



UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Evaluating FFT Implementations Developed for use on FPGA Hardware

by

Morag Brown
BRWMOR001

A dissertation submitted in fulfilment of the requirements for the degree of

Master of Science in Engineering

in the Department of Electrical Engineering,
Faculty of Engineering and the Built Environment

at the

University of Cape Town

11 February 2023

Supervisor: Dr Willem Francois Schonken

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Plagiarism Declaration

I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is my own.

I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this dissertation from the work(s) of other people have been cited and referenced.

I declare that this dissertation is my own work, and I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own. This dissertation has been submitted to the Turnitin module and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signed by candidate

Morag Brown
11 February 2023

Abstract

The Fast Fourier Transform (FFT) is an important algorithm within the field of radio astronomy, where it is used as a key processing component in the digital signal processing (DSP) backends of radio spectrometers. Field Programmable Gate Arrays (FPGAs) are a type of hardware commonly used to perform the various computations required to process incoming radio signals, including the FFT. As the FFT is a pivotal part of the DSP chain, the performance of the FFT implementation is a primary consideration when designing radio instruments. Here, performance measurements include numerical accuracy, logic resource utilisation and maximum clock rates at which the implementation can be operated.

This research project investigates, tests and compares three different FFT cores, namely the CASPER wideband, ASTRON wideband, and Xilinx SSR FFTs, implemented for use on FPGA hardware. The hardware on which these FFTs are tested are the Square Kilometer Array Reconfigurable Application Board (SKARAB) and the Xilinx Alveo AU50 Accelerator Card. Testing metrics have been selected specifically for radio astronomy applications and include Spurious Free Dynamic Range (SFDR), signal-to-noise ratio (SNR) preservation, quantisation noise, resource utilisation and achievable clock rates. A double precision floating point software FFT, the Fastest Fourier Transform in the West (FFTW), has been selected as the benchmark against which the numerical accuracy of the FFT cores under test are compared.

Results of this research project indicate that, when tested under the same conditions for default configuration options, the Xilinx core performs with the greatest degree of numerical accuracy and the ASTRON core with the least. While the CASPER core in its default state performs to a slightly lesser degree of accuracy than its Xilinx equivalent, this core can be configured in such a way that its performance is greatly improved - resulting in highly accurate results that surpass the performance of all other cores. The ASTRON and Xilinx cores do not provide the same configuration options that would allow for the same level of improvement. From a hardware resource utilisation perspective, ASTRON is the least resource efficient core and the CASPER core the most efficient. Both CASPER and Xilinx cores are capable of achieving clock rates between 450MHz-600MHz on Xilinx UltraScale+ hardware, while the ASTRON core failed to achieve a clock rate of 200MHz for the same test configuration.

Acknowledgments

My preferred title for this thesis is "*The Fast and the Fouriers: Gaussian Drift*". Unfortunately, I have been informed that I am not allowed to use this as an official title, and so it remains the name of this work in spirit only. Regardless, credit and thanks go to Omer Mahgoub for coming up with this name and making me laugh when I was at my most stressed.

Thank you to my supervisor, Dr. Willem Schonken, for both agreeing to supervise me and trusting me enough to get on with things without having to breathe down my neck. Thank you for the dry humour on the rare occasions that I required meetings, and the serious but still laid back attitude that went far to reassure me in my work. Thanks to Wesley New for agreeing to help me research potential topics way back when I first started at SRAO, and for all the encouragement and support that has been provided along the way.

Possibly the biggest thanks of all goes to Dr. Jonathon Kocz, who, for absolutely no benefit to himself, has invested an immeasurable amount of effort into both this project and my professional development over the past few years. Thank you for constantly holding me to such a high standard and always pushing me further - as frustrating as it's been at times, without it I would never have reached the place I am now. A further thank you for all the music and video games and chats over literature ranging from badly written fan-fiction to the masterpieces of Sir Terry Pratchett. Thanks to Amanda Kocz and Princess Peppa for just existing in general, and being the creators of such excellent motivation material in the form of promised cat pictures.

An inevitable thank you to my parents, Maggie and Jim, for all the support. Thank you to my mom for giving birth to me in the first place, and for all the millions of things she's done for me throughout the 26 (almost 27!) years since. Thanks for the fresh oat and raisin cookies and all the love given when I visited while in the midst of writing up. Give Loki a kiss on his head for me.

A massive thanks to my dad for always having my back and for all his efforts, especially over the last little while. Thank you for all the many, *many* re-reads of this thesis, and for suggesting improvements that could help more clearly link everything together. Thank you for the amount of effort put in as we both grew from this journey together. If you can find Zombie, tell him he's the best cat.

To my brother, Hugh, and sister-in-law, Tash, thank you for providing so much comfort and support over the years. I really have no words to describe how much you both mean to me, and how grateful I am for all you have done and continue to do.

Thank you to (takes a deep breath): Kathryn, Amish, Irshaad, Catherine, Ziyaad, Jason, the multiple Amy's, Omer, Gareth, Samantha, Savannah, Duduzile, Lliezel, Johan, Amrain and everyone else who forms the wonderful support system I have around me. Thank you to Kathryn and Amish for both having my back literally since day one at SRAO. And thank you to Sav and Sam for being so understanding of my need to drop off the face of the earth for a few months. Thank you to Gareth for being such a major catalyst in my professional development. And thank you to Johan for all the braais, cheeky humour and words of encouragement in the final stretch.

Thank you to Robin Van Wyk and Marc Welz for always having answers to my most obscure Linux questions, and for being so supportive and willing to help. Thank you to the The DSP Andrews, Andrew Martens and Dr. Andrew van der Byl, for tirelessly answering my endless stream of DSP related questions. Thank you to Dan Werthimer for his input on this project, and to the CASPER community in general for all support provided.

Lastly, thank you to Francois Kapp for effectively starting off the chain of events that would lead me to where I am today. Thanks for the guitar, and all the support along the way.

Table of Contents

Plagiarism Declaration	i
Abstract	ii
Acknowledgments	iii
Table of Contents	iv
List of Figures	vii
List of Tables	xvi
Glossary	xviii
1 Introduction	1
2 Literature and Theory Review	5
2.1 Radio Spectrometry	5
2.1.1 Calculating Power Spectral Density	5
2.1.2 Non-ideal Effects of a Non-ideal DFT	7
2.1.3 Finite Impulse Response Filters and Polyphase Filterbanks	8
2.2 The Fast Fourier Transform	10
2.2.1 Decimation-in-time	10
2.2.2 Decimation-in-frequency	12
2.3 Finite Precision Signal Processing	13
2.3.1 Fixed-point Representation	13
2.3.2 Floating-point Representation	14

2.3.3	Number Representations in FPGAs	14
2.3.4	Fixed-Point Arithmetic and Condition Handling	14
2.4	FPGA FFT Implementations	17
2.4.1	FPGA Design Flows	18
2.4.2	Existing Implementations	21
2.5	Other FFT Implementations	25
2.5.1	Intel Corporation Implementation	26
2.5.2	GPU Implementations	26
2.5.3	Software Implementations	27
3	Methodology	28
3.1	Requirements Analysis	29
3.2	Design and Implementation	30
3.3	Integration, Testing and Analysis	31
4	Requirements Analysis	32
4.1	High Level Requirements	32
4.2	Low Level Requirements	33
4.2.1	[HLR 1] Input Test Data	33
4.2.2	[HLR 2] System Architecture	37
4.2.3	[HLR 3] Output Analysis and Comparison	38
5	Test System Design and Implementation	42
5.1	Design Tool and Hardware Selection	43
5.1.1	Selection Process	43
5.1.2	Selected Tools	44
5.1.3	Selected Test Platforms	45
5.2	Test System Design	47
5.2.1	SKARAB-only Concept Designs	48
5.2.2	SKARAB and AU50 Concept Designs	50
5.3	Test System Implementation	53

5.3.1	Test Vector Generator	54
5.3.2	Test Design Implementation	56
5.3.3	System Architectures	63
6	Analysis System: Design, Integration and Testing	69
6.1	Analysis System Development	69
6.1.1	FFTW Test Benchmark	70
6.1.2	Output Analysis Process	73
6.2	System Integration	79
6.3	Formal Testing	81
6.3.1	Hardware Test Designs	81
6.3.2	Input Test Vector Data	82
7	Results and Analysis	85
7.1	Numerical Analysis Results	85
7.1.1	Test Case 0	85
7.1.2	Test Case 1	93
7.2	Hardware Resource Utilisation and Clock Rates	100
7.2.1	Resource Utilisation	100
7.2.2	Achievable Clock Rates	103
8	Conclusion	105
8.1	Numerical Performance	105
8.2	Resource Utilisation and Clock Rates	106
8.3	Suggestions for Further Work	107
	References	108
	Appendices	111
	Appendix A	112
A.1	AU50: ASTRON 1k, L = 4 Simulink Design	112
A.2	AU50: ASTRON 1k, L = 8 Simulink Design	113

A.3	SKARAB: ASTRON 1k, L = 4 Simulink Design	114
A.4	SKARAB: ASTRON 1k, L = 8 Simulink Design	115
A.5	AU50: CASPER 1k, L = 4 Simulink Design	116
A.6	AU50: CASPER 1k, L = 8 Simulink Design	117
A.7	SKARAB: CASPER 1k, L = 4 Simulink Design	118
A.8	SKARAB: CASPER 1k, L = 8 Simulink Design	119
A.9	AU50: Xilinx 1k, L = 4 Simulink Design	120
A.10	AU50: Xilinx 1k, L = 8 Simulink Design	121
Appendix B		122
B.1	TVG0 - FFTW PSDs	122
B.2	TVG0 - Improved PSDs	123
B.3	TVG0 - Hardware PSDs	125
B.4	TVG0 - Simulation PSDs	136
Appendix C		148
C.1	TVG1 - FFTW PSDs	148
C.2	TVG1 - Hardware PSDs	149
C.3	TVG1 - Simulation PSDs	158

List of Figures

1.1	MeerKAT FX-correlator F-Engine design flow. Coarse and fine delays align the signal received at different antenna positions in time, the PFB transforms time domain data into the frequency domain, and the corner turner transposes the spectral output to a set of sub-bands for correlation.	2
2.1	Frequency channel response of an FFT windowed using a rectangular windowing function. . .	8
2.2	Frequency channel response of a 16 tap, Hamming-windowed PFB vs a rectangular-windowed, standard FFT. An improvement in the PFB response can be seen when compared to that of the FFT.	9
2.3	Signal flow of a natural-order-in 8-point radix-2 DIT FFT. Sourced from [12].	11
2.4	2-point FFT butterfly. Sourced from [12].	11
2.5	Signal flow of a natural-order-in 8-point radix-2 DIF FFT. Sourced from [12].	12
2.6	Single precision (32 bit) floating-point representation. Sourced from [15].	14
2.7	Traditional FPGA design flow.	19
2.8	Real and imaginary even/odd symmetry of the Fourier Transform of complex input $y[n]$. Sourced from [36].	22
2.9	CASPER wideband FFT implementation. For a total N-point FFT with a wideband factor of L, the CASPER FFT performs L many $M=N/L$ FFTs in parallel, followed by an L point FFT. Sourced from [38].	23
2.10	ASTRON wideband FFT implementation. For a total N-point FFT and wideband factor of P, the ASTRON FFT computes P-many M-point ($M=N/P$) pipelined FFTs followed by a final P-point parallel FFT. Sourced from [39].	24
3.1	V-model for Systems Engineering. Sourced from [55].	28
3.2	Hierarchy of High Level Requirements and derived Low Level Requirements.	29
3.3	Diagram illustrating an iterative design and implementation process.	30
3.4	Test and Analysis System shown as a combination of distinct Test System and Analysis System components.	31

4.1	High Level Diagram of The Test and Analysis System based on the above high level requirements. HLR 1 and HLR 2 form the requirements of the test system, and HLR 3 defines the analysis system requirements.	33
4.2	High Level Diagram of The Test and Analysis System with corresponding high and low level requirements included.	41
5.1	Mid-Level Diagram of Test System. The test system refers to design components from generation of input data through to the capture of FFT output spectra, with testing performed in simulation and on actual FPGA hardware. The test platform refers to hardware and simulation environments and rounded blocks indicate communication and memory interfaces between the test platform and the rest of the architecture.	42
5.2	a) block diagram and b) internal layout of the SKARAB. Sourced internally from within SARAO.	45
5.3	Diagram of the Alveo U50 Data Center Accelerator Card. Sourced from [59].	46
5.4	Mid-level diagram of the test system containing system architectures for both simulation and hardware test platforms. Rounded blocks indicate communication and memory interfaces between the test platform and the rest of the architecture.	47
5.5	First hardware testing concept design. This design proposed using a signal generator as the method of test data generation and an onboard ADC to convert the analogue signal to digital for processing, with the result then output over a 40GbE network.	48
5.6	Diagram of the second concept design for hardware testing. This design proposed using the Dsim (a system that simulates MeerKAT digitiser data) to generate test data, with both input and output over 40GbE.	49
5.7	Diagram of the first concept design for simulation testing. This design would generate the input test signal in software within a MATLAB script, which would then be accessed by the Simulink design using <i>from workspace</i> blocks. The FFT output would then be written to <i>to workspace</i> blocks to be saved by a MATLAB script.	50
5.8	Diagram of the third concept design for hardware testing. This design would use the Xilinx DDS Compiler block for test data generation. FFT outputs would be written to a CASPER library BRAM snapshot block - a memory block that writes a pre-defined number of samples to memory when triggered.	50
5.9	Diagram of the second concept design for simulation testing. This design would use the Xilinx DDS Compiler block for test data generation. FFT outputs would be written to <i>to workspace</i> blocks and captured by a MATLAB script.	51
5.10	Diagram of the fourth concept design for hardware testing. Input test data would be generated in software and written to BRAM blocks on the hardware test platform for use as a look-up table. FFT outputs would be captured by BRAM snapshot blocks.	52
5.11	Diagram of the third concept design for simulation testing. Input test data would be generated in software, loaded into the MATLAB environment and passed to the BRAM blocks as initial values in simulation. The FFT output would then be captured by a MATLAB script from <i>to workspace</i> blocks.	52

5.12	Diagram of the implemented test system. Test data generation is performed by a script that generates and writes the test vector values to file. Both hardware and simulation test architectures load the test vector into the BRAM within their designs from this file, perform the FFT and write the FFT output to either BRAM snapshots (hardware) or <i>to workspace</i> blocks (simulation).	53
5.13	Usage of the TVG python script.	54
5.14	Flowchart of TVG operation. The TVG is capable of generating two test vectors based on the arguments passed from the command line. Both test vectors are comprised of a sinusoidal and WGN component, with the signal-to-noise ratio depending on the test vector case.	55
5.15	High Level Diagram of base Simulink design for FFT core testing. There are slight differences between FFT cores in the expected format of incoming data and how each core structures their output data. Differing logic on the inputs and outputs of each core is required to process incoming data from the BRAMs and write outputs, but in all cases, a base design comprised of control registers, BRAMs, the FFT under test and BRAM snapshots (hardware) or <i>to workspace</i> blocks (simulation) is required.	56
5.16	Diagram showing blocks required specific to each hardware platform.	57
5.17	Simplified Simulink block diagram of the base design for $L = 4$. Reset and enable signals control a BRAM address counter and the FFT core. The FFT subsystem represents each of the FFTs under test. The FFT output is then written to BRAM snapshots (hardware) and <i>to workspace</i> blocks (simulation) as separate <i>real</i> and <i>imaginary</i> components of the complex output.	58
5.18	ASTRON FFT block and timing diagram. When used to process two real-valued inputs, $x[n]$ and $y[n]$, the real and imaginary inputs of the core accept these two separate signals as inputs respectively. This core expects an input sync pulse on the same clock cycle as valid input data, and generates an output sync pulse on the same clock cycle as the first sample of valid output data.	59
5.19	Generation of <i>sync</i> input. Both ASTRON and CASPER cores provide input and output sync lines for data alignment purposes. The input sync pulse is generated using a counter and relational logic that is driven by software registers at run-time.	60
5.20	CASPER FFT block and timing diagram. This core only processes real-valued signals and can be configured to process one or multiple input signals. An input sync pulse is expected one clock cycle before the first valid sample of input data, and an output sync pulse is generated one clock cycle before the first valid output spectral sample.	61
5.21	Xilinx FFT block and timing diagram. This core can only process a single real-valued input if either of its real or imaginary inputs are set to zero. There are no input or output sync lines available with this core.	62
5.22	Directory structure of system architectures for both test platforms.	63
5.23	Usage of the <i>run_ffts.py</i> control script.	64
5.24	An example of running <i>run_ffts.py</i> for fpg files with a wideband factor $L = 4$.	64
5.25	Flowchart depicting operation sequence of hardware test script, <i>run_ffts.py</i> . The script connects to the hardware platform, obtains a lists of all test vector and fpg files and then performs a test for each test vector across all combinations for all wideband factors, FFT lengths and shifting schedules.	65

5.26	Flowchart depicting operation sequence of simulation test script, <i>sim_ffts.m</i> . Lists of input files and slx designs are created and simulations are performed for all wideband factors, FFT lengths and shifting schedules.	67
6.1	Mid-level diagram of Analysis System. This system analyses the spectral outputs of each FFT core under test as well as the timing and resource utilisation reports generated from the Vivado project of each final hardware test compile.	70
6.2	Directory structure of FFTW system.	70
6.3	Usage of the <i>run_r2c_fftw.py</i> control script.	71
6.4	Flowchart depicting operation of both <i>run_r2c_fftw.py</i> and <i>calc_r2c_fftw.exe</i> . For each input test vector file and tested FFT length, the Python script generates the FFTW output filename and calls the FFTW executable for each test combination. The FFTW executable then allocates memory, creates a real-to-complex FFTW plan and then computes the FFT. The spectral output is then written to file and the FFTW plan destroyed.	72
6.5	Usage of the <i>analysis.py</i> numerical analysis script.	73
6.6	Structure of results directory created by the <i>analysis.py</i> script.	73
6.7	An example template of a simulations results file generated by <i>analysis.py</i>	74
6.8	Flowchart detailing performance of numerical analysis script, <i>analysis.py</i> . For each wideband factor (denoted as <i>demux</i>), the FFTW output and FFT test outputs are loaded for each tested FFT length, with simulation and/or hardware analysis then performed. Analysis includes computing total spectral power, the normalised PSD, SFDR and SNR of the FFTW and FFT core outputs. The MSE between the FFT core output and the FFTW output is then calculated using the normalised PSD of each.	75
6.9	Resource utilisation report generated for an implemented Vivado project design.	77
6.10	Hierarchical power analysis report generated for an implemented Vivado project design. . . .	77
6.11	Design timing summary generated for an implemented Vivado project design.	78
6.12	Diagram depicting integrated Test and Analysis System. The test system is demarcated in blue and the analysis system in red. Inputs to the analysis system are not only the spectral outputs of the FFT cores under test, but also the resource and timing reports for each hardware compile.	79
6.13	Directory tree structure of integrated system.	80
6.14	Plot of Test Vector Generator 0 components and final test vector, for $f = 50\text{MHz}$. This test vector has been generated according to radio astronomy specific requirements. WGN has been generated for an RMS value of $\frac{1}{8}$ and a sinusoidal waveform has been generated with an amplitude derived from an SNR of -10dB (resulting in a sinusoidal amplitude of 0.056). . . .	83
6.15	Plot of Test Vector Generator 1 components and final test vector, for $f = 50\text{MHz}$. This test vector has been generated according to the general test case requirements. A sinusoidal waveform has been generated with an amplitude of 0.9, and a WGN model generated for an RMS value equal to half the precision of the implemented fixed-point system. Generating noise according to this parameter resulted in a test vector with an SNR of 92dB.	84

7.1	Non-normalised PSDs of ASTRON vs FFTW outputs.	91
7.2	Real and complex valued output magnitudes of CASPER hardware test vs scaled FFTW. . .	96
7.3	Real valued output magnitudes of ASTRON and Xilinx hardware tests vs scaled FFTW. . .	97
7.4	Diagram of a butterfly stage implemented for the CASPER FFT.	98
7.5	Real valued output magnitudes of 16-bit and 18-bit ASTRON.	99
A.1	astron_hw_test_au50_1k	112
A.2	astron_hw_test_au50_1k_8in	113
A.3	astron_hw_test_skarab_1k	114
A.4	astron_hw_test_skarab_1k_8in	115
A.5	casper_hw_test_au50_1k	116
A.6	casper_hw_test_au50_1k_8in	117
A.7	casper_hw_test_skarab_1k	118
A.8	casper_hw_test_skarab_1k_8in	119
A.9	xilinx_hw_test_au50_1k	120
A.10	xilinx_hw_test_au50_1k_8in	121
B.1	Normalised PSD of FFTW, $N = 1024$	122
B.2	Normalised PSD of FFTW, $N = 16384$	122
B.3	Normalised PSD of FFTW, $N = 65536$	123
B.4	Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shifting = improved.	123
B.5	Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shifting = improved.	124
B.6	Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shifting = improved, Rounding = even. . .	124
B.7	Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.	125
B.8	Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.	125
B.9	Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = alt.	126
B.10	Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = alt.	126
B.11	Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.	127
B.12	Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.	127
B.13	Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = alt.	128
B.14	Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = alt.	128

B.15 Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.	129
B.16 Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.	129
B.17 Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.	130
B.18 Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.	130
B.19 Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.	131
B.20 Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.	131
B.21 Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.	132
B.22 Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.	132
B.23 Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.	133
B.24 Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.	133
B.25 Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = alt.	134
B.26 Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.	134
B.27 Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.	135
B.28 Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.	135
B.29 Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.	136
B.30 Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.	136
B.31 Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.	137
B.32 Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = alt.	137
B.33 Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = alt.	138
B.34 Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.	138
B.35 Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.	139
B.36 Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = alt.	139
B.37 Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = alt.	140
B.38 Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.	140
B.39 Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.	141
B.40 Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.	141
B.41 Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.	142
B.42 Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.	142
B.43 Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.	143
B.44 Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.	143

B.45	Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.	144
B.46	Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.	144
B.47	Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.	145
B.48	Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = alt.	145
B.49	Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.	146
B.50	Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.	146
B.51	Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.	147
B.52	Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.	147
C.1	Normalised PSD of FFTW, $N = 1024$	148
C.2	Normalised PSD of FFTW, $N = 16384$	148
C.3	Normalised PSD of FFTW, $N = 65536$	149
C.4	Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.	149
C.5	Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.	150
C.6	Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.	150
C.7	Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.	151
C.8	Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.	151
C.9	Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.	152
C.10	Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.	152
C.11	Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.	153
C.12	Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.	153
C.13	Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.	154
C.14	Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.	154
C.15	Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.	155
C.16	Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.	155
C.17	Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.	156
C.18	Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.	156
C.19	Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.	157
C.20	Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.	157
C.21	Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.	158

C.22 Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.	158
C.23 Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.	159
C.24 Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.	159
C.25 Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.	160
C.26 Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.	160
C.27 Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.	161
C.28 Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.	161
C.29 Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.	162
C.30 Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.	162
C.31 Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.	163
C.32 Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.	163
C.33 Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.	164
C.34 Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.	164
C.35 Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.	165
C.36 Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.	165
C.37 Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.	166
C.38 Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.	166
C.39 Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.	167

List of Tables

2.1	Resource utilization of Monroe's optimizations compared to the equivalent CASPER implementations.	23
2.2	Resource utilization of the FFT v9.1 core on the Virtex 7 xc7vx690t device.	25
7.1	Simulation Test: Overflow matrix for wideband factor $L = 4$	86
7.2	Simulation Test: Overflow matrix for wideband factor $L = 8$	86
7.3	Numerical analysis results of reference FFTW outputs using Test Vector 0.	87
7.4	Numerical analysis results of simulation test outputs for $L = 4$, tested using Test Vector 0. . .	88
7.5	Numerical analysis results of hardware test outputs for $L = 4$, tested using Test Vector 0. . .	88
7.6	Numerical analysis results of simulation test outputs for $L = 8$, tested using Test Vector 0. . .	89
7.7	Numerical analysis results of hardware test outputs for $L = 8$, tested using Test Vector 0. . .	89
7.8	Numerical analysis results of CASPER and ASTRON simulation test outputs for an improved shifting schedule against nonoptimal shifting schedules.	92
7.9	Numerical analysis results of CASPER, $N = 16384$, $L = 4$ simulation tests for even and infinite rounding schemes.	92
7.10	Numerical analysis results of reference FFTW outputs using Test Vector 1.	93
7.11	Numerical analysis results of simulation test outputs for $L = 4$, tested using Test Vector 1. . .	94
7.12	Numerical analysis results of hardware test outputs for $L = 4$, tested using Test Vector 1. . .	94
7.13	Numerical analysis results of simulation test outputs for $L = 8$, tested using Test Vector 1. . .	95
7.14	Numerical analysis results of hardware test outputs for $L = 8$, tested using Test Vector 0. . .	95
7.15	Numerical analysis results of ASTRON 18-bit output data against 16-bit output data results. . .	99
7.16	Power requirements and percentage resource utilisation per FFT core for $L = 4$	101
7.17	Power requirements and percentage resource utilisation per FFT core for $L = 8$	101
7.18	Power requirements and percentage resource utilisation for a CASPER core using even and infinite rounding for $N = 16384$ and $L = 4$	102

7.19	Achieved and estimated clock rates (in MHz) for $L = 4$	103
7.20	Achieved and estimated clock rates (in MHz) for $L = 8$	103

Glossary

ASTRON The Netherlands Institute for Radio Astronomy.

Autocorrelation The convolution of a signal with itself, giving a measure of the correlation of the signal with a delayed version of itself as a function of the delay.

Biplex An FFT implementation architecture type that shares logic resources between two input data streams, halving the amount of complex arithmetic logic resources required for the same number of operations.

CASPER The Collaboration for Astronomy Signal Processing and Electronics Research, an open group of scientists and engineers producing astronomy signal processing resources to aid instrument design.

Convolution An integral of the product of a function shifted across another function, used in calculating both cross-correlation and autocorrelation.

Correlator The component of a radio instrument that correlates the inputs of multi-antenna systems, and calculates the PSD of received signals.

CPU Central Processing Unit, a type of integrated circuit that functions as the main control and processing unit of a computer system.

Cross-correlation The convolution of two distinct signals, measuring their similarity as a function of their relative displacement.

DFT Discrete Fourier Transform, the discretised form of the Fourier Transform for use in digital applications.

F-Engine The component of a correlator that computes the Fourier Transform of time domain input data.

FFT Fast Fourier Transform, an algorithm that computes the DFT in an efficient manner.

FFTW The Fastest Fourier Transform in the West, a C-based library for computing the DFT in software.

FIR Finite Impulse Response filter, a filter with a finite length impulse response calculated as the weighted moving average of an input.

Fourier Transform A generalisation of the Fourier Series that decomposes both periodic and aperiodic functions into their frequency components.

FPGA Field Programmable Gate Array, a type of integrated circuit in the family of re-configurable logic devices.

GbE Gigabit Ethernet, a computer networking term describing the transmission of Ethernet frames at a rate of Gigabits per second - a 10GbE technology can transfer 10 Gbps.

GPU Graphics Processing Unit, a type of integrated circuit capable of multi-threaded computation originally designed for rendering graphics.

Hermitian Symmetry A property of a real-valued Fourier Transform where positive frequency complex outputs are equal to the complex conjugate of negative frequency complex outputs, resulting in symmetry about the y-axis for real and imaginary components.

MeerKAT A SARAO-built (and operated) 64-antenna radio interferometer based in the Karoo desert.

PFB Polyphase Filterbank, an implementation of the DFT preceded by an FIR filterbank providing improved frequency channel response.

PSD Power Spectral Density (or power spectrum), a function describing the distribution of a signal's power across its frequency components.

Radio Interferometer A radio telescope array comprised of separate radio antennas that, when operated together, form a single telescope with greater resolution.

Radio Spectrometer An instrument that measures the PSD of cosmic radio emissions.

SARAO The South Africa Radio Astronomy Observatory, previously The SKA-SA, a subsidiary of the National Research Foundation that manages all radio astronomy activity in South Africa.

SSR Super Sample Rate, see "Wideband Factor".

Stationary and Ergodic A stochastic process with statistical properties that both do not vary with time and can be derived based on a (sufficiently long) single sample of the process.

Stochastic A random model (or more formally, well-defined by a random probability distribution).

WGN White Gaussian Noise, a noise model generated from samples taken from a zero-mean Gaussian probability distribution.

Wideband Factor The number of parallel inputs processed by an FFT core for a single input signal.

X-Engine The component of a correlator that computes the correlation of input data.

Xilinx A technology company that produces programmable logic semiconductor devices.

Chapter 1

Introduction

“the Fourier Transform is used to filter out undergraduate electrical engineering students who lack abstract mathematical and visualisation ability” - eric larsen

The *Fourier Transform* transforms an input function (typically in the time domain) into an output function of frequency - this frequency domain data represents the frequency composition of the input. Frequency domain decomposition is particularly useful in filtering applications or any area that requires spectral analysis, such as image processing, radar systems, telecommunications, spectroscopy, and radio astronomy.

The *Fast Fourier Transform* (FFT) is an algorithm that efficiently computes the *Discrete Fourier Transform* (DFT) used to perform the Fourier Transform in digital systems. This research project focuses specifically on a use case of the FFT within the field of radio astronomy, where radio spectrometers are used to measure the spectral composition of signals received from astronomical sources (referred to within this work as *sky signals*) through the calculation of the signal’s *power spectral density* (PSD) [1]. Chapter 2 provides the mathematical background of both the DFT and the calculation of the PSD within radio spectrometer instruments. However, a summary is given below to provide context to the research project topic.

Using both Parseval’s and Wiener-Khinchin theorems, the PSD can be computed by either calculating the autocorrelation of the signal and taking the Fourier Transform of the result, or by taking the Fourier Transform first and correlating afterwards [2]. Instruments that use the former correlation method are known as XF-correlators, while FX-correlators use the latter. Regardless of correlator type, the FFT is a fundamental processing component of any digital correlator instrument, with the correlator being one of the primary use-cases of the FFT in radio astronomy instrumentation.

As the FFT is an important component of radio astronomy signal processing, the performance of the FFT implementation used in the signal processing backend is an important consideration when designing correlator systems. Performance can be measured by the numerical accuracy of the output spectrum of the FFT, but also includes the percentage of total logic resources available on hardware used by the FFT, and clock rates at which the FFT implementation can be run. Different FFT implementations (also referred to as *cores*) may use varying amounts of logic resources or have different achievable clock rates depending on the specifics of how each is implemented. Cores that have more efficient resource usage and can achieve higher maximum clock rates would allow for more processing cores on a single hardware unit, and for a greater data processing rate. *Field Programmable Gate Arrays* (FPGAs) are a type of hardware commonly used as the processing units of correlators due to their reconfigurability, parallel computing capabilities and logic resources specifically designed for digital signal processing operations.

An example of an FPGA-based correlator system is the correlator implemented on the MeerKAT radio telescope [3]. The MeerKAT, developed and built by the *South African Radio Astronomy Observatory* (SARAO) as a precursor to the *Square Kilometer Array* (SKA) telescope, is a 64-antenna interferometer that makes use of an FX-correlator design. This correlator is implemented on 288 FPGA boards called *Square Kilometer Array Reconfigurable Application Boards* (SKARABs) [4]. The system that performs the Fourier Transform channelisation of this FX-correlator is called the F-Engine, while the X-Engine performs the correlation of the frequency domain data output by the F-Engine. Correlation is required in both the combination of the signal received across multiple antennas, and the calculation of the signal's PSD, however only the F-Engine is of interest for introductory context.

While additional logic is required to form an F-Engine, the FFT plays an integral role in processing signals received by the MeerKAT telescope (and any instrument performing power spectrum analysis). Figure 1.1 below depicts the design flow of a single F-Engine, where the *PFB* refers to a *polyphase filterbank* that is produced as a result of combining an FFT with a *finite impulse response* (FIR) filter. *Coarse* and *fine* delays align the signal received at different antenna positions in time, with coarse delay only able to compensate for integer multiples of a single clock cycle and fine delay applying a frequency-domain phase shift to fine-tune this compensation. The *corner turner* then transposes the spectral output to a set of sub-bands that are sent to corresponding X-Engines for further processing.

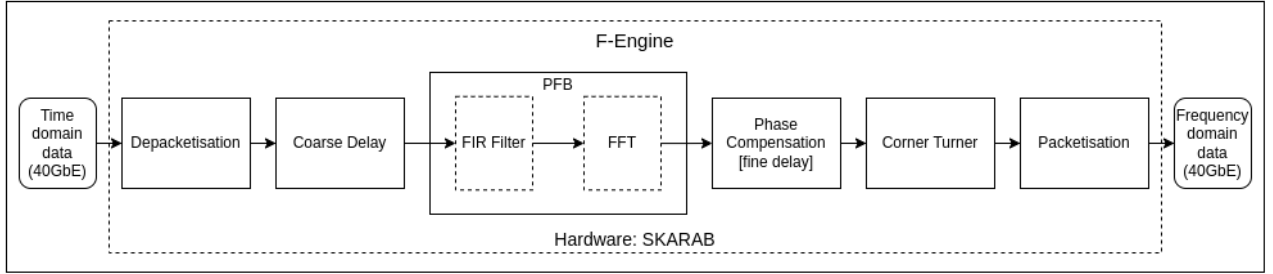


Figure 1.1: MeerKAT FX-correlator F-Engine design flow. Coarse and fine delays align the signal received at different antenna positions in time, the PFB transforms time domain data into the frequency domain, and the corner turner transposes the spectral output to a set of sub-bands for correlation.

The above context is provided to emphasise just how fundamental a role the FFT plays in radio astronomy correlator instruments, and why the performance of the core is of such interest. This thesis focuses on the performance of FFT cores implemented for use on FPGA hardware, specifically for use in radio astronomy. The FFT cores tested in this project are introduced in more detail in Section 2.4.2 of Chapter 2, but are listed here as the ASTRON wideband, CASPER wideband and Xilinx SSR FFT cores. The FPGA hardware on which they are tested are the *Square Kilometer Array Reconfigurable Application Board* (SKARAB) and the *Alveo U50 Data Center Accelerator Card* (AU50).

These cores have been tested in simulation and on hardware using fixed-point data and compared to the output of the equivalent double-precision floating point *Fast Fourier Transform in The West* FFTW implementation to measure the effects of quantisation on performance. The methods of these comparisons are detailed in Chapter 4, alongside the requirements of two input signals used to test the FFT cores. These input signals have been derived to test the FFT in the general case, at the extremes of the fixed-point representation's dynamic range; and in a radio astronomy case, with a signal representative of that typically detected by a radio instrument. Both signals contain a sinusoidal component and noise component modelled using a *white Gaussian noise* (WGN) distribution, with their *signal-to-noise ratios* (SNRs) differing according to the specific test case.

The fixed-point bit-width used to represent data for each core is a 16-bit quantisation capable of representing values in the range $[-1, 1)$. The cores have also been tested for an increasing number of FFT stages, with FFT lengths of $[2^{10}, 2^{14}, 2^{16}]$ having been selected, for wideband factors of both four and eight parallel samples processed per clock cycle. Longer FFT windows for the same sample rate provide finer granularity in the frequency domain; however, Section 2.3 describes the effects of finite precision arithmetic and explains that FFTs with more stages have a greater chance of overflow occurring. The FFT cores under test provide the option of supplying the core with a *shifting schedule* that downshifts the data being processed by the core at the stages specified by the schedule in an attempt to prevent overflow. However, shifting affects the fidelity of the data and shifting too aggressively can lead to the loss of important information. As such, the FFT cores have been tested for different shifting schedules to measure how shifting affects data. Details on the shifting schedules used for each input signal are given in Section 6.3.

Beyond numerical performance, an analysis on the resource utilisation and achievable clock rates of each core has also been performed. FPGAs contain a finite number of available resources, and an understanding of each core's resource usage is of interest. The maximum achievable clock rate for each core has also been measured for the design in which they are used, providing insight into the rate at which each core can be operated. Finally, tests have been performed with no optimisations applied to any core, as a measure of how well each performs "out of the box". Chapter 5 provides more detail on the specifics of the implemented test system architecture.

With introductory context to this research project having been provided, a brief summary of each chapter follows.

Chapter 2 details the theoretical background as mentioned above, with the aim of providing context for various design features and considerations present throughout this project. A review of the current literature regarding the performance of FFT implementations on FPGA hardware, as well as on *Graphical Processing Units* (GPUs) and *Central Processing Units* (CPU) is also presented in this chapter.

The following chapter, Chapter 3, outlines the design methodology used in the design and development of the system used to test the FFT cores of interest. The design methodology used follows the Systems Engineering approach, with the aim of mitigating potential downstream problems before they occur through a thorough analysis of system requirements.

The results of the aforementioned requirements analysis are documented in Chapter 4, with high-level and low-level system requirements having been identified. It is according to these derived requirements that the system used to test each FFT core and analyse their outputs has been designed and implemented.

The design and implementation process of the system used to test each FFT is then presented in Chapter 5. This chapter describes the various design revisions explored before detailing the implementation of the final test system. The output of this system as detailed in this chapter feeds directly into the subsequent chapter.

Chapter 6 describes the implementation of a system used to analyse the outputs of each FFT obtained from the test system of the previous chapter. The integration of the test and analysis components to form a single system is then detailed. The final section of this chapter presents the conditions under which formal testing has been conducted, providing details regarding input signal parameters, shifting schedules and a summary of the final system configuration as a whole.

The results of the test and analysis system are then presented in Chapter 7, with an analysis of these results provided. This discussion includes an analysis of the numerical performance of each FFT, and presents the clock rates achieved and resource utilisation of each core as a percentage of total available resources for each board.

The conclusions drawn regarding the performance of each core based on the analysis of results is presented in Chapter 8. Suggestions for additional work that could be performed to further test these cores beyond the scope of this project are then provided.

NOTE: *All relevant design files, scripts, programs, and results developed for and generated from this research project can be found in the following GitHub repository:
https://github.com/moragb96/brwmor001_msc-project [5].*

Chapter 2

Literature and Theory Review

“floating-point numbers are actually numbers that float when you put them in water, as opposed to fixed-point numbers that are fixed to the ocean floor” - me, showing an in-depth understanding of finite precision representations

This chapter both details the background theory necessary for this project and explores the available literature regarding this research topic. As the focus of this project is primarily on the implementation of signal processing algorithms on hardware specific to radio astronomy, a brief introduction to radio spectrometers is first given in a section on *Radio Spectrometry*. The mathematical context behind the *Fast Fourier Transform* (FFT) algorithm, a key component of any spectrometer, is then given. An overview of how digital systems represent real-world signals using *finite precision arithmetic*, and the impact this has on the implementation of the FFT on finite precision hardware, rounds out the sections on theoretical fundamentals.

Subsequent sections of this chapter then explore literature that describe how the above theory has been implemented as related to this project. A brief introduction to *Field Programmable Gate Arrays* (FPGA) design is given, before existing FFT implementations for use on FPGA hardware are investigated and a review of existing literature on the topic of testing the performance of these implementations is given. For completeness, FFT implementations for use on other hardware (namely *Central Processing Units* (CPUs) and *Graphics Processing Units* (GPUs)) are also explored; however, any further investigation into these forms of hardware falls outside the scope of this project.

2.1 Radio Spectrometry

Within radio astronomy, a spectrometer is an instrument used to study the spectral content of cosmic radio emissions. The *power spectral density* (PSD or *power spectrum*) of a signal describes the signal's spectral energy distribution over time - or rather, the power of the signal as distributed over its frequency components. By measuring the PSD of radio signals, a spectrometer allows astronomers to uncover and study characteristics of astronomical sources [1].

2.1.1 Calculating Power Spectral Density

This section details two methods by which the PSD may be calculated. As this project is implemented using digital systems that can only process discrete, sampled data, all equations are given as the discrete-case only.

Method 1

For a continuous-time, real-valued signal, $x(t)$, that is both stationary and ergodic, the non-windowed discrete-time representation of this signal is $x[n]$. Parseval's theorem [2] states that the power of $x[n]$ over all time can be calculated as:

$$P_{xx} = \lim_{N \rightarrow \infty} \frac{1}{2N} \sum_{k=-N}^{N-1} |X[k]|^2, \quad (2.1)$$

where $X[k]$ is the Fourier Transform of $x[n]$.

The distribution of signal power as a function of frequency (i.e. the PSD) with units W/Hz is then given by:

$$S_{xx}[k] = \lim_{N \rightarrow \infty} \frac{1}{2N} |X[k]|^2 \quad (2.2)$$

Method 2

According to the Wiener-Khinchin theorem[2], the PSD of a signal and its autocorrelation are Fourier Transform pairs. The PSD of $x[n]$ can therefore also be found by taking the Fourier Transform of the autocorrelation of $x[n]$. The discrete case is given below:

$$S_{xx}[k] = \sum_{m=-\infty}^{\infty} r_{xx}[m] e^{-j2\pi km} \quad (2.3)$$

The autocorrelation of $x[n]$, over all time, is given by the normalised discrete-time convolution of $x[n]$ with itself:

$$r_{xx}[m] = \lim_{N \rightarrow \infty} \frac{1}{2N} \sum_{n=-N}^{N-1} x[n]x[n-m] \quad (2.4)$$

Re-writing the above using angled brackets to denote averaging over time $[-N \leq n \leq N-1, (for N \rightarrow \infty)]$ for readability gives:

$$r_{xx}[m] = \langle x[n]x[n-m] \rangle \quad (2.5)$$

Substituting Equation 2.5 into Equation 2.3 shows that the PSD may be calculated as the Fourier Transform of the normalised convolution of $x[n]$ with itself:

$$S_{xx}[k] = \sum_{m=-\infty}^{\infty} \langle x[n]x[n-m] \rangle e^{-j2\pi km} \quad (2.6)$$

The convolution theorem shows that Equations 2.2 and 2.6 are equivalent as, for a real-valued "well-behaved" signal such as $x[n]$, the following is true:

$$|X[k]|^2 = \mathcal{F}\left\{ \sum_{n=-N}^{N-1} x[n]x[n-m] \right\}, \quad (2.7)$$

where $\mathcal{F}$ denotes the Fourier Transform.

While equivalent, these two equations show that the power spectrum of a signal can be calculated in two ways. These two correlation methods give rise to different correlation designs - *XF-correlators* first perform the autocorrelation and then take the Fourier Transform of the result (see Equation 2.3), while *FX-correlators* first take the Fourier Transform of the signal and then calculate the PSD (see Equation 2.2). Many recent correlator instruments have adopted an FX design approach over that of XF (see: MeerKAT [3], LEDA [6], HERA [7], CHIME [8]); however, in either case, the Fourier Transform is an important component in the signal processing architecture of a radio spectrometer.

2.1.2 Non-ideal Effects of a Non-ideal DFT

As it is an approximation of the continuous-time Fourier Transform, the response of a finite length DFT is non-ideal. Specifically, it can be thought of as the ideal, infinite length (but physically unrealisable) DFT, $X[k]$, of the signal $x[n]$ multiplied by the rectangle function, $\text{rect}_N[n]$:

$$X[k] = \sum_{n=-\infty}^{\infty} x[n]e^{\frac{-j2\pi kn}{N}} \quad (2.8)$$

$$X^*[k] = \sum_{n=-\infty}^{\infty} \text{rect}_N[n]x[n]e^{\frac{-j2\pi kn}{N}} \quad (2.9)$$

From the convolution theorem, it follows that the non-ideal DFT, $X^*[k]$, can be calculated as the convolution of the ideal DFT and the Fourier Transform of the rectangle function:

$$X^*[k] = \mathcal{F}\{\text{rect}_N[n]\} * X[k] \quad (2.10)$$

The Fourier Transform of the rectangle function is the *sinc()* function (and vice-versa):

$$\text{rect}_N(t) \xleftrightarrow{\mathcal{F}} \text{sinc}(\omega) \quad (2.11)$$

Convolution of $X[k]$ with the *sinc()* function gives rise to undesirable sidelobes in the output frequency channels. This non-ideal response introduces two main problems - *spectral leakage* and *scallop loss*.

Spectral leakage occurs when each frequency channel does not have a zero response outside its pass-band[1]. This non-zero response causes an input tone to appear in more than one output frequency channel.

Scallop loss occurs when signals are attenuated between frequency channel centres as a result of the channel's non-ideal shape.

A plot illustrating the frequency channel response of an FFT is shown in Figure 2.1 below. This plot has been generated using a modified version of the code supplied in the Jupyter Notebook project, "*Polyphase filterbanks: an interactive introduction*", written by Danny Price [9]. The code has been modified to use a single tap (i.e. a weighted averaging using only one sample) and window coefficients of a rectangular windowing function, as is the case for the non-ideal FFT defined in Equation 2.9.

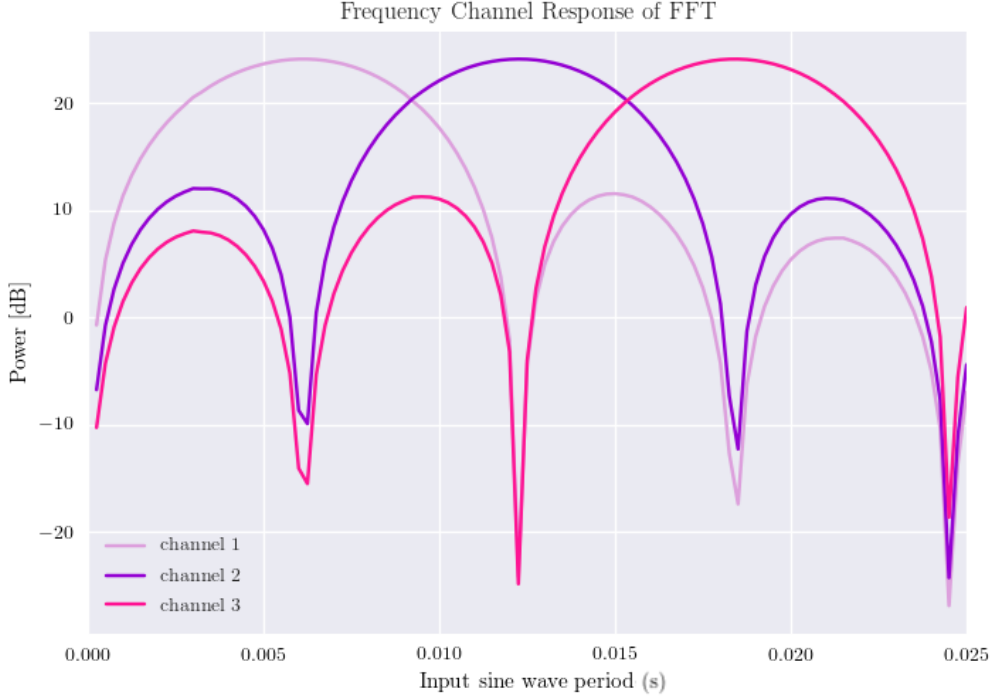


Figure 2.1: Frequency channel response of an FFT windowed using a rectangular windowing function.

The above plot clearly shows the non-uniform channel response with sharp roll-off and the undesirable sidelobes of a *sinc()* function. This non-ideal channel response and shape may hinder the accurate analysis of signals of interest and give way to imprecise results. However, the addition of a *polyphase decimated Finite Impulse Response* (FIR) filter before the DFT can improve channel response.

2.1.3 Finite Impulse Response Filters and Polyphase Filterbanks

In radio astronomy, a polyphase decimated FIR and DFT pair is called a *polyphase filterbank* (PFB). The polyphase FIR uses polyphase decomposition to decompose the input signal into P -many phases, effectively downsampling the signal by a factor of P for each phase. Each phase undergoes filtering in the form of a windowed moving average FIR filter of M taps, and the resulting filtered outputs are summed over P :

$$y[n] = \sum_{p=0}^{P-1} \sum_{m=0}^{M-1} h_p[m] x_p[n - m] \quad (2.12)$$

In this case, the chosen windowing function, $h()$, is the *windowed sinc()*. Multiplying each phase input by $\text{sinc}()$ ensures that the frequency channel response is as close to a rectangle function as possible (as per equation 2.11). However, digital systems can only process finite-length sets of data, and so the $\text{sinc}()$ function must be windowed to be of a finite-length.

The sharp cut-off at the boundaries of a $\text{sinc}()$ function windowed by a simple rectangular function introduces ripple and poor attenuation in the frequency response of the signal [10]. Frequency response can be improved by instead windowing with a smoother, tapered curve, with the Hann and Hamming windows commonly used within radio astronomy.

The complete PFB is formed by injecting the output of each FIR branch into a P-point DFT instead of taking the sum of the P branches. More details on the DFT are given in the following section, but the resulting PFB equation is shown below:

$$Y[k, n] = \sum_{p=0}^{P-1} \sum_{m=0}^{M-1} [h_p[m] e^{-\frac{jkp2\pi}{P}}] x_p[n - m] \quad (2.13)$$

Figure 2.2 below depicts the frequency channel response of a 16 tap, Hamming-windowed PFB against the frequency channel response of the FFT illustrated in Figure 2.1. This plot has been generated using the same introductory notebook by Danny Price [9], but with 16 FIR taps and windowing coefficients that apply the desired Hamming windowed $\text{sinc}()$ coefficient values to each phase. An immediate improvement in channel response shape is seen when compared to the response of the native FFT. The channel shape of the PFB is more rectangular, with sharper out-of-band attenuation and attenuated ringing effects and reduced scalloping loss.

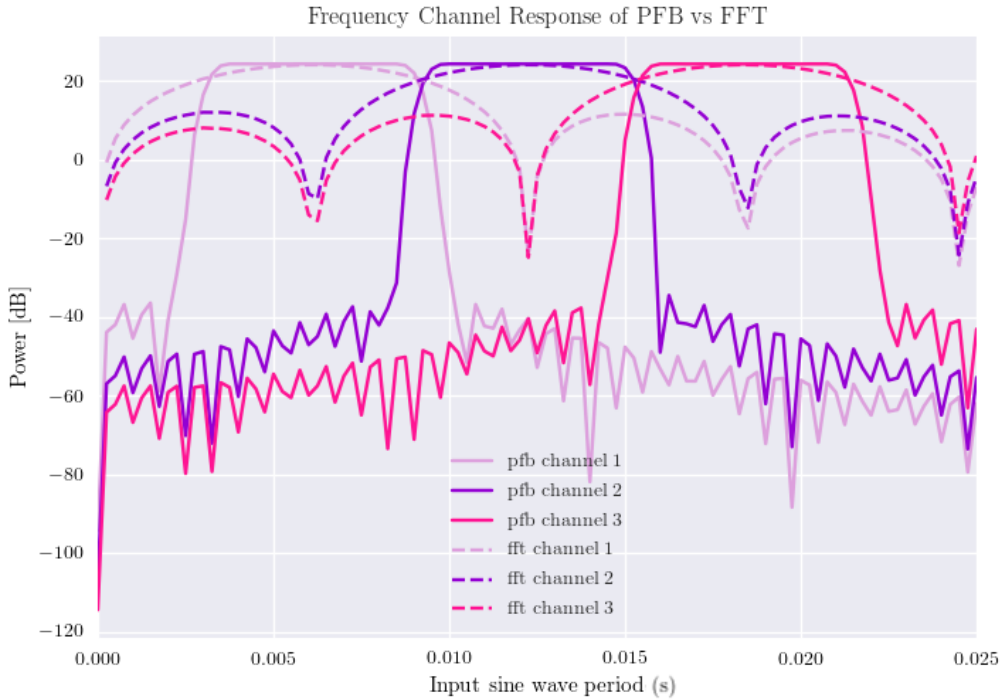


Figure 2.2: Frequency channel response of a 16 tap, Hamming-windowed PFB vs a rectangular-windowed, standard FFT. An improvement in the PFB response can be seen when compared to that of the FFT.

While the performance of both the polyphase FIR and the DFT affect the overall performance of the PFB (and spectrometer as a whole), this work focuses solely on the performance of DFT implementations for FPGA applications. As such, the Polyphase FIR is discussed no further, having only been mentioned to provide some insight into how the shortcomings of the non-ideal DFT may be mitigated, and context for further testing that could be performed using the specific implementations tested in this project - a topic that is discussed in Chapter 8.

The above provides a basic summary of the role the DFT plays in a radio spectrometer digital signal processing backend. The following section describes the *Fast Fourier Transform* (FFT) algorithm used to efficiently calculate the DFT. It is with variants of this algorithm that the FFT cores under test have been implemented, and as such an understanding of this algorithm is necessary when it comes to understanding how these cores function.

2.2 The Fast Fourier Transform

The *Discrete Fourier Transform* (DFT) transforms a finite duration signal of N samples into its frequency domain representation. For a discretely sampled, finite duration signal $x[n]$ of N total samples, the Discrete Fourier Transform is given by:

$$X[k] = \sum_{n=0}^{N-1} x_n W_N^{kn} \quad k = 0, \dots, N-1, \quad (2.14)$$

where

$$W_N = e^{-\frac{j2\pi}{N}}$$

Cooley and Tukey 1965 [11] detailed an algorithm that reduces the computational cost of the DFT from $O(N^2)$ operations to $O(N \log_2 N)$ operations. If a single operation takes 1ns to complete, this reduction in complexity increases the speed of a 10^9 -point DFT from 31.2 years to 30 seconds. This algorithm is one in a family of algorithms known as the Fast Fourier Transform (FFT).

The fundamental principle of these FFT algorithms is the recursive division of the DFT into smaller discrete Fourier Transforms. Two classes of the FFT, *decimation-in-time* (DIT) and *decimation-in-frequency* (DIF), are described below. The algorithm class selected for implementation depends on the practicalities of the implementation and how data flows within the system. This thesis focuses on FFT designs utilising cores that have been implemented using both *radix-2 decimation-in-time* and *decimation-in-frequency* algorithms.

2.2.1 Decimation-in-time

If $N = N_1 N_2$, the DFT can be recursively decomposed into N_1 DFTs of size N_2 . For an even N , a radix-2 DIT FFT recursively divides an N -point DFT into smaller DFTs [12]. The DFTs for both even and odd indexed inputs are computed separately and then combined to produce the complete DFT. This separation allows equation 2.1 to be written as:

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{N/2}^{kn} \quad (2.15)$$

$$X[k] = X_{\text{even}}[k] + W_N^k X_{\text{odd}}[k],$$

where

$$W_N = e^{-\frac{j2\pi}{N}}$$

This division is repeated recursively for a total of p stages, until there are $N/2$ many 2-point DFTs, where:

$$p = \log_2(N)$$

The radix-4 FFT follows the same logic as above, but instead divides an N -point DFT into $N/4$ many 4-point DFTs. The number of stages in a radix-4 decomposition is:

$$p = \log_4(N)$$

The diagram below shows the signal flow of an 8-point FFT implemented using radix-2 decimation-in-time. This diagram depicts a natural-order-in FFT where the input values are indexed in normal order (0 through 7) and the output sequence of DFT values is indexed in bit-reversed order (that is, the storage index of $x[1] = x[001]$ corresponds to the DFT output $X[4] = X[100]$, where the output's binary index is the bit-reversed form of the input's binary index). This bit-reversal is sorted post-DFT, but a natural-order-out variant exists where the input data is bit-reversed pre-DFT, resulting in an out of order input and normal order output.

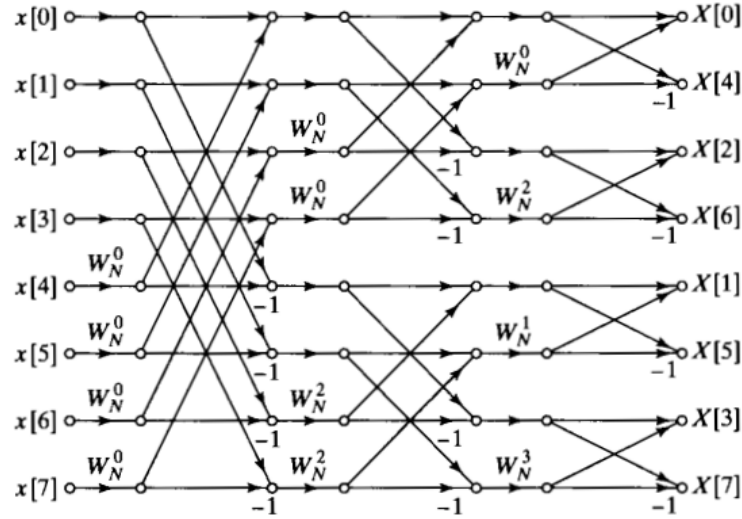


Figure 2.3: Signal flow of a natural-order-in 8-point radix-2 DIT FFT. Sourced from [12].

The butterfly structure of the smallest 2-point DFT with inputs $x[0]$ and $x[4]$ is shown below. Each butterfly performs one addition, subtraction and multiplication. The W_N term by which one of the butterfly inputs is multiplied is called a *twiddle factor*. Twiddle factors can be pre-computed and stored in look-up tables in order to improve computation time.

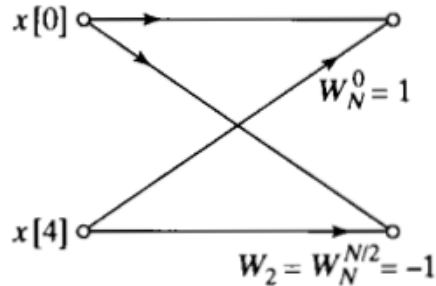


Figure 2.4: 2-point FFT butterfly. Sourced from [12].

2.2.2 Decimation-in-frequency

While decimation-in-time is based on forming subsequences of $x[n]$, decimation-in-frequency algorithms instead recursively decompose $X[k]$ into smaller subsequences of even and odd $X[k]$ indices [12].

Even-numbered frequency points can be obtained by:

$$X[2r] = \sum_{n=0}^{(N/2)-1} (x_n + x_{n+(N/2)}) W_{N/2}^{rn} \quad r = 0, \dots, (N/2) - 1 \quad (2.16)$$

Odd-numbered frequency points can be obtained by:

$$X[2r + 1] = \sum_{n=0}^{(N/2)-1} (x_n - x_{n+(N/2)}) W_N^n W_{N/2}^{rn} \quad r = 0, \dots, (N/2) - 1 \quad (2.17)$$

The equations above are the even and odd $N/2$ -point DFTs obtained by adding or subtracting the second half of the input sequence from the first half, and (in the case of the odd indexed sequence only) multiplying the result by W_N^n .

As with the radix-2 DIT algorithm, the DFT is recursively divided into these individual sequences a total of $\log_2 N$ times until $N/2$ 2-point DFTs are performed.

The signal flow diagram below depicts a natural-order-in 8-point DIF FFT. As with the natural-order-in DIT algorithm, the output indices are bit-reversed and are sorted to normal order post-DFT, and each butterfly within the FFT consists of one addition, subtraction and multiplication.

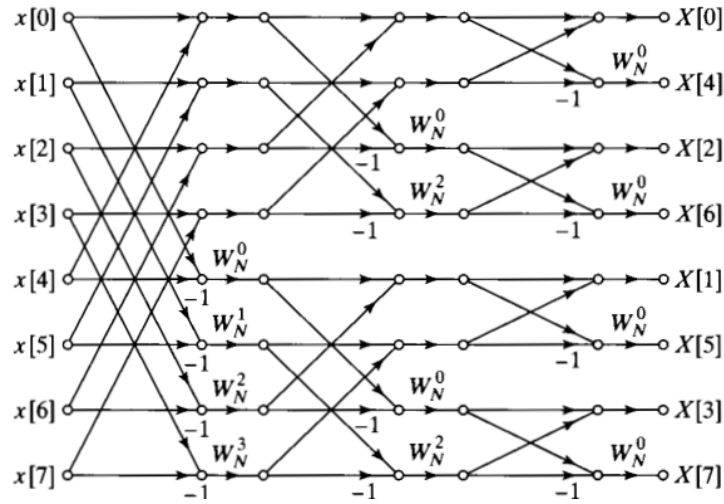


Figure 2.5: Signal flow of a natural-order-in 8-point radix-2 DIF FFT. Sourced from [12].

An understanding of the above realisations of the FFT is important when it comes to implementing and using such an algorithm on hardware. Digital systems can only represent numbers in a finite capacity, and performing finite precision arithmetic (such as the addition, subtraction and multiplication of each FFT butterfly) introduces problems surrounding bit growth, overflow handling, considerations into rounding schemes and the effects of quantisation.

The following section introduces digital number representation and the effects that this has on numerical accuracy. This is an important consideration in any digital system design, and as such is an aspect that must be considered when designing a spectrometer.

2.3 Finite Precision Signal Processing

As digital systems can only process signals represented with finite precision, real-world signals must be approximated as a set of discrete levels through a process known as *quantisation*. Numbers in a digital system are typically represented as binary numbers with integer and fractional parts stored as either fixed or floating point representations. Fixed point and floating point representations refer to the format in which this data is stored, with the *precision* (also called the *resolution*) of a representation referring to the smallest number that can be represented.

The unit typically assigned to a finite precision value is a *word*, with the number of bits assigned to representing a word of data called *word length*. *Bit width* is a term also used to indicate data size, however within this project a word refers to the width of a single sample value, while multiple samples may be concatenated together to form a vector with a greater bit width for processing within a design (see the implementation of the Xilinx SSR FFT test designs in Appendix A). The left-most bit of a binary number is the *most significant bit* (MSB), while the right-most bit is the *least significant bit* (LSB).

Two's complement [13] is typically used to represent signed finite-precision integers. The benefit of this is that addition and subtraction are functionally the same for signed and unsigned integers, allowing for the reuse of adders in hardware for both. However, as is shown below, signed representation reduces the precision and therefore *dynamic range* (the ratio of the largest and smallest values a system can represent) of a number representation.

2.3.1 Fixed-point Representation

Fixed-point representation assigns a fixed number of bits to both the fractional and integer parts of the binary form of a number[14]. The resolution of a fixed-point number is defined by its LSB.

An unsigned fixed-point number can be represented as:

$$N = UQp.q$$

Here, U indicates an unsigned number, Q indicates that the number is of binary, fixed-point type, p represents the integer part and q the fractional part. A UQ16.16 representation has a precision of 2^{-16} , minimum value of 0, a maximum value of $(2^{16}-2^{-16})$ and 16 bits are assigned to both integer and fractional parts (32 bits in total).

A signed fixed-point number can be represented in a similar manner with:

$$N = Qp.q$$

Here, the MSB is reserved as a sign-bit (with 0 representing a positive number and 1 a negative number). A Q15.16 representation of a 32 bit word has 15 bits assigned to the integer part, 16 bits assigned to the fractional part and the MSB assigned as a sign bit. Its precision is 2^{-16} , with a minimum value of -2^{15} and a maximum value of $(2^{15}-2^{-16})$.

2.3.2 Floating-point Representation

A drawback of fixed-point representation is a limit in range and precision in the representation of decimal numbers. In floating-point representations, the number of bits assigned to the integer and fractional parts is not fixed[14].

A floating-point number is represented by:

$$N = (-1)^S \times 2^E \times F$$

Here, S represents the sign-bit and is either 1 or 0, E is the exponent value and F is the fractional part (also called the *mantissa*). The floating-point number is stored in memory as a binary number *SEF*. The image below shows how a single precision (32 bit) floating-point binary representation is stored in memory.

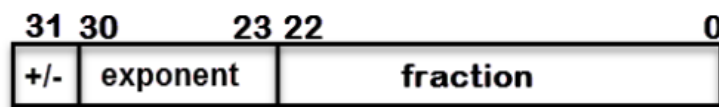


Figure 2.6: Single precision (32 bit) floating-point representation. Sourced from [15].

The integer part of a floating-point representation is normalised to have only one digit (assumed to be 1), so the fractional part is interpreted implicitly as *1.fraction*. Instead of storing the exponent value as a signed two's complement number, a bias of $(2^{e-1}-1)$ is applied to the exponent (where e is the number of bits assigned to E) and it is stored as an unsigned value. If the value stored by the exponent bits is greater than the bias, the exponent is positive, if the value is less than the bias, the exponent is negative.

2.3.3 Number Representations in FPGAs

Due to its exponential notation nature, an advantage of floating-point arithmetic is larger dynamic range compared to that of fixed-point. A high dynamic range is important when working with very large or very small numbers. However, floating-point FPGA designs generally consume more logic resources, require greater memory and consume greater power than a fixed-point equivalent [16]. As such, fixed-point representation is used in areas where factors such as resources, power consumption and cost are a concern.

While it is possible to utilise floating-point representations on FPGAs, only one of the FFT implementations under test within this project is capable of processing floating-point data. As such, all implementations are tested as fixed-point.

As was briefly mentioned at the end of section 2.2, finite-precision representation introduces the need for various considerations when it comes to arithmetic. The following explores a few key aspects of fixed-point arithmetic in particular.

2.3.4 Fixed-Point Arithmetic and Condition Handling

Section 2.2 showed that the FFT is comprised of many butterflies each performing only a single addition, subtraction and a multiplication (as shown in Figure 2.4). As a subtraction can be viewed as a negation followed by addition, an FFT is fundamentally a series of addition and multiplication operations. These two operations may cause bit growth and overflow, and may also require rounding and introduce further

quantisation error. These terms are explained below, with these explanations specific to the hardware and design toolflow used within this project. An alternative notation of fixed-point representations is given first for improved readability in the explanations that follow.

Alternative Fixed-Point Notation

From this point forward, fixed-point precision is represented with the notation:

$$N = \langle W, F \rangle, \quad (2.18)$$

where W represents the word length (including the sign bit when using signed data), and F represents the number of bits assigned to the binary fractional part. Furthermore, the binary representation of a number will be notated by:

$$\begin{aligned} N_{\langle W, F \rangle \text{ binary}} &= \langle (W - F)_{bits} \text{ } b \text{ } F_{bits} \rangle \\ 0.125_{\langle 4, 3 \rangle \text{ binary}} &= \langle 0b001 \rangle \end{aligned} \quad (2.19)$$

This notation is used throughout the following explanations on fixed-point arithmetic and condition handling, as well as in subsequent chapters detailing the design and implementation of the test system.

Addition and Multiplication

Both addition and multiplication have the potential to introduce bit growth, where the number of bits needed to represent a numerical value increases [17]. If this number of bits grows beyond the number of bits allocated to storing a numerical value, *overflow* occurs.

The sum of the addition of two X bit numbers may require $(X+1)$ bits if either number is at its maximum value. When multiplying two numbers of X bits and Y bits, the number of bits required to represent the product is $(X+Y)$ bits. As such, the possibility of overflow occurs within each butterfly and larger FFTs (that have a greater number of stages) have an increased likelihood of overflow due to the increased number of additions and multiplications.

Overflow handling

In the case of overflow occurring, data must be manipulated to fit within the allocated number of bits. The two most common ways of manipulation are *wrapping* and *saturation* [18]. An example of each is given for the below operation on two unsigned $\langle 4, 0 \rangle$ bit numbers:

$$0b1111 + 0b0001 = 0b10000 \quad (2.20)$$

If the sum of the above two values is also limited to an unsigned $\langle 4, 0 \rangle$ bit representation, overflow will occur.

Wrapping simply wraps the value around to the minimum representable value for the given representation. The sum would then wrap to the minimum value $\langle 0b0000 \rangle$.

Saturation saturates the result - in this case the saturated sum value would be $\langle 0b1111 \rangle$. For signed representations, the overflow saturates to the maximum representable value in the case of positive values, and to the minimum representable value in the case of negative values.

Each FFT under test includes an overflow status flag that signals when an overflow has occurred. However, overflow at any FFT stage has a cascading effect and will result in incorrect outputs at multiple frequency channels. It is therefore desirable to prevent overflow entirely where possible.

For this purpose, FPGA FFT implementations typically incorporate *shifting schedules* (also referred to as *scaling schedules*) that allow the designer to specify a *re-quantisation* of a value at any given stage of the FFT, where re-quantisation refers to a change in the number of bits assigned to representing a number. This scaling shifts the bit pattern representation of the number to the right by one bit, effectively dividing the value by 2, as a method to avoid bit growth and overflow. The shifting schedule determines at which specific butterfly stages of the FFT this shift must occur, with optimal schedules shifting just enough to avoid overflow without losing too much of the signal of interest. Efficient shifting is an important consideration in radio astronomy, where the signals of interest are often small compared to *Radio Frequency Interference* (RFI) and can be lost entirely if data is shifted too aggressively [4].

Another option is to allow for bit growth (to a specific number of bits) within the stages of the FFT, with a re-quantisation occurring after the final stage to reduce the number of bits back to the desired precision. However, bit growth in a design affects clock rate and resource usage - with higher bit widths increasing both. For example, the Xilinx DSP48 slices, blocks of logic combining a multiplier and adder, are limited to 18 bits within the multiplier [19]. Bit widths greater than 18 can be processed by the FPGA logic, but must then be split across multiple multipliers, which in turn utilises more logic resources and increases clock rate.

Rounding

While overflow handling refers to handling values beyond the maximum/minimum value representable by the number of bits allocated to the value, *rounding* refers to handling values with a resolution smaller than that of the resolution of the number of bits assigned to the fractional part of the value. The specific rounding schemes explained here are *truncation*, *round to infinity*, and *round to even* [20]. All examples given in the following explanations round to a final precision of signed $\langle 4,3 \rangle$ bit data.

Truncation removes any bits beyond the representation's precision/resolution without changing the remaining digits. This is the simplest method and requires no additional hardware resources, but has the poorest performance. For example, a signed $\langle 4,3 \rangle$ bit value has a resolution of 0.125 ($\langle 0b001 \rangle$ in binary). Using this precision, the value 0.24 ($\langle 0b0011110101110000101 \rangle$ in binary) would be truncated to 0.125, despite being closer to the value 0.25 ($\langle 0b010 \rangle$ in binary) - a value that is actually representable by the final precision.

Round to $\pm$ infinity adds (for positive values) or subtracts (for negative values) a bit corresponding to half the final representations resolution to the value before truncation. For the case above, half the resolution ($0.125 : \langle 0b001 \rangle$) would be 0.0625 ($\langle 0b0001 \rangle$ in binary). Adding this to 0.24 yields the following:

$$0b0011110101110000101 + 0b0001 = 0b0100110101110000101 \quad (2.21)$$

Truncating the result of this summation to $\langle 4,3 \rangle$ precision now gives $\langle 0b010 \rangle$ (0.25 in base 10) - a value closer to the original than in the case of simply truncating. However, while this is a more desirable result, it still introduces a bias to the set of data as mid-point values (values lying exactly between multiples of the resolution (e.g. $0.1875 : \langle 0b0011 \rangle$)) are always rounded away from 0. This method also requires the use of an adder in hardware, increasing resource utilisation above that of simple truncation.

Round to even performs a similar operation as rounding to $\pm$ infinity, but aims to reduce the bias introduced by rounding mid-point values away from 0. To do this, this method rounds the mid-point value to either positive infinity or negative infinity depending on whichever will result in an even LSB (i.e. an LSB value of 0).

For example, the number 0.1875 ($\langle 0b0011 \rangle$ in binary) lies exactly midway between $\langle 0b001 \rangle$ and $\langle 0b010 \rangle$. Rounding to negative infinity (i.e. $\langle 0b0011 \rangle - \langle 0b0001 \rangle$) results in $\langle 0b001 \rangle$. Rounding to positive infinity (i.e. $\langle 0b0011 \rangle + \langle 0b0001 \rangle$) results in $\langle 0b010 \rangle$. As the latter outputs a value with an even LSB, this value is the final result.

Rounding to even reduces the mid-point bias of infinite rounding by rounding both towards *and* away from 0 over an average of all data values. However, this method requires both an addition and a check against the LSB of the result, and is more resource intensive as the previous two methods as a result.

Quantisation Error

Despite being more resource efficient than floating-point, the lower dynamic range and precision of fixed-point arithmetic typically introduces more *quantisation error* into the system. *Quantisation error* is defined as the difference between a quantised signal and the real-world, continuous signal of which the quantised signal is an approximation [12]. Mathematically, this can be represented as:

$$e = \hat{x} - x, \quad (2.22)$$

where $\hat{x}$ is the quantised signal and x is the continuous signal.

Quantisation error is initially introduced through analogue-to-digital conversion, but the shifting schedules mentioned above, as well as any rounding and re-quantisation that occurs within a design compounds the issue.

Considering all of the above, spectrometer signal processing designs are implemented as a trade off between data widths providing a dynamic range large enough to sufficiently represent the signals of interest, while also optimising resource utilisation, clock rate and data rate. Data width and shifting schedule requirements also depend on the types of signals being received and how much RFI is present. A strong RFI presence may cause power to accumulate within individual frequency channels to the point of overflow, necessitating the use of more aggressive shifting. In the case of the MeerKAT 4k F-engine, PFB input data is 10 bits and output data is 18 bits, allowing for 8 bits of bit growth. The 32k F-engine, however, allows for a bit growth to 22 bits, as it was found that RFI could grow beyond 18 bits for a 16 stage FFT and the required shifting schedule overly degraded the *signal-to-noise ratio* (SNR) [4].

2.4 FPGA FFT Implementations

Field Programmable Gate Arrays (FPGAs) are a type of semiconductor hardware consisting of programmable logic cells that can be reconfigured to perform a wide range of digital logic operations[21]. These logic cells are connected by programmable interconnect. Configuring the logic cells and their interconnect allows a designer to implement multiple digital system designs of different functionality. The configurable logic cell matrices of which an FPGA is comprised allows for parallel operation, with multiple tasks executable per clock cycle. An FPGA is generally comprised of input/output blocks, configurable logic blocks (CLBs), programmable interconnect, block RAM (BRAM), DSP slices, and clock management blocks. Memory and communication interfaces between the FPGA and other hardware infrastructure are provided on the printed circuit board (PCB) on which the FPGA chip is mounted.

Configurable logic blocks are comprised of two components, flip-flops (FFs) and look-up tables (LUTs). Flip-flops are basic shift registers that latch and hold their current input value until the next clock cycle. A look-up table is a truth table implemented in hardware as a series of precomputed values stored in memory accessible through the select line of a series of multiplexers. DSP slices are blocks of accumulators and multipliers developed to perform the "multiply and accumulate" operations that form the basis of many DSP computations. BRAMs are block of random access memory distributed across the FPGA for storage.

On the lowest level, a digital system is defined by sets of registers, the binary information stored within them and the operations performed on this information. These operations are referred to as *register transfer operations* [22]. *Register transfer level* (RTL) is a design abstraction that represents a digital system defined by a set of registers, register transfer operations and the signals that control these operations and the flow of data in the system.

2.4.1 FPGA Design Flows

FPGA designs can be modelled on a low level with *Hardware Description Languages* (HDLs). These are low level languages used to describe the behaviour and structure of a digital system at the register transfer level. HDL code can be modelled in three different styles. *Structural* modelling specifies the logic gates of a system and their interconnect. *Dataflow* modelling specifies the flow of data in the system via boolean equations. *Behavioral* modelling is a high level description of how the design behaves using algorithms [14]. Two common HDLs are *Very High Speed Integrated Circuit HDL* (VHDL) and *Verilog*. FPGA design tools often incorporate the use of *Intellectual Property* (IP) Cores - predefined logic units that have been developed by a party for others to use, similar to libraries used in software development.

Models are verified in a simulation environment using *testbenches* - wrappers written in HDL that inject the system with input data and verify that the output is correct. Design simulations verify that the behaviour of the model is as expected.

Once behaviour has been verified, design constraints are assigned. Constraints specify the design's performance requirements in terms of timing, pin assignments and logic grouping specific to the FPGA hardware that is being used[23].

A synthesis tool then converts the RTL design and constraints into a gate-level netlist that is specific to an FPGA device. An implementation stage places and routes the netlist onto the FPGA device resources, with timing analysis carried out once this is complete to ensure that the design meets the desired timing performance.

Finally, a bitstream file is generated from the implemented design and uploaded to the FPGA where a hardware verification check is performed and (if successful) the design is put into operation.

The traditional design flow for an FPGA is shown below in Figure 2.7. In this diagram, solid arrows represent the main process of the design flow and dotted arrows represent the design revision process performed whenever a verification test fails.

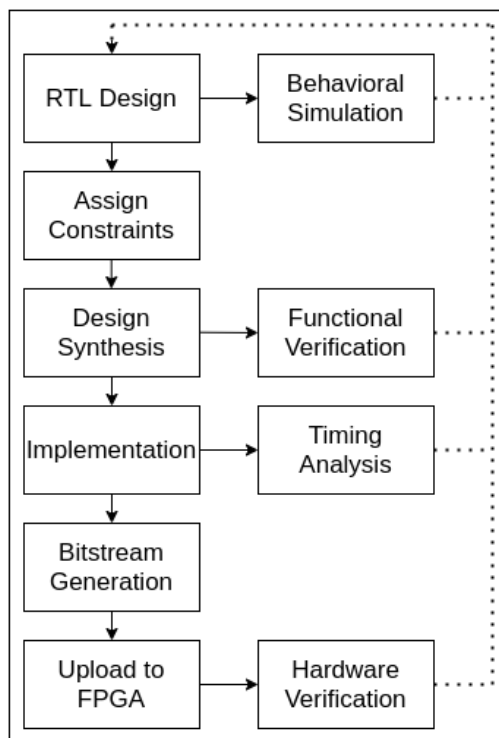


Figure 2.7: Traditional FPGA design flow.

Alternate design flows based on the traditional RTL flow have been created to abstract and facilitate the design process. Some of these flows are presented below to give context to the design of the testing framework described in later chapters and the different FFT implementations that are tested. The hardware used to test the FFT implementations uses a Xilinx FPGA chip, and as such, only an overview of Xilinx tools is given.

Xilinx Design Tools

FPGA manufacturer, Xilinx, provides a host of FPGA design tools. A brief overview of these tools is given below.

Vivado Design Suite

The Xilinx Vivado Design Suite is an Integrated Design Environment (IDE) that consolidates a number of design flows into one environment. These tools include different design methods, I/O and clock planning, synthesis, simulation, placement and routing, debugging and bitstream generation. A summary of some of Vivado's design capabilities is given below; more information on the Vivado IDE can be found in the *Vivado Design Suite User Guide*. [24].

RTL Design: RTL source files can be specified to follow the traditional design flow as shown above in Figure 2.7. VHDL, Verilog, SystemVerilog and XDC are amongst the supported source file types.

IP Design and IP Subsystem Design: Custom IP is made available through the Vivado IP catalogue, allowing for IP to be configured, instantiated and validated. IP can be included in a design in RTL or netlist format. The Vivado IP Integrator allows for the inclusion of IP into IP subsystems via an interactive block design interface.

High Level Synthesis (HLS): HLS enables the design of DSP functions using C/C++ and System C from which RTL modules are synthesised. C-based testbenches can be used to simulate the generated RTL to ensure behaviour is as expected.

Model Composer (previously System Generator)

Xilinx Model Composer/System Generator enables model-based design using MathWorks Simulink. A Xilinx specific blockset included within the DSP Simulink modeling environment via Model Composer allows for the development of FPGA designs without knowledge of the traditional RTL design flow.

DSP building blocks ranging from adders and multipliers to FFTs and filters implemented using Xilinx's IP core generators are available within the Xilinx blockset.

In addition to standard Simulink simulation, timing and resource analysis tools have been built into Model Composer allowing the designer to ensure correct functionality and performance before compiling for hardware.

More information on Model Composer can be found in the *Model Composer and System Generator User Guide* [25].

Vitis Unified Software Platform

All of Xilinx's software development capabilities have been combined into the Vitis software platform tool. Only the Vitis application acceleration development flow for FPGA software acceleration is of relevance to this research project.

Within Vitis, accelerated applications are developed with both a software and hardware component. The software component, developed in C/C++ using OpenCL API calls to manage interactions with the hardware, runs on an x86 or embedded processor. The software application on the processor passes control to kernels running on hardware that run accelerated computations and tasks before passing the output and control back to the processor. Kernels can be developed in C/C++, OpenCL or the traditional RTL design flow with the design imported into Vitis as a kernel. A platform for the targeted hardware is automatically created using Vivado and the accelerated kernels are built into this platform.

More information on Vitis can be found in the *Vitis Unified Software Platform Documentation* [26]

CASPER Toolflow

The Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) is a collaboration that aims to minimise the time-to-science when designing radio astronomy instrumentation [27]. CASPER develops and maintains open-source tools targeting specific hardware in an effort to minimise both the need for designing custom platforms, and difficulty in using pre-existing platforms. The tools include a set of DSP and Xilinx platform libraries that use MATLAB Simulink, Xilinx Model Composer, Xilinx Vivado Design Suite and Python to create FPGA designs.

Contributors to the collaboration either write design primitives in HDL or wrap lower level primitives available through Model Composer into more complex systems and then add these to the CASPER *mllib_devel* library. A user is then able to create a block design in MATLAB Simulink using the blocks from the CASPER libraries and Xilinx Model Composer blockset. This design can then be built for the specified target platform using the *jasper* command. This begins a series of Python script executions that creates and runs a Vivado project to build the FPGA *bit* file, that is then converted into an *fpg* file. This design process is referred to as the CASPER toolflow. The *fpg* file output by the toolflow can then be uploaded to CASPER supported hardware and controlled using a Python library called *casperfpga*.

More information on using this toolflow can be found on the *CASPER ReadTheDocs page* [28].

2.4.2 Existing Implementations

Much work has been carried out in implementing efficient FFT designs for FPGA hardware.

An ultra-high speed 256-point FFT was implemented for Xilinx Virtex 6 devices by Dreschmann et al. 2011 [29], with a chip resource utilization of around 35% and a post place-and-route maximum frequency of 248.3MHz. A pipelined decimation-in-frequency implementation for Virtex 6 devices by Yang et al. 2015 [30] yielded results comparable to the Xilinx FFT v7.1 core.

Kanders et al. 2019 [31] presented a one million-point FFT implementation on a single FPGA board, a technically challenging undertaking. The implementation yielded high-accuracy results at a maximum clock frequency of 233MHz with a large amount of hardware resources left available.

An optimised FFT implemented for FPGA architecture has been proposed by Ingemarsson and Gustafsson 2018 [32]. This proposed architecture implements a single-stream (one sample per clock cycle) radix-2 pipelined feedforward FFT that results in a reduction of required adders. The proposed architecture results in a reduced number of DSP slices and look-up tables when compared to the number required by other optimised and generic architectures.

Xilinx provides an HDL FFT core as part of their IP library, with FFT blocks also available within their Model Composer blockset. The far infrared heterodyne spectrometer instrument, GREAT, makes use of the Xilinx IP core generator to configure the FFTs of their advanced spectrometer core [33].

A wideband (multi-input) FFT is provided as part of the CASPER library DSP blockset. The MeerKAT radio telescope [3] currently makes use of the CASPER FFT (and other blocks available within the CASPER library) as part of its PFB infrastructure.

The *Netherlands Institute for Radio Astronomy*, ASTRON [34], has developed an HDL wideband FFT that was originally implemented for use on Intel devices but has since been modified for Xilinx AMD hardware. This FFT is currently used in the backend of the Westerbork Synthesis Radio Telescope in the Netherlands [35].

The following sections present the specific FFT implementations under test, with Chapter 5 providing details on how these cores were selected.

CASPER FFT

The CASPER toolflow provides a decimation-in-time, real-to-complex FFT capable of performing *multi-stream* and *multi-lane* FFTs. Here, multi-stream refers to the ability to perform separate FFTs of multiple distinct *real* input signals. A single complex FFT core can be used to compute the FFT of two real-valued FFTs, effectively halving the resource requirements for two distinct FFT computations. This is achieved by utilising the even/odd Hermitian symmetry inherent in the real/imaginary outputs of the Fourier Transform of a purely real input. Two real input signals, x_1 and x_2 , may be combined to form the complex signal:

$$y[n] = x_1[n] + jx_2[n] \quad (2.23)$$

Computing the FFT of $y[n]$ results in:

$$Y[k] = X_1[k] + jX_2[k] \quad (2.24)$$

An example of the even/odd symmetry of the real and imaginary parts of $Y[k]$ is shown below:

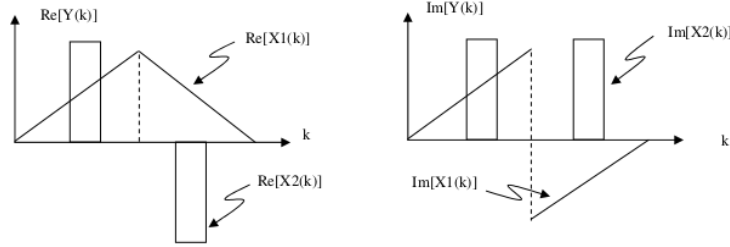


Figure 2.8: Real and imaginary even/odd symmetry of the Fourier Transform of complex input $y[n]$. Sourced from [36].

The individual spectra, X_1 and X_2 , in Figure 2.8 above can then be recovered with the following equations [36]:

$$\begin{aligned} X_1[k] &= \frac{1}{2}(Y[k] + Y^*[N-k]) \\ X_2[k] &= \frac{-j}{2}(Y[k] - Y^*[N-k]) \end{aligned} \quad (2.25)$$

This implementation also employs a *biplex architecture* [37] that utilises the butterfly structure of the FFT algorithm to its advantage. Figure 2.3 shows that half the inputs to the FFT are used in conjunction with the other half to calculate the outputs to the next stage. As a result, the logic resources of these butterfly calculations are idle for half the time as input data is buffered. The biplex core time-multiplexes two streams of input data across this logic as the alternate input is buffered, meaning the logic is never idle. This requires an initial buffering period of both input streams, but ultimately allows for two data streams to share the same computational logic - further reducing resource utilisation requirements.

Beyond improved resource utilisation, the biplex nature of the CASPER implementation allows access to the full complex output value per channel of both X_1 and X_2 every clock cycle. Without this additional architecture, the output of the core would alternate between $X_1[k]$ and $X_2[k]$ each clock cycle as $[X_1[0], X_2[0], X_1[1], X_2[1], \dots, X_1[N-1], X_2[N-1]]$. This biplex architecture therefore decreases the time required to read out a full spectrum worth of outputs for each stream of data.

A multi-lane FFT is referred to as *wideband* and allows multiple samples of a single input signal to be processed per FPGA clock cycle. An FFT capable of processing multiple input samples per clock cycle is required in cases where the ADC supplying digitised data to the FPGA has a sample rate greater than the maximum clock rate of the FPGA. As such, multiple samples must be processed by the FFT block every FPGA clock cycle. The wideband CASPER FFT accepts multiple time-demultiplexed inputs to accommodate this. In this work, the number of simultaneous inputs the block takes per input signal is referred to as the *wideband factor*.

For a total N -point FFT with a wideband factor of L , the CASPER wideband FFT operates first on each series of demultiplexed input samples, performing L many $M=N/L$ FFTs for the first $\log_2 M$ FFT stages. This is then followed by an L point FFT comprised of $\log_2 L$ stages. Figure 2.9 below depicts this wideband architecture.

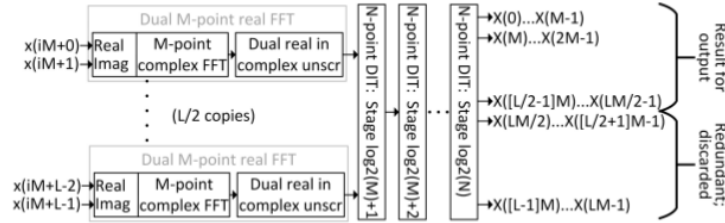


Figure 2.9: CASPER wideband FFT implementation. For a total N -point FFT with a wideband factor of L , the CASPER FFT performs L many $M=N/L$ FFTs in parallel, followed by an L point FFT. Sourced from [38].

Due to the symmetrical output of the FFT of a real valued input, the CASPER wideband only computes the positive frequency component outputs of the output spectrum. This reduces computational logic and storage requirements.

Monroe 2016 [38] presented an optimised, highly efficient implementation of the commonly used CASPER FFT algorithm. Optimisations include the use of a radix-4 butterfly structure and an implementation of a quadplex rather than biplex architecture. The results of his optimisations are shown below in Table 2.1, compared to the most resource efficient instance of the native CASPER FFT with both instances synthesised for the same Virtex-6 FPGA chip.

Length	Module	LUTs	FFs	DSPs	18k BRAMs
16k	CASPER	38 708	85 565	544	142
	Monroe 2016	11 396	16 529	250	128
	Fractional usage	0.294	0.193	0.459	0.901
32k	CASPER	103 923	224 373	1 216	139
	Monroe 2016	18 299	30 158	460	129
	Fractional usage	0.176	0.134	0.378	0.928

Table 2.1: Resource utilization of Monroe’s optimizations compared to the equivalent CASPER implementations.

While Monroe’s implementations are efficient, they have been heavily optimised through hand-placed routing specifically for a Virtex-6 device design. As such, they have not been included in the CASPER toolflow for general use on other hardware, and have therefore not been included in the implementations to be tested and compared in this project.

Astron FFT

The Netherlands Institute for Radio Astronomy, ASTRON, has designed an HDL implementation of a decimation-in-frequency wideband FFT [39] that was incorporated into the CASPER toolflow throughout 2021-2022.

This FFT implementation can be utilised as both a complex-to-complex and a real-to-complex FFT. In the real-to-complex case, the core also employs the Hermitian nature of a real valued FFT to compute the FFT of two real inputs using one complex core - enabling multi-stream processing in a manner similar to that of the CASPER wideband. However this application is not biplex, meaning that the outputs of two real valued inputs, x_1 and x_2 , are only accessible on every alternating clock cycle, delaying the readout of a full spectrum worth of data for each. The non-biplex architecture also results in an implementation that is less resource efficient than a biplex core.

Multi-lane processing is also available, where, for a wideband factor of P , this core computes an N -point FFT of P -many parallel input streams. P -many M -point pipelined FFTs are calculated (where $N = MP$) with the outputs of each pipelined FFT fed into a P -point parallel FFT that performs the final stage of the FFT over M -many iterations. The high-level concept of this wideband operation is analogous to the wideband operation of the CASPER wideband, and as with the CASPER wideband, only positive frequency components of each spectrum are computed. Figure 2.10 shows the wideband architecture for ASTRON's implementation.

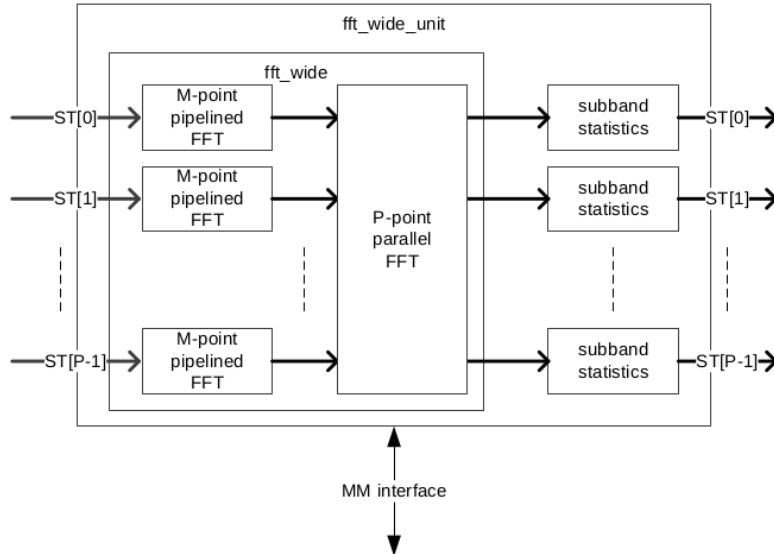


Figure 2.10: ASTRON wideband FFT implementation. For a total N -point FFT and wideband factor of P , the ASTRON FFT computes P -many M -point ($M=N/P$) pipelined FFTs followed by a final P -point parallel FFT. Sourced from [39].

Xilinx FFT

As of January 2023, the most current Xilinx FFT IP core is FFT v9.1 [40]. This core implements the Cooley-Tukey algorithm for transform sizes $N = 2^m$ (for $m = 3$ to 16). Both radix-2 and radix-4 decompositions are used by this core to compute the DFT; however, only the radix-2 implementation is of relevance to this project.

Xilinx FFT IP is available in both Vivado’s IP core generator and the Xilinx Model Composer DSP blockset. Architecture options for both Burst I/O (buffered data processing) and Pipelined Streaming I/O (continuous data processing) are available, with Burst mode using the decimation-in-time algorithm and Pipelined Streaming making use of decimation-in-time.

Model Composer offers three FFT block options. A simple FFT block computes the FFT of a single complex input, while a second FFT block also computes the FFT for a single input but offers more functionality than the simple block in the form of an AXI4-Stream-compliant interface, throttle schemes and more. A Vector FFT is available as part of Model Composers *Super Sample Rate* (SSR) Blockset - this calculates the FFT of vector-type inputs, allowing for multiple data samples to be processed every clock cycle.

The Xilinx SSR FFT is the Xilinx equivalent of the CASPER and ASTRON wideband FFT cores, where Super Sample Rate is the wideband factor of the FFT (i.e. number of samples processed in parallel per clock cycle). This implementation can be used as both complex-to-complex and real-to-complex. However, in the real-to-complex case, no logic is implemented to utilise both real and imaginary inputs as two distinct real valued input streams, and only one real input may be processed by a single complex core. The SSR FFT therefore allows multi-lane processing, but not multi-stream.

Resource utilization of the FFT v9.1 core for several Xilinx device families can be found in the *Performance and Resource Utilization for Fast Fourier Transform v9.1* IP document [41]. Table 2.2 below gives performance characteristics for several fixed-point configurations of the IP core as built for a Virtex 7 device.

Length	I/O Implementation	$F_{\max}$ (MHz)	LUTs	FFs	DSPs	36k BRAMs	18k BRAMs
1024	Radix-2 lite Burst	396	771	1331	2	0	3
1024	Pipelined Streaming	424	3353	5650	32	0	8
8192	Radix-2 Burst	396	1456	2500	10	1	35
8192	Pipelined Streaming	341	4998	9310	50	1	67

Table 2.2: Resource utilization of the FFT v9.1 core on the Virtex 7 xc7vx690t device.

2.5 Other FFT Implementations

While this project only tests FFT implementations that work with the CASPER toolflow and CASPER supported hardware, it is important to mention the literature surrounding other implementations for completeness. This section discusses the performance of various FFT modules implemented on other devices, including the Intel Corporations family of FPGAs, *graphics processing units* (GPUs), and *central processing units* (CPUs).

2.5.1 Intel Corporation Implementation

Intel Corporation acquired programmable logic manufacturer, Altera, in 2015 and now supplies FPGA hardware and design tools as part of their product catalogue. Among the IP cores that Intel provides for use on their FPGA hardware is an FFT core. Documentation for the core can be found in the *FFT IP Core v18.1 User Guide* [42]. Testing this IP core is not within the project scope, as the FPGA hardware and design tools provided by Intel are currently not supported by the CASPER toolflow; however, the performance of this core is briefly reviewed below.

The Intel FFT core is implemented using radix-2 or radix-4 decimation-in-frequency algorithm decomposition for transform sizes $N = 2^m$ (for $m = 6$ to 16). Block-floating point representation (a method of emulating floating-point arithmetic on a fixed-point processor) is used for an FFT of fixed transform length to balance a high dynamic range with minimum memory and size requirements. A variable streaming FFT using either a fixed-point, radix-2 single delay feedback FFT or a floating-point mixed radix-4/2 FFT is supported.

Leclère et al. 2015 [43] implemented an efficient version of the core before Altera was acquired by Intel. This implementation integrated additional elements into the design to achieve an FFT latency of 1084 clock cycles whereas the direct IP core implementation had a latency of 1402 clock cycles. The processing time for 10 FFTs of 2048 points each was also divided by a factor of around 1.94, resulting in a reduction in processing time for a more complex implementation with greater resource usage.

A radix-2 decimation-in-time FFT module was developed by Gonzalez-Concejero 2008 [44] and compared against the performance of both the Intel and Xilinx cores available at the time. The hardware on which the modules were tested were an Altera Stratix II and Xilinx Virtex 4 device respectively. The custom FFT design achieved better resource utilisation but inferior throughput and latency.

2.5.2 GPU Implementations

A graphics processor is comprised of a set of small multi-processing cores each with their own on-chip memory and registers, and an external global memory accessible by all cores, creating a re-programmable, parallel architecture that allows for high performance computing [45]. This parallel architecture makes the GPU well suited for FFT computation.

Moreland and Angel 2003 [46] describe how frequency domain processing had not been implemented on GPUs in the past due to being both *compute* and *I/O bound* - where *compute bound* is a limitation in throughput due to the GPU not being powerful enough to process the data for complicated requirements, and *I/O bound* is the limitation in throughput due to data transport bottlenecks. However, modern GPUs are better suited to the task. Further, Moreland and Angel designed a test system to render and filter an image with an FFT method described in their paper for a GPU and compared the result against the FFTW (a known software FFT implementation described in the following subsection). The GPU FFT was not fast enough for real-time usage with the state of the art of GPU hardware in 2003, and completed four 1024x1024 32-bit floating point Fourier transforms, piecewise complex multiplications and inverse Fourier transforms in 2.7 seconds, while the FFTW completed this operation in 0.625 seconds.

The CUFFT library supplied by *Compute Unified Device Architecture* (CUDA) (a parallel computing platform supplied by technology company, NVIDIA) provides FFT implementations for use on GPU hardware. Information on this library can be found in the *Cufft Library User's Guide* [47]. Cui et al. 2009 [48] studied guidelines that would assist programmers in achieving high performance applications with this library, and achieved improvement performances for both *single-precision*, *general matrix multiplication* (SGEMM) and FFT subprogramms on an NVIDIA GeForce GTX280 GPU. Experimental results released by a more recent study performed by Zhao and Zhao in 2018 [49] found that the CUFFT implementation outperformed the double-precision software FFTW implementation in terms of both performance and run-time.

Heterogenous platform framework, OpenCL, developed by Khronos group provides an FFT library for both GPU and CPU backends. The *clFFT* library [50] supports transform lengths of powers 2, 3, 5 and 7, and supports FFTs of up to three-dimensions. Any OpenCL compatible platforms, including GPUs and Intel FPGAs, are able to utilize this library.

2.5.3 Software Implementations

Many software implementations of the FFT for use on processing units such as CPUs are available. Some of the most commonly encountered of these are described below; however it is important to note that the performance of an FFT in software depends greatly on the processor used - any FFT will perform faster when run on the latest processor compared to the Intel Pentium processor of 1993. While this idea generally holds true for all hardware, all literature described thus far for FPGA and GPU benchmarks have been tested on a specific family of the hardware for a fair comparison. As of 2023, there is currently little in the way of formal research in comparing FFT performance for the same (or equivalent) CPU hardware. As such, these implementations are named below only, with the understanding that the performance of each is highly dependent on the processor on which the software is run.

The *Fastest Fourier Transform in The West* (FFTW) is a C subroutine library for calculating the FFT [51]. The FFT can be calculated for one or more dimensions of an arbitrary length input, for both real or complex inputs. The FFTW project also includes *benchFFT*, an FFT software benchmarking program, that has been used to analyse many other FFT implementations, results of which can be found on the *FFT Benchmark Results* page of the FFTW site [52]. Other commonly used software FFTs include those supplied by MATLAB [53]) and the Python mathematics and science libraries such as *numpy* [54].

Chapter 3

Methodology

“upstream: the quest to solve problems before they happen” - dan heath

The purpose of this research project is to test and compare the functionality and performance (in both simulation and on physical hardware) of different FFT cores developed for use on FPGA hardware. This section describes the Systems Engineering methodology used in the design of a system capable of evaluating the performance of each FFT under test. This approach transforms a set of overarching, conceptual requirements into a functional system.

The development of a *test system* is required to run each of the FFT implementations on FPGA hardware and in a simulation environment, with these FPGA hardware and simulation environments referred to as *test platforms*. Additionally, an *analysis system* is required to process and analyse the outputs of each test to compare performance.

These systems were developed according to specific requirements, which were in turn influenced by requirements specific to radio astronomy applications. An iterative and recursive Systems Engineering approach has been used in the development of each system. A Systems Engineering approach to any problem provides confidence that the needs are understood before any solution is attempted.

Figure 3.1 below depicts a V-model for Systems Engineering. Here, testing and verification is performed for each phase of the project, allowing for a more adaptive design process.

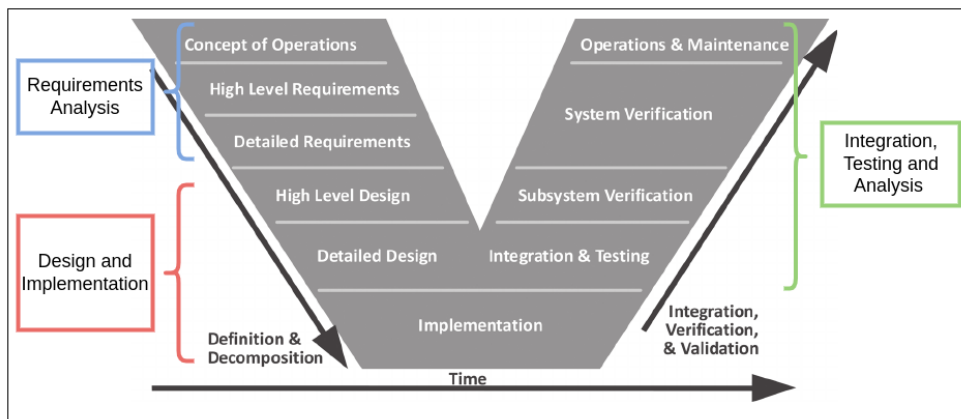


Figure 3.1: V-model for Systems Engineering. Sourced from [55].

In this project, the first three stages of the above model have been grouped together to form a single *Requirements Analysis* stage, and the next three stages form a *Design Specification and Implementation* stage. A third stage, *Integration, Testing and Analysis*, follows these - this is the stage in which the formal testing and analysis takes place. The following sections describe each of these three stages in more detail.

3.1 Requirements Analysis

This is the stage in which the exact desired functionality of the test and analysis system is conceptualised, and the system requirements (both high level and detailed) are defined through the performance of a thorough requirements analysis. High level requirements describe the general performance required of the system, while low level requirements define detailed specifications within these higher level requirements and are derived from specific use cases, limitations and conditions. Figure 3.2 below depicts this hierarchy.

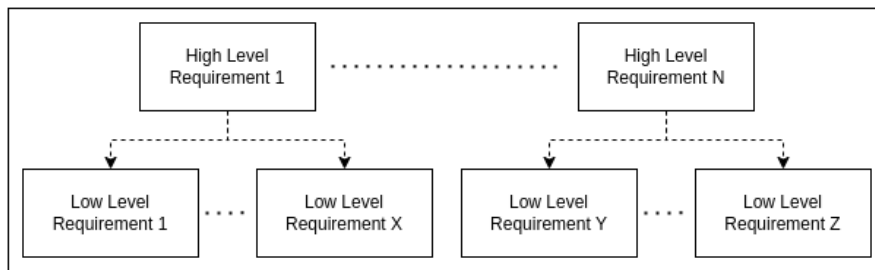


Figure 3.2: Hierarchy of High Level Requirements and derived Low Level Requirements.

The resulting high level and low level requirements inform design decisions made when designing the system. The system was designed to meet general functional requirements as defined by high level requirements to the specifications as defined by derived low level requirements. This design process and the implementation of the final design occurs in the next stage as defined in the following section. A report on the results of the requirements analysis is given in Chapter 4.

3.2 Design and Implementation

In this stage, the test system was specified based on both high and low level requirements and designed accordingly. The design process was iterative, and involved multiple design revisions before a final test system design could be implemented. This stage was where most time was spent once technical development began. This process is shown in the red, dotted box below:

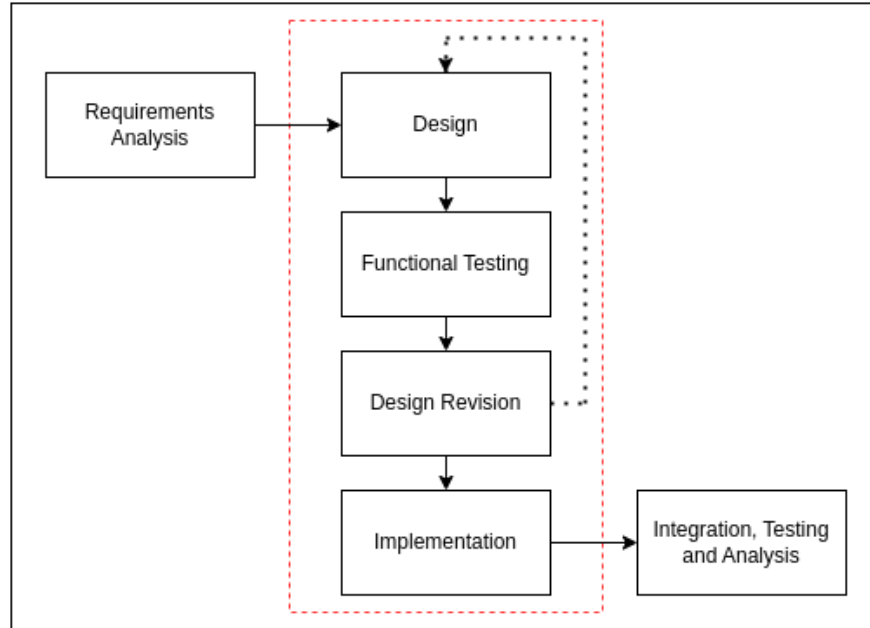


Figure 3.3: Diagram illustrating an iterative design and implementation process.

The first factors to be considered were the selection of design tools to be used and the FPGA hardware and simulation platforms on which the FFTs would be tested. This selection was made based both on the tools and hardware available at SRAO and the high level requirements defined by the requirements analysis process.

The design process involved exploring methods of input test data generation and injection, methods of transmitting and capturing the output spectrum of each FFT and interfacing with hardware and simulation designs on their respective test platforms. The resources available on the FPGA hardware platform and the server in which the FPGA is hosted influenced the means by which communication and data injection/capture requirements are satisfied within the hardware test system. Similarly, control and data flow of the simulation test system depended on the specifics of the simulation environment platform.

A final test system design was then implemented for both test platforms. Functionality verification was performed to ensure that the test system operated as expected, with an iterative debugging approach used to rectify any issues encountered during implementation. The output of this stage was a functional test system, with a detailed description of the design and implementation process presented in Chapter 5.

With a functional test system implemented, an analysis system could be developed to process and analyse the outputs of the test system. This development process, as well as the integration of the two systems to form a complete unit, is detailed in the following section.

3.3 Integration, Testing and Analysis

In this stage, the selected FFT implementations were tested using the test system developed in the previous stage and their outputs analysed. The development of an additional system was required to perform this analysis. The development of this analysis system was less time intensive than that of the test system and could only begin once a final test system had been implemented and the exact format of the outputs to be processed and analysed was known. However, a detailed requirements analysis of this system was performed alongside the requirements analysis of the test system. This aided the development of the test system, allowing for considerations as to output formats that would integrate with the analysis system. The derived low level requirements also influenced decisions as to the tools used to create the analysis system and the methods of analysis to be used based on identified testing criteria.

Before the final tests of the FFTs could be conducted, the test and analysis system components were integrated to form a complete unit. Together these two systems form the required *Test and Analysis System*. The integrated system is shown below in Figure 3.4.

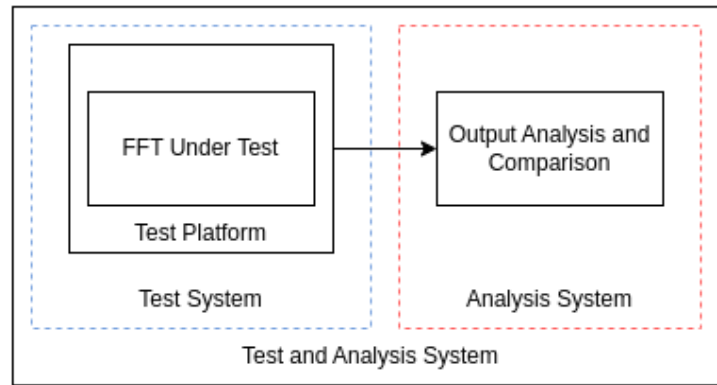


Figure 3.4: Test and Analysis System shown as a combination of distinct Test System and Analysis System components.

Upon the completion of this integration, testing could begin. The implementations were tested against specific criteria that assessed both the fidelity of the implementation output (i.e. ensures that the output of each FFT is correct and the degree to which it is accurate), and the performance and resource utilisation of the implementation's hardware compilation. Chapter 6 provides details on both system integration and formal testing.

Analysis and comparison of the test results allowed for conclusions to be drawn regarding the performance of each implementation. Suggestions for further work were also derived based on experience gained throughout the development and testing processes. These results and conclusions are presented in Chapters 7 and 8 respectively.

This chapter has described the design methodology followed in the development of a Test and Analysis System capable of evaluating the performance of the FFT implementations of interest. The following chapter reports on the outcome of the first stage of this process, a requirements analysis performed to obtain high and low level specifications according to which the system has been designed.

Chapter 4

Requirements Analysis

“not so fast, Fourier Transform!” - omer mahgoub

This chapter documents the results of the requirements analysis outlined in Section 3.1 of the Methodology chapter. The test and analysis systems could not be designed without requirement specifications. These specifications were formed based on the high and low level requirements resulting from the performance of an in-depth requirements analysis. High level test system requirements are specified first, with low level requirements following thereafter.

4.1 High Level Requirements

A preliminary analysis of the required general functionality of the test and analysis systems generated a set of high level requirements according to which the test system was designed. At the highest level, the test system required:

HLR 1 - Input test data on which the FFTs could be performed

HLR 2 - A system architecture providing a means of injecting test data into and reading output data from the FFT under test for both hardware and simulation test platforms

HLR 3 - Appropriate methods of FFT output analysis and comparison

These high level requirements are represented diagrammatically in Figure 4.1 below. In this high level depiction of the required test and analysis system, *HLR 1* and *HLR 2* form the requirements of the test system, while *HLR 3* define the analysis system requirements.

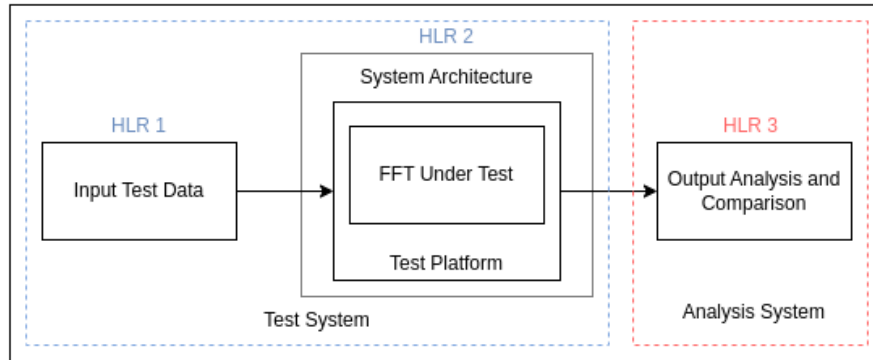


Figure 4.1: High Level Diagram of The Test and Analysis System based on the above high level requirements. *HLR 1* and *HLR 2* form the requirements of the test system, and *HLR 3* defines the analysis system requirements.

While it was expected that some modifications would be necessary to integrate each FFT under test into the system due to differences between implementations, a requirement was imposed that the test system be designed to test each FFT with only minimal changes needed to facilitate the testing of each.

As shown in Figure 3.2 of Chapter 3, lower level requirements stem from the above broad and generalised requirements based on the constraints and considerations of specific use cases. These constraints and considerations and the resulting low level requirements are described below.

4.2 Low Level Requirements

The topics explored within Chapter 2 and discussions with engineers and scientists within SARAO and CASPER informed the decisions made regarding more detailed system requirements. The mathematical and design theory provided by the aforementioned chapter informed requirements regarding the type of input test data used, design considerations regarding data representation and precision, and the specific performance metrics against which each FFT was to be compared. The literature review of Chapter 2 described the different FFT implementations under test, and examined the current state of the art regarding FPGA based FFT comparisons and performance. This guided the development of more detailed requirements for the test system architecture, as the details of how the implementations work and their expected inputs, outputs and interfaces affected the framework of the required test system.

4.2.1 [HLR 1] Input Test Data

Input data requirements were derived based on two considerations. The first considered the requirements of a radio astronomy use case for each FFT. The second considered testing the performance of each FFT from a general perspective, separate to any radio astronomy specific applications. Input signals and their parameters have been derived in their continuous-time form, with sampling to discrete, quantised input data values occurring in the practical implementation of the system.

For radio astronomy specific applications, the following requirements were identified:

LLR 1 - The input signal must be representative of the noise-like signals received from space

LLR 2 - An analogue to narrowband RFI must be present in the input signal to assess the performance of each FFT in an RFI-rich environment

In the general test case, the following requirements were identified:

LLR 3 - Test data must take the form of an input signal that can both verify the functionality of each FFT under test, and facilitate the evaluation of the accuracy of each FFT output

LLR 4 - Noise must be present in the input signal, both to assess each FFTs ability to process noisy signals and to randomise (and therefore reduce the effect of) error due to quantisation (i.e. noise dithering)

A general case requirement states that the input signal must allow for both functional verification of the FFT and the evaluation of the FFT output's accuracy. A periodic signal of known frequency was identified as providing a means of confirmation that the FFT output occurs at the expected frequency channel, with this output also numerically comparable to a well-defined FFT benchmark to determine accuracy. This signal was further identified as a suitable model of narrowband RFI, as a periodic signal of a single frequency can be interpreted as a narrowband signal with a bandwidth of one frequency channel. To this end, a real valued sinusoid of frequency f was selected in satisfaction of requirements *LLR 2* and *LLR 3*.

LLR 1 and *LLR 4* identify the presence of noise as requirements for both generalised and radio astronomy test cases. For this purpose, a *white Gaussian noise* (WGN) sample generated from a zero-mean Gaussian distribution has been selected to model the required noise. For a Gaussian distribution with a *mean*, μ , of zero, the distribution's *standard deviation*, σ , defines the *root mean square* (RMS) value of the generated noise over time interval $[\Delta T = T_2 - T_1]$:

$$\sigma = WGN_{RMS} = \frac{1}{\Delta T} \int_{T_1}^{T_2} [wgn_{\sigma}(t)]^2 dt \quad (4.1)$$

The required input signal has been derived as a function of time that combines a real valued sinusoid of some peak amplitude A with a white Gaussian noise model with an RMS of σ . The continuous-time definition of this function is:

$$x(t) = A \sin(2\pi ft) + wgn_{\sigma}(t) \quad (4.2)$$

While the above equation provides the type of input signal required for both general and radio astronomy test cases, it was determined that signals of differing *signal-to-noise ratios* (SNRs) would be required for each test case. Here, the SNR of a signal is defined as:

$$SNR = 20 \log_{10} \left(\frac{A_{RMS}}{\sigma} \right) \quad (4.3)$$

Where the RMS of a pure sinusoid can be calculated as follows:

$$A_{RMS} = \frac{A}{\sqrt{2}} \quad (4.4)$$

An analysis of test case requirements identified that the general case input would require a large SNR, with values both close to *full scale* (FS) (where full scale is the maximum value representable by the fixed-point representation used within the test system) and as small as possible to test across dynamic range. The radio astronomy case input would need to model a signal typically encountered when observing the sky, with the specifics of this input stemming from consultations with experts in the field. The parameters required for both test case inputs are detailed below.

Case 0 - Radio Astronomy

Discussions with experienced engineers and astronomers within the radio astronomy field informed the selection of parameters that define a suitable representation of a typical signal received by radio telescopes. In this case, parameters of a more powerful noise input modelling a noise-like sky signal and a weaker sinusoid mimicking the presence of RFI were required. Noise RMS and the input signal's SNR have been selected as:

$$\sigma = \frac{1}{8}FS, \text{ SNR} = -10dB \quad (4.5)$$

From the above, the required amplitude of the sinusoidal component could be calculated using Equations 4.3 and 4.4:

$$\begin{aligned} A_{RMS} &= \frac{(\frac{1}{8}FS)}{\sqrt{10}} \\ A &= \sqrt{2} \times A_{RMS} \end{aligned} \quad (4.6)$$

This case has an additional interpretation beyond that of modelling an environment in which RFI is present. Signals of interest within sky signals are often dominated by background radiation noise. As such, assuming an ideal, totally RFI-free environment, the input sinusoid may also be interpreted as a signal of interest concealed among noise.

Case 1 - General

The general test case specifies testing that determines how each core processes data at the extremes of the chosen fixed-point representation's dynamic range. This involves generating a sinusoid with a peak close to *full scale* (FS) and noise with values around the smallest representable number.

The amplitude, A, of the sinusoid for this case has been selected to be 90% FS:

$$A = 0.9FS \quad (4.7)$$

To generate noise at the scale of the system's smallest representable number, the noise model has been constrained to have a standard deviation equal to half the precision of the quantised fixed-point representation of the input data:

$$\sigma = \frac{1}{2}2^{LSB} \quad (4.8)$$

Doing so generates a noise input with values above or below half the precision, with minimum values smaller than the precision. Rounding performed on these minimum values during the quantisation process

will then stochastically toggle the LSB of the quantised value, allowing for the detection of the smallest possible noise values for the given fixed-point representation. Due to the stochastic nature of the noise model, values above the precision will also be produced (albeit more infrequently given the constrained standard deviation), allowing for the testing of slightly higher noise values.

With the selection of the fixed-point representation according to which data in the system is quantised in Chapter 5, the SNR of the resulting signal can be calculated using Equation 4.3:

$$SNR = 20 \log_{10} \left(\frac{A_{RMS}}{\frac{1}{2} 2^{LSB}} \right) \quad (4.9)$$

While this case is meant as a general test of the performance of each FFT core, it can also be interpreted as a worst case scenario of very strong narrowband RFI eclipsing a weak noise-like sky signal.

Signal Frequency

Finally, the requirements of the frequency, f , of the input sinusoid have been identified for both test cases. The resolution of the discrete frequency domain is finite, with *frequency channel resolution* (Δf) determined by the length of the FFT, N , and the rate at which the continuous-time input signal is sampled, F_s , according to:

$$\Delta f = \frac{F_s}{N} \quad (4.10)$$

An input containing a tone that is not an integer multiple of Δf requires a frequency channel output not representable by the FFT for the given frequency resolution. In such cases, spectral leakage occurs as the energy of the input tone is distributed across neighbouring output channels. This is undesirable for an ideal test case, and so the frequency value has been constrained as follows:

$$f = n\Delta f, \quad n \in \mathbb{Z} \quad (4.11)$$

The signal frequency has been further constrained such that the signal is well sampled with a frequency on the lower end of the bandwidth defined by the Nyquist theorem:

$$0 < f < \frac{F_s}{2} \quad (4.12)$$

With the above constraints defined, the derivation of such a function satisfying both general and radio astronomy test cases as described by Equation 4.2 gave rise to the following low level requirement:

LLR 5 - A means of generating the required input signal for use as test data

With test data requirements fully defined, the low level requirements of the system that will contain the FFT cores under test could be established. These requirements are detailed in the following section.

4.2.2 [HLR 2] System Architecture

System architecture requirements refer to the requirements surrounding everything from the injection of input test data to the capture of the output spectra of each FFT under test. These requirements are listed below:

LLR 6 - Suitable simulation and hardware platforms with which to test each FFT

LLR 7 - Simulation designs to test each FFT in simulation

LLR 8 - Hardware designs to be compiled for, and run on, the selected FPGA platform

LLR 9 - A means of communicating with and controlling simulation and hardware test platforms

LLR 10 - A means of injecting input test data into the FFT under test (for both hardware and simulation)

LLR 11 - A means of transmitting and capturing the output spectrum of each FFT under test (for both hardware and simulation)

Simulation and hardware platforms have been selected in such a way that the development of simulation and hardware designs (specified in *LLR 7 and LLR 8*) is possible using the same design tools. This constraint limits the number of software tools and libraries required, and allowed for the potential reuse of design components between simulation and hardware designs.

The length of the FFT affects frequency channel resolution, and therefore overall output accuracy. Due to an increased number of additions and multiplications, larger FFT lengths also affect the possibility of overflow and introduce greater data manipulation in the form of rounding. This informed the decision that the effect of FFT length on output performance should be measured, and simulation and test designs of different FFT lengths should be created. The following three FFT lengths (*N0*, *N1* and *N2*) were selected for testing:

- $N0 = 2^{10} = 1024$
- $N1 = 2^{14} = 16384$
- $N2 = 2^{16} = 65536$

Testing required that control and monitoring of selected simulation and hardware platforms also be possible, where control and monitoring includes methods by which test designs are loaded to the test platform, configured, and enabled and reset as necessary. It was expected that different means of communication would be necessary between simulation and hardware platforms; however, to simplify communication with each FFT test design the following requirement was imposed:

LLR 12 - For each test platform's respective communication method, the communication interface through which designs are controlled and monitored should be the same across test designs

The injection of test data, as referenced in *LLR 10*, requires the quantisation of input signal values to the fixed-point representation required by the FPGA hardware. Quantised test data is given the notation $x[n]$ to represent discrete time values, and is taken to be representative of voltage levels from an ADC of a radio telescope. Combining *LLR 3 and LLR 4* with *LLR 10* resulted in further requirements:

LLR 13 - The fixed-point representation of $x[n]$ must have a dynamic range capable of representing both stronger RFI-like signals and weaker noise-like sky signals

LLR 14 - The method of test data injection must be capable of handling data widths specified by *LLR 13*

An additional requirement arose when considering the differences in how each FFT core accepts input data and outputs frequency spectra data:

LLR 15 - Test designs for each FFT must be able to accept and transmit data in the same manner

While the above requirement would result in additional logic in each design to manipulate data to a standard form, standardising communication and data input/output across FFT test designs would also simplify the development of all communication systems and the system with which the outputs of each FFT were to be analysed and compared. The following section defines the requirements for the latter.

4.2.3 [HLR 3] Output Analysis and Comparison

Discussions with experienced engineers in the field identified two different measures regarding FFT performance that are of interest. The first is a numerical measure of the accuracy and range of each FFT output in hardware and simulation. The second focuses on hardware only, with an analysis of resource utilisation and achievable FPGA clock rates of each FFT. Measuring output performance in a numerical manner introduced the following requirement:

LLR 16 - A benchmark FFT against which the outputs of the FFTs under test can be compared

The selected benchmark FFT had to be a well-documented, customisable implementation of high precision and good performance. The FFTW C-based software library was briefly mentioned in Section 2.5. Both the speed and accuracy of this implementation have been benchmarked against a large number of FFT implementations, where the FFTW consistently performs well [52]. To this end, the FFTW has been selected as the FFT benchmark. Unlike the FFTs under test, the FFTW is a software implementation of a default double precision floating-point type. A separate infrastructure would need to be developed to utilise this FFT library. Despite communication differences, a constraint that the developed FFTW program must accept the same inputs as the FFTs under test and data must be output in the same format was placed to facilitate straightforward system integration.

The following requirements of the analysis system itself have been identified:

LLR 17 - A means of verifying the functionality of each FFT output

LLR 18 - A means of numerical analysis and comparison of each FFT output against the equivalent FFTW output

Plots of each FFT output spectrum against frequency provide a simple way of verifying functionality through confirming that the sinusoidal RFI component of the input test data maps to the expected frequency channel. Though this provides limited information beyond functional verification and other methods of analysis were required.

To ensure even comparison across the different FFT lengths to be tested and any shifting schedules that are applied during testing, the output of each FFT had to first be normalised. For an N-length FFT with output values $X[k]$, the PSD and power of the output spectrum can be calculated using Equations 2.1 and 2.2 in Chapter 2:

$$\begin{aligned} P_{xx} &= \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \\ S_{xx}[k] &= \frac{1}{N} |X[k]|^2 \end{aligned} \tag{4.13}$$

From the above, the normalised PSD, $\hat{S}_{xx}[k]$, can be calculated as:

$$\begin{aligned} \hat{S}_{xx}[k] &= \frac{S_{xx}[k]}{P_{xx}} \\ &= \frac{|X[k]|^2}{\sum_{k=0}^{N-1} |X[k]|^2} \end{aligned} \tag{4.14}$$

The calculation of $\hat{S}_{xx}[k]$ results in a normalised set of power spectral components, denoted in this work as $\hat{X}[k]$, upon which numerical analysis may be performed. Plots of each FFT core's normalised power spectrum can also be produced, with any interesting features and differences between these plots noted and compared.

Forms of Numerical Analysis

With the assumption that the output spectrum of the FFTW is correct, calculating the *mean squared error* (MSE) between the outputs of the fixed-point FFT core under test and the floating-point FFTW equivalent both verifies FFT core functionality and measures quantisation distortion. For an N-length FFT, if $\hat{X}[k]$ denotes the values of the normalised PSD output of the fixed-point FFT core under test, and $\hat{Y}[k]$ the normalised PSD values of the equivalent floating-point FFTW output, the MSE can be calculated as:

$$MSE = \frac{1}{N} \sum_{k=0}^{N-1} (\hat{X}[k] - \hat{Y}[k])^2 \tag{4.15}$$

While the error value can never be zero due to the effects of fixed-point quantisation and arithmetic, the distortion due to these effects would be small, resulting in errors in the scale range $(0 < MSE < 1^{-9})$. This error value would then provide a measure of the effects of fixed-point quantisation against floating-point, and is referred to in this work as *quantisation noise*. A larger error value would indicate an issue in the output of the core under test beyond that definable as quantisation noise, and so this error value could also be used as a means of numerically verifying FFT core functionality.

A calculation of the average power of each spectrum provides another means of measuring information loss within each FFT core's system when compared to the average power of the FFTW. This value is calculated as:

$$\begin{aligned} \text{Average Power} &= \frac{P_{xx}}{N} \\ \text{Average Power (dB)} &= 10 \log_{10} \left(\frac{P_{xx}}{N} \right) \end{aligned} \quad (4.16)$$

How the input signal's SNR is affected by each FFT process is also of interest within this project. The SNR of the output spectrum should be calculated and then compared against both the expected SNR of the generated input signal and the SNR of the equivalent FFTW output. Assuming a sinusoidal tone with a frequency that is an integer multiple of Δf and using the symmetry inherent in the Fourier Transform of a real signal, spectral SNR can be approximated as:

$$SNR \approx 10 \log_{10} \left(\frac{2\hat{X}_{max}}{[\sum_{k=0}^{N-1} \hat{X}[k]] - 2\hat{X}_{max}} \right) \quad (4.17)$$

An additional numerical metric of interest is the *Spurious Free Dynamic Range* (SFDR) of the output for each FFT core. The SFDR of a spectrum gives the difference in power, in dB, between the largest frequency component and its nearest (in magnitude) spur, measuring the range of each FFT under test. SFDR is calculated as:

$$SFDR = 10 \log_{10}(\hat{X}_{max}) - 10 \log_{10}(\hat{X}_{spur}) \quad (4.18)$$

The forms of analysis described above resulted in the further requirement:

LLR 19 - A computing environment in which the above numerical calculations and analysis can be performed

Other Means of Comparison

Beyond numerical performance and accuracy, an additional means of comparison pertaining only to the final hardware test designs is that of resource utilisation and achievable clock rates across the FFTs under test. Larger FFTs require more logic and memory resources in hardware, and the resulting longer and more complex logic path routing typically affects maximum achievable clock rates. Differences between how each FFT under test has been implemented may result in different resource utilisation and timing optimisation. Comparing resource and timing results would require the following:

LLR 20 - Access to resource and timing reports of the FPGA hardware compiles of the final implemented design for each FFT under test

A summary of the above low level requirements is given below in Figure 4.2, where Figure 4.1 is restated with the inclusion of corresponding LLR specifications for each HLR.

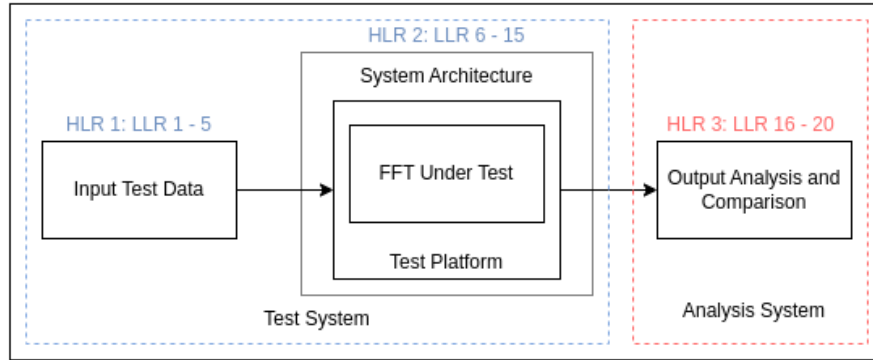


Figure 4.2: High Level Diagram of The Test and Analysis System with corresponding high and low level requirements included.

With the requirements analysis complete and design specifications and testing criteria defined, the design of the test system could begin. The following chapter describes the design process and the implementation of the final test system.

Chapter 5

Test System Design and Implementation

“this is my design” - will graham, hannibal tv series

This chapter details the design process undertaken in the development and implementation of the required test system. The test system as described refers to design components from generation of input data through to the capture of FFT output spectra, with testing performed in two environments - testing in simulation and testing on actual FPGA hardware. Figure 5.1 below depicts this system based on the identified high level and low level requirements of Sections 4.1 and 4.2 respectively, where *test platform* refers to hardware and simulation environments and rounded blocks indicate communication and memory interfaces between the test platform and the rest of the architecture.

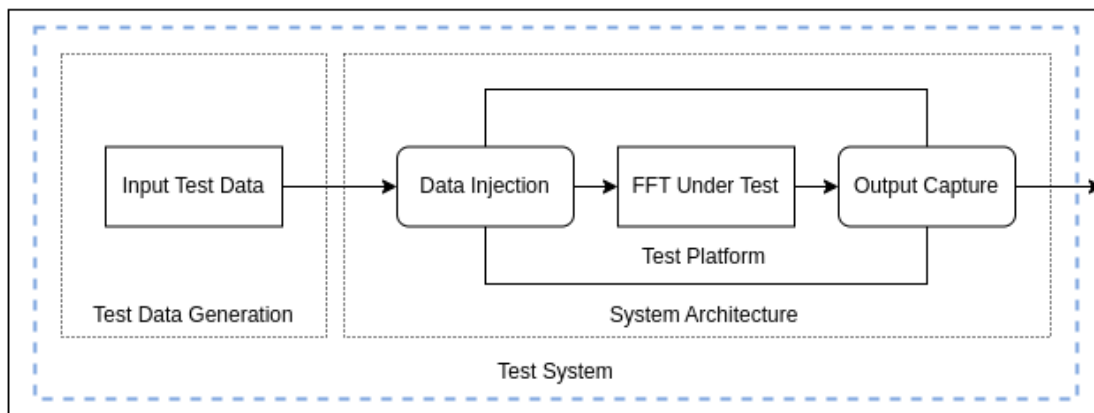


Figure 5.1: Mid-Level Diagram of Test System. The test system refers to design components from generation of input data through to the capture of FFT output spectra, with testing performed in simulation and on actual FPGA hardware. The test platform refers to hardware and simulation environments and rounded blocks indicate communication and memory interfaces between the test platform and the rest of the architecture.

Section 5.1 explains the reasoning behind the choice of design tool used for this project, as well as choice of FPGA hardware used in testing. Concept designs investigating different methods of generating and injecting input data and capturing output data for both hardware and simulation tests are then presented in Section 5.2. The implementation of the final test system is described in the final section of this chapter.

5.1 Design Tool and Hardware Selection

The test system design process could only begin once design tools, a simulation environment, and FPGA hardware had been selected. This section outlines the selection process for the tools and hardware used in this project. Also included in this section is an overview of the work performed to ensure compatibility between selected tools, test platforms and FFT cores. A complete list of selected tools and their required resources is presented in Section 5.1.2.

5.1.1 Selection Process

At the time of conception of this research project in 2020, the *Square Kilometer Array Reconfigurable Application Board* (SKARAB) had been officially deployed as part of the MeerKAT telescope’s backend since 2018 [4] and a suite of SKARABs configured for testing purposes was conveniently accessible on the SARAO network. The SKARAB’s board support package was also designed to be compatible with the CASPER toolflow, with all necessary communication, peripheral interconnect and memory cores available as blocks within the CASPER Simulink library.

Given the availability of SKARAB hardware and the already established use of the CASPER toolflow as a major design tool for the MeerKAT F-engine, the SKARAB was selected as the initial hardware test platform, and the CASPER toolflow adopted as the test system design tool. When the test system design process began in 2020, the latest release of the CASPER toolflow required the Ubuntu 16.04 operating system, MATLAB R2018a, Xilinx Vivado 2019.1, Xilinx System Generator 2019.1 (since updated and rebranded as Model Composer in later releases) and Python 2.7. While work was underway within the collaboration to update the toolflow to more recent OS and software releases, it was decided that support for the SKARAB would be locked to the Ubuntu 16.04 configuration. This decision was made based on the MeerKAT telescope being the largest use case for SKARAB hardware and, with the processing backend complete, no new features from later tools would be required for the telescope’s maintenance.

The FFT cores to be tested were then selected based on availability within the toolflow, with the requirements of radio astronomy signal processing further influencing the selection - namely that each core can process multiple time domain samples per FPGA clock and perform large FFTs (with FFT length $N \geq 2^{14}$). These cores have already been introduced in Chapter 2 as the CASPER wideband, ASTRON wideband and Xilinx SSR FFTs.

Basic simulation tests of these FFT cores uncovered an undocumented incompatibility between the Vivado System Generator 2019.1 version of the Xilinx SSR FFT and MATLAB R2018a, resulting in an erroneous spectral output from the core. This error was discovered and verified through analysis of the output to confirm that the result was not what was expected. The functionality of the SSR FFT core was then tested in simulation using later versions of MATLAB and Vivado tools and was found to work for System Generator 2019.1 when used with MATLAB R2019a and above. Rather than delaying the design process by upgrading the SKARAB-compatible CASPER library to work with later software versions, the initial solution was for all simulation testing to be performed using MATLAB R2019a. A blackbox containing the R2019a Xilinx SSR FFT core would then be created for use in MATLAB R2018a, allowing for the compilation of designs containing this core for use on the SKARAB.

A further problem arose when it was discovered that the Xilinx SSR FFT was not designed for use with the DSP slices available on the family of FPGA chip onboard the SKARAB. A post to the Xilinx forums [56] confirmed that this FFT core was designed to work with DSP48E2 slices only available on UltraScale+ devices, and not the DSP48E1 slices on the SKARAB's Virtex 7 chip. Discarding the Xilinx SSR FFT from the test pool was not desirable - the CASPER wideband FFT had been created specifically because Xilinx did not offer an FFT capable of processing multiple samples per clock cycle. As such, an investigation into the potential of using this new Xilinx core as a replacement to the CASPER wideband is of interest.

At this time, the *Hardware-Firmware Group* (HFG) within SARAO had acquired three Xilinx Alveo U50 Accelerator Cards utilising UltraScale+ FPGA chips. As part of a work package performed for SARAO by the author of this thesis and colleagues, these boards were brought into the latest version of the CASPER toolflow and, consequently, could now be used as a hardware platform in the test system of this project. It was then decided that the SKARAB would still be used as a test platform for the CASPER and ASTRON FFT cores, providing a basis of comparison in the performance of these cores on different hardware platforms. To ensure that all designs for all platforms could be created using the same tool versions, the latest CASPER toolflow release was modified by the author to provide support for the SKARAB.

Details on the complete selection of tools and hardware used in this project are given below.

5.1.2 Selected Tools

This section briefly details the tools used in the development of the test system, and the software library used to interface with and control the hardware test platforms during testing.

Design Tool - CASPER toolflow

This project uses the latest SARAO development branch of the CASPER toolflow library, `mllib_devel`, that has been forked from the latest official release. This branch has the same system and software requirements as the official release, but contains Alveo AU50 development work that has not yet been included in the main toolflow.

The latest CASPER toolflow requirements can be found on the CASPER GitHub [57] and Read the Docs [28] pages. The currently recommended OS is Ubuntu 20.04; however, implementation began prior to this upgrade and the test system has been developed using an Ubuntu 18.04 machine. The other tools used as part of the toolflow are as follows:

- CASPER `mllib_devel` version - https://github.com/ska-sa/mllib_devel, branch: `m2021a`
- MATLAB R2021a with Simulink
- Xilinx Model Composer 2021.1
- Python 3.8 with required libraries installed

Communication Tool - `casperfpga`

The `casperfpga` [58] Python library is used to interact with the selected SKARAB and Alveo AU50 hardware platforms. This library is capable of configuring the boards with different firmware design images and interacting with the design through various communication interfaces.

5.1.3 Selected Test Platforms

The specifications of the simulation and hardware platforms used in the test system are summarised below.

Simulation Platform - MATLAB R2021a with Simulink

Both MATLAB and its accompanying Simulink environment are required for simulation testing, with the same version used as in system design. The following toolboxes are required:

- DSP System Toolbox
- Fixed-Point Designer
- Signal Processing Toolbox

Simulation testing is performed on a Dell Precision 5530 workstation with an Intel Core i7-8850H CPU, 32GB of physical memory and 32GB of virtual swap space memory, and an Ubuntu 18.04 OS.

Hardware Platform - SKARAB

The SKARAB hosts a Xilinx Virtex 7 FPGA chip, 1GbE and 40GbE network interfaces, high speed transceivers, and has space for Hybrid Memory Cubes and ADC mezzanine cards. Figure 5.2 below shows both the block diagram and the internal layout for this board.

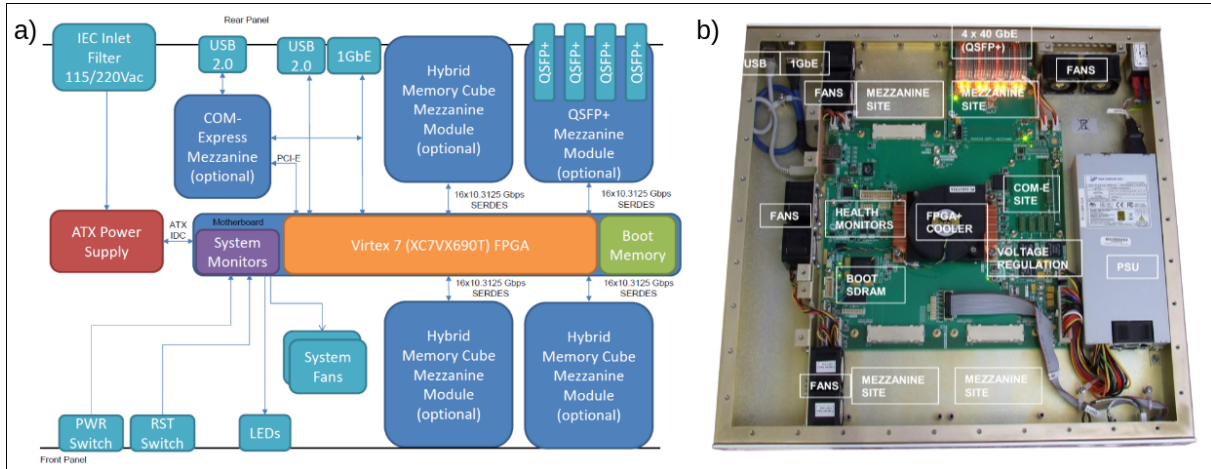


Figure 5.2: a) block diagram and b) internal layout of the SKARAB. Sourced internally from within SARAO.

The SKARAB used in this project is accessible using both 1GbE and 40GbE via a Dell Power Edge R140 server running Ubuntu 14.04. The Python version on this machine is Python 2.7, with the related *casperfpga* library version installed to facilitate communication.

Hardware Platform - Alveo U50 (AU50)

The Alveo U50 Data Center Accelerator Card [59] (referred to as *AU50* from this point forward) is a PCIe form-factor accelerator card containing an UltraScale+ FPGA chip, High Bandwidth Memory resources and 100GbE networking capabilities. This card is shown below in Figure 5.3.

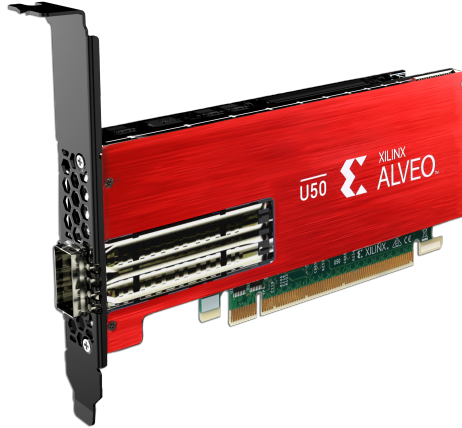


Figure 5.3: Diagram of the Alveo U50 Data Center Accelerator Card. Sourced from [59].

The AU50 used in this project is hosted in a Supermicro SuperServer 5039C-T server running Ubuntu 18.04. To communicate with the card, a Python 3.8 virtual environment with the latest version of *casperfpga* installed has been created, with communication currently only available via the PCIe bus and not 100GbE.

With all required tools and test platforms selected, the design process of the test system could begin according to the requirements described in Chapter 4. The following section describes this process, providing details on initial test system concept designs.

5.2 Test System Design

As FFT core tests were to be performed both in simulation and on hardware, separate system architectures were required for each. Figure 5.4 below depicts a mid-level diagram showing these two architectures within the test system:

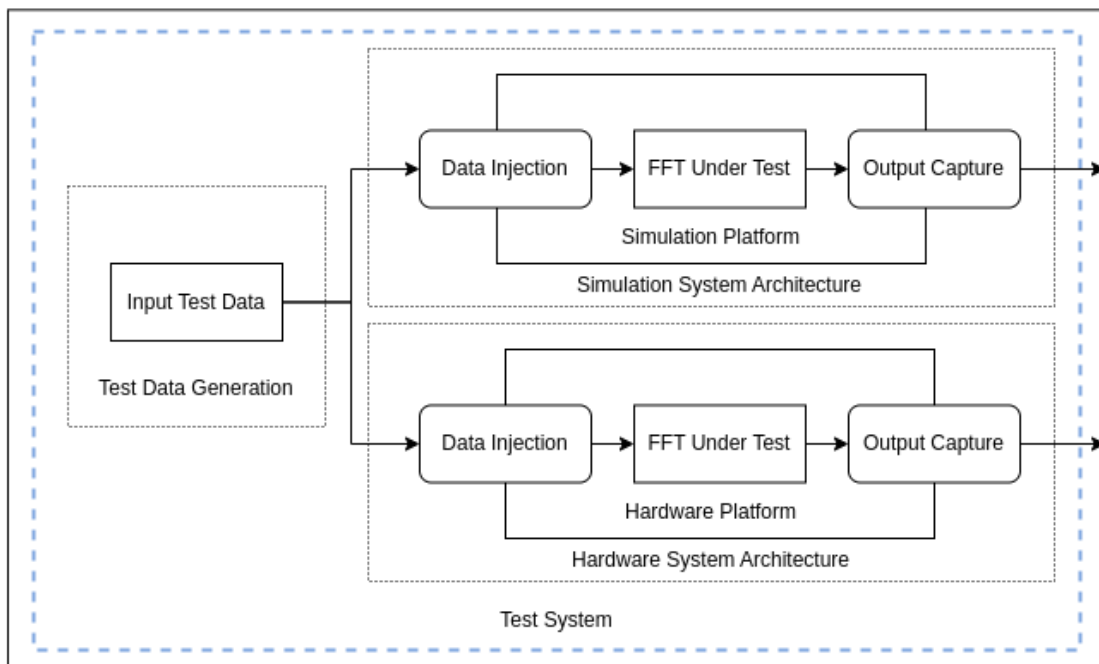


Figure 5.4: Mid-level diagram of the test system containing system architectures for both simulation and hardware test platforms. Rounded blocks indicate communication and memory interfaces between the test platform and the rest of the architecture.

Both simulation and hardware system architectures require a test platform onto which a design containing the FFT core under test can be loaded. The selected design tool, the CASPER toolflow, utilises the MATLAB-based Simulink environment as its graphical frontend for the development of block designs using Xilinx Model Composer and CASPER blockset libraries. These block designs can then be compiled for these hardware platforms using the *jasper* command as described in Section 2.4.2. The blocks supplied by the aforementioned tools are linked to IP cores that can be compiled for various FPGA hardware, but many also come with bit-accurate simulation models that allow for simulation testing. Blocks native to Simulink can also be used within these designs and while they cannot be compiled for hardware, they are used for simulation. The MATLAB and Simulink environment can therefore be used both to create and compile designs for use on hardware platforms and as the simulation platform itself.

Beyond the FFT core under test, the design itself must contain appropriate logic to process input data, allow access to output data and interface with the rest of the system. The design file with which the hardware platform is programmed is a type of bit file known as an *fpg file* generated by the CASPER toolflow from a Simulink design. Loading the simulation platform with a design entails loading the appropriate Simulink design containing the core under test in the MATLAB and Simulink environment.

As Simulink is used in the creation of design files for both simulation and hardware, an ideal test system would utilise the same design file developed in Simulink across both test platform types. This would allow for fair comparisons between simulation and hardware outputs due to analogous methods of data input and capture, and also simplify the integration of both system architectures into the whole test system. Additionally, as the FFTs are to be tested on two different hardware platforms, it should be possible to compile (and use) a single design file for both SKARAB and AU50 hardware platforms with only minimal changes required. Doing so requires that the methods of data injection to and capture from the test platform and all associated logic cores in the Simulink design utilise hardware resources common to both boards. While more complex from a design and development perspective, requiring the implementation of a single Simulink design to test on all platforms reduces implementation and integration complexity.

The test system concept design from which a prototype was created and revised for final implementation satisfies the above requirements. However, initial research investigated methods of input data generation and data flow for hardware testing that would require separate methods for simulation. The various concept designs for simulation and hardware testing are discussed below.

5.2.1 SKARAB-only Concept Designs

Section 5.1 explains that the design process began with the SKARAB having been selected as the sole hardware test platform. As such, the first two concept designs are based around the resources available on the SKARAB. The proposed methods of data injection and capture in *hardware concept designs* (HCDs) 1 and 2 also utilise communication blocks that would require the creation of a separate simulation design.

Hardware Concept Design 1

The first design for hardware testing proposed using a signal generator as the method of test data generation. The ADC installed on the SKARAB would convert the generated analogue signal to a discrete form that could then be processed by the FFT under test. The FFT output would be sent to an awaiting server via 40GbE. Figure 5.5 below depicts this proposed system.

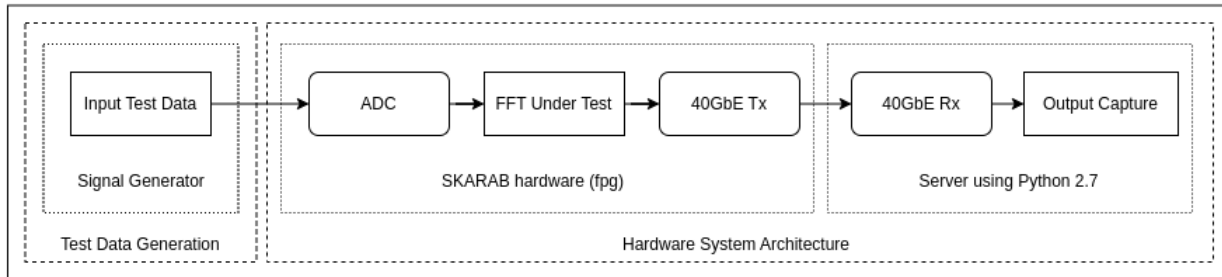


Figure 5.5: First hardware testing concept design. This design proposed using a signal generator as the method of test data generation and an onboard ADC to convert the analogue signal to digital for processing, with the result then output over a 40GbE network.

This system would only be able to test a pure sinusoidal input, as a standard signal generator would not be able to produce noise. Beyond this, even if an arbitrary waveform generator could be sourced, the AU50 does not have an ADC and this method could not be used for both hardware platforms. This necessitated further investigation.

Hardware Concept Design 2

The second hardware test design explored utilising the *Dsim* - a system developed by the *Correlator Beam-former* (CBF) team within SRAO that simulates the output of the MeerKAT digitiser for FX-correlator testing. The *Dsim* is able to generate a sinusoidal tone with noise that is multicast over a 40GbE network. This data could be received by the SKARAB's 40GbE interface, with the FFT output unicast to an awaiting server as in *HCD1*. This system is shown in Figure 5.6 below.

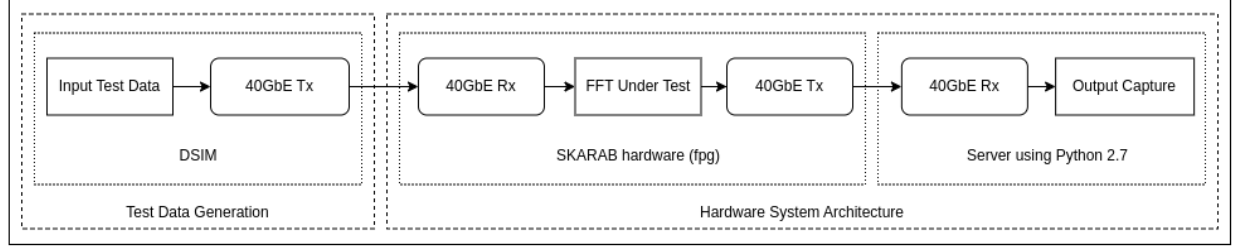


Figure 5.6: Diagram of the second concept design for hardware testing. This design proposed using the *Dsim* (a system that simulates MeerKAT digitiser data) to generate test data, with both input and output over 40GbE.

While this system could be used for a SKARAB-only test case, the addition of the AU50 as a hardware platform and the lack of a fully functional Ethernet core for use on the AU50 in the CASPER toolflow meant that this means of data generation could not be used across both cards.

Further, both *HCD1* and *HCD2* proposed methods of data generation, injection and output that could not be used in simulation testing. This would require the design of a separate simulation design, for which a *simulation concept design* (SCD) is proposed below.

Simulation Concept Design 1

A separate simulation design would be required for either of the above hardware concept designs. The base MATLAB tools provide functions that generate sinusoidal data and data drawn from a standard Gaussian distribution - this initial simulation design would generate the input signal in software within a MATLAB script using these functions. These values would then be made available to the Simulink design via *from workspace* blocks, with the FFT output written to *to workspace* blocks to be saved by a MATLAB script. This is shown below in Figure 5.7.

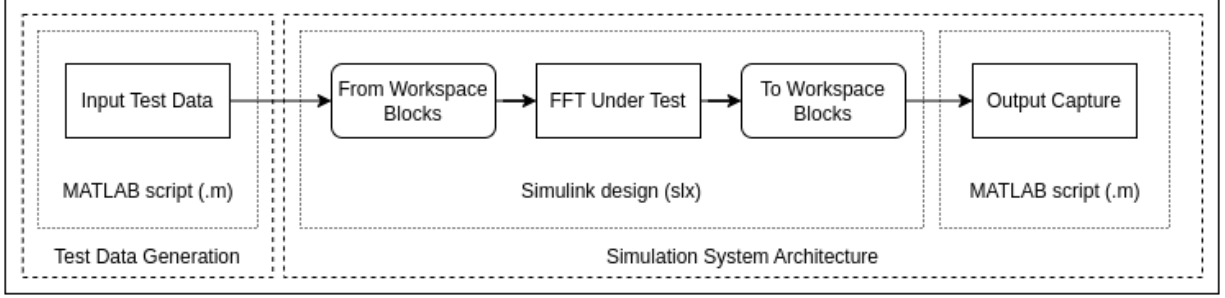


Figure 5.7: Diagram of the first concept design for simulation testing. This design would generate the input test signal in software within a MATLAB script, which would then be accessed by the Simulink design using *from workspace* blocks. The FFT output would then be written to *to workspace* blocks to be saved by a MATLAB script.

This simulation design would be usable no matter the hardware platform, as it is entirely independent of any hardware-specific logic resources. However, it is still desirable to have one type of Simulink design file for testing across all platforms to ensure that test inputs are the same for all tests.

5.2.2 SKARAB and AU50 Concept Designs

Upon the addition of the AU50 as a hardware platform, subsequent concept designs consider resources available on both boards. The following explores designs that utilise resources available to both hardware platforms and methods of data generation usable in both simulation and hardware testing.

Hardware Concept Design 3

The third hardware concept design investigated using the Xilinx *Direct Digital Synthesizer* (DDS) Compiler block available within the Model Composer library for data generation. The core sources sinusoidal waveforms of definable frequency and phase offsets, allowing for the generation of data within the test design itself. FFT outputs would be written to a CASPER library BRAM snapshot block - a memory block that writes a pre-defined number of samples to memory when triggered. Figure 5.8 below depicts this design.

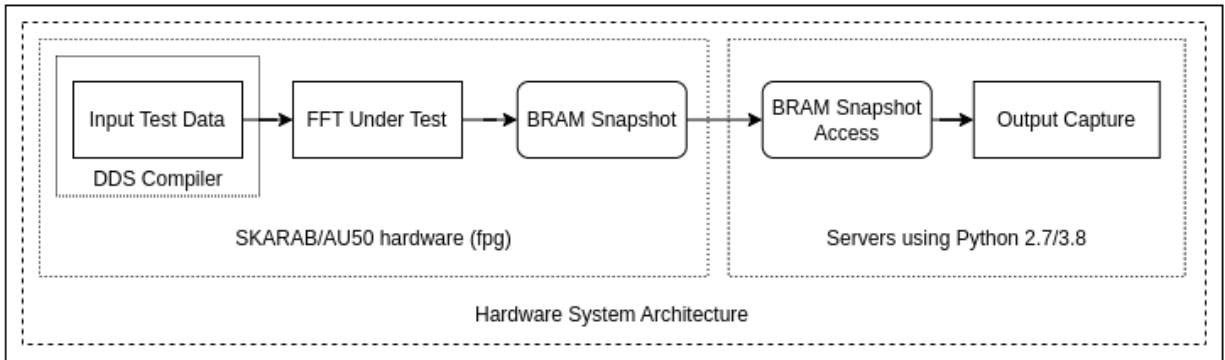


Figure 5.8: Diagram of the third concept design for hardware testing. This design would use the Xilinx DDS Compiler block for test data generation. FFT outputs would be written to a CASPER library BRAM snapshot block - a memory block that writes a pre-defined number of samples to memory when triggered.

The DDS Compiler block does not include noise in its model; however, the CASPER library includes a WGN generator block that could be included in the design for this purpose. An advantage of this design is that the DDS Compiler and WGN generator blocks can be used in simulation, requiring only one design between the two test types.

Simulation Concept Design 2

The simulation design for the above *HCD 3* is shown below in Figure 5.9. Here, the same methods of data generation are used, with the FFT output instead being written to *to workspace* blocks and captured by a MATLAB script.

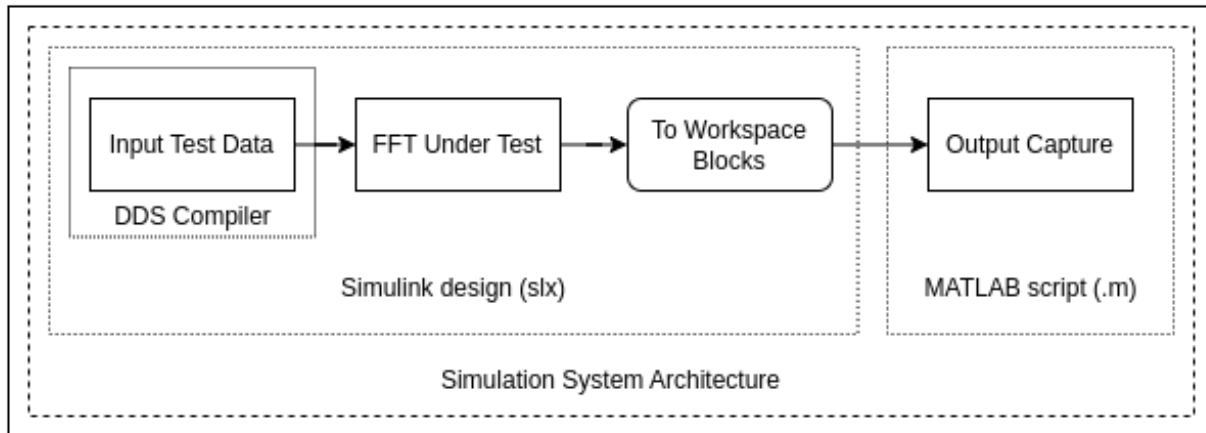


Figure 5.9: Diagram of the second concept design for simulation testing. This design would use the Xilinx DDS Compiler block for test data generation. FFT outputs would be written to *to workspace* blocks and captured by a MATLAB script.

While the above two concept designs are able to fulfil test requirements in terms of simulation and hardware, the reference FFT against which the FFT cores under test are to be compared (the FFTW) must also receive the same input test data. Test data that is generated on hardware must therefore also be captured, requiring either the simulation input to be captured via *to workspace* blocks, or the hardware input to be saved in BRAM - requiring large amounts of memory in the largest FFT test case. Further, while it is possible to seed the WGN block with a number that will generate the same random pattern, and the DDS compiler can be configured to output the same waveform, there is a chance of slight differences between generated data between test runs, which would not be satisfactory.

Hardware Concept Design 4

The final hardware concept design proposed generating input test data in software, in this case using Python, and writing this data to BRAM blocks on the hardware test platform for use as a look-up table. FFT outputs would be captured by BRAM snapshots in the same manner as was proposed in *HCD 3*. This method is shown below in Figure 5.10.

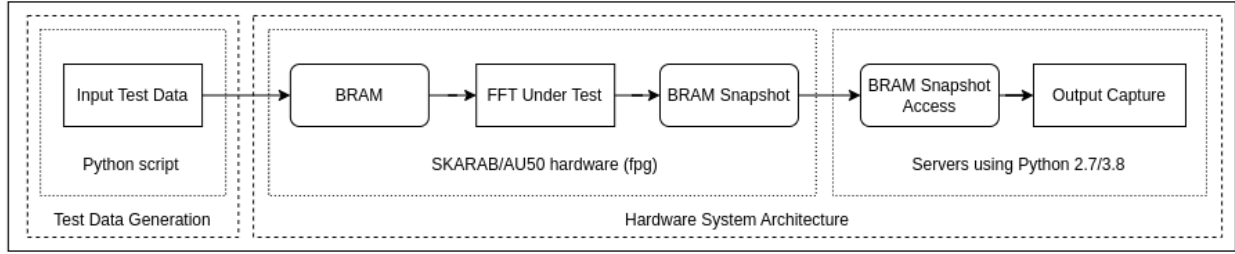


Figure 5.10: Diagram of the fourth concept design for hardware testing. Input test data would be generated in software and written to BRAM blocks on the hardware test platform for use as a look-up table. FFT outputs would be captured by BRAM snapshot blocks.

Generating input test data in Python allows for the generation of sinusoidal data and data drawn from a Gaussian distribution of specifiable μ and σ . A single set of test data long enough for all tested FFT lengths could be generated and saved to file for use in all hardware tests.

Simulation Concept Design 3

The BRAM blocks included in the CASPER library allow for initial values to be written to the block's simulation model as an array, meaning that the Python generated test data of *HCD 4* could be loaded into the MATLAB environment and passed to the BRAM block in simulation. As proposed in both *SCD 1* and *SCD2*, the FFT output would then be captured by a MATLAB script from *to workspace* blocks. Figure 5.11 belows depicts this concept design.

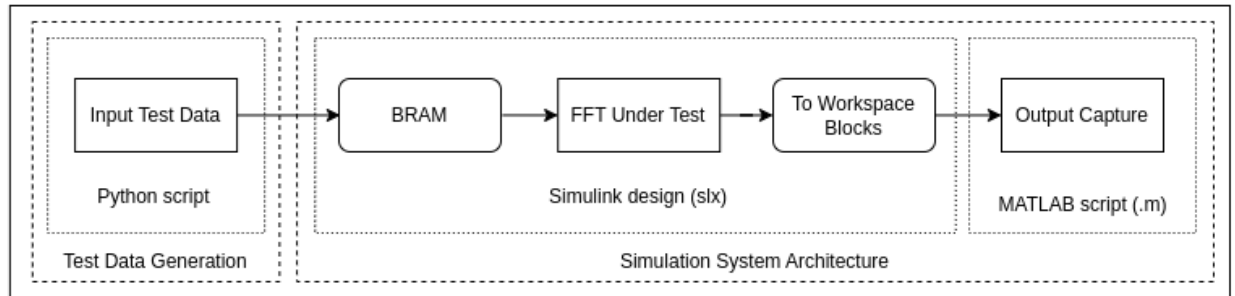


Figure 5.11: Diagram of the third concept design for simulation testing. Input test data would be generated in software, loaded into the MATLAB environment and passed to the BRAM blocks as initial values in simulation. The FFT output would then be captured by a MATLAB script from *to workspace* blocks.

Simulink specific blocks, such as the *to workspace* blocks, are ignored by the CASPER toolflow compile process. These blocks can therefore be included in hardware test designs without issue. To that end, the same design used for hardware testing can be used for simulation testing provided the FFT output is written to both *to workspace* blocks and BRAM snapshots.

The test system proposed in *HCD 4* and *SCD 3* satisfies the requirement that the same Simulink design be used for all platform tests. Further, the method of test data generation allows for the same data to be used across tests as well as input to the reference FFTW implementation. It is this concept design that has been selected for final implementation. The implementation process of this final test system is described in the following section.

5.3 Test System Implementation

The first step of the implementation process was setting up all libraries and environments required for testing. This includes the already mentioned *mllib_devel* library, an additional HDL github-hosted library, *casper_dspdevel* [60], containing a CASPER-compatible version of the ASTRON FFT core, and a Python 3.8 virtual environment in which all necessary CASPER toolflow Python dependencies are installed.

The implemented system has been developed based on proposed concept designs *HCD 4* and *SCD 3*. In this implementation, test data generation is performed by a script written using Python 3, with this data generator referred to as the *Test Vector Generator* (TVG). The TVG generates a discrete set of data called a *test vector* that is written to file for use in the test system. The test vector file is callable by both test system architectures, where it is written to BRAM and processed by the FFT under test.

The FFT output is then written either to BRAM snapshots or *to workspace* blocks depending on the test platform. From here, a Python or MATLAB script accesses the output data, processes it accordingly and saves it to file. The same Python and MATLAB scripts are also used to interface with and control the designs on their respective test platforms. This system is shown below in Figure 5.12.

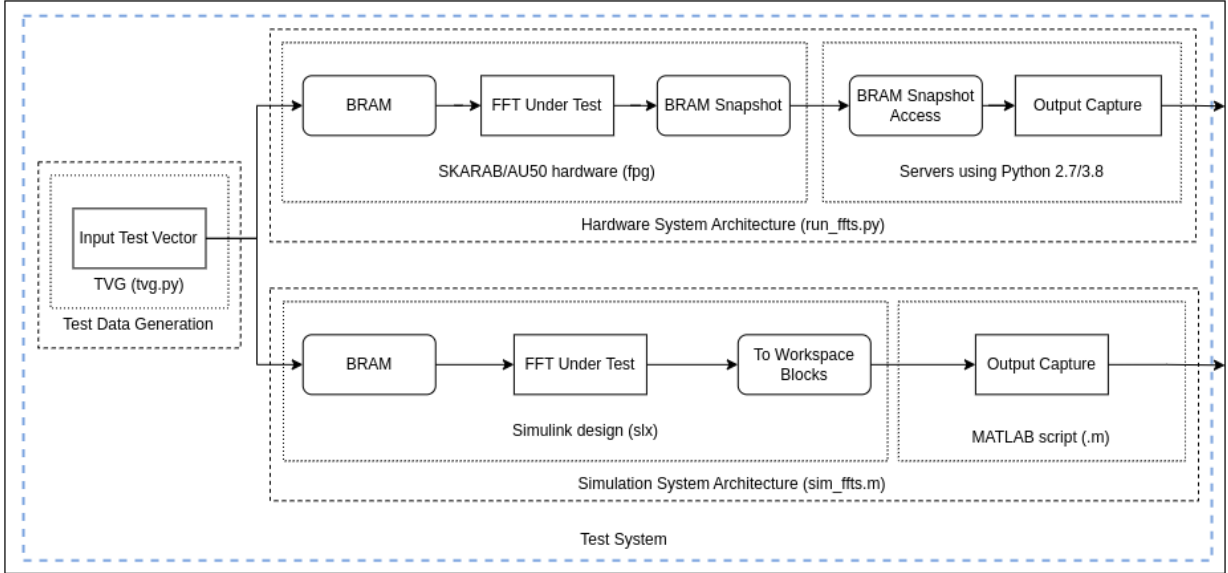


Figure 5.12: Diagram of the implemented test system. Test data generation is performed by a script that generates and writes the test vector values to file. Both hardware and simulation test architectures load the test vector into the BRAM within their designs from this file, perform the FFT and write the FFT output to either BRAM snapshots (hardware) or *to workspace* blocks (simulation).

The following sections describe the implementation of the TVG, the test designs to be loaded onto each test platform, and the surrounding system architectures of each test platform.

5.3.1 Test Vector Generator

The *Test Vector Generator* (TVG) has been implemented as a script written in Python 3 that parses a set of command-line arguments to generate the required test data. The usage of the implemented TVG script, *tv.py*, is shown below in Figure 5.13 and more detail on the specific implementation of the TVG is given after.

```

morag@5530:/media/morag/linux_storage/storage_home/university/test_system/fft_testing$ python3 tv.py -h
usage: tv.py [-h] [--fs FS] [--f1 F1] [--f2 F2] [--tv0] [--tv1]

generates input test vector for either/both test cases

optional arguments:
  -h, --help            show this help message and exit
  --fs FS               sample rate in MSPS (default: None)
  --f1 F1               frequency of first sinusoid in MHz (max <= fs/2) (default: None)
  --f2 F2               frequency of second sinusoid in MHz (max <= fs/2) (default:
                        None)
  --tv0                 case 0 - radio astronomy (default: False)
  --tv1                 case 1 - general (default: False)

```

Figure 5.13: Usage of the TVG python script.

Equation 4.2 describes the continuous-time case of a function satisfying the requirements of a signal that can be used to test the performance of each FFT implementation. The discrete-time form of this function for a sample rate F_s and vector length N is:

$$x[n] = A \sin \left[2\pi f \frac{n}{F_s} \right] + wgn_{\sigma}[n], \quad 0 < n < N - 1 \quad (5.1)$$

While the sinusoidal component is periodic and a full FFT length worth of periodic data could be generated by writing a single period to BRAM and iterating over these values, the noise component must be completely random across the entire FFT length. This requires a vector length equal to that of the FFT length.

The TVG has been designed to generate a vector of a length equal to the greatest FFT length to be tested, with the system architecture shortening the test vector as needed for smaller FFT lengths. The TVG therefore generates a vector of length $N = 2^{16}$ samples. The sample rate, F_s , used to generate the vector is specified at run-time as a command-line argument passed to the *tv.py* script that forms the TVG.

Data generated by the TVG is generated as a set of double-precision floating point values that the test platform system architecture manipulates for correct fixed-point representation within the test platform. The test system has been designed for data with a signed fixed-point representation of $\langle 16, 15 \rangle$ on each test platform, allowing for the representation of values in the range of $[-1:1]$.

Section 4.2.1 described two variants of the derived input signal to test each FFT for both general and radio astronomy cases. Depending on the arguments passed to the script, the TVG generates test vectors for either or both of these cases. For each case, the TVG generates vectors for two different frequencies, also passed to the script as arguments, for the same noise. Checks calculating the actual SNR of the generated test vector in both time and frequency domains and the actual RMS values of each generated component are also performed. Finally, plots for each test vector are generated.

Each test vector is then saved to file using the convention *"tvX_Fs_msp_s_fmhz_YdBsnr"*, where X is the case type, F_s the sample rate, f the sinusoidal frequency, and Y the SNR value. The corresponding plot for each test vector is saved as a PNG file using the same naming convention. The checks calculated for each test vector are collated in a single CSV file for ease of reference. The flowchart in Figure 5.14 depicts the general flow of the TVG for each vector case.

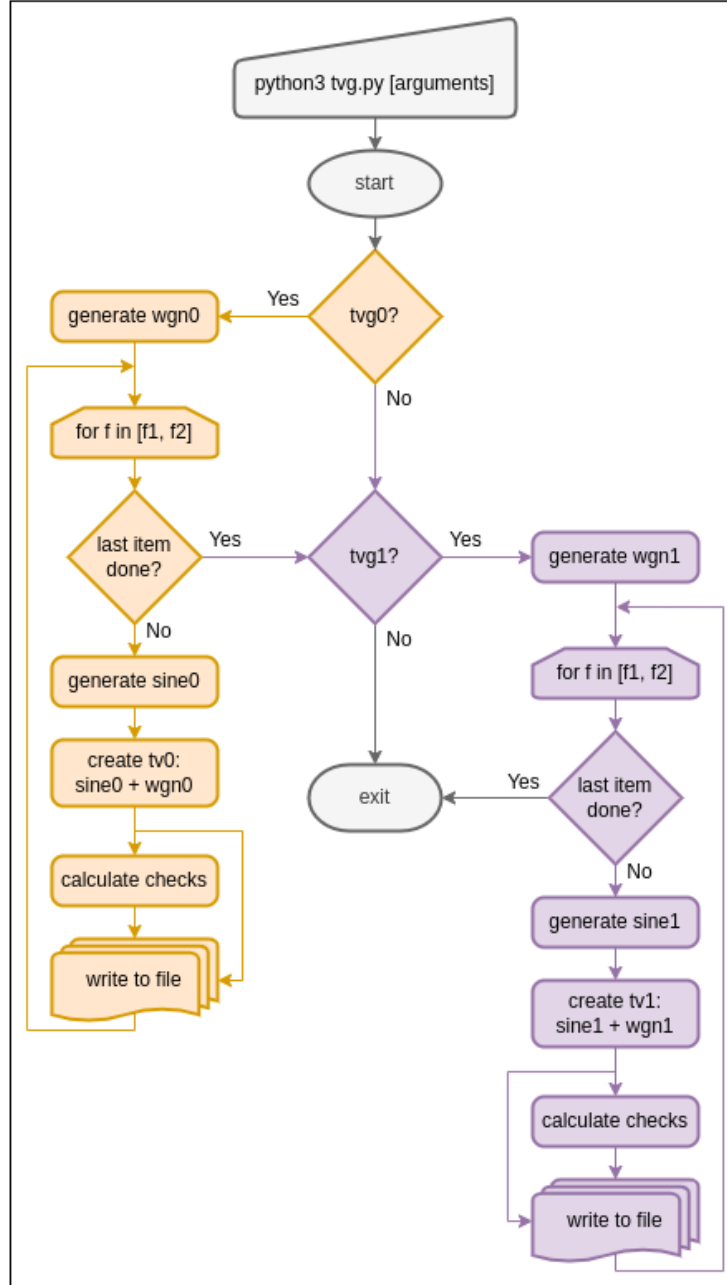


Figure 5.14: Flowchart of TVG operation. The TVG is capable of generating two test vectors based on the arguments passed from the command line. Both test vectors are comprised of a sinusoidal and WGN component, with the signal-to-noise ratio depending on the test vector case.

The noise RMS value, σ , for the general test case can be calculated using Equation 4.8. The index of the LSB in the bit pattern of a $\langle 16, 15 \rangle$ representation is -15 (i.e. 15 bits to the right of the binary point), resulting in $\sigma = 0.5 \times 2^{-15}$. This noise RMS value, coupled with a sinusoidal amplitude of 0.9 results in an SNR of 92dB.

The code written in the implementation of the TVG can be found in the Github repository for this project [5]. The output of this script is a set of files containing the values of test vectors fulfilling the input signal requirements to test for both radio astronomy specific and general uses cases of each FFT core. The BRAM inputs of each FFT test design can be loaded with these test vector values for processing by the FFT. The following section describes the implementation of these FFT test designs for each FFT core.

5.3.2 Test Design Implementation

This section describes the implementation of the Simulink design used both for simulation testing and compilation of the fpg file used for hardware testing. There are slight differences between FFT cores in the expected format of incoming data and how each core structures their output data. As such, differing logic on the inputs and outputs of each core is required to process incoming data from the BRAMs and write outputs. However, despite slight data processing logic requirements, many parts of the design remain common between cores. Figure 5.15 below depicts the template of a base design used to test each core, for a wideband factor of L .

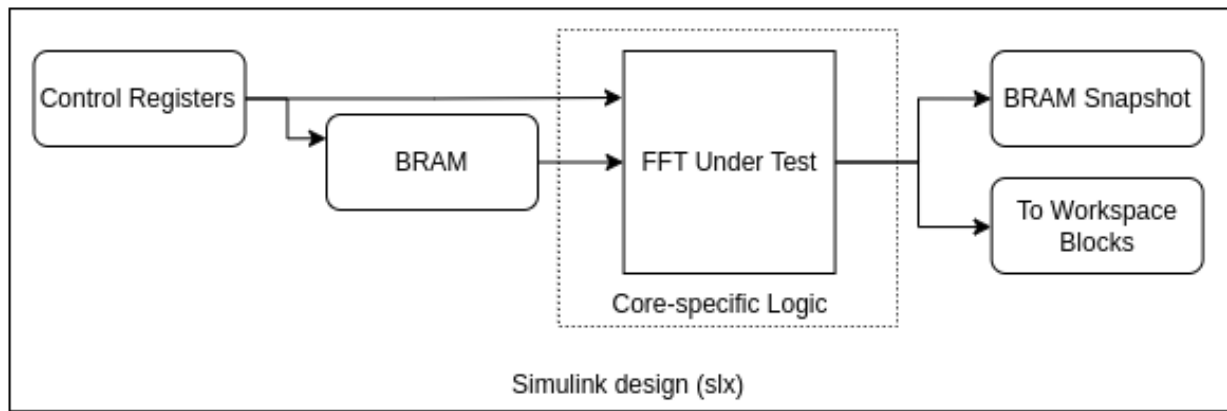


Figure 5.15: High Level Diagram of base Simulink design for FFT core testing. There are slight differences between FFT cores in the expected format of incoming data and how each core structures their output data. Differing logic on the inputs and outputs of each core is required to process incoming data from the BRAMs and write outputs, but in all cases, a base design comprised of control registers, BRAMs, the FFT under test and BRAM snapshots (hardware) or *to workspace* blocks (simulation) is required.

The following section describes the implementation of the base design containing commonly used features, with specific considerations needed to integrate each core into this design described in subsequent sections.

Base Test Design

All Model Composer designs require the inclusion of a *System Generator* block that provides information on which hardware the design is to be compiled for and at what clock rate. The CASPER toolflow provides *Platform blocks* for each supported platform that populates this System Generator block with the relevant information.

As the Simulink design is used for simulation testing and compilation to an fpg file for hardware testing, it has been implemented using logic resources available on both the SKARAB and AU50 hardware platforms. However, designs to be built for each platform must contain the relevant Platform block. Additionally, the SKARAB requires that a 40GbE core be instantiated in each design for communication purposes. A

40GbE subsystem instantiating this core is included in every design for the SKARAB, as well as the relevant SKARAB Platform block. Designs for the AU50 require only the correct Platform block be included and no specific communication core need be included. Figure 5.16 shows these requirements for each hardware platform.

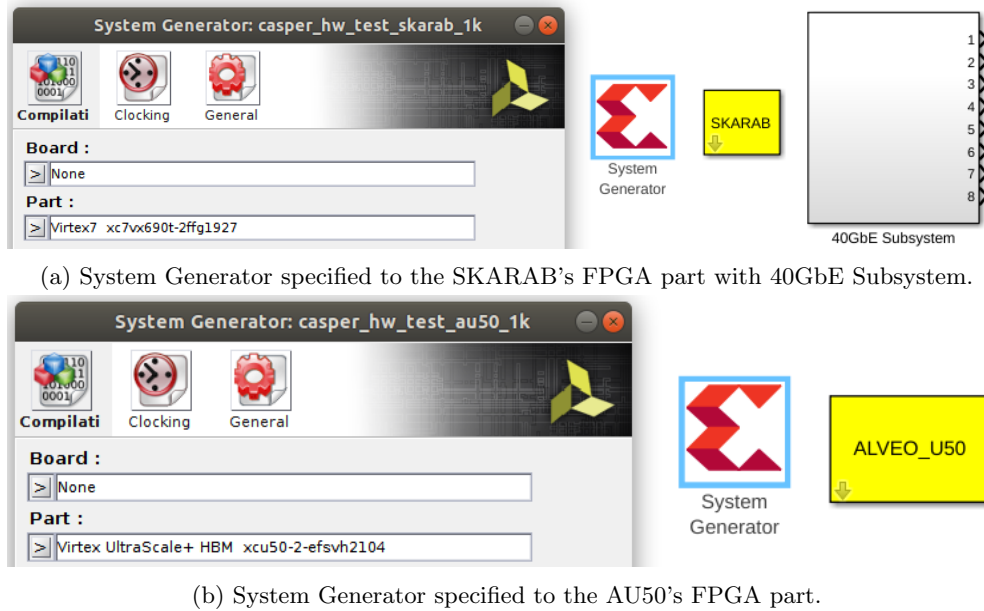


Figure 5.16: Diagram showing blocks required specific to each hardware platform.

Aside from requiring the relevant Platform block to build for a specific hardware platform and the 40GbE core required for communication with the SKARAB, all other logic within the implemented base design is common to both hardware types.

For a wideband factor of L , each FFT processes L -many parallel inputs derived from a single signal of length N decomposed into L -many down-sampled branches. One BRAM is required to store a single branch, requiring a total of L -many BRAM inputs. While BRAMs can be configured to store 16-bit data samples, the AU50 software communication architecture within casperfpga currently only supports 32-bit data writes. To maintain consistency between all designs, 32-bit BRAMs are used across all test platforms. Logic is included in the Simulink design to convert from 32-bit data to 16-bit and enable the representation of $\langle 16, 15 \rangle$ fixed-point fractional values. Typical astronomical implementations may use 18-bits to represent data within the FFT; however, a 16 bit representation has been used in this system to simplify output capture to snapshot blocks that are designed to store samples that are integer multiples of 16-bits.

Figure 5.17 below depicts the base design implemented in Simulink for $L = 4$. Here, reset and enable signals control a BRAM address counter and the FFT core. The FFT itself and all peripheral FFT-specific logic have been contained within a subsystem, representing the integration of each FFT core into the system. The output of the FFT is then written to snapshot and *to workspace* blocks as separate *real* and *imaginary* components of the complex output.

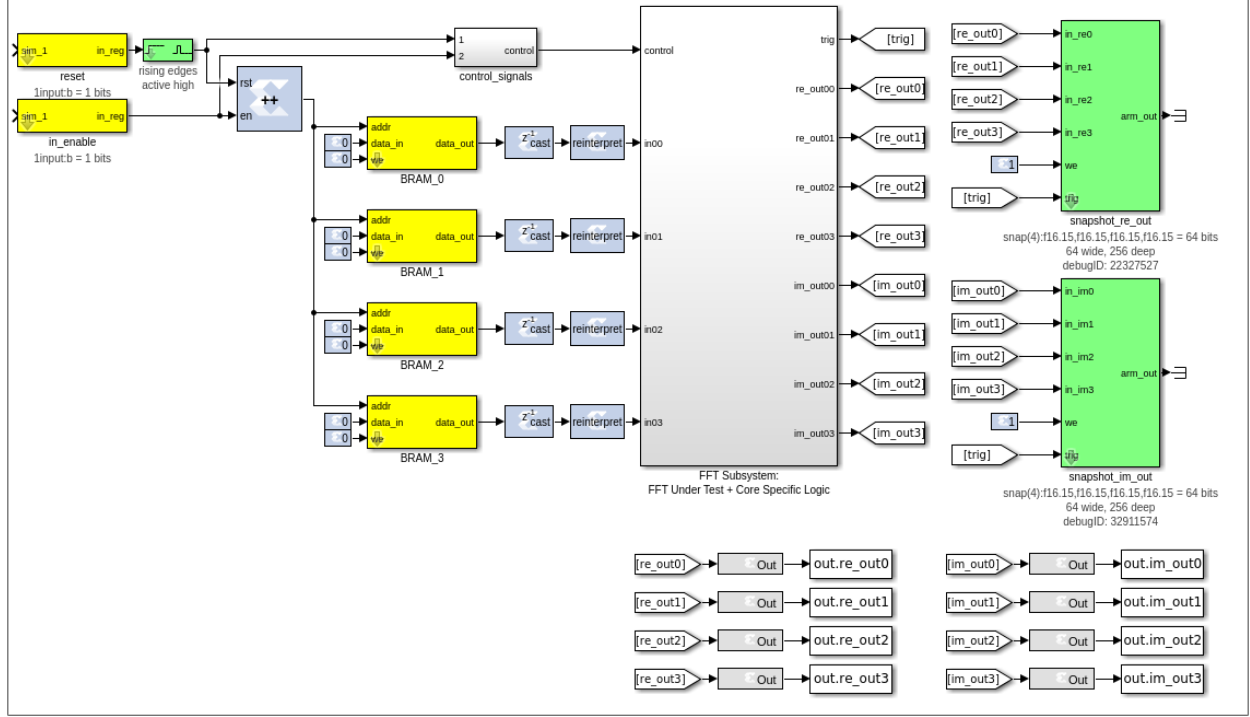


Figure 5.17: Simplified Simulink block diagram of the base design for $L = 4$. Reset and enable signals control a BRAM address counter and the FFT core. The FFT subsystem represents each of the FFTs under test. The FFT output is then written to BRAM snapshots (hardware) and *to workspace* blocks (simulation) as separate *real* and *imaginary* components of the complex output.

As the total length, N , of the input signal must equal the length of the tested FFT to ensure fully random noise across the entire FFT window, the depth of each BRAM must differ between designs that test different FFT lengths. Tested FFT lengths have been selected as $[2^{10}, 2^{14}, 2^{16}]$ for two wideband factors, $L = 4$ and $L = 8$. For each case BRAM depth is calculated as:

$$\text{BRAM depth} = \frac{N}{L} \quad , \quad N \in [2^{10}, 2^{14}, 2^{16}] \quad , \quad L \in [4, 8] \quad (5.2)$$

The BRAM address width, and therefore bit-width of the address counter, is then calculated as:

$$\text{Address width} = \log_2(\text{BRAM depth}) \quad (5.3)$$

Output data for each FFT is configured to the same $\langle 16, 15 \rangle$ representation as the input. The output snapshot of each design captures a valid spectrum worth of data from the FFT output. However, the format of the FFT output varies between FFT cores, requiring snapshot blocks of different depths. The following sections detail these, and other, differences between the cores that required consideration when integrating each FFT into the base design.

ASTRON

The ASTRON wideband FFT core can be configured to process a single complex input or two real-valued inputs using the method described in Section 2.4.2. When used as the latter, the real and imaginary inputs of the core accept two separate signals as inputs. Figure 5.18a below shows the ASTRON core configured for $L = 4$ and FFT length N . Figure 5.18b depicts timing and format diagrams for the input (top) and output (bottom) of the core for two input signals, $x[n]$ and $y[n]$. Here, $x[n]$ is passed to the core's real input, while $y[n]$ is passed to the imaginary input.

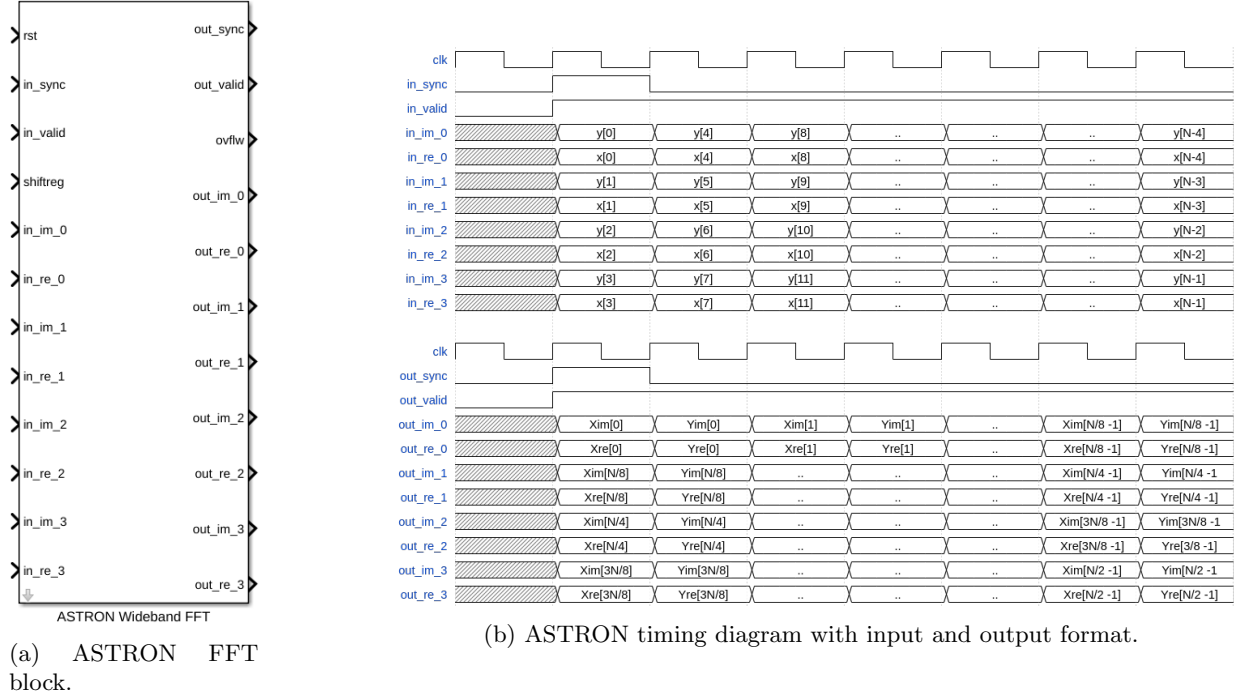


Figure 5.18: ASTRON FFT block and timing diagram. When used to process two real-valued inputs, $x[n]$ and $y[n]$, the real and imaginary inputs of the core accept these two separate signals as inputs respectively. This core expects an input sync pulse on the same clock cycle as valid input data, and generates an output sync pulse on the same clock cycle as the first sample of valid output data.

Due to the symmetry of a real-valued Fourier Transform, the ASTRON core discards negative frequency component outputs. As such, only half the spectrum for each input is computed by the core, requiring less storage resources when processing two real signals. A limitation of this core is the lack of decoding logic that allows it to only process a single input signal, meaning it always implements logic required for two input signals. In the case of these tests, only one real signal is processed and as it is not possible to configure the core to process only one signal, this results in greater resource usage. In the implemented design, the real input of the core is set to zero, and test data is sent to the imaginary input.

As illustrated in Figure 5.18b, data is output as streams of consecutive frequency channels divided amongst parallel output lanes. The outputs, $X[k]$ and $Y[k]$, of both inputs are interleaved, with only one of each available per clock cycle. As such, N -many samples of output data must be captured and the snapshot depth is set accordingly for all tested values of N .

The ASTRON and CASPER FFT cores both include a *sync* input and output for data synchronisation and alignment. As shown in Figure 5.18b, the ASTRON core expects an *in_sync* pulse on the same clock cycle as valid input data and outputs an *out_sync* on the same clock cycle as the first sample of valid output data. The generation of the input *sync* is included in the *control_signals* module in Figure 5.17, with the same logic used for both ASTRON and CASPER cores. This logic is shown below in Figure 5.19, with pulse generation taking two clock cycles in logic post-reset.

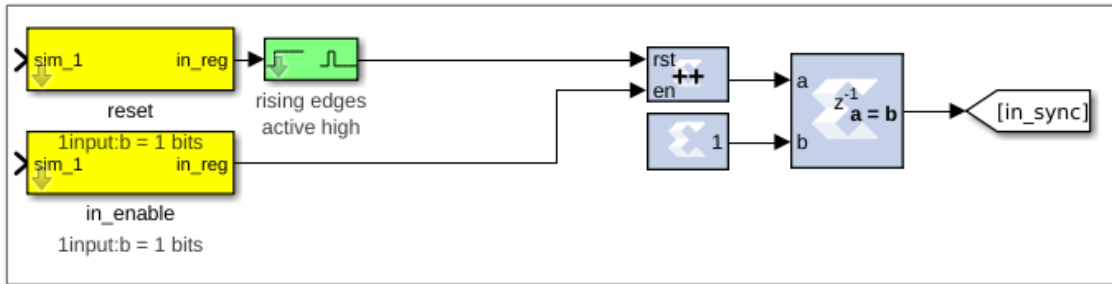


Figure 5.19: Generation of *sync* input. Both ASTRON and CASPER cores provide input and output sync lines for data alignment purposes. The input sync pulse is generated using a counter and relational logic that is driven by software registers at run-time.

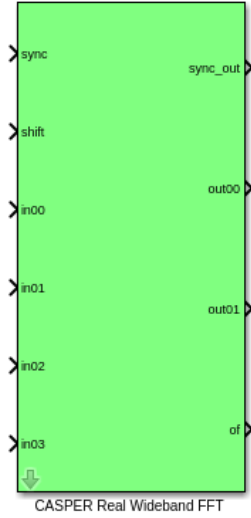
Also included in the *control_signals* module for this core is the line driving *in_valid*. This input is driven by the main *in_enable* register; however, a delay with a latency of two is required to correctly align this signal with the *in_sync* pulse and data input.

The snapshot block itself begins data capture one clock cycle after a pulse is received by its *trig* input. A pulse on *out_sync* signifies that the core has begun to output valid data, and this pulse is used to trigger the output snapshot blocks. As *out_sync* is aligned to the same clock cycle as the first valid data sample, output data is delayed by a single clock cycle before being input to the snapshot block to ensure all output samples are captured. The *out_sync* pulse is also written to a *to workspace* block for data alignment in simulation.

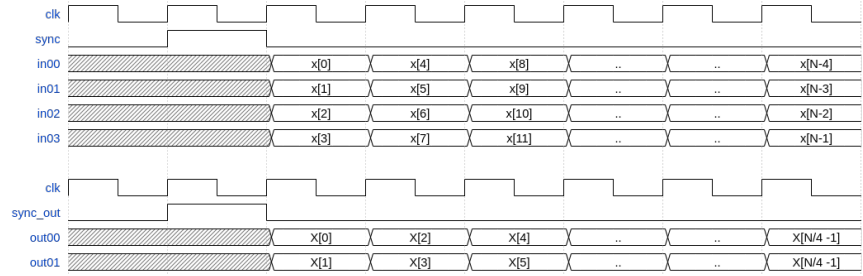
The FFT core itself has been configured to use default synthesis and implementation parameters, perform *infinite rounding* and use a *saturation* overflow scheme. The *guard enable* option has been selected to prevent overflow for input values $> 50\%FS$, with two guard bits added to the MSB of the input data representation, and the *stage data width* has been increased from 16-bits to 18-bits. The data processed by the core is therefore scaled down by a factor of 2^2 , as data representation has been increased from $\langle 16, 15 \rangle$ to $\langle 18, 17 \rangle$. Output data width has been kept at $\langle 16, 15 \rangle$, with a re-quantisation occurring within the core. The final design for the ASTRON core can be seen in Appendix A: A.1 through A.4.

CASPER

The CASPER Real Wideband FFT functions similarly to the ASTRON core. This core, however, only processes real-valued signals and can be configured to process one or multiple input signals. Figure 5.20a shows the CASPER core configured for a single input signal, $x[n]$, with $L = 4$ and FFT length N . A timing diagram with input and output data formats for this configuration is illustrated in Figure 5.20b.



(a) CASPER FFT block.



(b) CASPER timing diagram with input and output format.

Figure 5.20: CASPER FFT block and timing diagram. This core only processes real-valued signals and can be configured to process one or multiple input signals. An input *sync* pulse is expected one clock cycle before the first valid sample of input data, and an output *sync* pulse is generated one clock cycle before the first valid output spectral sample.

The biplex architecture of this core allows access to the output of multiple signals per clock cycle, decomposing the frequency channels of each output's spectrum along parallel lanes in the same manner as input data, as well as providing more efficient resource usage. As with the ASTRON core, only positive frequency channels are output by the CASPER FFT, with output data instead decomposed by frequency channel amongst parallel output lanes. The number of output lanes per input signal is $L/2$, and a positive spectrum worth of data is output every $N/4$ clock cycles. For a single input signal, only $N/2$ output data samples must be captured.

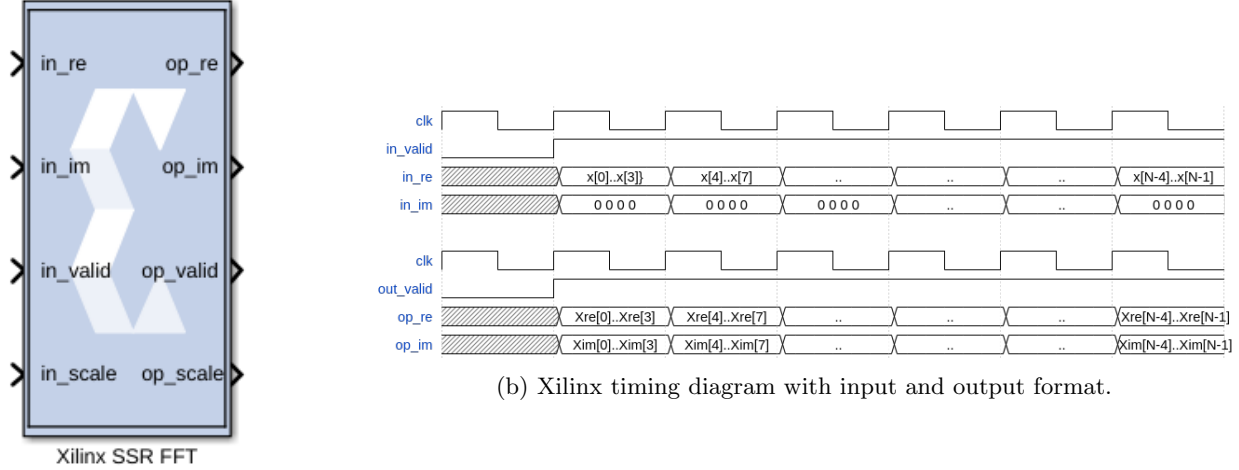
For 16-bit data, the CASPER core concatenates its real and imaginary components into a single 32-bit vector that is output by the core. Additional logic is implemented after each output lane to separate these components for capture.

While the same *sync* generation logic as depicted in Figure 5.19 is used to synchronise the CASPER core, the timing diagram in Figure 5.20b shows that a *sync* pulse is expected one clock cycle before valid data. This requires the addition of a single clock cycle delay on the input of the address counter input for correct alignment. The *sync_out* pulse is also output one clock cycle before valid output data, and this pulse has been used to trigger the output snapshots with no data line delays required and is written to a *to workspace* block for data alignment in simulation.

This core has been configured to use default implementation options, as well as *infinite rounding* and a *saturation* overflow scheme. The final design for this core can be seen in the CASPER designs displayed in Appendix A: A.5 through A.8.

Xilinx

The Xilinx SSR FFT is a purely complex-to-complex FFT, that can only process a single real-valued input if either of its real or imaginary inputs are set to zero. Figure 5.21a shows the Xilinx block configured for the same wideband factor as the other two cores; however, this block accepts a single vector input and the number of input ports does not change. Figure 5.21b portrays the timing diagram and data formats for a single real-valued signal, x/n , on the core's real input.



(a) Xilinx FFT block.

Figure 5.21: Xilinx FFT block and timing diagram. This core can only process a single real-valued input if either of its real or imaginary inputs are set to zero. There are no input or output sync lines available with this core.

This core processes a vector input of L -many consecutive input samples each clock cycle while *in_valid* is high, and outputs real and imaginary vectors of L -many consecutive output samples per clock cycle while *out_valid* is high. This requires additional logic on either side of the core to vectorise BRAM inputs and de-vectorise FFT outputs. BRAM inputs are concatenated to form a single scalar, and then converted to a vector for processing by the core. The output vector is then converted to a scalar, from which each output channel worth of data is sliced. This logic is not shown here, but can be seen in the final Xilinx designs in Appendix A: A.9 and A.10.

The output of this core is the full frequency spectrum, both positive and negative. This requires that N -many output samples be saved to acquire the full spectrum. As the core provides no synchronisation functionality, the rising edge of the *out_valid* output is detected for use as the output snapshot trigger. This requires delays on the output data before capture in the same manner as the ASTRON output, as *out_valid* is aligned with the first valid output data sample. The *out_valid* line is also captured to a *to workspace* block to align valid data in simulation.

The rounding and overflow schemes used by this core are undocumented, and it is not possible to configure specific implementation details.

The result of the above is the implementation of three designs integrating each core into the base design. To build each design for both hardware platforms, copies of these designs have been made for each board with the relevant blocks as shown in Figure 5.16 added. As the Xilinx SSR FFT is incompatible with the SKARAB's Virtex 7 FPGA, this core's design has not been compiled for use on the SKARAB.

Designs for each FFT core integrated into the base test design for both hardware platforms and wideband factors can be found in Appendix A. These designs have been configured for an FFT length of $N = 2^{10}$, with modified copies having been created for FFT lengths $N = 2^{14}$ and $N = 2^{16}$ where BRAM and snapshot depths have been adjusted accordingly and the FFT core configured to these lengths.

The captions of the figures in Appendix A depict the file naming convention used to name each design. This convention is important, as the test system architectures extract information about the test design (such as FFT core type and FFT length) from this filename. The system architectures that have been implemented for each test platform are detailed below.

5.3.3 System Architectures

The system architectures for each test platform have been implemented as a directory structure containing test vector input files, test platform specific design files, an outputs directory and a control script that automates testing. For *hardware testing*, this structure exists on each of the servers used to interface with the hardware platforms. For *simulation testing*, this structure exists within the main system development directory on the machine used both for development and simulation. The directory trees depicted in Figure 5.22 below show this structure for each test platform.

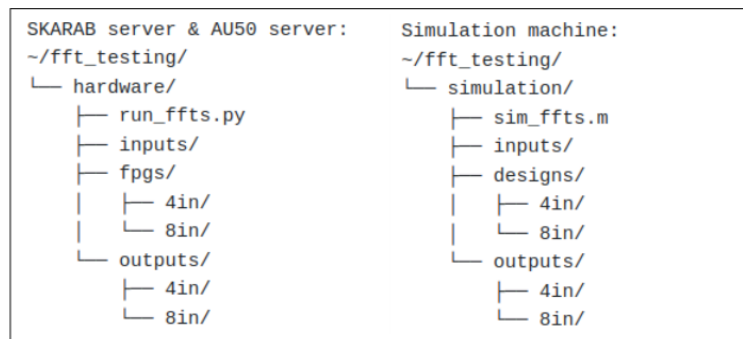


Figure 5.22: Directory structure of system architectures for both test platforms.

The function of the two control scripts across test platforms, *run_ffts.py* and *sim_ffts.m*, is the same. For each test vector input file, the scripts load the test vector values, iterate through and load each FFT core test design using the relevant test platform, run the design for the current test vector and write the FFT output to file. Designs are stored according to wideband factor, with each script determining which wideband factor is being tested based on the design file path it has been given. As such, hardware and simulation tests must be run separately for each wideband factor.

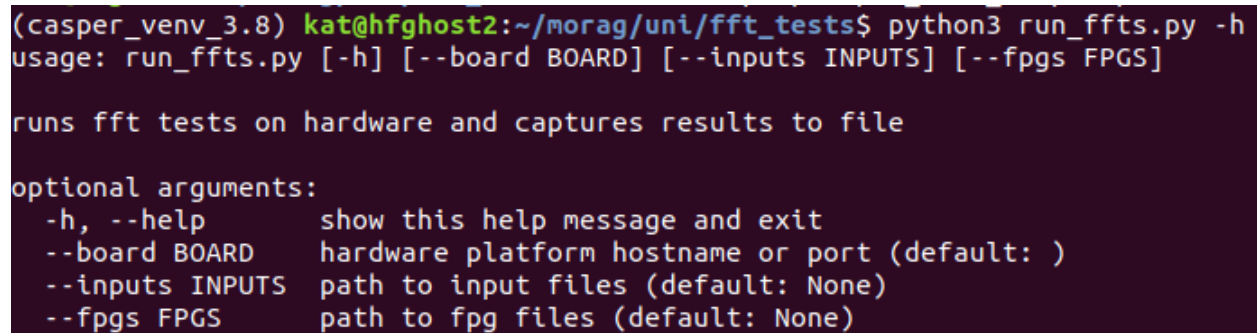
While functionally the same, each test platform's script performs this task in slightly different ways. The sections below outline how each is implemented, providing flowcharts to illustrate the test script's flow.

Hardware System Architecture

The control script that performs FFT testing on hardware has been written using Python with the *casperfpga* Python library to facilitate communications with each hardware platform. The *casperfpga* versions installed on the servers used in communicating with the SKARAB and AU50 use Python 2.7 and Python 3.8 respectively. As such, minor adjustments were made between the script written using the older Python version and the newer. These differences do not affect the overall function of the script, and are not described further.

Various standard Python libraries (such as *numpy*, *struct*, *os* and others) are used in this script, as well as the CASPER-specific *casperfpga* communication library. All required modules have been installed in the Python 3.8 virtual environment created on the AU50 server, and in the system version of Python 2.7 on the SKARAB server.

The *run_ffts.py* script for each hardware platform accepts command-line arguments that specify the board ID through which the hardware platform can be connected to, a path to the input test vector files and a path to fpg files to be tested. This usage is shown below in Figure 5.23.



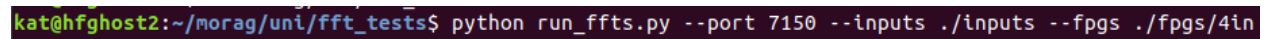
```
(casper_venv_3.8) kat@hfghost2:~/morag/uni/fft_tests$ python3 run_ffts.py -h
usage: run_ffts.py [-h] [--board BOARD] [--inputs INPUTS] [--fpgs FPGS]

runs fft tests on hardware and captures results to file

optional arguments:
  -h, --help            show this help message and exit
  --board BOARD         hardware platform hostname or port (default: )
  --inputs INPUTS       path to input files (default: None)
  --fpgs FPGS           path to fpg files (default: None)
```

Figure 5.23: Usage of the *run_ffts.py* control script.

Figure 5.24 below provides an example of using this script to test all fpg files with a wideband factor of $L = 4$. To test for $L = 8$, the path passed to *-fpgs* is replaced with *./fpgs/8in*.



```
kat@hfghost2:~/morag/uni/fft_tests$ python run_ffts.py --port 7150 --inputs ./inputs --fpgs ./fpgs/4in
```

Figure 5.24: An example of running *run_ffts.py* for fpg files with a wideband factor $L = 4$.

The script then tests each fpg design file for each FFT core and FFT length, using both test vectors generated by the TVG system. Using *casperfpga*, the script first connects to the hardware platform. Lists containing all input test vector files and fpg design files are then obtained, and the wideband factor being tested is extracted from the fpg path argument. For each input test vector file, the script then iterates through testing each fpg. The FFT core type and FFT length N is extracted from the fpg file name and the hardware platform is then programmed with the fpg file. The shift schedule to be used by the core is then selected, and the input test vector is read from file. A signal containing the first N -many samples is created and then scaled appropriately to the form of a 16-bit integer than can ultimately be represented as a $\langle 16, 15 \rangle$ value on hardware. The name of the output file for the current test is generated based on hardware platform type, FFT core type, FFT length, test vector input type and the selected shift schedule.

Before the test is performed, the output snapshots are armed to be ready to capture data upon trigger and the signal is written to the input BRAMs. Once input data has been written, the design is reset and enabled, beginning the test. Output data is then read from the snapshots and combined according to each core's output format (as shown in Figures 5.18, 5.20, 5.21) and written to the output file. The flow of this entire process is depicted below in Figure 5.25, with sub-processes representing function calls within the script.

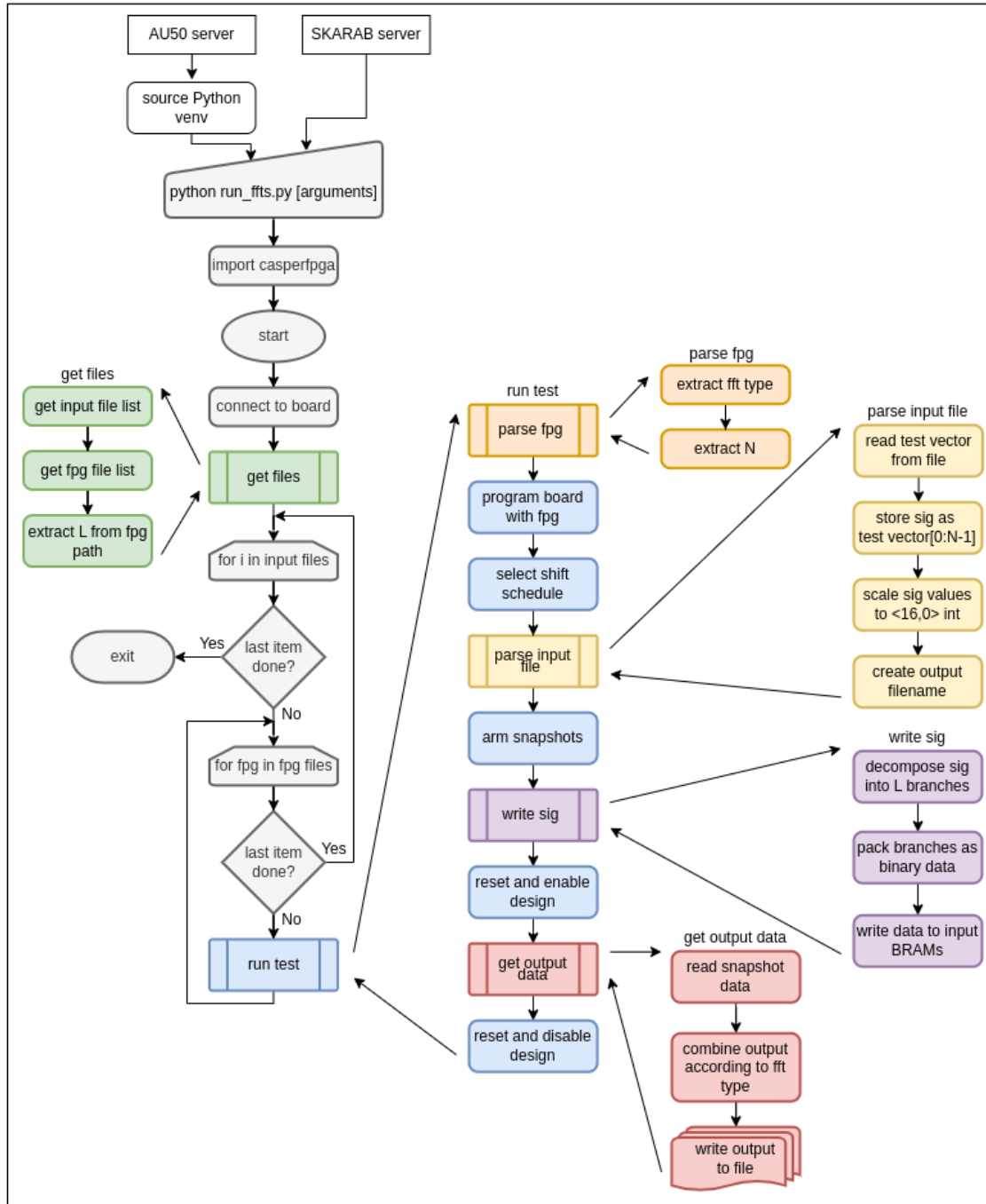


Figure 5.25: Flowchart depicting operation sequence of hardware test script, *run_ffts.py*. The script connects to the hardware platform, obtains a lists of all test vector and fpg files and then performs a test for each test vector across all combinations for all wideband factors, FFT lengths and shifting schedules.

The full scripts developed to run the hardware FFT tests for both the AU50 and the SKARAB are available in this project's GitHub repository [5].

Simulation System Architecture

The MATLAB control script for simulation testing, *sim_ffts.m* is structurally similar to that of the Python hardware testing control script in the general sense. However, this script does not require the use of additional libraries to facilitate communication with the test platform or load test designs. Further, the script is run from the MATLAB GUI environment, with paths to test vector input files, design files and the simulation outputs directory set as string variables at the beginning of the script.

As in hardware testing, this script tests each FFT core's design file for each FFT length and wideband factor. In this case, the design file is the Simulink slx design. Lists of input files and slx designs are created based on the paths set at the beginning of the script, with wideband factor to be tested determined by the slx design path. To simulate each slx file for the current input test vector, the slx model system is first loaded into memory without being opened in the Simulink GUI. The slx filename is then parsed to determine FFT core type, the hardware platform type being simulated and the FFT length N . A shift schedule is selected and the input test vector file is read from file. A signal of length N is created from this test vector and the output filename is generated based on hardware platform type, FFT core type, FFT length, test vector input and shifting schedule.

The input signal is then decomposed into L -many branches, scaled and cast as an integer for $\langle 16, 15 \rangle$ representation within the slx design. These scaled branches are stored as variables that are loaded as the initial vectors of each input BRAM - the simulation analogue to writing values to BRAM on hardware. With the test input loaded, the design is simulated. Once complete, the relevant *to workspace* outputs of the simulation are used to find the output *sync/valid* index to which output data is aligned - similar to triggering the hardware snapshot using the output *sync/valid* line. Valid output data is then stored and combined according to the output format of the FFT core under test and written to the output file.

Simulation testing also measures the time taken to simulate each design, and monitors each core's *overflow* output. Both simulation time and overflow status are written to the simulation output with the output spectrum data. Figure 5.26 below shows the simulation test process, with sub-processes once again representing function calls.

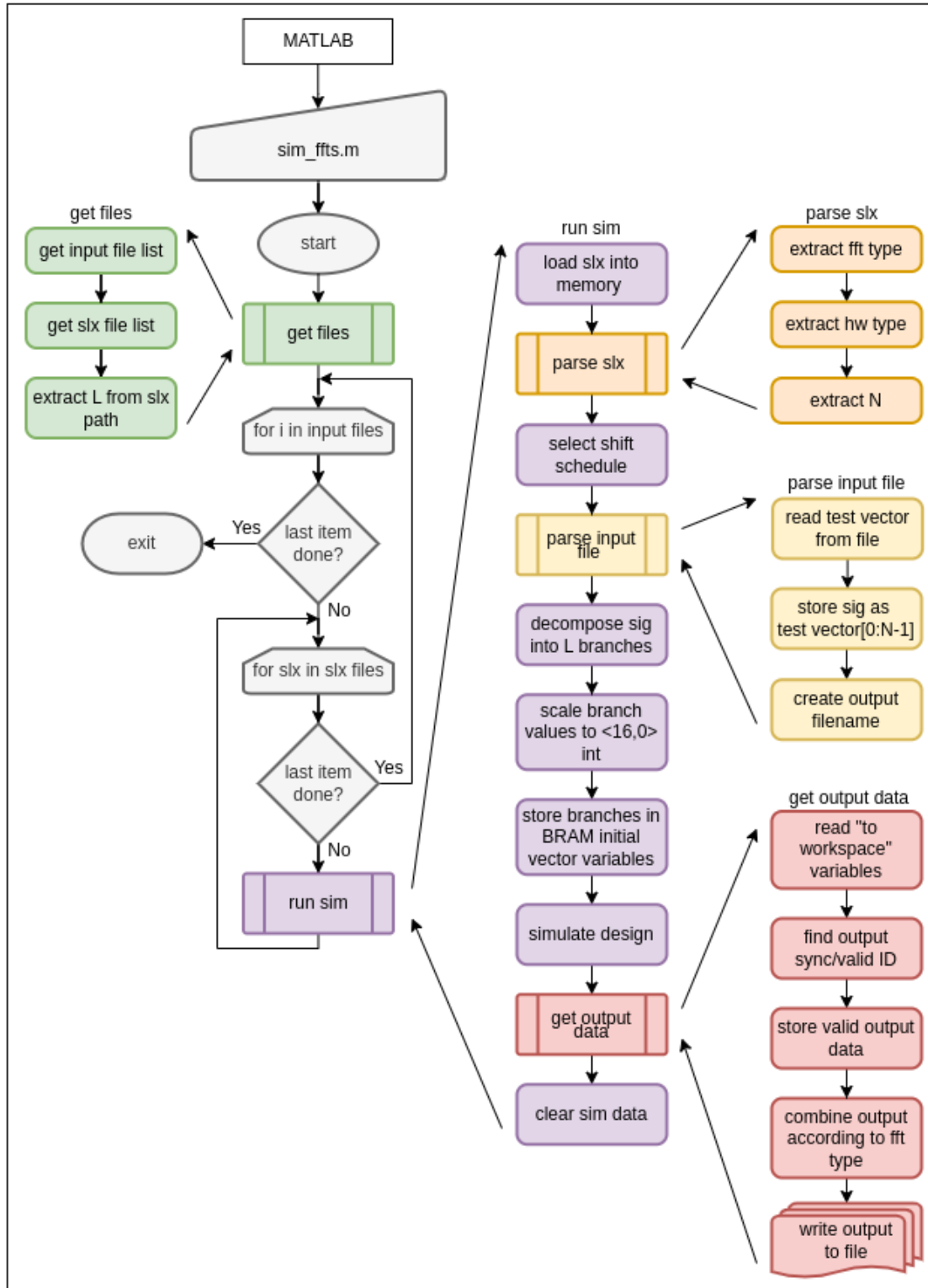


Figure 5.26: Flowchart depicting operation sequence of simulation test script, *sim_ffts.m*. Lists of input files and slx designs are created and simulations are performed for all wideband factors, FFT lengths and shifting schedules.

The research project GitHub repository [5] contains the MATLAB script developed to test each core in simulation.

With the implementation of the TVG, FFT core test designs and test platform system architectures completed using the Systems Engineering design approach, high level requirements *HLR 1* and *HLR 2* have been satisfied. The implemented TVG generates appropriate test data with which to test each FFT, satisfying the high level requirement as defined by *HLR 1*. The test vectors are also generated according to both radio astronomy specific and general test cases, satisfying derived low level requirements *LLR 1* to *LLR 5*.

The FFT core test designs and system architectures that have been developed provide means of data injection and capture, satisfying *HLR 2*. Low level requirements *LLR 6* to *LLR 15* have also been satisfied, as suitable test platforms have been selected, designs capable of testing each core have been developed, standard methods of communication, data injection and capture have been implemented for each test platform and the test system is capable of representing and processing data with a high dynamic range.

The following chapter describes the implementation of an analysis system that satisfies high level requirement *HLR 3* and its derived low level requirements *LLR 16* to *LLR 20*.

Chapter 6

Analysis System: Design, Integration and Testing

“I don’t like to ship a system that’s completely working and running perfectly...” - dan werthimer [61]

Chapter 5 described the design and implementation of the test system with which each FFT core of interest has been tested. This chapter describes the implementation of the system used in the analysis of the outputs of the test system, as well as the integration of both test and analysis systems to form the complete Test and Analysis System. The final section of this chapter details the formal testing that was conducted to generate outputs passed to the analysis system. The results of the analysis system are discussed in detail in Chapter 7.

6.1 Analysis System Development

Low-level requirements derived from the requirements analysis described in Chapter 4 state that the analysis system component of the Test and Analysis system must contain:

- A reference FFT for numerical comparison
- Methods of numerical analysis and an environment in which to perform this analysis
- Analysis of resource utilisation and timing reports for final hardware test designs

Figure 6.1 below depicts a mid-level diagram of the implemented analysis system satisfying these requirements. The inputs into the analysis system are the outputs of the test system. These inputs include the spectral outputs of each FFT core tested across test platforms, as well as the timing and resource utilisation reports generated from the Vivado project of each final hardware test compile.

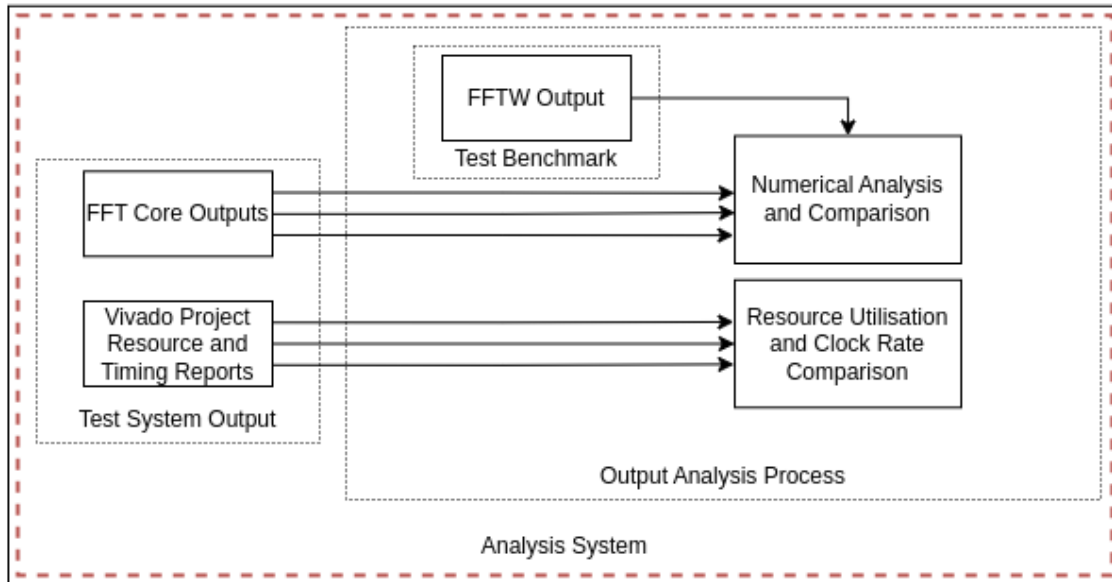


Figure 6.1: Mid-level diagram of Analysis System. This system analyses the spectral outputs of each FFT core under test as well as the timing and resource utilisation reports generated from the Vivado project of each final hardware test compile.

The following sections describe the implementation of each stage of the analysis system in more detail, beginning with the development of the reference FFTW system. The output analysis process follows after, describing both numerical analysis and the analysis of resource and timing reports.

6.1.1 FFTW Test Benchmark

The initial low-level requirement for the analysis system is a reference FFT against which the performance of the FFTs under test can be numerically compared. The double-precision floating point form of the FFTW C-library has been selected as this benchmark. As with the FFT cores under test, the FFTW requires a framework allowing for data input, processing and capture. This is shown below in the form of a directory tree, similar in nature to those created for both hardware and simulation tests described in Chapter 5.

```

FFTW benchmark system:
~/fft_testing/
├─ fftw/
│   ├── run_r2c_fftw.py
│   ├── calc_r2c_fftw.exe
│   ├── calc_r2c_fftw.c
│   ├── inputs/
│   └─ outputs/

```

Figure 6.2: Directory structure of FFTW system.

As shown above, this system contains directories for input and output files, an FFTW executable developed using C and a Python control script. This control script iterates through each input file and FFT length and calls the FFTW executable for each. The usage of this main Python script is shown below.

```
usage: run_r2c_fftw.py [-h] [--inputs INPUTS]

runs the r2c fftw executable for all given inputs and all fft lengths

optional arguments:
  -h, --help            show this help message and exit
  --inputs INPUTS       path to input files (default: None)
```

Figure 6.3: Usage of the *run_r2c_fftw.py* control script.

The FFTW C-library itself offers both complex-to-complex and real-to-complex FFT computations. For these tests, the real-to-complex version has been selected, so as to be in line with the operation of both ASTRON and CASPER FFTs. As with the ASTRON and CASPER cores, for an N -length FFT, the real-to-complex FFTW implementation outputs $N/2$ positive spectral samples and discards the mirrored negative components.

For each input test vector file and tested FFT length, the Python script generates the FFTW output filename and calls the FFTW executable, passing current input and output filenames and the FFT length to the program. Once the process is handed over to the FFTW executable, it parses these input arguments as necessary and creates and allocates memory to the real FFTW input (of double-precision floating point type) and complex FFTW output arrays. A one-dimensional, real-to-complex FFTW plan is then created, using the *FFTW_ESTIMATE* flag to specify a heuristic approach that decides on a plan quickly while not necessarily using the most efficient method of execution. With the plan created, the input test vector file is opened and N -many samples read and stored within the FFTW input address space. The input file is closed and the FFTW plan executed. Upon completion of the plan, the FFTW output is processed and written to file. The plan is then destroyed and previously allocated memory freed before the FFTW executable exits and passes control back to the Python script for the next iteration. This process is shown in Figure 6.4 below.

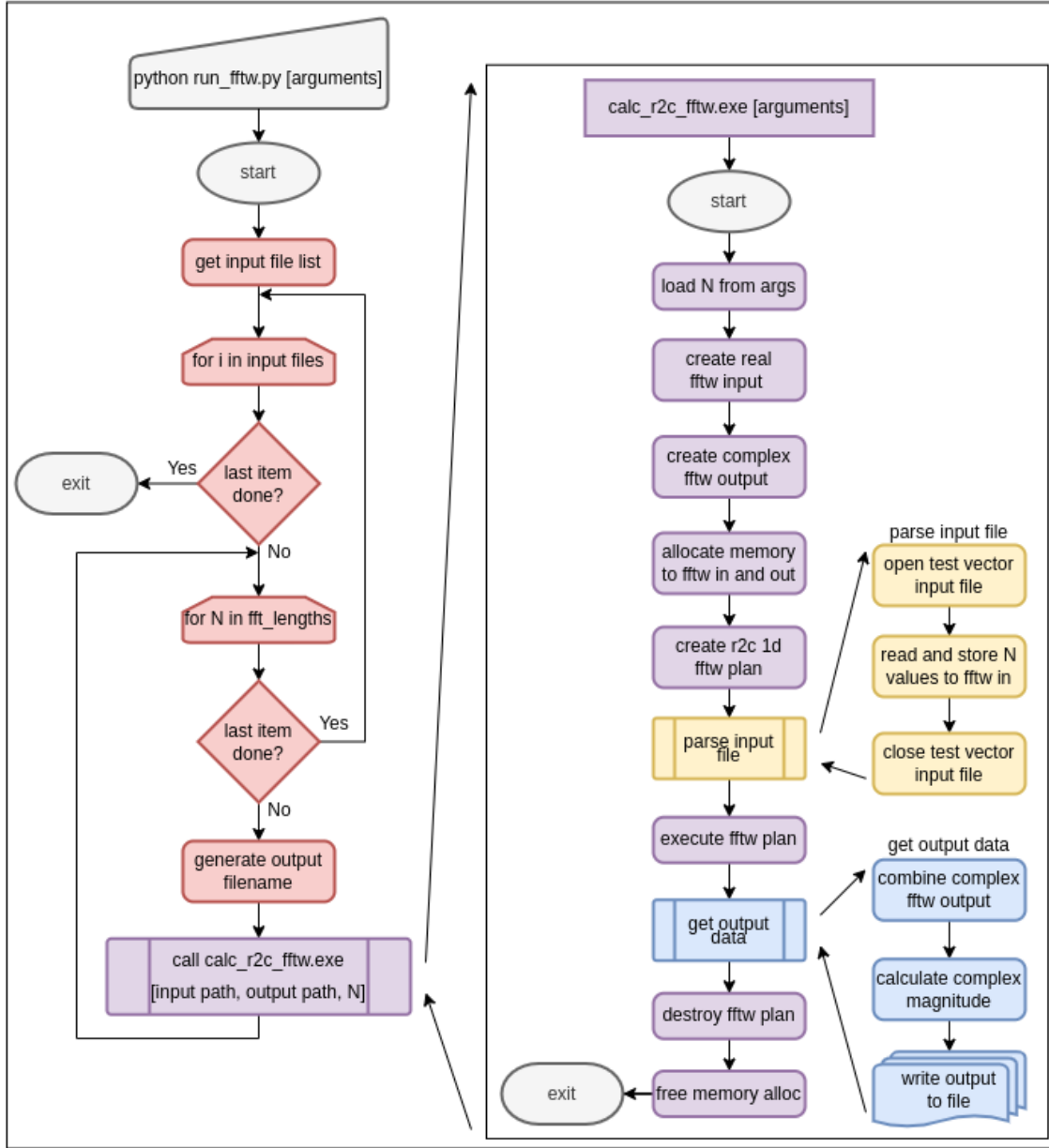


Figure 6.4: Flowchart depicting operation of both *run_r2c_fftw.py* and *calc_r2c_fftw.exe*. For each input test vector file and tested FFT length, the Python script generates the FFTW output filename and calls the FFTW executable for each test combination. The FFTW executable then allocates memory, creates a real-to-complex FFTW plan and then computes the FFT. The spectral output is then written to file and the FFTW plan destroyed.

The left-hand side of the above flowchart depicts the operation of the Python control script, while the right-hand side illustrates the processes and sub-processes of the FFTW C-program. The final outputs of this system are files containing the FFTW spectral outputs for each tested FFT length, providing a benchmark against which each FFT core can be compared through the output analysis process. The implementation of this numerical analysis process is described in the following section.

6.1.2 Output Analysis Process

The output analysis process consists of two distinct parts. The first is the numerical analysis of each FFT core under test's spectral output against that of the FFTW output, for both simulation and hardware testing. The second is the analysis and comparison of the percentage of hardware resources utilised by each FFT core and achievable clock rates. Both of these processes are explained below.

Numerical Analysis and Comparison

A Python script, *analysis.py*, has been developed to perform the numerical analysis computations as per the requirements specified in Section 4.2.3 of Chapter 4. The usage of this script is shown below in Figure 6.5.

```
usage: analysis.py [-h] [--inputs INPUTS] [--sim] [--hw]

does the thing

optional arguments:
  -h, --help            show this help message and exit
  --inputs INPUTS       path to input files (default: None)
  --sim                 analyses simulation outputs if specified (default: False)
  --hw                  analyses hardware outputs if specified (default: False)
```

Figure 6.5: Usage of the *analysis.py* numerical analysis script.

For each test vector input file, this script iterates through each FFT test output file for all wideband factors and FFT lengths and computes all numerical performance metrics defined in Section 4.2.3 against the equivalent FFTW output. Depending on the arguments passed to the script, analysis is performed on either or both simulation and hardware outputs.

The analysis script creates a separate results directory for each test vector file, with separate sub-directories created for the FFTW results files and each wideband factors. Each wideband factor sub-directory contains separate directories for simulation and hardware results. Figure 6.6 below illustrates the directory structure generated by the analysis script.

```
results/
├── tvg0_{Fs}mpps_{f}mhz_-10dBsnr/
├── tvg1_{Fs}mpps_{f}mhz_92dBsnr/
│   ├── fftw/
│   │   ├── normalised_psd/
│   │   └── fftw_results.csv
│   ├── 4in/
│   │   ├── simulation/
│   │   │   ├── normalised_psd/
│   │   │   └── simulation_results.csv
│   │   └── hardware/
│   │       ├── normalised_psd/
│   │       └── hardware_results.csv
│   └── 8in/
│       ├── simulation/
│       │   ├── normalised_psd/
│       │   └── simulation_results.csv
│       └── hardware/
│           ├── normalised_psd/
│           └── hardware_results.csv
```

Figure 6.6: Structure of results directory created by the *analysis.py* script.

For clarity, an example template of a simulations results file generated by the analysis system is shown below in Figure 6.7. Details on the functionality of the analysis system is provided after.

Input Signal: tvG0_Fs)mmps_(f)mhz_-10dBsnr										
FFT Length = 1024										
AU50										
FFT Core	Shifting Schedule	Average Power	Scaled Average Power	SAP dB	MSE	SFDR (dB)	SNR	Sim Time	Latency	Overflow?
astron										
casper										
xilinx										
SKARAB										
FFT Core	Shifting Schedule	Average Power	Scaled Average Power	SAP dB	MSE	SFDR (dB)	SNR	Sim Time	Latency	Overflow?
astron										
casper										
FFT Length = 16384										
AU50										
FFT Core	Shifting Schedule	Average Power	Scaled Average Power	SAP dB	MSE	SFDR (dB)	SNR	Sim Time	Latency	Overflow?
astron										
casper										
xilinx										
SKARAB										
FFT Core	Shifting Schedule	Average Power	Scaled Average Power	SAP dB	MSE	SFDR (dB)	SNR	Sim Time	Latency	Overflow?
astron										
casper										

Figure 6.7: An example template of a simulations results file generated by *analysis.py*.

A flowchart describing the operation of *analysis.py* is shown below in Figure 6.8. The script creates the appropriate results directory based on the test vector filename, and creates the FFTW results file within this directory. An iteration over each wideband factor is then performed, with the current factor stored as a variable called *demux*, referring to the factor by which the input signal has been demultiplexed in time. For each wideband factor, the FFTW output and FFT test outputs are loaded for each tested FFT length, with simulation and/or hardware analysis then performed.

Both simulation and hardware analysis processes compute the total spectral power, the normalised PSD, SFDR and SNR of the FFTW and FFT core outputs. The MSE between the FFT core output and the FFTW output is then calculated using the normalised PSD of each. These values are then written to the relevant results file under the current FFT length, and formatted according to hardware platform type, FFT core type and the shifting schedule used in testing. Simulation tests have also tested for simulation time, FFT core latency and overflow for the given shift schedule, and these values are also written to the simulation results file.

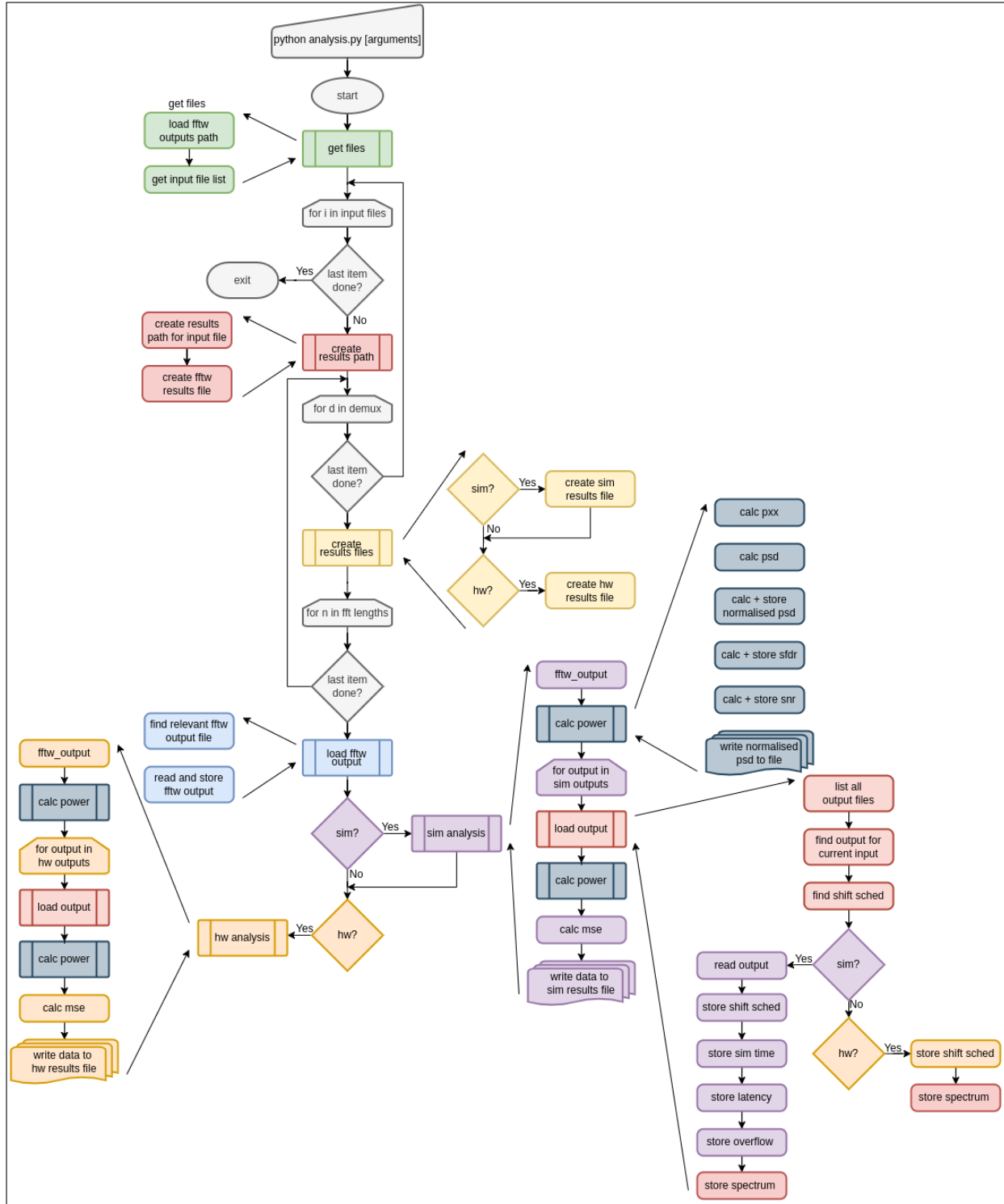


Figure 6.8: Flowchart detailing performance of numerical analysis script, *analysis.py*. For each wideband factor (denoted as *demux*), the FFTW output and FFT test outputs are loaded for each tested FFT length, with simulation and/or hardware analysis then performed. Analysis includes computing total spectral power, the normalised PSD, SFDR and SNR of the FFTW and FFT core outputs. The MSE between the FFT core output and the FFTW output is then calculated using the normalised PSD of each.

The analysis python script has been implemented to perform all calculations required to assess the numerical performance of each FFT core of interest. The following section describes how the resource utilisation and maximum achievable clock rates for each core have been determined.

Resource Utilisation and Clock Rate Comparison

A Xilinx Vivado project is created from the Simulink block design during the compilation process of the CASPER toolflow. This project is then synthesised and implemented for the specified hardware. Various reports providing information on resource usage, timing analysis, power requirements and more can be generated for both synthesis and implementation stages once each stage has finished. The synthesised design report is an estimate of the design's requirements before implementation, while the implemented design report provides information on the floorplan used to create the output bitstream.

Implemented design reports have been generated for each FFT core test design and analysed to extract information regarding resource utilisation and power requirements for each core. Timing reports have been analysed for test designs compiled for a range of FPGA clock rates to determine the clock rates at which each core can be used. Details on these reports are given below.

Resource Utilisation:

An example of an implemented design's utilisation report for a design containing an ASTRON FFT core is shown below in Figure 6.9. A summary of resource usage, shown in green (upper left), provides the total number of resources used by the entire design against total number of resources available. Additionally, the number of logic resources used by every module instantiated throughout the project hierarchy is also given. The ASTRON core's instance within this utilisation report is shown in red (bottom), with this providing access to resource usage data for the core specifically. For these tests, only resource data pertaining to each FFT core has been extracted from the generated reports, and not the total resource usage of each test design. Information regarding power requirements is available for the implemented design too, shown in orange (upper right). A summary of total on-chip power is given, with hierarchical power requirements provided too.

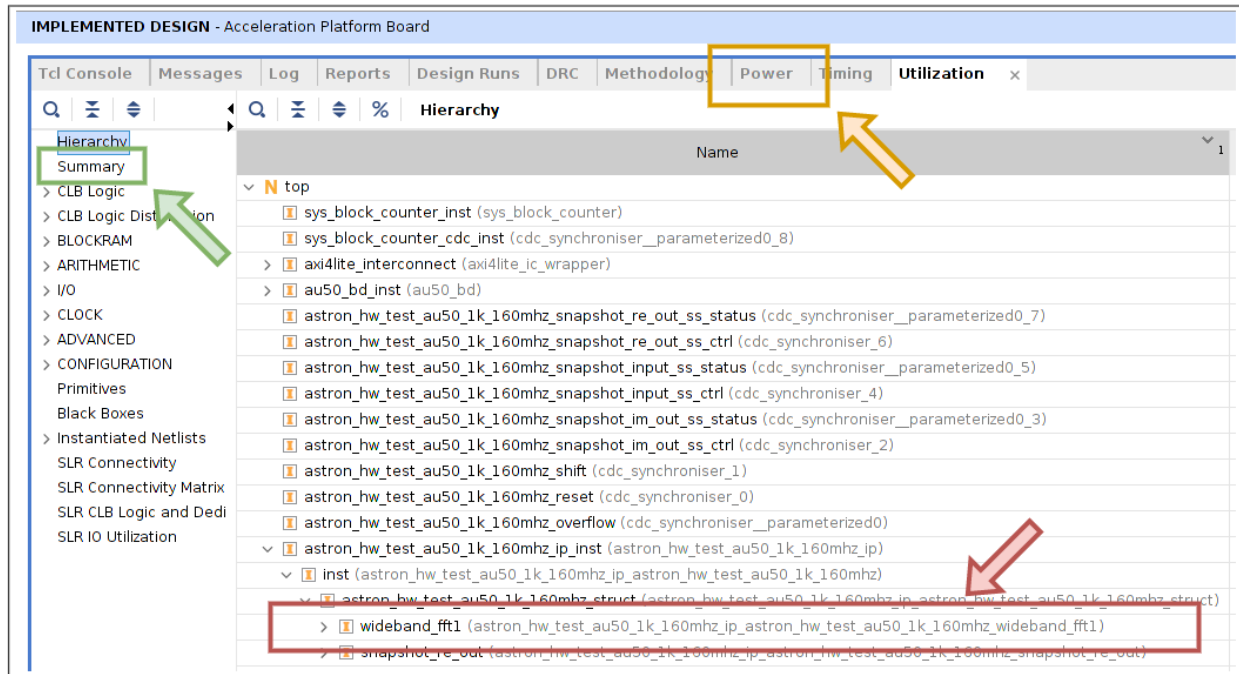


Figure 6.9: Resource utilisation report generated for an implemented Vivado project design.

The hierarchical power requirements of each logic core within the implemented design is shown in Figure 6.10 below. The power requirement for the ASTRON core is shown in red (bottom). The power required for each FFT core has been taken from the power analysis report of each FFT core test design for comparison purposes.

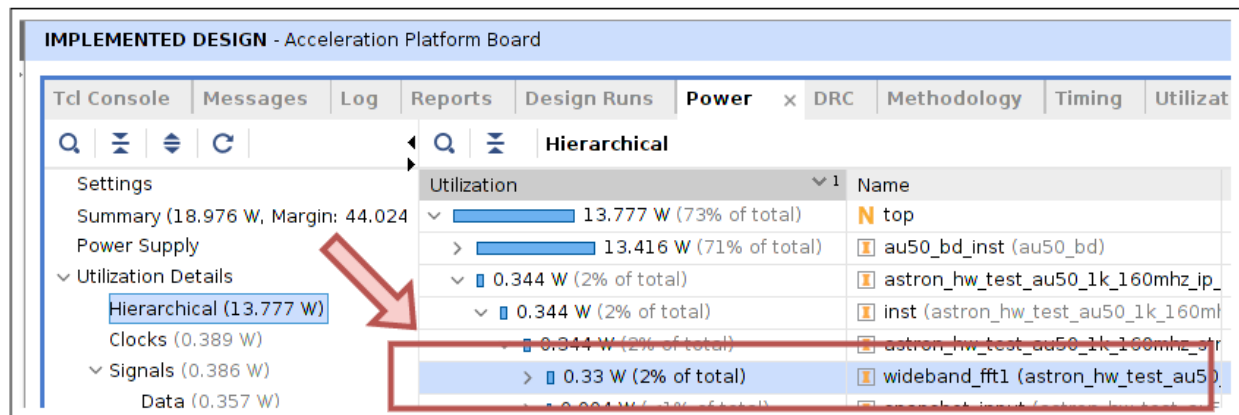


Figure 6.10: Hierarchical power analysis report generated for an implemented Vivado project design.

Beyond resource utilisation and power information, the clock rates that the FFT cores of interest can be successfully compiled for have also been assessed using timing reports generated for each test design. This process is described below.

Clock Rates:

Once the final Simulink test design had been implemented and functionally verified, each FFT core test design was compiled for a range of FPGA clock rates for both hardware platforms. This task was performed to determine the clock rates achievable by each core, with the timing report of each compile analysed for timing violations. An example of a design timing summary for the same ASTRON test design as used above is given below in Figure 6.11.

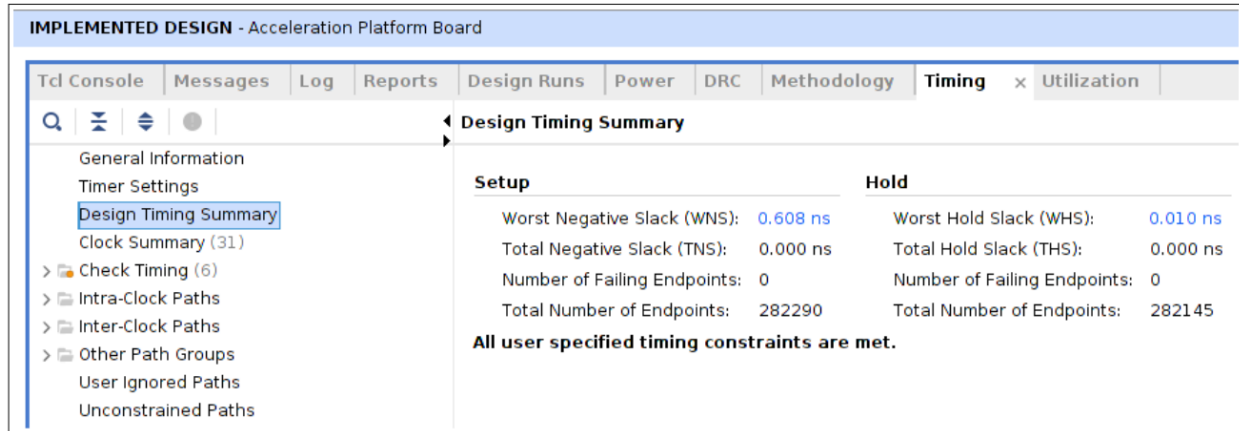


Figure 6.11: Design timing summary generated for an implemented Vivado project design.

An FPGA design compile with a negative *worst negative slack* (WNS) value has failed to meet the design's timing constraints, with this negative value indicating that the longest path delay between a pair of registers within the design is greater than the FPGA's clock period. Each core was compiled for an FPGA clock rate range of [200MHz : 600MHz] at increments of 50MHz and analysed for timing failures.

In the event of timing violations, an achievable FPGA clock rate estimate has been calculated using the period of the failed clock value one 50MHz increment above the nearest passing clock value and the failed WNS value. This estimate is calculated as follows:

$$\text{Estimated Clock Rate} = \frac{1}{(\text{Failed Clock Rate})^{-1} - WNS} \quad (6.1)$$

While the above equation provides a means of approximating an achievable clock rate, this is merely a theoretical value. In practice, the synthesis and implementation processes for the Vivado project may result in a compile that does not meet this timing constraint. Alternatively, it may be possible to compile a design that meets timing for a clock rate between this estimated value and the failed clock rate, depending on how the compilation process optimises and routes the design.

The implemented analysis system as described above satisfies the high level requirement defined by *HLR 3* and its derived low level requirements *LLR 16* to *LLR 20*. A system providing a reference FFT has been developed, numerical means of functional verification and comparison as described in 4.2.3 have been implemented using an appropriate computing environment and resource and timing reports for each FPGA compile are available. The following section describes the integration of the test system developed in Chapter 5 and the above analysis system to form the complete Test and Analysis System.

6.2 System Integration

A high level diagram of the fully integrated Test and Analysis System is shown below in Figure 6.12. Blocks outlined in blue indicate components of the test system, while figures outlined in red are analysis system components. As illustrated by Figure 6.1, the inputs to the analysis system are not only the spectral outputs of each FFT core test, but also include the resource and timing reports for each FFT core's hardware compile.

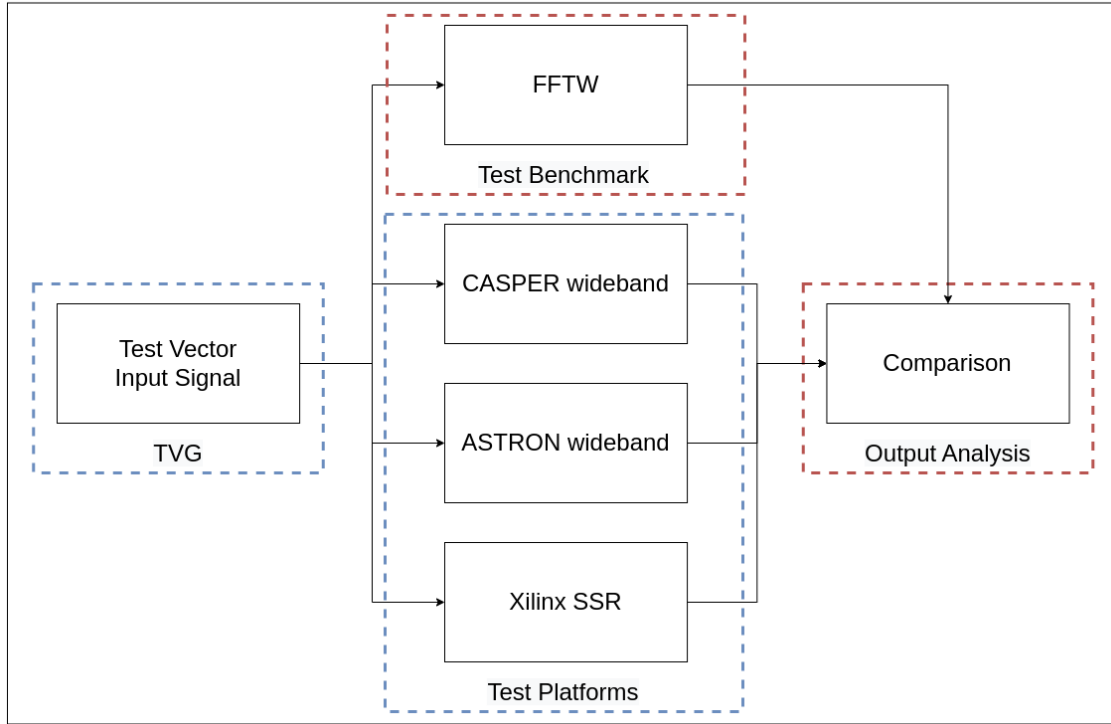


Figure 6.12: Diagram depicting integrated Test and Analysis System. The test system is demarcated in blue and the analysis system in red. Inputs to the analysis system are not only the spectral outputs of the FFT cores under test, but also the resource and timing reports for each hardware compile.

The directory tree structure for hardware and simulation system architectures are shown in Figure 5.22 in Section 5.3. Part of the integration process involved the deployment of test vector input files and appropriate fpg design files to the hardware system architecture on each hardware platform's server for testing. Section 6.3 further down describes the formal test process followed using these files. Once formal testing had been completed, the outputs of the hardware tests were copied from their servers to the machine that hosts the TVG, the simulation system architecture and the analysis system. These files are contained in a directory titled *fft_testing*, which holds all files and systems that form the Test and Analysis System. The tree structure for this directory and its sub-directories is given below in Figure 6.13.

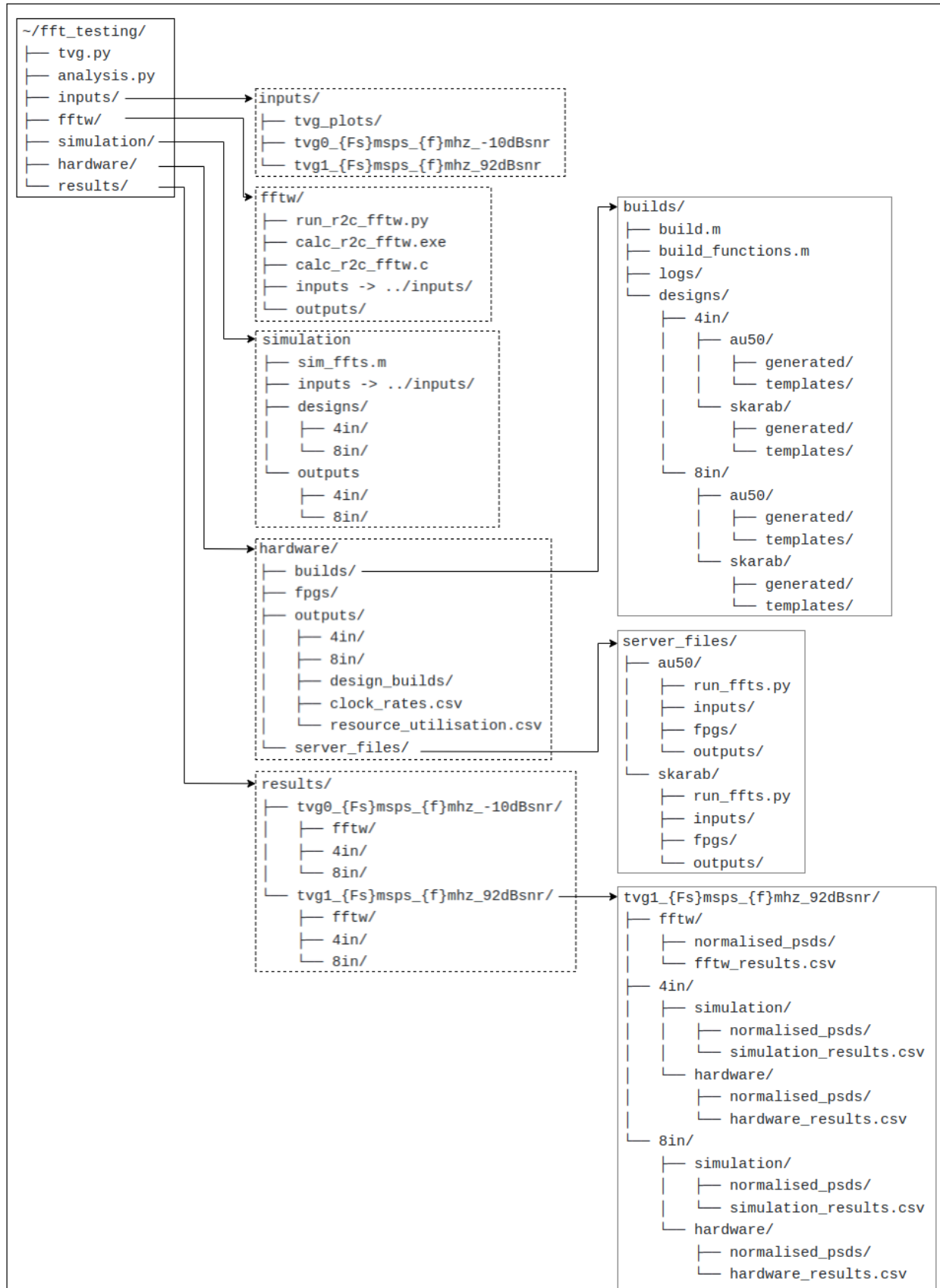


Figure 6.13: Directory tree structure of integrated system.

When run, the *tvf.py* script within the *fft_testing* directory writes its generated test vectors and their corresponding plots to the *inputs* sub-directory. The *analysis.py* script writes all its numerical analyses results to the *results* sub-directory, with an example of the inner structure of *results* given for the *tvf1* test vector input.

The *fft* sub-directory contains all files as depicted in Figure 6.2, with a symbolic link to the *inputs* folder one level above. The structure of the *simulation* sub-directory is as shown in Figure 5.22, again with a symbolic link to the TVG’s *inputs* folder. As the structure of the *fft_testing* system was created prior to the conduction of formal testing, placeholder *outputs* directories were created for both simulation and hardware tests.

The *builds* folder within the *hardware* sub-directory contains a hardware compilation system developed to automate the build process for each design. This build system generates Simulink designs for FPGA clock rates specified within *build.m* from the designs in each *templates* folder. Each generated file is then compiled for its hardware platform at the specified clock rate, with the resulting Vivado project and output fpg files saved to the *generated* folder. The template designs are the same designs that are stored in *simulation/designs*. This system was used when performing clock rate tests on each core’s test design, and was used again to generate the final hardware fpg files once a commonly achievable clock rate was found. The hardware fpg files used in formal testing are saved in the *fpgs* folder of the *hardware* sub-directory, with their corresponding built Vivado projects moved to *hardware/outputs/design_builds*. The outputs of each FFT core’s hardware test were copied to the *hardware/outputs* directory for both wideband factors, to be processed by the analysis system. Consolidated resource utilisation and clock rate results for each FFT core are also contained within the *hardware/outputs* folder.

This structure has been implemented with the *analysis.py* script located at the top level of the directory tree. This allowed numerical analysis to be performed on all test outputs contained within their relevant sub-directories relative to this script, removing the requirement for the provision of absolute paths.

With the integration of both test and analysis components complete, a final system satisfying all three high level requirements, *HLR 1*, *HLR 2* and *HLR3*, as illustrated in Figure 4.1 has been developed. The following section provides details on the formal testing conducted to generate the outputs on which numerical analysis has been performed using this complete system.

6.3 Formal Testing

The conditions under which formal testing has been conducted are described below. These conditions refer to the conditions under which each FFT core test design was compiled for formal testing, the specifics of the generated input test vectors used and the shifting schedules selected to test for each test vector.

6.3.1 Hardware Test Designs

As previously mentioned in Chapter 5, each FFT core has been tested in its default state. No optimisation options available for each core (if any) were applied, nor were any specific implementation strategies available through Xilinx Vivado selected for each core’s final compilation.

Section 6.1.2 described the process used to test the achievable clock rates for each FFT core. Once the implemented test system had been verified, these clock rate tests were performed before any others, as it was desirable to use the highest common clock rate across cores for all other tests. Chapter 7 discusses the results of these tests in more detail; however, the ASTRON FFT core failed to meet timing for the lowest tested clock rate of 200MHz. Based on this timing performance, a lower clock rate for the final hardware compiles was selected. As such, all FFT core test designs were compiled for an FPGA clock rate of 160MHz.

With the final compiles completed for the above conditions, the test vectors used in the formal tests were generated using the TVG. The sample rate and frequencies used to generate these test vectors and the shifting schedules used to test each are detailed below.

6.3.2 Input Test Vector Data

The TVG was implemented to generate test vectors for two separate frequencies. The second frequency output of the TVG could be used in situations where multiple signals to each FFT core are being tested, but for these tests, only one frequency output has been used.

Equations 4.11 and 4.12 define the requirements of the signal used to test each FFT core as:

- The sinusoidal component must have a frequency $f = n\Delta f$, where $n \in \mathbb{Z}$ and Δf is the frequency resolution defined by Equation 4.10
- The generated signal must be well sampled ((i.e $0 < f < \frac{F_s}{2}$, where F_s is the sample rate)

To satisfy both these requirements, the below sample rate and sinusoidal frequency have been selected for both test vector cases:

$$\begin{aligned} F_s &= 800 \text{MSPS} \\ f &= 50 \text{MHz} \end{aligned} \tag{6.2}$$

A frequency of $f = 50 \text{MHz}$ satisfies the condition that the test sinusoidal frequency is an integer multiple of the frequency resolutions for all three tested FFT lengths. This frequency value is also $\frac{1}{8}^{th}$ of the bandwidth specified by the Nyquist rate, and can be considered well sampled.

The test vector outputs for each test case are shown below and tested shifting schedules are given. The shifting schedules used differ for each test vector type, and an explanation as to why this is the case is provided.

TVG - Case 0

Test Vector 0 is the vector generated according to specific radio astronomy requirements. WGN has been generated for an RMS value of $\frac{1}{8}$ and a sinusoidal waveform has been generated with an amplitude derived from an SNR of -10dB. Using Equation 4.6, the sinusoidal amplitude has been calculated to be 0.056. Figure 6.14 below shows both WGN and sinusoidal components of the test vector, as well as the final test vector as a combination of both of these.

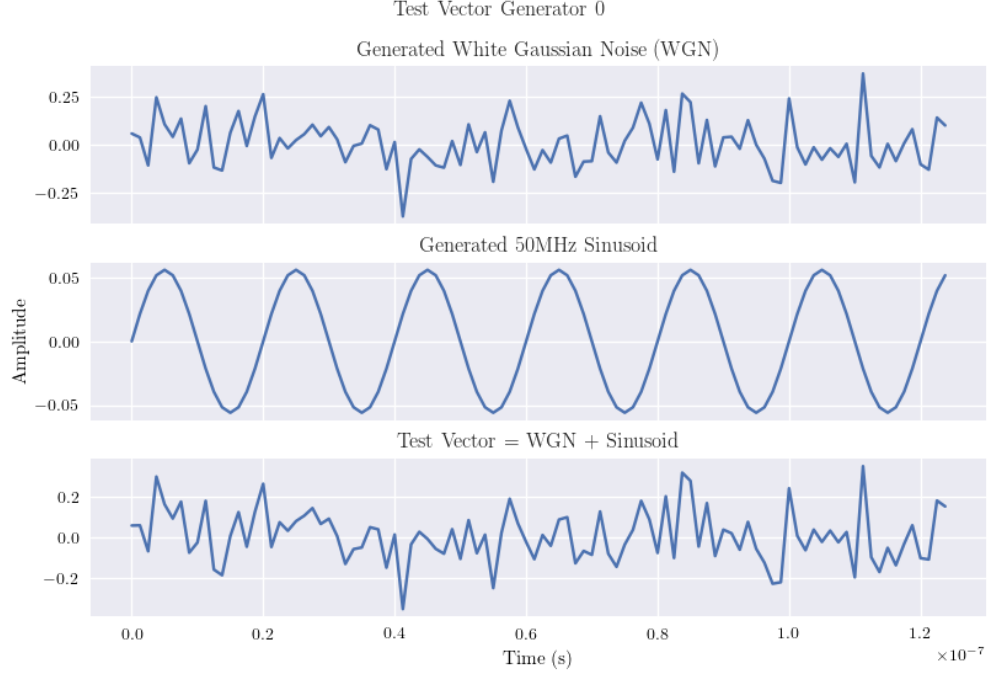


Figure 6.14: Plot of Test Vector Generator 0 components and final test vector, for $f = 50\text{MHz}$. This test vector has been generated according to radio astronomy specific requirements. WGN has been generated for an RMS value of $\frac{1}{8}$ and a sinusoidal waveform has been generated with an amplitude derived from an SNR of -10dB (resulting in a sinusoidal amplitude of 0.056).

The FFT core test designs have been tested using this test vector for a number of different shifting schedules. Shifting is necessary for any signal with a periodic component to prevent overflow from occurring in the frequency bin around which the periodic waveform's power will be concentrated. However, depending on the strength of the periodic signal, it is not always necessary to shift at every stage. In radio astronomy, it is preferable to use a shifting schedule that shifts just enough to prevent overflow while preserving as much of the data's precision as possible.

All FFT cores, for all FFT lengths and wideband factors, have been tested in simulation for the following shifting schedules:

- None - no shifting performed at any stage of the FFT
- Alternating - shifting performed at every alternate stage of the FFT, with the first shift occurring at the first stage
- All - shifting performed for all FFT stages

The outputs of each FFT core for the above shifting schedules were then examined for overflow and an overflow matrix created. Only the numerical analyses of simulation outputs that did not overflow have been presented, as overflow can cause a complete distortion of output data. Hardware tests were then performed using the same shifting schedules for comparison against their simulation counterparts.

With these basic shifting schedule tests completed, an attempt to find the ideal shifting schedule for each FFT core for an FFT length of 2^{16} was made. The results of this process are discussed in Chapter 7.

TVG - Case 1

Test Vector 1 has been generated according to the general test case requirements defined in Section 4.2.1. A sinusoidal waveform has been generated with an amplitude of 0.9, and a WGN model generated for an RMS value equal to half the precision of the implemented fixed-point system. The implemented system utilises a $\langle 16, 15 \rangle$ representation, resulting in a precision of 2^{-15} . Generating noise according to this parameter resulted in a test vector with an SNR of 92dB. Both sinusoidal and noise components are shown below in Figure 6.15 along with the resulting test vector.

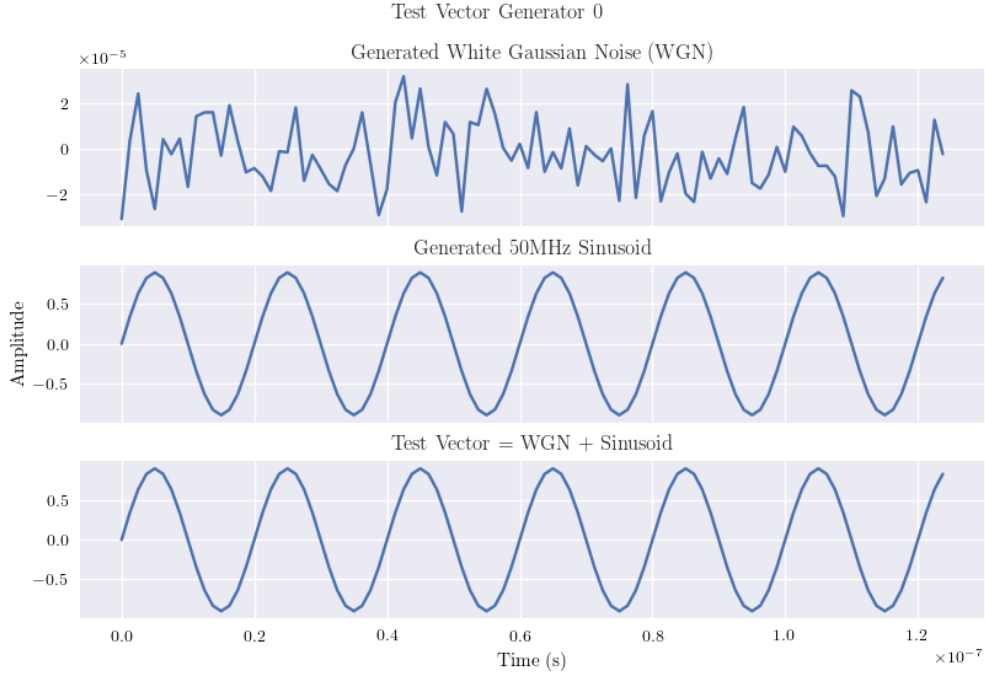


Figure 6.15: Plot of Test Vector Generator 1 components and final test vector, for $f = 50\text{MHz}$. This test vector has been generated according to the general test case requirements. A sinusoidal waveform has been generated with an amplitude of 0.9, and a WGN model generated for an RMS value equal to half the precision of the implemented fixed-point system. Generating noise according to this parameter resulted in a test vector with an SNR of 92dB.

The peak amplitude of the sinusoidal component of this test vector is at $90\%FS$. A periodic signal of this scale will cause overflow unless aggressively shifted, as the majority of the signal's power will be concentrated around a single frequency bin. This test vector has therefore only been tested for a shifting schedule that applies a shift at each stage of the FFT.

With the formal testing concluded and valid spectral outputs using effective shifting schedules gathered, these outputs were processed by the analysis system to generate numerical performance results. The resource utilisation and power requirements for the hardware test compiles were also gathered, and a matrix of maximum achievable clock rates for each core generated. These results are presented and discussed in the following chapter.

Chapter 7

Results and Analysis

“the entire universe is just a series of multiply and accumulates” - myself and robin van wyk

Chapters 5 and 6 described the design, implementation and integration of a Test and Analysis System capable of testing each FFT core of interest according to requirements specified in Chapter 4. The results of the formal testing conducted using this system under the conditions detailed in Section 6.3 are presented in this chapter.

Section 7.1 presents and discusses the results of the numerical analysis performed on each FFT core against the double-precision FFTW equivalent. Section 7.2 presents each core’s resource utilisation and power requirements and the results of the clock rate testing described in Section 6.1.2.

7.1 Numerical Analysis Results

Numerical analysis has been performed on the outputs of each FFT core for both simulation and hardware, for tests using both test vectors derived from the requirements identified in Section 4.2.1. All calculations have been performed using the normalised PSDs of each FFT output, as detailed in Section 4.2.3. The normalised PSDs of non-overflowed outputs for both test cases can be found in Appendices B and C respectively. No differences in numerical performance were found between hardware platforms, and only the PSD plots for simulation and hardware tests on the AU50 are given in these appendices. The results of the numerical analysis are presented in the sections below. The radio astronomy test case is presented first, with the general test case results presented after.

7.1.1 Test Case 0

The first tests performed using the radio astronomy test vector were simulation tests using three shifting schedules for each core, wideband factor and FFT length. This testing was conducted to determine whether overflow occurred for any of these shifting schedules. Only the numerical analysis results of spectral outputs from test cases using shifting schedules that successfully prevented overflow are of interest. For consistency, simulation tests were conducted using both AU50 and SKARAB test designs.

The same tests were then performed in hardware to confirm their fidelity to their simulation equivalents. Shorter simulation tests were then performed on the CASPER and ASTRON 16k FFT test designs for the wideband factor $L = 4$ to determine a more ideal shifting schedule for each. Numerical analyses of these test outputs were then performed for comparison against the results of shifting schedules that prevented overflow, but were not the best case shifting pattern that could be used.

Overflow Test Results

Overflow matrices for both wideband factors, $L = 4$ and $L = 8$, are given below in Tables 7.1 and 7.2. Here, a 1 signifies that an overflow as occurred and a 0 signifies no overflow.

AU50	N=1024			N=16384			N=65536		
	none	alt	all	none	alt	all	none	alt	all
ASTRON	1	0	0	1	0	0	1	1	0
CASPER	1	1	0	1	1	0	1	1	0
Xilinx*	1	0	0	1	1	0	1	1	0

SKARAB	N=1024			N=16384			N=65536		
	none	alt	all	none	alt	all	none	alt	all
ASTRON	1	0	0	1	0	0	1	1	0
CASPER	1	1	0	1	1	0	1	1	0

Table 7.1: Simulation Test: Overflow matrix for wideband factor $L = 4$.

AU50	1024			16384			65536		
	none	alt	all	none	alt	all	none	alt	all
ASTRON	1	0	0	1	0	0	1	1	0
CASPER	1	1	0	1	1	0	1	1	0
Xilinx*	1	1	0	1	1	0	1	1	0

SKARAB	N=1024			N=16384			N=65536		
	none	alt	all	none	alt	all	none	alt	all
ASTRON	1	0	0	1	0	0	1	1	0
CASPER	1	1	0	1	1	0	1	1	0

Table 7.2: Simulation Test: Overflow matrix for wideband factor $L = 8$.

* **NOTE:** The overflow flag for the Xilinx SSR FFT core tested using Vivado 2021.1 and MATLAB R2021a did not work in either simulation or hardware. As such, the overflow status for each test was estimated based on an analysis of MSE and SNR values and an inspection of all output spectra.

For each of the cases above, all cores overflowed when no shifting was applied. An aggressive shifting schedule that applied a shift at each stage prevented overflow in all configurations for all cores. For FFT lengths of 1024 and 16384, an alternating shifting schedule prevented overflow from occurring within the ASTRON core but did not prevent overflow within the CASPER core for any FFT length. This can be attributed to the use of guard bits and increased stage bitwidth in the ASTRON core, which allows for more growth in the data before shifting is required to prevent overflow.

The 1024 length Xilinx FFT for a wideband factor of $L = 4$ did not overflow for an alternating shifting schedule, while its $L = 8$ counterpart did. For all other FFT lengths across both wideband factors, only aggressive shifting prevented overflow within the Xilinx core. The specifics of how the Xilinx core has been implemented are unknown, and it is therefore not possible to determine whether the core performs any data manipulation that may help prevent overflow for smaller FFT lengths and wideband factors. It could also be that the Xilinx core did overflow for an FFT length of 1024 for $L = 4$ (see the note above regarding its non-functional overflow flag), but with minimal impact to its performance.

In the case of the CASPER FFT, the complex multiplication and addition/subtraction of the butterfly occurs before the shift is applied (if specified for that stage). Data bitwidth is increased during this operation within each stage to account for the natural bitgrowth that occurs during multiplication and addition. This bitwidth is then converted back to its original representation (in this case, 16-bits) post-shift before being output to the next stage. Even with the bit growth allowed within each butterfly before requantisation between stages, shifting is often necessary to prevent overflow, especially in cases where a strong periodic signal is present.

It is worth reiterating that the ASTRON FFT is a DIF implementation, where the largest twiddle factors are applied in the first stage and then decrease as stages progress (see Figure 2.5). This is in contrast to the DIT operation of the CASPER FFT, where the smallest twiddle factors are applied first (see Figure 2.3). With the largest twiddle factors applied first, the ASTRON core has an increased chance of overflow in the early FFT stages when compared to CASPER’s DIT implementation. To prevent overflow for signals containing a periodic component with an amplitude $> 50\%FS$, guard bits must be applied to the input data to the ASTRON core and the bitwidth of data at each stage increased.

The ASTRON FFT implementation that was tested applies two guard bits to the MSB of its input data and has a stage bitwidth that has also been increased by two, increasing the data representation for this core from 16-bits to 18-bits. This helps prevent overflow as data is able to grow beyond its original number of bits by a factor of four. As such, this core requires a less aggressive schedule than the others for the first two FFT lengths; however, an alternating shifting schedule did not prevent overflow for the ASTRON core for an FFT length of 65536.

As the length and number of stages of an FFT increases, the number of complex multiplications and additions applied to each data path increases too. This increases the chance of overflow occurring and larger FFTs require a more vigorous shifting method. In radio astronomy, it is ideal to use a shifting schedule that just prevents overflow without losing too much data precision. In such cases, shifting should only begin at the latest possible stage with shifting distributed among stages to preserve as much information as possible. The results of the numerical analysis performed on non-overflowed test outputs are given below.

Results Analysis

The numerical performance results of the reference FFTW software implementation are presented first in Table 7.3 below. The values presented provide a benchmark against which each FFT core’s performance is evaluated.

FFTW	Average Power	Average Power (dB)	SFDR (dB)	SNR (dB)
N=1024	0.017727	-17.514	9.0376	-10.155
N=16384	0.017144	-17.659	18.167	-10.227
N=65536	0.017269	-17.627	24.843	-10.245

Table 7.3: Numerical analysis results of reference FFTW outputs using Test Vector 0.

The tables displayed on the following two pages present the simulation and hardware results of each FFT core for both wideband factors. Tables 7.4 and 7.5 present results for the wideband factor $L = 4$ test case and Tables 7.6 and 7.7 present test results for a wideband factor of $L = 8$. In each case, the *scaled average power* (SAP) has been calculated using Equation 4.16 with the further application of a scaling factor to compensate for the downshifting that has occurred within each core.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017745	-17.509	4.3004E-12	9.037	-10.159
	ASTRON	all	0.018071	-17.43	2.2931E-09	9.0396	-10.227
	CASPER	all	0.017895	-17.473	5.2619E-10	9.0316	-10.192
	Xilinx	alt	0.017727	-17.514	5.7714E-13	9.0375	-10.155
	Xilinx	all	0.017689	-17.523	2.2217E-10	9.0381	-10.158
N=16384	ASTRON	alt	0.017168	-17.653	4.443E-13	18.164	-10.231
	ASTRON	all	0.018746	-17.271	2.9463E-09	18.058	-10.62
	CASPER	all	0.017961	-17.457	7.9862E-10	18.092	-10.428
	Xilinx	all	0.017161	-17.655	2.206E-11	18.179	-10.233
N=65536	ASTRON	all	0.020648	-16.851	2.9719E-09	24.333	-11.075
	CASPER	all	0.018946	-17.225	8.0465E-10	24.527	-10.66
	Xilinx	all	0.017346	-17.608	3.4588E-12	24.724	-10.253
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017745	-17.509	4.3004E-12	9.037	-10.159
	ASTRON	all	0.018071	-17.43	2.2931E-09	9.0396	-10.227
	CASPER	all	0.017895	-17.473	5.2619E-10	9.0316	-10.192
N=16384	ASTRON	alt	0.017168	-17.653	4.443E-13	18.164	-10.231
	ASTRON	all	0.018746	-17.271	2.9463E-09	18.058	-10.62
	CASPER	all	0.017961	-17.457	7.9862E-10	18.092	-10.428
N=65536	ASTRON	all	0.020648	-16.851	2.9719E-09	24.333	-11.075
	CASPER	all	0.018946	-17.225	8.0465E-10	24.527	-10.66

Table 7.4: Numerical analysis results of simulation test outputs for $L = 4$, tested using Test Vector 0.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017742	-17.510	2.6688E-12	9.0370	-10.158
	ASTRON	all	0.018084	-17.427	2.4324E-09	9.0396	-10.230
	CASPER	all	0.017886	-17.475	4.7620E-10	9.0316	-10.189
	Xilinx	alt	0.017724	-17.514	5.6558E-13	9.0391	-10.155
	Xilinx	all	0.017689	-17.523	2.4099E-10	9.0470	-10.148
N=16384	ASTRON	alt	0.017164	-17.654	4.6082E-13	18.163	-10.231
	ASTRON	all	0.018742	-17.272	2.9306E-09	18.058	-10.619
	CASPER	all	0.017959	-17.457	7.9544E-10	18.092	-10.428
	Xilinx	all	0.017157	-17.656	1.5468E-11	18.116	-10.232
N=65536	ASTRON	all	0.020645	-16.852	2.9676E-09	24.333	-11.074
	CASPER	all	0.018943	-17.225	8.0182E-10	24.527	-10.660
	Xilinx	all	0.017345	-17.608	5.4338E-12	24.773	-10.263
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017742	-17.510	2.6688E-12	9.0370	-10.158
	ASTRON	all	0.018084	-17.427	2.4324E-09	9.0396	-10.230
	CASPER	all	0.017886	-17.475	4.7620E-10	9.0316	-10.189
N=16384	ASTRON	alt	0.017164	-17.654	4.6082E-13	18.163	-10.231
	ASTRON	all	0.018742	-17.272	2.9306E-09	18.058	-10.619
	CASPER	all	0.017959	-17.457	7.9544E-10	18.092	-10.428
N=65536	ASTRON	all	0.020645	-16.852	2.9676E-09	24.333	-11.074
	CASPER	all	0.018943	-17.225	8.0182E-10	24.527	-10.660

Table 7.5: Numerical analysis results of hardware test outputs for $L = 4$, tested using Test Vector 0.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017745	-17.509	4.3004E-12	9.037	-10.159
	ASTRON	all	0.018071	-17.43	2.2931E-09	9.0396	-10.227
	CASPER	all	0.017912	-17.469	6.6492E-10	9.0302	-10.197
	Xilinx	all	0.017694	-17.522	2.2437E-10	9.0381	-10.159
N=16384	ASTRON	alt	0.017168	-17.653	4.443E-13	18.164	-10.231
	ASTRON	all	0.018746	-17.271	2.9463E-09	18.058	-10.62
	CASPER	all	0.018002	-17.447	8.8355E-10	18.048	-10.439
	Xilinx	all	0.017167	-17.653	1.5337E-11	18.126	-10.224
N=65536	ASTRON	all	0.020648	-16.851	2.9719E-09	24.333	-11.075
	CASPER	all	0.01904	-17.203	8.53E-10	24.663	-10.673
	Xilinx	all	0.017344	-17.608	5.5545E-12	24.887	-10.263
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017745	-17.509	4.3004E-12	9.037	-10.159
	ASTRON	all	0.018071	-17.43	2.2931E-09	9.0396	-10.227
	CASPER	all	0.017912	-17.469	6.6492E-10	9.0302	-10.197
N=16384	ASTRON	alt	0.017168	-17.653	4.443E-13	18.164	-10.231
	ASTRON	all	0.018746	-17.271	2.9463E-09	18.058	-10.62
	CASPER	all	0.018002	-17.447	8.8355E-10	18.048	-10.439
N=65536	ASTRON	all	0.020648	-16.851	2.9719E-09	24.333	-11.075
	CASPER	all	0.01904	-17.203	8.53E-10	24.663	-10.673

Table 7.6: Numerical analysis results of simulation test outputs for $L = 8$, tested using Test Vector 0.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017742	-17.51	2.6688E-12	9.037	-10.158
	ASTRON	all	0.018084	-17.427	2.4324E-09	9.0396	-10.23
	CASPER	all	0.017899	-17.472	5.7301E-10	9.031	-10.194
	Xilinx	all	0.017694	-17.522	2.1879E-10	9.047	-10.15
N=16384	ASTRON	alt	0.017164	-17.654	4.6082E-13	18.163	-10.231
	ASTRON	all	0.018742	-17.272	2.9306E-09	18.058	-10.619
	CASPER	all	0.017999	-17.447	8.7857E-10	18.048	-10.438
	Xilinx	all	0.017165	-17.654	1.653E-11	18.116	-10.234
N=65536	ASTRON	all	0.020645	-16.852	2.9676E-09	24.333	-11.074
	CASPER	all	0.019044	-17.202	8.9729E-10	24.653	-10.685
	Xilinx	all	0.017341	-17.609	5.3801E-12	24.887	-10.262
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	alt	0.017742	-17.51	2.6688E-12	9.037	-10.158
	ASTRON	all	0.018084	-17.427	2.4324E-09	9.0396	-10.23
	CASPER	all	0.017899	-17.472	5.7301E-10	9.031	-10.194
N=16384	ASTRON	alt	0.017164	-17.654	4.6082E-13	18.163	-10.231
	ASTRON	all	0.018742	-17.272	2.9306E-09	18.058	-10.619
	CASPER	all	0.017999	-17.447	8.7857E-10	18.048	-10.438
N=65536	ASTRON	all	0.020645	-16.852	2.9676E-09	24.333	-11.074
	CASPER	all	0.019044	-17.202	8.9729E-10	24.653	-10.685

Table 7.7: Numerical analysis results of hardware test outputs for $L = 8$, tested using Test Vector 0.

When comparing each core’s simulation and hardware test results, test platform performance matches well when analysing SAP, SFDR and SNR values; however, closer inspection of each MSE value reveals that hardware tests performed better than their simulation counterparts at times, with the opposite also true at other times. An extreme case is the ASTRON $N = 1024$, alternating shift schedule test, where the hardware result’s MSE is almost half the MSE of its simulation equivalent. This could be due to inaccuracies in the simulation model caused by modelling fixed-point arithmetic within a floating-point system, or given that the worst discrepancies occur for the FFT cores implemented in HDL, the Vivado simulator that simulates these cores may inject some level of error into the system.

Alternatively, the Xilinx $N = 65536$ case performed worse in hardware, with an MSE value 1.6 times that of its simulation result. That being said, this hardware MSE value is still the most accurate result within that test configuration and is of magnitude 10^{-12} . These minor discrepancies may be attributed to simulation tests being performed in an environment that models ideal conditions. In reality, many variables impact hardware performance; including electrical noise within a system, hardware temperature and even component degradation. There is no discernible difference between tests conducted on AU50 and SKARAB hardware platforms.

ASTRON’s performance is consistent across both wideband factors, while CASPER’s degrades slightly for a wideband factor of $L = 8$. In the worst case, the MSE for the CASPER hardware test for $N = 1024$ increases by a factor of 1.2 between wideband factors $L = 4$ and $L = 8$. It is worth noting that even with this slight deterioration, the CASPER FFT is more accurate when compared to the FFTW than its ASTRON equivalent.

Xilinx’s performance varies across wideband factors. For a hardware test of $N = 1024$ and shifting at every stage, the $L = 8$ case has an MSE that is 0.9 times the MSE of the $L = 4$ case. For $N = 16384$, the $L = 8$ MSE is 1.07 times the $L = 4$ MSE test result. Despite this variation, Xilinx is consistently the most accurate of all cores for all FFT lengths and wideband factors, with the smallest MSE and a worst case SNR difference from the FFTW equivalent of $0.018dB$ (hardware test, $N = 65536$, $L = 4$). The ASTRON’s worst case SNR difference is $0.83dB$ for the same configuration. CASPER’s numerical performance lies midway between the Xilinx and ASTRON cores for this particular test case when compared against outputs generated using the same shifting schedule.

A slight increase in average power is noticeable, particularly for ASTRON and CASPER cores, when compared to the average power of the FFTW output. This increase is most noticeable when shifting occurs at every stage, where the combination of such aggressive shifting and rounding to infinity superimpose, introducing a more pronounced bias to the output due to increased quantisation noise. This can be confirmed when analysing the the non-normalised PSDs of these cores against the PSD of the FFTW.

Figure 7.1 below shows the non-normalised PSDs of the ASTRON FFT hardware test output and the equivalent FFTW output for $N = 65536$ and an ASTRON wideband factor of $L = 4$. The ASTRON FFT spectrum has been scaled appropriately to compensate for the downshifting caused both by shifting at every stage and the use of guard bits and increased stage datawidths. In order to illustrate the scaling effect that has occurred due to quantisation noise, these PSDs have not been normalised. The magnified sections of the linear plot clearly show the effects of biasing caused by amplified quantisation noise on the ASTRON FFT output.

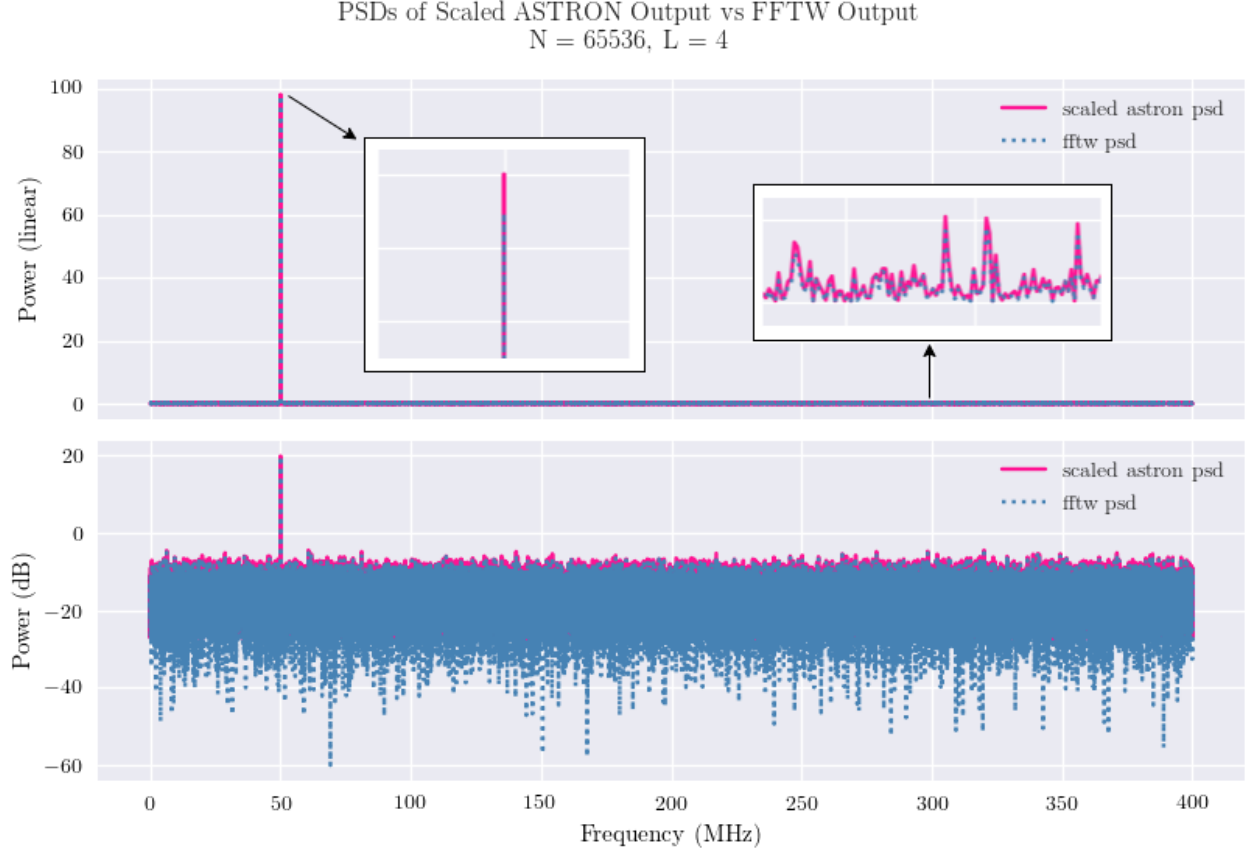


Figure 7.1: Non-normalised PSDs of ASTRON vs FFTW outputs.

In the single case where an alternating shift pattern prevented overflow for the Xilinx core, this core once again outperformed ASTRON, maintaining average power, SFDR and SNR fidelity to the FFTW’s equivalent to a highly accurate degree. An improvement in all numerical metrics can be seen for the ASTRON and Xilinx outputs for an alternating shifting schedule when compared to the results of shifting at every stage. In the ASTRON core’s case, for $N = 16384$ and $L = 4$, an alternating shifting schedules has reduced the MSE between the core’s output and the FFTW by a factor of $1.508e^{-4}$, and improved the spectral SNR by $0.39dB$. This illustrates the effect the shifting schedule can have on the precision of an FFT’s spectral output.

Improved Shifting Schedules

Simulation tests were conducted using the ASTRON and CASPER $N = 16384, L = 4$ FFT test designs to determine improved shifting schedules from those tested above. The Xilinx equivalent was not included in this test due to its non-functional overflow flag. Due to the differences between the DIF and DIT algorithms used by the ASTRON and CASPER cores, different shifting schedules are required for each. The following shifting schedule patterns have been selected:

- CASPER: 00010101111111
- ASTRON: 00001111101010

Table 7.8 below presents the numerical performance results of simulation tests using the above shifting schedules against their corresponding results presented in Table 7.4.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=16384	ASTRON	improved	0.017162	-17.654	3.8074E-13	18.163	-10.231
	ASTRON	alt	0.017168	-17.653	4.443E-13	18.164	-10.231
	ASTRON	all	0.018746	-17.271	2.9463E-09	18.058	-10.62
	CASPER	improved	0.017234	-17.636	4.6617E-12	18.154	-10.242
	CASPER	all	0.017961	-17.457	7.9862E-10	18.092	-10.428

Table 7.8: Numerical analysis results of CASPER and ASTRON simulation test outputs for an improved shifting schedule against nonoptimal shifting schedules.

The shifting schedule selected for the ASTRON core has not improved its performance significantly when compared to the results of an alternating shifting schedule; however, there has been a slight improvement in the MSE between the core's output and FFTW output. From these results, the alternating shifting schedule was already almost ideal, with only minimal improvement noticed when using the shift schedule selected above. The shifting schedule selected for the CASPER core has noticeably improved its performance in comparison to the aggressive shift case - SNR has been improved by $0.186dB$, MSE has been reduced by a factor of $5.837e^{-3}$ and the increase to total average power due to quantisation effects has lessened. The guard bits applied to the ASTRON core result in the core requiring less shifting than its CASPER equivalent. Both alternating and improved shift schedules for the ASTRON core resulted in a better overall performance than the CASPER core's improved shifting schedule, due to the lower number of shifts required for the ASTRON core.

Rounding Scheme Effects

When creating the test designs for use in the implemented test system, the CASPER core was configured to use a rounding scheme that rounds to infinity based on this scheme being the only option available for the ASTRON core; however, the CASPER core also provides an option to use an even rounding scheme. A simulation test has been performed for a single CASPER core test configuration to investigate the effects that different rounding schemes may have on performance.

This test was performed for an FFT length of $N = 16384$ and a wideband factor of $L = 4$, using both a shifting schedule that applies a shift at each stage and the improved shifting schedule presented above. The results of these even rounding tests are presented below in Table 7.9 against the results of their infinite rounding equivalents.

AU50: CASPER	Rounding	Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=16384	even	improved	0.017144	-17.659	8.4840E-14	18.163	-10.227
	even	all	0.017147	-17.658	7.8073E-12	18.126	-10.219
	infinite	improved	0.017234	-17.636	4.6617E-12	18.154	-10.242
	infinite	all	0.017961	-17.457	7.9862E-10	18.092	-10.428

Table 7.9: Numerical analysis results of CASPER, $N = 16384$, $L = 4$ simulation tests for even and infinite rounding schemes.

The above results show a significant improvement in the performance of the CASPER core when an even rounding scheme is used. For even rounding with an improved shifting schedule, the CASPER core yields an output with the lowest MSE against the FFTW output of any core, introduces no increase to average power within the spectrum and preserves the SNR to the same degree as the FFTW reference.

Xilinx provides no documentation on the rounding scheme used within its core's implementation; however, comparing the performance of the CASPER core when using even rounding against the performance of the Xilinx core suggests that it uses either an even or odd rounding scheme that reduces bias and quantisation effects.

Plots of the normalised PSDs for simulation tests performed for both improved shifting schedules and even rounding schemes can be found in Appendix B. This concludes the presentation and discussion of results for Test Case 0. The following section presents an analysis of the results of Test Case 1 core outputs.

7.1.2 Test Case 1

This test case was derived to test each FFT core's performance for a signal with a dynamic range at the extremes of the system's data representation. Given the large scale of the sinusoidal component of Test Vector 1, all FFT cores were tested using a shifting schedule that applied a shift at each FFT stage. The numerical analysis results of this test case are given below.

To provide a reference for each test's results, the numerical performance results of the reference FFTW software implementation are presented first in Table 7.10 below.

FFTW	Average Power	Average Power (dB)	SFDR (dB)	SNR (dB)
N=1024	0.40500	-3.9254	112.47	92.388
N=16384	0.40500	-3.9255	121.21	92.431
N=65536	0.40500	-3.9254	126.27	92.400

Table 7.10: Numerical analysis results of reference FFTW outputs using Test Vector 1.

The following tables present the simulation and hardware results of each FFT core for both wideband factors. Tables 7.11 and 7.12 present wideband factor $L = 4$ test results, with the test results for a wideband factor of $L = 8$ given in Tables 7.6 and 7.7. SAP values have been calculated using the same method as in Section 7.1.1 above.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40429	-3.9331	7.7511E-15	80.364	55.507
	Xilinx	all	0.40423	-3.9337	4.3623E-15	80.363	56.755
N=16384	ASTRON	all	0.40508	-3.9246	5.6854E-16	71.333	55.206
	CASPER	all	0.4051	-3.9244	1.2539E-13	80.364	43.437
	Xilinx	all	0.40494	-3.9261	3.2208E-11	64.339	31.386
N=65536	ASTRON	all	0.40513	-3.9241	3.8699E-15	71.333	47.989
	CASPER	all	0.40514	-3.924	5.0574E-13	80.364	37.397
	Xilinx	all	0.40507	-3.9247	1.2211E-12	73.373	35.483
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40429	-3.9331	7.7511E-15	80.364	55.507
N=16384	ASTRON	all	0.40508	-3.9246	5.6854E-16	71.333	55.206
	CASPER	all	0.4051	-3.9244	1.2539E-13	80.364	43.437
N=65536	ASTRON	all	0.40513	-3.9241	3.8699E-15	71.333	47.989
	CASPER	all	0.40514	-3.924	5.0574E-13	80.364	37.397

Table 7.11: Numerical analysis results of simulation test outputs for $L = 4$, tested using Test Vector 1.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40423	-3.9337	7.0361E-15	80.363	55.717
	Xilinx	all	0.40423	-3.9337	5.4261E-15	80.363	56.281
N=16384	ASTRON	all	0.40508	-3.9245	9.1065E-16	68.323	54.173
	CASPER	all	0.4051	-3.9244	1.2561E-13	80.364	43.433
	Xilinx	all	0.40499	-3.9256	7.6562E-14	77.353	44.508
N=65536	ASTRON	all	0.40513	-3.9241	4.6572E-15	68.323	47.586
	CASPER	all	0.40508	-3.9246	5.0575E-13	80.363	37.397
	Xilinx	all	0.40501	-3.9253	2.9582E-13	77.352	38.562
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40423	-3.9337	7.0361E-15	80.363	55.717
N=16384	ASTRON	all	0.40508	-3.9245	9.1065E-16	68.323	54.173
	CASPER	all	0.4051	-3.9244	1.2561E-13	80.364	43.433
N=65536	ASTRON	all	0.40513	-3.9241	4.6572E-15	68.323	47.586
	CASPER	all	0.40508	-3.9246	5.0575E-13	80.363	37.397

Table 7.12: Numerical analysis results of hardware test outputs for $L = 4$, tested using Test Vector 1.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40434	-3.9325	9.8768E-15	80.364	54.980
	Xilinx	all	0.40423	-3.9337	3.6879E-15	80.363	57.120
N=16384	ASTRON	all	0.40508	-3.9246	5.6854E-16	71.333	55.206
	CASPER	all	0.40510	-3.9244	1.5175E-13	80.364	43.022
	Xilinx	all	0.40499	-3.9256	7.0905E-14	76.384	44.675
N=65536	ASTRON	all	0.40513	-3.9241	3.8699E-15	71.333	47.989
	CASPER	all	0.40520	-3.9233	6.0493E-13	80.364	37.008
	Xilinx	all	0.40506	-3.9248	2.4500E-13	76.384	38.971
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40434	-3.9325	9.8768E-15	80.364	54.980
N=16384	ASTRON	all	0.40508	-3.9246	5.6854E-16	71.333	55.206
	CASPER	all	0.40510	-3.9244	1.5175E-13	80.364	43.022
N=65536	ASTRON	all	0.40513	-3.9241	3.8699E-15	71.333	47.989
	CASPER	all	0.40520	-3.9233	6.0493E-13	80.364	37.008

Table 7.13: Numerical analysis results of simulation test outputs for $L = 8$, tested using Test Vector 1.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40423	-3.9337	8.8256E-15	80.363	55.224
	Xilinx	all	0.40423	-3.9337	4.1368E-15	80.363	56.870
N=16384	ASTRON	all	0.40508	-3.9245	9.1065E-16	68.323	54.173
	CASPER	all	0.40505	-3.9249	1.5173E-13	80.364	43.023
	Xilinx	all	0.40499	-3.9256	7.9250E-14	76.384	44.433
N=65536	ASTRON	all	0.40513	-3.9241	4.6572E-15	68.323	47.586
	CASPER	all	0.40515	-3.9239	6.0468E-13	80.364	37.009
	Xilinx	all	0.40501	-3.9254	2.5534E-13	76.383	38.882
SKARAB		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=1024	ASTRON	all	0.40434	-3.9325	1.0486E-17	71.333	71.333
	CASPER	all	0.40423	-3.9337	8.8256E-15	80.363	55.224
N=16384	ASTRON	all	0.40508	-3.9245	9.1065E-16	68.323	54.173
	CASPER	all	0.40505	-3.9249	1.5173E-13	80.364	43.023
N=65536	ASTRON	all	0.40513	-3.9241	4.6572E-15	68.323	47.586
	CASPER	all	0.40515	-3.9239	6.0468E-13	80.364	37.009

Table 7.14: Numerical analysis results of hardware test outputs for $L = 8$, tested using Test Vector 0.

The general performance of each core matches the performance described for Test Case 0 when comparing results across test platforms. There are minor discrepancies between hardware and simulation test results, the cause of which having already been discussed in Section 7.1.1, and there are no differences in performance between SKARAB and AU50 hardware platform test cases.

While Test Case 0 resulted in an SNR preservation with a worst case SNR difference of $0.83dB$, an examination of each of the tables above shows significant spectral SNR degradation for every FFT core test. This deterioration increases with increasing FFT length, and is attributed to both the quantisation of such a low level of noise and the operation of the FFT itself. This is explained in more detail below.

Figure 7.2 below shows a magnitude plot of the complex output of the CASPER FFT hardware test against the FFTW equivalent that has been scaled down by the shifting schedule factor. The plot shows that the CASPER output has a significantly higher noise floor than that of the FFTW for the same level of scaling, with noise values with a magnitude equal to the $\langle 16, 15 \rangle$ system's precision of 2^{-15} .

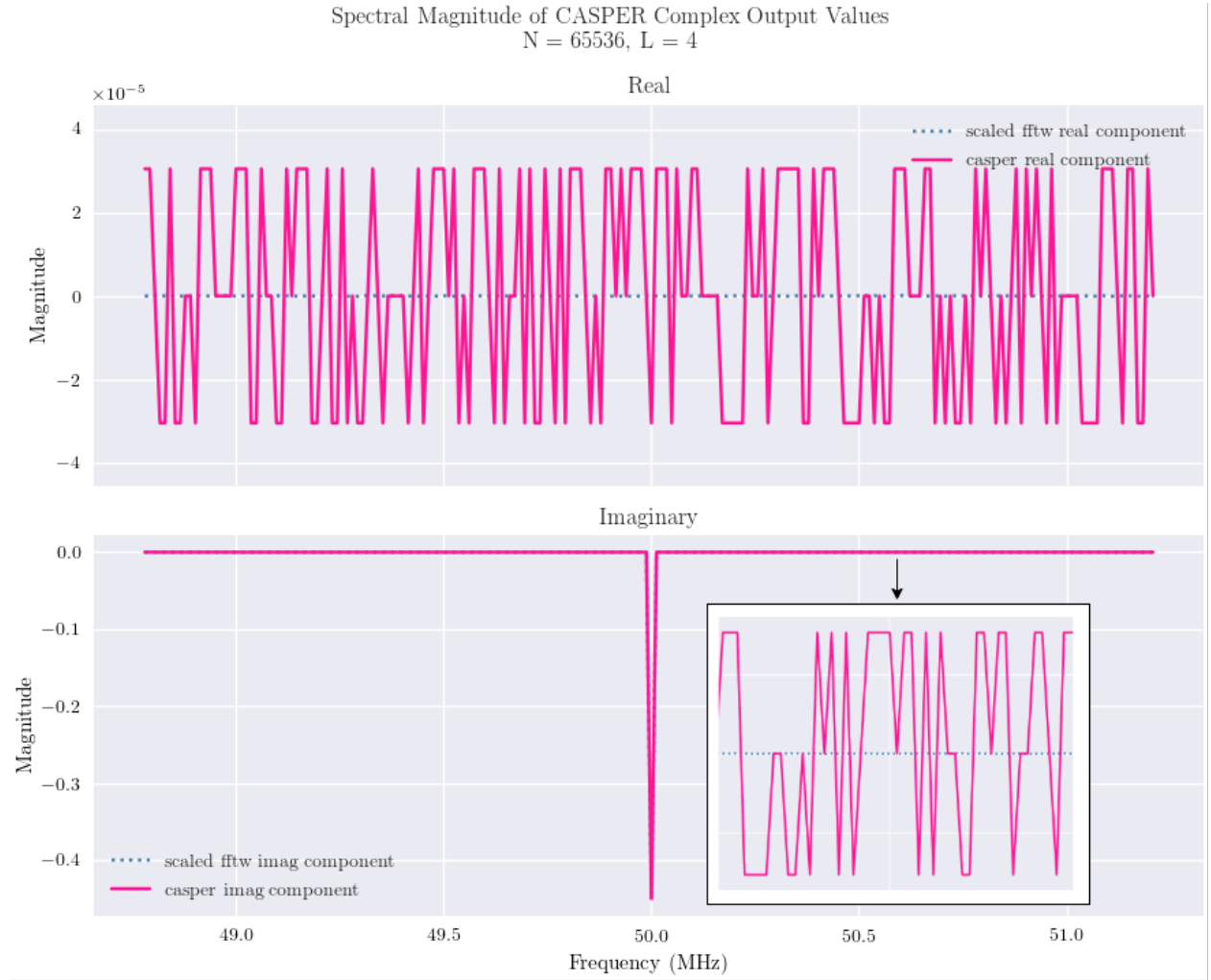


Figure 7.2: Real and complex valued output magnitudes of CASPER hardware test vs scaled FFTW.

Magnitude plots of a sample-set (of 200 data points) of the real outputs for the same ASTRON and Xilinx hardware tests, shown below in Figure 7.3, confirm the presence of the same level of quantisation noise for both cores. While this noise is present in the ASTRON core's output, there are less overall noise samples when compared to the CASPER and Xilinx cores.

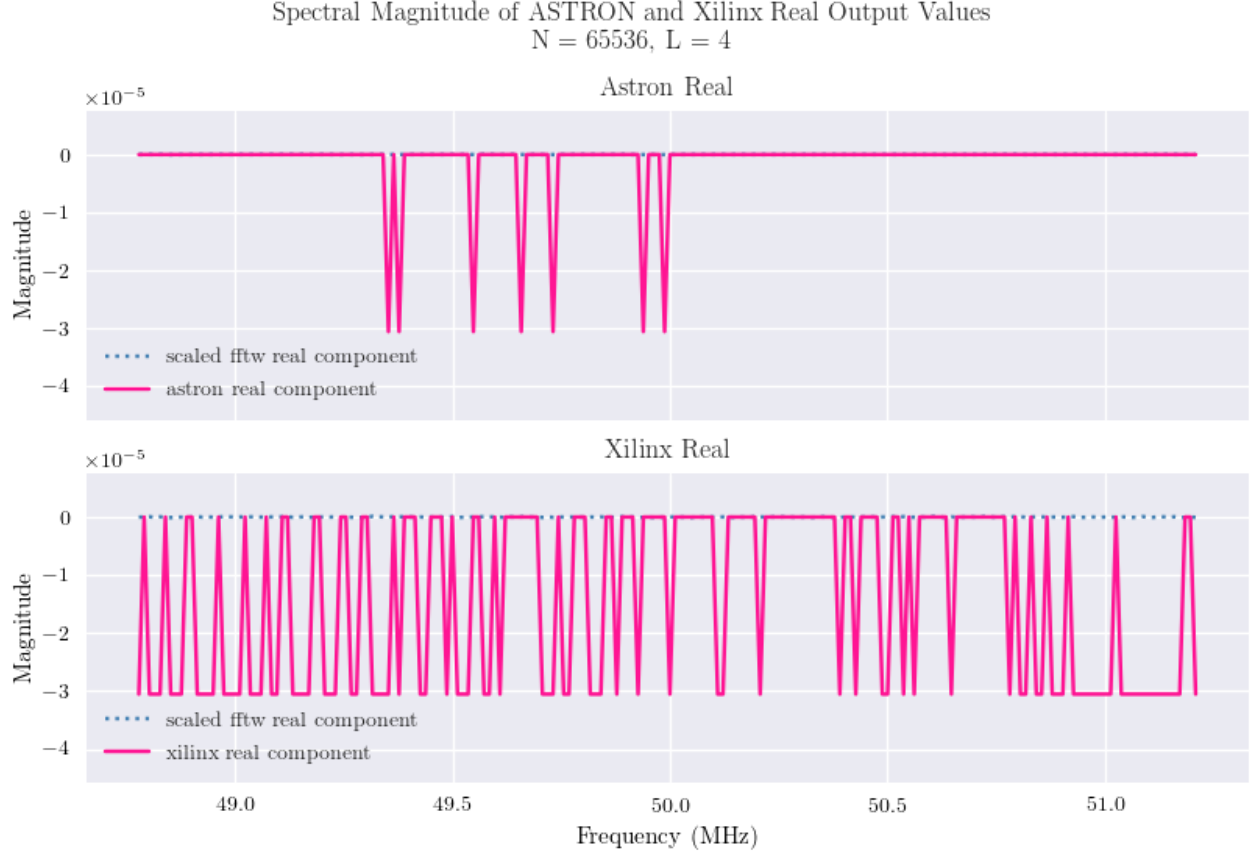


Figure 7.3: Real valued output magnitudes of ASTRON and Xilinx hardware tests vs scaled FFTW.

This noise can be explained when considering the quantisation of the low level noise values present in Test Vector 1 and the operation of the FFT itself. Due to the statistical properties of the noise model used, when the generated test vector is quantised to a $\langle 16, 15 \rangle$ fixed-point representation, the rounding that occurs during this quantisation results in random toggling of the LSB of the fixed-point input values. In the case of the smallest values of the input vector (where the sinusoid crosses the x-axis), these values are either rounded to zero, or to a positive/negative value with a magnitude equal to the system's precision (i.e. $+/- 2^{-15} = +/- 3.05175e^{-5}$).

The explanation for the amplified quantisation noise seen above is given using the CASPER implementation as a specific example. Figure 7.4 below depicts a typical butterfly implementation within the CASPER FFT. The first FFT stage for the CASPER core does not apply any twiddle factor to the input, and only performs a complex addition and subtraction (see Figure 2.3 for the signal flow of a DIT FFT algorithm).

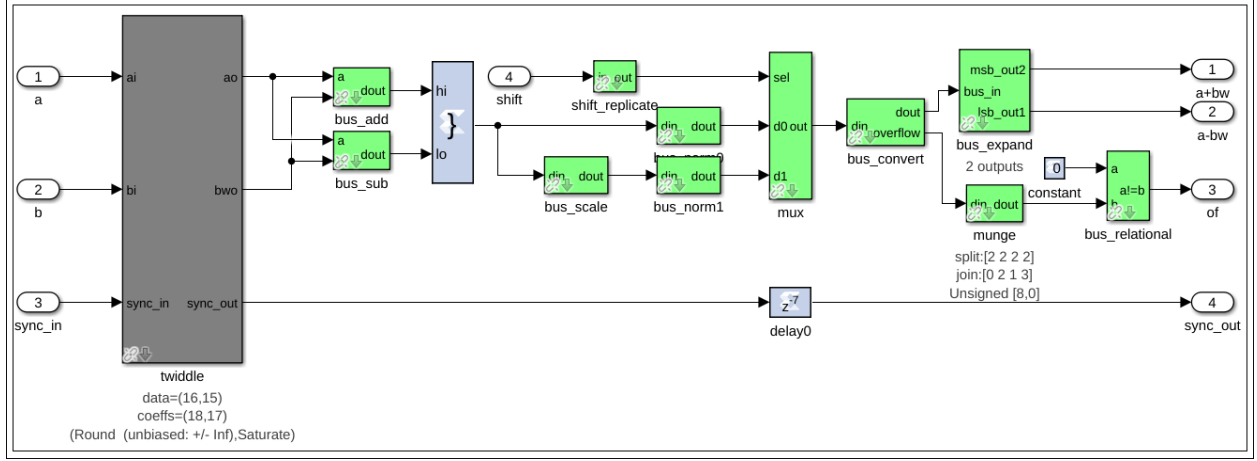


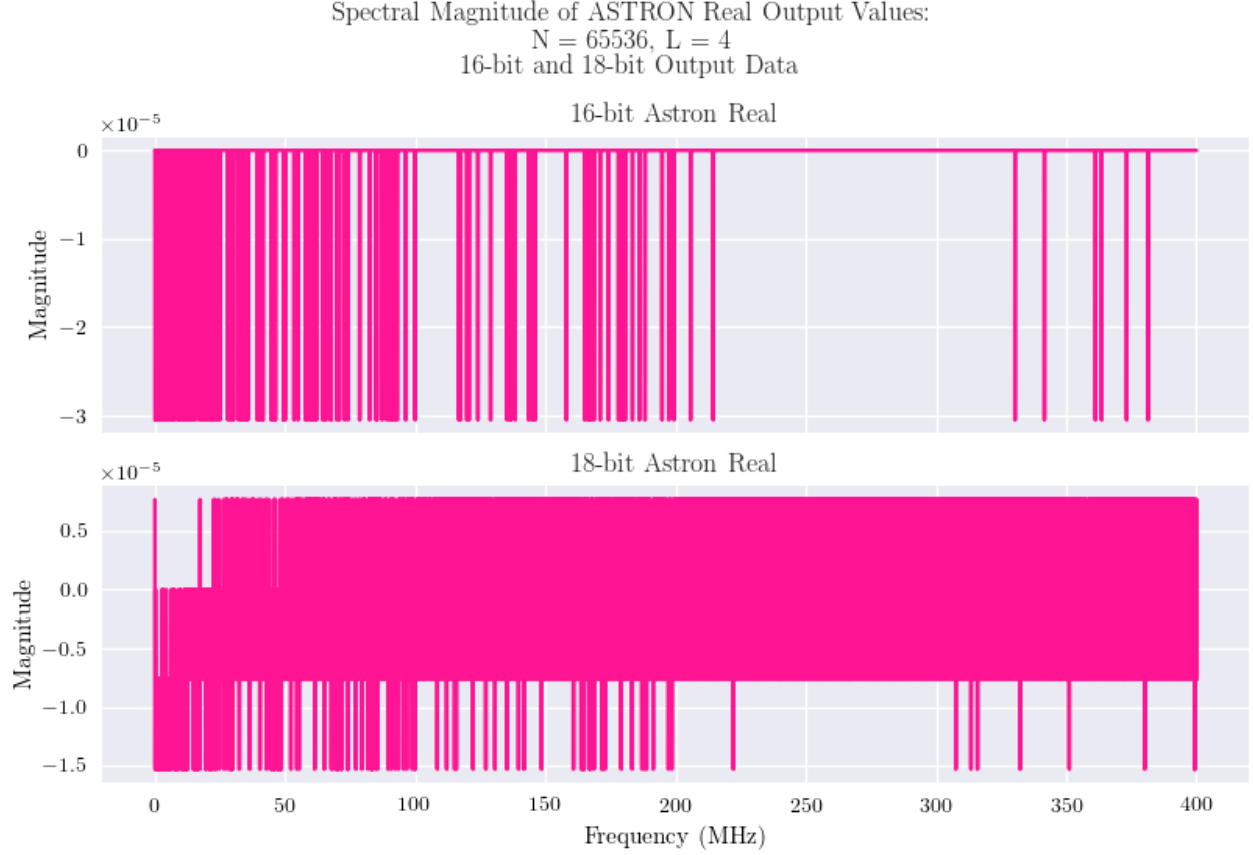
Figure 7.4: Diagram of a butterfly stage implemented for the CASPER FFT.

As the input vector's sinusoid has been generated using a frequency that is an exact integer multiple of each FFT length's frequency resolution, there will be instances where the inputs to the butterflies of the FFT's first stage are the rounded smallest values near the x-axis crossing. For stages where no complex multiplication occurs (as is the case with the first stage of a DIT implementation), these values pass through to the *add* and *sub* blocks. In this case, the outputs of these blocks have the following possible combinations:

$$\begin{aligned}
 \pm 3.05175e^{-5} - \pm 3.05175e^{-5} &= 0 \\
 0 + \pm 3.05175e^{-5} &= \pm 3.05175e^{-5} \\
 \pm 3.05175e^{-5} + \pm 3.05175e^{-5} &= \pm 2 \times 3.05175e^{-5}
 \end{aligned} \tag{7.1}$$

If a shift is set to occur for the current stage, nothing will happen to the output value of the first outcome, the second outcome will be downshifted to zero, and the last outcome will be downshifted to $3.05175e^{-5}$. This last outcome's value is now at the LSB of the system post-shift, and is propagated along to the next stage with the other zero values, where the same phenomenon occurs. This results in the presence of noise values at the LSB of the system in the output spectrum, with this effect amplified for larger FFT lengths with more stages and a greater number of output frequency channels. Additionally, the main frequency component of the input vector is also downshifted at each stage. This down-scaling of the main frequency component output value combined with the amplified output noise due to quantisation causes the SNR degradation observed in Tables 7.11 to 7.7.

ASTRON seems to perform better in this regard, with the best SNR preservation between all cores; however, the requantisation of the 18-bit stage data down to an output bitwidth of 16-bits reduces the precision of the output data as it rounds and then removes the last two LSBs worth of data. A simulation test of the ASTRON core was performed for an FFT length of $N = 65536$ and wideband factor $L = 4$, where the output bitwidth of the core was kept at 18-bits. Figure 7.5 below shows plots of the full real-valued spectral components for both the requantised 16-bit ASTRON output and the 18-bit ASTRON output cases.



The above plot reveals that the presence of noise in the ASTRON core is more pronounced than initially observed, and confirms that the requantisation down to 16-bit data reduces the noise floor. In the interests of a fair comparison, numerical analysis had been performed on the 18-bit ASTRON simulation test output, the results of which are presented below in Table 7.15 against the results of the other equivalent FFT cores.

AU50		Shift	SAP	SAP (dB)	MSE	SFDR (dB)	SNR (dB)
N=65536	ASTRON 16-bit	all	0.40513	-3.9241	3.8699E-15	71.333	47.989
	ASTRON 18-bit	all	0.40515	-3.9239	6.3960E-13	76.384	36.887
	CASPER	all	0.40514	-3.924	5.0574E-13	80.364	37.397
	Xilinx	all	0.40507	-3.9247	1.2211E-12	73.373	35.483

Table 7.15: Numerical analysis results of ASTRON 18-bit output data against 16-bit output data results.

The results above confirm that when output data is kept at 18-bits and precision is preserved, the ASTRON core's performance is reduced in a manner similar to both CASPER and Xilinx cores. While ASTRON's performance results are the most accurate between cores for 16-bit output data, the above 18-bit output data test yields the worst results between all equivalent cores. The above results imply that ASTRON only outperforms the other cores for 16-bit output data due to the requantisation of the output causing a loss in precision, and not because of the core's inherent performance. A full suite of tests for all ASTRON core test configurations would need to be performed; both to confirm this is true in the general case, and gather results indicative of the ASTRON core's true performance across all tests.

All core outputs experience minimal to no increase in average spectral power, as even with the high level of quantisation noise, the main frequency component is scaled down significantly due to the aggressive shifting schedule used and average power is conserved. Analysing the hardware test results for CASPER and Xilinx cores, Xilinx has the best performance, though not by much. The MSE results of both cores are all also values of magnitudes between e^{-12} to e^{-15} ; however, both experience severe SNR degradation that is exacerbated as FFT length increases (due to the extreme quantisation effects explained previously).

Preservation of SNR is perhaps the most important metric in analysing the performance of each FFT core, as the value provides a clear indication of how each core is affected by the quantisation noise introduced into the output. For Test Case 0, while MSE values and overall increase to average power varied between cores, each core preserved the input test vector's SNR with a worst case loss no greater than $0.83dB$. For Test Case 1, minimal increase to average power is observed, and each core returns an MSE value indicative of highly accurate results; however, the SNR of each core's output is greatly reduced due to the effects of quantisation, with a worst case difference of almost $60dB$ for $N = 65536$ test cases.

With the numerical results for both Test Case 0 and Test Case 1 presented, the following section presents the hardware resource utilisation for each core and their maximum achievable clock rates (for an upper limit of a tested clock rate of 600MHz).

7.2 Hardware Resource Utilisation and Clock Rates

This section presents the results of the resource utilisation of each final hardware compile and the maximum clock rates achieved for each core. Resource utilisation and power requirements have been gathered from the reports generated by Vivado for each implemented design, as described in Section 6.1.2. Clock rates have been determined based on the results of the test described within the same section. Resource utilisation data is presented first, with clock rate results presented after.

7.2.1 Resource Utilisation

Tables 7.16 and 7.17 below present the power requirements and resource utilisation of each FFT core for wideband factors $L = 4$ and $L = 8$. All core test designs have been compiled using an FPGA clock rate of 160MHz. All values refer only to the FFT core instantiated within the test design, and not the test design as a whole. Resource utilisation is presented as the percentage used of total available resources on each hardware platform.

AU50		Core Power	LUT as Logic	LUT as Mem	Flip-Flops	BRAM	DSP Slices
N=1024	ASTRON	0.33 W	1.07%	0.57%	0.68%	0.30%	2.42%
	CASPER	0.47 W	0.51%	0.68%	0.63%	0.78%	0.87%
	Xilinx	0.445 W	0.56%	0.59%	0.62%	0.00%	0.60%
N=16384	ASTRON	0.535 W	1.51%	5.18%	1.45%	5.95%	3.49%
	CASPER	0.719 W	0.68%	1.00%	0.94%	3.13%	1.14%
	Xilinx	0.848 W	0.79%	0.40%	0.79%	5.17%	0.91%
N=65536	ASTRON	1.133 W	1.78%	19.70%	3.33%	24.26%	4.03%
	CASPER	1.152 W	0.79%	1.12%	1.31%	11.79%	1.28%
	Xilinx	1.323 W	0.94%	0.43%	0.92%	18.79%	1.06%
SKARAB		Core Power	LUT as Logic	LUT as Mem	Flip-Flops	BRAM	DSP Slices
N=1024	ASTRON	0.617 W	2.16%	1.32%	1.36%	0.27%	4.00%
	CASPER	0.566 W	1.10%	1.63%	1.32%	0.71%	1.44%
N=16384	ASTRON	1.388 W	3.00%	12.00%	2.93%	5.99%	5.78%
	CASPER	0.878 W	1.49%	2.40%	1.97%	2.86%	1.89%
N=65536	ASTRON	2.495 W	3.44%	45.60%	6.74%	24.35%	6.67%
	CASPER	1.496 W	1.71%	2.69%	2.71%	10.85%	2.11%

Table 7.16: Power requirements and percentage resource utilisation per FFT core for $L = 4$.

AU50		Core Power	LUT as Logic	LUT as Mem	Flip-Flops	BRAM	DSP Slices
N=1024	ASTRON	0.656 W	1.98%	0.82%	1.22%	0.60%	4.57%
	CASPER	1.03 W	1.08%	1.28%	1.35%	1.15%	1.95%
	Xilinx	0.773 W	1.12%	0.78%	1.04%	0.00%	1.06%
N=16384	ASTRON	0.871 W	2.88%	5.52%	2.17%	5.95%	6.72%
	CASPER	1.393 W	1.39%	2.03%	1.78%	2.98%	2.49%
	Xilinx	1.485 W	1.60%	1.11%	1.50%	5.39%	1.41%
N=65536	ASTRON	1.49 W	3.37%	20.09%	3.99%	24.11%	7.80%
	CASPER	1.863 W	1.54%	2.25%	2.24%	10.75%	2.76%
	XILINX	2.187 W	1.85%	0.69%	1.59%	22.51%	1.76%
SKARAB		Core Power	LUT as Logic	LUT as Mem	Flip-Flops	BRAM	DSP Slices
N=1024	ASTRON	0.783 W	3.98%	1.91%	2.46%	0.54%	7.56%
	CASPER	1.162 W	2.25%	3.03%	2.75%	1.05%	3.22%
N=16384	ASTRON	0.998 W	5.77%	12.78%	4.38%	5.44%	11.11%
	CASPER	1.638 W	2.92%	4.78%	3.65%	2.72%	4.11%
N=65536	ASTRON	1.783 W	6.59%	46.49%	8.04%	24.22%	12.89%
	CASPER	2.312 W	3.24%	5.29%	4.59%	9.83%	4.56%

Table 7.17: Power requirements and percentage resource utilisation per FFT core for $L = 8$.

Both tables above show that as FFT length and wideband factor increase, the number of resources used increases too. In all cases bar the $N = 1024$ configuration, ASTRON is the least resource efficient for all resource types presented. The implementation of this core is such that it always computes two real FFTs when configured as a real-to-complex FFT. In the case of these tests only one real input test vector was processed, with the core's other data input set to process zero values. The resource utilisation for this core is therefore highly inefficient when only one input signal requires processing, but this inefficiency is reduced if multiple input streams are processed. The need for guard bits and an increased stage bitwidth to prevent overflow impacts the number of flip-flops and overall memory required by this core too.

Additionally, the ASTRON core does not utilise a biphase architecture that optimises the use of all logic resources for each clock cycle and halves the number of logic resources required for the same number of operations. While this core utilises the most resources of all tested cores in its native configuration, it provides an option to use a single pipelined FFT reorder buffer that would reduce the number of required BRAMs by one per wideband factor.

The CASPER core can be configured to process only one input stream of data and utilises the biphase method (described in Section 2.4.2) for this single input to optimise resource usage. Configuration options are also available when using the core that allow for the specification of BRAM thresholds, and some BRAM storage optimisations are automatically enabled within the design by default.

The Xilinx core generally uses less resources than both ASTRON and CASPER cores, with the exception of BRAMs, where its usage is similar to that of ASTRON's. An important note is that the Xilinx implementation is a full complex-to-complex FFT that has been configured to perform a real FFT by setting the imaginary input to zero. This is similar to this project's use of the ASTRON core, though it is not quite a fair comparison as Xilinx requires no additional logic within the core to implement using Hermitian symmetry to perform two real FFTs using the logic of one complex core. This core allows for minimal configuration as part of the Model Composer blockset, with only datawidths and the BRAM usage threshold selectable. Any optimisations that may be implemented within the core's architecture are unknown.

Despite using the greatest percentage of resources, Vivado reported that the ASTRON core requires less power than the others. The only exception to this is the SKARAB hardware compile for a wideband factor of $L = 4$, where ASTRON's power requirements are greater than the power required by the CASPER core. There is a direct correlation between increased resource utilisation and required power, and this is true in that the power requirement for each core increases as FFT length increases. The ASTRON core generally reporting the lowest power requirements despite using the most resources therefore differs from what was expected.

As one input stream to the ASTRON core is set to zero, it was hypothesised that the core had perhaps optimised for a zero input signal by not using the logic placed for that data stream. This was tested by setting the second input stream for the AU50, $N = 16384$, $L = 4$ ASTRON design to also process the outputs of the input BRAMs and recompiling. This resulted in the same power requirements given above.

A further power test was performed by compiling unmodified AU50, $N = 16384$, $L = 4$ ASTRON and CASPER designs for a lower clock rate of 100MHz and analysing their power requirements. In both cases, power requirements were reduced; however, the ASTRON core still required less power than the CASPER core. It is known that power requirements scale with clock rate, and it is possible that the additional synthesized logic could allow the design to run at a lower frequency for lower power requirements; however, a detailed review of the synthesized logic and implemented design floorplan would be needed to determine why the estimator reported such low power while using additional logic. This level of analysis was deemed as beyond the scope of this project, and has been included in Chapter 8 as a suggestion for further work.

For completeness, the CASPER test design used to perform a simulation test using even rounding in Section 7.1.1 was compiled, also at a clock rate of 160MHz, and resource utilisation examined. These results are presented below in Table 7.18.

AU50	Rounding	Core Power	LUT as Logic	LUT as Mem	Flip-Flops	BRAM	DSP Slices
N=16384	even	0.689 W	0.69%	0.90%	0.78%	3.13%	1.14%
	infinite	0.719 W	0.68%	1.00%	0.94%	3.13%	1.14%

Table 7.18: Power requirements and percentage resource utilisation for a CASPER core using even and infinite rounding for $N = 16384$ and $L = 4$.

In the even rounding case, the CASPER core reported a lower resource utilisation than its infinite rounding counterpart. These results are presented with the caveat that only one design for a single test configuration was compiled using even rounding, and more test samples would be needed to form a general conclusion as to how the rounding scheme impacts resource utilisation.

The results above were generated from the reports of hardware test designs compiled for a clock rate of 160MHz. The following section presents the maximum achievable clock rate for each core when used within the test designs developed for this research project.

7.2.2 Achievable Clock Rates

Each FFT core test design was compiled for a range of clock rates, [200MHz : 600MHz], at increments of 50MHz to determine the rates at which each core can be operated. Tables 7.19 and 7.20 below present matrices of achieved clock rates for all core test designs for wideband factors $L = 4$ and $L = 8$ respectively. Estimated maximum clock rate values are also presented, with the estimate calculated as described in Section 6.1.2. An entry of ‘-’ in the *Estimated* column means the core met the maximum tested clock rate and no estimation was required. An ‘x’ in the *Achieved* column means the core failed to meet timing for the minimum tested clock rate.

AU50	N=1024		N=16384		N=65536	
	Achieved	Estimated	Achieved	Estimated	Achieved	Estimated
ASTRON	400	444	250	294	x	179
CASPER	600	-	600	-	450	460
Xilinx	600	-	500	533	500	533

SKARAB	N=1024		N=16384		N=65536	
	Achieved	Estimated	Achieved	Estimated	Achieved	Estimated
ASTRON	200	217	200	222	x	195
CASPER	350	369	350	365	350	362

Table 7.19: Achieved and estimated clock rates (in MHz) for $L = 4$.

AU50	N=1024		N=16384		N=65536	
	Achieved	Estimated	Achieved	Estimated	Achieved	Estimated
ASTRON	400	413	300	318	x	190
CASPER	600	-	550	554	450	469
Xilinx	600	-	600	-	300	349

SKARAB	N=1024		N=16384		N=65536	
	Achieved	Estimated	Achieved	Estimated	Achieved	Estimated
ASTRON	200	210	x	192	x	190
CASPER	300	347	300	344	300	337

Table 7.20: Achieved and estimated clock rates (in MHz) for $L = 8$.

All FFT core types achieved higher clock rates for the AU50 hardware platform than for the SKARAB. A key aspect of design timing closure is how efficiently the design tool is able to route logic paths onto the FPGA fabric. As of January 2023, the Virtex-7 family of FPGA chips used on the SKARAB was released over a decade ago, while the Virtex UltraScale+ FPGA chips used on the AU50 are a newer form of technology. Increased achievable clock rates for newer FPGAs are the result of improved FPGA fabrication processes and upgraded resources (e.g. the DSP48E1 slices on Virtex-7 chips vs the DSP48E2 slices on UltraScale+ devices). As such, newer chips with more resources, improved technology and better logic layout planning will route more efficiently.

A closer examination of the results above reveals that the ASTRON core achieved lower clock rates than the CASPER and Xilinx cores for all test designs. A known fan-out issue on the core’s reset path limits operation to lower rates, especially for larger FFT lengths and wideband factors, where the reset fan-out is increased. For a wideband factor of $L = 4$, ASTRON did not meet timing for the lowest tested clock rate of 200MHz on either hardware platform for $N = 65536$. The same is true for a wideband factor of $L = 8$, with the core also failing timing at 200MHz for $N = 16384$ on the SKARAB.

Both CASPER and Xilinx achieved clock rates of 600MHz for $N = 1024$ and clock rates of 500-600MHz for $N = 16384$ for both wideband factors on the AU50. Achieved clock rates dropped for $N = 65536$ for both cores, with Xilinx dropping to 300MHz for a wideband factor of $L = 8$. On the SKARAB, CASPER consistently made timing for 350MHz and 300MHz clock rates for $L = 4$ and $L = 8$ respectively.

A decrease in achievable clock rate is to be expected as FFT lengths and wideband factors increase, as the logic paths that must be routed become longer and more complex. Vivado provides different implementation strategies that can be used to optimise the compile for different purposes. One such strategy optimises for improved timing, which may improve the rates achievable by these cores through more efficient routing. These cores have all been tested using the default implementation strategy that performs no optimisation.

The CASPER FFT core also provides the option to specify delays between components within the core. While these options have not been used in this work, a detailed analysis of the Vivado project’s timing report may reveal details as to where timing problems occur along the routed paths. Modifying delays between components along the problem path or implementing delays along other data paths can improve routing efficiency and assist in improving timing. CASPER also uses the *sync_in* pulse to reset necessary components as it propagates through the design ahead of data. This removes the need for a dedicated reset line and avoids potential fan-out issues as seen with the ASTRON core. There is no documentation on how the Xilinx core handles clocking and resets, and it provides no options to configure any latency values within its implementation.

With the presentation of hardware resource utilisation and achievable clock rate results concluded, this chapter has presented and discussed the results of all tests performed for each FFT core. The following chapter presents conclusions regarding the performance of the tested cores derived from the above results and analysis discussions.

Chapter 8

Conclusion

drops mic

This chapter summarises the results that were analysed and discussed in the previous chapter, remarking on particularly note-worthy performance outcomes. A conclusion as to which FFT core has performed the best based on all test results is then given, with suggestion for further work concluding the chapter.

8.1 Numerical Performance

For Test Case 0, the Xilinx core consistently performed best for a worst case shifting schedule, with CASPER's performance the second most accurate. In the case of an improved shifting schedule, the CASPER core's performance was greatly improved, but still slightly worse than the numerical results of ASTRON's improved shifting schedule output. The use of guard bits and increased stage bitwidth in the ASTRON core reduce the amount of shifting required to prevent overflow compared to the number of shifts required by the CASPER core. Requiring fewer downshifts results in improved numerical performance due to a reduced loss of data precision. For a non-optimal shifting schedule, ASTRON performed the worst. A likely reason for this is the requantisation from the 18-bit stage data down to 16-bit data on the output, resulting in a loss of precision as it rounds and then removes the last two LSBs of each output sample.

For the single simulation test performed where CASPER's rounding scheme was changed from infinite to even rounding, it outperformed all cores and resulted in the most accurate output. More tests would be needed to confirm this is the case across all FFT lengths and wideband factors, but this result indicates that an even rounding scheme greatly improves the numerical accuracy of the FFT output. While it is not fair to compare the performance of a CASPER core using an even rounding scheme to ASTRON's infinite rounding, it could be argued that CASPER is superior based purely on the fact that it provides even rounding as an option, while ASTRON can only round to infinity. The performance of the CASPER core with even rounding compared to the general performance of the Xilinx core suggests that Xilinx utilises an even (or perhaps odd) rounding scheme, resulting in its superior default performance as even/odd rounding reduces the amount of bias introduced through quantisation effects.

For Test Case 1, ASTRON seemingly performed the best of all three cores, but the reason for this has been identified as the effects of requantisation and not due to the core's actual performance. In extreme cases such as Test Case 1, the requantisation from 18-bits to 16-bits on the ASTRON outputs seemed to improve its performance, where it seemed to have the best SNR preservation and overall accuracy for 16-bit output data. If output data was kept at 18-bits and precision was maintained, it was found that the ASTRON core

actually performed worse when compared to the other core’s equivalent results. A full suite of tests would need to be performed to confirm this is the case across all test configurations, but the single simulation test that was performed indicates that this observed superior performance is the result of requantisation removing most of the output’s noise floor.

Otherwise, as for Test Case 0, Xilinx displayed the best performance when shifting at every stage and with CASPER using an infinite rounding scheme. For a signal with such an extreme dynamic range, all cores experienced severe quantisation effects, resulting in SNR degradation with a worst case loss of almost $60dB$ in some cases.

8.2 Resource Utilisation and Clock Rates

The CASPER core’s default configuration has been highly optimised through the implementation of a biplex architecture, resulting in the most efficient resource utilisation on average. The Xilinx cores utilised a similar number of BRAMs as their ASTRON core equivalents, but used less of all other resources - on a scale that is either equal to, or slightly less than, that of the CASPER core. The ASTRON core is the least resource efficient, with a worst case BRAM usage double that of the CASPER core’s usage.

Power requirements reported by Vivado for each of these test designs are somewhat perplexing, as the ASTRON core generally reported the lowest power requirements despite having the greatest resource usage. A hardware compile where the second input stream into the ASTRON core was set to process non-zero values confirmed that this lower power requirement is not due to some optimisation within the core causing it to not utilise the logic for the second stream if the input it set to zero, as the same power value was reported for a non-zero input. Hardware compiles for equivalent ASTRON and CASPER designs with a clock rate reduced from 160MHz to 100MHz confirmed that power requirements scale with clock rate, but with ASTRON still reporting a lower power requirement than the CASPER core. A thorough analysis of the floorplan, routing and exact resource types used for every core’s configuration would be required to determine the reason behind this; a task that was deemed beyond the scope of this project.

Both Xilinx and CASPER cores are able to achieve clock rates between 450 - 600MHz for $L = 4$ on newer hardware, with Xilinx’s clock rate performance only declining for the largest FFT length for $L = 8$ as path routing becomes more complicated. ASTRON’s clock rate is limited by a reset line fan-out issue, and in some cases did not meet timing for the lowest tested clock rate of 200MHz.

In general, CASPER has been proven to be either equal or superior to the other cores based on numerical, resource utilisation and clock rate performance test results when using its default configuration options, and so testing ceased at this point. The smaller tests that were performed using improved shifting schedules and an even rounding scheme indicate that it might be possible to improve CASPER’s performance further. While Xilinx already performs well numerically, it might be possible to make changes to its surrounding logic that increase its hardware resource efficiency, making it comparable to CASPER’s default optimised state. ASTRON’s implementation could also be modified to improve its performance and resource utilisation. The following section suggests some proposed modifications and outlines additional work that could be performed to test each core even further.

8.3 Suggestions for Further Work

The first suggestion to improve the performance of the ASTRON core would be to implement the use of an even rounding scheme - the results of the simulation test for the CASPER core using an even rounding scheme indicate a great improvement in numerical performance when compared to its equivalent infinite rounding tests.

Modifying the ASTRON core to utilise a biplex architecture in the same manner as the CASPER core would improve resource utilisation and order the spectral outputs so the outputs of multiple input streams could be accessed at the same clock cycle.

Decoding logic could be applied to the Xilinx core's output to create a real-to-complex version that uses Hermitian symmetry to calculate the FFT of two real inputs using a single complex core. This could be taken further, with the implementation of a biplex architecture that would result in a Xilinx core fully comparable to the CASPER core.

To further analyse the results presented in Chapter 7, an in-depth investigation into the reported power requirements of each core could be performed. As has already been mentioned, this would require a detailed review of each core's implemented design to determine why the estimator reported the requirements that it did - especially regarding the ASTRON core, where the estimator reported such low power requirements for the greatest amount of resources used.

Further tests of the CASPER core specifically could be performed to investigate by how much further its numerical performance could be improved. The core could be tested using bit growth between stages enabled, rather than requiring downshifting, or twiddle factor coefficients could be allocated more bits beyond their 18-bit default. A combination of both, with the addition of an even rounding scheme may provide a test of the CASPER core in its optimal state.

Finally, Figures 2.1 and 2.2 in Chapter 2 illustrate that the addition of a hamming-windowed FIR filterbank before an FFT greatly improves frequency channel response. To further test the performance of each FFT core, a PFB could be created using each core, with their resulting frequency channel shapes and overall performance compared. This would provide performance results specifically of interest to the radio astronomy community and, when coupled with the results of this report, could provide good insight as to which FFT core is the best to use when developing radio astronomy correlator instruments.

References

- [1] Danny C. Price. “Spectrometers and Polyphase Filterbanks in Radio Astronomy”. In: (July 2016). URL: <http://arxiv.org/abs/1607.03579>.
- [2] D. C. Champeney. “Power spectra and Wiener’s theorems”. In: *A handbook of Fourier Theorems*. First. Cambridgeshire: Cambridge University Press, 1987, pp. 102–117.
- [3] *MeerKAT*. URL: <https://science.ska.ac.za/meerkat>.
- [4] Andrew van der Byl et al. “MeerKAT correlator-beamformer: a real-time processing back-end for astronomical observations”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 8.01 (2021). ISSN: 23294221. DOI: 10.1117/1.jatis.8.1.011006.
- [5] Morag Brown. *brwmor001_msc_project*. 2023. URL: https://github.com/moragb96/brwmor001_msc_project.
- [6] *LEDA (Large-Aperture Experiment to Detect the Dark Ages)*. URL: <http://www.tauceti.caltech.edu/leda/index.php/technology>.
- [7] *HERA (Hydrogen Epoch of Reionisation Array)*. URL: <https://www.sarao.ac.za/science/hera/>.
- [8] *CHIME (The Canadian Hydrogen Intensity Mapping Experiment)*. URL: <https://chime-experiment.ca/en>.
- [9] Danny C Price. *PFB Introduction*. 2020. URL: https://github.com/telegraphic/pfb_introduction.
- [10] Steven W Smith. *The Scientist and Engineer’s Guide to Digital Signal Processing*. 1st. San Diego: California Technical Publishing, 1997. ISBN: 0-9660176-3-3.
- [11] James W. Cooley and John W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19.90 (1965), p. 297. ISSN: 00255718. DOI: 10.2307/2003354.
- [12] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Second. New Jersey: Prentice-Hall, Inc., 1999.
- [13] Bruno Paillard. *An Introduction To Digital Signal Processors INTRODUCTION 5 THE TMS320C5402: ARCHITECTURE AND PROGRAMMING 91*. Tech. rep. Université de Sherbrooke, 2002.
- [14] Cem Ünsalan and Bora Tar. *Digital System Design with FPGA Implementation Using Verilog and VHDL*. McGraw-Hill Education, 1389. ISBN: 9781259837913.
- [15] Scott Thornton. *Fixed point vs Floating point*. URL: <https://www.microcontrollertips.com/difference-between-fixed-and-floating-point/>.
- [16] Xilinx. “Reduce Power and Cost by Converting from Floating Point to Fixed Point Introduction : Xilinx Data Type Support”. In: 491 (2017).
- [17] Randy Yates. “Fixed-point arithmetic: An introduction. Technical Reference”. In: *Digital Signal Labs* 81 (2007). URL: <https://courses.cs.washington.edu/courses/cse467/08au/labs/15/fp.pdf>.
- [18] “Xilinx Model Composer - Managing Overflow”. In: (). URL: <https://docs.xilinx.com/r/en-US/ug1483-model-composer-sys-gen-user-guide/Managing-Overflow>.

- [19] Xilinx Inc. “Vivado Design Suite - Model-Based DSP Design Using System Generator”. In: *Ug958* (2019), pp. 1–174. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.
- [20] “Xilinx Model Composer - Overflow and Quantisation”. In: (). URL: <https://docs.xilinx.com/r/en-US/ug1483-model-composer-sys-gen-user-guide/Common-Options-in-Block-Parameter-Dialog-Boxes>.
- [21] Mark Zwoliński. *Digital System Design with VHDL*. Second Edi. Harlow: Pearson Education, 2004. ISBN: 978-0130399854.
- [22] M. Morris Mano and Michael D. Ciletti. *Digital design : with an introduction to the verilog hdl*. New Jersey: Pearson Education as Prentice Hall, 2013, p. 547. ISBN: 9780132774208. URL: <http://www.mypearsonstore.com/bookstore/digital-design-9780132774208>.
- [23] R. C. Cofer and Benjamin F. Harding. *Rapid System Prototyping with FPGAs*. Burlington: Elsevier, 2006. ISBN: 0-7506-7866-6.
- [24] Xilinx. “UG1301: Getting Started with Alveo Data Center Accelerator Cards”. In: UG1301 (2019), pp. 1–55.
- [25] Xilinx. “Model Composer and System Generator User Guide”. In: 1483 (2020).
- [26] Xilinx. “Vitis Unified Software Platform Documentation Application Acceleration Development”. In: 1393 (2020), pp. 1–580. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1393-vitis-application-acceleration.pdf.
- [27] *About the Collaboration*. URL: <https://casper.berkeley.edu/index.php/about/>.
- [28] *CASPER Toolflow*. 2022. URL: <https://casper-toolflow.readthedocs.io/en/latest/>.
- [29] Michael Dreschmann et al. “Implementation of an ultra-high speed 256-point FFT for Xilinx Virtex-6 devices”. In: *IEEE International Conference on Industrial Informatics (INDIN)* (2011), pp. 829–834. ISSN: 19354576. DOI: 10.1109/INDIN.2011.6035000.
- [30] Chen Yang et al. “Design of a configurable fixed-point FFT processor”. In: *IET Conference Publications* 2015.CP677 (2015), pp. 1–4. DOI: 10.1049/cp.2015.1307.
- [31] Hans Kanders et al. “A 1 Million-Point FFT on a Single FPGA”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.10 (2019), pp. 3863–3873. ISSN: 15580806. DOI: 10.1109/TCSI.2019.2918403.
- [32] Carl Ingemarsson and Oscar Gustafsson. “SFF—The Single-Stream FPGA-Optimized Feedforward FFT Hardware Architecture”. In: *Journal of Signal Processing Systems* 90.11 (2018), pp. 1583–1592. ISSN: 19398115. DOI: 10.1007/s11265-018-1370-y.
- [33] Klein, B. et al. “High-resolution wide-band fast Fourier transform spectrometers”. In: *A&A* 542 (2012), p. L3. DOI: 10.1051/0004-6361/201218864. URL: <https://doi.org/10.1051/0004-6361/201218864>.
- [34] *ASTRON*. URL: <https://www.astron.nl/>.
- [35] *WSRT-APERTIF - ASTRON*. URL: <https://www.astron.nl/telescopes/wsrt-apertif/>.
- [36] J Shima. “Computing the FFT of Two Real Signals Using a Single FFT”. In: 1.0 (2000), pp. 0–3.
- [37] R.F. Emerson and Pasadena Jet Propulsion Lab. California Inst. of Tech. “Biplex, pipelined FFT”. In: 0.7 (1976). URL: http://tmo.jpl.nasa.gov/progress_report2/42-34/34I.PDF.
- [38] Ryan Monroe. “Improving the Performance and Resource Utilization of the CASPER FFT and Polyphase Filterbank”. In: *Journal of Astronomical Instrumentation* 5.4 (2016), pp. 1–9. ISSN: 22511725. DOI: 10.1142/S2251171716410026.
- [39] Harm Jan Pepping. “UniBoard Wideband-FFT Module Description UniBoard Distribution list : Document history : UniBoard”. In: (2011).
- [40] Xilinx. “Fast Fourier Transform v9.1”. In: (2019).

- [41] Xilinx. *Performance and Resource Utilization for Fast Fourier Transform v9.1*. URL: https://www.xilinx.com/support/documentation/ip_documentation/ru/xfft.html.
- [42] Intel. “FFT IP Core User Guide”. In: (2017).
- [43] Jérôme Leclère, Cyril Botteron, and Pierre-andré Farine. “Implementing super-efficient FFTs in Altera FPGAs”. In: *EE Times Programmable Logic Designline*, 2015. Available online at www.eetimes.com. August 2014 (2015), pp. 1–14.
- [44] C. Gonzalez-Concejero et al. “An FFT/IFFT design versus Altera and Xilinx cores”. In: *Proceedings - 2008 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2008* (2008), pp. 337–342. DOI: 10.1109/ReConFig.2008.65.
- [45] David Luebke and Greg Humphreys. “How GPUs work”. In: *Computer* 40.2 (2007), pp. 96–100. ISSN: 00189162. DOI: 10.1109/MC.2007.59.
- [46] Kenneth Moreland and Edward Ange. “The FFT on a GPU”. In: *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2003), pp. 112–119.
- [47] NVIDIA Corporation. “Cufft Library User’s Guide”. In: *NVIDIA Corporation, Santa Clara, California* 5.5.5.5 (2013). URL: <https://developer.nvidia.com/cuda-toolkit-55-archive>.
- [48] Xiang Cui, Yifeng Chen, and Hong Mei. “Improving performance of matrix multiplication and FFT on GPU”. In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* (2009), pp. 42–48. ISSN: 15219097. DOI: 10.1109/ICPADS.2009.8.
- [49] Zhicheng Zhao and Yaquin Zhao. “The Optimization of FFT Algorithm Based with Parallel Computing on GPU”. In: *Proceedings of 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference, IAEAC 2018* Iaeac (2018), pp. 2003–2007. DOI: 10.1109/IAEAC.2018.8577843.
- [50] *clFFT: OpenCL Fast Fourier Transforms (FFTs)*. URL: <http://clmathlibraries.github.io/clFFT/>.
- [51] *FFTW Home Page*. URL: <http://www.fftw.org/>.
- [52] *FFT Benchmark Results*. URL: <http://www.fftw.org/speed/>.
- [53] *Fast Fourier transform - MATLAB fft*. URL: <https://www.mathworks.com/help/matlab/ref/fft.html>.
- [54] *Discrete Fourier Transform (numpy.fft) — NumPy v1.21 Manual*. URL: <https://numpy.org/doc/stable/reference/routines.fft.html>.
- [55] William C. Elm et al. “Integrating Cognitive Systems Engineering Throughout the Systems Engineering Process”. In: *Journal of Cognitive Engineering and Decision Making* 2.3 (2008), pp. 249–273. ISSN: 15553434. DOI: 10.1518/155534308X377108.
- [56] M Brown. *Support for Xilinx SSR FFT on non-ultrascale+ devices?* 2022. URL: https://support.xilinx.com/s/question/0D54U00005Seck4SAB/support-for-xilinx-ssr-fft-on-nonultrascale-devices?language=en_US.
- [57] *CASPER mlib_devel*. URL: https://github.com/casper-astro/mlib_devel.
- [58] *casperfpga*. URL: <https://casper-toolflow.readthedocs.io/projects/casperfpga/en/latest/>.
- [59] *Alveo U50 Data Center Accelerator Card*. URL: <https://www.xilinx.com/products/boards-and-kits/alveo/u50.html>.
- [60] *casper_dspdevel*. URL: https://github.com/talonmyburgh/casper_dspdevel.
- [61] *Dan’s Quotes on Scaling*. URL: https://casper.astro.berkeley.edu/wiki/Dan%27s_Quotes_on_Scaling.

Appendices

A.1 AU50: ASTRON 1k, L = 4 Simulink Design



A.2 AU50: ASTRON 1k, L = 8 Simulink Design

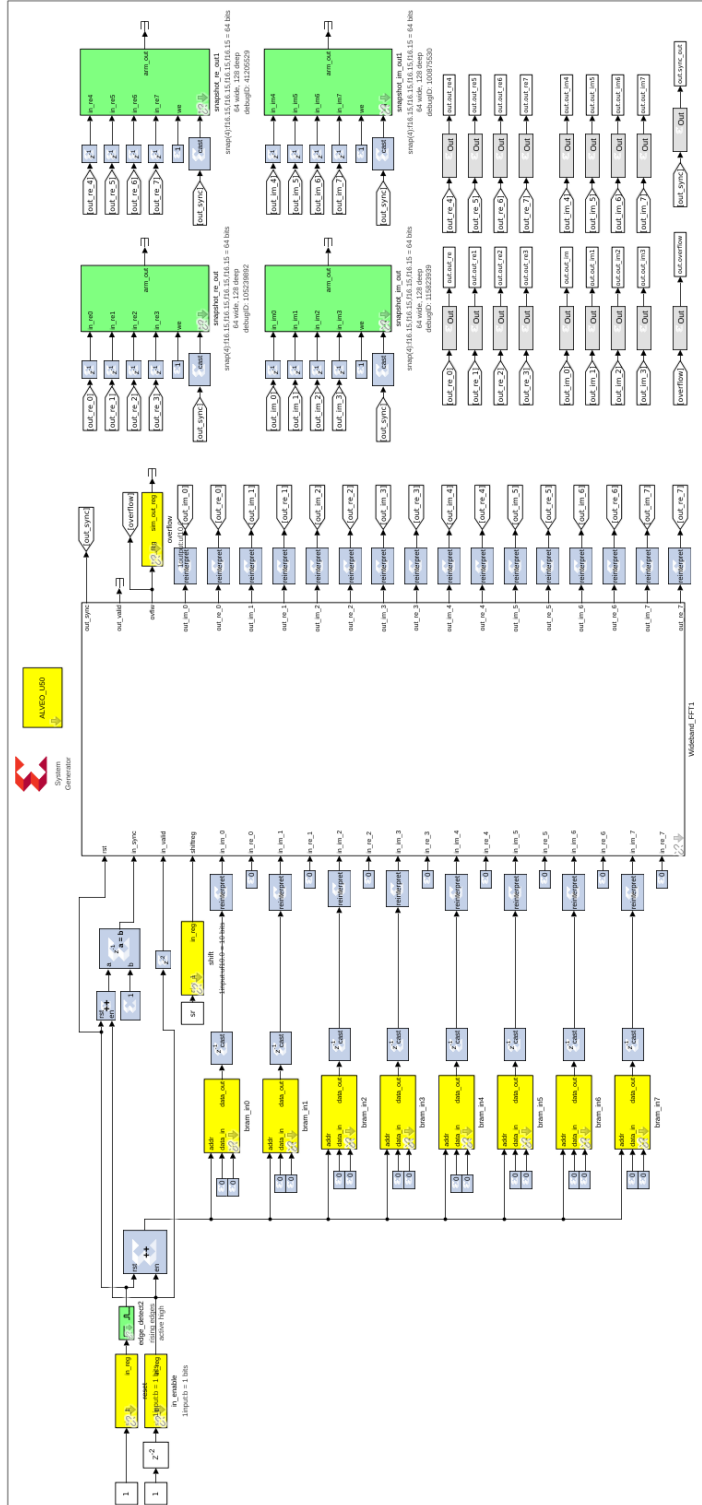


Figure A.2: astron_hw_test_au50_1k_8.in

A.3 SKARAB: ASTRON 1k, L = 4 Simulink Design

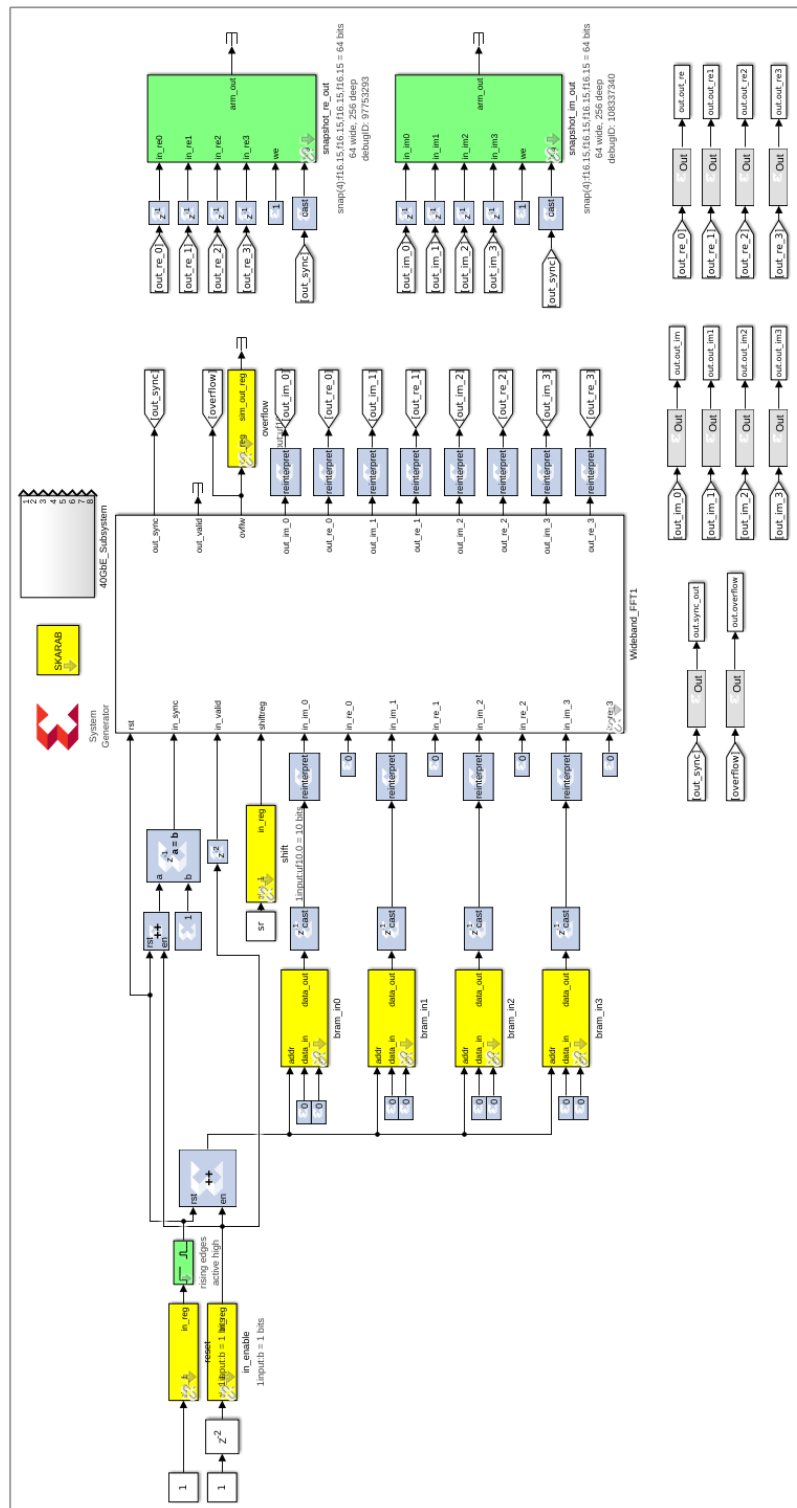


Figure A.3: astron_hw_test_skarab_1k

A.4 SKARAB: ASTRON 1k, L = 8 Simulink Design

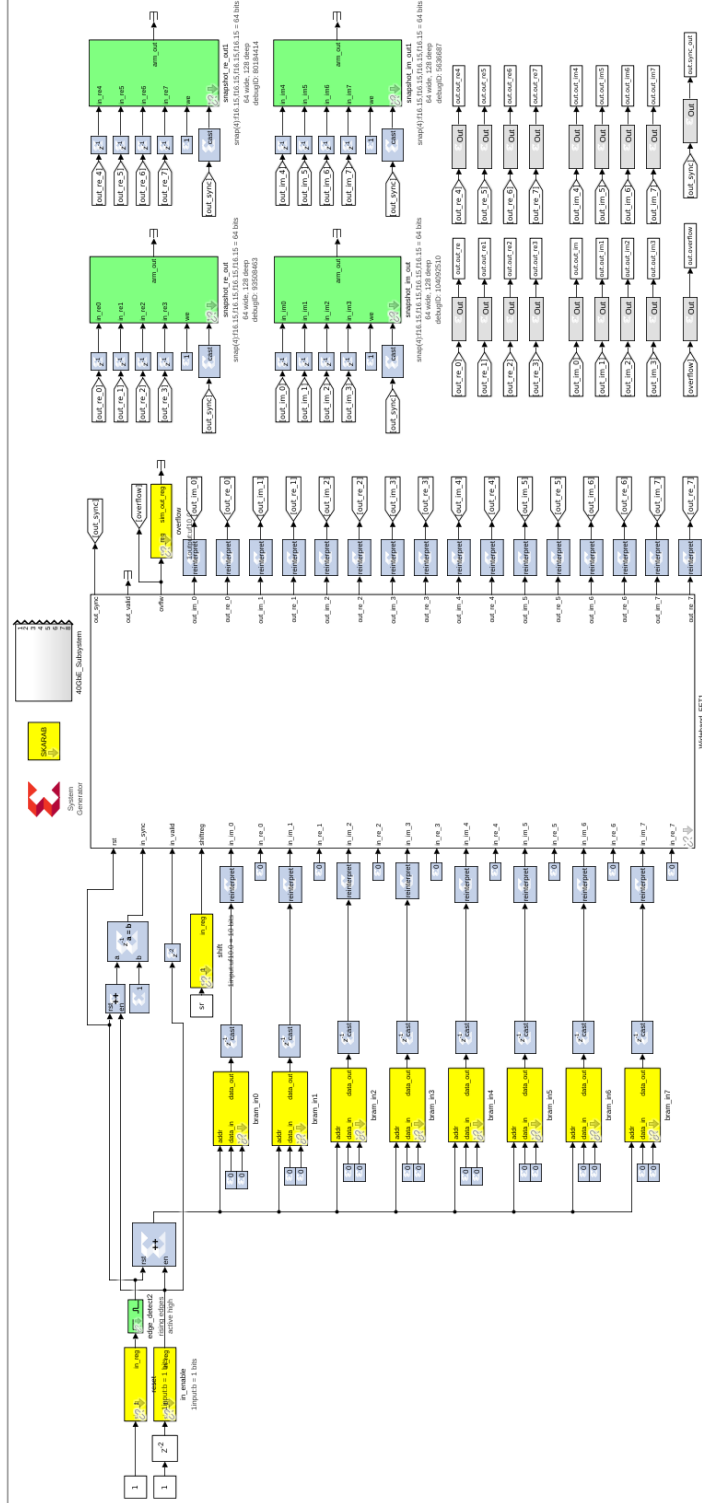


Figure A.4: astron_hw_test_skarab_1k_8in

A.5 AU50: CASPER 1k, L = 4 Simulink Design

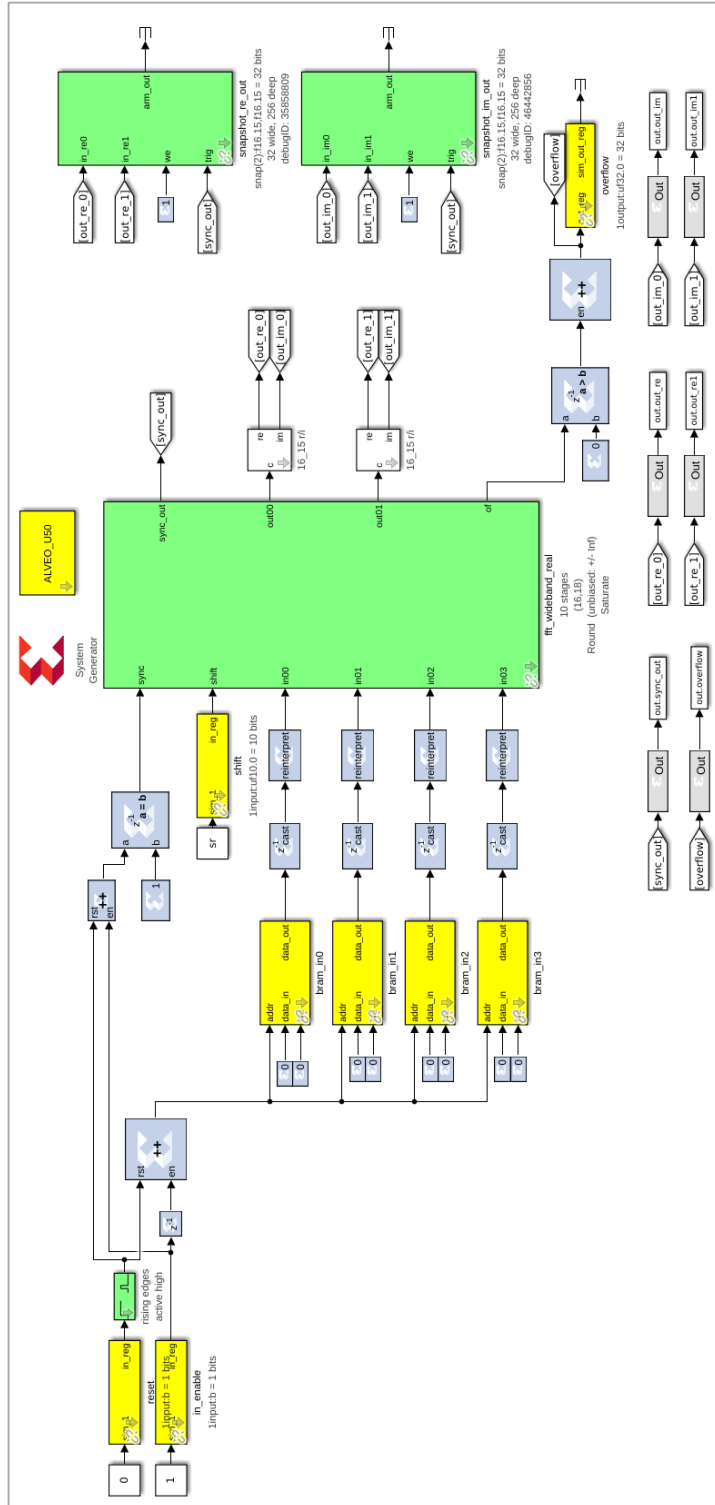


Figure A.5: casper_hw_test_au50_1k

A.6 AU50: CASPER 1k, L = 8 Simulink Design

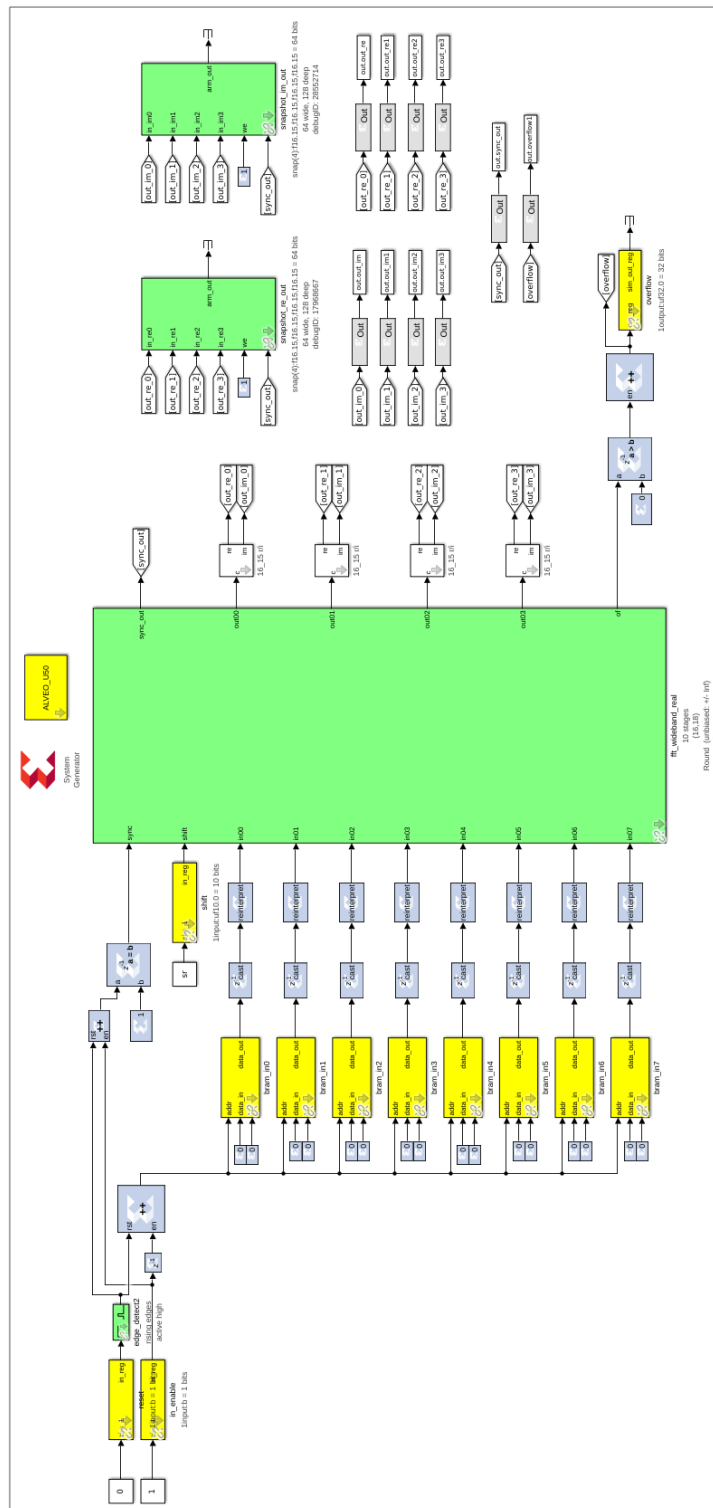


Figure A.6: casper_hw_test_au50_1k_8in

A.7 SKARAB: CASPER 1k, L = 4 Simulink Design

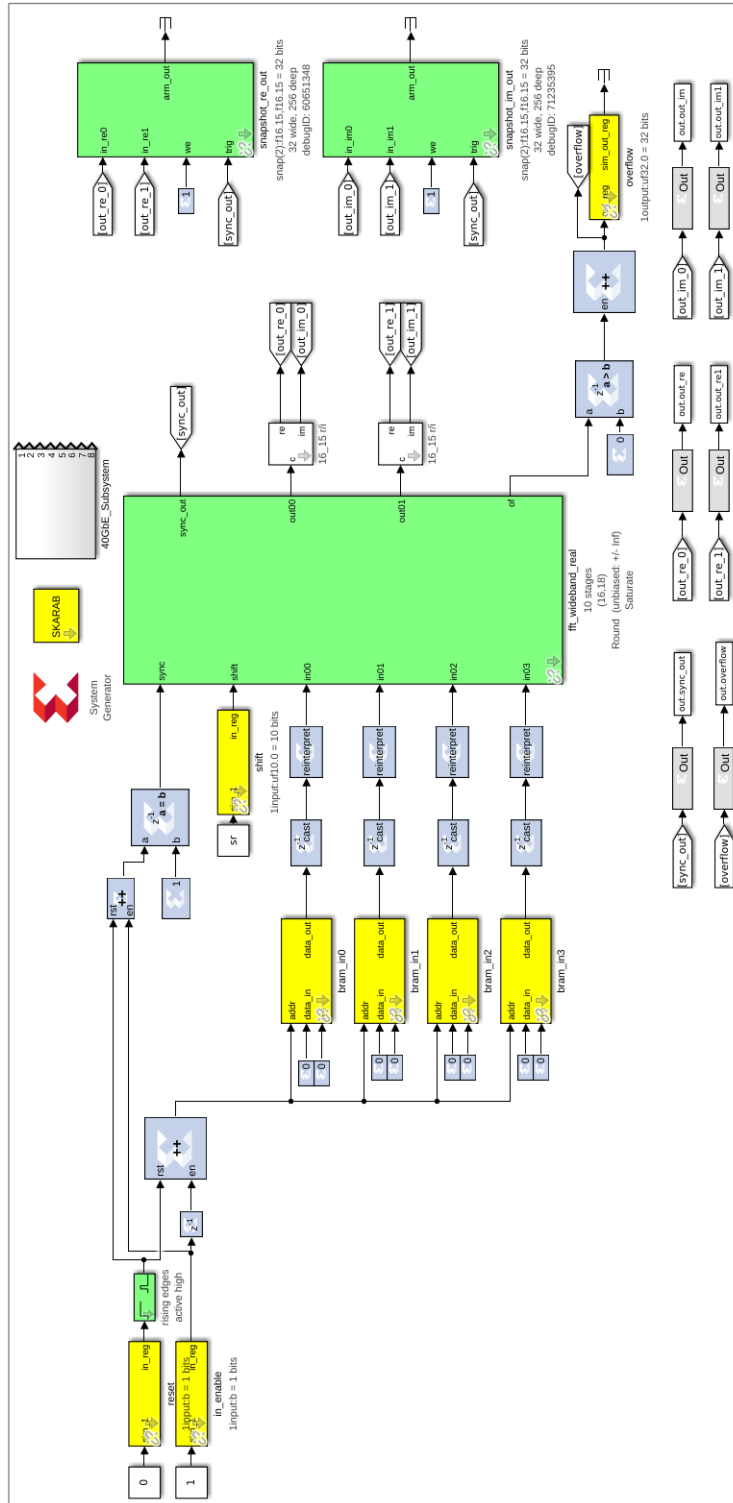


Figure A.7: casper_hw_test_skarab_1k

A.8 SKARAB: CASPER 1k, L = 8 Simulink Design

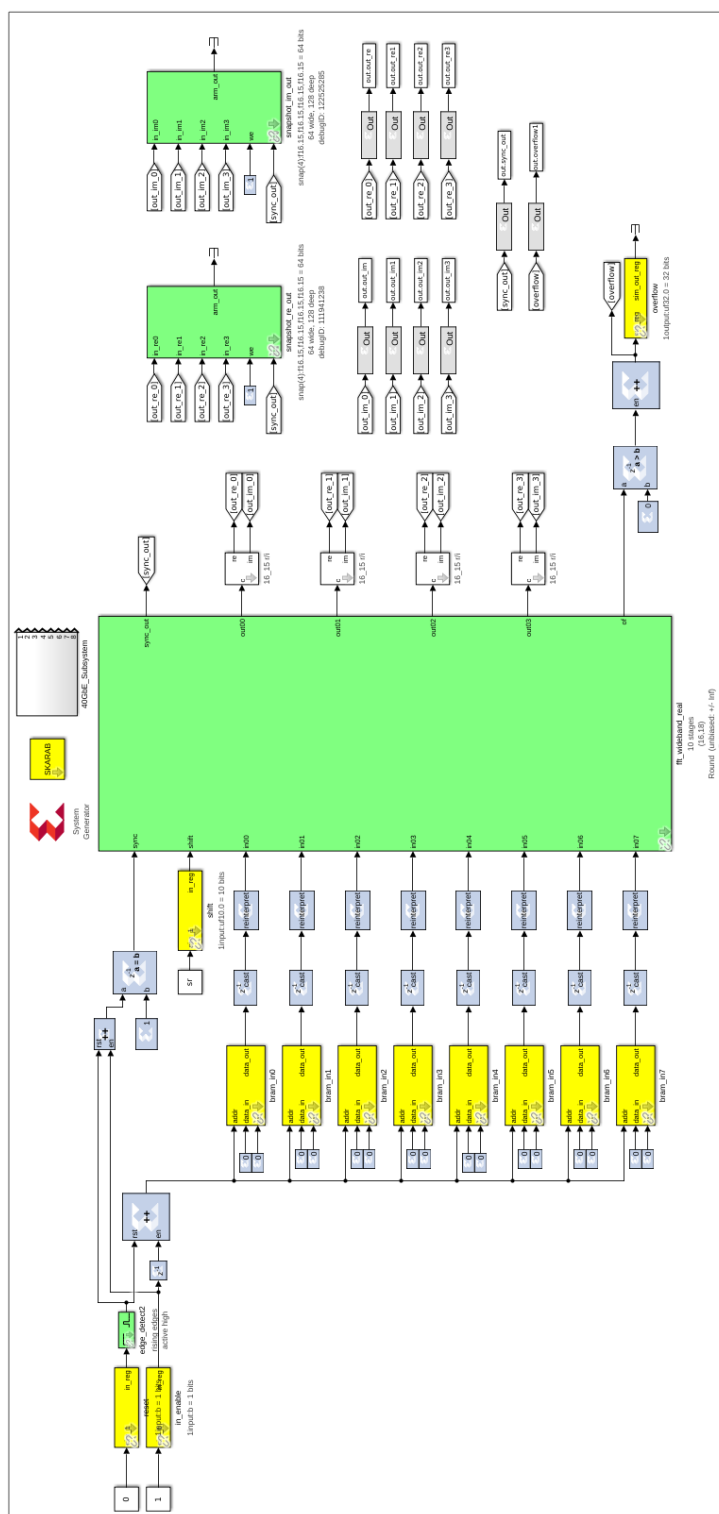


Figure A.8: casper_hw_test_skabab_1k_8in

A.9 AU50: Xilinx 1k, L = 4 Simulink Design

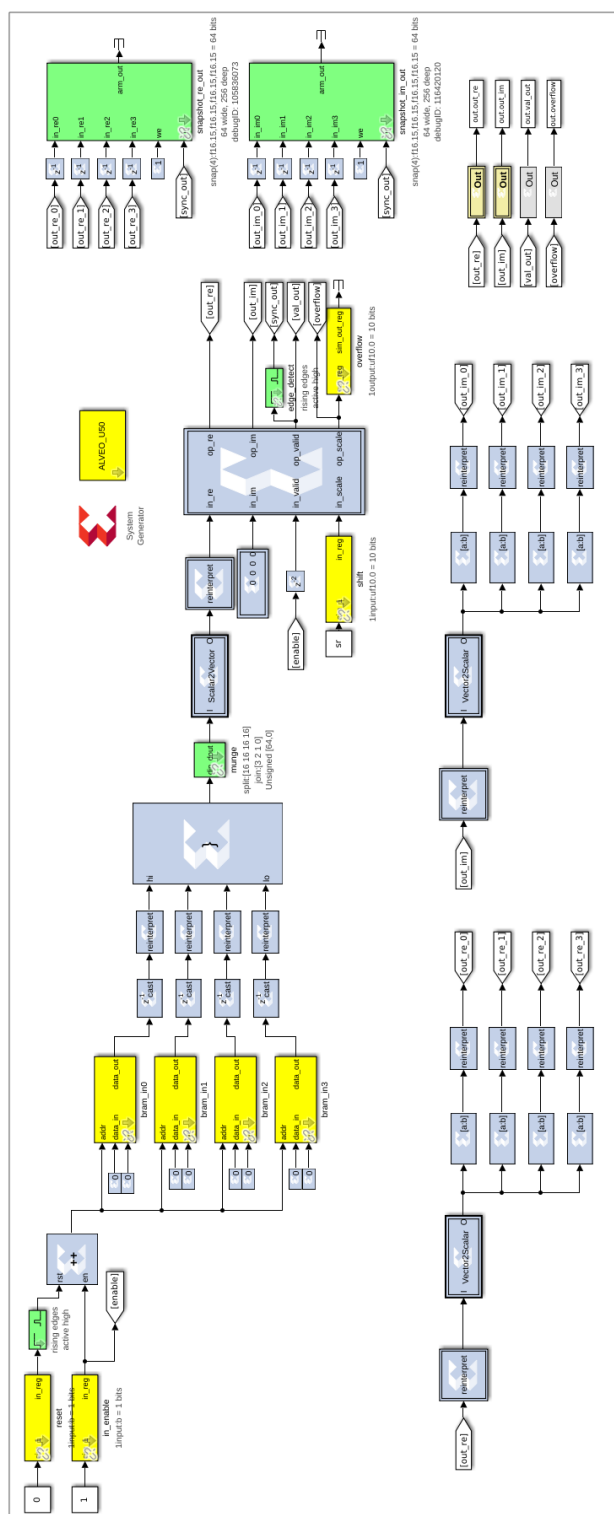


Figure A.9: xilinx_hw_test_au50_1k

A.10 AU50: Xilinx 1k, L = 8 Simulink Design

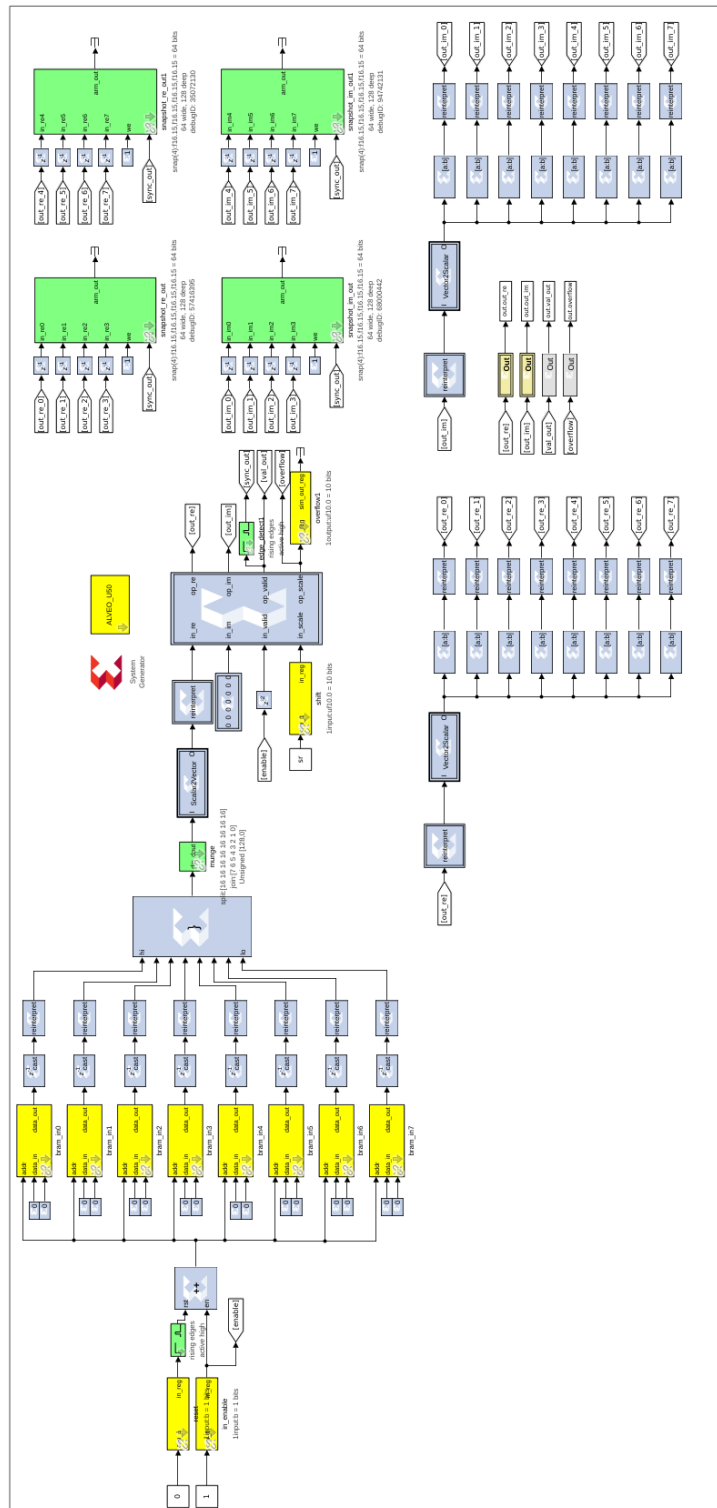


Figure A.10: xilinx_hw_test_au50_1k_8in

Appendix B

B.1 TVG0 - FFTW PSDs

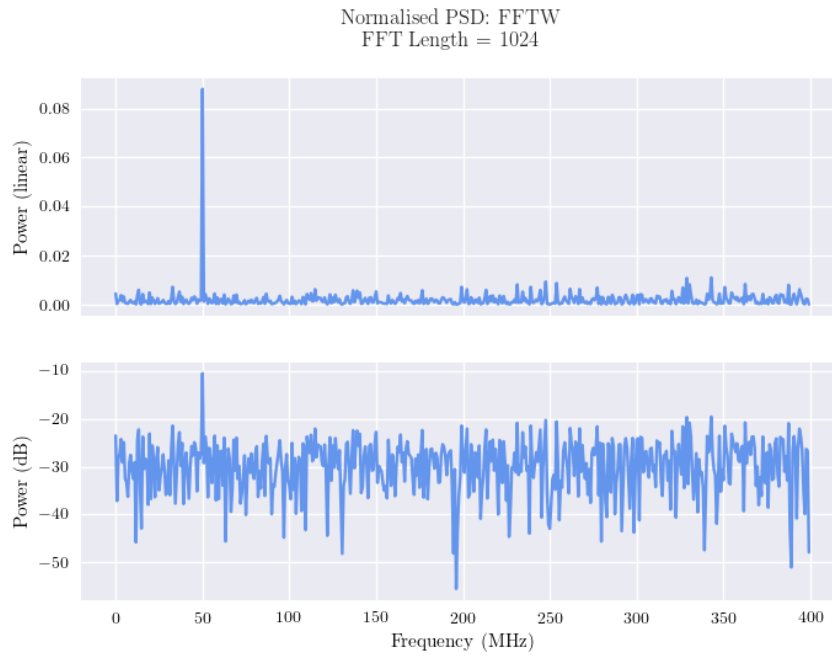


Figure B.1: Normalised PSD of FFTW, $N = 1024$.

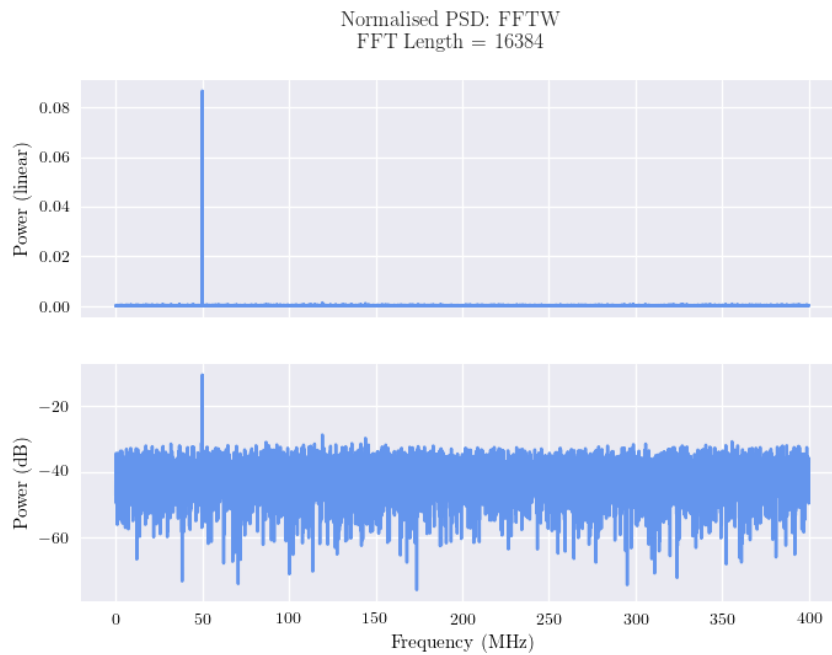


Figure B.2: Normalised PSD of FFTW, $N = 16384$.

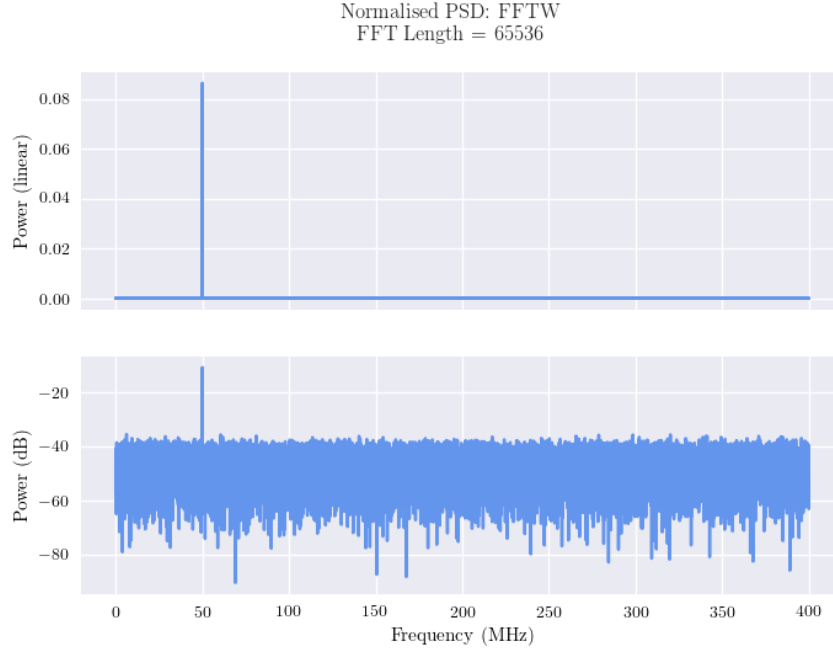


Figure B.3: Normalised PSD of FFTW, $N = 65536$.

B.2 TVG0 - Improved PSDs

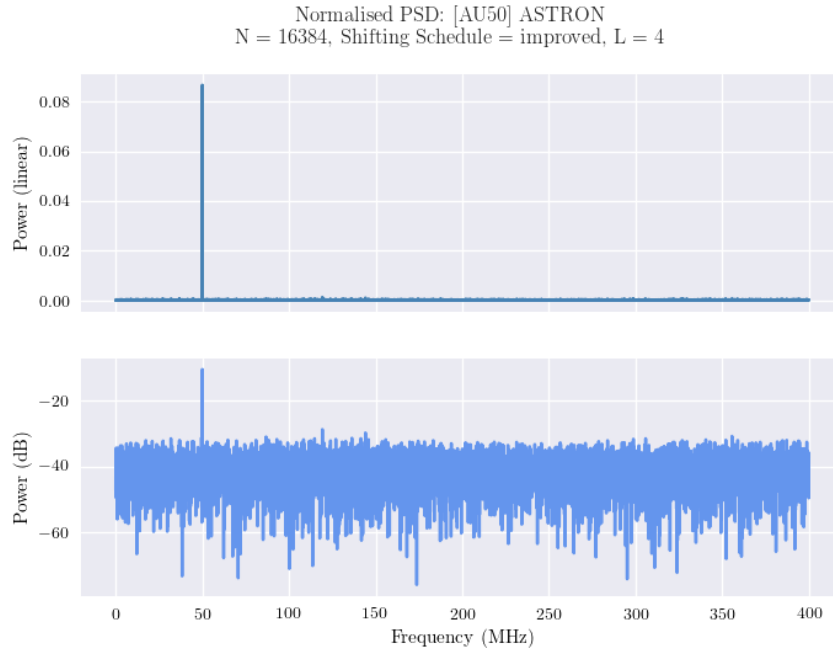


Figure B.4: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shifting = improved.

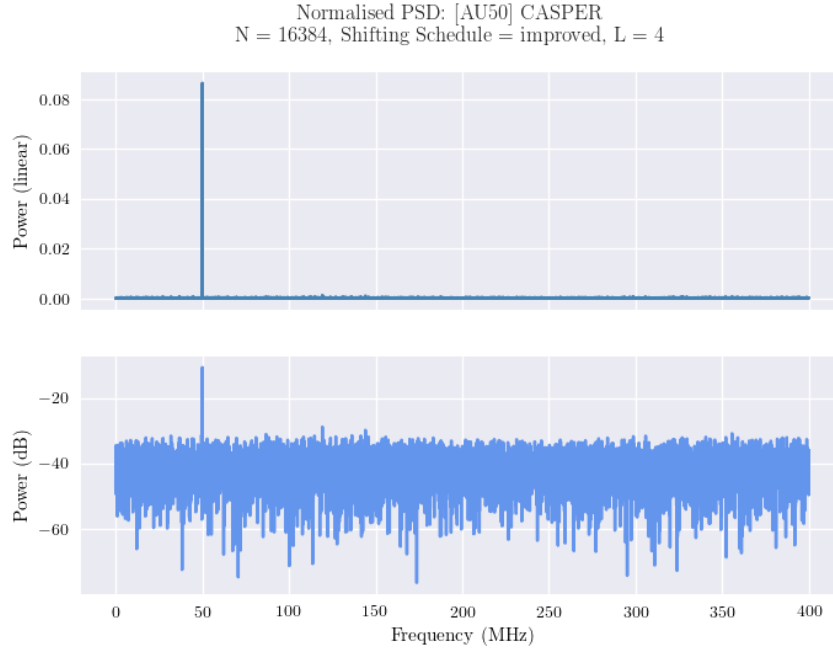


Figure B.5: Normalised PSD of CASPER, N = 16384, L = 4, Shifting = improved.

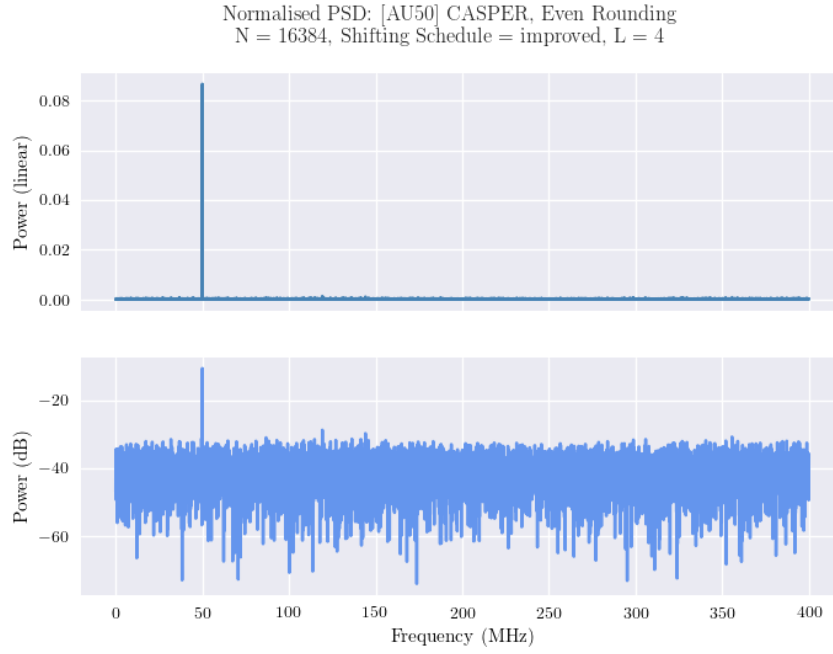


Figure B.6: Normalised PSD of CASPER, N = 16384, L = 4, Shifting = improved, Rounding = even.

B.3 TVG0 - Hardware PSDs

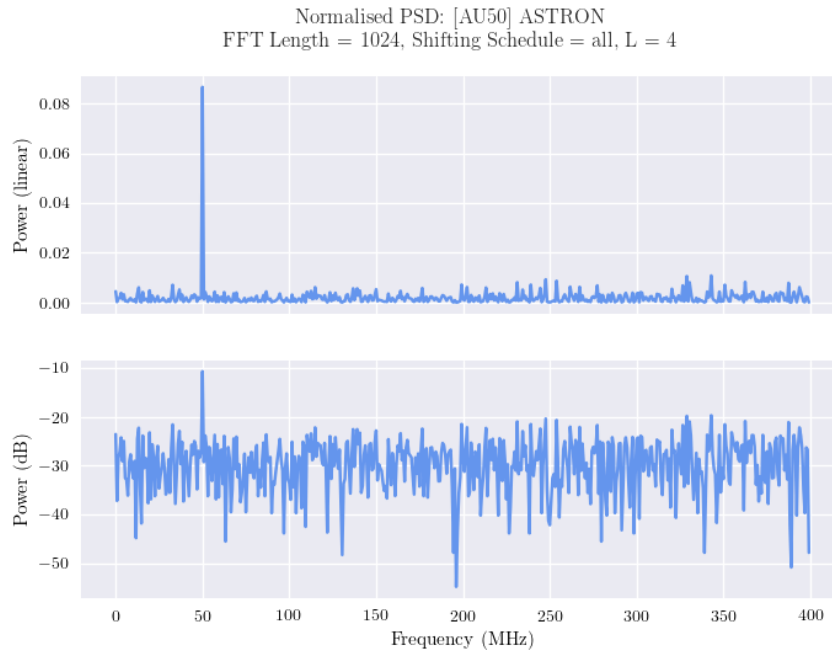


Figure B.7: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.

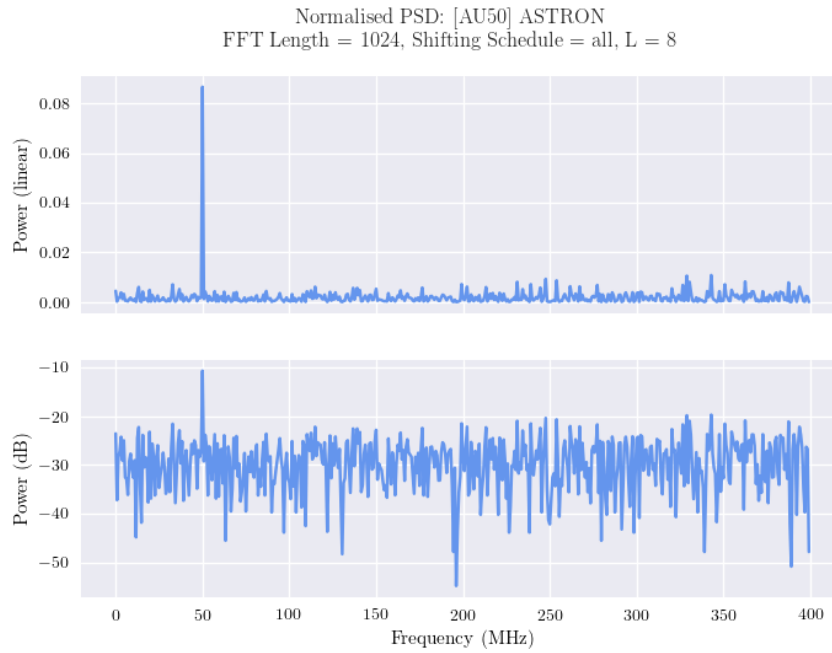


Figure B.8: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.

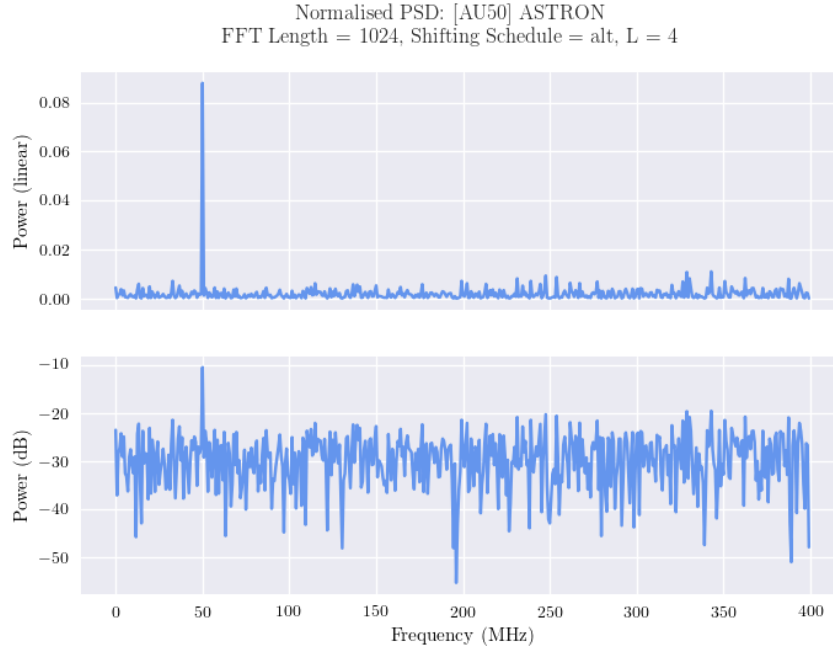


Figure B.9: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = alt.

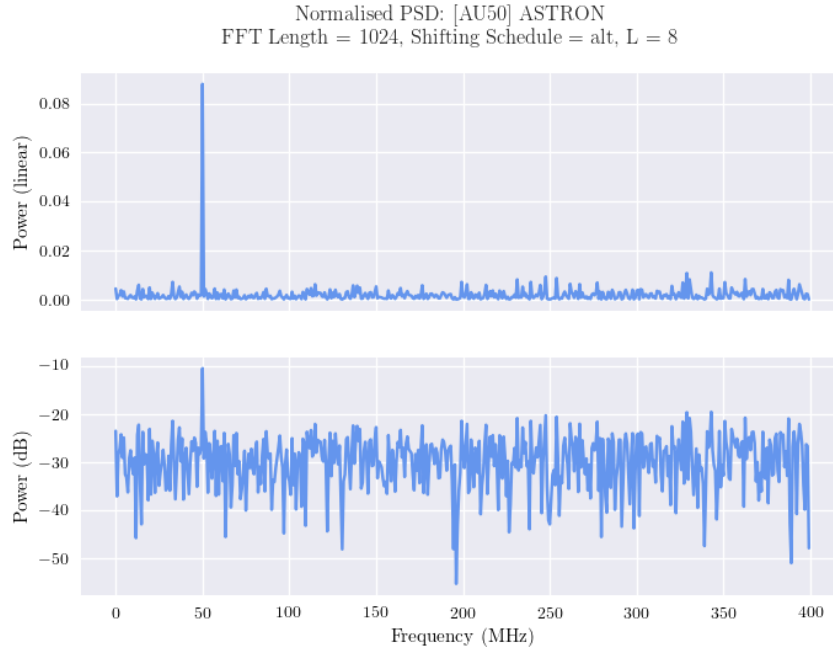


Figure B.10: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = alt.

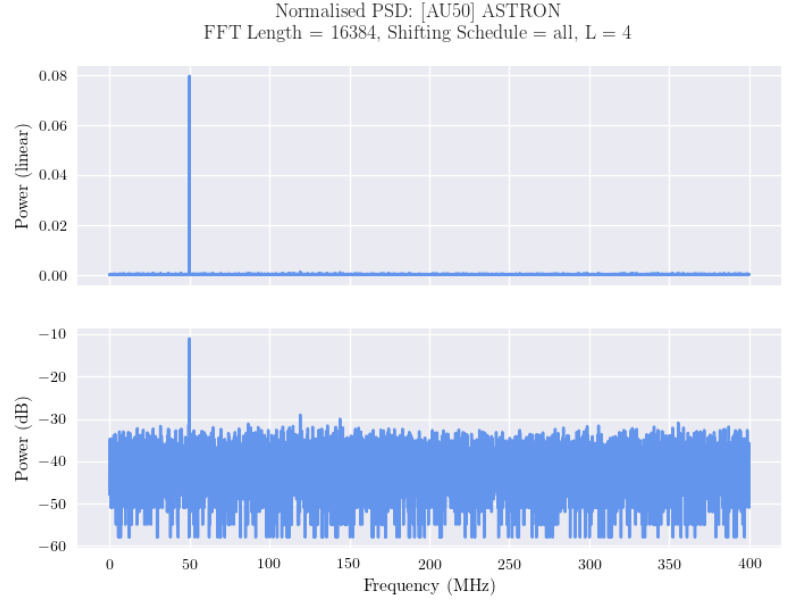


Figure B.11: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.

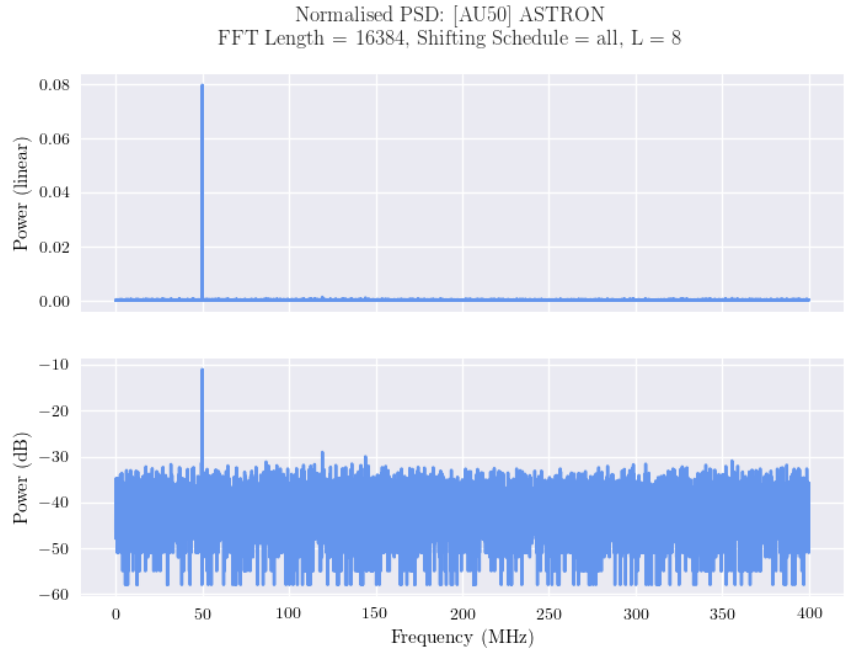


Figure B.12: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.

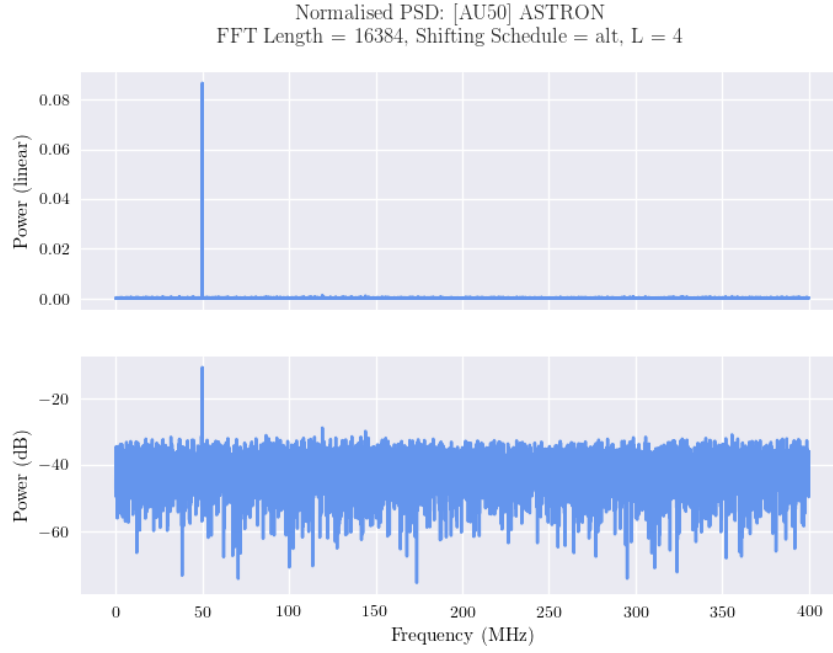


Figure B.13: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = alt.

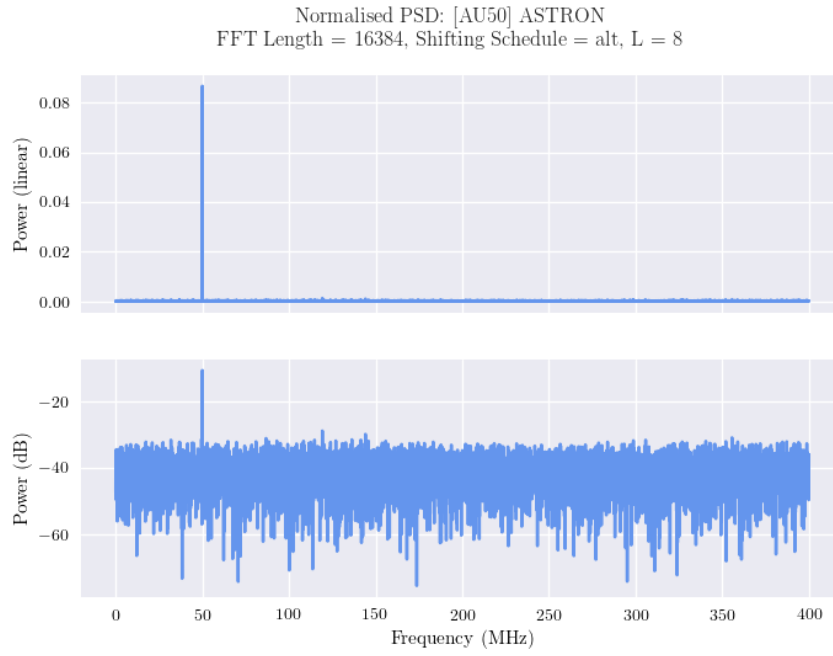


Figure B.14: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = alt.

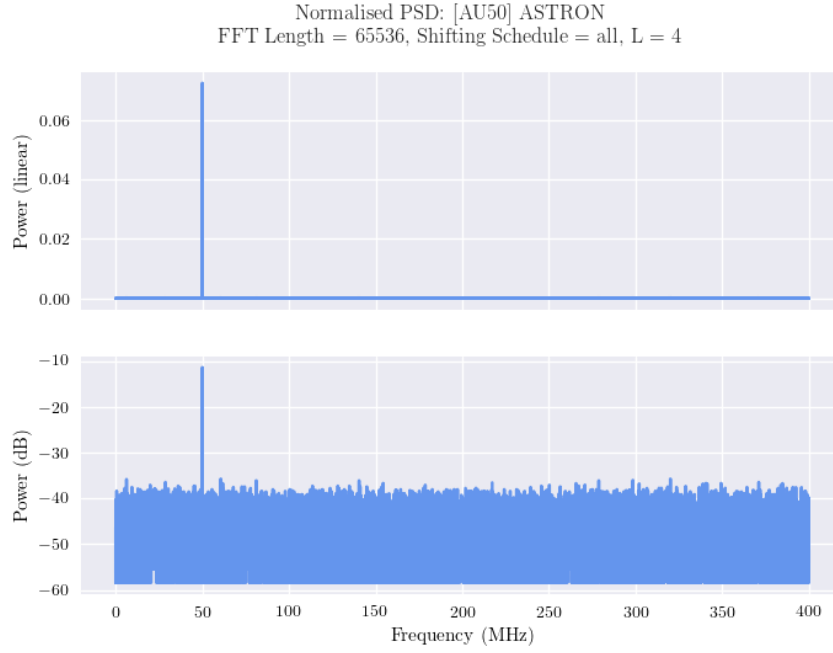


Figure B.15: Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.

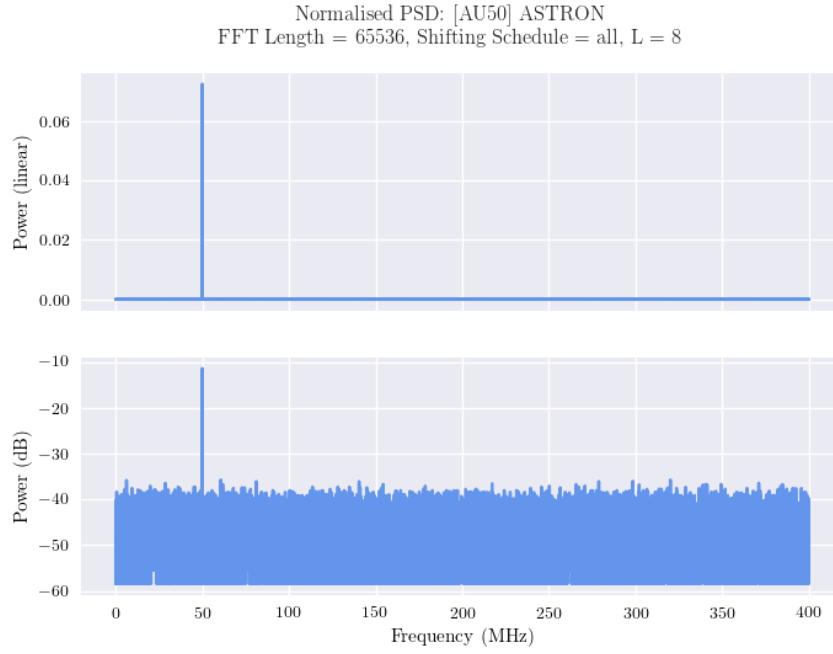


Figure B.16: Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.

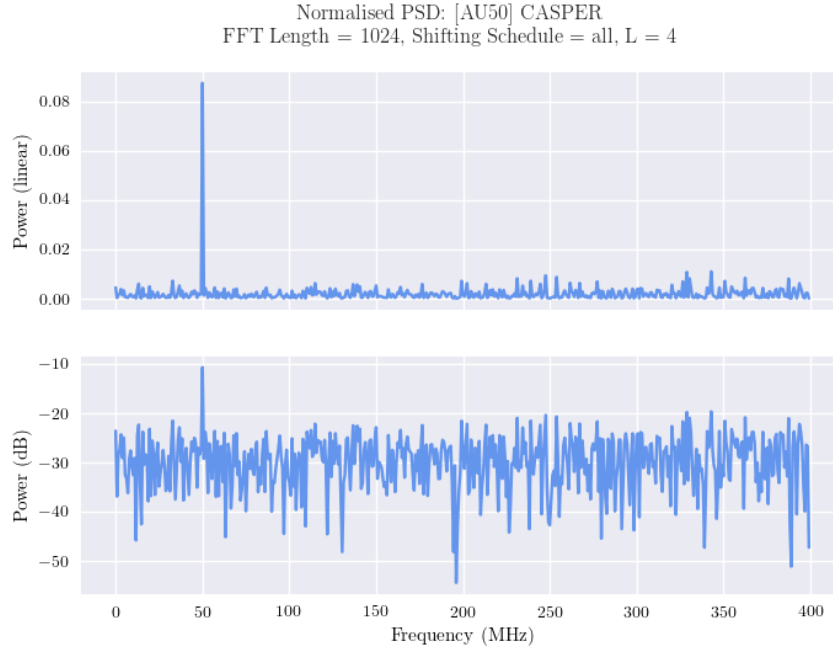


Figure B.17: Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.

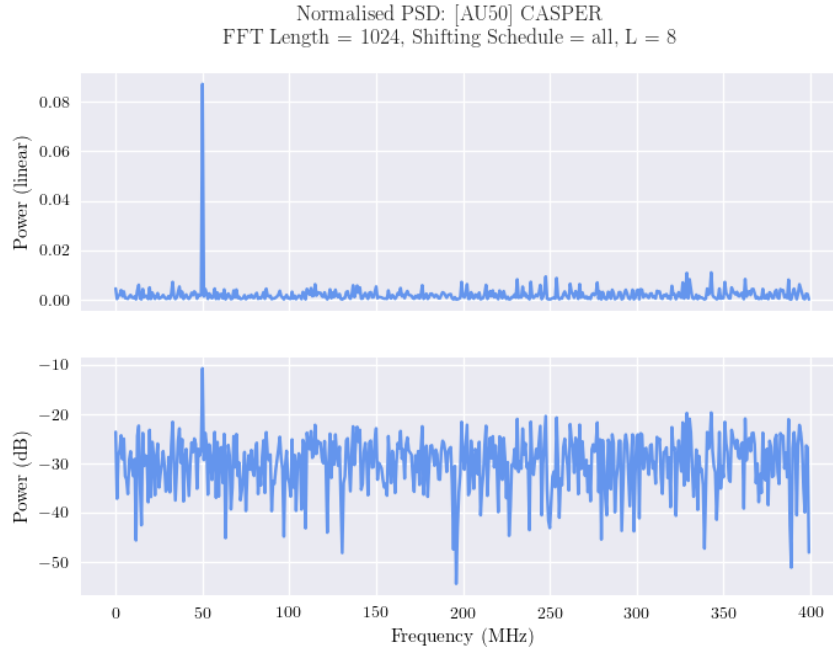


Figure B.18: Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.

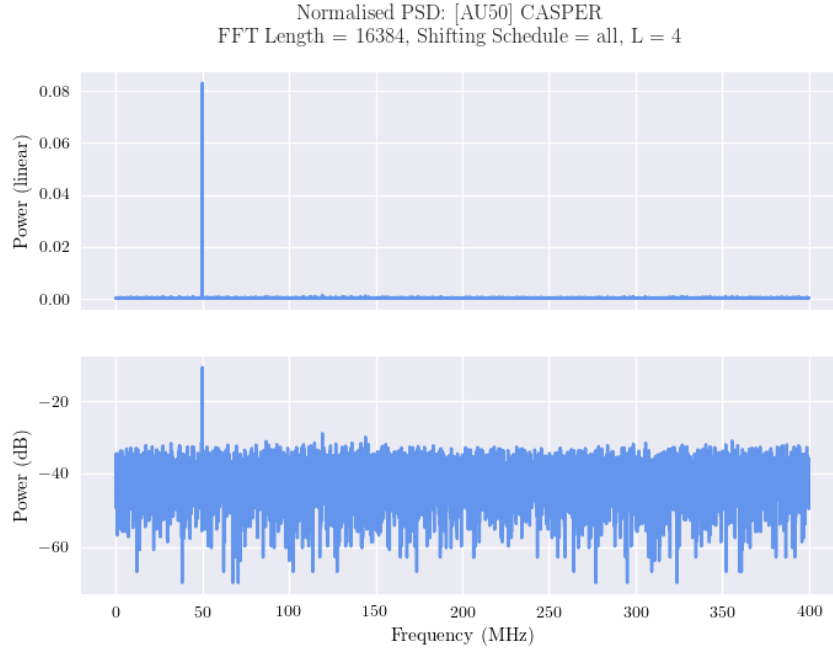


Figure B.19: Normalised PSD of CASPER, N = 16384, L = 4, Shift = all.

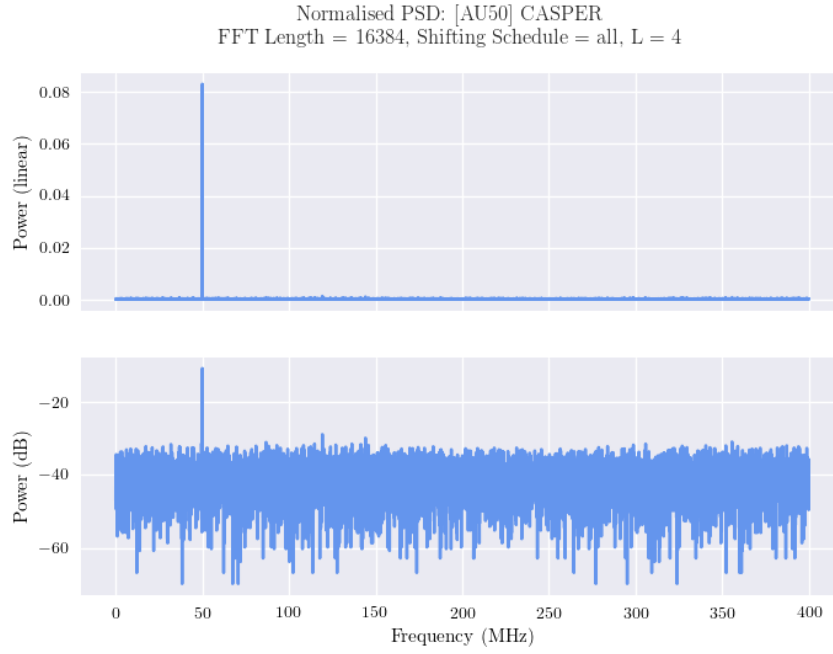


Figure B.20: Normalised PSD of CASPER, N = 16384, L = 8, Shift = all.

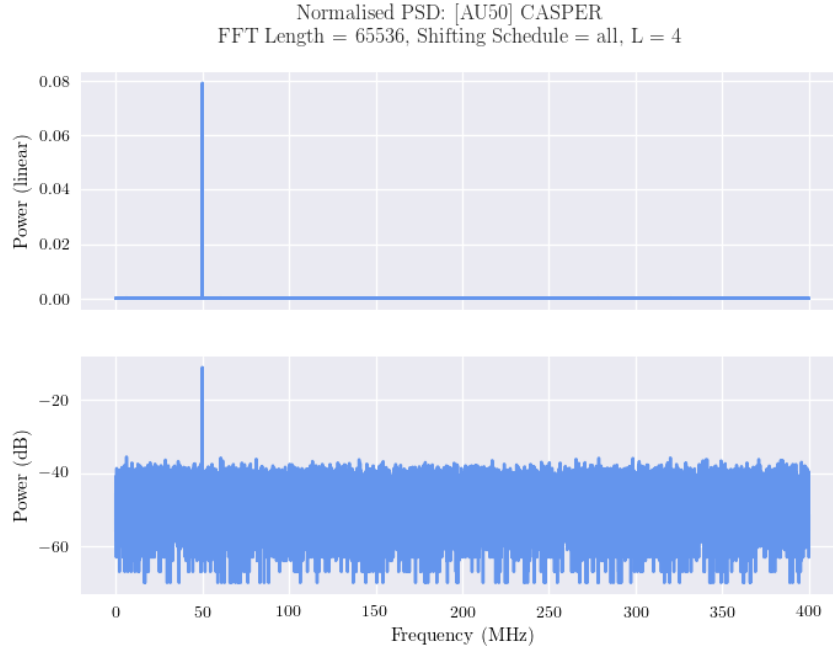


Figure B.21: Normalised PSD of CASPER, N = 65536, L = 4, Shift = all.

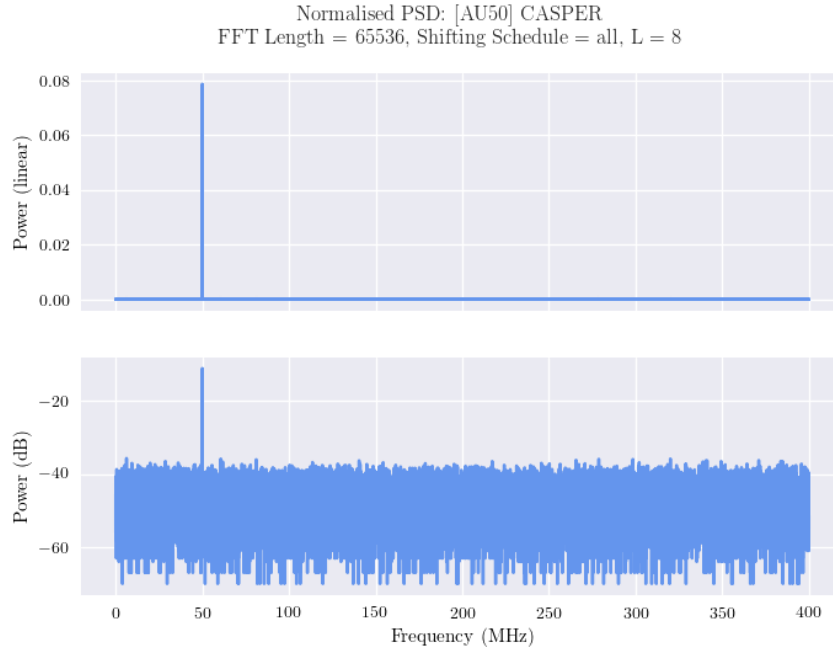


Figure B.22: Normalised PSD of CASPER, N = 65536, L = 8, Shift = all.

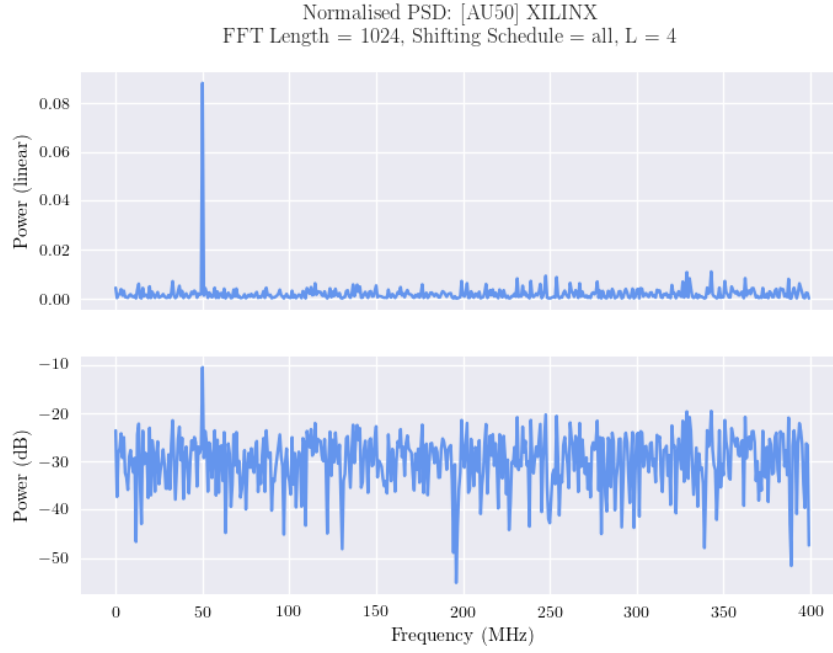


Figure B.23: Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.

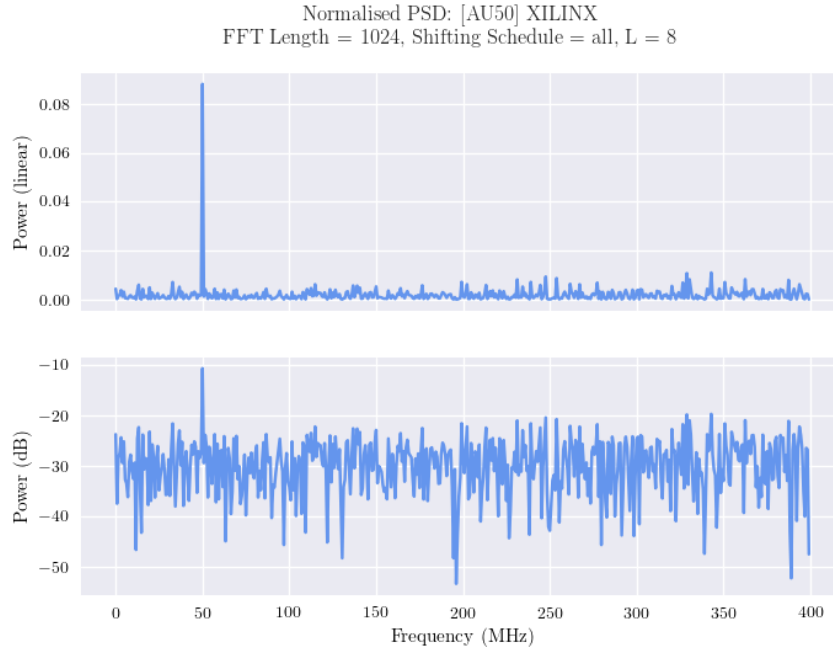


Figure B.24: Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.

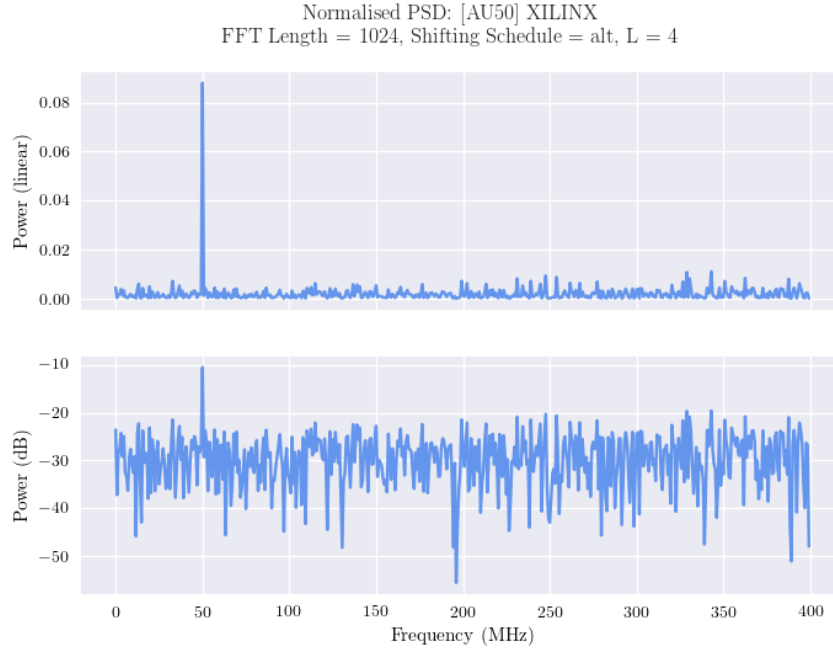


Figure B.25: Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = alt.

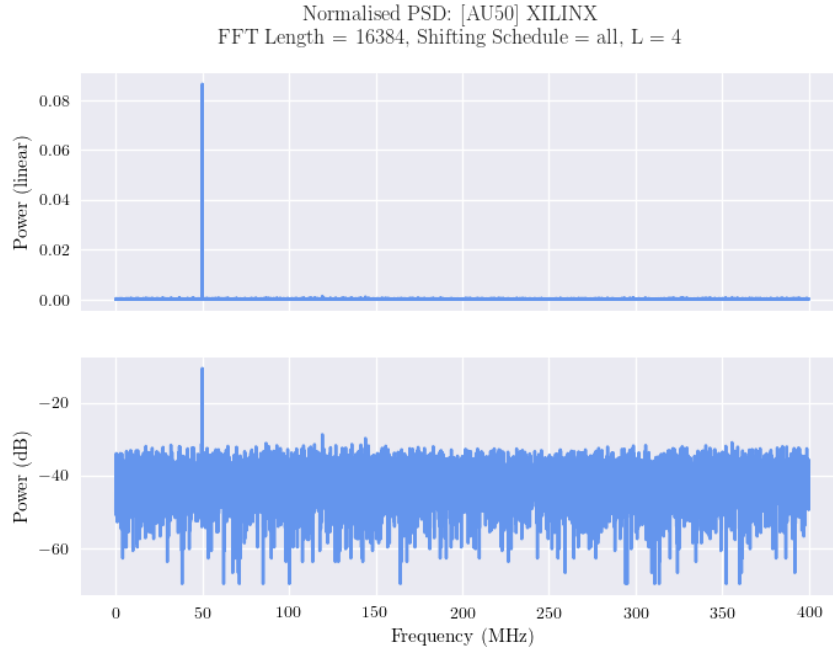


Figure B.26: Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.

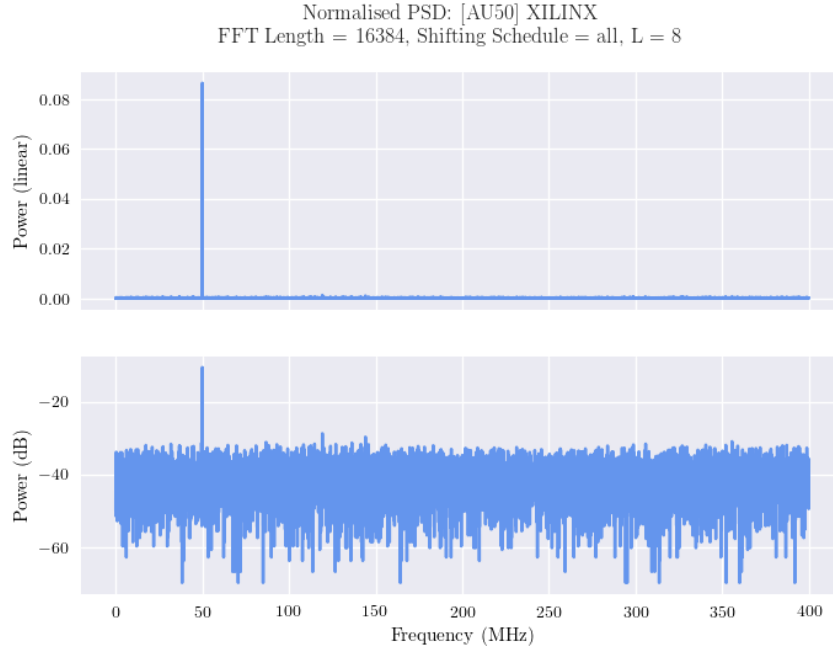


Figure B.27: Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.

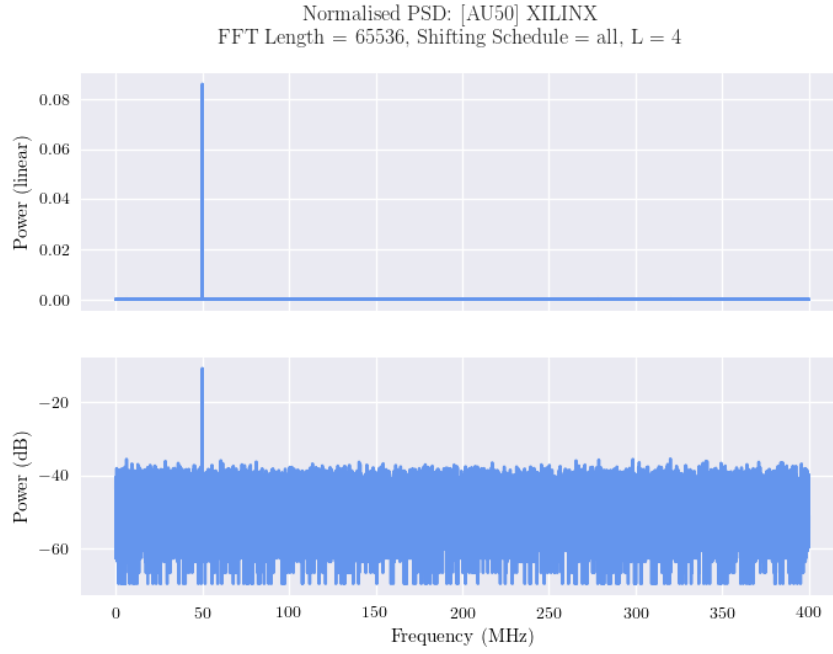


Figure B.28: Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.

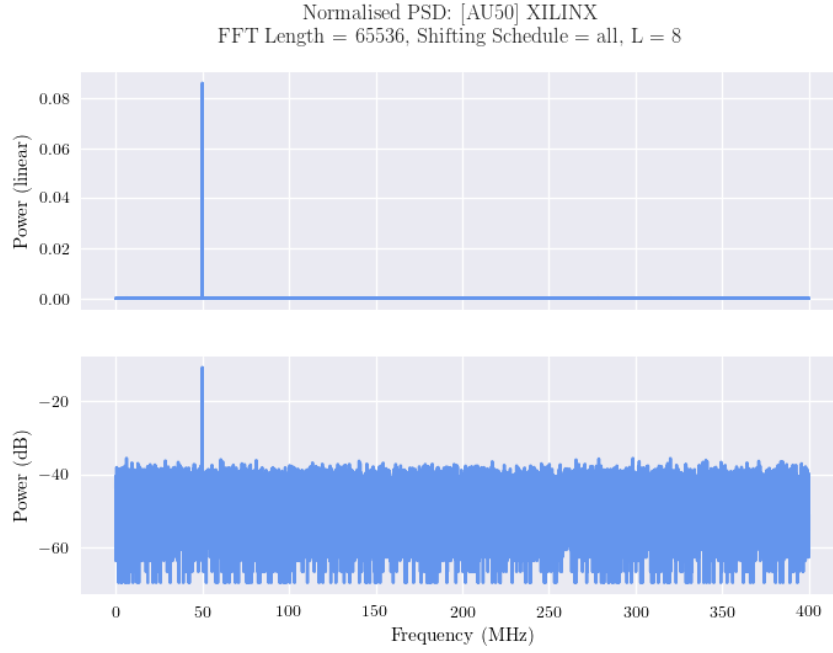


Figure B.29: Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.

B.4 TVG0 - Simulation PSDs

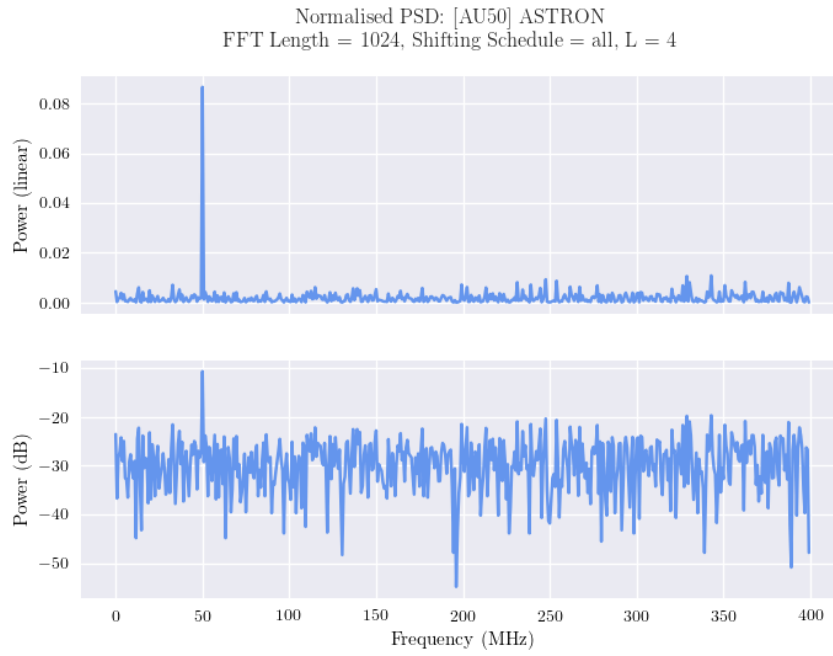


Figure B.30: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.

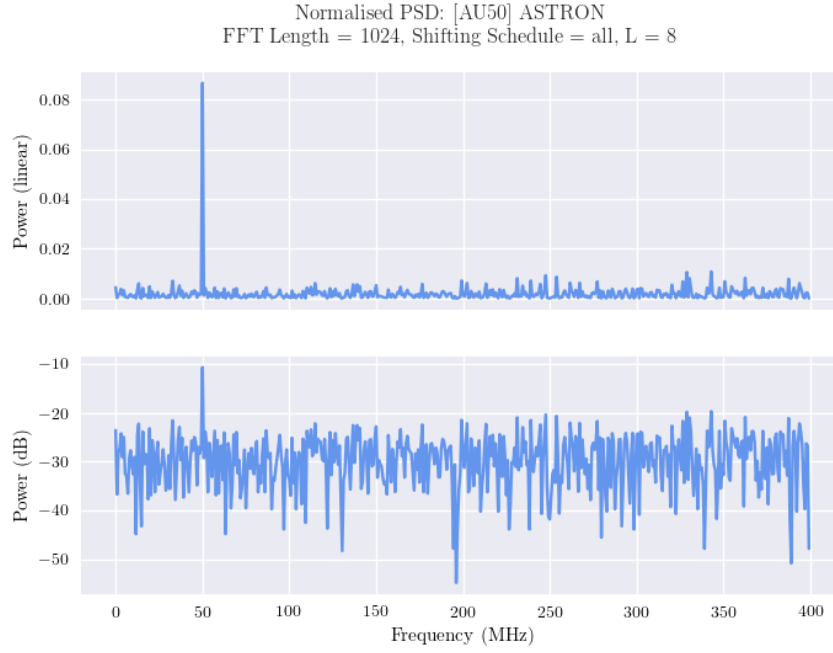


Figure B.31: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.

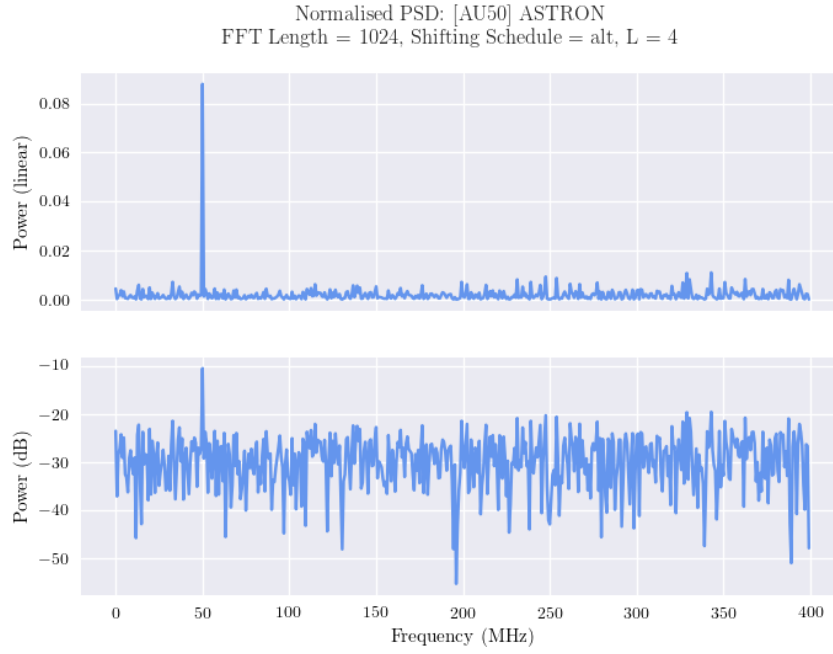


Figure B.32: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = alt.

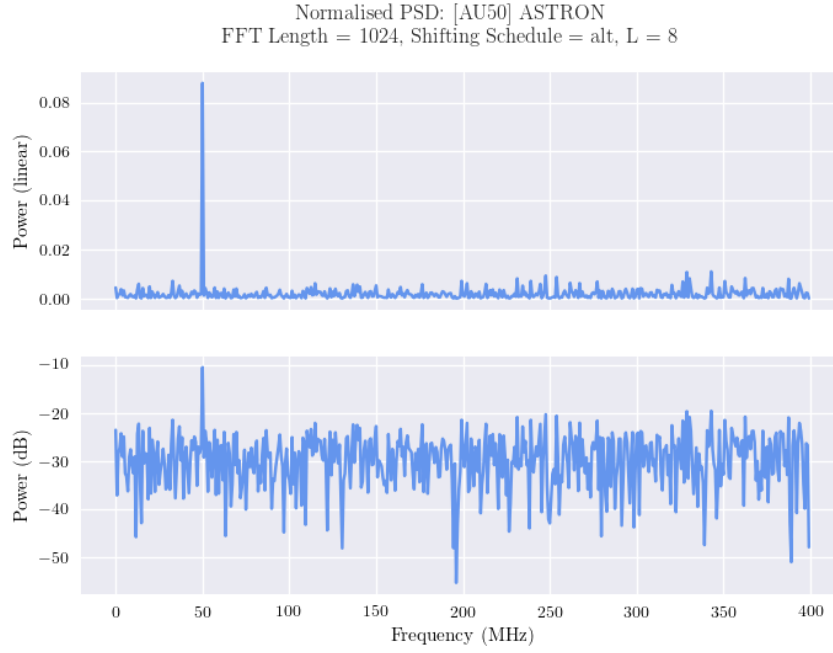


Figure B.33: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = alt.

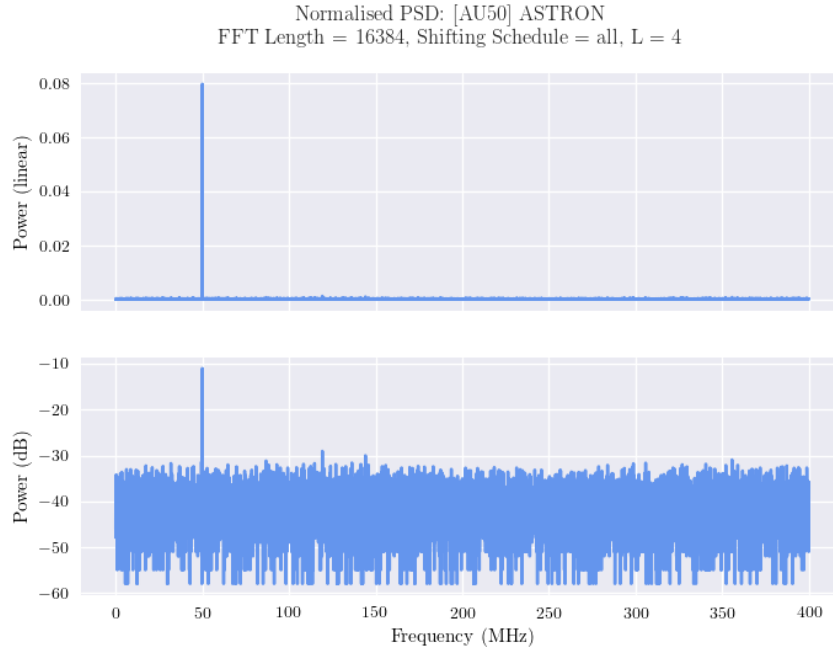


Figure B.34: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.

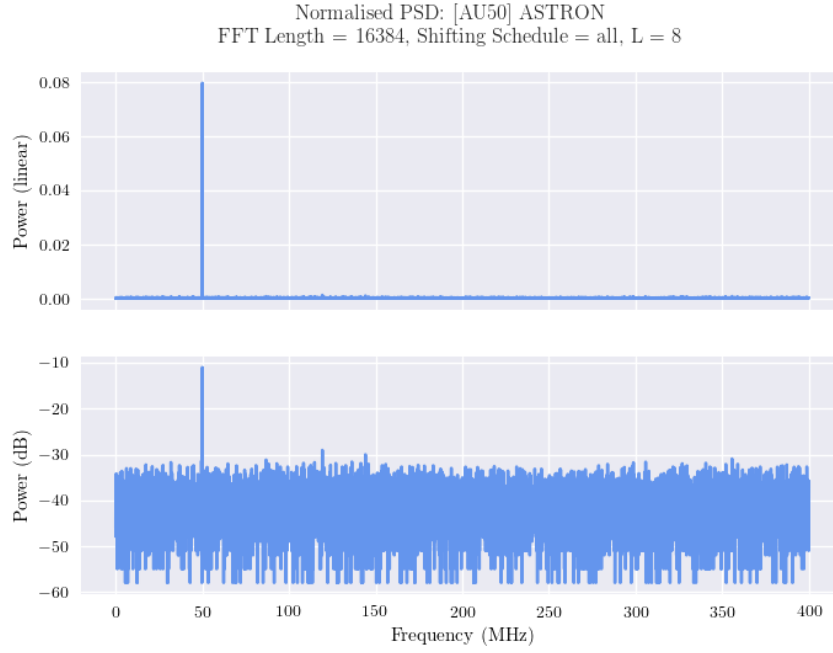


Figure B.35: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.

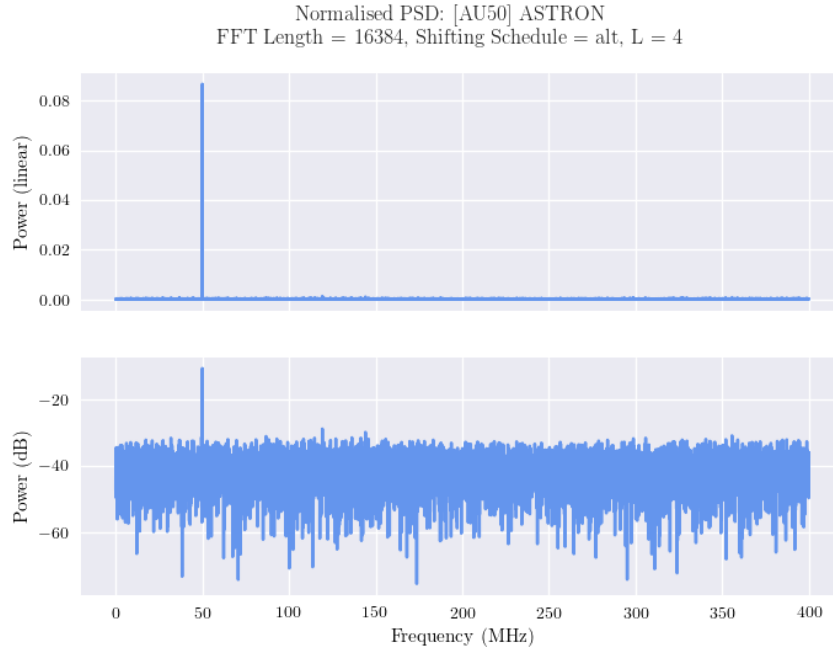


Figure B.36: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = alt.

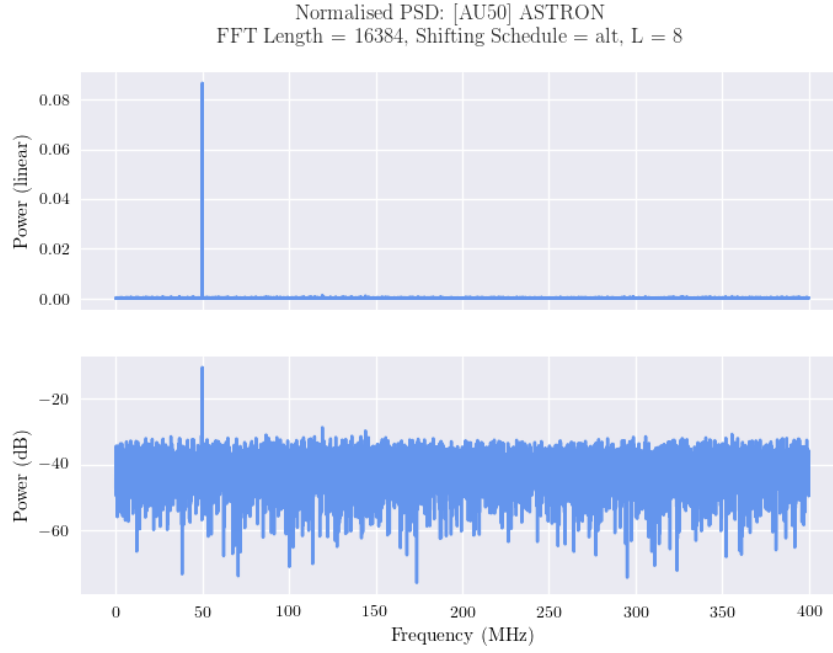


Figure B.37: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = alt.

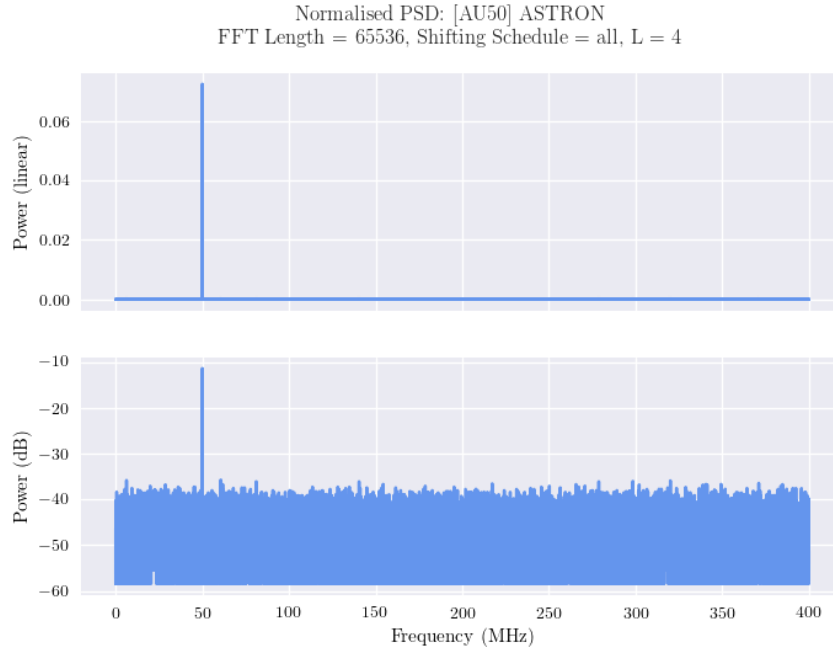


Figure B.38: Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.

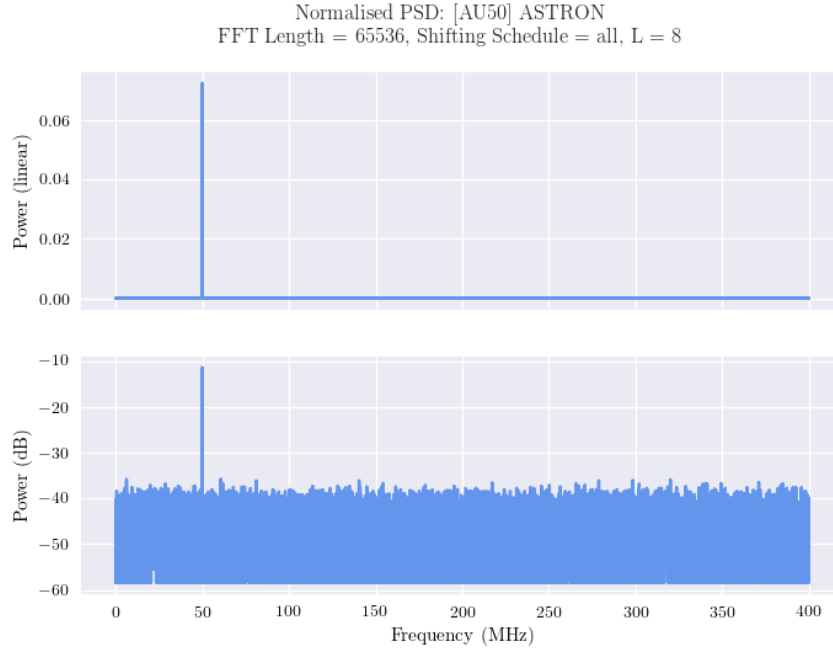


Figure B.39: Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.

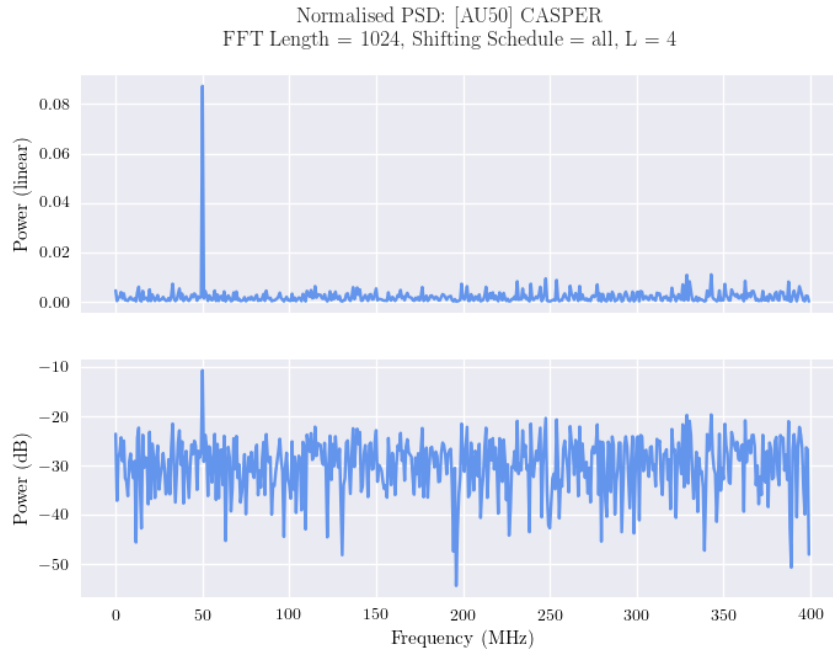


Figure B.40: Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.

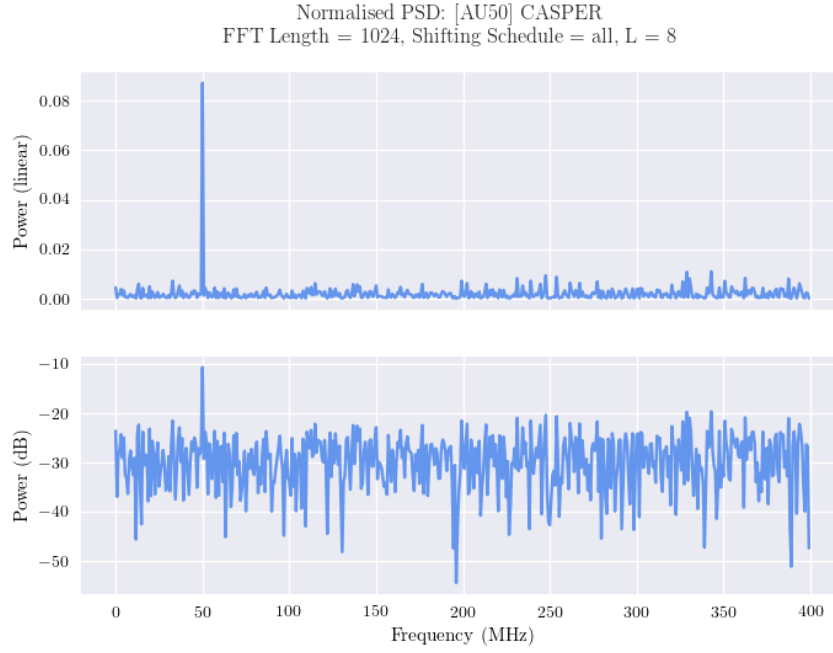


Figure B.41: Normalised PSD of CASPER, N = 1024, L = 8, Shift = all.

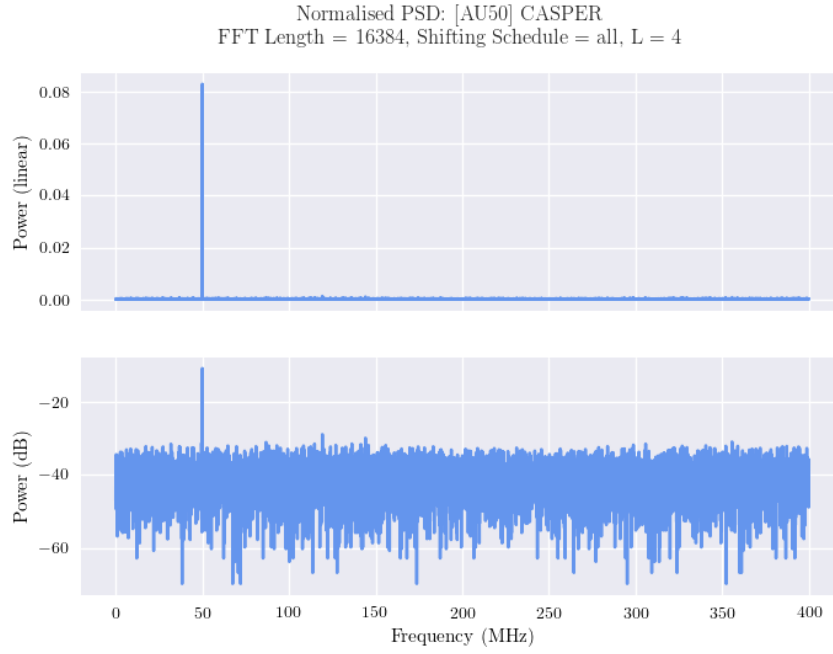


Figure B.42: Normalised PSD of CASPER, N = 16384, L = 4, Shift = all.

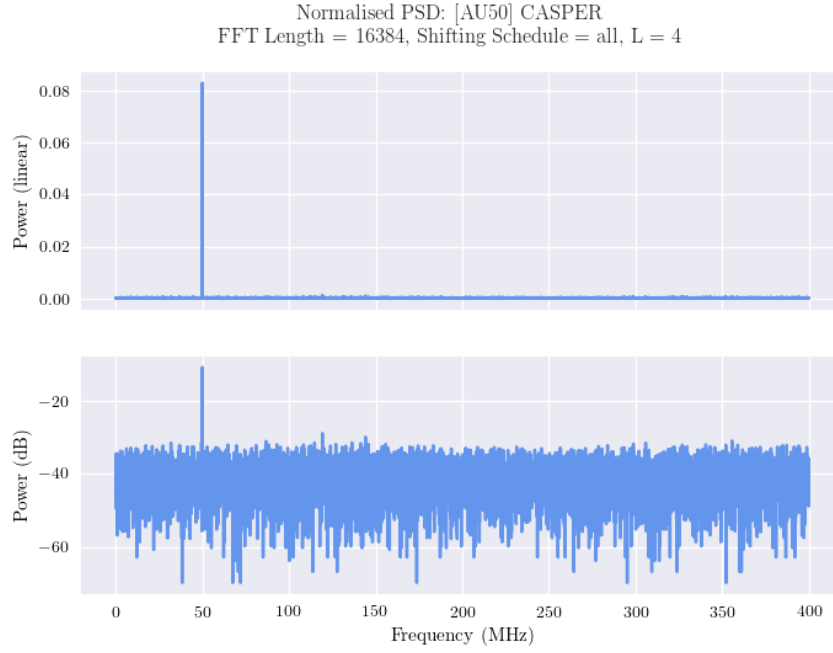


Figure B.43: Normalised PSD of CASPER, N = 16384, L = 8, Shift = all.

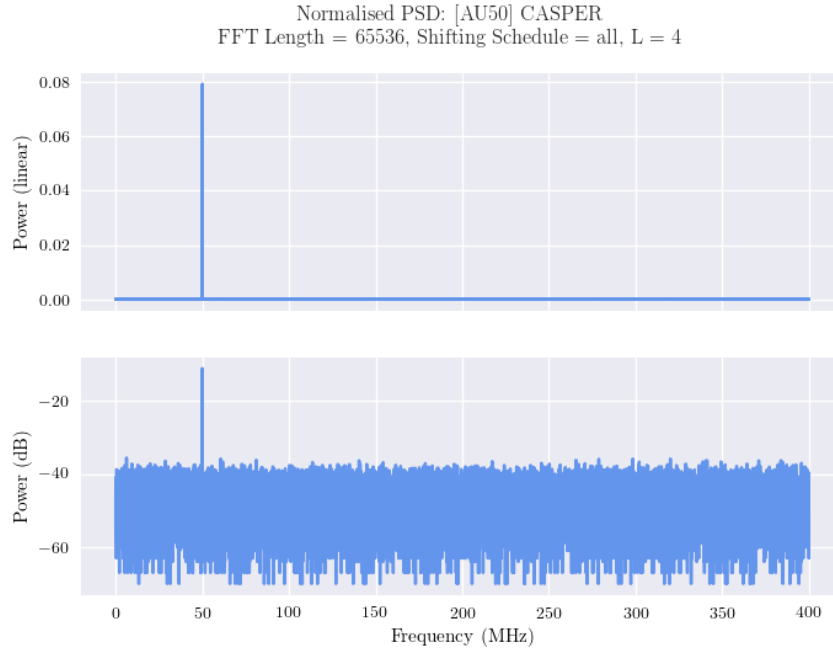


Figure B.44: Normalised PSD of CASPER, N = 65536, L = 4, Shift = all.

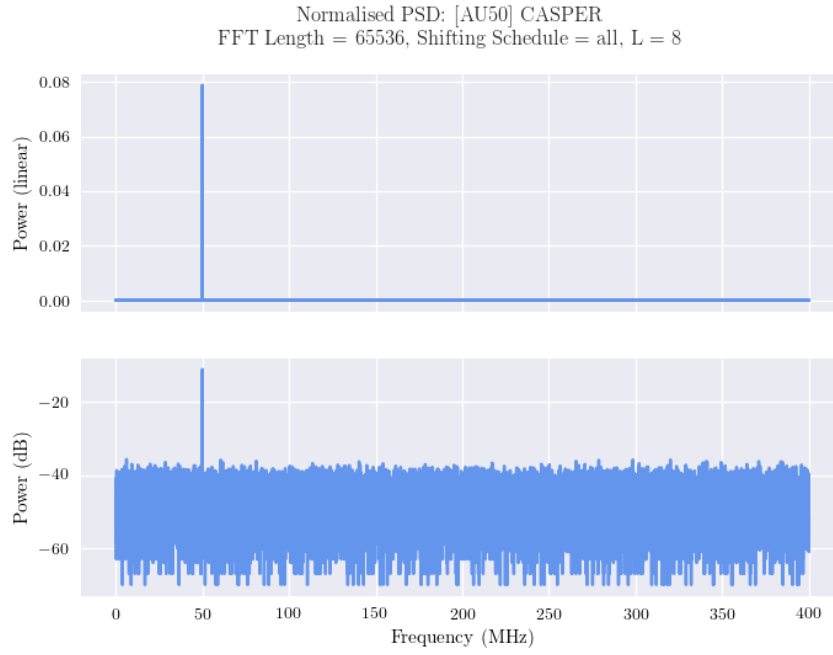


Figure B.45: Normalised PSD of CASPER, N = 65536, L = 8, Shift = all.

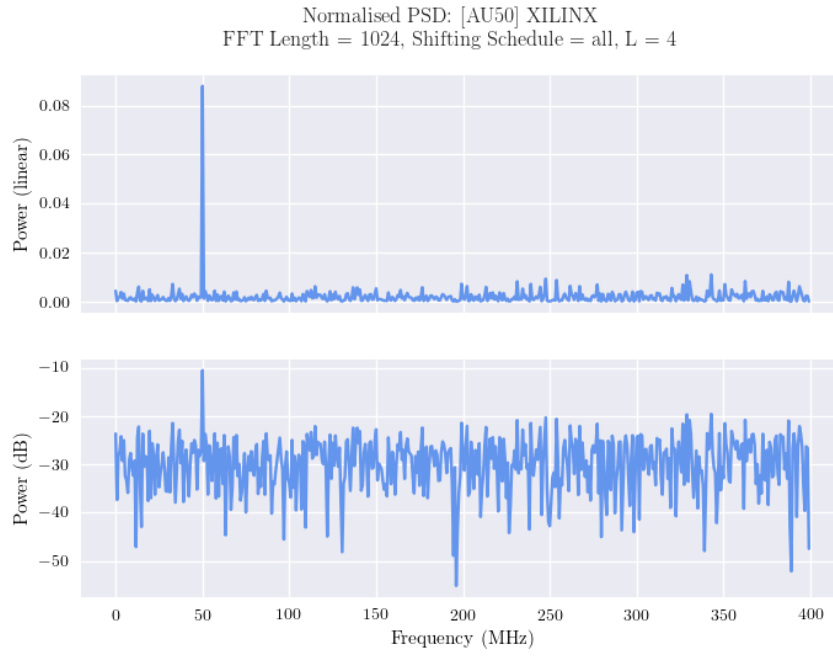


Figure B.46: Normalised PSD of Xilinx, N = 1024, L = 4, Shift = all.

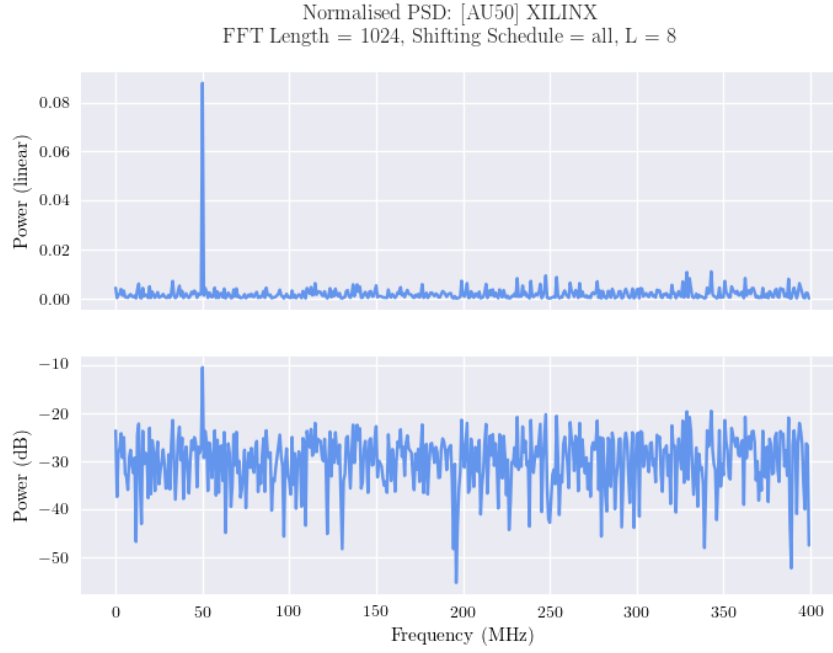


Figure B.47: Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.

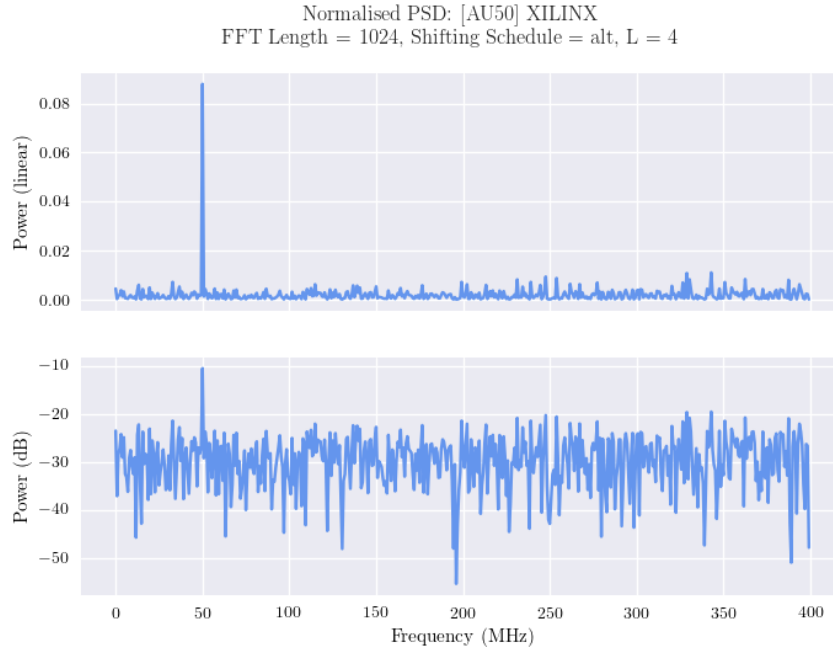


Figure B.48: Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = alt.

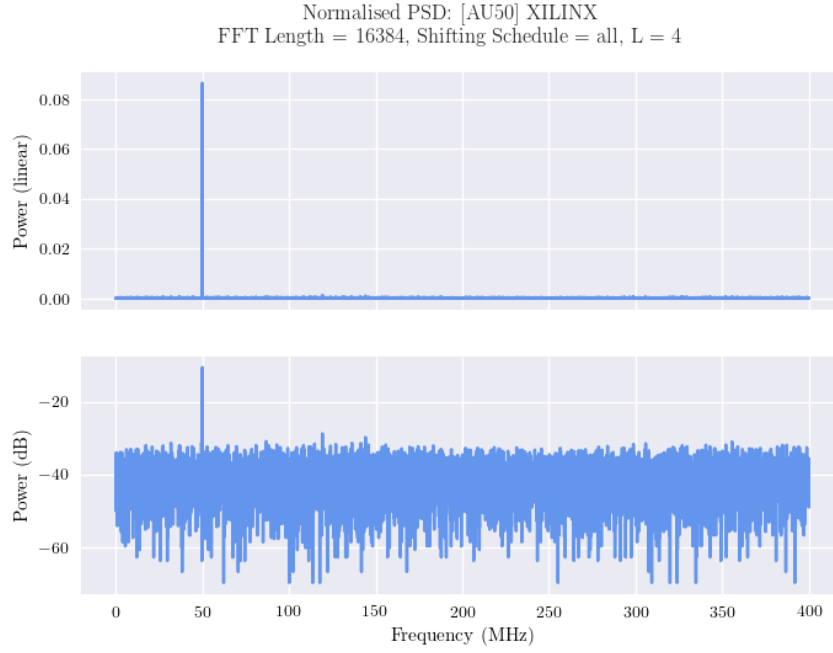


Figure B.49: Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.

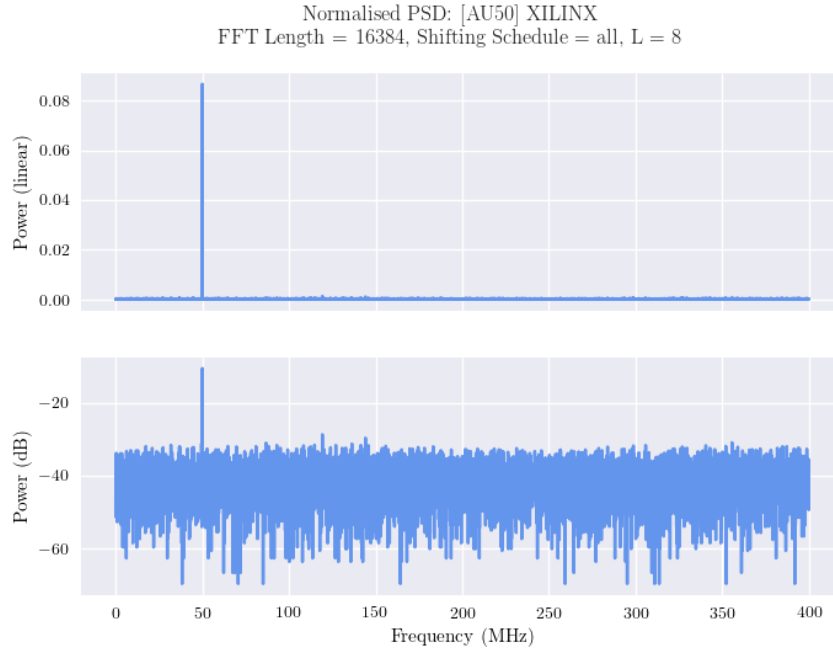


Figure B.50: Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.



Figure B.51: Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.

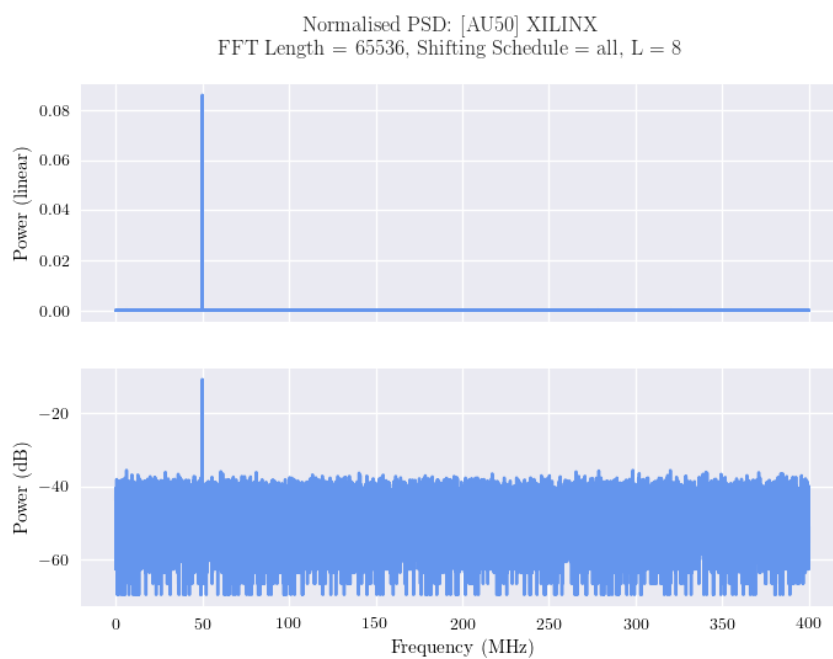


Figure B.52: Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.

Appendix C

C.1 TVG1 - FFTW PSDs

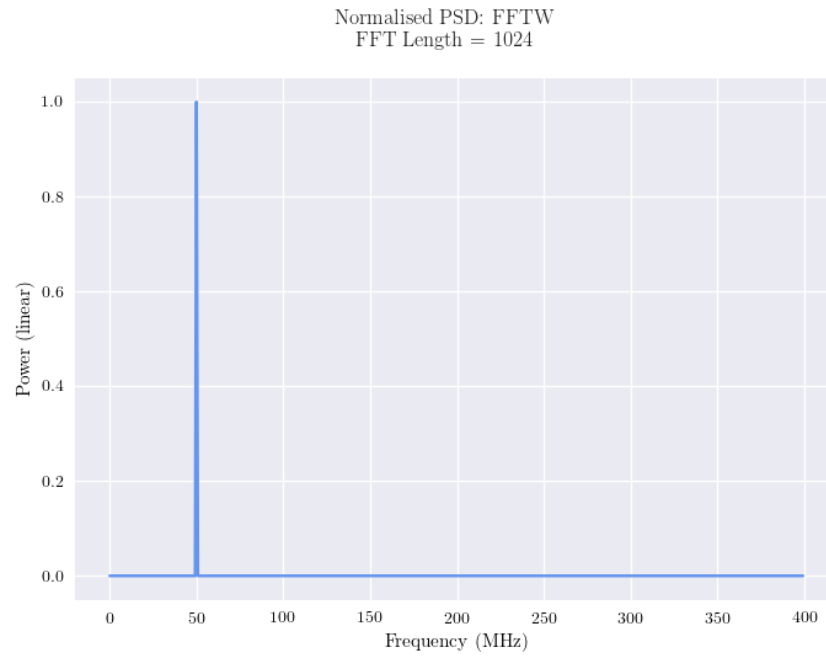


Figure C.1: Normalised PSD of FFTW, $N = 1024$.

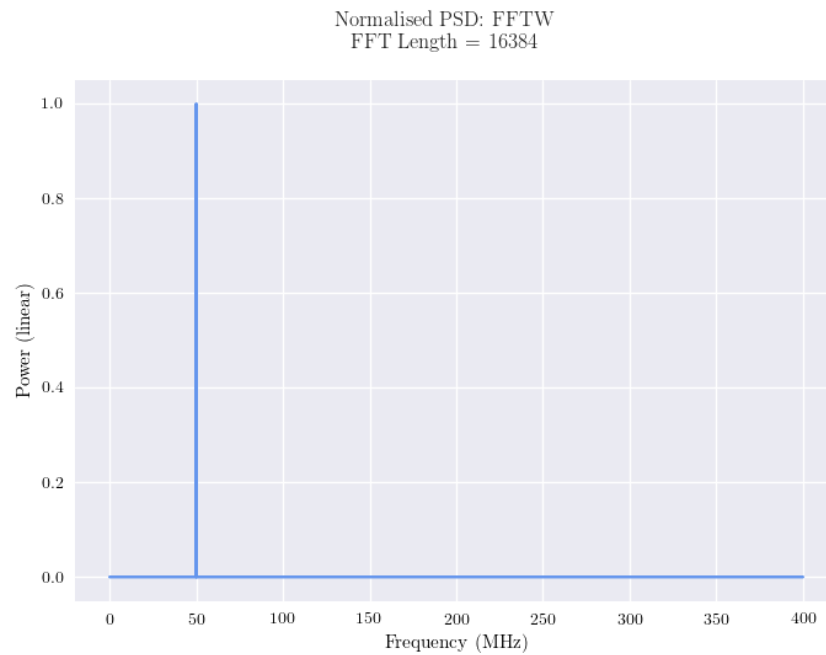


Figure C.2: Normalised PSD of FFTW, $N = 16384$.

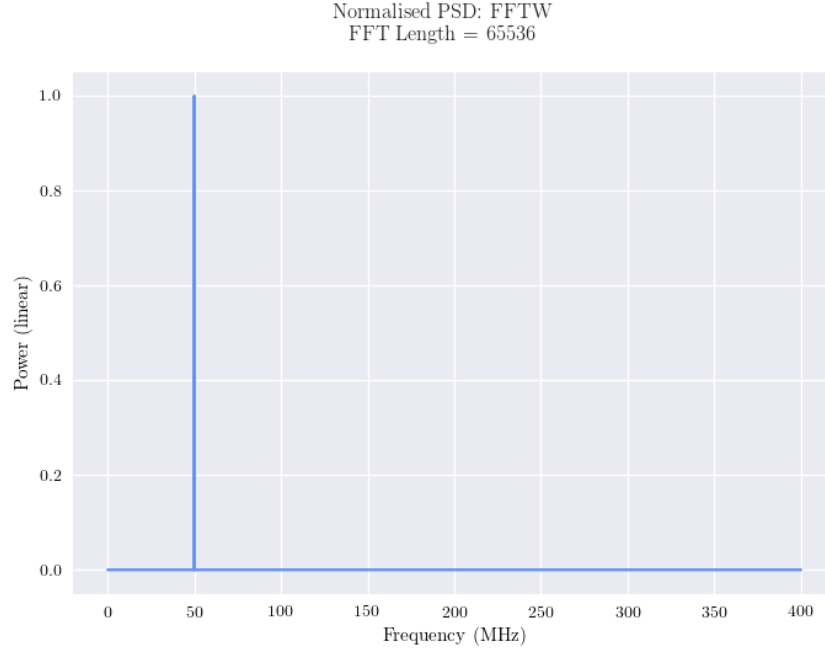


Figure C.3: Normalised PSD of FFTW, $N = 65536$.

C.2 TVG1 - Hardware PSDs

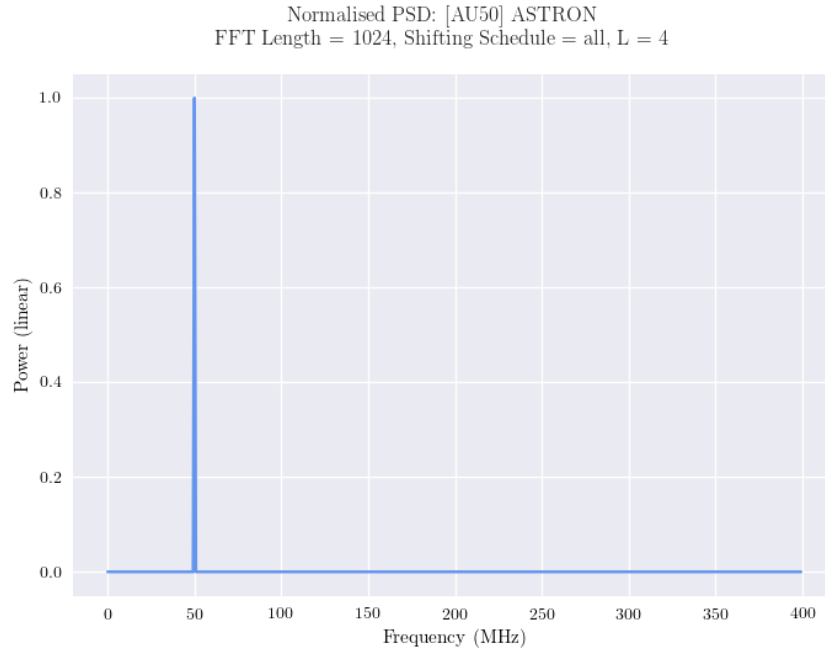


Figure C.4: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.

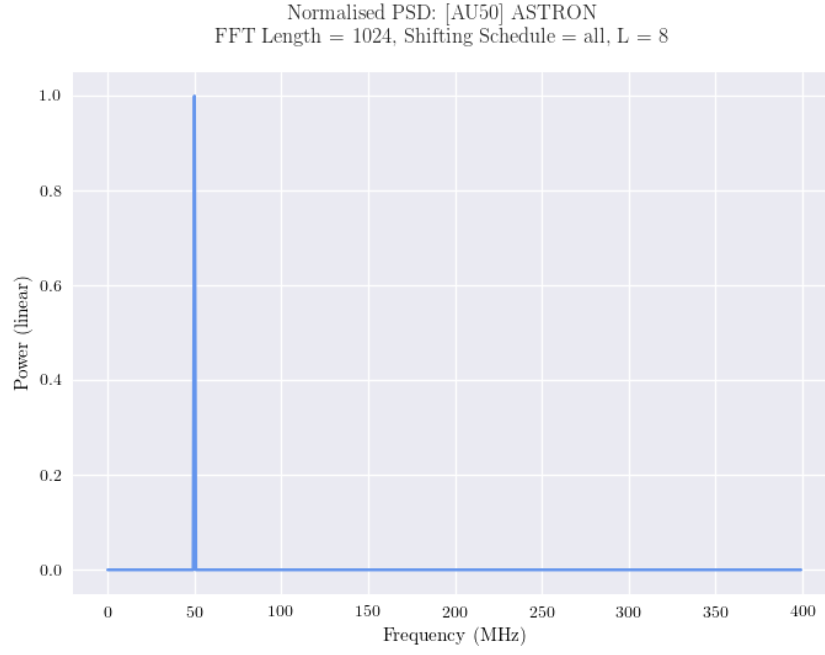


Figure C.5: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.

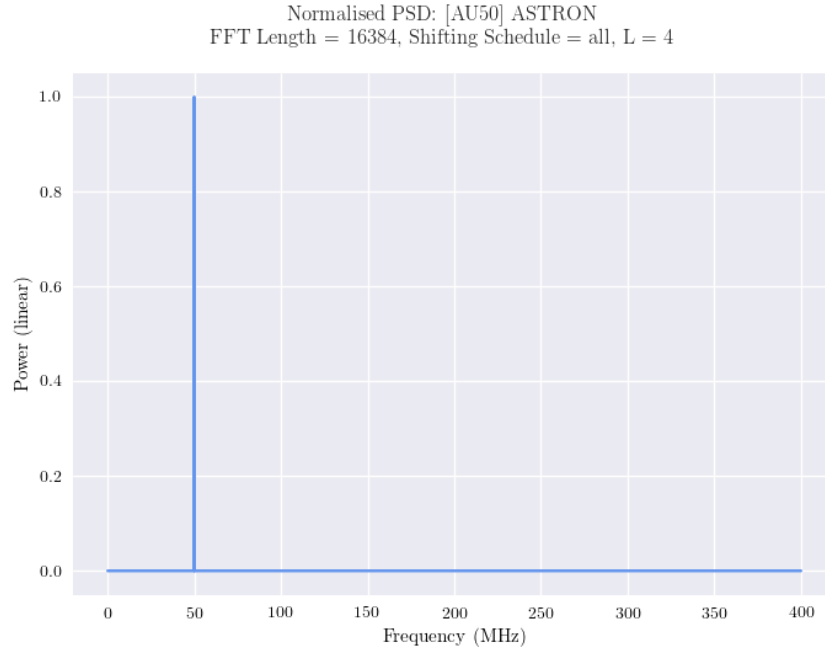


Figure C.6: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.

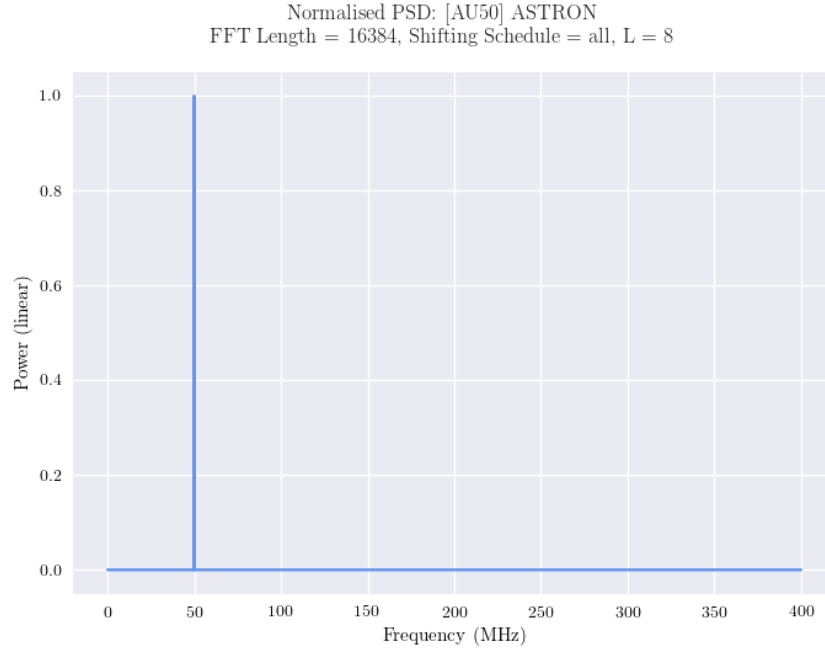


Figure C.7: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.

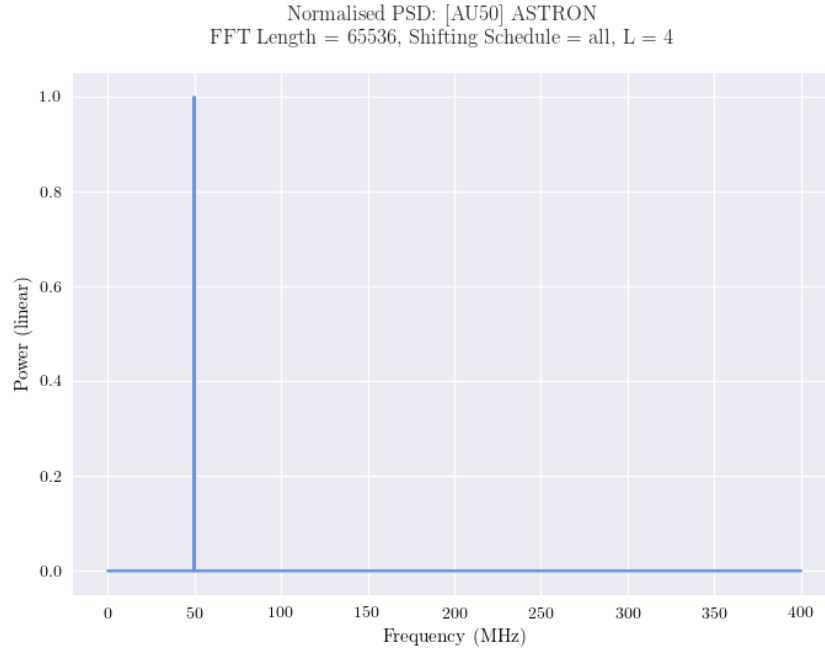


Figure C.8: Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.

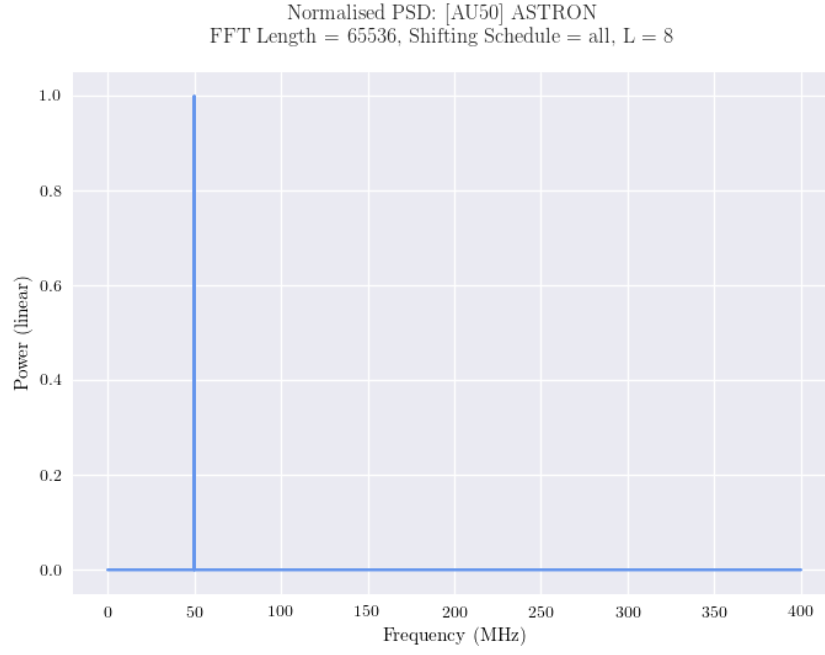


Figure C.9: Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.

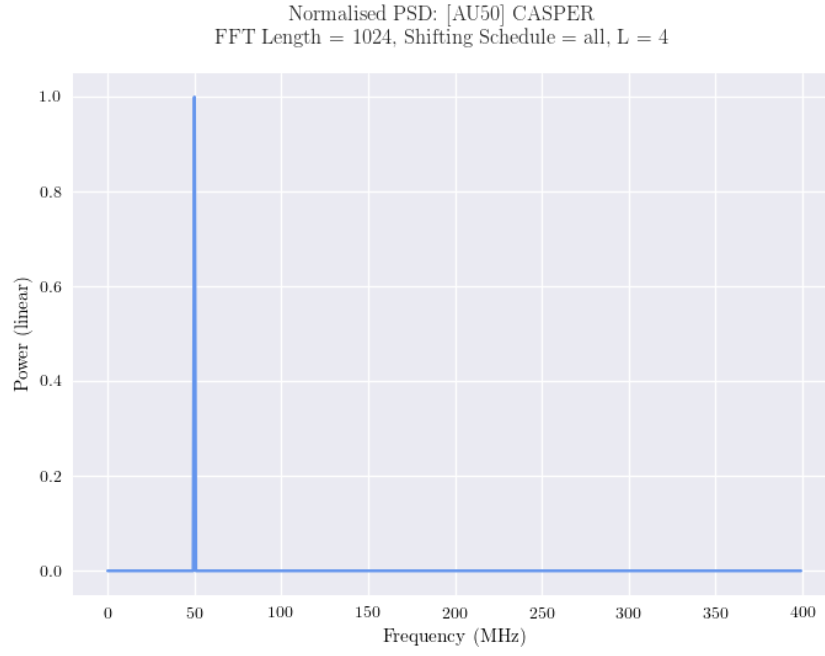


Figure C.10: Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.

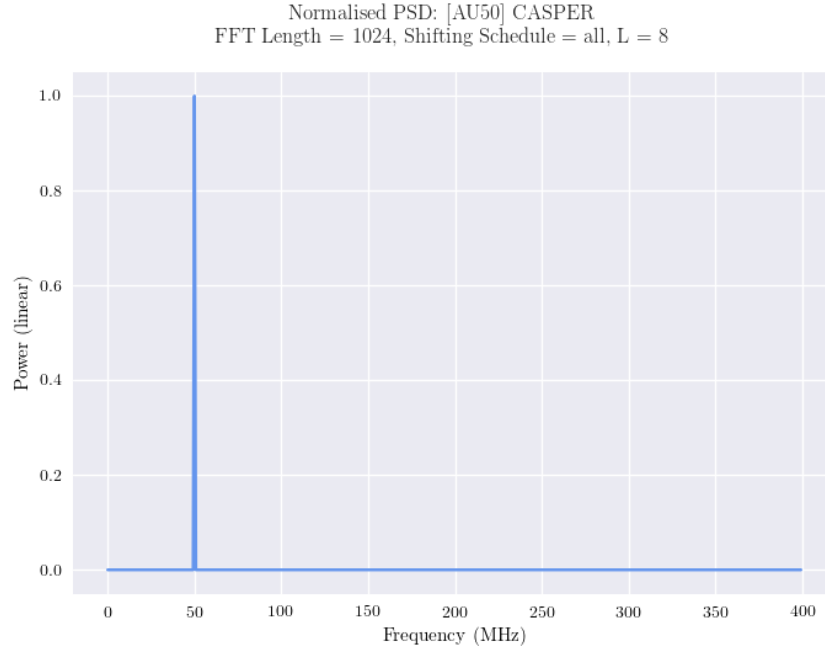


Figure C.11: Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.

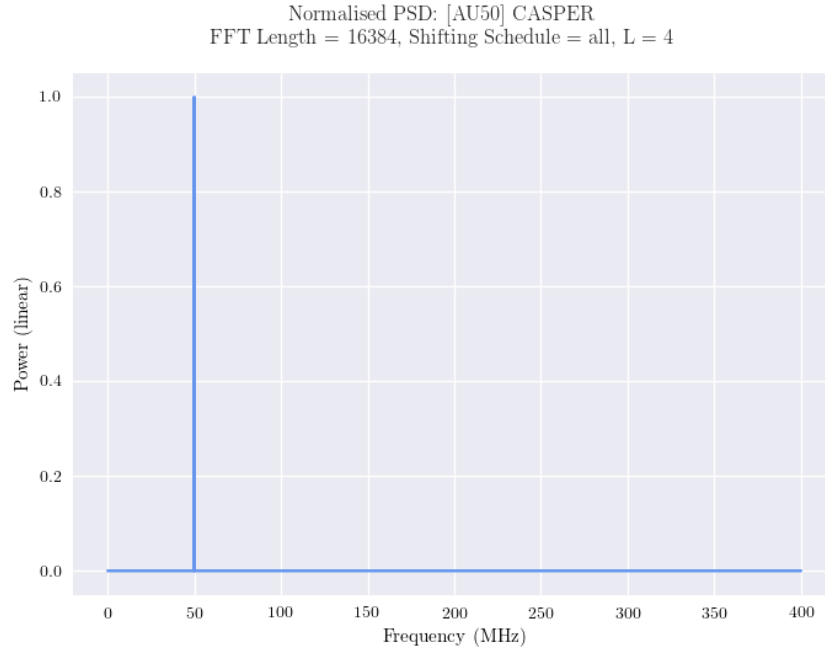


Figure C.12: Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.

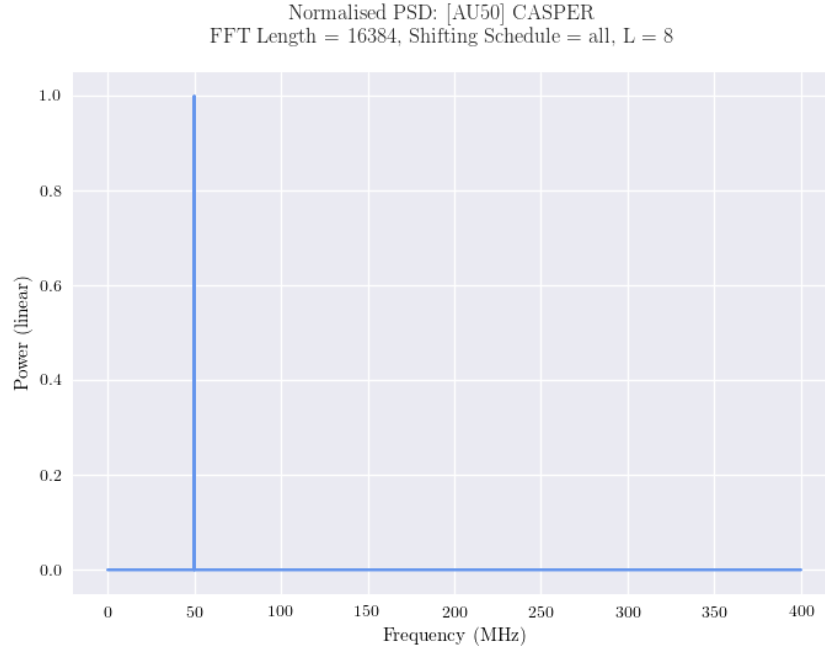


Figure C.13: Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.

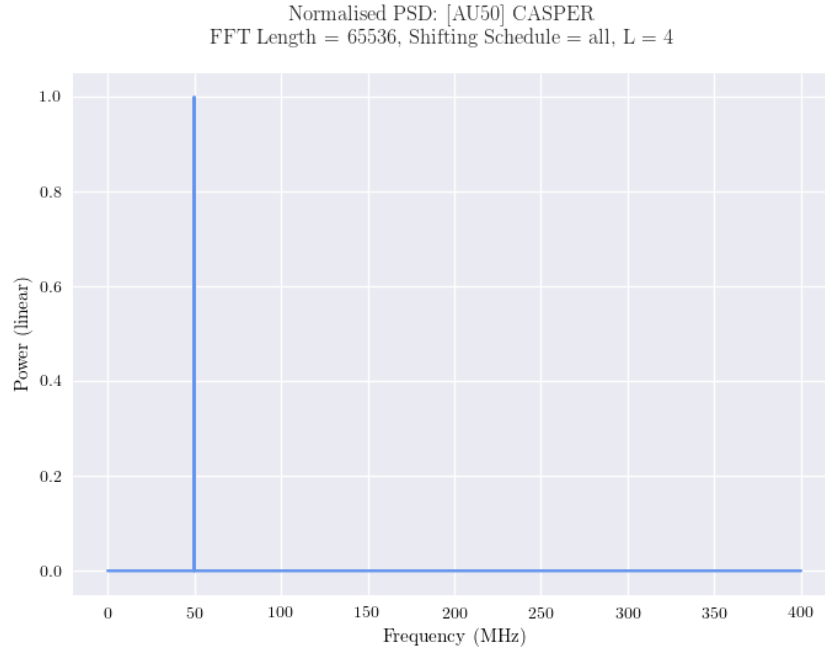


Figure C.14: Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.

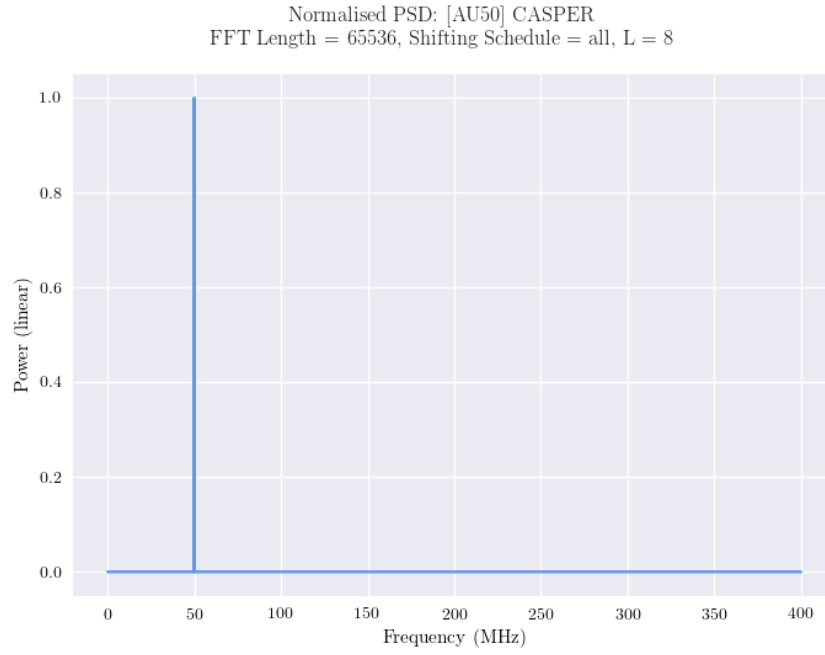


Figure C.15: Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.

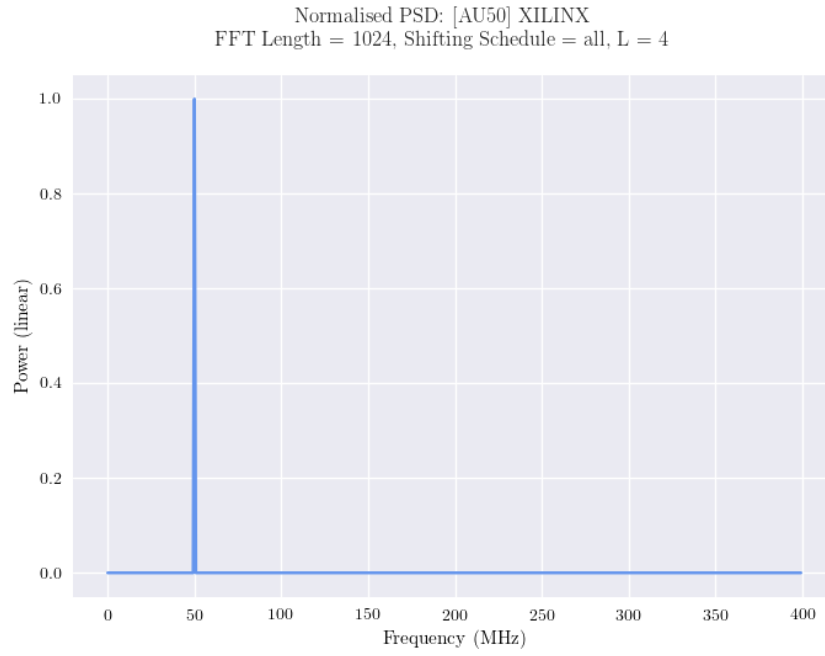


Figure C.16: Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.

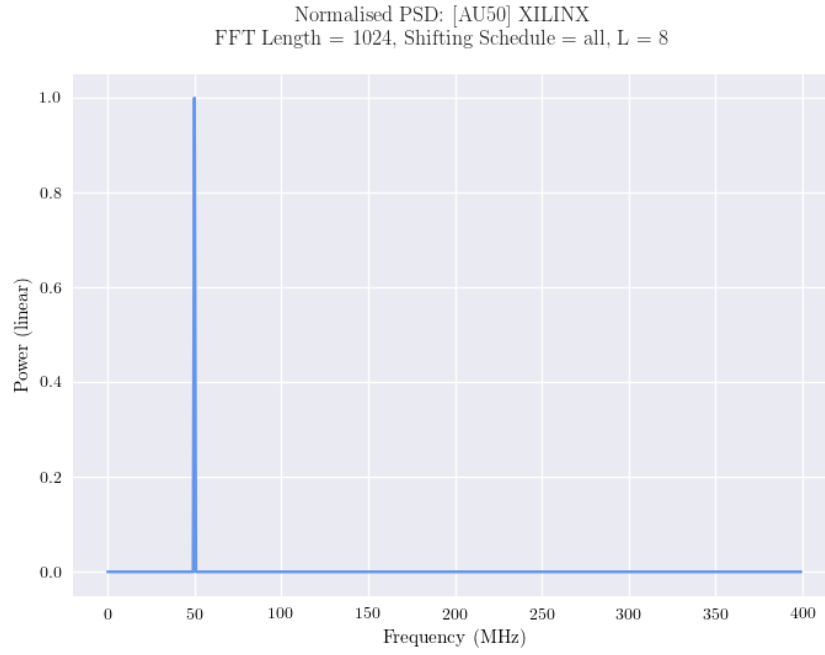


Figure C.17: Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.

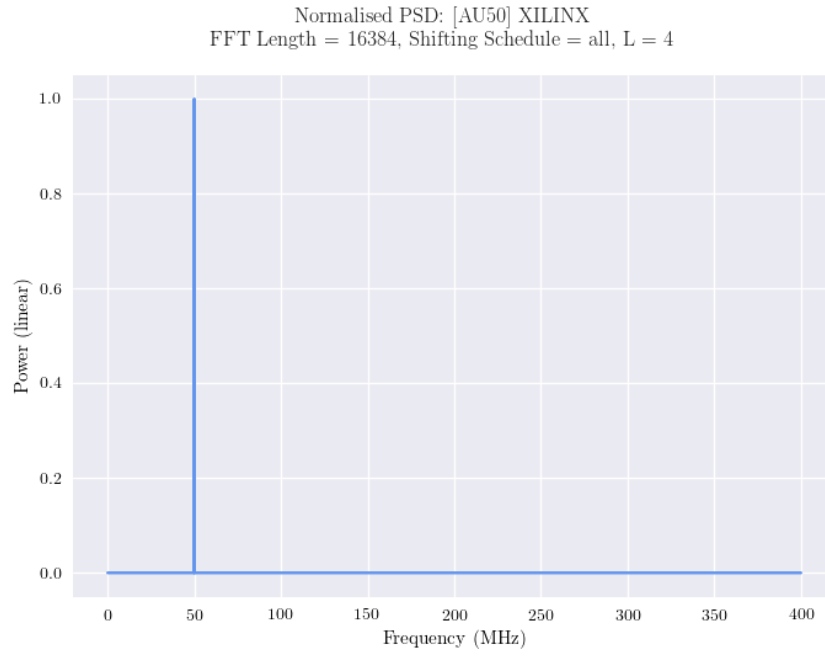


Figure C.18: Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.

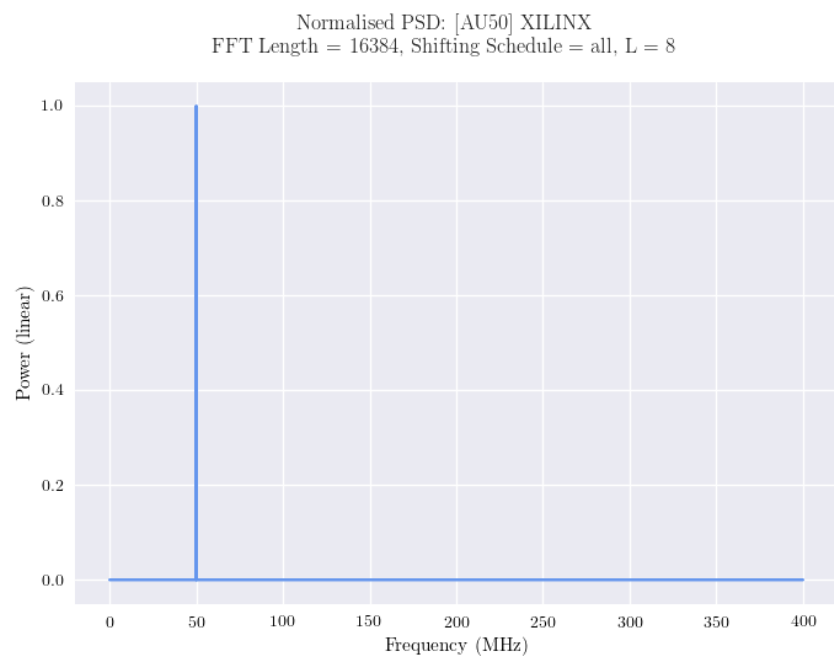


Figure C.19: Normalised PSD of Xilinx, $N = 16384$, $L = 8$, Shift = all.

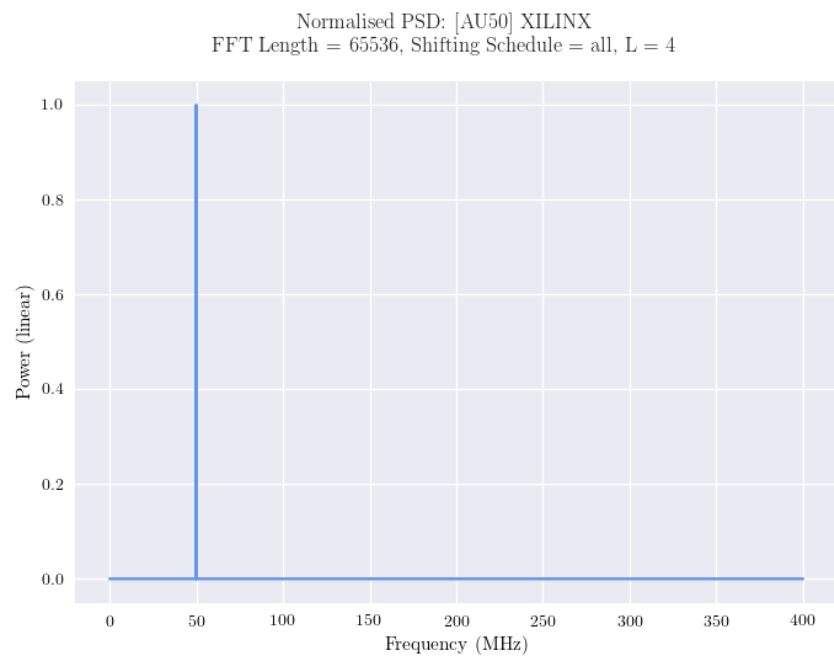


Figure C.20: Normalised PSD of Xilinx, $N = 65536$, $L = 4$, Shift = all.

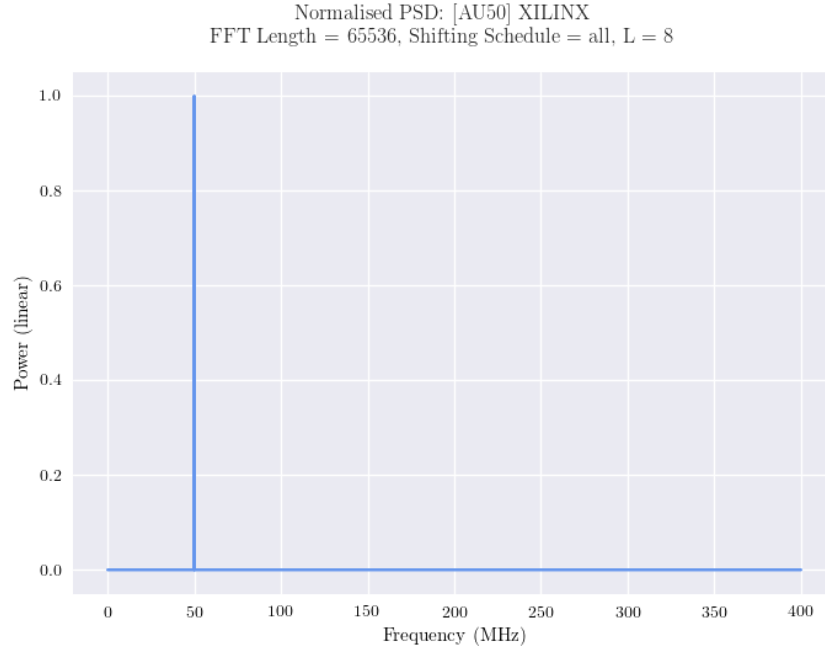


Figure C.21: Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.

C.3 TVG1 - Simulation PSDs

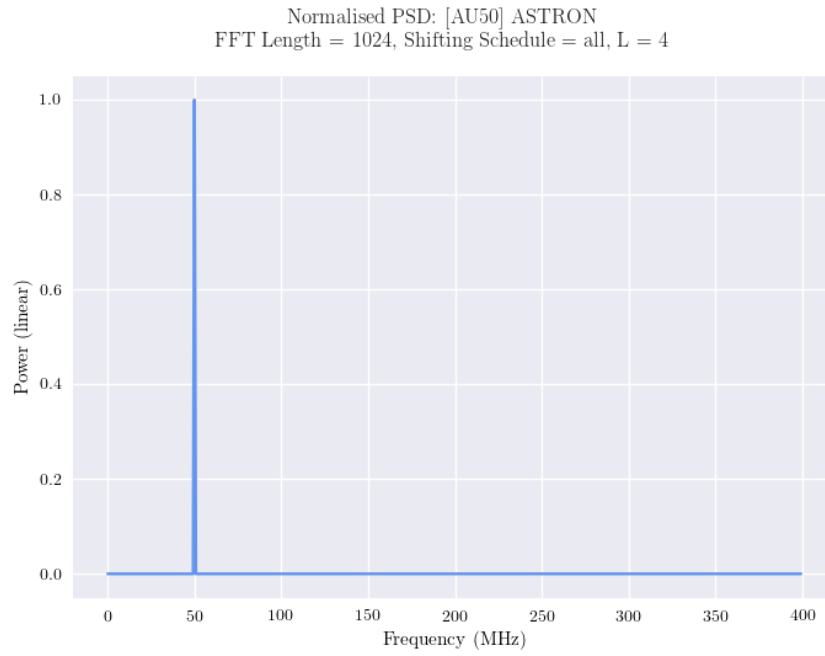


Figure C.22: Normalised PSD of ASTRON, $N = 1024$, $L = 4$, Shift = all.

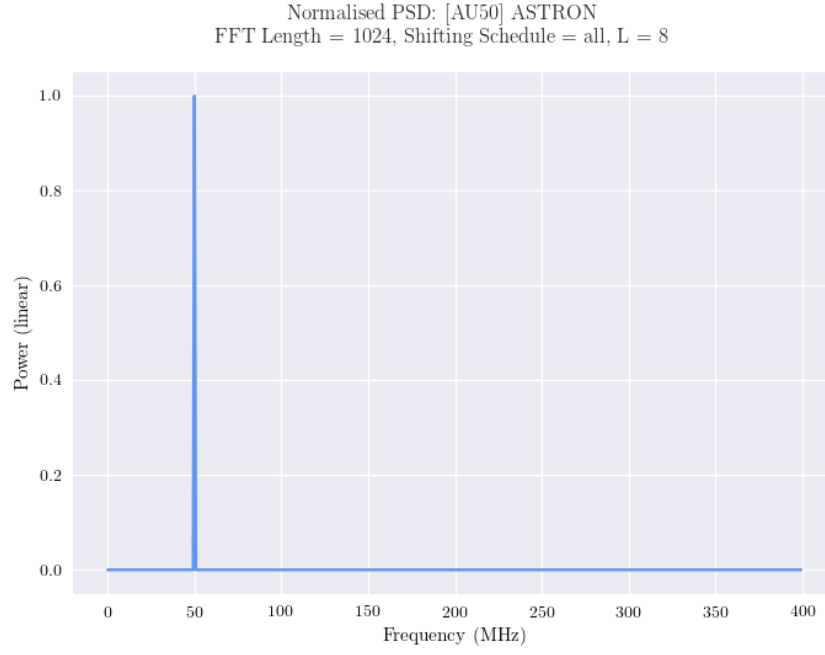


Figure C.23: Normalised PSD of ASTRON, $N = 1024$, $L = 8$, Shift = all.

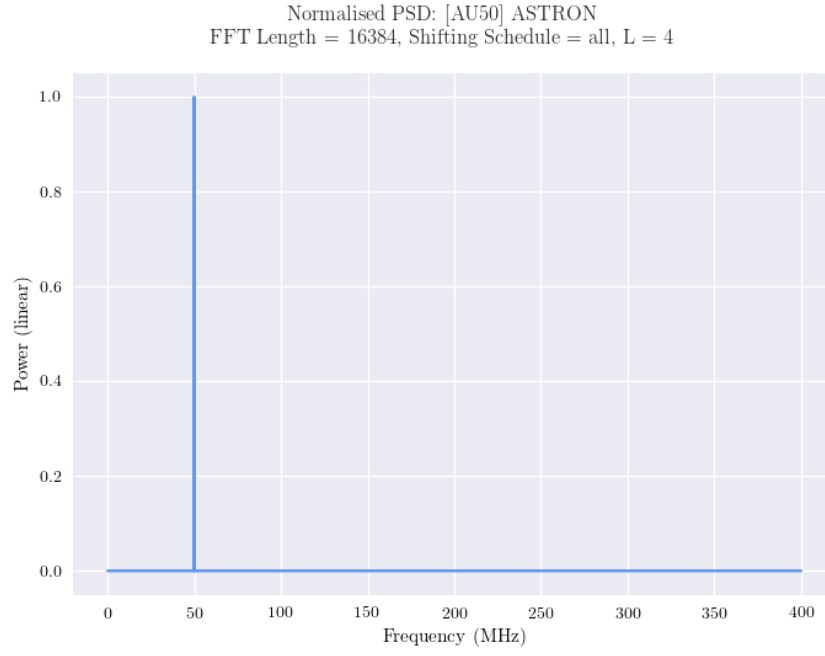


Figure C.24: Normalised PSD of ASTRON, $N = 16384$, $L = 4$, Shift = all.

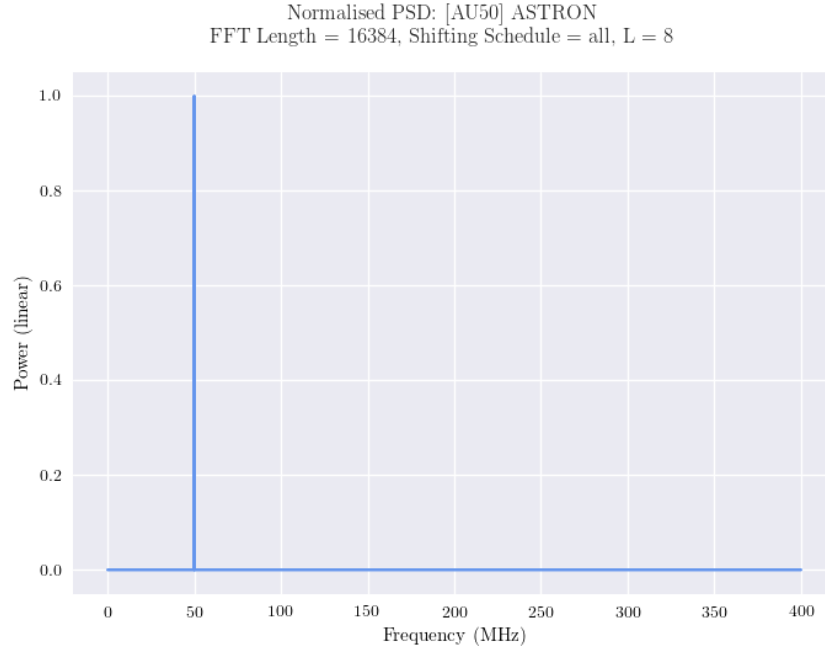


Figure C.25: Normalised PSD of ASTRON, $N = 16384$, $L = 8$, Shift = all.

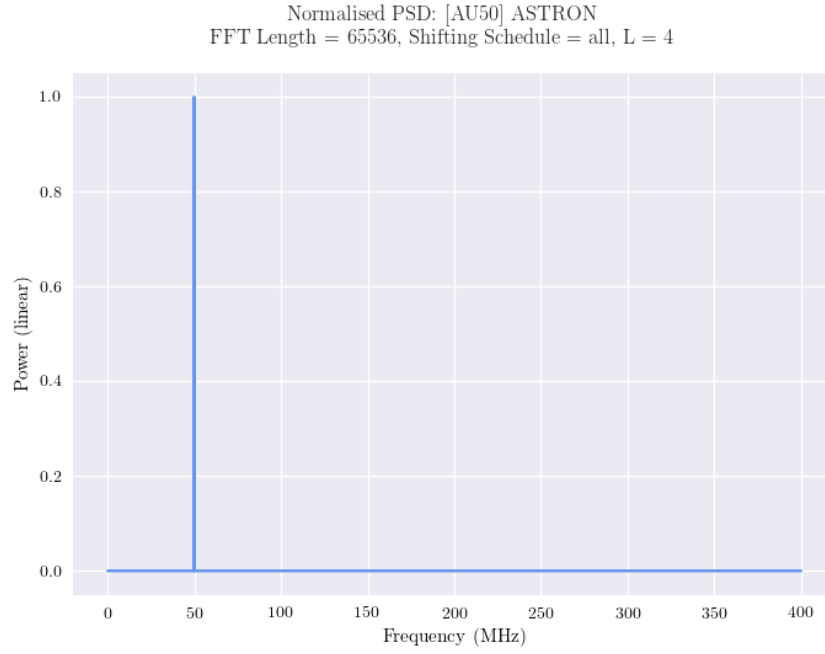


Figure C.26: Normalised PSD of ASTRON, $N = 65536$, $L = 4$, Shift = all.

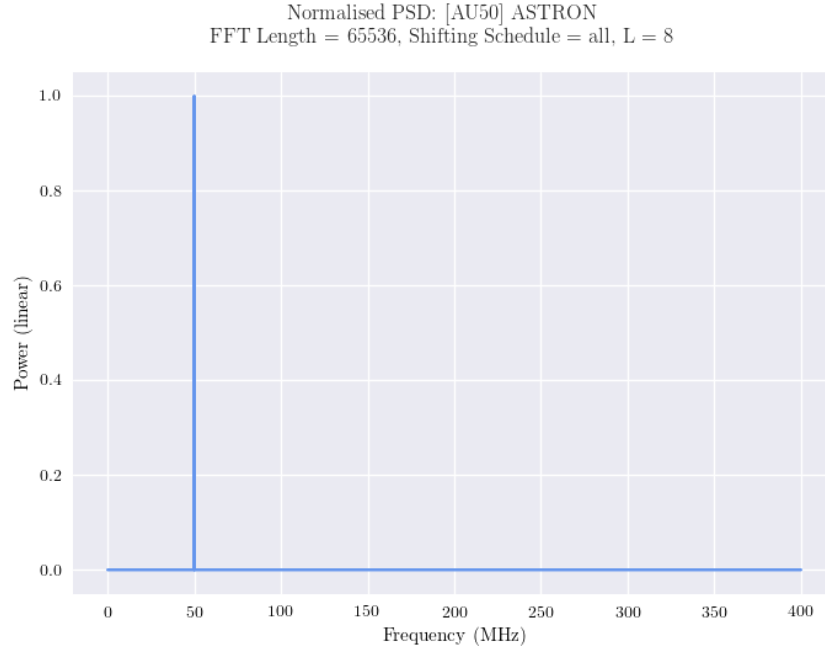


Figure C.27: Normalised PSD of ASTRON, $N = 65536$, $L = 8$, Shift = all.

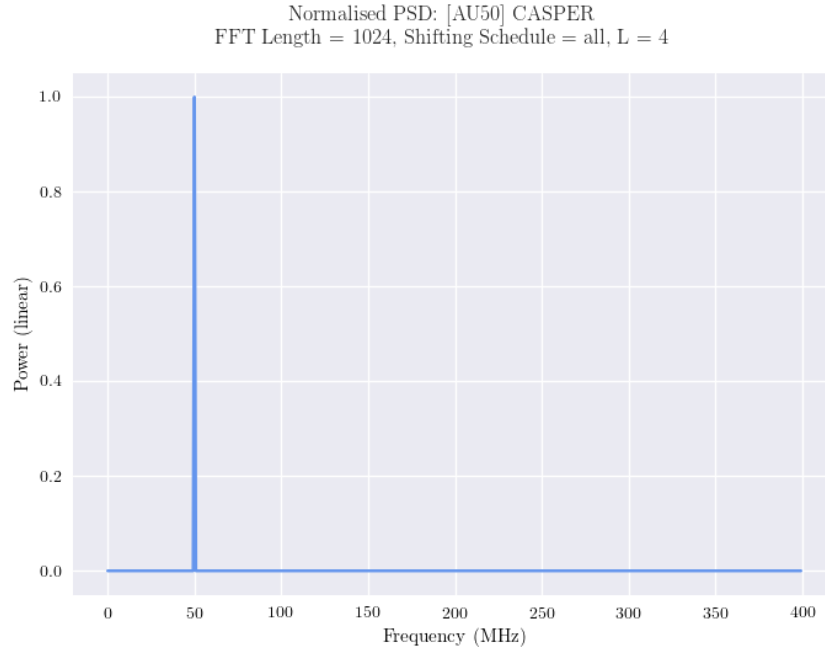


Figure C.28: Normalised PSD of CASPER, $N = 1024$, $L = 4$, Shift = all.

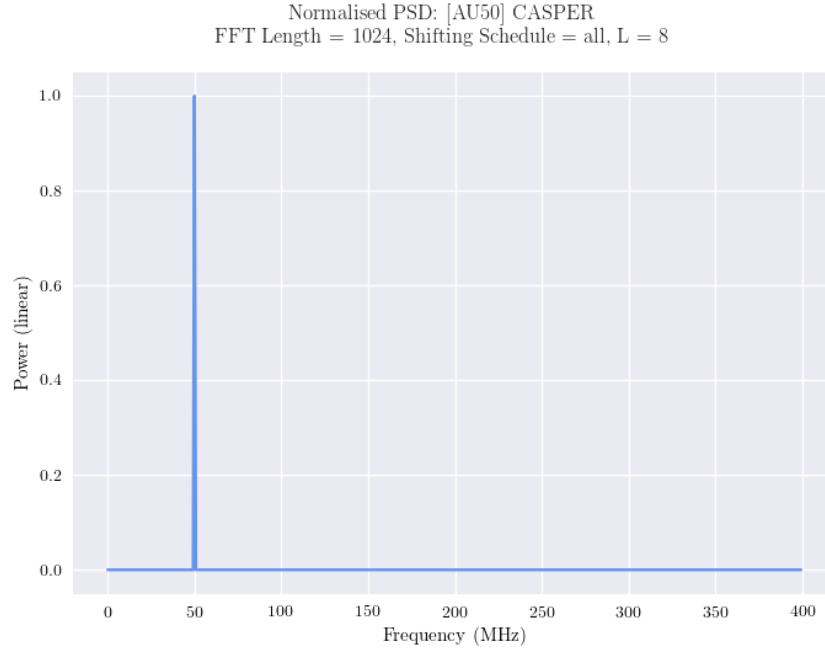


Figure C.29: Normalised PSD of CASPER, $N = 1024$, $L = 8$, Shift = all.

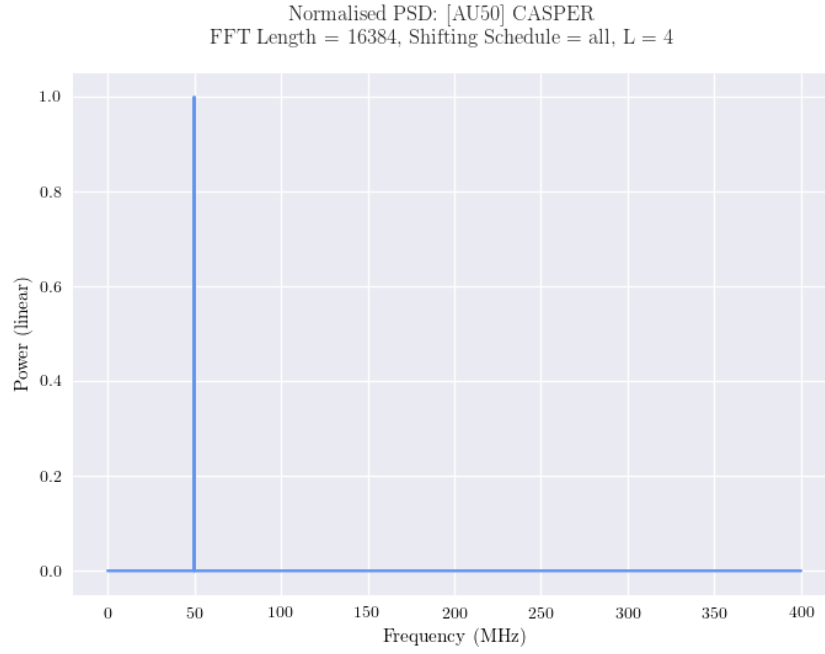


Figure C.30: Normalised PSD of CASPER, $N = 16384$, $L = 4$, Shift = all.

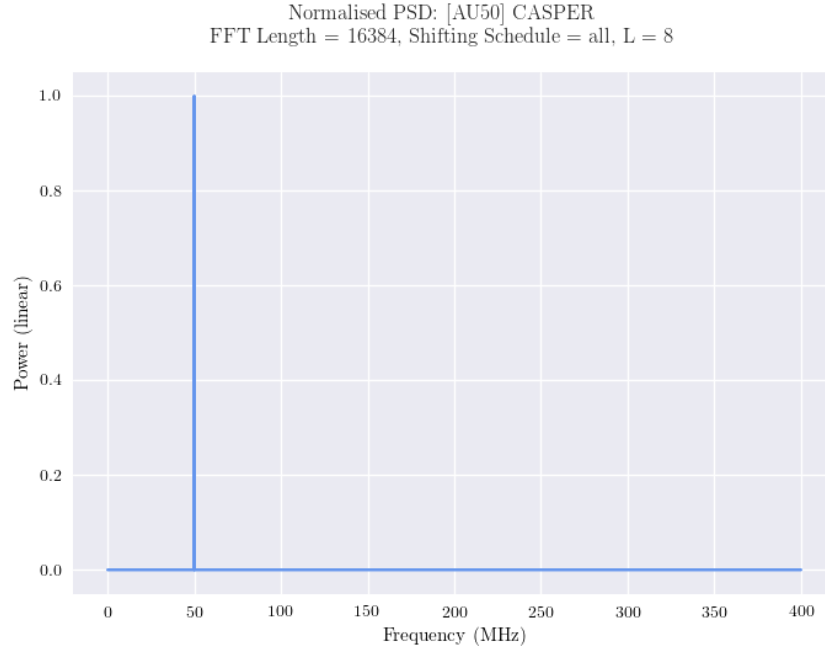


Figure C.31: Normalised PSD of CASPER, $N = 16384$, $L = 8$, Shift = all.

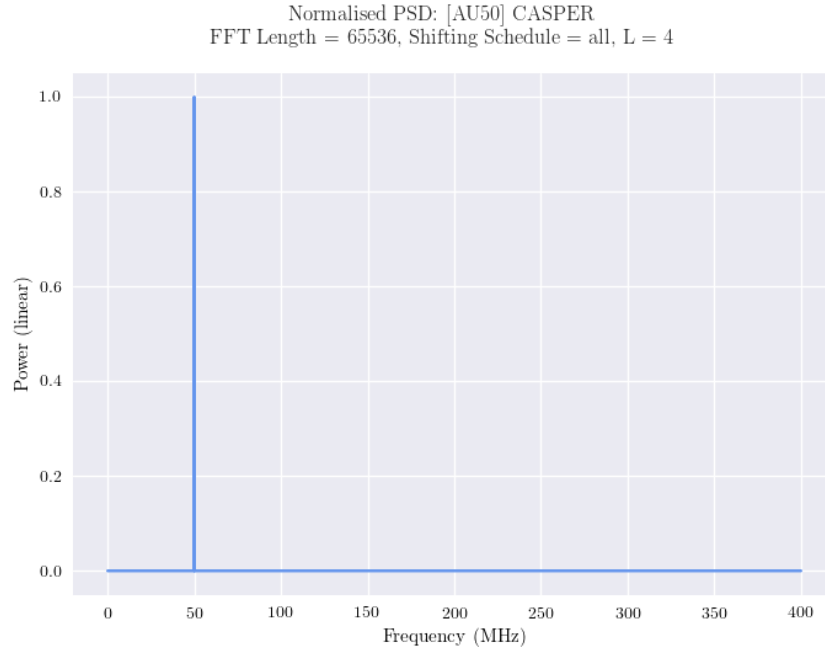


Figure C.32: Normalised PSD of CASPER, $N = 65536$, $L = 4$, Shift = all.

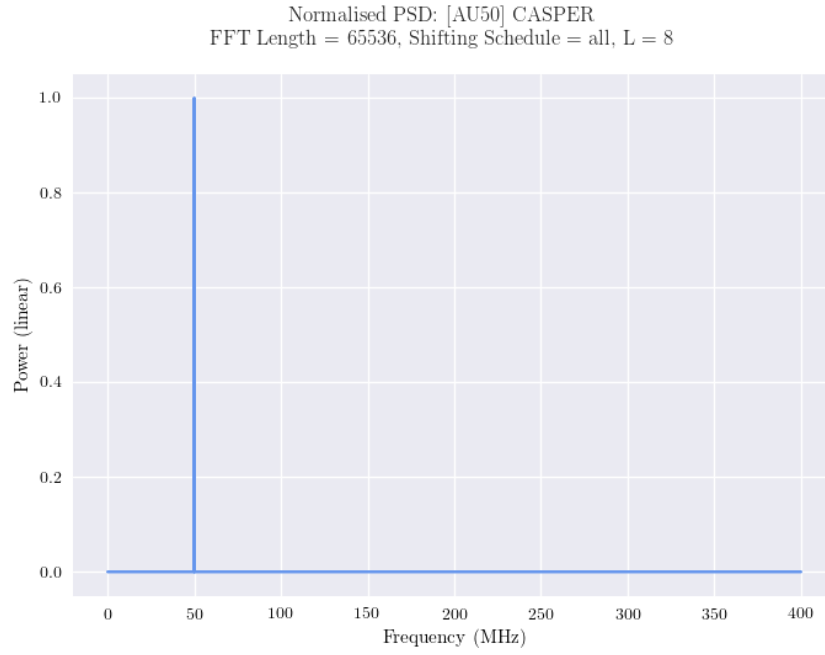


Figure C.33: Normalised PSD of CASPER, $N = 65536$, $L = 8$, Shift = all.

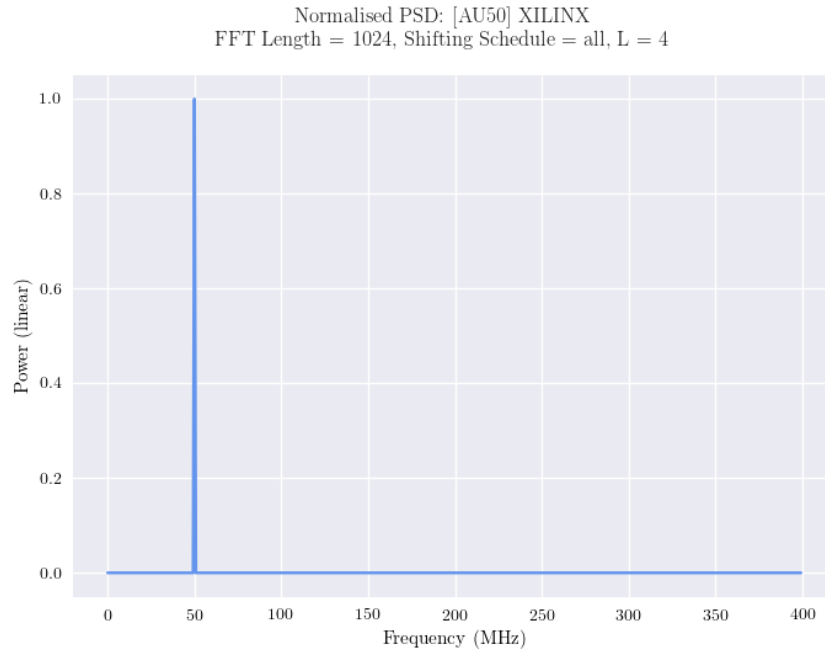


Figure C.34: Normalised PSD of Xilinx, $N = 1024$, $L = 4$, Shift = all.

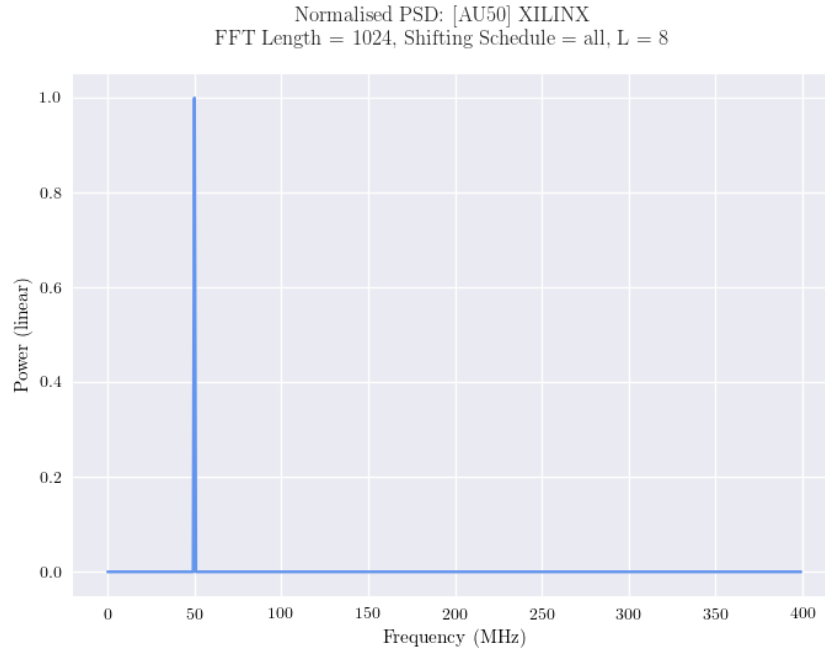


Figure C.35: Normalised PSD of Xilinx, $N = 1024$, $L = 8$, Shift = all.

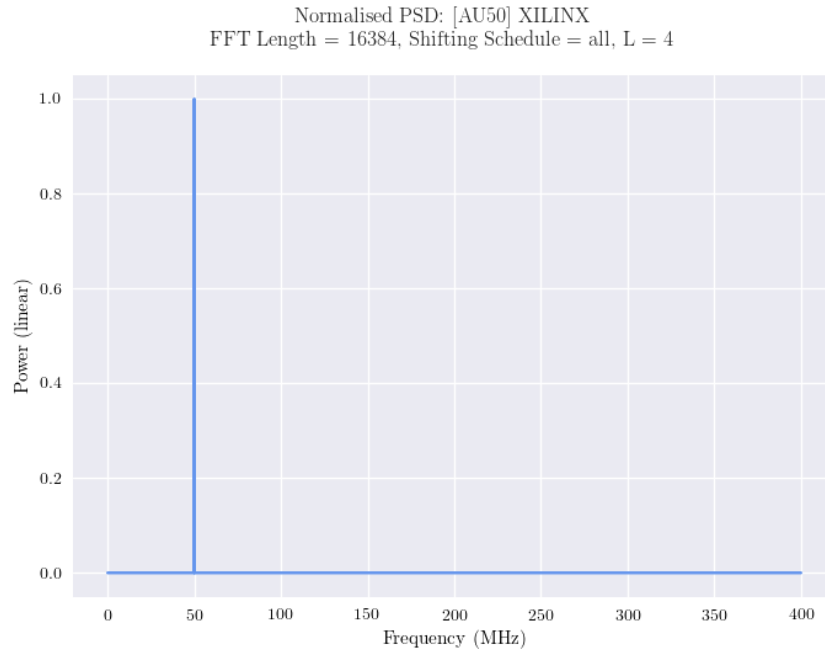


Figure C.36: Normalised PSD of Xilinx, $N = 16384$, $L = 4$, Shift = all.

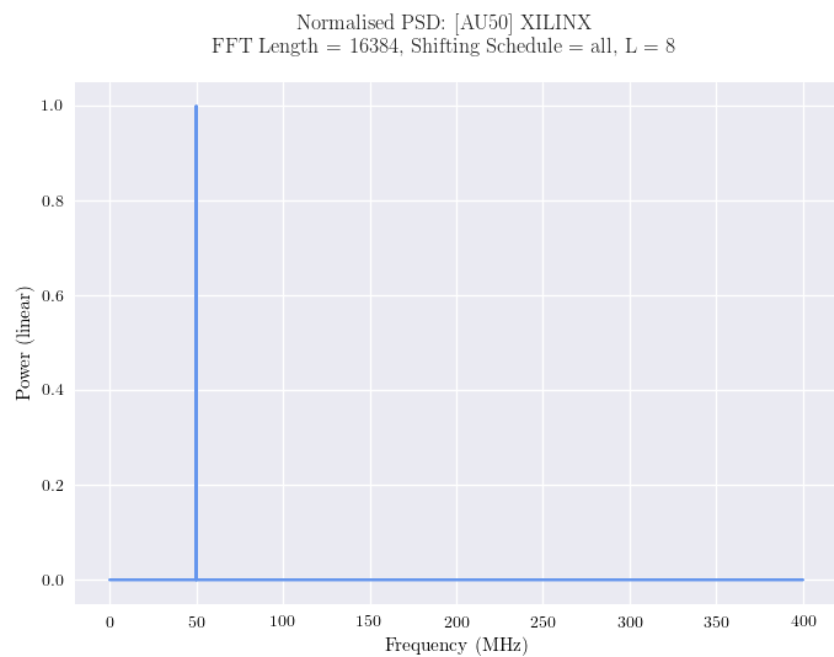


Figure C.37: Normalised PSD of Xilinx, N = 16384, L = 8, Shift = all.

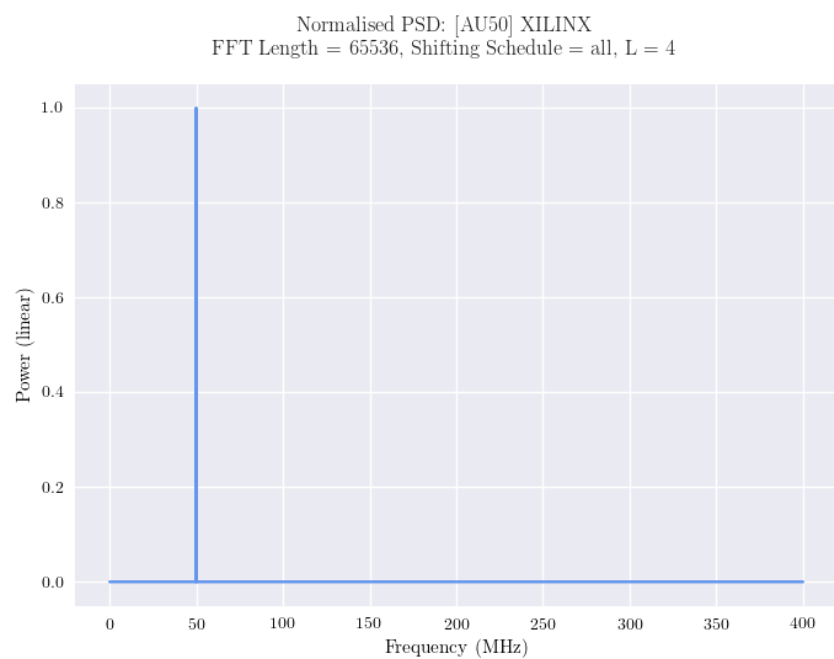


Figure C.38: Normalised PSD of Xilinx, N = 65536, L = 4, Shift = all.

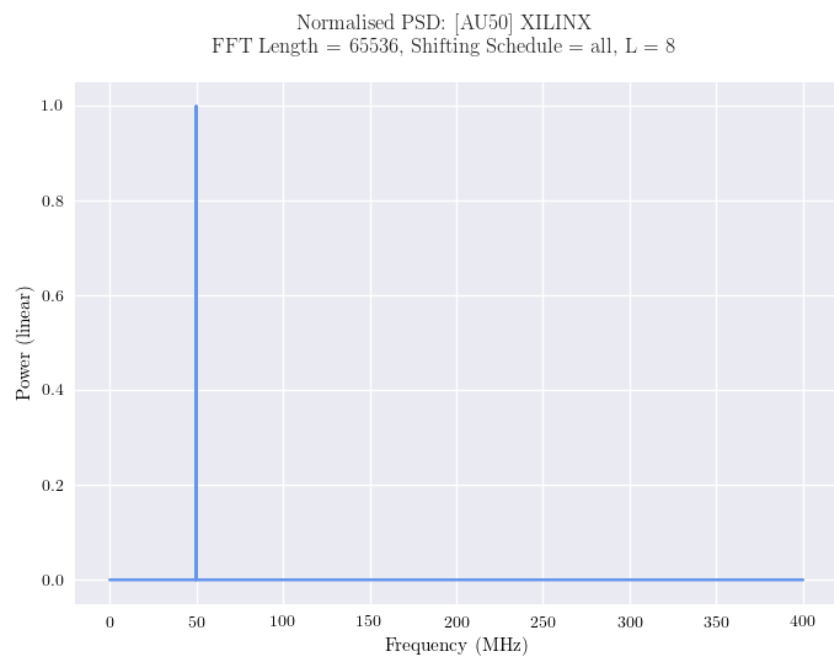


Figure C.39: Normalised PSD of Xilinx, $N = 65536$, $L = 8$, Shift = all.