named, and where they should be located. If this is realised, the IO-register mapping becomes deterministic and fixed. That allows for a quick change and recompilation of the model and final FPGA implementation without having to update IO locations in BRAM.

3.2.1. Required Model Transformations

In order to get the reference models working properly, the generated code needed to be free of all types except for single precision floating point. The most important change is therefore data-type conversion. This can be partly automatic by using the Matlab command *set_param('modelname', 'DataTypeOverride','single')*, But that is often not sufficient. Several Simulink blocks force inputs and/or outputs into a specific type. These blocks appear to not be affected by the data-type override command. These blocks have to be replaced by similar blocks which don't enforce data types. Another encountered problem was that switches are used to enable or disable part of the model. The conditions which trigger those switches have to be rewritten from boolean conditions into floating point conditions. The same holds for encountered reset signals which were boolean signals.

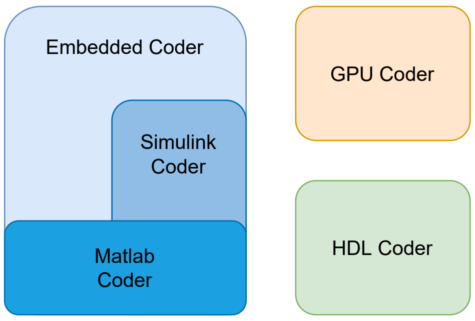

Figure 3.2: Matlab code generation tools overview.

3.2.2. Matlab Code Generation

Now the model is free of all illegal types, Matlab has to export the model into a readable format for LLVM. Matlab has several ways of generating code. There is 'GPU coder' to generate CUDA, 'HDL Coder' which generates Verilog or VHDL. There is 'Simulink Coder', for the generation of model-executable code and 'Matlab Coder' to generate code from general Matlab source. Furthermore, 'Embedded Coder' can be used to generate C or C++ to target specific or custom processors (compliant with several standards like MISRA-C, AUTOSAR, and ASAP2) from either Matlab code or Simulink models. The relation between these different tools is shown in Figure 3.2.

Embedded Coder is the tool used for targetting the soft-core. A couple of papers have been written about Embedded Coder, researching its use in critical applications [3.6], the performance [3.7], and more recently [3.8], where Embedded Coder is used to target a 'TI Launchpad F28069' development board. This is a similar use-case as described in this report, where the target is custom / non-standard hardware.

3.2.3. Configuring Embedded Coder

Before a Simulink model can become valid input of the compiler toolchain, the model needs to be changed into a usable format for use by LLVM. C-code is a functionality wise not extremely complex programming language which is well-supported by LLVM. More specifically, by Clang, the C/C++ frontend of LLVM. As mentioned above, Embedded Coder can be used for the generation of C-code from Simulink models.

This code generation has a couple of settings as it can optimise for different targets. However there are two main flavours, ERT (Embedded Real Time) and GRT (General Real Time). GRT generates a lot of boilerplate code while ERT is actually outputting very little code and the complexity of the produced code is also low. For example, code generated from a real (Simulink) control model is presented in Listing 3.1.

At Prodrive Technologies a lot of time has been spent on optimising the Matlab Embedded Coder for use with the mentioned (Vivado) HLS workflow. The Embedded Coder configuration which is used for that workflow can not directly be applied here. It for example still allows generation of the function *memset*, instead of zeroing all different parameters one by one. Other important settings are disabling support for 'non-finite numbers', and 'complex numbers' as well as optimisations which use bitshifts.

As long as Embedded Coder can be configured to prevent that certain language elements are generated (like the mentioned bitshifts), the compiler does not need to support those features. The downside is that manually written C-code will also be limited to the same language subset.

```c
// EncoderToElectricalAngle.h
typedef struct {
  int32_T Delay_DSTATE[4];              /* '<S2>/Delay' */
  uint32_T UnitDelay_DSTATE[4];         /* '<S3>/Unit Delay' */
} DW;

// EncoderToElectricalAngle.c
rtb_Sum_idx_0 =
  (rtU.PositionSensor[0] - rtDW.Delay_DSTATE[0])
  * 3L
  + rtDW.UnitDelay_DSTATE[0];
rtb_Sum_idx_1 =
  ((int64_T)(rtU.PositionSensor[1] - rtDW.Delay_DSTATE[1]) << 2)
  + rtDW.UnitDelay_DSTATE[1];
// (...)
if (rtb_Sum_idx_3 < 190L) {
  if (rtb_Sum_idx_3 < 0L) {
    rtb_Conversion_c = (uint32_T)(190L + rtb_Sum_idx_3);
  } else {
    rtb_Conversion_c = (uint32_T)rtb_Sum_idx_3;
  }
}
```

Listing 3.1: Example C-code generated by Matlab

3.2.4. Matlab & Hardware Intrinsics

Matlab will switch certain Simulink blocks with software implementations. For example, the square root is replaced if the operation is not supported by the target language. In this case, the C-programming language is the target and it does not have a way to specify a square root operation natively. One way to extend this Matlab functionality is by providing a code replacement library.

Code replacement libraries Matlab has the option to use so-called ‘code replacement libraries’. These libraries provide a way to add additional functionality to the code generator, mainly intended for target specific optimisations. It can be used to make proper use of hardware intrinsics like a custom multiply-add operation or a native square root operation. During the graduation project of Paul Moş [1.1] a custom code replacement library has been developed for the FP-CPU. This library is not up to date anymore and is therefore missing support for the later added square root and sine/cosine operations but that can be added without much effort.

This approach is currently being used in the ‘Simulink → CodeGen → HLS’ automated workflow. Because that workflow is similar, this suggests that this approach would also be feasible for the ‘Simulink → CodeGen → LLVM → FP-CPU’ workflow.

4. Compiler

This chapter introduces the LLVM-based PD-CPU compiler. As shown in Figure 4.1, the compiler is meant to both generate the assembly required for running the control algorithm provided, and also determine which configuration of the PD-CPU will be used. This results in a solution which in the future can be totally integrated and abstracted away for both the Control and FPGA engineer who will therefore have a reduced workload. According to the Problem Statement in Section 1.2 this would resolve the second main issues.

It is finally time to introduce LLVM, this happens in Section 4.1. After this introduction to LLVM, the implementation of the ISA in LLVM is explained. Followed by the 'lowering' process and an explanation of a custom *MachineFunctionPass* used to do part of the register allocation. The last sections look into the latency resource trade-off from a compiler perspective.

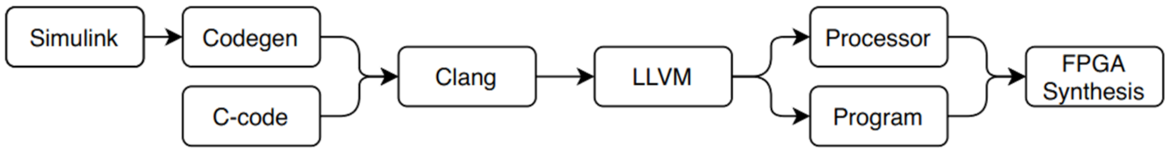

Figure 4.1: Schematic overview of the proposed workflow.

4.1. LLVM Background

LLVM, the compiler framework proposed as basis for this 'to-be-developed' toolchain, itself was once a master's thesis project, in 2002, by Chris Lattner [3.2]. The name, previously an acronym for Low Level Virtual Machine, has since lost its meaning, however the core goals are still present within the project. The first goal was to have aggressive multi-stage optimisation in the form of 'passes'. The second goal was to be a host for research and development projects, which still is the case as this project proves once again. The last goal was to have transparant behaviour towards the developer. This is debatable because of the complexity of LLVM. However, the modular architecture of llvm has a lot of features which improve transparency, like emitting all kinds of intermediate data during compilation and optimisation [3.2].

LLVM has become a mature compiler and active development is still going on, mature does not necessarily mean bug-free. A paper by Sun et al. [3.16] analysed bugs in gcc and llvm solved in the last decade and found that LLVM bugs have an average lifetime of 111 days while a GCC bug takes on average 200 days to get fixed. As mentioned, one of LLVMs goals was to be a host for compiler research in which it is succeeding. Sung, Paulsen and Wang presented CANAL [3.11], a framework that analyses cache timing by injecting additional variables and operations into the LLVM IR of a program. Skink, a static analysis tool which analyses LLVM IR to see if the error-block is not reachable, was developed by Cassez et al. [3.12]; Alive-FP, a domain specific language for expressing and verifying floating-point optimisations was presented by Menendez et al. [3.13]. All mentioned LLVM research was conducted in the past 4 years, this research not only benefits LLVM but also drives development of LLVM itself as some projects and optimisations get merged back into LLVM.

4.1.1. LLVM License

The LLVM source is distributed with the Apache License v2.0 with some LLVM specific exceptions. This means the source can be used for the purposes intended in within this project (create a 'derivative work' and 'use'). The most relevant part (license section 2) is quoted below.

> Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

LLVM was chosen as compiler framework due to the modular design, it being more modern than GCC while still being mature, and prior knowledge (both from Prodrive as due to previous experience using LLVM. As shown in Figure 4.2, the LLVM framework separates the frontend from the optimisations and the backend. Hence, the workload of creating a compiler for the PD-CPU is confined to implementing a code generation backend which can then make use of all the other LLVM tools and features.

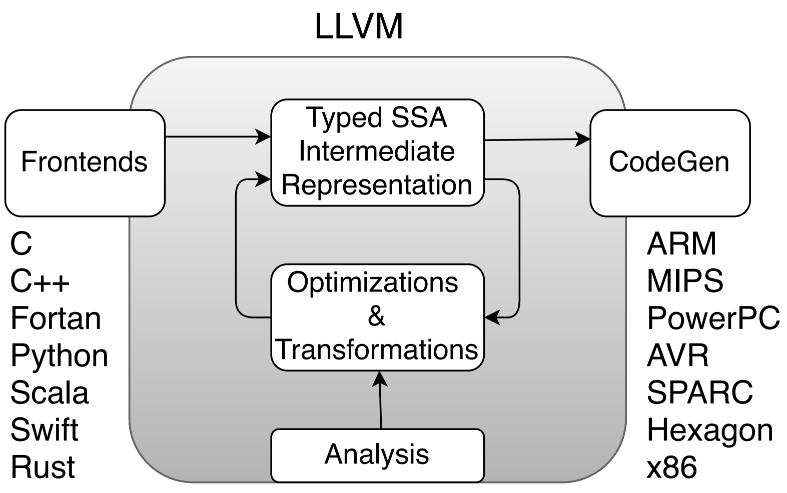

Figure 4.2: LLVM structure

4.1.2. General LLVM-backend Structure

Within the backend itself the structure is also very modular, because most functionality has been written separately, each in its own *Pass*. A pass can be either a *(Machine)FunctionPass* or a *ModulePass*, depending on if it interacts with a single function at a time or the entire module. There are a couple of other types but these are the common ones. There are passes which implement obvious optimisations like 'LICM' (Loop Invariant Code Motion), which happened to be one of the unmentioned pass-types *LoopPass*. Other passes are analysis passes, like the *MachineFunctionPass* 'Live Interval Analysis', which provides information on when values in registers are said to be 'live', i.e. valid and still required. This information, provided by analysis passes can be used by other passes. For example, Register allocation, which has been implemented several in multiple ways by MachineFunctionPasses (e.g. *RegAllocPBQP*). The fact that even core functionality like register allocation has been implemented in passes ensures that LLVM remains very flexible and extensible. This is also great for research or experimentation due to the ability to swap out passes in favour of other custom experimental ones.

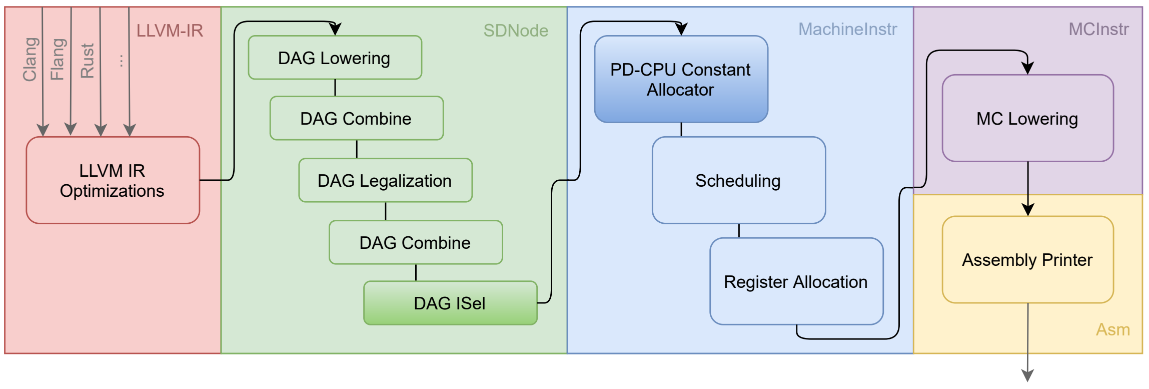

Figure 4.3: LLVM backend structure

LLVM is mostly written in C++, however a domain specific language named ‘TableGen’ is used to generate large amounts of C++ code or even entire components like an assembly printer during a pre-compilation step. This language is used to describe common elements occurring in all backends. TableGen is very efficient in for example describing instruction layout/behaviour, from this description it generates the required code for instruction selection. Other use-cases, or ‘Backends’ are documented in the LLVM documentation. [2.6]

Figure 4.3 presents a high level overview of an LLVM backend, in this case tailored to the PD-CPU as is clear by the highlighted PD-CPU specific ‘Constant Allocator Pass’ which will be further introduced in Section 4.4. Custom passes like this can be inserted everywhere due to the modular design. The other highlighted pass, Instruction Selection, is highlighted to indicate that these two highlighted passes are the locations in the backend which required the most customisation for the PD-CPU support. The LLVM-structure from Figure 4.3 is further explained in the following paragraphs.

LLVM-IR The LLVM Intermediate Representation is a low-level, type safe SSA (Static Single Assignment) representation. It is designed to represent all reasonable high-level language constructs properly as well as lowering easily and efficiently to most target platforms. LLVM-IR can be used in human-readable format or as an in-memory or on disk bitcode representation. LLVM-IR uses virtual registers i.e. an infinite number of not-physically-existing registers (as it is pre-register allocation) and is optimised by the target-independent optimisers present in LLVM. The optimisations can include some target specifics, which are obtained by running target analysis passes.

This LLVM language is at the core of LLVM. All front-ends produce LLVM IR and all backends consume LLVM IR. This way the core components of LLVM, arguably the most important one, the optimiser, can be reused across all frontend-backend combinations. The backend developed for the PD-CPU only needs to handle LLVM IR input and therefore does no longer have to worry about the C/C++ features which were present in the compiled source file. Which frontend was used, Clang, Flang, Rust, is no longer relevant at this point. Just as it also no longer matters how the control algorithm was implemented in Simulink.

SelectionDAG After the target independent code generation the LLVM-IR is lowered to the *SelectionDAG*, a Directed Acyclic Graph, representing the program structure and data/chain(control) dependencies. This graph contains generic nodes (e.g. *ConstantFP*, *CopyToReg*) as well as target specific nodes (e.g. *PDCPU::MOV*). On this DAG, a sequence of Combine/Legalise/Combine passes will be performed, where Combine is doing simplification and legalise is trying to get rid of all the illegal types and operations. For example, the PD-CPU backend only has a branch-on-greater-than instruction, meaning that all other types of comparisons will have to be rewritten using only this instruction. Legalise will try to do this for all illegal operations. Note that, for this specific case, Legalise alone is not sufficient and custom selection had to be done in order to get all the comparison’s working properly.

In the last stage, instruction selection, the SelectionDAG will be used to perform pattern matching with patterns defined in TableGen files, combined with custom selection rules defined by the target specific subclass of *TargetLowering*.

MachineInstr After instruction selection and pseudo-instruction expansion have been finished, all nodes should be representing actual machine instruction’s, the DAG *SDNode*s have been lowered to *MachineSDNode*s. Scheduling unwraps each *MachineSDNode* into corresponding *MachineInstr* and thereby linearises the program. After this, register allocation will map all the remaining virtual registers onto physical registers.

MCInstr The lowering from the *MachineInstr* to the *MCInst* used in the Machine Code Layer (MC Layer) happens in *AsmPrinter::EmitInstruction*. *MCInst* is a very simple representation which can be used to emit object files or printed as assembly. [2.4] Thereby completing the compilation process.

You may have an "oh my, why is this so complex?" reaction at this point. The TL;DR answer is "compilers are hard, let’s go fishing". (Eli Bendersky)

4.2. Implementing the PD-CPU ISA

The backends present in the current LLVM source-tree have a couple of things in common, including the integer support and a memory interface. Because these are both lacking in the PD-CPU ISA, this makes it a unique target. Both of these differences cause issues, for example, LLVM does not like being configured such that it cannot use integers of any size. Also, not having memory operations available presents issues when Clang places globals in memory. The following sections present several challenges and how they are overcome.

4.2.1. Compiling for a Reconfigurable Architecture

Having anything else than a fixed architecture poses an additional challenge. There are different levels of configurable architectures, ranging all the way from having an optional set of floating-point instructions like ARM [2.1], to being able to configure the entire datapath of a processor as is possible with a TTA (Transport Triggered Architecture) [3.4].

One of the early compilers for this TTA architecture, as presented by J. Hoogerbrugge in his doctoral thesis [3.5], uses boolean flags to indicate if a certain operation is permitted by the architecture. If not, it replaces those illegal operations by library functions, essentially 'legalising' the source by replacing calls which could be executed by hardware by algorithmic equivalents depending on if that specific hardware is available. The CGRA (Coarse Grained Reconfigurable Architecture) is closely related to the TTA and has less in common with other architectures targeted by LLVM than the FP-CPU. However, a compiler based on LLVM for such an uncommon architecture is still possible as shown by Adriaansen et al. [3.10].

4.2.2. Run-loop compilation

As mentioned before, both the FP-CPU and the PD-CPU use register mapped IO. When all inputs are available in the appropriate registers the CPU gets an external trigger to start execution of the control algorithm. This execution phase ends when the 'EOI' instruction is executed which halts the CPU. It is expected that at that moment all output register values are valid and can be used to set appropriate control parameters.

After some time, when the next sample of inputs appears, the process starts over. It is important that at this point the algorithm constants are still in the appropriate registers and that those values have not been erased or overwritten by the previous iteration. The way the compiler ensures this is by defining separate regions within the register file such that the registers which are needed the next iteration effectively appear as read-only. This is explained in depth in Section 4.3.6. The initial values of the register are set by use of a '.coe' file, which contains the initial data-memory content.

4.2.3. Function calls

Function calls often have architectural support. Those architectures have a jump instruction which stores the address of the instruction from where the jump was made (jump-and-link) in order for the function to return after execution (jump-register). Furthermore, there are ways to store and restore register contents, in order to execute a function call, often implemented using a stack.

Without the proper hardware support a solution is to inline all function calls. This can be done in a nice way by just lowering LLVM's threshold for inlining functions (by adding the parameters *-mllvm -Mulinline-threshold=100000* to Clang). This solution however does still not inline when it is not possible, like for recursive functions. Creating a custom LLVM pass or modify an existing one allows throwing errors if a call is encountered which cannot be inlined as there would then be no way to compile that code.

```
// Sometimes I believe the compiler ignores all my comments
static StringRef computeDataLayout(const Triple &TT) {
  return "e-p:32:32-f32:32-f64:32:32";
}
```

Listing 4.1: PD-CPU Datalayout string.

4.2.4. Datalayout

An LLVM backend defines a so-called ‘datalayout string’, this defines the endianness of the target-architecture as well as how many bits are used for different LLVM data-types. The string for the PD-CPU is shown in Listing 4.1 and the separate items are described in Table 4.1.

Table 4.1: PD-CPU Datalayout breakdown.

Part	Explanation
e	little endian
p:32:32	32-bit pointers, 32-bit aligned
f32:32	32-bit floating-point, 32-bit aligned
f64:32:32	32-bit 64-bit-floating-point, 32-bit aligned

4.3. Custom Lowering

‘Lowering’ is the process of moving from a higher abstraction level in LLVM to a lower abstraction level, i.e. closer to the target architecture. Information about the targets lowering preferences is accumulated in a target specific subclass of *TargetLowering*. Subclasses of *TargetLowering* have to implement a couple of lowering functions, *LowerFormalArguments*, *LowerReturn*, *LowerCall*, and finally *LowerOperations* must be implemented. The latter to handle all instructions / actions which have their lowering-action set to ‘Custom’. The subsections below provide information on the custom lowering required for this backend.

4.3.1. CopyToReg

The *ISD::CopyToReg* LLVM-IR operation is used and inserted to ensure that operations have their register operands in the proper register class or other cases where some kind of data movement is presumed to be needed. If this eventually turns out to be unnecessary, or if moves can be omitted this will happen in later compilation stages. The custom lowering done for this node is very simple, if a value is of type *f32*, and the register is not, a new *f32* general purpose register (GPR) will be allocated. The *ISD::CopyToReg* instructions will eventually be lowered to the *PDCPUISD::MOV* instructions.

4.3.2. SELECT

ISD::SELECT compares well to the ‘conditional move’ instructions present in several instruction sets, for example the x86 *CMOVEcc* [2.8]. The first issue with lowering this into a valid *PDCPUISD* instruction arises with the multiple possible variants of this instruction where the so called ‘condition code’, which determines what kind of comparison is happening, varies. All these comparisons, lesser than, equal, greater than or equal etc. have to be mapped onto the only present PD-CPU comparison instruction, the ‘greater than’. Table 4.2 shows proper transformations for the other possible comparisons.

Table 4.2: Transformations for comparison operations into ‘greater than’ format.

Operation	Original statement	Greater than conversion
gt	(a > b) ? foo() : faa()	
lt	(a < b) ? foo() : faa()	(b > a) ? foo() : faa()
gte	(a >= b) ? foo() : faa()	(b > a) ? faa() : foo()
lte	(a <= b) ? foo() : faa()	(a > b) ? faa() : foo()
eq	(a == b) ? foo() : faa()	(a > b) ? faa() : ((b > a) ? faa() : foo())
neq	(a != b) ? foo() : faa()	(a > b) ? foo() : ((b > a) ? foo() : faa())

In LLVM some of these transformations are implemented in *SelectionDAGLegalize::LegalizeSetCCCondCode*. In the case of the PD-CPU where only a ‘greater than’ comparison is present the by LLVM implemented transformations don’t suffice. Hence the described transformations in Table 4.2 are implemented in the *TargetLowering* subclass *PDCPUTargetLowering*.

LLVM makes a distinction between ordered and unordered floating point comparisons. The difference is in how NaN cases are handled (See Section 2.2.1). For the PD-CPU the assumption has been made that NaN cases don’t occur. Therefore the ordered and unordered comparisons are used interchangeably.

4.3.3. BRCOND

ISD::BRCOND is similar to *ISD::SELECT* in that a condition code requires transformation until it matches the 'greater than' instruction of the PDCPU. The difference is that *ISD::BRCOND* does not have an operand indicating the location to continue program flow if the condition is false. It is a conditional jump which means that if it is not taken, normal program flow continues. This makes the transformations harder to implement but the idea remains the same.

4.3.4. Bitcast Issue

A lot of meticulous changes where made to the input of Clang, to ensure that the only data-type required to compile would be a single precision float. It was a surprise to learn that Clang still decided to sometimes *Bitcast* a value from *float* to *i32*, to later store it with an embedded *Bitcast* back as *float*. An early fix was to custom lower these bitcasts into the *PDCPUISD::MOV* operation. This did not prevent an entire chain of incorrect types being generated. The entire conversion into a 32-bit int *i32* was pointless from a functionality perspective as well. It caused a lot of issues, for example, intermediate 'phi-nodes' had their constants converted into *i32*and this caused the floating-point only backend to crash.

The cause of this issue appeared to be an optimisation. After some investigation the optimisation pass 'instcombine' has a function named *combineLoadToOperationType* which tries to load values using the same type as in which the data will be used. For example, to prevent loading an integer and converting it to a pointer immediately afterwards. The pass tries to immediately load it as a pointer. In the case where a value is only stored without other operations happening in between, the loads are canonicalised into the 'cheapest' type which is an integer of the same size as the stored value, according to this pass. Modifying this function by providing an early exit prevents these casts into 'i32' values from being generated by Clang and therefore this no longer causes issues in the PD-CPU backend.

4.3.5. Unwanted Load-Store Optimisation

The upstream commit described in Appendix A, had similar consequences as the bitcast-issue from the previous section. The function *SelectionDAGLegalize::OptimizeFloatStore* would transform a floating point store into an integer store under certain conditions. Again, this integer would then cause all kinds of troubles, mostly crashes from the floating point only, backend. Transforming constants into *TargetConstantFP* removes this unwanted optimisation.

4.3.6. Register Info

Architecturally the BRAM acts as a register file, however looking a little broader, the BRAM is the only way to persist information. This means it also acts as a memory. The algorithm state which has to persist after a run of the control algorithm is finished. Furthermore, the register file also provides the interface for the input and output, register mapped IO.

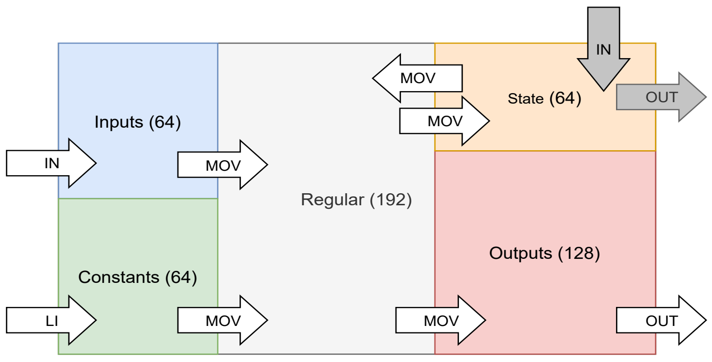

Figure 4.4: Schematic overview of the register layout.

The register file is divided in several regions for the below mentioned types/uses, and the remaining registers are used for 'normal' operation. This is shown in Figure 4.4. Variables can be moved between different registers only by the *PDCPUISD::MOV* instruction, this way it can be guaranteed that the non-regular register sections are only used for its intended purposes.

Input / **Output** For interfacing with the PD-CPU register mapped IO is used. This means that every input (and output) ends up in a register. The amount of input and output registers is based on a sample of control algorithms used at Prodrive Technologies.

Block State Initially reading from block state was handled similar as reading from an input and updating block state was handled the same as writing to an output. However it turned out that the updated block state values were sometimes used which invalidated that assumption (e.g reading from the block state would give the 'input' value and not the updated value which was written to the 'output'). Therefore a separate state-region has been defined within the register file. In this state region, both PseudoIN and PseudoOUT can map to the same value, therefore acting as a memory region with both reads and writes.

Constants The last section in the register file has been reserved for constants. There is no other way to embed the constants in the instructions (by means of immediate fields or alike). Therefore, the constant have to be placed into the registers and stay there to be used by future iterations of the algorithm.

4.4. Constant Allocator Pass

The lacking memory hierarchy poses some challenges. The model parameters, input, output and block states generated by Simulink are organised in structs. Separate structs, one for inputs, one for outputs etc. These are then stored as global variables, which Clang stores in memory. Because there is no memory or memory interface available on the PD-CPU these variables have to be allocated in specific registers. The constant allocator pass is executed before register allocation because the constant allocator pass allocates only some of the registers and the usual register allocation allocates the remaining virtual registers.

The name of the pass suggests that it is used to allocate constants, however the current use is way broader, it also allocates input, output and state registers.

```c
// Returns the value of 'angle'-degrees in radians.
float radians(float angle) {
  float pi_div_180 = 0.01745329251994329576923690768489;
  return angle * pi_div_180;
}
```

Listing 4.2: Example C-code using a floating point constant.

4.4.1. Embedding constants

In the preparation report [1.4] there has been some consideration on the implementation of immediates, which was deemed unnecessary and impractical because of the floating point data type. However; a way for the compiler to embed or generate constants is a necessity for almost all control algorithms.

Listing 4.2 shows an example of C-code which requires a constant. The function could be rewritten in several ways to change the constant but in the end a constant will always be necessary. The compiled LLVM IR version of the code is shown in Listing 4.3. The constant has now been converted into a hexadecimal representation of the 32 bit floating point value.

```llvm
; Compiled with: -O3 -ffast-math
; ModuleID = './radians.c'

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local float @radians(float %angle) local_unnamed_addr {
entry:
  %mul = fmul fast float %angle, 0x3F91DF46A0000000
  ret float %mul
}
```

Listing 4.3: (Condensed) Example LLVM IR using a floating point constant.

Usually constants are either generated by an instruction or being stored in the data-section of the program. This section ends up at a certain memory location and values will be fetched by a load operation. However, in this case the final destination for this constant is a fixed, reserved, constant register in the register file. This is being realised by first extracting the constant from the IR by creating a 'pseudo' instruction. This instruction matches and replaces the IR-level *ISD::ConstantFP* during instruction selection. This pseudo instruction is later expanded into an assembly pseudo 'li' as shown by Listing 4.4.

These 'li' instructions end up being spread out across the entire assembly file. To be able to use this as initialisation for the register file, all the 'li' instructions are hoisted into a new block, which is added at the start of the program. This hoisting is being done by a 'Machine Function Pass.' [2.7]

```
.text
.globl  radians                  # -- Begin function radians
.type   radians,@function
radians:                                 # @radians
radians$local:
# %bb.0:
    li      f1, 1.745329e-02
# %bb.1:                                 # %entry
    fmul    f10, f10, f1
    eoi
.Lfunc_end0:
# -- End function
```

Listing 4.4: Example assembly code using a floating point constant.

4.4.2. Pseudo Instructions

As stated before, Matlab generates a struct containing the input values, separate struct containing the output values, and a struct containing the model state. The structs are allocated in global scope, hence Clang inserts load and store IR instructions to access those values. These load / store instructions are lowered into custom 'PseudoIN' and 'PseudoOUT' pseudo instructions, this allows proper, custom, handling of these values. Eventually the pseudo's can be removed and the allocated register can be used as placeholder register for that IO value (or constant, or block state). The pseudo types are explained in Table 4.3.

Table 4.3: List of pseudo instruction types.

Pseudo	RegisterType	Function
PseudoIN	ireg / sreg	'Generates' an input value into a register such that it can be permanently allocated to a physical register. It replaces the load instructions in LLVM IR.
PseudoOUT	oreg / sreg	'Consumes' a register value, it replaces the store instruction in LLVM IR.
PseudoLI	creg	LI for 'Load Immediate', materialises a constant floating point value into a register of type 'Constant'.

The register type for the 'PseudoIN' and 'PseudoOUT' are either the Input/Output types or the 'Block State' registers. The only way this distinction can be made is by looking to the name of the global struct which values are loaded or stored. All inputs and outputs are gathered in blocks ending their name with '_Y' or '_U', if the name is different, it is assumed to be model state.

4.4.3. Hoisting

Because the *PseudoIN* and *PseudoLI* instructions are just placeholders for register allocation they don't need to be in-between the other instructions. Therefore they can be hoisted out of their basic block and grouped into a newly created block. This block will be inserted before the first block, go through register allocation and can be cut off before the assembly is generated. This way the constant and inputs have properly assigned physical registers and the CPU does not encounter these pseudo instructions. The process of moving instructions up is called 'hoisting'.

There is however an exception, if the PseudoIN loads a variable which represents a block or model state value instead of a model input, this instruction cannot be hoisted. This value can be modified during execution of the program by a PseudoOUT on the same variable. Therefore the location of the PseudoIN (and PseudoOUT) in the program does matter.

4.4.4. Register Allocation

After hoisting, the next step is to perform register allocation, where virtual registers are mapped onto physical registers. It is important that the virtual registers are mapped to to proper type mentioned in Table 4.3. For the input/output it holds that if it is a state variable it should be mapped onto an *sreg*, otherwise onto *ireg*/*oreg*.

It is not as trivial as mapping every occurrence of an PseudoOUT to a new register, multiple PseudoOUT instructions might actually write to the same address and therefore to the same output. Detecting these collisions is similar to alias analysis performed by LLVM [2.9]. Except, in this case we have unique identifiers as returned by *GlobalValue::getGlobalIdentifier()*. It is assumed that this is only used to differentiate elements of a couple of global struct parameters.

After the execution of this pass, normal register allocation will be performed to allocate all remaining virtual registers.

4.5. Latency & Resource Configuration

Different variants of the soft-core can be targeted by the same LLVM backend by the use of so-called ‘subtargets’. It can for example be useful to create a minimal subtarget which does not implement the FSQRT/SIN/COS operation to save resources in the FPGA when possible. In an ideal scenario these different configurations can be generated or at least configured without a lot of manual effort.

This however results in a ‘chicken or the egg’ problem, does the feature set of the PD-CPU gets defined first, and targeted by the compiler? Or does the compiler determine the feature set. Both situations have benefits. Initially the compiler would be the preferred determining factor, first compile using an imaginary maxed-out configuration of the PD-CPU and afterwards prune away unneeded operations. However, once an implementation of the PD-CPU has been generated and implemented in an FPGA design, it would be nice if the compiler could target that specific generated configuration. This way allows only updating the algorithm, the PD-CPU assembly, in the instruction registers/memory of the soft-core.

Both options have their use-cases and their implementations are not complicated, therefore both can be supported without issues. The more interesting topics with regards to the configurability are about optimisation. How to determine the latency resource trade-off. Different possibilities will be explained in the upcoming sections.

4.5.1. Predefined Set of Configurations

There are two parts to configuring the PD-CPU, on a macro scale there is the pruning of unwanted or unneeded IP blocks. On a smaller scale there is the configuration of the IP Blocks, how the operation delay and the running frequency of the IP/CPU should be configured. For the operation delay and frequency there are a lot of different options, but as benchmarked in Section 5.1, only a couple of them differ significantly. Therefore it is reasonable to take a couple of implementation’s, order them on ‘performance’ and use this set to do the latency resource trade-off. The control engineer can basically choose between a couple solutions and view the implementation cost in area upfront. This setting also has no effect on support for the generated assembly. It only influences how performant the execution of said assembly will be.

4.5.2. Dynamic Configuration

Dynamic configuration is the process of defining the exact implementation of the PD-CPU including the IP parameters at compile time and in theory use an optimisation algorithm to get to the most optimal solution for that specific control algorithm. This is impractical for multiple reasons, it tremendously increases the design space, opening up for all kinds of issues due to possible incompatible configurations and additional bugs due to the introduced complexity. The supposedly magical benefits are almost non-existent when looking at the differences in resource usage from the analysis in Section 5.1 and the difference which matters is the pruning of unused IP, which also happens in the section above.

4.5.3. Choose One, Optimise the Other

The optimal implementation, according to the interviewed control engineers would be an implementation where one of the latency and resource parameters can be constrained while the other is optimised. A possible request would be to create the lowest latency configuration with a maximum of 1200 LUTs. This can be obtained without much effort by just iterating all predefined configurations from Section 4.5.1, but also by intelligently configuring the dynamic configurations mentioned in Section 4.5.2.

In the end, the wisest choice from a business perspective is to not make this more complex than it already is by choosing for the predefined set of implementations and picking the optimal configuration in an automated way.

5. Results

The research question asks how motion-control algorithms can be compiled. After providing a solution in the previous chapters, this chapter will focus on the results. This chapter starts with an evaluation of the resource usage, thereby looking at different configurations. After that, the simulator used for the performance evaluation, is presented with the results of this performance evaluation in the last sections.

5.1. Resource Usage

The FP-CPU is very similar to the PD-CPU which was presented in Chapter 2. The differences are mostly details while the resource usage is dominated by the IP-blocks used. Therefore, analysing the current state of the art, the FP-CPU, will provide a good indication on the resource usage of the PD-CPU.

Benchmark Setup The FP-CPU is build using default IP blocks from Xilinx, the results in this section are obtained using the Xilinx Floating Point IP V7.1, of which the performance numbers are documented by Xilinx [2.5]. The synthesis target used is a member of the Zynq UlatraScale+ series, the 'xczu2cg-sfvc784-l i'. The tool-suite used is Vivado 2020.2.

5.1.1. FP-CPU Configurations

The implementation of the FP-CPU has been synthesised and implemented for several different configurations which differ in instruction latency of the several IP's and operation frequency. As the main reason for using this soft-core was to reduce the LUT usage, the first graph in Figure 5.1 is a Pareto graph of operating frequency versus LUT-usage of several configurations.

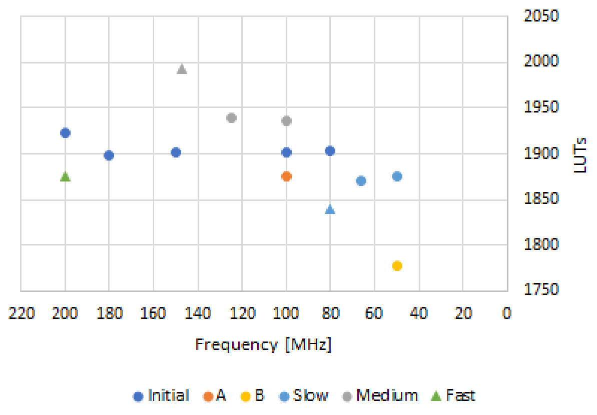

Figure 5.1: Pareto graph of several FP-CPU configurations across multiple operating frequencies versus their LUT usage. Configs are specified in Table 5.1.

The tested configurations show a nice spread across the different operating frequencies however, the amount of LUTs used does not change significantly. The difference between 1.7k and 2k LUT's is negligible. The configurations differ not only in size and operating frequency, but also in operation latency. I.e. the latency for an operation in cycles, e.g. a division takes 3 cycles in configuration 'Slow' but 8 cycles in the configuration 'Fast'. The mentioned cycles are soft-core cycles and their duration in seconds depends on the operating frequency of the FPGA. If the FPGA is running at 50MHz, a cycle takes exactly twice as much time when compared to running at 100MHz.

To get an indication on how performant a configuration is, the operation latency is averaged across all instruction types and shown in Figure 5.2. The first figure shows the average latency in cycles. 2 Cycles at 200MHz is however very different from 2 cycles at 50MHz. Therefore the second figure shows the same average latency but now expressed in nanoseconds.

Looking at Figure 5.2a, the fastest option of each of the three best configurations has been marked with a triangle. These configurations are pareto optimal when looking at Figure 5.2a. The same configurations are marked with a triangle in the other figures for clarity. Overall these configurations perform well and provide a good cover of the frequency range. These three 'reasonable' configurations are shown in more detail in Table 5.1.

Relevant gains in LUTs can be achieved by pruning unused parts of the CPU. By pruning all the non-basic operations (SIN, COS, FSQRT, FDIV), a pruned configuration might go as low as only using 497 LUTs.

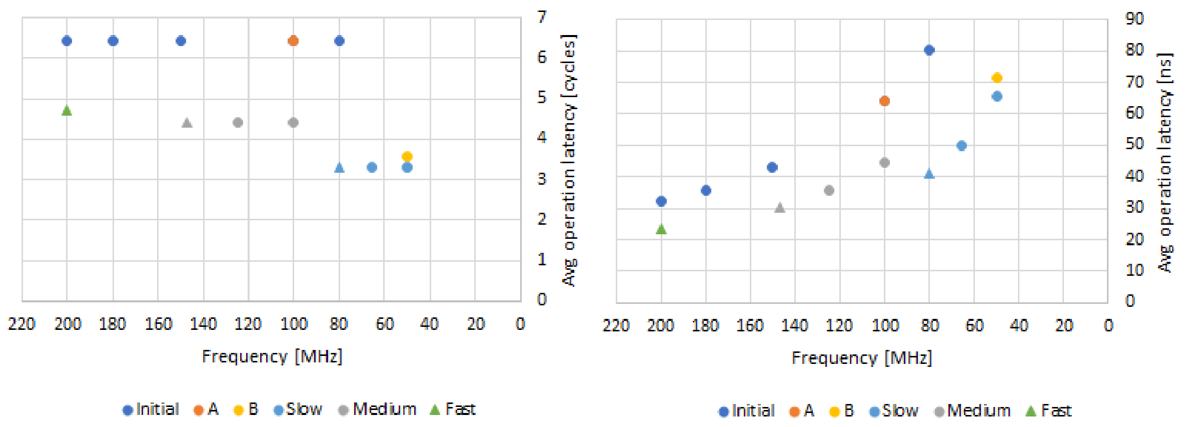

Figure 5.2: Pareto graph of several FP-CPU configurations across multiple operating frequencies versus (a) their average operation duration in cycles and (b) their average operation duration in nanoseconds. Configs are specified in Table 5.1.

5.1.2. Conclusions on Resource Analysis

The statements below are the result of analysing all data gathered. One important side-note is that the resources numbers in Table 5.1 are retrieved from an otherwise empty FPGA synthesis run. In larger designs the routing delay often overtakes the logic delay, hence the reported frequencies might have to be lowered in realistic scenarios. Overall the resource numbers are low which is great. Model-specific conclusions are presented later in this Chapter in Section 5.3.2.

IP size is not dependent on clock frequency. The IP sizes are almost only dependent on the chosen configuration, that is the IP setting like how much pipelining or parallelisation happens. This is also confirmed by the already mentioned list of IP resource usage published by Xilinx [2.5] as that is not dependent on clock frequency only on IP configuration. Small differences between the published numbers and the measured ones can be explained by optimisations which were set to 'global' during the testing of the different configurations.

Building Add & Subtract with DSP's saves 496 LUT. The tradeoff is that around 4 DSP's have to be used instead. Furthermore, this is only possible for the 'Slow' configuration as the DSP solution was too slow otherwise. In the slow configuration all gains from the addition and subtraction with DSP's are used to do a faster division and square root operation.

Removing potentially unused IP saves up to 1344 LUT. This number is taken from the 'Slow' configuration by removing sin/cos, square root and division. This leaves only 498 LUT for the total cpu implementation.

Addition & Subtraction can be merged. Using a single add/subtract IP instead of two separate ones can save up to 400 LUTs (for the faster configurations which don't use the DSP's yet). This almost halves the area used by those IP's. This modification has not been tested by synthesising as this would require modifying the CPU to fit the different interface of the new IP.

PD-CPU resource consumption is comparable. The resource numbers are dominated by the IP's. As these IP's are the same for the PD-CPU this analysis will still be useful after the changes to the FP-CPU have been finished.

Table 5.1: *Three possible FP-CPU configurations along with the original implementation.*

Operation	Cycles/Operation	Latency[Cycles]	LUTs	Registers	DSPs
Initial Configuration [100MHz]					
Add	1	2	407	135	0
Divide	5	11	243	193	0
Greater Than	1	1	35	0	0
Multiply	1	2	74	15	3
Sine Cosine	-	13	583	403	0
Square Root	9	14	129	113	0
Subtract	1	2	404	135	0
		Total	1875	994	3
Slow Configuration [50MHz - 80MHz]					
Add	1	1	196	82	2
Divide	3	3	391	73	0
Greater Than	1	1	35	0	0
Multiply	1	1	73	0	3
Sine Cosine	-	13	588	403	0
Square Root	3	3	365	74	0
Subtract	1	1	194	82	2
		Total	1842	714	7
Medium Configuration [100MHz - 147MHz]					
Add	1	1	421	33	0
Divide	5	5	278	93	0
Greater Than	1	1	35	0	0
Multiply	1	1	73	0	3
Sine Cosine	-	13	622	403	0
Square Root	9	9	144	102	0
Subtract	1	1	421	33	0
		Total	1994	664	3
Fast Configuration [200MHz]					
Add	1	2	409	135	0
Divide	8	8	194	66	0
Greater Than	1	1	35	0	0
Multiply	1	1	73	0	3
Sine Cosine	-	13	588	403	0
Square Root	6	6	168	73	0
Subtract	1	2	409	135	0
		Total	1876	812	3

5.2. Simulator

Synthesising and simulating FPGA implementations takes a long time due to the inherent complexity of FPGA design and implementation. Having a relatively light weight simulator available to quickly get metrics which otherwise would take a while to generate is very convenient. The ideal scenario would be to have a cycle and data-correct simulator, however due to the complex nature of floating point computations on different architectures such a simulator is a project an sich. For a straight forward HLS (High Level Synthesis) implementation in an FPGA it is easy to get a worst case estimation on how long it will take for valid output to appear. For arbitrary algorithms running on a soft-core that cannot easily be determined. *IF* the algorithm will halt within the required deadline, for every input, is a hard problem if the program is sufficiently complex. This problem in its general form is known as the universal halting problem and is undecidable as proven by Alan Turing in 1936 [3.1]. In restricted case such as these control algorithms, with limited input size and branching factor and most importantly without loops, it is possible to show that the algorithm has to finish within a certain amount of instructions.

5.2.1. Execution DAG

The simulator parses the assembly into a DAG. LLVM itself also uses a DAG, Directed Acyclic Graph, to represent a program. This is beneficial because a DAG has no duplicated nodes but still represents all possible execution traces.

Analysis of the DAG can provide the possible runtime of the input-program. By checking all possible paths through the DAG, taking into account the instruction delay for specific instructions, bounds can be found on the amount of cycles it will take to run specific control algorithms. The exact cycle count cannot be computed when there are input dependent branches or when there are more than 16 branches as this results in too much possible execution traces. However, it is very useful to know that an algorithm will always finish within a certain amount of cycles to guarantee finishing an iteration of the control loop in time.

The recursive algorithm shown in Listing 5.1 was created to compute the maximum possible cycle-count for any program. Using different settings for the instruction latency in the simulator allows performance simulation of the different configurations proposed in Table 5.1. Testing all configurations is a quick way to do a design space exploration.

```
func maxCycles() -> Int {
  var results = [Int]()
  for node in successors {
    results.append(node.maxCycles())
  }
  return (results.max() ?? 0) + blockDelay
}
```

Listing 5.1: Recursive algorithm for deducing the max cycle count.

5.2.2. Correctness

Due to the complexity of the compilation process it is hard to check, in an automated way, how correct the output is. The 'easy' way to do some verification on the results is to simply apply input values and check if the outputs match the expected values. The 'correct' baseline can be set by the Simulink model by using the inputs and outputs of the Simulink TestBench.

Analysing PD-CPU Assembly Listing 5.2 shows output of the PD-CPU compiler, it is cleaned up, censored and shortened but the essential pieces have not been modified except for cutting a part at line 23.

Line 8 till line 14 show the hoisted instructions (Section 4.4.3). The 'li' instructions are the assembly equivalent of the PseudoLI pseudo instruction and are only used as placeholder for register allocation. As shown the 'li' instructions all assign a constant to a 'c' (constant) register. These instructions can be removed and used as a mapping for initialisation of the data memory, e.g. in register 'c447' there should be a constant with value '1.534820e-05'. The hoisted 'in' instructions (Line 10 and 11), provide placeholders for the values in the input-struct named *SetpointRouting_U*. They are correctly mapped to an 'i' (input) register.

Lines 15 and 19 show an 'in' instruction which does not provide a value from an algorithm input but takes a value from *SetpointRouting_DW*, this struct contains the model's discrete block states. This instruction is not hoisted to the beginning of the program as the block state might be modified by 'out' instructions during the program execution as explained in Section 4.4.3.

Line 16 and 17 show that the operands of the FMUL on line 18 are moved from their respective register sections (state and constant) into the general purpose registers. This helps to guarantee that input values and constants remain valid. Future Work Section 7.2.2 is focussed on optimising these MOV instructions.

Finally lines 24 and 27, they show an ‘out’ instruction which correctly updates block state and output registers. However, where the ‘in’ instruction correctly forces a MOV with the register value into a general purpose register, the ‘out’ instruction does not force a value to be actually stored in the output (or state) register. This behaviour is not yet correct. It can be solved by implementing additional logic in LLVM for the PseudoOUT instruction expansion, or by implementing a simple rule in the assembler such that ‘out’ instructions will be replaced by MOV’s instead of being simply deleted.

```
.text                                                          1
.file    "SetpointRouting.c"                                   2
.globl   SetpointRouting_step     # -- Begin function          3
.type    SetpointRouting_step,@function                        4
                                                               5
SetpointRouting_step$local:                                    6
# %bb.0:                                  # %entry             7
  in     i383, SetpointRouting_U                               8
  li     c447, 1.534820e-05                                    9
  in     i384, SetpointRouting_U                               10
  in     i385, SetpointRouting_U                               11
  li     c448, 3.066659e-05                                    12
  li     c449, 1.989000e+00                                    13
  li     c450, -9.890000e-01                                   14
  in     s192, SetpointRouting_DW                              15
  mov    f1, s192                                              16
  mov    f2, c450                                              17
  fmul   f1, f1, f2                                            18
  in     s193, SetpointRouting_DW                              19
  mov    f2, s193                                              20
  mov    f3, c449                                              21
  fmul   f2, f2, f3                                            22
# (...)                                                        23
  out    SetpointRouting_DW, s193, f1                          24
  mov    f2, i383                                              25
  fadd   f1, f1, f2                                            26
  out    SetpointRouting_Y, o255, f1                           27
  eoi                                                          28
# -- End function                                              29
```

Listing 5.2: Example assembly generated by the PD-CPU compiler.

Conclusion As stated above, the generated assembly is not yet correct in all cases. Test cases were setup for simple comparisons and control flow. Compiling the reference models showed that produced output was not correct when manually compared with Simulink Test Bench data. The simulator however provides some guarantees, all instructions are formatted correctly, all constants and inputs have been correctly hoisted and all registers are defined before they are used.

5.3. Performance

Performance can be measured both in terms of latency and resource usage. In this section both the implementations and the compilers are compared. First of all the llvm-based compiler is compared with the previously used Python script. Then the results of compiling the reference models are analysed and lastly, several different configurations of the PD-CPU are benchmarked to get an idea on the configurability.

Remember that the resource numbers for the PD-CPU are derived from sampling the FP-CPU (Section 5.1).

5.3.1. Compiler Performance

Models ‘C’ and ‘D’, introduced in Table 1.1, are part of the reference set and have implementations available for the FP-CPU. Therefore they allow comparison between the Python script and the LLVM based compiler.

There is no simulator available for the FP-CPU. Hence, the accurate cycle count cannot be obtained easily. Because there is no code reuse and the assembly is mostly very linear in its execution, the statistics on all instructions present in the assembly file can be analysed without looking at code flow. These instruction statistics (for the entire program) are shown in Table 5.2. The table shows an overview of all instructions in the generated assembly file. Because there is no code reuse (refactoring reused code into functions) these statistics can be compared instead of looking into specific program execution traces.

Table 5.2: Per-operation-latency compiler comparison using ‘Model C’ and ‘Model D’.

Operation (Latency)	Model ‘C’	Model ‘D’	Model ‘C’	Model ‘D’
Compiler:	Python		LLVM	
MUL (2)	80	274	50	160
ADD (2)	44	74	42	96
SUB (2)	38	84	28	66
DIV (11)	77	154	44	121
FBGT (1)	30	77	22	79
SQRT (14)	0	28	0	28
MOV (1)	61	89	127	339
JMP (1)	98	234	16	52
SIN (13)	13	0	13	0
COS (13)	13	0	13	0
EOI (1)	1	1	2	2
TOTAL	455	1015	357	943

It is interesting to look at the difference between the Python script and the LLVM-based compiler. The Python script uses significantly more JMP instructions while the LLVM-based compiler uses a lot of MOV instructions. LLVM uses fewer jumps as it is able to analyse and properly schedule the code while the Python script just drop-in replaces lines by the proper assembly instructions. The larger amount of MOV instructions by LLVM can be explained because of the separate memory regions. Only the MOV instruction is allowed to take variables and place them in another region. This number can be improved fairly easily as will be explained in Future Work Section 7.2.2.

Note that ‘FBGT’ was counted as taking only 1 cycle, to not let this instructions complexity mess up this comparison by using the actual 3 cycles (including delay slots). This would favour the LLVM based compiler due to the lower number of FBGT instructions. The entire evaluation ignores the pipelining which happens with the PD-CPU but not with the FP-CPU as this is meant to evaluate performance of the compiler instead of the architecture.

5.3.2. Model Performance

The statistics of all compiled reference models are presented in Table 5.3. All the models have a floating point datapath. The exact implementations for both HLS and the PD-CPU are mostly the same with small changes here and there to get the models working properly. The PD-CPU configuration is the ‘Initial’ configuration as specified in Section 5.1. It might be that other configurations work better but those other configurations are all hypothetical but the ‘Initial’ configuration has been used in projects already and is therefore a more realistic configuration for benchmarking.

Table 5.3: *HLS versus PD-CPU performance, using the 'Initial' config of the PD-CPU and for all reference models.*

Model	Freq	Latency	HLS Freq	Lat	LUT	DSP	CPU Freq	Lat	LUT	DSP
Model A	800 kHz	300 ns	2673 kHz	374 ns	1654	8	3030 kHz	320 ns	1163	3
Model C	48 kHz	12 000 ns	820 kHz	1220 ns	6167	13	265 kHz	3770 ns	1745	3
Model D	60 kHz	16 667 ns	303 kHz	3301 ns	6381	5	106 kHz	9460 ns	1292	3
Model T	800 kHz	-	2179 kHz	459 ns	1859	5	571 kHz	1740 ns	1163	3
Model X	16 kHz	-	190 kHz	5255 ns	12466	38	219 kHz	4550 ns	1746	3

When comparing with the HLS solution, the PD-CPU solution has in general a longer latency. The latency is somewhere between 0.86 and 3.8 times the HLS latency. The resources (LUTs) are between 0.14 and 0.70 times the HLS LUT usage. This means that the goal of providing a solution with low resource usage and medium latency has been achieved. For Model A and Model X, both resources and latency are lower when using the CPU. This is especially significant for Model X as this is one of the larger models and the change in resource consumption is very significant.

When looking at the requirements, the PD-CPU is a sufficiently performant implementation and within the frequency and latency requirements. Except reference Model A, even though the PD-CPU performs better than the HLS solution, they both don't meet the latency requirements.

5.3.3. Configurability

To look at the configurability the reference model 'Model C' is used, this model is in the reference set because it is a very average model and one of the prime candidates for future use with the PD-CPU. This specific model has a minimum frequency requirement of 20KHz, however the preferred operating frequency is 48KHz, a limit imposed by the other hardware. The latency requirements (maximum time between input and output, when accounting for ADC and PWM generation, is 12µs when running at 48KHz.

Table 5.4 has been created by compiling Model C using the PD-CPU compiler and simulating the generated assembly in the simulator from Section 5.2. All configurations of the PD-CPU are able to fit within those requirements. That is no surprise as the model is currently using the FP-CPU. On a performance level this should compare very well with the PD-CPU 'Initial' configuration. The table shows for each model the maximum amount of cycles required for running an iteration of the control algorithm as well as the maximum frequency and latency which can be obtained using said configuration.

Table 5.4: *'Model C' performance for several FPGA implementations.*

Implementation	Clockspeed [MHz]	Max [cycles]	Max Freq. [kHz]	Latency [µs]	DSPs	LUTs (pruned)
PD-CPU Initial	100	394	253	3.93	3	1746
PD-CPU Slow	50	292	171	5.82	7	1477
PD-CPU Slow	80	292	267	3.74	7	1477
PD-CPU Medium	100	300	333	2.99	3	1850
PD-CPU Medium	147	300	490	2.03	3	1850
PD-CPU Fast	200	343	583	1.71	3	1708
HLS			820	1.22	13	6167

Note that the cpu will need to have both a fixed point to floating point and a floating point to fixed point converter to properly convert all input and output signals. The FP-CPU implementation of this wrapper used one of each to sequentially convert all input and output signals. This adds between 100 and 300 LUTs depending on the precision of the fixed point values (between 10 and 32 bits). The delay is one cycle for each input and output conversion, for 'Model C' this means an additional 34 cycles and around 200 LUTs.

6. Conclusion

The goal of this thesis was to compile motion control algorithms for a proprietary soft-core. The soft-core has a limited instruction set with only floating point instructions. This was expressed in the research question presented in the Introduction Section 1.2.1 and is repeated below.

How can motion-control algorithms be compiled for the, domain specific, PD-CPU instruction set architecture?

This thesis shows a lot of different angles on this research question and shows that, having the freedom to make smart choices across the entire domain of the custom architecture, development process, and compiler can save development effort in all those areas. E.g. by slightly modifying the Simulink model/code generation or the CPU architecture it becomes easier to build the compiler.

In the end this thesis resulted in a soft-core, the PD-CPU which improved the previously used FP-CPU. It specified an instruction set architecture for the soft-core, the PD-CPU ISA. An LLVM-based compiler backend was developed to target this architecture and a simulator was build to evaluate performance of the generated assembly.

The paragraphs below reflect quickly on some of the important topics of this thesis. After this two sections provide comments specifically related to the status of the project at Prodrive Technologies.

Compiling Control Algorithms The defined evaluation metric, a set of reference (control) models presented in Section 1.2.2 were all compiled by the new toolchain. It became clear that supporting a lot of the 'complexer' LLVM-IR features is not needed when only looking into the very specific subset that is required for compiling a floating point control algorithm. Even trivial things like dynamic array indexing are not necessary. What remains are basic floating point math operations and some trivial conditional control statements. This is exactly what is currently supported, almost anything complexer is unable to be executed by the hardware anyway.

Latency & Resource Tradeoff Within the boundaries of the PD-CPU ISA there is a lot of flexibility to choose different resource latency configurations. The LUT usage can be varied mostly by not implementing certain operations. The latency can be optimised depending on the operating frequency of the FPGA.

Reflection on Problem Statement The PD-CPU solution is able to target most of the control algorithms. When this solution would be polished and integrated, it would be a hands-off solution for the Control Engineer. Having a single source, multiple target solution is not completely realised. In theory C-code can be compiled using the PD-CPU based compiler, in practice however, the subset of C supported is so small that the C-code would have to be heavily modified or rewritten.

Performance Latency of the PD-CPU solution is either comparable or slower than an HLS solution but still within the model specific latency requirements. The FPGA resource usage however was in all tested cases an improvement, up to a 7x improvement in LUT usage. Only very strict latency requirements posed an issue as shown by the statistics on reference model 'Model A'.

6.1. Prodrive Applicability

The implementation analysis in Secion 5.3.2 clearly shows that the custom soft-core, the PD-CPU, has its place in implementations which don't require a lot of performance but have strong resource constraints. Paul Moş [1.2] showed that other soft-cores like MicroBlaze perform worse on latency and comparable on LUT usage.

The compiler analysis in Section 5.3.1 shows that the LLVM-based compiler outperforms the Python script from the previous implementation. Both can be improved but where the LLVM based compiler has a lot of room for improvements it also is the most unreliable solution. There is a huge discrepancy in size of the code base when comparing the LLVM-backend with the Python script.

Section 5.3.2 shows that the reference models tested work well with the PD-CPU. Especially the models with a large HLS size (Model C, D and X) show large improvements, between 3x and 7x reduction in LUT usage. The latency of those larger models is also within bound the required latency. This shows that PD-CPU is a good solution for the intended use-case.

6.2. Advice to Prodrive Technologies

Concluding this research is an advice to Prodrive Technologies, on how to progress with this obtained knowledge.

1. **Performance** The performance of the LLVM-based compiler with the PD-CPU is great as concluded above. The performance is in relevant cases better than the HLS solution. However, the FP-CPU and Python script solution is also good, and with some slight modifications it is very comparable to the LLVM-based solution (between 7% and 27% slower according to Section 5.3.1).
2. **Development Flow** The PD-CPU toolchain is currently not finished and there are still unsolved issues. For example, the integration with Simulink, the development process, and the build environments but also the stability and reliability of the compiler. Finally the PD-CPU needs to be build, tested and integrated. The FP-CPU solution however is already in use and integrated to a point where it is functional.
3. **Toolchain Development** The Python script used for the FP-CPU was modified by a ‘random’ control engineer (when this thesis was under construction), to implement support for for-loops. A similar situation would not occur with the PD-CPU compiler due to the inherent complexity of an LLVM-backend. In a business environment it is important to create maintainable and extendible tools. For this, low complexity and small code-bases are important measures.

Conclusion A soft-core solution fills a gap which, at the moment, cannot be solved by HLS solutions. Both the FP-CPU solution and the PD-CPU solution are able to solve this need. The performance of the ‘advanced’ LLVM-based PD-CPU setup seems to be slightly better. This however does not outweigh the massive effort required to get the compiler up-and-running, bug free, and well integrated into a, for a control engineer usable single button solution.

Please make it LEGENDARY rather than LEGACY. (Serkan Oktem)

The advice is therefore to continue with the FP-CPU, with a couple of improvements taken from this graduation research. First of all, implement the forwarding logic of the PD-CPU. This increases the performance of the FP-CPU to be almost as good as the PD-CPU. Furthermore, the PD-CPU simulator can be modified to provide support for FP-CPU assembly and can be integrated in the existing CI/CD workflow for the FP-CPU solutions. This can provide quick statistics on the runtime of the existing algorithm to provide statistics on the algorithms runtime and performance. Optionally, this FP-CPU simulator can also be used to validate the algorithm implementation using the Simulink TestBench.

7. Future Work

This section explains some improvements that came up during the research. These improvements each have their own reason for not being implemented which mostly came down to having lower priorities.

7.1. Latency Optimisation

The response time of a control model is a big determining factor in how well it performs. Reducing the latency, the time before the control algorithm finishes is therefore important. This can be achieved by speeding up the algorithm and/or the hardware, reducing algorithm complexity. Another solution is to make sure that outputs are computed as soon as possible. In this case the algorithm can defer new-state computations until after all outputs are computed. This reduces the effective latency.

Matlab has a flag which enables this behaviour, however this is not properly communicated towards the compiler as of now. In the compiler this would come down to hoisting the output instructions which don't output model-state as much as possible.

7.2. Instructions

A couple of improvements can done in relation to the instruction set or how instructions are implemented. These are mentioned in the sections below.

7.2.1. Unsigned addition using floating point hardware

One of the more relevant missing functionalities is dynamic array indexing. Because this forces the compiler to add pointer arithmetic into the source code. Pointers are integers and in the case of the PD-CPU these addresses need to be encoded using at least 9 bits.

A single precision IEEE-754 floating point number has a Mantissa of 23 bit's as shown by Figure2.1. The algorithm for floating point addition is shown below [2.10].

1. Align the decimal point (match exponent).
2. Add the significants.
3. Normalise the sum by shifting left or right while modifying the exponent accordingly.
4. Check overflow / do rounding.

This means that with the proper setup, the 23 mantissa bits can be used to pack a 16 bit integer. As specified by the C-standard (5.2.4.2.1 Sizes of integer types from the C standard [2.12])), the minimum size of an integer is 16 bits. Choosing 16 bit integer as an allowed type in LLVM would then allow the computation of for example, addresses for dynamic array indexing. This would only require the addition of some control logic in the implementation of the PD-CPU but no additional integer adder. This does not mean that all operations are supported for this 16-bit integer type. The only exposed instructions would be an 'add' which would take 2 floating point registers, produce one and handle all of them like they are 16 bit integer registers without the proper overflow behaviour.

7.2.2. Optimize MOV

MOV is the most used instruction by the LLVM backend even though the MOV instruction does no real useful work itself. As the MOV instruction is now dominating performance with around 1/3 of the time spend on moving values, this can lead to 15-20 percent performance improvement. Currently the main issue is that MOV is used on all inputs, constants and outputs. Allowing instructions to read from inputs and constants and write to outputs would reduce the amount of MOV's required significantly. Section 5.2.2 shows an example assembly file in which MOV instructions (on Lines 16 and 17) for example can be optimised away. An easy way to do this is by creating additional register subclasses and basically separating reads and writes into different classes. The normal general purpose registers can simply be in both the read and the write subclass.

7.2.3. Paralellism

Control algorithms can contain repetitive computations, they can also have to perform the same instructions multiple times for multiple in and outputs; for example, a motor drive which has to support 3 motors. In such cases there is a benefit to certain level of parallelism, where some instructions can happen in parallel as there are no data-dependencies between those separate runs of the algorithm.

Parallelism can be obtained by adding vector operations, but this requires large modifications of a large part of the toolchain Similar results can also be achieved by allowing another instruction to enter a different ALU IP block and for example adding a second divider IP. However, it should still be ensured that instructions finish in order.

7.3. Dynamic Input, Output, & Constant-Register Sections

The register sections as depicted in Figure 4.4 currently have their size hardcoded based on analysis of the reference control algorithms. There are obviously cases which don't fit these predefined sizes, therefore moving to a dynamic approach would be a good improvement.

7.4. CI/CD Integration

The ideal scenario from a Control Engineering perspective is that they commit their (Simulink) model into the source control solution and that from there the entire process is automated. This automated process includes the design space exploration, which can be achieved by kicking of several automated builds with different configurations, both HLS and for the PD-CPU. All of these runs could provide statistics on used resources, performance, correctness using data from the included Simulink Test Bench. Finally this allows the control engineer to make an educated choice on the actual implementation and configuration.

7.5. Simulink Test Bench Integration

It is hard to guarantee compiler correctness, the easiest way of testing is just compiling a control model, feeding it data and see if the output is correct. This is partly possible due to the development of the PD-CPU simulator which can test the program with specific input data very quickly, however it is still very cumbersome to have to create and verify input and output data.

The solution to this is to use the Simulink TestBench, this dataset is used in Simulink to verify that the model is performing as it should. Exporting this dataset, feeding it to the PD-CPU Simulator and verifying that the output is correct would be valuable, both during actual use of this PD-CPU as well as during the development of the compiler itself. The final goal is to implement this in the above mentioned CI/CD solution.

7.6. Improved Error Messages

Currently, the provided error messages are really bare bone. It is actually just a compiler crashing which often provides a, for a control engineer, vague message and a stack-trace. This is not sufficient for use in an environment where a control engineer is trying to develop a control algorithm instead of debugging a compiler. The bare minimum is that those messages are catched by the toolchain and replaced by a 'The compiler crashed, I am truly sorry' message.

Going one step further is to aggregate errors into common problems. For example, the floating point only compiler will crash when encountering a boolean or integer element in the source code. These errors could be catched, grouped and replaced by a useful message and if possible even a location in the source-code.

Appendix A. Upstream LLVM Commit

All of the LLVM-related development happened on an out-of-tree branch of LLVM. The development of the proprietary PD-CPU backend resulted however in some small changes to the LLVM core, of which one was a candidate to be pushed upstream. That specific change is documented in the following sections.

A.1. Context

The LLVM-IR has a lot of native opcodes to specify all kinds of DAG nodes, DAG nodes are used to represent the IR in a Directed Acyclic Graph containing all control and data dependencies as well as the program logic. The relevant nodes are *ISD::ConstantFP* and *ISD::TargetConstantFP*. The *target** opcodes are used in cases where further optimistion, folding or lowering is unwanted, to indicate that the target can, or should, handle this construct itself.

```
 1  SDValue SelectionDAGLegalize::OptimizeFloatStore(StoreSDNode* ST) {
 2    (...)
 3    // Turn 'store float 1.0, Ptr' -> 'store int 0x12345678, Ptr'
 4    // FIXME: We shouldn't do this for TargetConstantFP's.
 5    (...)
 6    if (ConstantFPSDNode *CFP =
 7      dyn_cast<ConstantFPSDNode>(ST->getValue())) {
 8      (...)
 9    }
10    return SValue(nullptr, 0);
11  }
```

Listing A.1: Relevant section of LegalizeDAG.cpp exposing the problem.

A.2. Problem & Solution

The problem occured when trying to prevent a store-optimisation which transformed a floating point constant store into an integer constant store as this was deemed a 'cheaper' operation (*SelectionDAGLegalize::OptimizeFloatStore*). This integer load and store are not possible on the PD-CPU. A fix in the backend to just convert a loaded Constant back into a ConstantFP resulted in an infinite loop where both transformations would happen alternately. The obvious solution was to change the transformation from *Constant* to *TargetConstantFP* to prevent further transformation. However this did not prevent the infinite loop which triggered an investigation into the LLVM code base. The relevant piece of code is shown in Listing A.1. The 'FIXME' on line 7 clearly indicates that it wasn't the intention to optimize *TargetConstantFP* nodes.

Fixing this 'FIXME' by adding the early return in Listing A.2 solved the infinite transformation loop by allowing *TargetConstantFP*'s to stay *TargetConstantFP*-nodes.

```
1  SDValue SelectionDAGLegalize::OptimizeFloatStore(StoreSDNode* ST) {
2    (...)
3    // Don't optimise TargetConstantFP
4    if (Value.getOpcode() == ISD::TargetConstantFP)
5      return SDValue();
6    (...)
7  }
```

Listing A.2: Relevant section of LegalizeDAG.cpp exposing the problem.

A.3. Commit

This change was committed and pushed upstream, both from an interest in the release process-point and for the sake of being a decent open-source user. The patch and review can be viewed at https://reviews.llvm.org/D93219.

Appendix B. Building LLVM

Prodrive Technologies makes extensive use of CI/CD workflows. At the moment most of the builds are running in Atlassian Bamboo while the (git) repositories themselves are stored in Atlassian Bitbucket. As this is the Prodrive Technologies standard, this is also chosen as solution for the LLVM fork. LLVM uses CMake, CMake is a cross-platform tool which generates native build files for Make, Visual Studio, Ninja etc. In this case we use CMake to generate Makefiles for our Linux build agents. The script used is shown in Listing B.1.

```
cmake ../llvm \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="RISCV;PDCPU" \
  -DBUILD_SHARED_LIBS=True \
  -DLLVM_ENABLE_BINDINGS=False \
  -DLLVM_BUILD_TESTS=True \
  -DCMAKE_BUILD_TYPE="Release"
```

Listing B.1: CI script to generate llvm Makefiles.

After the generation of the buildfiles the project has to be build, as CMake correctly passes the ‘parallel-jobs’ ‘-j’ option to Make, we make maximum use of the amount of virtual cores by querying ’/proc/cpuinfo’. The build command is shown in Listing B.2.

```
cmake --build . -j $(grep -c ^processor /proc/cpuinfo)
```

Listing B.2: CI build command.

After a successful generate and build the result will be tested. Running the LLVM test suite will be preceded by some PD-CPU specific tests, first to determine if the target has been registered properly as a supported target, followed by running only the PD-CPU specific tests. After this the LLVM test suite will be run. See the script in Listing B.3.

```
if ! ./bin/llc --version  | grep -q "pdcpu32" ; then
  echo "Error: PDCPU32 target not registered properly." 1>&2
  exit 1
else
  echo "PDCPU32 target has been correctly registered."
fi

if ! ./bin/llvm-lit -s -i -v test/MC/PDCPU ; then
  echo "[ERROR]: Failed to run PDCPU-MC tests." >&2
  exit 1
else
  echo "Correctly ran PDCPU-MC tests."
fi

# Similar for `test/CodeGen/PDCPU' & `test/Object/PDCPU'

make check-llvm-unit
make check-llvm
```

Listing B.3: CI script used for testing.

For each commit the above mentioned procedure of generate, build and test is run. Every night a nightly build also runs all these steps. For local development a bash script with similar functionality exists (it however does a debug instead of release build).

Appendix C. PD-CPU ISA Document

Attached below.

Specification Document

# of PD-CPU Instruction Set Architecture

Prodrive Technologies

Reference:	SPD6001203473R03
Reference date:	2021-03-15
Author(s):	Simon de Vegt
Status:	Accepted

Public

Template PN: 6001-1249-83XX
Template date: 2021-03-15

Document history

Rel.	Date	Changes
R01	2020-05-26	Initial release
R02	2021-01-22	Updated ISA to match the compiler implementation.
R03	2021-03-15	Change document confidentiality from Confidential to Public.

Name	R01	R02	R03
TUe			
Kees Goossens	x	x	
Kanishkan Vadivel	x	x	
Prodrive Technologies			
Jasper Kuijsten	x	x	
Serkan Oktem	x		
Simon de Vegt	A	A	A

Contents

List of Figures

List of Tables

List of Listings

References

[1.1]	Title:	EDD of MP4812 hardware
	Author (Company):	(Prodrive Technologies)
	ID, Version, Date:	EDD6982170384, R14, 2020-03-09
	File:	SPD6982170384R12
[1.2]	Title:	ip_fp_cpu
	Author:	Serkan Oktem
	Organization:	Prodrive Technologies
	Visited on:	2020-05-26
	URL:	https://bitbucket.prodrive.nl/users/serokt/repos/ip_fp_cpu/

1. Introduction

This document describes the instruction set architecture of a softcore, codeveloped with an LLVM backend, developed for use within Prodrive. The document is written as an interface between possible HDL implementations of the softcore and the developed LLVM backend.

1.1. Background and context

The PD-CPU, also referred to as PD-CPU32, PDCPU, PDCPU32, is a more general version of the FP-CPU, which in turn is a further developed DCDC-CPU.

This initial CPU, the DCDC-CPU, is developed for a specific project, the documentation can be found in the product EDD [1.1]. There were not sufficient FPGA resources available to use HLS, which is why a low-resource softcore was developed. This cpu was the basis for a more general version, the Floating Point CPU (FP-CPU); the name being derived from the floating point support (and with that, the lack of integer-support), which was already the case for the dcdc-cpu but not yet reflected in the name. The FP-CPU added support for a couple of more advanced operations, like a square root, sine, and cosine instruction. The documentation for the FP-CPU can be found in the repository [1.2].

To take this concept even further the FP-CPU is generalised into the Prodrive CPU (PD-CPU), a softcore, still with a floating-point base instruction set. However, this renaming opens the door for future additions of integer instruction set extensions. Furthermore, the PD-CPU is being codeveloped with a custom LLVM-backend. Having a compiler instead of scripts generating the code allows shifting some complexity away from the CPU-implementation towards the compiler as well as performance- and quality-of-life improvements.

1.2. Definitions and Abbreviations

1.2.1. Definitions

Marked text	Text needs to be changed or completed.
Marked text	Text has changed compared to the previous release.
Marked section	Section headers that are intended for review.

1.2.2. Abbreviations

CPU	Central Processing Unit
HLS	High Level Synthesis
ISA	Instruction Set Architecture

2. Platform Introduction

2.1. Memory

As can be observed in Chapter 3, the ISA contains no load and store operations. All instructions act on a single address space, addressed by 9 bits. This means a total of 512 ‘registers’ are available with single cycle access.

All elements in the data-memory are considered to be 32 bit floating point numbers and everything is stored in a little-endian format. Memory and register naming is specified in Table 2.1.

Table 2.1: PD-CPU register definition and naming.

Address	Mnemonic	Datatype
0x000	$res	fp32
0x001	$f1	fp32
0x002	$f2	fp32
⋮	⋮	⋮
0x1FF	$f512	fp32

Note: There is no dedicated return-address register, nor a stack-pointer register. If required, any register can be used as stack-pointer register. A return-address register cannot be impromptu created because the lack of a jump-register instruction.

3. PD-CPU32 Instruction Set

3.1. Instruction Format

The PD-CPU only has a single instruction-format, the instruction format is shown in Figure 3.1. The instruction is defined by a 4-bit *opc* (operation-code) field, followed by 3 9-bit operands. The first field, *res*, indicates the register where the result of the operation will be stored, the last two fields, *op1* and *op2*, indicate the input registers, except for the JMP instruction, where only the result operand is used as an offset. In other situations where only a single operand is required, *op2* is used, this allows for future extended address space usage where for example, the *MOV* instruction can use both operand fields to move data from a larger address space into the address space accessible by the other instructions. This would merely require defining a new instruction format.

Table 3.1: PD-CPU32 instruction formats.

	31	30 27	26 18	17 9	8 0
R-type	-	opc	res	op1	op2

3.2. Instructions

3.2.1. Computational (floating point)

As already mentioned before, the PD-CPU has a floating point datapath. This is reflected in the base instruction set where all instructions are floating point operations.

Table 3.2: PD-CPU32 computational floating point instructions.

Mnemonic	Opcode	Description
FADD	0000	res = op1 + op2
FSUB	0001	res = op1 - op2
FMUL	0010	res = op1 * op2
FDIV	0011	res = op1 / op2
MOV	0111	res = op2

3.2.2. NOP - No Operation

The *NOP* instruction is used to fill mandatory delay slots. E.g., if a conditional branch instruction forces the CPU to wait on the result of that operation before knowing where to continue. The NOP needs to be an operation without side-effects, often there are multiple possible ways to implement a NOP, by adding 0 to something, moving a value to its own address, jumping to the next address; However, in almost all cases side-effects, like modifying flags, should be prevented. The actual NOP instruction in the PD-CPU is implemented as a ‘MOV f7 f7’ and is therefore encoded by ‘0x77-7’ or ‘0b01110111—-0111’.

3.2.3. Control (floating point)

The control instructions are shown in Table 3.3 and are specified in the sections below.

Table 3.3: PD-CPU32 computational floating point instructions.

Mnemonic	Opcode	Description
JMP	0110	pc = pc + #res
FBGT	0101	pc = (op1 > op2) ? pc + #res : pc
EOI	1000	halt

3.2.3.1. JMP - Jump immediate

The *JMP* instruction ensures the following instruction is at the address of the current instruction, offset by the number specified in operand 2. Operand 2 is interpreted as a 9-bit signed integer, which means that jumping backwards is possible, hence loops can be implemented properly.

3.2.3.2. FBGT - Floating Point Branch on Greater Than

The *FBGT* instruction is the only available conditional branch instruction. The instruction will perform a conditional jump-offset(*res*) based on the greather-than comparison between *op1* and *op2*. The offset is a signed, 9-bit integer, allowing jumping 256 slots in both directions. Regardless of whether the branch is taken, the program counter will also be incremented by the regular increment logic, therefore, the next instruction will either be the one directly after the branch or the one at the 'current program counter + #res + 1'.

3.2.3.3. EOI - End Of Instructions

The *EOI* instruction is a contextual relevant instruction. The ISA is intended to be used on softcores which runs control algorithms. These algorithms run at intervals on a certain frequency; for example, depending on how often sensor data gets sampled. The goal is to execute a single run of the algorithm before new data is present, the *EOI* instruction indicates the end of such a run. This instruction will terminate execution and, if possible, halt the CPU. After this, an external trigger is required to start execution again with the program counter being reset to 0.

3.2.4. Extensions

All instructions shown in Table 3.4 are individual instruction set extensions; for example, the *FSQRT* can be present without a *SIN* being present.

Table 3.4: PD-CPU32 possible extensions.

Mnemonic	Opcode	Description
FSQRT	0100	res = sqrt(op2)
SIN	1001	res = sin(op2)
COS	1010	res = cos(op2)

3.2.5. Single Cycle Assumption

Data dependencies between sequential instructions can exist, by assuming that every instruction finishes, including writing back their value, within a single cycle, these data dependencies become a non-issue. This however means that the CPU will have to handle cases where instructions cannot be finished in a single cycle by stalling, as every 'cycle' a new instruction has to be issued. All pipeline slots should be filled, this also means forwarding logic is needed to solve the data-dependencies. The compiler will not insert NOP's after multi-cycle instructions, nor to solve data dependencies, as in this way, different implementation's with different operation delay's can be implemented and still execute the same assembly.

3.2.6. Mandatory delay

A new instruction can be issued every cycle, this enforces the responsibility of stalling onto the cpu-implementation. However, branches will have mandatory delay slots to avoid having to flush the pipeline and all the additional logic and complexity it would otherwise bring and this has to be implemented by the compiler. Every branch requires three mandatory delay-slots, instructions in these delay-slots are guaranteed to be executed, including jumps and branches. The logical use for the branch-delay-slots is to put branch-invariant instructions there or NOP's (and to not use them for fixed or relative jumps (*FBGT*) as this adds a lot of confusing complexity).