Customization of Ibex RISC-V Processor Core

Rahul Raveendran
ra1711ra-s@student.lu.se
Subhajit Bhuinya
su8350bh-s@student.lu.se

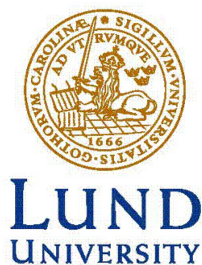
Department of Electrical and Information Technology
Lund University

Academic Supervisor: Liang Liu and Mohammad Attari

Supervisor: Antonio Vicent

Examiner: Erik Larsson

June 22, 2021



© 2021
Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgement

We would like to thank our academic supervisor, Liang Liu (Associate Professor, Lund University), assistant supervisor Mohammad Attari (PhD student, Lund University) and examiner Erik Larsson(Professor, Lund University) for all their guidance and support during the thesis. We would also like to express our gratitude to Per Odénius (ASIC Manager, Acconeer) for giving us the opportunity to work on this thesis with his team, as well as our Acconeer supervisor Antonio Vicent (ASIC Team Lead) who always encouraged us to think outside the box by sharing his extensive experience in the ASIC industry, which greatly aided us in forming our ideas and channeling our work in the right direction.
We are eternally grateful to our Acconeer colleagues Fredrik Andersson (Senior Embedded Developer) and Björn Samuelsson (Embedded Consultant) for assisting us from the very beginning of this thesis work to the very end. We are also thankful to the contributors of Ibex and RISC-V toolchain for their continuous support throughout the thesis.
Finally, we want to express our gratitude to our families and friends for all their support and for providing us with the opportunity to study at Lund University in Sweden.
To anyone we forgot to mention here, especially those who had even the smallest role in our thesis, from the bottom of our heart, we thank you.

Abstract

Despite, the fact that we now have highly powerful and reliable CPU cores from System on Chip (SoC) Intellectual Property (IP) giants, the silicon industry is still leaning toward an e cient CPU core with an open-source Instruction Set Architecture (ISA) at the moment, where RISC-V has proven to be extremely successful. The aim to customize an open-source processor core, as described in our thesis subject, in order to find a cheaper alternative to high-priced CPU cores that can be tailored for specific applications is one of the key contribution of our thesis. It was anticipated that after customization, the Ibex core will perform better, so we employed three reference algorithms to customize the Ibex Core. With a custom constructed profiler, the goal was to discover the bottlenecks in the algorithm and build specific instructions to resolve them. Following that, we added custom instructions support to both software and hardware. Finally, after implementing custom instructions, hardware simulations were conducted and synthesis was performed. Post-customization results delivered on the expected promise by drastically reducing execution time while still maintaining a degree of hardware or gate count optimization.

Popular Science Summary

The RISC-V Instruction Set Architecture (ISA) is a free and open standard based on Reduced Instruction Set Computer (RISC) concepts. The RISC-V ISA is free to use because it is distributed under open-source licenses that do not charge a fee. Through an open standard partnership, RISC-V has opened a new field of processor innovation. On core architectures, the RISC-V ISA provides a new level of open, extensible software and hardware flexibility, allowing implementation of custom instruction while retaining design innovation.
According to the current survey of the CPU industry, there are primarily two categories of CPU cores on the market. The first category uses an ISA that is licensed, such as cores from ARM, Intel etc which is typically expensive to purchase and the second category of CPU cores that are attempting to revolutionize the semiconductor industry by using an open-source RISC-V ISA that would be both freely available and customizable. This thesis aims to customize Ibex, a lowRISC processor core that runs on RISC-V ISA, demonstrating the RISC-V ISA's groundbreaking features while also proving to be a great alternative for simple, smaller applications. Ibex is a parameterizable open-source 32-bit RISC-V CPU core that is good for embedded applications. It is focused on hardware designers who want to integrate Ibex into their designs, as well as software developers who would like to create software running on Ibex. Since Ibex is open source, any Ibex user is encouraged to explore how the Ibex core can be used in their design and thus contribute to the open-source silicon development process. The Ibex core can be incorporated into any design that requires a compact, simple, and high-performance open-source processor core.

Table of Contents

List of Abbreviations xii

1 Introduction 1
- 1.1 Thesis Motivation 1
- 1.2 Goals and Challenges 1
- 1.3 Related Work 2
- 1.4 Thesis Methodologies and Outline 3

2 Ibex Processor Core 5
- 2.1 Introduction to Ibex 5
- 2.2 Ibex Architecture 5
- 2.3 Current state of Ibex 10

3 Algorithms and their Profiling 13
- 3.1 Algorithms 13
- 3.2 Profiler 15
- 3.3 Algorithm analysis using profiler 18

4 SDK Setup and Custom Instructions Implementation 21
- 4.1 SDK Setup 21
- 4.2 Custom Instruction Implementation Support 22

5 Simulation and Synthesis 31
- 5.1 Configurations used for Simulation and Synthesis 31
- 5.2 Simulation 32
- 5.3 Synthesis 33
- 5.4 Design Feasibility 40

6 Conclusion and Future Work 43
- 6.1 Conclusion 43
- 6.2 Future work 43

References 45

List of Figures

2.1 Ibex architecture [7] 6
2.2 Ibex verification setup [7] 11

3.1 Radix-2 FFT [13] 13
3.2 Profiler 16
3.3 Line profiler sample output 18
3.4 Functional profiler sample output 18

4.1 Binutils directory tree structure 23
4.2 GDB directory tree structure 23
4.3 Sample of riscv-opc.h contents 24
4.4 Spike directory tree structure 26
4.5 Ibex RTL directory structure 27
4.6 SMDRS custom instruction logic in ALU 29
4.7 ABS custom instruction logic in ALU 29
4.8 MAS custom instruction logic in ALU 30
4.9 Initial hardware architecture 30

5.1 Yosys synthesis flow 33
5.2 Execution time and area for FFT 35
5.3 Execution time and area for variance 35
5.4 Execution time and area for convolution 36
5.5 Leakage power in Ibex 37
5.6 Dynamic power in Ibex 39
5.7 Energy consumption in Ibex 40
5.8 Design feasibility 40

List of Tables

2.1 Instruction Fetch signals [7] 7
2.2 LSU Signals [7] 9
2.3 Synthesis state of Ibex [12] 12

4.1 ISS Selection Matrix 22
4.2 Instruction format 25
4.3 Custom Instructions Implemented 28

5.1 Ibex prefix nomenclature formation 31
5.2 Ibex su x nomenclature formation 31
5.3 IMC and EMC simulation cycle comparison 32
5.4 Instruction and cycles count for custom instructions 33
5.5 Area and maximum frequency comparison 34
5.6 Execution time for baseline and custom configuration 36
5.7 Leakage power for baseline and custom configuration 38

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
CSR	Control and Status Register
DIT	decimation-in-time
DSP	Digital Signal Processing
DV	Design Verification
EDA	Electronic Design Automation
EMC	Embedded Multiplication Compressed
ETH	Eidgenössische Technische Hochschule
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
GPRs	General Purpose Registers
ID	Instruction Decode
IEEE	Institute of Electrical and Electronics Engineers
IF	Instruction Fetch
IMC	Integer Multiplication Compressed
IoT	Internet of Things
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
LSU	Load Store Unit
MAC	Multiplication and Accumulation
PC	Program Counter
RISC	Reduced Instruction Set Computer
RoT	Root of Trust
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Input Multiple Data
SoC	System on Chip
STA	Static Timing Analysis

SW	SoftWare
UVM	Universal Verification Methodology
VCD	Value Change Dump
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WFI	Wait For Interrupt

Chapter 1

Introduction

1.1 Thesis Motivation

The aim of this thesis is to evaluate the performance of an open-source Ibex Central Processing Unit (CPU) core derived from lowRISC by tuning various hardware configurations and implementing unique RISC-V Instruction Set Architecture (ISA) extensions in software, such as the GNU toolchain and Spike Instruction Set Simulator (ISS). A processor is one of the key units in an electronic chip that handles computational, control, and data storage operations. But unfortunately, most current processor cores are expensive, and they are not always needed for small and simple applications for example bit-wise operations, simple arithmetic operations, control operations, etc. On the other hand, the fact that RISC-V is an open-source ISA and that Ibex works on RISC-V ISA core is the sole incentive for us to experiment and justify the core as a less expensive alternative to high-priced CPU cores that can be suitable for simple applications while still being robust and high-performing.

1.2 Goals and Challenges

Goals:

- Add custom instructions to the Ibex core to make it compatible with reference algorithms designed for specific application.
- To ensure that the Ibex processor core's performance and e ciency for all reference algorithms improves after customization.
- Reduce the number of processor gates used by reference algorithms for activity detection based on data from a radar sensor.

Challenges:

- The complexity level of ISA, the way instructions are organized in the disassembly, and the type of instructions that need to be mapped with trace log to get the cycle count for the real hardware are the reasons which made profiling of the reference algorithms for the lowest level of compiler optimization -O0 really challenging.

- After optimizing to the highest level (-O3), it was discovered that the instructions are structured out of sequence, making the building of a line profiler for (-O3) extremely difficult.
- Since both the RISC-V ISA and the Ibex core are open-source platforms, and are going through a continuous phase of update, implementing custom instructions in the Ibex processor core for specific application was possibly challenging.
- Adding custom instruction support to the toolchain, ISS, and hardware was difficult because we had to make sure that the functionality of any given custom instruction was properly implemented in all the platforms mentioned above so that when that instruction was run, the expected result was obtained.
- To ensure that the custom instructions implemented are compatible with both the Integer Multiplication Compressed (IMC) and Embedded Multiplication Compressed (EMC) baseline configurations, as the IMC configuration has 32 General Purpose Registers (GPRs) each of 32 bits in size, and the EMC configuration has 16 GPRs each of 32 bits in size.
- Following the evaluation of the synthesis results, hardware reuse and optimization proved to be a challenge because we wanted to achieve some sort of area optimization while not compromising on the execution time or performance already achieved, as we know that changing the hardware has a significant impact on the execution time.

1.3 Related Work

In terms of previous work on the Ibex processor core, formerly known as the Zero-Riscy, we have referred to a few papers that are important to the case study we will be conducting. Performance of three RISC-V cores, Riscy, Zero-Riscy, now known as the Ibex, and Micro-Riscy, were evaluated for their appropriateness in Internet of Things (IoT) applications that need high-end computational capability for IoT nodes, extreme energy efficiency, and low-cost implementation [1]. Since they are in sleep mode for most of the time, IoT devices must be able to work with extremely time-varying behaviour. At the same time, they must be responsive to external events, so that when they wake up, they must perform heterogeneous tasks such as managing interfaces to collect data from environmental sensors, storing data into memories, and so on. They must conduct relatively heavy digital signal processing in addition to the previously listed tasks in order to extract relevant information from sensor data [2]. Because of the wide range of computational requirements in application workloads, different cores can be used to serve the needs of different applications. So, essentially, an in-depth comparison of three cores is done in terms of area, output, power, and energy efficiency. The cores were compared using three application workloads, i. 2D-convolution which primarily included Digital Signal Processing (DSP) computations, ii. CoreMark which combines arithmetic and control code tasks and iii. Runtime workload that focuses on a simple set of embedded-system Runtime functions. Although the

Riscy core outperforms the Zero-Riscy core in DSP applications, there is no discernible difference in efficiency between the Riscy and Zero-Riscy cores, which are primarily designed for arithmetic and control mixed applications [1]. Furthermore, Zero-Riscy is 2 times smaller and consumes 2 times less power than the Riscy core [3]. Therefore, although Zero-Riscy is optimized for small area and low power, it still has hardware resources for multiplication and division, which are needed for signal processing algorithms like 2D-convolution, where Zero-Riscy can implement the hardware multiplication instructions. Zero-Riscy uses the least amount of energy in arithmetic-control mixed applications like CoreMark. Zero-Riscy's execution period is only 1.3 times longer than Riscy's, yet it consumes significantly less power. In terms of power and energy consumption, there is no discernible difference between the Zero-Riscy and Micro-Riscy for tasks involving only control code. Despite having a larger area than Micro-Riscy, Zero-Riscy is still the winner because it is 8.8 times faster and supports hardware resources for DSP applications [1]. In relation to our thesis, the previous study has provided us with a quick summary of the Zero-riscy core, now known as Ibex, in terms of area, performance, power and energy consumption, and current state when compared to two other cores working on similar ISA. When compared to other processor cores, the Zero-riscy or the Ibex core's suitability for IoT or embedded applications was a major motivator for us to focus more on the Zero-riscy or Ibex core and customize it for even better performance.

1.4 Thesis Methodologies and Outline

With the objective for the thesis defined, the steps taken to reach the goal are outlined below.

1. For proper execution and consistency of the reference algorithms, build the GNU compiler toolchain with baseline configurations RV32IMC and RV32EMC.
2. Build the ISS and execute the reference algorithms compiled with baseline configurations to verify the behavior of the real hardware.
3. Add support for both the baseline configurations to the Ibex software model, modify the reference algorithms to make them compatible with the Ibex model, execute them on the Ibex model with -O0 and -O3 compiler optimization to get the initial cycle count of the processor core.
4. Create a line profiler to profile the reference algorithms and obtain the initial cycle count for each line of code.
5. Analyze the output of the profiler and recommend custom instructions to ensure better performance.
6. Complete the implementation of the custom instructions by adding support for them to the GNU toolchain, ISS, and Ibex core.
7. Run the line profiler on reference algorithms after customizing the Ibex processor core to evaluate the difference in cycle count and ensure that performance has improved.

8. Analyze the area and gate count using synthesis on the customized Ibex core. Then, by modifying the Register Transfer Level (RTL) specification and reusing existing hardware, the same tailored features with a reduced area and fewer gates can be achieved.

9. Calculate the dynamic and leakage power using static timing analysis.

The rest of the thesis is outlined as follows:

- Chapter 2: Ibex Processor Core – This chapter begins with a brief overview of the Ibex processor core, accompanied by a thorough summary of the Ibex architecture and its current state.
- Chapter 3: Algorithms and their Profiling – The reference algorithms used, the construction of a line profiler, the e ects of algorithm profiling for the (-O0 and -O3) optimization levels of the compiler, and the results obtained after profiling are all covered in this chapter.
- Chapter 4: SDK Setup and Custom Instructions Implementation – This chapter begins with a brief overview of the Software Development Kit (SDK) setup including ISS and custom instructions and their importance, then moves on to the custom instructions used in the reference algorithms. The process of adding custom instruction support to the toolchain, ISS (Spike), and Ibex hardware is then described in detail.
- Chapter 5: Simulation and Synthesis Results – This chapter examines the simulation results from commercial and open-source Electronic Design Automation (EDA) tools, analyzing the profiling results obtained after custom instruction implementation, as well as the synthesis results from open-source synthesis tool and commercial synthesis tool, and their analysis to determine a viable configuration for embedded applications.
- Chapter 6: Conclusion and Future Work – This chapter concludes our thesis, sheds light on some key results, and leaves space for future research.

Chapter 2

Ibex Processor Core

2.1 Introduction to Ibex

Originally, this core was developed as part of the PULP platform [4] by a team from ETH Zurich under the name "Zero-Riscy". This core was later contributed to a non-profit company named lowRISC headquartered in United Kingdom under the name "Ibex," which is now in charge of maintaining and enhancing the core with additional features. The Ibex core is in a state of active growth, with several functional additions and enhancements to the current code. The Ibex CPU core is written in System Verilog and can be simulated using Verilator [5], an open-source simulator, as well as commercial simulators from Cadence, Synopsys, Aldec, etc. Design Verification (DV) of the CPU core is based on a Universal Verification Methodology (UVM) testbench and RTL simulation, which allows UVM to perform proper design verification. As previously mentioned, the Ibex core is highly parameterizable due to multiple configurations which can be tuned to analyze the results with di erent combinations. LowRISC, the company behind the Ibex core, is also working on the OpenTitan project with Google, which is an open-source initiative aimed at creating transparent, high-quality reference design and integration guidelines for silicon Root of Trust (RoT) chips [6]. By transparent we mean anyone can inspect, analyze, and add to this design to create more trustworthy chips. OpenTitan aims to create and sustain a logically stable architecture, including reference firmware, by using high-quality components.

2.2 Ibex Architecture

The architecture of the Ibex core is shown in figure 2.1.

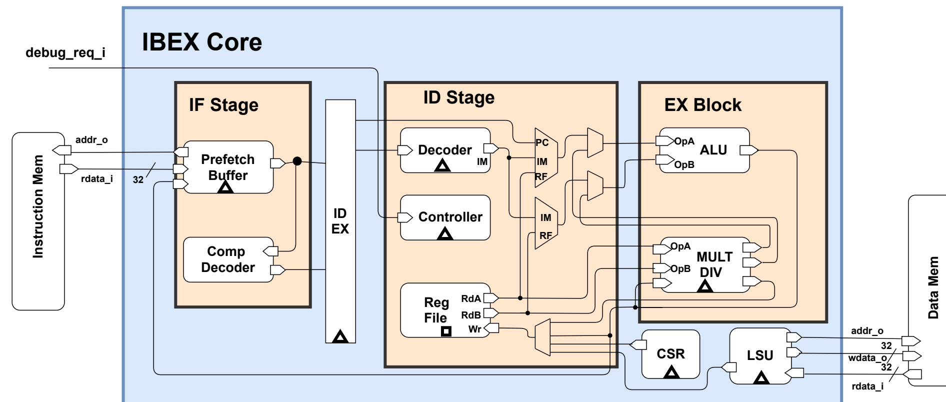


Figure 2.1: Ibex architecture [7]

2.2.1 Instruction Fetch (IF)

The prefetch bu er fetches instructions from memory and is capable of fetching one instruction per cycle if the memory subsystem allows it. If the instruction cache or instruction memory can serve one instruction per cycle, the IF stage of the core can supply one instruction to the Instruction Decode (ID) stage per cycle. For optimum output and timing termination, instructions are fetched into a prefetch bu er. This bu er only fetches instructions until it is complete. In the fetch First In First Out (FIFO), the instructions are stored along with the Program Counter (PC) from which they originated.

The instruction fetch stage manages the prefetch bu er (flushing it on branches, jumps, and exceptions, and starting prefetching from the required new PC), as well as supplying new instructions to the ID/EX stage with their PC. Despite the fact that compressed instructions are extended by the compressed decoder in the IF stage, so that the ID stage receives uncompressed instructions, the ID stage will still receive compressed instructions if an illegal instruction exception occurs. If Ibex is configured with an instruction cache rather than memory, the cache must be enabled by setting the parameter "ICache" to 1. The "icache" module has the same interfaces as the prefetch bu er, with two exceptions. First, a signal is sent to enable the cache, which is triggered by a custom Control and Status Register (CSR). Second, every time a "fence.i" instruction is executed, a signal to flush the cache is set.

By setting the "BranchPrediction" parameter to 1, Ibex can be configured to use static branch prediction. This increases e ciency by predicting that any branch with a negative o set will be taken, while any branch with a positive o set will be avoided. When the prediction is right, it eliminates a stall period from a taken branch. If a branch is incorrectly predicted to be taken, there is a mispredict penalty. The penalty is at least one cycle, or at least two cycles if the instruction is uncompressed or not aligned.
It should be noted that the "Branch Prediction" function is still in its early stages of development and has not been thoroughly tested.

Instruction Memory Interfaces

This interface is a condensed version of the data interface defined in the Load-Store Unit section. The key di erence is that the instruction interface does not support write transactions, resulting in less signals being needed. The signals used, as well as their intended purpose, are tabulated in table 2.1.

Table 2.1: Instruction Fetch signals [7]

Signal	Direction	Description
instr_req_o	output	Request valid, must stay high until instr_gnt_i is high for one cycle
instr_addr_o[31:0]	output	Address, word aligned
instr_gnt_i	input	The other side accepted the request. instr_req_o may be deasserted in the next cycle
instr_rvalid_i	input	instr_rdata_i holds valid data when instr_rvalid_i is high. This signal will be high for exactly one cycle per request
instr_rdata_i[31:0]	input	Data read from memory
instr_err_i	input	Memory access error

2.2.2 Instruction Decode

This stage decodes and executes the fetched instruction, which includes register read and write operations. This stage will be halted until all multi-cycle instructions have been completed.

Note: To flow down the pipeline, all instructions need at least two cycles. There will be one cycle in the IF stage and one in the ID/EX stage. Since not all instructions can be completed in one cycle in the ID/EX stage, they will occupy that stage until they complete. When multi-cycle instructions are not used, Ibex's overall Instructions Per Cycle (IPC) is one.

The Instruction Decode and Execute stage uses data from the instruction fetch stage to decode and execute instructions (which have been converted to the un-

compressed representation in the compressed instruction case). The instructions, including the register read and write, are decoded, and executed in a single cycle. The stage is divided into several sub-blocks, each of which is listed below.

Instruction Decode Block

The ID is in charge of the entire decoding and execution phase. It houses the multiplexers that determine what goes into the Arithmetic Logic Unit (ALU) inputs and where the write data for the register file come from. To control multi-cycle instructions, a small state machine is used, which stalls the entire stage while the multi-cycle instruction is being executed.

Controller

The state machine that controls the processor's overall execution is found in the Controller. It is in charge of:

- Handling core start-up from reset.
- Setting the PC for the IF stage on jumps/branches.
- Handling exceptions/interrupts (jump to acceptable PC, set specific CSR values).
- Controlling sleep/wakeup on Wait For Interrupt (WFI).

Decoder

The decoder receives uncompressed instruction data and sends appropriate control signals to the other blocks, allowing the instruction to be executed.

Register File

If the RV32E extension is disabled, Ibex has 32 registers each of size 32 bits and if it is enabled then Ibex has 16 registers each of size 32 bits. Register x0 is statically bound to 0 and has no sequential logic; it can only be read. The register file has two read ports and one write port, and data from the register file is available the same cycle that a read request is made. Since there is no write to read forwarding path, if one register is read and written at the same time, the read will return the current value instead of the value being written.

Execute Block

The ALU and multiplier/divider blocks are contained in the execute block.

Load Store Unit (LSU)

The core's LSU is in charge of accessing the data memory. Words (32 bits), half words (16 bits), and bytes (8 bits) can be loaded and stored. Any load or store will cause the ID/EX stage to stall for at least one cycle while waiting for an answer (whether that is awaiting load data or a response indicating whether an error has been seen for a store).

Data Memory Interface

Table 2.2 shows the signals used by the LSU.

Table 2.2: LSU Signals [7]

Signal	Dierction	Description
data_req_o	output	Request valid, must stay high until data_gnt_i is high for one cycle
data_addr_o[31:0]	output	Address, word aligned
data_we_o	output	Write Enable, high for writes, low for reads. Sent together with data_req_o
data_be_o[3:0]	output	Byte Enable. Is set for the bytes to write/read, sent together with data_req_o
data_wdata_o[31:0]	output	Data to be written to memory, sent together with data_req_o
data_gnt_i	input	The other side accepted the request. Outputs may change in the next cycle
data_rvalid_i	input	data_err_i and data_rdata_i hold valid data when data_rvalid_i is high. This signal will be high for exactly one cycle per request
data_err_i	input	Error response from the bus or the memory: request cannot be handled. High in case of an error.
data_rdata_i[31:0]	input	Data read from memory

CSR

All of the CSRs (control/status registers) are included in the CSR. This block handles all CSR reads and writes, and it additionally stores performance counters and increments them as required. Data from a CSR can be read in the same cycle as it is requested.

ALU

ALU is a strictly combinational block in the RV32I RISC-V Specification [8] that implements operations needed for integer computational instructions and comparison operations required for control transfer instructions. The ALU is used by other blocks for the following tasks:

- As part of the multiplication and division algorithms, Mult/Div uses it to perform addition.

- It uses PC + Imm to calculate branch targets.
- It uses Reg + Imm to calculate memory addresses for loads and stores.
- When performing two accesses to manage an unaligned access, the LSU uses it to increment addresses.

Multiplier/Divider Block (MULT/DIV)

The Multiplier/Divider (MULT/DIV) is a multiplication and division block powered by a state machine. The only di erence between the fast and slow models is this block. The long division algorithm is the same in all versions and uses the ALU block.

2.3 Current state of Ibex

2.3.1 Ibex configurations

Ibex currently uses parameters that are divided into two categories, which are fully verified and experimental. A fully verified parameter is one that is completely supported by most tools, while experimental means that the parameter is still being developed and needs to be updated. Some of the parameters in the fully verified category are RV32E (Embedded Extension), RV32M (Multiplication extension), RV32C (Compressed Extension), RV32B (Bit-Manipulation Extension) etc. RV32I (Integer extension) is the default configuration, with 32 GPRs of 32 bits each while RV32E is a configuration of 16 GPRs of 32 bits each.

RV32M allows the selection of three multiplication modes assisted by the Ibex core design.

- RV32MSingleCycle - Three parallel multiplier units are used in the single-cycle multiplier, which are mapped to hardware multiplier primitives on Field Programmable Gate Array (FPGA). As a result, it is the first choice for FPGA synthesis. This mode performs multiplication in one cycle, requiring three parallel 17-bit 17-bit multiplication units. The synthesis of Application Specific Integrated Circuit (ASIC) has not yet been checked, but it is estimated to take up 3-4 times the area of the fast multiplier.

- RV32MFast - The fast multi-cycle multiplier achieves a good balance of area and e ciency. For ASIC synthesis, it is the first option. Multiplication is done in three cycles. It uses a Multiplication and Accumulation (MAC), which is an internal multiplication and division block that can perform 17-bit 17-bit multiplication with a 34-bit accumulator.
- RV32MSlow - Performs multiplication at normal speed, requiring 30 to 33 clock cycles. The ALU block is used for addition.

The bit manipulation extension is included in RV32B, which is disabled by default. This parameter consists of three sub-extensions RV32BNone(default), RV32BBalanced,

and RV32BFull. In addition to these parameters, Ibex CPU core has parameters for selecting a register file from one of the three options: "RegFileFF" (default), "RegFileLatch", and "RegFileFPGA". Since Ibex core is an open-source core, there are several parameters that are still in the experimental stage, but to name a few, it has BranchTargetALU - This parameter provisions the use of a separate ALU for calculating branch target, allowing it to eliminate stall cycles from taken branches and increase the performance. WriteBackStage improves the performance of loads and stores.

2.3.2 Verification of Ibex

Figure 2.2 shows the verification structure of Ibex

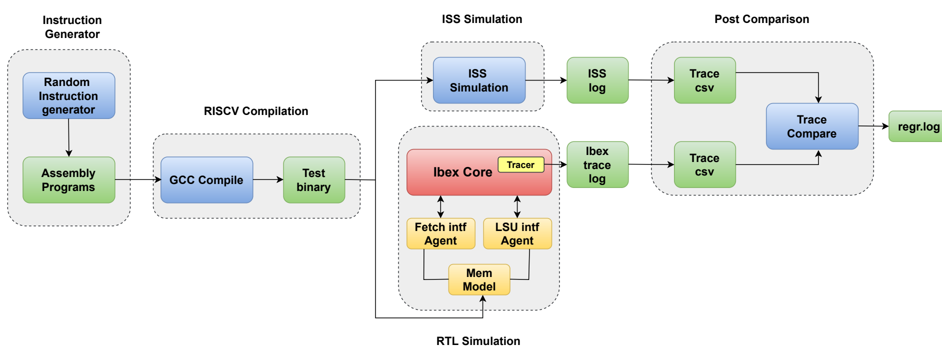


Figure 2.2: Ibex verification setup [7]

Ibex environment uses a SytemVerilog UVM testbench for verification. In order to run the above co-simulation flow we need some pre-requisites.

- A SystemVerilog simulator that supports UVM.
- A RISC-V ISS such as Spike.
- The RISC-V toolchain is used to compile or assemble the created programs prior to simulation.

At high level, the testbench generates compiled instruction binaries using an open-source RISCV-DV random instruction generator which is already integrated in Ibex generates assembly programs related to a specific set of instructions. These assembly instructions are then further compiled using the RISC-V toolchain and converted into corresponding binaries. These program files or binaries are loaded into the memory model of Ibex as well as supplied to the ISS. The final stage of this flow deals with log comparisons in order to assess the correctness of a simulation. The trace logs provided by the core and the chosen golden model ISS are both parsed to collect information about all register writebacks that occur. The data from these two sets of register writebacks are compared to ensure that the core is writing the correct data to the correct registers in the correct order.

2.3.3 Synthesis of Ibex

To help the readers concentrate on the current state of Ibex, we have put together a table that shows an in-depth overview of the Ibex core in di erent configurations. Table 2.3 shows the synthesis results of the Ibex simple system with an open-source synthesis tool, Yosys [9], timing analysis tool, OpenSTA [10] in terms of area and performance. Results are based on open-source standard cell library Nangate45 [11]. The output figures are currently based on CoreMark running on the Ibex simple system platform, while the Yosys synthesis area figures are based on the Ibex basic synthesis flow using a latch-based register file. When we look at the verification status, we can see that red indicates that a configuration has been properly checked, green indicates that it has been properly verified according to industry requirements, and amber indicates that some verification has been done but the configuration is still in the experimental stage.

Table 2.3: Synthesis state of Ibex [12]

Config	micro	small	maxperf
Features	RV32EC	RV32IMC, 3 cycle mult	RV32IMC, 1 cycle mult, Branch target ALU, Writeback stage
Performance (CoreMark/MHz)	0.904	2.47	3.13
Area - Yosys (kGE)	17.44	26.06	35.64
Area - Commercial (estimated kGE)	16	24	33
Verification Status	Red	Green	Amber

Chapter 3

Algorithms and their Profiling

3.1 Algorithms

Three reference algorithms used in this thesis are as follows

- Fast Fourier Transform (FFT)
- Variance
- Convolution

For radar signal processing, these three algorithms are the most widely utilized. The variance algorithm is used to detect activity in the channel when the radar subsystem wakes up. It is also known as the activity detection algorithm since it is used to identify activity in the channel once the subsystem comes out of the sleep mode. Convolution is a popular signal processing operation that is used for filtering, and in this case it is accomplished using a triangular smoothing filter. The FFT algorithm is used to analyze complex processes in frequency domain. It is also utilized for sensor calibration, some form of speed measurement, and gesture control.

3.1.1 FFT

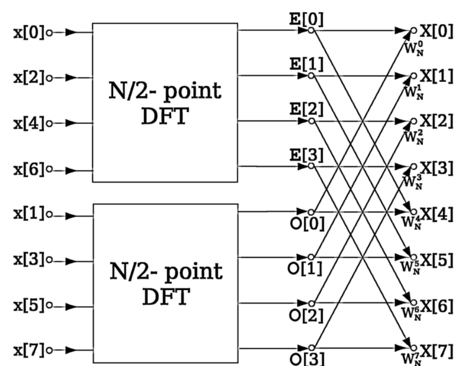


Figure 3.1: Radix-2 FFT [13]

Figure 3.1 shows the basic structure of a raidx-2 FFT. FFT is implemented using radix-2 Cooley-Tukey algorithm with decimation-in-time (DIT) [13]. With each recursive step, the radix-2 implementation divides the FFT of size N into two interleaved FFTs of size N/2. Radix-2 DIT computes the FFT of even-indexed and odd-indexed inputs separately before combining the two results to obtain the FFT of the entire series. The operation is performed with fixed-point precision on 64 input samples of complex data type. This algorithm is based on 16-bit integer arithmetic. Due to the fact that it is an integer arithmetic, FFT scaling has been used. For an FFT that works with 16 bits per complex component, there are 3 common approaches for handling the scaling; First one is the allocation of some 16 bits of each number to an exponent and use so called half-precision floating point numbers. This is a good choice if you add hardware support for that number representation, but it introduces a lot of overhead on a processor without such hardware support. Second one is the use of a common exponent for the full data array and adapt the scaling of the data as the calculation proceeds and last one is selection of a fixed scaling convention with no exponent in the output. This is the simplest but the least flexible approach, and if the characteristics of the input data are not very well-known, this approach is likely to lead to loss of precision or integer overflow. However, each radix-2 stage involves data additions and subtractions, with one of the operands being a complex number with unit absolute value multiplied, which implies that if the input data contains values that are too close to the 16-bit integer representation's limits, the result can overflow. It also means that the stored complex numbers absolute values can only increase by a factor of two. As a result, one or zero bits from the data must be shifted out. If the data is shifted, the exponent must be increased to maintain the represented values, even if rounding errors occur. The FFT algorithm is also Single Input Multiple Data (SIMD) friendly, which means it can use compositely packed data to perform parallel operations.

3.1.2 Variance

Variance algorithm initializes or updates a statistics state array with the variance score of the most recent input data sample from a data array containing all input data samples. This algorithms deals with 60,000 input samples, which is a huge amount. The 16-bit integer data type is used for the input data samples. In relation to the cumulative averages and variance, this algorithm computes and returns the variation score of the most recent input results. The variance score has a high rating, indicating that the most recent data varies greatly from the distribution of the most recent updates. The parameters used to construct this algorithm are mentioned as follows.

- Data array which contains all the input data samples.
- Data length which is the number of elements in the input data
- State array for the statistics state with the same number of elements as the data array. This state array is used for the purpose of updating the variation score.
- The weight of an update is determined by the smoothing shift parameter.

The smoothing change determines how often the average value and variance estimates are modified. The statistics state array is initialized with a zero smoothing shift, and the state array is updated with a positive smoothing shift. The update is slower for a larger smoothing shift, and the last sweep is measured against a longer background of sweep data, while a smaller smoothing shift results in faster dynamics, and the detector accepts a new static reflection as baseline after a shorter period.

3.1.3 Convolution

This algorithm is implemented using the triangular smoothing approach to an array of 16-bit complex type data. A running average is applied twice to the data sequence to form the triangular filter. The length of the running average is given by the "side_length" parameter. The output y at index n becomes: y[n] = (s * x[n] + sum_k = 1, ..., s - 1 (s - k) * (x[n - k] + x[n + k])) / s** 2, where x is the input data and s is the "side_length" parameter. The input data is padded with "pad_before" for negative indices and padded with "pad_after" for indices larger than or equal to "data_length". The primary sum is the first running sum before it is divided by the number of elements in the sum to calculate the first running average. This running average is bu ered in the smoothing bu er array. This array contains just the number of elements needed to form the second running sum. Circular bu ering is used as smoothing bu er and an element is overwritten as soon as it is no longer needed. The access index of the bu er wraps around to the beginning when it reaches the end of the array. The computations in the triangular smoothing function of the algorithm are split into three for loops to handle the boundary conditions at the start and end of the input data separately from the main computations in the middle of the input data. The parameters used to implement the convolution algorithm are as follows.

- Data array with complex data to be smoothed.
- "data_length" is the number of complex numbers in the data array.
- "smoothing_bu er" array with side_length complex numbers to serve as work memory.
- "side_length" is the length of the running average that is used to form triangular smoothing.
- "pad_before" is the input pad value to be applied before the input data array.
- "pad_after " is the input pad value to be applied after the input data array.

3.2 Profiler

To speed up the processor's execution, custom instructions are designed. The bottlenecks of the algorithms must be closely examined in order to generate custom instructions. On the Ibex processor, the study involves which sections of the algorithms require the most instructions and cycles to execute a particular feature.

As a result, algorithm profiling is needed to identify the critical areas. None of the open source ISSs support profiling, especially profiling at the line-by-line level.

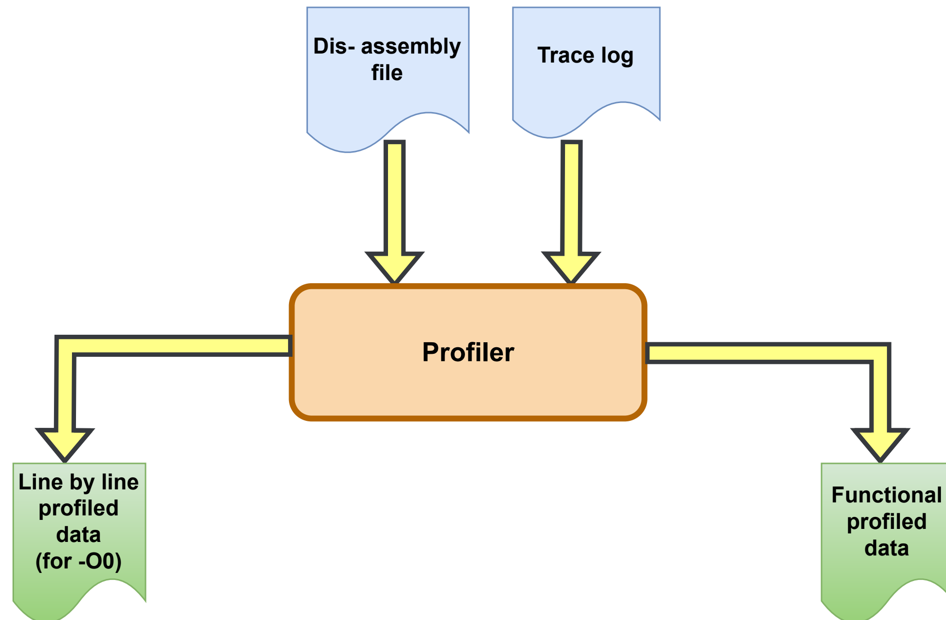


Figure 3.2: Profiler

Figure 3.2 shows the basic structure of the profiler. It is developed from scratch using python, that calculates the number of cycles elapsed in the actual Ibex hardware framework for the algorithm. The profiler can perform both line-by-line and functional profiling of the algorithm. Two files are required to estimate the cycles: one is the dis-assembly file, and the other is the trace log created by the Ibex software model. The object dump command creates a dis-assembly file from an executable(.elf) file. This file includes the C source code as well as the assembly format for it. To get the C source code and the assembly in the dis-assembly file, use the "-S" and "-D" switches with the object dump command. After running the algorithm on the Ibex SoftWare (SW) system, a trace file is created. The PC value/address, executed instruction, cycle number, and so on are all stored in this file.

The line profiler calculates the number of cycles spent on each line of the algorithm, allowing the user to see which areas of the algorithm consume the most and which consume the least. For the optimization stage -O0, line-by-line profiling is performed. Since the data within the dis-assembly file will be shu ed for optimization level -O3, line-by-line profiling will be ine ective. The line profiled data will be saved in a file named "<algo_name>.txt" by the profiler.

Profiling is often performed at the functional level in order to get a detailed understanding of the algorithm. The functional profiling logic implemented in the profiler is used to estimate the cycles spent exclusively and inclusively for each

function in the algorithm. Functional profiling requires the same input files as line-by-line profiling. Functional profiling is also possible at both the -O0 and -O3 compiler optimization levels. The functional profiled data will be saved in a file called "<algo_name>_functional.txt" by the profiler.

The following is a brief description of the steps taken to build the profiler:

- Generate the executable and dis-assembly file by compiling the algorithm using the RISC-V GNU Toolchain with the appropriate optimization level.
- Create the trace log by simulating the executable (.vmem) file created for the algorithm on Ibex.
- Pass the dis-assembly and trace file to the profiler.
- After removing the unwanted data from the dis-assembly file, the profiler will generate a list that only contains the required data such as C source code, PC value/address, and assembly information.
- As a next step, a list with the PC value/address and cycle number from the trace log is generated.
- After that, the assembly list and trace list are processed for line profiling and functional profiling.

Figure 3.3 shows the output sample of a line profiler. The first column gives the details of the functional flow, second column provides the corresponding C source code and the last column shows the number of cycles elapsed for that particular line.

Profiler data Created on -- 13/05/2021 09:52:48

Function Flow	C source code	Number of Cycles elapsed
main	int16_t i16_exponent = SIGNAL_I16_EXPONENT; // The signal, signal_i16, is generated with this exponent	3
main-->fft_prescale_i16_complex	const uint16_t n = 1 << length_shift;	6
main-->fft_prescale_i16_complex	uint32_t r2_max = 0;	2
main-->fft_prescale_i16_complex	for (uint16_t i = 0; i < n; i++) {	12
main-->fft_prescale_i16_complex	int32_t real = data[i].real;	10
main-->fft_prescale_i16_complex	int32_t imag = data[i].imag;	10
main-->fft_prescale_i16_complex	/*>> uint32_t r2 = (uint32_t)(real * real) + (uint32_t)(imag * imag); <<*/	7
main-->fft_prescale_i16_complex	if (r2_max < r2) {	5
main-->fft_prescale_i16_complex	r2_max = r2;	4
main-->fft_prescale_i16_complex	for (uint16_t i = 0; i < n; i++) {	12
main-->fft_prescale_i16_complex	int32_t real = data[i].real;	10
main-->fft_prescale_i16_complex	int32_t imag = data[i].imag;	10
main-->fft_prescale_i16_complex	/*>> uint32_t r2 = (uint32_t)(real * real) + (uint32_t)(imag * imag); <<*/	7
main-->fft_prescale_i16_complex	if (r2_max < r2) {	7
main-->fft_prescale_i16_complex	for (uint16_t i = 0; i < n; i++) {	12

Figure 3.3: Line profiler sample output

Figure 3.4 shows the output sample of functional profiled data. First column shows the function name, second column shows the cycles elapsed exclusively for that particular function and the last column shows the total cycles elapsed inclusively for that function.

Functional Profiler data Created on -- 13/05/2021 09:52:48

Function	Exclusive Cycles	Inclusive Cycles
main	52	61058
fft_prescale_i16_complex	3016	3016
fft_i16_complex	35480	57990
downscale_select_i16_complex	9070	9070
mul_i16_complex	13440	13440

Figure 3.4: Functional profiler sample output

3.3 Algorithm analysis using profiler

Profiler is used to analyse the bottlenecks of the three algorithms in terms of the number of cycles it consumes in each line of source code. Analysis example for one bottleneck in each algorithm is explained below.

3.3.1 FFT

Source code

```
int32_t real = (int32_t)a.real   (int32_t)b.real − (int32_t)a.
   imag   (int32_t)b.imag
```

Bottlenecks:

i This source code calculates the real part of the complex multiplication.

ii We discovered that a considerable number of cycles had elapsed for this particular source code after evaluating the results given by the line-profiler, most likely because this code belongs to the complex arithmetic function that is utilized most frequently in the algorithm.

iii This source code also requires a lot of instructions to obtain the outcome we see in the dis-assembly, thus the instruction count was too high, which was one of the bottlenecks that prompted us to seek for ways to improve it.

iv From the profiler results in table 5.4 this operation takes 19 cycles at a time by using 10 instructions before implementing custom instruction.

Potential improvements:

i Reduce the number of instructions by implementing the custom instruction, resulting in a lower cycle count when the total FFT algorithm is executed.

ii Since the source code uses two 16-bit data packed into a 32-bit register, one possible enhancement strategy is to employ SIMD, which takes advantage of the parallel operations between the compositely packed data.

3.3.2 Variance

Source code

```
if (sample >= average) {
abs_diff = (uint32_t)sample − (uint32_t)average;
} else {
abs_diff = (uint32_t)average − (uint32_t)sample;
}
```

Bottlenecks:

i This source code contains conditional branch instructions, which are implemented in the "statistics update" function, which is used to update the variation score in each iteration, resulting in a high cycle count when the entire algorithm is run.

ii The dis-assembly file revealed an instruction count of 12.

iii Profiler results in table 5.4 shows this operation takes 16 cycles to execute without the support of custom instruction.

Potential improvements:

- i Implementing custom instruction to execute the same operation with just one instruction, can reduce the total number of instructions.
- ii Reusing the existing functionalities and operators in hardware to implement the custom instruction can reduce the area.

3.3.3 Convolution

Source code

```
real = (value.real   factor + 0x8000) >> 16
```

Bottlenecks:

- i This source code is part of the convolution algorithm and is used to calculate the real part of a number in a scaling function.
- ii As a result, a considerable number of cycles are elapsed when the scaling function is employed for triangle smoothing implementation on an array of 16-bit complex type data.
- iii The number of instructions required to get the desired result is quite significant.
- iv Profiler results in table 5.4 show this operation takes 9 cycles to execute by using 6 instructions with normal instructions.

Potential improvements:

- i Reduce the number of instructions by replacing them with a single instruction that performs the same operation.
- ii Possibility of saving hardware resources by reusing them.

Chapter 4

SDK Setup and Custom Instructions Implementation

4.1 SDK Setup

4.1.1 Toolchain

The RISC-V GNU Compiler toolchain consists of cross-compilers for RISC-V C and C++. They have two di erent build modes.

1. ELF/Newlib toolchain (Generic)
2. Linux-ELF/glibc toolchain is sophisticated.

A riscv32-elf toolchain, which uses Newlib, is designed for embedded work. A riscv32-linux toolchain uses glibc and is intended for linux work. The embedded ELF toolchain is more straightforward to set up and use. To create linux applications or programs that use linux system calls, the linux toolchain is needed. The embedded ELF toolchain is used to compile small test programs or embedded applications. However, despite the availability of a Newlib port for RV32E, there is no linux glibc port for it. The ELF/Newlib toolchain was chosen for the thesis since it allows both IMC and EMC configurations.

4.1.2 ISS

An ISS is a simulation model that mimics the behavior of a mainframe or microprocessor by "reading" instructions and maintaining internal state that represents the processor's registers. It is normally written in a high-level programming language. An ISS is also provided with debugger in order to debug the algorithms before we proceed with the porting of algorithms in the target hardware.
Spike, riscv-ovpsim, Imperas Professional, Whisper and sail-riscv are the ISS's avaialble for RISC-V ISA. First three of these have been evaluated for the thesis work because of the proper guidelines and support.

The selection criteria for the ISS are shown in the table 4.1.

Table 4.1: ISS Selection Matrix

ISS	Custom instruction support	EMC support	Simulation cycles	Profiling	Open-source
Spike	✓	X	✓	X	✓
riscv-ovpsim	X	X	X	X	✓
Imperas Professional	✓	✓	✓	✓	X

Spike was chosen as the ideal ISS for the thesis work since it is open-source, and custom instruction support can be added to it. Spike, the RISC-V ISA simulator, implements the functional model of one or more RISC-V harts.

4.2 Custom Instruction Implementation Support

Custom instruction support is divided into two categories software support and hardware support.

Software support is further divided into 3 sub-categories:

(i) Adding support to the RISC-V GNU toolchain by updating the instructions in "riscv-binutils" and "riscv-gdb". Binutils is a collection of programming tools for creating and managing binary programs, object files, libraries, profile data, and assembly source code while GNU Debugger (GDB) is the debugging tool that comes with the GNU Toolchain. Debugger helps us to step through source code line-by-line and debug the issues.

(ii) Adding instruction support to the Spike.

(iii) Adding custom instructions to the algorithms with the help of inline assembly which is basically embedding the assembly instructions in the C program. This thesis demanded addition of inline assembly to the algorithms in both toolchain and Ibex software model.

After the software support has been given, we change the current hardware architecture to add custom instruction support to the hardware, which is the Ibex CPU core we are using.

4.2.1 Toolchain Support

To add custom instruction support to the GNU toolchain modification of three files are needed. In the "riscv-binutils" and "riscv-gdb" of the "ricv-gnu-toolchain" repository. The files and their directory structure inside riscv-binutils and riscv-gdb are shown in figure 4.1 and figure 4.2 respectively.

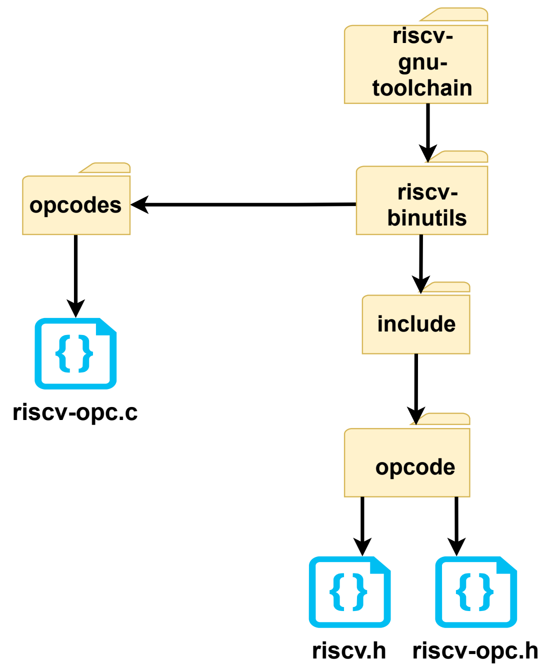


Figure 4.1: Binutils directory tree structure

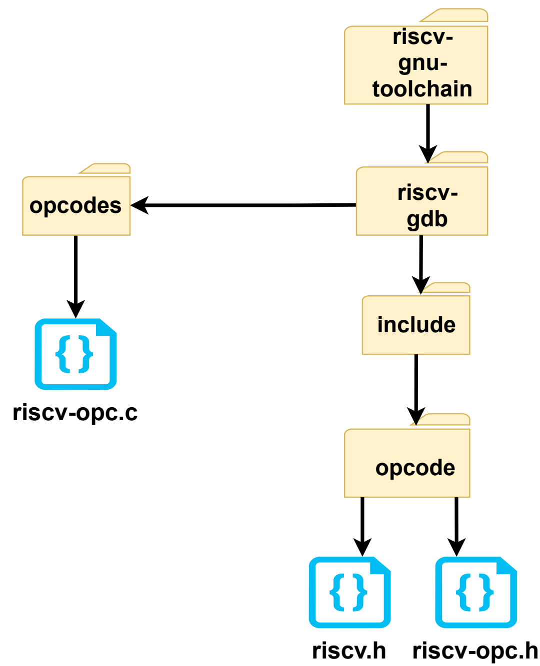


Figure 4.2: GDB directory tree structure

As a first step in adding custom instruction support to the toolchain, an opcode is created for each custom instruction in the "riscv-opc.h" file, which is present in both riscv-binutils and riscv-gbd with the appropriate mask and match bit. Following the generation of the opcode, "DECLARE INSN" is used to declare the newly created instruction on the same file. A sample of the contents inside "riscv-opc.h" is shown in figure 4.3.

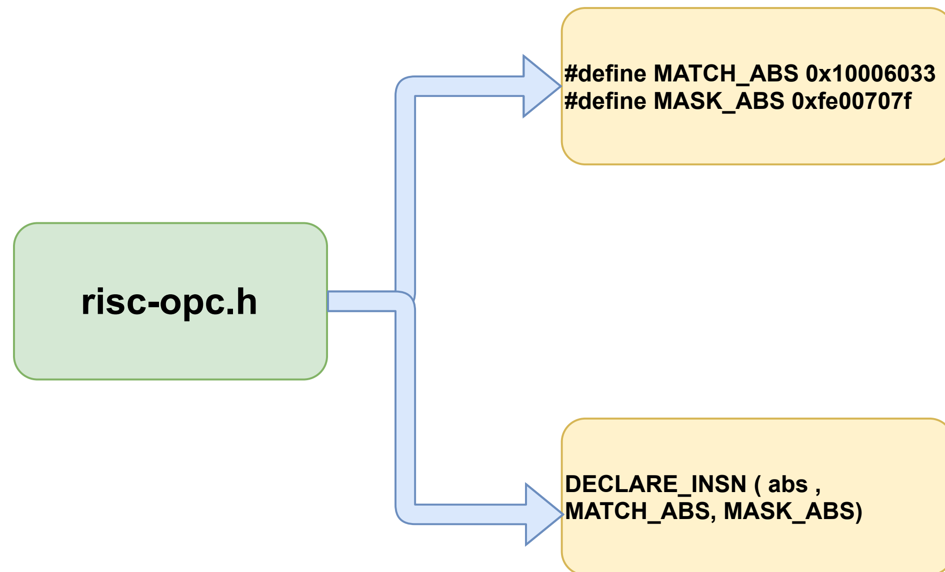


Figure 4.3: Sample of riscv-opc.h contents

The riscv opcode structure in "riscv-opc.c" holds all of the instructions available in the RISC-V ISA, and the custom instruction is defined in a very similar fashion. The structure of an instruction is described below along with a description of each field in table 4.2.

```
Const struct riscv_opcode riscv_opcodes[]=
{ ...
{"abs", 0, INSN_CLASS_I, "d,s,t", MATCH_ABS, MASK_ABS,
    match_opcode, 0 },
```

Table 4.2: Instruction format

Field	Description
0	xlen: Indicates Base ISA required for this instruction. 32 for (RV32), 64 for (RV64) and 0 for both.
INSN_CLASS_I	Instruction class. These are defined in: include/opcode/riscv.h enum_riscv_insn_class . . . . INSN_CLASS_I;
"d,s,t"	Instruction operands
MATCH_ABS	Instruction's primary opcode. This opcode is changed by arguments during assembly to generate the actual opcodes that are used.
MASK_ABS	Bit mask for the appropriate portions of the opcode when disassembling if pinfo is not INSN MACRO. Right instruction if the actual opcode anded with the match field equals the opcode field. This field is the macro identifier if pinfo is INSN MACRO.
match_opcode	A function to determine if a word corresponds to this instruction. It computes logic for match and mask as: (insn_encoding & mask == match)
0	Pinfo: This is INSN MACRO for a macro. Aside from that, it is just a set of bits that describe the instruction, including any applicable hazard information.

If the custom instruction belongs to the existing instruction class or already has the instruction encoding bit length set, then there is no need to modify else we need to make modifications in the "riscv.h" header file which also contains the instruction format as mentioned in 4.2.

After the addition of custom instruction support to the toolchain, the instruction was implemented in the algorithm using inline assembly. The algorithm was compiled and an object dump on the executable file was conducted to ensure that the custom instruction appeared in the dis-assembly file, suggesting that the instruction support had been properly implemented. Below is the sample of an inline assembly implementation of the "abs" custom instruction.

```
asm volatile
    (
    "abs   %[z], %[x], %[y]\n\t" // performs the absolute
     difference of 2 signed 32-bit values.
    : [z] "=r" (abs_diff)
    : [x] "r" (sample), [y] "r" (average)
    );
```

Where, z is an output register for the result of the "abs" instruction, which is "abs di ," and x and y are input registers for the input parameters "sample" and "average". Here "r" is used to indicate that the x, y, and z are used as registers in the inline assembly. The output register z is marked by the symbol "=r," where the "=" sign indicates that the register can only be read once a value has been set to it, whereas "+r" indicates that the output register can be read and written. Source code is compiled and an object dump is generated to verify that the custom instruction is present in the dis-assembly file.

4.2.2 ISS Support

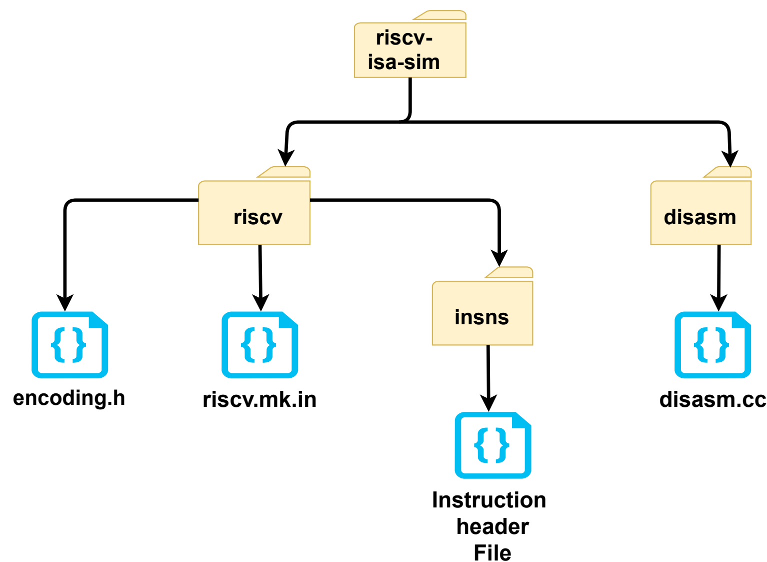


Figure 4.4: Spike directory tree structure

Figure 4.4 shows the directory structure of the files which need modification for adding custom instruction support in Spike. Custom instruction support begins with the addition of an appropriate mask and match bit to produce an opcode, as well as the declaration of each custom instruction using "DECLARE INSN" in the "encoding.h" file. Following this, custom instruction functionality must be developed in the form of an instruction header file and stored in the "insns" folder, which also contains the header file for all other instruction functions, so that Spike can recognize the operation performed by the custom instruction. For the

instruction to be decoded, the type of instruction must be stated in the "disasm.cc" file. Finally, we must add the instruction extension or instruction class to the "riscv.mk.in" file, which is basically a makefile for building support for certain classes of instructions in Spike. After the support has been properly implemented, the executable file generated by compiling the algorithm in the toolchain was run with the Spike executable command to ensure that the algorithm's output was correct.

4.2.3 Hardware Support

In order to implement the custom instruction "abs" functionality in the Ibex hardware three design files need to be modified. The directory structure of the files is shown in figure 4.5

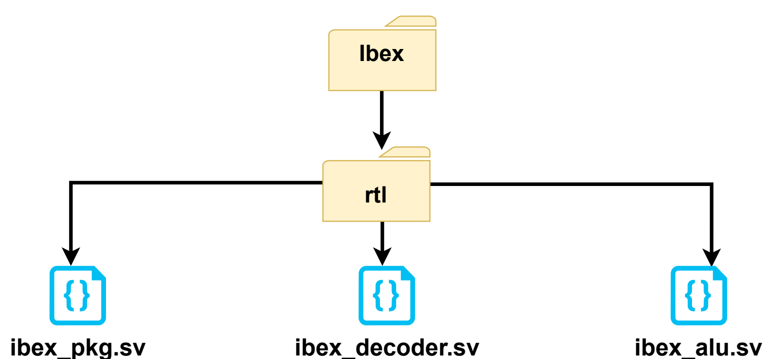


Figure 4.5: Ibex RTL directory structure

- First, "ibex_pkg.sv" file has been modified. The opcodes are defined under the "opcode_e" which is of type "enum" and the custom instructions are declared under "alu_op_e" which is also of type "enum".
- Second, "ibex_decoder.sv" file has been modified in two places: 1. "always_comb" block of the "Decoder" section 2. "always_comb" block of the "Decoder for ALU section".
- Finally "ibex_alu.sv " file has been modified to implement the functionality or the working of the instruction with hardware resources.

4.2.4 Implemented Custom Instructions

Custom instructions are categorized into 2 types.

1. SIMD - SIMD is an acronym for Single Input Multiple Data. SIMD operations allow multiple data to be processed with a single instruction. This type of instruction works with packed data and allows parallel operations to be performed on the packed data. This type of instruction is well-known for taking use of data-level parallelism.

2. Non-SIMD - Non-SIMD which is also a normal instruction and is not concerned with packed data.

Table 4.3 shows the 11 custom instructions implemented in Ibex and their operation.

Table 4.3: Custom Instructions Implemented

Instruction	Description	Algorithm	Category
abs	Absolute di erence between two numbers	Variance	Non-SIMD
rsra	Rounding after right shift operation	Variance	Non-SIMD
srsub	Subtraction followed by arithmetic right shift	Variance	Non-SIMD
sradd	Addition followed by arithmetic right shift	Variance	Non-SIMD
smdrs	Computes real part of complex multiplication	FFT	SIMD
kmxda	Computes imaginary part of complex multiplication	FFT	SIMD
sadd	Shift with rounding on complex number	FFT	Non-SIMD
kmda	Computes complex absolute square	FFT	SIMD
rsub16	16-bit signed parallel subtraction with 1 step right shift	FFT	SIMD
radd16	16-bit signed parallel addition with 1 step right shift	FFT	SIMD
mas	Multiplication and addition followed by right shift	Convolution	Non-SIMD

The custom instructions listed in the table 4.3 were built while keeping the algorithm bottlenecks in mind, as described 3.3. The instructions were implemented in the reference algorithms through inline assembly, as previously discussed in the toolchain support section and then the algorithms were made to execute on the core after providing suitable hardware support for the custom instructions.
Potential improvements for the bottlenecks which has been found out in 3.3 is implemented in hardware. The functionality of the custom instructions are implemented in ALU. Figures 4.6, 4.7 and 4.8 depict the hardware implementation of only three custom instructions, which are used in the algorithms listed in table 4.3. They are implemented to address the bottlenecks mentioned in 3.3.1, 3.3.2 and 3.3.3 respectively.

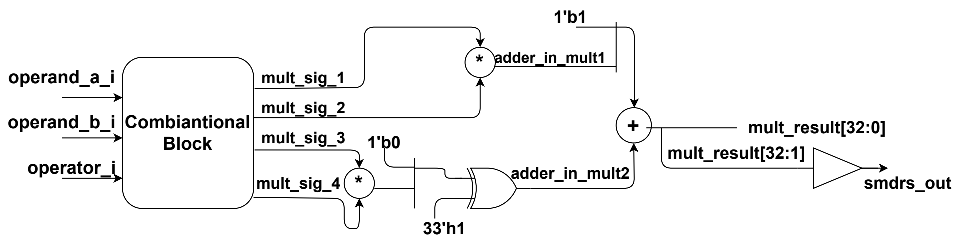


Figure 4.6: SMDRS custom instruction logic in ALU

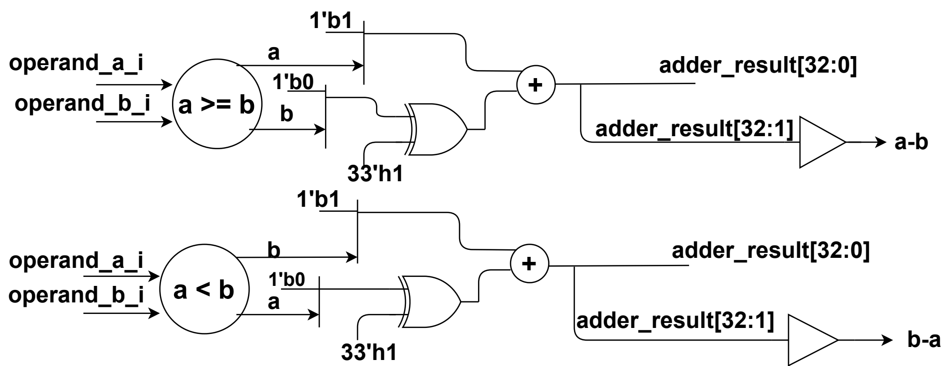


Figure 4.7: ABS custom instruction logic in ALU

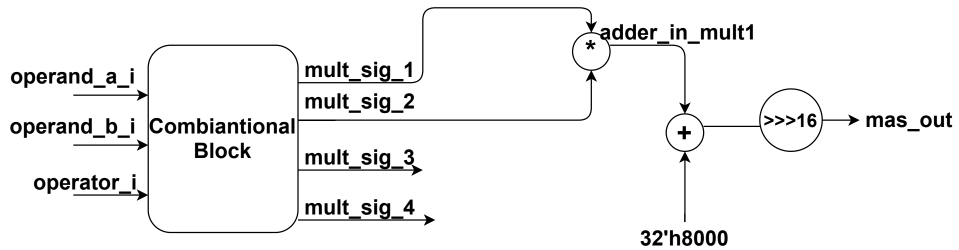


Figure 4.8: MAS custom instruction logic in ALU

Custom instructions were originally implemented to improve performance by reducing execution time, but when we implemented the functionality in hardware, we discovered that the instructions used a lot of extra logic, which introduced a lot of extra hardware resources, resulting in a huge increase in area, as indicated by the initial synthesis results. Following the examination of preliminary synthesis results, we attempted to build the functionality for the custom instructions listed in the table using currently available hardware resources. Following the hardware reuse methodology, we ran synthesis again, and the results showed that the area had been greatly optimized while still maintaining better performance. Figure 4.9 shows the initial hardware implementation of the three custom instructions SMDRS, KMXDA and KMDA which use dedicated hardware resources to perform the respective operations. Applying hardware reuse we were able to reduce the number of multipliers from 6 to 2 and the number of adders to 4 to 1 which can be reused for other instructions of type SIMD such as KMXDA and KMDA with the same operators as shown in figure 4.6. The MAS instruction showed in figure 4.8 also uses the same multiplier units.

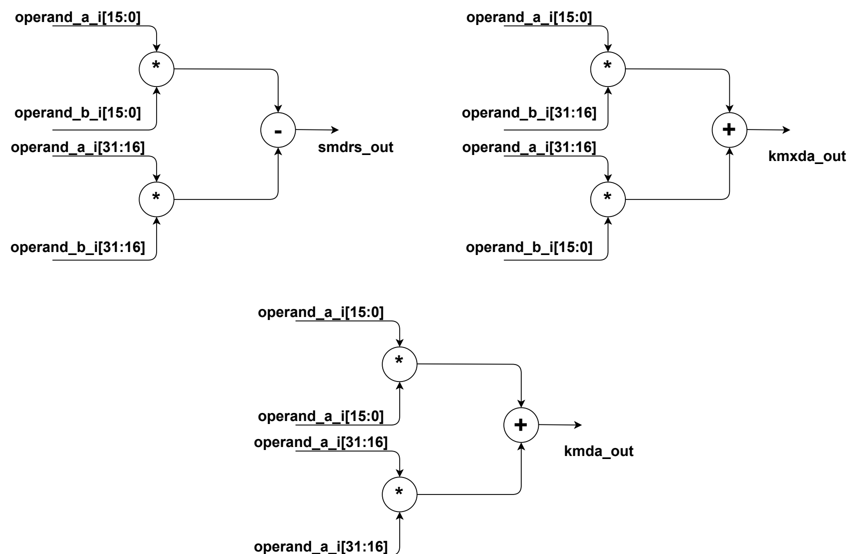


Figure 4.9: Initial hardware architecture

Chapter 5

Simulation and Synthesis

5.1 Configurations used for Simulation and Synthesis

Simulation and synthesis is carried out for 32 di erent configurations of Ibex including IMC/EMC, custom instruction, branch target ALU and write back support. Table 5.1 and 5.2 shows the ibex configuration nomenclature formation. From these tables 32 configurations have been formulated.

Table 5.1: Ibex prefix nomenclature formation

Configuration base name	ISA	Multiplier	Custom instructions
rv32imc_mfast_noci	IMC	mfast	X
rv32imc_mfast_ci	IMC	mfast	✓
rv32imc_msingle_noci	IMC	msingle	X
rv32imc_msingle_ci	IMC	msingle	✓
rv32emc_mfast_noci	EMC	mfast	X
rv32emc_mfast_ci	EMC	mfast	✓
rv32emc_msingle_noci	EMC	msingle	X
rv32emc_msingle_ci	EMC	msingle	✓

Table 5.2: Ibex su x nomenclature formation

Configuration su x	Branch Target	Writeback
nobt_nowb	X	X
bt_nowb	✓	X
nobt_wb	X	✓
bt_wb	✓	✓

5.2 Simulation

For simulation of algorithms with Ibex, two EDA tools are used.

1. An open-source tool which uses FuseSoC as a hardware package manager.
2. A commercial EDA tool.

Simulation process usually follows the steps below

1. Build the simulation model using FuseSoC when using an open source simulator. For commercial EDA tool this step is excluded.
2. Generate the Memory (vmem) file or elf file to load in the Ibex memory model, by compiling the algorithm using the RISC-V toolchain.
3. After building the simulator and software feed the elf file or the vmem file to the memory model already built and run the simulation.

Ibex was used to simulate the reference algorithms using IMC and EMC ISA. These algorithms are compiled with -O3 optimization, which is the highest level of compiler optimization, prior to simulation. "-O3" optimization was performed to see if, once the compiler has completed its highest level of optimization, customization, or in other words, custom instructions, may be used to boost performance even more. The simulation results of the reference algorithms with a comparison of IMC and EMC settings, is shown in the table 5.3. There is an increase in the number of simulation cycles for EMC when compared to IMC ISA. This is due to the EMC architecture's register configuration. In general, when the number of registers increases or decreases, the number of execution cycles reduces or increases, suggesting an increase or reduction in speed e ciency.

Table 5.3: IMC and EMC simulation cycle comparison

Algorithm	IMC/EMC	Simulation cycles		% Reduction
		Pre-custom instruction implementation	post-custom instruction implementation	
FFT	IMC	18887	16328	13.54
	EMC	23093	18230	21.05
Variance	IMC	2790587	2276744	18.41
	EMC	2804842	2378142	15.21
Convolution	IMC	140505	121428	13.57
	EMC	201674	173546	13.94

Table 5.4 shows the instruction count and simulation cycles for each functionality without custom instruction and with custom instruction. These data have been collected from the profiler and dis-assembly file with -O0 compiler optimisation.

Table 5.4: Instruction and cycles count for custom instructions

Functionality	Pre-custom instruction implementation		Post-custom instruction implementation		
	Number of instructions	Number of cycles	Instruction	Number of instructions	Number of cycles
Absolute di erence between two numbers	12	16	abs	4	6
Rounding after right shift operation	9	15	rsra	6	11
Subtraction followed by arithmetic right shift	6	10	srsub	4	7
Addition followed by arithmetic right shift	6	10	sradd	5	9
Computes real part of complex multiplication	10	19	smdrs	4	7
Computes imaginary part of complex multiplication	10	19	kmxda	4	7
Shift with rounding on complex number	5	7	sadd	4	6
Computes complex absolute square	7	14	kmda	4	7
16-bit signed parallel subtraction with 1 step right shift	14	21	rsub16	13	20
16-bit signed parallel additionwith 1 step right shift	14	21	radd16	13	20
Multiplication and addition followed by right shift	6	9	mas	4	7

[1] For post-custom instruction implementation the number of instructions include the surrounding instructions relevant to the execution of the custom instruction.

5.3 Synthesis

The Ibex core was synthesized with the aid of the open-source standard cell library Nangate45. This library is provided primarily for testing, research projects, and the exploration of various EDA flows, and it is non-manufacturable. A commercial synthesis tool was used for the thesis work.

5.3.1 Yosys Synthesis Flow

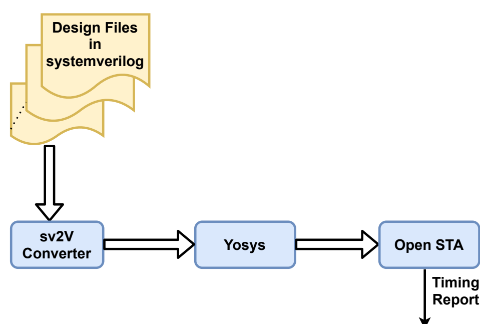


Figure 5.1: Yosys synthesis flow

Ibex's current configuration uses the open-source tool Yosys for synthesis and the open-source tool OpenSTA for producing timing reports. The model for synthesis using Yosys is shown in figure 5.1. Ibex employs SystemVerilog language for the design work. Yosys does not accept any of the SystemVerilog constructs that were available in Ibex. To convert design files from SystemVerilog to Verilog, an open-source converter called "sv2v" is used. With an emphasis on synthesizable language constructs, the sv2v converter converts SystemVerilog (IEEE 1800-2017) to Verilog (IEEE 1364-2005). Using the sv2v converter, the SystemVerilog Ibex design files are translated to Verilog, and the resulting Verilog design files are parsed into Yosys, which synthesizes the design by mapping it into the corresponding standard cell library. The resulting netlist is then parsed into the open-source timing analysis tool "OpenSTA." OpenSTA is a gate-level static timing verifier. It can be used as a stand-alone executable to verify a specification's timing. OpenSTA generates timing reports, and the netlist provided by Yosys generates an area use report. The Ibex synthesis results using the Yosys flow are shown in Table 2.3.

5.3.2 Synthesis Using Commercial tool

Synthesis has been performed for 32 di erent Ibex configurations formulated from tables 5.1 and 5.2.

5.3.2.1 Baseline and custom configuration comparison

The table 5.5 compares the baseline and custom configurations in terms of area and maximum frequency. The baseline configuration is "rv32imc_mfast_noci_nobt_nowb" and the custom configuration is "rv32imc_mfast_ci_nobt_nowb". Because of the inclusion of custom instruction logic on Ibex core, the custom configuration has considerably more area than the baseline configuration. As compared to the baseline configuration, the custom instruction configuration takes up 21.63 percent more space.

Table 5.5: Area and maximum frequency comparison

Configuration	Synthesis data		
	Area(m^2)	Time Period(ps)	Max. Frequency(MHz)
rv32imc_mfast_noci_nobt_nowb	21901.38	10100	99
rv32imc_mfast_ci_nobt_nowb	26639.37	10100	99

5.3.2.2 Execution Time and Area

Figure 5.2, 5.3 and 5.4 show the relation between execution time and area of FFT, variance and convolution algorithm.
While the figures for area based on synthesis results, the execution time is calculated as the product of the time-period used in synthesis and the cycles elapsed during simulation for the corresponding configuration and algorithm. Execution time is given by equation 5.3.2.1.

$$Execution\ Time = Time\ period_{(syn)} \quad Execution\ cycles_{(sim)} \qquad (5.3.2.1)$$

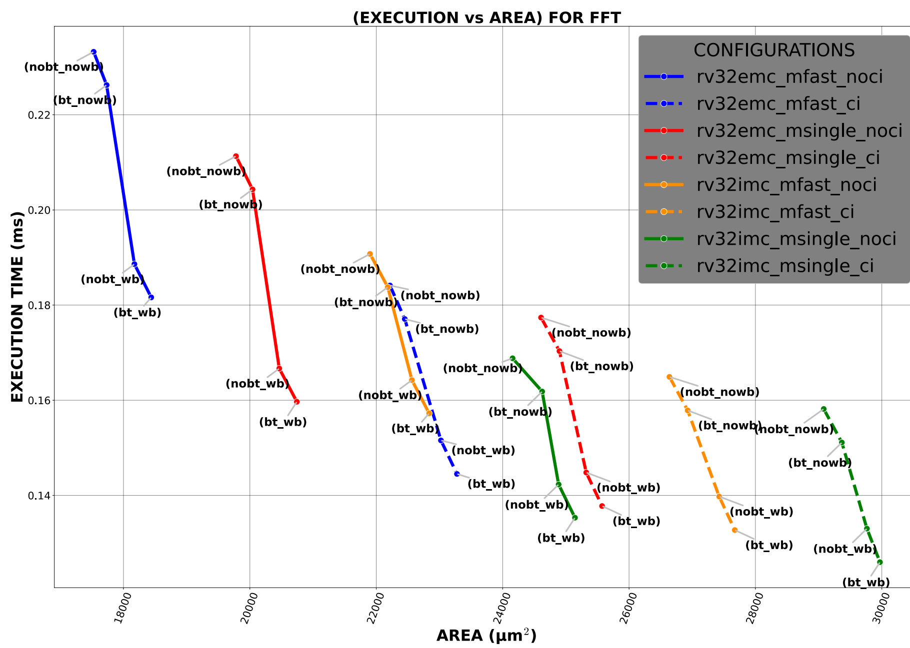


Figure 5.2: Execution time and area for FFT

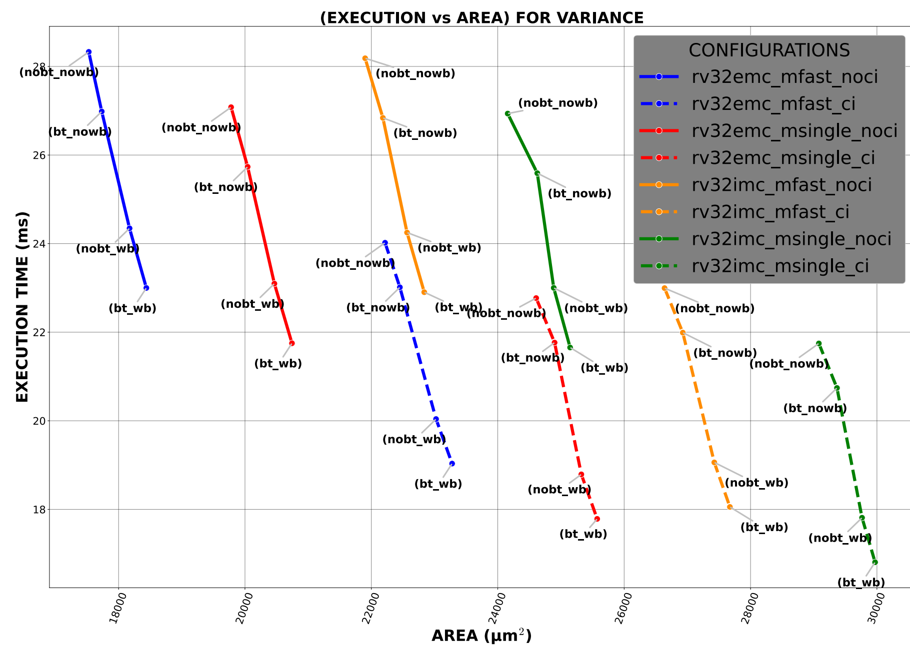


Figure 5.3: Execution time and area for variance

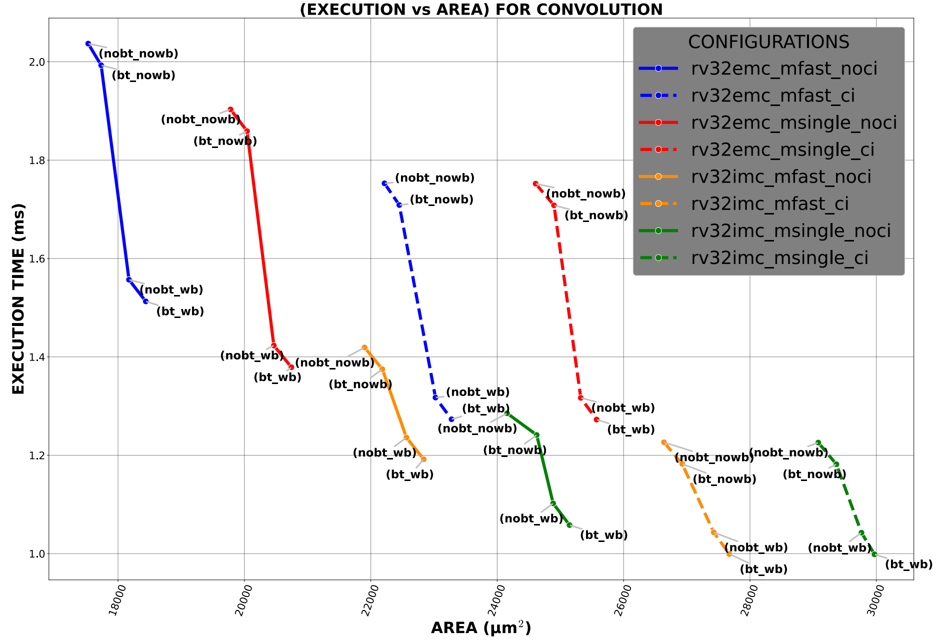


Figure 5.4: Execution time and area for convolution

According to the graphs, the EMC configuration with fast multiplier and disabled custom instruction, branch target, and write back support takes up less space, while the IMC configuration with single multiplier along with custom instruction, branch target, and writeback support takes up the most area. On the other hand, IMC configuration with single multiplier along with custom instruction, branch target, and writeback support has the fastest execution time, while the EMC configuration with fast multiplier and disabled custom instruction, branch target, and writeback has the slowest execution time. This pattern shows that as the area increases, the execution time decreases. The behavior is the same for all three reference algorithms.
The execution time for algorithms with baseline and custom configuration with "mfast" multiplier configuration is shown in table 5.6. After introducing custom instructions, FFT execution time decreased by 15.78%, while variance and convolution execution times decreased by 18.41% and 13.47%, respectively.

Table 5.6: Execution time for baseline and custom configuration

Algorithm	Configuration	Execution Time(ms)
FFT	rv32imc_mfast_noci_nobt_nowb	0.19
	rv32imc_mfast_ci_nobt_nowb	0.16
Variance	rv32imc_mfast_noci_nobt_nowb	28.18
	rv32imc_mfast_ci_nobt_nowb	22.99
Convolution	rv32imc_mfast_noci_nobt_nowb	1.41
	rv32imc_mfast_ci_nobt_nowb	1.22

5.3.2.3 Power consumption

Power in semiconductor devices is mainly classified into two categories.

1. Leakage power
2. Dynamic power

Leakage Power

The pattern between leakage power and area is shown in figure 5.5. The relationship between leakage power and area is linear, which means that as the area grows, so does the leakage power. With fast multiplier and custom instruction, branch target, and write back support disabled, the EMC configuration has less leakage power and takes up less area. IMC configuration with single multiplier along with custom instruction, branch target and writeback support enabled consumes more leakage power among all 32 configurations due to more area. The leakage power(static power) dissipated by a transistor is calculated as follows:

$$P_{leakage} = V_{DD} \quad I_{leakage} \tag{5.3.2.2}$$

where, V_{DD} is the supply voltage and $I_{leakage}$ is the current that flows in transistors in the absence of switching activity.

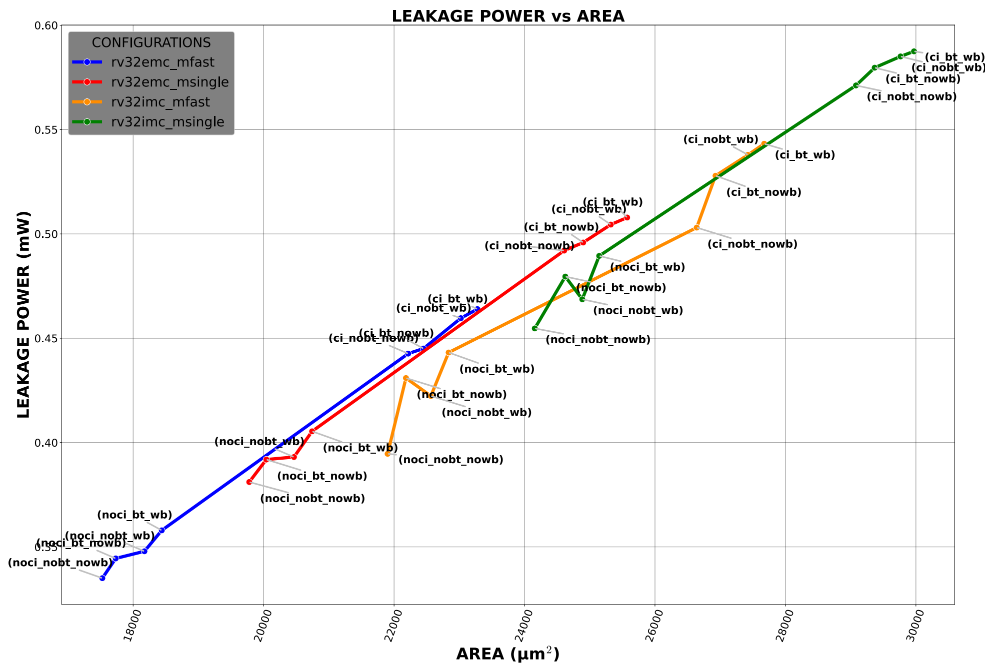


Figure 5.5: Leakage power in Ibex

Table 5.7 shows the leakage power and area consumption for baseline and custom configurations for "mfast" multiplier configuration. There is a 27.66% increase

in leakage power from baseline to custom configuration.

Table 5.7: Leakage power for baseline and custom configuration

Configuration	Synthesis data	
	Leakage power(mW)	Area (m^2)
rv32imc_mfast_noci_nobt_nowb	0.394	21901.38
rv32imc_mfast_ci_nobt_nowb	0.503	26639.37

Area(m^2)Time Period(ps)Max Frequency(MHz)

Dynamic Power

Sum of switching and short-circuit power determines the dynamic power of a circuit. When charging or dis-charging internal and net capacitances, switching power is dissipated. The power dissipated by short-circuit connection between the supply voltage and the ground at the time the gate switches state is known as short-circuit power. Dynamic Power is given by the equation 5.3.2.3.

$$P_{dyn} = P_{switching} + P_{short\ \ circuit} \tag{5.3.2.3}$$

where, $P_{switching}$ is the switching power and $P_{short\ \ circuit}$ is the short-circuit power.

Switching power is given by the equation 5.3.2.4

$$P_{switching} = \quad f \quad C_{eff} \quad V_{DD}^2 \tag{5.3.2.4}$$

where, is the switching activity, f is the switching frequency, C_{eff} is the e ective capacitance and V_{DD} is the supply voltage.

Short-circuit power is given by the equation 5.3.2.5.

$$P_{short\ \ circuit} = I_{sc} \quad V_{DD} \quad f \tag{5.3.2.5}$$

where, I_{sc} is the short-circuit current during switching, V_{DD} is the supply voltage and f is the switching frequency.

Figure 5.6 shows the dynamic power consumption for the baseline configuration and custom configuration with "mfast" multiplier configuration. Since these two configurations are the only completely tested configurations in the Ibex, dynamic power is only plotted for them. The switching activities of the signals are used to calculate dynamic power. For power analysis, synthesis tool reads a Value Change Dump (VCD) file created from simulation. The knowledge about signal value changes in the design is stored in a VCD file. This file is created by attaching VCD system tasks to a Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL) source file using a simulator. The VCD generation is done with the help of a commercial simulator.

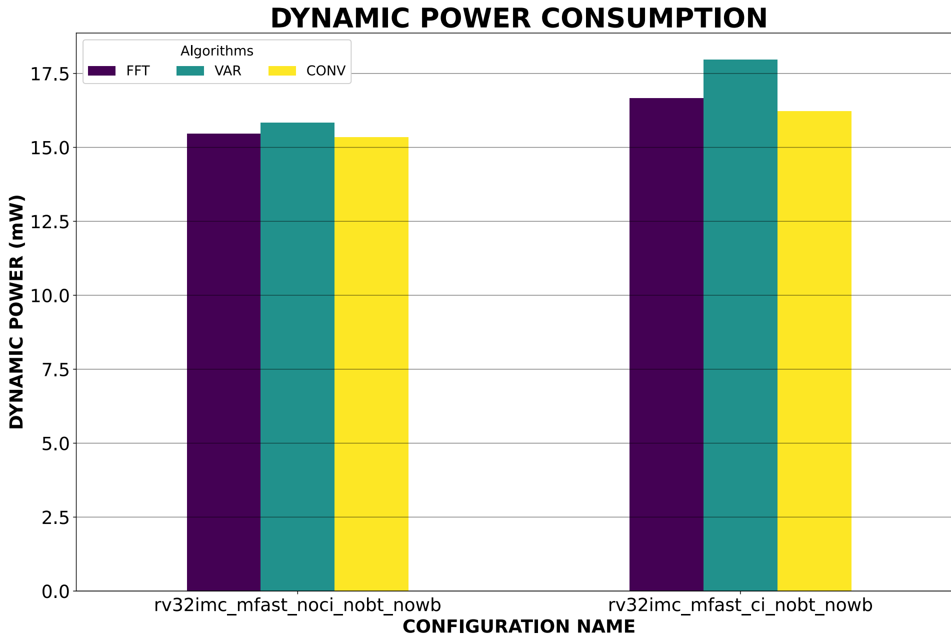


Figure 5.6: Dynamic power in Ibex

Since variance algorithm uses 60000 samples of data, the dynamic power for this algorithm is more when compared to FFT and convolution because it causes more changes in signal values. When comparing baseline and custom configuration, the dynamic power of the custom configuration increased by 9.03% on average.

5.3.2.4 Energy Consumption

Figure 5.7 shows the Energy consumption for the baseline configuration "rv32imc_mfast_noci_nobt_nowb" and "rv32imc_mfast_ci_nobt_nowb" which is of custom configuration. Energy consumption is plotted only for these configurations since these two are the only fully verified configurations. Energy consumption is calculated using the following formula,

$$Total\ Energy = Total\ Power \quad Execution\ time \tag{5.3.2.6}$$

where, Total Power is given by equation 5.3.2.7 and execution time is given by 5.3.2.1.

$$Total\ Power = P_{switching} + P_{short\ \ circuit} + P_{leakage} \tag{5.3.2.7}$$

As compared to FFT and convolution, the variance algorithm uses more energy. This is due to the fact that variance uses 60000 data samples, which means that more variations in signal values equals more dynamic power and energy. When comparing baseline and custom configuration, the custom configuration has an average energy consumption rise of 9.48%.

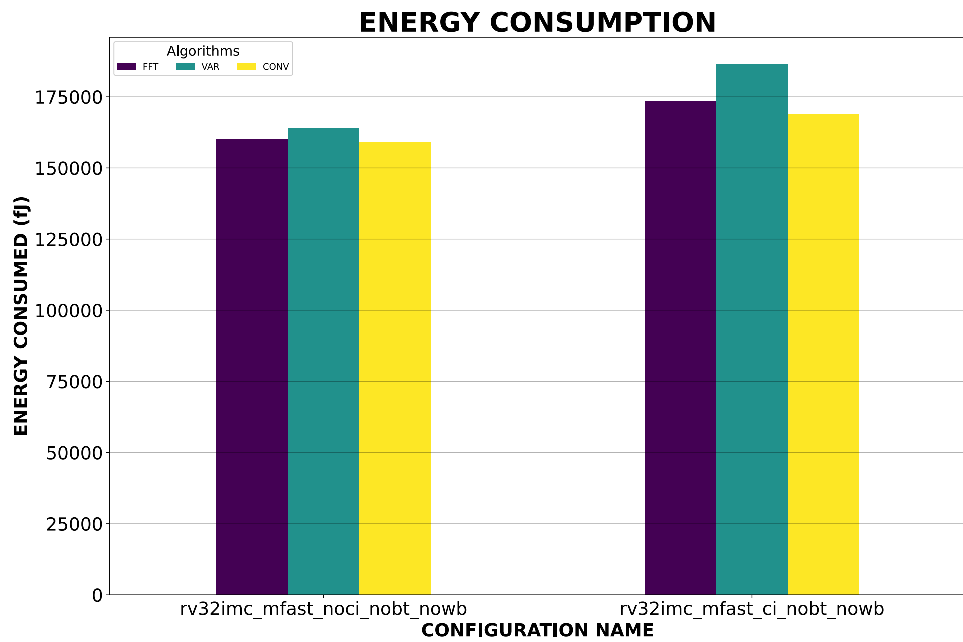


Figure 5.7: Energy consumption in Ibex

5.4 Design Feasibility

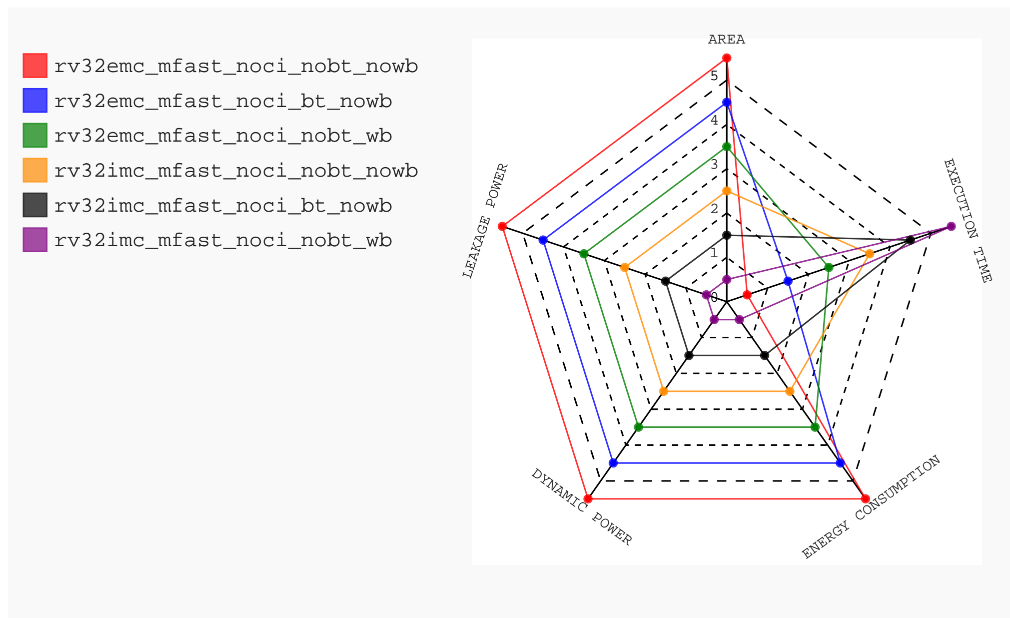


Figure 5.8: Design feasibility

When comparing the synthesis results of 32 different configurations, it was discovered that as area increased, so did power and energy, although execution time decreased significantly. As a result, we came up with six optimal configurations, three for IMC and three for EMC, as shown in figure 5.8 as a radar map.
The five metrics of area, leakage power, dynamic power, energy consumption, and execution time were compared between the setups. After analyzing the data from synthesis, we scaled the five metrics or parameters described previously in the range of (0.5-5.5) since the radar plot was constructed with the purpose of determining whether the design was feasible for embedded applications. For all factors in their respective domains, scale 0.5 is the worst and scale 5.5 is the best. In terms of all the criteria, scale 4.5 is slightly inferior to scale 5.5, while scale 1.5 is slightly superior to scale 0.5, and scale 2.5 and 3.5 is essentially an average. Both the IMC and EMC configurations are scaled in terms of increasing area and decreasing execution time. The first of the six configurations has the smallest area, consumes very low power, and uses extremely low energy, thereby making it ideal for embedded applications. So, we have come to a conclusion that if we want a feasible embedded application setup, we shall have to trade off execution time in exchange for low area and power. However, if the user wants a configuration with better performance, or in other words, a shorter execution time, they must pay a high hardware cost, as well as a high power and energy consumption. Among all six configurations that show gains in terms of area, power, and energy consumption, the radar plot clearly indicates that the configuration with the smallest area consumes the least amount of power and energy, whereas the configuration with the highest area has a shorter execution time or better performance.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We now live in the world of SoCs, where the processor core serves as the SoC's heart. Multiple cores allow for di erent processes to run at the same time, which increases the speed of the system as it enables your computer to perform multiple operations at the same time. With the ongoing advancement of SoCs, the processor core selection criteria has become one of the most essential factors nowadays. When it comes to the key selection criterion, both cost and performance are important considerations. Our thesis focuses on modifying or customizing a core that can be a cheaper option to many other cores presently on the market in order to attain almost the same level of performance as the more expensive processor cores. We were able to modify the Ibex core used in our thesis since it runs on an open-source RISC-V ISA. This allowed us to improve the core's performance. We tried to customize the core by implementing custom instructions in some of the algorithms that are really beneficial in radar signal processing, and then running those algorithms on the core to verify performance after customization. The results achieved strongly suggest that by implementing custom instructions, the bottlenecks observed in the reference algorithms were overcome or in other words we were able to find an open-source core that could still be modified to improve performance based on the requirements with low cost implementation.

6.2 Future work

Finally, our thesis resulted in a finding that leaves opportunity for more research. We examined the performance of the No-inline and Inline implementations for -O0 optimization and discovered that there was no di erence between the two approaches for -O0 optimization since instructions could be properly mapped, or in other words, the inline assembly implementation was exactly mirrored as the No-inline assembly version. However, at -O3 optimization, the highest level of compiler optimization, it was discovered that the No-inline implementation outperformed the Inline implementation due to the compiler optimization's instruction scheduling.

To make the most of instruction scheduling and take advantage of compiler optimization, it is required to replace inline assembly and attempt to o er instruc-

tion support at the compiler level, which necessitates compiler adjustments and hence remains a further scope of research topic.

References

[1] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand and Lucal, *Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications*, 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS),2017

[2] D. Rossi, I. Loi, A. Pullini, and L. Benini, *Ultra-low-power digital architectures for the internet of things*, in Enabling the Internet of Things, Springer International Publishing, pp.69-93, 2017

[3] YM. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gurkaynak, and L. Benini, *Near-threshold RISCv core with DSP extensions for scalable IoT endpoint devices*, IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, pp.1-14, 2017

[4] PULP Team, *PULP Platform*, www.pulp-platform.org

[5] Veripool, *Verilator*, www.veripool.org/verilator

[6] lowrisc Organisation, *lowRISC*, www.lowrisc.org

[7] lowRISC, *Ibex: An embedded 32 bit RISC-V CPU core*, www.ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html

[8] Andrew Waterman, Krste Asanovi, SiFive Inc, *The RISC-V Instruction Set Manual Volume I: User-Level ISA*, CS Division, EECS Department, University of California, Berkeley, 2017

[9] Cli ord Wolf, Johann Glaser, *Yosys Open Synthesis Suite*, www.clifford.at/yosys/

[10] James Cherry, William Scott, *Parallax Static Timing Analyzer*, www.github.com/The-OpenROAD-Project/OpenSTA

[11] The-OpenROAD-Project, *OpenROAD-flow-scripts*, www.github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/tree/master/flow/platforms/nangate45

[12] lowRISC, *Ibex RISC-V Core*, www.github.com/lowRISC/ibex

[13] Wikipedia, *Cooley–Tukey FFT algorithm*, www.en.wikipedia.org/wiki/Cooleyâ Tukey_FFT_algorithm

[14] Aneesh Raveendran, Vinayak Baramu Patil, David Selvakumar and Vivian Desalphine, *A RISC-V Instruction Set Processor-Microarchitecture Design and Analysis*, 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), 2016

[15] RISC-V, *RISC-V GNU Compiler Toolchain*, www.github.com/riscv/riscv-gnu-toolchain

[16] RISC-V, *Spike RISC-V ISA Simulator*, www.github.com/riscv/riscv-isa-sim

[17] Nitish Srivastava, *Adding custom instruction to RISCV ISA and running it on gem5 and spike*, www.nitish2112.github.io/post/adding-instruction-riscv/