A SOFT-CORE PROCESSOR ARCHITECTURE OPTIMISED FOR RADAR SIGNAL PROCESSING APPLICATIONS

by

René Broich

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Electronic Engineering)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology
UNIVERSITY OF PRETORIA

December 2013

SUMMARY

A SOFT-CORE PROCESSOR ARCHITECTURE OPTIMISED FOR RADAR SIGNAL PROCESSING APPLICATIONS

by

René Broich

Promoters:	Hans Grobler
Department:	Electrical, Electronic and Computer Engineering
University:	University of Pretoria
Degree:	Master of Engineering (Electronic Engineering)
Keywords:	Radar signal processor, soft-core processor, FPGA architecture, signal-flow characteristics, streaming processor, pipelined processor, soft-core DSP, processor design, DSP architecture, transport-based processor.

Current radar signal processor architectures lack either performance or flexibility in terms of ease of modification and large design time overheads. Combinations of processors and FPGAs are typically hard-wired together into a precisely timed and pipelined solution to achieve a desired level of functionality and performance. Such a fixed processing solution is clearly not feasible for new algorithm evaluation or quick changes during field tests. A more flexible solution based on a high-performance soft-core processing architecture is proposed.

To develop such a processing architecture, data and signal-flow characteristics of common radar signal processing algorithms are analysed. Each algorithm is broken down into signal processing and mathematical operations. The computational requirements are then evaluated using an abstract model of computation to determine the relative importance of each mathematical operation. Critical portions of the radar applications are identified for architecture selection and optimisation purposes.

Built around these dominant operations, a soft-core architecture model that is better matched to the core computational requirements of a radar signal processor is proposed. The processor model is iteratively refined based on the previous synthesis as well as code profiling results. To automate this iterative process, a software development environment was designed. The software development environment enables rapid architectural design space exploration through the automatic generation of development tools (assembler, linker, code editor, cycle accurate emulator / simulator, programmer, and debugger) as well as platform independent VHDL code from an architecture description file. Together with the board specific HDL-based HAL files, the design files are synthesised using the vendor specific FPGA tools and practically verified on a custom high performance development board. Timing results, functional accuracy, resource usage, profiling and performance data are analysed and fed back into the architecture description file for further refinement.

The results from this iterative design process yielded a unique transport-based pipelined architecture. The proposed architecture achieves high data throughput while providing the flexibility that a software-programmable device offers. The end user can thus write custom radar algorithms in software rather than going through a long and complex HDL-based design. The simplicity of this architecture enables high clock frequencies, deterministic response times, and makes it easy to understand. Furthermore, the architecture is scalable in performance and functionality for a variety of different streaming and burst-processing related applications.

A comparison to the Texas Instruments C66x DSP core showed a decrease in clock cycles by a factor between 10.8 and 20.9 for the identical radar application on the proposed architecture over a range of typical operating parameters. Even with the limited clock speeds achievable on the FPGA technology, the proposed architecture exceeds the performance of the commercial high-end DSP processor.

Further research is required on ASIC, SIMD and multi-core implementations as well as compiler technology for the proposed architecture. A custom ASIC implementation is expected to further improve the processing performance by factors between 10 and 27.

OPSOMMING

’N SAGTEKERNPROSESSEERDERARGITEKTUUR WAT VIR RADARSEINPROSESSERINGTOEPASSINGS OPTIMEER IS

deur

René Broich

Promotors:	Hans Grobler
Departement:	Elektriese, Elektroniese en Rekenaar-Ingenieurswese
Universiteit:	Universiteit van Pretoria
Graad:	Magister in Ingenieurswese (Elektroniese Ingenieurswese)
Sleutelwoorde:	Radarseinverwerker, sagtekernverwerker, FPGA-argitektuur, seinvloei-eienskappe, vloeiverwerker, pyplynargitektuur, DSP-sagtekern, DSP-verwerker, vervoergebaseerde verwerker

Die huidige radarseinverwerkerargitekture kort óf prestasie óf buigbaarheid betreffende maklike modifikasie en hoë oorhoofse ontwerptydkoste. Kombinasies van verwerkers en FPGAs word tipies hard bedraad in ’n noukeurig gemete en pyplynoplossing om die vereiste vlak van funksionaliteit en prestasie te bereik. So ’n vasteverwerker-oplossing is duidelik nie vir nuwe algoritmiese berekeninge of vinnige veranderinge tydens veldtoetse geskik nie. ’n Meer buigsame oplossing gebaseer op hoëprestasie-sagtekernverwerkerargitektuur word voorgestel.

Om so ’n verwerkingsargitektuur te ontwikkel, is data- en seinvloei-eienskappe van gemeenskaplike radarseinverwerkingalgoritmes ontleed. Elke algoritme word afgebreek in seinverwerking- en wiskundige berekeninge. Die berekeningvereistes word dan geëvalueer met behulp van ’n abstrakte berekeningsmodel om die relatiewe belangrikheid van elke wiskundige operasie te bepaal. Kritieke gedeeltes van die radartoepassings word geïdentifiseer vir argitektuurseleksie en optimeringdoeleindes.

’n Sagtekern-argitektuurmodel, wat volgens die dominante operasies gebou is en wat beter aan die kernberekeningsvereistes van ’n radarseinverwerker voldoen, word voorgestel. Die verwerkermodel word iteratief verfyn op die basis van die vorige sintese sowel as kodeprofielresultate. Om hierdie herhalende proses te outomatiseer, is ’n sagteware-ontwikkelingomgewing ontwerp. Die sagteware-ontwikkelingsomgewing maak vinnige argitektoniese ontwerpsverkenning van die ruimte deur middel van die outomatiese generasie van ontwikkelinggereedskap (samesteller, binder, koderedakteur, siklus-akkurate emulator/simulator, programmeerder en ontfouter) sowel as platform-onafhanklike VHDL-kode van ’n argitektuurbeskrywinglêer moontlik. Saam met die bord-spesifieke HDL-gebaseerde HAL-lêers word die ontwerpslêers gesintetiseer deur die verkoper-spesifieke FPGA-gereedskap te gebruik en prakties geverifieer op ’n doelgemaakte hoëprestasie-ontwikkelingbord. Tydsberekeningresultate, funksionele akkuraatheid, middele-gebruik, profiele en prestasiedata word ontleed en teruggevoer in die argitektuurbeskrywinglêer vir verdere verfyning.

Die resultaat van hierdie iteratiewe ontwerpsproses is ’n unieke vervoergebaseerde pyplynargitektuur. Die voorgestelde argitektuur bereik hoë deurvoer van data en verleen terselfdertyd die buigsaamheid wat ’n sagteware-programmeerbare toestel bied. Die eindgebruiker kan dus doelgemaakte radar-algoritmes in sagteware skryf, eerder as om dit deur ’n lang en komplekse HDL-gebaseerde ontwerp te doen. Die eenvoud van hierdie argitektuur lewer hoë klokfrekwensie en deterministiese reaksietye en maak dit maklik om te verstaan. Verder is die argitektuur skaleerbaar wat prestasie en funksionaliteit betref vir ’n verskeidenheid vloeiverwerkerverwante toepassings.

In vergelyking met die Texas Instruments C66x DSP-kern was daar ’n afname in kloksiklusse met ’n faktor van tussen 10,8 en 20,9 vir die identiese radar op die voorgestelde argitektuur oor ’n verskeidenheid tipiese bedryfstelselparameters. Selfs met die beperkte klokspoed wat haalbaar is op die FPGA-tegnologie, oorskry die voorgestelde argitektuur die prestasie van die kommersiële hoëspoed-DSP-verwerker.

Verdere navorsing is nodig oor ASIC, SIMD en multi-kern-implementering, sowel as samesteller-tegnologie vir die voorgestelde argitektuur. Daar word voorsien dat ’n pasgemaakte ASIC-implementering die verwerkingprestasie tussen 10 en 27 maal sal verbeter.

LIST OF ABBREVIATIONS

AAU	Address arithmetic unit
ABI	Application binary interface
ADL	Architecture description language
AESA	Active electronically scanned array
AGU	Address generation unit
BRF	Burst repetition frequency
BRI	Burst repetition interval
CAM	Content-addressable memory
CFAR	Constant false alarm rate
CLB	Configurable logic block
CORDIC	Coordinate rotational digital computer
CPI	Coherent processing interval
CW	Continuous wave
DDS	Direct digital synthesizer
DFG	Data flow graphs
DFT	Discrete Fourier transform
DIF	Decimation-in-frequency
DIT	Decimation-in-time
DLP	Data-level parallelism
DPU	Data processing units
DSP	Digital signal processor
EPIC	Explicitly parallel instruction computing
EW	Electronic warfare
FM	Frequency modulating
FU	Functional unit
GPGPU	General purpose computing on GPUs
HDL	Hardware descriptive language
HLS	High-level synthesis
HRR	High range resolution
HSDF	Homogeneous synchronous data flow

I/Q	In-phase and quadrature
IDFT	Inverse discrete time Fourier transform
IF	Intermediate frequency
ILP	Instruction-level parallelism
IP	Intellectual property
LE	Logic elements
LFM	Linear frequency modulated
LNA	Low noise amplifier
LUT	Look up tables
LWDF	Lattice wave digital filter
MMU	Memory management unit
MTI	Moving target indication
NCI	Non-coherent integration
NISC	No instruction set computer
NLE	Noise-level estimation
OISC	One instruction set computer
PRF	Pulse repetition frequency
PRI	Pulse repetition interval
RCS	Radar cross section
RSP	Radar signal processor
SAT	Summed area table
SDF	Synchronous data flow
STA	Synchronous transfer architecture
STC	Sensitivity time control
SW	Sliding window
TLP	Thread-level parallelism
TTA	Transport triggered architecture

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

In a modern radar system, the architecture of the radar signal processor is one of the most important design choices. The amount of useful information that can be extracted from the radar echoes is directly related to the computational performance that the radar signal processor can deliver. To meet the required real-time performance, field programmable gate arrays (FPGAs) have become the most popular technology because of their re-programmability and resource intensive nature. Radar signal processing algorithms commonly exhibit parallelism and limited cyclic dependencies, making the FPGA a highly suitable technology. For such cases, considerable performance improvements can be achieved over conventional microprocessor and dedicated coprocessor solutions [1, 2].

However, with evolving requirements in the constantly changing radar processing field (largely due to radar technology advances and electronic warfare (EW) counter-measures), an FPGA lacks flexibility in terms of ease of modification and design time overhead. The design cycle for developing the radar signal processor in a hardware descriptive language (HDL), is often much longer than the period between consecutive releases of new specifications or requirements. Also, new algorithms or algorithm combinations based on MATLAB simulations or sequential C program implementations are typically difficult or cumbersome to convert to the inherently concurrent HDLs, further widening the gap between the theoretical concept and the practical implementation. The repetitive HDL-based redesign is time consuming and clearly not feasible in the long run, especially for small design changes made during field tests.

Soft-core processors (embedded on the FPGA), digital signal processors (DSPs) and conventional micro-processors shift the redesign methodology to a software based approach, but generally lack the real-time performance required for processing radar data.

It is clear that a more versatile solution is required. A new soft-core processing architecture that is optimised for radar signal processing applications is proposed. Since the pulse-Doppler radar is most commonly used for detecting, tracking, and classification functions in modern radar systems, it will be the main focus of this investigation.

1.2 MOTIVATION

In the past the focus of the radar signal processor design has simply been to achieve a desired level of functionality and performance. Combinations of multiple application specific integrated circuits (ASICs), DSPs, FPGAs, and coprocessors were hard-wired together to form a fixed solution for a specific task. Little or no emphasis has been placed on attempts to establish a versatile, flexible and easy to use processing architecture for radar processing applications.

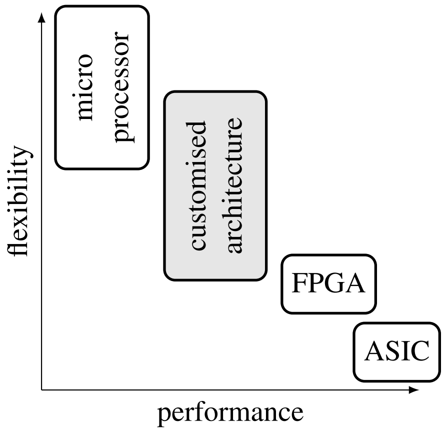


Figure 1.1: Bridging the gap between flexibility and performance

From an FPGA perspective, challenges concerning rapid implementation and high-level optimisation of algorithms are posed. Some notable attempts have been made to improve the overall design process of FPGA systems; software abstraction layers [3], library based tool chains [4], rapid implementation tool-suites for translating high-level algorithms into HDL [5, 6] and high-level synthesis tools [7–9]. Table 1.1 summarises these different FPGA mapping approaches and compares their relative characteristics.

Clearly none of these mapping approaches achieve the right balance between speed, flexibility and ease of implementation. Simple parameter or functional changes are not in-field changeable and require hours of re-compilation and debugging because of latency mismatches or timing closure problems. In the rapidly changing field of radar processing, the implementation of custom algorithms

Table 1.1: FPGA Algorithmic Mapping Techniques

Category	**Comments**
HDL synthesis Tools (e.g. Xilinx ISE, Altera Quartus II, Synopsis)	High performance, low latency, time consuming HDL design; exact cycle by cycle behaviour needs to be specified, requires specialised expertise, difficult to design for functional scalability and flexibility, long compile / synthesis times, timing closure problems, complex and iterative debugging phases, design often relies on vendor specific primitive and interface blocks
Graphical block diagram tools (e.g. Mentor Graphics HDL Author)	Vendor independent, code visualisation as modular blocks, library based design, graphical state machine editor, facilitates design collaboration, poor integration with design tools, same issues as with HDL synthesis tools
IP libraries / generators (e.g. OpenCores [10], Xilinx CORE Generator System [11], Altera MegaWizard Plug-In Manager [12])	Library of IP cores that can be generated and reused to increase design productivity
Soft-core processor (e.g. NIOS II, MicroBlaze, LEON)	Sequential execution model; software is more flexible than HDL, easy to program with high-level languages such as C/C++, quick compilation and reprogramming, easy to debug with breakpoints and code stepping, limited FPGA resource usage, aimed at general purpose computing and control applications, can be accelerated by extending the ISA or adding custom logic into their memory space, lacks DSP performance, not real-time, high latency, limited performance scalability
C to HDL compiler (e.g. Xilinx Vivado HLS, Synopsys Synphony C Compiler, Mentor Graphics Catapult C, Cadence C-to-Silicon)	C-language is not well-suited for mapping FPGA hardware; lacking constructs for specifying concurrency, timing, synchronisation. Restructuring the C code is necessary for the tool to detect parallelism, poor integration with design tools, same issues than with synthesis tools after HDL creation
OpenCL to HDL compiler (e.g. Altera SDK for OpenCL [13] [14])	FPGA used as an accelerator to a host PC processor, experimental
Transaction-level modelling (e.g. SystemC)	Early architectural exploration, C++ for simulating the entire system, functional verification, untimed specification, same issues than with synthesis tools after HDL creation
Dataflow to HDL (e.g. ORCC [15])	Compiles RVC-CAL actors and XDF networks to any source code (C/Java/VHDL/Verilog), needs to be synthesized using traditional tools
Visual design tools (e.g. National Instruments LabView for FPGA, MATLAB/Simulink plug-ins: Xilinx System Generator, Altera DSP Builder)	Graphical interconnection of different DSP blocks, multirate systems, fused datapath optimisations. Difficult to describe and debug complex designs, no design trade-off optimisation, lacking support for: parameterised designs, parallelism, control signals, interfaces

remains cumbersome with long design cycles on current hardware architectures, often requiring system level changes even for small modifications. It is clear that a high-level programming model that simplifies the algorithmic implementation for a radar signal processor, is required. Given the flexibility limitations of current radar signal processors, it is theorised that a better solution based on a custom processor and a software development environment is possible. The proposed architecture is envisioned to consist of multiple low-level functional units applicable to the various radar signal processing algorithms. Each of these would be configured and interconnected from a high-level, thus bridging the gap between performance and flexibility as shown in Fig. 1.1.

1.3 OBJECTIVES

The research objective of this study is to develop a framework that captures common low-level processing constructs that are applicable to a pulse-Doppler radar signal processing architecture. These low-level constructs will be combined into a new processing architecture that exhibits the required performance for radar signal processing, and is programmable from a software environment. This architecture would have to support numerous variations of the generic radar algorithms as outlined in Appendix A, as well as a portfolio of signal processing operations as listed in Appendix B. For example, the Doppler processing requirements for a high range resolution (HRR) radar system would be significantly different compared to a standard radar system. Some of the research questions that will be addressed are listed below:

- What does radar signal processing entail from a processing perspective?
- What performance enhancement techniques do current processors employ?
- Which characteristics make an architecture optimal or well-suited for a specific application?
- What procedures are followed to design, optimise, debug and verify an architecture?
- What is the optimal architecture for a radar signal processor?

The aim is to develop a deterministic real-time soft processor architecture that is capable of fully utilizing the FPGA in the radar signal processing paradigm. Emphasis will be placed on the software configurable (embedded software rather than FPGA firmware) aspects, while balancing computational performance. Algorithms written purely as FPGA firmware will almost certainly outperform any algorithms written for a soft-core architecture residing on that same FPGA. The key difference of the proposed architecture compared to conventional FPGA and DSP based solutions however, is a much better match of the architecture to the core computational as well as the system level architectural needs of a radar signal processor.

It should be clear that the focus of this study is not to develop a radar system, but rather to design and optimise a soft-core architecture such that radar systems can easily be implemented from a high-level. The end user would thus be able to write and test new algorithms as embedded software rather than FPGA firmware.

1.4 CONTRIBUTION

Among the vast amount of implementation details relating to radar signal processors (RSPs), there is very limited material covering an architecture that is programmable and reconfigurable from a high-level software environment. Using the architectural knowledge gained through the iterative design and verification process, a higher level framework is presented. This framework is employed to explore the design space of an optimal RSP architecture that can be implemented as a soft-core processor on an FPGA. The results of this study, and specifically the architecture of the proposed soft-core processor, will be submitted for publication in the IEEE Transactions on Aerospace and Electronic Systems journal. The background study analysing the processing and computational requirements of a typical radar signal processor was presented at the 2012 IEEE Radar Conference in Atlanta, GA, USA [16].

The following list summarises the primary contributions made by this work:

- a computational requirement analysis of radar algorithms,
- a framework for processor architecture design,
- a software tool chain with assembler and simulator,
- the *FLOW* programming language, and
- the proposed processing architecture for software-defined radar applications.

1.5 OVERVIEW

The structure of this document outlines the approach that was followed in obtaining the optimised architecture. Chapter 1 identifies the problem and provides a brief literature overview, emphasising the need for a reprogrammable radar processing architecture.

Chapter 2 introduces the problem from a signal processing perspective. Firstly, the background material relating to radar systems is covered. Typical radar signal processing algorithms are discussed and broken down into common digital signal processing operations. The data and signal flow characteristics of the various radar algorithms are then quantitatively analysed based on a test-case, such that operations can be ranked according to their relative importance for optimisation purposes.

Various computational architectures and processing technologies are discussed and compared in Chapter 3, examining their applicability to radar systems. Each of these processing architectures are potential architectural templates, which could be implemented on an FPGA.

Chapter 4 proposes a suitable architectural template that is capable of handling the processing requirements established in Chapter 2. The architecture is discussed from a higher level without the implementation details of the operations and algorithms. The optimisation procedure and implementation alternatives for the operations and algorithms on the proposed architecture are then discussed in Chapter 5.

In Chapter 6 these implementation options are amalgamated into a unified architecture. Trade-offs and algorithm combinations are discussed from a performance and resource usage perspective. The final architecture is presented and the hardware implementation alternatives are evaluated.

The proposed architecture is verified and quantified in Chapter 7. Processing performance, FPGA resource usage, and ease of implementation are discussed and compared against other architectures. Finally, the advantages and shortcomings of the proposed architecture are discussed in Chapter 8, followed by a discussion on alternative solutions and suggestions for further research.

The following table outlines the notations that are used throughout the document for variables and constant expressions.

Table 1.2: Notation Convention

	Description		**Description**
N	Number of samples per PRI	P	Number of pulses per burst / CPI
L	Filter length	T	Number of transmit samples
K	FFT point length	R	Range
f_s	Sampling frequency	T_s	Sampling interval
f_d	Doppler frequency	τ	Transmit pulse width
f_p	Pulse repetition frequency	T_p	Pulse repetition interval

CHAPTER 2

COMPUTATIONAL ANALYSIS OF RADAR ALGORITHMS

2.1 OVERVIEW

In order to propose an optimised soft-core processing architecture, an in-depth computational analysis of the algorithms used in pulse-Doppler radar systems is necessary. As a first step, a typical pulse Doppler radar signal processor is implemented as a MATLAB [17] model. This high-level model of the radar system:

1. Provides a mechanism for analysing data as well as signal-flow characteristics of the radar algorithms,
2. Provides an overview of the entire radar system and its processing as well as memory requirements,
3. Establishes a baseline to verify the correctness of the implementation during development, and
4. Sets a benchmark for comparing the performance of the implementation against.

The radar signal processor algorithms are then broken down into mathematical and signal processing operations (kernels). Kernels that exhibit high recurrence or significant computational requirements are identified and sorted according to their relative processing time and importance. Kernels with a high computational density are given priority in the optimisation process in later chapters.

The first step thus involves an analysis of a typical radar signal processor. An overview of the radar system as a whole is provided in the next section, identifying the various processing algorithms that are typically used in such a radar signal processor.

2.2 RADAR OVERVIEW

A radar system radiates electromagnetic waves into a certain direction of interest, analysing the received echo once the signal has bounced back. The analysed information can be used for target detection, tracking, classification or imaging purposes. Radar systems can be divided into two general classes; namely continuous wave (CW) and pulsed radars.

2.2.1 Continuous Wave Radar Systems

In the CW configuration, the radar system simultaneously transmits and receives. Fig. 2.1 shows a basic CW radar with a dual antenna configuration.

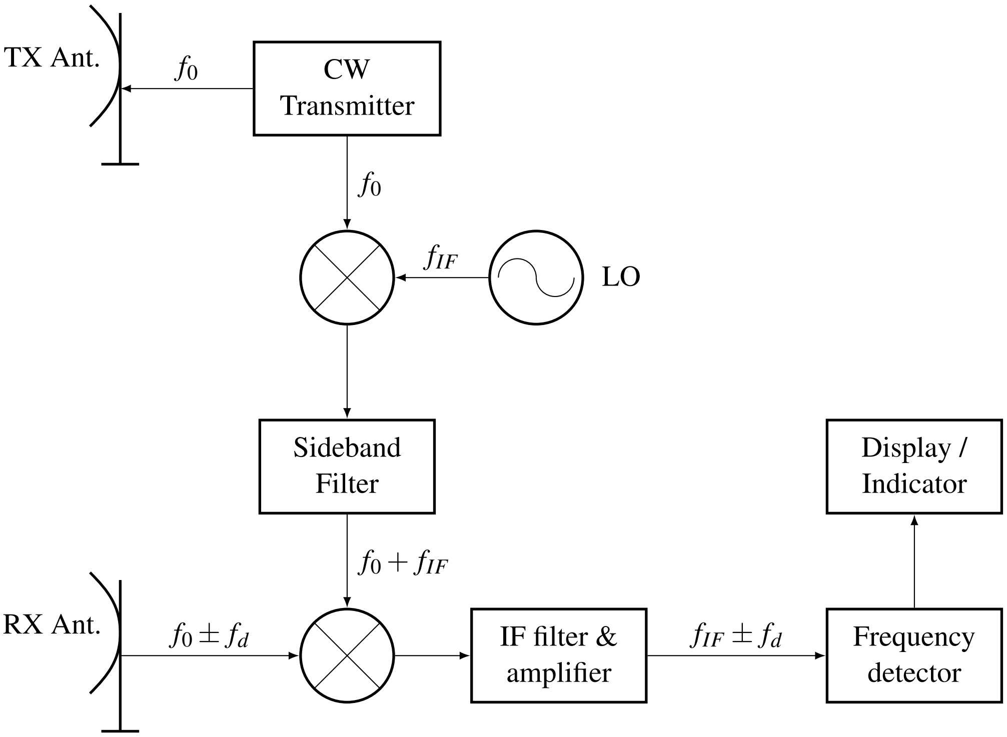


Figure 2.1: CW radar block diagram

Although a single antenna monostatic configuration is theoretically possible for CW radar systems, the transmitter leakage power is typically several magnitudes larger than what can be tolerated at the receiver, making such systems impractical. Additionally, the received echo-signal power is typically in the order of 10^{-18} or less that of the transmitted power. Regardless, complete isolation between transmitter and receiver is practically not realisable in monostatic configurations, even with separate antennas. It is thus up to the radar to differentiate between transmitted and received signals.

In the case of a moving target, the received echo will have a Doppler frequency shift relative to the transmitted signal of

$$f_d = \Delta f = \frac{2v_r f_0}{c}, \tag{2.1}$$

where f_0 is the transmitted frequency, c is the propagation velocity (3×10^8 m/s in free space), and v_r is the relative radial velocity between target and radar.

Thus this frequency shift can be used to detect moving targets, even in noisy environments and with low signal returns. In the implementation depicted in Fig. 2.1, the transmitted signal is a pure sinusoid at a radar frequency f_0. The received signal is mixed down to the intermediate frequency (IF) of the local oscillator (LO). A bank of filters acts as a frequency detector and outputs the presence of a moving target to a display. To measure target distance however, a change in transmitter waveform is required as a timing reference. This can be accomplished by frequency modulating (FM) the carrier, as with FM-CW radars.

These FM-CW radars are relatively simple, low power systems often used for proximity fuses, altimeters or police speed-trapping radars. Most commercial as well as military radar systems are pulsed radars; more specifically pulse-Doppler radars as discussed in the next section.

2.2.2 Pulse Radar Systems

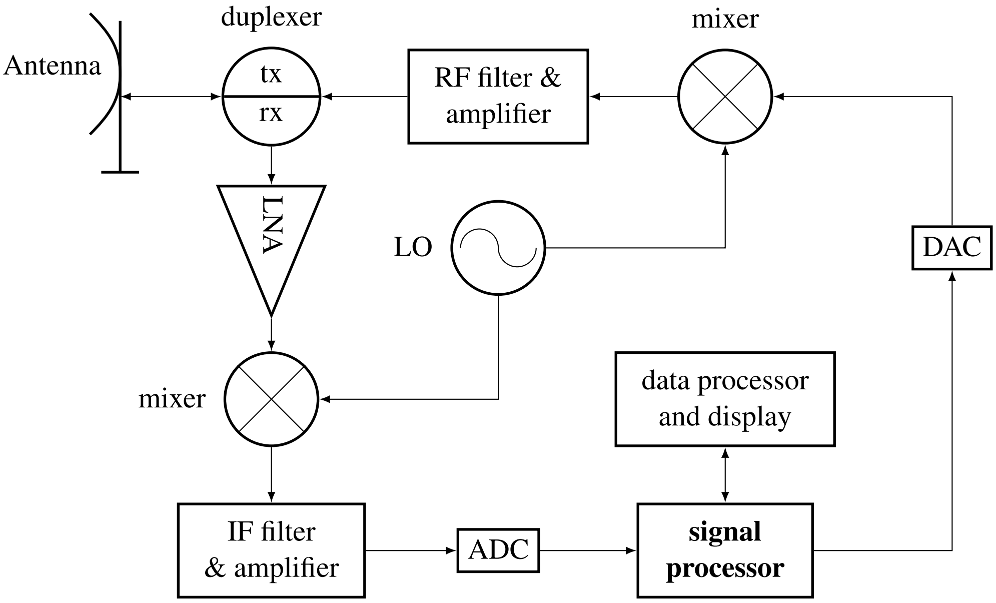


Figure 2.2: Pulse radar block diagram

A simplified pulse radar system is shown on the previous page in Fig 2.2. Pulses that are to be transmitted are converted to an analogue signal, up-converted to the radar frequency, amplified and filtered. The duplexer is set to transmit mode and the pulse is radiated at the speed of light in the direction the antenna is facing. After the transmission is complete, the duplexer is switched back to receive mode. The receiver now waits for a returned signal. The time it takes for a signal to travel towards the target, reflect off the target and return back to the antenna is thus

$$t_0 = \frac{2R}{c}, \tag{2.2}$$

where c is the speed of light and R is the range to the target. Once a radar echo is received, it is amplified by the low noise amplifier (LNA) and down-converted to some IF frequency (typically in the centre of the ADC band) by the mixing and filtering operations. In practical implementations, there are usually more IF up and down-conversion stages. Optionally the received signal may also be fed through a sensitivity time control (STC), a variable attenuator to provide increased dynamic range (by decreasing the effects of the $1/R^4$ return power loss) and receiver saturation protection from close targets. The mixing operations require the LO to be extremely stable in order to maintain phase coherence between transmitted and received pulses; a vital prerequisite for pulse-Doppler radars and most modern radar algorithms. The IF signal is then amplified and passed to the analogue to digital converter (ADC) of the radar signal processor (RSP). When phase coherency between transmitter and receiver is maintained from sample to sample, the pulse radar is known as a pulse-Doppler radar, because of its ability to detect Doppler frequencies by measuring phase differences. Thus the ADC samples both in-phase (I) and quadrature (Q) channels of the received echo in a pulse-Doppler radar. This can be accomplished by oversampling and performing digital I/Q demodulation (e.g. a Hilbert transform) or by using two ADC's and a 90 degree out of phase analogue mixing operation as shown in Fig. 2.3.

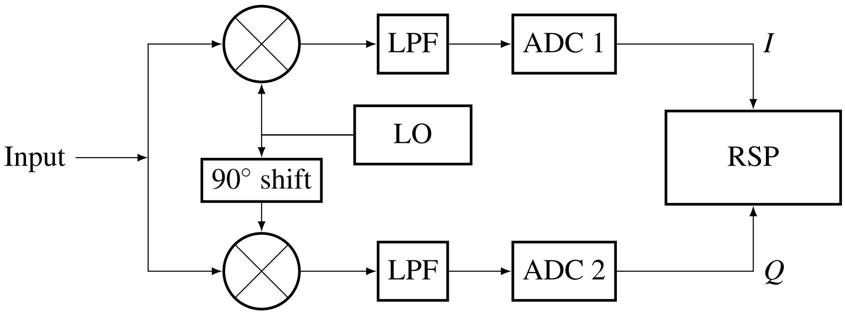


Figure 2.3: Conventional quadrature channel demodulation

A pulsed radar system sends out numerous pulses at a predetermined pulse repetition frequency (PRF); typically between 100 Hz and 1 MHz. Fig. 2.4 shows how its inverse, the pulse repetition interval (PRI), represents the time between consecutive transmissions. The transmit pulse width τ is typically 0.1 to 10 microseconds (μs) and the receive / listening time is typically between 1 microsecond and tens of milliseconds.

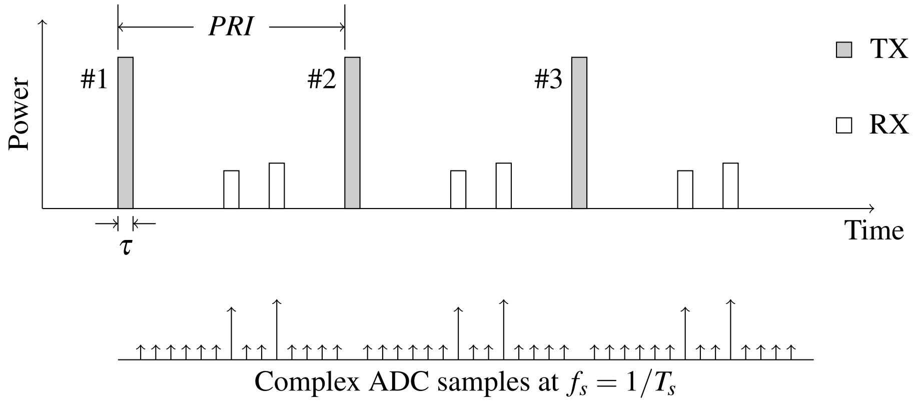


Figure 2.4: Pulsed radar waveform

During the receiving time, the ADC collects numerous "fast-time" range samples for each pulse number. If a transmitted pulse (at the carrier frequency f_c) of the form

$$x(t) = A\cos(2\pi f_c t + \theta), \qquad -\frac{\tau}{2} < t < \frac{\tau}{2} \tag{2.3}$$

is returned from a hypothetical target at range R_0, the received pulse $y(t)$ would be

$$\begin{aligned} y(t) &= x(t - \frac{2R_0}{c}) && (2.4)\\ &= A'\cos(2\pi f_c t + \theta - \frac{4\pi R_0}{\lambda}), \qquad -\frac{\tau}{2} + \frac{2R_0}{c} < t < \frac{\tau}{2} + \frac{2R_0}{c}. && (2.5)\end{aligned}$$

In order to determine the unknown amplitude A' (modelled by the radar range equation [18]) and phase shift $\theta' = -(4\pi R_0)/\lambda$, the complex sampling and demodulation techniques of Fig. 2.3 are employed to remove the carrier frequency $\cos(2\pi f_c t + \theta)$. With $y(t)$ as the input, the discrete time analytic signal $y[n]$ becomes

$$\begin{aligned} y[n] &= y_I[n] + j \cdot y_Q[n] && (2.6)\\ &= A'\cos(-\frac{4\pi R_0}{\lambda}) + j \cdot A'\sin(-\frac{4\pi R_0}{\lambda}) && (2.7)\\ &= A'e^{-j\frac{4\pi R_0}{\lambda}}, \qquad n = 0, ..., N-1 && (2.8)\end{aligned}$$

where n is the sample number corresponding to the time of $t = nT_s$. Note how the phase of the analytic signal is directly proportional to the range of the target. A change in range by just $\lambda/4$ changes the phase by π; a range increment of barely 7.5 mm in an X-band radar. This capability is key to most modern radar algorithms such as Doppler processing, adaptive interference cancellation, and radar imaging.

The complex I/Q samples of the analytic signal are stored in a matrix with pulses arranged in rows beneath each other as shown in Fig. 2.5.

Pulse # (slow time)	1	2	3	4	5	6	7	8	9	10	11	12	13	**N**
P	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$5\angle 30^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$6\angle 50^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$
2	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$5\angle 20^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$6\angle 50^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$
1	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$5\angle 10^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$6\angle 50^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$	$0\angle 0^\circ$

Range bin / Sample number
(fast time)

Figure 2.5: Data matrix: range-pulse map

Each sample in the matrix can now be addressed with $y[p][n]$, where p is the pulse number and n is the range sample. The number of pulses that are stored together are determined by the coherent processing interval (CPI), sometimes also referred to as dwell, pulse burst or pulse group. Typical CPIs are chosen to be between 1 and 128 pulses (P), consisting of between 1 and 20000 range samples (N). Each column in the CPI matrix represents different "slow-time" samples (separated by the PRI) for a specific discrete range increment (often called a range bin) corresponding to a range of

$$R_n = \frac{cnT_s}{2}, \tag{2.9}$$

where n is the sample number and T_s is the sampling interval. The two hypothetical targets introduced in Fig. 2.4 and Fig. 2.5 are clearly visible at range samples 7 and 10. At an ADC sampling frequency of $f_s = 50$ MHz, these would represent targets at a range of 21 and 30 m with each range bin covering 3 m of distance (provided the pulse width τ is short enough to isolate consecutive targets or pulse compression techniques are used). Note how the phase is changing in the first target; a trait of a moving target because of the Doppler shift.

To extract the required information in the presence of noise, clutter, and interference (intentional [e.g. jamming] or unintentional [e.g. from another radar system]), the pulse-Doppler RSP performs various functions as discussed in the next section.

2.2.2.1 Pulse-Doppler Radar Signal Processor

The RSP of a pulse-Doppler system is responsible for both waveform generation as well as analysis of the returned signal as shown in Fig. 2.6. In this case a monopulse antenna configuration is depicted, however a variety of different antenna configurations (e.g. phased array, bi-static, active electronically scanned array (AESA), or multiple-input multiple-output (MIMO)) could be used. The transmitter side of the RSP simply generates the desired pulses (usually linear frequency or phase modulated) at the pulse repetition frequency set by the timing generator, and passes them to the digital to analogue converter (DAC).

On the receiver side, the functionality of the radar signal processor is typically divided into the front-end streaming processing, and the processing that is done on bursts of data. Although only the summation (Σ) channel is shown, the processing requirements are similar on all three of the monopulse channels.

The streaming radar front-end processing is usually very regular and independent of the data. It includes basic signal conditioning functions, digital IF down-conversion, in-phase / quadrature component demodulation (typically a Hilbert transformer), filtering, channel equalisation, and pulse compression (binary phase coding or linear frequency modulation). Although these functions remain the same for most applications, samples need to be processed in a high-speed stream as the data from the ADC becomes available. With sampling rates as much as a few gigahertz in some cases, this type of data independent processing can demand computational rates in the order of 1 to 1000 billion operations per second (1 gigaops (GOPS) to 1 teraops (TOPS)) and is thus often implemented in ASICs, FPGAs or other custom hardware.

Pulse-Doppler processing, ambiguity resolution, moving target indication (MTI), clutter cancellation, noise level estimation, constant false alarm rate (CFAR) processing, and monopulse calculations are all done on bursts of data in pseudo real-time. After each transmitted pulse, received bursts of data are stored and grouped together for processing. These algorithms are commonly changed with new requirements, and many different implementation variations each with their own advantages and disadvantages exist.

The back-end processing of a radar system performs higher-level functionality such as clustering, tracking, detecting, measuring, and imaging. These functions make decisions about what to do with the information extracted from the received radar echoes, and are not as regular as the front-end

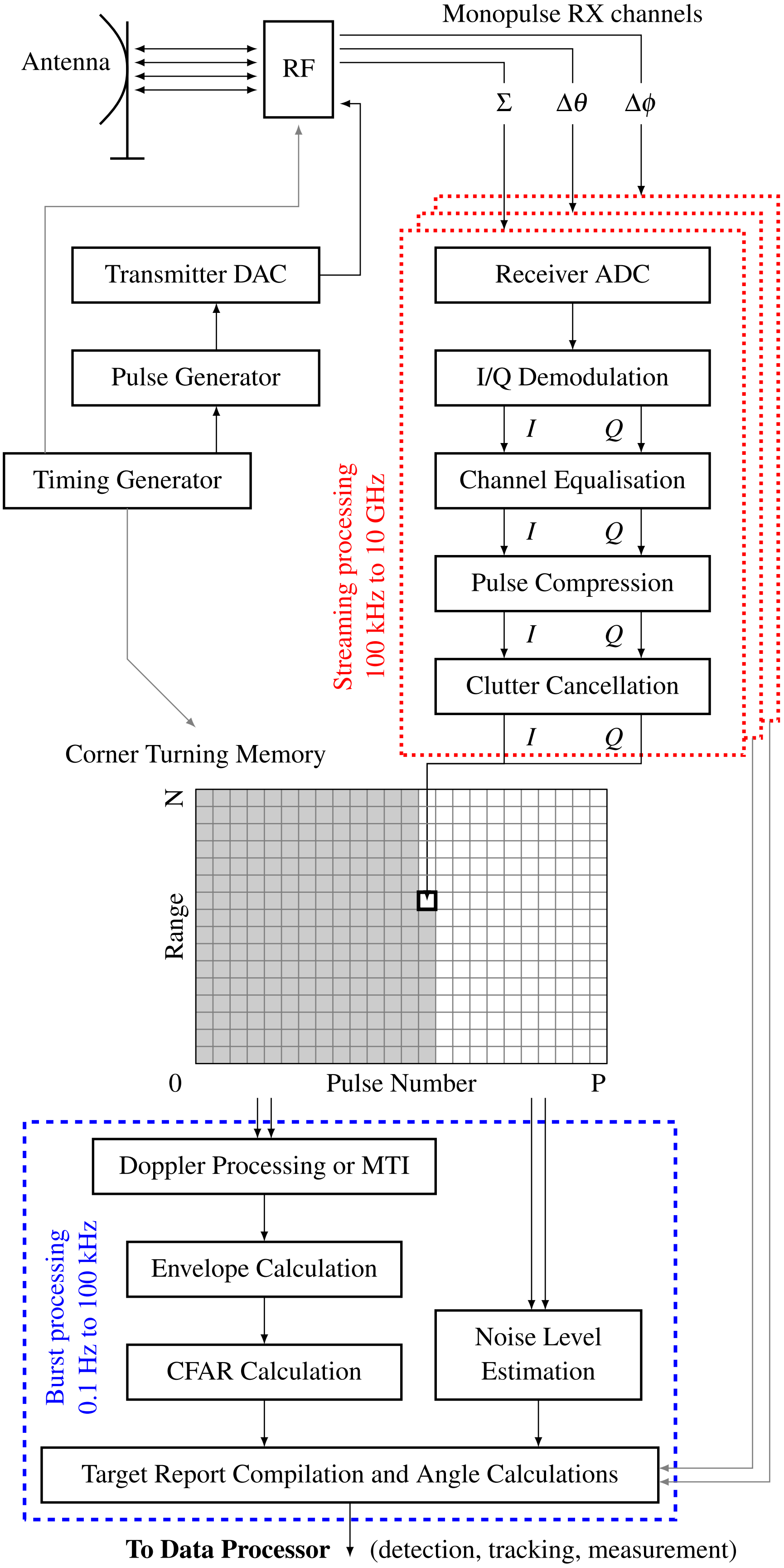


Figure 2.6: Radar signal processor flow of operations

processing. The data throughput for this processing level is much more limited so that almost any general-purpose processor or computer can be used for such a task. The radar processing requirements from front- to back-end processing therefore reduce in computational rate and regularity, but increase in storage requirements for each successive stage. Refer to Appendix A for more details on each of the algorithms in Fig. 2.6.

Note that Fig. 2.6 is just one possible way of implementing a simple RSP. Many variations to each of the algorithms exist and new ones are constantly being added or modified with radar technology advances. For example, MTI is typically only implemented when the computational requirements for pulse-Doppler processing cannot be achieved. Also, the order of the algorithms may be changed; clutter cancellation could be performed before the corner turning memory operation or even after Doppler processing. The next section examines these radar algorithms from a computational perspective, identifying the throughput and data-rate requirements.

2.3 COMPUTATIONAL REQUIREMENTS

This section analyses the various radar algorithms from a computational perspective, identifying the relative importance of each mathematical operation. To determine these processing requirements, the radar signal processor in Fig. 2.7 is used as a test-case.

Although only some of the many implementation variations are shown in the test-case, it represents a variety of mathematical operations that are typically found in RSPs and many of the operations also apply to the higher level data processing. Different implementation options for each algorithm are shown in the horizontal direction, while the radar algorithm stages are listed in the vertical direction. For example, I/Q Demodulation could be implemented as a standard mixing operation, as a simplified mixing operation or as a Hilbert filter. Similarly, amplitude compensation, frequency compensation or no compensation could be used for the channel equalisation stage.

The typical process of optimising a processor for a specific application is to run the application code through a software profiler on the target architecture. The profiling tool identifies hotspots and analyses the call graphs to determine the most time consuming functions. These processing intensive critical portions are then optimised either by instruction set extensions, parallelisation or hardware acceleration.

In this case such an approach would be suboptimal, since the processing architecture is yet to be determined. Profiling radar signal processing applications on a standard personal computer (PC) or

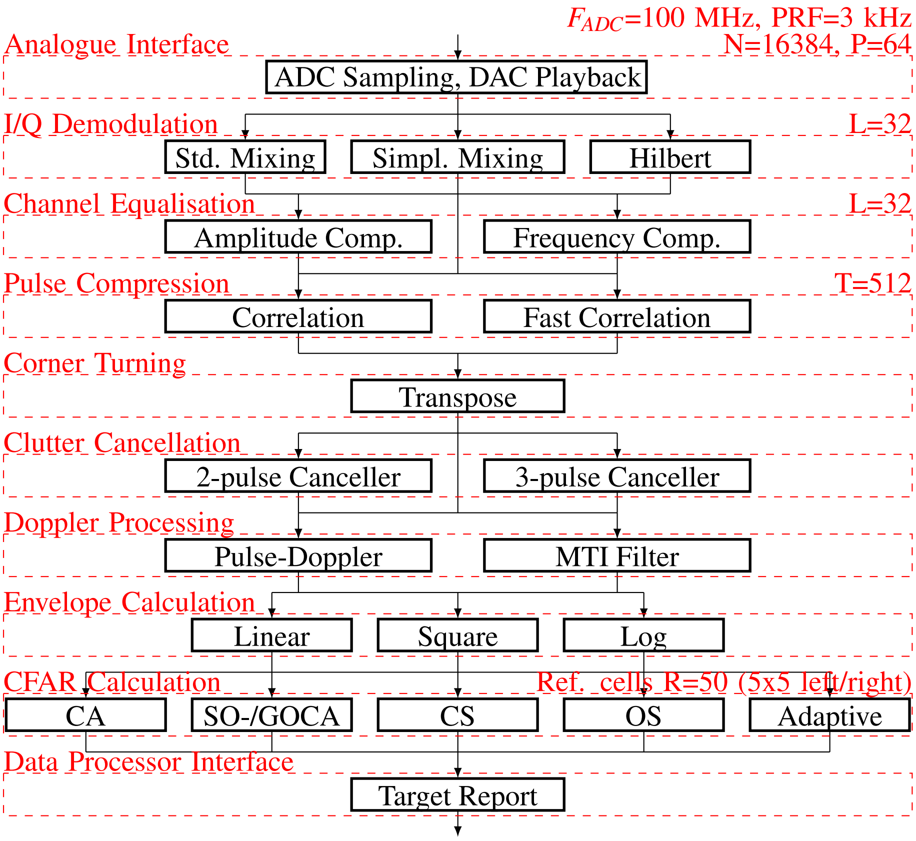


Figure 2.7: RSP test-case flow of operations

reduced instruction set computing (RISC) processor would only achieve feasible results if such a processor is used as the architectural template. Since the architectural template should be designed to cater for the computational requirements, an analytical approach is used instead.

Determining the computational requirements of an application is an important step in the architectural design process. Rather than analysing the complexity or growth rate of each algorithm (based on big O-notation for example) an analytical approach was used to extract approximate quantitative results. Quantitative analysis is a well-known technique for defining general purpose computer architectures [19, 20], and is typically more accurate than growth rate analysis (in which constants, factors and design issues are hidden).

Depending on the target architecture, various models of computation can be assumed to perform this quantitative analysis. Common models in the field of computational complexity theory include Turing machines, finite state machines and random access machines [21]. To avoid any premature architectural assumptions, the model of computation is defined as an abstract machine in which a set of predefined primitive operations each have unit cost. Since the architecture is embedded on an FPGA and applications are inherently memory-access or streaming based, an abstract machine in which only memory reads, writes, additions and multiplications are considered to be significant operations is chosen for the model of computation. For each algorithm in Fig. 2.7, a pseudo-code listing is used to find an expression for the required number of additions/subtractions, multiplications, as well as memory reads and writes. For example, the pseudo-code for the amplitude scaling of the channel equalisation operation is given as:

```
loop p=0 to P-1
 loop n=0 to N-1
   amp_comp[p][n].RE = iq_dem[p][n].RE*amp_coeff
   amp_comp[p][n].IM = iq_dem[p][n].IM*amp_coeff
```

This requires P×N×2 memory writes and multiplications, as well as P×N×2+1 memory reads if the above model of computation is used. The identical methodology is used to analyse each of the remaining algorithms, which are covered in Appendix A.

The quantitative requirements are then calculated by substituting the parameters from Fig. 2.7 into each of these analytical expressions. The results are graphically depicted in Fig. 2.8. All figures are in millions of operations required for processing one burst (a burst is $t = P \times T_p = 21.3$ ms in the test-case).

The correlation in Fig. 2.8 overshoots the axis limit by a factor of more than 6 (equally distributed between multiplications, additions/subtractions and memory reads), emphasising the computational requirement difference between the time domain method (correlation) and the frequency domain method (fast correlation) for matched filtering. It is interesting to note that the number of memory writes are significantly less than the number of memory reads, with the exception of the algorithms that make use of the fast Fourier transform (FFT) or sorting operation. The cell averaging (CA-)CFAR algorithms require the least amount of processing of the CFAR classes, especially when implemented with the sliding window (SW) optimisation. Although order statistic (OS-)CFAR typically performs better than CA-CFAR in the presence of interferers, the processing requirements are more than 33 times as high. The next section looks at how these radar algorithms are broken down into mathematical and signal processing operations.

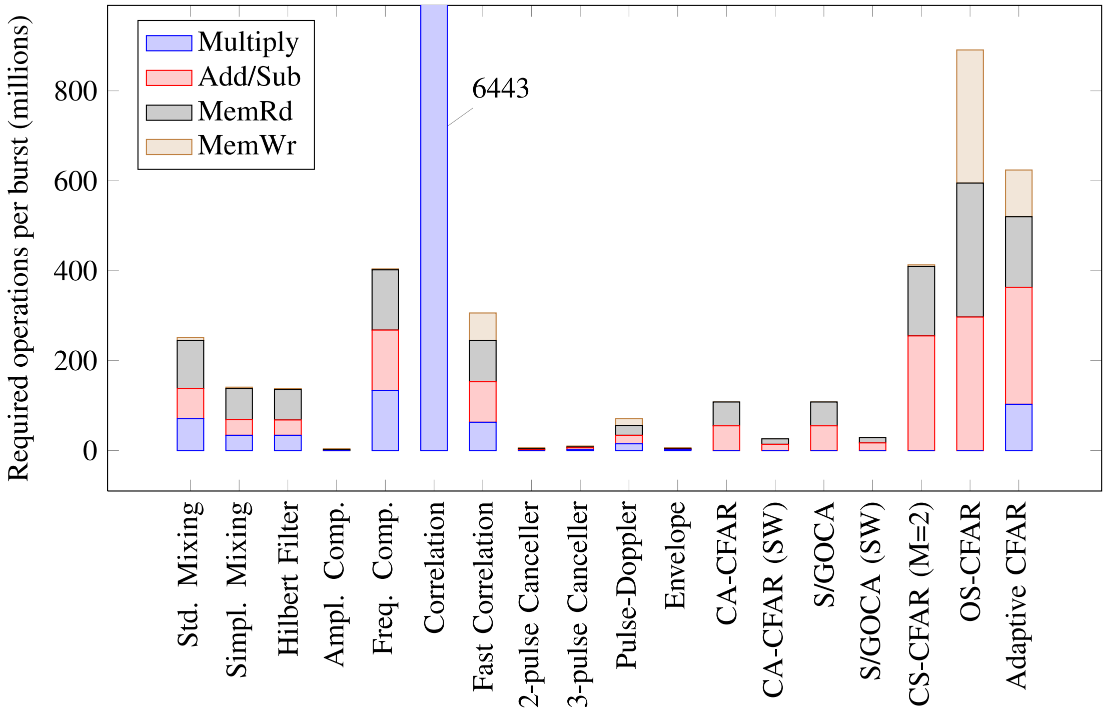


Figure 2.8: Quantitative Computational Requirements

2.4 COMPUTATIONAL BREAKDOWN INTO MATHEMATICAL OPERATIONS

Most of the radar signal processing algorithms follow a fairly natural breakdown into common mathematical and signal processing operations as shown in Table 2.1 and Fig. 2.9. Refer to Appendix B for a complete list of signal processing operations that are typically required in radar signal processing applications.

The number to the right of each of the mathematical operations in Fig. 2.9 indicates the number of algorithms that make use of that particular operation. For example, the finite impulse response (FIR) filter structure is used in all three of the discussed I/Q demodulation algorithms, in two Doppler algorithms, in channel equalisation and in one of the pulse compression algorithms.

The CFAR algorithms primarily break down into numerical sorting or minimum/maximum selection as well as block/vector summation operations. Although functions such as the summed area table (SAT) and the log-likelihood function are used mainly in the adaptive CFAR implementation, they could be used for other algorithm classes on the data processor level.

Table 2.1: Summary of mathematical operations required for radar algorithms

Radar Algorithm	Mathematical Operations
Analogue Interface	Counter, comparator, trigonometric functions
I/Q demodulation	FIR filter, digital filter, interleaving, negation
Channel equalisation	Element-wise complex multiplication, digital filters
Pulse compression	Correlation / FIR filter, element-wise complex multiplication, FFT, IFFT (typically more than 4k samples)
Corner turning	Matrix transpose
Clutter cancellation	Matrix multiplication / digital filter
Doppler processing	Element-wise multiplication, FFT (typically 512 or less)
Envelope calculation	Squaring, square-root, logarithm, summation
Constant false alarm rate	Block summation, sliding window, sum area table, scalar multiplication / division by a constant, comparison, sorting, minimum / maximum selection
Target report	Mono-pulse calculation, element-wise division, comparison
Data processor	Higher level decisioning, branching, filter, convolution, correlation, matrix multiplication, sorting, FFT, angle calculations

2.5 RELATIVE PROCESSING REQUIREMENTS FOR EACH OPERATION

The most important of these mathematical operations need to be selected for optimisation, without giving too much importance to one specific implementation alternative. Logically only mathematical operations making up a fair amount of processing time should be optimised. To find the normalised percentage usage, the different implementation options for each algorithm are thus weighted according to their computational requirements. Algorithms with low computational requirements should be favoured over those with high requirements. For example, pulse compression can be implemented with the fast correlation or as a FIR filter structure. The FIR filter structure has a substantially larger computational requirement in terms of operations per second, but may be better suited for streaming hardware implementations, and thus cannot be neglected. Within radar algorithm groups, each implementation is assigned a percentage of total computational requirements for that group. This number is then subtracted from 100% and normalised such that the added weights of each group add up to unity. Table 2.2 shows the normalised processing requirements for each mathematical operation across the different implementations that were discussed.

It comes as no surprise that the FIR filter and FFT operations have the highest usage. Although only used in 3 of the 7 discussed CFAR algorithms, the high computational requirements of the sorting operation indicate its importance for optimisation purposes. Figure 2.10 graphically confirms the above results for 5 different RSP implementation alternatives.

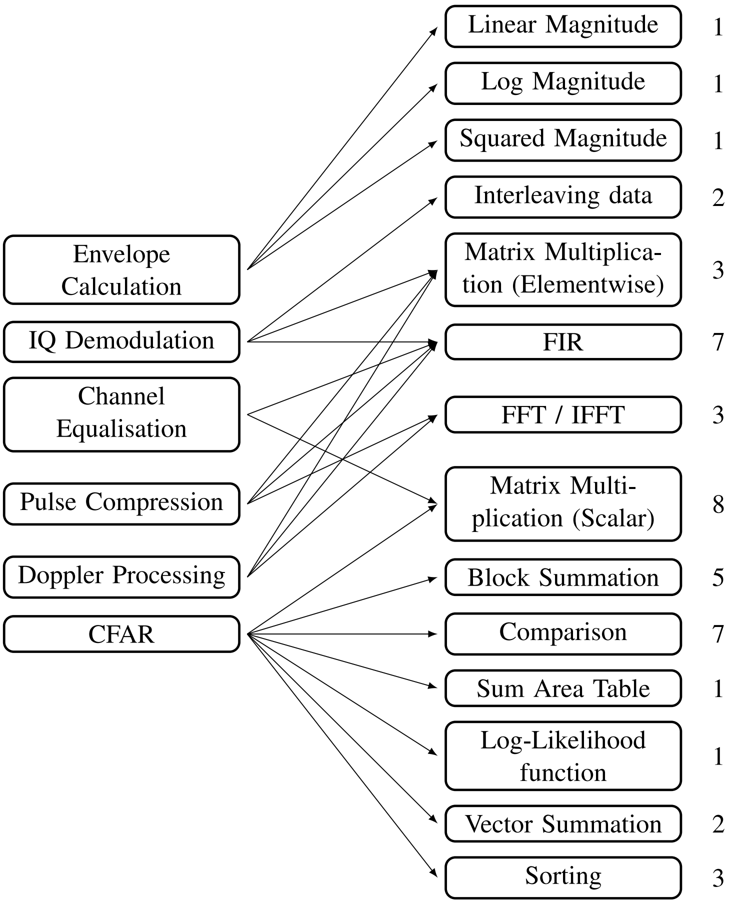


Figure 2.9: Radar signal processor algorithmic breakdown

Table 2.2: RSP normalised percentage usage

Mathematical Operation	%
FIR	56.63
FFT / IFFT	22.50
Sorting	9.57
Block Sum	3.53
Log-Likelihood	3.36
Sum Area Table	1.53
Matrix Multiplication (Elementwise)	1.51
Matrix Multiplication (Scalar)	0.49
Interleaving data	0.18
Log Magnitude	0.17
Comparison	0.17
Linear Magnitude	0.16
Squared Magnitude	0.15
Register Sum	0.07

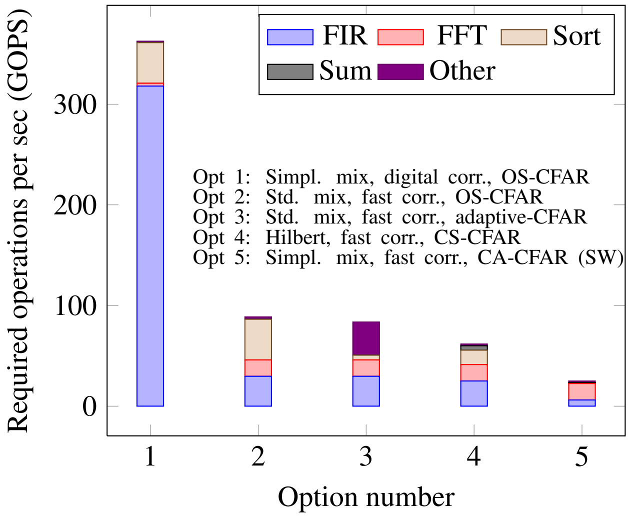


Figure 2.10: Complete RSP computational requirements

Options 1 to 4 all make use of amplitude and frequency compensation, the 3 pulse canceller, pulse-Doppler processing and the linear magnitude envelope calculation. In the simplest case (option 5), the RSP is constructed without channel equalisation and clutter cancellation, consisting of only Hilbert filtering, fast correlation, pulse-Doppler processing, linear magnitude and cell averaging CFAR. The large 'other' block in option 3 primarily consists of the log-likelihood function. It is interesting to note that the summation operation does not require much resources compared to the FIR filter, FFT and sorting operations.

Fig. 2.11 depicts the algorithmic density range (the range between the minimum and maximum algorithmic processing time relative to the total processing time) of the different mathematical operations when all possible dataflow permutations of the test-case in Fig. 2.7 are considered. Each of the possible combination options of the test-case are broken down into the comprising mathematical operations, and sorted according to the percentage processing time. For example, when option 1 is selected, the FIR filter operation requires 87.7 % of the total processing time of that specific RSP implementation. Similarly the FIR filter in option 5 consumes 24.6 % of the processing time. Options 1 and 5 happen to be the extremes for the FIR filter operation (representing the maximum and minimum processing times respectively) and thus determine the range of the first bar in Fig. 2.11. Only options 1 and 5 are shown as lines on the diagram; all other possible combination options are hidden for clarification purposes. Besides the FIR filter, the FFT operation, and the CFAR comparisons, all other operations have a minimum processing time of 0; meaning that those operations are not required in one or more implementation alternatives. The maximum of each bar thus represents the processing

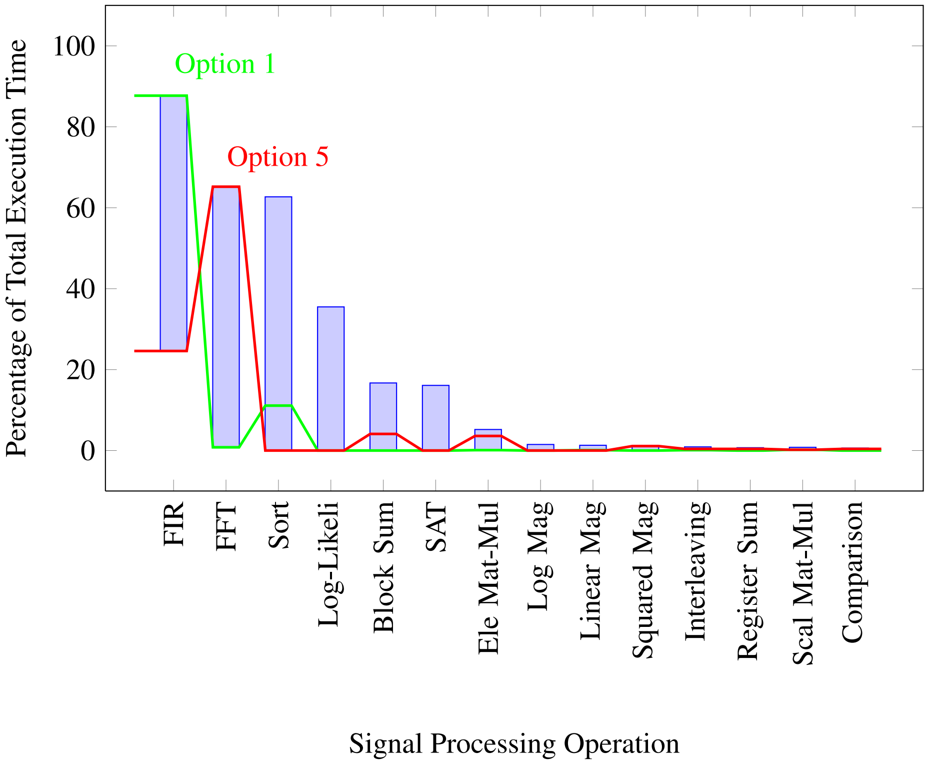


Figure 2.11: RSP algorithmic density range

percentage required in the option with the highest computational requirements for that operation. For example, the sorting operation could require as much as 63 % of the processing time in one or more of all possible permutations of the test-case.

The need for an optimised FIR filter and FFT operation is apparent based on the results of the analysis in this chapter. Even though the sorting and log-likelihood functions consume a considerable amount of processing time, they are only used in a small subsection of processing algorithms. Regardless, all mathematical operations identified in this chapter as well as Appendix B need to be supported by the architectural solution, and the necessary arithmetic and control-flow requirements need to be catered for.

To handle the throughput requirements of the radar algorithms discussed in this chapter, a suitable processing architecture is required. The next chapter discusses some of the currently available processing architectures and technologies as well as their applicability to radar.

CHAPTER 3

CURRENT PROCESSING TECHNOLOGIES

3.1 OVERVIEW

The main processing task of the radar signal processor is the extraction of target information. A variety of different high performance technologies can be used for this task; FPGAs, ASICs, DSPs, CPUs, GPUs, coprocessors, soft-core processors, or conventional microprocessors.

Performance is not the only concern when it comes to choosing a processing technology for the radar signal processor. Radar systems operate in very constrained environments, placing stringent limits on the available size, weight and power consumption. These computational density (GFLOPs/m^3) constraints are typically not achievable with commercial high performance systems or similar building-sized systems. Additionally, specifications demanding extreme shock, vibration, humidity, and temperature conditions need to be met.

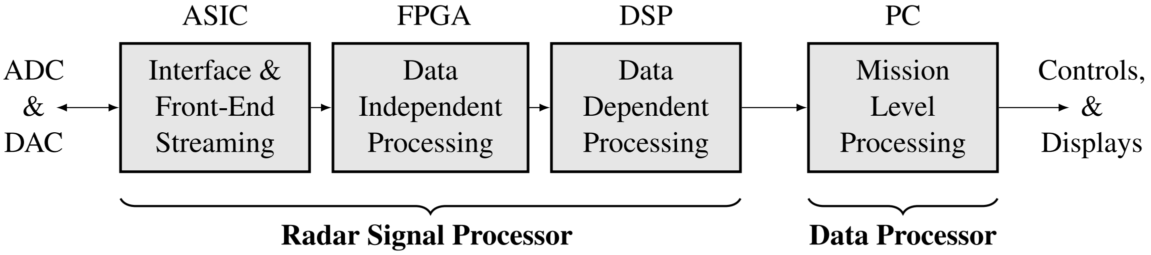


Figure 3.1: Processing technology stages in a radar system

The computing platform or technology is highly dependent on the processing stage within the radar system. Fig. 3.1 depicts an example of functional partitioning across various technology domains [22]. In this example, high-speed custom digital logic (such as ASICs) is used in the front-end processing chain of the radar signal processor, while reconfigurable logic (such as FPGAs) is used for the majority of the data-independent operations. Data dependent operations are usually hosted on commercial off-the-shelf (COTS) processors (such as DSPs) either in fixed or floating-point format, while commercial computer systems are frequently used as radar data processors and for user interface and control purposes. With the recent advances in FPGA technology, the entire radar signal processor can however be implemented as a system-on-chip design. Replacing the ASIC and COTS technologies with reconfigurable technology increases both the flexibility as well as the design time of the system, emphasising the need for a higher level design methodology.

For each of the technologies in the following sections, the processing architecture as well as the applicability to radar processing is investigated. Refer to Appendix C for a discussion of the most common processing architectures and the various optimisation techniques used in these processing technologies. Each of these architectures is a potential candidate and could be used as an architectural template for the proposed architecture.

3.2 DIGITAL SIGNAL PROCESSORS

A DSP is a microprocessor with an architecture designed specifically for real-time signal processing applications. DSPs often feature slight variations from traditional sequential instruction set architectures to increase throughput or reduce complexity. One such variation is the use of a mode set instruction. Depending on the mode set, the same instruction performs differently. Most DSP operations are comprised of sum-of-product calculations, and thus require multiplying, adding, looping, and fetching new data values. The architecture of modern DSPs is tailored to do these operations (or multiple of them) in parallel in a single clock cycle.

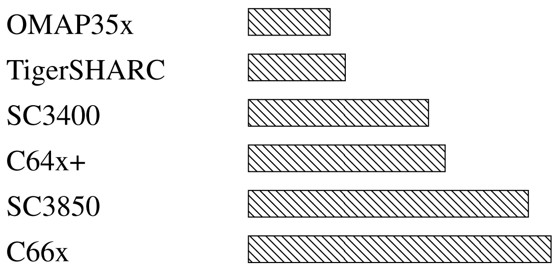


Figure 3.2: BDTI Speed Scores for Fixed-Point Packaged Processors [23]

The chart in Fig. 3.2 compares the performance results of the highest performing commercially available DSPs. The results are based on an average of 12 digital signal processing algorithm kernels; namely 4 types of FIR filters, an infinite impulse response (IIR) filter, vector dot product, vector addition, vector maximum, Viterbi decoder, a sequence of control operations, a 256-point FFT, and bit unpacking [24]. The architecture of the two highest performing DSPs, Texas Instruments' C66x and the Freescale's SC3850, are discussed in the following sections.

3.2.1 Architecture of the Texas Instruments C66x Core

The C66x core is the successor to the popular C64x+ architecture, adding floating-point support and extending the number of parallel multipliers [25]. Although there are some distinct differences in the C64x+ and the C66x functional units, the block diagram in Fig. 3.3 summarizes their dual datapath and very large instruction word architectures.

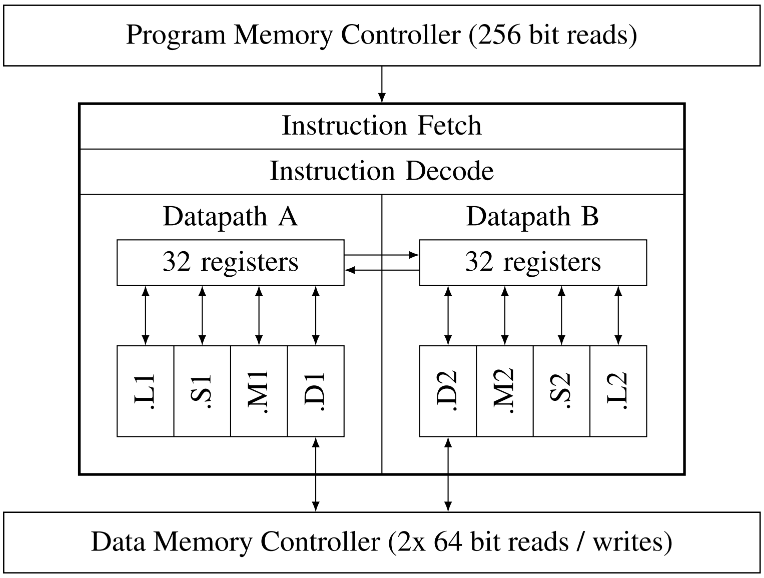


Figure 3.3: Texas Instruments C64x+ and C66x CPU Architecture

The two datapaths each consist of 4 functional units, initially designated as multiplier (.M) unit, ALU (.L) unit, control (.S) unit and data (.D) access unit. Each parallel functional unit is a 32-bit RISC core with its own instruction set, of which instructions are not only limited to their initial designations. For example, the "ADD" instruction can be executed on the .L, .S and .D units rather than just the arithmetic logic unit (ALU). Provided the application is not memory bound, up to 16 single precision floating-point operations or 32 MAC operations can be executed per cycle per core (2-way single instruction multiple data (SIMD) Add/Subtract on .L1, .L2, .S1, and .S2 as well as 4-way SIMD Multiply on .M1 and .M2). When 8 parallel C66x cores (each with 32 KB L1 and 512 KB L2 cache) are integrated onto a single chip (TMS320C6678) running at the maximum clock frequency of 1.25 GHz, a theoretical performance of 160 GFLOPs or 320 GMACs could be achieved. However, data memory throughput and control-flow restrictions substantially impede this theoretical performance in any practical or constructive processing tasks.

3.2.2 Architecture of the Freescale SC3850 Core

Similar to the C66x core, the Freescale SC3850 core also makes use of a VLIW instruction set architecture to meet the performance requirements of modern DSP systems [26]. It features four identical data ALUs and two address generation units, making it a 6 issue statically scheduled VLIW processing core. The program counter (PC) unit supports up to four zero-overhead hardware loops. Fig. 3.4 summarises the architectural features of the SC3850 core.

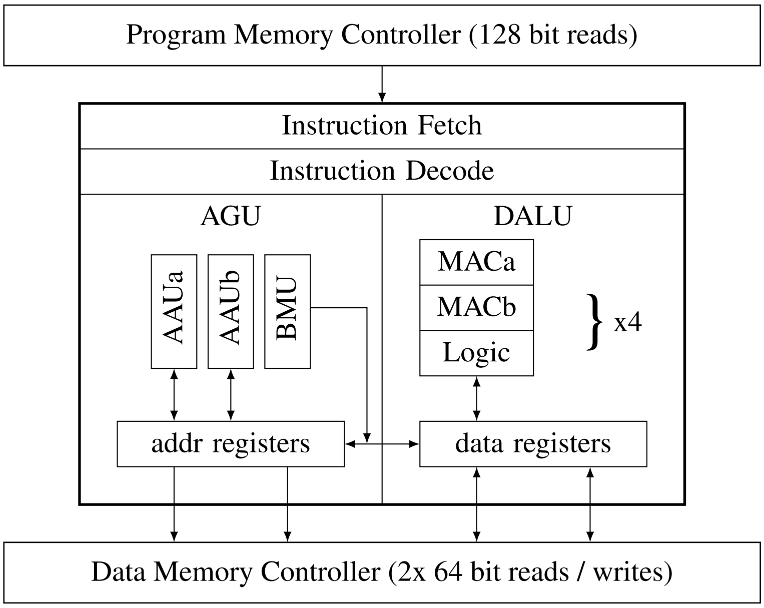


Figure 3.4: Freescale SC3850 CPU Architecture

Each data arithmetic and logic unit (DALU) contains two 16x16 multipliers for SIMD2 multiplication and accumulation, as well as application specific instructions for FFT, Viterbi, multimedia and complex algebra operations. The 16 data registers can be used as 8-, 16- or 32-bit data words (fractional or integer data types) and allow accumulation up to 40 bits. When using the data registers as packed 8-bit registers, a total of 16 instructions can be executed per clock cycle with the SIMD4 instructions on each DALU.

On the address generation unit (AGU) side there are 27 32-bit registers; 16 for address calculation and 11 offset registers. Each address arithmetic unit (AAU) drives one memory access and performs an address calculation per clock cycle. The two 64-bit data buses can thus be used concurrently to access the 32 KB L1, the 512 KB L2 cache or the switching fabric for a throughput of 128 Gbps at a clock frequency of 1 GHz.

The Freescale MSC8156 DSP adds 6 of these SC3850 cores (total peak performance of 48 GMACs), 1 MB shared memory, a DDR-2/3 controller and a coprocessor (MAPLE-B baseband accelerator: Turbo/Viterbi decoder, FFT/IFFT, CRC) onto a single chip running at 1 GHz [27].

3.2.3 Usage in Radar

Many conventional radar signal processors rely on arrays of off-the-shelf DSPs together with coprocessors hardwired for a particular function [28–32]. Newer DSP radar systems use multi-cored DSP processors with integrated hardware accelerators for signal processing routines such as filters and transforms [33–35]. In both implementations, identical processors (and identical firmware) are typically pipelined and carefully timed so that blocks of data can be assigned in a round-robin manner. This method ensures that each processor finishes just in time before new data is allocated. The output of this array of processors will be at the same rate as the block input data, but delayed by the time it takes a single processor to process the block. DSP architectures are popular for implementing radar algorithms because of their ease of use and high-speed instruction clock. However, the limitation of these designs is that the entire array of processors is usually only fast enough to perform a single function of the radar signal processor, effectively requiring multiple of these arrays for each algorithm. Additionally, modern DSPs are not as well-suited for strict timing and low latency streaming applications as older generations of DSPs, since they include speculative execution, branch prediction, complex caching mechanisms, virtual memory and memory management units. Multiple re-runs of the same application program yields varying instruction cycle counts, making the processing response times non-deterministic. The careful firmware design, timing requirements as well as complex hardware interconnections between processors usually results in a long development time.

3.3 FPGAs

Field programmable gate arrays are re-programmable devices for implementing custom digital logic circuits. Internally they are arranged in grids of programmable cells referred to as logic elements (LE) or configurable logic blocks (CLB). Each of these cells contain configurable SRAM-based look up tables (LUT), carry chains, multiplexers and registers. The logic blocks are surrounded by programmable I/O pins, which support a variety of transceiver, interface and communication standards.

Additionally, DSP blocks, dedicated memory, clocking resources and built in IP (such as hard-core processors, Ethernet MACs, serialisers and deserialisers) are often integrated into the array of programmable blocks [36, 37]. Typically FPGAs are configured with a binary file, which is compiled from a HDL such as Verilog or VHDL [38, 39].

The internal LUTs can be configured to perform various functions from complex combinatorial logic to simple logic gates. With these LUTs and the clocked registers in the logic blocks, any sequential or combinatorial digital circuit can be synthesised. It is thus possible to implement any of the processing architectures discussed in Appendix C on an FPGA. Some common architectures and their applicability to radar signal processors are discussed in the following subsections.

3.3.1 Streaming / Pipelined Architecture

The streaming or pipelined architecture is the most common for FPGA applications. A custom hardware pipeline for a specific application is created with either traditional HDL design flows, graphical or fused datapath tools. The fused datapath tools can maintain a higher bit accuracy and throughput by joining two consecutive floating-point operations in the processing chain [40, 41].

For radar signal processing applications, this streaming architecture on an FPGA is applicable to either the entire radar system or smaller sub-parts thereof. Various radar signal processors rely on FPGAs to provide an interface to the high-speed ADC and DAC as well as perform some data independent front-end streaming operations such as I/Q demodulation, filtering, channel equalisation or pulse compression [42–48]. The remaining processing and data dependent stages are then handled by other processing technologies such as PC clusters or DSPs. Various other implementations use a single FPGA to realise the entire radar system in a pipelined architecture [49–52].

3.3.2 Adaptive Hardware Reconfiguration

Since the LUTs of modern FPGAs are reprogrammable during execution, a partially reconfigurable adaptive hardware platform is possible. A set of profiles for a variety of different applications could be created, and activated based on the current computational requirements [53]. Effectively a set of coarse-grained blocks or components would be predefined and stored in memory, only to be configured into hardware when needed. Depending on the size of the reconfigurable portion, the dynamic reconfiguration can take between tens of milliseconds to hundreds of milliseconds. Similarly, hardware could be generated for a specific application on demand and configured in an unused section of the FPGA. Adaptive hardware reconfiguration has had some exposure in radar applications, especially where limited FPGA resources are available and multiple different modes of operation are required [54, 55].

3.3.3 Hard-core Processors Embedded on FPGAs

Some FPGAs include built in hard-silicon processors between the reconfigurable logic. For example, select devices from the Xilinx Virtex 4 and 5 families feature an embedded 550 MHz PowerPC processor. More recently, Xilinx integrated a 1.0 GHz dual-core ARM Cortex-A9 into the 28-nm Zynq-7000 FPGA [56]. Altera includes a similar ARM core (at 800 MHz) into some of their Arria V and Cyclone V FPGAs as a hard-core fixed processor. System-on-chip (SoC) devices are also available; for example the Intel E6x5C Series adds an Altera Arria II GX FPGA device onto the same package as a low power 1.3 GHz Intel Atom Processor, connecting the two via PCIe x1 links [57].

Embedded processors are generally used as control or for interface purposes in radar systems [58]. They are aimed at general purpose sequential computing, and lack the DSP performance needed for radar signal processing, although they frequently feature high clock speeds.

3.3.4 Generic Soft-core Processors Embedded on FPGAs

Another approach to simplify the HDL design phase is by utilising a soft-core processor and a C-compiler. Since the soft-core architecture is embedded within the FPGA structure, custom instruction or hardware coprocessors and accelerators are easily integrated into the datapath with tools such as Altera SOPC Builder or Xilinx Platform Studio.

Generic soft processors such as the NIOS II [59], ARM Cortex-M1, Freescale V1 ColdFire, MIPS MP32 or the MicroBlaze [60] are thus often used in conjunction with HDL-based intellectual property (IP) to create a functional system. This has been done numerous times in the radar field, but is only viable with applications requiring limited processing as with automotive radar systems [61, 62]. Multiple soft processors can be instantiated on the same FPGA to create a symmetric multiprocessor (SMP) system to achieve a slightly higher performance [63]. These soft-core processor based systems allow high-level control (from a software development environment) at the expense of performance, primarily as a result of the low clock speed, control-flow restrictions and limited instruction level parallelism. Similar to the hard-core processors embedded on FPGAs, the generic soft-core processors generally lack the real-time performance required in applications such as radar. Instead they are frequently used for control and configuration purposes or as an interface to the data processor [64–66].

3.3.5 Custom Soft-core Processors Embedded on FPGAs

Unlike the generic soft-core processors, custom soft-core processors are optimised for a specific application, and are not necessarily fixed to a specific FPGA vendor's devices. Multiple custom soft microprocessors exist for a variety of different applications, ranging from specialised and optimised communications processors all the way to simple educational RISC cores [10].

One notable attempt at a high-throughput soft-core processor is the GPU-inspired soft-core processor [67, 68]. With the long term goal of creating a soft-core processor to fully utilise the FPGA potential, a soft-core processor that executes AMDs GPGPU CTM applications (by supporting the r5xx ISA application binary interface (ABI) directly) was created. While it represents a step in the right direction for high performance soft-core processors, it is not optimised for digital signal processing applications. As a result of limited concurrent memory accesses and only a single SIMD4 ALU, multiple threads are executed in a lock-step manner for a maximum of one ALU instruction per clock cycle.

Another method of achieving a high-throughput soft-core processor is by vectorising the ALU [69, 70]. Such soft vector processors are essentially SIMD processors that perform the same operation on all data-elements in the vector. As such they are well-suited for applications that can be vectorised, but leave the majority of data-dependent signal processing operations unoptimised.

VLIW soft processors seem like a natural progression of soft-core processors, especially since the DSP counterparts all make use of the VLIW paradigm. However, increasing the number of parallel processing units severely decreases the performance of the processor, as the register file quickly becomes the bottleneck [71]. When the number of ports to/from each processing element are limited, some performance improvements can however still be achieved [72].

In another instance, the first generation Texas Instruments TMS32010 DSP core was ported to an FPGA as a soft-core processor [73]. However, the low clock speeds and limited instruction-level parallelism make the core impractical for high performance processing. Regardless, there have been very limited attempts at a high performance soft-core DSP or streaming signal processing architecture among the vast selection of processors and IP.

3.4 ASIC AND STRUCTURED ASIC

The same processing architectures that can be implemented on an FPGA can also be transferred to a standard-cell ASIC. Although the non-recurring engineering costs are extremely high for this architecture, the clock rate of an ASIC implementation compared to an FPGA implementation can be considerably higher, at a fraction of the unit cost for large batches.

Structured ASICs are a trade-off between FPGAs and ASICs. They keep the pre-defined cell structure of the FPGA-based design (or any other predefined block structure) and fix the interconnect and LUT SRAM for a specific configuration. Thus higher performance and lower power consumption can be achieved compared to FPGAs. The predefined structure makes structured ASICs less complex to design, achieving a faster time-to-market than ASICs. FPGA manufacturers include tools that facilitate transferring a synthesised FPGA design to a "hardcopy" structured ASIC. Table 3.1 compares the various characteristics of standard cell ASICs, structured ASICs and FPGAs [74, 75].

Table 3.1: FPGA/Structured-ASIC/ASIC Comparison

Criteria	**FPGA**	**Structured ASIC**	**Standard-cell ASIC**
Speed Reduction (seq)	24.7	3.2	1
Speed Reduction (logic)	3.4	1.6	1
Power consumption (seq)	-	1.1	1
Power consumption (logic)	14	1.5	1
Area Usage (seq)	-	8.0	1
Area Usage (logic)	32	2.1	1
Unit costs	High	Medium	Low (high qty)
NRE cost	Low	Medium	High
Time-to-market	Low	Low-Medium	High
Reconfigurability	Full	No	No
Market Volume	Low-Medium	Medium	High

The results are based on both combinatorial (logic) and sequential benchmarks of a 65-nm standard cell ASIC, a 65-nm NAND2-based structured ASIC, and a 65-nm Xilinx Virtex-5 XC5VLX330. It should be noted that the standard cell ASIC performance can be further improved with full-custom designs. With a full-custom design ASIC implementation, 3 to 8 times the performance, 14.5 times less area, and 3 to 10 times less power consumption can be achieved compared to standard cell ASICs [75].

Several licensable cores are commercially available for ASICs. Companies such as ARM (Cortex series), CEVA (X and TeakLite families), and Tensilica (Xtensa and ConnX cores) supply synthesisable process-independent (often configurable) cores, which can be embedded on chip with any required additional interface or custom logic.

Early digital radar systems heavily relied on custom ASIC implementations to achieve the desired performance, as other technologies were simply not available. More recent systems typically use reconfigurable logic rather than ASICs to implement front end streaming processing, mainly because of limited batch quantities, increased flexibility and the lack of high initial NRE costs.

3.5 CONSUMER PERSONAL COMPUTER SYSTEMS

Standard off-the-shelf computer systems found in today's businesses and homes often outperform the custom processors in legacy radar systems. Although their architecture is tailored towards general purpose computing, the huge market driving their development ensures consistent performance growth even for signal processing applications.

Architecturally, modern processors microcode the x86 instruction set into lower level instructions that map more directly to the execution units. On the Intel Sandy Bridge micro-architecture, 6 parallel execution units receive instructions from the hardware scheduler (which features out of order execution, register renaming, branch prediction, and speculative execution) in order to exploit instruction-level parallelism. Each of the execution units can perform a subset of operations ranging from ALU and SIMD operations to memory accesses and address generation. Three execution units directly interface to L1 D-Cache, able to perform two 128-bit memory loads and a 128-bit memory store operation each clock cycle. L2 Cache is connected to L1 Cache via a single 256-bit bus.

Multiple of these cores are then integrated into a single chip. This parallel processor architecture is well matched to radar signal processing applications, since multiple channels can be processed in parallel [76]. Offering significantly reduced platform costs, clusters of these standard architectures were used for some radar applications [77, 78]. Although high-level programming languages together with parallel processing frameworks such as OpenMP, MPI and TBB can be used, programming real-time signal processing applications under strict latency requirements is challenging mainly because of operating system overheads, non-deterministic response times and data-input and output overheads.

3.5.1 Vector Processing Extensions

Modern CPUs include vectorised instructions in each of their processing cores. Vectorisation instruction set extensions (MMX, SSE, and more recently AVX and AVX2) enable arithmetic SIMD operations on selected extra-wide (packed) registers for high throughput signal processing applications. Each of the 16 registers in the AVX extensions is 256-bit long, and allows packing of bytes, words, double words, singles, doubles or long integers for SIMD32 to SIMD4 operations respectively. Such vector operations are useful for many signal processing applications, achieving significantly higher computational performance than some high-end DSPs [79]. As dataset sizes increase beyond cache-sizes however, significant performance drops are experienced as a result of the additional memory fetch latency.

3.5.2 Graphics Processing Unit

Another method of processing streaming data is with the GPU. Support for general purpose computing on GPUs (GPGPU) has become a standard feature amongst vendors such as NVIDIA and AMD. The details of the underlying low-level architecture however, are kept strictly confidential. NVIDIA provides a low-level programming language called PTX (Parallel Thread eXecution) that doesn't expose any the underlying instruction set or architectural features. Similarly, AMD provides a Close-to-the-Metal (CTM) API [80], an abstracted SDK hiding the GPU control hardware but exposing some details of the underling fragment processor.

General purpose computing support is obtained through configurable shader processors. Shader operations are by default very regular in nature, applying the same operation to each vertex (or pixel). The architecture can thus be statically scheduled, deeply pipelined, multi-threaded and make use of vectorised SIMD operations, making it well-suited for radar applications [81, 82]. As the GPU is controlled and configured by a host program on the CPU (through calls to the compute-interface of the API driver), loading and offloading the computed results adds additional latency and creates a bottleneck for practical real-time signal processing applications on GPU platforms.

3.6 COMMERCIAL MICROPROCESSORS

Embedded low-power RISC processors are becoming increasingly popular for smart-phone, tablet and other mobile processing platforms. Low power microprocessors such as the ARM Cortex series are aimed at consumer multimedia applications, and feature multimedia specific signal processing and SIMD vectorisation instructions, making them useful for many medium performance DSP related tasks. As a result such processors have been used in multi-core packages for mobile traffic radar systems [83], which only require limited processing capabilities. For higher performance radar systems, ARM processors are simply used for control or interface applications [84,85], in many instances similar to the embedded FPGA soft- or hard-core processors.

3.7 APPLICATION SPECIFIC INSTRUCTION-SET PROCESSORS

Contrary to the design of general purpose processors, the application specific instruction-set processors (ASIP) design flow is driven by a set of predefined target applications, usually in the high performance embedded processing field [86]. As such binary and backwards compatibility is not of significant importance, and the design is generally optimised for cost and power efficiency [87]. Typically, a RISC (although sometimes SIMD or VLIW) processor is used as an architectural template providing basic programmability and thus some flexibility. After algorithm analysis and profiling, an optimised instruction set is proposed and simulated. This process is repeated until a processor capable of delivering the required performance for the particular application is synthesised. The resulting architecture is thus a trade-off between flexibility and application specific performance, offering limited programmability to adapt for small changes (e.g. in standards or protocols) at a later stage.

Of particular interest is the ASIP design methodology [88–91]. Numerous tools have been developed that automate the entire process and simplify design space exploration, functional and performance simulation as well as generation of a HDL-based target processor description (e.g. Synopsys Processor Designer). A number of architecture description languages (ADLs) have emerged for the purpose of programmatically describing the ASIP architecture; MIMOLA, UDL/I, EXPRESSION, LISA, ISDL, and nML to name a few [92]. The ADL file is used as an input to the development tool, which is iteratively refined based on the simulation as well as synthesis and profiling results.

Surprisingly ASIPs have not featured much in radar systems, although some papers on image and telecommunication processing related applications mention radar as an alternative application [93, 94].

3.8 OTHER PROCESSING ARCHITECTURES

Multiple smaller start-up companies have emerged with many-cored RISC processor arrays (MIMD) for various applications. ZiiLabs' ZMS-40 processor features a quad-core ARM with 96 programmable array processors for media processing (58 GFLOPS). Kalray's 256-core multi-purpose processor array (MPPA-256) delivers 230 GFLOPS or 700 GOPS with a power consumption of 5 Watt. Picochip's tiled-processor connects each of the 430 3-way VLIW processors via a compile-time scheduled interconnect for wireless infrastructure applications. XMOS adds 4 tiles (of 8 time threaded pipelined RISC processors) into their xCORE for timing sensitive I/O micro-controller applications. Tilera incorporates up to 72 cores at 1.2 GHz into the TILE-Gx family for networking and cloud-based applications. Adapteva's E64G401 features 64-cores at 800 MHz for general purpose computing up to 102 GFLOPS, which is incorporated into their low-cost Parallella boards together with a Xilinx Zynq7010 (dual core ARM processor and FPGA). Clearspeed's CSX700 64-bit floating-point 192-core array processor yields a maximum performance of 96 GFLOPS at 250 MHz for low-power signal processing applications.

Another interesting development on exascale computing is Convey's Hybrid Core HC1. A large array of FPGAs is loaded with application specific "personalities", and interfaces to a standard Intel x86-64 processor through cache-coherent shared memory via the north bridge controller. The array of FPGAs thus act as an application specific coprocessor, which is programmed by the host CPU from a high-level programming development environment (Fortran/C/C++). A similar development is the Stretch S6000 architecture; it features an array of custom FPGA-like configurable logic that acts as an instruction set extension to the Tensilica Xtensa core, providing acceleration for various applications without the overhead of the FPGA interconnect.

A rather unconventional architecture is the counterflow pipeline processor architecture [95–97]. This architecture allows temporary results to be reused directly in the instruction pipeline without stalling the entire pipeline until data is written back into the register file. This approach attempts to replace the long path delays between functional units in traditional architectures with pipelined local processing units. Register values flow in the opposite direction of the instruction pipeline, allowing successive instructions to grab values before they reach the register file for writing. This methodology alleviates congestion at the register file and is suited for asynchronous as well as synchronous operation.

Another processor that attempts to reduce power consumption with asynchronous execution is the SCALP processor [98]. This superscalar processor lacks a global register file, and explicitly forwards the result of each instruction to the functional unit where it is needed next.

3.9 DISCUSSION

Modern silicon processes can integrate over 2 billion transistors on a single chip. As such the amount of on-chip arithmetic and computational resources are no longer a bottleneck, leaving the challenge of enabling an effective interface to fully utilise these raw computational resources. In the general purpose processing and DSP paradigm, these challenges have been overcome with various instruction sets optimisations (vectorisation, SIMD, and instruction set extensions) and micro-architectural techniques (dynamic instruction scheduling, superscalar, rotating register files, speculation buffers, register renaming, pipelining, out-of-order execution, and branch prediction) as described in Appendix C. Together with memory caching techniques, these optimisations attain significant utilisation of the underlying processing resources by extracting parallelism from the mostly unpredictable and irregular instruction stream of general purpose processing tasks.

In the streaming processing paradigm however, some of these techniques are actually detrimental to the application performance. Application, task, data and instruction-level parallelism is not sufficiently propagated down to the computational resources, mostly due to inadequate capturing in high-level sequential programming languages as well as hardware-based dynamic scheduling mechanisms which cannot extract sufficient parallelism from the sequential instruction stream. The regular instruction stream and data access patterns of most stream processing applications enable static scheduling with large degrees of parallelism, provided that the programmer/compiler has explicit control over the low-level processing resources. The general purpose processing optimisations and techniques inherently deny this low-level control. As such the optimisation for streaming processing applications becomes a complicated task which requires detailed insights into each mechanism of the processing architecture. Additionally, caches typically perform poorly on streaming applications. The optimisation procedure is thus mostly based on trial and error, experimenting with various cache sizes, compiler optimisation settings, loop unrolling, different high-level coding structures and styles, profiling tools, assembly optimisation, software pipelining and instruction reordering. On multi-cored and many-cored platforms insight into the performance measures is further obscured by shared memory controller throughput and arbitration, memory hierarchy, access times, and inter-process communication mechanisms.

For these reasons, an architecture with much finer control over each low-level computational resource is proposed. Table 3.2 summarises some of the desirable and undesirable features of a radar signal processor.

Table 3.2: Radar signal processor desirable and undesirable features

	Feature	Comments
✓	Deep Pipelines	Deeply pipelined computational resources achieve high throughput and clock frequencies; well-suited for the regular data access patterns of a RSP
✓	Vectorisation and SIMD	Increases performance for many signal processing operations or for multiple channels
✓	Multiple cores	Exploit coarse-grained and task-level parallelism; time-division multiplexing or multi-channel processing
✓	Instruction-level parallelism	RSP algorithms can benefit from more than the 2-8 parallel execution units of current superscalar and VLIW architectures
×	Hardware scheduling	RSP and streaming applications are better scheduled statically at compile-time; deterministic and less overhead than dynamic instruction scheduling
×	Memory caching	Explicit control over fast on-chip memory is preferred to exploit high data locality; scratch-pad rather than cache
×	Branch prediction and speculation buffers	RSP applications feature very limited branches; speculative execution mechanisms just add overhead
×	Register renaming and rotating register files	Adds hardware complexity; can be done in software during compile time for the statically scheduled RSP applications
×	Out-of-order execution	Adds no value as data-dependencies are resolved at compile time
×	Interrupts	RSP applications have a very regular control flow that typically does not need to be interrupted; interrupts are of low importance
×	Data bus arbitration	Simple point-to-point data buses are preferred for deterministic and low latency accesses
×	MMU, virtual memory	Not required for streaming and signal processing operations
×	Central register file	Localised registers preferred for higher memory bandwidths
×	High-level abstraction	Simplification is preferred over abstraction; an easy-to-understand architecture enables intuitive optimisation; the limitations and advantages of the processing architecture become transparent

The most important characteristics for a radar signal processing architecture are thus architectural transparency, deterministic performance (repeatable and predictable), and full control over horizontal as well as vertical instruction-level parallelism (explicit pipeline control to avoid stalls and enable pipelining of multi-cycle arithmetic operations). These characteristics differ substantially from current processing architectures, which seem to focus on higher levels of abstraction and task-level parallelism through multiple cores.

CHAPTER 4

PROPOSED ARCHITECTURE TEMPLATE

4.1 OVERVIEW

This chapter introduces a processing architecture that is well matched to the computational requirements outlined in Chapter 2. The processing architecture is presented at an abstracted level without the datapath optimisations for any of the algorithms. The optimisation procedure as well as the signal processing and algorithmic radar datapath optimisations are covered in Chapter 5, while the final architecture is presented in Chapter 6. Note that this work is not based on any prior literature, and deduced entirely from the data and control-flow requirements of the radar algorithms.

4.2 IDEAL RADAR SIGNAL PROCESSING ARCHITECTURE

At this point it would be useful to consider the ideal processing architecture for radar signal processing from a fundamental and conceptual perspective. Based on generic "ideal DSP" wish-lists covered in literature [99, 100] as well as the findings covered in Chapter 2, the following list summarises some desirable architectural characteristics of a radar signal processor.

- High throughput and performance (instruction-level parallelism, fast instruction cycle; not necessarily clock speed)
- Easily programmable and debuggable (high-level control, development environment, in-field changeable - fast compile times, code profiling capability)
- Large dynamic range (high precision, floating-point)
- Low power consumption
- Small size and low weight (stringent requirements on airborne platforms, spacecraft, missiles, UAVs, vehicles)

- Direct high-speed peripheral interfaces (ADC / DAC / MGT / Ethernet / IOs)
- Streaming and real-time (deterministic, low latency)
- Parallel arithmetic operation execution
- Fast memory with multiple ports (simultaneous read and write)
- No or low overhead loops
- Dedicated address generation units
- Support general purpose computing (control-flow based: if-then, case, call, goto)
- Scalable in terms of performance (multi-core support)
- Excellent customer support and documentation

An architecture that meets all of the above criteria is practically not achievable, and some compromises and trade-offs will have to be made.

4.3 SIMPLIFIED ARCHITECTURE

One of such trade-offs is the fundamental architectural design choice between control-flow and data flow. Streaming hardware implementations are commonly based on fixed and pipelined dataflow implementations that achieve high throughput but extremely limited flexibility, while traditional control-flow architectures offer high levels of flexibility, but limited scalability in terms of performance.

To overcome the limitation in performance scalability, commercial DSP solutions make use of VLIW architectures to exploit instruction-level parallelism. Since the clock speed of an FPGA is substantially lower than that of commercial DSPs, a similar FPGA processing architecture will have to bridge the performance gap by parallelisation methods. An FPGA VLIW implementation would thus have to have more than the typical 8 parallel execution units. When implementing such a VLIW architecture, it quickly becomes apparent that the register file becomes the biggest bottleneck to performance. Execution units require multiple port access to the register file and have to cater for different processing latencies for the various instructions. These accesses to the register file severely limit the maximum performance of the entire processor.

The VLIW register file optimisation attempt inspired an entirely different architecture as shown in Fig. 4.1. Rather than an instruction word controlling the execution units, the instruction word defines how data is routed between the lower level functional units. Unlike a dataflow architecture however,

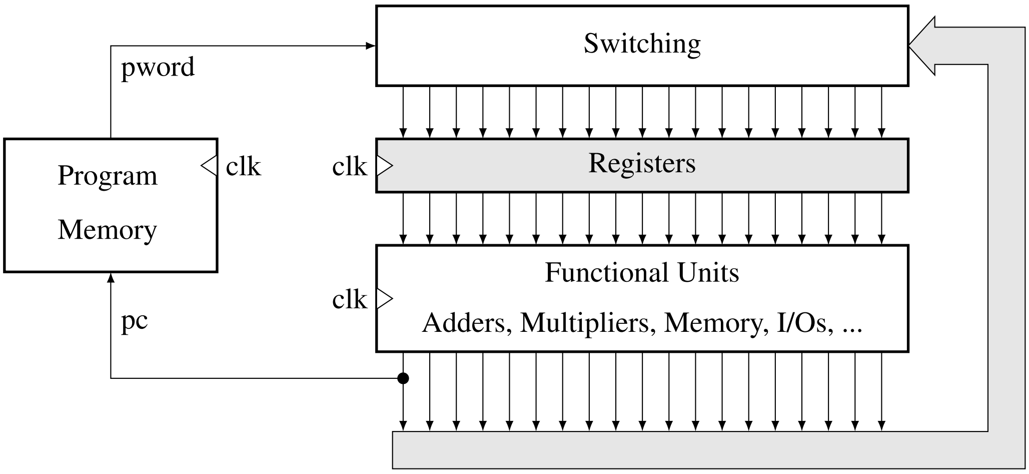


Figure 4.1: Basic Processor Architecture

this approach still retains the program memory and program counter. Thus no token matching logic is required, overcoming the bottleneck of traditional dataflow architectures.

The switching matrix of Fig. 4.1 is a simple multiplexer, selecting a functional unit output for each register, based on a slice of the program word. Functional unit outputs are connected to the multiplexer inputs. The multiplexer outputs are in turn connected to the clocked registers, and each register output has a fixed connection to a specific functional unit input. Fig. 4.2 shows how the switching matrix is implemented for the case of 32 functional units with 32-bit wide registers.

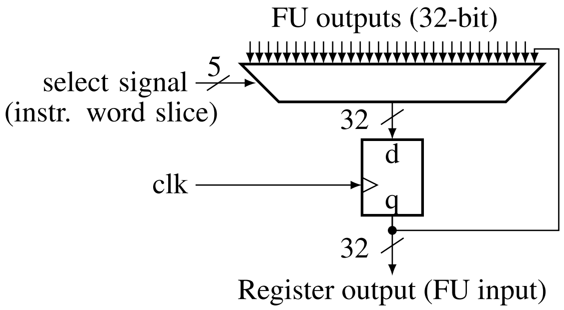


Figure 4.2: Register switching matrix architecture

It consists of a simple D-flip flop register that is fed back into a multiplexer, such that when nothing is selected on the multiplexer select signal, the register remains unchanged. It is thus possible to assign any functional unit output to each register every clock cycle. Note that registers cannot be read; only functional unit outputs can be read and in turn written to one or more registers.

This architecture represents a dataflow architecture that is reconfigured every clock cycle, as each assembly instruction stores a specific static dataflow pattern. Instruction-level parallelism is thus fully exploited with control over both horizontal as well as vertical operations in the datapath. There is no instruction set, only a source select signal for each register that routes data from any functional unit to that register. This matches well to the streaming requirements of a radar signal processor, with a fixed and deterministic latency determined only by the internal functional units.

4.4 DATAPATH ARCHITECTURE

The datapath is split into two sides; the data processing side, and the address generation / program flow control side. The data processing side is implemented as floating-point registers and functional units, while the address generation / program flow control side requires integer registers and functional units. Separating the datapath into integer and floating-point registers reduces the number of multiplexer inputs for each register. Fig. 4.3 depicts the simplified processor architecture with some arbitrary functional units added for illustration purposes. Data flows in a counter-clockwise circular motion for both the integer as well as floating-point sides. The instruction word does not control any of the functional units, instead it controls what functional unit output is selected for each individual register. Additionally it includes a constant, which can be assigned to any register via the multiplexers it controls.

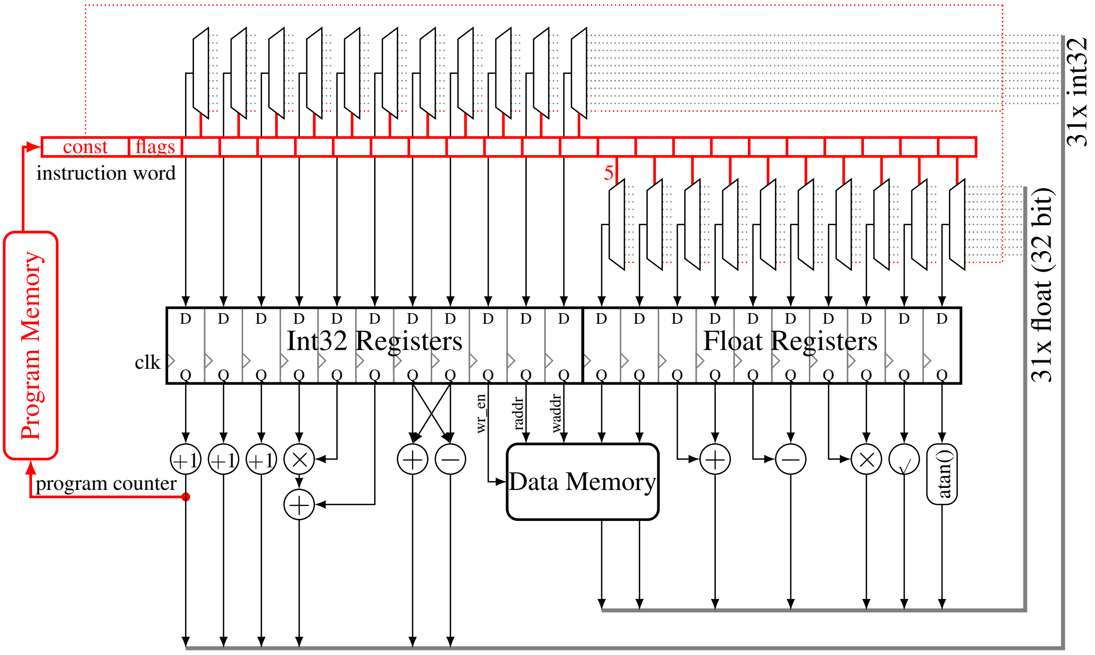


Figure 4.3: Simplified Processor Architecture

Functional units in the datapath can be deeply pipelined and have any arbitrary latency, depending on the target clock frequency. A functional unit does not necessarily have to be an arithmetic operation; external IO ports, delay registers, memory or even register buffers (used as temporary variable storage or function call arguments) are all valid functional units.

The performance of this architecture is quite easily scaled up in performance by increasing the number of functional units and the number of registers. However, the number of functional units are directly related to the size of the multiplexer in the switching matrix. Depending on the targeted technology, increasing the size of the multiplexer beyond for example 128, may severely limit the maximum clock frequency. Similarly, scaling the number of registers directly affects the width of the instruction word as well as the number of multiplexers, influencing the required silicon area. Thus some practical limitations are imposed on the scalability of the instruction-level parallelism.

Registers are named according to the functional unit they are connected to. Registers (functional unit inputs) have the suffixes '_a', '_b' or '_c' while functional unit outputs are assigned the suffixes '_o' or '_p'. For example, the inputs to the integer adder (the first integer adder being referred to as *IAdd0*) are connected to the registers *IAdd0_a* and *IAdd0_b*, and produce the functional unit output *IAdd0_o*. Since the integer subtractor shares the same input registers, *IAdd0_a* is equivalent to *ISub0_a* and *IAdd0_b* is equivalent to *ISub0_b*.

From an assembler/compiler point of view, functional unit outputs are assigned to registers as:

```
; REGISTER_a/b/c = FU_OUTPUT_o/p
FAdd0_a = DMem0_o
FAdd0_b = DMem0_p
FSqr0_a = FAdd0_o
DMem0_a = FSqr0_o
|| ; instruction delimiter - starts new instruction block
; next assembly instruction
```

It should be noted that the order of the instructions in an instruction block does not matter, as each assignment occurs concurrently. In this code segment, the two 32-bit output words from the data memory are fed into the floating-point adder, which is connected to the square root input, and the output of the square root is written back into memory, all in a single assembly instruction. This architecture allows multiple datapaths to run in parallel and provides a mechanism for creating a deep pipeline with multiple functional units. Applications can thus produce one or more streaming data output every clock cycle, greatly improving on the performance of traditional processing architectures by fully exploiting all functional units in parallel. Obviously the memory read and write address as well as the write enable strobes need to be updated accordingly on the integer side for such a streaming application.

4.5 CONTROL UNIT ARCHITECTURE

As it is in Fig. 4.3, the program counter can be assigned any constant from the instruction word or be calculated with the integer registers. No conditional branching or calling is supported. Calling support is added with a stack functional unit. Conditional branching is achieved by making the assignment of a new value to the PC register based on a condition pass flag. Table 4.1 summarises the various functional units associated with the program flow control.

Table 4.1: Control Function Units

FU Name	Inputs	Outputs	Formulae	Description
PC	_a	_o	$o = a$	Program Counter
Stack	push, pop	_o	$o = STACK[TOP]$	Stack
RCon	_a,	le,eq,gr	$a <=> 0$	Int32 Condition Check
FCon	_a,	le,eq,gr	$a <=> 0$	Float Condition Check
Cons	const	_w0, _w1, _d, _f	$w0/1 = LO/HI(\text{const})$ $d = \text{const}, f = \text{const}$	Constant from instruction word slice

The program counter register (PC0_a) can be directly written to via the switching matrix. When no assignment is made, it is incremented by default. The current program counter value is fed back into the switching matrix, such that it can be used for calculations on the integer side (e.g. jumps into a look up table, case statements, non-linear program flow).

The stack unit is a simple clocked last-in-first-out (LIFO) buffer, with a depth equal to the required number of levels in the calling hierarchy. It only has a single output, Stack0_o, which represents the current value on top of the stack. When the “pop” flag from the program word is asserted, the current value on the top of the stack is “popped” off, and the next value appears on the output (or zero if the stack is empty). The “push” flag increases the current program counter value and adds it to the top of the stack. A function call thus requires the function address to be assigned to the PC while asserting the “push” flag. Returning from the function is then achieved by assigning the stack output to the PC and asserting the “pop” flag.

The two conditional check units (RCon and FCon for integer and floating-point respectively) each only have a single input port. The input is compared against 0, and one or more of the following internal flags are set: *_less, _equal, _greater*. These flags form part of a multiplexer input, which produces the condition pass flag output. A 4-bit segment from the instruction word (cond_sel) is used to select which condition is assigned to the cond_pass flag. The following conditions are selectable from the instruction word:

no condition	peripheral_irq	proc_start	not proc_start
RCon0 < 0	RCon0 ≤ 0	FCon0 < 0	FCon0 ≤ 0
RCon0 > 0	RCon0 ≥ 0	FCon0 > 0	FCon0 ≥ 0
RCon0 $= 0$	RCon0 $\neq 0$	FCon0 $= 0$	FCon0 $\neq 0$

When no condition is selected, or the selected condition is true, the cond_pass flag is high and allows writes to the PC. When the selected condition is false, the write request to the PC is ignored and the PC is incremented, resuming with the next instruction in program memory.

The following code segment shows how zero-overhead hardware while loops can be implemented with the above functional units. Similarly, a FOR-loop could be implemented by assigning a counter to the integer conditional check unit, with the initial condition (number of loop cycles) assigned in the previous instruction.

```
PC0_a = PC0_o    ; assign PC to itself; i.e. don't increase PC, loop here
FCon_a = DMem0_o1; loop depends on e.g. data memory output
[FCon_a > 0]     ; condition select: allow PC write when FCon_a > 0
||               ; instruction delimiter - starts new instruction block
; next assembly instruction after loop condition has failed
```

Initial conditions, addresses or constant values are assigned via the “Cons” functional unit. The “Cons” unit uses a 32-bit slice of the instruction word as an input and directly outputs the constant to the floating-point as well as the integer multiplexers. Four output ports are defined: _w0, _w1, _d, _f. On the integer side, the words _w0 and _w1 (Int16) are the sign extended lower and upper words of the 32-bit constant respectively, while the _d output (Int32) is the full double-word constant from the instruction word. The _f output routes the constant through to the floating-point side as a single precision representation of the constant. From the assembler point of view, the above ports can be assigned any number, defined value or address. In turn they can be assigned to any register of the proposed architecture as shown below:

```
Cons0_f = 1.2345
FSqr0_a = Cons0_f                  ; calculate the square root of 1.2345
||
Cons0_w0 = (25+MY_DEFINED_VALUE/2)
IAdd0_a = Cons0_w0                 ; assign a compiler calculated value
Cons0_w1 = addr_fft_function
PC0_a = Cons0_w1                   ; jump to address in current program
```

4.6 MEMORY ARCHITECTURE

Since most algorithms exhibit alternating horizontal and vertical data dependencies, the processing chain typically involves reading memory, performing some mathematical operations, and writing back the processed memory.

Both data and coefficient memories are functionally identical from the processor point of view, providing fixed access times to any memory location after issuing the read address. Memory architectures such as external QDR memory, SRAM or internal FPGA Block RAM exhibit deterministic latency and are thus well-suited for this purpose. DDR3 SDRAM poses a problem when used as data memory in the processing loop. The activation of a row (based on the row address) requires a certain minimum amount of time, and rows in multiple banks need to be refreshed more than 100 times per second. Once a row is activated, the behaviour simplifies and columns can be addressed with data becoming available after the specified CAS latency. DDR memory is thus better suited for loading or offloading large blocks of data after inner-loop processing is complete.

In most signal processing algorithms simultaneous reading and writing of complex numbers (with a separate read and write address bus) greatly improves the algorithmic performance. Dual-ported 64-bit memory is thus mapped directly into the datapath as a functional unit. Fig. 4.4 shows the memory architecture of both the data as well as coefficient memory functional units.

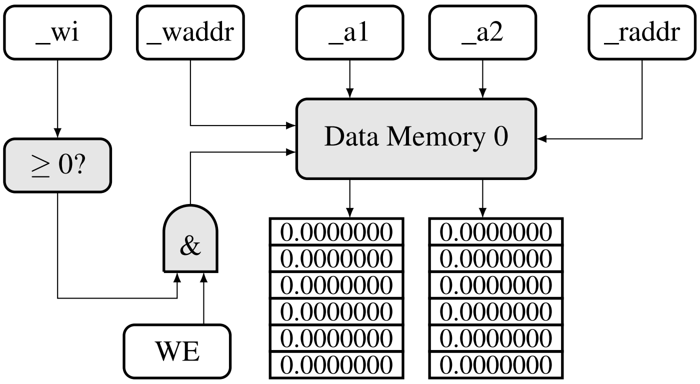


Figure 4.4: Data Memory Functional Unit

Each memory functional unit (DMem and CMem) thus has a write enable flag (_WE) and 3 input ports on the integer side (_wi, _waddr, _raddr). On the floating-point side two inputs (_a1, _a2) and two outputs (_o1, _o2) are provided for simultaneous reading and writing of complex-valued data.

In order to simplify the control-flow, an additional input signal with a comparator is inserted before the write enable signal of the memory controller. Only when this write inhibit (_wi) signal is greater or equal to zero and the write enable flag (_WE) from the instruction word is asserted, data is written to memory. Connecting a simple counter to this write inhibit signal in the inner loop allows the write enable signal to be asserted any number of clock cycles after the inner loop processing commences, accounting for the computational latency between reading and writing memory. This mechanism simplifies software pipelining and removes the need for a prologue before the inner loop, further reducing complexity and program code size.

4.7 FUNCTIONAL UNITS

This section outlines the functional units used for performing the actual calculations, both on the integer (address / program counter) and floating-point (data calculations) side. For each functional unit, the input and output ports as well as the naming convention are given. Multiple of the same functional units can be added to the architecture; for example IAdd0, IAdd1 and IAdd2 are collectively referred to as IAdd<n>.

4.7.1 Integer Functional Units

The functional units on the integer side are given in Table 4.2. One of the most fundamental operations on the integer side is the increase function. When the output is connected to the input a counter is created, which is used for loop control counting, linear address generation, write inhibit counters, activity counters and any other application requiring a linear counter.

Equally important are the add and subtract functional units. Together with the multiply adder, they are used for address offset calculations and matrix address translation from row/column format to a linear address.

The integer buffer unit is used for temporary variable storage (e.g. parameters in function call) or delaying a result by a single clock cycle for alignment purposes. When more clock cycles of delay are required, the variable delay operation provides a tapped delay register, capable of selecting between 1 and 32 clock cycles of latency. This operation is needed for synchronisation and alignment purposes when the processing latency needs to be matched to the address generation latency or vice versa.

Table 4.2: Integer Function Units

FU Name	Inputs	Outputs	Formulae	Description
IInc	_a	_o	$o = a+1$	Int32 Increase
IAdd	_a, _b	_o	$o = a+b$	Int32 Adder
ISub	_a, _b	_o	$o = a-b$	Int32 Subtractor
IMac	_a, _b, _c	_o	$o = a \times b + c$	Int32 Multiply Adder
IBuf	_a	_o	$o = a$	Int32 Buffer
IDel	_a, _b	_o	$o = \text{delay}_b[a]$	Int32 Variable Delay
RotR	_a, _b	_o	$o = a >> b$	Int32 Rotate Right
RotL	_a, _b	_o	$o = a << b$	Int32 Rotate Left
IRev	_a, _b	_o	$o = a[\text{0\_to\_b}]$	Int32 Bitwise Reverse
IncZ	_a, _b, _c	_o, _p	$o = a+1 < b?a+1:0$ $p = a+1 < b?c:c+1$	Int32 Increase, Compare, and Conditional Zero
IIOPort	_a	_o	$IO_{out} = a$, $o = IO_{in}$	Int32 I/O Port
IReg	int addr	_o	$o = REG[\text{addr}]$	Register Select Buffer
IDebug	_a		$debug = a$	Debug trace to ChipScope

The bitwise rotate left and right functional units are used for multiplying or dividing by factors of 2. They can also be used for calculating base-2 logarithms and exponentials. The bitwise reverse functional unit is used by the FFT algorithm for bit-reversed addressing.

The IncZ functional unit is one that is surprisingly not featured on modern instruction sets. It is however a very useful functional unit in the address generation and control-flow paradigm. Under normal operating conditions the output *IncZ0_o* is assigned to input *IncZ0_a*, which forms a continuous counter that resets to zero when the value *IncZ0_b* is reached. When the output *IncZ0_p* is assigned to the *IncZ0_c* input, an up-counter counting the number of overflows on the *IncZ0_a* side is achieved. This instruction can thus be used to transpose arbitrary dimensioned matrices, for FFT address calculation purposes, as a circular address buffer, or simply as a counter and comparator.

The I/O port interface resides on the integer side, and similar to all other functional units, can be assigned and read every clock cycle. At a width of 32-bits and a clock frequency of 100 MHz, a single IIOPORT functional unit can provide 3.2 Gbits of full duplex bandwidth to peripherals, coprocessors or general purpose I/O pins.

The integer and floating-point debug registers are routed to a logic analyser (such as integrated Xilinx ChipScope ILA or an external logic analyser port) to provide a clock-by-clock snapshot of the internal debug register values. These snapshots can be loaded into the development environment for exact comparisons between runtime and simulated results.

The *IReg* functional unit differs substantially from the *IBuf* and any other functional unit. The input port of the register select buffer is simply a multiplexer select signal, which allows any register (based on an address) value to be read directly and fed through to the output port, rather than selecting a functional unit output. An indirect register addressing mode is created with this functional unit, allowing unused functional unit registers to be used as general purpose registers or temporary variable storage.

4.7.2 Floating-Point Functional Units

To increase dynamic range and simplify scaling between processing stages, all functional units in the data processing path were implemented as floating-point values. The single precision floating-point format (32-bit) was chosen since the limited number of bits of the analogue converter interfaces did not justify the additional resource usage of the the double precision (64-bit) arithmetic functions. The IEEE single precision floating-point type features a 1 bit sign bit (s) and 23 fractional bits (mantissa m) along with an 8-bit exponent (x). Eq. 4.1 shows how to calculate the decimal value from a floating-point type. Special numbers such as positive infinity, negative infinity and not a number (NaN) are defined when the exponent is 0xFF.

$$n = (-1)^s \times (1 + m \times 2^{-23}) \times 2^{x-127} \tag{4.1}$$

Although more on chip resources (silicon area or LUT) are required for floating-point arithmetic compared to fixed-point arithmetic, the advantage in dynamic range, lack of scaling between processing steps and faster design time is usually significant in radar processing algorithms. Since most radar signal processing related algorithms are streaming in nature, the increased latency of the floating-point arithmetic does not have a major influence on the overall performance.

Table 4.3 summarises the different functional units that reside on the floating-point side. The majority of functional units in the datapath are self-explanatory mathematical operators; the adder, subtracter, multiplier, square root, sine, cosine, and arctangent are fundamental operations used in multiple algorithms and to calculate other composite operations.

The float-to-integer and integer-to-floating point conversion functional units facilitate data exchange between the two sides. Sometimes indexes are required for calculations, e.g. pulse generation uses a counter on the integer side to calculate the phase for the sine and cosine functional unit. Also, since

Table 4.3: Floating-Point Function Units

FU Name	Inputs	Outputs	Formulae	Description
ItoF	_a	_o	$o = \text{int\_to\_float}[a]$	Int32 to Floating-Point
FtoI	_a	_o	$o = \text{float\_to\_int}[a]$	Floating-Point to Int32
FAdd	_a, _b	_o	$o = a + b$	Floating-Point Adder
FSub	_a, _b	_o	$o = a - b$	Floating-Point Subtractor
FMul	_a, _b	_o	$o = a \times b$	Floating-Point Multiplier
FDiv	_a, _b	_o	$o = a/b$	Floating-Point Divider
FDot	_a$_{0..7}$, _b$_{0..7}$	_o	$o = \sum[a_i \times b_i]$	Floating-Point Dot Product
FBuf	_a	_o	$o = a$	Floating-Point Buffer
FDel	_a, _b	_o	$o = \text{delay}b[a]$	Floating-Point Variable Delay
FSwap	_a, _b	_o, _p	$o = \text{min}[a,b]$ $p = \text{max}[a,b]$	Floating-Point Compare and Swap
FSqr	_a	_o	$o = \sqrt{a}$	Floating-Point Square Root
FSinCos	_a	_o, _p	$o = \sin(a)$ $p = \cos(a)$	Floating-Point Sine and Cosine
FDebug	_a		$debug = a$	Debug trace to ChipScope
FReg	int addr	_o	$o = REG[\text{addr}]$	Register Select Buffer

the data memory ports are only accessible from the floating-point side, these conversion functional units (or special pass-through register through the *FReg* and *IReg* units) allow integer values to be stored in and read from memory. Similarly, I/O ports are only mapped on the integer side, and thus require the conversion functional units to bridge the two sides for floating-point I/O access.

The floating-point dot product has 16 inputs and a throughput of 1 result every clock cycle. Internally the outputs of all 8 multipliers are connected to a balanced adder tree consisting of 7 adders (3 levels deep), giving a latency of

$$FDOT_{LAT} = FMUL_{LAT} + 3FADD_{LAT} \tag{4.2}$$

For algorithms requiring data comparison, the *FSwap* functional unit takes two input values, sorts them and outputs the larger value to the _p port and the smaller value to the _o port.

The floating-point buffer, register select buffer and variable delay functional units are identical to the integer *IBuf*, *IReg* and *IDel* respectively. The delay unit is used extensively on the floating-point side to match processing latencies in streaming applications that have two or more distinct datapaths that are joined again at a later stage.

CHAPTER 5

ARCHITECTURAL OPTIMISATION PROCESS

5.1 OVERVIEW

The design and optimisation of a processor architecture is frequently an iterative refinement process as shown in Fig. 5.1. This process consists of alternating processor definition, practical or theoretical implementation, and application profiling stages.

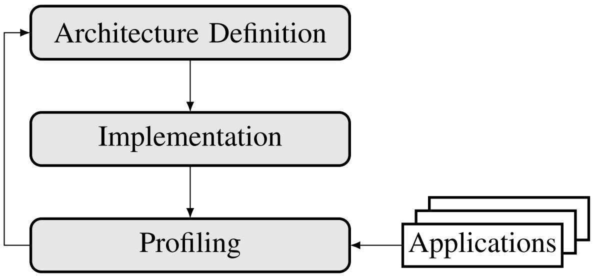


Figure 5.1: Architecture Design Process

The application profiling stage includes functional as well as performance verification, but also takes other factors such as resource usage and power consumption into consideration. After the first few iterations it becomes clear that this design process is a time consuming and tedious task of which many aspects can be automated. As such a software development environment was designed to automate this architectural design space exploration phase, bearing a close resemblance to the ASIP design methodology [88–91].

This chapter introduces the software development environment for the proposed architecture, followed by a description of the algorithmic implementation process within this environment. Each of the algorithms required for a typical radar application are then discussed from an implementation and optimisation perspective. The algorithmic implementations on the proposed architecture determine the quantity and type of functional units, while the respective optimisations thereof are used to iteratively refine the processor model.

5.2 SOFTWARE DEVELOPMENT ENVIRONMENT

A substantial portion of the architectural design process relies on a software development environment to enable efficient design feedback for debugging and optimisation of the architecture. To automate the design process of Fig. 5.1, both the hardware implementation and the relevant software tools are generated from an architectural definition. The design flow of this software-based approach is shown in Fig. 5.2.

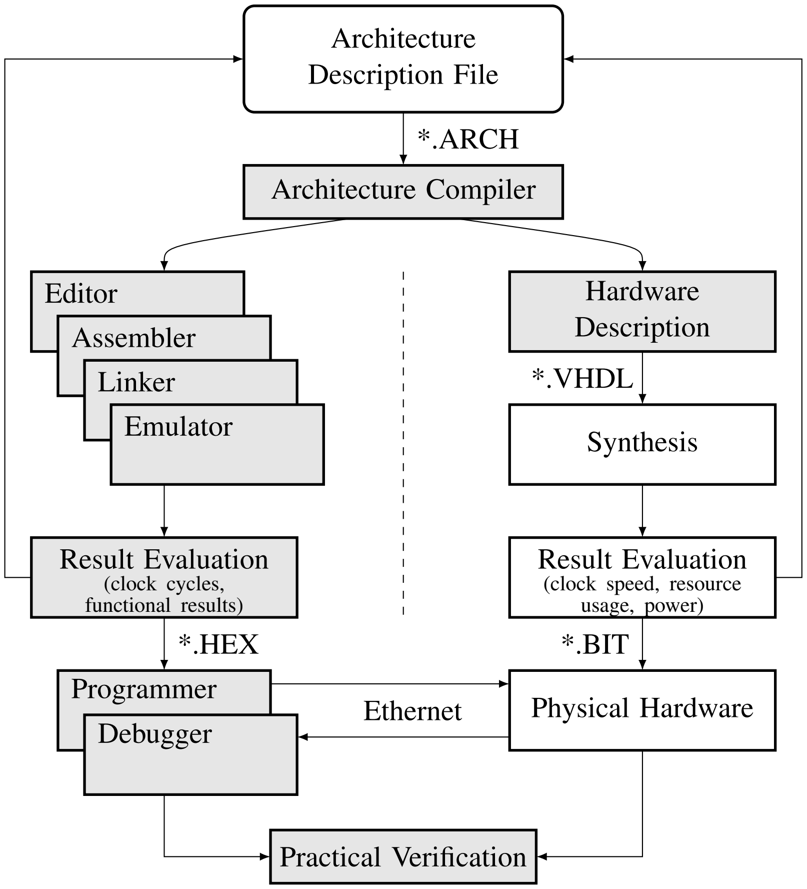


Figure 5.2: Software-based Architecture Design Flow

The architecture is defined by an architecture description file (**.ARCH* file), which forms the foundation of the entire software development environment. It consists of a list of the various functional units, registers, and their respective input as well as output ports for interconnection purposes. Refer to Appendix D for an example of this architecture description file.

From a software architecture point of view, each functional unit is an inherited object with a virtual execute function, VHDL instantiation template and graphical drawing routine (additional functional units are easily added into the software development environment by copying the functional unit template object and reimplementing these functions). Based on the **.ARCH* file, the software development environment creates one or more of each functional unit type, and stores these instantiations in a list. This functional unit list is then used to generate the VHDL source files for the processor core implementation, a graphical depiction of the processor architecture, and all the required development tools such as code editor, assembler, linker, cycle accurate emulator / simulator, debugger and programmer.

Since the architectural template (defined in Chapter 4) has a regular and well defined structure, the generation of a synthesisable RTL model for the target processor is a trivial process. Firstly, all the output ports of the functional units in the instantiation list are grouped into either integer or floating-point categories. Each functional unit output port has a unique name, which is used for declaring the VHDL signals for both categories. Both integer and floating-point register signals are declared as arrays. A generate statement iterates over all register indexes, instantiating a multiplexer with input signals from the respective integer or floating-point categories. The select signal of the multiplexers is a fixed slice of the program word, calculated by the register index. Objects in the functional unit list locally store the associated input register indexes, linking to the register array. When the instantiation function of an item in the functional unit list is called, the code segment of that functional unit type is written to the VHDL file with its local register indexes. For most functional units on the integer side, this is a single line of source code using the VHDL syntax of the respective mathematical operation. The functional units on the floating-point side instantiate components that were generated using the Xilinx CORE Generator tool [11] (for the Altera design flow, similar components can be generated with the MegaWizard Plug-In [12]). Lastly common elements such as the debug registers and the condition pass logic are instantiated based on a pre-written code segment.

Together with the board specific HDL-based hardware abstraction layer (HAL) files, the generated VHDL design files are then synthesised using the vendor specific FPGA tools (in this case Xilinx ISE, but similarly on Altera Quartus II). Timing results, functional accuracy, resource usage, profiling and performance data are then analysed and used by the designer to further refine the architecture through the architecture description file as shown in Fig. 5.2. The generated **.BIT* file is programmed into flash memory on the development board. When the development board boots up, the FPGA is loaded with this firmware containing the processor core, memory and high-speed analogue converter interfaces.

The application running on the processor core (*.*HEX* file) is then programmed via the gigabit Ethernet interface. The programming tool of the development environment supports scripted memory pre-loading (exported from MATLAB or the development environment in *HEX*, *BIN* or *TXT* formats) and exporting after execution is complete. Once the hardware platform is released from reset, an optional debugger trace can be enabled from the Xilinx ChipScope interface. This debug trace is used to compare the runtime results against the simulated results on a cycle by cycle basis to ensure congruency. These mechanisms simplify the practical verification of the proposed architecture, by running the application directly on the high performance development board. This can also have considerable performance benefits, as the practical runtime is typically a few orders of magnitude faster than the simulation time.

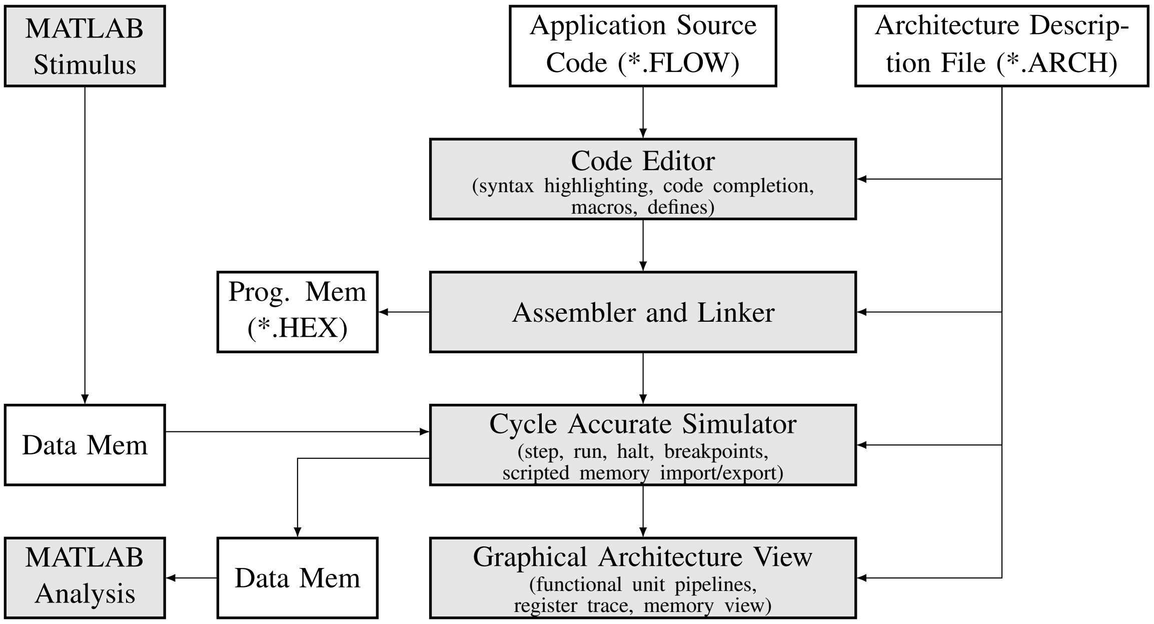

MATLAB Stimulus
Application Source Code (*.FLOW)
Architecture Description File (*.ARCH)
Code Editor
(syntax highlighting, code completion, macros, defines)
Prog. Mem (*.HEX)
Assembler and Linker
Data Mem
Cycle Accurate Simulator
(step, run, halt, breakpoints, scripted memory import/export)
MATLAB Analysis
Data Mem
Graphical Architecture View
(functional unit pipelines, register trace, memory view)

Figure 5.3: Software-based Simulation Flow

The left side of the design flow in Fig. 5.2 is more concerned with the development and simulation aspect of the processing architecture. Similar to the right side, the *.*ARCH* file is used as the foundation for the development tools as shown in Fig. 5.3. For the code editor, it provides the mechanism for syntax highlighting and dynamic code completion. For each assignment in the *.*FLOW* language, the code completion determines which assignments are possible with the selected register and pops up with the relevant functional unit outputs. Macro's and defines using compile time evaluated expressions for both floating-point as well as integer constants are also possible in the editor, easing maintainability and ensuring better code readability.

The *.*FLOW* language maps almost directly to the assembly datapath routings on the proposed architecture. Each assignment line in the source code simply determines the constant on the select signal of the related multiplexer. The second pass of the assembly process thus only involves a lookup in a map, followed by setting the appropriate bits in the program word. The first pass determines label addresses and resolves defines and macros such that the constants are updated correctly on the second pass of the source code. When the entire application has been assembled, the program memory is exported and written to a *.*HEX* file, which can be programmed onto the physical hardware as described earlier.

The assembled source files are also used by the emulator. This cycle accurate simulator plays an important role during debugging, architecture design, code profiling and understanding the dataflow paths in the proposed architecture. As such a substantial amount of effort has been placed into data visualisation and ease of use. Fig. 5.4 depicts the methodology of the simulator showing how data flows between registers with each instruction step. With each consecutive clock cycle step in the simulator, a time instance of the architecture is appended to the bottom of the current view, updating the relevant register values and latency slots of multi-cycle functional units. This mechanism allows the user to trace changes in the registers and latency slots between clock cycles by simply scrolling up and down.

Fig. 5.4 represents the dataflow of an envelope calculation. Registers are marked blue, while latency slots are shown in orange. Instructions 0 and 1 are identical, routing the data memory output to the two multipliers, their output to the adder, and its output to the square root operation, which in turn is connected to the input port of the memory. On the integer side, an increment functional unit is used as a counter with its output connected to the memory read address. Functional unit outputs are given in the last of the orange boxes, with multi-cycle (pipelined) operations shifting down the values in the pipeline each clock cycle.

In run mode, the graphical view is suspended for simulation performance reasons, and only resumed once a breakpoint is reached. Breakpoints can be set at source-code line numbers, defined in-line, called once a cycle count is reached or be more complex data-matching criteria. In-line breakpoints also allow scripted data memory imports and exports (text, binary or hexadecimal formats) at multiple

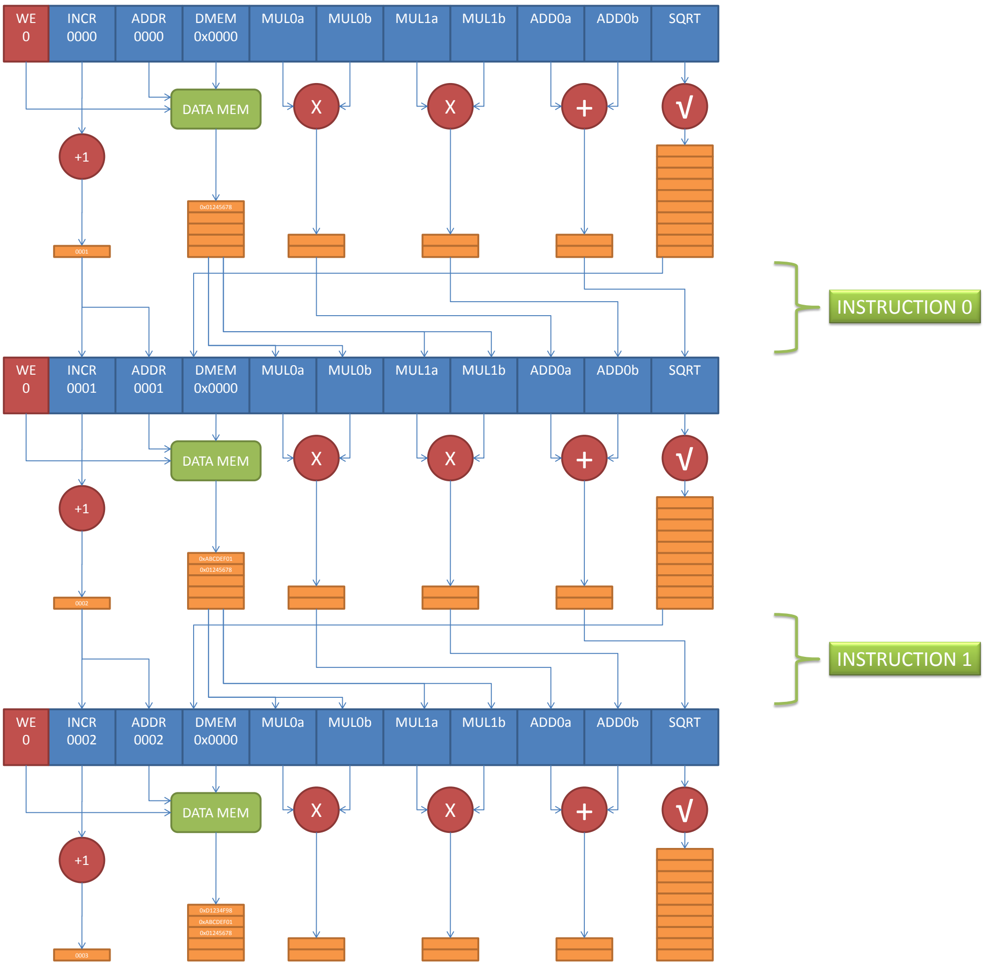


Figure 5.4: Proposed Architecture - Simulator Methodology

locations in the program code, making it easy to verify execution stages in the program flow by analysing the data with tools like MATLAB.

Functional verification and performance profiling are important aspects of the simulator during algorithm development. The design flow typically involves implementing and compiling the algorithm in the *FLOW* language, pre-loading the data memory, running the simulator (or releasing the hardware from reset), recording the elapsed clock cycles, exporting the memory and finally verifying the results for functional correctness. The development tools were designed with this principle in mind, integrating the verification process into a ’Run on Hardware’ and ’Run in Simulator’ button. Refer to Appendix D for screen-shots of the software development environment, as well as source code examples in Appendix E.

5.3 ALGORITHM IMPLEMENTATION PROCEDURE

The previous section introduced the software development environment for the proposed architecture. The software-based design flow reduces the design time and overcomes many typical error-prone design challenges through automatic generation of the relevant design tools and the RTL-model. It is thus possible to change the underling processor architecture by simply changing the architecture description file (e.g. creating new functional units or experimenting with different functional unit combinations and number of registers). Such a process enables the gradual refinement of a processor architecture for an optimised solution based on a particular set of algorithms and applications.

The implementation of this set of algorithms is covered in the following subsections. The software-based design process is used for the algorithm implementation and verification, facilitating the architecture refinement in the process. The aim is to determine the hardware/software boundary for each algorithm; finding a balance between performance and re-usability of each functional unit. On the one extreme, a functional unit could be an optimised hardware coprocessor for a specific algorithm, while the opposite extreme would rely on a few transcendental RISC functional units for all algorithms. Regardless, a sufficient quantity and the appropriate types of functional units need to be included in the final architecture to support all the operations described in Appendix B. The following steps outline the general procedure that was followed in the implementation of the various algorithms on the proposed architecture template.

1. Draw the dataflow graph for the data processing section of the algorithm; Start by drawing the memory read ports, showing the data moving through the mathematical functions all way the to the memory write ports with connecting lines. Ideally all these operations should happen simultaneously in a single clock cycle in the inner loop to create a pipeline.
2. Are there enough functional units for each of the mathematical functions in step 1? If not, split the dataflow into smaller portions or add more functional units. If yes, are there enough functional units / memory ports to process two or more parallel data streams simultaneously?
3. Draw the control-flow graph for the memory address generation
4. Is an outer loop required? What changes each outer loop?
5. Convert the dataflow and control-flow graphs into *FLOW* language assignments. Use the software development environment to calculate the write latency and adjust the memory write enable strobe delay.

Based on these steps, the majority of operations follow a straightforward mapping to the architecture described in Chapter 4. During this implementation process, it becomes clear if an algorithm would benefit from having more mathematical functional units or a specialised one, implicitly defining the boundary and trade-offs between performance for an individual algorithm and general purpose performance for all algorithms. This implementation and optimisation effort is used to determine both the type and quantity of functional units, which are required for the set of algorithms that are typical in radar signal processing. The implementation options for each of the algorithms and mathematical operations are discussed in the following sections.

5.3.1 Envelope Calculation

The envelope calculation (linear magnitude operation) is well-suited for the illustration of the implementation process, and is thus covered first. The signal flow graph of this operation is shown below in Fig. 5.5.

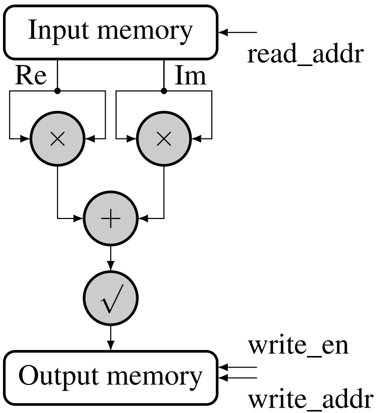


Figure 5.5: Signal Flow Graph for the Envelope Operation

For the squared magnitude and log-magnitude envelope calculations, the square root operation is removed or replaced by a logarithm and multiplication by 2 stage respectively. The above signal flow graph simply translates to the following dataflow routings on the proposed architecture:

```
FMul0_a = DMem0_o     ; RE*RE
FMul0_b = DMem0_o
FMul1_a = DMem0_p     ; IM*IM
FMul1_b = DMem0_p
FAdd0_a = FMul0_o
FAdd0_b = FMul1_o     ; RE*RE + IM*IM
FSqr0_a = FAdd0_o     ; SQRT(RE*RE + IM*IM)
DMem0_a = FSqr0_o
```

Every move operation in this listing occurs in parallel, making the process a software pipeline capable of producing a new output every clock cycle. The write enable signal is asserted when the first output is available from the square root operation. This processing latency entails the *DMEM_LAT* + *FMUL_LAT* + *FADD_LAT* + *FSQR_LAT*, which initialises the write-inhibit counter, controlling the data memory write enable signal through a comparator. The number of iterations of the inner loop instruction is controlled by the control-flow methods of Section 4.5. A counter is connected to the RCon comparator that controls the condition pass flag; when it fails the program counter flow continues normally and exits the inner loop. Read and write addresses can be generated with simple linear up-counters and some initial start addresses as shown below:

```
Cons0_w0 = READ_ADDR
Cons0_w1 = WRITE_ADDR - (DMEM_LAT+FMUL_LAT+FADD_LAT+FSQR_LAT)
DMem0_raddr = Cons0_w0
IInc0_a = Cons0_w0
IInc1_a = Cons0_w1
||                      ; inner loop:
IInc0_a = IInc0_o       ; increase counter0
DMem0_raddr = IInc0_o   ; assign counter0 to read address
IInc1_a = IInc1_o       ; increase counter1
DMem0_waddr = IInc1_o   ; assign counter1 to write address
```

This implementation implies a two-word-wide data bus on the input memory side, and a single word-wide data bus on the output side. Another implementation option is to integrate the envelope operation into the write stage of the previous operation, reducing the cycle count by an iteration over the entire data set. If the data memory cannot read and write simultaneously (or the data bus is only a single word wide), the dataflow graph would have to operate in a lock-step fashion with alternating memory accesses. Similarly, when more than one memory data bus is available, parallel streams of execution are possible provided enough multipliers, adders and square root functional units are accessible.

However, even if multiple two-word-wide data buses and sufficient multipliers and adders are available, adding a second square root operation (for a speed-up by a factor of 2) to the architecture might not justify the overall performance improvement gained by this addition. Based on the percentage processing time spent on the envelope calculation as shown in Table 2.2, an additional functional unit might be better suited to slightly improve the performance of another algorithm of higher importance.

5.3.2 FIR Filter Operation

The FIR filter structure of Fig. 5.6 can be mapped directly to a hardware implementation. For any given filter length, a filter of lower order is realised with the same hardware implementation by setting the unused coefficients equal to zero, at the expense of additional latency cycles. With such a FIR filter structure as a functional unit (single data input and output port), coefficients could be address-mapped into the main memory space for a high performance streaming solution.

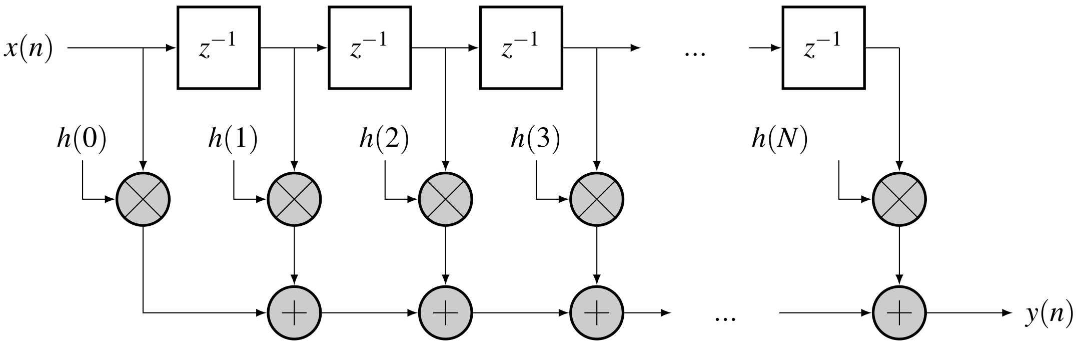


Figure 5.6: Direct-form FIR filter implementation

However, the multiply reduction structure of this hardware implementation cannot be re-used by other processing operations that would otherwise benefit from a similar structure. The vector dot product, convolution, correlation, interpolation and the matrix multiplication algorithms all require element-wise multiplication followed by a reduction of the products into a single result via summation. As such all of these operations can benefit from the dot product structure as depicted in Fig. 5.7.

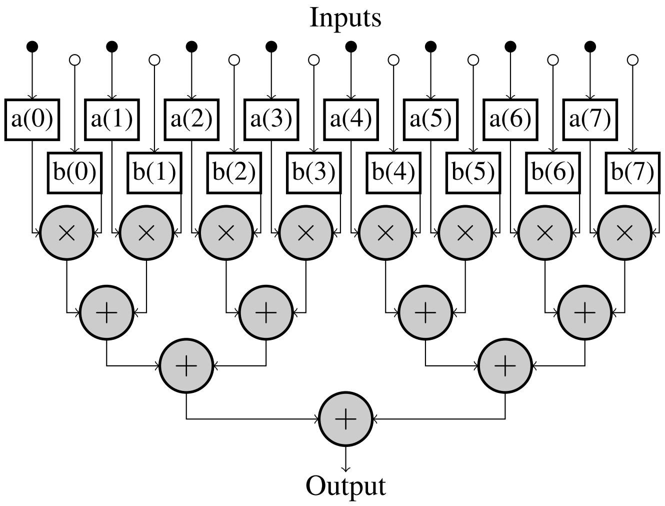


Figure 5.7: Eight Point Dot Product Hardware Implementation

Each of the input ports are connected to registers in the proposed architecture. Assigning registers the value of the register to the right, shifts the entire data-stream to the left, allowing new input data to be clocked in on the right-most port every clock cycle. This connection effectively implements a streaming FIR filter of length 8. For higher order filters, the entire data-stream is filtered with the first 8 coefficients, and the output stream written to a temporary memory location. The next 8 coefficients are then loaded into the relevant registers, followed by a second filtering stage that reads the previously calculated partial results and adds them to the calculated output stream, writing the results back to memory. This process is repeated until the required filter length is reached. Care must be taken with the number of iterations and the output address offset, as they decrease and increase by 8 after each partial filtering stage respectively.

5.3.3 FFT Operation

Similar to the FIR filter, the FFT operation could also be implemented as a coprocessor in a functional unit with a single input and output port. A suitable streaming pipelined FFT architecture is given in [101], which could be coupled with FIFO buffers on the input and output ports with flags indicating processing start and completion events.

Only supporting a few selected point sizes and no resource reuse, an optimisation of the core architecture was chosen over the FFT coprocessor alternative. Section B.4.3 introduces the Radix-2 FFT operation from a dataflow perspective. The FFT butterfly translates to the signal flow graph in Fig. 5.8 on the proposed architecture.

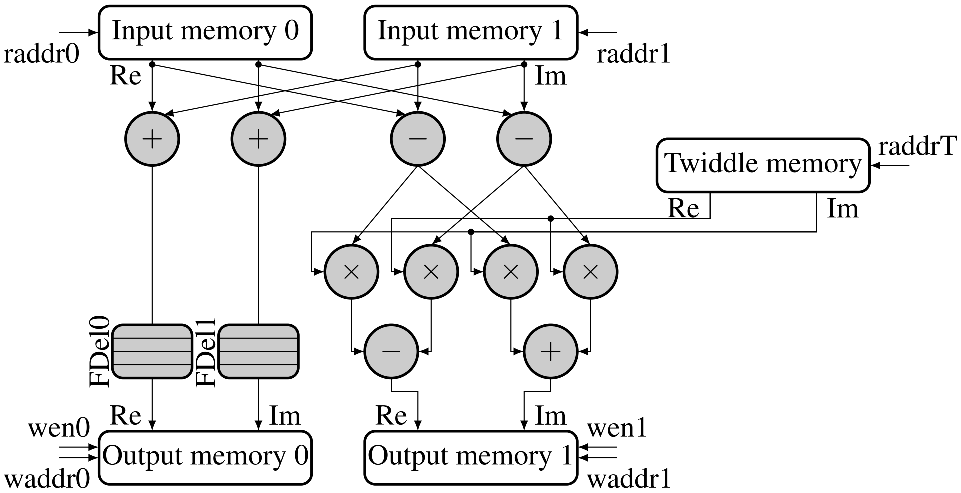


Figure 5.8: Signal Flow Graph for the FFT Operation

Fig. 5.8 depicts the real functional units that are required for a streaming approach. As long as the read and appropriate write address are changed every clock cycle, a new butterfly can be calculated every instruction cycle. The latency between memory read and memory write is:

$$PROC_{LAT} = DMEM_{LAT} + FSUB_{LAT} + FMUL_{LAT} + FADD_{LAT}. \tag{5.1}$$

The control-flow and address generation for the FFT operation control ‘wen0’, ‘wen1’ and ‘raddr0’, ‘raddr1’, ‘waddr0’, ‘waddr1’ respectively. Since the read data is written back to the same memory locations after the butterfly computation, the write addresses ‘waddr0’ and ‘waddr1’ are simply the read addresses ‘raddr0’ and ‘raddr1’ delayed by the processing latency $PROC_{LAT}$.

From the FFT dataflow pattern in Fig. B.3 it is clear that there are data dependencies between stages. If memory is read in stage 2 before it is written back to memory in stage 1, invalid values would be produced. This adds overhead between processing stages since the control-flow has to suspend reading data for the next stage until the $PROC_{LAT}$ of the previous stage has elapsed. To hide this latency, multiple FFT operations can be executed stage by stage.

Another issue is the alignment of output memory 0 and 1. The ‘output memory 0’ write enable strobe needs to be asserted after $DMEM_{LAT} + FADD_{LAT}$, or delayed along with its input data to match the latency of the ‘output memory 1’ calculation. In this implementation the data is delayed to simplify the control-flow.

The address generation pattern for an FFT with N=16 (at an offset of 0x0000 in data memory) is shown below in Table 5.1.

Table 5.1: FFT Address Generation Pattern (N=16, format="addr0:addr1 / addrT")

	Stage 0	**Stage 1**	**Stage 2**	**Stage 3**
Iteration 0	00:08 / 0	00:04 / 0	00:02 / 0	00:01 / 0
Iteration 1	01:09 / 1	01:05 / 2	01:03 / 4	02:03 / 0
Iteration 2	02:10 / 2	02:06 / 4	04:06 / 0	04:05 / 0
Iteration 3	03:11 / 3	03:07 / 6	05:07 / 4	06:07 / 0
Iteration 4	04:12 / 4	08:12 / 0	08:10 / 0	08:09 / 0
Iteration 5	05:13 / 5	09:13 / 2	09:11 / 4	10:11 / 0
Iteration 6	06:14 / 6	10:14 / 4	12:14 / 0	12:13 / 0
Iteration 7	07:15 / 7	11:15 / 6	13:15 / 4	14:15 / 0

Note how the difference between addr0 and addr1 is always $N >> (stage+1)$. Thus the following address generation code applies:

$$\begin{aligned} addr0 &= b*(N>>s)+i \qquad (5.2) \\ addr1 &= addr0+N>>(s+1), \qquad (5.3) \end{aligned}$$

where b represents the block number, s is the current stage number and i is the current index value. The difference between consecutive addresses is 1 unless the end of a block is reached. The end of a block is reached when the current index value exceeds $(N>>(s+1))$. The following pseudo code is thus performed every iteration to increase the index and the block number.

```
if (i++ > (N>>(s+1)))
    i = 0
    b++
```

The stage number s is increased and the block number reset when the number of executed butterflies in the current stage exceeds $N/2$. The following code depicts how this is done:

```
if (j++ > (N/2))
    s++
    b = 0
```

The twiddle factor address can be calculated from the current index value and the stage number as shown below:

```
addrT = i<<s
```

This address access pattern could be precomputed and stored in the program memory, with each instruction cycle accessing and writing back a different memory location. The precomputed access pattern is however fixed to a certain size of N, and uses unnecessary amounts of program memory. For bigger sizes of N, this address access pattern becomes excessively large and impractical for the proposed long instruction word architecture. The preferred method would thus be to implement the address generation circuitry and dynamically calculate the address access patterns. The following address flow graph depicts how this address generation scheme is implemented on the proposed architecture (Fig. 5.9)

The blue dashed lines represent registers that are re-initialised before each stage, while the red register values are set during the function initialisation. These initial conditions are set with the traditional control-flow methods.

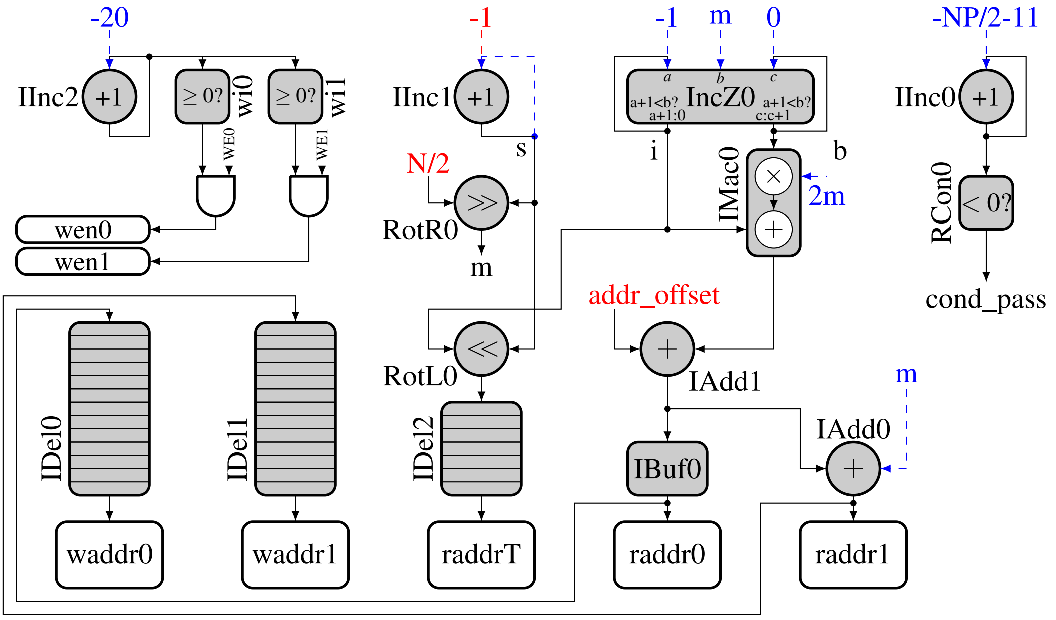


Figure 5.9: Address Flow Graph for the FFT Operation

The *IInc0* counter keeps track of the number of butterflies to be calculated in each stage. Once it reaches 0, the *condpass* flag clears and the inner loop is interrupted. *PC0_a* is no longer assigned to *PC0_o* and the control-flow continues sequentially rather than looping at the inner loop FFT instruction. The outer loop then initialises the registers for the next inner loop stage.

RotR0 calculates the new m value when the stage number is increased. Since the m value halves for each consecutive stage, the m value from the previous stage is the current $2m$ value. After each stage, the current m value (which becomes $2m$ in the next stage) is read from *RotR0_o* and assigned to *IMac0_b*, followed by assigning *IInc1_o* to *IInc1_a* to increase the stage number. The newly calculated m value (on *RotR0_o* a clock cycle later) is then assigned to *IncZ0_b* and *IAdd0_b*. The write inhibit counter (*IInc2_a*) is reinitialised to -20 to account for the address generation, memory read and data calculation latency. Similarly, the i and b values are reset by writing -1 and 0 to *IncZ0_a* and *IncZ0_c* respectively. The $-\frac{NP}{2}-11$ value requires an additional Buffer/Adder/Increment functional unit, so it can be assigned to *IInc0* after each stage without being recomputed and adding more latency into the outer loop.

Bit-reversed addressing could be incorporated into the last stage of the address generation logic. The delay just before the write addresses could be reduced, and the bit reversal function inserted in between. Since the twiddle factors are 0 for the last stage, the IRev and RotL functional units are not used at the same time and could thus be secondary functional units.

A Radix-4 FFT requires $\log_2 N$ stages each consisting of $N/4$ butterflies, making it 4 times as fast as the Radix-2 implementation if a butterfly is executed every clock cycle. However, for a Radix-4 FFT implementation capable of executing a butterfly every clock cycle, the required number of functional units become excessive on the proposed architecture. A single radix-4 butterfly consists of 8 complex adders/subtractors and 3 complex multipliers and needs a data memory with 4 simultaneous complex reads and writes as well as a twiddle factor memory with 3 complex read ports. This translates to 22 real adders/subtractors, 12 real multipliers, 4 DMem and 3 CMem units on the datapath side. Unless the radix-4 butterfly is implemented as a single functional unit, the number of functional units and thus the number and size of multiplexers cause a bottleneck and reduced the maximum clock frequency. Additionally, quad-ported memory with simultaneous read and write capability further reduce the maximum clock speed.

5.3.4 Transpose Operation

Transposing or corner-turning a matrix simply requires the output port of the memory to be connected to its input port. The appropriate address and write enable signals then need to be generated with the traditional control-flow mechanisms. This requires the *IncZ* functional unit to generate the row and column addresses, which are connected to the *IMac* unit to calculate the transposed linear write address. An additional integer adder is required for writing back the result to an offset in memory (any address other than 0). An *IInc* functional unit is then used for the input address, the write inhibit signal as well as the inner loop counter.

For data buses wider than the individual data elements, the transpose operation becomes a bit more complex, requiring a large amount of registers to buffer the read data before it is reassembled into a packed data-word and written back to memory.

5.3.5 Summation Operation

The block summation for a sliding window or moving average calculation is easily implemented with a log-sum structure [102] as shown in Fig. 5.10. The latency of the adders does not affect the operation, provided it remains the same for all adders in the log-sum structure. Each additional stage that is added into the structure requires a delay register of $d = 2^{stage-1}$ clock cycles. With the depicted structure consisting of 3 stages, $N = 2^{stages} = 8$ values will be summed. Depending on how

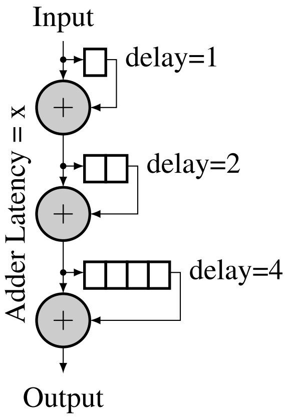


Figure 5.10: Log-Sum Technique for a Moving Average Calculation

many values need to be summed, $\log_2(N)$ adders are connected in series with the appropriate delay lines using the *FDel* functional units.

The *FDot* instruction could also be used for summing or averaging 8 numbers with the multiplication factors set to 1 or 1/N respectively. Since input data can be left shifted, the FDot instruction acts as a sliding window over the input samples, summing the 8 previous values.

5.3.6 Sorting Operation

For standard sequential CPUs, data dependent sorting algorithms like heapsort or quicksort are well-suited because of the dynamic branch prediction, speculative execution and high clock rates of the CPU architecture. For dataflow based processors, data independent sorting networks are better suited as discussed in Section B.8. These sorting networks typically require more operations than the equivalent sorting algorithm, but do not suffer from pipeline stalls or control-flow restrictions. They can thus be statically scheduled and use deeply pipelined arithmetic compare and swap operations.

From an implementation point of view, the dual ported data memory outputs are connected to the two input ports of the *FSwap* operation, of which the outputs are connected to the memory input ports again. The *FSwap* operation simply compares the two input ports, and outputs the larger value to the *p* port, while the smaller value is outputted on the *o* port. The control-flow section of the sorting operation then simply generates the addresses and delayed write enable strobe according to one of the sorting network structures from Section B.8. The latency of the datapath needs to be considered because of data dependencies between stages, requiring delays between stages or a rearrangement of comparators in the consecutive stage.

5.3.7 Timing Generator

The timing generator can be implemented with a control-flow loop that waits a certain number of iterations before continuing. It calls both the DAC playback and the ADC sampling routines and keeps track of the number of pulses required for the burst. It also increases the write address for the ADC sampling routine to ensure the samples are written to the appropriate memory locations. With such a core implementation of the timing generator, the operations before the corner turn could be completed during the PRI dead-time, shortening the processing overhead after the completed burst.

Regardless, the core implementation of the timing generator restricts the minimum burst repetition interval (BRI) of the radar, as processing cannot occur during the sampling and playback operations. To support pipelined sampling and processing, a dual-ported memory (in addition to the data memory) with a separate timing controller is required.

5.3.8 Pulse Generation

The transmit pulse is stored in data memory and played back once the timing generator calls the dac_playback function. From an implementation perspective, the data memory output ports are directly connected to the DAC ports, and enabled once the DAC_EN flag is high. The pulse playback function then simply needs to point the read address to the memory location of the transmit pulse and linearly increase the address until the end of the pulse is reached.

Using the *FSinCos* and *FMul* functional units in the datapath and the *IInc* together with *ItoF* functional units on the integer side, the transmit waveform could also be generated dynamically during execution.

5.3.9 ADC Sampling

The ADC sampling process is called from the timing generator. A direct connection from the ADC to the memory input ports is provided, and if the CUSTOM_SELECT signal of the multiplexer is enabled via the program word, ADC data is directly streamed into the main data memory. The ADC sampling process then simply linearly increases the write address with an *IInc* counter.

Similarly to the DAC, 4 interface ports are provided. Each data memory interface (*DMem0* and *DMem1*) provides two write ports, which are normally used for real and imaginary components. In the ADC sampling process all 4 input ports are used for receiving de-serialised data from the ADC, which thus runs at 4 times the clock speed of the processor.

If data sampling rates higher than 4 times the processor clock frequency are required, a dual ported block memory can be mapped into the *CMem* space and sampled or played out with flags. This method allows the processor to simultaneously sample and process the previous data as the core is not involved in the sampling process.

5.3.10 IQ Demodulation

The IQ demodulation algorithm is implemented by calling the FIR filter function with the Hilbert filter coefficients loaded into the coefficient memory. Since the Hilbert transform produces the Q component from the I component, the I component needs to be delayed by the group delay of the filter to align the real and imaginary parts. A decimation stage removes the unnecessary data, since the full bandwidth is still preserved with complex data. From an implementation perspective, both the alignment and the decimation can be done with a single iteration over the filtered data in memory.

5.3.11 Channel Equalisation

The channel equalisation is implemented with a simple FIR filter function call. A FIR filter with 32 coefficients requires 4 iterations over the input data with the length-8 *FDot* implementation.

5.3.12 Pulse Compression

The pulse compression operation can be realised with either a matched filter in the time domain, or a fast filtering operation by means of a spectrum multiplication in the frequency domain. With larger number of transmit pulse samples, the frequency domain method becomes computationally more efficient than the time domain matched filter.

The frequency domain method involves converting the sampled range-line to the frequency domain via the FFT, performing a complex multiplication by the precomputed spectrum of the transmit pulse, and performing an inverse FFT again. The FFT and IFFT translate to a functional call to the P×N-point FFT subroutine with the forward and reverse twiddle factors loaded into memory respectively. The spectrum multiplication and bit-reversed addressing can be done in a single iteration of the range-pulse map for each burst between the FFT and IFFT. After the IFFT operation, the multiplication by

$1/N$, bit-reversed addressing, the matrix transpose and Doppler window multiplication can all be done in a single iteration.

The time domain matched filter simply involves a function call to the FIR filter routine, with the conjugated and time reversed transmit pulse samples as coefficients.

5.3.13 Corner Turning

The corner turning operation is simply a matrix transpose, and is implemented as discussed in Section 5.3.4. For the fast correlation implementation of pulse compression, it can be incorporated into the bit-reversed addressing after the IFFT operation.

5.3.14 Moving Target Indication

The moving target indication algorithm simply removes stationary clutter with a filter. It thus involves a call to the FIR filter function with the appropriate coefficients loaded into memory.

5.3.15 Pulse-Doppler Processing

Pulse-Doppler processing simply involves performing an FFT operation over the pulse-to-pulse samples in each range-bin, or simply a call to N×P-point FFT subroutine. To suppress side-lobes, a window can be applied prior to the FFT. This window multiplication can be done as part of the matrix transpose during the bit reversed addressing of the pulse compression IFFT operation.

5.3.16 CFAR

The CFAR implementation depends primarily on the algorithm class that is selected. For the simple case of 1 dimensional cell averaging CFAR, a call to the moving average function (or FIR filter function with coefficients set to 1) is required before the threshold comparison. Each moving average value is then multiplied by the CFAR constant and compared against the cell under test, requiring two simultaneous read ports. The threshold comparison is done with the *FSub* and *FCon* functional units. When the *FCon* greater than 0 condition passes, the program flow loop is interrupted and adds the current range and Doppler index into a list. Alternatively, the detections could be marked on the range-Doppler map.

For the two dimensional CA-CFAR, the averaging of any number of reference cells is achieved by first computing a summed area table by looping over the vertical and horizontal dimensions of the range-Doppler map and accumulating the sums. Since the *FAdd* functional units are multi-cycle operations, partial sums are kept for multiple range-lines to sustain the maximum throughput (limited by the data memory access speeds). The reference cell summation can then be calculated by reading the edges of the reference window in the sum area table as described in Section A.13.1.1. The threshold comparison and target listing is then performed similar to the 1 dimensional case.

Other CFAR classes follow a straightforward implementation by calling the appropriate sorting, summed area table or moving average calculation functions.

5.3.17 Data Processor Interface

The data processor interface consists of assembling the target detection list from the CFAR process into a packet that can be transmitted to the data processor via a communication peripheral. The different peripheral interfaces are FIFO memory mapped into the address space of the *CMem* functional unit. For most instances, the data processor is a standard PC with a gigabit Ethernet connection, which is also used for displaying the radar data. In the next chapter, the required functional units are grouped together, and the final architecture is proposed.

CHAPTER 6

FINAL ARCHITECTURE

6.1 OVERVIEW

This chapter presents the final architecture with all the necessary functional units. Each functional unit is evaluated from the perspectives of generality, algorithmic reuse, and performance. Too fine-grained functional units add overhead to the core architecture, while too coarse-grained functional units are only applicable to very specific algorithms and cannot be reused for other similar algorithms. A processor consisting of only hard-wired coprocessors optimised for a specific application may be optimal in terms of performance, but lacks flexibility for algorithmic variations or future algorithms. Similarly, a processor architecture that is too focused on general purpose computing usually lacks the real-time performance from a signal processing perspective. The aim is thus to balance these two extremes, providing a solution that is as customisable as possible, whilst still delivering the required performance for radar signal processing.

6.2 AMALGAMATION OF THE DIFFERENT FUNCTIONAL UNITS

This section serves to determine the required number and input connections of the different functional units. Table 6.1 and 6.2 summarise the functional unit requirements for the various signal processing operations, control-flow constructs and radar algorithms as determined by the implementations in the previous chapter.

Several functional units are interchangeable or can be used for purposes other than the intended one. For example, the *IAdd* can be used as an additional *IInc*, as a subtractor, a decrement function or simply as a temporary register in form of an *IBuf*. Many of the integer functional units are only used in the loop or address setup calculations, and do not form part of the inner loop.

Table 6.1: Integer functional unit usage

	Cons	**PC**	**Stack**	**RCon**	**IDebug**	**IInc**	**IAdd**	**ISub**	**IMac**	**IBuf**	**IDel**	**RotR**	**RotL**	**IRev**	**IncZ**	**ItoF**	**IReg**	**IIOPort**
General Purpose	✓	✓	✓	✓		✓	✓	✓	✓							✓	✓	✓
Branch (Goto)	✓	✓																
Call	✓	✓	✓															
Conditional Branch	✓	✓		✓														
Loop	✓	✓		✓		1												
FFT	✓	✓	✓	✓		4	2	1	1	1	3	1	1		1			
FIR	✓	✓	✓	✓		4	2	1	1		1							
CA-CFAR	✓	✓	✓	✓		4	3											
Moving Average	✓	✓	✓	✓		3	3											
Spectrum Multiply	✓	✓	✓	✓		3	3								1			
Bit reverse	✓	✓	✓	✓		3	3		1	1			1	1	1			
Envelope	✓	✓	✓	✓		3	3											
Windowing	✓	✓	✓	✓		3	3											
Transpose	✓	✓	✓	✓		2	3		1						1			
Memory Copy	✓	✓	✓	✓		3	3											
Memory Clear	✓	✓	✓	✓		1	3											
Delay	✓	✓	✓	✓		1		1										
Signal Generation	✓	✓	✓	✓		2		1								1		
DAC Output	✓	✓	✓	✓		2	1	1		1								
ADC Sample	✓	✓	✓	✓		2	1	1		1								

The majority of callable functions (with arguments: read address memory offset, write address memory offset, and number of iterations) simply require 3 *IAdd*, 3 *IInc* and the standard *Cons*, *PC*, *Stack* and *RCon* functional units on the integer side. These functional units provide loop control, read and write address generation with offsets as well as a variable write delay counter. Regardless of the operations on the floating-point data side, the address generation and control-flow model (of reading data memory in a linear stream and writing it back to memory some variable clock cycles later) remains constant.

For all of the above operations, a core architecture optimisation was chosen over the coprocessor alternative. The coprocessor option provides a usable solution for a few fixed applications, but does not allow for resource re-use or optimisation. If the performance from the core is not sufficient, a coprocessor can easily be added at a later stage.

As explained in Section 4.4, the number of combined output ports of all functional units determines the multiplexer size, directly influencing the required silicon area and maximum clock frequency. In the proposed architecture, each input register is associated with a multiplexer and requires a slice of the program word. The number of registers therefore directly determine the program memory

Table 6.2: Floating-point functional unit usage

	DMem	CMem	FCon	FReg	FDebug	FtoI	FAdd	FSub	FMul	FDiv	FDot	FBuf	FDel	FSwap	FSqr	FSinCos	FLog	FATan2
General Purpose	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓						
Standard Arith.	✓						✓	✓	✓	✓					✓	✓	✓	✓
FFT	2	1					3	3	4				2					
FIR	2	1					1				1	1						
CA-CFAR	2							1	1				2	1				
Moving Average N=8	1						3					1	2					
Transpose, Window	2								2									
Spectrum Multiply	2						1	1	4									
Bit reverse	1																	
Envelope	1						1		2						1			
Transpose	1																	
Memory Copy	1	1																
Memory Clear	2																	
DAC Output	2																	
ADC Sample	2																	
Polar->Rect Convert	1								2							1		
Rect->Polar Convert	1						1		2						1			1
Log-Sum N=16	1						4						3					
Log-Magnitude	1						1		3								1	
Dot Product N=8	1										1							
Phase Shift	1						1	1	4							1		
Matrix Multiply	2						1				1							
Scaling	1									1								
Compile Target List	1		1											1				
Sort	2													1				

width and the required silicon area for the multiplexers. Minimising the number of registers and the number of functional unit output ports reduces the required silicon area, while reducing the number of functional unit output ports increases the maximum clock frequency.

One method of minimising the number of registers is by connecting multiple functional units to the same registers. This is possible when the functional units are not used in parallel or require the same input connections. For example, both the floating-point adders and subtractors (connected to the memory output) for the FFT operation in Fig. 5.8 share the same input connections, and could thus share the same input registers (provided no other operation requires separate input registers). Similarly, the rotate left and bit-reverse functional units are never used concurrently in an inner loop, allowing common input ports between them.

For a throughput of one or more kernel operation every clock cycle, many algorithms benefit from data memory with two ports as well as a separate cache or coefficient memory. Each of the two data memory ports should be capable of reading and writing a complex number simultaneously every clock cycle. A single ported data memory causes many operations to become memory bound as the ALU datapath is idle every second clock cycle. Data memories with more than 2 simultaneous ports hamper the access time, throughput and memory clock speed, severely limiting the maximum clock frequency of the entire processor.

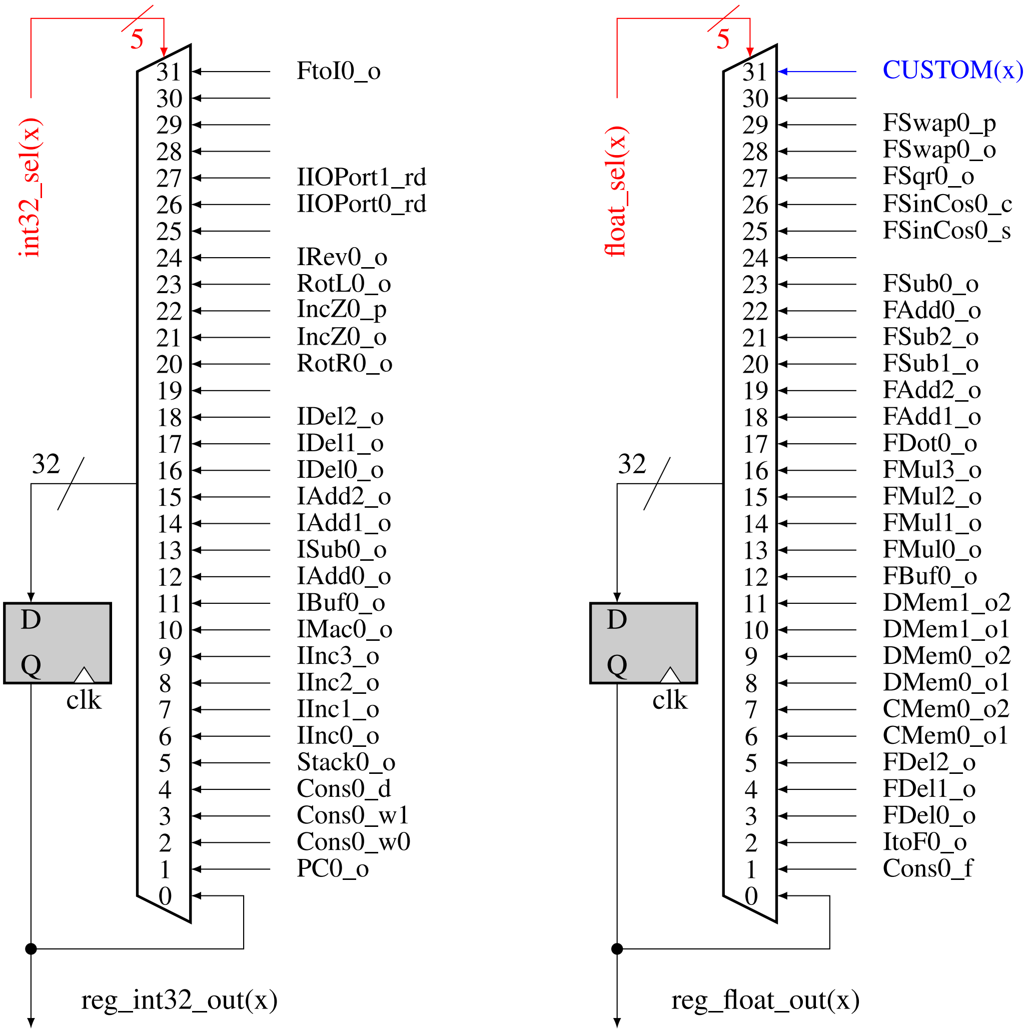
When such a dual ported data memory is used, some operations become ALU bound again, as there are not enough functional units. For example, the envelope operation could be sped up by a factor of 2, if another square root operation were available. However, the percentage processing time improvement as a function of the total processing time does not justify the additional resource requirements for a second square root functional unit.

Combining the arithmetic RISC-type functional units into more complex CISC-type functional units is also possible (e.g. complex multiplier, FFT butterfly). However, the core architecture provides a software defined mechanism of combining arithmetic functional units into a software pipeline without incurring additional overhead. Hardware combined functional units would thus have the same latency as the software pipelined implementation at the expense of not being able to reuse the arithmetic units for other operations.

6.3 FINAL ARCHITECTURE

Fig. 6.1 depicts the switching matrix and register architecture. The first multiplexer input is directly connected to its associated register output. When the select signal is zero, the register is assigned to itself and therefore remains unchanged. The input ports on the right of both Fig. 6.1a and Fig. 6.1b are the functional unit outputs. In this implementation, there are 32 multiplexer inputs (31 functional unit outputs) making the select signal for the multiplexer $log_2(N) = 5$ bits wide.

Registers are 32 bits wide and divided into integer as well as floating-point sides to avoid the multiplexers getting too large. The integer registers are used for memory address generation and program flow (e.g. branching and loop control), while the floating-point registers are used for data processing. The program memory width is determined by the number of registers and the multiplexer select width, and can become rather large. For the 92 registers used in this implementation, 460 bits are required for the multiplexer select signals. Additionally a 32 bit constant, a 4 bit condition code and 12 flags form the program word as shown in Fig. 6.2.



(a) Integer Side

(b) Floating-Point Side

Figure 6.1: Switching Matrix and Register Architecture

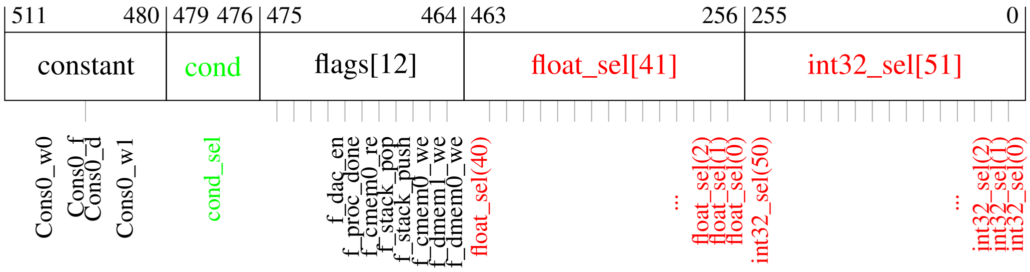


Figure 6.2: Program Word Format

The constant from the program word is also routed to the multiplexer input, making it possible to assign a value to any register. On the integer side, the constant can be split into two different 16 bit constants or kept in its 32 bit form.

On the floating-point side, the *CUSTOM(x)* multiplexer input allows a customised input specific to each individual register (blue lines in Fig. 6.1 and Fig. 6.4b). For example, the ADC data would typically be routed directly to the data memory input registers, and not to any other register. It would thus make sense to route the ADC data only to the multiplexer connected to the data memory input, and not to any other multiplexer. Similarly, the *FDot* inputs need to be shifted each clock cycle for FIR filtering, requiring a connection to the adjacent register via the multiplexer. No other registers require a direct connection to these input registers from the *FDot* functional unit, making additional global multiplexer ports pointless. Instead the *CUSTOM(x)* signal is used to route data between adjacent registers as shown in Fig. 6.4b.

The first register, the program counter, deviates slightly from Fig. 6.1. It uses the conditional code from the program word (cond_sel) to determine whether assignments to the program counter are permitted. If the condition check fails, or if no new assignment for the PC is selected, the multiplexer selects input port 0. Unlike the other registers however, input port 0 is not directly connected to the register output, but instead increased by one. Thus, if the condition passes the new value is assigned, else the program counter is increased and the program execution continues normally. Fig. 6.3 depicts the program counter architecture. Note that the program counter can still be assigned to itself, thus repeatedly executing the current instruction for looping or end-of-program purposes.

The final architecture is shown in Fig. 6.4a and Fig. 6.4b for both the integer and floating-point sides. Because of the independent select signals for each multiplexer, the proposed architecture provides direct control over horizontal as well as vertical instruction-level parallelism in both the data- and control-path.

6.4 ARCHITECTURE IMPLEMENTATION ON XILINX VIRTEX 5

The final architecture was implemented on a Xilinx Virtex 5 SX95T FPGA for verification purposes. The various floating-point arithmetic functional units (multiply, divide, add, subtract, square root, compare, integer to float, float to integer) were generated using the floating-point wizard of Xilinx CORE Generator [11]. A conservative clock frequency of 100 MHz was initially chosen as a proof

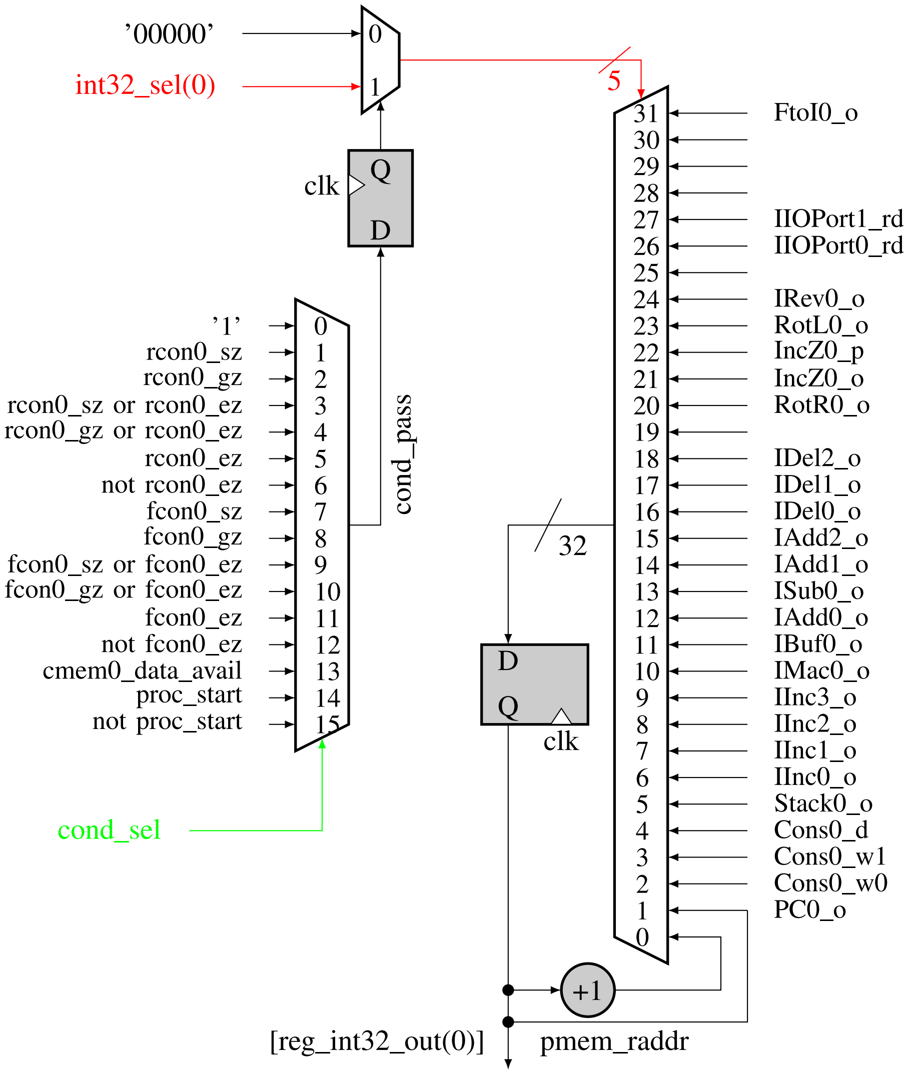
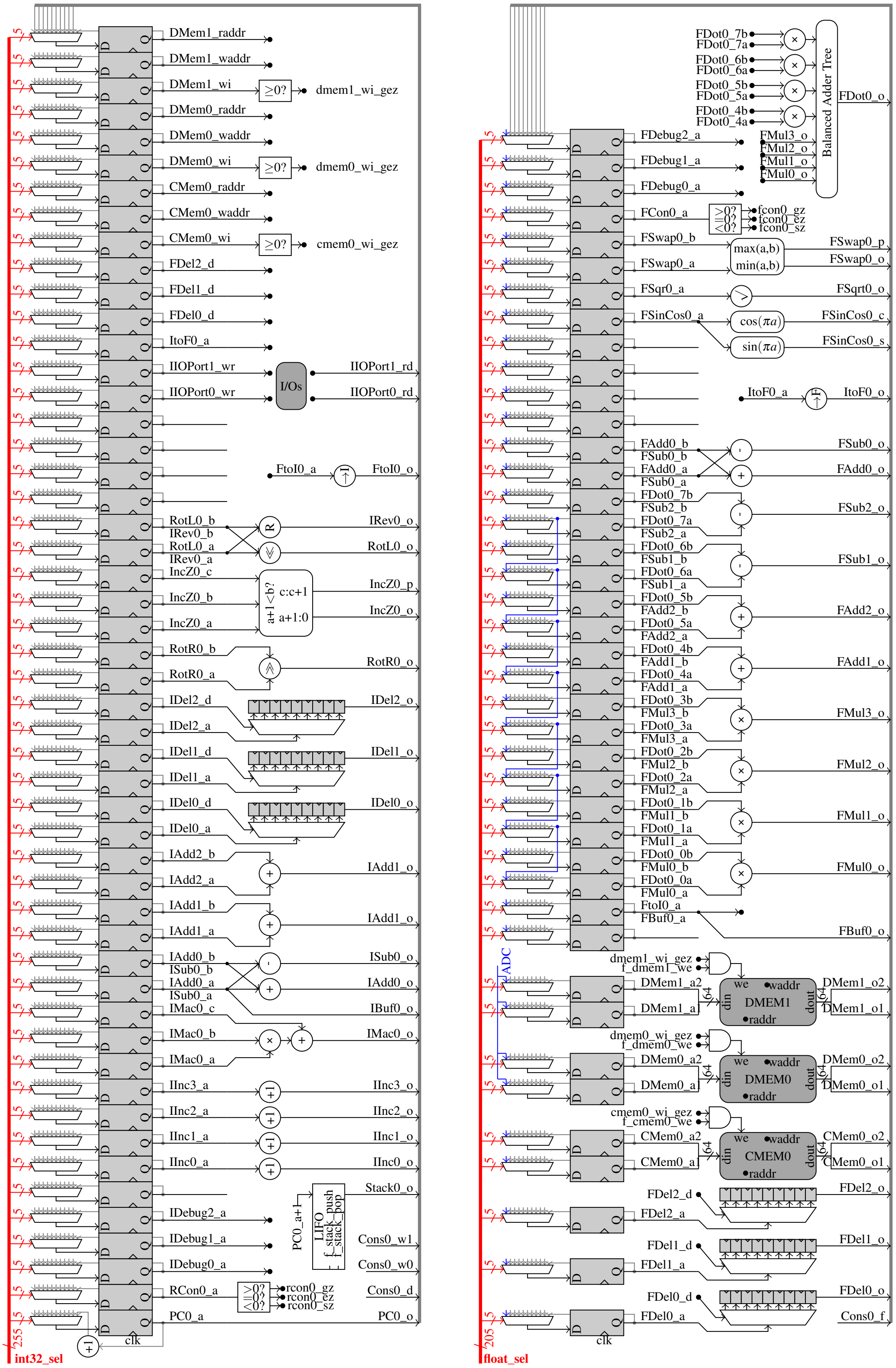


Figure 6.3: Program Counter Architecture

of concept to avoid timing closure problems. As such the amount of pipelining in the functional units was kept to a minimum as shown in Table 6.3.

Each 32-to-1 multiplexer uses 10 look up tables on the Virtex-5 architecture, resulting in a path delay of 2.011 ns (0.641 ns logic, 1.370 ns route, 2 levels of logic). A total of 320 LUTs is thus required for each 32-bit register multiplexer pair. The coefficient and program memory was implemented using internal block RAM. The main data memory requires a minimum of 2 memory slots of size $N \times P$ complex samples (for operations that cannot be executed in place; e.g. matrix transpose) plus additional coefficient storage (pulse compression spectrum, twiddle factors FFT, twiddle factors IFFT, twiddle factors Doppler, Doppler window, transmit pulse). The absolute minimum memory space is



(a) Processor Architecture: Integer Side **(b)** Processor Architecture: Floating-Point Side

Figure 6.4: Final Processor Architecture

Table 6.3: Functional unit latency

FU Name	Latency Cycles	Max Path Delay
IMac	1	7.136 ns
ItoF	2	6.939 ns
DMem	5	(external QDRII)
FtoI	2	7.693 ns
FMul	2	6.210 ns
FDiv	10	7.465 ns
FAdd	2	7.652 ns
FSub	2	7.652 ns
FSinCos	7	7.517 ns
FSqr	10	7.174 ns

thus

$$MEM_{min} = 2PN+N+N/2+N/2+P/2+P+T \quad \text{complex float} \tag{6.1}$$

$$= 16(P+2)N+8P+8T \quad \text{bytes,} \tag{6.2}$$

more than 16.5 MB for the test-case in Fig. 2.7. In this implementation a stack depth of 16 and a program memory depth of 1024 was chosen, which proved to be more than sufficient for the tested radar signal processors. Fig. 6.5 shows the top level of the firmware instantiating the soft-processor core.

Peripherals are memory-address mapped into the CMem functional unit and coupled via FIFO buffers with a flag indicating received data availability. Additionally a “start” flag is provided for synchronisation purposes between multiple cores. These flags can be used as conditionals in the soft-core to poll for certain conditions.

A JTAG interface via the standard Xilinx USB programmer allows soft-core BIT file programming and access to the internal BSCAN interface. This BSCAN debugging interface can be used for programming the program memory, preloading the external data memory (QDRII+ memory chip) and accessing the internal ChipScope logic analyser trace (dbg_regs) of the *IDebug* and *FDebug* functional units for verification purposes.

The ADC interface de-serialises and de-couples the high-speed differential inputs (400 MSPS, 10-bit) into a parallel 4-word wide single precision floating-point (128-bits) stream that is connected to the soft-core processor. Similarly, the DAC interface converts and then serialises the floating-point data coming from the soft-core processor into the 12-bit wide fixed point stream used by the DAC IC. For

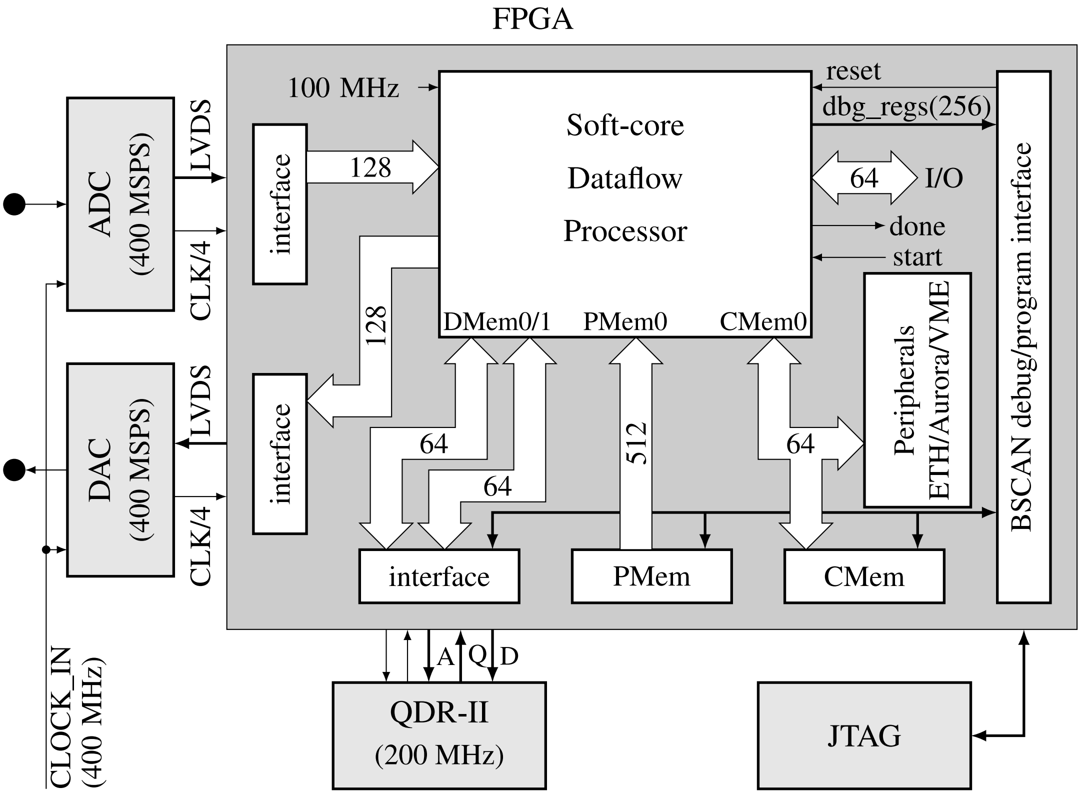


Figure 6.5: Hardware and Firmware Interface: Top Level

ADC and DAC sampling rates higher than 400 MSPS, dual port memories with sample-enable and playback-enable flags can be mapped into the CMem space.

CHAPTER 7

VERIFICATION AND QUANTIFICATION

7.1 OVERVIEW

This chapter presents the performance results of both the signal processing operations as well as the radar algorithms on the proposed architecture. Although there are many verification metrics, the primary focus is algorithmic performance. Other metrics of the core architecture include latency, resource usage (area or LUT), power consumption, ease of programmability and composite metrics based on ratios of these metrics.

For performance comparison purposes, the architectures of the Texas Instruments C66x core and Intel's AVX SIMD vector extensions (e.g. in the 2nd and 3rd generation Intel Core i7 processors) are used. The comparison is based on the number of clock cycles required for each algorithm or operation, rather than the processing time. Since the two competing architectures are implemented in custom CMOS silicon processes, the achievable clock rates are in the gigahertz ranges and a comparison based on processing time would favour these architectures. An implementation of the proposed architecture in a full custom ASIC is expected to achieve comparable if not higher clock frequencies than the competing architectures. The clock cycle based comparison can thus be seen as a normalised performance metric (which can also be used as a measure of architectural efficiency), while the processing time represents the actual performance.

As a final step, a simple radar signal processor is implemented directly from the high-level development environment to demonstrate the processing capability of this architecture. The implementation is evaluated in terms of functionality and performance characterised against an identical implementation on the Texas Instruments C66x architecture, from both a clock cycle as well as processing time perspective.

7.2 SIGNAL PROCESSING PERFORMANCE RESULTS

Table 7.1 summarises the required number of clock cycles for some selected signal processing operations. Note that only a few signal processing operations were implemented as a proof of concept of the proposed architecture; more elaborate signal processing libraries will be written at a later stage.

Table 7.1: Architectural clock cycles (N=No. of samples, P=No. of pulses, L=Filter length)

Algorithm	**Clock Cycles**
FIR Filter	$5+ceil(\frac{L}{8})[N+23]-8[\frac{ceil(\frac{L}{8})-1}{2}]ceil(\frac{L}{8})$
Inplace Fast Fourier Transform (FFT)	$(\frac{PN}{2}+21)\log_2(N)+8$
Clear DMEM	$\frac{N}{2}+5$
Copy DMEM to DMEM/CMEM	$N+10$
Transpose in DMEM	$N+11$
Bit reversed addressing in DMEM	$N+11$
Elementwise operations in DMEM	$N+17$
8-point Moving Average in DMEM	$N+28$

Provided that there are enough functional units to do the required operation in a stream, only a single iteration over the data values is required. For most operations, the cycle count is thus simply N with a few extra clock cycles for the memory read and functional unit latencies. With the FIR filter operation, multiple iterations are required if the filter length exceeds the length of the *FDot* functional unit (8 in this implementation). Each additional iteration is offset by 8 data values from the previous iteration to align the data stream, requiring 8 less clock cycles with each consecutive iteration cycle. The FFT operation has an outer loop latency of 21 clock cycles and an 8 clock cycle setup requirement. The outer loop latency can be hidden by performing the same stage of multiple FFT operations back-to-back before proceeding to the next stage.

Note that these results are based on callable functions (with arguments for point sizes, input and output addresses, and number of samples) in a crude **.FLOW* assembly implementation on the proposed architecture. Further optimisations are possible when the iterations are fixed (based on #defines or macros) or optimised for specific point sizes. For example, the outer loop latency of a single 8-point FFT (using the above function) exceeds the calculation latency of the butterfly operation by a factor of more than 5. Embedding the address generation code into the program flow would greatly reduce the datapath idle time between stages at the expense of code flexibility, program memory size and reuse for other point sizes.

7.2.1 Comparison to other Architectures

Fig. 7.1, Fig. 7.2, and Fig. 7.3 graphically compare the FFT clock cycle performance of the Texas instruments C66x core and the Intel AVX instruction set extensions with the proposed architecture. All calculations are performed on single precision floating-point values. Note that the Texas Instruments implementation is based on a Radix-4 algorithm, while the implementation on the proposed architecture is a Radix-2 FFT algorithm.

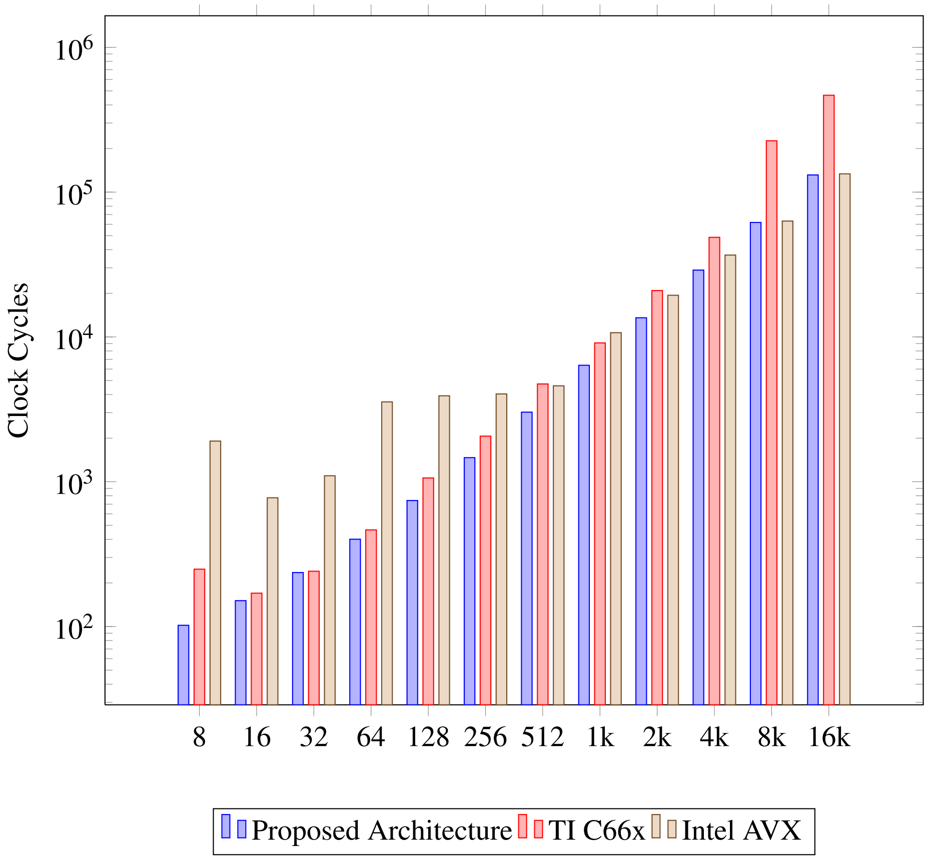


Figure 7.1: FFT 1D clock cycle comparison N=8 to 16384

The required clock cycles for performing a single FFT on each of the three architectures is shown in Fig. 7.1 above. Note that these results were obtained by averaging the clock cycle results of over 40 repeated runs. The proposed architecture required the identical number of clock cycles for each repeated run, while the C66x and the AVX clock cycle count varied significantly. The relative standard deviation of the AVX results was as much as 240 percent for some point sizes, while settling to about

10 percent for the larger point sizes. Texas Instruments' C66x architecture was a bit more consistent, and achieved a relative standard deviation of less than 6 percent over the entire range. The initial performance results from the Texas Instruments C66x core were based on simulator profiling of the dsplib *DSPF_sp_fftSPxSP* function call, however actual measured results (with all data in L2RAM) required about 30% more clock cycles.

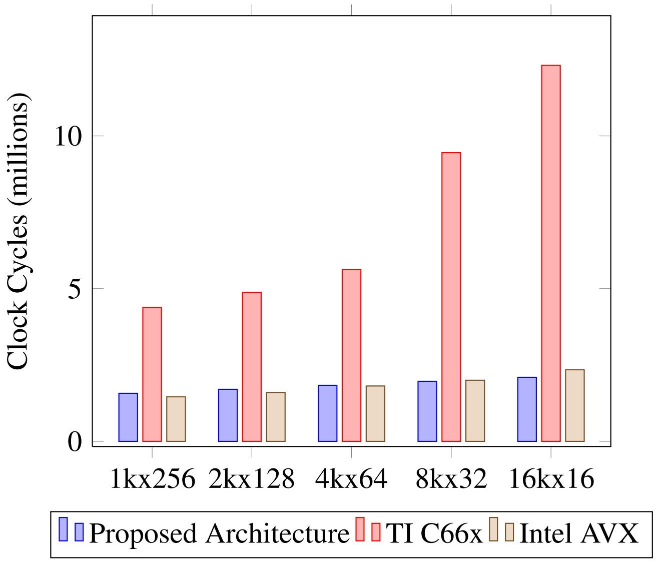


Figure 7.2: FFT 2D clock cycle comparison N=1024×256 to 16384×16

The best case results (rather than the averaged results) of the AVX technology performed slightly better than the proposed architecture for point sizes between 512 and 2048, achieving very similar results for larger point sizes. The wide SIMD registers of the AVX architecture work well for the FFT operation, where multiple elements fit into a single register; operations can thus occur in the registers without having to re-fetch data from main memory. As the point size increases, the wide registers do not favour data locality any more, and memory has to be fetched from DDR. For larger or multidimensional FFTs, the performance of the AVX extensions thus start degrading, as the algorithms become more memory bound.

Regardless, the AVX methodology works quite well even in the two-dimensional case. The 2-dimensional performance of the 3 architectures is shown in Fig. 7.2 for sizes that are common in radar signal processing. The proposed architecture achieves very similar clock cycle results to the AVX extensions, even though the Intel architecture features a much wider data memory bus, 8-wide SIMD operations and exceeds the number of arithmetic primitives of the proposed architecture by several factors. In comparison to Texas Instruments' architecture, the proposed architecture does the same calculation in about a third of the clock cycles of the C66x implementation. Note that the data-set does not fit into the local L2RAM of the C66x architecture and is executed from DDR memory instead.

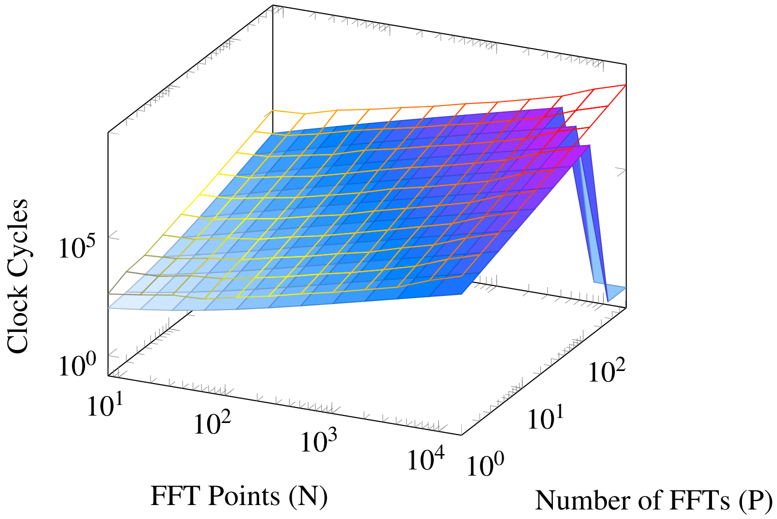


Figure 7.3: 2D FFT clock cycles; surface plot is the proposed architecture, mesh plot is the C66x architecture

Fig. 7.3 depicts the performance difference between the proposed architecture and the C66x core architecture for point sizes ranging from 8 to 16384 and the number of FFTs ranging between 1 and 512. The last 3 data-sets of the proposed architecture (for 8192-pt × 512, 16384-pt × 512 and 16384-pt × 256) did not fit into the off-chip memory on the development board, and were thus not computed. The proposed architecture performs bit-reversed addressing after the in-place FFT library function. This bit reversal could be incorporated into the last stage of the library function, further reducing the number of clock cycles by N for any point size. Regardless, the proposed architecture performed more than 3 times faster than the C66x core, and identically to the AVX implementation for the dimensions applicable to radar signal processing.

The FIR filter performance is compared in Fig. 7.4 and Fig. 7.5 for filter lengths of 8 and 32 respectively. The proposed architecture performs faster than the competing architectures in both cases, with a clock cycle performance gain of more than 7 for the filter length of 8.

The complex vector multiplication is an indication of the performance of element-wise operations in the datapath. Fig. 7.6 compares the clock cycle performance difference between the proposed architecture and the AVX implementation.

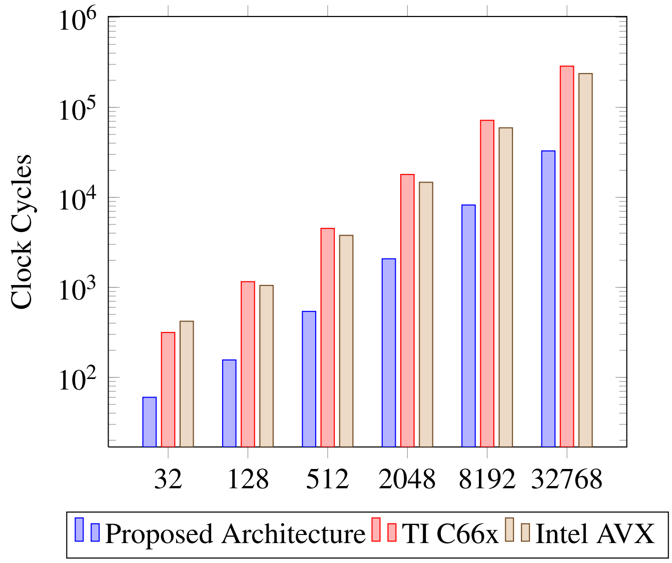


Figure 7.4: FIR filter clock cycle comparison (L=8), N=32 to 32768

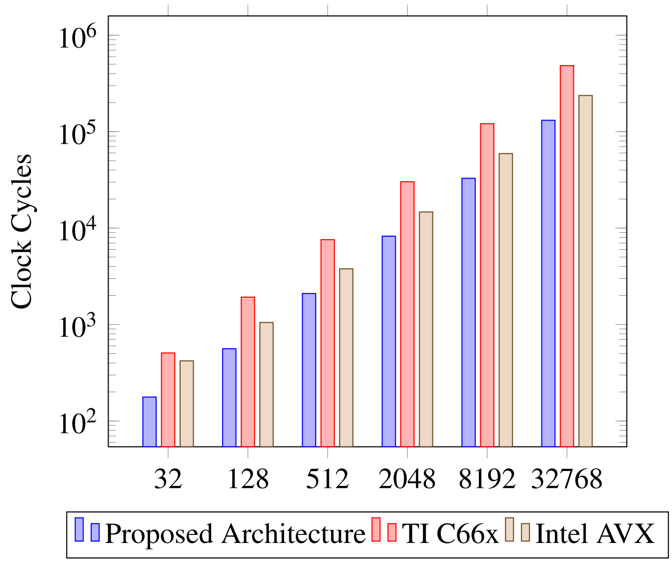


Figure 7.5: FIR filter clock cycle comparison (L=32), N=32 to 32768

The proposed architecture achieves a clock cycle performance improvement of at least 20 percent for small sizes of 4096 and less. For larger or two-dimensional sizes, the performance difference between the proposed architecture and the AVX implementation is further improved. The vector multiplication operation is mostly memory bound, as both vectors need to be read (2× 64-bit) and the result written (1× 64-bit) simultaneously. When a vector by matrix multiplication is required, the vector can be read from the CMem functional unit, while two data streams are processed in parallel directly from main memory.

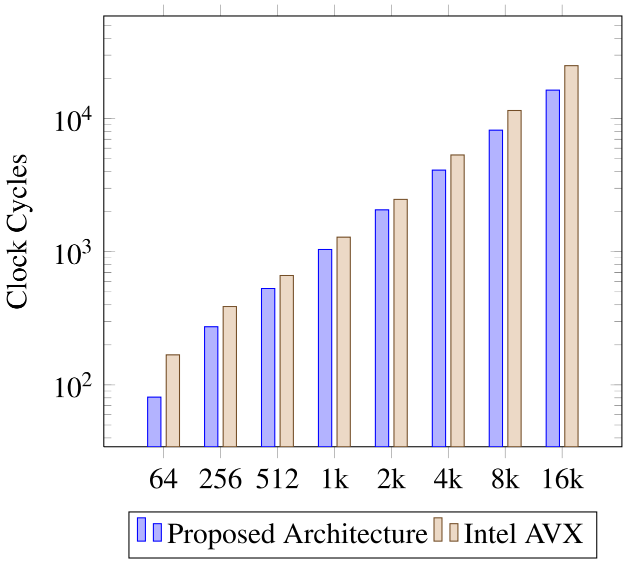


Figure 7.6: Vector complex multiply clock cycle comparison N=64 to 16384

7.3 RADAR ALGORITHM PERFORMANCE RESULTS

A test-case similar to that shown in Fig. 2.7 is used to validate and profile the radar implementation. Processing time is measured for a single burst, consisting of a range-Doppler map of 32 × 8192 complex samples. The flow of processing subcomponents is shown in Fig. 7.7.

Each of these subcomponents have been implemented and practically validated on the Xilinx Virtex 5 FPGA platform with an ADC, DAC, and QDR interface as shown in Fig. 6.5. The shaded operations in Fig. 7.7 are combined into a single pipelined operation that only loops over the range-Doppler map ($N \times P$ iterations) once, rather than for each individual sub-operation. This is possible since there are sufficient processing functional units in the datapath. Table 7.2 shows the measured results in terms of number of clock cycles and respective processing time for a single burst.

The processing time as shown is based on the reference clock frequency of 100 MHz. Based on the path delay analysis in Section 6.4, much higher clock rates are achievable, especially if deeper pipelined functional units are used in the datapath. Deeply pipelined functional units slightly increase the required number of clock cycles because of the increased latency, but allow substantially higher clock frequencies.

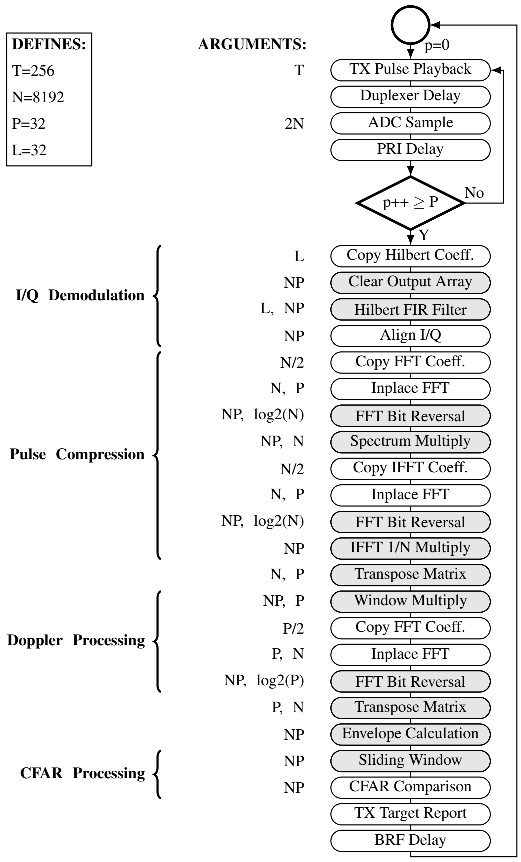


Figure 7.7: RSP test-case flow chart

7.3.1 Comparison to other Architectures

The identical radar signal processing chain (as shown in Fig. 7.7) is implemented on the Texas Instruments C66x core by using the optimised signal processing library functions from the Texas Instrument's *DSPLIB*, and compared against the proposed architecture. An implementation based on Intel's AVX instruction set was not realised as no direct high-speed connection to an ADC and DAC was available, and this peripheral interface is expected to be the major bottleneck for a practical radar system.

Fig. 7.8 compares the total number of clock cycles required for the radar signal processor of the proposed architecture (surface plot) with the C66x architecture (mesh plot). The last 5 data-points of the proposed architecture were not computable due to limited external memory on the development board. There is an almost constant offset between the C66x results and the proposed architecture results on the log scale, a difference of more than an order in magnitude. The C66x implementation requires between 10.8 and 20.9 times the number of clock cycles compared to the equivalent imple-

Table 7.2: Radar performance

Algorithm	Clock Cycles	Time (ms)
32x DAC Playback	2240	0.02
32x Deadtime Delay	0	variable
32x ADC Sample	131328	1.31
32x PRI Delay	0	variable
FIR Hilbert Filter	1050144	10.50
Memory Re-align	262175	2.62
Load FFT Coeff	4106	0.04
32x FFT8192	1704217	17.04
Bitrev+Spectrum Multiply	262167	2.62
Load IFFT Coeff	4106	0.04
32x IFFT8192	1704217	17.04
Bitrev+Windowing+Transpose	262159	2.62
Load FFT32 Coeff	26	0.00
8192x FFT32	655473	6.55
Bitrev+Transp.+Envelope+Moving ave	262189	2.62
CFAR processing	262173	2.62
Send target report to data processor	262154	2.62
TOTAL	6828874	68.29

mentation on the proposed architecture for typical radar operating parameters. Based purely on clock cycles, the proposed architecture outperforms the C66x implementation by an average factor of 13.9 over the parameter ranges in Fig. 7.8.

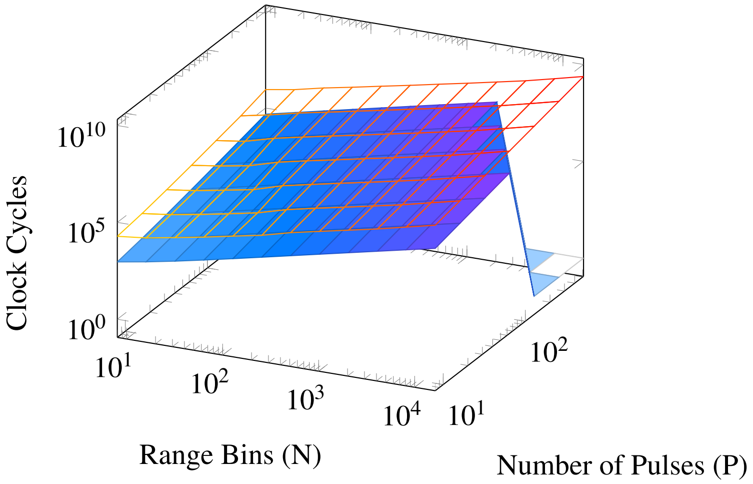


Figure 7.8: RSP clock cycles as a function of range and number of pulses

A similar variation was also implemented on the fixed point Texas instruments C64x+ core. The single-core processing time performance of this implementation is compared to the proposed architecture and the C66x in Fig. 7.9 for a single burst. Note how this comparison favours the PC and DSPs, as they are running at 2.1 GHz, 625 MHz and 1200 MHz respectively, with all calculations on the C64x+ in fixed point at either 16 or 32 bit depending on the processing stage. The ADC sampling process, I/Q demodulation and the data processor interface are excluded from this comparison, as they are handled by other processing interfaces in the C64x+ based radar system.

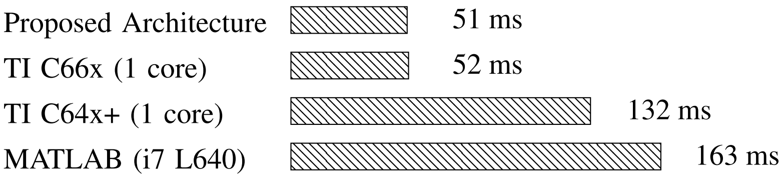


Figure 7.9: Processing-time comparison between TI DSPs, MATLAB and the proposed architecture

7.4 FPGA RESOURCES USED

Table 7.3 summarises the resource usage of the various core elements. Some of the elements of the proposed architecture are not shown explicitly, as they were absorbed into other modules by the synthesizer tool. The combined LUT usage of the multiplexers is 22118 (some multiplexers on the integer side are less than 32 bit wide and thus use less than 320 LUTs), about 50 % of the total LUT usage of the proposed architecture. The remaining resources are dedicated mostly to address generation and data processing functional units, achieving a high ratio of processing resources to control-flow overheads. The Xilinx tools reported a power consumption of 5.23 Watts, which is comparable to the 6.5 Watts of single-cored high performance DSP processors. The power consumption is not expected to increase with clock frequency, as the ASIC processes are more power efficient.

7.5 DESIGN TIME AND EASE OF IMPLEMENTATION

For many high performance systems, the actual algorithmic performance is not the only metric in choosing the processing architecture. The design time, ease of implementation, support and software development environment also play a vital role in the decision making process.

The primary reason for the proposed soft-core processor is to speed up the development time of the current HDL-based FPGA design flows. As the development tools for the proposed architecture are still in their infancy, they cannot be compared against the matured software IDE suits of Intel and Texas Instruments. Regardless, the low-level assignment-based *FLOW* language is easy to understand

Table 7.3: FPGA resource usage

Component	LUT usage	BRAM usage	DSP48E usage
CMem memory	142	57	0
DMem interface	25	0	0
PMem memory	0	15	0
ADC interface	269	0	0
DAC interface	600	0	0
FAdd0/1/2	252	0	2
FSub0/1/2	252	0	2
FCon	39	0	0
FDel0/1/2	991	0	0
FDot0	2356	0	38
FMul0/1/2/3	74	0	3
FSinCos0	413	2	5
FSqr0	432	0	0
FSwap0	108	0	0
FtoI0	182	0	0
ItoF0	169	0	0
IAdd0/1/2	32	0	0
ISub0	32	0	0
IDel0/1/2	991	0	0
IInc0/1/2/3	1	0	0
IMac0	32	0	3
IncZ0	82	0	0
RotR0	26	0	0
Stack0	13	0	0
MUX (x 81)	320	0	0
TOTAL	41783	74	70

and maps directly to the underlying hardware architecture. Optimising an algorithmic kernel on the proposed architecture is straightforward, as the architecture is simple to understand and its limitations as well as advantages are transparent. In most cases, implementing a processing system only requires calling a few signal processing library functions with arguments such as number of iterations, point or filter lengths and memory addresses. This makes the design flow very similar to the commercial processing approaches, with function calls to optimised signal processing operations.

The commercial software tools make the first stage of algorithm development quick and relatively effortless, as the design flow is based on a high-level programming language such as C and performance libraries or intrinsics. Optimising for performance on these architectures however, requires an in depth understanding of the complex underlying architecture, a task that is not necessarily trivial.

CHAPTER 8

CONCLUSION

8.1 OVERVIEW

The proposed architecture was designed to bridge the gap between high performance radar signal processors and soft-core architectures, allowing for faster practical evaluation of new radar algorithms. From the extensive literature covering soft-core processors, there has been very limited material on custom soft-core processors for streaming signal processing applications, especially in the field of radar signal processing.

The first step of the architectural design process was to analyse the various radar signal processing algorithms. The algorithms were broken down into mathematical kernels and ranked according to their relative processing requirements. Operations exhibiting high computational requirements were then selected for optimisation purposes and architectural selection.

The next step involved an examination of current processing architectures and technologies as well as their usage in radar signal processing applications. Architectural traits that were well matched to the prior computational requirements were selected and a new architectural template was proposed.

The various radar algorithms were implemented and then performance profiled on this architectural template. With each implementation and profiling stage, the architectural template is incrementally refined into the final architecture. A software development environment was designed to simplify and facilitate this iterative process, generating the development tools as well as HDL-based description of the processor core from an architectural description file.

For practical verification, the generated core was synthesised on a Xilinx Virtex-5 FPGA. The custom development board includes a high-speed ADC and DAC, making the implementation a complete radar signal processor system. The proposed architecture was compared against COTS processing alternatives, and surpassed their performances over a wide range of typical operating parameters.

The primary contributions made by this work consist of the architectural framework with the relevant optimisation process and guidelines, the final architecture, the software tool chain and *FLOW* programming representation, a survey of radar processing systems, and a computational requirement analysis of typical radar algorithms. The answers to the original research questions of Section 1.3 are summarised in Table 8.1 below.

Table 8.1: Answers to original research questions

Research Question	Short Answer
What does radar signal processing entail from a processing perspective?	Streaming, fixed processing chains consisting of mostly FFT, IFFT, FIR and sorting operations along both vertical and horizontal dimensions in the data matrix.
What performance enhancement techniques do current processors employ?	Vectorisation and SIMD, superscalar, VLIW, instruction set extensions, dynamic instruction scheduling, micro-coding, rotating register files, speculation buffers, hardware looping mechanisms, register renaming, pipelining, out-of-order execution, caching, branch prediction, co-processors, multiple cores
Which characteristics make an architecture optimal for a specific application?	Streaming support, burst processing support, memory hierarchies, interface mechanisms, deterministic and real-time responses, interrupt support, dynamic instruction scheduling support, high-level compiler support, architectural transparency, inter-process communication mechanisms, memory management units, hardware abstraction, power consumption, physical size, dynamic range, performance scalability, operating system support, customer support
What procedures are followed to design, optimise, debug and verify an architecture?	Iterative design, functional / performance evaluation and architectural model refinement based on a development environment and designer intuition
What is a well-suited architecture for a radar signal processor?	Low latency, predictability and transparency, determinism, deeply pipelined functional units (no pipeline stalls), explicit cache / scratch-pad control, small uncached instruction memory, no dynamic hardware scheduling techniques, full control over low-level computational resources

This chapter critically discusses the advantages and shortcomings of the proposed architecture, and provides suggestions on future research topics and optimisation alternatives.

8.2 RESULT DISCUSSION

The advantages and shortcomings of the proposed architecture are considered from various perspectives; namely performance, system interface, latency, support for general purpose computing, architectural efficiency, assembly optimisation, and performance scalability.

8.2.1 Performance

The performance results of the proposed architecture compared to the COTS processing alternatives show an interesting trend. In the case of the one-dimensional clock cycle performance of the individual operations, the proposed architecture only slightly outperforms the competing architectures. In the two dimensional case and when the entire radar signal processing chain is profiled, the proposed architecture performs several orders of magnitude better as shown in Fig. 7.8 and below in Fig. 8.1.

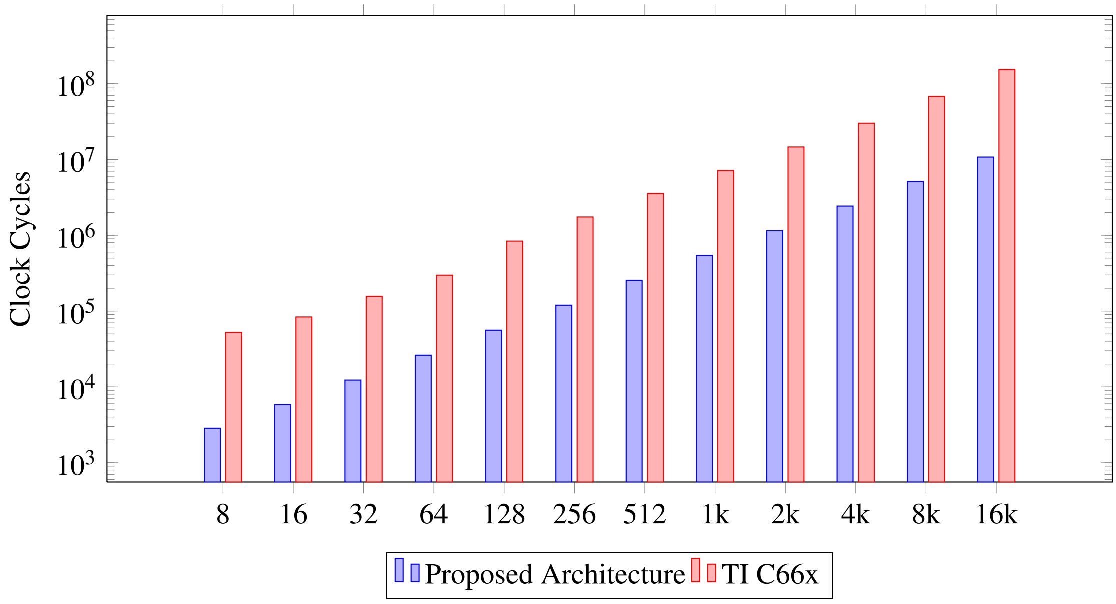


Figure 8.1: RSP clock cycle comparison, 32 pulses per burst

This comparison, based purely on the required number of clock cycles for processing a radar burst, shows a performance improvement to the C66x core by an average factor of 14 over the range of typical operating parameters. This suggests an exceptional architectural match to the core signal processing needs of the radar signal processor. Note that this comparison is based on single core performance of both architectures. A comparison based on processing time favours the competing architecture (clock frequency of 1.2 GHz versus 100 MHz), yet the proposed architecture still achieves a marginally better performance compared to the C66x core. Implementing the proposed architecture on a custom ASIC is expected to yield substantially higher clock rates, surpassing the computational performance of the competing architectures by several factors.

8.2.2 System Interface

DSP systems, by definition, are designed to process analogue signals in the digital domain. As such they conceptually consist of an ADC, a DSP processor, and a DAC. The DSP processor thus runs continuously, processing the input stream according to some algorithm and passing the output data to the analogue interface. Surprisingly, none of today's DSPs even include interfaces to high-speed ADC and DAC (multi-GSPS, parallel LVDS lines) chips, forcing designers to use FPGAs, ASICs or other dedicated front end ICs as interfaces between the analogue devices and the DSP (or PC). Since these FPGAs or ASICs are required either ways, the proposed architecture greatly simplifies the system design, providing a complete system-on-chip solution based on a single FPGA or ASIC. External system interfaces are easily mapped into the architectural multiplexers, providing a low latency and high-speed streaming mechanism directly into the core registers.

8.2.3 Latency

Additionally, modern DSPs look a lot more like CPUs rather than dedicated streaming signal processors. The architecture of these DSPs is primarily optimised for batch or packet processing in communication infrastructure applications (such as 3G-LTE, 3GPP/3GPP2, TD-SCDMA, and WiMAX). Due to the large market driving these developments, they feature many general purpose processing mechanisms for hardware scheduling, branch prediction, speculative execution, multi-level caches, virtual memory, shared memory and switching fabric controllers, memory management units (MMU) and support for operating systems. As such their performance and response times are no longer deterministic, with varying number of clock cycles between different re-runs of the same application, making them unsuitable for applications that require strict timing control (real-time processing). The proposed architecture does not feature any of these general purpose processing mechanisms, and is thus much better suited for streaming and timing critical signal processing applications (such as radar signal processing), providing fixed and deterministic processing latencies.

8.2.4 General Purpose Computing

The lack of these general purpose processing mechanisms in the proposed architecture comes at the expense of limited performance for control-flow orientated or multi-threaded applications. Applications with an unpredictable control-flow feature suboptimal performance when scheduled statically. Another disadvantage of the proposed architecture is limited code-interruptibility and backward-compatibility. Interrupting the software pipeline during an inner loop calculation would require a context backup of all core registers as well as the various partial results inside the pipelined functional units. As such a context switching mechanism would both complicate as well as limit the maximum performance of the proposed architecture. Interrupts should thus be disabled during inner loops, or, for the case of high priority interrupts, the last few inner loop iterations (based on the number of inner loop latency cycles) would have to be rerun after the interrupt service routine returns. Since most streaming applications (such as the radar signal processing operations) are highly regular and data-independent, the lack of interrupts is not necessarily a limiting factor of the proposed architecture.

8.2.5 Architectural Efficiency

Commercial CPUs and DSPs feature a multitude of functional units ranging from mathematical to arithmetic, logical and SIMD functions. However, the control-flow restrictions of their respective out-of-order and VLIW architectures allow a maximum of 6 or 8 instructions to be issued every clock cycle, while the remaining functional units sit idle. Additionally, the datapath has various routing restrictions between functional units, further restricting the number of simultaneous execution units that can be used in a constructive way. The proposed architecture features less functional units than the traditional architectures, but exhibits full control over each low-level execution unit on a cycle-by-cycle basis. Of the 6828874 required clock cycles for processing a burst in Table 7.2, 99.96% are used for actual processing; the remaining 0.04% make up the architectural overhead and consist of functional unit latencies, function calls, loop setups and other control-flow related overheads. The proposed architecture thus exhibits very little architectural overhead and is able to fully utilise the ALU and other on-chip resources, making the core architecture extremely efficient.

8.2.6 Optimisation

A lacking feature of the proposed architecture is a supporting high-level compiler. The competing architectures are driven by large markets, ensuring consistent growth on both the hardware and software sides. As such they have mature development environments with optimised compilers, development support and debugging tool chains. Even though their architectures are hard to understand and require a significant amount of insight for optimisation purposes, the software tools and performance libraries achieve acceptable results and manual optimisation is not typically required. Creating an algorithmic kernel on the proposed architecture requires thinking on a streaming dataflow level rather than a sequential or control-flow paradigm. The majority of algorithms are implemented with software pipeline mechanisms, avoiding the need for an explicit loop prologue or epilogue. The *FLOW* assignment based language maps directly to the functional units on the proposed architecture, making it easy to understand and optimise based on the graphical development tools. The simplicity of the proposed architecture makes trade-offs between various performance optimisations and implementations transparent.

8.2.7 Scaling in Performance

Modern digital signal processors are easily up-scaled in the number of processing cores; various development boards feature numerous interconnected multi-cored DSPs. Although the interconnection mechanisms (or the shared memory controllers) on these architectures usually pose bottlenecks for distributed computing, their limitations and applications are well characterised. The proposed architecture is similarly scalable in number of cores, but can be scaled on a functional unit level as well. Additional functional units are easily added provided sufficient silicon area or LUTs are available, and the application is not memory-bound. Pipelining multiple cores for streaming or distributed computing applications could be achieved with interconnections based on FIFO-coupled input and output ports, which are mapped into the core architecture as functional units. Such mechanisms are deterministic in terms of throughput and latency, simplifying inter-processor communications and system level design.

8.3 SIMILAR ARCHITECTURES

Additional research revealed two processing architectures that feature some resemblance to the proposed architecture. The first of these architectures is the Multi-logic-unit processor [103–105]. As the name suggests, the architecture features multiple arithmetic logic units. These architectures use program word slices to directly control various ALU units simultaneously. Similar to VLIW architectures, they rely on a shared register file, which creates a bottleneck in the datapath at the crossbar switching network. Also, only a single functional unit inside of each ALU can be used simultaneously, limiting the maximum ILP to the number of ALUs.

Another architecture-class, which is closely related to the proposed architecture, is the transport triggered architecture (TTA) [106–108]. A transport triggered architecture can be seen as a one instruction set computer (OISC); the only available instruction is to move data from a source register address to a destination register address. Register files, memory access units as well as functional modules all have a fixed number of input and output ports with queues and special trigger-ports for initiating operations. As data is moved between registers, the processing inside the functional module happens as a side effect of the move operation. Multiple transport buses can be provided, exposing instruction-level parallelism similar to VLIW architectures. The exposed datapath of TTA architectures provides finer grained control over functional units compared to VLIW architectures though, allowing direct data movement between functional modules and bypassing the register file completely. The TTA architecture is used in the commercially available MAXQ processor family from Maxim Integrated [109]. These processors feature a single transport bus, and as such each instruction can move one data word from a module output port (16-to-1 multiplexer) to a module input port (1-to-16 multiplexer) in a clock cycle. Each instruction also specifies both the input and output register addresses internal to the selected module. TTA architectures can also be used as templates for C-to-HDL compilers or other application-specific instruction-set processors (ASIP) [110].

A simplification of the TTA architecture is the synchronous transfer architecture (STA) [111, 112]. The STA architecture removes the register-file, trigger-ports and queues from the critical path of the TTA architecture, using synchronous communication between arithmetic logic modules (somewhat resembling [113]). The assembly instruction thus contains both transfers and operations that supply control signals to explicitly trigger control functional units.

The proposed architecture could thus be seen as a further optimisation of the STA architecture. In the proposed architecture, the instruction word is only used to specify the routings for each register (from a specific functional unit output port to a register), and not for instruction control to modules via opcodes. The proposed architecture thus allows even finer grained control and can use every functional unit simultaneously, rather than using multiple modules simultaneously. Additionally, the proposed architecture provides various architectural optimisations for loop control and streaming applications as explained in Chapters 4 and 6.

8.4 ALTERNATIVES

The proposed soft-core architecture is not the only processing solution that is programmable from a high level and well-suited for radar signal processing applications. Various other alternatives based on completely different architectures are possible as discussed in this section.

8.4.1 High-level FPGA Synthesis Tools

Developers usually design and describe mathematical algorithms in a sequential matter in high-level languages such as MATLAB or C. A limitation of this approach is that the inherent serial execution denies parallelism. Various architectural techniques can extract instruction-level parallelism from the sequential instruction stream, but these approaches are only scalable to a limited extent. Although these high-level synthesis tools are constantly evolving [8,9,114–116], the abstraction of time through high-level programming languages limits design trade-offs and optimisation prospects. Additionally, the sequential program descriptions rely on a flat memory structure, which inherently denies locality. Cache memories provide an illusion of both temporal as well as spatial memory locality, but these approaches become very inefficient for streaming approaches or when the working set does not fit into the cache.

To overcome these limitations in the streaming signal processing domain, various FPGA synthesis tools enable developers to work on higher levels of abstraction and functionality [7,117]. These tools attempt to address the needs of developers, allowing them to describe systems through an abstracted graphical interface that exposes the inherent parallelism in the algorithms. Although these developments are still in their infancy and various issues still need to be overcome, multiple frameworks for algorithm development and implementation have been demonstrated.

8.4.2 Grid Processor

Another architecture that is well-suited for algorithms exhibiting high levels of temporal locality, is the grid-processor. Various programmable processing units are arranged in a grid, with memory outputs or ADC ports connected on top of the grid and memory input or DAC ports connected at the bottom of the grid. This arrangement works well for streaming applications where accesses to the same data values tend to be clustered in time, and are not reused later.

The proposed architecture could be used as such a grid-processor, when the data processing (floating point) side is duplicated numerous times. The memory access functional units of each grid-processor would then be replaced with interface ports to the adjacent processors. When a clock-by-clock reconfiguration of the multiplexer connections is not required, the PC and instruction word could be completely removed and replaced with latches that are initialised on power-up.

8.4.3 GPU-based Processor

Another alternative processing platform for radar signal processing is the GPU. Modern GPU architectures feature large arrays of configurable processors that are well-suited for streaming and data-independent processing. Provided the external interfaces to the ADC and DAC can sustain the required data throughput and low latency requirements, the GPU makes a capable radar signal processor.

8.5 FUTURE RESEARCH

This section provides some ideas on how to further optimise the proposed architecture for performance and resource utilisation. Future research includes a custom ASIC implementation of the proposed architecture as well as adding compiler support for a high-level programming language.

8.5.1 Further Performance Optimisations

Although the proposed architecture was optimised for performance of radar signal processing applications, the architecture could easily be extended to support more algorithms from the radar data-processor (or any other signal processing application) by adding the required functional units.

One way to increase data-parallelism and thus also the performance of the proposed core architecture is by adding vectorisation support. In multi-channel radar signal processors, the same operations are applied to each channel simultaneously. By extending the registers and functional units on the data processing side of the proposed architecture to vectors and SIMD operations respectively, such a multi-channel system is easily realised (e.g. phased array or mono-pulse radars).

Another method of increasing performance is by pipelining. When throughput is more important than atomic latency, each functional unit in the datapath can be further pipelined, reducing critical path lengths and in turn increasing the maximum clock frequency. However, data dependent or single value calculations take multiple clock cycles to complete with such deeply pipelined arithmetic or functional units.

The proposed architecture is also well-suited for adding coprocessors directly into the critical datapath based on a custom functional unit. Coprocessors such as complex FIR filters or FFT processors can thus be mapped directly into the core architecture to increase throughput.

For many streaming signal processing applications, multi-core implementations are a simple way of increasing system performance. A multi-cored version of the proposed architecture is achieved by adding special functional units into the datapath. Depending on the required multi-core arrangement, different inter-process communication mechanisms are possible.

When the system is used in a pipelined fashion, where the processed data is passed along the processing chain to the next core, simple FIFO-coupled input and output ports provide communication between adjacent cores.

For distributed computing applications, an entirely different mechanism, which allows communication between all cores, is required. Such a mechanism could be a simple functional unit that provides bus arbitration to a shared memory resource, or a more advanced inter-process communication mechanisms based on hardware semaphores and mutexes.

8.5.2 Resource Usage Reduction

The resources used by the proposed architecture could be further reduced using several different methodologies, each with their own advantages and disadvantages, as described in this section.

One way to reduce the routing congestion and thus the resource utilisation, is by removing the global reset signal going into the proposed architecture. Rather than an explicit reset signal going to all registers and functional units, the program word at address zero could be set in such a way that the zero constant is assigned to each register in the architecture. This would save LUT resources on the FPGA, as the reset signal would only have to be routed to the PC register. The reset signal should be asserted for a minimum number of clock cycles determined by the maximum latency of the internal functional units. This method would also allow non-zero reset values, for example the *NaN* could be assigned to all floating-point registers.

The debug registers and multiplexers use a substantial amount of chip resources that do not constructively add to the processing performance. An alternative mechanism of saving a debugging trace could be implemented. For example, a single bit in the program word could indicate that all registers are to be saved into a dual ported RAM for the next X instruction cycles before the processor halts. The saved values could then be read out to the debugging environment at any later stage via a dedicated debugging peripheral interface (e.g. via USB or Ethernet).

The program word of the proposed architecture is extremely wide, yet most program word slices are only fully utilised in the inner loops. During setup and control-flow instructions, only a small portion of the program word is used. One mechanism to reduce the program memory bandwidth, is to compress the instruction word. Another mechanism would be to store the long instruction words in a separate dedicated memory, and have the shorter control-flow instruction word point to an address in this dedicated memory when needed.

To save on-chip resources without degrading the performance of the proposed architecture, the multiplexer connections could be further reduced. In most applications, the multiplexer select signals only take a few discrete values with the majority of connections left unused. Based on the desired end application, these multiplexer sizes could be reduced at the expense of flexibility with reduced inter-functional-unit connectivity.

Also, many functional units are seldom used and never used concurrently in the inner loops. Such functional units could be mapped into the memory space or grouped into secondary functional unit clusters with common input ports. The output ports of each cluster could then be passed to a secondary multiplexer with a registered output port connection to the main multiplexers. Only a single functional unit in a cluster can thus be used concurrently. This reduces the width of the main architectural multiplexers and saves on-chip resources at the expense of limited ILP and an added latency cycle on the secondary/clustered functional units.

Another method of reducing the resource usage and possibly increasing performance, is segmenting the functional units into localised groups based on functionality. Each functional unit in the localised group would have direct multiplexer connections to every element in the same group, but connections to other groups would be of secondary concern. This method thus resembles the integer and floating-point functional unit split in the proposed architecture, with a much finer grained split in the datapath.

8.5.3 ASIC Implementation

A substantial amount of effort is required to implement the proposed architecture in a custom ASIC. The proposed architecture is a prototype demonstrating its characteristics and performance, still requiring many system level design choices and trade-offs before taking the design to a final product. Regardless, an ASIC implementation of the proposed architecture is expected to achieve considerable performance, power and area-usage improvements over the FPGA implementation. Such an implementation would overcome many issues inherent to FPGAs: routing congestions, datapath latencies, maximum clock frequencies, and the FPGA technology-gate overhead.

8.5.4 Compiler Support

Although performance libraries based on manually-optimised DSP functions have been implemented, adding high-level compiler support to the proposed architecture greatly simplifies the development of new algorithms. Possible high-level language support could include MATLAB and C/C++. However, as the serial execution of these languages typically denies parallelism, alternative programming languages could be better suited. For example, functional programming languages such as Haskell might be better matched to extracting parallelism than traditional sequential programming languages.

Regardless of the programming language, the proposed architecture is well-suited for compiler-based static scheduling because of deterministic and fixed latencies in the core architecture, interconnects and functional units. Many developers find it more difficult to write parallel programs compared to sequential programs, however experimental results show that compilers can evolve parallel programs with less computational effort than the equivalent sequential programs [103]. A significant amount of research has been done on compilers for both TTA as well as STA architectures [110, 118–122]. Of particular value to the proposed architecture would be the LLVM-based TCE toolset [123], which could be used as a basis for designing a compiler for the proposed architecture.

8.6 CONCLUDING REMARKS

The main focus of this research was to design a processing architecture that is optimal for radar signal processing applications. Constructs of both sequential processors and dataflow machines were merged into a tightly coupled solution, capable of fully exploiting each of the underlying processing resources concurrently. This novel soft-core processing architecture features an excellent match to the core computational requirements of the RSP. The proposed architecture outperforms competing architectures in both number of clock cycles and processing time, despite the limited clock frequencies achievable in the FPGA technology. Table 8.2 summarises the characteristics of this architecture.

Table 8.2: Proposed architecture - Characteristics Summary

Property	**Characteristics**
Number of Cores	1
Clock Frequency	100 MHz (ASIC estimated: 1.0 - 2.7 GHz)
Streaming Performance	High (data independent processing)
Burst Processing Performance	Medium to High (software pipelines)
General Purpose Performance	Low to Medium (no hardware scheduling)
Interrupt Support	Limited (impractical context backup)
Latency	Low (deterministic and real-time)
Interface Capabilities	Excellent (direct streaming interfaces to peripherals and external systems)
Architectural Efficiency	Excellent (extremely low overhead, high ALU utilisation)
Performance Scalability	Excellent (add/remove functional units)
Power Consumption	Low to Average
Code Compatibility	None (not backwards compatible between implementations)
Code Density	High (horizontal and vertical)
Compile Times	Low (less than a second)
Resource Usage	Average (medium to high on an FPGA)
Ease of Use	Good (software based: function calls to optimised DSP routines, lacking high-level compiler)

The software-based development environment enables quick algorithmic changes and instant compile times during field tests, greatly improving the ease of use compared to the FPGA design flow and compilation times. An ASIC implementation of the proposed architecture would be extremely well-suited for integration into the transmit/receive (TR-)modules of AESA and MIMO radar systems, featuring low power consumption, high processing throughputs and instant front-end processing mode changes for various operational requirements (e.g. communications, radar, electronic warfare techniques and jamming modes).

REFERENCES

[1] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proc. Comput. Digit. Tech.*, vol. 152, no. 2, pp. 193–207, Mar. 2005.

[2] P. Yiannacouras, J. Steffan, and J. Rose, "Data Parallel FPGA Workloads: Software versus Hardware," in *IEEE Int. Conf. Field-Program. Logic Appl.*, Aug. 2009, pp. 51–58.

[3] J. Greco, G. Cieslewski, A. Jacobs, I. Troxel, and A. George, "Hardware/software Interface for High-performance Space Computing with FPGA Coprocessors," in *IEEE Aerospace Conf.*, Jan. 2006, pp. 1–10.

[4] Y. He, C. Le, J. Zheng, K. Nguyen, and D. Bekker, "ISAAC - A Case of Highly-Reusable, Highly-Capable Computing and Control Platform for Radar Applications," in *IEEE Radar Conf.*, May 2009, pp. 1–4.

[5] J. McAllister, R. Woods, S. Fischaber, and E. Malins, "Rapid implementation and optimisation of DSP systems on FPGA-centric heterogeneous platforms," *J. Syst. Architect.*, vol. 53, no. 8, pp. 511–523, 2007.

[6] O. Cret, K. Pusztai, C. Vancea, and B. Szente, "CREC: A Novel Reconfigurable Computing Design Methodology," in *Int. Proc. Parallel Distr. Processing*, Apr. 2003, pp. 8–16.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[8] Berkeley Design Technology, Inc. (2012, May) BDTI Certified Results for the AutoESL AutoPilot High-Level Synthesis Tool. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot

[9] ——. (2012, May) BDTI Certified Results for the Synopsys Synphony C Compiler. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/Synphony

[10] (2012, Nov.) Opencores. [Online]. Available: http://opencores.org/

[11] Xilinx. (2012, Nov.) Xilinx CORE Generator System. [Online]. Available: http://www.xilinx.com/tools/coregen.htm

[12] Altera Corporation. (2013, Aug.) MegaWizard Plug-Ins. [Online]. Available: http://www.altera.com/products/ip/altera/megawizd.html

[13] ——. (2012, Nov.) OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity. [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html

[14] D. Singh. (2011, Mar.) Higher Level Programming Abstractions for FPGAs using OpenCL. Altera Corporation. [Online]. Available: http://cas.ee.ic.ac.uk/people/gac1/DATE2011/Singh.pdf

[15] (2012, Nov.) Open RVC-CAL Compiler. [Online]. Available: http://orcc.sourceforge.net/

[16] R. Broich and H. Grobler, "Analysis of the Computational Requirements of a Pulse-Doppler Radar Signal Processor," in *IEEE Radar Conf.*, May 2012, pp. 835–840.

[17] Mathworks. (2010, Nov.) MATLAB - The Language Of Technical Computing. [Online]. Available: http://www.mathworks.com/products/matlab/

[18] M. A. Richards, *Fundamentals of Radar Signal Processing*, 1st ed. New York: McGraw-Hill, 2005.

[19] B. Kienhuis, E. Deprettere, K. Vissers, and P. Van Der Wolf, "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," in *IEEE Int. Conf. Appl.-Specific Systems, Architectures & Processors*, Jul. 1997, pp. 338–349.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2002.

[21] A. Blass and Y. Gurevich, "Abstract state machines capture parallel algorithms," *ACM Trans. Comput. Logic*, vol. 4, no. 4, pp. 578–651, Oct. 2003.

[22] M. A. Richards, *Principles of Modern Radar: Basic Principles*, 1st ed. North Carolina: SciTech Publishing, 2010.

[23] Berkeley Design Technology, Inc. (2011, Feb.) Speed Scores for Fixed-Point Packaged Processors. [Online]. Available: http://www.bdti.com/MyBDTI/bdtimark/chip_fixed_scores.pdf

[24] ——. (2011, Aug.) BDTI DSP Kernel Benchmarks (BDTImark2000). [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/BDTIMark2000

[25] *TMS320C66x CPU and Instruction Set Reference Guide (SPRUGH7)*, Texas Instruments, Nov. 2010.

[26] *StarCore DSP SC3850 Core Reference Manual (SC3850CRM Rev. C)*, Freescale, Jul. 2009.

[27] *MSC8156 SC3850 DSP Subsystem Reference Manual (SC3850SUBRM Rev. E)*, Freescale, Feb. 2010.

[28] J. Alter, J. Evins, J. Davis, and D. Rooney, "A Programmable Radar Signal Processor Architecture," in *IEEE Nat. Radar Conf.*, Mar. 1991, pp. 108–111.

[29] Y. Rin, B. Sie, and L. Yongtan, "An Advanced Digital Signal Processor for the HRR Polarimetric MMW Active Guidance Radar," in *IEEE Nat. Proc. Aerospace & Electronics Conf.*, vol. 1, May 1993, pp. 370–376.

[30] Y. Lan, Y. Zhaoming, J. Jing, Z. Delin, and T. Changwen, "A High-Speed Multi-Channel Data Acquisition and Processing System for Coherent Radar," in *Int. Conf. Signal Processing Proc.*, vol. 2, 1998, pp. 1632–1635.

[31] F. Wang, T. Long, and M. Gao, "A digital signal processor for high range resolution tracking radar," in *Int. Conf. Signal Processing*, vol. 2, Aug. 2002, pp. 1441–1444.

[32] S.-Q. Hu and T. Long, "Design and Realization of High-performance Universal Radar Signal Processing System," in *Int. Conf. Signal Processing*, Oct. 2008, pp. 2254–2257.

[33] D. Anuradha, P. Barua, A. Singhal, and R. Rathore, "Programmable Radar Signal Processor For a Multi Function Radar," in *IEEE Radar Conf.*, May 2009, pp. 1–5.

[34] F. Wen, W. Zhu, and B. Chen, "Design of Universal Radar Signal Processor Architecture Based on Crosspoint Switch," in *Int. Conf. on Multimedia Technology (ICMT)*, Oct. 2010, pp. 1–4.

[35] D. Wang and M. Ali, "Synthetic Aperture Radar on Low Power Multi-Core Digital Signal Processor," in *IEEE Conf. High Performance Extreme Computing (HPEC)*, Sep. 2012, pp. 1 –6.

[36] *Virtex-5 FPGA User Guide (UG190 v5.3)*, Xilinx, May 2010.

[37] *Virtex-5 FPGA XtremeDSP Design Considerations User Guide (UG193 v3.4)*, Xilinx, Jun. 2010.

[38] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," *Proc. IEEE*, vol. 81, no. 7, pp. 1013–1029, Jul. 1993.

[39] *FPGA Architecture - White Paper (WP-01003-1.0)*, Altera Corporation, Jul. 2006.

[40] E. E. Swartzlander and H. H. Saleh, "FFT Implementation with Fused Floating-Point Operations," *IEEE Trans. Computers*, vol. 61, no. 2, pp. 284–288, Feb. 2012.

[41] *Achieving One TeraFLOPS with 28nm FPGAs - White Paper (WP-01142-1.0)*, Altera Corporation, Sep. 2010.

[42] R. Lazarus and F. Meyer, "Realization of a Dynamically Reconfigurable Preprocessor," in *IEEE Nat. Proc. Aerospace & Electronics Conf.*, May 1993, pp. 74–80.

[43] T. Tuan, M. Figueroa, F. Lind, C. Zhou, C. Diorio, and J. Sahr, "An FPGA-Based Array Processor for an Ionospheric-Imaging Radar," in *IEEE Symp. Field-Programmable Custom Computing Machines*, 2000, pp. 313–314.

[44] S. Sumeem, M. Mobien, and M. Siddiqi, "A Pulse Doppler Radar using Reconfigurable Computing," in *Int. Multi Topic Conf.*, Dec. 2003, pp. 213–217.

[45] Y. Shi, X. Gao, Z. Tan, and Y. Wang, “Research of Radar Signal Processor Based on the Software Defined Radio,” in *Int. Conf. Wireless Comm., Networking & Mobile Computing*, Sep. 2007, pp. 1252–1255.

[46] H. Nicolaisen, T. Holmboe, K. Hoel, and S. Kristoffersen, “High Resolution Range-Doppler Radar Demonstrator Based on a Commercially Available FPGA Card,” in *Int. Conf. Radar*, Sep. 2008, pp. 676–681.

[47] S. Xu, M. Li, and J. Suo, “Clutter Processing for Digital Radars Based on FPGA and DSP,” in *Int. Conf. Wireless Comm. Signal Processing*, Nov. 2009, pp. 1–4.

[48] B. Liu, W. Chang, and X. Li, “Design and Implemetation of a Monopulse Radar Signal Processor,” in *Int. Conf. Mixed Design of Integrated Circuits & Systems (MIXDES)*, May 2012, pp. 484–488.

[49] R. Stapleton, K. Merranko, C. Parris, and J. Alter, “The Use of Field Programmable Gate Arrays in High Performance Radar Signal Processing Applications,” in *IEEE Int. Radar Conf.*, 2000, pp. 850–855.

[50] V. Winkler, J. Detlefsen, U. Siart, J. Buchler, and M. Wagner, “FPGA-based Signal Processing of an Automotive Radar Sensor,” in *European Radar Conf.*, 2004, pp. 245–248.

[51] T. Darwich, “Pulse-modulated Radar Display Processor on a Chip,” in *Int. Parallel & Distributed Processing Symp.*, Apr. 2004, pp. 128–132.

[52] S. Lal, R. Muscedere, and S. Chowdhury, “An FPGA-Based Signal Processing System for a 77 GHz MEMS Tri-Mode Automotive Radar,” in *IEEE Int. Symp. Rapid System Prototyping*, May 2011, pp. 2–8.

[53] S. Bayar and A. Yurdakul, “Self-Reconfiguration on Spartan-III FPGAs with Compressed Partial Bitstreams via a Parallel Configuration Access Port (cPCAP) Core,” in *Research in Microelectronics & Electronics*, Apr. 2008, pp. 137–140.

[54] N. Harb, S. Niar, M. Saghir, Y. Hillali, and R. Atitallah, “Dynamically Reconfigurable Architecture for a Driver Assistant System,” in *IEEE Symp. Application Specific Processors*, Jun. 2011, pp. 62–65.

[55] E. Seguin, R. Tessier, E. Knapp, and R. Jackson, "A Dynamically-Reconfigurable Phased Array Radar Processing System," in *IEEE Int. Conf. Field Programmable Logic & Applications (FPL)*, Sep. 2011, pp. 258–263.

[56] Xilinx. (2013, Mar.) Zynq-7000 All Programmable SoC. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm

[57] Altera Corporation. (2013, Mar.) Intel Atom Processor E6x5C Series. [Online]. Available: http://www.altera.com/devices/processor/intel/e6xx/proc-e6x5c.html

[58] P. Long, C. He, and L. Yuedong, "The SoPC Based Design for Real-Time Radar Seeker Signal Processing," in *IET Int. Radar Conf.*, Apr. 2009, pp. 1–4.

[59] Altera Corporation. (2011, Apr.) Nios II Processor: The World's Most Versatile Embedded Processor. [Online]. Available: http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[60] Xilinx. (2010, Sep.) MicroBlaze Soft Processor Core. [Online]. Available: http://www.xilinx.com/tools/microblaze.htm

[61] S. Le Beux, V. Gagne, E. Aboulhamid, P. Marquet, and J.-L. Dekeyser, "Hardware/Software Exploration for an Anti-collision Radar System," in *IEEE Int. Midwest Symp. Circuits & Systems*, vol. 1, Aug. 2006, pp. 385–389.

[62] J. Saad, A. Baghdadi, and F. Bodereau, "FPGA-based Radar Signal Processing for Automotive Driver Assistance System," in *IEEE/IFIP Int. Symp. Rapid System Prototyping*, Jun. 2009, pp. 196–199.

[63] J. Khan, S. Niar, A. Rivenq, and Y. El-Hillali, "Radar based collision avoidance system implementation in a reconfigurable MPSoC," in *Int. Conf. Intelligent Transport Systems Telecomm.*, Oct. 2009, pp. 586–591.

[64] B. Mingming, L. Feng, and X. Yizhuang, "Dynamic Reconfigurable Storage and Pretreatment System of SAR Signal Processing using Nios II Architecture," in *IET Int. Radar Conf.*, Apr. 2009, pp. 1–4.

[65] R. Djemal, “A Real-time FPGA-Based Implementation of Target Detection Technique in Non Homogenous Environment,” in *Int. Conf. Design Technology Integr. Systems*, Mar. 2010, pp. 1–6.

[66] P. Jena, C. V, B. Tripathi, R. Kuloor, and W. Nasir, “Design and Implementation of a Highly Configurable Low Power Robust Signal Processor for Portable Ground Based Multiple Scan Rate Surveillance Radar,” in *Int. Radar Symp.*, Jun. 2010, pp. 1–4.

[67] J. R. C. Kingyens, “A GPU-Inspired Soft Processor for High-Throughput Acceleration,” Master’s thesis, University of Toronto, 2008.

[68] J. Kingyens and J. Steffan, “A GPU-Inspired Soft Processor for High-Throughput Acceleration,” in *IEEE Int. Symp. Parallel Distributed Processing*, Apr. 2010, pp. 1–8.

[69] P. Yiannacouras, J. Steffan, and J. Rose, “Portable, Flexible, and Scalable Soft Vector Processors,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 8, pp. 1429–1442, Aug. 2008.

[70] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, “Vector Processing as a Soft Processor Accelerator,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 12:1–12:34, Jun. 2009.

[71] M. Purnaprajna and P. Ienne, “Making wide-issue VLIW processors viable on FPGAs,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 33:1–33:16, Jan. 2012.

[72] A. Jones, R. Hoare, I. Kourtev, J. Fazekas, D. Kusic, J. Foster, S. Boddie, and A. Muaydh, “A 64-way VLIW/SIMD FPGA architecture and design flow,” in *IEEE Int. Conf. Electronics, Circuits & Systems*, Dec. 2004, pp. 499–502.

[73] C. Choo, J. Chung, J. Fong, and S. E. Cheung, “Implementation of Texas Instruments TMS32010 DSP Processor on Altera FPGA,” in *Global Signal Processing Expo & Conf.* San Jose State University, Sep. 2004.

[74] L. Noury, S. Dupuis, and N. Fel, “A Reference Low-Complexity Structured ASIC,” in *IEEE Int. Sym. on Circuits & Systems*, May 2012, pp. 2709–2712.

[75] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[76] K. Hwang and Z. Xu, "Scalable Parallel Computers for Real-Time Signal Processing," *IEEE Signal Process. Mag.*, vol. 13, no. 4, pp. 50–66, Jul. 1996.

[77] M. Kalkuhl, P. Droste, W. Wiechert, H. Nies, O. Loffeld, and M. Lambers, "Parallel Computation of Synthetic SAR Raw Data," in *IEEE Int. Geoscience & Remote Sensing Symp.*, Jul. 2007, pp. 536–539.

[78] T. Maese, J. A. Hunziker, H. S. Owen, C. Reed, R. Bluth, and L. Wagner, "Modular Software Architecture for Tactical Weather Radar Processing," in *Int. Interactive Information & Proc. Systems Conf.*, Jan. 2009.

[79] *Signal Processing on Intel Architecture: Performance Analysis using Intel Performance Primitives*, Intel, Jan. 2011.

[80] J. Hensley, "AMD CTM overview," in *ACM SIGGRAPH courses*, 2007.

[81] C. Fallen, B. Bellamy, G. Newby, and B. Watkins, "GPU Performance Comparison for Accelerated Radar Data Processing," in *Symp. Application Accelerators in High-Performance Computing*, Jul. 2011, pp. 84–92.

[82] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, and Y. Deng, "Evaluating the Potential of Graphics Processors for High Performance Embedded Computing," in *Design, Automation Test in Europe Conf. Exhibition*, Mar. 2011, pp. 1–6.

[83] Y. Zeng, J. Xu, and D. Peng, "Radar Velocity-Measuring System Design and Computation Algorithm Based on ARM Processor," in *Int. Conf. Intelligent Control & Automation*, Jul. 2010, pp. 5352–5357.

[84] R. Khasgiwale, L. Krnan, A. Perinkulam, and R. Tessier, "Reconfigurable Data Acquisition System for Weather Radar Applications," in *Symp. Circuits & Systems*, Aug. 2005, pp. 822–825.

[85] S. Winberg, A. Mishra, and B. Raw, "Rhino Blocks Pulse-Doppler Radar Framework," in *IEEE Int. Conf. Parallel Distributed & Grid Computing*, Dec. 2012, pp. 876–881.

[86] Z. Liu, K. Dickson, and J. McCanny, "Application-specific instruction set processor for SoC implementation of modern signal processing algorithms," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 52, pp. 755–765, 2005.

[87] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded software in real-time signal processing systems: application and architecture trends," *Proc. IEEE*, vol. 85, no. 3, pp. 419–435, 1997.

[88] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, pp. 1338–1354, 2001.

[89] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman, "PICO: automatically designing custom computers," *Computer*, vol. 35, pp. 39–47, 2002.

[90] A. Hoffmann, F. Fiedler, A. Nohl, and S. Parupalli, "A methodology and tooling enabling application specific processor design," in *Int. Conf. VLSI Design*, 2005, pp. 399–404.

[91] V. Guzma, S. Bhattacharyya, P. Kellomaki, and J. Takala, "An integrated ASIP design flow for digital signal processing applications," in *Int. Symp. Appl. Sci. on Biomedical & Comm. Tech.*, 2008, pp. 1–5.

[92] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. European Design & Test Conf.*, 1995, pp. 503–507.

[93] J.-H. Yang, B.-W. Kim, S.-J. Nam, Y.-S. Kwon, D.-H. Lee, J.-Y. Lee, C.-S. Hwang, Y.-H. Lee, S.-H. Hwang, I.-C. Park, and C.-M. Kyung, "MetaCore: an application-specific programmable DSP development system," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, pp. 173–183, 2000.

[94] L. Zhang, S. Li, Z. Yin, and W. Zhao, "A Research on an ASIP Processing Element Architecture Suitable for FPGA Implementation," in *Int. Conf. Comp Science & Software Engineering*, vol. 3, Dec. 2008, pp. 441–445.

[95] R. Sproull, I. Sutherland, and C. Molnar, "The counterflow pipeline processor architecture," *IEEE Des. Test. Comput.*, vol. 11, no. 3, pp. 48–59, 1994.

[96] P. Balaji, W. Mahmoud, E. Ososanya, and K. Thangarajan, "Survey of the counterflow pipeline processor architectures," in *System Theory Symp.*, 2002, pp. 1–5.

[97] B. Childers and J. Davidson, "Custom wide counterflow pipelines for high-performance embedded applications," *IEEE Trans. Computers*, vol. 53, pp. 141–158, 2004.

[98] P. B. Endecott, "SCALP: A Superscalar Asynchronous Low-Power Processor," Ph.D. dissertation, University of Manchester, 1996.

[99] M. Smith, "How RISCy is DSP?" *IEEE Micro*, vol. 12, no. 6, pp. 10–23, Dec. 1992.

[100] L. Wanhammar, *DSP Integrated Circuits*. Academic Press, 1999.

[101] D. R. Martinez, R. A. Bond, and M. M. Vai, *High Performance Embedded Computing Handbook*. CRC Press, 2008.

[102] S. Sun and J. Zambreno, "A Floating-point Accumulator for FPGA-based High Performance Computing Applications," in *Int. Conf. Field-Prog. Technology*, Dec. 2009, pp. 493–499.

[103] S. M. Cheang, K. S. Leung, and K. H. Lee, "Genetic Parallel Programming: Design and Implementation," *Evol. Comput.*, vol. 14, no. 2, pp. 129–156, Jun. 2006.

[104] W. S. Lau, G. Li, K. H. Lee, K. S. Leung, and S. M. Cheang, "Multi-logic-unit processor: A combinational logic circuit evaluation engine for genetic parallel programming," in *European Conf. Genetic Programming*, 2005, pp. 167–177.

[105] Z. Gajda, "A Core Generator for Multi-ALU Processors Utilized in Genetic Parallel Programming," in *IEEE Design & Diagn. of Electr. Circuits & Systems*, 2006, pp. 236–238.

[106] H. Corporaal, "A different approach to high performance computing," in *Int. Conf. High-Performance Computing*, Dec. 1997, pp. 22–27.

[107] ——, *Microprocessor Architectures from VLIW to TTA*. John Wiley, 1998.

[108] Y. He, D. She, B. Mesman, and H. Corporaal, "MOVE-Pro: A Low Power and High Code Density TTA Architecture," in *Int. Conf. Embedded Computer Systems*, Jul. 2011, pp. 294 –301.

[109] *MAXQ Family Users Guide*, 6th ed., Maxim Integrated, Sep. 2008.

[110] O. Esko, P. Jaaskelainen, P. Huerta, C. de La Lama, J. Takala, and J. Martinez, "Customized Exposed Datapath Soft-Core Design Flow with Compiler Support," in *Int. Conf. Field Programmable Logic & Applications (FPL)*, Sep. 2010, pp. 217–222.

[111] G. Cichon, P. Robelly, H. Seidel, T. Limberg, G. Fettweis, and V. Chair, "SAMIRA: A SIMD-DSP architecture targeted to the Matlab source language," in *Proc. Global Signal Processing Expo & Conf.*, 2004.

[112] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Synchronous Transfer Architecture (STA)," in *Proc. Systems, Architectures, Modeling & Simulation*, Jul. 2004, pp. 126–130.

[113] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths," in *Conf. Advanced Research in VLSI*, Mar. 1999, pp. 23 –40.

[114] Xilinx. (2013, Jul.) Vivado ESL Design. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

[115] Synopsys, Inc. (2013, Jul.) Synphony C Compiler - High-Level Synthesis from C/C++ to RTL. [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx

[116] Cadence Design Systems, Inc. (2013, Jul.) C-to-Silicon Compiler. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx

[117] Mathworks. (2013, Jul.) HDL Coder. [Online]. Available: http://www.mathworks.com/products/hdl-coder/

[118] J. Hoogerbrugge and H. Corporaal. (1997, Apr.) Resource Assignment in a Compiler for Transport Triggered Architectures. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.3906

[119] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Compiler Scheduling for STA-Processors," in *Int. Conf. Parallel Computing in Electrical Engineering*, Sep. 2004, pp. 45–60.

[120] J. Guo, J. Liu, B. Mennenga, and G. Fettweis, “A Phase-Coupled Compiler Backend for a New VLIW Processor Architecture Using Two-step Register Allocation,” in *IEEE Int. Conf. Application-specific Systems, Architectures & Processors*, Jul. 2007, pp. 346–352.

[121] X. Jia and G. Fettweis, “Code Generation for a Novel STA Architecture by Using Post-Processing Backend,” in *Int. Conf. Embedded Computer Systems*, Jul. 2010, pp. 208–215.

[122] ——, “Integration of Code Optimization and Hardware Exploration for A VLIW Architecture by Using Fuzzy Control System,” in *IEEE Int. SOC Conf.*, Sep. 2011, pp. 36–41.

[123] Tampere University of Technology. (2012, Nov.) TTA-based Co-design Environment (TCE). [Online]. Available: http://tce.cs.tut.fi/

[124] M. I. Skolnik, *Introduction to Radar Systems*, 2nd ed. Mcgraw-Hill College, 1980.

[125] ——, *Radar Handbook*, 3rd ed. McGraw-Hill Professional, Jan. 2008.

[126] K. Teitelbaum, “A Flexible Processor for a Digital Adaptive Array Radar,” *IEEE Aerosp. Electron. Syst. Mag.*, vol. 6, no. 5, pp. 18–22, May 1991.

[127] R. G. Lyons, *Understanding Digital Signal Processing*, 2nd ed. Prentice Hall, Mar. 2004.

[128] T. Long and L. Ren, “HPRF pulse Doppler stepped frequency radar,” *Science in China Series F: Information Sciences*, vol. 52, pp. 883–893, 2009.

[129] C. Venter and K. AlMalki, “RATIP: Parallel Architecture Investigation (GPU),” CSIR, Tech. Rep., Mar. 2011.

[130] S. Vangal, Y. Hoskote, N. Borkar, and A. Alvandpour, “A 6.2-GFlops Floating-Point Multiply-Accumulator With Conditional Normalization,” *IEEE J. Solid-State Circuits*, vol. 41, no. 10, pp. 2314–2323, Oct. 2006.

[131] Z. Luo and M. Martonosi, “Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques,” *IEEE Trans. Computers*, vol. 49, no. 3, pp. 208–218, Mar. 2000.

[132] M. Ayinala, M. Brown, and K. Parhi, "Pipelined Parallel FFT Architectures via Folding Transformation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 6, pp. 1068 –1081, Jun. 2012.

[133] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, ser. The Art of Computer Programming. Addison-Wesley, 1997, vol. 3.

[134] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comp. Conf.*, 1968, pp. 307–314.

[135] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB Journal*, vol. 21, pp. 1–23, 2012.

[136] S. W. Smith, *Digital Signal Processing - A Practical Guide for Engineers and Scientists*. Newnes, 2003.

[137] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[138] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface*, 4th ed. Morgan Kaufmann, 2008.

[139] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, "Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor," *IEEE J. Solid-State Circuits*, vol. 41, pp. 179–196, 2006.

[140] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *SC'03*, Phoenix, Arizona, Nov. 2003.

[141] S. Rixner, "Stream processor architecture," Ph.D. dissertation, Rice University, 2001.

[142] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the Imagine Stream Architecture," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 14, Mar. 2004.

[143] R. Wester, “A dataflow architecture for beamforming operations,” Master’s thesis, University of Twente, Dec. 2010.

[144] M. Freeman, “Evaluating Dataflow and Pipelined Vector Processing Architectures for FPGA Co-processors,” in *Conf. Digital System Design: Architectures, Methods & Tools*, 2006, pp. 127–130.

APPENDIX A

RADAR SIGNAL PROCESSING ALGORITHMS

A.1 OVERVIEW

This chapter provides more details on each of the radar signal processing algorithms [18,22,124,125] of Fig. 2.6. For each algorithm, the computational requirements as well as possible implementation alternatives are considered.

A.2 TIMING GENERATOR

The timing generator is the underlying synchronisation mechanism for both the radar transmitter and the radar receiver. It ensures that the receiver is blanked during the transmission time and that the transmitter is disabled during the reception time. In a practical system, the timing generator also controls various subsections of the analogue front-end including the antenna duplexer and the various amplifiers. The STC also requires the timing information from the timing generator to avoid saturation in the receiver from close targets and to amplify far target returns (increase dynamic range).

From a processing point of view, the timing generator consists of a few software configurable timers. Radar parameters such as the PRI, CPI and receiver listening time can thus be adjusted in real time to focus on certain regions of interest or achieve better counter-jamming performance through staggered PRF or pulse burst mode techniques.

In a hardware implementation the timing generator could be a simple up-counter that is reset once it reaches the “PRI” value as shown in Fig. A.1. A comparator asserts the transmit enable signal if the counter value is less than the transmit length. Similarly the receive enable signal is high when the counter value is between the receive start and the receive stop values.

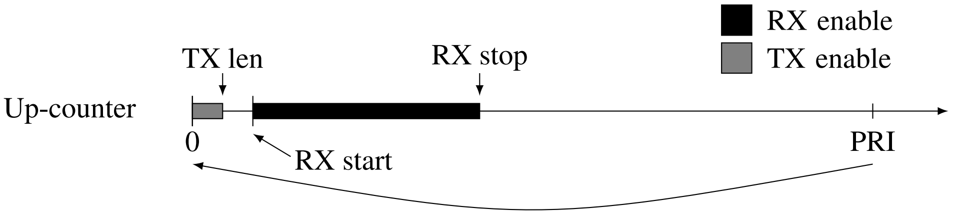


Figure A.1: PRI counter

A.3 PULSE GENERATOR

For simpler systems that do not make use of pulse compression techniques, the pulse generator simply outputs a basic pulse with a pulse width of τ. The unmodulated pulse width determines the capacity to distinguish between targets that are in close proximity to each other. The RF modulated signal of a simple pulse with $\tau = 1\ \mu s$ is shown in Fig. A.2. It should be clear that the radar can only differentiate between the two targets at 4 μs (at a range of 600 m) and 5 μs (at a range of 750 m) if they are separated by at least $R_0 \geq c\tau/2$, a resolution of 150 m in this example.

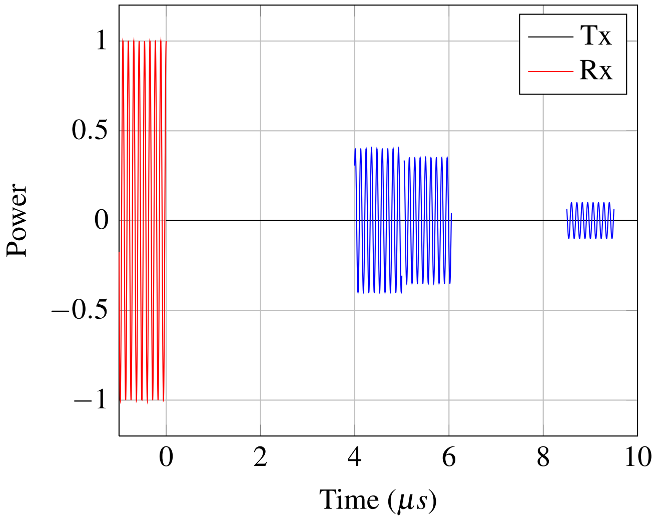


Figure A.2: Range discrimination for a pulsed radar without pulse compression

Since a long pulse width is desired in order to increase the average transmitter output power (and thus also the reflected power from a target), the transmitted pulse is either modulated in frequency or phase. A frequency modulated waveform is shown in Fig. A.3. Note how the frequency of a pulse is swept linearly across a bandwidth B over the entire pulse width τ. Alternative waveforms for pulse compressed systems make use of pseudo random phase modulations that are matched at the receiver, filtering out undesired signals that do not match the transmitted sequence.

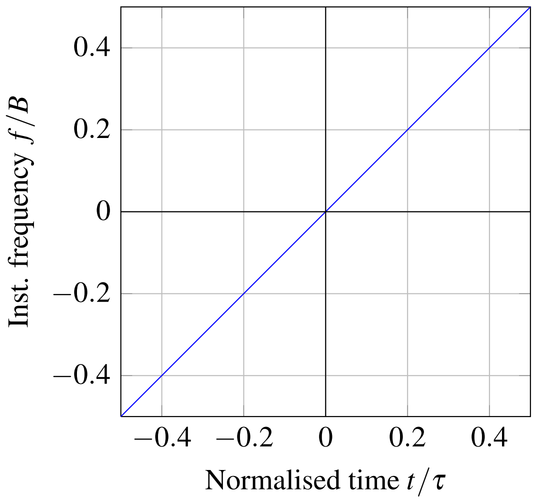


Figure A.3: Instantaneous frequency of a linear frequency modulated radar pulse

Mathematically the linear frequency modulated (LFM) signal can be written as:

$$y(t) = \cos(2\pi(f_c + B\frac{t}{\tau})t), \qquad -\frac{\tau}{2} < t < \frac{\tau}{2} \tag{A.1}$$

where f_c is the carrier frequency or zero in the baseband signal case. The baseband time domain signal for this LFM chirp waveform is shown in Fig. A.4.

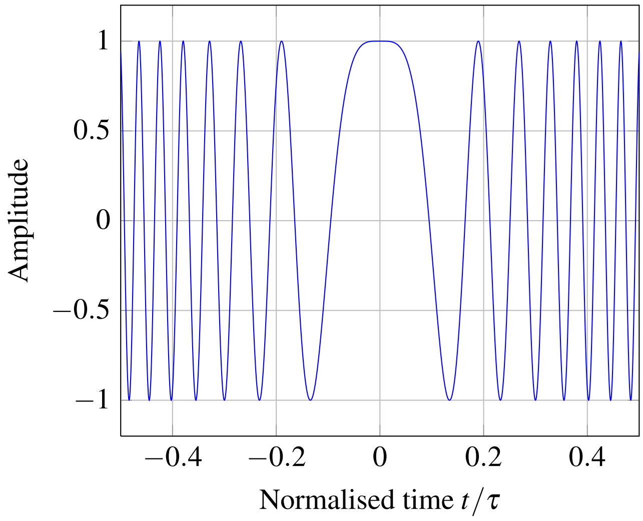


Figure A.4: Linear frequency modulated radar pulse

It can be shown that the pulse compressed waveform has a much higher range resolution than the unmodulated waveform [18]. Although the pulse width remains constant, the Rayleigh resolution after pulse compression becomes $1/B$ seconds (for time-bandwidth products greater than 10); a range resolution of $R_0 = c/(2B)$ or 3 meters for a pulse with 50 MHz bandwidth. Thus the pulse energy can be controlled through the pulse width τ while separately controlling the range resolution through the pulse bandwidth B.

Since the waveform that is transmitted has to be matched to the receiver for pulse compression, it is typically precomputed. Practically this precomputed data is often stored in lookup tables on hardware or in memory for a software implementation. Other hardware implementations have a parallel input DAC directly access the data lines of a memory chip, and thus only require a counter on the read address lines.

A.4 ANALOGUE INTERFACE

The analogue interface features both the input sampling process on the receive side as well as the output pulse playback on the transmit side as shown in the pseudo code below.

```
loop p=0 to P-1
   loop t=0 to T-1
      DAC_DATA = pulse_playback[t]

   loop n=0 to 2*N-1
      input[p][n] = ADC_DATA
```

The timing generator initiates the pulse playback as well as the sampling process at the PRF. A fast memory interface as well as a processor architecture that can sustain the data rates of the ADC and DAC is necessary. Computationally P×2×N memory writes and P×T memory reads are required.

A.5 I/Q DEMODULATION

The process of in-phase and quadrature sampling is often implemented digitally after analogue to digital conversion [125]. In many cases the input signal is centred around some intermediate frequency f_{IF}, and directly converted to the baseband real and imaginary components with two digital multiplication operations by the in-phase and 90 degree out-of-phase local oscillator. These digital mixing operations are easily implemented if the instantaneous frequency is a quarter of the ADC sampling frequency. Fig. A.5 depicts how the mixing operations reduce to simple multiplications by either 0, +1 or -1 if the ADC sampling frequency is four times the instantaneous frequency of the radar [126].

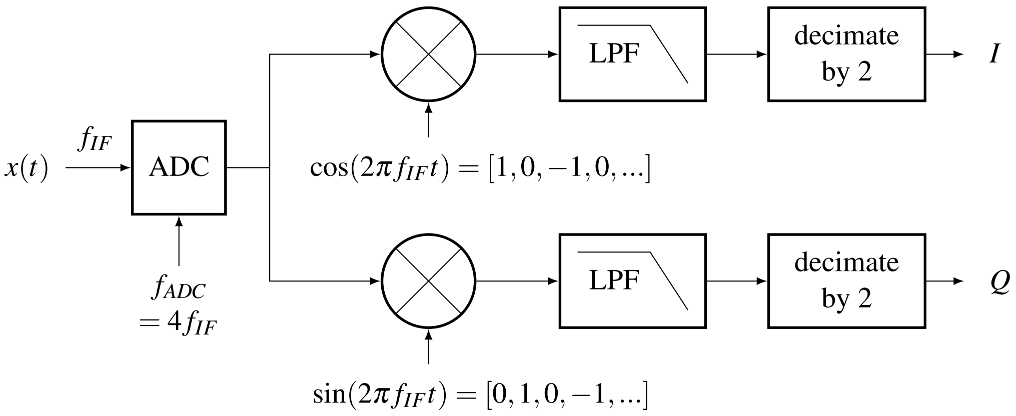

$x(t)$
f_{IF}
ADC
f_{ADC}
$= 4f_{IF}$
$\cos(2\pi f_{IF}t) = [1, 0, -1, 0, ...]$
$\sin(2\pi f_{IF}t) = [0, 1, 0, -1, ...]$
LPF
LPF
decimate by 2
decimate by 2
I
Q

Figure A.5: Digital I/Q sampling

Digital low pass filters are used to eliminate the negative frequency as well as DC components of both I and Q after the mixing operation. The filter output is then decimated by two to discard redundant data. The low pass filters (LPF) are typically designed to have a cut-off corner frequency at f_{IF}. However, since the signal is complex, the full bandwidth of $f_{ADC}/2$ is still preserved.

Alternatively, a Hilbert transform can be used to extract the Q-channel from the real sampled data [127]. The FIR approximation of a Hilbert filter is essentially a band-pass filter (BPF) that has a constant 90-degree phase shift. Fig A.6 shows how I/Q demodulation can be performed with a Hilbert filter.

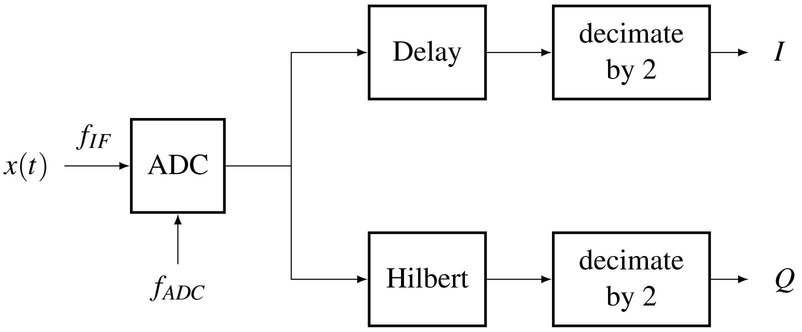


Figure A.6: Digital I/Q demodulation using a Hilbert filter

Unlike the mixing method however, the Hilbert transform method does not shift the frequency band to baseband. A high pass filter for both the I and Q channel as well as a decimation stage can however be added to bring the signal of interest down to baseband. Note how the I-channel input is delayed by the same latency the Hilbert filter introduces (for a K-tap FIR filter, group delay G = (K-1)/2 samples).

The Hilbert transform is often realised as a systolic array structure, while the filters are typically implemented as either FIR, IIR or similar classes of digital filters. Implementation details for these types of filters are discussed in Sections B.5, and B.6 respectively.

I/Q demodulation can also be performed in the frequency domain, similar to MATLAB's *hilbert* function. Since the analytic signal has a one-sided Fourier transform (i.e. no negative frequencies), the forward FFT can be used to convert the real-valued signal to the frequency domain, the negative frequencies multiplied by zeros, and an inverse FFT to get the complex time domain analytic signal. Computationally this approach is quite intensive for the typically long range lines of a radar system.

The performance requirements for I/Q demodulation strongly depend on the required resolution and acceptable filter stop-band attenuation. Since most high frequency ADCs have between 10 and 14-bit resolutions, a 16-bit filter of length 32 (order 31) is usually more than adequate. The mixing operations are simply digital multiplications by sine and cosine as shown below (where P is the number of pulses per burst, and N is the number of range bins per pulse).

```
loop i=0 to P-1
 loop j=0 to 2*N-1
   input_re[i][j] = input[i][j]*sin(2*pi*fIF*j/fadc)
   input_im[i][j] = input[i][j]*cos(2*pi*fIF*j/fadc)
```

Thus P×2×N×2 multiplications and memory writes as well as P×2×N×3 memory read operations are required provided that a lookup table is used for the trigonometric functions. The filter requirements (length L) are therefore P×N×L×2 multiplications, additions, and memory reads as well as P×N×L coefficient memory reads.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L-1
    iq_dem[i][j].RE += input_re[i][j*2-l]*lpf_coeff[l]
    iq_dem[i][j].IM += input_im[i][j*2-l]*lpf_coeff[l]
```

When the ADC sampling frequency is 4 times as high as the IF frequency, the above mixing operation simplifies to multiplications by 1,0,-1,0 and 0,1,0,-1 for the I and Q channels respectively, reducing the computational requirements to P×2×N memory reads and writes as well as P×N negations.

```
loop i=0 to P-1
 loop j=0 to N-1
```

```
   if (N=even)
     input_re[i][j] = input[i][j*2]
     input_im[i][j] = input[i][j*2+1]
   else
     input_re[i][j] = -input[i][j*2]
     input_im[i][j] = -input[i][j*2+1]
```

In this case the filtering stage can also be simplified, since only every second filter coefficient (even coefficients for I, odd coefficients for Q) is needed [126]. The computational requirements are thus reduced to P×N×L multiplications, additions, and memory reads, as well as P×N×L coefficient memory reads as shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L/2-1
    iq_dem[i][j].RE += input_re[i][j-l*2]*lpf_coef[l*2]
    iq_dem[i][j].IM += input_im[i][j-l*2-1]*lpf_coef[l*2+1]
```

```
loop i=0 to P-1
 loop j=0 to N-1
   iq_dem[i][j].RE = input[i][j*2]
   loop l=0 to L-1
     iq_dem[i][j].IM += input[i][2*j-l]*hil_coeff[l]
```

Thus P×N×(L+1) memory reads as well as P×N×L coefficient reads, multiplications and additions are required.

A.6 CHANNEL EQUALISATION

In practical systems channel equalisation is an important prerequisite for radar systems. The transfer characteristics of RF front-end components as well as ADC converters are not consistent across the frequency bandwidths of radar systems. Additionally the gains between the different mono-pulse antenna channels are not matched. The aim of the channel equalisation operation is to equalize all the incoming channels to have the same filter response, compensating for channel-to-channel distortions and performing matched filtering to the transfer characteristics of the RF components. Coefficients for this matched filter are determined by sweeping the input frequency and measuring the resulting response. These coefficients can be altered for calibration purposes at a later stage.

To compensate for such imbalances in amplitude, both the I and the Q channels are multiplied by a fractional scaling factor as shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
   amp_comp[i][j].RE = iq_dem[i][j].RE*amp_coeff
   amp_comp[i][j].IM = iq_dem[i][j].IM*amp_coeff
```

Frequency compensation typically involves a digital filter such as a complex FIR filter or two real FIR filters. The case of a complex-valued FIR filter is shown in the pseudo code below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop l=0 to L-1
    freq_comp[i][j] += amp_comp[i][j-l] * freq_coeff[l]
```

The complex multiplication in the above filter translates to 4 real multiplications and an addition as well as a subtraction. Since the input, coefficients and output is complex, 2 memory accesses need to be performed for each variable, resulting in P×N×L×4 memory reads, multiplications and additions.

A.7 PULSE COMPRESSION / MATCHED FILTERING

Pulse compression is thus a technique that allows the pulse energy to be controlled separately from the required range resolution. Typically the transmit signal is modulated to produce a chirp signal as discussed in Section A.3. On the receiving side, the incoming data (in the fast-time / range dimension) is correlated with the transmitted waveform.

Fig. A.7 shows how the output of this correlation yields a narrow compressed pulse with a main-lobe width of approximately $1/B$ when the transmit signal aligns perfectly with the receive signal. Although the pulse width remained at 1 μs as in Fig. A.2, the range resolution improved by a factor of 50 (from 150 m to 3 m).

The main-lobe width (Rayleigh resolution) does not depend on the duration of the transmitted pulse in pulse-compressed radar systems. Instead, increasing the pulse width increases the pulse energy, at the cost of a convolution filter with more taps. An improved range resolution on the other hand, only requires a wider signal bandwidth.

The pulse compression matched filtering operation can practically be implemented in several ways. The most straightforward implementation uses a digital correlation as discussed in Section B.13.

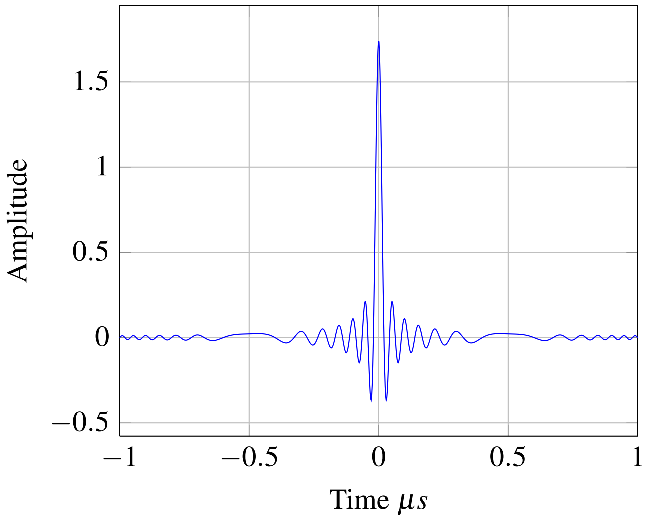


Figure A.7: Pulse Compression output of a LFM waveform with $\tau = 1$ μs and $B = 50$ MHz

When the known template signal (the transmitted chirp) is time-reversed and conjugated, the convolution method in Section B.12 can also be used. The fast convolution method (Section B.12.1) using the Fourier transform, multiplication in the frequency domain, and the inverse Fourier transforms becomes advantageous when

$$\log_2(N) < T, \tag{A.2}$$

where N is the number of range samples and T is the number of convolution taps (number of samples used for the transmitted waveform). Another possible implementation uses the FIR filter structure (Section B.5) to implement the required arithmetic involved in computing the product of filter taps against the incoming data.

The process of pulse compression (or matched filtering) involves correlating the transmitted pulse against the received signal. The most straightforward implementation of this matched filter is derived from the correlation integral and shown below.

```
loop i=0 to P-1
 loop j=0 to N-1
  loop t=0 to T-1
    pc[i][j] += freq_comp[t+j] * tx_pulse*[t]
```

Similar to the FIR filter for the frequency compensation, P×N×T×4 memory reads, multiplications and additions are required. However, since the transmit pulse length is usually longer than the frequency compensation filter lengths, the processing requirements are much more demanding.

The fast convolution method on the other hand, converts the input sequence to the frequency domain with an FFT, performs a complex multiplication by the frequency spectrum of the time-reversed complex conjugated transmitted pulse, and converts the result back to the time domain using the inverse FFT.

```
loop i=0 to P-1
  in_freq[i] = FFT(freq_comp[i])
  loop j=0 to N-1
    fil_freq[i][j] = in_freq[i][j] * FFT(tx_pulse*[-t])[j]
  pc[i] = IFFT( fil_freq[i] )
```

As discussed in Section B.4.3, the total computational requirements for an FFT are P×N/2×$\log_2$N×4 real multiplications and memory writes as well as P×N/2×$\log_2$N×6 real additions and memory reads. Bit inverted addressing is required for either memory reads or writes, depending on the selected DIT or DIF algorithm type. The multiplication stage requires P×N×4 multiplications and memory reads, as well as P×N×2 additions and memory writes, provided the inverted and conjugated spectrum of the transmitted pulse is precomputed. The IFFT processing requirements are similar to those of the FFT, except for an added stage that divides all output values by N and uses swapped real and imaginary components of the forward FFT twiddle factors [127].

A.8 CORNER TURNING MEMORY

Most of the burst processing algorithms work on the “slow-time" samples. Since the majority of the signal processing libraries (such as the FFT function) are written for row access, and the cache performance improves as a result of spatial locality in the row dimension, the entire pulse-range matrix is often transposed as shown in Fig. A.8

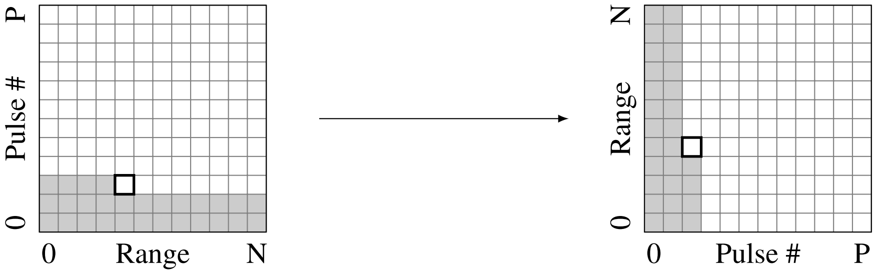


Figure A.8: Corner turning memory operation

This process also acts as a buffer. When an entire CPI of data has been collected, it is corner turned and saved in a new matrix, so that the pulse-range map can be overwritten by the new streaming samples from the next CPI.

From a hardware implementation perspective, the pre-processed streaming range-line data can be written to memory in transposed order at this stage. In software implementations, one could avoid transposing the input matrix explicitly by accessing the data in column vector format rather than row vector format. The transpose operation is however still performed to improve cache performance and make use of optimised library functions. Two separate sections of memory are then required as the transpose operation is difficult to perform in-place.

Computationally, the corner turning operation is simply a matrix transpose in memory as shown below.

```
loop i=0 to N-1
 loop j=0 to P-1
   transp[i][j] = pc[j][i]
```

If the memory architecture permits non-linear memory accesses without latency or throughput penalties, this step can be incorporated into the next processing stage. Otherwise P×N memory reads and writes are required with the appropriate address calculation circuitry (a multiply-adder).

A.9 NON-COHERENT INTEGRATION

Although not shown in 2.6, the non-coherent integration (NCI) operation is vital to simpler radars that do not measure or cannot process the returned phase. In such systems, the NCI sums the returns from numerous pulses before performing the threshold check, diminishing the effects of any noise that is centred around a mean of zero.

A.10 MOVING TARGET INDICATION

Doppler processing can be divided into two major classes: pulse-Doppler processing and moving target indication (MTI). When only target detection is of concern, the MTI filter is usually adequate. MTI filters are often low order; even a first or second order FIR high-pass filter (such as 2 or 3 pulse MTI cancellers) can be used to filter out the stationary clutter. Higher order filter types are typically not used as they only provide modest performance improvements over the pulse cancellers [22].

MTI assumes that repeated returns from stationary targets have same echo amplitude and phase. Thus subtracting successive pulses from each other should cancel out stationary targets. MTI merely provides an indication of a moving target; no information about the targets velocity is extracted. The

MTI operation is essentially a high pass filter (HPF) typically implemented as a delay and subtract operations or a FIR filter as discussed in Section B.5.

MTI filters can also be used to cancel clutter prior to Doppler processing. The clutter cancellation operation can be described by the pseudo code below. It simply subtracts the previous pulse value from the current pulse value for each sample in the range line.

```
loop i=0 to N-1
 loop j=0 to P-1
   cc[i][j] = transp[i][j] - prev
   prev = transp[i][j]
```

The processing requirements for the two pulse canceller (above pseudo code) are thus N×P×2 memory reads, subtractions and memory writes, while the 3-pulse canceller (pseudo code below) requires an additional N×P×2 multiplications and additions.

```
loop i=0 to N-1
 loop j=0 to P-1
   cc[i][j] = pc[j][i] - 2*prev1 + prev2
   prev2 = prev1
   prev1 = pc[j][i]
```

A.11 PULSE-DOPPLER PROCESSING

For a target directly approaching the radar with velocity v, R_0 in Eq. 2.8 is replaced with $R_0 - vpT_p$ (where T_p is the PRI) to give

$$\begin{aligned} y[p][n_0] &= A'e^{-j\frac{4\pi}{\lambda}(R_0 - vpT_p)} && \text{(A.3)} \\ &= A'e^{-j\frac{4\pi R_0}{\lambda}} e^{j2\pi(\frac{2v}{\lambda})pT_p}, \qquad p = 0, ..., P-1 && \text{(A.4)} \end{aligned}$$

Since the $e^{-j(4\pi R_0)/\lambda}$ component does not change with increasing time pT_p, extracting the Doppler frequency component

$$f_d = \frac{2v}{\lambda}, \tag{A.5}$$

can be accomplished with a K-point discrete Fourier transform (DFT) over all P pulse samples. The Doppler resolution then becomes f_p/K. For an X-Band radar, the wavelength is

$$\lambda = \frac{c}{f} = 0.03m \tag{A.6}$$

Thus there will be a $f_d = 66$ Hz Doppler shift for each m/s of target radial velocity. When the targets radial velocity becomes too large, it wraps in Doppler and becomes ambiguous. This happens when $f_d >= f_p$. If the direction of the Doppler shift is not known, the unambiguous velocity is halved again, placing the following constraint on the targets velocity:

$$v < \frac{\lambda f_p}{4}. \tag{A.7}$$

From a computational perspective, Doppler processing translates to an independent K-point fast-Fourier transform (FFT) over the “slow-time" pulse-to-pulse samples representing a discrete range bin as shown in Fig. A.9. K has to be a power of 2 and each “slow-time" sample vector is zero padded when K is larger than P.

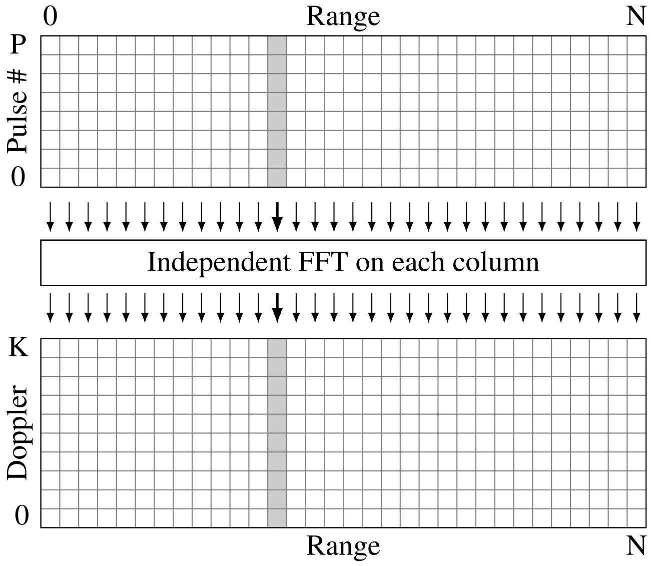


Figure A.9: Doppler processing algorithm

For HRR or stepped frequency implementations of Doppler processing [128], the memory accesses of the FFT operation become disjoint as only every N-th pulse sample is used. Library-based FFT functions assume data input in linear order, and thus require a memory realignment operation to group the input samples by their frequency steps prior to the FFT operation.

Optionally a Blackman or a Hamming window can be applied to the input data prior to the Doppler FFT operation to suppress side-lobes at the expense of a slightly worse noise bandwidth. Pulse Doppler processing is thus computationally more demanding than MTI, but improves signal-to-noise ratio (SNR) performance and provides target velocity information.

Computationally N independent FFTs over all P samples are required, each of which are preceded by a windowing function as shown below.

```
loop i=0 to N-1
 loop j=0 to P-1
   win[i][j].RE = pc[j][i].RE * w[j]
   win[i][j].IM = pc[j][i].IM * w[j]

loop i=0 to N-1
  dop[i] = FFT(win[i])
```

The processing requirements for the FFT stage are thus N×P/2×$\log_2$P×4 real multiplications and memory writes as well as N×P/2×$\log_2$P×6 real additions and memory reads. As before, bit-inverted addressing on either the input or output as well as twiddle-factor coefficients are required. The window function requires N×P×2 multiplications and memory writes as well as N×P×3 memory reads.

A.12 ENVELOPE CALCULATION

Envelope calculation (or linear detector) takes the I and the Q value of each sample as input and calculates the complex modulus (magnitude) as shown in Eq. A.8.

$$|x| = \sqrt{x_I^2 + x_Q^2} \tag{A.8}$$

Although one of the more simple algorithms, the envelope or magnitude calculation is computationally demanding because of the square root operation. Depending on the processing architecture, the lack of a square root operation may have a significant impact on performance. Historically the linear detector was used to scale the output in fixed point processors, but the square-law detector is often preferred on modern floating-point processors to simplify the computational requirements. The square-law detector (or squared magnitude operation) does not make use of the square root $(x_I^2 + x_Q^2)$. Another variation takes the (linear) magnitude and scales the output logarithmically. In such a case the square root operation simplifies to a multiplication by 2 after the log operation.

Practically, the sum of squares operation can be implemented as a complex multiplication with its complex conjugate if such an operation is available on the specific architecture.

The linear magnitude takes the square root of the sum of the squares of real and imaginary parts, requiring N×P×2 multiplications and memory reads as well as N×P additions, square roots, and memory writes as shown below.

```
loop j=0 to N-1
 loop i=0 to P-1
   env[j][i] = SQRT(dop[j][i].RE^2 + dop[j][i].IM^2)
```

The log magnitude and squared magnitude operations have similar computational requirements to the linear magnitude operation, with the exception of requiring a logarithmic operation instead of a square root operation, or neglecting it entirely.

A.13 CONSTANT FALSE ALARM RATE

The process of detection compares each radar measurement to a threshold; if the test cell x_t is greater than the threshold $\hat{T}$, a target detection decision is made.

$$P_D = \begin{cases} H_1 & \text{if } x_t \geq \hat{T} \\ H_0 & \text{if } x_t < \hat{T} \end{cases} \tag{A.9}$$

The radar detector is designed such that the highest possibility of detection can be achieved for a given SNR and probability of false alarm (P_{FA}). If the statistics of the interference are known a priori, the threshold can be chosen for a specific probability of false alarm. However, since the parameters of these statistics are typically not known in advance, the CFAR detector tracks changes in the interference levels and continuously adjusts the threshold to keep the probability of false alarm at a constant. The CFAR detection process thus greatly improves the simple threshold detection performance by estimating the current interference level rather than just assuming a constant level.

A.13.1 Cell Averaging CFAR

The simplest implementation of the CFAR detector is the cell averaging (CA) variant. The CA-CFAR detector estimates the interference by averaging R reference cells x_r around each test cell. CA-CFAR relies on two basic assumptions:

- The reference cells contain interference with the same statistics as test cell, and

- The reference cells do not contain any targets.

The CFAR window (consisting of all the reference cells used for the interference estimation) is computed for each cell in the range-Doppler map. It can be any arbitrary size across one dimension in the range vector or across two dimensions in the range-Doppler map. Fig. A.10 depicts the case of a two dimensional window consisting of a 5x5 matrix on each side of the test cell (the CFAR window: R = 2×WIN_R×WIN_D cells). At the extremes, Doppler cells are wrapped around; that is, the reference cells from the opposite side of the Doppler dimensions are used. In the range dimension, cells are either duplicated or single-sided reference windows are used.

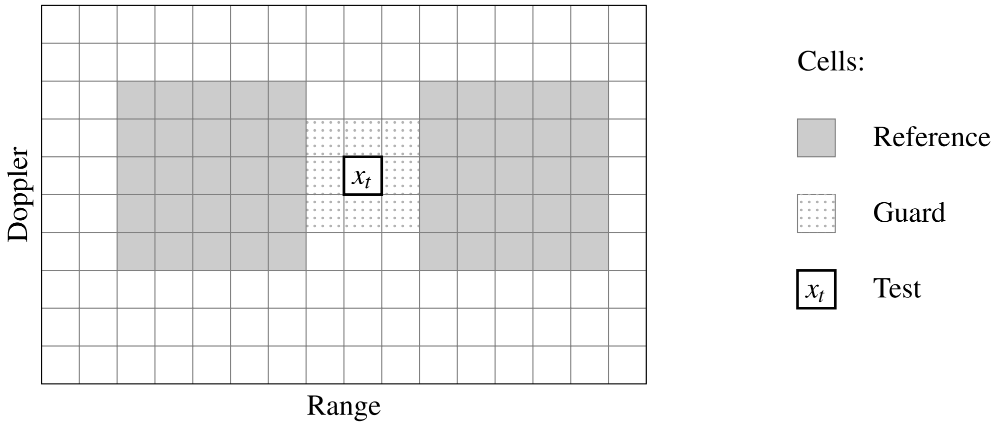


Figure A.10: Cell averaging CFAR window

The estimated threshold $\hat{T}$ can now be computed by multiplying the average interference for each cell by the CFAR constant α as shown in Equations A.10 and A.11.

$$\hat{T} = \frac{\alpha}{R}\sum_{r=1}^{R} x_r \qquad \text{where} \tag{A.10}$$

$$\alpha = R[P_{FA}^{-1/R} - 1] \tag{A.11}$$

To minimize target masking effects, smaller of cell averaging (SOCA-) CFAR uses the quantitatively smaller of the two (left and right) windows. Similarly greater of cell averaging (GOCA-) CFAR uses the larger of the two windows to suppress false alarms at clutter edges. Another variation of CFAR determines the mean of both the left and right CFAR window, and depending on their difference, makes a logical decision whether CA-, GOCA-, or SOCA- CFAR is to be used.

Heterogeneous environments may bias the threshold estimate. The censored (CS-) CFAR discards the M largest (both largest and smallest in the case of trimmed mean (TM-) CFAR) samples in the window, making it much more suitable for such environments than the CA-CFAR. The interference power is then estimated from the remaining cells as in the cell averaging case. Sorting the reference cells is extremely demanding from a computational perspective, and hence a minimum / maximum selection algorithm that loops over all the reference cells is often used for smaller values of M.

A.13.1.1 Sequential implementation

Numerous approaches and methods for calculating the CFAR threshold from a software perspective are possible; an iterative method, a sliding window method and a sum area table method.

Iterative Method

In the iterative CFAR calculation the reference cells are iteratively summed for each test cell according to Equation A.10. Although this method is simple, it is computationally intensive and leaves a lot of room for optimisation. Processing architectures that have vectorised single instruction multiple data (SIMD) instruction sets (which feature packed additions) are well-suited for this CFAR approach. However, the data alignment requirements of these instructions typically don't support the iterative process without some modifications.

Sliding Window Method

The CFAR sliding window method relies on the fact that the threshold for the test cell x_t can be calculated using the reference window of the previous test cell x_{t-1}; new cells that are sliding into the window are added, while old cells that are sliding out of the window are subtracted.

In Fig. A.11, the reference window for the current test cell x_t (shaded grey) is calculated by subtracting the cells directly on the left of each window from the previous windows and then adding the rightmost cells of the current windows.

The old cells that are sliding out of the window do not necessarily have to be re-computed, since their sum could be stored in a circular buffer and subtracted from the previous window.

Sum Area Table Method

The CFAR summed area table (SAT) method [129] makes use of a precomputed sum matrix to determine the sum of the relevant reference cells. The SAT matrix is defined in Eq. A.12.

$$SAT(i,j) = \sum_{a=0}^{i} \sum_{b=0}^{j} x(a,b) \tag{A.12}$$

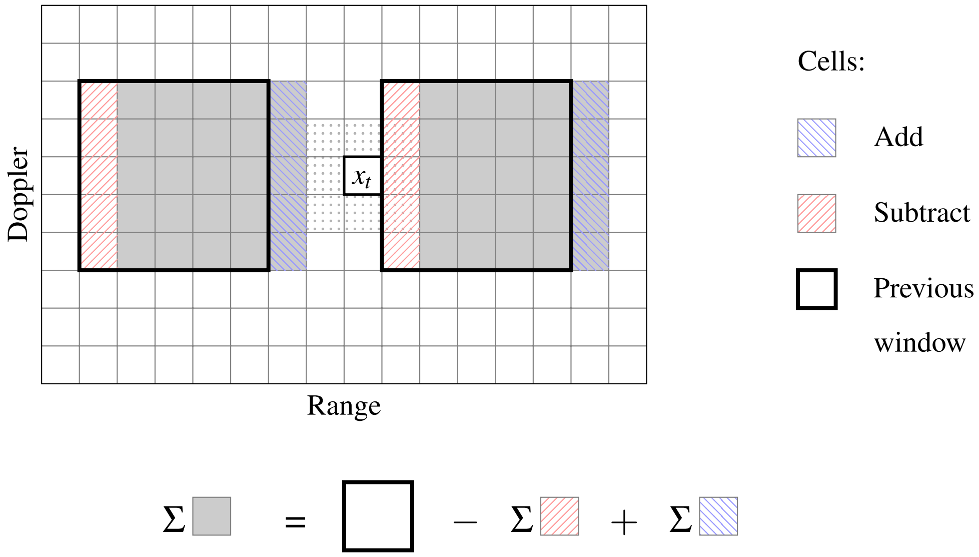


Figure A.11: CFAR sliding window method

Each entry in the SAT thus represents the sum of all the original reference cells below and to the left of that entry.

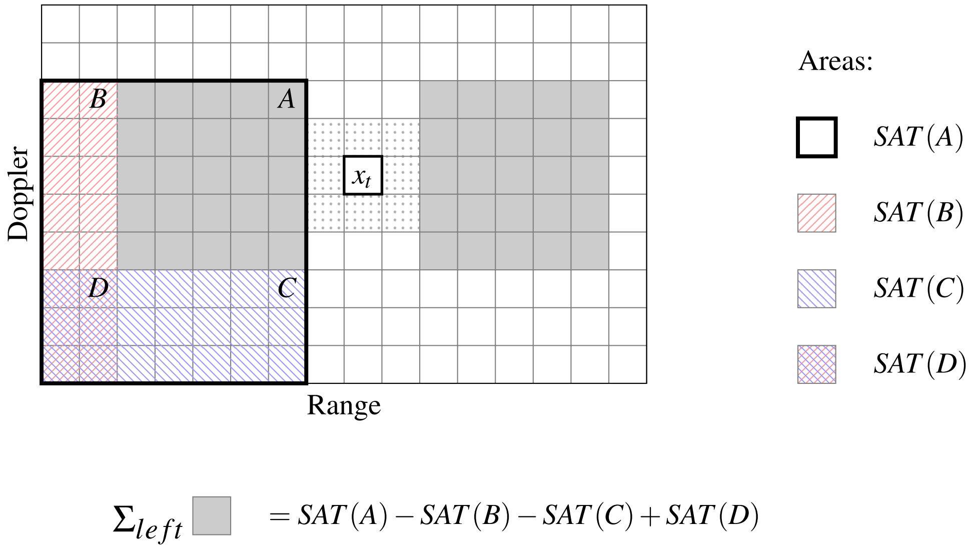


Figure A.12: CFAR sum area table method

For example $SAT(A)$ is the sum of all cells enclosed in bold in Fig. A.12. In order to calculate the left reference window (shaded grey), the areas to the left ($SAT(B)$) and below ($SAT(C)$) are subtracted from the total area ($SAT(A)$). Since the intersecting area ($SAT(D)$) was subtracted twice from the total area, it has to be added to the result again. The same principle applies to calculating the right reference window.

This method thus only requires the lookup of 8 values in the SAT matrix to calculate the entire reference window for each test cell. Care needs to be taken with the number of bits used to represent the SAT, since overflows can easily happen with large range-Doppler maps.

The iterative, sliding window and sum-area-table methods for calculating the CFAR threshold can be ported to a multi-cored implementation. The range-Doppler map is split into range blocks (leaving the complete Doppler dimension) and computed individually on each processing core. Care needs to be taken with overlapping range samples, which need to be copied to the local core before each job can be scheduled.

A.13.1.2 Hardware implementation

The CFAR algorithm is often realised in hardware as a tapped delay line or a shift register with an adder tree structure. Depending on the length of the CFAR window and the number of guard cells used, this delay line may become extremely long, even for the one dimensional window. For the two dimensional window this approach becomes highly impractical as the shift register has to be longer than the range line. If multiple memory read buses are available, multiple tapped delay lines could be used for a multi-dimensional window.

A.13.2 Adaptive CFAR

Adaptive CFAR algorithms iteratively split the window (including the test cell) at all possible points and then calculate the mean of the left and right windows. The most likely splitting point M_t that maximises the log-likelihood function based on the calculated mean values is then chosen. The final step performs the standard CA-CFAR algorithm on the data in which the test cell is located. In such a case, a one dimensional SAT of all reference cells can be formed as shown in Eq. A.13.

$$SAT(i) = \sum_{a=0}^{i} x(a) \tag{A.13}$$

The sum of all cells in the windows to the left of the transition point M_t is now simply sum_area[M_t], while the sum of the right window is sum_area[R] - sum_area[M_t]. The log-likelihood function [18] is defined in Eq. A.14.

$$L(M_t) = (R - M_t)\ln(\beta_r) - M_t \ln(\beta_l), \qquad M_t = 1, ..., R-1 \tag{A.14}$$

where β is the sample mean of the reference cells either to the right or the left of M_t. An iteration over the (R-1) possible transition points is required to select M_t such that L(M_t) is maximised. If M_t is smaller than R/2, the left window (otherwise the right window) is selected as the CFAR statistic.

A.13.3 Order Statistic CFAR

Similar to CS and TM-CFAR, order statistic (OS-) CFAR also requires the samples in the window to be numerically sorted. Rather than discarding samples however, OS-CFAR selects the K-th sample from the sorted list as the interference statistic. This value is then multiplied by a fractional and used in the comparison against the current cell under test. Although a selection algorithm could be used, well established sorting algorithms such as merge-sort can sort R samples with R×log$_2$R comparisons. Since the reference window has to be sorted for each cell in the range-Doppler map, the sorting stage is very computationally demanding compared to CA-CFAR, but provides improved performance in multi-target environments.

A.13.4 Computational Requirements

For the CA-CFAR case, both sides of the window are summed together as shown below.

```
loop j=0 to N-1
 loop i=0 to P-1
   sum_left = 0, sum_right = 0
   loop l=0 to WIN_R-1
    loop k=-WIN_D/2 to WIN_D/2
      sum_left += env[j+GUARD+l][i+k]
   loop l=0 to WIN_R-1
    loop k=-WIN_D/2 to WIN_D/2
      sum_right += env[j-GUARD-l][i+k]
   if (env[j][i] > alpha * (sum_right+sum_left))
     target.RANGE = j
     target.VELOCITY = i
```

Thus N×P×(R+1) additions, N×P×(R+1) memory reads, as well as N×P fractional multiplications, and comparisons are required. The number of memory writes depends on the expected number of targets.

Sliding window techniques, where previously summed columns are stored in registers and reused in the next iteration, further reduce the number of memory reads and additions to N×R + N×(P-1)×WIN_R×2 + N×P and N×R + N×(P-1)×(WIN_R×2+2) respectively.

Computationally the GOCA and SOCA CFAR classes are similar to the CA-CFAR with the exception of one less addition and one extra comparison for each N×P loop. The sliding window approach to SOCA and GOCA requires 2 more additions (as well as 1 extra comparison) per cell in order to keep the two window sums separate.

For CA-CFAR with small values of M (between 1 and 3), a selection algorithm as shown below is well-suited.

```
loop m=0 to M-1
   min = 0, max = 0
   loop d=1 to R-1
     if x[min] > x[d] min=d
     if x[max] < x[d] max=d
   delete x[min]
   delete x[max]
```

This selection algorithm thus requires M×(R-1)×2 comparisons, M×R memory reads and M×2 memory invalidations per cell in the range-Doppler map.

The merge-sort for the CS-CFAR sorts R samples with $R\times\log_2 R$ comparisons, memory writes and memory reads. Since this sorting operation is done for each cell in N×P, the computational requirements very demanding.

The Adaptive CFAR one dimensional SAT method involves the following calculation.

```
sum = 0
loop r=0 to R-1
   sum = sum + x[r]
   sum_area[r] = sum;
```

The log-likelihood function requires 3 additions, 2 divisions, multiplications and logarithm calculations, as well as 1 memory read and write per transition point as shown below.

```
loop mt=1 to R-1
   L[mt] = (R-mt) * ln((sum_area[R]-sum_area[mt])/(R-mt)) -
          mt * ln( sum_area[mt] / mt )
```

A.14 NOISE LEVEL ESTIMATION

Once a detection is made, the data processor needs to decide whether a detection is a false alarm or not. One of the factors influencing this decision is the measured noise level during pulse reception. The noise-level estimation (NLE) subtracts successive samples from each other (similar to the 2 pulse MTI canceller), leaving a crude estimation of the noise level.

A.15 MONOPULSE CALCULATIONS

The monopulse calculation is used to improve directional accuracy of a target within the current beam. It makes use of the Δ_{AZ}/Σ and Δ_{EL}/Σ ratios to estimate the bearing (azimuth and elevation angles) relative to the main beam. These angle estimates are then averaged over numerous CPIs by the data processor. Practically it involves calculating a ratio of the sum and difference channels, which is then passed to the data processor for detection purposes.

A.16 DATA PROCESSOR

The data processor interface involves compiling the results from the CFAR stage into a set of target reports that can be sent to the data processor over an appropriate media (typically in packet format over gigabit Ethernet).

If the CFAR process writes the range and Doppler bin indexes to an array once a positive target identification is made, the computational requirements only involve reading that list and writing it to the media access control (MAC) interface together with some control fields in the packet structure. When the CFAR process outputs a hard-limited range-Doppler map, an additional iteration over the matrix is required (N×P memory reads and comparisons), writing the indexes of the positive identifications into a new array. The size of the index array, and thus also the number of memory writes, is determined by the probability of false alarm, which is chosen for a specific system at design time.

The post detection processing functions of the data processor are application dependent. The data processor typically receives a set of target reports (including false alarms) from the signal processor and performs combinations of higher level functions such as clustering, radar cross section (RCS) measurement, tracking, detection, classification, surveillance or scanning.

The processing occurs on bursts of target reports that are received every CPI. These target reports include critical target information such as beam direction, range, Doppler, signal-to-noise ratio estimates, and radar-specific parameters for the current CPI. Since the CPI is typically in the range of a few milliseconds to a few seconds, the processing requirements are not as intensive as with the radar signal processor and standard microprocessors or personal computers (PC) are often used as the data processor.

APPENDIX B

COMMON SIGNAL PROCESSING OPERATIONS

B.1 OVERVIEW

This chapter introduces some of the signal processing operations commonly used by the radar signal processor. Operations are analysed from an algorithmic as well as a computational perspective, giving details for both parallel and sequential (dataflow and control-flow) implementations. These operations are listed in no particular order in the following sections.

B.2 BASIC ARITHMETIC OPERATIONS

The most fundamental mathematical operations used in a typical processor are summarised in Table B.1.

Table B.1: Fundamental arithmetic operations used in a typical processor

Mathematical Operation	Comment
Addition	Crucial; typically single cycle operation
Subtraction	Crucial; typically single cycle operation
Multiplication	Crucial for most DSP algorithms, iterative additions
Division	Quotient and remainder. Radix-2 non-restoring algorithm
Complex Addition	Two real additions. SIMD instruction
Complex Subtraction	Two real subtractions. SIMD instruction
Complex Multiplication	Four real multiplications, one real addition and subtraction
Square Root	CORDIC
Logarithm	Leading ones detector
Exponential	Iterative multiplications, binary exponentiation
Trigonometric Functions	CORDIC or DDS look up tables with Taylor series correction

Addition, subtraction and multiplication form the basis of almost all DSP related algorithms and typically sustain a throughput of 1 output sample per clock cycle on modern processing technologies. Integer multiplication or division by powers of two can be accomplished with left or right bit-shifting respectively. For high-speed implementations or floating-point variations, the throughput is still sustained at the expense of additional latency cycles. Because of the streaming nature of DSP algorithms, this added latency does not cause any significant dependency issues and hardly impacts the overall performance.

The complex multiplication can be broken down to 4 real multiplications, 1 real addition and 1 real subtraction. With a few rearrangements, one multiplication can be saved at the expense of one extra addition and two additional subtractions as shown in Eq. B.3.

$$\begin{aligned}
\mathbf{a}*\mathbf{b} &= (a_i + ja_j)(b_i + jb_j) && \text{(B.1)}\\
&= (a_ib_i - a_jb_j) + j(a_ib_j + a_jb_i) && \text{(B.2)}\\
&= (a_ib_i - a_jb_j) + j((a_i + a_j)(b_i + b_j) - a_ib_i - a_jb_j) && \text{(B.3)}
\end{aligned}$$

A routine that counts how many times sequential odd numbers (1,3,5,7,9,...) can be subtracted from an input number could be used for the square root operation, when no hardware support is available. The number of required clock cycles for this operation would however vary depending on the input number. Hardware implementations step through 2 bits per cycle to calculate the square root. Thus it is typically faster to perform a square root operation compared to a division, which works through 1 bit per stage.

Not all of the operations in Table B.1 need to be explicitly supported by the instruction set, as they can be calculated using iterative, recursive, numeric or bit manipulation methods. For example, modern PCs (such as the x86) typically include the elemental trigonometric functions (Sine, Cosine, ArcTan) in their instruction set, but derive other trigonometric functions from these or through polynomial expansions. The coordinate rotational digital computer (CORDIC) algorithm or the direct digital synthesizer (DDS) can also be used for trigonometric function calculations.

B.3 ACCUMULATION

The accumulate operation is used in numerous DSP operations. It involves adding an input number to an internal sum value. On single cycle operations (fixed point addition on most modern processing technologies), the accumulator can be implemented as shown below in Fig B.1.

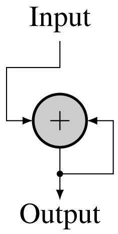


Figure B.1: Signal Flow Graph for the Accumulate Operation

The output of the adder is simply connected to one of its inputs. The other adder input now serves as the accumulator input, where data can be clocked in every cycle. To reset the accumulator value (bypass with a new value on the input), the output is disconnected from the other input.

The floating-point addition operation usually requires several latency cycles to compute an output value. Since the previously computed accumulated value is required for each new input value, a new value can only be input at multiples of the latency cycle count. This poses a problem for streaming applications where new input values become available every clock cycle. If the dynamic range is known beforehand, the floating-point input number can be converted to fixed point, accumulated in fixed point, and the output value converted back to floating-point representation. When the precision is not known beforehand, delayed addition or similar techniques for floating-point accumulation can be used [102, 130, 131].

B.4 FOURIER TRANSFORM

The Fourier transform is one of the most important signal processing operations in radar. It transforms a time-domain input sequence to the frequency domain representation and vice versa with the inverse Fourier transform.

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{B.4}$$

For digital signal processing the above integral is solved either with the discrete Fourier transform or the fast Fourier transform as discussed below.

B.4.1 Discrete Fourier Transform

The discrete time Fourier transform is shown below in Eq. B.5.

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nk/N} \tag{B.5}$$

Rewriting Eq. B.5 with $W_N = e^{-j2\pi/N}$ gives

$$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{nk} \tag{B.6}$$

From a computational perspective, the following pseudo code describes the DFT algorithm:

```
const double ph0 = - 2 * PI / N
for (k=0; k<N; k++)
    Complex t = 0.0
    for (n=0; n<N; n++)
        t += x[n] * exp(ph0*n*k)
    y[k] = t
```

Computationally there are N^2 complex multiplications and additions in the above algorithm. If the twiddle factors are precomputed and stored, $2N^2$ complex memory reads and N complex memory writes are required. Even for relatively small Fourier transform sizes these requirements are severely stringent.

B.4.2 Inverse Discrete Fourier Transform

The inverse discrete time Fourier transform (IDFT) is defined in Eq. B.7.

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j2\pi nk/N} \tag{B.7}$$

There are numerous tricks for using the DFT to compute the IDFT:

- Reversing the input: $IDFT(x[n]) = DFT(x[N-n])/N$,
- Complex conjugate of both input and output: $IDFT(x[n]) = DFT^*(x^*[n])/N$,
- Swap real and imaginary parts on both input and output: $IDFT(x[n]) = swap(DFT(swap(x[n])))/N$.

B.4.3 Fast Fourier Transform

Although the DFT is extremely simple to understand, it is rather inefficient and the computational requirements become excessive when N is large. The Fast Fourier transform eliminates the redundant arithmetic operations of the DFT by reusing previously calculated values. The Cooley-Tukey Radix-2 algorithm is one of more popular algorithms, with the restraint of N having to be a power of two. When N is not a power of two, the input data is zero padded to the next power of two.

The performance improvement comes from its capability to segment an N-point DFT into two N/2-point DFTs. This may not seem like it helps with the computational requirements, but iterative splitting yields a simplified structure that only requires a few arithmetic operations. The pseudo-code for the non-recursive decimation-in-frequency Radix-2 algorithm is shown below.

```
N = 2^pow
for stage=pow to 1 step -1
    m = 2^stage
    mh = m/2
    for k=0 to mh-1
        W = exp(-j*2*PI*k/m)
        for r=0 to N-1 step m
            addr0 = r+k
            addr1 = r+k+mh
            A = x[addr0]
            B = x[addr1]
            x[addr0] = (A + B)
            x[addr1] = (A - B) * W
bit_reverse(x)
```

The inner loop FFT butterfly operation is depicted graphically in Fig. B.2.

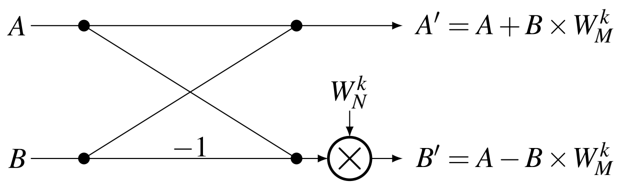


Figure B.2: FFT Decimation-in-Frequency Butterfly

Where $W_N^k = \exp(-j2\pi k/N)$ is the twiddle factor. The decimation-in-frequency FFT butterfly can be described algorithmically as:

```
A′[addr0] = (A[addr0] + B[addr1])
B′[addr1] = (A[addr0] - B[addr1]) * W[addrT]
```

The FFT dataflow structure for the case of N=16 is shown below in Fig. B.3.

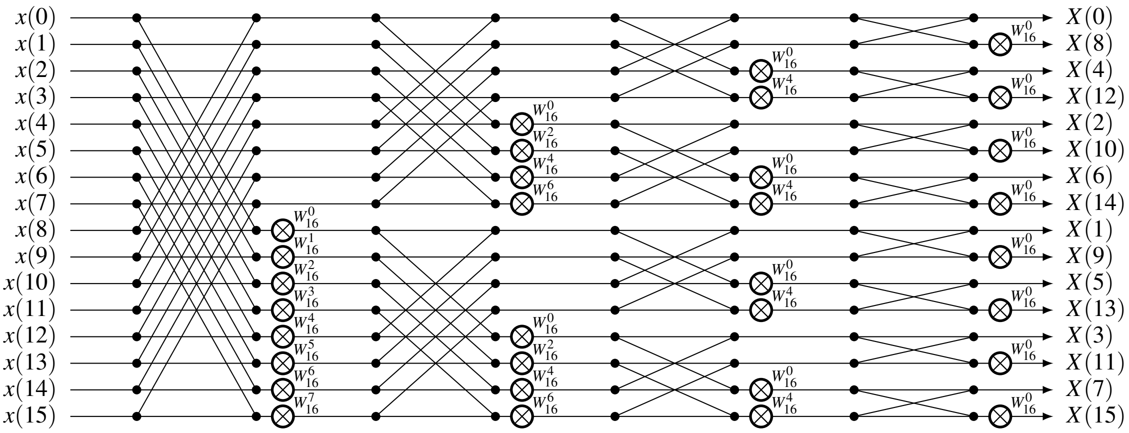


Figure B.3: FFT Radix2-DIF dataflow pattern

Note how there are $\log_2 N = 4$ stages, each consisting of $N/2 = 8$ butterflies. The above structure and butterfly are for a decimation-in-frequency (DIF) type algorithm, better suited for when the input sequence is complex. Decimation-in-time (DIT) algorithms subdivides the input data into odd and even components, and are better matched for real input data sequences from a standpoint of general hardware.

As shown in Fig. B.3, the radix-2 FFT of length N has $\log_2 N$ stages, each consisting of N/2 butterflies. Each butterfly requires one complex multiplication, addition and subtraction. Thus the total computational requirements for an FFT are $\mathrm{P}\times\mathrm{N}/2\times\log_2\mathrm{N}\times 4$ real multiplications and memory writes as well as $\mathrm{P}\times\mathrm{N}/2\times\log_2\mathrm{N}\times 6$ real additions and memory reads. Bit inverted addressing is required for either memory reads or writes, depending on the selected DIT or DIF algorithm type. The bit inverted addressing can be implemented as a precomputed lookup table in main memory if a bit-reversal instruction is not available on the architecture.

The radar requirements for the FFT operation dictate a wide range of FFT lengths; small length FFTs of 8 or less for pulse-Doppler processing, all the way to extremely large sizes across the entire range line for pulse compression or fast filtering. Hardware implementations (FFT coprocessors) rely on highly pipelined structures that can sustain a high throughput bandwidth [132]. These FFT coprocessors however typically only support a few fixed or limited range of lengths. For purely sequential implementations, DSP manufacturers often include radix-2 butterfly instructions to speed up the implementation. Since decimation in time or frequency algorithms permit FFTs of length N to be computed with two FFTs of length N/2, a hybrid soft-core implementation consisting of several smaller memory mapped FFT-coprocessors could be software coordinated to achieve larger length FFTs. Optimisations such as these can improve the performance as well as simplify the programming model from a HDL design to a library based software environment, permitting quick evaluation of new radar algorithms during field-tests.

Other implementations, such as the radix-4 FFT, require less complex multipliers (75%) than the radix-2 FFTs. In modern processing architectures however, the performance limiting factor is commonly not the arithmetic multiplier performance, but the memory bandwidth. The real saving from the radix-4 implementation is the number of memory accesses, since the number of stages is further halved. For a radix-4 implementation the input sequence is required to be a power of 4 samples long.

B.4.4 Inverse Fast Fourier Transform

Similar to the inverse discrete Fourier transform, the inverse FFT can be computed with the forward FFT. One of the simplest methods is to use the same algorithm, but different twiddle factors ($W_N = e^{j2\pi/N}$). The output will still have to be scaled by 1/N though.

B.5 FINITE IMPULSE RESPONSE FILTERS

Another crucial signal processing operation is the FIR filter. An important characteristic of the FIR filter is that it can be designed to have a linear phase response, making it useful for a wide range of applications that require little or no phase distortions such as the Hilbert transformer. Additionally, FIR filters are inherently stable and require no feedback. The transfer function for a causal FIR filter of order L (length $L+1$) is shown in Eq. B.8.

$$H(z) = h(0) + h(1)z^{-1} + ... + h(L)z^{-L} \tag{B.8}$$

The FIR filter structure (and hardware systolic array implementation) as derived from the transfer function is depicted in Fig. 5.6. The input sample feeds into a shift register, which delays the input sequence by one sample every clock cycle. Each delayed sample is multiplied by a unique coefficient, while the multiplier outputs are summed to produce an output sample every clock cycle. A filter of order L has $L+1$ taps, coefficients and multipliers and needs L adders.

Mathematically each output sample can be calculated as follows:

$$y[n] = \sum_{i=0}^{L} b_i x[n-i] \tag{B.9}$$

Software tools for designing and analysing various types of FIR filters are readily available (for example MATLAB's *fdatool* [17]) and simplify the coefficient generation process. FIR filter frequency responses have a linear phase if the impulse response is either symmetric or anti-symmetric as shown in Eq. B.10 or Eq. B.11.

$$h[n] = h[L-n], \qquad n = 0, 1, ..., L \tag{B.10}$$

or

$$h[n] = -h[L-n], \qquad n = 0, 1, ..., L \tag{B.11}$$

The direct FIR filter structure maps well to fused multiply accumulate instructions available on most DSP's. These purely sequential implementations (such as DSPs or general purpose CPUs), however, require numerous iterations over the array for a single output sample. The processing requirements are thus quite stringent; at a sampling frequency of 200 MHz and a FIR filter length of 512, more than 102 billion multiplications and additions are required every second - a throughput of 102 GMAC's per second. From a hardware implementation perspective, the direct form FIR filter structure maps well to systolic array structures [101] (as shown in Fig. 5.6), capable of producing an output sample every clock cycle at the expense of additional resource usage.

So far only FIR filters with purely real input data and coefficients were discussed. When the input data is complex, the same algorithm applies with complex coefficients. Since the complex multiplications and additions are computationally more demanding though, both I and Q channels are often filtered separately with two real-valued FIR filters when there are no special frequency response requirements.

B.6 INFINITE IMPULSE RESPONSE FILTERS

An infinite impulse response (IIR) filter is a recursive filter that requires less coefficients that an equivalently specified FIR filter. It can be designed to have a sampled version of the frequency response corresponding to that of an analogue filter. An IIR filter with numerator order L and denominator order M is mathematically defined as

$$y[n] = \sum_{i=0}^{L} b_i x[n-i] - \sum_{j=1}^{M} a_j y[n-j]. \tag{B.12}$$

The first summation in Eq. B.12 is identical to the FIR filter. The second summation provides feedback, including previously calculated outputs multiplied with their relevant coefficients in the calculation. For each output sample the computational requirements are thus:

Number of multiplications:	$L+M+1$
Number of additions / subtractions:	$L+M$.

Since the IIR filter order is typically less for an equivalently specified FIR filter, less processing and memory is required. However, if the filter is not designed properly, the filter could become unstable as a result of overflow in the datapath. IIR filter coefficients are very sensitive to quantisation limitations, making them hard to implement practically in fixed point processors with limited dynamic range. Because of data dependencies (requiring the previous output sample before the next one can be calculated), the IIR filter is not as well-suited for high-speed streaming applications compared to the FIR filter. For example, pipelining and SIMD vectorisation cannot be used in the datapath because of the dependency between loop iterations. For these reasons IIR filters are not often used for practical implementations of radar signal processors.

B.7 PHASE SHIFT

The digital phase shift operation is a common DSP operation used in a variety of algorithms. To phase shift an input sample (I_i,Q_i) with an angle of θ, the phase shifted output (I_{ps},Q_{ps}) is given by:

$$I_{ps} = I_i(\cos\theta) - Q_i(\sin\theta) \tag{B.13}$$

$$Q_{ps} = I_i(\sin\theta) + Q_i(\cos\theta). \tag{B.14}$$

Eq. B.13 reminds us of the complex multiplication operation. The most straightforward technique of digitally phase shifting an input signal is thus by multiplying it with a complex coefficient. The real and imaginary components of the complex coefficient are determined by the value of $\cos\theta$ and $\sin\theta$ respectively.

B.8 SORTING

For some CFAR processing classes, the reference cells need to be sorted in ascending or descending order. The sorting operation is computationally and memory intensive even for the small number of reference cells in the CFAR window. Sequential sorting algorithms (such as Heapsort, Mergesort or Quicksort) are well documented and characterised in terms of performance and memory usage on commercial PC systems [133]. Sorting networks are data independent and thus much better suited for parallel hardware implementations. The compare and swap operation, which is used by all the sorting networks, is represented by a vertical line as shown below in Fig. B.4.

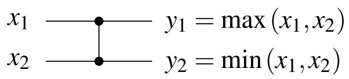


Figure B.4: Network Comparator

Depending on the required output order (ascending or descending), y_1 could also be $\min(x_1, x_2)$ while $y_2 = \max(x_1, x_2)$, as long as the convention is kept the same throughout the selected algorithm.

B.8.1 Odd-Even Transposition Sort

One of the simplest parallel sorting network implementations is the odd-even transposition sort. Adjacent (odd, even) pairs in the input stream are compared and swapped if they are in the wrong order. In the second step, adjacent (even, odd) pairs are compared and swapped. This alternating sequence is repeated until the list is sorted as shown in Fig. B.5.

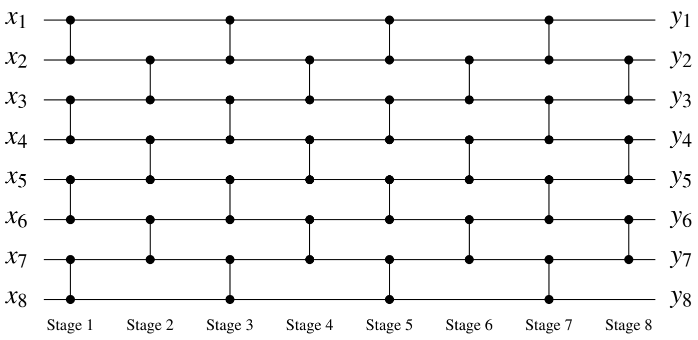


Figure B.5: Odd-Even Transposition Sorting Network

There are N stages each consisting of either $N/2$ or $N/2-1$ comparisons for the even and odd stages respectively. Thus a total of $N(N-1)/2$ comparators and swaps are required for the even-odd transposition sorting network. Provided that there are enough parallel comparators, each stage could be completed in a single step. The algorithm would thus take N steps to complete the sorting operation. In a pipelined implementation a throughput of 1 sorted output array can be achieved each clock cycle with a latency of the 8 stages.

B.8.2 Bitonic Merge-Sort

The Bitonic mergesort improves on the simple odd-even transposition network. The input array is partitioned into two equally sized sub-arrays, which are recursively sorted in parallel and finally merged. The original algorithm [134] was devised for input sequences of powers of 2, but later modifications proposed an algorithm for arbitrary sizes. Fig. B.6 shows the bitonic sorting network for $N = 8$.

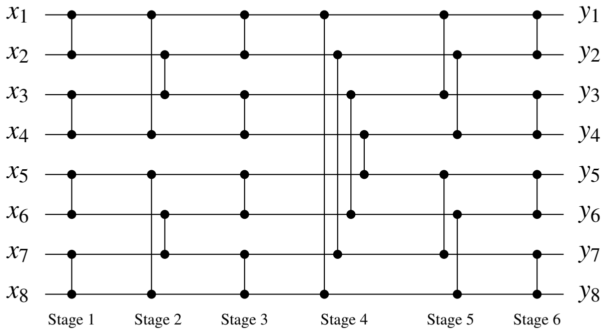


Figure B.6: Bitonic Sorting Network

The algorithm is well-suited for a parallel architecture such as an FPGA. The number of stages (directly related to the latency) is $\log_2(N)(\log_2(N)+1)/2$, each consisting of $N/2$ comparisons. The total number of comparators for a streaming implementation is thus

$$C = \frac{N\log_2(N)(\log_2(N)+1)}{4} \tag{B.15}$$

Note that the algorithm is highly regular in dataflow with each signal path having the same length, and the number of comparators remaining constant in each stage.

B.8.3 Odd-Even Merge-Sort

Another related data-independent sorting algorithm is Batcher's odd-even mergesort. It sorts N input samples in the same number of stages as the bitonic mergesort, but uses fewer comparators [135]. The required number of comparators is given in Eq. B.16.

$$C = \frac{N\log_2(N)(\log_2(N)-1)}{4} + N - 1 \tag{B.16}$$

The odd-even mergesort is a popular sorting network on graphic processing units (GPUs). On FPGAs and ASICs additional registers will have to be used to buffer the data as some paths route through fewer comparator stages as others (the number of comparators per stage is not a constant).

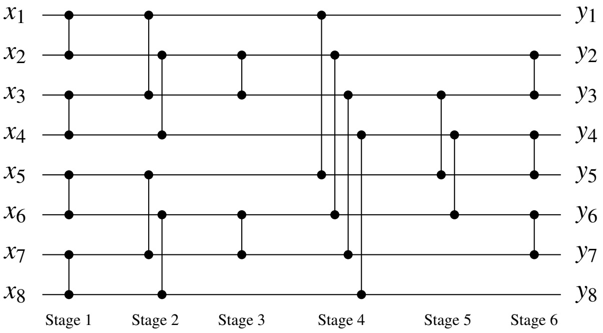


Figure B.7: Odd-Even Merge-Sort Network

B.9 ARRAY OPERATIONS

Array operations refer to elementwise arithmetic operations performed on two input arrays or an array and a scalar or constant. Mathematically, the operation can be described as

$$y[n] = a[n] \,\Theta\, b[n] \qquad \text{or} \tag{B.17}$$

$$y[n] = a[n] \,\Theta\, c, \tag{B.18}$$

where Θ represents any arithmetic operation as listed in Table B.1. This arithmetic operation is performed iteratively over all N elements in the array. From a computational perspective, the array operations are performance limited by memory bandwidth, when the number of elements that can be fetched and written back via the memory data bus is exceeded by the number of effective parallel arithmetic functional units that can be performed each clock cycle. The instruction bandwidth and architectural execution restrictions may limit the number of effective operations that can be performed each clock cycle, even when a sufficient number or type of arithmetic functional units is available. SIMD instructions are well-suited for array operations.

B.10 MATRIX MULTIPLICATION

The process of matrix multiplication is used extensively by the data processor for higher level processing. Equations B.19 to B.23 describe the process from a mathematical perspective.

$$\mathbf{C}_{np} = \mathbf{A}_{nm} \times \mathbf{B}_{mp} \tag{B.19}$$

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,p} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,p} \end{pmatrix} \tag{B.20}$$

where

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + ... + a_{1m}b_{m1} \tag{B.21}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22} + ... + a_{1m}b_{m2} \tag{B.22}$$

$$c_{1p} = a_{11}b_{1p} + a_{12}b_{2p} + ... + a_{1m}b_{mp} \tag{B.23}$$

Each element in the resulting matrix (c_{xy}) is the dot product of row x in matrix A (m-elements) with column y of matrix B (also m-elements). Computationally, the matrix multiply operation thus requires $n \times p$ dot-products of length m, each requiring m multiplications and $m-1$ additions.

Number of multiplications:	npm
Number of additions / subtractions:	$np(m-1)$.

The matrix multiplication operation is both computationally and memory intensive. Storing two copies of the matrix, one in transposed and one in normal order, with data bus widths the size of a row and column would simplify these memory requirements. A dot-product operation of length m would be required to improve the computational throughput.

B.11 MATRIX INVERSION

Matrix inversion is an important component of beam-forming, which is commonly used in multiple antenna systems for beam steering purposes. The concept of matrix inversion can be seen as a form of matrix division as shown in Eq. B.24,

$$\boldsymbol{AA}^{-1} = \boldsymbol{I}, \tag{B.24}$$

where $\boldsymbol{I}$ is the identity matrix (each diagonal element $i_{jj} = 1$, every other element is zero). Only square matrices with their determinant not equal to zero are invertible (i.e. nonsingular or nondegenerate). The matrix inverse or reciprocal matrix $\boldsymbol{A}^{-1}$ can be computed using numerous methods, such as LU Decomposition or Gaussian Elimination.

B.12 CONVOLUTION

The convolution operation expresses the amount of area overlap between two functions as one is shifted over the another. Convolutions are used in a variety of mathematical and engineering applications including probability theory, weighted moving average calculations and the output responses of linear time-invariant systems. The convolution integral is defined in Eq. B.25 and Eq. B.26 for the continuous and discrete cases respectively.

$$(f*g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{B.25}$$

$$x[n]*h[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m] \tag{B.26}$$

From a computational perspective, the direct implementation of the above summation has a computational cost of N^2 (complex or real) multiplications and $N(N-1)$ (complex or real) additions for N input samples.

B.12.1 Fast Convolution

A substantial reduction in computational requirements can be achieved by performing the convolution in the frequency domain. The fast convolution method multiplies the spectrum of the two input sequences, and then converts the result back to the time domain with the IFFT as shown in Eq. B.27.

$$\mathbf{x}*\mathbf{h} = \mathrm{IFFT}(\mathrm{FFT}(\mathbf{x}) \times \mathrm{FFT}(\mathbf{h})) \tag{B.27}$$

The computational cost for N input samples then becomes:

Number of complex multiplications:	$N + 3N(\log_2 N)/2$
Number of complex additions:	$3N\log_2 N$

When the spectrum of one of the input sequences can be precomputed, only two (rather than three) FFTs and N multiplications are required.

B.13 CROSS-CORRELATION

Similar to the convolution operation, the correlation operation also performs a sliding dot product of two input functions. However, rather than representing the amount of overlap, the cross-correlation operation is a measure of the two functions similarity as a function of time lag between them. This has a variety of applications in pattern matching, signal detection and almost any statistical analysis involving dependence. For example, the matched filter used for pulse compression is a correlation between the transmitted pulse template and the received data stream. The correlation integral is defined in Eq. B.28 and Eq. B.29 for the continuous and discrete cases respectively.

$$(f \star g)(t) = \int_{-\infty}^{\infty} f^*(\tau)g(t+\tau)d\tau \tag{B.28}$$

$$x[n] \star h[n] = \sum_{m=-\infty}^{\infty} x^*[m]h[n+m] \tag{B.29}$$

Comparing the correlation and convolution integrals, it should be evident that the cross-correlation of functions $f(t)$ and $g(t)$ is equivalent to the convolution of $f^*(-t)$ (time inverted and conjugated $f(t)$) and $g(t)$. Convolution and correlation are thus identical for real-valued functions when $f(t)$ is symmetric. Computational requirements for the correlation are identical to the convolution, and can also be computed in the frequency domain as with the fast convolution.

B.14 DOT PRODUCT

The dot product (or scalar product / inner product) is an important concept in linear algebra and physics. Geometrically it can be interpreted as the length of the projection of one vector onto another when their starting points coincide. The dot product can also be used as a subcomponent of multiple other signal processing operations such as FIR filtering, matrix multiplication, convolution or any other sum of products application. The dot product of two vectors **a** and **b**, each of length N, is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a[i]b[i] \tag{B.30}$$

Thus N multiplication and $N-1$ additions are required for the computation. Digital signal processors use a multiply accumulate instruction in a loop, while increasing the address pointers of the input arrays, to calculate the dot product. For a hardware implementation capable of sustaining a throughput of 1 output sample every clock cycle, a balanced adder tree structure is often used. Fig. 5.7 shows an example of an eight point dot product with a balanced adder tree structure.

New input samples (a(0)-a(7) and b(0)-b(7)) can be clocked in every clock cycle, with the result appearing on the output port after the combined latency of the multipliers and adder tree (1x MUL_LAT + 3x ADD_LAT in this case).

B.15 DECIMATION

Decimation (or downsampling) refers to the process of reducing the sampling rate of a signal. Practically decimating by a factor of N involves only making use of every N-th sample in the input stream and discarding all other samples. Fractional decimation factors require an integer interpolation stage followed by an integer decimation stage. Software downsampling implementations simply rearrange the memory or use a modified addressing scheme in the next process that makes use of the decimated data. On hardware systems, the decimation process introduces another clock domain at a fraction of the input sampling frequency.

B.16 SCALING

Scaling is an important process for fixed point processors, commonly inserted before and/or after processing stages such as FFTs or filters. For example, the change in dynamic range before and after the digital pulse compression algorthm is very large, causing finite word length effects when not considered. Input (or output) data is left or right shifted in order to balance dynamic range and avoid data overflow. Even on floating-point processors, scaling is still required for normalisation or converting data to fixed point for other interfaces (e.g. DAC output). When the simple left or right shifting operation is not adequate, or the scaling factor is not known beforehand, the scaling operation is preceded by finding the maximum and/or minimum value in the input stream. Subsequently all elements in the input stream are then divided or multiplied by that scaling factor.

Rather than scaling the input data to some predetermined range, data could also be scaled logarithmically to decrease the dynamic range, for example for graphical representation purposes.

B.17 INTERPOLATION

Interpolation is the process of increasing the sampling rate (upsampling). The simplest form of interpolation is linear interpolation, which estimates the value of a new datapoint from a straight line drawn between the two closest known datapoints. For example, the linear interpolation between datapoints x(1) and x(2) is given as

$$x(1.5) = \frac{x(1)+x(2)}{2}, \tag{B.31}$$

which is halfway between the two datapoints. Although very simple to calculate, linear interpolation is not very precise. In practical systems, interpolation filters are used rather than linear interpolation methods. To interpolate by a factor of N, N-1 zeroes are inserted between consecutive low rate samples before the data is passed through a low pass filter running at the interpolated rate. The low pass filter is typically a FIR filter or a cascaded integrator-comb filter (a FIR filter class that does not require multipliers).

B.18 RECTANGULAR AND POLAR COORDINATE CONVERSION

Converting rectangular coordinates to polar coordinates and vice versa is a common operation in the radar signal processing paradigm. Real and imaginary samples (i.e. I/Q samples) are converted to amplitude and phase components as shown in Fig. B.8.

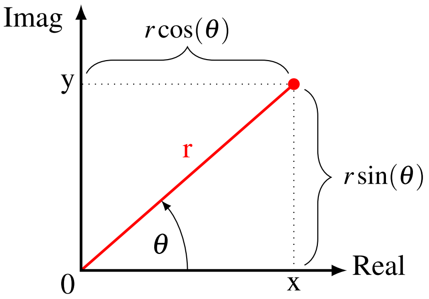


Figure B.8: Rectangular - Polar coordinate conversion

Mathematically these conversions are given in Eq. B.32 to Eq. B.35.

$$x = r\cos(\theta) \tag{B.32}$$

$$y = r\sin(\theta) \tag{B.33}$$

$$\theta = \text{atan2}(x,y) \tag{B.34}$$

$$r = \sqrt{x^2+y^2} \tag{B.35}$$

From a computational perspective, CORDIC algorithms are well-suited for these type of conversions.

B.19 MOVING AVERAGE

The moving average operation is commonly used to smooth out short term fluctuations in a dataset in order to highlight long term trends, making it a type of low pass filter. A simple moving average calculates the sum of N data points, which is then divided by N. The moving average calculation can thus also be used as a sliding window alternative for the CA-CFAR algorithm. In dataflow architectures the summation of the last N values can be implemented as a log-sum structure as shown in Fig. 5.10. In software implementations sliding window techniques similar to those discussed in Section A.13.1.1 can be applied, followed by a multiplication by $1/N$.

B.20 FINITE WORD LENGTH EFFECTS

All of the signal processing operations discussed in this chapter are prone to finite word length effects. Since the radar processing chain includes multiple FIR filters and FFT operations, the signal degradation through these operations is additive and a certain number of bits need to be maintained to reduce distortion and round-off noise at the end of the processing chain.

Additionally the ADC quantisation errors, limited dynamic range, scaling errors (the rounding or truncation between stages to avoid data overflow), precision of filter coefficients as well as twiddle factors all constitute to signal degradation as a result of finite word lengths in fixed point arithmetic [127].

Although a datapath consisting of floating-point arithmetic units solves many of these issues, the round-off noise still plays a significant factor. Since the result of an arithmetic operation is either rounded up or down, depending on the data, the error can become additive (rather than random) between consecutive stages in the worst case scenario. This round-off from single precision floating point arithmetic can cause a variable to drift by as much as 1 part in 40 million multiplied by the number of arithmetic stages it passed through [136].

Historically the hardware for fixed point arithmetic was always considered to be much faster and simpler than equivalent floating-point arithmetic. With modern processing technologies however, the difference in computational throughput is minimal, and the additional silicon area requirements hardly influence the cost [137]. The large dynamic range, and lack of scaling between stages (from both a performance and ease of programming model) make floating-point arithmetic very attractive for radar signal processors.

APPENDIX C

CURRENT PROCESSING ARCHITECTURES

C.1 OVERVIEW

This chapter provides a brief overview of the architectural features of current processors with emphasis on parallelism, execution model, instruction set architecture, and micro-architecture.

Commercial PC systems have retained the same execution model over the past few decades, only changing the microprocessor architecture that implements the execution model [101]. Features such as instruction-, data- and thread-level parallelism, horizontal and vertical microcoding, pipelining, caching, branch prediction, vectorisation, instruction set extensions, out-of-order execution, and multi-threading all aim at improving the overall performance, without changing the underlying execution model. Depending on the application, this instruction-set based execution model might not be the most optimal architecture. In order to propose an optimal radar signal processing architecture, the different architectures and performance enhancement techniques that are typically used in high performance processing architectures are investigated.

C.2 INSTRUCTION SET ARCHITECTURE

One of the most fundamental concepts of a processing architecture is that of the instruction set. The instruction set architecture refers to the way a processor is defined in regard to the interface between hardware and software, the type and number of registers, word size (bit level parallelism), mode of memory addressing, and the various address and data formats. The majority of instruction set architecture approaches can be categorised into Reduced Instruction Set Computing (RISC), Complex Instruction Set Computing (CISC) or Very Long Instruction Word (VLIW) architectures.

C.2.1 Reduced Instruction Set Computing

The simplest instruction set architecture is undoubtedly the RISC architecture. RISC instructions are frequently single cycle instructions that directly link to functional units like multipliers, adders, arithmetic logic units and memory accesses in order to achieve high clock rates. Instructions are register based rather than memory based, with special instructions for loading and storing registers to and from data memory.

Traditional RISC processors execute one instruction every cycle, and hence only a single functional unit is used at a time (horizontal waste, since other functional units sit idle). When an operation requires multiple cycles to complete, execution is halted until the instruction completes (vertical waste, since new data could be clocked into the functional unit every clock cycle for a pipelined solution).

C.2.2 Complex Instruction Set Computing

In contrast to RISC, the CISC instruction set architecture includes more complex instructions, requiring multiple clock cycles or data memory accesses to execute a single instruction. These instructions were designed to simplify assembly language programming, representing high-level functions rather than simple arithmetic operations. Modern CISC architectures, such as Intel's x86, use embedded micro-code to translate the CISC instructions to the underlying RISC-like functional units.

C.2.3 Very Long Instruction Word

Very Long Instruction Word architectures attempt to reduce the horizontal waste of RISC processors at the expense of wider program memory widths. Each instruction is made up of multiple (typically 8 or less) RISC instructions, which are dispatched to the relevant execution units. Thus multiple existing functional units can be used in parallel (instruction-level parallelism), provided the compiler schedules the instructions appropriately.

Although VLIW architectures achieve much higher performance than similar RISC architectures, their instruction sets are not backwards compatible between implementations. Also, memory accesses are not deterministic, making static scheduling by the compiler very difficult. Explicitly Parallel Instruction Computing (EPIC) architectures add on to the VLIW idea by providing explicit cache control mechanisms with extra cache pre-fetching instructions.

C.2.4 Instruction Set Extensions

Each of the above instruction set architecture approaches can be optimised and extended for application specific tasks. For example, applications requiring numerous checksum computations would benefit from a custom assembly instruction that supports this operation. Instruction set optimisation involves both removing unused and less frequently used instructions (instruction set reduction) as well as identifying groups of repeatedly executed operations, which are combined into optimised custom instructions.

A popular method of improving digital signal processing performance in general purpose computing architectures is by adding vectorisation extensions into the instruction set. Multiple vector operations can be executed concurrently with this architecture, by packing data words into large single-instruction-multiple-data (SIMD) registers.

Special loop control instructions are often added into the instruction set for applications requiring fixed iterations without the overhead of loop exit controls or counters. These loop control instructions as well as ring buffers or other specialised modulo addressing instructions also simplify software pipelining (or modulo scheduling), which further improves performance.

The instruction set can also be extended with custom arithmetic operations depending on the end application. Saturation arithmetic, bit-reversed addressing modes for FFT calculations, or operations such as multiply accumulate (MAC) instructions, complex multiplication and inverse approximation are included in various commercial processing architectures.

C.3 MICRO-ARCHITECTURE

The micro-architecture is more concerned with computer organisation, defining the method in which an instruction set architecture is implemented on a processor, and describing the various datapaths, data processing elements, and storage elements. The design methodology focuses on increasing execution speed by maximising the amount of useful instructions that can be executed in a single instruction cycle.

One method to achieve this is through instruction-level parallelism. Horizontal parallelism is utilized by having control over the multiple functional units in parallel, either by VLIW, microcoding, or superscalar mechanisms. In superscalar designs, instructions are still issued from a sequential instruction stream, but the CPU hardware checks for data dependencies between instructions dynamically and decides which instructions can be executed in parallel. Microcoding is a technique that allows the higher level assembly instructions to be split and dispatched to multiple functional units concurrently, adding an abstraction layer between instruction set architecture and the underlying functional units.

Vertical parallelism on the other hand is utilized through pipelining. Adding registers in the datapath of arithmetic operations and in the control-flow execution path allows the CPU to be clocked at much higher frequencies at the expense of additional latency. When the data and instruction stream is continuous, high throughputs can be achieved through such data or instruction pipelining techniques. In most general purpose processing applications however, a continues stream of data and instructions is not available due to dependencies, branching, memory access latencies and peripheral accesses amongst other factors, causing the pipeline to stall.

To overcome these pipeline stalls, out-of-order execution, register renaming, rotating register files, speculative execution and other similar techniques are often built into the micro-architecture. In the out-of-order execution paradigm, instructions without data dependencies are re-ordered and execute dynamically while waiting for other operations to complete or input data to become available. The effectiveness of out-of-order execution can be further improved by adding register renaming support circuitry, which can use different physical registers for independent sequential program flow sequences that would otherwise have used the same architectural registers. Speculative execution and branch prediction techniques preempt the instruction flow and start executing the predicted route while waiting for the branch condition to become available. When the prediction is incorrect, the calculated values are discarded and execution jumps to the correct branch location.

Pipeline stalls can also be preemptively avoided by the compiler with loop unrolling, software pipelining, and static scheduling around hazards and functional unit latencies. Different physical registers are used in loop unrolling to avoid dependencies in processors that do not have dynamic register renaming or rotating register file circuitry.

Another method to avoid wasting CPU cycles while waiting for dependencies is multi-threading. The temporal multi-threading methodology switches to another processing thread when execution of the current thread is stalled because of data or control-flow dependencies. Once the dependency is cleared, execution of the previous thread resumes. The context switching between threads does not incur any overhead as register files and state hardware is duplicated on such multi-threaded CPUs. Simultaneous multi-threading on the other hand allows multiple threads to execute simultaneously in the various pipeline stages and functional units, improving the overall efficiency of superscalar CPUs by exploiting thread level parallelism.

C.4 MEMORY ARCHITECTURE

The memory architecture of a processor plays an important role, as the primary memory is typically the bottleneck in most high performance applications [138]. Table C.1 summarises some of the most popular memory technologies.

Table C.1: Memory Technology Comparison

Memory Type	Access Time	Capacity	Volatile	Write Endurance	Price per GB
Flip Flops	< 1 ns	words	No	∞	High
SRAM	< 10 ns	MB	No	∞	High
DRAM	< 20 ns	GB	No	∞	Moderate
Flash	< 100 ns	GB	Yes	1M	Moderate
Magnetic disk	< 25 ms	TB	Yes	1.2M hrs	Low
Optical disk	< 500 ms	GB	Yes	1-10	Moderate

The need for memory hierarchies should be obvious, as a single non-volatile technology achieving low latency, high throughput, large capacity and unlimited write cycles is not currently available. As memory accesses usually exhibit both temporal and spatial locality, these caching memory hierarchies work reasonably well for general purpose processing applications. Flip flops and latches are used as internal working registers (storing single data values), while low latency volatile memory is typically used for cache memory (storing the current dataset that is being processed). The slightly slower DRAM then holds multiple and larger datasets and the various active programs. The non-volatile disk- or flash-based memory on the other hand, provides a vast amount of storage space for data and programs at much slower access speeds.

The traditional Harvard / Neumann memory architecture model is less frequently used nowadays, as it varies throughout the different levels of the memory hierarchy. For example, a certain processor might have separate program and data memory caches, but store both data and programs in non-volatile memory or DDR memory during runtime.

C.5 HARDWARE COPROCESSORS

Hardware coprocessors supplement the main CPU in form of a custom processor created for specific tasks, such as floating-point arithmetic, signal processing, fast Fourier transforms, graphic operations, encryption and decryption. Tasks are allocated to the coprocessor from the main processor to offload processor intensive or timing stringent tasks.

One method of connecting a coprocessor to the CPU is by mapping it directly into the memory space, and having the CPU move data and commands through the data bus. The data bus can however be hindered by bus arbitration and typically operates at a much slower speed than the processor clock frequency. To overcome these issues, a DMA controller is often added to allow the coprocessor to operate independently on blocks of data in a shared memory space.

Another method of connecting the coprocessor to the CPU is via dedicated I/O ports. I/O ports do not have arbitration and are usually clocked faster than the data bus. Data movement to and from the coprocessor thus achieves higher throughputs and lower latency than the processor bus interface.

Even faster connection speeds can be achieved by integrating the coprocessor directly into the datapath of the CPU. With this instruction pipeline connection, the coprocessor responds to special assembly instructions, executing them directly based on operands and updating the relevant status flags. Floating-point units and memory address calculation units fall into this category and blur the line between coprocessors and instruction set extensions.

C.6 MULTIPLE CORES / PARALLEL PROCESSOR

Most of the performance enhancement techniques discussed so far focused on exploiting bit, data and instruction-level parallelism in the horizontal as well as vertical dimension. A more coarse-grained approach to parallelism duplicates multiple processing cores on the same chip (thread or task-level parallelism) in a linear or tile based manner. On an even more macro scale, multiple of such chips are duplicated in a system as a cluster or grid of multi-cored machines (distributed computing or MPP).

Regardless on the scale at which multi-cored processing is implemented, additional complications are posed in the fields of scheduling, data dependencies, race conditions, synchronisation, shared memory arbitration, and inter process communication. These issues have been covered extensively in general purpose processing literature, and multiple frameworks for handling parallel processing workloads exist (e.g. OpenMP, Intel Thread Building Blocks (TBB), Message Passing Interface (MPI), POSIX Threads).

C.7 VECTOR / ARRAY PROCESSOR

A vector processor executes the same instruction on large data sets such as arrays or vectors. A single multiply instruction could thus multiply two arrays each consisting of e.g. 32 element vectors of double precision floating-point numbers (SIMD).

An example of a vector processor is the cell processor [139], consisting of one scalar processor and eight vector processors. The master processor controls and allocates tasks to each of the eight memory mapped coprocessors.

C.8 STREAM PROCESSOR

A stream processor applies the same operations to all elements of an incoming data stream. Practically it consists of a customised hardware pipeline (a fixed data flow graph) for a specific application, compromising flexibility for parallel performance. Applications such as media encoding and decoding, graphics rendering, digital filtering and image processing all exhibit data parallelism and locality and are thus well-suited for such a pipelined processing architecture.

Examples of stream processors include Stanford's Merrimac [140] and Imagine [141, 142] architectures as well as modern GPUs. Systolic arrays form a subsection of stream processors. The difference is that systolic arrays can exhibit non-linear array structures with data flowing in multiple directions, and often contain small RISC processors with their own local instruction and data memory rather than just mathematical or signal processing kernels.

C.9 DATAFLOW PROCESSOR ARCHITECTURES

A dataflow machine executes programs described by data flow graphs (DFG), rather than by a sequential instruction stream. Each operation in a DFG is represented as a node, while the data connection between nodes is represented by arcs. Data travels along the arcs in tokens (data packets). The execution of a node is governed by the availability of tokens on each of its inputs. When all of the required inputs are available, the node fires and consumes the input tokens, producing one or more output tokens. The execution of nodes is thus data dependent rather than control-flow dependent. When nodes produce and consume a fixed number of tokens per firing, the DFG is known as a Synchronous Data Flow (SDF) graph. Homogeneous Synchronous Data Flow (HSDF) graphs are a further subset of SDF graphs, which only consume and produce a single token per input [143]. Both SDF and HSDF graphs simplify the compiler architecture of the dataflow machine as they allow static scheduling.

Dataflow graphs simplify the construction of complex processing chains, as each node controls the flow of data to neighbouring nodes without any global control bottleneck [144]. Because of this data dependent processing nature, DFGs are useful for exposing parallelism in algorithms, graphically depicting the mathematical dependencies between processing steps. Applications exhibiting data parallelism or streaming operations are thus better described by DFGs rather than the traditional control-flow based methods.

Describing control-flow based applications with DFGs becomes a bit more cumbersome though, as additional mechanisms for branching, merging, function calls, recursion and loop control need to be added. Avoiding race conditions and deadlocks is then handled by attaching tags to tokens. These tags are used (amongst others) to distinguish between different loop iterations or contexts such that nodes will only fire when their inputs have matching tags.

Dataflow architectures are typically divided into static dataflow machines and dynamic dataflow machines, based on whether multiple instances of execution (reentrant subgraphs: e.g. loops, recursion, sub-program invocation) are permitted. Static dataflow machines use a simple model, in which tokens destined for the same node cannot be distinguished in context, and thus do not support modern programming constructs like procedure calls and recursion. Since only one token per arc is supported, an acknowledge signal is uses to signal that the next firing can occur. Unlike dynamic dataflow machines, this static dataflow model can use conventional program memory to store the data dependency and transformation tags.

Dynamic dataflow machines keep track of the different contexts by attaching them to the tokens in the form of context tags. Since tokens need to be matched in context, a content-addressable memory (CAM) is required. This sort of associative memory is not cost-effective and becomes impractical as the memory requirements to store tokens waiting for a match tend to be rather large. Instead of the associative memory, hashing techniques are used in the explicit token store dataflow architecture. In this architecture, a separate frame memory is used to group waiting tokens with the same tags until the node can fire. Regardless, the token matching logic turns out to be a huge bottleneck for dataflow machines in addition to the efficiency problems in broadcasting data and instruction tokens to multiple nodes.

C.10 NO INSTRUCTION SET COMPUTER

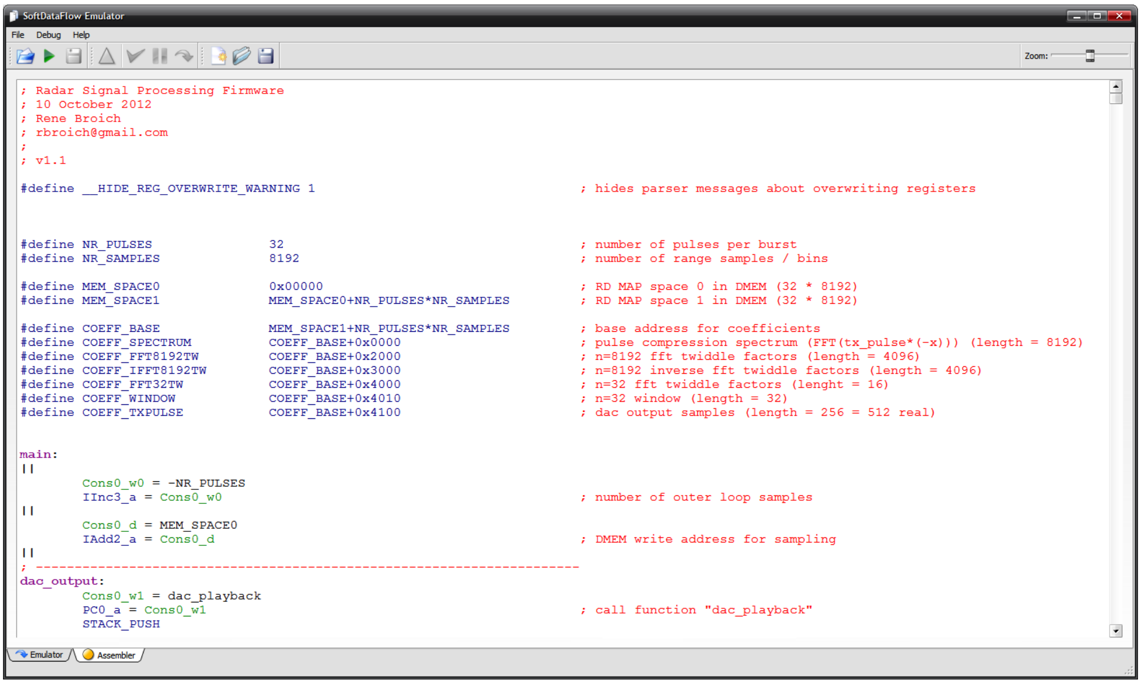
A No Instruction Set Computer (NISC) architecture does not have any predefined instruction set or microcode. Instead functional units, registers and multiplexers of a given datapath are directly controlled from a compiler generated instruction word (termed “nanocode”). Since the compiler has knowledge of all functional units and latencies in the datapath, the program flow can be statically scheduled with full parallel (horizontal and vertical) control over the functional units.

NISC architectures are common in high-level synthesis (HLS) or C to HDL compilers, as their datapaths are highly efficient and can be generated automatically, bridging the gap between custom processors and hardware descriptive languages.

APPENDIX D

SOFTWARE DEVELOPMENT ENVIRONMENT SCREENSHOTS

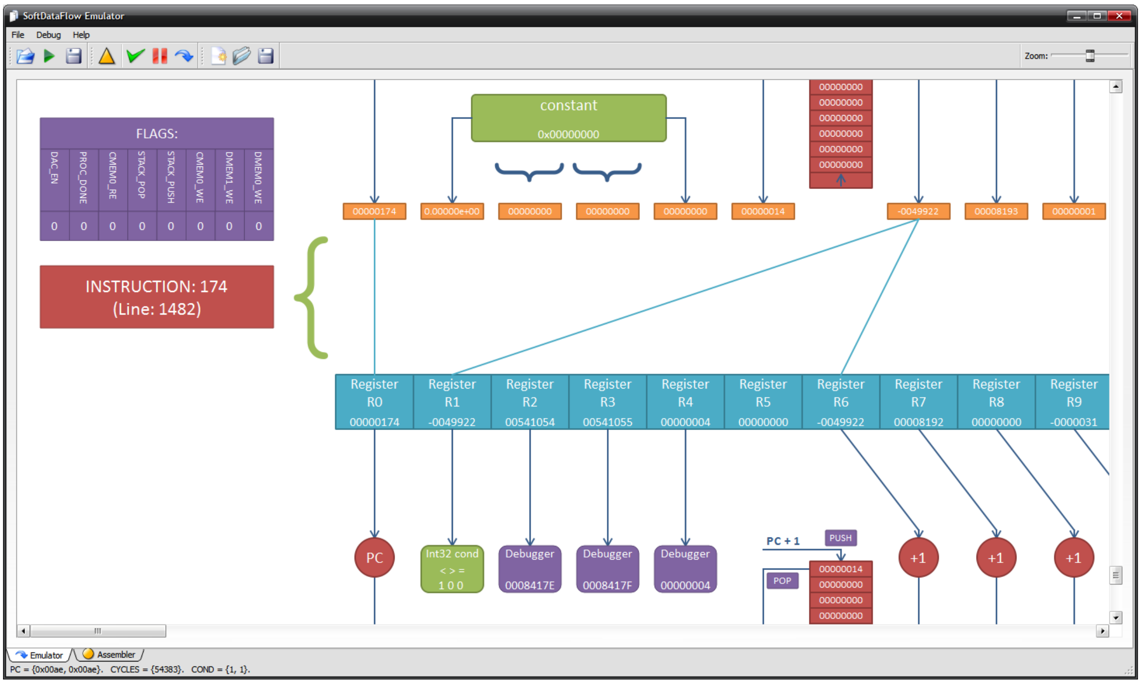
Fig. D.1 shows the programming environment for the *FLOW* language of the proposed architecture. Each instruction is essentially a move instruction that happens in parallel with any number of other move instructions. The instruction delimiter “||” starts a new parallel move portion.



```
; Radar Signal Processing Firmware
; 10 October 2012
; Rene Broich
; rbroich@gmail.com
;
; v1.1

#define __HIDE_REG_OVERWRITE_WARNING 1                           ; hides parser messages about overwriting registers



#define NR_PULSES               32                               ; number of pulses per burst
#define NR_SAMPLES              8192                             ; number of range samples / bins

#define MEM_SPACE0              0x00000                          ; RD MAP space 0 in DMEM (32 * 8192)
#define MEM_SPACE1              MEM_SPACE0+NR_PULSES*NR_SAMPLES  ; RD MAP space 1 in DMEM (32 * 8192)

#define COEFF_BASE              MEM_SPACE1+NR_PULSES*NR_SAMPLES  ; base address for coefficients
#define COEFF_SPECTRUM          COEFF_BASE+0x0000                ; pulse compression spectrum (FFT(tx_pulse*(-x))) (length = 8192)
#define COEFF_FFT8192TW         COEFF_BASE+0x2000                ; n=8192 fft twiddle factors (length = 4096)
#define COEFF_IFFT8192TW        COEFF_BASE+0x3000                ; n=8192 inverse fft twiddle factors (length = 4096)
#define COEFF_FFT32TW           COEFF_BASE+0x4000                ; n=32 fft twiddle factors (lenght = 16)
#define COEFF_WINDOW            COEFF_BASE+0x4010                ; n=32 window (length = 32)
#define COEFF_TXPULSE           COEFF_BASE+0x4100                ; dac output samples (length = 256 = 512 real)


main:
||
        Cons0_w0 = -NR_PULSES
        IInc3_a = Cons0_w0                                       ; number of outer loop samples
||
        Cons0_d = MEM_SPACE0
        IAdd2_a = Cons0_d                                        ; DMEM write address for sampling
||
; -----------------------------------------------------------------------
dac_output:
        Cons0_w1 = dac_playback
        PC0_a = Cons0_w1                                         ; call function "dac_playback"
        STACK_PUSH
```

Figure D.1: Screenshot of the Emulator depicting the *FLOW* Programming Window

The development environment features code auto-completion and syntax highlighting. In the debugging environment of Fig. D.2, a new instance of the architecture is shown for each instruction cycle. The lines connecting consecutive instructions correspond to the assignments in the *FLOW* program, and depict how data flows through the registers. The functional units with multi-cycle latencies have multiple orange boxes with the numeric results propagating through them every clock cycle.



PC = {0x00ae, 0x00ae}. CYCLES = {54383}. COND = {1, 1}.

Figure D.2: Screenshot of the Emulator depicting the Debugging Window

The final architecture as shown in the development environment is shown in Fig. D.3. The final architecture description file (which is used to generate the VHDL processor source files and the graphical representation in the simulator) is shown below:

```
; arch_descr_v31.arch
; order in which they are drawn
NR_INT32_INP_REG=51
NR_FLOAT_INP_REG=41
NR_INT32_OUT_REG=32
NR_FLOAT_OUT_REG=31

; funcname_nr<intIN<intOUT<floatIN<floatOUT

PC_0<0<1<<
CONS_0<<2,3,4<<1
SPACING<-400
RCON<1<<<
IDEBUG_0<2<<<
IDEBUG_1<3<<<
IDEBUG_2<4<<<
STACK_0<<5<<
IINC_0<6<6<<
IINC_1<7<7<<
IINC_2<8<8<<
IINC_3<9<9<<
IMAC_0<10,11,12<10<<
IBUF_0<13<11<<
IADD_0<13,14<12<<
ISUB_0<13,14<13<<
SPACING<50
```

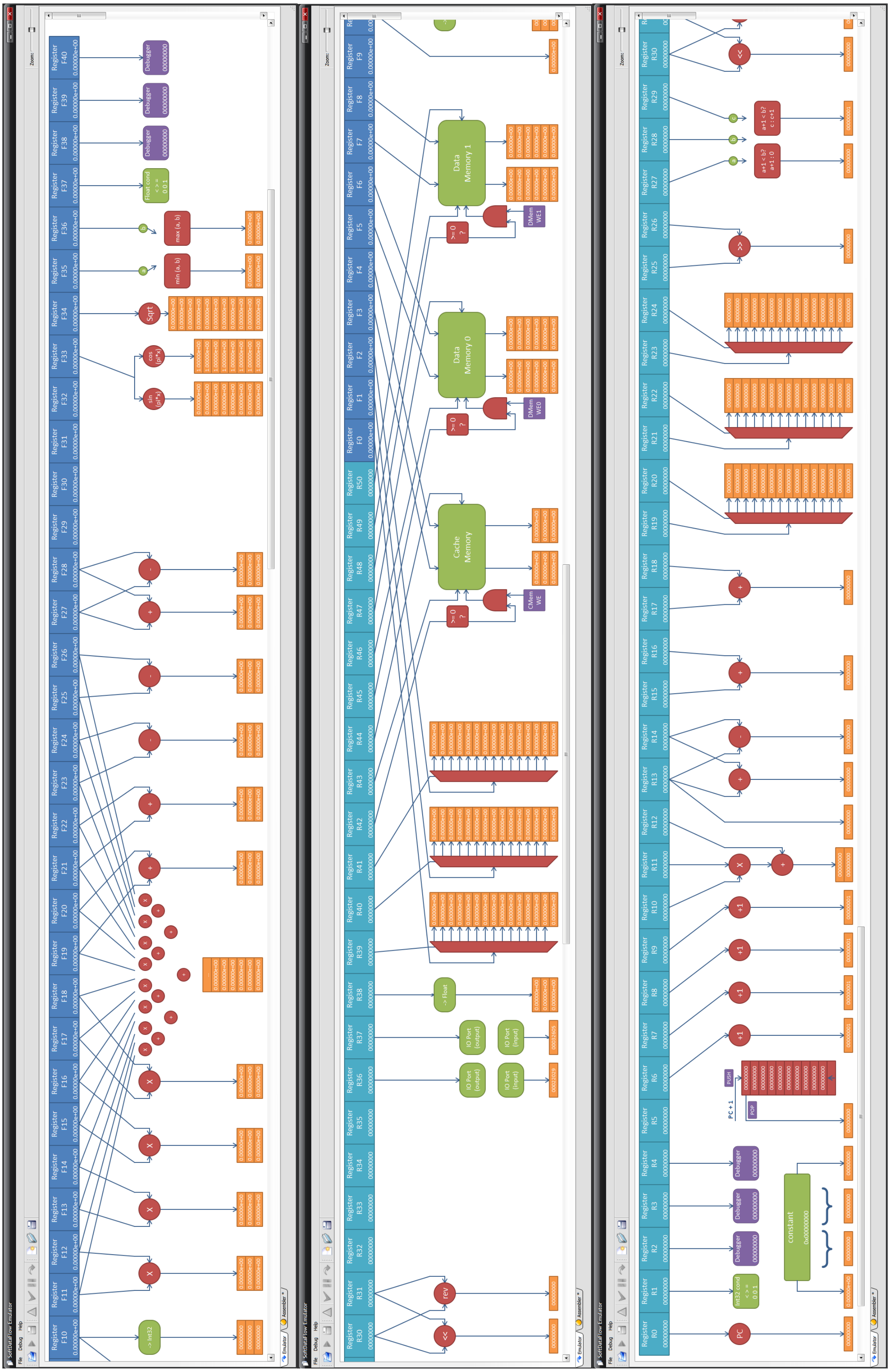


Figure D.3: Screenshot of the Emulator depicting the Architecture

```
IADD_1<15,16<14<<
SPACING<50
;ISUB_1<15,16<15<<
SPACING<50
IADD_2<17,18<15<<
SPACING<50
IDEL_0<19,20<16<<
IDEL_1<21,22<17<<
IDEL_2<23,24<18<<
;ROTL_1<25,26<19<<
SPACING<50
ROTR_0<25,26<20<<
SPACING<100
INCZ_0<27,28,29<21,22<<
SPACING<50
ROTL_0<30,31<23<<
IREV_0<30,31<24<<
;ROTR_1<30,31<25<<
SPACING<400
IIOPORT_0<36<26<<
IIOPORT_1<37<27<<
ITOF_0<38<<<2
FDEL_0<39<<0<3
FDEL_1<40<<1<4
FDEL_2<41<<2<5
SPACING<200
CMEM_0<43,44,42<<3,4<6,7
SPACING<100
DMEM_0<46,47,45<<5,6<8,9
SPACING<100
DMEM_1<49,50,48<<7,8<10,11
SPACING<50
FBUF_0<<<10<12
FTOI_0<<31<10<
SPACING<50
FMUL_0<<<11,12<13
SPACING<50
FMUL_1<<<13,14<14
SPACING<50
FMUL_2<<<15,16<15
SPACING<50
FMUL_3<<<17,18<16
FDOT_0<<<11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26<17
FADD_1<<<19,20<18
SPACING<50
FADD_2<<<21,22<19
SPACING<50
FSUB_1<<<23,24<20
SPACING<50
FSUB_2<<<25,26<21
SPACING<50
FADD_0<<<27,28<22
FSUB_0<<<27,28<23
SPACING<200
;FADD_3<<<29,30<24
SPACING<100
FSINCOS_0<<<33<25,26
FSQR_0<<<34<27
FSWAP_0<<<35,36<28,29
FCON<<<37<
;CUSTOM_SEL<<<<31
FDEBUG_0<<<38<
FDEBUG_1<<<39<
FDEBUG_2<<<40<
```

SPACING is a dummy functional unit that simply controls the horizontal gaps between functional units in the graphical representation for readability purposes.

APPENDIX E

APPLICATION SOURCE CODE

This section provides example source code of a few selected algorithms in the *FLOW* programming language. Note that the cycle count for some of the algorithms could still be reduced by a few clock cycles at the expense of generality and/or program memory size. For example, rather than dynamically assigning initial conditions for registers based on the function arguments, these could be precomputed by the compiler based on a few #defines. Similarly, the control-flow that generates address and write enable signals could be replaced by a predefined control sequence in program memory, thus avoiding the reinitialisation of the control-flow generation code between consecutive processing stages.

E.1 ENVELOPE CALCULATION

```
; ----------------------------------------------------------------------
; ----------------------------------------------------------------------
;       Envelope Processing (sqrt(x^2 + y^2)) in DMEM
;
;       arguments:      IAdd2_a = NR_WORDS_TO_PROCESS (N)
;                       IAdd1_a = WRITE ADDR_IN_DMEM
;                       IAdd0_a = READ ADDR_FROM_DMEM
;
;       cycle count:    N+28
;
;       macro definition
        IInc0_a = IInc0_o
        DMem0_raddr = IInc0_o   ; DMEM read address

        FMul0_a = DMem0_o1
        FMul0_b = DMem0_o1
        FMul1_a = DMem0_o2
        FMul1_b = DMem0_o2
        FAdd0_a = FMul0_o
        FAdd0_b = FMul1_o
        FSqr0_a = FAdd0_o
        DMem0_a1 = FSqr0_o

        IInc1_a = IInc1_o
        DMem0_waddr = IInc1_o   ; DMEM write address

        IInc2_a = IInc2_o
        DMem0_wi = IInc2_o      ; DMEM write inhibit
        DMEM0_WE = 1
{{ m_envelope

envelope:
        Cons0_w0 = 17
        IAdd2_b = Cons0_w0
||
        Cons0_w0 = -7-17        ; wr_inhibit delay
        IAdd1_b = Cons0_w0      ; IAdd1_o = ADDR_IN_DMEM - wr_inhibit
        DMem0_wi = Cons0_w0     ; wr_inhibit
        IInc2_a = Cons0_w0      ; wr_inhibit

        Cons0_w1 = -1
        IAdd0_b = Cons0_w1      ; IAdd0_o = ADDR_IN_DMEM - 1
        IAdd2_b = Cons0_w1      ; IAdd2_o = (NR_WORDS_TO_COPY+17) - 1
        IAdd2_a = IAdd2_o
||
        Cons0_f = 0.0
        DMem0_a2 = Cons0_f

        IInc0_a = IAdd0_o       ; read address for DMEM
        IInc1_a = IAdd1_o       ; write address for DMEM

        RCon0_a = IAdd2_o       ; setup loop initial condition register
        PC0_a = PC0_o           ; setup loop
||
        }} m_envelope

        IAdd2_a = IAdd2_o       ; NR_WORDS_TO_COPY--
        RCon0_a = IAdd2_o
        [RCon0 >= 0]            ; loop here while RCon0 >= 0
        PC0_a = PC0_o
||
        }} m_envelope   ; wait for cond pass before assigning PC0_a
||
        }} m_envelope
        PC0_a = Stack0_o        ; return to calling address
        STACK_POP
||
        }} m_envelope   ; next instr is still exec. after PC assignment
||
; ----------------------------------------------------------------------
```

E.2 FFT ALGORITHM

```
; ----------------------------------------------------------------------
; ----------------------------------------------------------------------
;       Inplace Fast Fourier Transform (FFT)
;
;       arguments:      IAdd1_a = ADDR_IN_DMEM
;                       RotR0_a = N (for N-point fft)
;                       ISub0_b = P (number of of consecutive FFTs)
;
;       cycle count:    (P*N/2+21)*log2(N)+8
;
;       macro definition
        RotR0_b = IInc1_o       ; m value
        RotL0_a = IncZ0_o       ; j
        RotL0_b = IInc1_o       ; j<<stage, twiddle address

        IncZ0_a = IncZ0_o       ; j = j+1 if (j+1<m), else 0
        IncZ0_c = IncZ0_p       ; r = r if (j+1<m), else r+1

        IMac0_a = IncZ0_p       ; r*(2*m) + j
        IMac0_c = IncZ0_o

        IAdd1_b = IMac0_o       ; addr_offset + r*(2*m) + j

        IAdd0_a = IAdd1_o       ; IAdd0_o = IAdd1 + m = xaddr1
        IBuf0_a = IAdd1_o       ; IBuf0_o = IAdd1 = xaddr0

        IDel0_a = IBuf0_o       ; xaddr0
        IDel1_a = IAdd0_o       ; xaddr1
        IDel2_a = RotL0_o       ; delay twiddle address

        CMem0_raddr = IDel2_o
        DMem0_raddr = IBuf0_o
        DMem1_raddr = IAdd0_o

        DMem0_waddr = IDel0_o
        DMem1_waddr = IDel1_o
        DMEM0_WE = 1
        DMEM1_WE = 1

        FAdd0_a = DMem0_o1      ; FAdd0o = a.re + b.re
        FAdd0_b = DMem1_o1      ;
        FAdd1_a = DMem0_o2      ; FAdd1o = a.im + b.im
        FAdd1_b = DMem1_o2      ;
        FSub0_a = DMem0_o1      ; FSub0o = a.re - b.re
        FSub0_b = DMem1_o1      ;
        FSub1_a = DMem0_o2      ; FSub1o = a.im - b.im
        FSub1_b = DMem1_o2      ;
        FDel0_a = FAdd0_o       ; Pipeline bubble for an.re=a.re+b.re
        FDel1_a = FAdd1_o       ; Pipeline bubble for an.im=a.im+b.im
        FMul0_a = FSub0_o
        FMul0_b = CMem0_o1      ; (a.re - b.re) * t.re
        FMul1_a = FSub1_o
        FMul1_b = CMem0_o2      ; (a.im - b.im) * t.im
        FSub2_a = FMul0_o
        FSub2_b = FMul1_o       ;bn.re=(a.re-b.re)*t.re-(a.im-b.im)*t.im
        FMul2_a = FSub0_o
        FMul2_b = CMem0_o2      ; (a.re - b.re) * t.im
        FMul3_a = FSub1_o
        FMul3_b = CMem0_o1      ; (a.im - b.im) * t.re
        FAdd2_a = FMul2_o
        FAdd2_b = FMul3_o       ;bn.im=(a.re-b.re)*t.im+(a.im-b.im)*t.re
        DMem0_a1 = FDel0_o      ; an.re
        DMem0_a2 = FDel1_o      ; an.im
        DMem1_a1 = FSub2_o      ; bn.re
        DMem1_a2 = FAdd2_o      ; bn.im

        IInc2_a = IInc2_o
        DMem0_wi = IInc2_o
        DMem1_wi = IInc2_o      ; write inhibit signals

        IDebug0_a = IBuf0_o
        IDebug1_a = IAdd0_o
        IDebug2_a = IDel2_o
        FDebug0_a = FDel0_o
        FDebug1_a = FSub2_o
        FDebug2_a = FAdd2_o
{{ m_fft_inplace

fft_inplace:
        Cons0_w0 = 1
        RotR0_b = Cons0_w0      ; RotR0_o = N >> 1, i.e. N/2
        Cons0_w1 = 0
        ISub0_a = Cons0_w1      ; ISub0_o = 0 - P
||
        IMac0_a = RotR0_o       ; N/2
        IMac0_b = ISub0_o       ; -P

        IBuf0_a = RotR0_o       ; save N/2 temporarily

        Cons0_w0 = -12
        IMac0_c = Cons0_w0      ; IMac0_o = N/2 * (-P) - 12

        Cons0_w1 = 0
        RotR0_b = Cons0_w1      ; RotR0_o = N >> 0, i.e. N
||
        IMac0_b = RotR0_o       ; N = initial m*2
        RotR0_a = IBuf0_o       ; N/2 = m start value
        IncZ0_b = IBuf0_o
        IAdd0_b = IBuf0_o

        Cons0_w0 = 13
        IDel0_d = Cons0_w0
        IDel1_d = Cons0_w0      ; delay xaddr for writing
        Cons0_w1 = 7
        IDel2_d = Cons0_w1      ; twiddle address delay
||
        IInc3_a = IMac0_o       ; -inner loop counter-1

        Cons0_w0 = -20
        IInc2_a = Cons0_w0      ; write inhibit
        DMem0_wi = Cons0_w0
        DMem1_wi = Cons0_w0     ; write inhibit signals

        Cons0_w1 = 4
        FDel0_d = Cons0_w1      ; Pipeline bubble for an.re and an.im
        FDel1_d = Cons0_w1
||
        IInc0_a = IInc3_o       ; assign inner loop counter

        Cons0_w0 = -1
        IncZ0_a = Cons0_w0
        IInc1_a = Cons0_w0      ; stage counter start at 0
        Cons0_w1 = 0
        IncZ0_c = Cons0_w1
||
fft_stage_loop:
        }} m_fft_inplace

        RCon0_a = IInc0_o       ; setup loop condition register
        PC0_a = PC0_o           ; setup loop
||
        }} m_fft_inplace

        [RCon0 < 0]             ; loop here while RCon0 >= 0
        RCon0_a = IInc0_o
        IInc0_a = IInc0_o       ; inner loop counter
        PC0_a = PC0_o
||
        }} m_fft_inplace
        IMac0_b = RotR0_o       ; save old m value = new m*2
        IInc1_a = IInc1_o       ; increase stage counter
||
        }} m_fft_inplace
||
        }} m_fft_inplace
        RCon0_a = RotR0_o       ; setup condition check

        IncZ0_b = RotR0_o       ; new m
        IAdd0_b = RotR0_o

        IInc0_a = IInc3_o       ; reassign assign inner loop counter
||
        }} m_fft_inplace
        [RCon0 > 0]             ; test cond: (N/2>>stage_cnt++)>0
||
        }} m_fft_inplace
        Cons0_w0 = fft_stage_loop
        PC0_a = Cons0_w0        ; loop while condition true, i.e.

        Cons0_w1 = -20
        IInc2_a = Cons0_w1      ; write inhibit
        DMem0_wi = Cons0_w1
        DMem1_wi = Cons0_w1     ; write inhibit signals
||
        Cons0_w0 = -1
        IncZ0_a = Cons0_w0
        Cons0_w1 = 0
        IncZ0_c = Cons0_w1
||
        PC0_a = Stack0_o        ; return to calling address
        STACK_POP
||
        ; next instr is still executed after PC0_a assignment
||
; ----------------------------------------------------------------------
```

E.3 DELAY

```
; ----------------------------------------------------------------------
; ----------------------------------------------------------------------
;       Delay N cycles
;
;       arguments:      ISub0_b = Delay
;
;       cycle count:    N+1
;

delay:
        Cons0_w0 = 8     ; account for call/return/loop setup overhead
        ISub0_a = Cons0_w0
||
        IInc0_a = ISub0_o
        RCon0_a = ISub0_o       ; setup loop condition register
        PC0_a = PC0_o           ; setup loop
||
        [RCon0 < 0]             ; loop here while RCon0 >= 0
        RCon0_a = IInc0_o
        IInc0_a = IInc0_o
        PC0_a = PC0_o
||
                                ; wait for cond to be true again
||
        PC0_a = Stack0_o        ; return to calling address
        STACK_POP
||
                                ; wait for return
||
; ----------------------------------------------------------------------
```