RISC-V Processor Core Customization and Verification for m-NLP Applications

Robin A. T. Pedersen

Thesis submitted for the degree of
Master in Electrical Engineering, Informatics and Technology
60 credits

Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

RISC-V Processor Core Customization and Verification for m-NLP Applications

Robin A. T. Pedersen

RISC-V Processor Core Customization and Verification for m-NLP Applications

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This work experimented with using a custom instruction set extension for computing electron density from sensor data received from a multi-needle langmuir probe (m-NLP). Custom instructions were designed and implemented on a RISC-V processor core. The aim being to both speed up the computation and to reduce transmission data, in order to increase spatial resolution and overcome limitations of the communications downlink. The central stategy is to use a small core with specialized acceleration instead of a bigger core with excess features, in order to meet the m-NLP project's potential future of fitting a processor on the same die as other electronics. An attempt was also made to use plain integers for computations, while other theses have used floating point numbers either on an FPU (floating point unit) or using software emulation. Emphasis is placed on verification of the implemented design, using tools like the universal verification methodology (UVM) and SystemVerilog assertions (SVA).

Precision of the results, speed of computation, and size of output data was found to either improve upon or match that of previous work. It was found that the precision of the results are marginally within acceptable limits given the range of electron densities of interest. Precision measurements were done both against real data from an ICI-2 sounding rocket and generated stimuli. By measuring the cycle count of computations, it was possible to compare the speed of different implementation. In this case it was found that with the new modifications, the time to compute electron density is in fact reduced from a pure C language implementation and even more so compared to emulated floating point. The reduction in data transmission achieved by previous theses was maintained. It now requires a 29-bit integer per sampling, instead of 64 bits of raw data, being slightly lower than 32-bit floating point numbers. It was found that even with the timing paths introduced by the new extension, computations can be done on FPGA at 14 times a desired sampling rate of the m-NLP system.

Acknowledgments

Thanks to my three official supervisors Ketil Røed, Mohammed Saifuddin, and Philipp D. Häfliger, for guidance and technical discussions and for helping me organize the work and solving technical challenges. Thanks also, to other staff at UiO for good advice and review; particularly thanks to Girish Aramanekoppa and Candice Quinn for helpful feedback on my writing. Thanks to my fellow students for useful discussions. Thanks to my family for lodgings and advice, and to my girlfriend for care. This work would have been of lesser quality if not for the help of everyone above.

Contents

Nomenclature

This is a list of the most central terms used.

n_e – Electron density
ADC – Analog to digital converter
ALU – Arithmetic logic unit
ASIC – Application specific integrated circuit
ASM – Assembly code
CISC – Complex instruction set computer
CR – Constrained random
DSP – Digital signal processing
DUT – Device under test
EX – Execution stage
FPGA – Field programmable gate array
FPU – Floating point unit
GPR – General purpose register
IC – Integrated circuit
ICI – Investigation of Cusp Irregularities
ID – Instruction decode stage
IF – Instruction fetch stage
IO – Input output
ISA – Instruction set architecture
ISS – Instruction set simulator
LLS – Linear least squares
LSU – Load store unit
LUT – Look up table
PC – Program counter
PMP - Physical memory protection
RISC – Reduced instruction set computer
RISC-V – A RISC instruction set architecture
RTL – Register transfer level
STA – Static timing analysis
SV – SystemVerilog
SVA – SystemVerilog assertions
TLM – Transaction level modeling
UVM – Universal verification methodology
Xmnlp – The new custom RISC-V extension for m-NLP purposes
libc – C standard library
m-NLP – Multi-needle Langmuir probe

Chapter 1

Introduction

This introductory chapter sets the premise. A brief exposition presents the theme, and is followed by an account of the *goals* of the project. Following that is some background theory which is helpful to know about before reading the methods chapter. Lastly is a description of the project plan, which elaborates on the project goals by setting specific requirements.

The overall structure of this document is strictly in the IMRaD format, and its ordering largely follows the chronological progression of the project.

1.1 Exposition

There exists a research initiative, 4DSpace [1], whose goal is to understand plasma instabilities and turbulence in the ionosphere. For that purpose, a multi-needle Langmuir probe (m-NLP) [2] [3], developed by the University of Oslo, is used to study the plasma. This instrument is subject to continued development.

Such a probe has three aspects of relevance for this work: First, spatial resolution is dependent on the frequency of sampling. Second, communication channels have finite bandwidth. Third, an on-board processor coordinates various tasks.

Previous work [4] has experimented with processing the sensor data in-flight, as opposed to transmitting the raw data. And one effort [5] investigated compression methods for rocket telemetry data. Both of these had a goal of reducing the amount of transmitted data. Other work [6] attempted to find processor cores suitable for use in 4DSpace. While *this* work continues in the same spirit, pursuing new ways of improving the in-flight computation.

1.2 Goals

Two more factors are of significance: 1) There is an interest for implementing a *small* processor on the same IC (integrated circuit) as other mission-related electronics [7] [6]; 2) RISC-V has particular support for custom user-defined extensions [8].

Hence, the main goal of this work is to customize a RISC-V processor core with new instructions and assess their utility for m-NLP purposes. A special emphasis will be put on the verification of said customization.

Relating to the exposition above, the main goal can be subdivided into three sub-goals: Firstly, increase the speed of the computation. Secondly, maintain transmitted data size below that of sending raw data. Thirdly, ascertain the suitability of custom RISC-V processors for the m-NLP project.

The original granted thesis proposal was purely about RISC-V verification. For at the time in 2018, the maturity of the RISC-V ecosystem was deemed unsatisfactory. The goal was to study and improve the feasibility of utilizing open source processor designs by giving confidence through verification. The work was to start with UVM (more on that later) and possibly progress onto formal verification. But in January 2019, Google released their UVM-based RISC-V instruction generator [9]. And subsequent investigations revealed more. Notably, in March 2019 the RISC-V foundation opened a review for choosing an official formal specification [10]. It felt like it would not make a significantly meaningful contribution to compete with such professional grade existing solutions.

1.3 RISC-V

> Given that the industry has been revolutionized by open standards and open source software — with networking protocols like TCP/IP and operating systems (OS) like Linux — why is one of the most important interfaces proprietary?
>
> The Case For RISC-V [11]

This section is a primer to RISC-V, seeking to make later parts of the text more understandable.

RISC-V (pronounced *risk five*) is an open source instruction set architecture (ISA), creating a standardized interface for processors. It defines aspects such as: registers which must exist, instructions that are available, encoding of instructions, exceptions, primitive data types, and memory addressing [8]. RISC-V is independent of microarchitecture, is cleanly modularized, and attempts to learn from old mistakes in ISA designs.

Some of the popular ISAs today, like ARM and x86, are not open source, meaning that the author companies hold exclusive rights. In non open source, usage may require expensive licenses and can disallow users from changing the design, which can impede both academic endeavors and market competition. Complete processors does exist under open source licenses, such as OpenSPARC and LEON [12], and OpenRISC, SPARC, and MIPS [11], to name a few. Although RISC-V is permissively licensed and open source, and it is popular with much activity [13].

1.3.1 Basics

RISC-V is a load-store architecture [8]. This means that there is a clean separation between instructions that only operate on memory, and those who do arithmetics. The RISC in RISC-V stands for reduced instruction set computer. This is in contrast to CISC (complex instruction set computer) architectures where one instruction can be further divided into several sub-instructions by the hardware.

There are 31 general purpose registers (GPR), from `x1` up to `x31`. The special purpose register `x0` is always zero. Certain configurations allow for just 16 GPRs, and optionally one can have separate floating point registers.

1.3.2 Modules

RISC-V was created with the idea of modularity in mind [11]. This means that not every RISC-V processor needs to support *all* of the specification. If one is developing a low power embedded device, one may be satisfied with just some basic functionality. General purpose computers also have different needs than research- and supercomputers. Hence, RISC-V was made to have a small base, and have the option to be extended with other modules. [1]

The *base* instruction set is called `I` (I standing for Integer). It contains basic integer operations such as addition, xor, and shift-left; along with load/store operations; and a few other instructions considered to be very minimal while still enough to be usable. There exists variants of the basic integer module, i.e. 32-bit, 64-bit, 128-bit, and an "embedded" smaller variant.

Other modules have more specialized contents[2]. For instance, the `M` module (for Multiplication and division) is not part of the base `I` set, because not all computing platforms need this. Further more, there exists `F` for Floating point, `D` for Double-precision floating point, `A` for Atomic instructions, and so on. Table 1.1 shows the extent of the modules available.

Custom instructions sets are the ones that a core implementor might want to create for themselves. In the notation style used above, these are written as `Xname`, where `name` is the name of the custom extension. In this master's thesis, the aim is to create a custom instruction set extension called `Xmnlp`.

1.3.3 Instruction Formats

As the instruction fetch unit of a processor schedules new instructions to be executed, the rest of a core must deduce the meaning of the instruction. An instruction is a word of data, a bundle of bits, divided into certain bit fields. The so called *instruction formats* are the encoding schemes for how the bundles of bits may be interpreted to meaningful semantics. [3] Figure 1.1 shows the 4 basic instruction formats.

[1] Sometimes "modules" are just referred to as "extensions".

[2] Be aware that not all modules in the specification are finalized and ratified yet.

[3] A RISC-V implementation with a fetch unit is a "core", and one without is a "hart".

Table 1.1: Enumeration of existing RISC-V standard modules.

Base	Extension
RVWMO (Weak memory ordering)	Zifencei
RV32I (32-bit integer)	Zicsr
RV64I (64-bit integer)	M (Mult/Div)
RV32E (Embedded, less GPRs)	A (Atomic)
RV128I (128-bit integer)	F (Floats)
	D (Doubles)
	Q (Quads)
	C (Compressed)
	Ztso
	Counters
	L (Decimals)
	B (Bits)
	J (JIT)
	T (Transactional)
	P (Packed SIMD)
	V (Vectors)
	N (Interrupts)
	Zam

The *opcode* is the main identifying field which the decoder uses to decide which instruction is encountered. But multiple instructions may share the same opcode. For instance, the various load instructions (load byte, load half-word, etc) use the `LOAD` opcode. From that, funct3 and funct7 are used to single in on which one exact instruction one is dealing with.

The other fields are used for arguments to the instruction. `rs1` and `rs2` selects which registers to use as operands, and `imm` caries a number to be immediately used as an operand. Lastly, `rd` is the destination register where results are stored.

1.3.4 Core Implementations

Since the goal is to write a RISC-V extension, a processor core to build upon is necessary. To get the most out of the time available, a sensible choice is to choose an existing open source processor to work on.

Choosing a core is not easy. A core and its surrounding system can be both nuanced and complicated. And by the time one has enough knowledge to make an educated decision, one might already be fairly committed to that one core. The process risks being a shrouded trade-off.

Candidates Some use-cases may have criteria such as low power usage, or support for eccentric RISC-V extensions. But the criteria chosen here are different from that: 1) The time it takes to run a hello world was used as an indication of user-friendliness; 2) Whether the supported toolchain is open source was also a criterion, because there can be problems with using proprietary tools (free license limiting features, not supporting my

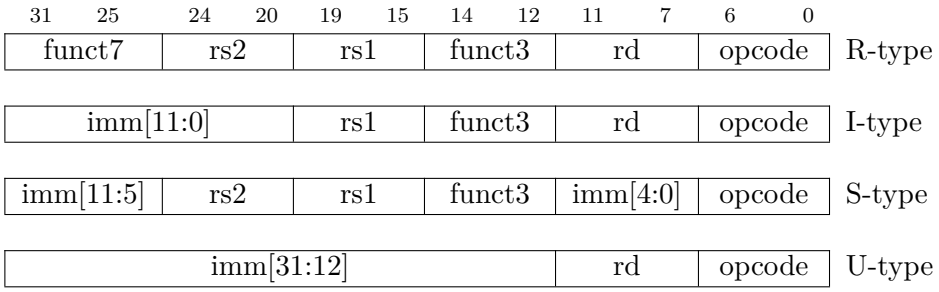


Figure 1.1: RISC-V base instruction formats [8].

computer's OS, etc); 3) I also wanted the freedom to turn off the FPU and other unnecessary features, if present; 4) Acclaim and merits of good performance was heeded as a positive bias in the decision making.

Below is a brief enumeration of the cores evaluated (See [13] for a comprehensive list).

- DIY core - Making an own core from scratch. This was deemed to have too much uncertainty regarding the time necessary to get it working. (More on this in Section 2.3).
- Picorv32[4] - This core was evaluated because 1) it is a very small core (3000 LOC), and 2) I am already familiar with other impressive works of this author. Ultimately, I found it too cumbersome to work with.
- Rocket chip - Made by UC Berkeley Architecture Research. Downside is that it is written in the language Chisel, and I would prefer to keep the overhead minimal.
- PULP Platform - 1) Ariane might be bigger than necessary. 2) RI5CY is a good candidate. 3) Ibex is very similar to RI5CY, and the differentiation is not too significant.
- Freedom - Used in the SiFive SoCs. Also written in Chisel.
- Others - There are many cores available at [13]. Then there are also many not listed at that link.

The core chosen to work on was RI5CY [14][5]. Mainly because I know it so well from working with it before. It is also small, and so it fits the use case. The makers are ETH Zurich and Università di Bologna. It was made for low power, and has been implemented in silicon [15]. Additionally, many options in RI5CY are configurable via parameters in the source code, e.g. the FPU can be left out if so desired.

[4]It is worth checking out the other tools from this author.

[5]RI5CY, originally part of the PULP Platform, was later adopted by The OpenHW Group and renamed CORE-V CV32E40P.

1.4 m-NLP

1.4.1 Langmuir Probe

A significant part of the 4DSpace initiative is the multi-Needle Langmuir probes (m-NLP). They are used to measure plasma electron density.

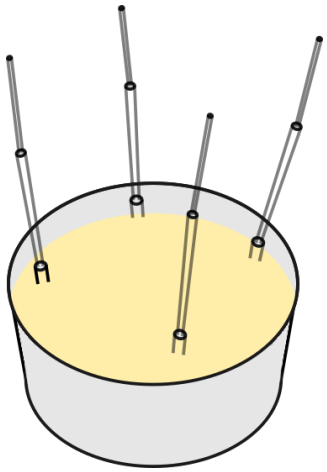


Figure 1.2: Sketch of payload carrying m-NLP probes. [4]

A previous master's thesis by Bekkeng at UiO experimented with prototype development of a multi-needle Langmuir probe in 2009 [16]. And continued development of this system has been the theme of more than one thesis to follow [4] [6]. Use of the instrument sees continued use today [17]. A sketch of a spacecraft payload is shown in Figure 1.2.

Electron Density The instrument measures electron density in the ionosphere [16]. The probes are biased to a reference potential. Current can then arise in accordance with the difference in potential of the probe bias and the surrounding plasma. A typical relationship between the current and voltage in shown in Figure 1.3.

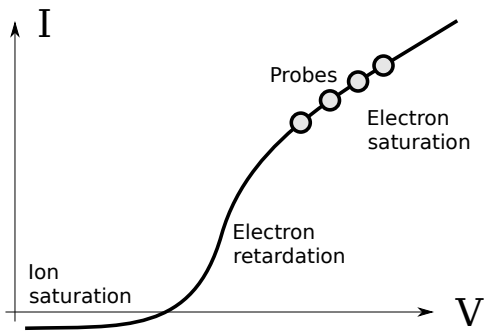


Figure 1.3: I-V characteristics of Langmuir probe in plasma. The measurement points of the four probes are marked as circles. [17]

It can be shown that the electron density relates to the difference between the square of the currents over the difference of bias voltage [2]. This is

shown in Equation 1.1.

$$n_e = \sqrt{C\frac{\Delta(I^2)}{\Delta V}} \tag{1.1}$$

The constant C is given in Equation 1.2. Where m_e is the mass of an electron, q is the charge of an electron, r is the radius of the probe, l is the length of the probe.

$$C = \frac{m_e}{2q(q2rl)^2} \tag{1.2}$$

Note that this only holds under certain conditions [17], but such specifics are outside the scope of this work. For instance is it assumed a "β-factor" of 0.5.

Multi-Needle Using just one probe is possible. But then one will have to sweep across a voltage range. Such a sweep has been found to be too slow [16]. As the space craft travels through space, the spatial resolution is highly dependent upon the sampling rate.

Another way to do it [16] [2] is to use four probes biased at different potentials. These probes are sampled simultaneously and a higher time resolution is achieved. One can derive the electron density independently of the electron temperature and the potential of the spacecraft.

With the multi-needle approach, calculating n_e can be done using linear regression. In the electron saturation region, the four probes can be approximated by a straight line. And since the sample points are only two-dimensional, one can use simple linear regression. Hence, the ratio $\frac{\Delta(I_C^2)}{\Delta V}$ in the n_e expression (Equation 1.1) is the *slope* of the approximated line. This is calculated with the formula [4] shown in Equation 1.3.

$$slope = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \tag{1.3}$$

Where n is the number of probes, x is the bias potential V, and y is the current squared I^2.

This computation is the primary focus of the new RISC-V extension.

1.4.2 Previous Theses

These are the theses that were most influential to me. The following is a brief overview.

Prototype Development

Bekkeng [16] did the original multi-needle prototyping. That thesis laid much of the ground work that is built upon by these other theses, including this one.

Instrument Dataflow

Kosaka [4] investigated possibilities of improving the m-NLP instrument. This thesis has been of much help to me. It showed among other things that an FPU is not necessary for running the computation in a timely manner. In order to i.a. reduce the total data transfer and test whether the computations should be done in flight, it looked at improving the performance of in-flight electron density computations. In the spirit of that thesis, I am not only removing the FPU, but I'm moving away from floating points altogether.

Softcore Processor

Bartas investigated softcore processors' fit-for-purpose for 4DSpace [6]. They used a LEON3 processor and tested much of the same calculation that I am doing in this work. Additionally, they investigated ASIC implementations and power consumption, among other things. The thesis served as a useful reference, because they had already investigated some points regarding processor cores so that I could more quickly get an idea of what I wanted to do. Moving in the direction of RISC-V continues some of the same spirit that was done by Bartas.

1.4.3 RISC-V Acceleration

The work of Kosaka looked at accelerating the in-flight computation. And the work of Bartas looked at implementing an open processor core to do the computation. In this thesis, the main idea of both of these theses will be continued; to use open processor cores with custom extensions to accelerate the computation.

RISC-V seems to be a viable candidate. The momentum it has gained, and kept, makes it likely to be relevant in the future. Its openness makes a good fit for academic work, such as the one revolving around the m-NLP project.

1.5 Verification

Some background information on verification can be helpful in order to understand the reasoning later in the text. This chapter gives a basis to justify the choices made about methodology and implementation mechanisms. What follows is an overview of the techniques that exist, and a few elaborations on the most important paradigms.

1.5.1 Techniques and Methodologies

This is a discussion about which verification methodologies exists, with their pros and cons. The goal in gathering this information was to decide how to best verify the resultant device. The intention is to be a primer for the succeeding text, as well as to show how that choice is far from a straight and narrow path.

Terminology

There exists many different ways of doing verification. For instance, differentiating between static and dynamic [18]. Static techniques are e.g. manual code review, linting software, formal verification, all of which do not *run* the design. Dynamic, contrarily, implicate *running* the design in either simulation, FPGA, emulator, or similar.

An attempt was made to create a taxonomy in [19]. However, a strict hierarchy has problems, like deciding whether the top-level node should be the differentiation between functional and non-functional verification, etc.

When searching for literary resources on verification, a vast array of terminology is encountered. The following list is an attempt to categorize and map out those terms. Figure 1.4 illustrates part of the problem.

- Methodologies and Frameworks – AVM, ESL, eRM, OSVVM, OVM, RVM, TLM, URM, UVM, UVVM, VMM.

- Isms – Agent-based, Aspect-oriented, Assertion-based, Class-based, Coverage-driven, Constraint-driven, Event-driven, Feature-driven, Functional-driven, Interfaces-based, Metric-driven, Model-driven, Network-based, PSS-based, System-based.

- Implementation – Graph-based, Hw sw co-verification, Block-level, Chip-level, Integration-level, System-level.

- Paradigms – Simulation-based, Constrained-random, DirectedTests, Formal (model/equivalence checking), Formal (property checking), HLS (high level synthesis), Linting (structural analysis?), PortableStimulus, Reviews, Self-Checking, Model-based.

- Languages – C/C++, DPI (direct programming interface), e (specman), HiLo (forerunner to verilog), OpenVera, OVA (OpenVera Assertions), PSL (Property Specification Language), SCV (SystemC verification), SuperLog, SVA (SystemVerilog Assertions), SystemC, SystemVerilog, Verilog, VHDL, myHDL, nmigen.

- Coverage – Branch, Expression, FSM, Functional, Line, Mutation, Path, Toggle, System, UCIS (Unified Coverage Interoperability Standard), Connectivity checking.

1.5.2 Main Functional Classes

This section covers the three main classes of functional verification which are considered for this work. It gives a narrowed-down look at a subset of the technologies; honing in on a decision. Excluded from this list are honorable mentions like portable stimulus and graph-based verification. (The apparent order of increased utility is unintentional).

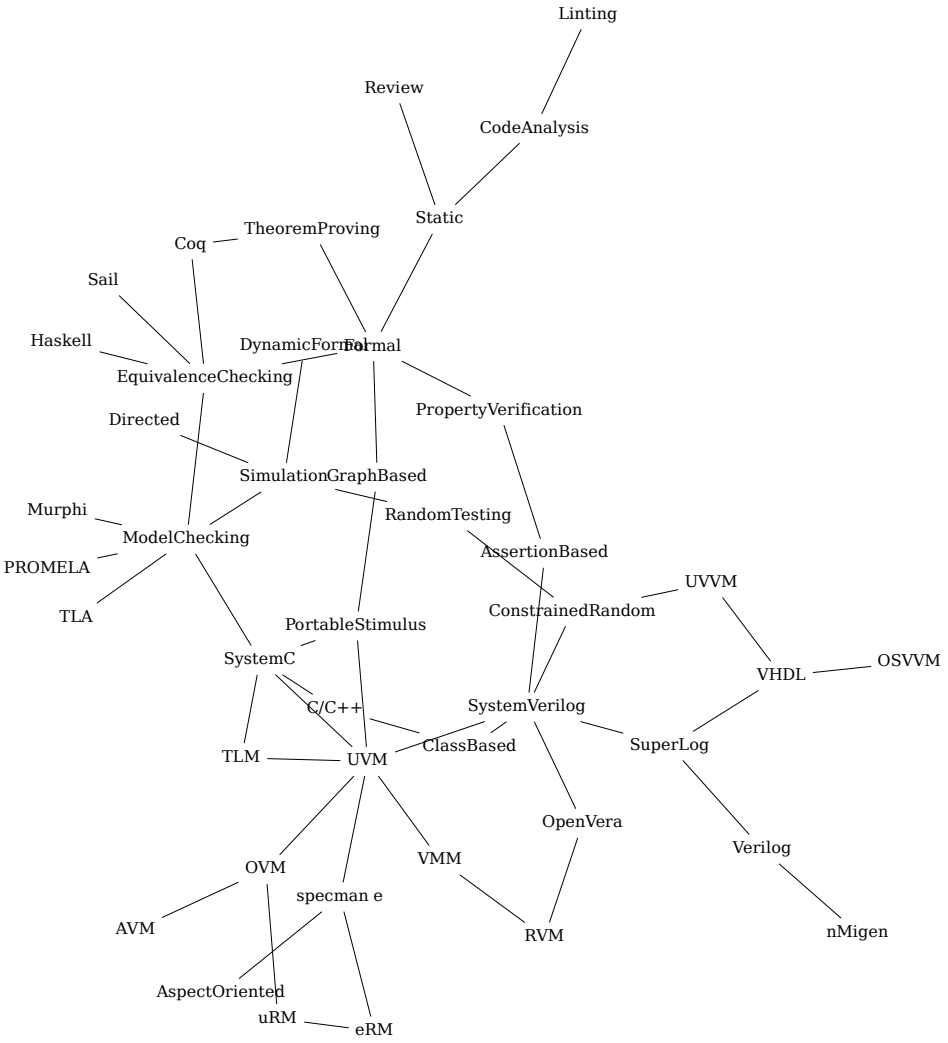


Figure 1.4: Map of a small part of existing technologies, illustrating the complexity involved.

Deterministic Simulation

Deterministic simulation, also called directed tests, is the least complicated technique in principle. Within this mode, one runs a predetermined sequence [19, p. 71] of stimuli onto a design and checks for known expected results. This is the basic testbenches that are widely known — explicitly specifying the stimuli one wants to exercise with. The tests are usually hand-written for specific cases of input/output, similar to a check-list. For big designs, the effort of covering all cases by manual labor is increasingly difficult.

Random Pattern Simulation

Feeding random input to a system, it can cover many cases that directed tests is liable to overlook. This often relates specifically to constrained random (CR). This is done [20] by first applying constraining requirements

to certain variables (e.g. `constraint a {x < y;}`) and letting a "solver" generate a wide range of values in accordance with the constraints.

For a complex design, it could be unlikely [21, p. xiv] that CR would stumble upon certain bugs. And even when all tests pass, it can be difficult to know how much of the design was verified (metric driven verification [20] deals with this measurement problem). It can be computationally expensive [21] to run extensive CR tests, and exhaustive testing is usually infeasible.

CR is a typical [21][20] way of doing simulation-based verification. And UVM is one commonly used methodology for implementing CR verification, with a reference implementation released under an open source license. Because CR is both popular and because it can be an efficient way of engineering a verification system for a design, this is a good candidate as a tool to be used.

Formal

Formal verification [21] is an approach to functional verification, by using mathematical analysis to assert properties of a designs behavior. For a component to be verified as correct, it's implementation must logically imply its specification.

Functional verification can cover whole spaces of possible states. It is often contrasted with simulation-based verification, which must try specific values by stepping through each one (See Figure 1.5). It can *prove* (or disprove) mathematically that a design is a correct realization of the specification. If the size or complexity of some logic is too big, formal might not be the most feasible alternative (combinatorial explosion). Off-the-shelf tools exist for a price, but open source projects are not as readily available.

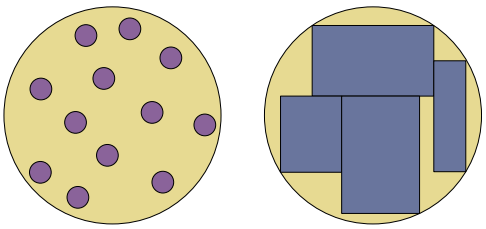


Figure 1.5: Coverage of simulation (the left circle) and formal (the right circle). [21]

There are roughly two main categories of formal verification. In formal property verification (FPV), properties are specified using e.g. assertions (assert, assume, and cover statements. [22, p. 368]), and can then be proved to either comply or not with the specified properties [21, p. 10]. And formal equivalence verification (FEV) compares [21, p. 11] different abstraction levels of a design (model, RTL, netlist, layout, ...) to see if they are equivalent.

(I had wished to use this technology, but time and resources available regrettably did not allow for it).

1.6 Planning

The goals of the project are expounded in the sections below. Priorities of the work are specified, followed by specific target requirements.

1.6.1 Quality Characteristics

The qualities discussed here are used to define the scope and goals of both design and testing. For this purpose, a taxonomy of system qualities is used [23]. Figure 1.6 shows an overview of this taxonomy, along with visual annotations to indicate priorities in this project.

Top Priority *Functional*: The functionality must be appropriate to meet the goals in mind. It shall also completely meet all the stated requirements, and correctly produce precise results. Lest it has zero utility.

Performance: This is the second most important characteristic, so as to speed up the computation of electron density. Hence, *time* is of especial concern. Second to this is taking heed of area, and so *resource* use is also of some interest.

Secondary Priorities *Portability*: The module should be installable in new systems, preferably also adaptable to changes in surrounding source code, and also replaceable by not having an overly idiosyncratic interface and in case a better solution is made.

Usability: Aesthetics, user error protection, and accessibility is of no concern. The important part is that the system can be learned, operated, and that its utility is recognizable.

Maintainability: This is important for being able to adjust to future needs.

Inconsequentials As indicated by de-emphasis in Figure 1.6, some aspects are of no concern. *Compatibility*, whether the device can communicate with other devices and share common resources, is beyond the scope. Only one aspect of *reliability* is considered: fault tolerance, in that erroneous usage does not propagate. *Security* is ignored altogether, because concerns of authentication and data protection should be handled at a higher level of a spacecraft's system.

1.6.2 Requirements

As explained in the section about planning, requirements set the foundation of many things. In the name of pursuing tractable results, certain requirements are prioritized above others. It is important to keep the scope manageable, since other parts of the project also needs attention. The target requirements are not numerous, but that does not mean they are easy. Targets for power usage and similar is not considered.

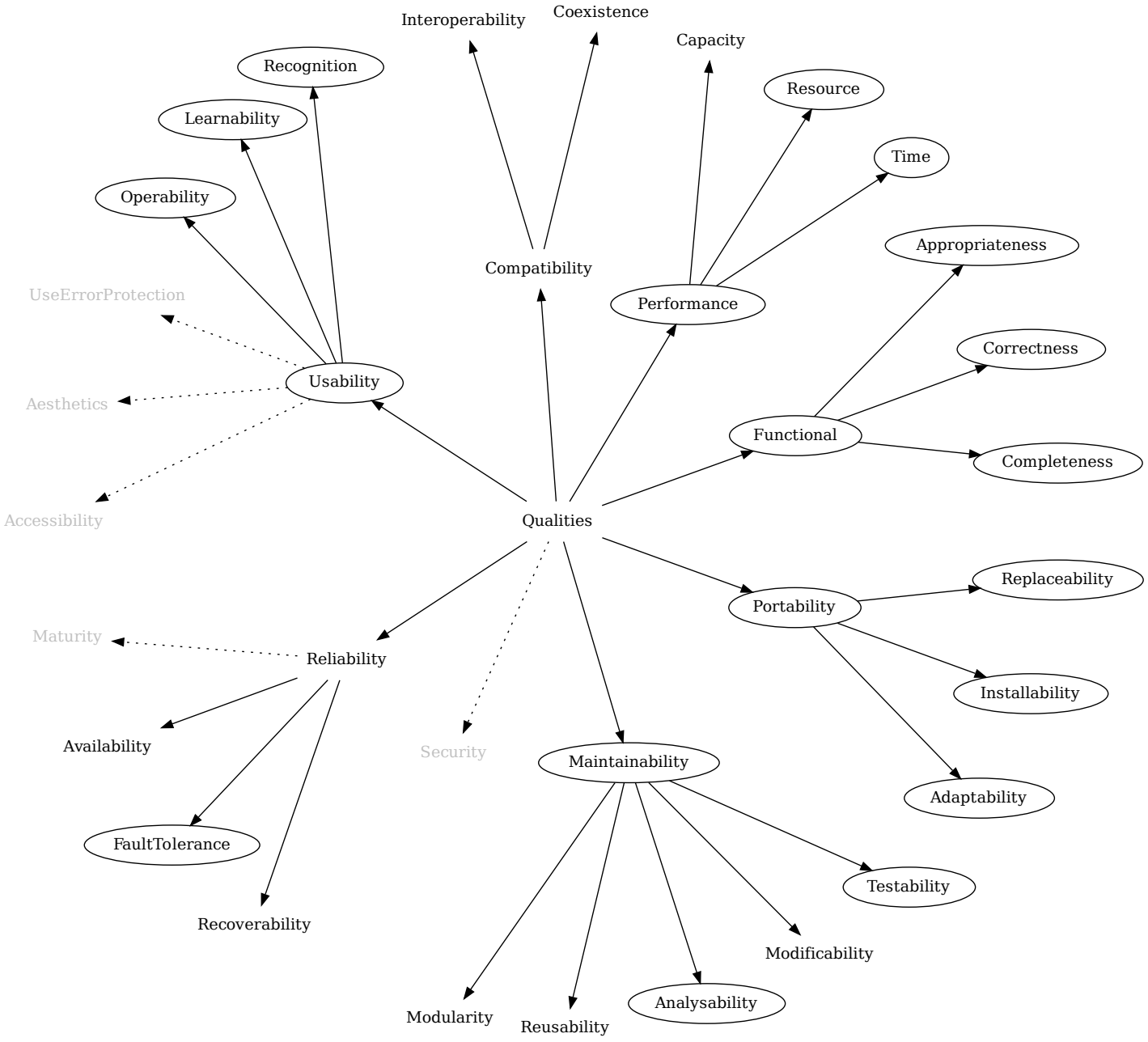


Figure 1.6: Graph of quality characteristics. Encircling denotes elevated priority, and graying represents lowered priority.

Functional The scope of the task is narrowed in, so as to spend the energy on the technical aspects of customizing a core. In other words, keep it simple, and more finesse can be built later if so desired. Of the three parameters that were calculated in previous theses (electron density, platform potential, correlation of determination) [4], only electron density

will be computed. Omitting having the beta factor[6] adjustable has the benefit of making it easier to compare to the works of Kosaka and Bartas. This means that the linear fitting technique [24] cited for its applicability for onboard-computation is used, instead of the non-linear technique.

Precision of the results is the primary functional requirement. The targeted range is the minimum and maximum electron density that is regarded. Bounds were found in Bekkeng [16] and in source code that I was granted access to [25]. The step resolution must also be good. As far as I could tell, there was no unanimous call for a specific resolution. One option is to use the minimum density ($10^8 m^{-3}$) as the step size, giving approximately 13 bits of resolution. Another option is to insist on e.g. 16 bit resolution, which corresponds to a step size of $\sim 1.5 \cdot 10^7$. The targets are summarized in Table 1.2.

Table 1.2: Target range of electron density.

Target	Density $[m^{-3}]$
Minimum	1e8
Maximum	1e12
Resolution 1	1e8
Resolution 2	1.5e7

Timing The computation frequency f_c must be *faster* than the input stream of new samples f_s.

$$\frac{f_c}{f_s} > 1$$

Meaning, roughly (disregarding overhead of other tasks running on the processor), that the clock frequency divided on the number of cycles needed for whole computation is bigger than the sample frequency:

$$\frac{f_{clk}/\text{cycles}}{f_s} > 1$$

and

$$f_{clk} > f_s \cdot \text{cycles}$$

The sampling rate is desired to be 20 kHz in the future [7]. So however many cycles it takes for a complete computation of the slope, must fit within this time. This ultimately determines the minimum clock frequency. In addition to this, a comparison is made against sample rates that are *known* to have been used in m-NLP systems. These are 5787 Hz (6k) [26] and 1 kHz [27].

As a guidepost (without strict necessity), inspiration can be taken from previous master's projects. Bartas [6] and Kosaka [4] both used 100 MHz and 160 MHz in their experiments. If the unmodified core is at all capable

[6]The equation for electron saturation current includes an exponent which is assumed to be 0.5, but can vary under non-ideal situations [17].

of these speeds, then it is interesting to see whether the modifications has a negative impact on it.

The targets are summarized in Table 1.3.

Table 1.3: Minimum target clock speed, along with some guideposts.

Target	Frequency
Lower 1	1 kHz × N-cycles
Lower 2	6 kHz × N-cycles
Primary	20 kHz × N-cycles
Upper 1	100 MHz
Upper 2	160 MHz

Resources By choice, there are not any strict requirements for resource use in this work. This is because the ultimate implementation medium is as of yet undecided, and because the more generic measures translates better for weighing overall performance. A reasonable target is to have the finished core fit onto a typical FPGA as used in courses at UiO. That is, either on an Altera Cyclone V DE1-SoC board or on a ZedBoard Zynq board.

Regarding transmission of data, the output result must be smaller than transmitting all the sample data. That is, as a set of sample is 4x 16-bit, the result should be less than 64-bytes. Otherwise the only benefit of the system is a potential speed-up.

Chapter 2

Methods

This chapter is divided mostly chronologically as: Design, Implementation, New Core, Verification, Post-processing, Measurements.

Design is about how the m-NLP calculations were transformed into new assembly instructions. For instance, what number representation to use, how big each step of the computation should be, and how to balance optimizations for speed, simplicity, and reuse. Implementation, then, is about implementing the design into a processor core, and expounds on the details and internal workings of fitting new instructions into a core. And as a small aside, an entirely new processor core was written from scratch. This started out as a fun hobby project, but the learning outcome proved to be helpful in grounding my understanding of customizing a processor core.

After those steps, the sections about verification, post-processing and measurements deals with testing the implementation. Verification checks that the implementation is bug-free and works as intended. While post-processing deals with getting a correct measure of electron density as a result. And finally, the measurement section details the protocols which yields the final benchmarks of Results.

2.1 Design: m-NLP Computations

This section discusses the transformation of the mathematical computation into hardware that computes it. It searches for the best way to compute the electron density. Different approaches are considered and the final solution is elaborated.

2.1.1 Electron Density

As discussed in Section 1.4 m-NLP, the electron density n_e can be computed as follows.

$$n_e = \sqrt{C\frac{\Delta(I^2)}{\Delta V}}$$

Let us assume for a second that the differences are computed as $\Delta x = x_2 - x_1$, so that a point can be made. In order to frame this computation in a format more suitable for digital computation, the n_e equation is drawn

as a signal-flow graph in Figure 2.1. The input numbers come in from the left and pass through each sub-operation corresponding with the equation. It becomes apparent that some of these sub-calculations are independent of each other and can be parallelized. These are candidates for aggregate instructions.

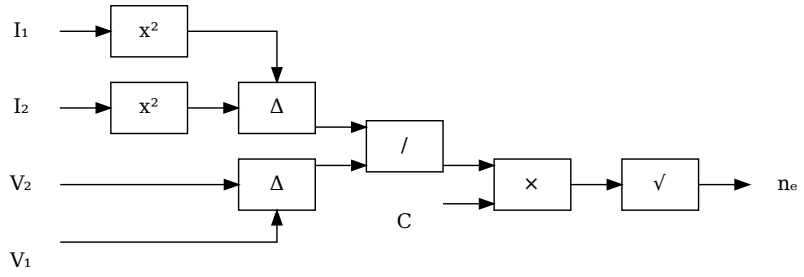


Figure 2.1: Signal-flow graph of electron density calculation.

The worst case time of executing the whole computation is achieved by doing just one sub-calculation in sequence. Although, pipelining in the processor will be of help, as illustrated in Table 2.1. If each operation takes one CPU cycle and nothing is done in parallel, the computation can complete in 11 cycles at best on a 5-stage processor. With instruction-level parallelism it is possible to run some of these operations at the same time.

Table 2.1: Sequential execution on 5-stage pipeline.

	1	2	3	4	5
1	X^2				
2	X^2	X^2			
3	ΔI	X^2	X^2		
4	ΔV	ΔI	X^2	X^2	
5	/	ΔV	ΔI	X^2	X^2
6	×	/	ΔV	ΔI	X^2
7	√	×	/	ΔV	ΔI
8		√	×	/	ΔV
9			√	×	/
10				√	×
11					√

Several options are possible for solving this. For instance, the final multiplication can be done either before or after the division. Or, one can defer the final multiplication and squaring until post-processing on a computer *after* the tiny processor has done its job. Everything could be done in one whole operation, either stalling the pipeline or severely sacrificing clock frequency. One alternative could be to use a standard RISC-V digital signal processing (DSP) module, but then there is a loss of potential case-

specific speed-up and one must also accept extra functionality which might not be necessary. The chosen solution is described in the section that follow.

2.1.2 Least Squares

Since the equation is solved using linear least squares, the situation is a bit more complex but the same principles hold. The ratio of $\Delta(I^2)/\Delta V$ is gathered from 4 points of probe data, and linear least squares fitting is used to obtain a decent approximation to the I-V slope. As a reminder, the formula is repeated in Equation 2.1, where n is the number of probes and x is the bias voltage and y is the current squared.

$$slope = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \tag{2.1}$$

This, too, can be illustrated as a signal-flow graph, depicted in Figure 2.2. Each node shows either multiplication, squaring, summing, subtraction, og division. Using such graphs were found to be helpful in reasoning about partitions of the calculation and finding suitable assembly code instructions.

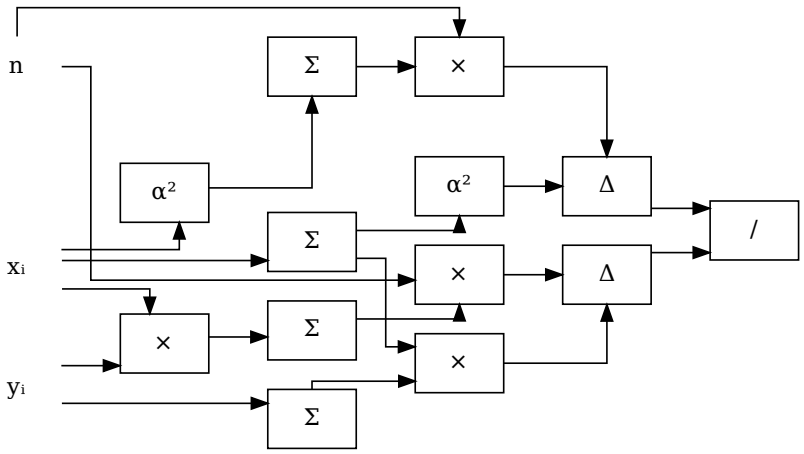


Figure 2.2: Signal-flow graph of LLS slope equation

A representation in the form of a script is shown in Listing 2.1.

Listing 2.1: LLS calculation in Octave/Matlab.

```
XSum = sum(x);
YSum = sum(y);
XYSum = sum(x .* y);
XSqrSum = sum(x.^2);
top = 4 * XYSum - XSum * YSum;
bot = 4 * XSqrSum - XSum.^2;
slope = top / bot;
```

2.1.3 Number Representation

Numerical data types can broadly be categorized as either integers or floats. The floats are usually limited to either either single-precision (32-bit) or double-precision (64-bit). Moreover, they can be used on an actual FPU or emulated in software based on integers. In addition to the typical IEEE 754 floating points, one could experiment with the *posit* [28] number format. Or one could use fixed-point floats represented by integers.

The incoming signals are represented as 16-bit integers. Whereas previous work [4] [6] has used floating point numbers to perform the calculations. Here, it is decided to do most of the computation directly in pure integers. Respecting the range of input currents and bias voltages, it is possible to convert back to physical units in post-processing.

Early attempt An initial attempt was done using the ranges of allowed input currents and bias probes, shown in Equation 2.2.

$$n_e = \sqrt{C\frac{I_2^2 - I_1^2}{V_2 - V_1}} = \sqrt{\frac{i_{I_2^2} - i_{I_1^2}}{i_{V_2} - i_{V_1}}}\sqrt{C \cdot \frac{I_{range}^2}{V_{range}}} \tag{2.2}$$

Where I is current, V is voltage, C is a constant, and i are integer representations. And the physical values were represented as in Equation 2.3. Where $r_i \in [0, 1]$ is the ratio of a reading relative to its defined range.

$$\begin{aligned} V_2 - V_1 &= [V_{min} + r_2(V_{max} - V_{min})] - [V_{min} + r_1(V_{max} - V_{min})] \\ &= (i_{V2} - i_{V1})\frac{V_{max} - V_{min}}{2^N - 1} \end{aligned} \tag{2.3}$$

This method was rejected as it does not translate easily to the 4-point LLS method. Instead, directly treating the inputs without associating them with physical sizes way chosen. Physical units are done in post-processing, detailed further in Section 2.5.2 Electron Density.

2.1.4 Partitioning

The equation to be computed consists of multiple computational nodes, or "steps". These must be broken down into chunks constituting single instructions.

Problem

Such computational steps can sometimes be done in parallel, or they might potentially be done out-of-order, and the various parts can be grouped in a multitude of different ways. The bigger the equation, the more possible permutations of subdivisions.

How does one decide which way one would like to partition an equation for effective execution? The problem is dependent upon several factors. For instance, what subdivisions are even possible? And what are the system's

constraints in regard to timing, size, power, etc? Another problem is, what work-methods exists for figuring this out (apart from trial-and-error)?

The following list is just a few complications to consider:

Pre-calculate	-	If some parts of the equation are static and unchanging, it can be calculated once and shared multiple times.
Multi-cycle	-	Instructions may have the ability to stall the pipeline.
Own-registers	-	The module could be allowed to have internal registers, or it could rely solely on general purpose registers.
Fixed-location registers	-	Can it be reasonable to omit `rs1` and `rs2` fields, and let instructions use predetermined registers, allowing for more input per instruction?
Assume extensions	-	Is it reasonable to demand the presence of e.g. the "M" module?

Some approaches are more speculative and experimental than others[1]. For example, squaring of two pre-specified registers (i.e. not assembly operands) done in parallel and writing back the results, is a very non-RISC thing to do and could be problematic for the processor pipeline.

Area and complexity can be saved by avoiding FPU, MULTDIV, DSP, and the like. So maybe some instructions that typically already exists in other modules could still belong to the Xmnlp set.

Cramming too much into one instruction could bring misfortune, because doing the whole equation as 1 instruction will take too much time. The critical path will make sure the clock frequency plummets.

Graph Permutations

One way to map out the multitudes of ways to order the calculations is by viewing the flow graph as a matrix. That is to configure the flow graph into a grid and assign each node into a position in a matrix. Such a configuration is alluded to in figure 2.3. A 2-point simplified calculation is used as an example because its low complexity allows for discussing more complicated treatment of it. The points made still holds for any other signal-flow graph.

With the graph represented in matrix form, one can use scripts to try out all permutations of partitioning and check for valid solutions. Listing 2.2 demonstrates one way to do it. For each horizontal starting point L, all possible endpoints $R = L + 1$ are tried out, and the space between them are checked to be a valid instruction.

[1] In hindsight, the chosen solution was indeed experimental.

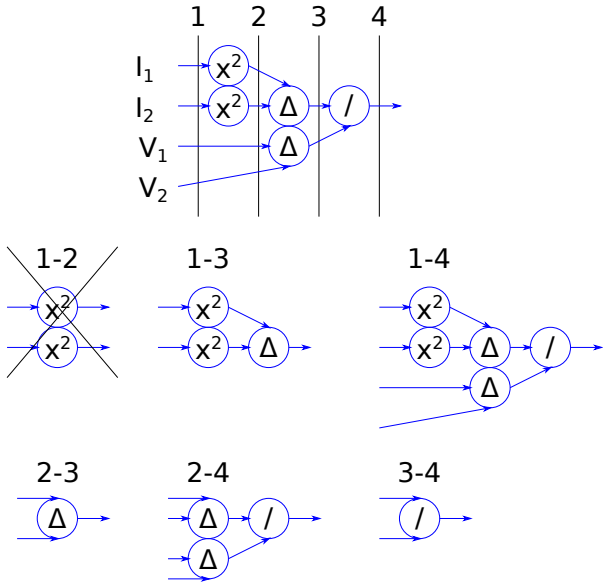


Figure 2.3: Example partitioning of 2-point n_e equation

Listing 2.2: Example traversal of flow graph in matrix form. All horizontal groupings are tried out.

```
for L = 1..end
  for R = (L + 1)..end
    if "only 1 output"
      => can be an instruction
```

For the graph in Figure 2.3, grouping 1-2 can be considered illegal, as long as SIMD instructions (single instruction, multiple data) are not used. It fails the condition of having only 1 output. However, that condition is alone insufficient to filter out suited candidates for instructions. A few more necessary conditions are i.a. 1) the end node's dependencies are covered by the start nodes' dependencies, 2) the output node is not part of the start nodes, and 3) edges can only point in one direction.

Brute Force Permutations

To find all possible partitions of an equation, scripting languages were employed in an attempt to brute force all possible combinations. Graphs were represented in matrix form as a list of lists[2]. Edges are kept in a separate matrix from the node values (names).

The first attempt was done in python. Listing 2.3 draws the graph of Figure 2.3 above.

[2]As one might have noticed from the illustrations throughout this document, I am really fond of graphs.

Listing 2.3: Abbreviated code structure for representing graphs as lists.

```
nodes = [ \
["I1", "sq", "diff", "div"], \
...
]
edges = set()
edges.add(((0,0), (0,1)))
...
```

From that structure, permutations could be generated e.g. by functions like: Listing 2.4

Listing 2.4: For every edge, form a group with every other edge, repeat N levels deep.

```
def permutateNIter(pile, remains, i):
    if (i == 0):
        return pile
    else:
        curpile = pile.copy()
        for r in remains:
            newpile = augmentPile(curpile, r)
            newremains = remains.copy()
            newremains.discard(r)
            news = permutateNIter(newpile, newremains, (i - 1))
            pile.update(news)
        return pile
```

Scheme (programming language) was used instead of python. The reason being that python provided such a great overhead for working with lists in comparison with scheme, which is a LISP-like language. As a result, graph-handling code became faster to write.

Consider the arbitrary graph defined by the edges in Listing 2.5. The corresponding edges are shown in Figure 2.4.

Listing 2.5: Testing-purpose definition of edges in a graph. Coordinates read as m.n (that is y.x).

```
(define
  edges-simple
  '(
    ((0 1) (0 2))
    ((2 0) (0 2))
    ((0 0) (0 1))
    ((1 0) (0 1))
  )
)
```

A program was written to generate all partitions from an edge specification, and retain only the legal ones. Filtering functions remove the invalid partitions based on graph properties. Example output of the program is shown in Listing 2.6. This result was used to derive the colored partitioning in Figure 2.4. Note that the outermost partition has three inputs, and therefore risks not being realizable in the available instruction

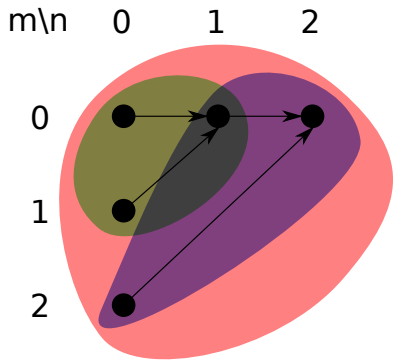


Figure 2.4: Valid partitions of a simple graph.

formats. In the end, feeding the LLS slope signal-flow graph into this script yields every potential instruction that may comprise the Xmnlp instruction set.

Listing 2.6: Legal partitions indicated by permutation script.

```
((0 0) (1 0) (0 1))
((0 0) (1 0) (2 0) (0 2))
((0 1) (2 0) (0 2))
```

Trial and Error

A different method than the one above was tried out; the slightly less systematic method of trial and error. After spending time working on the intricate affair of permutations, the time available became a concern.

To start, a simple instruction was made that calculates the whole LLS equation in one go — not insisting upon being synthesizable. The instruction `loadv` takes in all the x-values (4x 16-bit fits in two operands), and `lls` takes all the y-values and returns the slope. Obviously, this would come at the cost of a huge combinatorial path if implemented in hardware.

Continuing improvement on this solution, persistent refactoring was able to evolve the implementation into a slicker one. This demonstrates the difference between big-design-up-front and so called agile development. Compared to the more organized attempts at partitioning the equation, trial and error yielded equally satisfactory results in terms of achieving manageable partitions of the equation. Actual performance implications was still to be seen.

Design Challenges

Along with the problems discussed above, there are certain difficulties relating to loss of precision, complexity of the implementation, performance, and the like. Here are some such problems, explaining what may go wrong, and discussion of possible solutions. Wherever this analysis is inadequate, later testing might be able to uproot remaining problems.

In devising new instructions, if one instruction does "too much" work, then the timing will go awry. A long instruction forced into one clock cycle will demolish the maximum clock frequency. Or, it will need more complex logic to fix[3].

If max clock frequency is unacceptable under given sample rate requirements, then *internal pipelining* can be used. That is, while one instruction is being executed, pending instructions must wait while the execution unit takes the number of cycles that it needs. This is bad because: 1) it is more CISC-like than RISC-like[4], 2) it needs more control signals than single-cycle instructions, 3) it resembles an external co-processor more than ISA extensions.

Another solution is more fine-grained partitioning of the equation to be calculated. In other words, make instructions that do less. If done right, the effort might yield only 1-cycle instructions. Even 2-cycle or 3-cycle instructions are preferable over one huge one. The drawback is that it is hard to concoct such partitions.

At the end of the computation is a division, which risks a loss of precision. If `top` and `bot` (numerator/denominator) are close values, then the ratio is close to 1 and integers will truncate or round.

Any solution will depend on which values are typically dealt with, on the environment of the sensors, and accompanying electronics. The output should map $[0, \text{INTMAX}]$ to the $[\min, \max]$ of expected ratios. If the input to the division always has a ratio $>> 1$ (either by chance or manipulation), then there is less of a problem. If the range of possible inputs are known and deterministic, then maybe custom logic can be made to map it to the integer range of the output. Should all else fail, then a last way out is letting software emulate floats for just this one division. A hypothetical alternative, is to use a very small subset of an FPU to do only the division.

There is a choice in what the equation itself looks like before partitioning into instructions. And this choice will affect resolution and implementation details. Consider the least squares equation (in simplified form):

$$slope = \frac{4 \cdot XYsum - XsumYsum}{4 \cdot XXsum - XsumXsum} \tag{2.4}$$

It can be multiplied by $\frac{1/4}{1/4}$ and preserve its meaning:

$$slope = \frac{XYsum - XsumYsum/4}{XXsum - XsumXsum/4} \tag{2.5}$$

In performing this calculation stage-wise using integers, choosing mult or div has implications for preservation of data. If it happens by nature of the system that the numbers are guaranteed to have certain properties, then the division might make sense. Usually, though, multiplication makes

[3] As it so happens, this became the case regarding wide multiplication (Section 2.6.5).

[4] CISC is "Complex Instruction Set Computer", where each instruction is translated to sub-instructions (Fun: check out MISC, OISC, ZISC!).

the most sense because it will preserve the data, at the cost of more bits to handle.

In calculating LLS, intermediary stages exceeds the 32-bit word length (see Figure 2.5). This is problematic because it cannot be stored in the general purpose registers.

Finding a solution is difficult because there is a large scope to consider. The equation can be altered, the architecture changed, input data can be changed, conventions can be broken, etc. One solution is to change to a 64-bit processor, but this is costly and complicates matters in other ways. Results can be stored in 2 registers in a combined double word, but this puts particular demands on the processors design (might limit choice of processors to use). The input size can be reduced to 15-bit, but this diminishes resolution. Internal registers can be > 32 bits, but this idiosyncrasy costs resources and might make working with the architecture awkward.

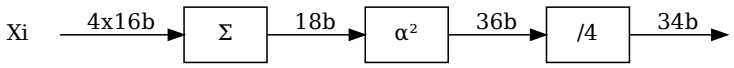


Figure 2.5: Example of intermediary calculations requiring >32bit.

2.1.5 Final Sets

Among all the possible partitions, 3 different sets were created. Each set is alone capable of computing LLS, with some assistance from the "I" base instructions. Each iteration is an improvement on the previous one, in regards of improving the efficacy of the implementation. The first sets were used as stepping-stones in writing the code. In the following text, as well as throughout this work, x is the bias voltage and y is the current squared.

Common to all the Xmnlp instruction sets, is `loadv`. This is because the voltage biases are held constant for each run. Several computations for different currents are done with the same voltage. Having a fixed bias is the normal way of operating the m-NLP instrument [3]. This instruction comes at the cost of having internal state and more registers, but benefits are less overhead and more parallelism.

- loadv - Stores 4 half-words inside the mnlp module.

Xmnlp0 - Initial As a basic starting point, this instruction set contains only one "god instruction"[5]. During synthesis and timing analysis, this did indeed show extremely poor performance.

- lls - Computes the whole lls from input half-words to result

[5]Referring to the OOP term "god object", an instance of a class which has too many responsibilities.

Xmnlp1 - Revised This instruction set investigates slightly more complex instructions. It computes the four terms of the LLS equation. "I" base instructions are needed to complete the full computation. The set is "regular" in the sense that each of the instructions performs a semantically similar operation.

- xsum - Sums the x-values.
- ysum - Sums the y-values.
- xysum - Multiplies x and y and sums it up.
- xxsum - Squares x and sums it up.

Xmnlp2 - Final The goal here is to create as small partitions as possible, while 1) avoiding duplicating the work of the base instruction set modules, 2) without creating a too long critical path, and 3) enabling parallelism. To understand the interplay between these instructions, it might be helpful to refer to Figure 2.6. In order to perform the operations relating to `top` and `bot` in parallel, while re-using the shared parts, the following instructions where devised.

- abaddab - Adds two operands together while heeding and storing overflow bits (that is, hjADDkl reads hj and writes kl).
- xxs - Retrieves the internally stored sum of x squared.
- xsxs - Retrieves the internally stored product of the x-sum.
- xsys - Adds the x-sum (stored internally) to the y-sum given as an operand.

Overview Figure 2.6 shows the flow through the system. This is the core of the computation, shown as a signal-flow graph with annotations about bit-width and instruction names. This overview shows what could have been just one big accelerator without tying each partition to a named instruction. By observing the red labels, one can understand the order of which the instructions must be called.

Fine-tuning As the project developed, weaknesses were observed in the chosen instruction set. These problems, and solutions to them, are discussed in section 2.6.5 Complication: Critical Path, 2.6.6 Complication: Multiplier Synthesis, and 2.6.7 Complication: Division Time. Both approaches, Xmnlp2 and the various fixes, are explained and measured.

2.1.6 Sub-Partitions

Because a small 32-bit core was chosen, the specialized Xmnlp instructions must accommodate those values in the computation which overflows 32 bits. An attempt was made to unify the benefits of a monolithic memory

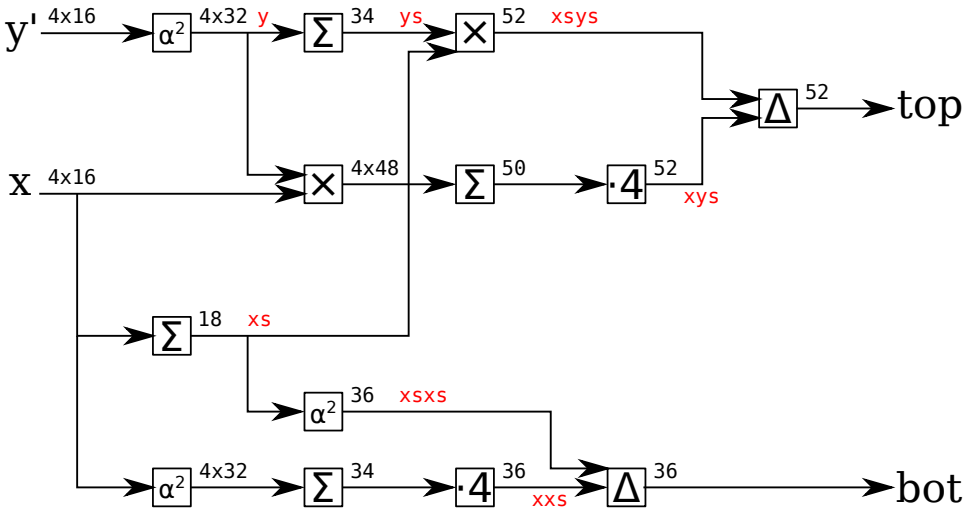


Figure 2.6: Signal-flow graph with labels on those nodes that represents steps in the computation.

mapped device with those benefits that comes from RISC-like granularity of instructions. Figure 2.6 does not give a complete picture; most of those computational nodes does not fit in one single instruction. Hence they need to be broken down.

The following is a detailed description of how sub-partitioning of the operations facilitate the complete computations. In the figures below, ellipses represent values in general-purpose registers, text-only nodes are internal registers to mnlp module, and prefixed subscripts are number of bits.

Y values The y-axis of the graph that gives the LLS slope, is in m-NLP context the square of the current I^2. That means that the input data mutates from a 4x 16-bit vector, to a 4x 32-bit vector. Figure 2.7 shows a basic signal-flow graph of the operation.

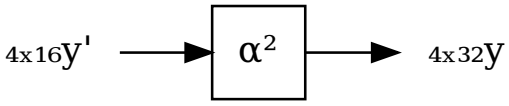


Figure 2.7: Low-detail signal-flow graph of the squaring stage. Where y'=I, and y=I², and pre-fixed subscript denotes number of bits.

Here, the "M" set's multiplication instruction can be used. But if the Xmnlp instruction set is to be used on a simpler rv32i processor, then a multiplication instruction can easily be added to the set. If it is found that

a dedicated squaring circuit is even faster, then that could potentially be desired in the future.

Ysum The process of summing 4 values of 32-bit each is not straight forward on a 32-bit architecture. One might ask the question of why not use a 64-bit processor, but the idea is to have a light-weight processor with a heavy-weight accelerator built in. Overflow bits are carefully tracked by the mnlp module, in order to avoid loss of precision. The abstract view is in Figure 2.8.

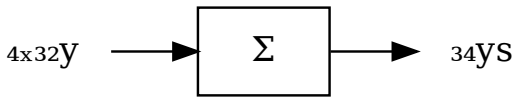


Figure 2.8: Abstract view of summing y-components.

Figure 2.9 shows the details of calculating the sum of squared current values. Squaring is done with "M" type pre-existing instructions. Summation is done with custom overflow-respecting instructions.

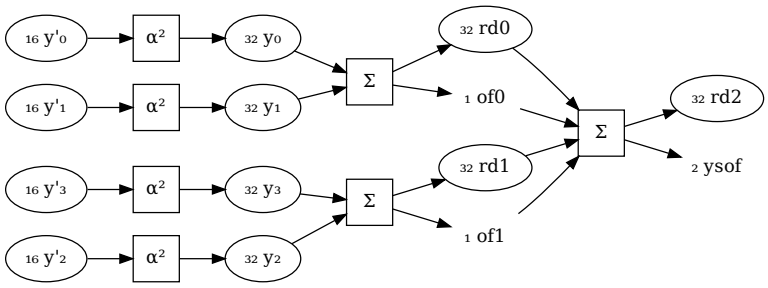


Figure 2.9: Detailed signal-flow graph of the squaring and summing of y values.

Xsum Ysum Equation dictates, the sums must be multiplied. The abstract signal-flow is shown in Figure 2.10.

Summing X is handled internally in the mnlp module to enable parallelism and speed-up. The sum of Y is saved in general purpose registers, but overflow bits are internally handled by the mnlp module. This is done to reduced the number of instructions that needs to be called and to reduce the number of general purpose registers in simultaneous use. The result is 52 bits big and is to split into a low L and a high H word. Figure 2.11 shows the detailed signal-flow graph of this.

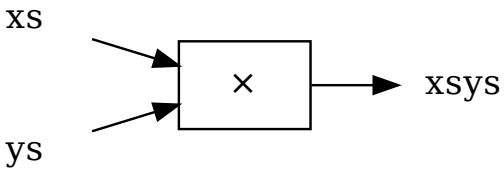


Figure 2.10: Abstract view of XsumYsum.

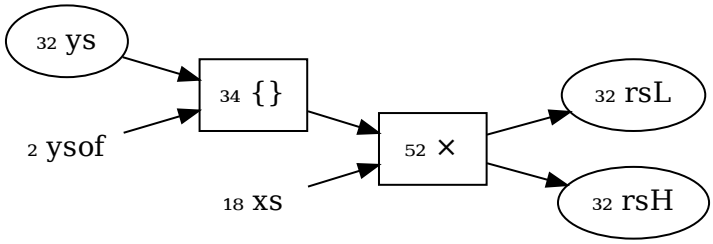


Figure 2.11: Detailed graph of XsumYsum.

XYsum X and Y values must be multiplied and summed. Multiplication is re-used from the RISC-V "M" instruction set module. The remainder of work is the sum and multiplying by 4. Figure 2.12 gives the abstract view. The multiplication is done in two steps, `mul` and `mulh`.

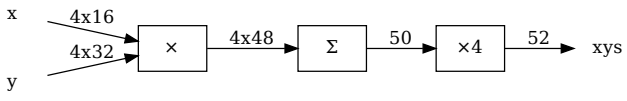


Figure 2.12: Abstract view of the xysum procedure, with bit-annotations.

After y_0, x_0 are multiplied, then y_0x_0, y_1x_1 are summed, before it is all summed together and left-shifted. Figure 2.13 shows the process.

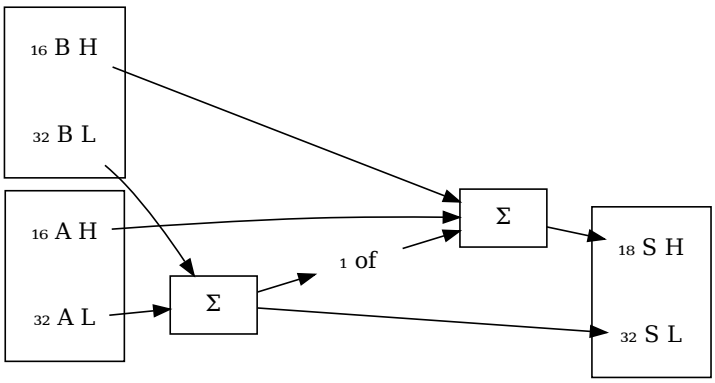


Figure 2.13: Two-stage summing handling overflows.

2.2 Implementation: RTL Code

Writing the code was not just about writing the computational module on its own. A fitting place within the processor core also had to be found. This entails adhering to how the RISC-V specification is structured. And the development toolchain should be adapted to accommodate the new additions. Thus, the following text explains the choices made for embedding a new computational module into a pre-existing processor core.

Roughly, the steps include 1) modify the decoder to recognize opcodes and other fields, 2) figure out how the ALU works and mimic its signaling, and 3) route control signals such as enable, operands, and result.

2.2.1 Pipeline Stages

In order to add new functionality to the core, its architecture was analyzed. Starting off with the fact that RI5CY has a 4-stage pipeline [14]. The stages being IF (instruction fetch), ID (instruction decode), EX (execution), and LSU (load-store unit).

For Xmnlp, the only relevant parts are ID and EX. Because, within the ID stage is the decoder which can recognize new opcodes and bit fields. Within ID is also a controller, which handles all the signaling and does operand forwarding on Xmnlp's behalf. And the EX stage is where execution is done, so that is where the new module must reside. Figure 2.14 shows the processor pipelines and the two points of insertion for Xmnlp.

2.2.2 Opcode

Before any instructions can be implemented in a processor, a choice of opcodes must be made. I place some care in this, because the RISC-V specification is carefully modular and a bad choice could be problematic for future changes. More than one opcode could be used for different

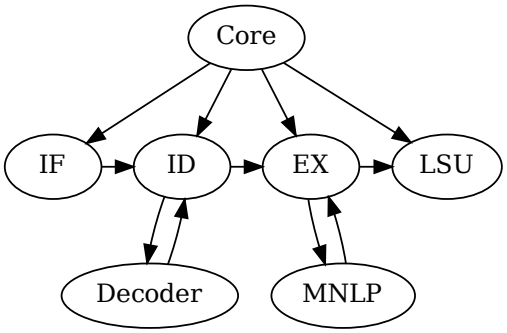


Figure 2.14: Pipeline stages of the processor core

instructions if needed. Or one could re-purpose opcodes for those standard extensions one is sure not to use. Although, the RISC-V specification has taken consideration for custom extensions [8]. As shown in Table 2.2, there are some opcodes dedicated for custom use.

Table 2.2: RISC-V base opcode map, inst[1:0]=11. [8]

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

A problem was that RI5CY had already taken up many opcodes. In addition to the standard extensions, it uses `custom-0`, `custom-1`, `custom-2`, `custom-3`, and one `reserved`. The appealing choice of `0b1111111` was then chosen, but this is the $\geq$ `80b` opcode and the assembler had trouble with it. So instead, the final decision was to use `0b11_101_11`, reserved.

2.2.3 Instruction Encoding

When the opcode is recognized in the decoder, the other bit fields of the instruction is used to locate the specific action to execute. The field `funct3` (3-bit wide) identifies classes of instructions. Examples are: `loadv`, `addops`, `xyops`, `xxops`.

To identify the exact instructions within a class, the field `funct7` (7-bit wide) is used. This can differentiate between e.g. `xsysl` and `xsysh`. Signals like write enable and which registers to use can then be controlled correspondingly. Any instruction that does not fit the recognizable fields is flagged as an illegal instruction.

2.2.4 Implementation Proceedings

Before it became apparent how to best proceed, a simple attempt was made using the tentative opcode `1111111` and the instruction `sqdiff` (squaring two operands and returning the difference). Before compiler support, hardcoded instructions were tested in assembly code in the following manner. According to the instruction formats (Section 1.3.3), bitfields were filled out, converted to binary, and written into the program in hexadecimal. The notation used is e.g. `7'h` for a 7 bit hex, and `5'd` for a 5 bit decimal.

1. `/* funct7=7'd0, rs2=5'd2, rs1=5'd1, funct3=3'd0, rd=5'd31, opcode=7'h7F */`

2. `/* 0000000  00010  00001  000  11111  1111111 */`

3. `.word 0x00208FFF`

The pre-existing modules ALU and MULTDIV gave inspiration for how to route the appropriate signals. As it was evident that they have similar functionality and both live within EX, and are then likely to implement instructions in the way desired for Xmnlp. Mimicking ALU made control signals relatively straight forward, although routing throughout the modules hierarchy later turned out to be the cause of a few bugs. Listing 2.7 shows how the existing infrastructure for the ALU was leeched onto by the new mnlp module.

Listing 2.7: ID-Stage utilizing ALU's operands.

```
always_ff @(posedge clk, negedge rst_n)
begin : ID_EX_PIPE_REGISTERS
[...]
  mnlp_en_ex_o <= mnlp_en;
  if (mnlp_en) begin
    mnlp_operand_a_ex_o <= alu_operand_a;
    mnlp_operand_b_ex_o <= alu_operand_b;
  end
[...]
```

The core's top-level merely connects Xmnlp signals between ID and EX. And the mnlp module is housed by the EX stage, along with ALU and MULTDIV. EX is already part of managing operand forwarding and write-back so this is exploited on this end as well. After the very first instruction was in place, the rest of the instructions were added as described in the Design: m-NLP Computations section.

2.2.5 Compiler Support

Getting compiler support for new instructions is surprisingly easy! The situation would be different if one wishes for C code to automatically translate to custom extensions when that would result in optimized code. But if one simply wants support for assembly code, the method is simple.

The compiler used is the ubiquitous `gcc` (GNU C compiler). This method also shares some steps in common with the Customizing The ISS section.

1. First, assembly mnemonics must be associated with the bit fields it accepts, as for example: `adda rd rs1 rs2 31..25=0 14..12=4 6..2=0x1D 1..0=3`.

2. From this definitions, a C header file can be automatically generated. This header contains two definitions for each instruction `MATCH_INSTRNAME`, `MASK_INSTRNAME`, and one macro call `DECLARE_INSN(...)`. And it need only be appended to the compiler's own opcodes header.

3. The third and final step — before compiling the compiler — is to register the instructions type and mnemonic: `{"loadv", 0, {"I", 0}, "d,s,t", MATCH_LOADV, MASK_LOADV, match_opcode, 0}`

One's assembly code will now generate object code as desired.

2.3 Simple Hex Instruction Transputer V

An entirely new processor was written. It was originally meant just as a hobby project for the free time, but the lessons learned turned out to be of much help in the Xmnlp project as well. The motivations for building it was 1) to gain first-hand knowledge of processors' inner workings, and 2) I wanted something simpler than what I had experienced from other cores. Resulting from it was relevant lessons about modifying a core, automated testing, adding new functionality, the cost of implementation, and tooling. The following discussion tries to give a brief overview of the new core and stay relevant to the overall topic.

The core is 32-bit, and strictly supports only the `I` standard module. Everything runs in unprivileged mode, meaning that there are no protections concerning which instructions can be called at any given time. Nor is there any memory protection. There are also no control status registers. Only open source tools were used, notably Icarus simulator and Verilator simulator, and gtkwave waveform viewer. A classic 5-stage pipeline was chosen because it is well understood and simple. RI5CY, on the other hand, is a 4-stage core [14].

2.3.1 Bootstrapping

When starting up, the very first steps where done by printf and inspecting waveform traces. The goal being to get an absolute minimum system up and running, so that automatic testing can get started. This also gives a clue for how to start introducing new instructions in existing cores.

Thus, the first instructions were `LUI`, `BEQ`, `JAL`, `ADDI`, and `SW`. LUI (load upper immediate) because it can *reset* the general purpose registers; BEQ (branch if equal) to handle decisions in testing; JAL (jump and link) is a stepping-stone before BEQ; ADDI (add immediate) is just a very primal

instruction; SW (store word) for halting simulation by writing to a magic number address. Listing 2.8 alludes to how ADDI was used to gradually introduce operand forwarding.

Listing 2.8: Testing operand forwarding.

```
        addi    x1, x0, 1
        nop
        nop
        nop
        addi    x1, x1, 1
        nop
        nop
        addi    x1, x1, 1
        nop
        addi    x1, x1, 1
        addi    x1, x1, 1
```

2.3.2 Automated Testing

Automated testing was an excellent tool for knowing whether changes to the RTL preserved all functionality. They give instant feedback and give clues for debugging. It turned out that writing such tests are quite difficult; false positives are surprisingly easy to create. For instance, if BEQ or JAL is implemented incorrectly, a LUI may appear to work correctly even though it does not. The tricky part is creating a test that makes as few assumptions as possible, and which defaults to failure. Listing 2.9 shows all the tests which together complete successfully in under a second upon running `make test`.

Listing 2.9: Overview of RV32I test suite.

```
rv32i
  ctrltran
    beq.S bge.S bgeu.S blt.S bltu.S bne.S jal.S jalr.S
  intcomp
    regimm
      addi.S andi.S auipc.S lui.S ori.S slli.S slti.S sltiu.S
      srai.S srli.S xori.S
    regreg
      add.S and.S or.S sll.S slt.S sltu.S sra.S srl.S sub.S xor.S
  loadstore
    lb.S lbu.S  lh.S  lhu.S  lw.S  sb.S  sh.S sw.S
```

Some problems turned up after continued testing. For instance, operand forwarding was not always handled correctly. Repeating multiple jump instructions consecutively also resulted in disaster. Sign-extension was another area of unexpected problems.

There also exists a set of official assembly tests for RISC-V compliance, called `riscv-compliance`. At the time of writing, 40 out of 48 such tests pass. All of the arithmetic and control transfer instructions work, but e.g. the misaligned jump tests does not pass yet.

Case in Point The following demonstrates a case of testing and debugging. At one time, a test timed out instead of halting gracefully. Because of the decision to use one-hot signaling, the problem was easy to spot. Figure 2.15 shows an ADDI instruction repeating a pattern along with the PC (program counter) register address which it corresponds to.

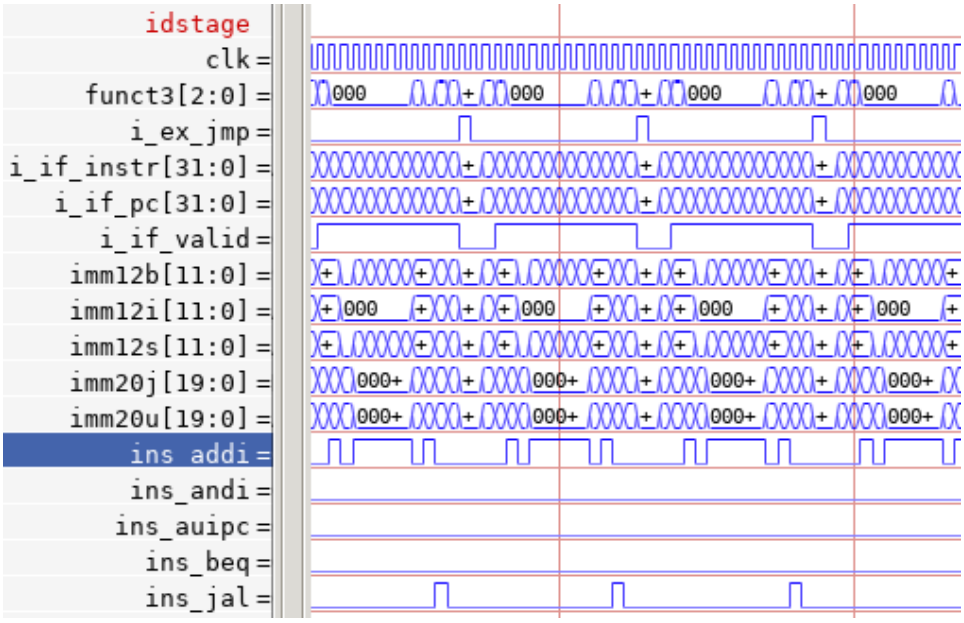


Figure 2.15: Repeating pattern of ADDI instructions indicating infinite loop.

Since the PC is propagated through the pipeline, it was possible to find the offending instruction via an object dump of the ELF (an executable format) binary. And it turned out to be a RET (return) instruction, which is a pseudo instruction [8] corresponding to a JALR. So, after writing new automatic tests for JALR, and getting to pass those, the original problem was solved.

2.4 Verification: Functional

This section is concerned with functional correctness. Or, informally, "bug hunting". It shows the process of using various verification tools and designing testbenches on both block-level and core-level. The goal being to ensure that the implementation matches the intended functionality. An outline of the process is illustrated in Figure 2.16.

2.4.1 ASM-C Testing

The very first verification was done in assembly. Opcodes and other bit fields were manually put together as for example `.word 0x00208FF7`. Thanks to the work on the Simple Hex Instruction Transputer V, I had developed an efficient method for writing such tests, allowing simple debugging:

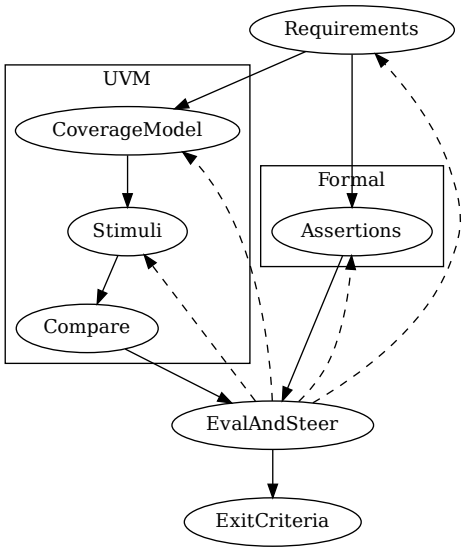


Figure 2.16: Functional testing phases.

1. `_start: addi x31, x0, 0: /* Initialize a default OK return value. */`
2. `test: [some test]; beq x3, x4, pass; /* Do a test, check success. */`
3. `fail: addi x31, x0, 0xAA; j halt; /* Set return ID, jump to exit */`
4. `pass: nop; /* Make the pass be listed in object dumps. */`

When assembly mnemonics were in place, using the instructions became way easier. And it can even be called from C with relative ease:

```
asm volatile ("adda %0, %1, %2" : "=r"(myret) : "r"(var1), "r"(var2));.
```

Tests written in C was able to check all the basic functionality of the instructions before moving on to direct bit wiggling in SystemVerilog.

2.4.2 UVM

While the introductory chapters covered the basics of verification, this section is about the specifics of the chosen methodology, UVM – The universal verification methodology. It is used here as a means for constrained random verification. The UVM is aimed at [29] making reuse of verification components easier, and to aid in interoperability, and reduce cost of intellectual property (IP) use. The specification is quite big and it took time to hone in on a suitable subset. This chapter elaborates on the choices made and the resulting architecture.

Learning

The UVM specification is quite large, weighing in at almost 500 pages [29] along with SystemVerilog (SV) standard at almost 1300 pages [30][6]. During the learning period, an onslaught of questions arose: Which classes are the bare minimum necessary for a profitable outcome? Is it advisable to use SV concept such as `program` or `clocking` blocks? How much responsibility should the top-level SV module have, contra the `uvm_test`? Is it really any point in having sequences when multiple `uvm_test`-expanded classes can do the same job? How does one logistically bundle related classes into separate files? There are countless of such kind of questions and it proved quite exhausting to try and answer them all.

To get through the learning, several experimental designs where tested. This was done to filter out those UVM constructs that are superfluous, or rather not appropriate for smaller designs. Experiments were carried out on a hamming weight counter (sum of ones), a simple shift register module, and even a small SystemC example. A maximalist testbench tried to use as many of the prescribed UVM classes as possible. While a minimalist setup tried to have as few classes as possible. Directed tests where translated to UVM for comparison. And one experiment tried out being phasing-based instead of class-based. Much where learned from these experiments, for instance that `uvm_agent` is unneeded in some cases.

Five sources were primarily used while learning:

- A paper from a convention "UVM Rapid Adoption: A Practical Subset of UVM" [31]
- A tutorial website "UVM TestBench architecture" [32]
- The source code repository of another tutorial website "uvm-tutorial-for-candy-lovers" [33]
- An introductory book on UVM "The UVM Primer" [34]
- The UVM specification itself [29]

It was found that there is not a definite consensus on usage of UVM. The resources above all favor different parts of UVM, and sometimes they are contradictory. For instance, three of the resources extends `uvm_monitor`, but the UVM primer book uses the parent class `uvm_component`. Some use `uvm_subscriber` and some use `uvm_scoreboard` instead. Some use `uvm_transaction` in stead of its child class `uvm_sequence_item`. Since UVM is a big standard that has evolved over time, my inkling is that part of this is caused by 1) historical legacy 2) too much redundancy in the options available. With all of these discrepancies, majority voting and personal preference was used to decide which concept to adopt.

[6] For comparison, the C89 standard is less than 250 pages.

Subset

From the whole of the UVM specification, a small subset was extracted. The following is the resultant collection of UVM concepts, sufficient to build quite intricate testbenches without getting lost in complexity.

UVM is object-oriented, and therefore consists of classes. After pondering the questions discussed above, the subset that emerged is shown in Figure 2.17. This is sufficient to exercise the DUT, collect coverage, and check results against an oracle reference model. The marked* classes are often extended several times to create finer granularity testbenches. For instance, `uvm_test` can be extended to `base_test`, which can further be extended into `addition_test` and `subtraction_test`. Descriptions of these classes are in the Topology paragraph below. Note also that sequence items are sometimes called "transactions", alluding to transaction level modeling (TLM).

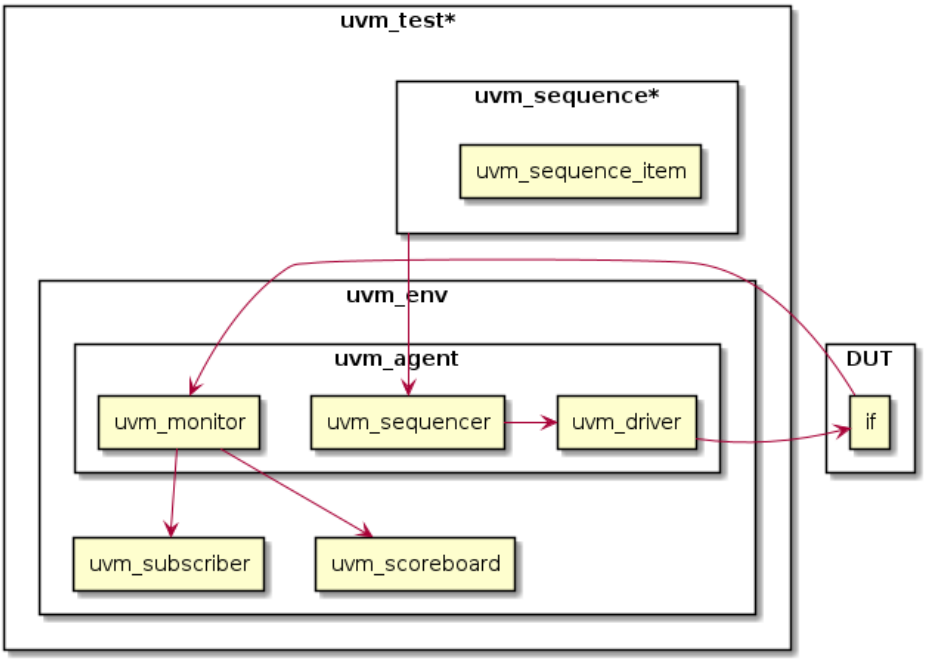


Figure 2.17: Minimalist set of UVM classes needed for a testbench. (For small designs, contents of `agent` and `environment` can reside directly within `test` instead).

The second most important concept of UVM is *phases*. When running a UVM testbench, certain steps ("phases") are run in a predefined sequence (some also run in parallel). A graphical representation of the phases in UVM is shown in Table 2.3.

Table 2.3: Overview of phases in UVM.

Predefined phases	Common phases	Build	
		Connect	
		End of elaboration	
		Start of simulation	
		Run	
		Extract	
		Check	
		Report	
		Final	
	Runtime phases	Reset	Pre reset
			Post reset
		Configure	Pre configure
			Post configure
		Main	Pre main
			Post main
		Shutdown	Pre shutdown
			Post shutdown

Each class defines what actions it wants to do in a certain phase, and the execution environment invokes that class' appropriate phasing procedure when the time is nigh. There exists many phases in the standard, but the level of detail that they facilitate is not necessary to use in its entirety. The *build* and *connect* phase is fairly common, but I decided not to use it. This is because I rejected the "factory pattern" and "global database" approach that is typically used in UVM. The factory and database patterns were rejected because typical object orientation like `driver = new();` is much simpler than `driver = my_driver::type_id::create("driver", this);`. Any purported benefits to the contrary did not reveal themselves. In conclusion, the phases that were used in this work are shown below.

- `run`: `reset`, `main`, `post_main`.
- `extract`, `check`, `report`, `final`.

Topology

Figure 2.18 shows an overview of the resultant testbench. The topology is almost identical for the block-level and the core-level verification. Where the first difference is that in the block-level testbench, there is no `core` in the testbench, only the `mnlp` module itself. And the second difference is that, for core-level, the monitor has two outgoing ports separating stimuli from response. Thirdly, the sequences are named and grouped differently, so as to also stimulate RV32I instructions. The driver and monitor of the core-level testbench is more complex than for block-level, but the topology remains the same.

At the top left are a set of `sequences`, each responsible for generating one type of operation stimuli. For instance, the reset sequence sets the relevant

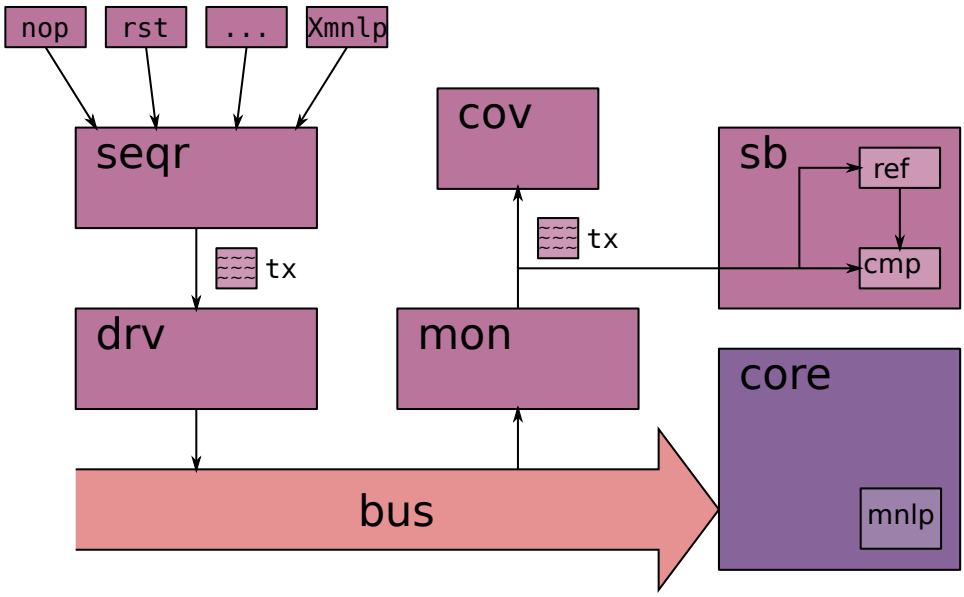


Figure 2.18: Topology of UVM testbenches.

data so as to induce a reset of the DUT (device under test). Within the main test class, the sequences are started up and connected to the `sequencer`. The sequencer, in turn, forwards the data to its connected `driver`. So called transactional level modeling (TLM) transactions (`tx`) are used as the messaging medium. The driver possess all the knowledge of how to drive this sequence item on to the `bus` of the core.

On the right-hand side, the `monitor` also holds know-how about the bus. It snoops up all the traffic that is taking place. It records the stimuli that is applied, and also the response that the DUT lets out. This information is then passed on (as transaction items) to both the `coverage` object and the `scoreboard`. The coverage class is responsible for counting the variations of stimuli applied and reporting the end result. The scoreboard does two things, 1) it feeds the DUT's stimuli to a `reference` model, and 2) it feeds the DUT's response to a `comparison`. Results from the reference is also fed to the comparison, and the two results (DUT and ref) are then compared, logged, and reported.

Using a reference model written differently from the RTL module has a good benefit. That is, they have their own areas of flaws, so any disagreements between the two will expose bugs. Writing in two distinct languages may help even more in this regard.

2.4.3 Coverage Goals

Through the use of coverage, the goal was to 1) have a target to drive verification towards, and 2) get a numerical metric for the thoroughness of the verification.

Coverpoints where use on the input of the DUT, so as to define all stimuli of interest. The same was done on the output, to be certain that all

scenarios are covered. Wide signals were divided into bins, differentiating between low, medium, and high range values. And a coverpoint for overflow was also tested, because this is a corner case of interest for testing a.o. internal registers.

Expanding on this, so called crosses were used. This is to make sure that all relevant combinations of the above coverpoints are tested. For example, all bins of the A operand has been individually tested against all bins of the B operand. To ensure internal overflow registers are exercised, the overflow coverpoint was crossed with the operator coverpoint. This means that all operators are tested for both non-overflow and overflow. The same was done for operators crossed with non-zero results.

2.4.4 Assertions

Assertions are a handy tool in verification. They are a sort of strict formal specification of desired behavior. An example is "it shall never be true that reset is asserted AND output is non-zero" shown in Listing 2.10. Assertions were used in the RTL module to check the internal state relating to the instructions. For instance, to check the `abADDab` instructions influence on overflow registers. It was also used to check that the internally conducted computations can only change under certain conditions.

Listing 2.10: Asserting that output is quiet under reset

```
assert property (
  @(posedge clk) (rst_n == 0)
  |-> (result_o == 0)
);
```

One particular benefit of assertions is that it can silently apply to most verification efforts. That is, the same assertions can be used in directed tests, constrained random, and formal. It can often be much shorter to write an assertion, vs writing a whole reference model or specific test to catch all scenarios of one behavior. One drawback is that SVA (SystemVerilog assertions) in particular is very much unsupported in open source tools.

Assertions can be useful in testing internal registers. Internal behavior is not always easy to catch through blackbox testing where one only tests a device through its endpoints. Internals are only visible from the outside via logical inference, and one needs certain sequences of stimuli for that to be possible. Assertions, on the other hand, can fit inside the RTL itself. Listing 2.11 shows part of the intention behind the `ADDa` instruction; to never affect the B overflow register.

Listing 2.11: Asserting ADDA's effect upon overflow bit B.

```
AddaLeaveB:
assert property (
  @(posedge clk)
  ((operator_i == MNLP_OP_ADDA) && (en_i))
  |->
  ##1 !$changed(ofb_q)
);
```

2.4.5 Core-Level

Core-level verification, as opposed to block-level, needs some additional furnishing. Although the topology is almost the same, the driver and all the other components must be made anew so that they match the interface of the core. For instance, the instruction-side bus of the core has a specific protocol that the driver and monitor had to be able to speak. The biggest difference is that the reference model must also support RISC-V concepts like the register file with general purpose registers and also some RV32I instructions. Making constraints for the randomized stimuli is also required more finesse than the block-level.

The following section contains a lot of source code listings. There are not too many ways of conveying the information as efficiently.

Simulator

As a golden model, any RISC-V instruction set simulator (ISS) can be used. The one chosen here is called `spike`[35], and is hosted by the RISC-V foundation. Since it is open source, there was nothing in the way of modifying it for a new purpose.

It had to be compiled with a non-default flag to enable so called commit logs. This makes executions of the simulator emit records of every memory- and register access, as shown in Listing 2.12.

Listing 2.12: Observing a register access with spike simulator in interactive mode.

```
$ spike -d pk hello.c
: reg 0 t0
0x0000000000000000
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
: reg 0 t0
0x0000000000001000
```

The commit log will then show that same register access (`t0` is an alias for `x5`), shown in Listing 2.13. With this feature enabled, the simulator is *almost* ready to be used as a reference model.

Listing 2.13: Commit log showing a register access.

```
3 0x0000000000001000 (0x00000297) x 5 0x0000000000001000
```

The examples above are running spike with a so called proxy kernel, which makes it possible to run C programs using libc (the C standard library) but it was found to not be the most suitable for running ”bare metal” applications. It is problematic because the proxy kernel and bootloader does a lot of actions, which lessens observability. And also, because managing the toolchain is a hassle (e.g. compile for 32 bit, etc). Consequently it was necessary to find a way to run programs without the proxy kernel. In that regard, consider the program in Listing 2.14.

Listing 2.14: Testing-purpose assembly program.

```
.global _start
_start:
        nop
```

Through experimentation with the proxy kernel and libc-linked programs, it was found that the spike simulator always runs its own bootloader which jumps to a given user-code address[7]. With a linker script it was possible to get the simulator to enter the program above, as demonstrated in Listing 2.15.

Listing 2.15: Instruction set simulator entering user-code.

```
$ make run
spike -d --isa=RV32I --priv=m addi.elf
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x000000000000100c (0x0182a283) lw      t0, 24(t0)
:
core   0: 0x0000000000001010 (0x00028067) jr      t0
:
core   0: 0xffffffff80000000 (0x00000013) nop
```

The corresponding linker script is shown in Listing 2.16. It does not have to be complicated.

Listing 2.16: Simple linker script to place the program entry point at the RISC-V implementation's boot address.

```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )
SECTIONS
{
  . = 0x80000000;
  .text : { *(.text) }
  .data : { *(.data) }
  _end = .;
}
```

Executables Generation

Yet another thing that proved necessary was to generate `.elf` executables from the instructions that were generated by the UVM testbench. It can be simple enough to generate random instructions, but a different challenge to get an ISS to execute that code. The generated instructions can be regarded as pure binary, informally "hex files". Problem is, the ISS expects an `elf`[8] executable.

Converting a hex file to an ELF file can be done with a RISC-V variant of a typical unix tool `riscv64-unknown-elf-objcopy`. But the ISS expects

[7] This was not apparent from readily available documentation.

[8] ELF is a format for executables files, used e.g. on linux computers.

the ELF file to be compiled for a certain machine type. This was handled with good old ”man pages” and the flag `--alt-machine-code=243` (243 representing RISC-V). The problem is once again not solved, because the ELF file ends up as a ”relocatable” object. Another problem is that since there is no metadata in a raw hex dump, this relocatable will not have a normal symbol table. Hence, there is no reference handle for which to place at the desirable boot address.

The method for creating runnable executables ended up pretty intricate. Summarized, the gist of it lies in a new linker file (Listing 2.17). Gcc allows linker scripts to directly include object files. Some initial boot code (a.o. zeroing the register file) is placed in `.text.init`. Followed by `.text` including the generated instructions from `instrs.o`. Finalized by any necessary clean-up code that is placed in `.text` from within the assembly code.

Listing 2.17: Linker script that forces the position of an object file.

```
OUTPUT_ARCH("riscv")
ENTRY(_start)
SECTIONS
{
  . = 0x80000000;
  .text.init : { *(.text.init) }
  .text : {
    instrs.o;
    *(.text);
    }
  .data : { *(.data) }
  _end = .;
}
```

Managing Execution

There are some maintenance tasks that must be done. For instance, if one does not implement some exit procedure, then the simulation will hang forever as it enters a default trap handler[9].

Firstly, it is necessary to establish a communications channel between the running program and the host simulator. Certain symbols need to be defined. Upon loading an ELF executable, the simulator checks whether the ELF is 32 or 64 bit, whether the ELF is flagged as executable, if the machine code is RISC-V, etc. [10] Most importantly it checks for the existence of the symbols `tohost` and `fromhost`. Doing a write access to `tohost` will now send data to the ISS.

Running a bare metal executable is very easy in itself, but simulation will not halt automatically. Exit is simply done by writing `1` to `tohost` (Listing 2.18).

[9] A ”trap handler”, aka ”exception handler”, is jumped to when the processor encounters an exceptional or illegal instruction or scenario.

[10] Gritty details can be found at `https://github.com/riscv/riscv-isa-sim/blob/master/fesvr/elfloader.cc`.

Listing 2.18: Trap handling in spike ISS.

```
.text

.global _start
_start:
        la  t0, trap_vector
        csrw  mtvec, t0
        csrr  t1, mtvec
1:
        bne t0, t1, 1b

[...]

_exit:
        la      t1, tohost
        li      t2, 1
1:
        sw      t2, 0(t1)
        j       1b

trap_vector:
        j       _exit
```

This code also shows trapping of exceptions. Exceptions can be caused by interrupts, illegal instructions, and other things that are not strictly normal program flow. When designing custom instructions, there is likely that one will want to simulate them using an ISS. In the scenario where a testbench generates custom instructions, but the ISS is not yet modified to accommodate, the exception handler is triggered. The code above simply jumps to the exit procedure upon any exception.

Customizing The ISS

Adding support for new instructions in the instruction set simulator needs only a few steps [36]. Although the instruction semantics requires some extra work. Luckily, the reference model written for block-level verification can be reused inside the ISS. Since the spike ISS is open source, and the reference model is written in C, the only thing that remains is connecting to the simulators infrastructure.

The method used for modifying the ISS is as follows.

1. Specify mnemonic and bit fields, in a new file:
   `xxsl rd rs1 rs2 31..25=2 14..12=6 6..2=0x1D 1..0=3`

2. Generate headers, using RISC-V tools. These must be merged with the ISS' headers. E.g.:
   `#define MASK_XXSL 0xfe00707f`

3. Specify semantics. Here, just delegate the responsibility to the reference model:
   `WRITE_RD(sext_xlen(xmnlp_compute(1, 1, MNLP_OP_XXSL, RS1, RS2)));`

4. Attach to infrastructure: 1) add reference model to build scripts etc, 2) add mnemonics to ISS' internal list, 3) specify instruction format.

2.5 C Interface and Post-Processing

Using the mnlp module requires post processing to derive electron density, and using the core itself can be done in C instead of directly in assembly.

2.5.1 C Code

A C-interface was made to easy working with the ASM (assembly code). The functions are all written in assembly, hiding the details. And the interface simply looks like this:

```
uint64_t bot_asm (uint16_t x0, uint16_t x1, uint16_t x2, uint16_t x3);
uint64_t top_asm (uint16_t x0, uint16_t x1, uint16_t x2, uint16_t x3,
                  uint16_t y0, uint16_t y1, uint16_t y2, uint16_t y3);
```

The `top` function assumes that `bot` has been called first to initialize internal registers. Arguments are received via registers `a0` through `a7`. Even though the inputs are 16 bits, they are received as 32-bit words in the assembly code function, as per the ABI (application binary interface). Pairing them into 32-bit variables in C code would only complicate the assembly code handling.

Division One difficult aspect of producing the final result is the division at the end of the calculation. Before the problem domain was completely understood, this was implemented using C library functions. The difficulty of the problem domain is that the numerator and denominator has natural constraints on how they can relate to each other. Without knowing the specifics, it is difficult to exploit this relation. Hence, a general case of division must be assumed. An attempt at finding the extremas was done using partial derivatives and substitution, but the complexity was not easily surmountable.

Because a small 32-bit processor is used, the only way to do 64-bit division is using software routines. One can call upon this from within assembly by jumping to `__udivdi3` from libc. The verification and measurements would be the judge of this decision. See also Section 2.6.7 Complication: Division Time for a discussion of a solution to this.

2.5.2 Electron Density

With the aim of translating from LLS slope to electron density n_e, it was recognized that it is not necessary to do so within the in-flight computation. All the unique information is contained within the slope. Any remaining steps can be computed in post-processing, so long as this minimizes bandwidth usage.

A correctional scaling constant can be derived from the non-LLS 2-point equation. Recall that electron density being given by $n_e = \sqrt{C\Delta(I^2)/\Delta V}$. Focusing only at the division, as in Equation 2.6, it is possible to separate the ADC-given integer representation from the physical units and other system properties.

$$\begin{aligned}\frac{I_2^2 - I_1^2}{V_2 - V_1} &= \frac{\left(I_{2adc}\frac{dnr}{R \cdot bitspan}\right)^2 - \left(I_{1adc}\frac{dnr}{R \cdot bitspan}\right)^2}{V_{2adc}\frac{dnr}{bitspan} - V_{1adc}\frac{dnr}{bitspan}} \\ &= \frac{\Delta(I_{adc}^2)}{\Delta(V_{adc})}\frac{dnr}{bitspan}\frac{1}{R^2}\end{aligned} \tag{2.6}$$

The dynamic range dnr is 20 V ($\pm$ 10 V), the $bitspan$ is 2^{16} because the input data is 16 bit, and the resistance R is 4.7e6 Ω [37]. Other constants were already discussed in Section 1.4.1 Electron Density.

In this, the physical readings are represented as a ratio of its dnr given by the ADC reading and the available bitspan. An example is shown in Equation 2.7.

$$V_1 = \frac{V_{1adc}}{bitspan} dnr \tag{2.7}$$

Substituting these integer representations into the n_e equation, the post-processing procedure becomes apparent. See Equation 2.8. Even though this was derived from the 2-point calculation, the same constant is applicable to the 4-point LLS version.

$$n_e = \sqrt{C \cdot slope_{adc} \cdot \frac{dnr}{bitspan \cdot R^2}} \tag{2.8}$$

Necessary Number of Bits So, the integer-represented slope can be used to calculate n_e. But how many bits are required? The smaller the better for transmission, but too small and precicion will be sacrificed.

Consider first the relationship between the slope and n_e squared in Equation 2.9.

$$n_e^2 = slope \cdot C \tag{2.9}$$

Given a constrained scope on the n_e values of interest, namely $n_e \in [1e8, 1e12]$ (as discussed earlier). One can calculate the number of bits required to represent each of those limits. Limits of the slope is shown in Equation 2.10, and the integer ADC-variant in Equation 2.11.

$$slope := \frac{limit^2}{C} \tag{2.10}$$

$$slope_{adc} := \frac{limit^2}{C} \cdot \frac{R^2 \cdot bitspan}{dnr} \tag{2.11}$$

Required number of bits can then be derived using $\log_2$. See Equation 2.12.

$$N_{bits} = \log_2(slope_{adc}) \tag{2.12}$$

The upper limit of $n_e := 1e12$ needs the number of bits $\log_2(106252.25282) = 16.697$. Meaning that one needs a minimum of 17 bits to hold all of the information.

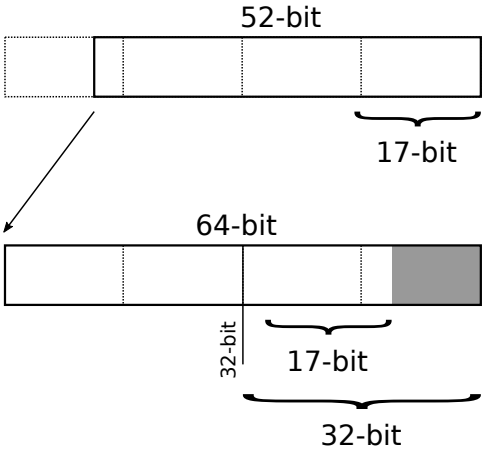
Likewise, the lower limit $n_e := 1e8$ is found to be $\log_2(0.0010625) = -9.8783$. This means that 10 bits are *missing*. And the lowest possible integer-slope must be 0.0010625, but this is way below what is representable as integers. A solution to this is to left-shift the numbers by 10 before dividing the slope. Then $0.0010625 \cdot 2^{10}$ becomes 1.0880. Any additional bits beyond this is advantageous when doing integer division.

2.5.3 Enhancement

The precision was improved. Thanks to the findings in the 2.5.2 Electron Density section, fine-tuning the relevant bits is now possible. The `top` value is 52 bits wide, but must be represented in a 64-bit variable and there is room to spare (12 bits). Left-shifting this value will increase resolution of integer divisions, but there might arise side-effects from this.

The upper limit of the slope requires 17 bits, and 12 bits are available for left-shifting. A total of 29 bits is within 32 bits, thankfully, because that preserves the goal of minimizing the transmitted data. The lower limit of the slope requires 10 additional bits to meet the minimum electron density. Which is now satisfied by the 12-bit left-shift. Figure 2.19 illustrates the point.

Figure 2.19: Utilizing excess bits.



2.6 Measurements

2.6.1 Cycle Counts

Cycle counts are a medium-independent measure of the duration of computations. The following is an explanation of the protocol used for measurements, while the results themselves can be found in Section 3.2.1 Cycle Count.

The code being run is that which constitutes a whole computation of LLS slope from a 4-probe input to the answer on the output. But the measurement is done in three steps: 1) first the `bot` (denominator) result is computed and measured, 2) then the `top` (numerator) result, and 3) finally the finishing division. These three measurements are done for three cases: A) plain integers, B) software-emulated floats, and C) Xmnlp-accelerated integers.

Manually observing waveforms is a hassle. Instead, a memory mapped time-stamper was set up. During simulation, the functional module serving as RAM treats certain accesses as timestamp commands. Hence, before and after a full calculation, a timestamp can report the current number clock cycle (or rather simulation tick).

One problem is that jumps, memory access, and returns take a few cycles themselves (Figure 2.20), making the measurement unjust. This could simply be subtracted from the measurements. But they are all included in these measurements for the following reasons: jumps are likely to be part of real life usage, and counting the timestamps makes for a more conservative measure (less optimistic results).

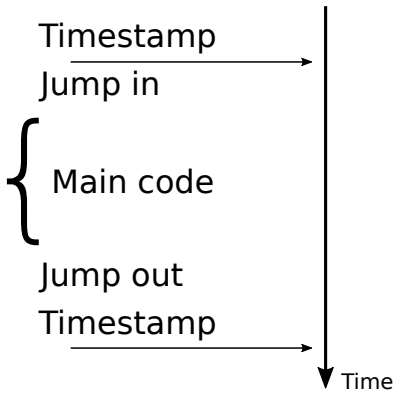


Figure 2.20: Timestamping and jumps add to the measurement.

Being hand-written, the assembly program has an advantage in its compactness and frugal concerns for time, with careful management of register use. On the other hand, the C code has the benefit of compiler optimizations (`-O3`). Integer calculations in C is done with as small variable type sizes as possible, but there is a disadvantage of having to use 64 bit variables (`uint64_t`) even for 36-bit numbers. The `floats` calculation is expected to be much slower, and is included in part as a curious reference.

2.6.2 Synthesis

Even though cycle count gives a measure of whether timing has improved. Two more things are important in the interest of timing: 1) did modifications ruin the maximum clock frequency, and 2) is the RI5CY core running on an FPGA good enough for m-NLP needs?

At first an Altera DE1-SoC board was used, but the development tool did not support some SystemVerilog keywords which were used in the core. For

example the `inside` keyword. So instead, a ZedBoard Zynq was used, along with the Vivado software tool [38]. At times, the open source tools Yosys and Icetime [39] were also used for quick synthesis and timing profiling.

Top Level The top level of the synthesis is done very simple in order to avoid complexity to sneak in and introduce a mistake somewhere. That means, only instruction and data memory connection (in addition to clock and reset) where routed to the IO of the FPGA. Interrupts, memory protection, i.a. was hard coded to not be enabled. Even if assuming that these would impact the critical path, the difference from unmodded vs customized core is easier to compare with only the bare essentials of signals. Signal connections are shown in Figure 2.21.

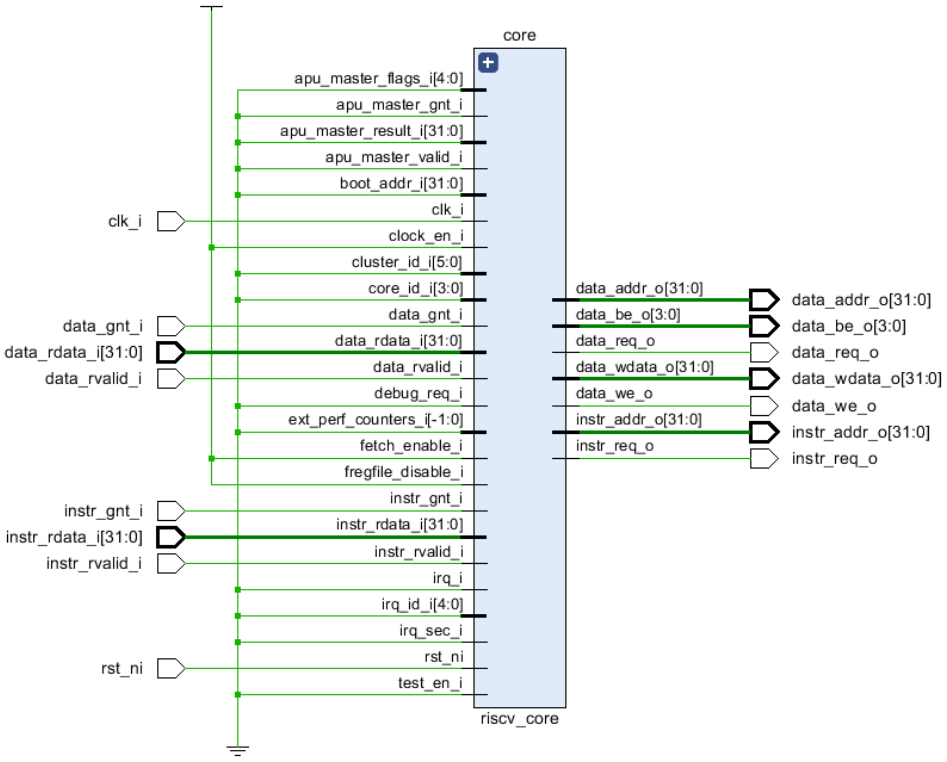


Figure 2.21: Pinout of top-level module used in synthesis and timing analysis, as produced by Vivado.

When this is run in synthesis, the top-level ports will get assigned to IO pins of the FPGA, and port delays are specified for use in static timing analysis [40]. In contrast to hard-coding the ports which lead to too much logic being optimized away, and leaving ports unconnected which *removes* logic[11].

Test Setup The method for finding the maximum frequency has a misleading pitfall that must be avoided. When a design is under-constrained,

[11] I did both of these mistakes :) Much time was spent on those blunders.

Vivado will lazily just barely meet the timing requirement and the reported frequency is not the maximum possible. If instead, the design is over-constrained, the tool will try its best up until a certain limit of effort. Through much experimentation with the tool, this was the most reliable method for finding the maximum possible frequency.

Comparisons towards sample rate is done by combining the cycle count with the maximum clock frequency. This rate of computation is also compared against all three sampling rates of interest: 1 kHz, 6 kHz, and 20 kHz. This is to see which of the targets, if not all, can be achieved.

2.6.3 ICI-2 Reference

Data from a real rocket launch has been gathered and is used here for testing purposes. The data is from the ICI-2 rocket launched in 2008 [37]. Figure 2.22 shows the data converted to electron density, both unfiltered and with filtering *before* computing the slope.

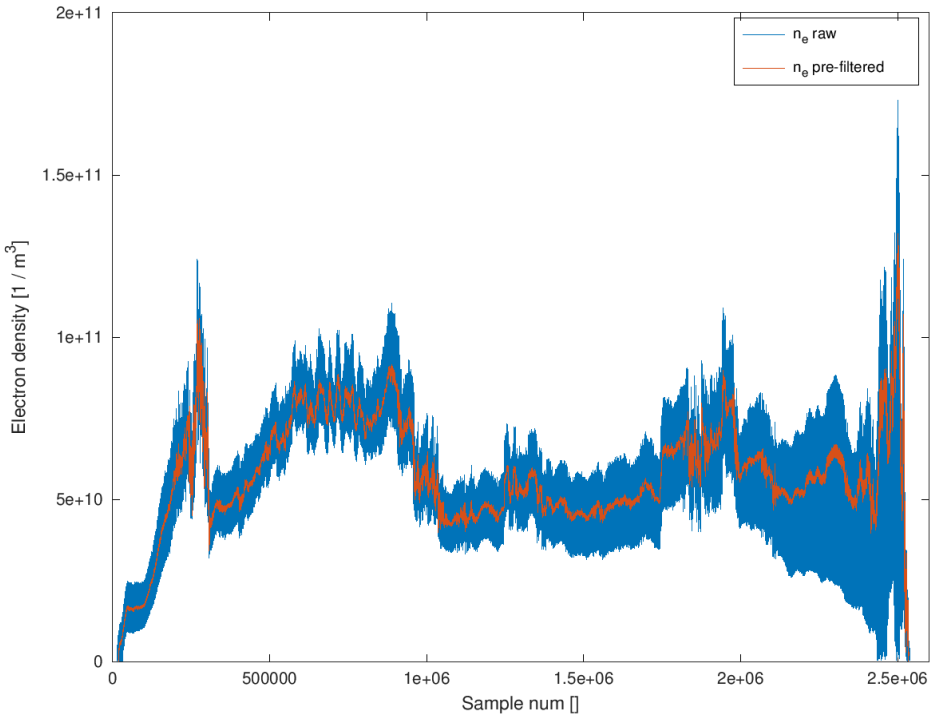


Figure 2.22: Electron density gathered from ICI-2 sounding rocket [37]. The blue plot is calculated from raw data, while the orange one has rocket spin filtered out beforehand.

Electron density in this sample set spans from $5.5964e7$ to $1.3200e11[1/m^3]$. This is counted after cropping away the noise of initiation and closedown (otherwise, NaN would occur). This is of relevance for whether or not the data set is well suited as a test input for trustable results. Indeed, it is "real" data, but factors like gain and startup time can have an impact. It is a bit surprising that the data shows a density *below* that which is required from target specs, but this might be data from before entering a valid area. The

maximum of this data set does not stimulate the maximum of the target range. Usefulness of the set perseveres because of other properties like e.g. its derivative. The minimum and maximum values are shown in Table 2.4.

Table 2.4: Extrema of ICI-2 data set $[1/m^3]$.

Measure \ Source	ICI-2	Target
min	5.5964e7	1e8
max	1.3200e11	1e12

2.6.4 Fidelity Measures

To be on the safe side, the data from the ICI-2 (cf. the ICI-2 Reference section) launch is not trusted to be a comprehensive enough set of stimuli. It does test a broad variation, but as an addition a more targeted input set is generated.

A collection of four data points have both a degree of spread and a position between max/min of the range of 16-bit integers. Hence, the generated stimuli uses a mix of narrow spread (e.g. 1, 2, 3, 4) and wide spread (e.g. 1 up to int16 maximum). These different spreads are then distributed over the available range (e.g. 1000, 1001, 1002, 1003). Figure 2.23 illustrates the idea of having sets of inputs 1) bundled tightly together vs spread out, 2) cover low, medium, or high values, 3) use steep and level incline, and 4) combining these variations for both axes. The reason is to cover as wide variation of inputs as possible to test every corner of the system.

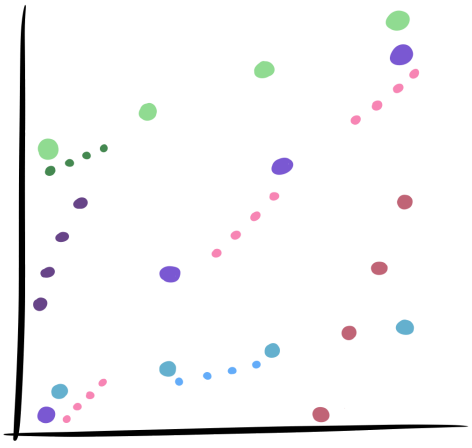


Figure 2.23: Artist rendition of spread stimuli. X-axis is voltage inputs, and Y-axis is current inputs. [by anonymous donation]

Generated data has three parts to it: 1) deliberately spread data, 2) randomized (but monotonous) data), and 3) a selection of the ICI-2 data. The whole set of input stimuli is then a congregation. The randomized input is made sure to be all monotonic, because a negative electron density is not

supported. Finally, this input set is applied in a "cross" so that each set of 4 values is applied to the x-axis, and for each such x-value *all values* are run on the y-axis. Meaning there is $quadruple^2$ tests to run. Figure 2.24 shows the electron density derived from the generated input stimuli.

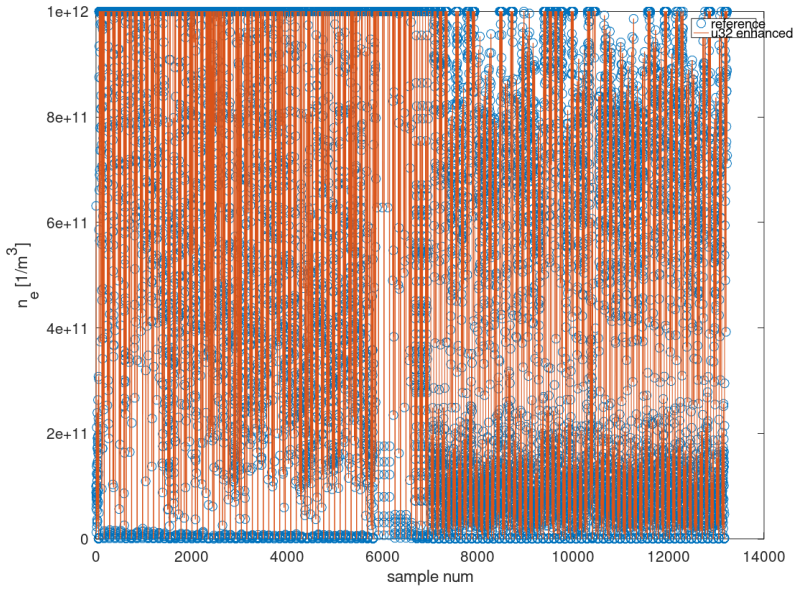


Figure 2.24: Electron density from generated stimuli.

2.6.5 Complication: Critical Path

Problem A problem was revealed when running synthesis and timing analysis. Resource use was suspiciously low, and timing results were optimistically high. The wrong ports where tied to constants, and other wrong ports where unconnected. Refer to Figure 2.21 for the correct setup.

After correcting this, the true critical path was revealed, and patches where needed. Some of the multiplications in the mnlp module took too long.

Solution One solution would be to write a custom multiplier and pipeline it. A simpler solution was chosen, namely to derive the main clock and let the multiplication run on half the speed. In the meanwhile, the processor pipeline may stall for one extra cycle.

Figure 2.25 shows the main `clk` and the derived `clk2`. The derived clock was implemented as an enable bit. In the figure, there is also shown an incoming `long` instruction, indicating that the requested operation is of a type that needs and extra cycle. Below that is a `ready` signal that can stall the pipeline so that no new instructions are requested. In the figure, this shows the *intended* behavior of such a ready-signal.

In Figure 2.26 the `long` signal arrives *in the middle of* a `clk2` pulse. The chosen solution was to wait one cycle and continue as normal from a

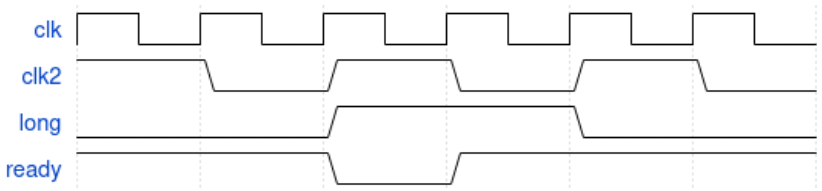


Figure 2.25: Timing diagram for `ready` logic, when long instr arrives at a clk2 posedge.

positive edge of the derived clock.

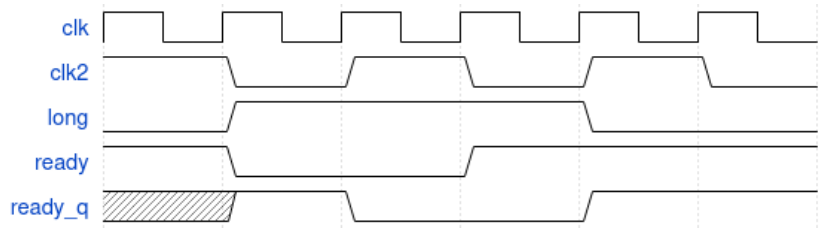


Figure 2.26: Timing diagram for `ready` logic, when long instr arrives in the middle of a clk2 cycle.

Logic to control the ready signal can be derived from the signals `clk2`, `long`, `ready_q`. Where `ready_q` is a registered version of the ready signal as it was on the previous `clk` edge. Figure 2.27 shows a highlighted truth table which unites the two signal traces shown above. The resulting logic is simply: `!ready = long && (clk2 || ready_q)`.

clk2	long	ready_q	ready
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.27: Truth table of not-ready signal, in relation to ”long” instructions.

2.6.6 Complication: Multiplier Synthesis

Another problem presented itself shortly after. The synthesis tool in Vivado refused to generate logic for the multiplication, due to it being a ”wide multiplication”. This might have been a licensing issue or a configuration issue.

The solution that was chosen was to build a shift-and-add multiplier. Performance-wise it may pose as a penalty, but that will show up in timing analysis. And there is no guarantee that it is portable.

2.6.7 Complication: Division Time

The final part of the computation is a division. Since no custom instruction had been written for this, software routines were used instead. Problem is that it takes so many cycles to complete that the gains of the custom instructions becomes comparatively less significant (Section 3.2.1 Cycle Count).

The solution was to write an unsigned integer long division circuit. This follows the same principle as doing long division by hand on paper. The method takes exactly as many cycles to complete as there are bits in the numerator.

This adds a new instruction to the set, `slope`. It takes two words which together comprise the `top` numerator. The `bot` denomenator is handled internally. As a result, the whole computation can be done without any of the x-side instructions.

Thanks to the testbenches, implementing it with certainty of its functionality went fast and smoothly. Its signaling is a one-pulse `start` along with `numerator` and `denomenator` signals. When the division is done, the results are given in a `result` signal and a `valid` signal lets the supervising block proceed. Including the divider in the mnlp module follows roughly the same principles as in Section 2.6.5 Complication: Critical Path. For example, the start signal is given by:

```
!clk2_q && (operator_i == MNLP_OP_SLOPE) && !ready_d1 && !divrunning_q
```

Chapter 3

Results

A system has been made and this chapter puts it to the test in regard of functionality, performance, and other qualities.

3.1 Functional

This section is about how well the output of the mnlp module matches its intended purpose. It discusses the functional quality characteristics as discussed in Section 1.6.1. Precision and error margin is the aspect that gets the most attention here.

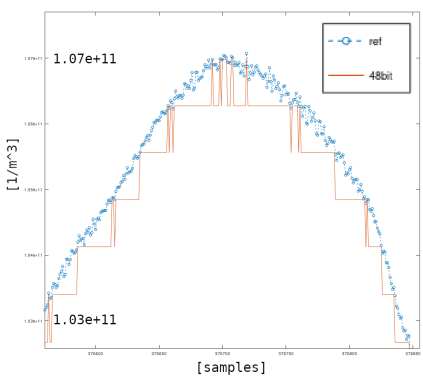
3.1.1 Correctness

The following are measurements of system precision. Outputs from the system is post-processed using Octave/Matlab. The results are then compared with reference solutions done purely in Octave/Matlab. Since the system uses only integer arithmetics, it is expected that Octave solution does not match exactly. The point is to inspect whether there is incongruity between target requirements and the systems intrinsic uncertainty.

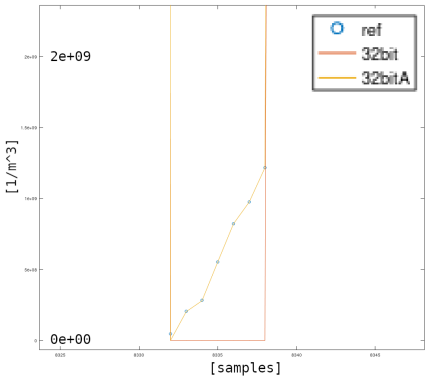
Integer calculation of `top` and `bot` (cf. Figure 2.6) has no error. Simulation results show that the output is 100% as desired. This is a trivial truth, because all operations up until that point is purely additive (multiplication, squaring, left-shift, and addition) and the number of bits have been carefully managed. The loss of precision begins in the division that follows (excluding the precision that is given by the ADC and front-end). Figure 3.1 shows the problems of certain types of division.

Different techniques for division was experimented with. For instance using 32-bit output, 64-bit output, and single-precision floats. A requirement is that the output is 32-bit so to actually have any effect on the down-link bandwidth limit. The absolute error of these divisions are shown in Table 3.1. All of these divisions are done as 64-bit `libc` software divisions, except the float one. The difference lies in how either the inputs or outputs are treated.

The unsigned 32-bit result (u32) variant is just an AND-mask applied to the u64 results. The ”enhanced” variant uses a left shift before dividing.



(a) A futile attempt at reducing bits by right-shifting.



(b) Loss of precision thanks to integer division.

Figure 3.1: Pitfalls of division with insufficient bit-depth; Electron density.

Table 3.1: Absolute errors for different division types.

measure \ division	u64	u32	u32 enhanced	float
max error	3.0448e+09	3.0448e+09	4.6785e+07	5.0251e+06

48-bit and a 16-bit bit-shifted variants were abandoned. And the floats are single-precision. Some interesting observations can be made from this:

1. u64 and u32 results are equal. This is because of the nature of the system as discussed in Section 2.5.3.
2. The non-enhanced variants are *above* the required resolution of $1e8[1/m^3]$, and are not acceptable.
3. Floats are the best, despite them not being optimized for the specific range of interest.
4. Enhancing the integer division gives a satisfactory resolution of $4.6785e7$.

Figure 3.2 illustrates the precision of the different methods along with the target.

Coverage All the testbenches run with *zero* errors. That is, the testbenches that check instruction-level correctness, and not the tests of n_e precision. This has been an implicit requirement all along. To back this up, Table 3.2 summarizes the coverage.

As can be seen, there is near 100% coverage on the coverpoints and crosses that were specified. However, this does not take into account metrics like branch coverage. The `en` coverpoint at 50% falsely indicates a defeat, where in reality it only reminds that this particular point is nonsensical. The `operatorXnonzero` cross at 80% is also not too worrisome, because an output of zero is statistically not so easy to hit.

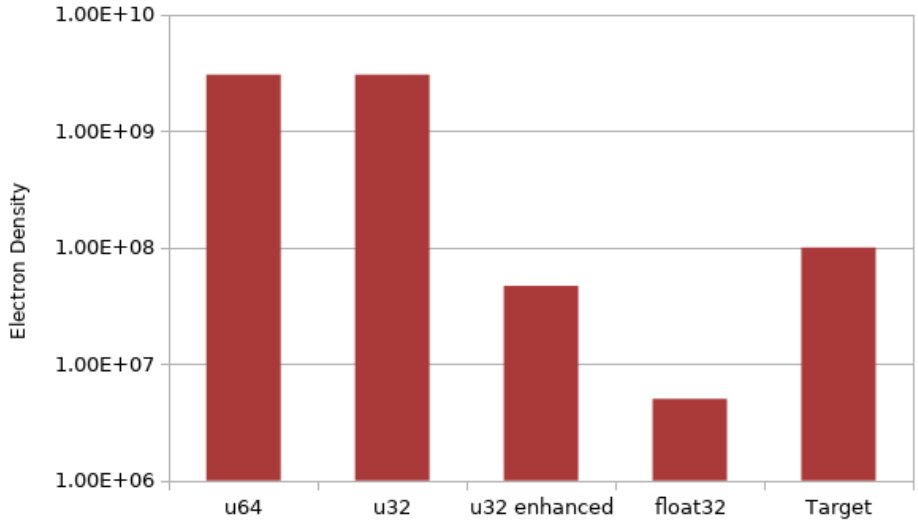


Figure 3.2: Comparison of lowest detectable change in electron density.

Table 3.2: Coverage results.

Covergroup	Metric	Status
Coverpoint cg::rst	100%	Covered
Coverpoint cg::en	50%	Uncovered
Coverpoint cg::operator	100%	Covered
Coverpoint cg::op_a	100%	Covered
Coverpoint cg::op_b	100%	Covered
Coverpoint cg::result	100%	Covered
Coverpoint cg::overflow	100%	Covered
Cross cg::opaXopb	100%	Covered
Cross cg::operatorXoverflow	100%	Covered
Cross cg::operatorXnonzero	80%	Uncovered

All of this is no guarantee that the covergroups were wisely selected or is a reliable testament to the correctness of the device, but it is a good indication.

3.1.2 Completeness

A lot of parameters and aspects are involved in usage of the m-NLP system at large. Not all of which has been included in the scope of this project. A summary of features are shown in Table 3.3. This summary includes both those features which have been successfully implemented (marked with ”Yes”), and some that have been dealt with in other works but not in this project (unmarked). Some of the features which are not implemented might be possible to do in post-processing or is possible to implement in RTL without too much overhead.

Table 3.3: List of implemented features.

Feature	Implemented
Calc electron density	Yes
Calc correlation coeff	
Heed ne min/max	Yes
Adjustable no. probes	
Adjustable adc bitdepth	Yes
Adjustable gain	
Adjustable offset	
Adjustable voltage range	Yes
Adjustable beta factor	
Handle invalid channel (3 probe)	
Heed probe size	Yes
Pre-filter spin	
Post-filter spin	Yes

3.1.3 Appropriateness

Functional appropriateness is difficult to assess. The custom instructions does yield electron density as wanted, so in this regard the system is appropriate. Resolution is also within the targeted bounds which were gathered from dependable sources. This matter is further covered in Chapter 4 Discussion.

3.2 Performance

Again driven by the system's qualities plan from Quality Characteristics. This section covers time and resources. The timing aspects of the device is measured, both in generalized and absolute terms of speed. The resource utilization is also measured. However, excluded from the discussion is the system's capacity for concurrency and storage and similar, as it is hardly of relevance.

Timing measures check whether the custom instructions provides any speed-up vs a plain C approach, and whether it fulfills timing requirements. A speed-up should be substantial for the laborious exercise of customization to be justifiable. Also, the resource utilization should not have grown cumbersome.

There are two sides to speed: maximum clock frequency, and time to complete a whole computation. Maximum clock frequency need not be high, as long as the time to complete a computation is low. One whole computation must fit within the timing requirements for sampling frequency. Both clock frequency, the architecture of the customization, and the software plays a role in the total time. A benefit of exceeding expectation is to facilitate for a potential gain in spatial resolution.

3.2.1 Cycle Count

Computation time can be given in a relative measure, namely the number of clock cycles. It is a generic measure that is independent of the implementation medium, although it is still bound to the architecture which run the code. Such a measure will be a fixed number regardless of clock frequency. In fact, it will be a determining factor in deciding the minimum clock frequency. This measurement is also very useful for justly comparing two different algorithms or two different platforms. "top" and "bot" is a shorthand for the numerator and denominator as used in Figure 2.6.

Findings Here are the resulting cycle counts found when running the program as described in Section 2.6 Measurements. Cycles are derived from the simulation clock period of 10 ns. *Int* tests are done using integer types in C, *SW float* is done in C and ran without FPU[1], *Custom ASM* are computations done using the custom instructions.

Results from the "bot" calculation is shown in Table 3.4. The value from this computation is only necessary to calculate once, when the probe biases are sat. Hence, however great the speed-up is, it will not have a significant impact. Unless, if future m-NLP systems will have more fine-grained control of the probe biases, then this timing result would be of more significance.

Table 3.4: Cycles to compute the "bot" part of the full calculation. Parenthesis show values from before the 2-cycle multiplier was introduced.

Program	Cycles	Ratio
Int	50	1
SW float	1298	26
Custom ASM	34 (29)	0.68 (0.58)

The "top" calculations are a more critical piece, as they are run for each new incoming sample. Unfortunately the speed-up here (20% reduction) is not as good as the "bot" calculation (32% reduction). Table 3.5 summarizes the results.

Table 3.5: Number of cycles used to compute "top" part. Parenthesis show results from before the new multiplier was installed.

Program	Cycles	Ratio
Int	92	1
SW float	2017	21
Custom ASM	74 (73)	0.80 (0.79)

Depending on the input values, the software division algorithm varies in the amount of time it needs. Inputs used in the C case are a cross sweep of the numerator range (52-bit) and the denominator range (36-bit), i.e. `for i in 1:step:max52bit` crossed with `for j in 1:step:max36bit`. Table 3.6

[1]The same effect can be achieved on a processor *with* an FPU, if one compiles with flags that disallows floating point instructions.

summarizes the division measurements. Originally, the C division was used also for the final step of the custom assembly version. With the latest `slope` instruction the divion has a fixed time, for either 52-bit or 64-bit. Both are included, but since the time is fixed a mid-range value is of no interest.

Table 3.6: Minimum, maximum, and mid-range (aka mid-extreme) of division cycle counts.

Type	Min	Max	Mid-range
Int	41	195	118
SW float	95	2090	1093
Custom ASM	67	79	N/A

Figure 3.3 shows the time of a full computation with and without the custom instructions, revealing the Achilles' heel of the system. That is, even though "top" and "bot" calculations have achieved a speed-up, the division part of it evens out the differences. "bot" is only calculated once by itself, while "top" and the division is done together repeatedly.

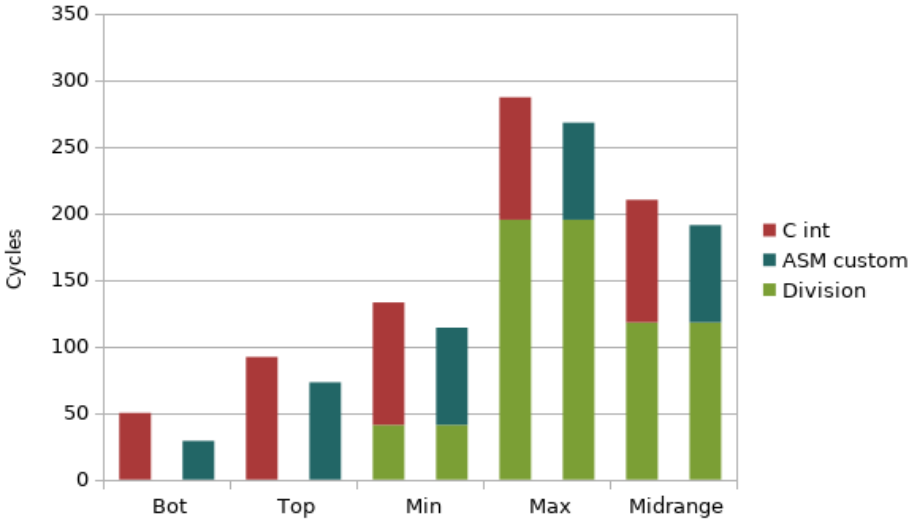


Figure 3.3: Completion times comparing C and ASM variants, using C division. Time of division in C depends on given inputs.

When measuring the fixes introduced in Section 2.6.7 Complication: Division Time, the division has a much lesser impact. Figure 3.4 shows the difference. The "bot" computation is not included because it is now part of the `slope` instruction doing the division. As a result, the division time is almost halved, from 287 cycles (worst case C) to 153 cycles (64-bit custom assembly). The best case C is faster still, but 1) if this was invoced by realistic inputs then the ASM division can also be improved accordingly, and 2) if realistic inputs does vary like this then the ASM division can be easily modified in the future to use fewer cycles based on the first 1-bit in the numerator.

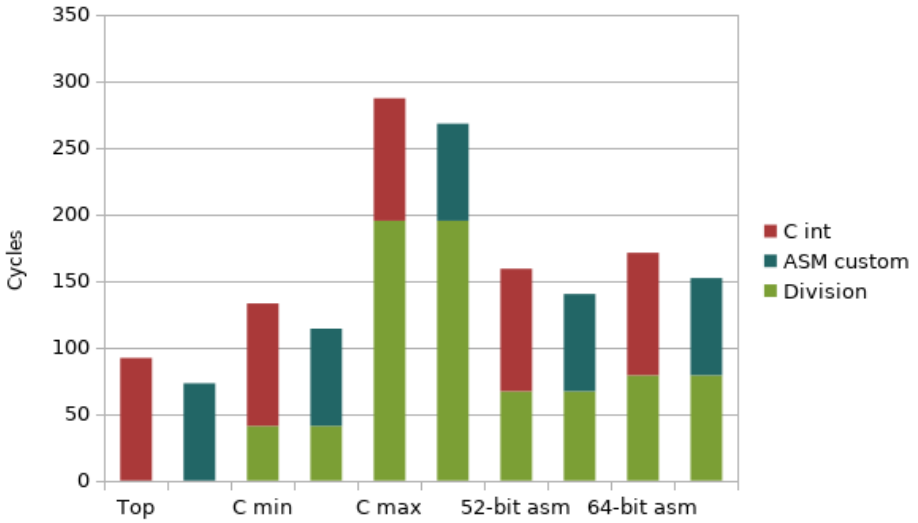


Figure 3.4: Completion times comparing C and ASM. C max is division in C, and the two others are in custom ASM.

Hardware Comparison Kosaka [4] ran the routine on a TMS570LS1224 microcontroller. They did 100000 samples and measured the time it took with and without FPU at different frequencies. Comparing the speed of two different hardware platforms directly does not compare the algorithms themselves as good as cycle counts does. Since they have both specified the clock speed and also measured the time of execution, the general measure of cycle counts can be derived (Equation 3.1).

$$\text{Cycles}_{comp} = \frac{\text{Time}_{tot}}{100000} f_{clk} = \text{Time}_{comp} \frac{\text{Cycles}}{\text{Time}} \tag{3.1}$$

Table 3.7 lists the scores. The results show some inconsistency, in that the clock frequency appears to affect the cycle count. This is possibly due to the uncertainty of the timing function used. The results are still usable for a comparison.

Table 3.7: Cycle count of one electron density calculations on an ASIC processor.

100 MHz		160 MHz	
FPU	No FPU	FPU	No FPU
560	4030	676	5019

This hardware comparison is only useful as an indication and not an absolute measure. This is because there are too many differences in the examined objects. Kosaka's process seem to include the full electron density calculation in-flight, whereas I use post-processing to take the square root and multiply with the correctional coefficient. The performance will differ depending on whether e.g. the scaling factor is pre-calculated, if n_e is cropped to a maximum and minimum value, and if there is applied scaling

to fit the correct number of bits.

Figure 3.5 shows the cycle counts of 1) hardware integers (custom instructions) and 2) hardware double-precision floats, compared vs 3) software single-precision floats, and 4) software double-precision floats. Kosaka had fixed input values (not varying them), and did not do a plain integer calculation. There are some interesting things to note. Their emulated double-precision floating points (4500 cycles) are comparable to my single-precision floats (2500 cycles). Additionally, my plain integer calculation at ~ 150 is faster than their FPU-backed >500 cycles.

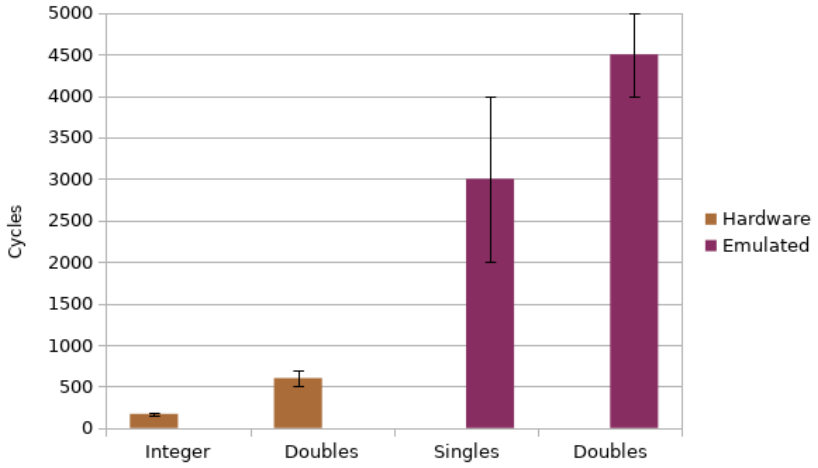


Figure 3.5: Cycle count estimates of a full computation with hardware integers (Xmnlp) and doubles (FPU), and software singles and doubles. All the "doubles" results are derived from Kosaka [4].

3.2.2 Timing Analysis

An FPGA was used to synthesize and implement the core. How the synthesis was setup has already been described in Section 2.6 Measurements. The intention is twofold: 1) to see whether the system requirements can be reached as an FPGA softcore, and 2) to see if the modifications introduced a new critical path. Whether the FPGA is sufficient for speed requirements is of interest for future mission hardware choices. And if there is no new critical path, that means that the entire cycle count speed-up can be utilized without having to sacrifice clock speed.

According to timing requirements (discussed in Section 1.6.2), calculations must fit within a certain time slot. For a given sample rate, the period time must house a complete computation. Not only that, but part of the time might be taken up by transmission and other tasks. The computation time must therefore be so much smaller than the sample rate as is possible, $f_c/f_s > 1$.

The number of cycles for a whole calculation (worst case), plus a safety buffer, must fit within the available time. Number of cycles, times target

sample frequency, gives the minimum clock frequency (Equation 3.2).

$$\min(f_{clk}) = \text{cycles}_{comp} \cdot f_{sampl} \tag{3.2}$$

For example, 150 cycles targeting 20 kHz requires a minimum clock frequency of 3 MHz. But this is just the slowest acceptable pace, and it is also interesting to know if it can go faster.

Analysis Results The design could be synthesized at a clock period of $23ns$, about $43.5MHz$ (Figure 3.6). Since the synthesis tool does not try to zealously optimize beyond the specified constraints, one can only find the maximum frequency by over-constraining the design. Constraining for $22ns$ fails in timing analysis (Figure 3.7). Warnings from synthesis and implementation has been scrutinized (to the best of ability) to ascertain the validity of the results (see Section 2.6.5 Complication: Critical Path).

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0,118 ns	Worst Hold Slack (WHS):	0,075 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7532	Total Number of Endpoints:	7532

All user specified timing constraints are met.

Figure 3.6: Timing is OK at 23 ns clock period.

Setup		Hold	
Worst Negative Slack (WNS):	−0,967 ns	Worst Hold Slack (WHS):	0,065 ns
Total Negative Slack (TNS):	−94,684 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	351	Number of Failing Endpoints:	0
Total Number of Endpoints:	7535	Total Number of Endpoints:	7535

Timing constraints are not met.

Figure 3.7: Timing FAILs at 22 ns clock period.

Measurements of speed was also done on the unmodified core, *before* any modifications. Before applying the Xmnlp modification, the plain core was capable of running at $22ns$, about $45.5MHz$. But this need not mean the critical path is within the mnlp module itself, as the control logic can have added to another path; as is discussed shortly.

Critical Path But indeed, as is illustrated by Figure 3.8, the new mnlp module *is* the holder of the critical path. Although not by much! The worst slack is at $0.118ns$ while second place is at $0.147ns$. And the podium is even shared between `mnlp`, `mult` (the core's multiplication unit), and `alu` (the core's ALU).

Name	Slack ^1	Levels	High Fanout	From	To
Path 1	0.118	31	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mnlp_operand_a_ex_o_reg[24]/D
Path 2	0.147	31	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_dot_op_a_ex_o_reg[24]/D
Path 3	0.244	31	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/alu_operand_b_ex_o_reg[24]/D
Path 4	0.302	32	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/alu_operand_b_ex_o_reg[30]/D
Path 5	0.309	32	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_dot_op_a_ex_o_reg[31]/D
Path 6	0.317	31	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_operand_a_ex_o_reg[24]/D
Path 7	0.340	32	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_dot_op_a_ex_o_reg[26]/D
Path 8	0.345	32	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mnlp_operand_b_ex_o_reg[30]/D
Path 9	0.361	31	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_dot_op_b_ex_o_reg[24]/D
Path 10	0.362	32	161	core/id_stage_i/alu_operator_ex_o_reg[2]/C	core/id_stage_i/mult_dot_op_b_ex_o_reg[16]/D

Figure 3.8: Worst near-offending timing paths in the design (custom core at 23 ns).

Findings Comparing the results to the requirements. Table 3.8 tallies the conclusion. For a sample rate of $20kHz$ a clock frequency of $3.06MHz$ is needed (at a cycle count of 153), but the design is actually capable of running at $43MHz$. That yields a computation frequency 14 times above the sampling frequency. Meeting the bonus guidance targets would have been enjoyable, but is mostly inconsequential.

Table 3.8: Meeting clock period requirements.

Target	Clock (MHz)	Achieved (Y/N)	Ratio (%)
1 k f_s	0.15	Y	286
6 k f_s	0.92	Y	46.7
20 k f_s	3.1	Y	13.9
(actual)	43	-	1
Guide 1	100	N	0.43
Guide 2	160	N	0.27

FPU-Enable An attempt was also made to run static timing analysis while the FPU is enabled. The unmodified core was used for this, and the only change made was enabling a parameter at top-level and including the FPU source code.

Surprisingly, the timing deteriorated substantially. Worst case slack was at -25 ns, while the synthesis and implementation aimed at a 21 ns clock period. This indicates a minimum period of at least 46 ns, yielding a maximum frequency of 21 MHz. Note that nothing else was changed but enabling the FPU flag. If additional configuration could fine tune the results, that is not accounted for here.

All timing requirements and results are summarized in Figure 3.9.

3.2.3 Resource Utilization

Resource utilization of the FPGA is not a critical metric in and of itself. Though it is very interesting to see the impact of the customization, and even to see how demanding the unmodified core is. The following section

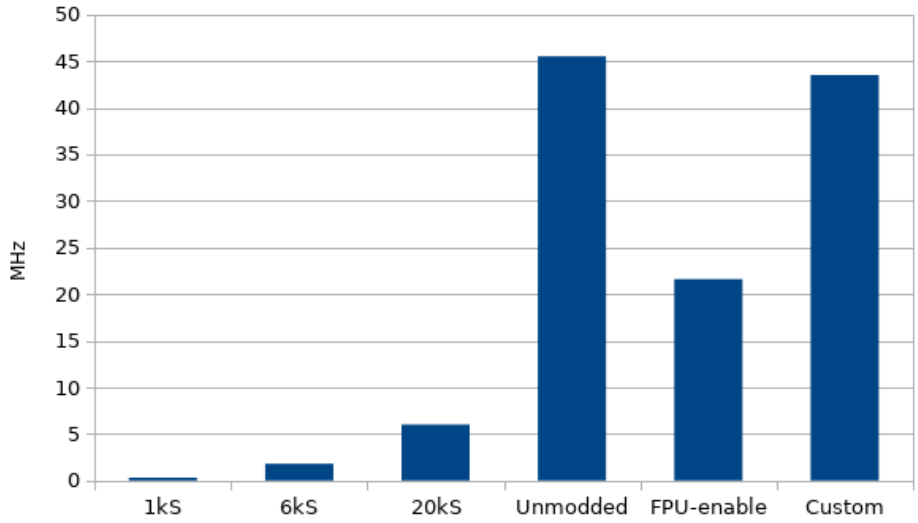


Figure 3.9: Timing requirements for 1k, 6k, and 20k samples/sec, and max frequency of three configurations of the core.

summarizes area usage and other FPGA resources (which also gives a slight hint towards ASIC use). Power use and similar has not been regarded.

Unmodded "plain" core Baseline resource utilization for an unmodified RI5CY core is shown in Figure 3.10. This is the result of place and route.

Resource usage is surprisingly low in light of the apparent area use illustrated in the place and route results. In Figure 3.11 it is shown that lookup tables and flip-flop use has hardly made a dent. Since memory protection (PMP) is turned off, along with interrupt handling and a few other extraneous features (for this purpose), the low resource use represents a pretty minimal core. Lastly, because no internal block ram was set up, IO pins comprise the main bulk of the resource use.

Customized core Figure 3.12 shows the corresponding results of place and route for the customized core. It is still well within reasonable limits.

IO usage and flip-flops remain the same, as shown in Figure 3.13. However, the lookup table is slightly up to 13.81% from 12.14%, as could be expected from *adding* logic. It is interesting to note that the mnlp module yields a 14% increase in LUT use. Another thing to observe is that the DSP (digital signal processing blocks) went up to 5% from 3%. This increase did likely get a significant contribution from the DIY multiplier that was written.

FPU-Enable The floating point unit was enabled, just like in Section 3.2.2 Timing Analysis. Look up table and DSP block use was the only ones to change noticeably. LUT usage at 19% is the biggest one of all three, not surprisingly. DSP use was at 4%, indicating that the DIY multiplier was not so frugal in resource use.

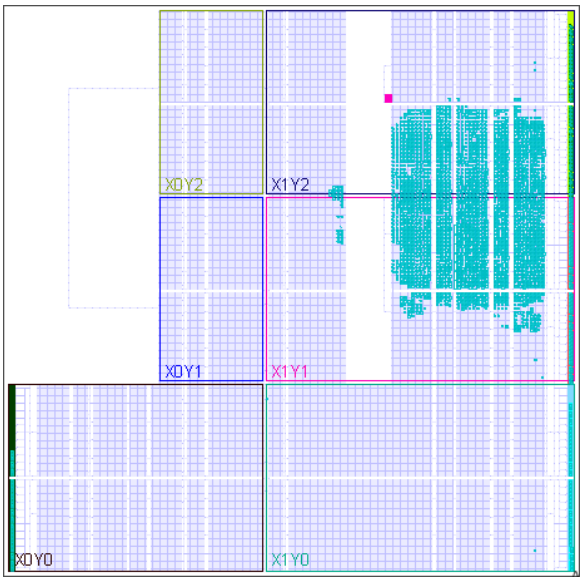


Figure 3.10: Place-and-route results for *unmodified* RI5CY core targeting 22ns clock period.

Resource	Utilization	Available	Utilization %
LUT	6460	53200	12.14
FF	2263	106400	2.13
DSP	7	220	3.18
IO	173	200	86.50

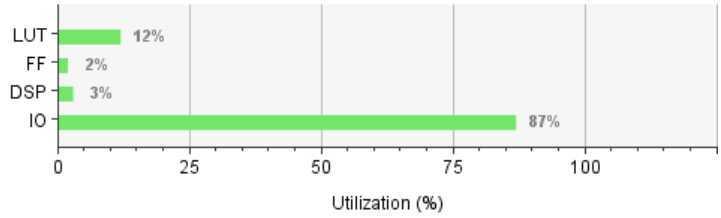


Figure 3.11: Summary of resource utilization, unmodded core at 22 ns.

3.3 Further Qualities

The most prioritized system's qualities have been discussed above, functional and performance qualities. Whereas some other significant properties have not been discussed yet. And those are: usability, portability, maintainability, and reliability. They are only tersely review in this section.

Usability The following is a self-assessed review of the project's usability.

In terms of operability, the system is fairly easy to use. Reasoning about the instructions' interplay can be challenging, and assembly programming is not easy in the first place. But calling upon C functions makes the system quite trivial to operate.

Learnability depends on how deep one looks. Simply writing code using the system is not so difficult. Understanding the signaling of the RTL

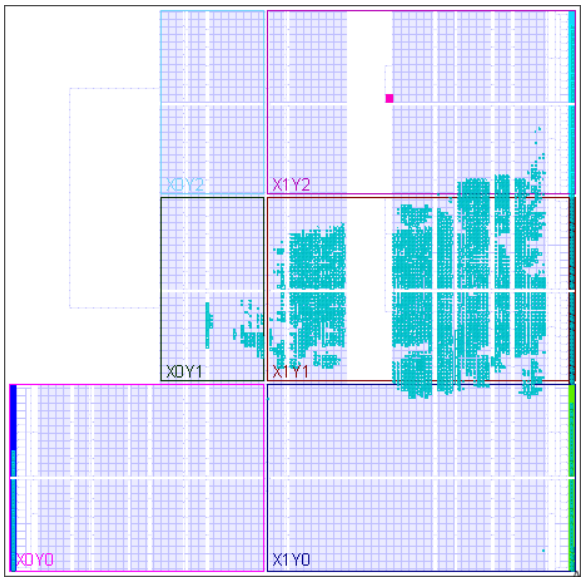


Figure 3.12: Place-and-route results for *customized* core at 23 ns clock period.

Resource	Utilization	Available	Utilization %
LUT	7346	53200	13.81
FF	2414	106400	2.27
DSP	12	220	5.45
IO	173	200	86.50

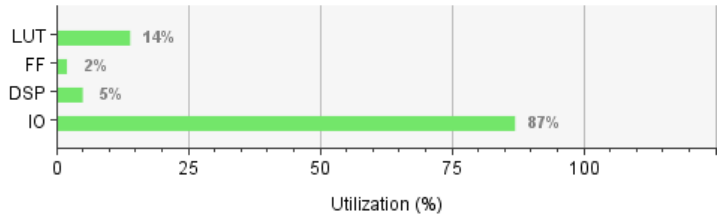


Figure 3.13: Summary of resource utilization, custom core at 23 ns.

module should not be too hard either, but understanding the possible states of the module is not immediately obvious. As long as one bases one's reasoning on the computational flow as shown in Figure 2.6 (Section 2.1.5) much confusion can be avoided. The verification infrastructure, on the other hand, is at times more complex than the module itself.

It is hard to gauge how easy it is to recognize the appropriateness of the system is for a use case. This is discussed more in Section 4.2 Conclusion.

Portability Portability regards how easy it is to fit the system into a new environment.

The installability of the system proved to be a relatively smooth experience, as detailed in Section 2.2 Implementation: RTL Code. If more time was available, and attempt could have been made to fit Xmnlp into the

Simple Hex Instruction Transputer. The most peculiar about the module is that it depends on stalling the pipeline with a ready signal. Supporting new opcodes and routing the control signals depends on the given core and its flexibility regarding change.

Replaceability is rather not so good. Although adapting to software changes should be easy, the hardware is more challenging. This is discussed more in the Conclusion (Section 4.2).

The systems ability to adapt to changes in its environment depends. Opcodes and control signals are most vulnerable to change. Especially when the core is a separate project from Xmnlp, and does not care or know about keeping changes compatible. The custom assembly mnemonics currently only supports `gcc`. Considering all of that, the interface to the module *is* simple, so it is not unreasonable to expect it to tolerate environmental changes.

Maintainability Maintainability is, roughly, about how easy it is to modify the system, fix bugs, and add new features.

Analysability regards the ease of failure analysis and anticipating the impact of making changes. The source code is cleanly written but the logic is not obvious. Separate functionality is organized in SystemVerilog "tasks", and there is a clean separation between combinatorial and sequential code. Implementing a new operator does not involve many steps at all. Understanding the ecosystem it lives within may require understanding of RISC-V and involves using several software tools.

Testability, on the other hand, is regrettably not so good. A trade-off was committed for optimization; to be a bit cryptic and specialized, at the cost of not being general and simple. Some of the instructions that were implemented, have side-effects within the computational module. This makes testing harder, because the observability of the instructions effect depends on interplay between several instructions.

Reliability The only reliability aspect deliberately pondered in detail is that of fault tolerance. And the answer is simple: 1) it is the responsibility of the decoder stage to ensure that signaling is correct, 2) the responsibility of the programmer to rationally follow their interests, and 3) should surrounding systems go down then the mnlp module dies with it. When encountering an invalid operator code, the Xmnlp system simply outputs zeroes and silently defers error handling to a higher level.

Chapter 4

Discussion

4.1 Limitations

Measurement Reliability The cycle count measures have some uncertainty in them. While the number of cycles are measured exactly, the efficiency of the algorithms can vary. For instance, the C versions depends upon the compiler's optimization, and the assembly code depends upon how it was written. However, this is representative of realistic usage of the system. And to make a conservative comparison, the highest optimization level was used for C code.

Regarding the precision measurements. The C code might introduce a small error as it outputs its results in decimal base with printf. And Octave/Matlab, while serving as the golden reference, might contribute a small error itself. Here, the differences observed between the various methods of calculations are great enough to not warrant any suspicion in regard of the abovementioned contributions.

The 17-bit LLS slope gave insufficient precision, and were in need of the left-shifting enhancement. With a maximum n_e of $1e12[1/m^3]$, and 17 bits of resolution, the precision should be at an acceptable $7e6[1/m^3]$. The math indicated that a left-shift was necessary, and results agree. One possible explanation is that it is an artifact of the test stimuli. The stimuli is ensured to be monotonic, but it might be too comprehensive and generate ratios that are smaller than what can be expected in the field. If the truth is that only 17 bits are needed, then that would be even better for reducing transmission of data.

Verification Coverage Coverage of functional testing was near 100%. While this is good for inciting confidence, it does depend on the quality of the defined covergroups. Additionally, it would have been interesting to see the results of formal verification, but neither the time nor the tools were available.

The System in General One disadvantage of calculating results in-flight is that one loses the ability to try out different methods on the raw data. Different ways of filtering and analyzing the data could provide new

angles and insights, but that possibility is lost when only processed data is transmitted. However, such experimentation can still be done before the in-flight algorithms are decided, or the in-flight calculation can be toggleable.

The probe bias is assumed to be given in 16-bits, the same size as for current readings. The intention behind this is to facilitate greater resolution and flexibility over allowable biases, because they relate to the floating potential. However, considering that probe biases used are e.g. {2.5, 4, 5.5, 10} volt [37], using a whole 16 bits on this is too generous. If this was taken into account earlier, a more tightly optimized design could have been made.

Overall the design is more complicated than strictly necessary. The cause of this is a conflict of two ideas: 1) to adhere to the strict rules of RISC-like instructions, and 2) to get the performance benefits of a stand-alone monolithic accelerator. What this resulted in is a hybrid that is not immediately intuitive to reason about. Considering how RISC-V itself takes inspiration from literally decades of experimentation in the field, one take-away might be that such instructions mature best in an evolutionary manner. Although, I am happy that a pure integer method worked, even without emulated floats.

4.2 Conclusion

The goal was primarily to customize a RISC-V processor core for m-NLP purposes. Where the three sub-goals were to increase the speed of computation, keep the output data smaller than sending individual samples, and judge whether custom RISC-V processors are a good fit for m-NLP systems.

As a whole, the goal has been achieved. A working instruction set is implemented in a processor core, the results are computed as expected, and the code is synthesizable. The scope had been deliberately limited to only parts of the relevant m-NLP parameters, in order to focus on the processor customization aspect.

Regarding speed improvements, the goal was reached. Comparing a plain C variant at 22ns clock period, versus a custom ASM variant at 23ns, the achieved reduction in time is 44% (cf. worst case C division), 24% (cf. mid-range), and -20% (an increase, cf. best case C). If the derived cycle count from previous theses is reliable, the new count is 4 times shorter than the FPU case and 30 times shorter than the emulated floats version.

The second goal of maintaining reduced data transmission is also achieved. Consider that one set of samples is 4x 16-bit, or 64-bit. Since it was found that only 17 bits are needed to represent the highest electron densities, and an additional 10 bits are needed to meet the desired resolution, then 32 bits are sufficient to hold the result.

The final goal of being a good fit for the m-NLP systems is more difficult to conclude. Requirements were sat for maximum and minimum electron densities to be able to handle, and these requirements were fulfilled. For example, the target precision was to detect electron densities

of $1e8[1/m^3]$, and the achieved resolution was $4.7e7[1/m^3]$. Furthermore, several requirements were sat for desired sampling rate. Implemented on a ZedBoard, this frequency was beaten by over 14 times. The proposed instruction set has slightly cumbersome complexity, but new instructions can be implemented as quickly as one day.

4.3 Future Works

The following are general suggestions for future work. Some relate to the 4DSpace initiative at whole, and some pertain to this particular implementation.

- Pursuing RISC-V in general is a good idea. For m-NLP purposes, one could continue experimenting with custom extension, and additionally evaluate the usefulness of other standard extensions. Regarding standard extensions, one possible interest is to monitor the progress of the `P` and `V` extensions (currently in draft) [8], because they relate to digital signal processing and might be able parallelize the computations. One could benefit from doing a survey of other core implementations. And although it might be too much for the purpose, a 64-bit core makes a lot of sense considering the numbers which are treated, and it is worth taking a look at.

- More of the methods relevant for m-NLP could be implemented. That is, if one continues with the custom extensions approach, more of the parameters can be computed. This thesis focused mostly on the making of custom extensions, so future work may focus on making an even better fit for m-NLP. For example, calculate platform potential and correlation coefficient in addition to electron density. And also look into making use of the latest developments in theory, including the possibility to adjust the beta factor [24]. See Section 3.1.2 Completeness for more inspiration on this.

- Mapping out more detailed and specific needs of the m-NLP and 4DSpace project. As I read previous theses and research papers, I found it difficult to piece together a clear cut definition of requirements. Although that could be a natural property of research initiatives. More theoretical work could be done to extract a clear map of pending tasks and explicit requirements.

- Custom extensions is still worth more experimentation. There exists a possibility that better thought out instructions can be faster, more accurate, and easier to use. If making custom extensions becomes more commonplace, the cost and overhead of development might diminish. For example, one could 1) make it even more CISC-like, 2) be more strictly RISC-like, 3) experiment with posits a floating point alternative to IEEE 754, or 4) do a compare against a memory mapped accelerator.

Appendix A

Appendices

A.1 Source Code

Source code for the project is not unloaded unto this document. Instead, links to repositories on the university server is provided. An exception is made for Listing A.1, which contains the main Xmnlp RTL module responsible for computing all of the instructions.

The repositories listed below are the most central ones. Throughout the work, about 10 repositories were used in total. Most of them were used for personal experimentation and are not included here.

A *fork* [1] of RI5CY resides in: `https://github.uio.no/master-projects/riscy`. This contains a full processor core with the Xmnlp modification. The processor source code is in `/rtl` directory, including my mnlp module. All of my testbenches, including UVM benches, are found in `/tb/mnlp`. Subdirectories should be named somewhat self-explanatory, and README files give brief overviews in most directories. Another directory worth checking out is the pre-existing directory `/tb/core`. My addition can be found explicitly by running a `diff` against commit `3c3400b0`.

A fork of the instruction set simulator spike: `https://github.uio.no/master-projects/spike`. This includes recognition for the new assembly mnemonics. A copy of the Xmnlp reference model is included to enable functional support for the new instructions. Almost all of my additions can be found by searching for "abaddab".

Post-processing, partitioning, and other scripts were done in: `https://github.uio.no/master-projects/riscv-xmnlp`. The most important directory is `/opcodes`, containing scripts for modifying the compiler toolchain. Then, `/testvec` holds scripts used for testing n_e precision. Partitioning scripts are in `/partitions`, ISS experimentation in `/iss`, and ICI-2 related scripts in `/confidential`.

A few links to 3rd party sites can also be useful. The compiler toolchain `https://github.com/riscv/riscv-gnu-toolchain`. Tools necessary for

[1] A fork is a copy of an existing project that is developed in a different direction than the original work.

generating headers from opcode definitions `https://github.com/riscv/riscv-tools`. And an open source verilog simulator which runs some of the tests in the RI5CY repo `https://www.veripool.org/wiki/verilator`.

The core of the Xmnlp extension is done by the module in Listing A.1.

Listing A.1: Main computational module.

```
module mnlp
(
  input  logic        clk,
  input  logic        rst_n,

  input  logic        en_i,
  input  logic [ 4:0] operator_i,
  input  logic [31:0] op_a_i,
  input  logic [31:0] op_b_i,

  output logic [31:0] result_o,
  output logic        ready_o
);

  logic        clk2_q;       // Divided clock
  logic        ofa_q;        // Overflow A (clocked)
  logic        ofb_q;        // Overflow B (clocked)
  logic        prevloadv_q;
  logic        ready_d1;
  logic [17:0] xsum_q;
  logic [31:0] xx_q [0:3];
  logic [33:0] xxsum_q;
  logic [35:0] xsxs_q;
  logic [51:0] xsys_q;

  logic        longinstr_w;  // Instruction needs extra cycle
  logic        ofa_w;        // Overflow A (combinatorial)
  logic        ofb_w;        // Overflow B (combinatorial)


  always_comb begin
    case (operator_i)  // Check for "long instructions"
      MNLP_OP_XSYSL,
      MNLP_OP_XSYSH,
      MNLP_OP_XSXSL,
      MNLP_OP_XSXSH,
      MNLP_OP_XXSL,
      MNLP_OP_XXSH:  longinstr_w = 1;
      default:       longinstr_w = 0;
    endcase
  end


  always_comb begin
    result_o = 0;
    ready_o  = 1;

    ofa_w = 0;
    ofb_w = 0;

    if (en_i) begin
```

```
    case (operator_i)
      MNLP_OP_LOADV:    ;

      MNLP_OP_ADDA,
      MNLP_OP_AADD,
      MNLP_OP_ADDB,
      MNLP_OP_ABADDAB:  xxaddxx();

      MNLP_OP_XSYSL,
      MNLP_OP_XSYSH:    xsys();

      MNLP_OP_XSXSL,
      MNLP_OP_XSXSH:    xsxs();

      MNLP_OP_XXSL,
      MNLP_OP_XXSH:     xxs();

      default:          ;
    endcase

    if (longinstr_w && (clk2_q || ready_d1))
      ready_o = 0;
  end

  if (|op_a_i || |op_b_i || prevloadv_q)
    ;  // Hack for sensitivity lists and task arguments
end


always @(posedge clk) begin
  if (rst_n == 0) begin
    clk2_q      <= 0;
    xsum_q      <= 0;
    ofa_q       <= 0;
    ofb_q       <= 0;
    xx_q[0]     <= 0;
    xx_q[1]     <= 0;
    xx_q[2]     <= 0;
    xx_q[3]     <= 0;
    xxsum_q     <= 0;
    prevloadv_q <= 0;
    ready_d1    <= 0;
    xsxs_q <= 0;
    xsys_q <= 0;
  end else begin
    clk2_q   <= ~clk2_q;
    ready_d1 <= ready_o;

    prevloadv_q <= 0;
    if (en_i) begin
      case (operator_i)
        MNLP_OP_LOADV:   sync_loadv();
        MNLP_OP_ABADDAB,
        MNLP_OP_ADDA,
        MNLP_OP_ADDB:    sync_xxaddxx();
        default:         ;
      endcase
    end
```

```
    if (clk2_q) begin
      xsxs_q <= xsum_q * xsum_q;
      sync_xsys();
    end

    xxsum_q  <=
      {1'b0, xx_q[0]} + {1'b0, xx_q[1]}
      + {1'b0, xx_q[2]} + {1'b0, xx_q[3]};
  end
end


task sync_loadv();
  // xs = x0 + x1 + x2 + x3
  xsum_q  <=
    {1'b0, op_a_i[15:0]} + {1'b0, op_a_i[31:16]}
    + {1'b0, op_b_i[15:0]} + {1'b0, op_b_i[31:16]};

  xx_q[0] <= op_a_i[15:0]  * op_a_i[15:0];
  xx_q[1] <= op_a_i[31:16] * op_a_i[31:16];
  xx_q[2] <= op_b_i[15:0]  * op_b_i[15:0];
  xx_q[3] <= op_b_i[31:16] * op_b_i[31:16];

  prevloadv_q <= 1;
endtask


task sync_xxaddxx();
  ofa_q <= ofa_w;
  ofb_q <= ofb_w;
endtask


task sync_xsys();
  logic [51:0] sum;
  logic [33:0] part;
  logic [33:0] ys;

  ys = {ofb_q, ofa_q, op_a_i};

  sum = 0;
  for (int i = 0; i < 18; i++) begin
    part =  ys  &  {34{xsum_q[i]}};
    sum += part  <<  i;
  end
  xsys_q <= sum;
endtask


task automatic xxaddxx();
  logic [33:0] tmp;

  tmp = {2'b00, op_a_i} + {2'b00, op_b_i};
  if (operator_i == MNLP_OP_AADD)
    tmp = tmp + {33'd0, ofa_q};
  else if (operator_i == MNLP_OP_ABADDAB) begin
    tmp = tmp + {ofa_q, 32'd0};
    tmp = tmp + {ofb_q, 32'd0};
  end
```

```
    result_o = tmp[31:0];

    if (operator_i == MNLP_OP_ADDA) begin
      ofa_w = tmp[32];
      ofb_w = ofb_q;
    end else if (operator_i == MNLP_OP_ADDB) begin
      ofa_w = ofa_q;
      ofb_w = tmp[32];
    end else if (operator_i == MNLP_OP_ABADDAB) begin
      ofa_w = tmp[32];
      ofb_w = tmp[33];
    end
  endtask


  task automatic xsys();
    if (prevloadv_q)
      ready_o = 0;

    if (operator_i == MNLP_OP_XSYSL) begin
      result_o = xsys_q[31:0];
    end else if (operator_i == MNLP_OP_XSYSH) begin
      result_o = {12'd0, xsys_q[51:32]};
    end
  endtask


  task automatic xsxs();
    if (prevloadv_q)
      ready_o = 0;

    if (operator_i == MNLP_OP_XSXSL) begin
      result_o = xsxs_q[31:0];
    end else if (operator_i == MNLP_OP_XSXSH) begin
      result_o = {28'd0, xsxs_q[35:32]};
    end
  endtask


  task automatic xxs();
    if (prevloadv_q) begin
      ready_o = 0;
    end

    if (operator_i == MNLP_OP_XXSL) begin
      result_o = {xxsum_q[29:0], 2'b00};
    end else if (operator_i == MNLP_OP_XXSH) begin
      result_o = {28'd0, xxsum_q[33:30]};
    end
  endtask

endmodule
```

A.2 Large Latex Documents

I want to share a little script, in hopes that it may be useful to other people. When working with large documents, it is easy to get lost in the mass of information. Since documents are structured as tree graphs, I thought to use the file system's directory hierarchy for organizing sections.

This makes changing order easy, it's easy to move a subsection to another place.

Using a "miller columns" file browsers, it is easy to navigate.

For instance, here is a small part of my directory hierarchy:

```
...
1-Premise/20-RISC-V
1-Premise/20-RISC-V/00-about
1-Premise/20-RISC-V/20-Modules.tex
1-Premise/20-RISC-V/30-Instruction_Formats.tex
...
```

The awk script is quite simple:

```
/\// {
    if ($N ~ /about/) {
        printf "        \\subfile{%s}\n", $N
    } else if ($1 == "" || $2 == "") {
    } else if ($3 == "") {
        printf "\\chapter{"
        name = $2
    } else if ($4 == "") {
        printf "  \\section{"
        name = $3
    } else if ($5 == "") {
        printf "    \\subsection{"
        name = $4
    } else if ($6 == "") {
        printf "      \\subsubsection{"
        name = $5
    } else if ($7 == "") {
        printf "ERROR '%s'\n", $N
    }

    gsub("_", " ", name)
    gsub("^[0-9]*-", "", name)
    gsub("\.tex$", "", name)
    if ($N !~ /about/)
        printf "%s}\n", name

    path = $N
    if (path ~ /\.tex$/) {
        printf "        \\subfile{%s}\n", path
    }
}
```

Using it in the treatise is done by `cat`ing the result into a file, and then inputing it into the main latex file, like so:

```
\input{build/sections.tex}
```

A.3 Management

Throughout the project, milestones and subtasks were maintained. From an early stage, an attempt was made to draw up a schedule. But this initial plan proved to be challenging to follow in the long run, as the plan changed and milestones needed revising.

The research question, in all honesty, did not solidify until halfway through the work. I was relatively new to the world of scientific work practice, and insisting upon a topic that no professor had *directly* in their repertoire.

Figure A.1 illustrates how the plan evolved over time. Please bear in mind that the study credits are not distributed equally over this range of time, but is rather consentrated towards the end.

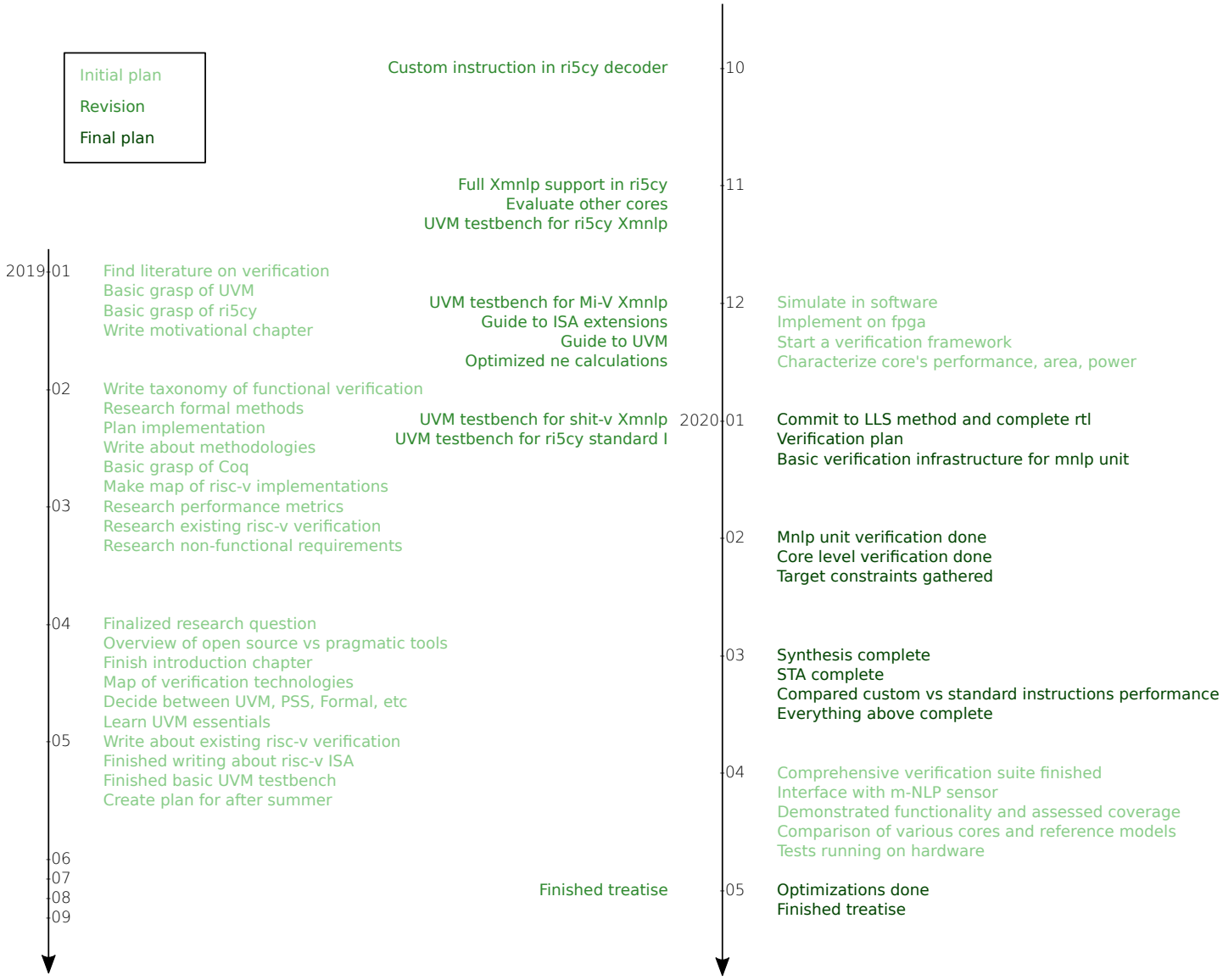


Figure A.1: Evolution of thesis milestones.

Coordination of the work was done using an issue tracker. The intention being to keep planning overhead out of the mind and free up mental resources. This turned out to be helpful in keeping up the pace, but adjusting the usage policy was needed as the management overhead grew too big to handle at a certain point. Being online, this facilitated the advantage of communicating with supervisors directly on issues.

Issues were ranked on priority (low, medium, high). Additionally, tags were used to indicate the type of work ("design" for coding, "research" for learning, "writing" for thesis ideas), but this practice was dropped later. Milestones (a bundle of issues) were also used, in order to 1) group related tasks, and 2) manage time. It was not always easy to meet the deadlines.

Every single day I wrote a journal, detailing my work and decisions. It served well as a thinking-tool and reference for writing. The wiki functionality of github was also used to document findings en route. Time was also spent researching topics like harking/sharking/tharking, how to efficiently conduct meetings, workflow methods, defining a research question, definition of done, axiomatic design, TRIZ, Five whys, a.o..

Appendix B

Bibliography

Bibliography

[1] Faculty of Mathematics and University of Oslo Natural Sciences. 4dspace - strategic research initiative. `https://www.mn.uio.no/fysikk/english/research/projects/4dspace/index.html`. [Online; accessed 2020-05-11].

[2] K S Jacobsen, A Pedersen, J I Moen, and T A Bekkeng. A new langmuir probe concept for rapid sampling of space plasma electron density. *Measurement Science and Technology*, 21(8):085902, jul 2010.

[3] T A Bekkeng, K S Jacobsen, J K Bekkeng, A Pedersen, T Lindem, J-P Lebreton, and J I Moen. Design of a multi-needle langmuir probe system. *Measurement Science and Technology*, 21(8):085903, jul 2010.

[4] Eirik Nobuki Kosaka. Development and characterization of the data flow for a spacecraft payload instrument, 2018.

[5] Marius Elvegård. Sounding rocket telemetry compression, 2016.

[6] Bartas Venckus. Softcore hdl processor for implementation in fpga and asic, 2019.

[7] Supervision meetings. (Personal communication). University of Oslo.

[8] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, 2019.

[9] Google. Risc-v instruction generator – riscv-dv. `https://github.com/google/riscv-dv`. [Online; accessed 2020-05-03].

[10] RISC-V Foundation. Isa formal spec public review. `https://github.com/riscv/ISA_Formal_Spec_Public_Review`. [Online; accessed 2020-05-03].

[11] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. Technical report, University of California at Berkeley, 2014.

[12] Eli Greenbaum. Open source semiconductor core licensing. *Harvard Journal of Law & Technology*, 25, 2011.

[13] RISC-V Foundation. Risc-v cores and soc overview. `https://riscv.org/risc-v-cores/`, 2020. [Online; accessed 2020-04-24].

[14] PULP Platform and OpenHW Group. Cv32e40p (formerly ri5cy). https://github.com/openhwgroup/cv32e40p. [Online; accessed 2020-04-26].

[15] A. Pullini, D. Rossi, I. Loi, A. Di Mauro, and L. Benini. Mr. wolf: A 1 gflop/s energy-proportional parallel ultra low power soc for iot edge processing. In *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, pages 274–277, 2018.

[16] Tore André Bekkeng. Prototype development of a multi-needle langmuir probe system, May 2009.

[17] M. F. Ivarsen, H. Hoang, L. Yang, L. B. N. Clausen, A. Spicher, Y. Jin, E. Trondsen, J. I. Moen, K. Røed, and B. Lybekk. Multineedle langmuir probe operation and acute probe current susceptibility to spacecraft potential. *IEEE Transactions on Plasma Science*, 47(8):3816–3823, 2019.

[18] Mike Bartley. Trends in functional verification. https://www.testandverification.com/resources/published-articles/, 2015. [Online; accessed 2019-02-06].

[19] Brian Bailey, Grant Martin, and Thomas Anderson, editors. *Taxonomies for the development and verification of digital systems*. Springer, New York, 2005.

[20] Ahmed Yehia. The top most common systemverilog constrained random gotchas. In *DVCon Europe 2014 Proceedings*, 2014. https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2014/proceedings/T5_1_paper.pdf.

[21] Erik Seligman, Tom Schubert, and V Achutha Kirankumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. 01 2015.

[22] Stuart Sutherland. Systemverilog for design : a guide to using systemverilog for hardware design and modeling, 2006.

[23] International Organization for Standardization. Systems and software quality requirements and evaluation (square). Technical Report ISO/IEC 25010:2011, 2011.

[24] H Hoang et al. A study of data analysis techniques for the multi-needle langmuir probe. *Meas. Sci. Technol.*, 29, 2018.

[25] University of Oslo mNLP Team at Department of Physics. science.c. (Personal communication).

[26] A. Spicher, W. J. Miloch, L. B. N. Clausen, and J. I. Moen. Plasma turbulence and coherent structures in the polar cap observed by the ici-2 sounding rocket. *Journal of Geophysical Research: Space Physics*, 120(12):10,959–10,978, 2015.

[27] Tore Andre Bekkeng, Espen Sorlie Helgeby, Arne Pedersen, Espen Trondsen, Torfinn Lindem, and Joran Idar Moen. Multi-needle langmuir probe system for electron density measurements and active spacecraft potential control on cubesats. *IEEE Transactions on Aerospace and Electronic Systems*, 55(6):2951–2964, 2019.

[28] John L. Gustafson and Isaac Yonemoto. Beating floating point at its own game: Posit arithmetic. 2017.

[29] IEEE. *Std 1800.2-2017 Standard for Universal Verification Methodology Language*, 2017.

[30] IEEE. *Std 1800 SystemVerilog - Unified Hardware Design, Specification, and Verification Language*, 2 edition, 2011.

[31] Stuart Sutherland and Tom Fitzpatrick. Uvm rapid adoption: A practical subset of uvm. DVCon San Jose, 2015. `http://sutherland-hdl.com/papers/2015-DVCon_UVM-rapid-adoption_paper.pdf`.

[32] Verification Guide. Uvm testbench architecture. `https://www.verificationguide.com/p/uvm-testbench-example.html`. [Online; accessed 2019-05-29].

[33] cluelogic. Uvm tutorial for candy lovers. `https://github.com/cluelogic/uvm-tutorial-for-candy-lovers`. [Online; accessed 2019-05-29].

[34] Ray Salemi. *The UVM Primer*. Boston Light Press, 2013.

[35] The Regents of the University of California. Spike risc-v isa simulator. `https://github.com/riscv/riscv-isa-sim`. [Online; accessed 2020-05-08].

[36] Nitish Srivastava. Adding custom instruction to riscv isa and running it on gem5 and spike. `https://nitish2112.github.io/post/adding-instruction-riscv/`, 2017. [Online; accessed 2020-05-13].

[37] Ketil Røed. (Personal communication). University of Oslo.

[38] Xilinx. Vivado design suite - hlx editions - 2019.2. Available at `https://www.xilinx.com/support/download.html`.

[39] Clifford Wolf. Project icestorm. `http://www.clifford.at/icestorm/`. [Online; accessed 2020-05-20].

[40] Xilinx. *Vivado Design Suite Tutorial - Using Constraints*. Xilinx, 2014.