Design and evaluation of architectures for efficient generation of control sequences

DAVID ALBACETE SEGURA
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Design and evaluation of architectures for efficient generation of control sequences

David Albacete Segura
da5730al-s@student.lu.se

Acconner AB
Department of Electrical and Information Technology
Lund University
Universidad Politécnica de Madrid

Academic Supervisors: Joachim Rodrigues (LU) & Pedro Malagón (UPM)

Supervisor: Antonio Vicent

Examiner: Pietro Andreani

September 19, 2023

© 2023
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Ultra millimeter-wave (mmWave) radars have become a vital sensor in automotive, surveillance, and consumer electronics thanks to its precise measurements and low power consumption. However, they required a precise control to coordinate their different modules and produce meaningful data. In this thesis work, we introduce an approach to perform the control and coordination of the radar based in microcode that is served as instructions its different modules. The control system is implemented with five different architectures following two strategies. The first one consist in storing the microcode in memory from where they are latter read with direct memory access. On the other hand, second approach consists in customizing the processor to generate and push the instructions directly to the rest of the radar. The Pulpissimo system on chip is used as development platform with the Ibex RISC-V core configuration. The architectures are evaluated in terms of throughput of microcode words per clock cycle, energy consumption and area after synthesis. The results show that architectures which implement more parallelism achieve more robust throughput in exchange of a bigger area and higher energy consumption. Additionally, customizing the processor show a better throughput than architectures which use independent modules.

Popular Science Summary

Ultra millimeter-wave (mmWave) radars have become a vital sensor in automotive, surveillance, and consumer electronics. Their importance lies in their ability to be integrated into System on Chips (SoCs) and produce accurate position, speed or direction measurements of the surrounding objects. However, to produce meaningful data, keeping a precise periodicity in the sampling and processing of the measurements is key. This is managed keeping precise control and coordination of the different modules of the radar.

Nowadays, there is no standard nor common SoC architecture for mmWave radars which results in wide heterogeneous set of solutions that can be found on the market. Commonly, the radar architecture can be split in two. An analog part which transmits, receives and samples the electromagnetic signals, and a digital part in charge of the control and processing of the data. The operation of Acconeer's sensor relies in a set of microcode instructions precomputed by the control system before taking any measurement. These are pushed during the operation into the different modules of the radar to coordinate the sampling and processing of data. In this thesis work, this control system is implemented using five different architectures ranging from the usage of dedicated memories and direct memory access, to the customization of a Central Processing Unit (CPU). All of them have been evaluated in terms of number of microcode instructions pushed per clock cycle, area and energy dissipation.

Acknowledgments

I want to thank Acconeer for giving me the chance to work on my master's thesis with them. I am grateful to my colleagues Pontus, Saurabh, Subhajit, Nishant, and Martin for making the work environment enjoyable and for being always available to solve doubts. Special thanks to my supervisor Antonio for his guidance and constant support, this thesis work wouldn't have been possible without him.

I also want to express my gratitude to Lund University and Universidad Politécnica de Madrid for allowing me to pursue a double degree program and specialize in embedded electronics. Thanks also to Joachim and Pedro for reviewing the work done.

Most importantly, I am thankful for my friends, my family, María, Alfonso, Lola, and my girlfriend Cristina, who have always been there for me and supported me whenever I needed it. Their encouragement has meant a lot to me.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

SoC	System on Chip
DMA	Direct Memory Access
CPU	Central Processing Unit
FIFO	First In First Out
PCR	Pulse Coherent Radar
SNR	Signal to Noise Ratio
HWAAS	Hardware Accelerated Average Sampling
RTL	Register Transfer Level
ASIC	Application Specific Integrated Circuit
RISC	Reduced Instruction Set Computer
IO	Input Output
HPC	High Performance Computing
ISA	Instruction Set Architecture
IF	Instruction Fetch
ID	Instruction Decode
IM	Instruction Memory
EX	Execute
DM	Data Memory
WB	Write Back
AMBA	Advanced Microcontroller Bus Architecture
AHB	Advanced High-performance Bus
AXI	Advanced eXtensible Interface
APB	Advanced Peripheral Bus
IC	Integrated Circuit
IP	Intellectual Property
CPI	Cycles per Instruction
CISC	Complex Instruction Set Architecture
PC	Program Counter
CSR	Control and Status Registers

PULP	Parallel Ultra Low Power
ETH	Eidgenössische Technische Hochschule
IoT	Internet of Things
SDK	Software Development Kit
TCDM	Tightly Coupled Data Memory
HWPE	Hardware Processing Element
NoC	Network on Chip
JTAG	Join Test Action Group
DSP	Digital Signal Processing
PMP	Physical Memory Protection
GP	General purpose
ALU	Arithmetic Logic Unit
LSU	Load Store Unit
PnR	Place and Route
UART	Universal Asynchronous Receiver-Transmitter
SPI	Serial Peripheral Interface
FLL	Frequency-Locked loop
KB	Kilobytes
MHz	Megahertz
ASR	Addressed Shift Register
SRC	Shift Ring Counter

Chapter 1

Introduction

Radars are electrical systems that transmit a signal towards a region of interest and receive the echoes produced by its reflections in the surrounding objects. Based on the elapsed time between transmission and reception, and the properties of the transmitted and received signal, radar devices are able to detect presence and measure distance or speed [1]. This technology can be embedded in SoCs, being suitable for a wide range of applications within the fields of consumer, biomedical or automotive electronics. Some examples are obstacle detection [2], heart rate monitoring [3] or speed measurement [4][5].

Despite some attempts to achieve open standards for embedded radar sensors such as Googles Ripple initiative [6], nowadays, the software and hardware of these products remain heterogeneous in the market. In general, these SoCs are radiating elements that usually operate at high frequencies and with high data throughput. Consequently, even without a common standard, they face similar challenges that range from the typical memory bottleneck, present in the majority of digital architectures, to ensuring real time constraints to produce meaningful data. Common ways to deal with these challenges are memories as data exchange point between sensor and application system and Direct Access Memory (DMA) modules for data transactions, as can be seen in the Texas instruments IWR1843 [7] and Acconeers A121 [8].

This project aims to evaluate different architectures for the control system of a SoC radar sensor. To achieve this objective, first, a baseline design is defined and implemented with a CPU that produces microcode to operate a radar SoC. In the baseline, the microcode is stored in a memory and later pushed to the different modules through DMA and two FIFO buffers. Second, performance of the implemented architecture is measured in terms of throughput, power consumption and area. Third, four different approaches are proposed and implemented to evaluate and compare the different trade-offs between throughput, area and power consumption.

1.1 Background and main objective

This thesis work has been founded and carried out in collaboration with Acconeer AB.

To measure a point, Acconeer's mmWave Pulse Coherent Radars (PCRs) transmit a short pulse of known phase and uses the time difference between the transmitted and received signal to detect presence [9]. During normal operation, the sensor measures presence at consecutive points within a distance range. This set of measurements are known as sweeps. The points from consecutive sweeps are stored orderly in memory and grouped in frames to be later retrieved. This procedure is illustrated in Figure 1.1 which shows the structure of a frame composed by three consecutive sweeps targeting the same distance range.

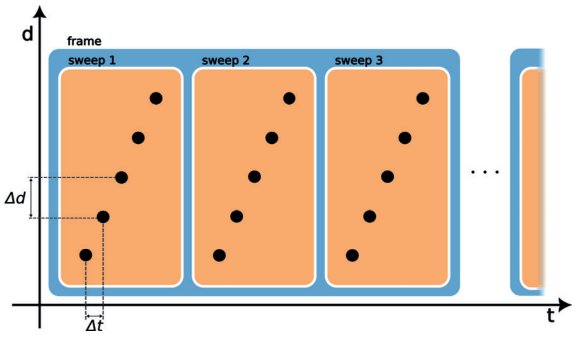


Figure 1.1: Acconeers RADAR sensor operation.

In ideal conditions, each point could be measured once. However, because of the exponential decrease in the Signal to Noise Ratio (SNR), as the distance increases, the accuracy of the measurements also decays exponentially. This scenario is illustrated by Equation 1.1, which presents the radar equation showing the SNR and its relationship with distance. The parameters showed in the equation 1.1, are the following:

- P_t: transmit power.
- G_t: transmit antenna gain.
- G_r: receiver antenna gain
- λ: radar wavelength.
- σ: radar cross section.
- d: distance.
- K: Boltzmann constant.
- T_0: reference temperature, 290 K.
- F: noise figure.
- B: effective noise bandwidth.
- L_s: transmit losses.

$$SNR = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 d^4 K T_0 F B L_s} \tag{1.1}$$

To attenuate this inconvenience, Acconeers PCRs implement a Hardware Accelerated Average Sampling (HWAAS) mechanism. This consists in measuring one point multiple times and compute the average in hardware to improve the SNR of each measurement.

One of the key features of Acconeer's sensors is their flexibility which allows the variation of parameters such as the step size (Δd), the distance range or the number of points to average during the operation. This flexibility relies on a microcontroller that precomputes and pushes microcode to coordinate the analog and digital modules of the radar. The microcode, consisting of 32-bit control words as illustrated in Figure 1.2, is grouped into chunks. These chunks determine the specific sequence in which the control words are employed, enabling various functions ranging from system setup to perform measurement. Both, the content of fields and the structure of the chunk is highly dependent in the function to fulfill. However, in the context of this thesis work, the focus is not on the specific function of each field, but rather on the time and power required to compute and push them. The fields of the microcode instruction that play a significant role for this thesis work are contained in the first thirteen bits:

- Type: Two types of control word, A if 0 and B if 1.

- Field_A and Field_B: used for decompression.

- C_end: Last control word of the chunk.

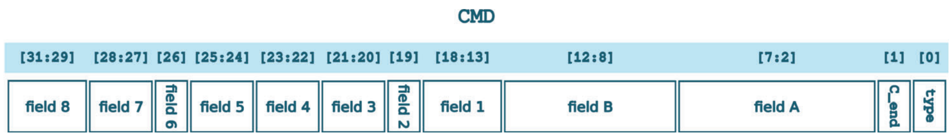


Figure 1.2: Microcode instruction structure.

In typical radar applications such as speed measurement keeping a precise periodicity between the measurements is mandatory, and, thus, the system must not run out of microcode. Figure 1.3 shows the proposed baseline solution of the radar architecture in this thesis work, which consists in buffering the microcode before being pushed to the destination modules. Multiple solutions could be addressed to compute and push the control commands into the modules of the radar, resulting in different impacts in throughput, area, and energy consumption. Therefore, the main goal of this thesis work is to propose, implement and compare multiple solutions for the control system of Acconeer's radar in terms of throughput, area and energy consumption.

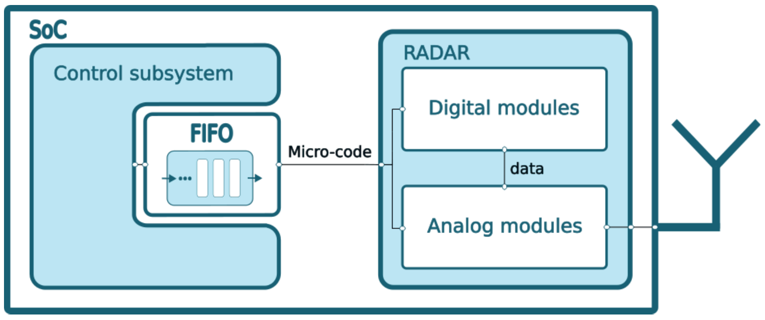


Figure 1.3: Proposed radar SoC architecture.

1.2 Work methodology

For confidentiality reasons, this project has involved selecting an embedded open-source platform on top of which a reference use case that relates to Acconeers sensors has been implemented. Using this baseline as a reference design (A0), a total of four different modifications (A1 to A4) have been proposed, implemented and evaluated. Due to the availability of the analysis tools and the long simulation time, two use cases have been used for evaluation:

- A shorter test consisting of linear sweep of 100 points with constant $hwaas$.
- A sweep of 10 points with $hwaas = i^d$ where i stands for the point number within the sweep.

The first test has been used to simulate energy consumption while the second one for throughput. Additionally, to evaluate how the system performs under stress, a periodic interrupt has been configured in the ten points sweep use case. This interrupt consists of multiple data operations to keep the memory and processing bandwidth busy. The interrupt is described more in depth in the Annex for each architecture.

The metrics used for comparison of the proposed solutions have been the following:

- **Performance** in terms of throughput which is measured as microcode instruction pushed per clock cycle.
- **Power consumption** measured based on the switching activity at gate level.
- **Area** after Application Specific Integrated Circuit (ASIC) synthesis.

1.3 Tools

The tools used in this project have been Mentor QuestaSim for the compilation and simulation of hardware modules. Power consumption was simulated at gate level with Ansys PowerArtist and, lastly, hardware modules have been later synthesized with Cadence Genus.

1.4 Thesis Outline

This report is structured as follows:

- In chapter 1, radars fundamentals and a proposed architecture is introduced and explained together with the operation of the sensor. The main objective of the thesis is defined and the methodology followed throughout the development of the thesis is described.
- In chapter 2, Reduced Instruction Set Computers (RISC) processors and common digital radar SoC architectures are introduced and summarized for better understanding of this thesis work. The candidate platforms for the baseline design are introduced and the final choice is discussed.
- In chapter 3, the baseline and its integration in the chosen open source platform is described together with the other 4 proposed solutions.
- In chapter 4, the results obtained for each optimization are discussed and compared.
- In chapter 5, the conclusions and future work of this thesis are presented.

Chapter 2

Prior work

2.1 SoC Architecture

SoCs have different architectures depending on the applications they are designed for. In microcontrollers or general purpose platforms targeting embedded applications, an extended architecture may include multiple peripherals, Input-Output (IO) interfaces, and memories connected to a Central Processing Unit (CPU) through one or more buses. The specific IO interfaces and peripheral modules vary based on the SoC's requirements, but there are commonly used approaches or standards to describe the CPU, bus organization, and memory organization system.

The memory organization system in SoCs is designed to balance speed and capacity. It involves using different levels of caches, with smaller ones placed closer to the CPU to minimize latency. The CPU first checks the closest cache level for data and instructions. If the needed information is not found, it looks in the higher cache levels and updates the closest cache with the retrieved data [13]. The number and size of cache levels depend on the system's requirements. Typically, embedded applications may only need one level, while High Performance Computing (HPC) applications might require three or four levels .

A CPU is defined by the Instruction Set Architecture (ISA), and the microarchitecture. The ISA is an abstract model of a CPU which defines its behavior, this englobes the instructions, memory and addressing models, and the IO interface of a family of implementations. On the other hand, the implementation of the ISA is known as microarchitecture and it may differs from one processor to another [13]. Therefore, while two processors may executed the same code, their different microarchitectures may lead to different, performance, area and power consumption. Despite the different microarchitectures, several processors are built upon a 5-stage pipeline microarchitecture, depicted in Figure 2.1. While the number of stages may differ, most implementations can be divided into the following building blocks [13]:

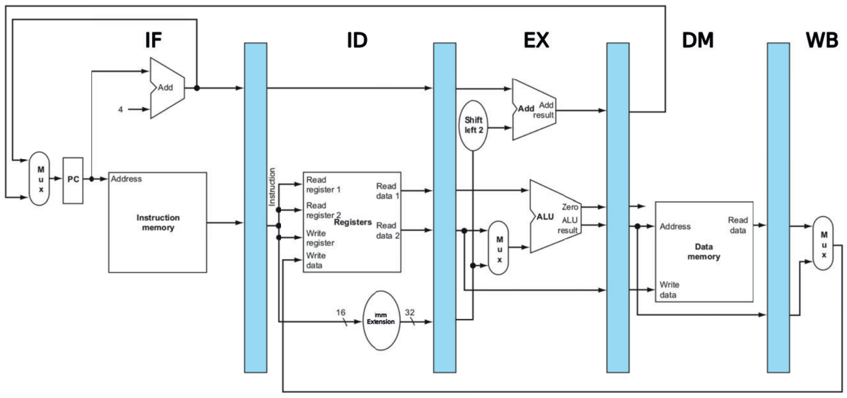


Figure 2.1: 5-stage pipelined processor (*extracted from [13]*).

- **Instruction Fetch (IF):** Fetches the next instruction to execute from Instruction Memory (IM).
- **Instruction Decode (ID):** Decodes the instruction, retrieves operands from the immediate fields of the instruction and from the General-Purpose registers.
- **Execute (EX).** Performs computational operations.
- **Data Memory (DM):** Interface with data memory.
- **Write Back (WB):** Writes results of the execution or data retrieved from memory in the general-purpose registers.

As for the previous modules, the bus organization also depends on the requirement of the system. Nowadays, ARM's Advanced Microcontroller Bus Architecture (AMBA) specification is an extended approach to define the buses and interconnection of the SoC. This standard defines various bus interfaces for different parts of the SoC. The primary buses defined in this standard are the following:

- **Advanced High-performance Bus (AHB).** The AHB is a high-performance bus that serves as the backbone of the SoC. It connects the main system components, such as processors and memories.
- **Advanced eXtensible Interface (AXI).** The AXI is a high-performance, scalable, and configurable bus protocol. It supports the connection of complex and high-bandwidth IP cores that require advanced features like burst transfers or multiple outstanding transactions.
- **Advanced Peripheral Bus (APB).** The APB is a low-power, low-bandwidth bus protocol designed for connecting peripheral devices with lower performance requirements.

In the following sections, a deeper overview of the standardized technologies used in this thesis work is given. These are RISC-V ISA and APB bus protocol.

2.1.1 RISC-V

RISC-V is an open standard ISA based on RISC principles which aim to enhance performance by compiling software into a fine-grained set of instructions. This standard is defined under the following principles[10]:

- **Cost, Simplicity and Program size:** Given the high production costs associated with integrated circuits (ICs), minimizing the design area is a critical factor to maximize the number of ICs produced per silicon wafer. The complexity of the ISA directly impacts the amount of hardware resources required to implement a processor, making desirable having instructions that perform simple and independent tasks. Moreover, the memory size significantly contributes to the overall area. Hence, it is key for the ISA to enable an efficient conversion from software to instructions.

- **Modularity and flexibility:** To face efficiently heterogeneous applications and use cases, RISC-V ISA defines different sets of instructions for arithmetic operations, instruction sizes or number of registers.

- **Performance.** The end metric to evaluate the performance of a processor is time per program. However, since this metric depends on the clock frequency, performance of designs with multiple clocks may be difficult to compare. Hence, two more commonly used metrics are the number of executed instructions and Cycles Per Instruction (CPI). RISC processors prioritize the usage of a larger number of shorter instructions whereas Complex Instruction Set Computer (CISC) processor uses fewer instructions that may take more time to execute but perform more tasks.

- **Isolation of architecture from implementation:** Designing instructions for a specific microarchitecture might boost performance for that processor. However, it might as well be problematic for other pipelines. Therefore, since RISC-V establish a standard ISA, it must fit properly different implementations.

- **Ease of programming, compiling and linking.** The RISC-V standard is based on a register-memory ISA model in which data is moved from memory to registers in order to perform operations with it. To facilitate the work of the compiler in allocating data, RISC ISA formats establish a set of 32 registers (with the exception of RV32E, which only uses 16 registers for smaller designs). Another desirable feature is performance predictability, which is managed thanks to the RISC principles since all instructions take a similar number of clock cycles to execute. Lastly, relative branching to the address of the instruction being executed (called program counter or PC) is also implemented to support dynamic linking of libraries, allowing more efficient library compilation and function calls.

- **Room for growth.** Saving space for new instructions is important to allow customizations or future enhancements without changing the ISA formats.

RISC-V ISAs

RISC-V standard is divided in two specifications: privileged and unprivileged. The former sets three privilege levels which grant access to different hardware features. These modes are Machine (M), Supervisor (S) and Hypervisor (H) mode together with address translation and page memory management methods [11]. On the other hand, the unprivileged specification defines a fourth privilege level known as User (U) mode, the memory consistency model, 4 base ISAs and 17 extensions [12].

In this thesis work, only M and U modes have been utilized since these are the ones supported by the chosen processor. M mode is the only mandatory mode and grants access to all hardware features. It is employed to execute boot code, handle exceptions, or switch to H or S modes if implemented. Conversely, U mode is employed for running regular application code. When switching between modes, the processor stores the context (PC and register content) in memory and Control Status Registers (CSR) respectively. Upon returning to the previous mode, the context is restored. Transitioning from U to M mode can occur either due to an exception or the execution of an environment call (*ecall*) instruction. To perform the latter transition, the Machine mode return (*mret*) instruction has to be executed [11][12].

The base ISAs define a set of instructions required for compiling and executing minimal software, specifying the length of the data word and the number of registers. The primary base ISA is RV32I, which establishes a data word size of 32 bits and a set of 32 general purpose registers. It is important to note that the standard does not assign a specific function to each register. However, there exists a software convention to utilize them for specific purposes as shown in Table 2.1. The other three base ISAs are variations of RV32I, as they share the same instructions but differ in either data word length, such as RV64I or RV128I with 64-bit and 128-bit word lengths respectively, or in the number of registers, as RV32E, which only utilizes the 16 lower registers of RV32I [12].

RISC-V base instruction set (which is shared by the 4 base ISAs) is composed of 5 distinct types of instructions: Integer Computational, Control Transfer, Load Store, Memory Model, and Control Status Register (CSR). Furthermore, the Integer Computational instructions are categorized based on the destination and source of the data into Register-to-Register and Immediate-to-Register instructions. On the other hand, the Control Transfer instructions are divided into conditional and unconditional instructions [12].

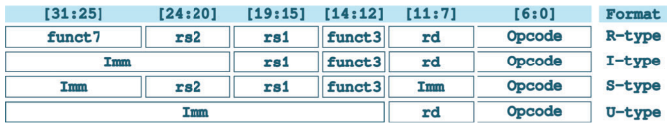
The encoding is done with 4 different formats shown in Figure 2.2. The instruction is identified by the *Opcode*, *funct3* and *funct7* fields. Rs1 and rs2 indicates the source registers of the data while *rd* is the destination register to store the result. Lastly, data can also be encoded in the instruction through the Immediate (Imm) field [12].

For example, the instruction add immediate (*addi*), I-type instruction, is iden-

Register	Convention
x0	hard-wired zero register
x1	return address
x2	stack pointer
x3	global pointer
x4	thread pointer
x5	temporal data / alternate link register
x6-x7	temporal data
x8	saved register / frame pointer
x9	saved register
x10-x11	function arguments / return value
x12-x17	function arguments
x18-x27	saved registers
x28-x31	temporal data

Table 2.1: RV32I and RV32E general purpose registers and conventional usage.

tified by the *Opcode* as an Integer Computational Immediate-to-Register instruction and the *funct3* field specifies that is an addition. The operands are taken from the *Imm* field (12-bit) and the register *rs1*, and the result is stored in the register *rd*.



[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Format
funct7	rs2	rs1	funct3	rd	Opcode	R-type
Imm		rs1	funct3	rd	Opcode	I-type
Imm	rs2	rs1	funct3	Imm	Opcode	S-type
Imm				rd	Opcode	U-type

Figure 2.2: RISC-V base instruction formats.

2.1.2 Advanced Peripheral Bus protocol

APB is an communication protocol designed by ARM as part of its AMBA specification. It is a memory mapped protocol designed to perform read and write operations from a master or requester (commonly an APB bridge) to control registers of peripheral modules known as completers [14]. The APB bus specification defines an interface with a total of 17 signals. However not all of them are needed to implement a basic bus communication, Table 2.2 describes the simplified APB interface used in this project and Figure 2.3 represents its operating states.

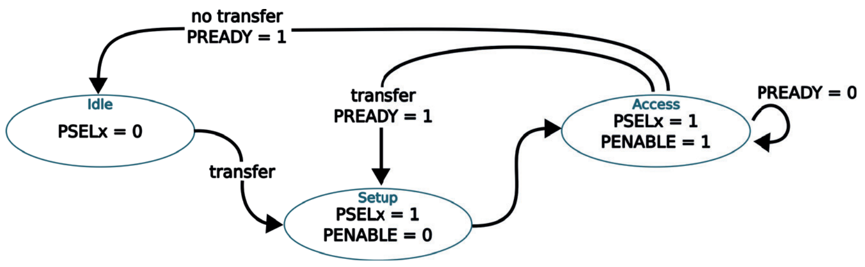


Figure 2.3: APB interface state diagram [14].

Signal	Width	Description
PCLK	1 bit	Clock signal. All signals are sampled with the rising edge of the clock.
PRESETn	1 bit	Active-low reset signal.
PADDR	Up to 32 bits	Byte address. The width depends on the number bytes that can be accessed through the bus.
PSEL	1 bit	The requester generates one PSEL for each completer. If high, the completer needs to attend the request.
PWRITE	1 bit	If set to low, indicates read access otherwise write access.
PENABLE	1 bit	Indicates the second and subsequent cycles of the transfer.
PWDATA/PRDATA	32 bits	Data bus.
PREADY	1 bit	Indicates that the completer can attend a request.

Table 2.2: Simplified version of the APB interface [14].

A typical transaction would happen as follows. Initially, the bus is in the Idle state, awaiting a transaction to be initiated by the requester through the assertion of the PSEL signal relevant to the target completer. This action prompts the bus to transition into the Setup state. Subsequently, in the Setup state, the requester triggers the PENABLE signal, causing the interface to enter the Access state, where it awaits the PREADY signal from the completer. Upon receipt of the asserted PREADY signal, data transfer occurs, after which the interface returns to the Setup state. However, if all the data has already been transferred, the interface transitions back to the Setup state.

2.2 Candidate Platforms

As mentioned in Section 1.2, two platforms have been evaluated to implement a baseline system and avoid working with Acconeer's propietary designs. The platforms that have been considered are the Parallel Ultra Low Power (PULP) Platform and Chipyard[17]. Additionally, within the PULP platform two different SoCs, Pulpino[15] and Pulpissimo[16] were considered for this thesis work.

The PULP platform is an open source project carried out by ETH Zürich and Universita di Bologna, to achieve scalable hardware and software research for low power applications that range from the field of Internet of Things (IoT) to HPC[18]. On the other hand, Chipyard is a framework that facilitates the design and evaluation of comprehensive system hardware using agile methodologies[17]. Among these two, we opted for the PULP platform since it uses a more extended hardware design language, SystemVerilog. Within the PULP platform, both Pulpino and Pulpissimo are targeted for low power and edge computing applications. In this case, the selected SoC is Pulpissimo because Pulpino dates from 2016 and it was harder to find support and documentation for it.

2.2.1 Pulpissimo

As it was previously mentioned, Pulpissimo is a SoC targeted to low power IoT applications. It supports the RISC-V cores, CV32E40P (formerly RI5CY) and Ibex (formerly ZERO RI5Y and MICRO RI5Y) [16]. As seen in Figure 2.4, the Pulpissimo SoC has two main interconnects: the Tightly Coupled Data Memory (TCDM) and the Peripheral interconnect (APB bus). Typically, the TCDM interconnect is used for heavier data transactions, while the peripheral interconnect is meant for configuration purposes. Pulpissimo has a DMA module for transferring data to/from the IO interfaces and a module (HWPE) to attach Hardware accelerators. Moreover, the TCDM interconnect enables simultaneous memory accesses from different masters (core, DMA, and HWPE) to separate memory modules, achieving efficient data processing [16].

Pulpissimo utilizes a master/slave architecture (Figure 2.4). The masters in the SoC include the DMA module, the AXI port used for connecting multiple Pulpissimo SoCs, the Hardware Processing Element (HWPE) module used for hardware acceleration, the core, and the JTAG port. In contrast, the peripherals and memory devices within Pulpissimo work as slaves.

Following this architecture, the TCDM acts as the primary bus of the system, providing low-latency access to the slaves. It consists of two channels: the request channel, which allows a master to write or read data from a slave, and the response channel, which retrieves data from the slave if necessary. The signals comprising the interface of the TCDM bus, are described in Table 2.3.

To manage access arbitration between masters, the bus implements three in-

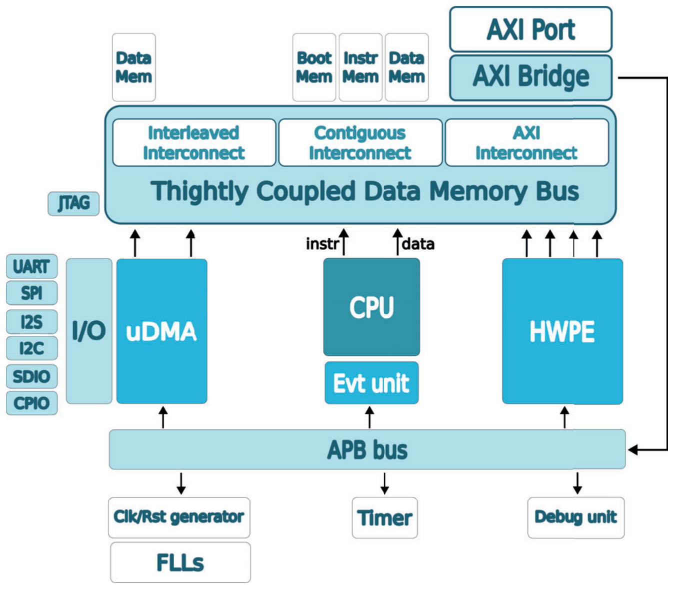


Figure 2.4: PULPissimo SoC.

Signal	Width	Channel	Description
Required	1 bit	Request	Set high to request new data
Address	32 bits	Request	System address to be accessed
Wen	1 bit	Request	High for write operations, otherwise read
Wdata	32/64 bits	Request	Data sent by the master
Be	wdata_width/8 bits	Request	Byte enable
Granted	1 bit	Request	Indicates if the master has access to address sent
Rdata	32/64 bits	Response	Data retrieved from the slave
Rvalid	1 bit	Response	Data retrieved is valid

Table 2.3: TCDM interface.

terconnects: the interleaved interconnect, which provides access to the interleaved memory space, the contiguous interconnect, which allows access to the remaining addresses and the AXI interconnect which connects the TCDM bus with the peripheral interconnect. The AXI interconnect also implements a port called AXI plug meant to connect the SoC with other Pulpissimo SoCs for multicore applications. The internal structure of the interleaved and contiguous is depicted in

Figure 2.5. Each interconnect has a request network per slave that employs a Round Robin algorithm to arbitrate accesses among the active masters. Additionally, each master has a corresponding response network that routes back the data sent by the slave. The interconnect latency is fixed and configurable, set to 1 clock cycle in this project [19].

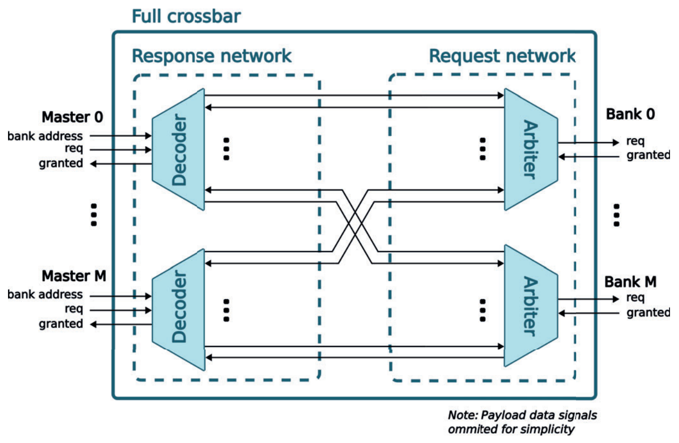


Figure 2.5: Full crossbar interconnect [19].

Masters in Pulpissimo can be classified based on the addresses they can access, the address space can be seen in Appendix A, Table A.1. Full masters, such as the core, DMA, AXI port, and JTAG, have access to both the contiguous and interleaved interconnects. On the other hand, interleaved masters (HWPE) only have access to the interleaved interconnect. The reason for this is to have a dedicated memory for hardware acceleration which would be the interleaved memory.

Also shown in Figure 2.4, a peripheral APB bus is used to access the configuration registers of the different modules. This bus is connected to the contiguous interconnect through AXI and APB bridges[18].

2.2.2 Pulpissimo: RISC-V cores

Two cores can be implemented in Pulpissimo: the CV32E40P and the Ibex Core. The CV32E40P (formerly RI5CY) is RISC-V core targeted for Near-Threshold operation in scalable IoT applications. As shown in Figure 2.6, it is composed by a 4-stage pipeline (IF, ID, EX and WB/MEM) and it supports the RV32I ISA and RV32FMC extensions. It implements as well microarchitectural modifications and custom instructions to enhance its performance in tightly coupled multi-core clusters and Digital Signal Processing (DSP) applications [20][21].

On the other hand, the Ibex Core is a smaller core targeted for arithmetic and control tasks in IoT applications. It is a 2-stage (IF, ID/EX/MEM/WB) RISC-V processor with support for the RV32I (formerly ZERO-RI5CY) or RV32E (formerly

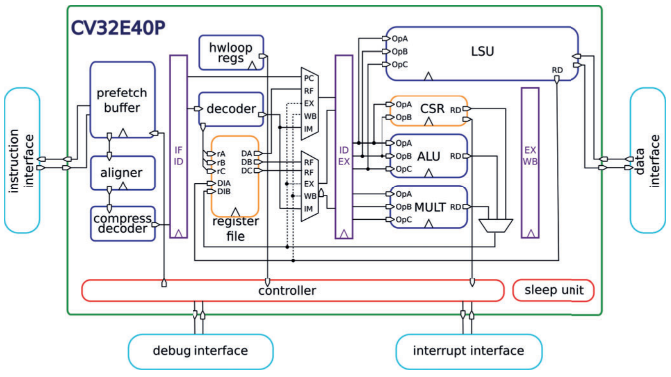


Figure 2.6: CV32E40P RISC-V core (*extracted from* [21]).

MICRO-RI5CY) ISAs and the extension RV32MC. Additional functionalities can be enabled to reduce area or increase performance. However, in this thesis work we have only considered the default core configuration. This implements a one entry instruction buffer, no branch predictor, a slow multi-cycle multiplier and a write back mechanism in the second stage.

This project has been carried out with the Ibex core because it targets control and arithmetic tasks [22] [23]. Figure 2.7 depicts the default configuration of the Ibex core.

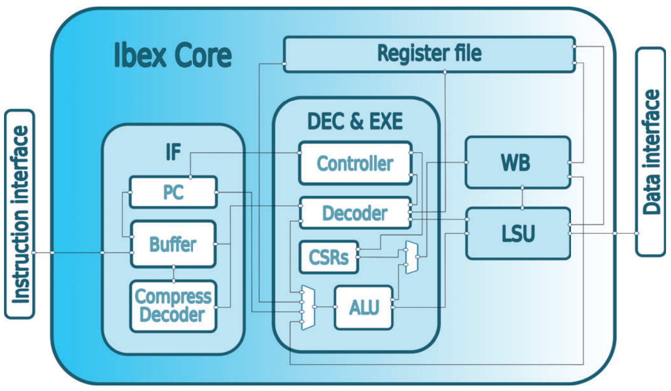


Figure 2.7: Ibex default configuration.

The modules implemented in this configuration are the following:

- **Program Counter.** The PC holds the memory address of the next instruction that is being read from memory.
- **Buffer.** 2-entry instruction buffer. It allows the core to continue with the execution even if the bus is stalled.

- **Compress decoder.** Decoder for RISC-V compressed instructions (RV32C).
- **Controller.** It controls the overall executions, it is responsible for: the startup of the core, performing address jump, exceptions, debugging and, system sleep and wake up.
- **Decoder.** Regular decoder of the CPU targeting RV32IM instructions.
- **Register file.** 32 Genearal Purpose (GP) registers used to store most recently used data.
- **Control and Status Registers (CSRs).** Set of registers used to monitor and modify the state of the CPU.
- **Arithmetic Logic Unit (ALU).** Module used for arithmetic operations.
- **Write Back (WB).** The WB stores the data retrieved from memory or the result of the arithmetic operations in the GP registers.
- **Load Store Unit (LSU).** The LSU interfaces data memory to perform load and store operations.

2.2.3 Pulpissimo: ASIC flow

Pulpissimo design flow is mainly centered around Mentor QuestaSim which is used to compile and simulate the SoC. The simulation consists of C programs that are compiled together with the SDK and the boot code into a binary file, which is later flashed into the instruction memory. To verify the correct execution of the test, the testbench produces a core.log file that contains the executed instructions, their execution time in terms of clock cycles and nanoseconds, and the contents of the registers and memory addresses accessed.

Pulpissimo provides the bender tool used for version control together with git. Bender is able to generate filelist for a range of tools such as genus which eases the integration of the synthesis flow [18].

Chapter 3

Proposed Architectures

3.1 A0: Baseline system

3.1.1 A0: Baseline hardware

The main aim of the baseline design is to set a reference system able to produce and store microcode in memory, to later retrieve it and push it to an output interface following a programmable sequence.

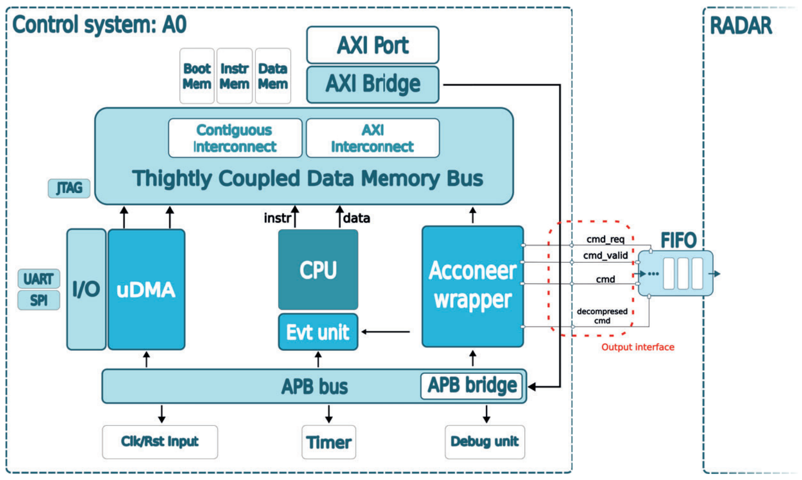


Figure 3.1: Baseline system architecture.

Figure 3.1 shows how Pulpissimo SoC architecture was modified to perform this task. First, the IO interfaces were removed to reduce synthesis time except UART and SPI interfaces that remained for debug and flash purposes. The FLL module was also removed since PnR and tape-out are out of the scope of this thesis. Second, the contiguous interconnect was left to see the impact in throughput of sharing a data memory. Third, the HWPE was replaced with a module called Acconeer Wrapper to retrieve control words from memory and push them to the radar interface. For simplicity, in this project we will assume that this interface is composed by the following five signals:

- **cmd:**. 32-bit microcode word.
- **dec_cmd:** 48-bit word, output of the decompression module.
- **cmd_valid:** 1-bit signal that validates the pushed data.
- **cmd_req:** 1-bit signal used by the radar to request more data.

A more in-depth description of the Acconeer wrapper can be seen in Figure 3.2 showing the principal connections. The wrapper is attached to the SoC through two interfaces: a TCDM full master interface to retrieve the microcode from memory and an APB interface to configure the module. The configuration of the Acconeer wrap is hold in the APB registers module, which also multiplexes APB write operations to the FIFO buffer and decompression module. Table 3.1 shows the module configuration addresses. A 4-entry FIFO buffer called chunk FIFO is implemented to store the starting addresses of the microcode chunks in memory. This FIFO is loaded through the APB bus and provides the addresses to the DMA module.

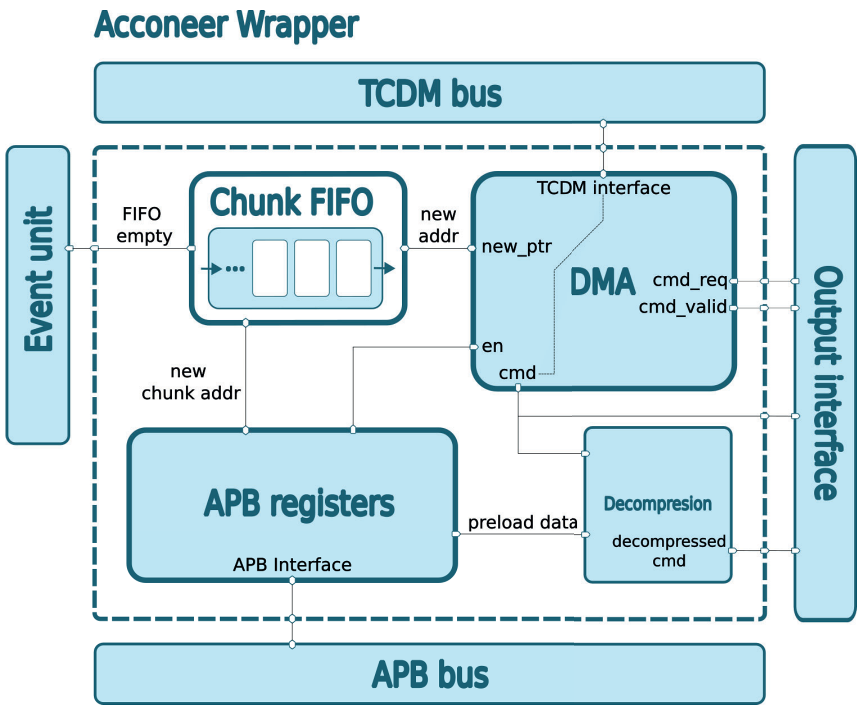


Figure 3.2: Acconeer wrap.

Module	From Address	To Address	Description
Acconeer wrapp control	0x1A10C000	-	Bit 0: starts DMA.
FIFO state	0x1A10C00C	-	Bit 0: FIFO full Bit 1: FIFO empty
Decompression module	0x1A10C100	0x1A10C3FC	Used to expand the content of the control word.
FIFO input	0x1A10C400	-	FIFO input

Table 3.1: Acconneer wrap memory map.

The DMA state machine and its interfaces are depicted in Figures 3.4 and 3.3. By default the output signals remain low and the registers maintain their state. Two internal counters are implemented to keep the track of the number of words fetched in the burst and the address to request the microcode word. The first one is increased by one while the second one is increased by four since the microcode words are 32-bit long.

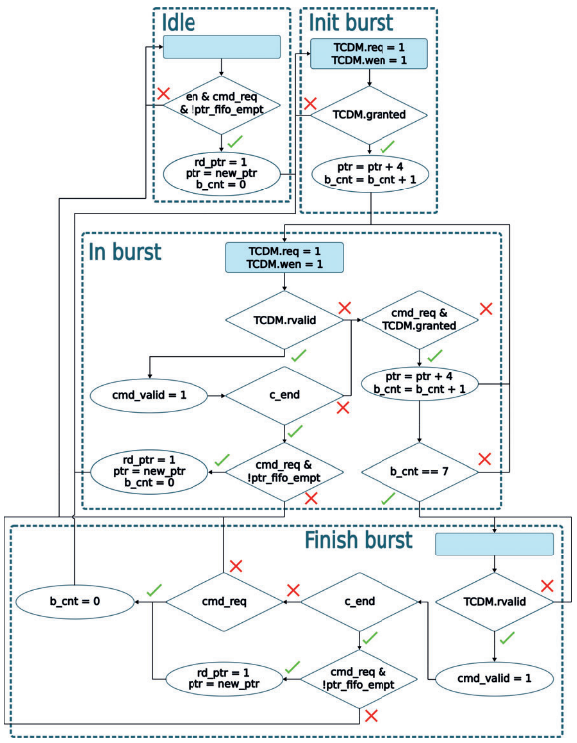


Figure 3.3: DMA ASM chart.

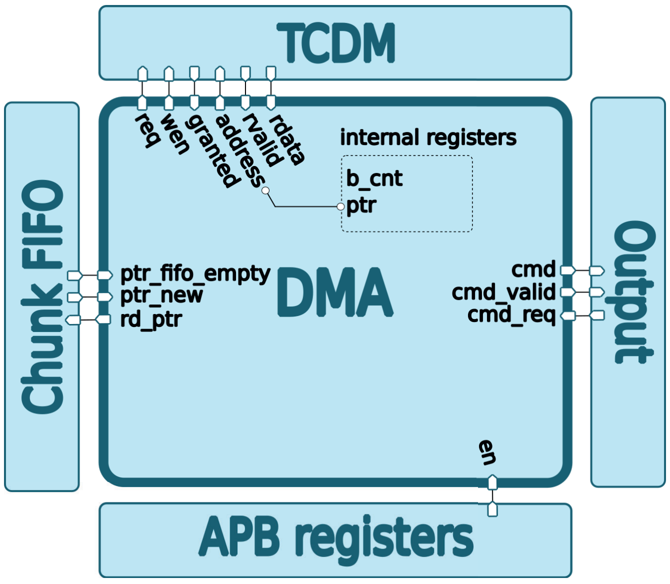


Figure 3.4: DMA interfaces.

Once enabled, the DMA fetches microcode from memory in bursts of eight words. It starts in an Idle state and remains there until the module is enable, the FIFO buffer is not empty, and microcode is required from the output interface. The control words are fetched and pushed through three states: *Init burst*, *In burst* and *Finish burst.* In *Init burst*, the first word of the burst is requested. However no microcode word is pushed since there is one clock cycle delay to read from memory. In the state *In burst* microcode words are fetched and pushed until either a *c_end* field is detected high or eight memory read operations are performed. In the first case, a complete chunk has been pushed, therefore, if more commands are required and the Chunk FIFO is not empty, the process starts over from the state *Init burst.* Otherwise the DMA returns to Idle state. In the second case, the DMA transits to *Finish burst* where it receives the last word requested from memory, and restarts a new burst if required or requests a new chunk if the *c_end* is high.

3.1.2 A0: Baseline software

As mentioned in Section 1, a sweep is managed thanks to a sequence of microcode chunks that are pushed from the control system to the radar modules. In our baseline system, to perform a sweep, a total of eight different chunks are precomputed. Two to set-up the radar, four to perform the measurements, and two to store the data. Figure 3.5 illustrates the sequence of chunks that are fetched and pushed by the DMA in both use cases (linear and exponential sweeps). The chunks corresponding to the measurements of points are repeated according to the *hwaas* parameter which, in the linear sweep is $hwaas = 3$ and in the exponential sweep follows the equation 3.1 where i stands for the point index within the sweep.

$$hwaas = \begin{cases} 3 & \text{if } i < 3 \\ i^4 & \text{if } 3 \leq i \leq 8 \\ 8^4 & \text{if } i = 9 \end{cases} \tag{3.1}$$

The chunks C1, C2, C7 and C8 are composed by twelve control words each, while the chunks C3, C4, C5, and C6 contain thirty six words each. This results in a total of 192 control words stored in memory for both use cases, which translates to 28836 and 927036 pushed words in the linear and exponential sweeps respectively. To reload the buffer with new chunk addresses, the CPU remains in a pooling loop reading the state of the FIFO buffer and pushing new addresses if there is room. Additionally, the periodic interruption used to evaluate the system in stress conditions has been configured with 100, 10 and 1 microsecond (us) in the case of the exponential sweep. The interruption has to increment and multiply a volatile variable which forces the core to read it from memory each time is accessed. See appendix B for more information.

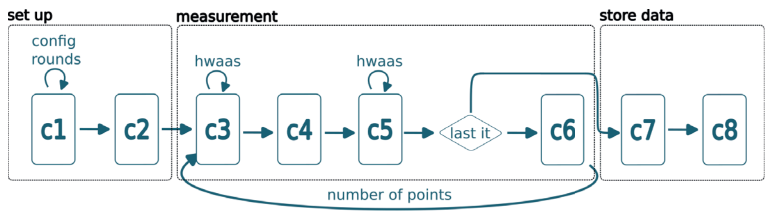


Figure 3.5: Chunk sequence used to perform a sweep.

3.1.3 A0: ASIC flow

To check the correct performance of the different architectures, a virtual module has been implemented in the testbench. This module is directly connected to the output interface of the control system. It stores the control words pushed by the Acconeer wrap in a CSV file and is later compared with a reference. If both match, throughput is measured, and synthesis and power analysis are performed.

Throughput is measured by storing three constants in the internal registers of the CPU at the start and end of the program, and when the microcode starts to be pushed. These constants can be identified in the simulation output files, as well as the time in number clock cycles when the constants were stored. These are used as time stamps.

Synthesis is performed with Genus using the filelist produced by bender as input, together with Acconeer's technology files. The instruction and data memories are replaced with two 32-bit single-port memory IPs of 8 Kilobytes (KB) each. The whole process is done for a main clock of 125 Megahertz (MHz) and a JTAG clock of 10 MHz. Lastly, the results of the synthesis are taken into account only if the timing information is correct.

Power Artist analyzes the power consumption from bender's filelist, Acconeer's technology files (physical model of logic cells), and a simulation database generated with Mentor QuestaSim. First, Power Artist elaborates the design with logic cells. Second, it computes the switching activity within the design from the simulation database. Lastly, the average power consumption between the start and end timestamps is computed from the switching activity and the information in the technology files.

3.2 A1: Dedicated memory for commands

In a real-life scenario, the processor is likely to access memory simultaneously with the DMA, forcing this last module to stall. This situation often occurs in designs involving hardware acceleration for data-intensive applications like artificial intelligence. A common solution to this problem is to utilize a dedicated memory module exclusively for a hardware accelerator.

Initially, Pulpissimo had the interleaved interconnect and memory bank for this purpose, and all masters were connected to it. This resulted in a larger area.

In this architecture, a new memory IP is used to store the microcode and is attached to the interleaved interconnect, which is implemented with three masters, the CPU, the DMA and the JTAG port. The purpose of this experiment is to explore how the throughput improves when there is no arbitration between the DMA and the CPU. Additionally, it aims to analyze how the implementation of an additional 8 KB memory IP and a interconnect affects the area and power consumption. Figure 3.6 illustrates the architecture A1.

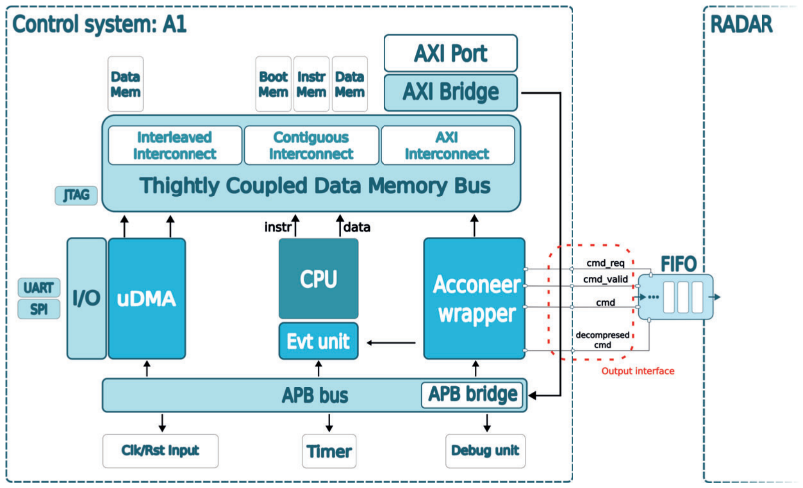


Figure 3.6: A1 system architecture.

Software remained the same as for the baseline architecture, with the exception of the instantiation of the chunks in memory. While for the baseline, they were a simple array stored in the contiguous memory, for A1 architecture, the linker script had to be modified to include the new memory. This one is addressed from 0x1C04000 to 0x1C06000.

3.3 A2: Processor customization - Pushing commands from immediate field

Architectures A0 and A1 perform APB read operations to load new chunk addresses into Acconeer's wrapper FIFO. This approach provides higher modularity and independence between modules. However, in a nutshell, this approach implies two memory accesses per microcode word: one to store the command in memory and one to read it with the DMA, which may result in higher power consumption. Moreover, there are three different buses between the CPU and the FIFO: the TCDM, AXI, and APB, which result in four and three clock cycle latencies between successive write or read operations, respectively. An approach to address these two issues is pushing the microcode directly from the core. This would reduce the number of memory accesses per pushed word and avoid the latency introduced by the buses.

A2 architecture, as shown in Figure 3.7, does not instantiate the DMA nor the chunk FIFO, but it still implements the decompression module. Since, now the only master accessing the memory is the CPU, no arbitration is needed and, therefore, the interleaved memory and interconnect may be removed as well.

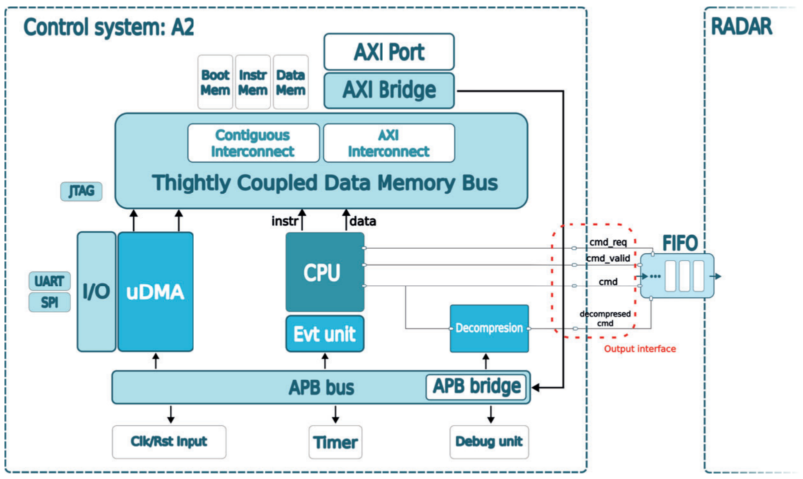


Figure 3.7: A2 and A3 system architectures.

There are two ways to address the task of pushing microcode from the CPU: computing the control words or chunks on-the-fly or encoding them in the instructions. Initially, the first approach seems promising as it would significantly reduce memory usage. However, it would introduce a periodic delay each time a command or chunk is computed. The length of this delay depends on the operations needed for the computation and the General Purpose (GP) registers available in the CPU. On the other hand, the second approach involves computing the commands during compile time. Despite the fact that this would result in higher memory usage (each command consumes four bytes), it also avoids the periodic delay. Additionally, the CPU performs fewer operations, leading to lower power consumption. Considering these factors, the A2 architecture focuses solely on exploring the second approach.

3.3.1 A2: Processor microarchitecture and ISA modifications

RV32I ISA was modified with four new instructions to push the commands directly from the *immediate* field. Since all the commands are pushed through the same interface, no destination address is required and the last 25-bit of the instruction (7 to 31) can be used to encode the command.

The new instructions are *ppA*, *pcA* to encode A microcode words with *opcodes* 0xb and 0x5b respectively, and, *ppB* and *pcB* to encode B words with *opcodes* 0x2b and 0x7b respectively. The new formats for the instructions are displayed in Figure 3.8, *ppX* format (*ppA* and *ppB*) targets chunks C2, C3, C4, C5, C6 and C7 while *pcX* (*pcA* and *pcB*) targets C1 and C8.

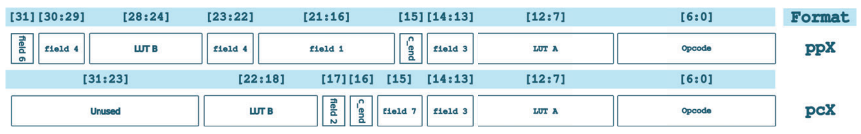


Figure 3.8: A2 instruction formats

The modifications done in the microarchitecture to support the new four instructions are shown in Figure 3.9. First, the decoder of the Ibex's ID stage is modified to recognize *ppX* and *pcX* instructions formats and extract the *immediate*. Additionally, it only validates the executed instruction if the fetch was correct. This avoids pushing erroneous commands when the pipeline needs to be flushed due to a jump or exception. A microcode decoder called *CMD mux* is added in the second stage to extract the microcode word from the 25-bit immediate and convert it into the microcode word. Lastly, the PC module is modified to stop the execution if a *ppX* or *pcX* instruction are decoded until a new control word is required (cmd_req = high).

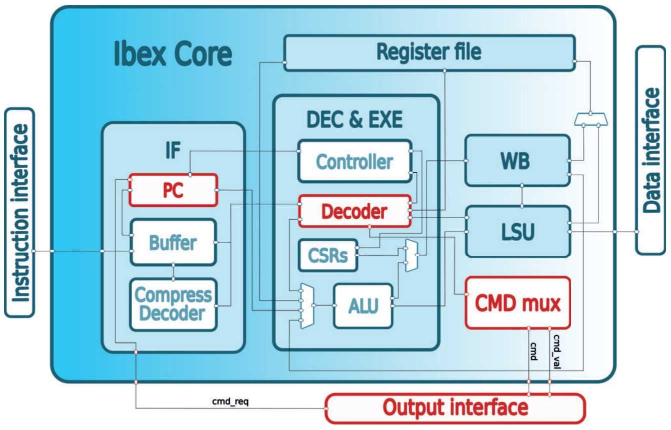


Figure 3.9: Ibex microarchitecture modifications for A2 system.

3.3.2 A2: Assembler support and software

One of the advantages of this architecture is that the computation of the microcode is done at compile time saving latency and power. However, this would require compiler support whose customization is out of the scope of this thesis work. Instead, the assembler was modified as done previously in [24] to support the new instructions. The process followed to add new instructions is described more deeply in Appendix C.1. Each microcode segment was then encoded with a function using an inline-assembly block containing the relevant *ppX* or *pcX* instructions. Subsequently, the chunks were incorporated by invoking functions in the sequence illustrated in Figure 3.5.

3.4 A3: Processor customization - Hardware accelerated chunk sequence

The main aim of architecture A3 is to speed up the execution of the sequence followed to push the chunks. In architecture A2, the Ibex executes jump instructions in two clock cycles, one to decode the instruction and another to flush the prefetch buffer [23]. Since each chunk is encoded in a different function, four extra clock cycles are needed for each chunk. Two to jump from the main function to the chunk function and two to jump back to the main. Moreover, if the compiler does not unroll the for loops, the delay increases because the core has to check the loop condition, calculate the jump address, and update the loop variable.

Architecture A3 explores the usage of a for loop accelerator to speed up the execution of the sequence. The target type of execution sequence to be accelerated is shown in Figure 3.10. It is composed of an outer for loop (number of points) which iterates over a series of inner loops, all of which have a known number of iterations that correspond to the *hwaas* parameter.

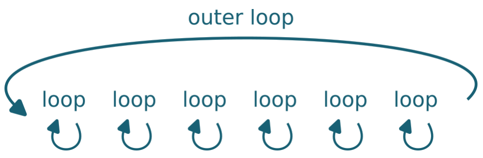
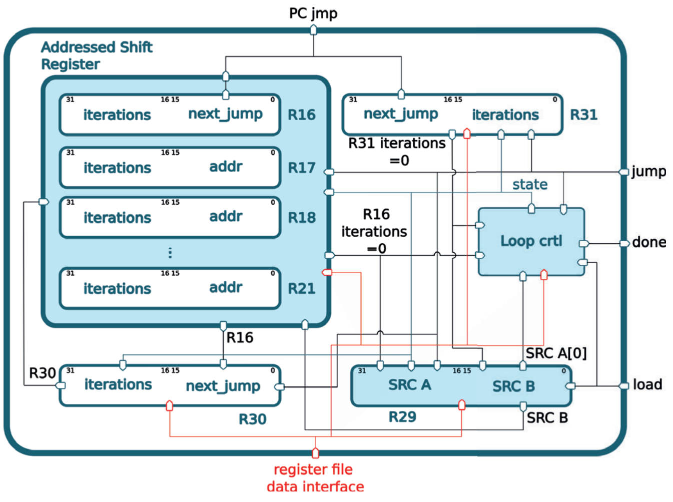
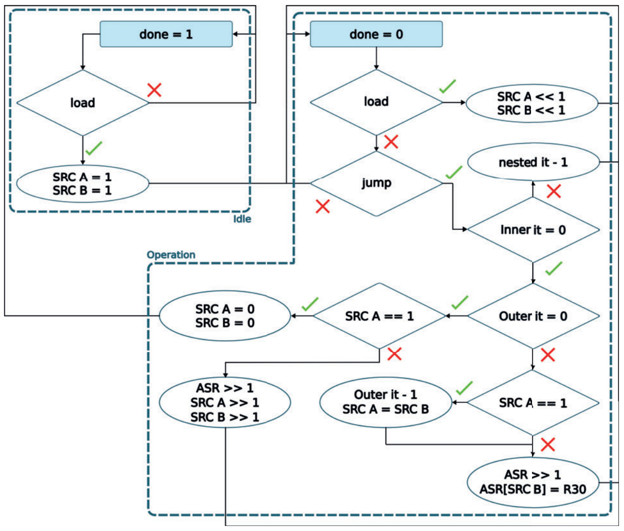


Figure 3.10: Target accelerated sequence in A3 architecture.

The implemented module as well as its ASMD diagram are shown in Figure 3.11.



(a) Chunk sequence accelerator module.



(b) Chunk sequence accelerator ASMD chart.

Figure 3.11: Chunk sequence accelerator.

The new of the registers of the for loop accelerator goes as follows:

- **Register 16 to 21:** Addressed Shift Register (ASR), a shift register whose positions can be accessed based on an address. The lower half (bits 15 to 0) of each register stores the lower 16 bits of the chunk address composing the sequence. Their upper half (bits 31 to 16) stores the number of times the chunk should be iterated (inner iterations).
- **Register 21 to 28:** Unused.
- **Register 29:** Shift Ring Counters (SRCs), counter which is incremented by shifting left 1 position and decremented by shifting right. The bits of SRC A and B reference register 16 to 20. SRC A is used to track the last function of the sequence while SRC B points the last register loaded.
- **Register 30:** It keeps a copy of register 16 which is only updated each time the address contained by register 16 is changed.
- **Register 31:** It contains the upper half of the address of the functions composing the sequence in bits 31 to 16 while in bits bits 15 to 0 the number of times to iterate over the whole sequence (outer iterations).

To save area, the upper 16 GP registers of the CPU were reused for this architecture but maintaining and prioritizing the regular data interface of the registers. This was managed compiling the Ibex hardware with 32 GP registers and compiling software with the RV32E extension instead of RV32I. This forces the compiler to use only the lower 16 registers of the CPU. These GP registers are implemented in the register file module of the CPU. Figure 3.12 shows in red the changes needed to integrate the accelerator in the CPU. The for loop accelerator is implemented in the register file module and its output connected to the PC module. Additionally, the changes done in architecture A2 were kept to be able to decode the commands.

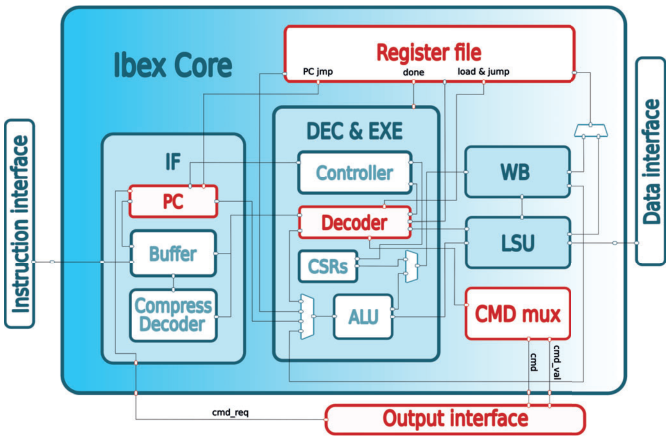


Figure 3.12: Ibex microarchitecture modifications in architecture A3.

The control of the registers is managed through the 16-bit counters stored in register 16 and 31, the input signals, and the value of the SRCs. The module operates in the following way.

First, the CPU loads the respective addresses and iterations into the corresponding registers 16 to 20, and 31. This is done with an *addi* instruction which has been modified to trigger the load signal if it targets any of the ASR registers. This load signal, increases (shifts left) the SRCs by one position. Register 30 is automatically loaded with a copy of register 16. The next address to jump to, is composed by concatenating the upper and lower halves of register 31 and 16, respectively. The module remains in Idle state until a first value is loaded, otherwise no jump can be performed with this module.

Each time a jump signal is received, counter of register 16 is decremented if it is not 0, otherwise the contents of the ASR are shifted and SRC A is decremented (shift right). If the number of outer iterations is not 0, the shift consist of a circular shift among the values of register 16 to 21. This is done by loading the value of register 30 in the register pointed by SRC B. The execution has conclude when the values of both counters, inner and outer iterations are 0, and when the SRC B is 1.

To trigger the jump signal, a variation of *jalr* instruction, the *fjr* instruction, is added with same *opcode* and *funct3* 0x7 (format I, see Figure 2.2). The *fjr* serves as a specific jump instruction for for loops. It triggers the jump signal of the sequence accelerator, saves the current PC in a GP register, and sets the PC module to select the output of the accelerator as the next instruction address. Additionally, *ppA*, *pcA*, *ppB* and *pcB* instructions were modified to perform the same operation as the *fjr* instruction if the *chunk_end* field is high, but without saving the PC. At the end of the execution, when the last chunk_end is received, the module is already empty, therefore that jump instruction is bypassed and the next *JALR* instruction retrieves the PC stored with *fjr* at the beginning of the loop.

3.5 A4: Processor customization - Parallel pipeline

Architecture A2 and A3 focuses on improving the throughput while reducing area and power. However, they don't consider the case where the CPU has to execute multiple tasks at the same time, apart from decoding and pushing microcode. Architecture A4 proposes a parallel pipeline called command pipeline which focus on delivering the microcode while the main CPU is available to perform other tasks.

3.5.1 A4: ISA

The main pipeline implements the RV32E ISA with C and M extensions as the previous architectures, while the command pipeline implements the following instructions previously introduced in A2 and A3:

- ***ppA, ppB, pcA and pcB:*** To decode and push the commands, with A3 modification that uses c_end field to jump to a new chunk address.
- ***fjr:*** To operate the chunk sequence accelerator of architecture A3 and store the return address in register 28.
- ***jalr:*** It is used to jump back to the address stored by *fjr* instruction.
- ***wfi:*** It forces the Command pipeline to return to Idle state.

Apart from the already mentioned, two new instructions were added to control the command pipeline, *start_p* and *synch_p* with opcode 0x1b and funt3 field 0x0 and 0x1 respectively. Both of them are I-format, however, while *start_p* is only implemented in the main pipeline, both implement *synch_p*. The first instruction enables the operation of the command pipeline while the second one is used to synchronize the execution between pipelines.

3.5.2 A4: Architecture

At system level, the architecture is modified added the interleaved interconnect and a dedicated memory as in architecture A1. However this time, the dedicated memory is used as instruction memory for the command pipeline. This way we avoid arbitration between the two pipelines of the processor since they are fetching instructions from different memories.

Regarding the microarchitecture of the CPU, the modifications with respect to the baseline Ibex core are marked in red in Figure 3.13. The command pipeline is composed by two stages IF and ID, and it operates the *CMD mux*, and for loop accelerator. However, The regular data interface of the accelerator is still connected to the main pipeline. This allows loading data into the accelerator with immediate-to-register instructions as it was done in architecture A3.

To addapt the for loop accelerator to the new architecture, the following registers were given a new purpose:

- Register 27 stores the return address from the loop.
- Register 28 stores the start address of the program run by the command pipeline.

To register the state of the command pipeline, a 2-bit CSR with address 0x800 was added, bit 0 indicates if its executing code, while bit 1 stall its execution. Lastly, the module Lock is implemented to coordinate the execution of both pipelines.

The IF stage acts as a TCDM master fetching instructions from the address provided by the PC. It stalls the execution of the command pipeline if no access was granted to the bus or if a *ppx or pcx* instruction was fetched but no microcode instructions were required.

The PC module is a simple mux controlled by the controller in the ID stage. It selects the next instruction address to fetch, the multiple options are:

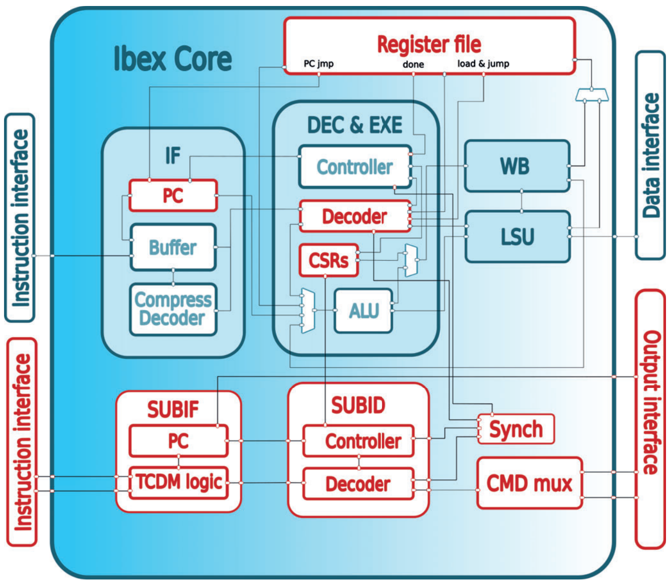


Figure 3.13: Ibex microarchitecture modifications in architecture A4.

- **PC:** Execution stalled.
- **PC+4:** Next instruction address.
- **Register 16/27:** Jump address, it can be the chunk address provided by the chunk sequence accelerator (GP register 16) or the start address of the command pipeline (GP register 27).
- **Register 28:** For loop return address.

The ID stage is composed by the controller and the decoder. As in the main pipeline, the decoder extracts the immediate field of the instruction and generates the corresponding control signals to operate the chunk sequence accelerator, the lock, the CMD mux and the controller.

Controller

An ASM diagram describing the controller is shown in Figure 3.14, the input and outputs of the module are described in table 3.2. For better clarity, the diagram shows the selected address of the PC in the IF stage in each state.

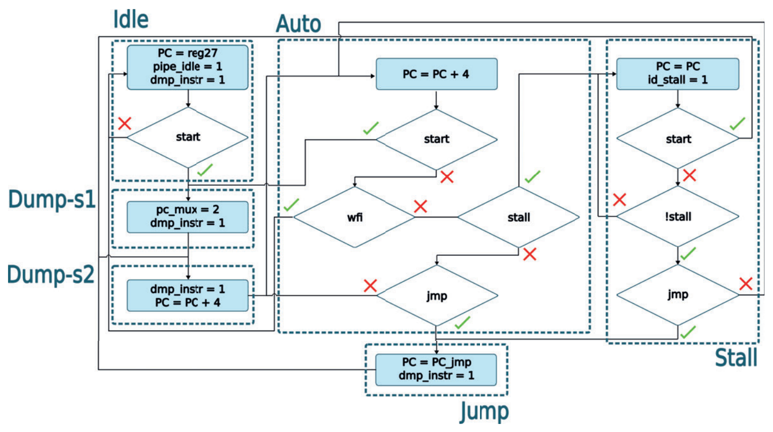


Figure 3.14: ASM diagram of the command pipeline controller.

Port name	Type	Description
start	Input	Starts the execution of the command pipeline. It is connected to the decoder of the main pipeline and it is only asserted if a *st_p* instruction is decoded
stall	Input	Stall the execution of the command pipeline. It is connected to bit-1 of CSR 0x801
wfi	Input	Forces the command pipeline to idle state. It is connected to the decoder of the command pipeline and it is only asserted if a *wfi* instruction is decoded
jmp	Input	Connected to the decoder of the command pipeline. It is asserted if a jump is performed. This happens when the instructions *jalr* or *fjr* are executed, or if the chunk_end field of a *ppX* or *pcX* is high
pipe_idle	Output	Connected to bit 0 of CSR 0x800. It also disables TCDM logic from fetching new instructions
dmp_instr	Output	invalidates the decoded instruction of the command pipeline
id_stall	Output	It disables TCDM logic from fetching new instructions
PC_sel	Output	Two bit signal which controls the PC module of the IF stage

Table 3.2: Command pipeline controller ports.

The controller consists of a moore state machine with 6 states: Idle, Dump-s1, Dump-s2, Auto, Jump and Stall:

- **Idle :** The idle state is the initial state of the Controller. This sets the PC multiplexer to select the initial instruction address (GP register 27). In this state, the output of the decoder is invalidated since no instruction has been fetched yet. Once the start signal is trigger the controller transists to Dump-s1.
- **Dump-s1, Dump-s2 :** These states are used to model the delay to read from memory. Since no buffer is implemented in the IF stage, it takes two clock cycles to fetch the first instruction at system start or after each jump. After Dump-s2, the controller transists to the Auto state.
- **Auto :** In the Auto stage the instructions are fetched successively. If a *wfi* instruction is decoded or the main pipeline triggers the start signal again, the controller restart the execution from the Idle state. If the command pipeline is stalled or a jump signal is decoded the controller transits to the Stall or Jump state respectively. Otherwise, it remains in Auto state fetching the next instruction address.
- **Stall** : In the Stall state, the controller remains fetching the same instruction until the stall input is de-asserted, then it transists to the Auto state.
- **Jump :** The jump state sets the PC to the output of the sequence accelerator and then transits to Dump-s1 to invalidate the next two instructions.

Synchronization module

A deeper overview of the Synchronization module can be seen in Figure 3.15. This module is used to coordinate the execution of both pipelines with the instruction *synch_p*. A description can be seen in Table 3.3.
The module is composed by 4 logic circuits: the command and main pipeline locks, the reset circuitry and the machine mode flag.

The command and main pipeline locks are two analogous registers that are set to 1 when a *synch_p* instruction is decoded in the corresponding pipeline stalling its execution. They are reset once both pipelines have executed a *synch_p* instruction with the same id thanks to the reset circuitry.
Lastly, the machine mode flag, bypass the Main Pipeline Lock while the CPU is in machine mode. This allows the core to attend interruptions even if a *synch_p* instruction was previously issued from the main pipeline.

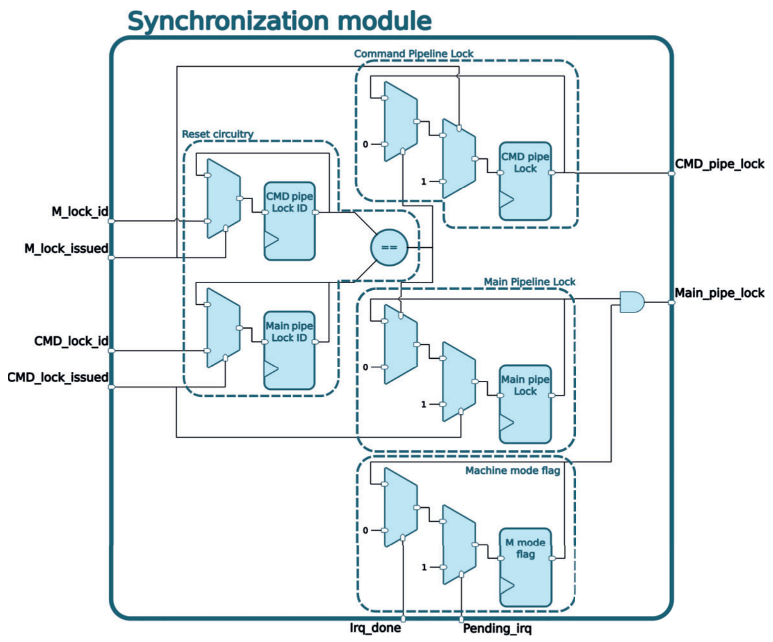


Figure 3.15: Synchronization module of architecture A4.

Port name	Type	Description
CMD_pipe_lock	Output	It is connected to the controller of the Command pipeline. If asserted, the controller stops its execution.
Main_pipe_lock	Output	It is connected to the controller of the Main pipeline. If asserted, the controller stops its execution.
CMD_lock_issued	Input	It is connected to the decoder of the Command pipeline. Triggered when a *synch_p* instruction is decoded.
Main_lock_issued	Input	It is connected to the decoder of the Main pipeline. Triggered when a *synch_p* instruction is decoded.
CMD_lock_id	Input	5-bit Id field encoded in the rd field of the *synch_p* instruction.
Main_lock_id	Input	5-bit Id field encoded in the rd field of the *synch_p* instruction.
Pending_irq	Input	It indicates if a interruption is going to be attended.
Irq_done	Input	It is triggered when the execution returns from M mode. This is when *eret* instruction is executed.

Table 3.3: Command pipeline Synchronization module ports.

Chapter 4

Results and Discussion.

4.1 Performance

The performance has been evaluated in terms of throughput. This is known as the number of microcode instructions pushed per clock cycle (cmd/clk). It was evaluated in the sweep use case with variable *hwaas* across four different scenarios varying the interruption period. The interruption periods were set at 1, 10, and 100 microseconds, while one scenario had no interruptions. The test involved a total of 927,036 micro-code words and resulted in a confidence interval of 0.00342 cmd/clk. Table 4.1 shows the results normalized with respect to the throughput obtained for the baseline in the scenario without interruption. The absolute throughput values are shown in Table C.2, in the Annex C.3. Figure 4.1 also shows a comparison of the throughput obtained for each architecture across the different scenarios.

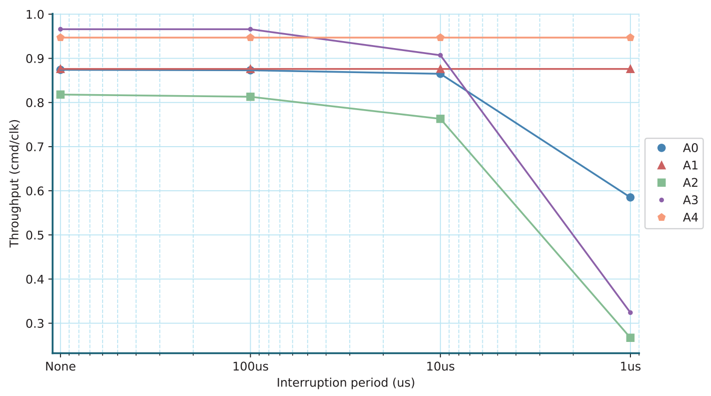


Figure 4.1: Comparison of the throughput obtained with the different architectures.

Architecture	**Interruption period (us)**			
	None	**100**	**10**	**1**
A0	1.00	1.00	0.98	0.66
A1	1.00	1.00	1.00	1.00
A2	0.93	0.92	0.87	0.30
A3	1.10	1.10	1.03	0.37
A4	1.08	1.08	1.08	1.08

Table 4.1: Throughput variation of each architecture with respect to the baseline performance in the scenario without interruptions.

The baseline system establishes a reference throughput of 0.874 cmd/clk for the use case without interruptions. Under an interruption period of 100 microseconds, the system functions correctly. However, as the interruption period decreases to 10 microseconds, the throughput begins to decrease. It experiences a significant drop of approximately 34% when the interruption period is reduced to 1 microsecond. This decline occurs due to the arbitration process in the contiguous interconnect. Both the CPU and the DMA access the memory simultaneously, leading to the DMA having to pause the micro-code flow when the CPU accesses the memory causing the reduction in throughput.

On the other hand, Architecture A1 addresses this problem by using separate memories for the CPU and the DMA. As shown in Figure 4.1, Architecture A1 maintains a consistent flow of micro-code words regardless of the interruption periods. However, it's important to note that in both Architecture A0 and A1, if the interruptions become more frequent or longer in duration, the system could reach a point where the chunk FIFO runs out of available addresses, causing a stall in the DMA operation.

In architecture A2, the results indicate that directly pushing the micro-code from the immediate field of the instruction is not a better solution. The throughput is lower than the baseline system in all scenarios. Although, the CPU in this architecture doesn't share memory bandwidth, the micro-code flow stalls when handling interruptions. This results in a larger decrease in throughput compared to architectures A0 and A1, especially with a one microsecond interruption period.

The Chunk Sequence Accelerator from architecture A3 outperforms other architectures in scenarios without interruptions and with interruptions occurring every 100 microseconds. However, like Architecture A1, the throughput decreases for shorter interruption periods due to the CPU's limited processing capacity, which can only handle one instruction per clock cycle at most.

Lastly, the parallel pipeline implemented in architecture A4 achieves a slightly lower throughput compared to architecture A3 in the first two scenarios. This is because while architecture A3 can execute jump instructions in two clock cycles, architecture A4 requires an additional clock cycle. However, in scenarios where the system is under greater stress, architecture A4 is capable of sustaining the micro-code flow at the same throughput as in the ideal scenario.

4.2 Area

Synthesis was performed using two methods: one with module grouping (global optimization) allowed and another without it. Allowing module grouping leads to better optimization results, resulting in a smaller area. However, this approach makes the analysis more difficult due to module merging. All the results were obtained using a mainstream process node from Global Foundries under standard conditions.

4.2.1 Synthesis without global optimization

Table 4.2 shows the results obtained for each module with synthesis flow that avoids module merging. The values have been normalized with respect to the modules of the baseline, the absolute values can be found in Table C.3 in Annex C.3. Additionally, Table 4.3 highlights the 50 critical paths with the shortest slack. Analyzing the critical paths can help understand where Genus had to allocate more logic or optimize further to meet timing requirements. The groups in Table 4.2 correspond to the modules shown in Figure 4.2. Architectures A1 and A4 have the largest area due to additional memory IP usage for storing micro-code or instructions. Among the two, Architecture A4 is the largest due to extra core logic (parallel pipeline, chunk sequence accelerator, Lock) resulting in a 10% area increase with respect to the baseline. On the other hand, Architectures A2 and A3 achieve the smallest areas as they utilize one less memory module.

In architecture A0, the Interconnect takes up the most space. Adding a new master to the contiguous and AXI interconnects reduces the slack in the interconnect's logic paths, requiring Genus to optimize the module's area for timing. As shown in Table 4.2, 29 out of the 50 paths with the lowest slack in Architecture A0 involve the AXI interconnect. Similarly, the Acconeer Wrapper is larger in A0 than in A1, even though it is the same module. This is because connecting the DMA to the contiguous and AXI interconnects makes the Acconeer Wrapper critical, resulting in a larger area for this module. In the other architectures, the Acconeer Wrapper consists only of the decompression module and is directly connected to the CPU, explaining its smaller size compared to architectures A0 and A1.

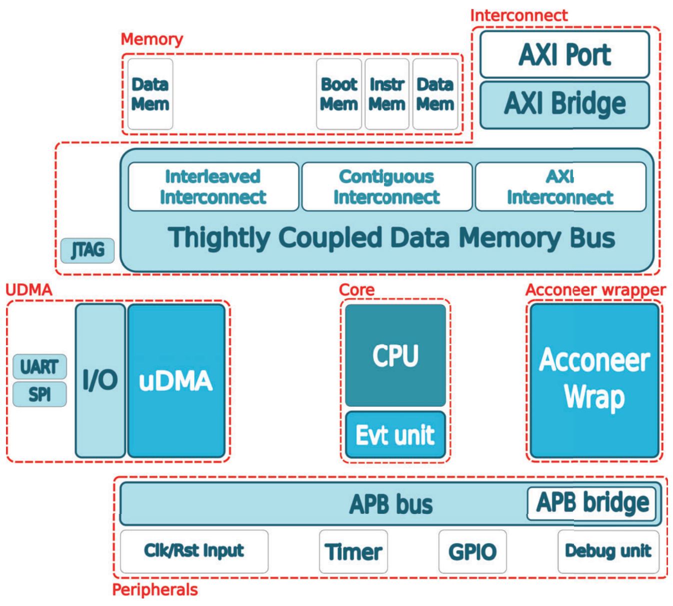


Figure 4.2: Groups used for synthesis without global optimization.

Architecture	Area (um)						
	Core	Memory	Interconnect	UDMA	Peripherals	Wrapper	Total
A0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
A1	0.99	1.18	0.95	0.99	1.00	0.88	1.08
A2	1.00	1.00	0.95	1.06	1.00	0.80	0.99
A3	1.06	1.00	0.95	1.06	1.00	0.77	1.00
A4	1.08	1.18	0.94	0.99	1.00	0.77	1.09

Table 4.2: Area per module normalized with respect to area of each module of the baseline.

Besides the memory IPs, the module which consumes the most area is the core. Comparing the area consumption of this module in architecture A0 and A4, reveals that the CPU customization with the parallel pipeline is more costly than implementing the DMA and FIFO buffer of the Acconeer Wrapper.

Architecture	Critical Path		N
	Start Point	End Point	
A0	Ibex.IF_stage	TCDM.Axi_interconnect	3
	TCDM.Axi_interconnect	Ibex.LSU	6
	TCDM.Axi_interconnect	Ibex.Register_file	19
	TCDM.Axi_interconnect	Peripherals.Timer	1
	Ibex.IF_stage	Acconeer_Wrapper.dma	1
	UDMA	UDMA	20
A1	Ibex.IF_stage	Memory.Data_mem	32
	TCDM.Axi_interconnect	Ibex.IF_stage	17
	TCDM.Axi_interconnect	Ibex.CSR	1
A2	TCDM.Axi_interconnect	Ibex.IF_stage	3
	Ibex.IF_stage	Memory.Instr_mem	17
	Ibex.IF_stage	Ibex.CSR	2
	Ibex.IF_stage	Ibex.IF_stage	28
A3	Ibex.IF_stage	Memory.Instr_mem	32
	TCDM.Axi_interconnect	Ibex.LSU	18
A4	TCDM.Axi_interconnect	Ibex.IF_stage	1
	UDMA	UDMA	19
	Ibex.IF_stage	Ibex.CSR	27
	TCDM.Axi_interconnect	Ibex.Register_file	3

Table 4.3: Main critical paths for each architecture.

4.2.2 Synthesis with global optimization

The results obtained from synthesis, allowing global optimization, are presented in Table 4.4 and Figure 4.3. The results of Table 4.4 are normalized with respect to results obtained for the baseline, the absolute values can be found in Table C.4 in Annex C.3. As it can be seen, Architecture A4 has the largest area, followed by A1 due to the additional memory. Across all architectures, more than half of the total area is occupied by memory IPs, while approximately 20% to 23% is attributed to sequential logic, and 15% to 18% is allocated for combinational logic. Lastly, inverters, buffers, and clock gates account for approximately 1% to 2% of the total area. As it happened for the previous synthesis flow, the total area variations are contained within a range of the 10% compared with the baseline. This means that, although the final area is smaller due to module merging, there are no significant changes in the critical paths across the different modules.

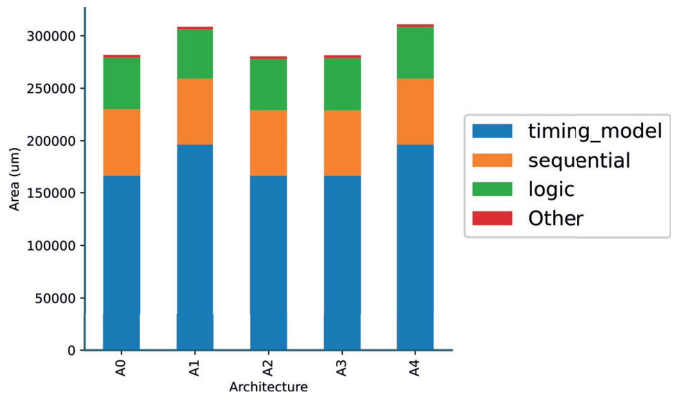


Figure 4.3: Comparison of the area obtained with the different architectures.

Architecture	Area (um)				
	IPs	Sequential	Logic	Other	Total
A0	1.00	1.00	1.00	1.00	1.00
A1	1.18	0.98	0.95	0.97	1.09
A2	1.00	0.98	0.99	1.00	0.99
A3	1.00	0.98	1.01	1.02	0.99
A4	1.18	0.98	1.00	0.98	1.10

Table 4.4: Area obtained after synthesis with global optimization.

4.3 Energy consumption

Power consumption has been measured at gate level also with a mainstream process node from Global Foundries in standard conditions. The switching activity to compute the power consumption has been obtained from the linear sweep test with 100 points. Table 4.5 and Figure 4.4 show the energy consumed by each architecture to produce and push the microcode.

As shown, the baseline and the architecture A1 are the ones with the highest energy consumption followed by architecture A4, while architectures A2 and A3 consume the least. In architectures A0, A1 and A4, the modules with the highest energy consumption are the memories due to the constant accesses performed by the DMA or command pipeline and the Ibex. On the other hand, in architectures A2 and A3, the memories consume less power since it is the Ibex the only module performing memory accesses.

The core module is the second highest energy-consuming module across all architectures. Its consumption is higher in architectures A0 and A1 due to pre-

Architecture	Energy (nJ)						
	Core	Memory	Interconnect	UDMA	Peripherals	Wrapper	Total
A0	413.586	497.491	415.456	8.261	136.789	191.538	1663.122
A1	419.333	496.200	387.575	8.273	137.145	190.798	1639.326
A2	300.913	255.201	334.977	17.819	120.713	162.951	1192.576
A3	278.006	232.405	307	16.305	110.353	162.534	1106.604
A4	382.679	562.235	328.281	16.76	114.015478	162.656	1566.628

Table 4.5: Energy consumed in a linear sweep

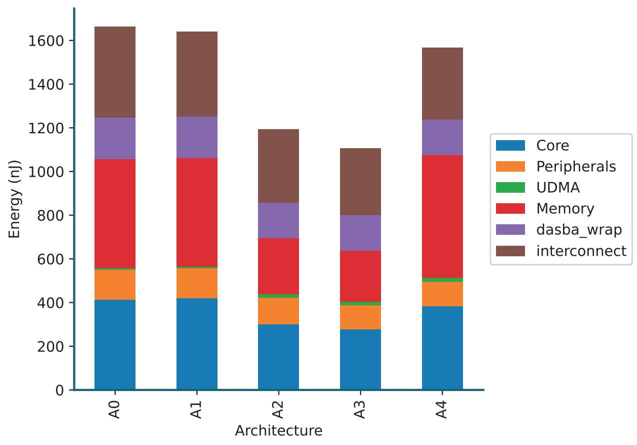


Figure 4.4: Comparison of the energy consumed in each architecture.

computing and storing commands in memory, as well as constant APB reads to monitor the state of the FIFO buffer in the Acconeer Wrapper. In contrast, the core in architectures A2 and A3 consumes less power as it stalls until more micro-code words can be pushed. Furthermore, despite not performing APB reads, the implementation of a parallel pipeline (architecture A4) that continuously fetches, decodes, and pushes data results in a significant increase compared to architectures A2 and A3.

Similarly, the power consumption of the interconnect in architectures A0 and A1 is higher compared to the other architectures. This is due to both the CPU and DMA performing memory accesses simultaneously. In contrast, architectures A2 and A3 involve only the CPU accessing the memory, resulting in lower energy consumption. In architecture A4, although both pipelines access the memory, the main pipeline remains idle until the second pipeline completes the decoding and pushing of micro-code. This idle state contributes to lower power consumption compared to architectures A0 and A1.

Lastly, the consumption of the Acconeer Wrapper is higher in architectures A0 and A1 since it reads and pushes microcode from memory while in architectures A2, A3 and A4, it is only used for decompression.

4.4 General comparison

In the previous sections the Throughput, Area and Power consumption results were introduced and discussed separately for the different architectures. This sections shows a general overview of all the results for each architecture. Figure 4.5 compares the Throughput, Area and Energy consumption for architectures A0 to A4. Each axis represent a different parameter and is normalized between 0 and 1, where 0 means the worst and 1 the best. Throughput results are presented for interruption periods of 100us 10us 1us and inf while area results are shown with and without memory IPs using global optimization.

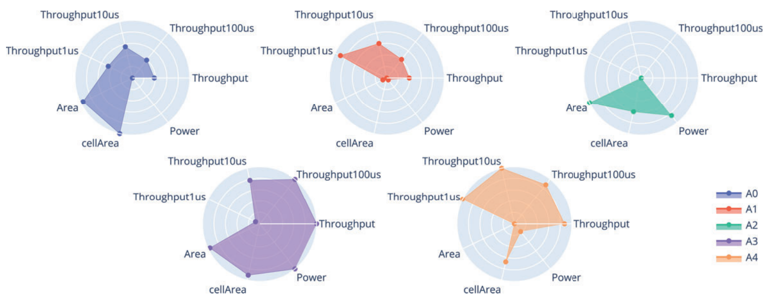


Figure 4.5: General comparison of the different architectures.

As seen, architecture A3 maintains a high throughput and achieves the best power consumption, with the second smallest area among all architectures, when the CPU is not heavily utilized. However, in scenarios where the CPU is occupied with other tasks, architectures A1 and A4 perform better than A3. A0 and A2 have the smallest area, but A2 has the lowest throughput overall and struggles even more when the CPU is stressed. On the other hand, A0 shows resilience in handling multiple tasks for the CPU, although it is still outperformed by A1 and A4. This increased robustness of A1 and A4 to stress scenarios comes in exchange of implementing an additional memory IP which results in an area increase of around 10 % in comparison with the baseline. This increase may seem insignificant, but it will depend in the requirements of the final application.

In conclusion, architectures that incorporate specific modules for microcode decoding or computation achieve a more robust throughput across different stress scenarios. This makes architectures A0, A1, and A4 suitable for SoCs targeting a wide range of applications. Furthermore, incorporating dedicated memory IP to mitigate memory arbitration enhances this resilience at the cost of increased power and area consumption, with architectures A1 and A4 generally delivering the best performance at return of higher area and power consumption. Among these two, the customization of the Core module in architecture A4 enables higher throughput and lower power consumption, while architecture A1 achieves a smaller area. Consequently, the choice between architectures A1 and A4 depends on the specific requirements of the SoC in terms of throughput, area, and power consumption.

Chapter 5

Conclusions and future work.

5.1 Conclusions

Meeting timing requirements is critical radar SoC to produce meaningful measurements. To achieve enhanced flexibility for a wide variety of use cases and scenarios, The proposed reference radar SoC is operated based on a set of microcode which is precomputed and pushed to the different modules that make up the radar. Therefore, achieving a high throughput of microcode words is key to avoid faulty outputs or system failure.

In this thesis work, a total of 5 different approaches from specific memories to store the microcode to CPU customizations are presented and evaluated in terms of throughput, area and power consumption in different stress scenarios. Considering throughput, using parallel modules to push the microcode is an approach that manages the best results across scenarios with different levels of stress for the system. However, this comes in return of bigger area and higher power consumption. The processor customization, results to be the best approach in terms of throughput, as long as it can produce the microcode flow in parallel with other tasks. However, when the CPU becomes occupied with multiple tasks, it struggles to sustain a continuous microcode flow.

5.2 Future work

The results of this thesis work have uncovered new areas to explore, which means there is a need for additional studies to expand our knowledge. In this section, some potential future research directions are outlined.

First, it would be of interest integrating the different architectures presented in Acconeer's radar SoC and compare them with the current system. Additionally, if any other than the current is selected for the control system of the radar, the design will have to pass through a new verification process.

Second, regarding the results, Genus and Power Artist tool provide a wider range of options than the one used in this thesis work. A further analysis with

these tools to throw more light into the obtained results could help to understand the difference in area an power consumption between the different architectures and how to improve it.
z Lastly, the custom instruction used in this thesis work could be accommodated to be more aligned with RISC-V design principles. Using 4 instructions to encode the microcode makes the ISA dependent on the the control word structure. Instead, only one instruction could be used to push 25-bit of data from the immediate field. Additionally, instead of adding new jump instructions for the chunk sequence accelerator, the current ones (*jal* and *jalr*) could be modified to use it, but this would require support from the compiler to recognize for loops and load the addresses in the accelerator.

Appendix A

Pulpissimo memory map

This appendix describes the address distribution for the different Pulpissimo modules.

Module	Access	Start/End addresses
AXI plug	AXI bridge	0x1000000 0x1040000
Boot ROM	Contiguous interconnect	0x1A000000 0x1A002000
FLL	APB bus	0x1A100000 0x1A100000
GPIO	APB bus	0x1A101000 0x1A102000
UDMA	APB bus	0x1A102000 0x1A104000
SoC Control	APB bus	0x1A104000 0x1A105000
Advanced Timer	APB bus	0x1A105000 0x1A106000
Event generator	APB bus	0x1A106000 0x1A109000
Interrupt Unit	APB bus	0x1A109000 0x1A10B000
Timer	APB bus	0x1A10B000 0x1A10C000
HWPE	APB bus	0x1A10C000 0x1A10F000
Stdout	APB bus	0x1A10F000 0x1A110000
Debug Unit	APB bus	0x1A110000 0x1A120000
Data memory	Contiguous interconnect	0x1C000000 0x1C008000
Instruction Memory	Contiguous interconnect	01C008000 0x1C010000
Interleaved Memory	Interleaved interconnect	0x1C010000 0x1C080000

Table A.1: Pulpissimo memory distribution [16].

Appendix B

Interruptions

This appendix describes the interruptions used to stress different architectures evaluated. Table B.1 compares how the interruption is compiled for the different architectures and its duration in clock cycles. As it can be seen, architectures A0 and A1 perform less accesses to memory than A2, A3, and A4 resulting in shorter interruptions (less instructions). Appart from loading the variables used in the variables used in the interruption, the memory accesses are also used to save the content of hte GP registers when switching from U to M mode at the beginning of the interruption and to restore it back when returning to U mode at the end. The amount of registers saved in memory when switching mode is calculated during compile time and it depends in the number of GP registers available. The number of arithmetic and jump instructions used in each architecture is constant and they are used to calculate memory addresses and to jump and return from the memory addresses where the interruption service is stored respectively.

Architecture	Total	Arithmetic	Jump	Memory	Duration
A0	25	8	2	15	85
A1	25	8	2	15	45
A2	28	8	2	18	50
A3	28	8	2	18	49
A4	28	8	2	18	66

Table B.1: Comparison of the interruptions used to stress the different architectures.

The C code corresponding to the interruption is shown bellow. The variables *var_a* and *var_b* are only used to stress the system and are not use in any other part of the code. The macro *TIMER_LO_EVT* is an HEX number that points to the bit that has to be cleared or set with the functions *rt_irq_mask_clr* and *rt_irq_mask_set* respecitvely to attend the interrupt.

```
volatile uint32_t var_a, var_b;
void __attribute__((interrupt)) timer_service() {
  rt_irq_mask_clr(TIMER_LO_EVT);
```

```
    var_a += 1;
    var_a *= var_a;
    var_b = 0;
    rt_irq_mask_set(TIMER_LO_EVT);
}
```

Appendix C

ISA and CPU customization

This appendix describes the modifications needed to implement new instructions in the Ibex Core with assembler support.

C.1 Assembler support

To utilize the new instructions from a C program, modifications are required in the assembler of the RISC-V toolchain. These modifications enable the use of the new instructions within an inline-assembly block. The process is similar to the one described in [24]. However, in this the toolchain used in this thesis work, is already adapted for the PULP platform, which may result in some differences in the steps involved.

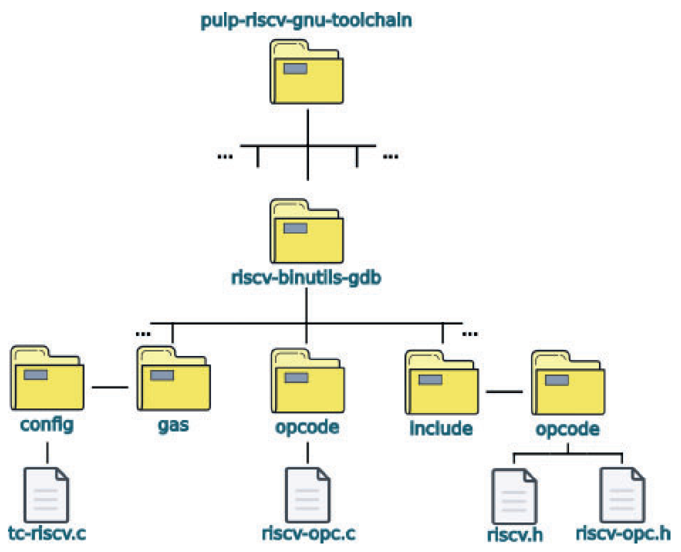


Figure C.1: PULP platforms's RISC-V toolchain directory structure.

Figure C.1 displays the directories of interest within the structure of the PULP platform RISC-V toolchain repository. The files that need to be modified to add support for new instructions to the assembler are the following:

- **riscv-opc.h** Includes the opcodes of all the instructions and other macro variables related to the codification of the different instruction format.
- **riscv.h** Includes macro functions and other utilities to encode/decode and the instructions.
- **riscv-opc.c** Instantiates the instructions.
- **tc-riscv.c** Contains the functions to parse and validate the inline-assmebly blocks.

To modify the toolchain, the following steps have to be followed:

First, the mask and match macros of each instruction have to be added to the *riscv-opc.h* file as macros. The mask is a macro that indicates the identifier bits of the instruction, while the match indicates the actual values of the identifier. These values are later used to identify the instruction using the following function.

```
static int match_opcode (const struct riscv_opcode *op, insn_t
   insn)
{
  return ((insn ^ op->match) & op->mask) == 0;
}
```

Secondly, the instructions need to be declared in *riscv-opc.c*. This file contains a struct called *riscv_opcodes*, which instantiates all the existing instructions and ISA extensions. The fields of the struct are described in Table C.1. To include a new instruction, a new entry must be added to the struct.

Thirdly, if the custom instruction uses a new format, its operands have to be added in the functions *validate_riscv_insn* and *riscv_ip* in the *tc-riscv.c* file. The first function is used to validate the instantiation of the instruction when the toolchain is compiled, and the second is used to encode the inline-assembly blocks into the binary code that is executed by the CPU.

In this thesis work, the operand "y" was added for the *ppX* and *pcX* instruction formats, referencing bits 31 to 8. Additionally, the "y" operand is also encoded in the *st_p* instruction for simplicity, but its content remains unused during execution.

The following code chunk shows the instantiation of the custom instructions used in this thesis work:

```
const struct riscv_opcode riscv_opcodes[] =
{
  /* name,        isa,    operands, match, mask, match_func, pinfo.
      */
```

Field	Description
Name	Instruction abbreviation. The abbreviation used later in the inline-assembly block.
ISA	ISA or ISA extension that defines the instruction. It is used during compilation of the toolchain to select the types of instructions that the compiler will support.
Operands	String defining the type of operand that the instruction uses. For instance, the characters "d," "s1," and "s2" reference the fields "rd," "rs1," and "rs2" in Figure 2.2, respectively.
Match	Match macro variable.
Mask	Mask macro variable. If the instruction does not use all the bits, those have to be included in this here.
Match_opcode	function used to validate the instruction. Usually, it is the condition shown in the equation ??.

Table C.1: *riscv_opcodes* struct fields.

```

{"psa",        "I",    "y",   MATCH_DASBA_SA,  MASK_DASBA_RV32,
  match_opcode, 0 },
{"psb",        "I",    "y",   MATCH_DASBA_SB,  MASK_DASBA_RV32,
  match_opcode, 0 },
{"pca",        "I",    "y",   MATCH_DASBA_CA,  MASK_DASBA_RV32,
  match_opcode, 0 },
{"pcb",        "I",    "y",   MATCH_DASBA_CB,  MASK_DASBA_RV32,
  match_opcode, 0 },
{"fjr",        "I",    "d",   MATCH_DASBA_FJ,  MASK_DASBA_FJ_RV32
  | MASK_RS1 | MASK_IMM,   match_opcode, 0 },
{"st_p",       "I",    "y",   MATCH_DASBA_SUBPIPE_CRTL,
  MASK_DASBA_RV32,   match_opcode, 0 },
{"synch_p",    "I",    "d",   MATCH_DASBA_SUBPIPE_SYNC,
  MASK_DASBA_SUBPIPE_SYNC | MASK_RS1 | MASK_IMM,   match_opcode
  , 0 },
...
```

C.2 Hardware support

Adding support for a new instruction also requires modifications in the CPU microarchitecture. To recognize and generate the corresponding control signals, the decoder in the ID stage must be modified to identify the new *opcodes*, which are defined in the *Ibex_pkg.sv* file.

The decoder entity is defined in the file *ibex_decoder.sv*. It consists of two combinational blocks (*always_comb*), each with a *switch* statement that decodes the identifiers of the instructions. While both blocks can identify all types of instructions, the second *switch* statement generates control signals specifically targeting the EX_module. The following code describes how the new instructions were implemented in the decoder:

```
logic [6:0] opcode;
assign opcode = instr_i[6:0];
always_comb begin
unique case ( opcode )
  /* default signal value */
  OPCODE_PCB: begin
    /* PCB control signals */
  end
  OPCODE_PCA: begin
    /* PCA control signals */
  end
  OPCODE_PPB: begin
    /* PPB control signals */
  end
  OPCODE_PPA: begin
    /* PPA control signals */
  end

  ...

  OPCODE_JALR: begin
    if (instr_i[14:12] == 3'b000) begin
      /* JALR control signals */
    end else if (instr_i[14:12] == 3'b110) begin
      /* FJR control signals */
    end
  end
  ...

  OPCODE_SYSTEM: begin
    if (instr_i[14:12] == 3'b000 && instr_i[31:20] == 12'h105)
     begin
      /* WFI control signals */
    end
  end
  ...

  OPCODE_SYNCH: begin
    /* SYNCH_P control signals */
  end
  OPCODE_ST_P: begin
    /* ST_P control signals */
  end
endcase;
end;
```

C.3 Absolute result values

Architecture	Interruption period (us)			
	None	100	10	1
A0	0.874	0.873	0.865	0.585
A1	0.876	0.876	0.876	0.876
A2	0.818	0.812	0.762	0.267
A3	0.966	0.966	0.906	0.324
A4	0.947	0.947	0.947	0.946

Table C.2: Throughput measured in the variable sweep use case.

Architecture	Area (um)						
	Core	Memory	Interconnect	UDMA	Peripherals	Wrapper	Total
A0	55002.462	160526.977	18236.592	32652.925	35879.331	9128.735	332261.473
A1	54734.013	190790.418	17338.934	32551.100	35899.583	8093.122	360248.519
A2	55142.677	160526.977	17328.595	34649.017	35967.268	7341.377	331817.182
A3	58450.152	160526.977	17330.423	34689.130	36072.077	7089.695	334981.064
A4	59725.589	190790.418	17158.551	32608.651	36001.452	7088.989	364191.271

Table C.3: Area without global optimization.

Architecture	Area (um)				
	IPs	Sequential	Logic	Other	Total
A0	166166.49	63960.053	49083.182	2549.098	281758.823
A1	196412.06	62747.362	46923.811	2490.062	308573.295
A2	166166.49	62940.461	48744.965	2564.386	280416.301
A3	166166.49	62772.998	49921.200	2603.900	281464.588
A4	196412.06	62877.662	49091.650	2515.230	310896.601

Table C.4: Area obtained after synthesis with global optimization.

References

[1] M. A. Richards, W. A. Holm, and J. A. Scheer, Principles of Modern Radar, 1st ed, vol. 1, 3 Vols. Edison, NC: SciTech, 2010.

[2] F. Alberius and E. Rolander, “Arbitrary motion Synthetic Aperture Radar,” thesis, Lund University, Lund, 2023.

[3] M. Al Nasser and L. Gromert, “Heart Rate Measurement using a 60 GHz Pulsed Coherent Radar Sensor,” thesis, Lund University, Lund, 2022.

[4] A. Mogyla and G. Khlopov, "Measurement of range and speed of high-speed targets by millimeter wave pulse radar," *2016 9th International Kharkiv Symposium on Physics and Engineering of Microwaves, Millimeter and Submillimeter Waves (MSMW)*, Kharkiv, Ukraine, 2016, pp. 1-3, doi: 10.1109/MSMW.2016.7538164.

[5] W. Kleinhempel, "Automobile Doppler speedometer," *Proceedings of VNIS ’93 - Vehicle Navigation and Information Systems Conference*, Ottawa, ON, Canada, 1993, pp. 509-512, doi: 10.1109/VNIS.1993.585683.

[6] "Ripple Radar Sensor API", CTA-5400, Consumer Technology Association, San Francisco, California, U.S, Jun, 2020.

[7] Texas Instruments, "IWR1843 Single-Chip 76- to 81-GHz FMCW mmWave Sensor", IWR1843 datasheet, Sept, 2019 [Revised Jan 2022].

[8] Acconeer, "A121 – Pulsed Coherent Radar (PCR)", A121 Datasheet, 2022 [Revised Mar 2022].

[9] Acconeer, "A111 – System Overview", Acconeer Handbook, https://docs.acconeer.com/en/latest/handbook/a111/system_overview.html 2019 (accessed 23 May 2023).

[10] D. A. Patterson and A. Waterman, The RISC-V Reader: An Open Architecture Atlas. Berkeley, California: Strawberry Canyon LLC, 2018.

[11] “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Docu- ment Version 20211203”, Editors Andrew Waterman, Krste Asanović, and John Hauser, RISC-V International, December 2021.

[12] “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213”, Editors Andrew Waterman and Krste Asanovi ´c, RISC-V Foundation, December 2019.

[13] J. L. Hennessy, D. A. Patterson, and K. Asanovic, Computer Architecture: A Quantitative Approach. Cambridge (U.S), Massachusetts: Morgan Kaufmann, 2019.

[14] ARM, “AMBA 3 APB Protocol” ARM IHI 0024B, Sept. 2003 [Revised Aug. 2004].

[15] PULP platform, "Pulpino", Pulpino datasheet, Jun, 2017 [Revised Jun 2017].

[16] PULP platform, "Pulpissimo", Pulpissimo datasheet, March, 2021 [Revised March 2021].

[17] Berkeley Architecture Research, "Chipyard", Chipyard, https://chipyard.readthedocs.io/en/stable/index.html 2019 (accessed 6 Jun 2023).

[18] PULP platform, "Pulpissimo", Pulpissimo, https://github.com/pulp-platform/pulpissimo (accessed 6 Jun 2023).

[19] PULP platform, "TCDM Interconnect Overview", Cluster interconnect, https://github.com/pulp-platform/cluster_interconnect/blob/master/rtl/tcdm_interconnect/ (accessed 6 Jun 2023).

[20] M. Gautschi et al., "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, pp. 2700-2713, Oct. 2017, doi: 10.1109/TVLSI.2017.2654506.

[21] Open Hardware Group, "Introduction", CV32E40P user manual, https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/intro.html (accessed 6 Jun 2023).

[22] P. Davide Schiavone et al., "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 2017, pp. 1-8, doi: 10.1109/PATMOS.2017.8106976.

[23] Lowrisc, "Ibex: An embedded 32 bit RISC-V CPU core", Ibex documentation, https://ibex-core.readthedocs.io/en/latest/index.html (accessed 6 Jun 2023).

[24] R. Raveendran and S. Bhuinya, “Customization of Ibex RISC-V Processor Core” thesis, Lund University, Lund, 2022.

Printed by Tryckeriet i E-huset, Lund 2023

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-952
http://www.eit.lth.se