A scalable, portable, FPGA-based implementation of the Unscented Kalman Filter

Jeremy Soh BE (Hons 1)

A thesis submitted in fulfillment of the requirements of the degree of
Doctor of Philosophy

School of Aerospace, Mechanical and Mechatronic Engineering
Faculty of Engineering & Information Technologies
The University of Sydney

Submitted February 2017; revised October 2017

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the University or other institute of higher learning, except where due acknowledgement has been made in the text.

Jeremy Soh

17 October 2017

Abstract

Sustained technological progress has come to a point where robotic/autonomous systems may well soon become ubiquitous. In order for these systems to actually be useful, an increase in autonomous capability is necessary for aerospace, as well as other, applications. Greater aerospace autonomous capability means there is a need for high performance state estimation. However, the desire to reduce costs through simplified development processes and compact form factors can limit performance.

A hardware-based approach, such as using a Field Programmable Gate Array (FPGA), is common when high performance is required, but hardware approaches tend to have a more complicated development process when compared to traditional software approaches; greater development complexity, in turn, results in higher costs.

Leveraging the advantages of both hardware-based and software-based approaches, a hardware/software (HW/SW) codesign of the Unscented Kalman Filter (UKF), based on an FPGA, is presented. The UKF is split into an application-specific part, implemented in software to retain portability, and a non-application-specific part, implemented in hardware as a parameterisable IP core to increase performance. The codesign is split into three versions (Serial, Parallel and Pipeline) to provide flexibility when choosing the balance between resources and performance, allowing system designers to simplify the development process.

Simulation results demonstrating two possible implementations of the design, a nanosatellite application and a Simultaneous Localisation and Mapping (SLAM) application, are presented. These results validate the performance of the HW/SW UKF and demonstrate its portability, particularly in small aerospace systems. Implementation (synthesis, timing, power) details for a variety of situations are presented and analysed to demonstrate how the HW/SW codesign can be scaled for any application.

Acknowledgements

I would like to thank my supervisor, Dr. Xiaofeng Wu, for his guidance and countless insights over the course of this research.

I would also like to thank my friends and family for all their love and support in allowing me to continue studying for so long. I would like to thank, in particular, my sister Steph, for helping me with proofreading in the final hours.

Table of Contents

List of Figures

List of Tables

Abbreviations & Non-standard Terms

μC	Microcontroller
AC	Alternating Current
ADC	Analog-to-Digital Converter
ADCS	Attitude Determination & Control System
AES	Advanced Encryption Standard
ALU	Algorithmic Logic Unit
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
AXI4	Advanced eXtensible Interface 4
BRAM	Block RAM
BRIEF	Binary Robust Independent Elementary Features
COTS	Commercial-off-the-shelf
CRC	Cyclic Redundancy Check
DAC	Digital-to-Analog Converter
DMA	Direct Memory Access
DSP	Digital Signal Processor
DSP48	Xilinx DSP Primitive
EEG	Electroencephalogram
EKF	Extended Kalman Filter
FF	Flip-flop
FIFO	First-In/First-Out
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSL	GNU Scientific Library
hard-core	Design implemented with specialised hardware
HDL	Hardware Description Language
HPC	High Performance Computing
HW/SW	Hardware/software
I/O	Input/Output
IC	Integrated Circuit
IoT	Internet of Things
IMU	Inertial Measurement Unit
IP core	A single, modular HDL design unit
LUT	Lookup Table
many-core	A system with multiple or many processor cores
MEMS	Microelectromechanical System
MPPT	Maximum Power Point Tracking
NoC	Network-on-Chip
OBC	Onboard Computer
PCB	Printed Circuit Board
PE	Processing Element
PL	Programmable Logic
PMU	Performance Monitoring Unit
PS	Processor System

radhard	Radiation Hardened
RAM	Random Access Memory
RF	Radio Frequency
ROM	Read-Only Memory
RTL	Register Transfer Level
SDR	Software Defined Radio
SEE	Single Event Effect
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SIFT	Scale-Invariant Feature Transform
SLAM	Simultaneous Localisation And Mapping
Slice	Fundamental logic block (on the FPGA)
SoC	System-on-Chip
soft-core	Design specified as IP and implemented on programmable logic
SoPC	System-on-Programmable-Chip
SRAM	Static Random Access Memory
SVD	Singular Value Decomposition
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver Transmitter
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
UT	Unscented Transform
UKF	Unscented Kalman Filter
VLSI	Very Large Scale Integration

Nomenclature

k	Discrete time step
$\mathbf{x}_k$	State estimate vector
$\mathbf{z}_k$	Observation estimate vector
f	Process or `predict` system model
h	Observation or `update` system model
$\mathbf{u}_{k-1}$	Control input
$\mathbf{w}_{k-1}$	Process or control noise
$\mathbf{v}_{k-1}$	Observation or measurement noise
$\mathbf{Q}_k$	Process noise covariance
$\mathbf{R}_k$	Observation noise covariance
$\hat{\mathbf{x}}^-_{k\|k-1}$	*a priori* state estimate
$\mathbf{P}^-_{k\|k-1}$	*a priori* state covariance
$\mathbf{F}_k$	Process model Jacobian
$\hat{\mathbf{z}}_{k\|k-1}$	Predicted observation
$\mathbf{S}_{k\|k-1}$	Observation covariance
$\mathbf{H}_k$	Observation model Jacobian
$\mathbf{K}_k$	Kalman gain
$\tilde{\mathbf{z}}_k$	Observation
$\mathbf{I}$	Identity matrix
$\hat{\mathbf{x}}_k$	Current state estimate
$\mathbf{P}_k$	State covariance
$\mathbf{x}^a_k$	Augmented state vector
$\mathbf{P}^a_k$	Augmented covariance
$\hat{\mathbf{x}}^a_k$	Current augmented state estimate
M	Number of augmented state variables
M_{state}	Number of state variables
M_{obs}	Number of observation variables
$\boldsymbol{\mathcal{X}}_k$	Sigma points
N	Number of sigma points
W_0	Sigma point centre weighting coefficient
W_1	Sigma point spread weighting coefficient
$\boldsymbol{\mathcal{X}}^x_k$	State sigma points
$\boldsymbol{\mathcal{X}}^w_k$	Process noise sigma points
$\boldsymbol{\mathcal{X}}^v_k$	Observation noise sigma points
$\boldsymbol{\mathcal{X}}^x_{k\|k-1}$	Process model transformed sigma points
$\boldsymbol{\mathcal{Z}}_{k\|k-1}$	Observation model transformed sigma points
$\mathbf{P}_{xz,k\|k-1}$	Cross covariance
$\boldsymbol{\sigma}$	Spherical simplex sigma points weighting coefficient matrix
N_{PE}	Number of processing elements
$\backslash$	Matrix 'left' divide
$/$	Matrix 'right' divide
$\mathbf{L}_1$	Cholesky decomposition product
$\mathbf{L}_2$	LDL decomposition product
$\mathbf{D}$	LDL diagonal product
m, n, p	Matrix sizes
$\tilde{\boldsymbol{\mathcal{X}}}_k$	State sigma point residuals
$\tilde{\boldsymbol{\mathcal{X}}}_k$	Observation sigma point residuals
$\mathbf{F}$	Intermediate LDL decomposition product
P	Depth of the memory buffer
q	Unit quaternion
$\mathbf{q}$	Vector part of unit quaternion
$\mathbf{q}^\times$	Skew-symmetric matrix of $\mathbf{q}$
$\mathbf{z}_g$	Gyroscope measurement
$\boldsymbol{\omega}_T$	True angular velocity
$\boldsymbol{\beta}$	Gyroscopic bias
$\boldsymbol{\eta}_g$	Gyroscopic noise

$\boldsymbol{\eta}_d$	Gyroscopic bias drift
$\mathbf{z}_a$	Accelerometer measurement
$\mathbf{a}_T$	True acceleration vector
$\boldsymbol{\eta}_a$	Accelerometer noise
$\mathbf{z}_m$	Magnetometer measurement
$\mathbf{m}_T$	True magnetic vector
$\boldsymbol{\eta}_m$	Magnetometer noise
dt	Simulation time step
$A_q(\mathbf{q})$	Rotation matrix between the body frame and local frame
$\mathbf{v}$	Velocity in the world frame
$\mathbf{v}_R$	Velocity in the UAV frame
$\mathbf{u}_{xyz}$	Linear control inputs
$\mathbf{u}_{\psi\theta\phi}$	Angular control inputs
$\boldsymbol{\eta}_{u,xyz}$	Linear control input noise
$\boldsymbol{\eta}_{u,\psi\theta\phi}$	Angular control input noise
$\mathbf{L}_i$	Landmark representation
$x_i,\ y_i,\ z_i$	Co-ordinates of the UAV when the landmark was first seen
α	Azimuth to the landmark
β	Elevation to the landmark
ρ	Inverse depth to the landmark
$\mathbf{z}_c$	Pinhole camera measurement
$f_u,\ f_v$	Distance from the centre of aperture of the camera to the centre of the image plane
$x_P,\ y_P,\ z_P$	Co-ordinates of a point in the UAV frame
$\boldsymbol{\eta}_c$	Camera measurement noise
$\mathbf{p}$	Position of the UAV
$x_L,\ y_L,\ z_L$	Co-ordinates of the landmark in the world frame
$l_x,\ l_y,\ l_z$	Co-ordinates of the new landmark in the world frame
n	Number of landmarks

Chapter 1

Introduction

Throughout human history technological progress has, fundamentally, always been about efficiency. Overall human endeavour may have been about how to best shape the world and everything in it as we desire, but each singular advancement, from the simplest tool to the most complex of machines, has been about how to effect the same or greater outcome with the same or lesser human effort.

The last few centuries, starting with the Industrial Revolution, have been particularly fruitful as we have seen productivity gains unlike any time period before. The ultimate goal, of course, being systems or machines which require minimal or no human effort to operate yet still yield productivity gains, quality of life or other outcomes which we may enjoy; the logical conclusion of this vein of technological progress is the completely autonomous system, with some artificial intelligence, that does work for our benefit. Indeed now, in this Information Age, humanity sits on the cusp of a 'robotics revolution' where increasingly intelligent robotics and other autonomous systems may well soon become ubiquitous.

The push for greater autonomy is not without its challenges, however. Just as an individual must know their own capabilities in order to understand how best to affect their surrounding environment, a truly independent robotic or autonomous system must know *itself*. The system must be capable of internal or external (or preferably both) sensing such that the system may infer relevant information about itself in

relation to its environment. External or environmental sensors may, for example, take the form of accelerometers to detect the orientation of the system with respect to local gravity. Internal or system sensors may take the form of, for example, an angular position sensor to detect the orientation of a robotic limb with respect to the system's main body. In general, however, the system must make use of some set of one or more sensors (internal or external) and combine the information from each together to produce a singular estimate of the system's state. This process is important not only for autonomous guidance or navigation purposes, but may also be relevant to the application itself; for example, a fixed-position camera on a satellite may not be able to produce meaningful data unless it is known which way a certain face on the satellite is pointing.

However, there is another trend that further complicates matters, particularly in the aerospace field. For aerospace applications where size and weight are at a premium and overall costs are much higher than similar ground applications, the push is not only for increased autonomy but also increased miniaturisation. In these applications, both space (astronautic) and aeronautical, the primary concern is fuel efficiency as fuel is one of the major costs involved; smaller and lighter air-/space-craft means less fuel is needed thus reducing overall costs. For space applications, the costs are predominantly related to the fuel needed by the launch vehicle to insert the satellite into orbit; whereas, for aeronautical applications, the costs are mostly for the fuel necessary to stay airborne for the desired length of time.

In the case of space applications, miniaturising the satellite may mean that the launch costs are no longer prohibitively expensive and so the barrier to space access is reduced. In particular, nanosatellites (a categorisation of satellite sizes can be seen in Table 1.1) have gained popularity in recent years and have seen many successful missions; though many are still for educational or technology-demonstration purposes rather than specific scientific objectives, partially due to the limitation in attitude determination and control capabilities (Bouwmeester and Guo, 2010). The aggressive miniaturisation of the spacecraft structure and subsystems, however, not to mention the trend towards commercial-off-the-shelf (COTS) components, has meant cheaper,

less precise sensors have had to be used and complex algorithms that could potentially compensate for them tend to be infeasible due to limited computing power. Using multiple small satellites in some formation or constellation is one proposal to boost the competitiveness of small satellites, especially if the satellites can be mass produced – insofar as a satellite can be mass produced – which would keep overall costs down.

Satellite size	Mass (kg)
Large	> 1000
Medium	500 -1000
Mini	100 - 500
Micro	10 -100
Nano	1 - 10
Pico	< 1

Table 1.1: Classification of satellite types (Vladimirova and Wu, 2007).

In a similar manner for aeronautical applications, the explosion in the mass production of Unmanned Aerial Vehicles (UAVs) have made them extremely attractive in a number of consumer and research applications. Aggressive miniaturisation of these systems also drastically reduces costs compared to manned aircraft for certain applications as well as opening up new applications which may have been prohibitively expensive if using manned aircraft. This mass production of UAVs has lowered the cost to a point where UAVs acting in some formation or constellation could be attractive in, for example, surveying or mapping applications. Many UAV systems are still controlled remotely, however, requiring skilled pilots to remain close by 'in-the-loop'. For UAVs to continue to appeal in existing applications as well as generate new applications, a greater amount of autonomy is required which, of course, means better attitude determination and control despite restricted electrical power, computing power and physical space.

One approach to alleviate this computing issue has been to translate all the necessary functionality into hardware and use a hardware-based rather than software-based solution. Specialised hardware implemented as an Application Specific Integrated Circuit

(ASIC) has the potential to be much faster, more power efficient and save on physical space when compared to software on a general-purpose microprocessor or microcontroller. The downside is the obvious portability issues between applications which often translates into difficult, complex and, perhaps more importantly, expensive, development processes.

Specialised hardware can also be implemented on a Field Programmable Gate Array (FPGA) device. FPGAs, or rather FPGA-based implementations, tend to exhibit the benefits of a full ASIC implementation but the reconfigurable nature of these devices help with portability and simplify development somewhat. In particular, FPGAs are becoming increasingly popular in space applications mainly for two reasons: they allow the implementation of computationally intensive algorithms, and their reconfigurability allows them to recover from faults due to radiation or update the algorithms based on changing mission objectives. Hardware-based solutions, either ASIC or FPGA, also allow multiple functions to be implemented on a single chip, in a so-called System-on-Chip (SoC), which frees up valuable real estate within the autonomous system normally required by multiple processors. An extreme case of this is where *all* of the system's computing is implemented on one or more SoC devices; however, there are few examples of an actual aerospace implementation in this manner.

Thus implementing functionality as specialised hardware offers large performance gains over software implementations in terms of speed and power but also tends to greatly increase the length and complexity of development as well as limits reusability. While FPGAs certainly reduce the impact of these issues somewhat compared to ASICs, even FPGAs have longer and more complex development times when compared to the traditional software approaches.

When it comes to state estimation in particular, algorithms will generally contain a model to represent the specific autonomous system it has been implemented on. This means if these algorithms are implemented solely in hardware, the hardware will be application or system specific. This can restrict the reuse of designs between systems or applications as the algorithm's hardware may be difficult to adapt if certain

parameters change (e.g. the set of sensors or actuators); software implementations, however, can easily deal with such a change. While FPGA-based implementations are usually slightly more portable over ASIC designs, they still do not compare to the generality of software that features proper hardware abstraction.

This lack of portability is usually not a problem with large, expensive, satellites/aircraft which may only be produced in low numbers, but with the trend towards mass production and entirely COTS systems, a more portable approach to state estimation that attempts to leverage the performance gains of hardware but retain the portability and low development effort of software is needed.

1.1 Thesis motivations

Small (micro-, nano-, pico-) satellites and micro-UAVs are emerging technologies that have the potential to be of great academic and commercial use, but only if a balance can be found between two diametrically opposed forces that act on their design: the desire, and need, for high performance and the desire to reduce costs. High performance, especially in state estimation, is necessary for these technologies to be advantageous over traditional aerospace systems in relevant applications.

The desire to reduce the costs of these technologies has led to their miniaturisation and heavy use of COTS components so that some level of economy-of-scale may be achieved. Though both component and development costs can be reduced in this way, this approach, in turn, leads to a reduction in the resources available (e.g. electrical power, computing power and physical space) aboard those systems, impacting performance.

Specialised hardware, e.g. ASIC/FPGA-based systems, can achieve high performance, even for severely resource-constrained systems, but tends to increase the development complexity of these systems; in this way, using specialised hardware may reduce component costs and meet performance and miniaturisation requirements while development costs are typically increased. This issue is illustrated in Figure 1.1

which depicts the balance between development complexity and performance for different embedded systems; greater complexity during the development process means a greater investment in resources, personnel and time becomes necessary which leads to higher development costs.

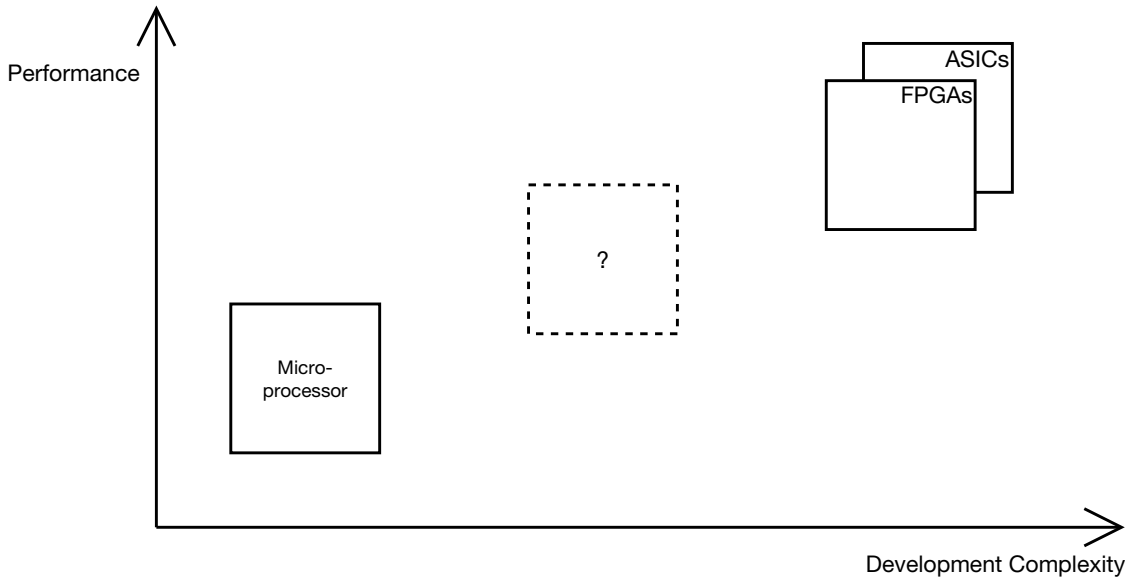


Figure 1.1: The performance versus development complexity trade-off for different types of embedded systems.

Software approaches, e.g. microprocessor-based systems, generally have lower performance compared to specialised hardware but have much simpler, and thus cheaper, development processes. It is, however, possible to draw upon aspects of both hardware and software approaches and combine them into a hardware/software codesign. This codesign could deliver the high performance of specialised hardware but, by using software techniques, e.g. modularity or abstraction, could also alleviate some of the high development costs associated with such hardware. If this codesign approach is applied to a prolific state estimation algorithm then the performance and miniaturisation requirements could be met while keeping development costs low.

1.2 Thesis overview

In this thesis, a library containing a scalable, portable, hardware/software (HW/SW) codesign of the Unscented Kalman Filter (UKF), based on an FPGA, is presented. One way software approaches keep development costs low is to create software libraries that can be reused between applications regardless of hardware changes; this thesis will outline an attempt to do the same: creating a generic, FPGA-based state estimation library that can be easily applied to any application.

In order to be as generic as possible, the algorithm implemented in this library must be widely applicable, and none are more widely used in state estimation than the Kalman Filter family of algorithms; the UKF, in particular, has recently gained popularity in many applications. Despite the popularity, very few examples of hardware UKFs exist and, as far as the author is aware, no examples of a device-independent HW/SW codesign of the UKF.

The codesign implements the application-specific parts of the UKF as software, which allows the library to be rapidly adapted to new applications thus incorporating the portability and ease of development of software approaches. The codesign then implements the non-application-specific parts of the UKF as hardware, accelerating many parts of the UKF algorithm, thus incorporating the greater performance of hardware approaches.

The codesign is implemented as a fully parameterisable, self-contained 'black-box' (IP core) which aims to minimise the necessary input from system designers when applying the codesign to a new application, such that overall development complexity is reduced. The library contains, and this thesis will describe, three separate designs each of which are useful in different situations providing maximum flexibility to system designers. The rest of this thesis is organised as follows:

Chapter 2 gives a brief background of the relevant fields. A short introduction to FPGAs and their development processes is given, followed by a broad set of examples of fields/applications where FPGAs are currently commonly used. The System-on-Chip concept is introduced then, for this context, the hardware/software codesign approach

is described. Finally, a short introduction to state estimation and, in particular, the Kalman Filter family of estimation algorithms is given; existing work on hardware implementations of Kalman filters is also described.

Chapter 3 describes the proposed hardware/software codesign of the UKF. The overall design methodology is outlined before three different variants of the codesign are introduced. First is the Serial design which deliberately does not embrace the main advantage of FPGAs, wide parallelism, in order to reduce resource usage to a bare minimum; the Serial design is intended for use in severely resource-constrained applications or incorporation into a SoC. The second variant, the Parallel design, does adopt the main advantage of FPGAs, wide parallelism, in the primary datapaths in order to increase performance; the Parallel design is intended for use as a coprocessor. Finally, the Pipeline design takes the parallelism even further by splitting the UKF into a five-stage pipeline for additional performance over the Parallel design; the Pipeline variant is intended for use as a standalone system.

Chapter 4 presents implementations of the HW/SW codesign in two example applications: attitude determination for nanosatellites and the state estimation part of a Simultaneous Localisation and Mapping (SLAM) system for a micro-UAV. The nanosatellite application is further segmented into two: attitude determination of a single nanosatellite and attitude determination for a small (5) constellation of nanosatellites. These example applications demonstrate the proof-of-concept for each of the three codesign variants.

Chapter 5 presents implementation details for a variety of example applications to demonstrate the flexibility of the library as a whole. The implementation details cover resource usage, power consumption and execution time of all three codesign variants which have been parameterised for those example applications. An analysis of how latency is impacted by various parameters is also given to demonstrate the scalability of the codesign.

Chapter 6 concludes this thesis by summarising the work presented in each chapter as well as the main contributions of this thesis, then discussing potential future work.

1.3 Summary of publications

Soh, J., & Wu, X. (2017a). A Five-Stage Pipeline Architecture of the Unscented Kalman Filter for System-on-Chip Applications. *IEEE Transactions on Industrial Electronics. PP*(99): 1-1. doi: 10.1109/TIE.2017.2740844

Soh, J., & Wu, X. (2017b). An FPGA-Based Unscented Kalman Filter for System-On-Chip Applications. *IEEE Transactions on Circuits and Systems II: Express Briefs, 64*(4):447-451. doi: 10.1109/TCSII.2016.2565730

Soh, J., & Wu, X. (2014). A Modular FPGA-based implementation of the Unscented Kalman Filter. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pages 127-134. doi: 10.1109/AHS.2014.6880168

Soh, J., & Wu, X. (2012). A FPGA-based approach to attitude determination for nanosatellites. In *Industrial Electronics and Applications (ICIEA), 2012 7th IEEE Conference on*, pages 1700-1704. doi: 10.1109/ICIEA.2012.6360999

Chapter 2

Background

This chapter presents background information and literature on relevant topics. An overview of Field Programmable Gate Arrays (FPGA) and its applications is given to demonstrate the FPGA's effectiveness as a development platform for increasing system performance. An overview of System-on-Chip (SoC) techniques, including hardware/software (HW/SW) codesign, is given to elaborate their advantages as a design methodology for reducing development complexity. A brief overview of state estimation and the Kalman Filter family of algorithms is given to establish the suitability of the Unscented Kalman Filter (UKF) for use in a generic state estimation library. Finally, a summary of the background information is given, justifying the proposed HW/SW codesign of the UKF.

2.1 Field Programmable Gate Arrays

An FPGA is an integrated circuit containing an array of programmable digital logic components, sometimes called logic cells or elements, and a hierarchy of interconnections, or programmable switches, that allow the components to be connected together in some fashion; a conceptual diagram of the FPGA structure is shown in Figure 2.1.

Each of the logic cells, contain basic circuit elements which can be used to implement the desired function. The exact composition of these cells, in general, differ depending

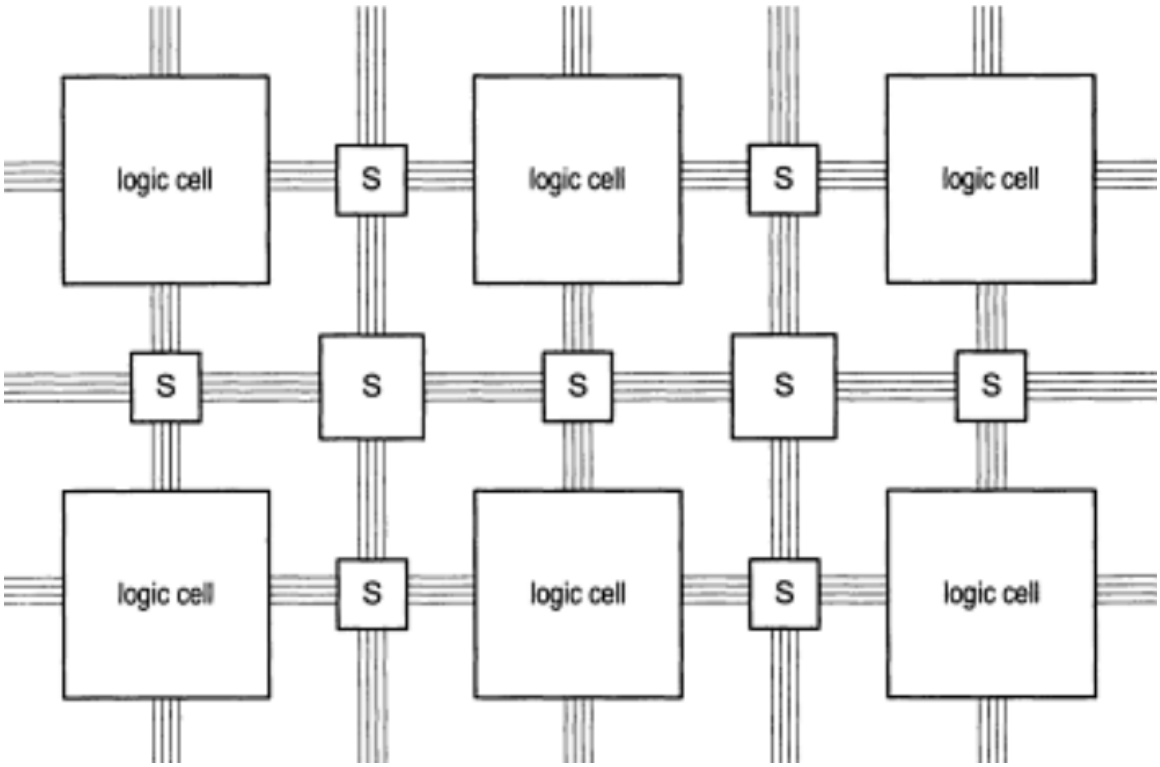


Figure 2.1: Structure of an FPGA device (Chu, 2008)

on the vendor, or even between device families by the same vendor, and can be loosely grouped as fine-grained or coarse-grained (Todman et al., 2005). Fine-grained components tend to include flip-flops (FFs) and/or n-input lookup tables (LUTs), while coarse-grained components tend to include Arithmetic Logic Units (ALUs) or function generators; additional possible elements for either include memory blocks, multipliers, carry logic or multiplexers among others.

FPGAs were originally used as prototype devices for Application Specific Integrated Circuits (ASIC), but increased sophistication in manufacturing techniques led to an increase in transistor, and thus resource, density; furthermore, an increase in the volume of production led to a reduction in cost which allowed the FPGA to replace ASICs in many applications. FPGAs can also be partially or completely reconfigured post-manufacture, unlike ASICs, hence the moniker 'field programmable'. The configurations available range from simple logic gates such as AND or XOR operations, to full blown processor units and their associated peripherals.

Functionally, digital logic is made up of combinational and sequential logic. Combinational logic is a circuit where the output is a function of the present input(s) only; as such, combinational logic may sometimes be considered as time-independent. Sequential logic is a circuit where the output is a function of the present input as well as past inputs; sequential logic can be considered to be combinational logic with memory. Furthermore, sequential logic can be divided into synchronous and asynchronous logic. Synchronous sequential logic *synchronises* changes in the output to a regular clock signal and all logic elements change their output at the same time. Asynchronous sequential logic changes their output directly whenever the inputs change; thus, ensuring the stability of the system can be much trickier with asynchronous logic, especially when input signals may arrive 'out of order' or in an unexpected order. For the purposes of this thesis, only synchronous sequential logic is considered and any reference to sequential logic should be regarded as synchronous.

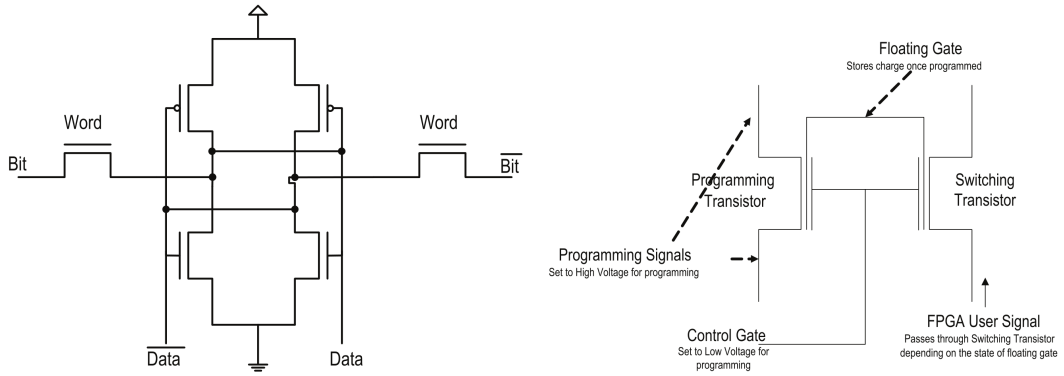
Each logic cell in an FPGA contains some combination of basic circuit elements that may be divided into active and passive components; the main passive component is the LUT and the main active component is the FF. Combinational logic is mostly implemented using passive components and sequential logic is mostly implemented using active components. Passive components such as LUTs do not draw power unless a signal is being propagated through them. This means designs that predominantly utilise passive components tend to draw very little power (beyond the power necessary to maintain the block's configuration). Active components such as FFs constantly generate and propagate new signals and so are always drawing power; elements like FFs also require clock signals to be distributed to them which in itself draws power. Designs that contain a large amount of active components will tend to use a large amount of power. Some active components may have additional control inputs to enable or disable the component; this can help save power when the component isn't being used. In general, designs will contain a mix of both types of logic elements, but in applications where it is desired to minimise the power consumption, a designer may prefer to use more passive components than active.

The maximum speed that a design can be run is a function of the propagation delay:

the time it takes for a signal to be propagated through the design. In the case of a purely combinational logic design, this is the time it takes for a signal at the input of the design to travel through the logic to the output. In the case of a design with synchronous sequential logic elements, this is either the time between the input and any sequential element, the time between any two sequential elements or the time between a sequential element and the output, whichever is longest. For designs using synchronous sequential elements, the propagation delay must be shorter than the time between clock pulses in order to guarantee stability of the design. If the propagation delay is longer than the time between pulses, elements in the design may become 'metastable' where an element cannot settle on a definitive state; the design behaves unpredictably at this point and usually fails. Increasing performance of a design is about shortening the propagation delay as much as possible, which also allows a potential increase in clock frequency. In order to do so, a designer may want to simplify any combinational logic so that the path through it is shorter, or if that is not feasible, insert additional sequential elements into the logic to break up long paths; thus, applications where high performance is desired may end up using proportionally more sequential elements like FFs. The potential downside to using additional sequential elements is that it increases the latency of the circuit. For a purely combinational design, the latency is just the propagation delay of the circuit. Because sequential elements are synchronised to the clock signal, successive sequential elements are 'delayed' with respect to preceding sequential elements when generating the desired output. For designs including sequential elements, the latency is the delay in clock cycles between the input and the output of the circuit. Each additional sequential element placed in succession adds an additional clock cycle to the latency. Although adding to the latency of the circuit means it takes longer to 'run', the reduction in the propagation delay may allow an increase in clock frequency which makes up for the additional latency.

2.1.1 FPGA technologies

There are three main technologies used to actually implement the programmable logic cells on FPGAs, all with their advantages and disadvantages: Static Random Access Memory (SRAM), Flash memory, and antifuses. Figure 2.2 depicts common implementations of SRAM and Flash cells. SRAM- and Flash-based FPGAs are implemented in the same way as SRAM and Flash memories. Antifuses are simply the opposites of fuses - rather than start at low resistance and then break ('blow') the connection with increasing current, antifuses start at high resistance and create ('burn-in') a connection with increasing current. Subsequently, antifuse devices cannot be reconfigured, a major draw point of FPGAs, but are obviously non-volatile and have a greater radiation tolerance than SRAM FPGAs.



(a) SRAM (b) Flash

Figure 2.2: Programmable logic technologies (Kuon et al., 2008)

SRAM-based FPGAs are the most popular type of FPGA[i] for multiple reasons. SRAM memories are an extremely common type of memory with widespread use, from ordinary computers to embedded systems, meaning the manufacturing techniques used are extremely mature; this leads to SRAM FPGAs enjoying high resource densities. As with their traditional use as memories, SRAM logic cells can be reprogrammed an indefinite number of times - a property obviously very useful for FPGAs. The main disadvantage of SRAM is their volatility - only while powered does the FPGA hold its configuration. This also means that additional circuitry and/or

[i] http://www.grandviewresearch.com/industry-analysis/fpga-market

memory is required to hold the configuration while the FPGA is unpowered, then load the configuration once it is powered; in some cases, load time may be non-trivial and can have an effect on the design of the system as a whole.

Antifuse FPGAs are actually now somewhat more popular than Flash-based FPGAs, particularly in telecommunications and automotive applications. This is largely due to their replacement of traditional ASICs in those applications. Flash-based FPGAs, as with Flash memories, are non-volatile. This means once programmed, Flash FPGAs can be instantly started up as is, without the need of some sort of external configuration circuitry. Compared to SRAM technologies, Flash technologies aren't as mature and tend to lag by several 'generations', meaning lower resource densities; this means Flash FPGAs cannot implement designs as complex as an SRAM FPGA of similar cost could. Another drawback is that Flash FPGAs cannot be reprogrammed indefinitely. One of the latest Flash FPGAs, the Actel ProASIC3 (Actel, 2012), for example, can be reconfigured a maximum of 500 times; however, for many applications this is sufficient.

2.2 FPGA development

FPGA development has a number of differences compared to software development. A typical software toolchain – for example, C – involves writing an application then compiling it down to assembly that is specific to the target processor. The assembly contains individual low-level instructions that the processor executes in a sequential sequence. The assembler translates the assembly into raw binary and a linker links together multiple source files or libraries to create one large binary file containing the entire application; a diagram of the C compilation process can be seen in Figure 2.3. The software development process involves largely thinking about how sequential instructions can produce the desired output and only at the end of the sequence is the output produced. For scientific computing, software development is relatively straightforward since mathematical algorithms tend to be sequential and recursive; this translates well to the von Neumann (or rather, the more modern Harvard/modified-Harvard) archi-

tecture of traditional microprocessors where instructions are executed sequentially.

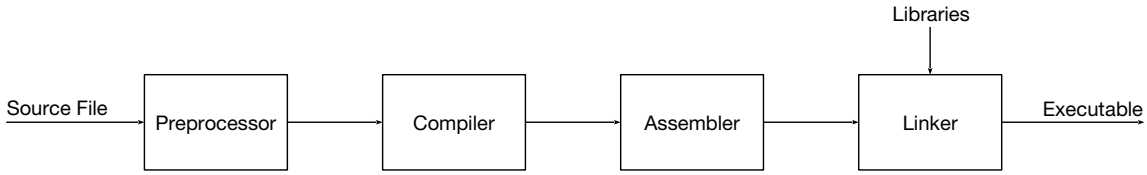


Figure 2.3: C software development process

FPGA development usually involves using a Hardware Description Language (HDL) to specify the design (Todman et al., 2005); although specifying the design as a set of schematics is also possible. More modern FPGA design tools have high-level synthesis (HLS) capabilities (e.g. Xilinx Vivado HLS[ii]) where the design may be specified in a high-level programming language, such as C++, and that high-level language is translated into hardware by the toolchain; the traditional approach, however, is to use an HDL.

In comparison to a set number of instructions that execute sequentially, hardware designs implement a series of low-level digital logic that is arbitrarily combined to produce the desired output. The process begins by detailing the design in an HDL (or schematic) then behaviourally/functionally simulating the design to ensure it produces the desired output. A functional design may then be synthesised by a synthesiser to produce a netlist of the digital logic components that function as originally described. The netlist is then translated and mapped onto the specific components featured by the target device. Finally, the design is placed onto the actual locations of the components in the target device and the routes between the components connected up. The configuration, including both the components and the routes, is stored as a bitstream which is downloaded into the configuration memory of the target device. Translation, mapping, placing and routing is commonly referred to as the implementation step; the full development process can be seen in Figure 2.4. Additional simulations, usually to determine timing performance, can be conducted post-synthesis as well as post-implementation.

[ii]https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, previously AutoPilot

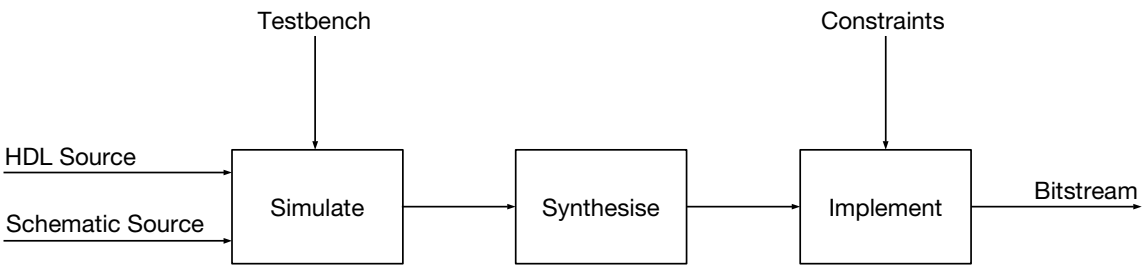


Figure 2.4: FPGA HDL development process

A HDL provides some abstraction from the low-level digital logic, but 'programming' an FPGA is more accurately described as designing a hardware architecture than programming. This is in contrast to software development where programs can be described at a high, functional level and advanced toolchains handle the precise implementation in the background (Bacon et al., 2013). High-level synthesis tools may be able to provide the same capabilities in the future but currently still lag behind software tools in maturity; a survey of HLS tools is given by Daoud et al. (2014).

Another difference from software programs is that hardware 'executes' all at once, i.e. different logic blocks all execute concurrently. Though there are some ways to produce a delay in time, in general, a hardware design is always 'on', producing some output. This parallelism is one of the greatest benefits of hardware designs (e.g. García et al., 2014; Lacey et al., 2016) but can also make it difficult to maintain data coherence if future outputs of a particular algorithm depend strongly on past outputs.

Optimised hardware designs also tend to be application-specific. While the FPGA device itself can be reconfigured for different applications, even modular HDL designs may have limited reusability; this is often an issue for designers, not only when considering new applications but also when considering new FPGA devices. However, the loss in flexibility is often offset by the boost in performance, whether that be in speed (e.g. Herbordt et al., 2007; Kumar, Joshi, et al., 2010; Brzoza-Woch and Nawrocki, 2016) or in reduced power consumption (e.g. Kestur et al., 2010; Hamada and Shibata, 2013).

The lack of portability, the low level of abstraction and concurrent execution leads to

significant design challenges even for experienced designers, let alone beginners that are considering whether to use an FPGA or a microprocessor for their next application. This design complexity is the reason why, despite FPGAs gaining popularity amongst some designers in many applications, they are still not as widely used as traditional general-purpose microprocessors (Jones et al., 2010; Bacon et al., 2013).

2.2.1 Number Representation

When working at the hardware level, it becomes necessary to consider how numerical values will be represented. Larger representations (in terms of total bits) increase the complexity and resource usage of the system, not only due to more complex arithmetic and memory, but also because communication channels (e.g. bus widths) have to be extended. The representation chosen will affect the dynamic range (the ratio of the largest and smallest possible values that can be represented) and the precision of variables (the smallest number that can be added to another and still be represented) used which will in turn affect the operation of the system, particularly arithmetic operations.

Digital logic operates in binary for which there are two main representations: fixed-point and floating-point. As the name suggests, fixed point representation uses a fixed number of bits to represent the fractional component of the number (see Figure 2.5); for a given word length, this then implies a fixed number of bits used to represent the integer part as well.

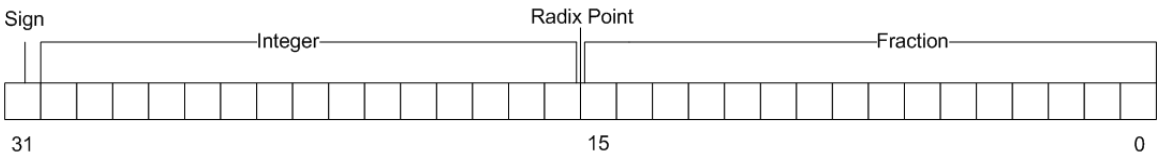


Figure 2.5: Binary fixed point representation of numbers (32-bit example)

Floating point representation instead breaks the number into two parts, the mantissa and exponent. The mantissa is a value between -1.0 and 1.0 which is then multiplied by two to the power of the exponent to give the desired value (see Figure 2.6). In

general, the size of the mantissa and exponent can vary with application or as the designer requires; however, the IEEE Standard for Floating Point Arithmetic (IEEE 754-2008) defines strict size limits for each. This standard is what will be adopted here and further discussion will assume mantissa and exponent sizes as summarised in Table 2.1.

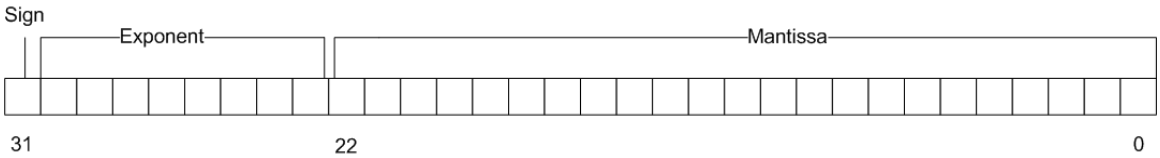


Figure 2.6: Binary floating point representation of numbers (32-bit example)

Total size (bits)	Name	Mantissa (bits)	Exponent (bits)
16	Half precision	10	5
32	Single precision	23	8
64	Double precision	53	11

Table 2.1: Floating point representation as defined by IEEE 754-2008

Floating point arithmetic is far more complex than fixed point arithmetic making it far slower and more expensive to implement (in terms of size/area of circuitry and overall complexity of circuitry) for a given word size. Fixed point representation, while easier and faster to implement, lacks the dynamic range and precision that floating point representations can offer. A summary of the dynamic range and most precise values that can be represented by either scheme is given in Table 2.2. The smallest value, or how precise a number can be, is of particular importance as it will have the most impact on system operations depending on the data type chosen.

Low precision variables set a limit on how small a change the system can make and still be registered which limits the overall algorithmic precision of the system. Low precision variables may also introduce rounding/truncation errors which then compound to limit the overall accuracy of the system.

	Dynamic Range (dB)		Precision	
Bits	Fixed point	Floating point	Fixed point	Floating point
16	48	175	3.9×10^{-3}	9.8×10^{-4}
32	96	1523	1.5×10^{-5}	1.2×10^{-7}
64	193	12312	2.3×10^{-10}	2.2×10^{-16}

Table 2.2: Dynamic range and most precise values fixed and floating point data types can represent, for a given word length. Fixed point representation assumes half the word size is used to represent the fractional component. For floating point representation, the precision varies depending on the size of the numbers, so listed here is the smallest number that can be added to 1 and still be represented.

2.3 Applications of FPGAs

FPGAs were originally used as prototyping devices for ASIC designs due to their reconfigurability and it is this benefit that the early uses of an FPGA as a standalone device sought to exploit. One of the earliest applications was artificial neural networks (ANNs) (Zhu and Sutton, 2003) where the dynamic reconfigurability allowed for rapid prototyping of ANN designs but, perhaps more importantly, also allowed for adaptive topologies which, in turn, allowed the ANN to 'evolve'; early implementations had limited practicality due to the inherent complexity of ANNs and size constraints of FPGAs at the time, but work continues on more recent, higher density devices with greater success (e.g. Gomperts et al., 2011; Qiu et al., 2016).

Another early application making use of this feature was using FPGAs to implement reconfigurable softcore processors (e.g. Davidson, 1993; Wittig and Chow, 1996; Zheng et al., 2001); 'soft' or 'softcore' here simply refers to the fact that the processors are implemented on (reconfigurable) FPGA fabric as opposed to 'hard' or 'hardcore' processors which refers to processors that are implemented as their own specialised/customised hardware in an IC i.e. traditional ASIC designs. These softcore processors functioned similar, if not identically, to traditional hardcore microprocessors but while the processor core was usually left unchanged, the peripherals (I/O such as UART, variable GPIO, interrupt controllers, ALUs etc.) could be reconfigured at will depending on changing environment or changing device/mission objectives.

In addition to more flexible embedded systems, softcore processors allow for easier development since development can continue in software and potentially reuse modules from previous/related hardcore processor implementations of the system. These days even FPGA vendors offer softcore processors for inclusion on their devices if desired (optimised for their FPGAs of course, e.g. Xilinx's Microblaze core[iii]), though other vendors providing more general softcore processors do exist as well (e.g. Aeroflex Gaisler's LEON series[iv]).

The rise in resource densities and more efficient routing, which increased timing performance, led to the use of FPGAs in some applications, not so much for their reconfigurability, but for their ability to outperform (in terms of speed and/or power) traditional microprocessor implementations. This was due to the fact that hardware approaches allowed for massive parallelism that was beneficial to some algorithms. Parallelism meant that, although FPGAs have had to use clock frequencies an order of magnitude or more slower than what microprocessors use, the overall throughput, in some cases, was still higher. An application that took early advantage of this property, and still does, was that of control (e.g. Jung et al., 1999; Krach et al., 2003; Kim, 2000; Monmasson and Cirstea, 2007; Rossi et al., 2011; Chekired et al., 2014; Hartley et al., 2014). FPGAs were of particular use to control systems thanks to the ability, by virtue of being hardware, to perform computations in a set period of time that did not change which is extremely important for reliable control; development was simplified somewhat as well since specialised real-time operating systems used on microprocessor systems were no longer necessary.

Similar motivations made FPGAs popular for cryptography; at least the part involving the actual process of encryption/decryption. The benefits of rapid encryption/decryption for communication are obvious and the release of the Advanced Encryption Standard (AES) saw a flurry of activity in FPGA development (e.g. Hodjat and Verbauwhede, 2004; Chodowiec and Gaj, 2003; Saggese et al., 2003); development of optimised designs for even greater performance still continues today (e.g. Dyken

[iii] `http://www.xilinx.com/tools/microblaze.htm`

[iv] `http://www.gaisler.com/index.php/products/processors`

and Delgado-Frias, 2010; Hoang and Nguyen, 2012) as well as on other cryptography techniques (e.g. Aysu et al., 2013; Azarderakhsh and Reyhani-Masoleh, 2015).

Parallelism and an increasing amount of specialised signal processing slices made FPGAs attractive for high speed digital communication systems, particularly for modulation/demodulation. This field was, perhaps, one of the first to truly combine both the FPGA's strengths - performance and reconfigurability - with its use of FPGAs for Software Defined Radio (SDR) (e.g. Cummings and Haruyama, 1999; Zhigang et al., 2003; Ye et al., 2007; Amiri et al., 2011; Wu et al., 2017; Maheshwarappa et al., 2017). The explosion in wireless communication schemes spanning wide frequency bands and many modulation/encoding schemes led to issues with generality: the hardware (both analog and digital) used in receivers would usually only work with a single frequency or a small band of frequencies around some centre as well as one or a small number of related modulation methods; if another frequency or modulation scheme was desired, new hardware was needed at potentially great cost. SDR techniques moved modulation and many other receiver functions to software, thereby reducing the hardware needed for the RF frontend as much as possible. FPGAs made it possible to have high-speed signal processing (largely due to parallelised and heavily optimised arithmetic datapaths) necessary for high data rates, but still had the ability to reconfigure the processing 'hardware' to suit another frequency or modulation scheme allowing greater use in a variety of applications.

More recently, FPGAs have seen use in the field of High Performance Computing (HPC). HPC refers to using extremely large computing clusters to perform large-scale calculations or simulations of some kind; a common application is weather modelling. Here, FPGAs are used to implement particularly time intensive parts of the algorithms or models they are running, thus gaining significant speed-ups over pure microprocessor implementations (e.g. El-Ghazawi et al., 2008; Dimond et al., 2011). FPGAs are also being used as part of heterogeneous 'many-core' processors that use different types of hardware – e.g. microprocessors, graphics processing units (GPU), ASICs, FPGAs, DSPs etc. – all together to implement extremely complex and/or large-scale algorithms (e.g. Stratikopoulos et al., 2014; Chen and Prasanna, 2016).

The algorithm is broken into parts and implemented on the type of hardware that would benefit that part most; an introduction on how these algorithms are mapped onto many-core systems is given by Singh et al. (2013).

In some cases, moving the entire algorithm to the FPGA for processing, rather than just parts of it, is beneficial due to overhead incurred in moving data on and off the FPGA; this is mainly for highly repetitive algorithms or algorithms that require storing a lot of intermediate states. This approach is also beneficial for embedded systems that require using complex algorithms that would otherwise be unable to be implemented on traditional microprocessors. Some examples include: solving the least squares problem (Yang, Peterson, et al., 2009); Singular Value Decomposition (SVD) for, among other things, SDR (Wang, Cunningham, et al., 2010; Ledesma-Carrillo et al., 2011); image processing in vision-based control systems (Honegger et al., 2014); signal processing for a human sensing radar application (Wang, Liu, et al., 2013).

FPGAs have also played a role in advancing the emerging field of evolvable hardware; in this field, the FPGA is used to implement an algorithm that can self-update and then self-reconfigure (Lambert et al., 2009). There are numerous problems to be solved before true self-reconfiguring hardware can be developed (Haddow and Tyrrell, 2011), but more recent work in the field is encouraging (e.g. Salvador et al., 2013; López et al., 2014).

Nowadays, usage of FPGAs can be to instantiate customised, direct hardware implementations (IP cores) of the section of the algorithm they are tasked to run or instantiate softcore processors including a set of standard peripherals, custom peripherals or custom hardware as the application requires; part, or all, of the FPGA can be reconfigured at will. For some applications, the FPGA can be used as a co-processor, supplementing a traditional microprocessor instead. These various levels of coupling between a hardcore processor and an FPGA are illustrated in Figure 2.7.

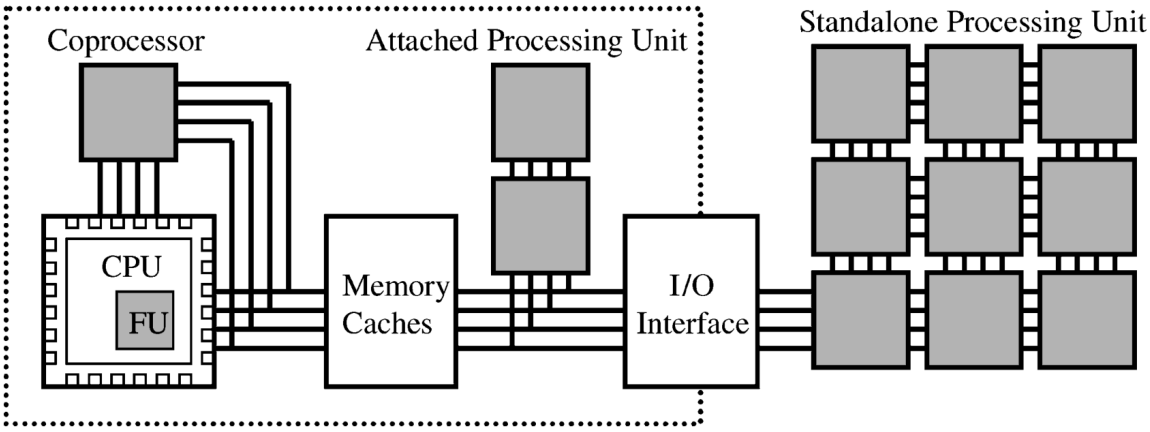


Figure 2.7: Common coupling schemes for FPGAs (Shaded grey boxes) (Compton and Hauck, 2002)

2.3.1 Space Applications

FPGAs have not seen nearly as much use in space (astronautic) applications as in terrestrial applications, mostly because of the development complexity, but recently, interest has been increasing for much the same reasons as their popularity in select terrestrial uses: performance and reconfigurability. For space applications, a major limiting factor for all electronics in operation is the effects of radiation. Though radiation-hardened (radhard) devices do exist, for both FPGAs and traditional microprocessors, these are typically more expensive and can be an entire generation or more behind non-radhard devices, which limits their performance.

FPGAs have been, and still are, useful in payload data processing due to their ability to accelerate calculation of complex algorithms and potentially reduce power consumption. This is important because, as higher fidelity instruments/sensors are launched and the associated data volume increases, online processing or compression reduces the information that needs to be sent back to Earth; power, fault tolerance and orbital constraints are severe limiting factors in how much data can be transferred.

FPGA reconfigurability allows mission objectives to be more flexible based on changing conditions than when using traditional microprocessors, especially given the inherent difficulty with accessing the satellite after it has been launched. The reconfig-

urability can also be used as a fault tolerance technique to simply 'refresh' parts of, or the whole, FPGA to deal with radiation faults. Although space is an extremely harsh environment, the savings in area, mass and power (i.e. performance gains) garnered by FPGAs as well as the desire to create true adaptive/autonomous systems continues to make them an extremely attractive research area (Fossati and Ilstad, 2011).

In line with these motivations, one of the first FPGAs to be used in a satellite was a payload data processing module (HPC-I) onboard the Australian FedSat (Fraser et al., 2000). Although not the first FPGA flown in space, the HPC-I module represented the first intentional application of an FPGA as a proper reconfigurable computing system (Bergsman, 2003). Apart from a technology demonstration of the reconfigurability aspects, the HPC-I was designed to implement a series of image processing (filtering, compression etc.) algorithms for online payload data processing that could be changed as necessary (Dawood et al., 2002; Williams et al., 2002; Visser et al., 2002).

To study the effects of radiation further, FPGAs themselves have flown, albeit radhard versions, in order to characterise the faults caused; one example is the CFESat (Caffrey et al., 2009). In addition to the radiation study, the CFESat's FPGAs were also used for the usual data processing purposes, transforming data from high-throughput sensors to more manageable packets of information for transmission back to Earth.

There is scant literature on the use of an FPGA as the main onboard computer (OBC); although it has been proposed before in principle (Zheng et al., 2001). One of the only, if not the only, planned missions actually using this method is the Flying Laptop (Grillmayer et al., 2005; Huber et al., 2007; Kuwahara et al., 2009; Fritz et al., 2015), which is yet to launch as of 2017. The Flying Laptop will use four redundant FPGAs, implementing identical hardware to perform all necessary computing with a Flash voter/command FPGA for fault mitigation. Other current research for planned or 'theoretical' missions has mainly focused on data processing (e.g. Sharma, Kulkarn, et al., 2010; Fiethe et al., 2012; Hopson et al., 2012) or fault mitigation (e.g. Iturbe et al., 2011; Dumitriu et al., 2012; Siozios and Soudris, 2012).

2.4 System-on-Chip

The term System-on-Chip (SoC) comes from the field of Very Large Scale Integration (VLSI) where individual hardware units or 'black boxes' (IP cores) that perform some dedicated function are arranged and connected together on a single ASIC chip. Typical SoCs may include a microcontroller or microprocessor core, DSPs, memories such as RAMs or ROMs, peripherals such as timers/counters, communication interfaces such as USB/UART, analog interfaces such as ADCs/DACs or other analog, digital, mixed-signal or RF hardware. Previously, each of these components may have had their own ASIC and were connected together on a PCB but, in accordance with Moore's Law, resource densities of silicon chips have massively increased over time so now these components are able to be integrated together on a single chip; an example SoC can be seen in Figure 2.8.

As SoC usually refers to VLSI structures, the same approach used on an FPGA is sometimes also referred to as System-on-Programmable-Chip (SoPC) to avoid confusion; here, SoC will be used in reference to the *methodology* of integrating multiple subsystems together on a single piece of hardware, regardless of the type of hardware. Indeed the SoC *methodology* extends even to other non-electronic types of hardware: recently, biological chambers have been added alongside sensing electronics for use in the fields of microbiology (e.g. Ghallab and Ismail, 2014 gives a survey of 'lab-on-a-chip' systems) and medicine – e.g. a bio-sensor by Yang, Xie, et al., 2014 for Internet of Things (IoT) enabled healthcare; photonic circuits have also been incorporated into electronic systems, largely for optical communications (e.g. see a survey by Kish et al., 2018).

SoC designs for FPGAs have become more popular recently as the increase in resource densities allowed more complex logic to be implemented. The push towards SoCs on FPGAs is driven by the desire for greater autonomy in a variety of systems; the most obvious example is field robotics, but autonomous systems such as 'smart' rooms and satellites also have a need for small form factor and high performance computing solutions that the FPGA is well placed to deliver.

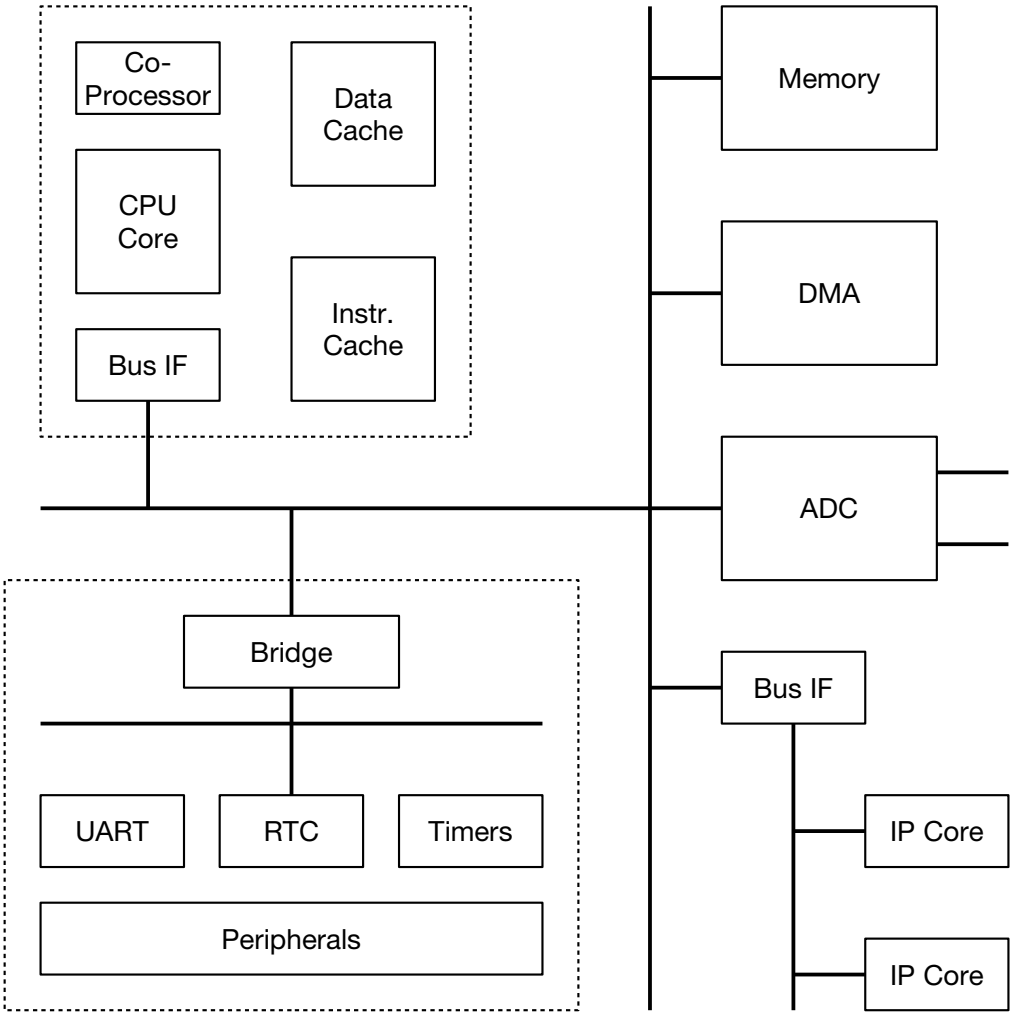


Figure 2.8: Example of a typical System-on-Chip

Computer vision is a growing area of research that exemplifies this need. Image processing algorithms are among the more computationally intensive algorithms but many applications of computer vision (e.g. mobile robotics, field surveillance, assembly line inspection etc.) require the computer system to be small and unobtrusive. It is unsurprising then that computer vision researchers are among the more prominent users of FPGA SoCs; examples include: standard encoding (Lehtoranta et al., 2005), compression (Tumeo et al., 2007), facial recognition (He et al., 2009; Al-Mahmood and Agyeman, 2017), plain object recognition ('Object Recognition on a Chip' by Schaeferling and Kiefer, 2011), object recognition *and* tracking (Kowalczyk and Kry-

jak, 2017 include pan-tilt actuation of the camera as well) or feature detection and matching (e.g. a SIFT & BRIEF implementation by Wang, Zhong, et al., 2014). In many of these examples, the FPGA is actually used as a coprocessor to provide hardware acceleration, outputting the data to a processor or communication interface; because the designs are specified as IP cores, however, integration into an even larger scale SoC is possible in the future.

Apart from computer vision, FPGA-based SoCs have flourished in many of the same areas that FPGAs originally were valuable, for example: ANNs (Biradar et al., 2015), HPC (e.g. Baklouti et al., 2015 propose a general single-instruction-multiple-data (SIMD) many-core processor with potential application in video processing), cryptography (Bossuet et al., 2013 gives a review and explains how crypto-engines need flexibility/reconfigurability to handle new types of attacks but also need to be fast and unobtrusive to their parent application), RF processing (e.g. Pang et al., 2014), control systems (e.g. Zhong et al., 2016; Guo, Pan, et al., 2017) or high-performance signal processing (e.g. Flesch et al., 2017 uses a hardware/software codesign SoC for Earth and planetary spectrography).

2.4.1 Intra-chip communication

Communication between modules in a SoC is a huge area of research on its own, especially as SoC designs integrate more and more functionality; with greater functionality, efficiently moving data around to where it is needed becomes harder too. Communication may be facilitated via a simple bus if there aren't many modules, especially if it's unlikely all modules will need to communicate with each other - e.g. in a single master situation, only the main computer will need to communicate with the other modules. A common bus used in SoC designs is the AXI4 interface specified by ARM (ARM, 2013); however, this specification is proprietary. An example of an open-source, free (as in 'free' software), bus specification is the Wishbone interface (Sharma and Kumar, 2012). Bus techniques begin to suffer performance issues in designs where there are many peripherals that each need to be accessed in a timely

manner and when multiple master devices want to use the bus; since bus control is locked to a single bus master at a time, arbitration between masters is required, which is non-trivial for larger numbers of masters.

There are also methods that borrow techniques from the field of computer networking such as circuit-switching (e.g. Wiklund and Liu, 2003) or network-on-chip (NoC) (e.g. Kumar, Jantsch, et al., 2002; Prasad et al., 2016) techniques. Circuit-switched SoCs are analogous in functionality to the old telephone system switchboards with a centralised switch to facilitate communication between peripherals being used. Though circuit-switched networks may sometimes outperform bus networks, they suffer many of the same drawbacks: when there are a large number of peripherals that need to be connected and/or multiple master devices, the 'switchboard' can become unwieldy and slow. NoCs are analogous in functionality to packet-switched computer networks (like the Internet) and so have the same advantages and disadvantages. NoCs allow many peripherals to communicate at once and easily scale with larger numbers of peripherals, but, in general, do not guarantee arrival of packets and the routers that facilitate traffic are quite complex themselves (which, of course, can affect performance of the SoCs 'actual' functionality).

2.4.2 Partial Runtime Reconfiguration

SRAM FPGAs have an additional capability not present in Flash FPGAs: the ability to reconfigure one section, or partition, at a time without affecting the operation of the rest of the FPGA. Here, the FPGA must be divided into a single static region, that obviously does not change, and one or more reconfigurable regions, which may be reconfigured at will. This capability expands the flexibility of the FPGA even further but also adds a number of design challenges.

Communication between regions is difficult because wires can no longer be routed easily between them and must be facilitated by carefully designed slices (Huebner et al., 2004), though if care is taken, ordinary communication schemes can be implemented - e.g. bus-based interconnect with the bus residing in the static region (Oetken et al.,

2010). However, the advantage is that subsystems may be time-multiplexed in the dynamic regions (Becker et al., 2007), allowing the SoC to achieve more than would otherwise be possible. This can also be used in fault recovery schemes to reconfigure only the partition that has detected faults, rather than the whole FPGA.

2.4.3 Hardware/Software Codesign

As VLSI technology matured, designers began to see that the increase in development complexity for hardware or ASIC designs was impacting their ability to bring products to market quickly. The associated increase in the complexity of microprocessors led many designers to realise that these microprocessor units could be included into system designs and some of the functionality shifted to software to reduce their time-to-market. Microprocessors can be considered a SoC on their own but they can also be included in much larger SoC designs and this is where the idea of hardware/software codesign first began; reviews of the field by Michell and Gupta (1997), Wolf (2003) and Teich (2012) give a comprehensive history of hardware/software codesign.

Early research/work in codesign explored how best to partition functionality, with the software part residing on some microprocessor, the hardware part in an ASIC with shared memory or buffers, and having some communication bus between the two (see Figure 2.9). In this case, the target architecture is fixed but as work in this field progressed, much more elaborate architectures began to rise with multiple ASICs and even multiple microprocessors integrated together into a single SoC. Heavy reuse of hardware blocks, IP cores, as well as software meant that, despite the complexity, development time could be reduced but superior quality products could be achieved.

The accelerating scale of these SoCs integrating more and more functionality – i.e. systems within systems – as transistor densities swelled led to huge demand for similarly powerful design tools. With hardware and software being designed concurrently, design tools with a higher level of abstraction were required not only for design itself but also for simulation and validation. Higher abstraction simulators and streamlined development toolchains in turn allowed the hardware/software codesign approach to

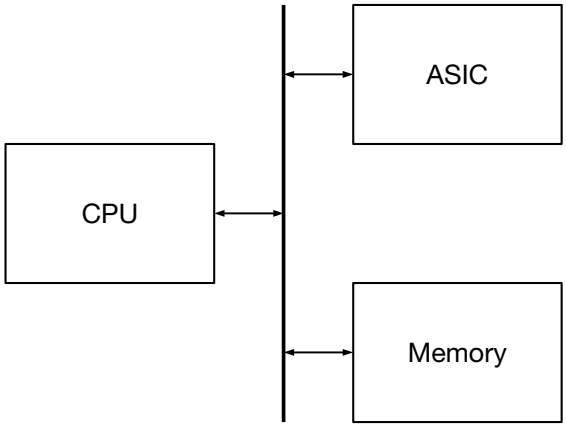


Figure 2.9: An example of a target architecture for early hardware/software codesign implementations

both cut development times further and handle the addition of even greater functionality. Thus hardware/software codesign often refers to not only the separation of the design but also to the design tools and methodology that support such separation and allow the overall development cycle to be shortened, if not necessarily simplified. More current work in this area is aimed at introducing flexibility and runtime adaptability to systems (Teich, 2012); i.e. systems which can change or reconfigure in real-time in response to faults or some other event.

2.5 State Estimation

A system's state can be any, or all, information relevant to the operation of that system. This can be anything from the position and orientation of a spacecraft to the electrical current being supplied to an electric motor or the concentration of reactants in a chemical reaction. Knowing the current state of a system is necessary to understand how that system is operating or to alter its operation in order to produce some more desirable state. In general, not all information regarding a system is observable, however, and must be inferred via other means. To this end, the system is represented by a series of system models which model the evolution of the system based on the previous known state and/or system inputs and/or observations of the

system. State *estimation* is then the process of using these models to assess the operation of the system based on what information is available.

Consider the general system described for discrete time, k:

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \mathbf{w}_{k-1}\right) \tag{2.1a}$$

$$\mathbf{z}_k = h\left(\mathbf{x}_k, \mathbf{v}_k\right) \tag{2.1b}$$

where f and h are the system's process and observation models respectively; $\mathbf{x}$ and $\mathbf{z}$ are the state and observation vector respectively; $\mathbf{u}$ is the control input and $\mathbf{w}$ and $\mathbf{v}$ are respectively the process/control and measurement/observation noise which are assumed to be zero-mean Gaussian white noise terms with covariances $\mathbf{Q}$ and $\mathbf{R}$.

Perhaps the most famous method used for state estimation of systems in this form, from its inception to the current day, is the Kalman Filter and its associated variants. The original and seminal Kalman Filter proposed by Rudolph Kalman was shown to be optimal for linear system models and the assumption that any noise in the system could be modelled via zero-mean Gaussians (Kalman, 1960). Despite its popularity, a flaw in the original Kalman Filter becomes immediately apparent: many 'real-world' systems are not linear and, certainly in aerospace applications, system models tend to be highly non-linear. For these non-linear systems, a variant of the original Kalman Filter was quickly adapted: the Extended Kalman Filter (EKF).

2.5.1 Extended Kalman Filter

The Extended Kalman Filter is a method of state estimation for non-linear systems where the noise is still assumed to be zero-mean Gaussians. The EKF handles non-linear system models by first taking a Taylor Series expansion of the system models truncated at the first order term. These Jacobians are evaluated at each time step for the discrete system and used within the Kalman Filter equations. This alteration essentially linearises the non-linear system models around the current state estimate.

The system models, in this case, must obviously be differentiable in order to use analytical solutions. More realistically, discrete approximations which are recalculated at each time step are used instead.

For the discrete system given by (2.1) where f and h are now non-linear, the formalisation of the EKF contains two parts: the `predict` step and the `update` step. The `predict` step is given by:

$$\hat{\mathbf{x}}^{-}_{k|k-1} = f\left(\mathbf{x}_{k-1|k-1},\ \mathbf{u}_{k-1}\right) \tag{2.2}$$

$$\mathbf{P}^{-}_{k|k-1} = \mathbf{F}_{k-1}\mathbf{P}_{k-1|k-1}\mathbf{F}^{T}_{k-1} + \mathbf{Q}_{k-1} \tag{2.3}$$

where $\hat{\mathbf{x}}^{-}_{k|k-1}$ and $\mathbf{P}^{-}_{k|k-1}$ are the *predicted*/a priori state and covariance respectively based on the previous state and covariance and:

$$\mathbf{F}_k = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k-1|k-1},\ \mathbf{u}_{k-1}} \tag{2.4}$$

is the process model Jacobian. The `update` step starts with the *predicted* observation and covariance, $\hat{\mathbf{z}}_{k|k-1}$ and $\mathbf{S}_{k|k-1}$ respectively, based on the previous state estimate:

$$\hat{\mathbf{z}}_{k|k-1} = h\left(\hat{\mathbf{x}}^{-}_{k|k-1}\right) \tag{2.5}$$

$$\mathbf{S}_{k|k-1} = \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}^{T}_{k} + \mathbf{R}_k \tag{2.6}$$

where:

$$\mathbf{H}_k = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k|k-1}} \tag{2.7}$$

is the observation model Jacobian. Next, the Kalman gain is calculated:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^{T}_{k}\mathbf{S}^{-1}_{k} \tag{2.8}$$

Finally, the updated state estimate and covariance, $\hat{\mathbf{x}}_k$ and $\mathbf{P}_k$ respectively, is calcu-

lated:

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \left(\tilde{\mathbf{z}}_k - \hat{\mathbf{z}}_{k|k-1}\right) \tag{2.9}$$

$$\mathbf{P}_k = \left(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k\right) \mathbf{P}^-_{k|k-1} \tag{2.10}$$

where $\tilde{\mathbf{z}}_k$ are the current set of observations.

The EKF is, and has been, the most widespread method for non-linear state estimation (Gelb, 1974). It has also become the de facto standard by which other methods are compared when analysing their performance. Various surveys of the field have noted that the EKF is: 'unquestionably dominant' (Nørgaard et al., 2000), 'the workhorse' of state estimation (Simon, 2006; Crassidis, Markley, and Cheng, 2007) and the 'most common' non-linear filter (Patwardhan et al., 2012). Despite some shortcomings, the relative ease of implementation and still-remarkable accuracy have propelled the EKF's popularity. However, the EKF does have its flaws; due to the linearisation of the system models, the EKF is no longer, in general, an optimal filter unlike the original Kalman Filter. The calculations of the Jacobians themselves can be extremely computationally demanding, depending on just how far from linearity the system models are. Truncation at the first order term in the Taylor expansion necessarily means the EKF can only be accurate to the first order. Additionally, as the system models become 'highly' non-linear, performance of the EKF suffers dramatically.

The EKF's inadequacies become more and more apparent as autonomous systems with greater and greater amounts of autonomy and precision are desired. Truncating the Taylor expansion at higher order terms is certainly possible for greater accuracy, but calculating the Hessian of a system model or any other higher order derivatives simply compounds the issues already mentioned. It was noted by Nørgaard et al. (2000) that the problem is in the linearisation via Taylor expansion itself; as long as this approach to handling the non-linearity is used, these issues will remain. Nørgaard et al. go on the propose a class of derivative-less filters to handle non-linear systems via *statistical* linearisation instead of *analytical* linearisation. A special case of this

class of filters is the Unscented Kalman Filter (UKF).

2.5.2 Unscented Kalman Filter

The Unscented Kalman Filter (UKF) (Julier and Uhlmann, 1997; Wan and Van Der Merwe, 2000) takes a very different approach to deal with non-linearities in the system models. While the EKF attempts to deal with non-linearities by using the Jacobian to linearise the system model, the UKF instead models the current state as a probability distribution with some mean and covariance. Following this, a deterministic sampling technique known as the Unscented Transform (UT) is applied. A set of points called 'sigma' points, are drawn from the probability distribution and each of them propagated through the non-linear system models. The new mean and covariance of the transformed sigma points are then recovered to inform the new state estimate. The crucial aspect of the UT is that the sigma points are drawn deterministically, unlike random sampling methods like Monte Carlo algorithms, drastically reducing the number of points necessary to recover the 'transformed' mean and covariance.

Using this approach, the UKF has been shown to perform much better than the EKF when the system models are 'highly' non-linear (Nørgaard et al., 2000; Simon, 2006; Crassidis, Markley, and Cheng, 2007; Kandepu et al., 2008; Patwardhan et al., 2012); and this is true for a variety of different applications such as physics (Sitz et al., 2002), aerospace (Crassidis and Markley, 2003; Van Dyke et al., 2004; Giannitrapani et al., 2011) and industrial applications (Xiong et al., 2007; Zhou et al., 2011; Jafarzadeh et al., 2012). This approach to non-linear models is far superior and when validated using Taylor expansion or Monte Carlo methods, this approach can be shown to be accurate to the third order in the case of Gaussian noise or, at least, second order for non-Gaussian noise, depending on the selection of certain parameters in the algorithm (Julier and Uhlmann, 2004). That said, the UKF is not completely infallible: Perea et al. (2007), for example, found that it is still possible for the UKF to diverge or converge to an incorrect state estimate when attempting to fuse data using non-linear observation models of multiple independent sensors that have contrasting accuracies.

The issue is no worse than the EKF, however, meaning the UKF is still viable in plenty of circumstances.

The formalisation of the UKF for the discrete system (2.1) where again, f and h are non-linear is as follows. Define an augmented state vector, $\mathbf{x}^a$, with length M that concatenates the process/control noise and measurement noise terms with the state variables as:

$$\mathbf{x}_k^a = \begin{bmatrix} \mathbf{x}_k \\ \mathbf{w}_k \\ \mathbf{v}_k \end{bmatrix} \tag{2.11}$$

The augmented state vector and associated augmented state covariance, $\mathbf{P}^a$, are initialised with:

$$\begin{aligned} \hat{\mathbf{x}}_0^a &= E\left[\mathbf{x}_0^a\right] = \begin{bmatrix} \hat{\mathbf{x}}_0 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \\ \mathbf{P}_k^a &= E\left[(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a)(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a)^T\right] = \begin{bmatrix} \mathbf{P}_k & 0 & 0 \\ 0 & \mathbf{Q}_k & 0 \\ 0 & 0 & \mathbf{R}_k \end{bmatrix} \end{aligned} \tag{2.12}$$

where $\hat{\mathbf{x}}_0$ is the expected value of the initial (regular) state and $\mathbf{P}_k$ is the (regular) state covariance. The current augmented state and covariance are used to generate the set of sigma points, $\boldsymbol{\mathcal{X}}$, using:

$$\begin{aligned} \boldsymbol{\mathcal{X}}_{0,k} &= \hat{\mathbf{x}}_k^a \\ \boldsymbol{\mathcal{X}}_{i,k} &= \hat{\mathbf{x}}_k^a + \left(\sqrt{(M+\lambda)\mathbf{P}_k^a}\right)_i \qquad i = 1, \ldots, M \\ \boldsymbol{\mathcal{X}}_{i,k} &= \hat{\mathbf{x}}_k^a - \left(\sqrt{(M+\lambda)\mathbf{P}_k^a}\right)_{i-M} \qquad i = M+1, \ldots, 2M \end{aligned} \tag{2.13}$$

where i refers to the i-th column of the matrix 'square-root'; $\lambda = \alpha^2(M+\kappa) - M$ is a scaling parameters; α determines the spread of sigma points about the mean (usually set to some small positive value e.g. 10^{-3}); and κ is a secondary scaling parameter

that is usually set to zero. Each sigma point has an associated weight given by:

$$\begin{aligned} W_0^{(m)} &= \frac{\lambda}{(M+\lambda)} \\ W_0^{(c)} &= \frac{\lambda}{(M+\lambda)} + (1-\alpha^2+\beta) \\ W_i^{(m)} = W_i^{(c)} &= \frac{1}{2(M+\lambda)} \quad i = 1, \ldots, 2M \end{aligned} \tag{2.14}$$

where (m) and (c) denote whether the weight is used for a mean calculation or a covariance calculation, and β is used to incorporate prior knowledge of the distribution around the mean ($\beta = 2$ is optimal for Gaussian distributions). The number of generated sigma points is $N = 2M + 1$ and they can be segmented and associated with their respective state and noise terms via:

$$\boldsymbol{\mathcal{X}}_{i,k} = \begin{bmatrix} \boldsymbol{\mathcal{X}}^x_{i,k} \\ \boldsymbol{\mathcal{X}}^w_{i,k} \\ \boldsymbol{\mathcal{X}}^v_{i,k} \end{bmatrix} \tag{2.15}$$

The `predict` step begins with the sigma points being propagated through the system model:

$$\boldsymbol{\mathcal{X}}^x_{i,k|k-1} = f\left(\boldsymbol{\mathcal{X}}^x_{i,k-1|k-1}, \mathbf{u}_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{i,k-1|k-1}\right) \tag{2.16}$$

The state and covariance are then predicted as:

$$\hat{\mathbf{x}}_k^- = \sum_{i=0}^{N-1} W_i^{(m)} \boldsymbol{\mathcal{X}}^x_{i,k|k-1} \tag{2.17}$$

$$\mathbf{P}_k^- = \sum_{i=0}^{N-1} W_i^{(c)} \left[\boldsymbol{\mathcal{X}}^x_{i,k|k-1} - \hat{\mathbf{x}}_k^-\right] \left[\boldsymbol{\mathcal{X}}^x_{i,k|k-1} - \hat{\mathbf{x}}_k^-\right]^T \tag{2.18}$$

For the `update` step, the sigma points that were updated in the `predict` step are propagated through the observation model:

$$\boldsymbol{\mathcal{Z}}_{i,k|k-1} = h\left(\boldsymbol{\mathcal{X}}^{x}_{i,k|k-1}, \boldsymbol{\mathcal{X}}^{v}_{i,k-1|k-1}\right) \tag{2.19}$$

The mean and covariance of the observation-transformed sigma points are calculated:

$$\hat{\mathbf{z}}_{k|k-1} = \sum_{i=0}^{N-1} W_i^{(m)} \boldsymbol{\mathcal{Z}}_{i,k|k-1} \tag{2.20}$$

$$\mathbf{S}_{k|k-1} = \sum_{i=0}^{N-1} W_i^{(c)} \left[\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right] \left[\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right]^T \tag{2.21}$$

followed by the cross-covariance:

$$\mathbf{P}_{xz,k|k-1} = \sum_{i=0}^{N-1} W_i^{(c)} \left[\boldsymbol{\mathcal{X}}^{x}_{i,k|k-1} - \hat{\mathbf{x}}^{-}_{k|k-1}\right] \left[\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right]^T \tag{2.22}$$

and the Kalman gain:

$$\mathbf{K} = \mathbf{P}_{xz,k|k-1}\mathbf{S}^{-1}_{k|k-1} \tag{2.23}$$

Finally, the current system state is estimated by:

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}^{-}_{k|k-1} + \mathbf{K}\left(\tilde{\mathbf{z}}_k - \hat{\mathbf{z}}_{k|k-1}\right) \tag{2.24}$$

where $\tilde{\mathbf{z}}$ is the current set of observations and the current covariance is updated with:

$$\mathbf{P}_k = \mathbf{P}^{-}_{k|k-1} - \mathbf{K}\mathbf{S}_{k|k-1}\mathbf{K}^T \tag{2.25a}$$

$$= \mathbf{P}^{-}_{k|k-1} - \mathbf{P}_{xz,k|k-1}\mathbf{K}^T \tag{2.25b}$$

where the expression for the Kalman gain, (2.23), is substituted.

Other benefits of the UKF include no longer having to calculate and evaluate potentially complex Jacobians at every time step. This benefit may be countered by the

fact that, in practice, the UKF execution time is much higher. St-Pierre and Gingras (2004), for example, compared the EKF and UKF for an integrated navigation system, for a car, where odometer, IMU and GPS data were fused, but found an order of magnitude increase in computation time for the UKF despite the increase in positional accuracy. Holmes et al. (2009) experimented with using the UKF for monocular SLAM while Kurt-Yavuz and Yavuz (2012) compared all the major SLAM solutions, including the UKF, and both found similar problems with the UKF computation time despite greater accuracy. This issue may be, at least in part, because multiple calculations of the system models must now occur, specifically, one instance for each sigma point. Though this may be an issue for complex models using the traditional processor approach, it is here where hardware designed for an FPGA has the potential to benefit, by reducing execution times back down to feasible levels.

2.5.2.1 Spherical simplex sigma points

There exists various other sigma point selection strategies and, in order to minimise computational effort, a selection strategy involving a minimal set of samples is highly desired. The spherical simplex set of points (Julier, 2003; Julier and Uhlmann, 2004) can be shown to offer similar performance to the original UKF with the smallest number of sigma points required ($M + 2$). Using this sigma point selection strategy keeps the size of certain matrices down and so reduces the computational demand of the UKF compared to other sigma point selection strategies. The sigma point weights and a coefficient matrix is generated by choosing $0 \leq W_0 \leq 1$, then calculating W_1:

$$W_1 = W_i = \frac{(1 - W_0)}{(M + 1)} \qquad i = 1, \ldots, M + 1 \tag{2.26}$$

The vector sequence is initialised as:

$$\boldsymbol{\sigma}_0^1 = [0]\,, \quad \boldsymbol{\sigma}_1^1 = -\left[\tfrac{1}{\sqrt{2W_1}}\right] \quad \boldsymbol{\sigma}_2^1 = \left[\tfrac{1}{\sqrt{2W_1}}\right] \tag{2.27}$$

Then the vector sequence is expanded for $j = 2, \ldots, M$ via:

$$\boldsymbol{\sigma}_i^j = \begin{cases} \begin{bmatrix} \boldsymbol{\sigma}_0^{j-1} \\ 0 \end{bmatrix} & i = 0 \\ \begin{bmatrix} \boldsymbol{\sigma}_i^{j-1} \\ -\frac{1}{\sqrt{j(j+1)W_1}} \end{bmatrix} & i = 1, \ldots, j \\ \begin{bmatrix} \mathbf{0}_{j-1} \\ \frac{j}{\sqrt{j(j+1)W_1}} \end{bmatrix} & i = j+1 \end{cases} \tag{2.28}$$

Here $W_i^{(c)} = W_i^{(m)}$, $i = 0, \ldots, M+1$ and the actual sigma points are drawn from:

$$\boldsymbol{\mathcal{X}}_{i,k} = \hat{\mathbf{x}}_k^a + \left(\sqrt{\mathbf{P}_k^a}\boldsymbol{\sigma}\right)_i \tag{2.29}$$

which replaces the original sigma point selection strategy given by (2.13).

2.5.3 Hardware Kalman Filters

Implementing complex algorithms like the Kalman Filter can be a difficult task even in software. Given the additional complexities of hardware designs, there are not many examples of hardware, hardware/software or FPGA-based Kalman Filters (of any variant), although they do exist.

Quinchia and Ferrer (2011) implemented the (linear) Kalman Filter entirely as a hardware IP core and attached it to a softcore processor as a peripheral device; this improved the performance of their algorithm, however the implementation is completely application specific. Cruz et al. (2013) implemented just the `update` step of the EKF into an IP core for a localisation subsystem in a wheeled robot; the work is continued by Contreras et al. (2015a) to implement the `predict` step as another IP core as well. In both cases, the IP cores are connected to a softcore processor over a communication bus as a peripheral device. Similarly, Idkhajine et al. (2012) implemented an EKF IP core which connected to a processor over a communication bus for a synchronous AC drive current controller.

There are some examples of hardware/software codesigns of the EKF as well. In fact, Bahri et al. (2013) simply extends the work done by Idkhajine et al. (2012) to a codesign in order to better balance 'control performance, controller complexity and design flexibility'. Similarly, Contreras et al. (2015b) presents an alternative implementation to Contreras et al. (2015a) where the EKF `predict` step is implemented in software on the processor for much the same reasons. Aung et al. (2013) implements most of the EKF in hardware but leaves calculation of the models in software to take advantage of existing standardised software for motor control in automotive applications, thus reducing their time-to-market. Tertei et al. (2014), in developing a visual SLAM system, implemented the matrix multiplier of an EKF in hardware and attached it to the processor as an IP core to behave like a coprocessor.

Another approach to the hardware/software codesign, other than partitioning functionality, is to take advantage of the softcore processor specification and modify it to include custom instructions. Bonato, Peron, et al. (2007) used such an approach with the EKF for the localisation system and controller of a wheeled robot. Guo, Chen, et al. (2012) used this method to develop an EKF velocity estimator for automotive vehicles; Guo, Chen, et al. also used an additional IP core of a matrix multiplier. This approach means that although the implementation can be easily used between applications, it is now restricted to a particular device (or device family).

Bonato, Marques, et al. (2009) continue their work (Bonato, Peron, et al., 2007) by implementing application-specific aspects of the EKF – the system models – into software on a softcore processor while implementing non-application-specific parts of the EKF into a hardware IP core attached to the processor. They take advantage of certain features of their application – the SLAM problem for mobile robotics – to restrict the sizes of the state and observation vectors, allowing the IP core to be agnostic with respect to the sensors used. With known state and observation vector sizes, Bonato, Marques, et al. use 4 processing elements (PEs) to balance on-chip memory requirements with off-chip memory bandwidth requirements. Though their implementation is an application-specific EKF, it is not restricted to any particular FPGA device or (SLAM) system hardware.

There are only a small number of examples of UKF implementations as well. Previous work by the author (Soh and Wu, 2012) attempted an application-specific hardware implementation of the UKF as a standalone IP core that would interface directly with the sensors and actuators of a nanosatellite; however, this design lacked performance optimisation and generality.

A patent by 姬红兵 et al. (2013) describes a method of implementing a UKF entirely in hardware on an FPGA. An example embodiment is given involving tracking a single moving object in a 2-D plane with three passive sensors in a fixed location. The position, velocity and acceleration of the object is tracked with state vector of size 6. The sensors measure the distance to the object in both dimensions as well as the angle to the object with respect to the horizontal axis; the observation vector is size 3. Computation times for up to 6 PEs are disclosed. The matrix inversion (see (2.23)) is handled by using Singular Value Decomposition (SVD) to calculate the pseudoinverse matrix (i.e. for target square matrix, $\mathbf{A}$, with SVD, $\mathbf{A} = \mathbf{USV}$, where $\mathbf{U}$ and $\mathbf{V}$ are unitary and $\mathbf{S}$ is diagonal, the pseudoinverse can be computed via: $\mathbf{A}^{-1} = \mathbf{US}^{-1}\mathbf{V}$).

This patent discloses an application-specific implementation of the UKF and does not mention how higher dimensionality variants of the example embodiment (e.g. a 3D variation) may be handled nor how the design can be translated to handle other, completely different, applications. The patent further discloses an implementation onto, specifically, a Xilinx XC4VFX140 FPGA (Xilinx, 2010) and it is unclear whether the design can be implemented on other devices. Though a parallelisation scheme is described (up to 6 PEs), it is not clear whether an arbitrary number of PEs can be used.

Entirely hardware, application-specific implementations of the UKF are also proposed by Ramchandani et al. (2012) and Yang, Deng, et al. (2017). Ramchandani et al. propose using the UKF as part of a Maximum Power Point Tracking (MPPT) system for photovoltaic cells. Their design does not appear to contain a parallelisation scheme and while algorithm performance is reported, hardware performance is not. Yang, Deng, et al. propose using the UKF to estimate dynamic characteristics of thalamo-

cortical cells in the study of Parkinson's disease. Their design uses parallel calculation blocks in the propagation of the sigma points through the system models and most of the matrix calculations, but does not appear to be arbitrarily parameterisable. Algorithm performance, but not hardware performance, is reported.

One HW/SW codesign approach is a patent by 刘仙 et al. (2014) which describes a method of implementing the UKF, along with a neural dynamics model, to remove interference from electroencephalogram (EEG) signals. The invention uses a soft-core processor with custom instructions to implement the UKF on a Altera Cyclone IV EP4CE15F17C8 (Altera, 2016) device; this makes the implementation not only application-specific, but also device-, or device family-, specific.

2.6 Summary

This chapter gave background information and reviewed relevant literature on FPGAs, FPGA development processes, FPGA applications, System-on-Chip techniques and state estimation. A short introduction to FPGAs and FPGA development was first given. The SoC methodology of integrating multiple 'black boxes' (IP cores) into a single device was then described; hardware/software codesign techniques for SoCs were also introduced. A short synopsis of the most popular state estimation method, the Kalman Filter family of algorithms, was presented; existing attempts at implementing hardware or HW/SW Kalman Filters and its variants were also given.

FPGAs are capable of increasing the performance of complex algorithms but the development process – translating these algorithms to hardware – is more difficult and expensive when compared to software development. Creating reusable IP cores for SoC systems does not necessarily reduce development costs for the initial application but helps reduce development costs of subsequent applications that use that IP core. Using a HW/SW codesign can help reduce development costs further since, not only is software easier to develop for initial applications, it is also easier to adapt for subsequent applications. Although the EKF has been the most popular state estimation algorithm for non-linear systems, it suffers linearisation problems which can reduce

the algorithm's performance in highly non-linear systems. The UKF is a newer variant that has shown superior algorithm performance over the EKF but at the cost of slightly higher computational requirements. There are very few examples of hardware or HW/SW Kalman Filters, of any variant, and those that exist are largely application- or device-specific; these approaches are unlikely to reduce development costs of subsequent applications.

Small aerospace systems need to retain high performance state estimation to have any advantage over larger systems but miniaturisation efforts put considerable pressure on the available resources (e.g. computing power, physical space). Using an FPGA for computation can help deliver the performance necessary but increases development costs. Using SoC techniques can help save physical space as multiple functions are integrated onto a single chip; using a HW/SW codesign as well may mitigate the increase in development costs brought on by using an FPGA. The UKF is a popular and widely applicable state estimation algorithm and its increased computational requirements can be relieved by the greater performance of an FPGA. Hence, a HW/SW codesign of the UKF is an excellent approach to creating a generic state estimation library which may be highly beneficial in the development of small aerospace systems.

Chapter 3

Hardware/Software Codesign of the Unscented Kalman Filter

In this chapter, the proposed hardware/software (HW/SW) codesign of the UKF is described. The overall design methodology is outlined starting with the hardware/software partitioning strategy and how the hardware and software parts are intended to interact; generation of header files to ensure the coherence of relevant parameters between the hardware and software parts is then detailed. The main contributions of this thesis – three variants of a HW/SW UKF – are described before, finally, a summary of the advantages and disadvantages of each variant is given.

3.1 Design overview

The first exercise in the hardware/software codesign is to divide the UKF algorithm into two parts. For maximum performance, it is desirable for as much of the algorithm as possible to be implemented in hardware. However, to maintain portability, any part of the algorithm that is application-specific would be better implemented in software. This is so that the application-specific parts can make use of the faster and simpler development processes that using software entails. Reviewing the UKF algorithm, only the two system models, the `predict` and `update` models, are application-specific.

The `predict` model, (2.16), models the evolution of the system's state between time steps. This model, also known as the process model, usually encapsulates the behaviour of the system subject to control actions or external forces; for example, consider a 3D rigid-body dynamics model of a multi-rotor micro-UAV which accounts for gravity and air resistance. Changing to a different system (e.g. from a multi-rotor UAV to a fixed-wing UAV) would require the `predict` model be updated to reflect the new system. Different applications usually require different systems, and so the `predict` model is highly application-specific; keeping the `predict` model in software allows rapid re-development if the system or application changes.

The `update` model, (2.19) models the measurement of the state of the system at the current time step. Measurements of the system's state are carried out by sensors incorporated into the system; for example, a multi-rotor UAV may use an IMU to determine its attitude and a GPS receiver to determine its position and speed. If the set of sensors carried by the system change (e.g. using a camera instead of an IMU for attitude), then the `update` model would need to be updated to reflect the new sensors, even if the base system does not change. The appropriate sensor set depends strongly on the type of base system used and what it is being used for. So as with the `predict` model, the `update` model is highly application-specific.

Apart from the two system models, the rest of the UKF can be viewed as, essentially, a series of matrix manipulations. The only changes to the rest of the UKF when either of the system models change is the size of the vectors and matrices used in the UKF calculations. The sizes of these vectors and matrices are fixed for a particular formulation of the UKF and so they can be treated as parameters that are set at synthesis. Fixing the parameters at synthesis means that only the bare minimum of hardware resources are needed but the hardware can still be easily used for different applications with different vector/matrix sizes; rather than needing to redesign any functionality, the hardware can simply be synthesised with different parameters. Thus, the rest of the UKF can be designed and then used as a parameterisable, modular 'black-box' (IP core) and implementing it for any given application only requires the appropriate selection of parameters.

When it comes to designing hardware, there are three main considerations: the performance of the hardware (which may include data throughput, maximum clock frequency etc.), the logic area (on-chip resources) used and the power consumption of the hardware. During development these considerations are usually at odds with each other - specifically performance is usually opposed by logic area and power consumption. In order to increase the performance of the design, additional resources often have to be used which, in turn, may increase the power consumption; these increases in resource/power cost may make the implementation infeasible for a given application. Due to these considerations, there are a number of different strategies that can be applied to hardware design; which strategy is preferable depends on the requirements of the application. Possible design strategies include: minimising resource usage, minimising power consumption, maximising performance, or maintaining a balance of all three.

As each different design strategy is appropriate depending on the situation, no single strategy is suitable for the codesign if it is to be as widely applicable as possible. With the stated goal of creating a generic library that can be easily applied to any application, rather than introduce just one design, three different designs are presented, each with advantages and disadvantages. Each of the designs is developed with different applications in mind but is otherwise easily swapped with the others, allowing the greatest flexibility to system designers.

The first design, which will be referred to as Serial, is the most basic and is intended to fit within a greater SoC or to be used as a coprocessor to a standard microprocessor. The second design, referred to as Parallel, exploits the hardware advantage of parallelism in order to accelerate the computation of low-level arithmetic; this design is intended for use as a coprocessor. The third design, referred to as Pipeline, further exploits parallelism to allow multiple instances of the algorithm to be calculated at once; this design is intended, along with a dedicated microprocessor, for use as a standalone subsystem.

Although three different designs are presented, the variation in design only applies to the hardware part (the IP core). The same partitioning method is used for all three

design variants and the software part is still developed as per the requirements of the application. The hardware/software partition strategy is summarised in Table 3.1.

Hardware		Software	
Non-application-specific		Application-specific	
$\boldsymbol{\mathcal{X}}_{i,k-1}$	(2.29)	$\boldsymbol{\mathcal{X}}^{x}_{i,k\|k-1}$	(2.16)
$\hat{\mathbf{x}}_k^-$	(2.17)	$\boldsymbol{\mathcal{Z}}_{i,k\|k-1}$	(2.19)
$\mathbf{P}_k^-$	(2.18)		
$\hat{\mathbf{z}}_{k\|k-1}$	(2.20)		
$\mathbf{S}_{k\|k-1}$	(2.21)		
$\mathbf{P}_{xz,k\|k-1}$	(2.22)		
$\mathbf{K}$	(2.23)		
$\hat{\mathbf{x}}_k$	(2.24)		
$\mathbf{P}_k$	(2.25a)		

Table 3.1: Summary of the hardware/software partitioning of the UKF. See Section 2.5.2 for details of the UKF algorithm. All three variants use this partitioning method.

The actual physical implementation of the hardware/software UKF on an FPGA can be seen in Figure 3.1. The hardware part is implemented as a standalone IP core and the software part is implemented on a general-purpose microprocessor. The processor acts as the main controller which, in addition to implementing the system model software, controls the hardware IP core. The precise method of controlling the IP core is dependent on the design variation and is elaborated on in the following sections.

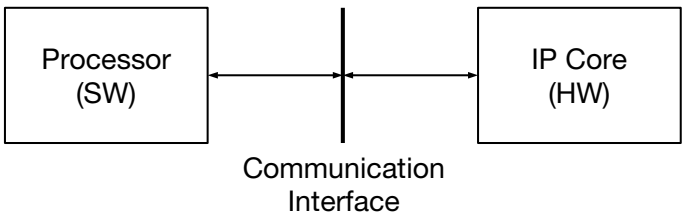


Figure 3.1: The hardware/software partition on the FPGA

The processor communicates with the IP core over some communication interface. Any intra-chip communication method would be sufficient and would be driven mostly by the requirements of the application; viable interfaces include point-to-point, bus or NoC interfaces. The IP core contains memory buffers at the interface in order to

receive data from the processor as well as to temporarily store data that needs to be read by the processor. The communication interface is the same between all three variants but the specifics of the memory buffers are not.

In this thesis, the communication interface between the two parts is an AXI4 bus. All variants are implemented using single precision arithmetic (IEEE-754); this gives a decent balance of dynamic range and resource usage which should be sufficient for the majority of applications. All hardware in the codesign is developed using the Verilog HDL and all software in the codesign is developed using C. Although C is used here, in general, any type of software (i.e. programming language) may be used as long as it contains the ability to interact with the communication interface connecting the hardware and software parts.

3.1.1 Header generation

There are a number of parameters chosen during the design of the UKF algorithm that need to be recorded so that data is correctly interpreted during the operation of the codesign. These parameters are application-specific as they depend on the design of the `predict` and `update` models. The parameters are:

- Length of the augmented state vector (2.11)
- Number of sigma points (2.29)
- Number of state variables (2.1a)
- Number of observation variables (2.1b)
- Sigma weighting coefficients W0/W1 (2.26)

These parameters set the expected structure of data read from or written to the memory buffer by both the hardware and software parts. The hardware part also uses these parameters to instantiate memory blocks of the correct size, initialise state machines and initialise the control loops that handle the various vectors/matrices.

The software part uses these parameters to initialise the control loops that handle the sigma points propagation process.

To ensure that the recorded parameters are consistent between the hardware and software parts, the parameters are first stored in a plain-text parameter file. This parameter file is then used by a configuration script to generate the appropriate headers for inclusion into the source files of both the hardware and software parts (see Figure 3.2). In order to move to a new application, a designer updates the parameter file depending on the requirements of the application and the design of the UKF for that application; the configuration script is then run to regenerate the header files for the new application. This is a simple process that can easily occur alongside the software development of an application.

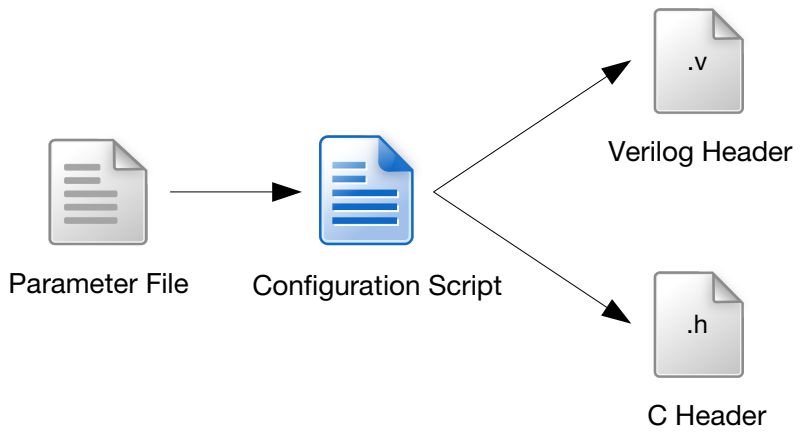


Figure 3.2: Generation of header files

A number of additional hardware-only parameters are also recorded by the parameter file:

- Latency of the floating-point arithmetic (e.g. Table 3.2):
  - Multiply
  - Accumulate
  - Fused multiply-add
  - Subtract

  - Square-root
  - Divide

- Number of processing elements
- Sigma weighting coefficient matrix
- Depth of the memory buffer

In this case, additional files for the hardware part only are generated. The sigma weighting coefficient matrix (see Section 2.5.2) is stored by the hardware part as a ROM. The ROM needs to be initialised with the desired values at synthesis time so, the configuration script generates the matrix as per (2.28) and exports the values to an initialisation file. The script then adds the current path to the HDL header so that the matrix values can be fetched during synthesis.

For this thesis, the header generation scripts were written in Matlab to help maintain consistency with simulation models; for a standalone library, however, a freer scripting language with suitable text manipulation tools may be more appropriate.

3.2 Serial design

In this section, the Serial variant of the HW/SW UKF is introduced. This section is based on, and the Serial design was first presented in, Soh and Wu (2014). The division of the UKF algorithm and how the different parts are calculated is outlined. The three major datapaths, associated secondary arithmetic and memory blocks that compose the IP core and its control scheme for the Serial design are also described.

The Serial design design strategy is to minimise the area and power consumption as much as possible with the intent to include the design into a greater SoC. As such, the design forgoes one of the main benefits of hardware implementations: wide parallelism. The performance of this design will suffer but the low resource cost

will allow it to be used in severely resource-constrained applications; for example, on board a nanosatellite or micro-UAV.

The UKF algorithm can be logically divided into two parts: the `predict` step (for the Serial design, we consider sigma points generation as part of the `predict` step) and the `update` step. The algorithm must first be initialised with an augmented state estimate and covariance before any calculations can begin. After the algorithm is initialised, the `predict` and `update` steps can be calculated as necessary. The `predict` step can be calculated independently, but the `update` step must always be preceded by a `predict` step.

Though the algorithm can be logically divided into two parts, the hardware implementation attempts to reuse as much of the resources as possible and so only separates the low-level functions into modules at an HDL level. The Serial design contains the following modules and memories which are controlled via a large state machine:

- Modules
  - Triangular linear equations solver (`trisolve`)
  - Matrix multiply-add
  - Calculate mean/covariance
  - Calculate sigma point residuals (Subtract)
  - Floating-point arithmetic
- Memory
  - Augmented state
  - Augmented covariance
  - Sigma weighting matrix
  - Cholesky decomposition
  - State/Observation sigma point residuals
  - Observation residual

- Observation covariance
- Cross covariance
- Kalman gain

All non-arithmetic modules are instantiated only once and the floating-point arithmetic is shared between these modules. Each of the floating-point arithmetic modules are deeply pipelined in order to maintain a reasonable synthesisable clock frequency. The floating-point arithmetic modules and their clock cycle latencies are listed in Table 3.2. A state machine ensures there are no data collisions and that only one module at a time can feed data into the floating-point arithmetic pipelines.

Module	Latency (cycles)
Multiply	8
Accumulate	22
Fused multiply-add	11
Subtract	11
Square-root	28
Divide	28

Table 3.2: Basic arithmetic modules and their latencies

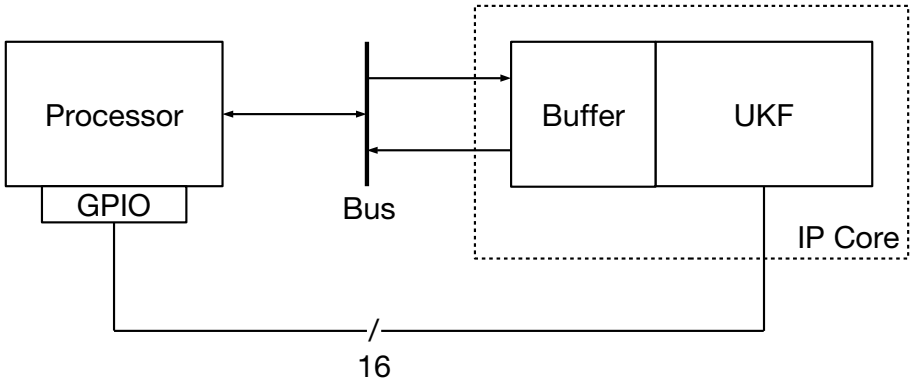


Figure 3.3: Top-level block diagram of the Serial design

The IP core for the Serial design consists of the UKF hardware and a memory buffer which is attached to a communication (AXI4) bus. The interface between the hardware and software parts also features a set of 16 control lines which are used by

the processor to control the IP core; a top-level block diagram of the Serial design illustrating the structure can be seen in Figure 3.3.

The 16 control lines are simply digital signals that can be attached to either a General Purpose Input/Output (GPIO) module or an interrupt controller. These control lines allow the processor to initiate the IP core calculations, to know where the IP core calculations are up to and to know when the processor needs to deliver model data. The complete list of control lines can be seen in Table 3.3. The memory buffer itself is a simple FIFO in either direction. Both the processor and IP core make use of the control signals to know how to interpret the data that is in the buffer at any one time.

No.	I/O	Function
0	I	Reset
1	I	Enable
2	I	Initialise state
3	O	State initialised
4	I	Initialise covariance
5	O	Covariance initialised
6	I	Start `predict` step
7	O	`predict` step complete
8	O	Request `predict` model
9	I	`predict` model complete
10	I	Start `update` step
11	O	`update` step complete
12	O	Request `update` model
13	I	`update` model complete
14	O	Not Connected
15	O	Not Connected

Table 3.3: Control lines for the Serial design. I/O is with respect to the IP core; so I: PS $\rightarrow$ IP core and O: PS $\leftarrow$ IP core.

3.2.1 State machine

The Serial design IP core has five top-level states: an idle state, two initialisation states and one state each for the two parts of the UKF (`predict` and `update`); the

top-level state diagram can be seen in Figure 3.4. The IP core remains in the idle state until given a signal by the processor to perform an initialisation, the `predict` step or the `update` step; these steps can be performed independently as required, or as the available sensor data will allow. Once any of the steps are completed, the IP core returns to an idle state. The state machine only enables hardware that is necessary for the current state; hardware that is not being used by the current state is kept disabled to conserve power.

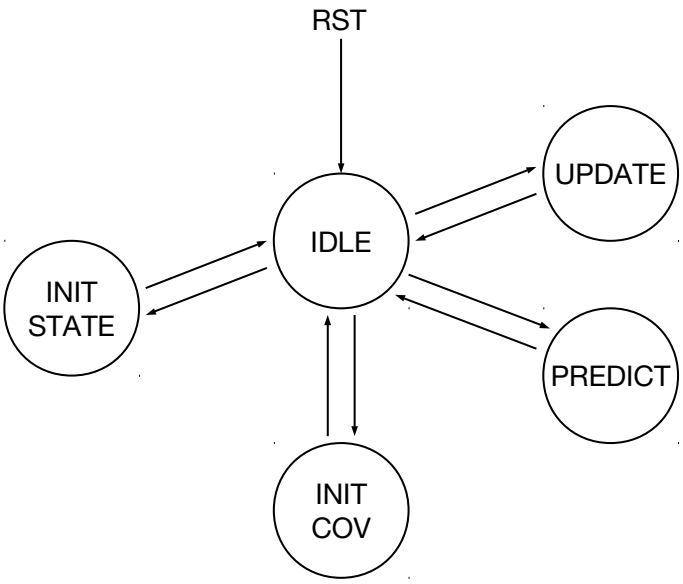


Figure 3.4: Top level state diagram for the Serial design. Transition conditions are omitted for clarity, but are all tied to the appropriate control signals from Table 3.3.

The initialisation process (i.e. the two `init` states) involves the processor copying the initial augmented state estimate and covariance into the memory buffer where it is copied again into a local memory block. The local memory block for the augmented state estimate and covariance is used by the IP core to keep track of the current estimate in expectation of the next iteration of the algorithm.

3.2.2 Predict step

The `predict` step for the Serial design generates the new set of sigma points and calculates the a priori state estimate. A block diagram of the `predict` step architecture,

showing the data flow between hardware modules, can be seen in Figure 3.5.

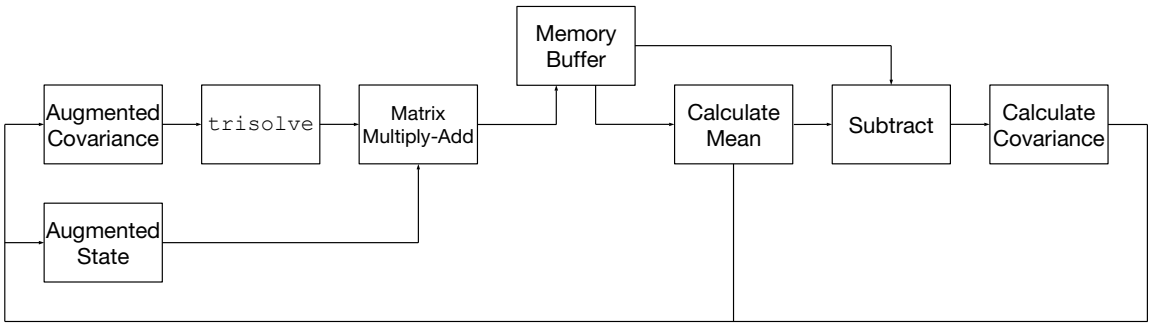


Figure 3.5: Block diagram of the `predict` step for the Serial design

The `predict` step begins by using the current augmented state vector and covariance, stored in a local memory block, to calculate the new sigma points via (2.29). To calculate the new set of sigma points, first the matrix 'square-root' of the current augmented covariance must be calculated. For some matrix $\mathbf{A}$, matrix $\mathbf{B}$ is the square root of $\mathbf{A}$ if:

$$\mathbf{A} = \mathbf{B}\mathbf{B} \tag{3.1}$$

A number of methods to calculate $\mathbf{B}$ for any given $\mathbf{A}$ exists; for example, via direct diagonalisation, Schur's algorithm or Denman-Beavers iteration to name a few. However, if $\mathbf{A}$ is positive definite – which the covariance matrices in the UKF necessarily are – the Cholesky Decomposition (Golub and Van Loan, 1996) can be used. The Cholesky Decomposition is somewhat simpler than other methods but is still stable and accurate and, especially in aerospace applications, is favoured for the UKF (Rhudy et al., 2012). The Cholesky Decomposition is implemented via a triangular linear equations solver (i.e. `trisolve` in Figure 3.5) which is described in Section 3.2.2.1.

The 'square-root' of the augmented covariance is then multiplied by the sigma coefficients weighting matrix and the current augmented state vector added column-wise; this is implemented by the matrix multiply-add module described in Section 3.2.2.2. The new sigma points are placed in the memory buffer (to the processor) and the appropriate control lines are set. Once the processor has propagated the sigma points

through the `predict` model, it places the transformed sigma points back into the memory buffer and signals the IP core to proceed via the appropriate control line.

The mean of the transformed sigma points is calculated which is also the a priori state estimate (2.17). The transformed sigma points and the mean are then used to calculate the 'sigma point residuals' via a `subtract` operation. From the 'sigma point residuals', the covariance of the set of transformed sigma points is calculated which is also the a priori covariance (2.18). Calculation of the mean and covariance is implemented by the calculate mean/covariance module described in Section 3.2.2.3; this section also describes the details of the 'sigma point residuals'. Once the calculations are complete, the IP core writes the a priori state and covariance to the local augmented state and covariance memory blocks as well as the memory buffer so that both the processor and IP core has the current state estimate.

Since each of the hardware modules are only instantiated once and reused between the different calculations, a state machine is necessary to control the hardware and prevent data collisions; this is also necessary because the floating-point arithmetic is shared between the major modules. A state diagram of the `predict` step can be seen in Figure 3.6. This state machine occurs entirely within the `predict` state in Figure 3.4. Each state allows the calculations for its eponymous module to complete before transitioning. The `wait` state is for when the new set of sigma points are placed into the memory buffer and the IP core requests the processor to propagate them through the `predict` model; once the processor signals it has done so, the IP core transitions to the next state.

3.2.2.1 Triangular linear equations solver

In addition to the matrix 'square-root', the Choleksy Decomposition is also used in the Kalman gain calculation which involves a matrix inversion (see 2.23). Directly computing a matrix inversion is extremely computationally demanding so, rather than directly inverting the matrix, an algorithm called the matrix 'right divide' is used here. The matrix 'divide' defines operators \ and / which denote the matrix

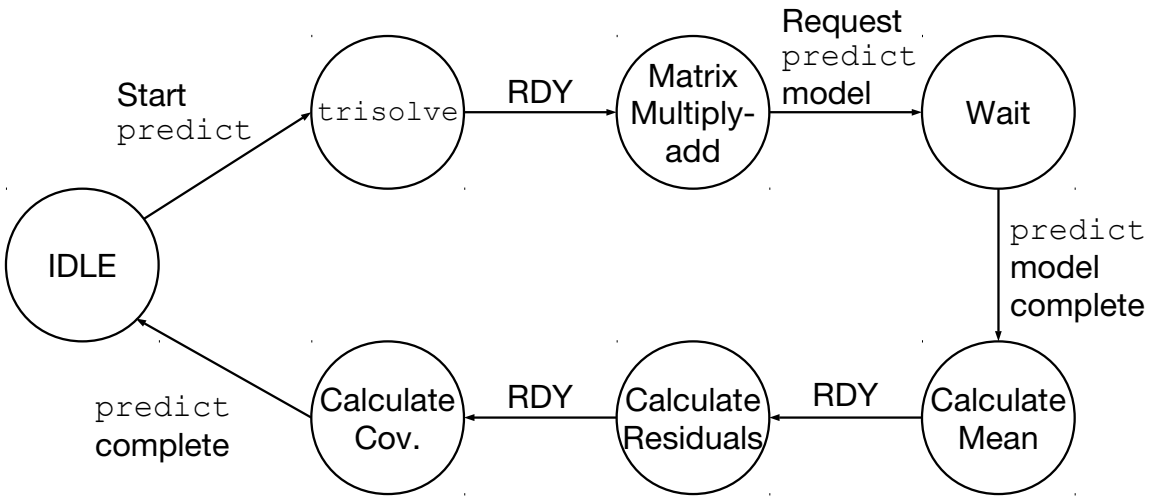


Figure 3.6: State diagram of the `predict` step for the Serial design

'left divide' and matrix 'right divide' respectively, such that for $n \times n$ square matrix $\mathbf{A}$ and $n \times 1$ vector $\mathbf{b}$:

$$\mathbf{A} \backslash \mathbf{b} \equiv \mathbf{A}^{-1}\mathbf{b} \tag{3.2a}$$

$$\mathbf{b} / \mathbf{A} \equiv \mathbf{b}\mathbf{A}^{-1} \tag{3.2b}$$

This is a computational algorithm which, for the system $\mathbf{Ax} = \mathbf{b}$:

1. If $\mathbf{A}$ is triangular, performs back substitution or forward elimination to solve for $\mathbf{x}$ directly
2. If the matrix is positive definite, performs the Cholesky decomposition on $\mathbf{A}$, then performs step 1
3. Otherwise performs an LU decomposition with partial pivoting, then performs step 1

The Cholesky and LU decomposition reduces the matrix $\mathbf{A}$ into its upper and lower triangular components, such that:

$$\begin{aligned} \mathbf{A} &= \mathbf{LU} \qquad \text{(LU)} \\ \mathbf{A} &= \mathbf{LL}^T \quad \text{(Cholesky)} \end{aligned} \tag{3.3}$$

where $\mathbf{L}$ is a lower triangular, $\mathbf{U}$ is the upper triangular matrix and $\mathbf{L}^T$ is the transpose of $\mathbf{L}$, thus this algorithm solves for $\mathbf{x}$ via:

$$\mathbf{Ax} = \mathbf{b} \tag{3.4}$$

$$\mathbf{LUx} = \mathbf{b} \tag{3.5}$$

$$\mathbf{x} = \mathbf{U}\backslash\mathbf{L}\backslash\mathbf{b} \tag{3.6}$$

Note that the number of operations the Cholesky decomposition requires is roughly half the LU decomposition as it doesn't have to calculate a second triangular matrix.

To calculate the Cholesky Decomposition, set $L_{11} = \sqrt{A_{11}}$ and $L_{21} = A_{21}/L_{11}$, then the rest of the matrix can be calculated element-wise via:

$$L_{ij} = \frac{1}{L_{jj}}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}\right) \quad \text{for } i > j \tag{3.7a}$$

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} \tag{3.7b}$$

The calculation may proceed either by row or by column. Here, the multiply-accumulation dominates the calculation in both instances so the time complexity for the decomposition is roughly $\mathcal{O}(n^3)$ operations where n is the size of the target matrix[i]. The main problem, however, is the need to perform a (scalar) square-root and a divide operation in each iteration (i.e. the calculation of a given row/column). There are multiple multiplication and accumulation operations per iteration so they can both be easily pipelined, but the square-root and divide makes the datapath inefficient. This is because square-root and divide operations are much more complex than multiplication and accumulation and so their calculations usually have a much longer latency (e.g. see Table 3.2); furthermore, only one division per element or square-root operation per row/column is necessary making their pipelines inefficient. Since a divide operation is in every non-diagonal calculation, successive elements in

[i]In fact, in can be shown that it requires $n^3/3$ floating point operations (Golub and Van Loan, 1996)

a row/column must wait for this long latency divide operation to complete before proceeding; similarly, a square-root operation for every diagonal element means the calculation for the next row/column must wait on this long latency operation.

Given these inefficiencies, instead of the original decomposition:

$$\mathbf{A} = \mathbf{L}_1 \mathbf{L}_1^T \tag{3.8}$$

where $\mathbf{L}_1$ is lower triangular, an alternative version of the decomposition is used:

$$\mathbf{A} = \mathbf{L}_2 \mathbf{D} \mathbf{L}_2^T \tag{3.9}$$

where $\mathbf{L}_2$ is lower triangular and its diagonal terms are unit elements, $\mathbf{D}$ is diagonal and the two versions are related by:

$$\mathbf{L}_1 = \mathbf{L}_2 \sqrt{\mathbf{D}}^{\text{ii}} \tag{3.10}$$

This alternate version of the Cholesky Decomposition is sometimes referred to as the LDL Decomposition and to calculate it, first set $L_{11} = 1, D_1 = A_{11}$ and $L_{21} = A_{21}$, then the element-wise calculation is:

$$L_{ij} = \frac{1}{D_j} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} D_k \right) \quad \text{for } i > j \tag{3.11a}$$

$$D_j = A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2 D_k \tag{3.11b}$$

At first glance, this alternative version looks worse with additional terms in the L_{ij}

[ii] Recalling (3.1), the matrix square-root of a diagonal matrix is simply the (scalar) square-root of its elements

calculation, however let:

$$F_{ij} = L_{ij} D_j \quad i > j \tag{3.12a}$$

$$F_{jk} = L_{jk} D_k, \quad k = 1, \ldots, j-1 \tag{3.12b}$$

and rewrite the element-wise calculation as:

$$F_{ij} = \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} F_{jk} \right) \quad \text{for } i > j \tag{3.13a}$$

$$D_j = A_{jj} - \sum_{k=1}^{j-1} \frac{F_{jk}^2}{D_k} \tag{3.13b}$$

In this form, the divide operation is moved from the F_{ij} calculation to the D_j calculation. If we calculate the Decomposition by row, then the D_j calculation only needs to be calculated once per row and can be performed in parallel, meaning the long latency divide operation has little to no effect on the main datapath. Furthermore the square-root operation is eliminated completely unless it is desired to recombine the LDL products in order to recover the original Cholesky Decomposition (i.e. $\mathbf{L}_1$) product; the recombination process can occur in parallel as well. Calculating by row also means the expression for F_{ij} can now be considered as simply solving a series of triangular linear equations, i.e. solving for $\mathbf{y}$ in the system:

$$\mathbf{F}_{i-1} \mathbf{y}_i^T = \mathbf{a}_i^T \tag{3.14}$$

where $\mathbf{y}_i$ is the next row in the LDL Decomposition, $\mathbf{a}_i$ is the i-th row of $\mathbf{A}$ and $\mathbf{F}_{i-1}$ is the LDL Decomposition triangular matrix product already calculated so far with the elements given by (3.13a). Considering the matrix 'divide' and the system given by (3.4) now, if we use the LDL Decomposition on $\mathbf{A}$:

$$\mathbf{L}_2 \mathbf{D} \mathbf{L}_2^T \mathbf{x} = \mathbf{b} \tag{3.15}$$

then this system can be similarly reduced to a triangular linear form:

$$\mathbf{L}_2\mathbf{c} = \mathbf{b} \tag{3.16a}$$

$$\mathbf{D}\mathbf{e} = \mathbf{c} \tag{3.16b}$$

$$\mathbf{L}_2^T\mathbf{x} = \mathbf{e} \tag{3.16c}$$

This means the Cholesky Decomposition as well as the forward elimination and back substitution can all be treated as solving a series of triangular linear equations (Yang, Peterson, et al., 2009), meaning the same hardware can be reused for each operation; Figure 3.7 depicts the full `trisolve` datapath for the Serial design, including the division and the recombination process to recover $\mathbf{L}_1$.

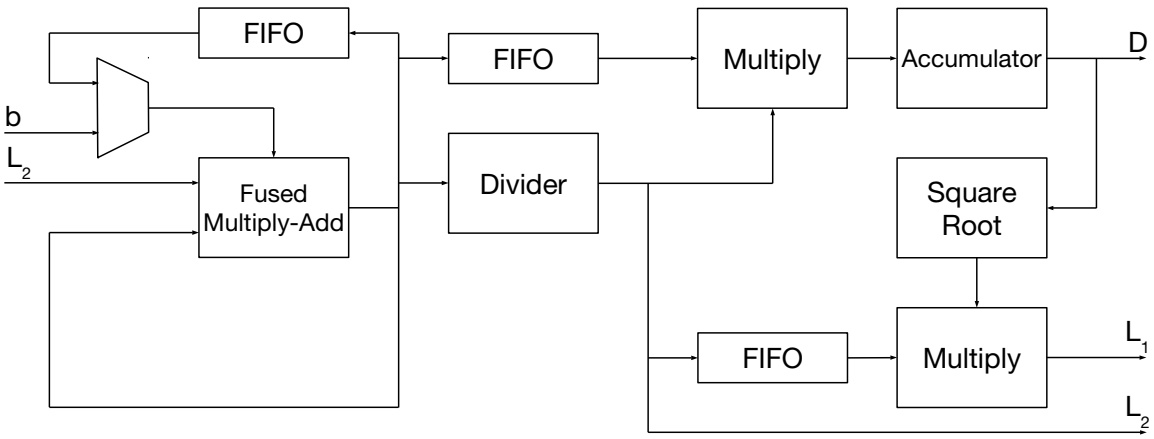


Figure 3.7: Triangular linear equations solver (`trisolve`) for the Serial design

The two inputs to the `trisolve` datapath feed into the fused multiply-add module and are denoted by $\mathbf{L}_2$ and $\mathbf{b}$. When performing the matrix 'right divide', these two inputs refer to the LDL Decomposition product and the dividend vector in (3.15), respectively. When performing the Cholesky Decomposition, $\mathbf{L}_2$ is the LDL Decomposition product calculated so far and $\mathbf{b}$ is actually a row of the target matrix $\mathbf{A}$ in (3.9).

Although in terms of the algorithm, the Decomposition proceeds by row, in terms of the hardware, all elements in a column are calculated at once in order to make full

use of the fused multiply-add pipeline. Consider the expanded form of (3.14):

$$\begin{aligned} y_1 &= a_1 \\ y_2 &= a_2 - F_{21}y_1 \\ y_3 &= a_3 - F_{31}y_1 - F_{32}y_2 \\ &\vdots \\ y_n &= a_n - \sum_{k=1}^{n-1} F_{nk}y_k \end{aligned} \tag{3.17}$$

For an $n \times n$ target matrix, the first iteration through the fused multiply-add pipeline calculates $n-1$ elements of the first column (i.e. $a_k - F_{k1}y_1,\ k = 2, \ldots, n$); the result is fed back through a FIFO to be used in the calculation for subsequent columns. The next iteration calculates the $n-2$ elements of the second column (i.e. $F_{k2}y_2,\ k = 3, \ldots, n$) which is added to the previous results and so on for the whole Decomposition. This means the fused multiply-add pipeline eventually becomes inefficient as the number of calculations necessary decreases by one every iteration.

The results of the fused multiply-add, F_{ij} (3.13a), are passed to the divider and a FIFO for two other calculations: the next row in the LDL Decomposition product $\mathbf{L}_2$ (3.11a) and the next element in the LDL diagonal product D_j (3.13b). Crucially, these two calculations occur in parallel with the calculation of F_{ij}, resulting in minimal delay in starting the calculation of the next row. The LDL diagonal product is calculated by multiplying the result of the divider with F_{ij} and accumulating it with the previous elements in the row as per (3.13b). After the LDL diagonal product is calculated, it and the LDL Decomposition product are used to recover the next row in the original Cholesky Decomposition product $\mathbf{L}_1$ via (3.10). This is necessary because of the subsequent matrix multiply-add between the augmented covariance square-root and the sigma weighting coefficient matrix (see (2.29)).

3.2.2.2 Matrix multiply-add

The matrix multiply-add datapath is a standard 'naive' element-wise multiplication and accumulation; in the 'naive' approach, for the expression:

$$\mathbf{R}^{m\times p} = \mathbf{A}^{m\times n}\mathbf{B}^{n\times p} + \mathbf{C}^{m\times p} \tag{3.18}$$

the element-wise calculation is given by:

$$R_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} + C_{ij} \tag{3.19}$$

The elements of the matrix to be added, **C**, can simply be injected into the accumulation directly, instead of performing an additional matrix addition after a matrix multiplication. A diagram of the datapath can be seen in Figure 3.8. The inputs are elements from the three matrices and the multiplexer handles the insertion of elements of **C** into the accumulation.

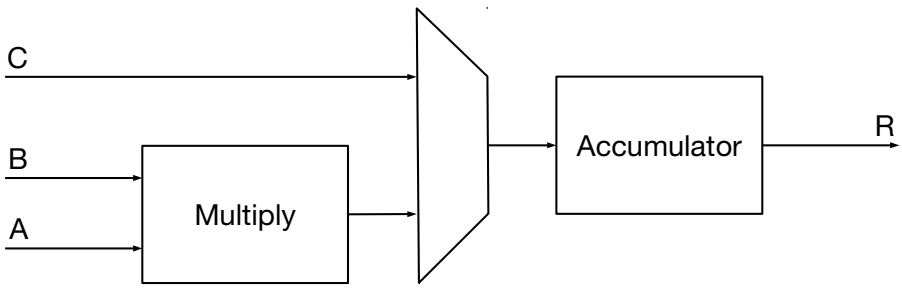


Figure 3.8: Matrix multiply-add operation for the Serial design

In order to calculate the whole matrix, a series of counter loops to fetch the appropriate elements in memory is used:

for $i = 1 : m$ **do**
  for $j = 1 : p$ **do**
    $R_{ij} = 0$
    for $k = 1 : n$ **do**
      $R_{ij} = R_{ij} + A_{ik}B_{kj}$

end for

$R_{ij} = R_{ij} + C_{ij}$

end for

end for

Because the Serial design aims to minimise resource use, these loops are NOT unrolled and so the datapath processes only one element of the result matrix, $\mathbf{R}$, at a time. The multiplication and accumulation dominate the calculation, so the time complexity of this 'naive' approach is $\mathcal{O}(mnp)$ operations. In order to maximise re-usability, a number of small tweaks to the datapath are made to handle other operations. The full list of supported operations is:

$$\mathbf{r} = \mathbf{Ab} + \mathbf{c} \tag{3.20a}$$

$$\mathbf{r} = \mathbf{AB} + \mathbf{C} \tag{3.20b}$$

$$\mathbf{R} = \mathbf{c} - \mathbf{Ab} \tag{3.20c}$$

$$\mathbf{R} = \mathbf{C} - \mathbf{AB} \tag{3.20d}$$

where $\mathbf{R}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are matrices and $\mathbf{r}$, $\mathbf{b}$, $\mathbf{c}$ are vectors. Multiplying and accumulating a matrix with vectors is really just the special case where $p = 1$. Subtracting the matrix multiplication result from the third matrix, $\mathbf{C}$, instead of accumulating it, simply involves negating the sign bit of the element-wise calculation, R_{ij}, using an `xor` operation.

3.2.2.3 Calculated mean/covariance

Calculating the mean and covariance of the transformed sigma points are both very similar, meaning both can be calculated by the same datapath. Consider the calculation for the mean of the `predict` step transformed sigma points:

$$\hat{\mathbf{x}}_k^- = \sum_{i=1}^{N} W_i \boldsymbol{\mathcal{X}}_i^x \tag{3.21}$$

This is a simple column-wise multiply-accumulation. Consider the calculation of the covariance:

$$\mathbf{P}_k^- = \sum_{i=1}^{N} W_i \left[\boldsymbol{\mathcal{X}}_i^x - \hat{\mathbf{x}}^-\right] \left[\boldsymbol{\mathcal{X}}_i^x - \hat{\mathbf{x}}^-\right]^T \tag{3.22}$$

The subtraction looks like it will cause inefficiencies in the datapath, similar to the division operation in the original Cholesky Decomposition. However, let $\tilde{\boldsymbol{\mathcal{X}}}_i = \boldsymbol{\mathcal{X}}_i^x - \hat{\mathbf{x}}^-$ be the i-th column of $\tilde{\boldsymbol{\mathcal{X}}}$, then the covariance calculation reduces to:

$$\mathbf{P}_k^- = \sum_{i=1}^{N} W_i \tilde{\boldsymbol{\mathcal{X}}}_i \tilde{\boldsymbol{\mathcal{X}}}_i^T \tag{3.23}$$

This 'sigma point residuals' matrix $\tilde{\boldsymbol{\mathcal{X}}}$ is of size $M_{state} \times N$ where M_{state} is the number of state variables and N is the number of sigma points as before. The element-wise calculation is then:

$$P_{ij}^- = \sum_{k=1}^{N} W_1 \tilde{\mathcal{X}}_{ik} \tilde{\mathcal{X}}_{jk} \tag{3.24}$$

This expression involves two multiplications followed by an accumulation; if these 'sigma point residuals' are calculated first with a `subtract` operation, then both the mean and covariance calculations simply involve a series of multiplications and accumulation. A diagram of this datapath for the Serial design can be seen in Figure 3.9.

The input to the datapath is either the transformed sigma points to calculate the mean, or the residuals to calculate the covariance. The FIFO is used to skip the first multiplication when calculating the mean; the multiplexer selects which value is calculated. To calculate the whole covariance matrix, the memory fetches are:

for $i = 1 : M_{state}$ **do**

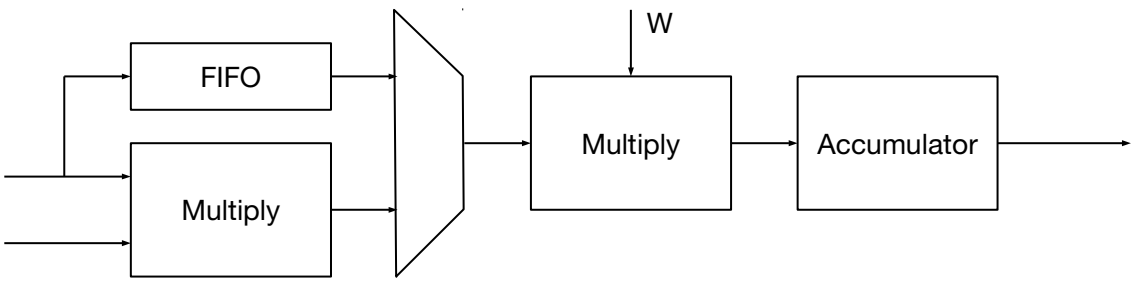


Figure 3.9: Calculate mean/covariance operation for the Serial design. W refers to the sigma points weights W_0, W_1 which are parameters (see Section 3.1.1).

for $j = 1 : M_{state}$ **do**
  $P_{ij}^{-} = 0$
  for $k = 1 : N$ **do**
    $P_{ij}^{-} = P_{ij}^{-} + W_1 \tilde{\mathcal{X}}_{ik} \tilde{\mathcal{X}}_{jk}$
  end for
end for
end for

with a time complexity of $\mathcal{O}(M_{state}^2 N)$. This module is also used to calculate the observation mean (2.20) and covariance (2.21) as well as the cross covariance (2.22) which are necessary in the `update` step. For the observation 'sigma point residuals', let $\tilde{\boldsymbol{\mathcal{Z}}}_i = \boldsymbol{\mathcal{Z}}_i - \hat{\mathbf{z}}$ be the i-th column of the $\tilde{\boldsymbol{\mathcal{Z}}}$ which is of size $M_{obs} \times N$ where M_{obs} is the number of observation variables. The observation covariance calculation proceeds in the same manner as the state covariance but with complexity $\mathcal{O}(M_{obs}^2 N)$ while the cross covariance has complexity $\mathcal{O}(M_{state} M_{obs} N)$.

3.2.3 Update step

The `update` step corrects the a priori state estimate with a set of observations to generate the new state estimate. Many of the calculations in the `update` step are very similar to the `predict` step; a block diagram of the `update` step architecture showing the data flow between hardware modules can be seen in Figure 3.10. The individual modules are reused from the `predict` step and the datapaths operate as

described in the previous sections (Sections 3.2.2.1, 3.2.2.2 and 3.2.2.3).

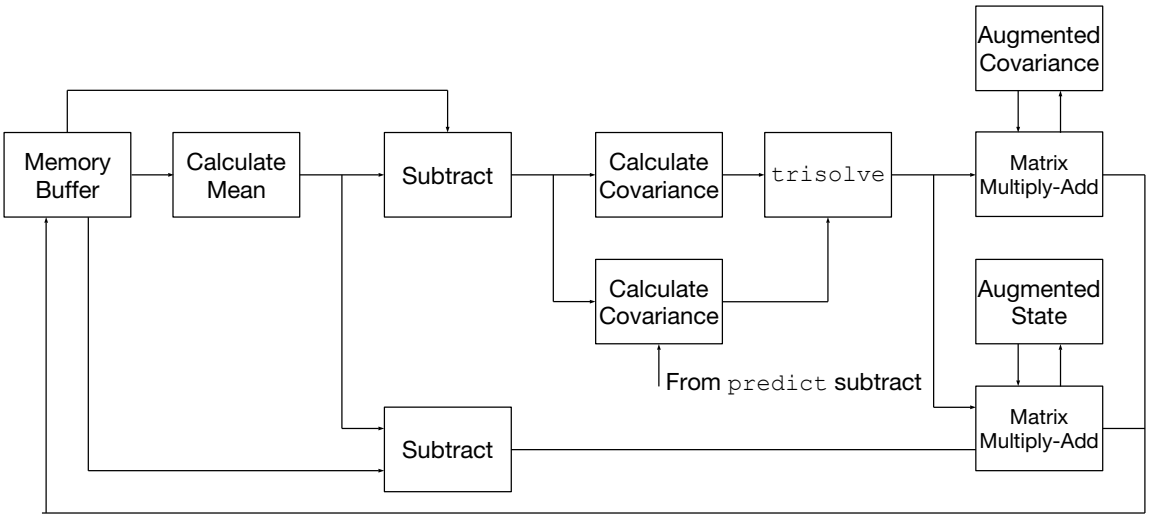


Figure 3.10: Block diagram of the `update` step for the Serial design

The `update` step starts with the copying of the current sensor observations, $\tilde{\mathbf{z}}$ (2.24), from the memory buffer into a local memory block; the IP core then requests the `update` transformed sigma points from the processor. Once the processor has placed the transformed sigma points in the memory buffer, the IP core is signalled to proceed and the transformed sigma points are used to calculate the observation mean (2.20). Similar to the `predict` step, the observation mean is used to calculate the `update` 'sigma point residuals' (`subtract`) before the covariances are calculated. The observation covariance (2.21) is calculated with the `update` 'sigma point residuals' and the cross covariance (2.22) is calculated with both the `predict` and `update` 'sigma points residuals'. The observation residual, $\tilde{\mathbf{z}} - \hat{\mathbf{z}}$ (2.24), is calculated with a `subtract` operation. Then the Kalman gain (2.23) is calculated by the matrix 'right divide' (`trisolve`). Finally, the new state estimate and covariance are calculated with the matrix multiply-add module as per (2.24) and (2.25b).

After the current state estimate and covariance are calculated, they are, like the `predict` step, written back to the processor memory buffer as well as, respectively, the augmented state and covariance local memory blocks. In this way, the `predict` and `update` steps can be performed asynchronously depending on the available sensor data and the processor is always up-to-date with the latest state estimate.

The process described in this section is summarised by a state diagram of the `update` step which can be seen in Figure 3.11. This state machine executes entirely within the `update` state in Figure 3.4. As with the `predict` step, the `wait` state is for when the processor is propagating the sigma points through the `update` model.

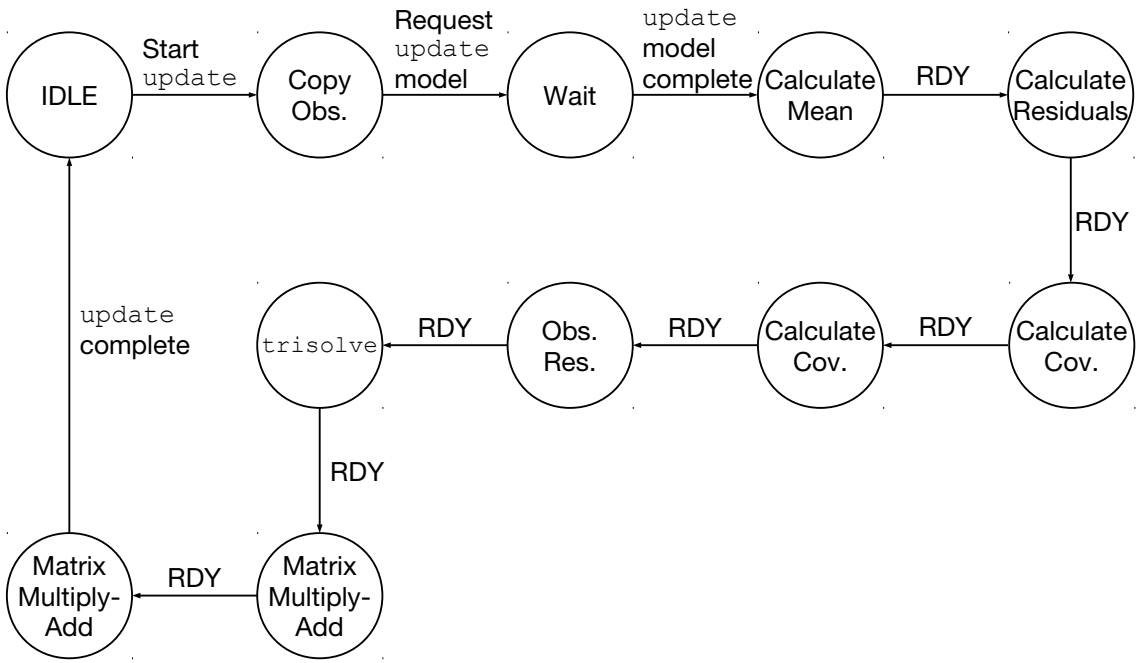


Figure 3.11: State diagram of the `update` step for the Serial design

3.3 Parallel design

In this section, the Parallel variant of the HW/SW UKF is introduced; the work in this section was first presented in Soh and Wu (2017b). The division of the UKF algorithm, which is slightly different to the Serial design, is outlined. The Parallel design also tweaks, rather than redesigns, aspects of the three major datapaths, secondary arithmetic, memory blocks and the control scheme that compose the IP core.

The Parallel design reintroduces the main benefit of hardware implementations: wide parallelism. This design strategy will use much more resources than the Serial design, but also increases performance. The design does so by encapsulating certain parts of

the major datapaths into a sub-module called a processing element (PE), then uses multiple instances of these PEs in parallel, allowing multiple elements of an algorithm to be calculated at once. The increase in resources used is not only for the extra processing elements, but also in the additional overhead needed to deal with the parallel memory structure that is also necessary to feed to the parallel processing elements. This overhead also means that the special case where the number of processing elements is 1, does not quite reduce to the Serial design; if a system designer wanted absolutely minimum resources used, it would still be better to use the Serial design. Nonetheless, the number of PEs used in the design is parameterisable, allowing for some trade-offs by the system designer between resources used and performance.

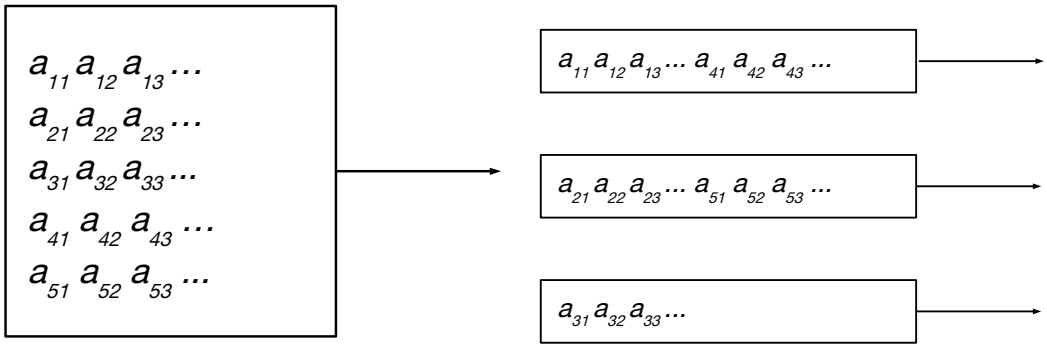
The Parallel design logically separates the UKF into three parts instead of the two that the Serial design uses. This is because, with the added parameter controlling the number of processing elements, slightly finer grained modules may become desirable to a designer. As with the Serial design, though the algorithm is logically divided into parts, the hardware implementation attempts to reuse as much of the resources as possible and so, at the HDL level, only separates the low-level functions into modules. The Parallel design contains the following modules and memories which are controlled via a large state machine:

- Modules
  - Memory prefetch
  - Triangular linear equations solver
  - Matrix multiply-add
  - Calculate mean/covariance
  - Calculate sigma point residuals (Subtract)
  - Memory serialiser
- Memory
  - Augmented state vector (Serial)

- Augmented covariance (Serial)
- Sigma weighting matrix (Serial)
- Cholesky decomposition (Parallel)
- State/Observation sigma point residuals (Parallel)
- State/Observation sigma point residuals (Serial)
- State mean/covariance (Parallel)
- Observation mean (Serial)
- Observation mean (Parallel)
- Observation residual (Serial)
- Observation covariance (Serial)
- Cross covariance (Serial)
- Cross covariance (Parallel)
- Kalman gain (Parallel)
- Kalman gain transposed (Serial)

Since there are additional processing elements per module now, the floating-point arithmetic modules can no longer be shared between functions as in the Serial design. Each of the modules instantiates their own floating-point arithmetic modules which means that, again, even in the special case where there is only one processing element, the Parallel design still uses more resources than the Serial design. Regardless of the number of processing elements, the floating-point arithmetic modules are still deeply pipelined with latencies described by Table 3.2 as in the Serial design.

There is now also some differences in the structure of the memory blocks in the Parallel design. A diagram of the different memory schemes can be seen in Figure 3.12. 'Serial' memory uses a single memory block; the 'Serial' designator here is in reference to the fact that only a single value of the stored matrix/vector can be accessed at one time. The memory instantiated in the Serial design is only structured in this way. The 'Parallel' memories use multiple memory blocks in parallel, controlled by the number



(a) Serial (b) Parallel

Figure 3.12: Memory structures in the Parallel design

of PEs parameter. The matrix/vector is spread out over these memory blocks such that multiple values of the matrix/vector may be accessed at once.

It is necessary in many cases to be able to convert between the two memory schemes so two new modules are introduced: a memory 'prefetch', which fetches data from a serial memory block and places it into parallel memory blocks, and a memory 'serialiser', which collects data from calculations in a parallel scheme and outputs it in a sequential fashion for storage in a serial memory block. The main need for this is the interaction between the memory buffer and the IP core. The memory buffer is necessarily a serial memory block as the processor handles memory access in a sequential manner but multiple values need to be fed to the additional processing elements in each module for them in order for them to be useful.

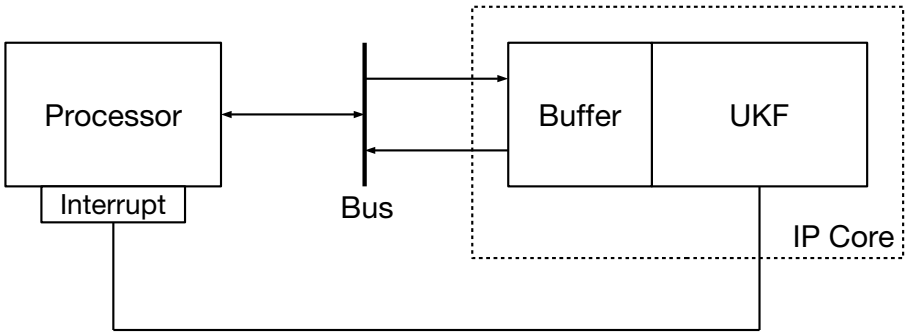


Figure 3.13: Top-level block diagram of the Parallel design

The top-level block diagram of the Parallel design can be seen in Figure 3.13. The

top-level design is very similar to the Serial design but here, the control scheme has changed: instead of having digital control lines, the control register has been incorporated into the memory map of the memory buffer; a single interrupt line remains. Instead of a simple FIFO, the memory buffer for the Parallel design has a proper internal memory map to ensure the control information and data is coherent between the processor and the IP core; the full memory map can be seen in Figure 3.14. Similar to the control lines in the Serial design, the control register allows the processor to reset or enable the IP core as a whole as well as start one of the core's functional steps via a state machine. The control register also records the current state of the IP core. Data required by the IP core (e.g. transformed sigma points) must be placed in the memory buffer at the appropriate address by the processor before signalling the IP core to begin its calculations.

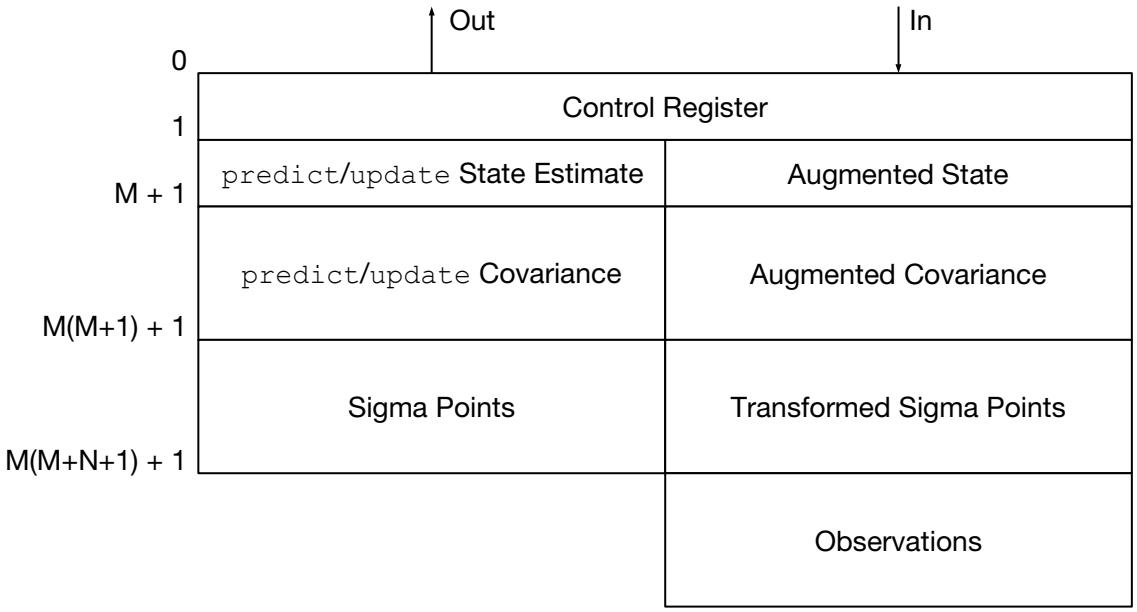


Figure 3.14: Memory map for the Parallel design. The exact addresses (left) are dependant on the selection of certain parameters (see Section 3.1.1). As in Section 2.5.2, M is the length of the augmented state vector and N is the number of sigma points. In/Out is with respect to the IP core; so In: PS $\rightarrow$ IP core and Out: PS $\leftarrow$ IP core

The control register may be polled by the processor to control the IP core; alternatively, the core may also be configured with an optional interrupt line that may be

attached to the processor's interrupt controller or external interrupt lines. Values in the control register can be seen in Table 3.4.

No.	I/O	Function
0	O	NC
1	O	Idle
2	O	Interrupt flag
3	O	UKF initialised
4	O	`sig_gen` step complete
5	O	`predict` step complete
6	O	`update` step complete
7	O	NC
8	I	Reset
9	I	Enable
10	I	Interrupt clear
11	I	Initialise UKF
12	I	Start `sig_gen` step
13	I	Start `predict` step
14	I	Start `update` step
15	I	NC

Table 3.4: Control register for the Parallel design. I/O is with respect to the IP core; so I: PS $\rightarrow$ IP core and O: PS $\leftarrow$ IP core. NC = Not connected.

3.3.1 State machine

The IP core is controlled by a state machine which has five states: `idle`, `init`, `sig_gen`, `predict` and `update`; see Figure 3.15. The IP core waits in the `idle` state for instructions from the processor before beginning one of the UKF steps. The processor sets the relevant bit in the control register to transition to a new state and once the UKF step is done, the relevant bit in the control register is set before the IP core transitions back to `idle`.

During the `init` state, the processor initialises the internal memory of the IP core with initial values for the augmented state and covariance; this is the same process as in the Serial design. The `sig_gen` state handles the calculation of the latest set of sigma points. After the new sigma points have been propagated through the `predict`

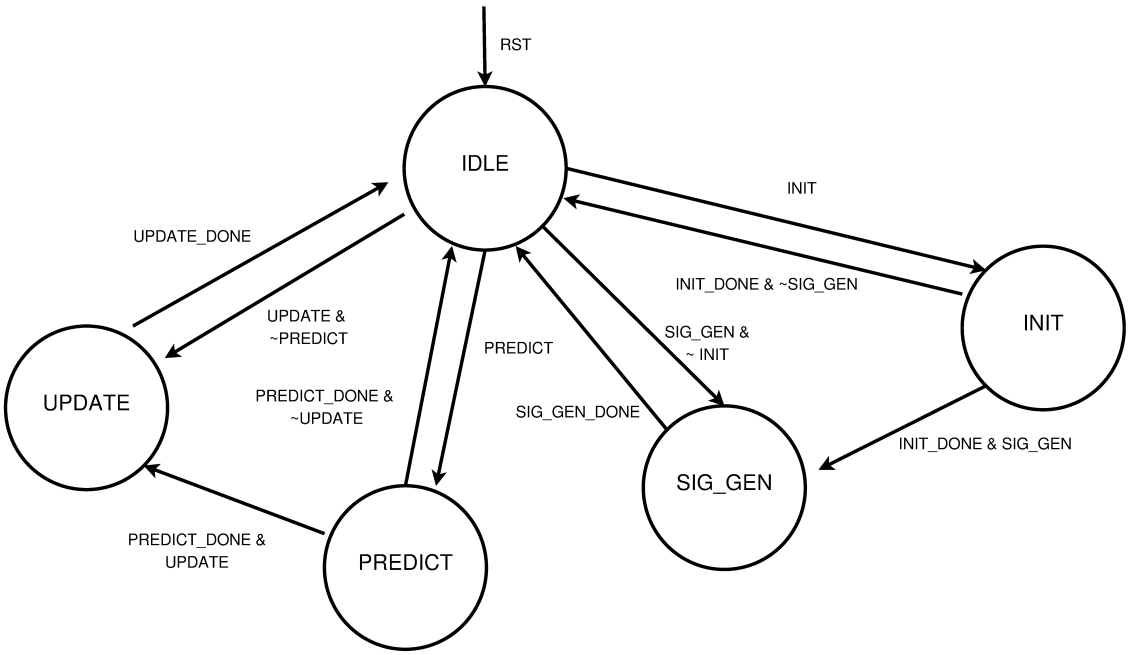


Figure 3.15: Top-level state diagram for the Parallel design

model, the `predict` state uses the transformed sigma points to calculate the a priori state and covariance. Similarly, the `update` state uses the `update` transformed sigma points to calculate the current state and covariance.

The `init` state may be performed in conjunction with the `sig_gen` step to either initialise new, or reset old, state estimates. Otherwise the `sig_gen` step utilises previously calculated values for the augmented state and covariance (which are stored in the internal memory). Similarly, the `predict` and `update` steps may be performed together if valid observations are available, or independently as required.

3.3.2 Sigma points generation

The functionality of the `sig_gen` is the same as the first half of the `predict` step from the Serial design (i.e. before the processor is signalled to propagate the sigma points): taking the matrix 'square-root' of the augmented covariance, then multiplying the result by a weighting matrix, before adding the augmented state column-wise. After the sigma points are calculated they are written to the memory buffer, a control bit

is set to signify completion to the processor and, if the interrupt line is included, an interrupt generated. A block diagram of this step can be seen in Figure 3.16 showing the data flow between modules.

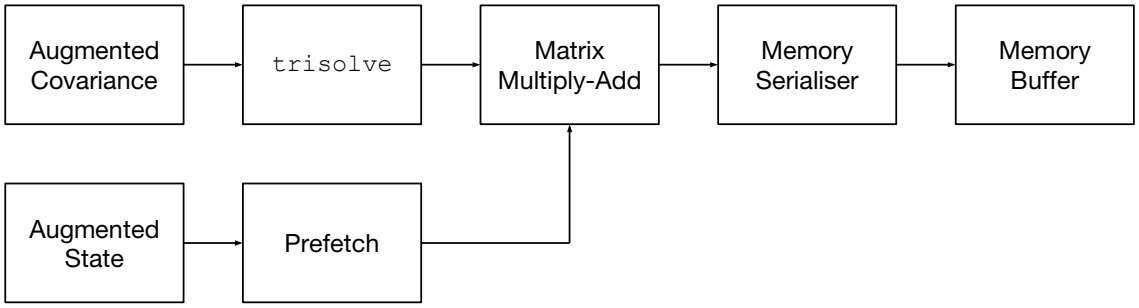


Figure 3.16: Block diagram of the `sig_gen` step for the Parallel design

The main difference from the Serial design (cf. Figure 3.5) is the need to introduce a memory prefetch as well as a memory serialiser module which adds a small amount of overhead to the `sig_gen` step; these two modules are necessary due to the matrix multiply-add now featuring a parallelised datapath. The `trisolve` and matrix multiply-add modules are functionally the same as the Serial design but have small tweaks, described in the next two sections, to implement the parallelisation.

A state diagram summarising the process for the `sig_gen` step can be seen in Figure 3.17; like in the Serial design, the state machine is necessary so that the hardware for the `trisolve` and matrix multiply-add module can be reused later. This state machine executes entirely within the `sig_gen` state in Figure 3.15.

3.3.2.1 Triangular linear equations solver

The triangular linear equations solver for the Parallel design is functionally the same as the `trisolve` module in the Serial design, implementing the alternate LDL Decomposition. However, for the Parallel design, the fused multiply-add module and feedback FIFO has been encapsulated to form a processing element which can be instantiated multiple times in parallel; the `trisolve` datapath for the Parallel design can be seen in Figure 3.18. The PEs now output to a demultiplexer which ensures

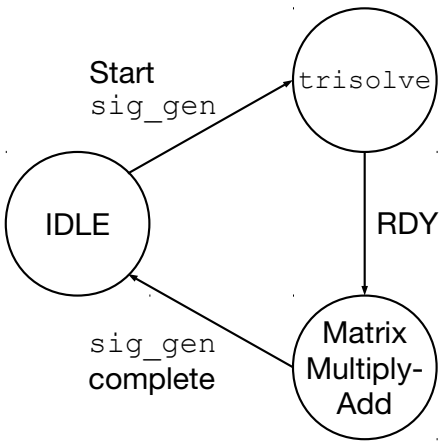


Figure 3.17: State diagram of the `sig_gen` step for the Parallel design

values are passed to the subsequent calculations in the correct order. The three latter calculations, the LDL Decomposition product (3.11a), the diagonal product (3.13b) and the original Decomposition product (3.7a), are not parallelised because these calculations require much fewer operations and so parallelisation is not necessary; the second 'half' of the datapath dealing with these three calculations operates in exactly the same way as the Serial design.

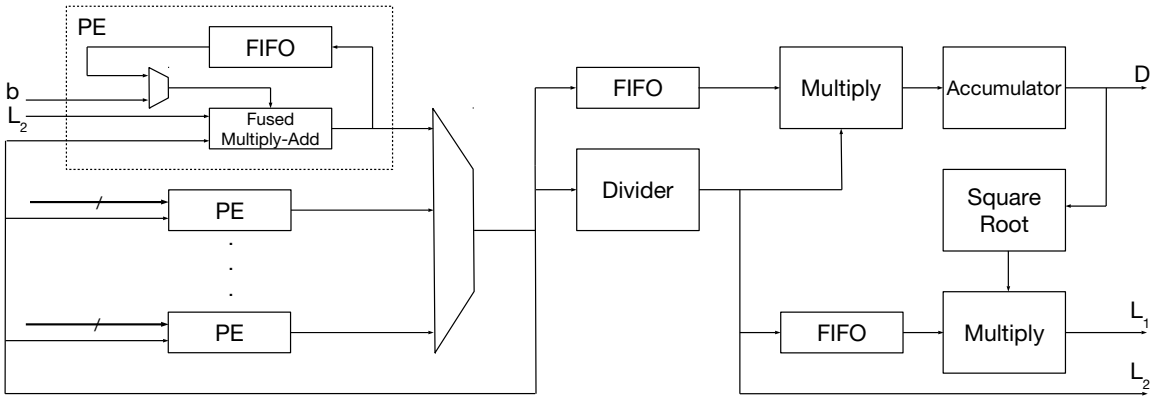


Figure 3.18: Triangular linear equations solver for the Parallel design

Recalling the expanded triangular system given by (3.17), the additional processing elements in this design calculate elements of the current column in parallel. For example, in the 2 PE case, $F_{21}y_1$ and $F_{31}y_1$ are calculated in parallel. As noted in the Serial design, the processing element pipeline quickly becomes inefficient as one less calculation is necessary each iteration; this problem is exacerbated in the

Parallel design as more elements are calculated per iteration. Also noted in the Serial design is that the calculation of any given row of the Cholesky Decomposition requires the values of the all rows before it. Due to this data dependency, row calculations cannot be calculated independently and thus the Cholesky Decomposition cannot be parallelised effectively. However, this is not an issue with forward elimination or back substitution when solving a linear triangular system. Solving the system given by (3.16) results in a vector so, if the dividend of the matrix 'right' divide is also a matrix (of size $m \times n$, e.g. see (2.23)), then m forward eliminations and back substitutions are required. Crucially these operations *are* independent, so the `trisolve` datapath can be properly pipelined and the back substitution and forward elimination effectively parallelised. The issue with the Cholesky Decomposition is sufficient that additional processing elements in this datapath may not be as useful as expected; despite the reshuffle of operations and multiple PEs, the `trisolve` module still has the potential to limit performance of the IP core.

3.3.2.2 Matrix multiply-add

The matrix multiply-add module only has a minor tweak compared to the Serial design: the entire datapath from the Serial design (cf. Figure 3.8) has been enclosed as one processing element and additional PEs are added to handle calculations in parallel; the matrix multiply-add datapath for the Parallel design can be seen in Figure 3.19. Each PE is responsible for calculating at least one row of the result matrix.

The following loops account for the whole matrix:

for $i = 1 : N_{PE} : m$ **do**
  for $j = 1 : p$ **do**
    $R_{ij} = 0$
    $\dots$
    $R_{xj} = 0$
    for $k = 1 : n$ **do**
      $R_{ij} = R_{ij} + A_{ik}B_{kj}$

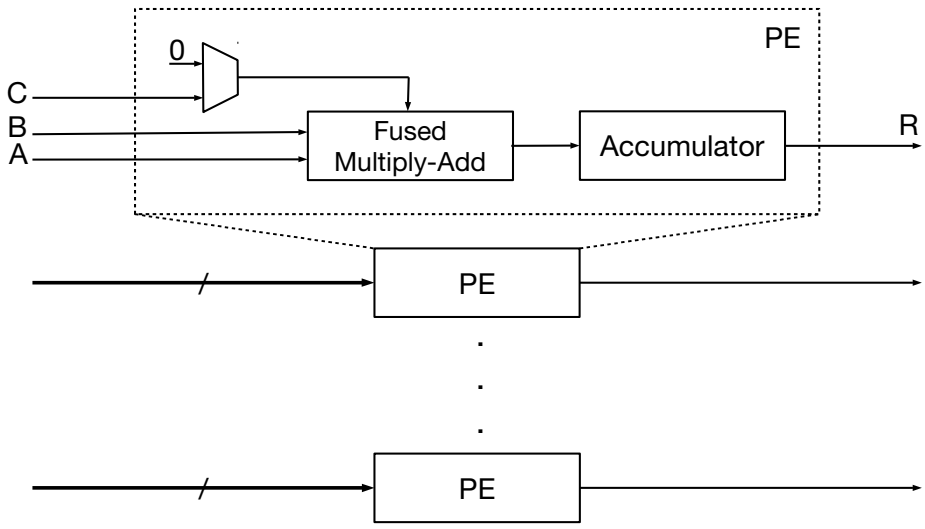


Figure 3.19: Matrix multiply-add operation for the Parallel design

$\ldots$

$R_{xj} = R_{xj} + A_{xk}B_{kj}$

end for

$R_{ij} = R_{ij} + C_{ij}$

$\ldots$

$R_{xj} = R_{xj} + C_{xj}$

end for

end for

where N_{PE} is the number of processing elements and $x = i + N_{PE} - 1$. This parallelisation reduces the complexity of the module to $\mathcal{O}(mnp/N_{PE})$ and, in the case where $N_{PE} \geq m$, the complexity is reduced to just $\mathcal{O}(np)$. The supported operations by this module are the same as in the Serial design.

3.3.3 Predict step

The `predict` step for the Parallel design encompasses the second half of the `predict` step from the Serial design (i.e. after the new sigma points have been propagated through the `predict` model); the architecture for the `predict` step can be seen in

Figure 3.20 showing how data flows between each module.

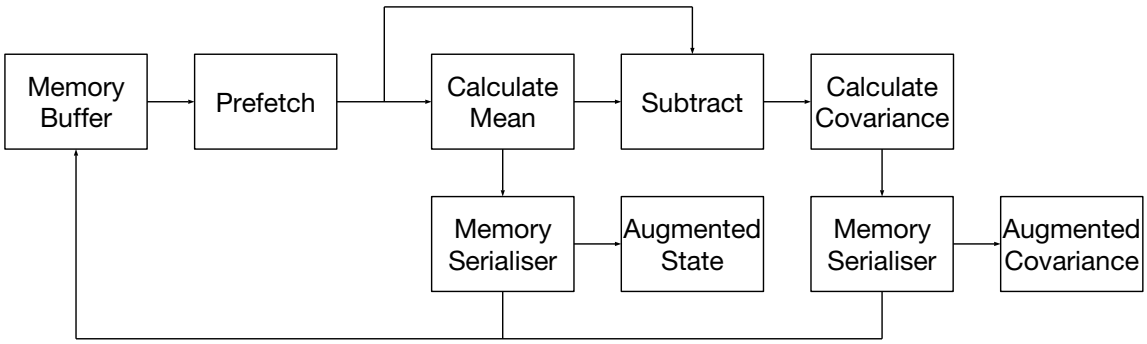


Figure 3.20: Block diagram of the `predict` step for the Parallel design

The processor may initiate a `predict` step once it has placed valid transformed sigma points into the memory buffer. The prefetch module fetches the transformed sigma points from the memory buffer and places them into a parallel memory structure. As with the Serial design, the mean of the transformed sigma points is written back to both the augmented state vector memory and the memory buffer as the a priori state estimate. The sigma point residuals are once again calculated first before the covariance calculation. The new covariance is written back to the augmented covariance memory and the memory buffer as the a priori covariance. Memory serialisers are necessary after the mean and covariance calculation as the memory buffer is a serial memory (see Figure 3.12). As with the `sig_gen` step, once the `predict` step is completed a control bit is set to notify the processor and, if included, an interrupt generated.

A state diagram of the Parallel `predict` step can be seen in Figure 3.21. This state machine includes an additional state for the prefetch operation and occurs within the `predict` state in Figure 3.15.

3.3.3.1 Calculation of mean/covariance

Similar to the matrix multiply-add operation, the module for calculating the mean and covariance in the Parallel design merely encapsulates the datapath from the Serial design (cf. Figure 3.9) into one processing element then adds additional PEs to the

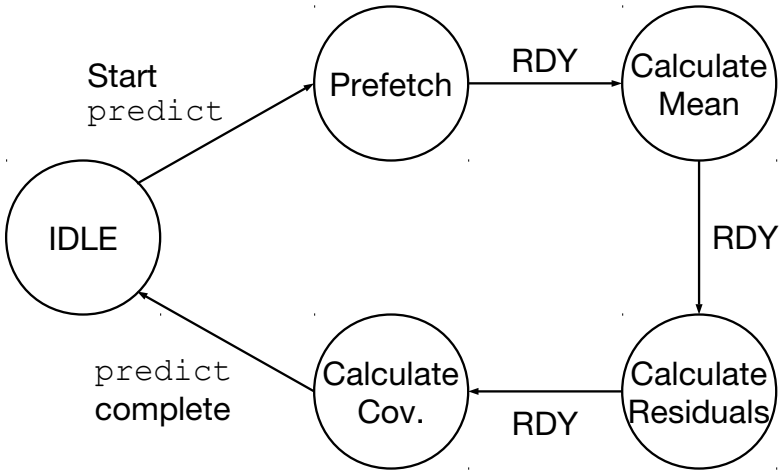


Figure 3.21: State diagram of the `predict` step for the Parallel design

datapath in order to calculate additional rows in parallel; the datapath for the Parallel design can be seen in Figure 3.22.

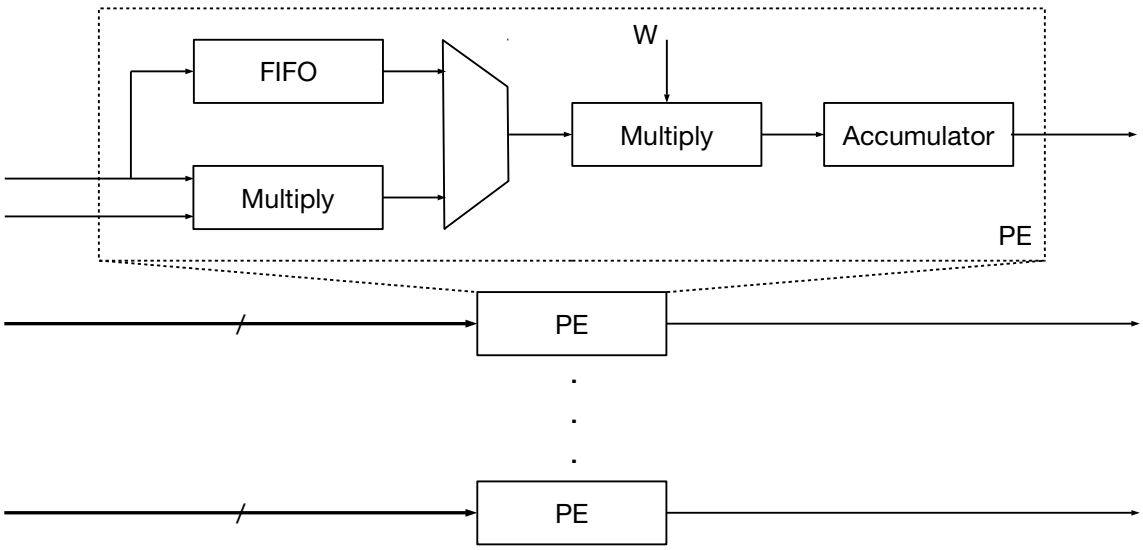


Figure 3.22: Calculate mean/covariance operation for the Parallel design. W refers the sigma points weights W_0, W_1 which are parameters (see Section 3.1.1).

The memory fetch loop for the state covariance is:

for $i = 1 : N_{PE} : M_{state}$ **do**

  for $j = 1 : M_{state}$ **do**

    $P_{ij}^{-} = 0$

$\ldots$

$P^-_{xj} = 0$

for $k = 1 : N$ **do**

$P^-_{ij} = P^-_{ij} + W_1 \tilde{\mathcal{X}}_{ik} \tilde{\mathcal{X}}_{jk}$

$\ldots$

$P^-_{xj} = P^-_{xj} + W_1 \tilde{\mathcal{X}}_{xk} \tilde{\mathcal{X}}_{jk}$

end for

end for

end for

where $x = i + N_{PE} - 1$. The time complexity of this operation is $\mathcal{O}(M^2_{state} N / N_{PE})$ which further reduces to $\mathcal{O}(M_{state} N)$ if $N_{PE} \geq M_{state}$. A similar process occurs for the observation covariance and cross covariance.

3.3.4 Update step

The `update` step for the Parallel design is very similar to the `update` step in the Serial design (cf. Figure 3.10) but has some small tweaks to accommodate the parallel memory structures; the architecture for the `update` step can be seen in Figure 3.23 showing the data flow between modules.

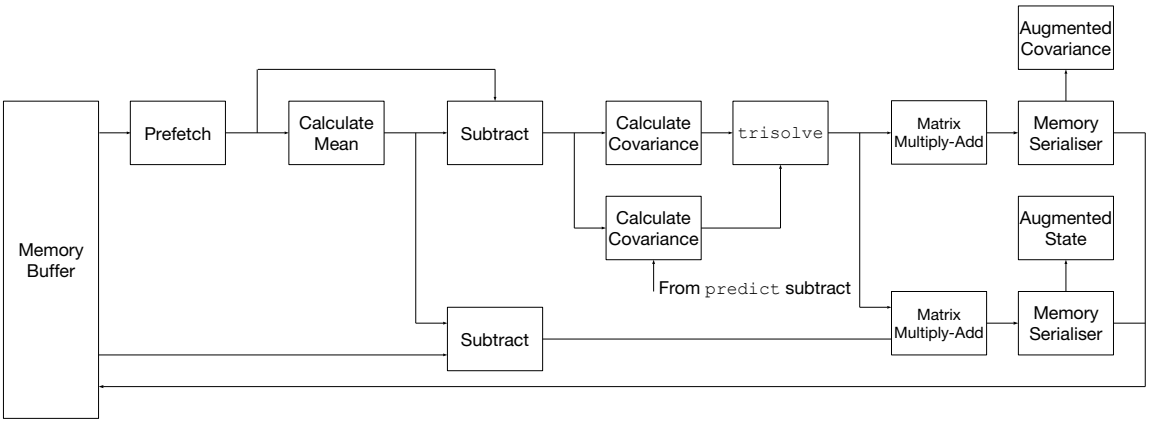


Figure 3.23: Block diagram of the `update` step for the Parallel design

As with the `predict` step, the processor must first place the valid transformed sigma

points into the memory buffer before signalling the IP core to begin. First, the prefetch module converts the transformed sigma points into a parallel memory structure. The mean and 'sigma point residuals' are calculated, then used to calculate the observation covariance. The `update` 'sigma point residuals' are also combined with the `predict` 'sigma point residuals', which were calculated during the `predict` step, to calculate the cross covariance between the two system models. The observation residual, $\tilde{\mathbf{z}} - \hat{\mathbf{z}}$ (2.24), is calculated with the current set of observations in the memory buffer. The observation and cross covariance are used to calculate the Kalman gain before the matrix multiply-add modules use the Kalman gain and the a priori state estimate and covariance to calculate the new state estimate and covariance. The new estimates overwrite the a priori estimates in the internal memory and are also written into the memory buffer such that both the core and the processor have the most recent estimate. The core notifies the processor upon completion, setting a control bit and/or generating an interrupt.

A state diagram of the `update` step illustrating the described process for the Parallel design can be seen in Figure 3.24. This state machine occurs within the `update` state in Figure 3.15. The state machine for the Parallel `update` step is very similar to the Serial design (cf. Figure 3.11) but with an additional state to handle the prefetch module and instead of a `wait` state, the IP core assumes the processor has already handled the transformation of the sigma points.

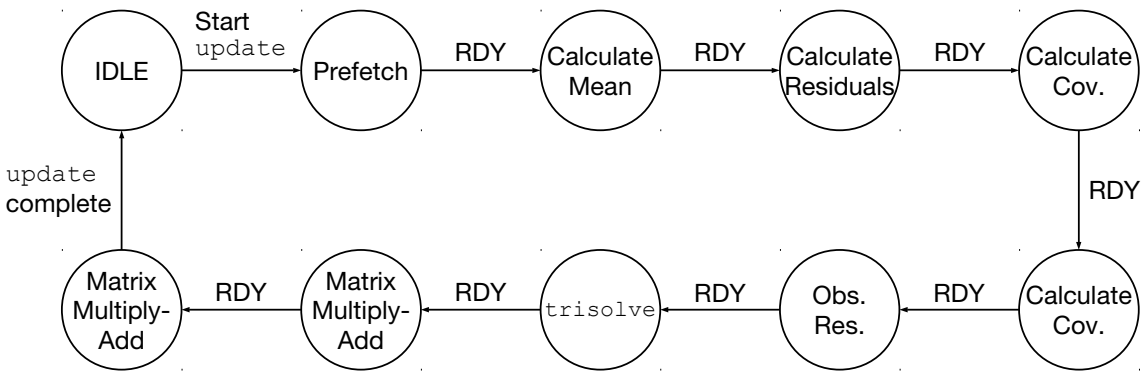


Figure 3.24: State diagram of the `update` step for the Parallel design

3.4 Pipeline design

In this section, the Pipeline variant of the HW/SW codesign is introduced; this section is based on work first presented in Soh and Wu (2017a). The three major datapaths remain the same as in the Parallel design; however, changes to the division of the UKF algorithm and hardware implementation, in order to accommodate calculation of multiple UKF instances simultaneously, are described.

The Pipeline design reinforces the main benefit of hardware implementations – wide parallelism – with a 'high-level' pipeline to increase performance even further. This design strategy uses the most resources but also has the highest performance in terms of algorithm throughput, though not necessarily in terms of algorithm latency.

The Pipeline design makes use of the parallelised datapaths as described in the Parallel design and retains the same logical separation of the UKF with three major steps: `sig_gen`, `predict`, `update`. However, unlike the previous two designs, the Pipeline design also sections the UKF hardware into multiple parts at an HDL level, meaning no hardware is reused between sections. This allows each section to run independently and in parallel, a necessity for a hardware pipeline. The top-level block diagram of the Pipeline design can be seen in Figure 3.25. The IP core features a memory buffer and three sub-modules, one for each of the major steps.

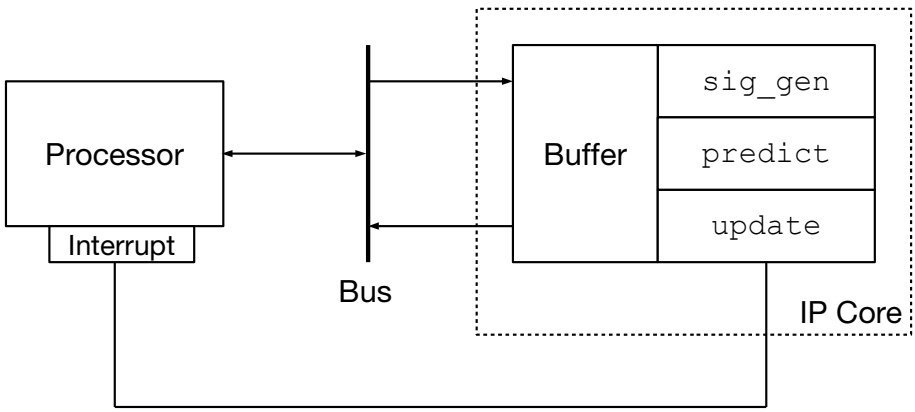


Figure 3.25: Top-level block diagram of the Pipeline design

In order to accommodate sections of the UKF running independently, the memory

buffer is itself sectioned into three parts as well; the memory map for the Pipeline design can be seen in Figure 3.26. The IP core modules assume that valid data is in the correct section of the memory buffer once any of the steps has been signalled to start; the processor must place the required data (e.g. transformed sigma points) in the appropriate section beforehand.

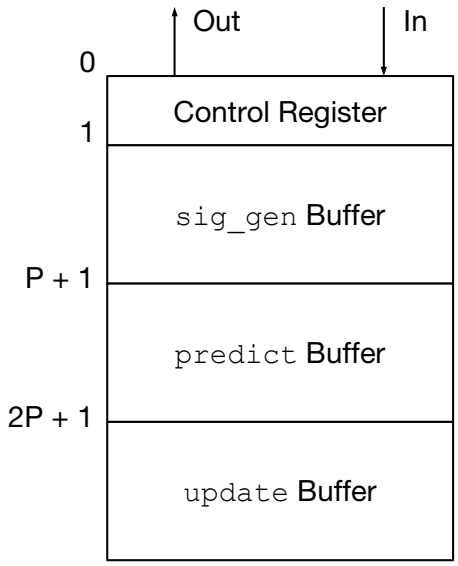


Figure 3.26: Memory map for the Pipeline design. The exact addresses (left) are dependant on the parameter P, which controls the depth of the memory buffer (see Section 3.1.1). In/Out is with respect to the IP core; so In: PS $\rightarrow$ IP core and Out: PS $\leftarrow$ IP core.

The control scheme is similar to the Parallel design, where a control register at the top of the memory buffer. The control register allows the processor to reset or enable the IP core as well as to start or stop any of the three core modules as desired; the control register also records the current state of the three modules as well as the IP core as a whole. The control register may be polled by the processor to control the IP core or use an optional interrupt line that may be attached as one of the the processor's external interrupt sources. The IP core generates an interrupt to signify the completion of calculations from any of the three modules as well as readiness to accept new data. The control register can be seen in Table 3.5.

With the three sub-modules able to operate independently, the IP core may be used

No.	I/O	Function
0	O	NC
1	O	Idle
2	O	Interrupt flag
3	O	`sig_gen_a` step complete
4	O	`sig_gen_b` step complete
5	O	`predict` step complete
6	O	`update` step complete
7	O	NC
8	I	Reset
9	I	Enable
10	I	Interrupt clear
11	I	NC
12	I	Start `sig_gen` step
13	I	Start `predict` step
14	I	Start `update` step
15	I	NC

Table 3.5: Control register for the Pipeline design. I/O is with respect to the IP core; so I: PS $\rightarrow$ IP core and O: PS $\leftarrow$ IP core. NC = Not connected.

as a pipeline where the `sig_gen` module is able to accept new data while the `predict` step is still calculating the previous estimate; the pipeline stages can be seen in Figure 3.27. The `sig_gen` step contains two large matrix operations – `trisolve` and the matrix multiply-add – which can lead to long calculation times for large matrices; because of this, the `sig_gen` step is broken up into two stages – the first two stages – for the pipeline. The third stage is the software 'stage' where the processor propagates the sigma points through the system models. The final two stages are simply for the `predict` and `update` steps. The selection of these five stages was due to the logical separation of the UKF allowing for easy control of the IP core by the processor; ease of control was prioritised over hardware efficiency so the performance of the pipeline may not necessarily be as high as it could be.

Each of the three sub-modules are parameterisable to use multiple PEs, within their respective datapaths, at the designer's discretion. Since no hardware is reused, there is no longer a need for a state machine to control access/data flow for individual modules. The organisation of memory blocks in the Pipeline design is the same as

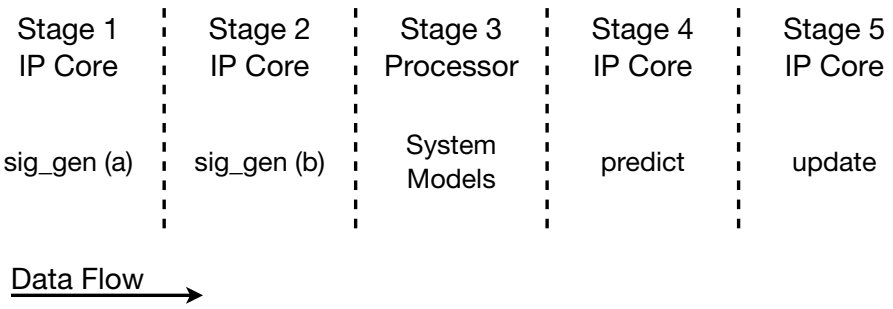


Figure 3.27: Stages of the five-stage UKF pipeline

in the Parallel design with 'serial' and 'parallel' memory blocks (see Figure 3.12); additionally, some memory blocks are FIFOs which are explained further in the next three sections. The full list of instantiated modules and memories used by the Pipeline design are:

- Modules
  - Sigma points generation
    - Memory prefetch
    - Triangular linear equations solver
    - Matrix multiply-add
    - Memory serialiser
  - Predict
    - Memory prefetch
    - Calculate mean/covariance $\times 2$
    - Calculate sigma point residuals (Subtract)
    - Memory serialiser
  - Update
    - Memory prefetch $\times 2$
    - Calculate mean/covariance $\times 3$
    - Calculate sigma point residuals (Subtract)

    - Triangular linear equations solver
    - Matrix multiply-add ×2
    - Memory serialiser ×3

- Memory
  - Sigma weighting matrix (Serial)
  - State sigma points residuals (FIFO)
  - `predict` state estimate/covariance (FIFO)
  - Sigma points generation
    - Cholesky decomposition (Parallel)
  - Predict
    - State mean (Parallel)
    - State sigma point residuals (Parallel)
  - Update
    - State sigma points residuals (FIFO)
    - `predict` state estimate/covariance (FIFO)
    - Observation mean (Parallel)
    - Observation sigma points residuals (Parallel)
    - Observation covariance (Serial)
    - Cross covariance (Parallel)
    - Observation residual (Serial)
    - Kalman gain (Serial)

3.4.1 Sigma points generation

A block diagram of this module showing the flow of data can be seen in Figure 3.28. The `sig_gen` module for the Pipeline design is very similar to the Parallel design (cf. Figure 3.16) except that there are no longer local/internal memory blocks for

the augmented state and covariance, and the module operates in two stages for the pipeline (see Figure 3.27).

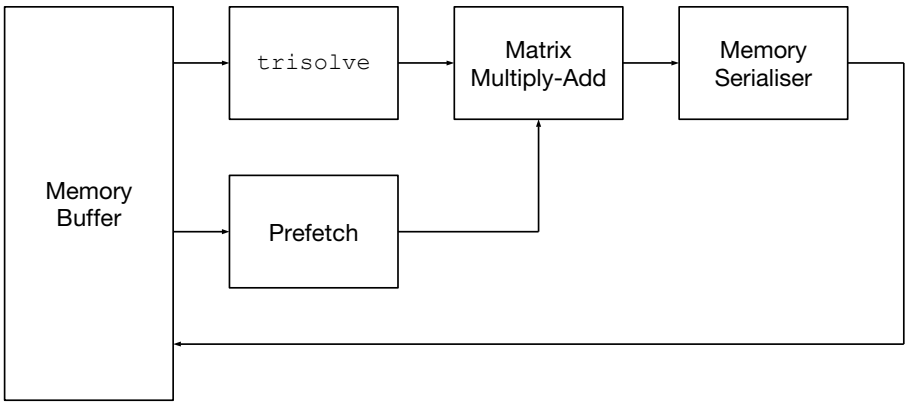


Figure 3.28: Block diagram of the `sig_gen` step for the Pipeline design

To start the `sig_gen` module, the processor must first place the current augmented state and covariance estimate into the memory buffer. The first stage (`sig_gen` (a)) contains the matrix 'square-root' and a prefetch module to hold the augmented state vector. The second stage (`sig_gen` (b)) contains just the matrix multiply-add. The `sig_gen` module is able to accept new data (i.e. the augmented state and covariance of another UKF instance) once the first stage (`sig_gen` (a), i.e. the `trisolve` module) has completed. The control register contains control bits to start the `sig_gen` module as well as bits to signify the completion of either pipeline stage; an interrupt is also generated when either stage has completed. The datapath for the triangular linear equations solver (`trisolve`) and the matrix multiply-add modules are the same as described in the Parallel design (Sections 3.3.2.1 and 3.3.2.2 respectively).

3.4.2 Predict step

The architecture for the `predict` step can be seen in Figure 3.29 showing the data flow between modules. The functionality of the `predict` step in the Pipeline design is also very similar to the Parallel design (cf. Figure 3.20) except that the a priori state estimate and covariance are output to a FIFO (in addition to the memory buffer). This is because these values are necessary during calculations in the `update` step and

there are no longer any local memory blocks for the augmented state and covariance; once the values are output into the FIFO, the `predict` module can continue with the next UKF instance without losing any data. Similarly, the `predict` 'sigma point residuals' are necessary for calculation of the cross covariance (see (2.22)) and are output to a FIFO as well.

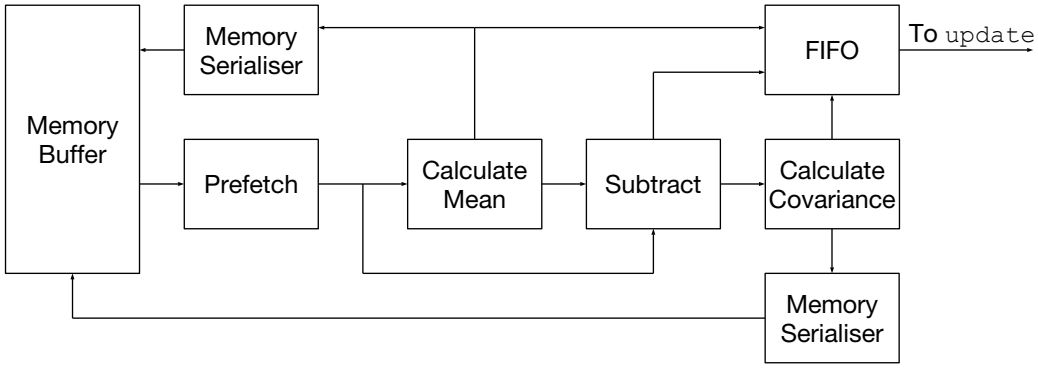


Figure 3.29: Block diagram of the `predict` step for the Pipeline design

The processor may initiate a `predict` step once it has placed a valid set of transformed sigma points into the memory buffer. The processor must propagate the sigma points, generated from the `sig_gen` module, through both the `predict` model as well as the `update` model as both the `predict` and `update` steps are calculated together in succession (they are the two final stages of the pipeline). As with the `sig_gen` step, the control register contains control bits the processor may use to start the `predict` step and record its completion; an interrupt is generated on completion as well. The datapath for the calculation of the mean/covariance is also the same as in the Parallel design (see Section 3.3.3.1).

3.4.3 Update step

The architecture for the `update` step in the Pipeline design, showing the data flow between modules, can be seen in Figure 3.30. The `update` module is functionally similar to the Parallel design (cf. Figure 3.23) but has some key practical differences since none of the hardware is reused; there is also the additional FIFO with intermediate values from the `predict` step.

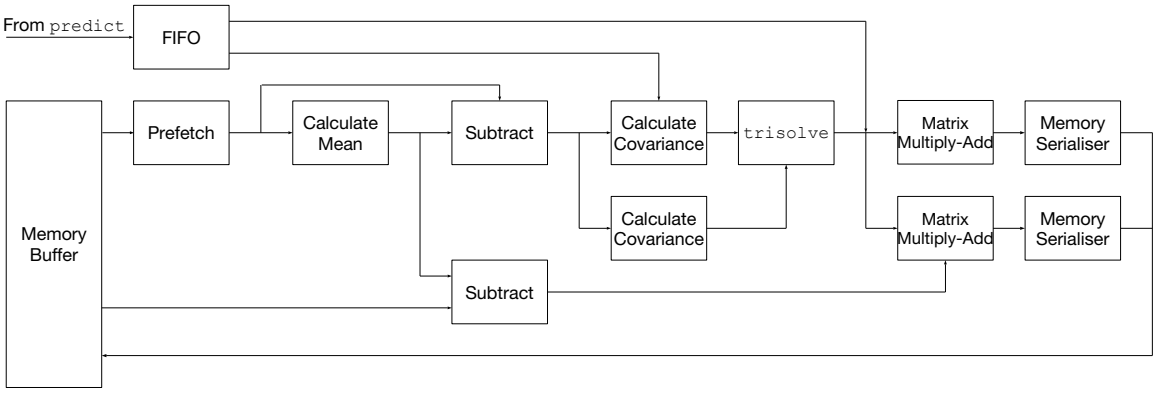


Figure 3.30: Block diagram of the `update` step for the Pipeline design

The `update` step proceeds immediately after the `predict` step (as the fifth and final stage of the pipeline). As with the Parallel design, a prefetch stage converts the transformed sigma points into a parallel memory structure then calculates the mean and 'sigma point residuals'. However, in the Pipeline design, the two covariance calculations occur in parallel (the calculation covariance module is instantiated twice). These two covariances are used to calculate the Kalman gain which is then used, along with the a priori state estimate and covariance, to calculate the new state estimate and covariance; for this calculation as well, the matrix multiply-add module is instantiated twice and the two matrix multiply-add calculations occur in parallel. Finally, the new state estimate is written to the memory buffer for the processor to collect. The control register contains a control bit signifying completion of the `update` step only and, as usual, an interrupt is generated on completion.

3.5 Summary

In this chapter, the FPGA-based HW/SW codesign of the UKF was presented. The partitioning strategy for the UKF was introduced, splitting the UKF into application-specific and non-application-specific parts; the method of implementing these two parts on the FPGA was also described. Implementing the application-specific parts in software that runs on a processor allows the design to retain flexibility and portability, while implementing the rest of the algorithm in hardware as an IP core allows for

the acceleration of large and cumbersome matrix operations, increasing performance. A parameterisation scheme for the codesign is outlined which allows the IP core to handle different applications with different vector/matrix sizes. These parameters are collected into a parameter file which is used to generate header files that are included into the hardware and software source files so that both parts know how to interpret the data passed between them.

To maximise the flexibility of the codesign, three variants were presented. The Serial design aims to use the least amount of hardware resources possible and so does not make use of parallelism in its main datapaths. This is so that the Serial design can be easily integrated into larger SoC or fault-tolerant systems but also means that the performance boost may be modest. The Parallel design does use parallelism in the main datapaths which increases performance but will also increase the amount of resources used. With the additional resource usage, the Parallel design may only be feasible as a coprocessor unless high-end FPGAs are used. The Pipeline design features parallelism not only in the main datapaths but also at the top-level, allowing multiple instances of the UKF to be calculated at once. The Pipeline design has the highest performance but will also use the most amount of resources and so is intended for use as a coprocessor only.

The approach presented means that the codesign is highly flexible and can be easily ported to any application. Once a system designer has formulated the UKF for an application, the application-specific parts are developed in software, the parameter file is updated and the IP core synthesised for the new set of parameters. From the Serial to Parallel to Pipeline design, the designer can scale the performance of the codesign depending on the level of hardware resources available; the designer can fine-tune the balance between resources and performance further in the Parallel and Pipeline designs by altering the number of processing elements. Thus, the proposed HW/SW codesign is both a scalable and portable implementation of the UKF and suitable for use in a generic state estimation library.

Chapter 4

Testing and Validation of the Hardware/Software Codesign

To validate the implementation of the hardware/software UKF and demonstrate its effectiveness, two example applications involving nanosatellites and micro-UAVs are presented in this chapter. The nanosatellite application emulates the attitude determination subsystem of a nanosatellite in two related situations: a single uncontrolled nanosatellite and a small constellation of uncontrolled nanosatellites. For the micro-UAV application, the state estimation part of a visual Simultaneous Localisation And Mapping (SLAM) system is considered.

It is envisioned that a system designer looking to use the HW/SW UKF in a new application simply formulates the UKF appropriately for that application, i.e. formulates the system models (2.1), and sets the algorithm parameters (see Section 3.1.1). Then, once the UKF algorithm has been defined, the HW/SW codesign detailed in Chapter 3 can then be used to actually implement the UKF and accelerate its performance. The example applications in the next two sections attempts to employ this process.

4.1 Nanosatellites

In order for nanosatellites to gain wider use for more specific scientific objectives, such as remote sensing, they also have to operate in a capacity where they can be considered beneficial over a single, larger satellite. One such proposal to this end is to have multiple satellites fly in formation with a homogeneous sensor set on-board; however, this approach still requires much higher accuracy pointing capabilities. Furthermore, each satellite's attitude determination and control system (ADCS) will likely need to be working at high sampling frequencies in order to maintain the satellites' formation for this type of multi-nodal sensing. This results in an increased computational load which is problematic for nanosatellites. Translating the necessary functionality into hardware and implementing it onto an FPGA is one approach to alleviate such problems.

FPGAs are becoming increasingly popular in space applications as mentioned in Section 2.3.1 and an FPGA allows multiple subsystem functions to be implemented as a SoC, which frees up valuable real estate within the satellite normally required by multiple processors. For this kind of nanosatellite application, it is envisioned that the Serial design could handle attitude determination and control for that nanosatellite as part of a greater SoC that handles all of the necessary computation for the nanosatellite. The Parallel and Pipeline design could be used in larger nano/microsatellites where greater performance may be required.

This section describes two related nanosatellite applications. The first involves the attitude determination subsystem of a single uncontrolled nanosatellite and is based on simulation results first presented in Soh and Wu (2014). The second simulates a small constellation of five uncontrolled nanosatellites, one of which is considered the 'lead' nanosatellite which uses the UKF to calculate the attitude of all five nanosatellites in the constellation; this second application is based on results first presented in Soh and Wu (2017a).

The UKF was implemented using a number of methods for validation and comparison purposes. Once formulated, the UKF was first implemented in Matlab (SW) to

validate the design of the UKF algorithm. Next, the UKF was implemented again using the HW/SW codesign on an FPGA development board in order to validate the codesign. Finally, the UKF was implemented a third time in C (SW), but on the same FPGA development board, to provide a performance benchmark the HW/SW codesign could be compared to. The three different implementations of the UKF used for this example application are summarised in Table 4.1

Implementation	Platform	Purpose
Matlab (SW)	PC	Validation of UKF algorithm
Codesign (HW/SW)	Dev. board	Validation of codesign
C (SW)	Dev. board	UKF implementation performance comparison

Table 4.1: Summary of the different UKF implementations for the nanosatellite example application

The FPGA development board used was the Zedboard (AVNet, 2014), featuring a Xilinx Zynq-7000 series XC7Z020 (Xilinx, 2016), seen in Figure 4.1. The relevant features of the board are:

- Dual ARM Cortex-A9 processor system (PS) @ 667 MHz
- The equivalent of an Artix-7 device in programmable logic (PL)
- AXI4 PS-PL interface

All three variants of the HW/SW codesign were implemented. The hardware part of the codesign, the IP core, was developed in Verilog and synthesised and implemented using Vivado 2014.1; basic arithmetic (i.e. Table 3.2) was implemented using floating point IP cores from Xilinx's IP catalogue. All designs used a single precision (IEEE 754-2008) number representation. The target synthesisable frequency for the IP core was 100 MHz and the Parallel and Pipeline version were instantiated with two processing elements (PEs) for the whole design (i.e. each individual module had two processing elements). The software part of the codesign was implemented in C as bare-metal application on the processor system. The general purpose AXI4 interface between the PS and the PL was used by the two parts to communicate with

Figure 4.1: Zedboard development board used for two of the UKF implementations

each other (@ 100 MHz as well). The C (SW) implementation of the UKF was a bare-metal application that used the GNU Scientific Library (GSL) for its vector and matrix manipulations.

To test the different UKF implementations, a simulator was constructed in Matlab to model the nanosatellites' motion. Both the single nanosatellite case and the nanosatellite constellation case are modelled in this simulator; the details are given in Section 4.1.5. Simulated sensor measurements were generated from the nanosatellites' motion and passed to each of the three UKF implementations which is acting as the attitude determination subsystem. For the Matlab implementation, the simulated measurements could be passed directly. For the HW/SW codesign and the C (SW) implementations, the simulated measurements were first exported to a C header which was included during compilation. The Matlab and C (SW) implementations of the UKF were applied to both modelled cases but for the HW/SW codesign implementation, the Serial and Parallel variants were applied to the single nanosatellite case while the Pipeline variant was applied to the nanosatellite constellation case.

4.1.1 System Model

Each individual nanosatellite is modelled as a 1U CubeSat. The attitude of the nanosatellite is represented by the unit quaternion $q = [\mathbf{q},\ q_0]^T$ where $\mathbf{q} = [q_1,\ q_2,\ q_3]^T$ and which satisfies $q_1^2 + q_2^2 + q_3^2 + q_0^2 = 1$.

The kinematic equations for the satellite in terms of quaternions are given by:

$$\dot{\mathbf{q}} = \frac{1}{2}\left(q_0 I_{3\times 3} + \mathbf{q}^{\times}\right)\boldsymbol{\omega} \tag{4.1a}$$

$$\dot{q}_0 = -\frac{1}{2}\mathbf{q}^T\boldsymbol{\omega} \tag{4.1b}$$

where $\boldsymbol{\omega}$ is the angular rate and $\mathbf{q}^{\times}$ is the skew-symmetric matrix of $\mathbf{q}$ given by:

$$\mathbf{q}^{\times} = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \tag{4.2}$$

4.1.2 Sensor Model

We consider a basic sensor set common on nanosatellites - a three-axis MEMS IMU including an accelerometer, gyroscope and magnetometer. We use the standard gyroscopic model for the gyroscope:

$$\mathbf{z}_g = \boldsymbol{\omega}_T + \boldsymbol{\beta} + \boldsymbol{\eta}_g \tag{4.3}$$

$$\dot{\boldsymbol{\beta}} = \boldsymbol{\eta}_d \tag{4.4}$$

where $\boldsymbol{\omega}_T$ is the true angular velocity, $\boldsymbol{\beta}$ is the gyroscopic bias, $\dot{\boldsymbol{\beta}}$ is the gyroscopic bias drift and $\boldsymbol{\eta}_g, \boldsymbol{\eta}_d$ are noise terms that are assumed to be zero-mean Gaussians. Similarly, we model the accelerometer and magnetometer as:

$$\mathbf{z}_a = \mathbf{a}_T + \boldsymbol{\eta}_a \tag{4.5}$$

$$\mathbf{z}_m = \mathbf{m}_T + \boldsymbol{\eta}_m \tag{4.6}$$

where $\mathbf{a}_T$ is the true local acceleration vector, $\mathbf{m}_T$ is the true local magnetic vector and $\boldsymbol{\eta}_a, \boldsymbol{\eta}_m$ are again, zero-mean Gaussian measurement noise terms.

4.1.3 Predict Model

We use a dead-reckoning model and the gyroscopic data to predict the motion of the nanosatellite. However, it is necessary to account for the gyroscopic bias drift so we estimate the current gyroscopic bias as well. Let the state vector be:

$$\mathbf{x} = [\mathbf{q},\ q_0,\ \boldsymbol{\beta}]^T \tag{4.7}$$

The `predict` model, f, is then:

$$f(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) = \boldsymbol{\mathcal{X}}^x_{k-1|k-1} + f'(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) \cdot dt \tag{4.8}$$

$$f'(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) = \begin{bmatrix} \frac{1}{2}\left(q_0 I_{3\times3} + \mathbf{q}^\times\right)\mathbf{z}_g \\ -\frac{1}{2}\mathbf{q}^T\mathbf{z}_g \\ \mathbf{0}_{3\times1} \end{bmatrix} + \mathbf{w}_k \tag{4.9}$$

where dt is the time step between samples, $\mathbf{w}_k = [\boldsymbol{\eta}_q,\ \dot{\boldsymbol{\beta}}]^T$ is the process noise and $\boldsymbol{\eta}_q$ is assumed to be a zero-mean Gaussian.

4.1.4 Update Model

The accelerometer and magnetometer data is used to correct for the gyroscopic bias, so the observation model, h, is:

$$h(\boldsymbol{\mathcal{X}}^x_{k-1|k-1},\ \boldsymbol{\mathcal{X}}^v_{k-1|k-1}) = \begin{bmatrix} A_q\left(\mathbf{q}\right) g\mathbf{b}_a \\ A_q\left(\mathbf{q}\right)\mathbf{b}_m \end{bmatrix} + \mathbf{v}_k \tag{4.10}$$

where $\mathbf{b}_a$ and $\mathbf{b}_m$ are the respective body frame vectors, g is the magnitude of the gravity vector (assumed 8.94 m.s^{-2} at an altitude of 300 km), $\mathbf{v}_k = [\boldsymbol{\eta}_a,\ \boldsymbol{\eta}_m]$ is the measurement noise and $A_q(\mathbf{q})$ is the rotation matrix between the body frame and local frame given by:

$$A_q(\mathbf{q}) = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_0q_2 + q_1q_3) \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_0q_1 + q_2q_3) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{4.11}$$

4.1.5 Simulation Model

Collecting all the relevant terms, the initial augmented state vector is given by:

$$\mathbf{x}_0^a = [\mathbf{q},\ q_0,\ \boldsymbol{\beta},\ \mathbf{0}_{4\times1},\ \mathbf{0}_{3\times1},\ \mathbf{0}_{3\times1},\ \mathbf{0}_{3\times1}]^T \tag{4.12}$$

and the initial augmented covariance is a diagonal matrix with diagonal terms:

$$\mathrm{diag}(\mathbf{P}_0^a) = [\mathbf{1}_{6\times1},\ \boldsymbol{\eta}_q,\ \dot{\boldsymbol{\beta}},\ \boldsymbol{\eta}_a,\ \boldsymbol{\eta}_m] \tag{4.13}$$

The state vector length is 7, the number of observation variables is 6 and the augmented state vector length is 20. The quaternion noise term was modelled with covariance $\boldsymbol{\eta}_q = 10^{-6}$. The simulated sensor set was homogeneous so the modelled errors are the same for each nanosatellite. The gyroscopic bias drift was modelled with covariance $\boldsymbol{\eta}_d = 10^{-2}\ {}^\circ/s^2$. The measurement noise terms were modelled with covariances: $\boldsymbol{\eta}_g = 10^{-1}\ {}^\circ/s$, $\boldsymbol{\eta}_a = 10^{-2}g$, $\boldsymbol{\eta}_m = 10^{-2}\ gauss$.

Each individual satellite was modelled as undergoing a different motion, including: a steady state, slow oscillations about one or more axes and full tumbling. The motion was modelled using Euler angles in a local ground frame which is relevant in most remote sensing applications; here, we use roll-pitch-yaw to refer to rotations about the x-y-z axis respectively. An example of the simulated truth data can be seen in Figure 4.2 and the simulated motions of each of the five nanosatellites can be seen in Table 4.2.

Satellite No.	Type of motion
1	Sinusoidal oscillation over all three axes with an amplitude of 5° and natural frequency $\omega_n = 0.01$ Hz
2	Sinusoidal oscillation over all three axes with an amplitude of 5° and natural frequency $\omega_n = 0.05$ Hz
3	Full tumbling about the roll axis with frequency 0.01 Hz
4	Sinusoidal oscillation about the roll/pitch axes with an amplitude of 45° and natural frequency $\omega_n = 0.002$ Hz
5	Sinusoidal oscillation about the pitch/yaw axes with an amplitude of 2° and natural frequency $\omega_n = 0.1$ Hz

Table 4.2: Modelled motions for each of the nanosatellites. The motion for the first satellite is reused between the single nanosatellite case and the nanosatellite constellation case.

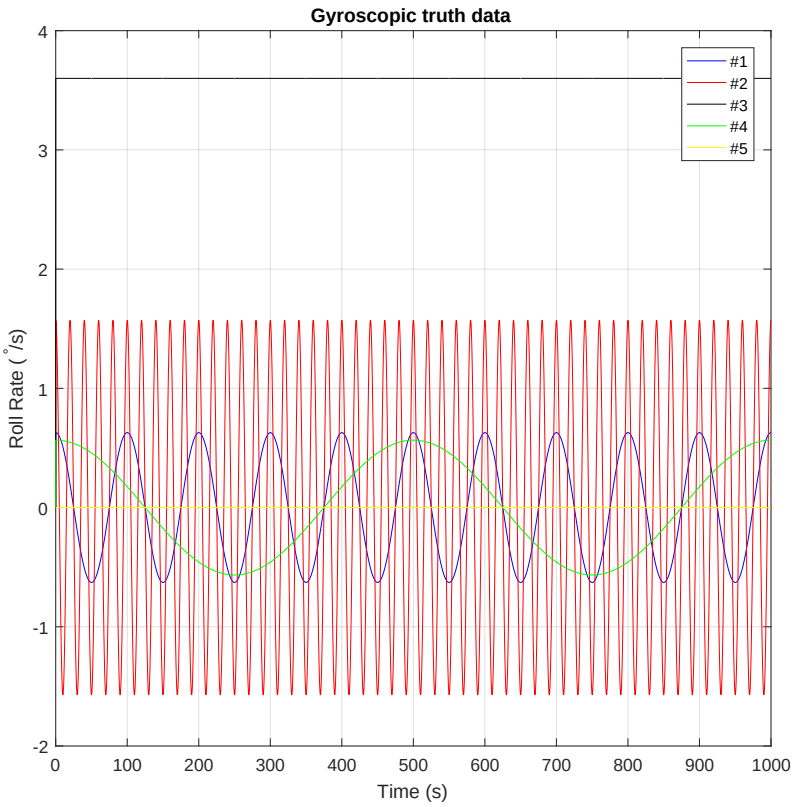
To generate the sensor measurements, the simulated motions where converted into the body frame via rotation matrix with 1-2-3 referring to roll-pitch-yaw respectively:

$$A_{euler} = \begin{bmatrix} c_1c_2 & c_1s_2s_3 - s_1c_3 & s_1s_3 + c_1s_2c_3 \\ s_1c_2 & s_1s_2s_3 + c_1c_3 & s_1s_2s_3 - c_1s_3 \\ -s_2 & c_2s_3 & c_2c_3 \end{bmatrix} \tag{4.14}$$

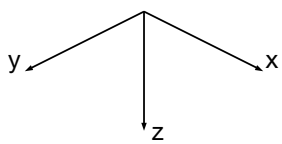
It is assumed that the magnetometer is aligned with the x-axis ($\mathbf{b}_m = [1, 0, 0]$) and the accelerometer is aligned with the z-axis ($\mathbf{b}_a = [0, 0, 1]$). Next, using the sensor models described earlier, noise terms were added to the sensor 'truth' data which was then sampled at 1 Hz to simulate measurements from an actual set of sensors; an example of the simulated gyroscopic measurements for one of the nanosatellites can be seen in Figure 4.3.

4.1.6 Results

The UKF was simulated in Matlab environment as well as on the Zedboard development board. For the two Zedboard implementations, the simulated sensor dataset was loaded into the onboard memory (RAM) and the UKF simulated as if it were receiving data from the actual sensors. The dataset used in all three implementations



(a) 'Truth' data



(b) Reference frame

Figure 4.2: Simulated 'truth' roll for all five nanosatellites

was the same. State estimates from the UKF were stored on the Zedboard for the duration of the simulation then read back into Matlab afterwards for analysis. Both applications, single nanosatellite and nanosatellite constellation, were simulated in this way.

For the first simulation, with a single nanosatellite, all three implementations produced (within working precision) the simulation results in Figure 4.4a and 4.4b; these figures show the absolute attitude error (i.e. the difference between the UKF estimated attitude and the simulated 'truth') of the nanosatellite. In Figure 4.4a, the top graph shows the first tenth of a second of the simulation, highlighting early convergence of the filter to the truth from an initial noisy estimate. The bottom figure shows the first second of the simulation, highlighting the ability of the filter to maintain its accuracy ($< 0.1°$ error) after convergence. Figure 4.4b shows that the UKF

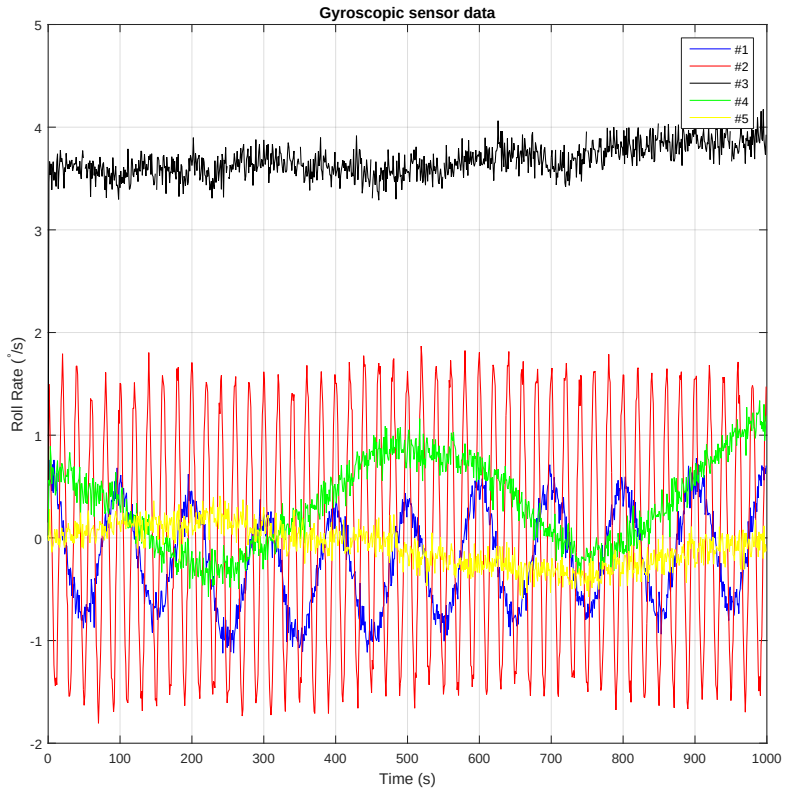
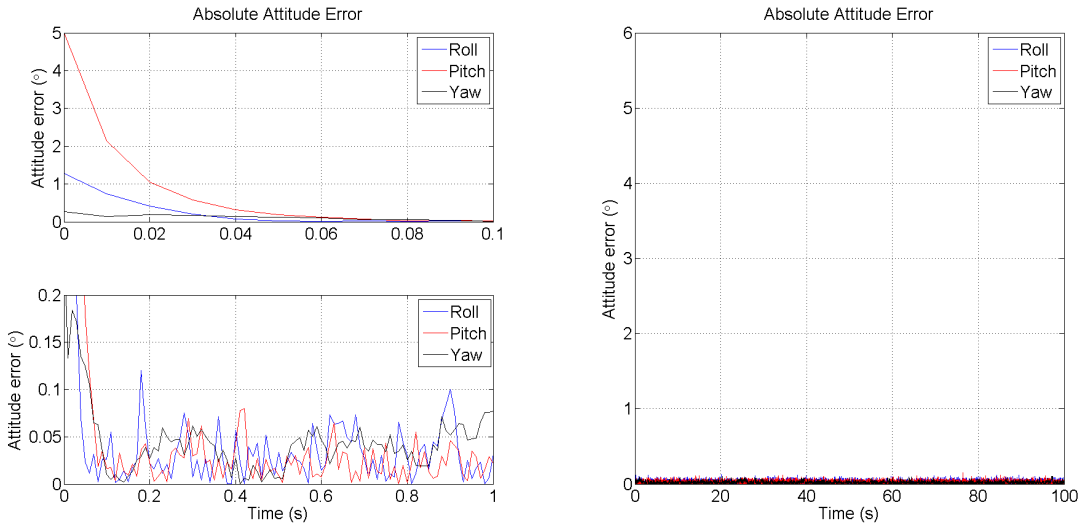


Figure 4.3: Sample of the simulated gyroscopic sensor data for all five nanosatellites. For clarity, only 'measurements' for the roll axis is shown.

is able to correct for the inaccuracies arising from the gyroscopic bias and bias drift over the full duration of the simulation.

These results demonstrate that there are no implementation issues when taking the UKF to a HW/SW codesign; the codesign, and IP core, is able to completely replicate software-based implementations of the UKF. The overall latency of the C (SW) implementation and the HW/SW codesign (Serial and Parallel) for the single nanosatellite case were measured using the ARMv7 Performance Monitoring Unit (PMU) and can be seen in Table 4.3. This overall latency is the time taken to complete one full iteration of the UKF (all steps). The Serial design offers a modest 1.8× increase in performance over the C (SW) implementation and can be run at ≈ 2 kHz which is more than adequate for the sampling frequency assumed by the simulation. The Parallel design offers a slightly better 2.4× speed-up, for the 2 PE case, over the C



(a) At the very beginning of the simulation.

(b) For the full simulation

Figure 4.4: Absolute attitude error

(SW) implementation. Note that the processor system operates at a clock frequency more than 6 times the frequency used by the IP core (667 MHz vs. 100 MHz), yet the IP core is still able to out perform the C (SW) implementation.

	SW	Serial	Parallel
Total	660	363	272

Table 4.3: Overall latency for the single nanosatellite. All values in μs.

For the second simulation, with a constellation of five nanosatellites, five instances of the UKF were calculated, one for each nanosatellite, and all three implementations produced (within working precision) the simulation results in Figure 4.5; these figures once again show the absolute attitude error of the nanosatellite. The UKF is again able to quickly converge after initialisation and maintain its accuracy over the duration of the simulation. Though the UKF is able to display good accuracy ($< 0.2°$) for most of the nanosatellites, it exhibits slightly poorer performance ($< 1°$) for nanosatellites undergoing the more erratic motions.

The overall latency for the C (SW) implementation and the Pipeline variant of the HW/SW codesign for the nanosatellite constellation case was also measured and can

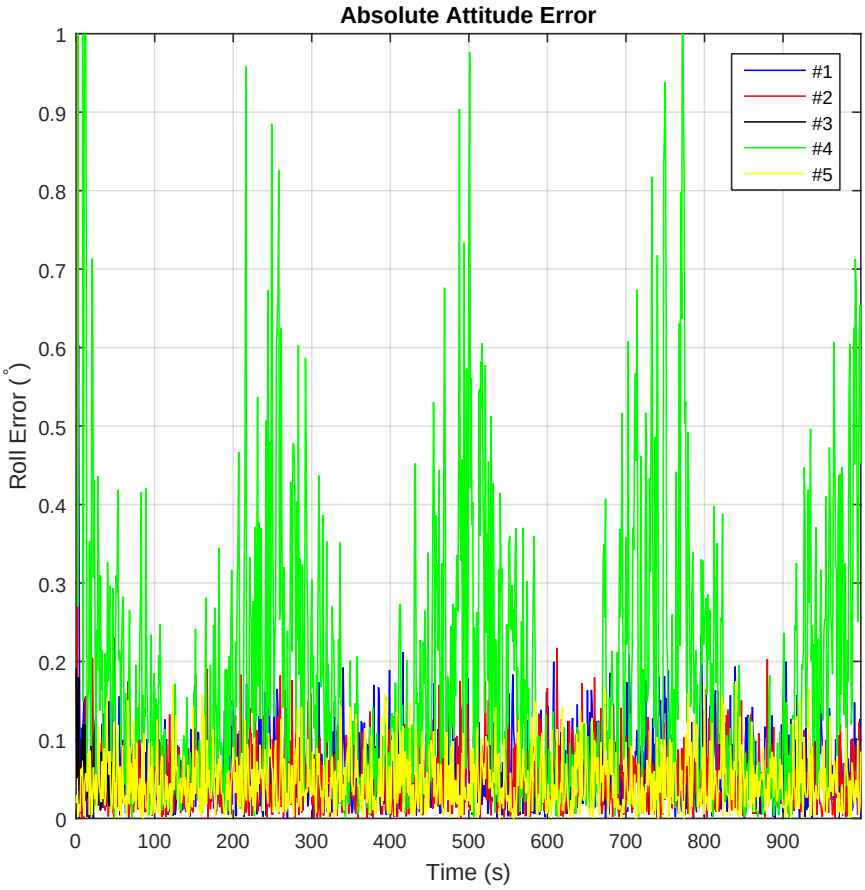


Figure 4.5: Absolute attitude error for the full simulation

be seen in Table 4.4. The C (SW) implementation must calculate each of the 5 UKF instances sequentially while the Pipeline design is able to accept data for a new instance after the first stage is complete. The Pipeline design offers a 2.75× performance gain in the 2 PE case.

	SW	Pipeline
Total	3347	1219

Table 4.4: Overall latency for the nanosatellite constellation. All values in μs.

4.2 Simultaneous Localisation And Mapping

Autonomous navigation is one of the main focus areas in increasing the capabilities, and thus viability, of robotic/autonomous systems in any given application. Autonomous navigation is an extremely complex process with many aspects that have to work together within several layers of abstraction. State estimation, to serve localisation and mapping subsystems, sits in a 'middle' layer with state estimates passed to 'higher' level path planning or other decision making algorithms, while requiring information about the environment from 'lower' level perception subsystems. For an unknown environment or an environment with only limited existing information about it, the SLAM problem is how an autonomous system can construct or update a map of the environment while simultaneously keeping track of its location within that environment. The issue is that, in general, for a system to localise itself within an environment, a map of that environment is required but in order to generate a map of the environment, the system needs to know where it is within that environment. Many solutions to the SLAM problem exist, including: the EKF, particle filters and, of course, the UKF; an informative treatise on SLAM is given by Durrant-Whyte and Bailey (2006).

Early SLAM solutions focused on ground robotics and, as such, were restricted to 2D. Here, the full 3D case is considered which is a straightforward, if not necessarily simple, extension. Common sensor sets used in SLAM solutions include laser rangefinders and cameras for vision-based navigation. The application modelled here is a small quadcopter UAV with a single, fixed pinhole camera with static landmarks in an enclosed room. There are, of course, many more parts to vision-based navigation than just the SLAM system. Image processing capabilities are needed to extract relevant features or landmarks from the camera data stream as well as perform data association between extracted features and previously known landmarks. As mentioned, information generated from the SLAM solution can be passed to higher-level navigation algorithms. The main concern here, however, is state estimation and, in particular, the performance of the UKF, so it is assumed that all feature extraction and data association has already been handled.

As with the previous nanosatellite application, multiple methods are used to implement the UKF for testing and validation. A Matlab implementation is used again to validate the formulation of the UKF, however, the codesign implementation is tested a little differently.

As mentioned in Section 3.1, the hardware part – the IP core – implements the non-application-specific parts of the UKF; when moving to a new application, the IP core only needs to be instantiated with the correct set of parameters (see Section 3.1.1) rather than redesigned. In the previous section (Section 4.1), it was already demonstrated that the HW/SW codesign is capable of completely replicating the UKF algorithm so here the focus is on the performance of the IP core. As long as the design of the UKF algorithm is validated (and thus that the selected parameters are correct) then the performance of the IP core can be tested alone, without the software part.

In addition to this, the simulation environment necessary to test a full SLAM solution (including other parts, e.g. data association) is quite complex and beyond the scope of this thesis. The complexity makes it difficult to construct the required simulator in an embedded platform. For this reason, the performance of the IP core is not tested using the Zedboard platform as in the nanosatellite application, but instead by using Xilinx's Vivado Simulator (Xilinx, 2017b) to perform a behavioural simulation. Behavioural simulation is ordinarily used only to verify functionality but as long as operational (e.g. processor setting a control bit) and timing assumptions (e.g. input clock frequency) made during the behavioural simulation are verified post-implementation, Vivado Simulator provides the execution time of the IP core accurate to the clock cycle.

In this case, the IP core used the floating point IP cores from Xilinx's IP catalogue for its basic arithmetic (i.e. Table 3.2) and was synthesised with a target frequency of 100 MHz. With the assumptions for the IP core validated post-implement, Vivado Simulator was used to measure performance. The HW/SW codesign variants tested here are the Serial design and the two processing element (PE) case of the Parallel design.

4.2.1 System Model

The outputs of any SLAM solution is the pose of the autonomous system or robot, in this case the UAV, as well as the positions of relevant landmarks in the environment which can be used to generate a map. The UAV pose includes 3D Cartesian co-ordinates in some world frame for the UAV's position and quaternions, as in 4.1.1, for the UAV's attitude. The motion of the UAV is modelled with the angular rates given by (4.1) and the linear rates given by:

$$\mathbf{v} = A_q(\mathbf{q})\mathbf{v}_R \tag{4.15}$$

where $\mathbf{v}_R$ is the linear velocity of the UAV in the UAV frame and $A_q(\mathbf{q})$, given by (4.11) as before, is the rotation of the UAV frame with respect to the world frame. The UAV linear and angular rates are controlled via inputs:

$$\mathbf{u}_k = [\mathbf{u}_{xyz},\ \mathbf{u}_{\psi\theta\phi}] \tag{4.16a}$$

$$\mathbf{u}_{xyz} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + \boldsymbol{\eta}_{u,xyz} \tag{4.16b}$$

$$\mathbf{u}_{\psi\theta\phi} = \begin{bmatrix} u_\psi \\ u_\theta \\ u_\phi \end{bmatrix} + \boldsymbol{\eta}_{u,\psi\theta\phi} \tag{4.16c}$$

where $\mathbf{u}_{xyz}$ is the desired linear motion for the x, y and z axes respectively, $\mathbf{u}_{\psi\theta\phi}$ is the desired angular motion about the roll, pitch, yaw rotational axes respectively and $\boldsymbol{\eta}_{u,xyz}$ and $\boldsymbol{\eta}_{u,\psi\theta\phi}$ are the zero-mean Gaussian control noise terms.

Landmarks in the environment are represented using the inverse depth parameterisation. For i-th landmark $\mathbf{L}_i$:

$$\mathbf{L}_i = [x_i,\ y_i,\ z_i,\ \alpha,\ \beta,\ \rho] \tag{4.17}$$

where x_i, y_i, z_i are the co-ordinates of the UAV when the landmark was first seen in

the world frame, α and β are the azimuth and elevation to the landmark respectively, when it was first seen in the world frame and ρ is the inverse depth (i.e. $\rho = 1/d$ where d is the distance to the landmark) of the landmark. The inverse depth parameterisation is common for vision-based SLAM solutions as it provides low linearisation errors at low parallax and has the ability to represent any distance from the system immediately; features at very large distances that are effectively 'infinite' from the system would ordinarily be unusable, attracting additional processing to treat or discard those sensor readings, but, in this parameterisation, is simply treated as zero. Full details of the inverse depth parameterisation is presented by Civera et al. (2008) (who also provides a monocular SLAM example implementation).

4.2.2 Sensor Model

The sole sensor used in this application is the aforementioned pinhole camera, fixed to the front of the UAV and its aperture aligned perpendicular to the UAV x-/roll axis. Since it is assumed that feature extraction and data association has already been performed, the camera observations are already rotated (ordinarily the mapping of the 3D co-ordinates to a 2D image plane described here is a perspective projection with a 180° rotation in the image plane) and the sensor readings are simply the camera/image frame co-ordinates to the landmark. The pinhole camera model for the co-ordinates of some point P, which exists in the environment, in the camera frame is given by:

$$\mathbf{z}_c = \begin{bmatrix} z_{c,u} \\ z_{c,v} \end{bmatrix} = \begin{bmatrix} f_u \frac{y_P}{x_P} \\ f_v \frac{z_P}{x_P} \end{bmatrix} + \boldsymbol{\eta}_c \tag{4.18}$$

where f_u and f_v are the distances from the centre of the aperture of the camera to the centre of the image plane (i.e. the camera co-ordinate frame), x_P, y_P, z_P are the Cartesian co-ordinates of the point P in the UAV frame and $\boldsymbol{\eta}_c$ is a zero-mean Gaussian noise term.

4.2.3 Predict Model

The predict model uses a dead reckoning model and the control inputs to predict the motion of the UAV. The positions of known landmarks are also tracked so let the state vector be:

$$\mathbf{x} = [\mathbf{p},\ \mathbf{q},\ q_0,\ \mathbf{L}_1,\ \ldots,\ \mathbf{L}_n]^T \tag{4.19}$$

where $\mathbf{p} = [p_x,\ p_y,\ p_z]$ is the Cartesian position of the UAV in the world frame. The predict model, f is then:

$$f(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) = \boldsymbol{\mathcal{X}}^x_{k-1|k-1} + f'(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) \cdot dt \tag{4.20a}$$

$$f'(\boldsymbol{\mathcal{X}}^x_{k-1|k-1}, \boldsymbol{\mathcal{X}}^w_{k-1|k-1}) = \begin{bmatrix} A_q(\mathbf{q})\mathbf{u}_{xyz} \\ \frac{1}{2}\left(q_0 I_{3\times3} + \mathbf{q}^\times\right)\mathbf{u}_{\psi\theta\phi} \\ -\frac{1}{2}\mathbf{q}^T\mathbf{u}_{\psi\theta\phi} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \tag{4.20b}$$

where dt is the time step between control inputs.

4.2.4 Update Model

The update model uses new measurements of one of the landmarks to update the state of both the UAV and that landmark. The observation model, h, is:

$$h(\boldsymbol{\mathcal{X}}^x_{k-1|k-1},\ \boldsymbol{\mathcal{X}}^v_{k-1|k-1}) = \begin{bmatrix} f_u \dfrac{y_L}{x_L} \\ f_v \dfrac{z_L}{x_L} \end{bmatrix} + \mathbf{v}_k \tag{4.21}$$

where $\mathbf{v_k} = \boldsymbol{\eta}_c$ is the observation noise and x_L, y_L, z_L are the co-ordinates of landmark in the world frame calculated via:

$$\begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = A_q(\mathbf{q}) \left(\rho \left(\mathbf{L}_{i,xyz} - \mathbf{p}_{k-1} \right) + \begin{bmatrix} \cos\alpha\cos\beta \\ \sin\alpha\cos\beta \\ \sin\beta \end{bmatrix} \right) \tag{4.22}$$

where $\mathbf{L}_{i,xyz}$ is the Cartesian position of the UAV in the world frame when the i-th landmark was first seen and $\mathbf{p}_{k-1}$ is the a priori estimate of the position of the UAV in the world frame.

As the UAV moves around the environment, the number of visible landmarks is not necessary static. If a new landmark is detected, the state vector needs to be expanded and initialised with the new information. Adding a new landmark to the tracking is done by passing the current state and observation to an inverse sensor model, h^{-1}, given by:

$$\mathbf{L}_{i,0} = h^{-1}(\mathbf{x}_{k-1}, \mathbf{z}_k) \tag{4.23}$$

$$h^{-1}(\mathbf{x}_{k-1}, \mathbf{z}_k) = \begin{bmatrix} \mathbf{p}_{k-1} \\ \arctan\left(l_y, l_x\right) \\ \arctan\left(l_z, \sqrt{l_x^2 + l_y^2}\right) \\ \rho_0 \end{bmatrix} \tag{4.24}$$

$$\begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} = A_q(\mathbf{q}) \begin{bmatrix} 1 \\ \frac{z_{c,u}}{f_u} \\ \frac{z_{c,v}}{f_v} \end{bmatrix} \tag{4.25}$$

where l_x, l_y, l_z are the co-ordinates of the newly detected landmark in the world frame. New landmarks are detected when observations cannot be associated with existing known landmarks; as mentioned, this data association is assumed to be handled already. When known landmarks have not been seen in some time, they are usually deleted from tracking to reduce computational burden; this process is again handled by the data association process that is not being considered here.

4.2.5 Simulation Model

With all the relevant terms, the initial augmented state vector is:

$$\mathbf{x}_0^a = [\mathbf{p},\ \mathbf{q},\ q_0,\ \mathbf{L}_1,\ \ldots,\ \mathbf{L}_n,\ \mathbf{0}_{3\times1},\ \mathbf{0}_{3\times1},\ \mathbf{0}_{2\times1}]^T \tag{4.26}$$

and the initial augmented covariance is a diagonal matrix with diagonal terms:

$$\mathrm{diag}(\mathbf{P}_0^a) = [\mathbf{1}_{(7+6n)\times1},\ \boldsymbol{\eta}_{u,xyz},\ \boldsymbol{\eta}_{u,\psi\theta\phi},\ \boldsymbol{\eta}_c] \tag{4.27}$$

The length of the state vector is $7 + 6n$ where n is the number of known features, the number of observation variables is 2 and the augmented state vector has $15 + 6n$ variables. The maximum number of landmarks considered in this simulation is 3 (i.e. $n = 3$) so the maximum state and augmented state vector has 25 and 33 variables respectively. The control noise terms are modelled with covariances $\boldsymbol{\eta}_{u,xyz} =$ [0.002812, 0.004349, 0.002248] m.s^{-1} and $\boldsymbol{\eta}_{u,\psi\theta\phi} =$ [0.01993, 0.03476, 0.03223] rad.s^{-1} while the observation noise is modelled with covariance $\boldsymbol{\eta}_c = 5$.

A diagram of the initial setup of the simulation area can be seen in Figure 4.6. At the beginning of the simulation, the UAV is 'hovering' in one position (the origin) and is initialised with a slightly noisy estimate of its own position. The three landmarks are scattered at different positions and altitudes around the simulation area. All three landmark are in view of the UAV at the start and so the UAV has an initial (inaccurate) estimate of their positions; each of the estimates have relative high uncertainties (the green ellipses).

4.2.6 Results

The UAV is flown around in a polygon shape, roughly 3×2 m in size, maintaining its initial altitude and orientation (i.e. the UAV does not rotate during the flight); the final status of the simulation area can be seen in Figure 4.7. The SLAM solution has been able to track to the motion of the UAV along its path, i.e. the estimated

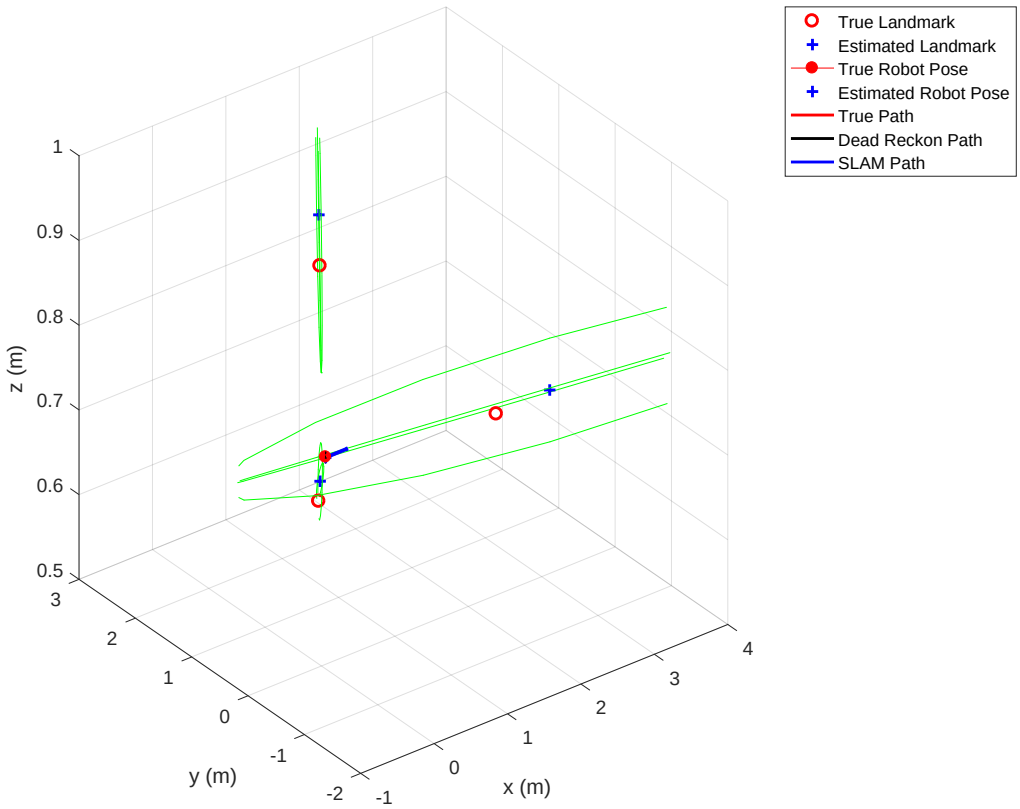


Figure 4.6: The initial UAV and landmark position estimates for the SLAM simulation. The green ellipses are the covariance (representing the uncertainty) of the various position estimates.

and true paths converge; a dead reckoning path (integrating the control actions) is included for comparison. The SLAM solution has also been able to estimate the positions of the three landmarks with a high amount of certainty; the estimated and true positions overlap and the uncertainty (green ellipses) have shrunk considerably around the estimate.

A closer look at the UAV path (distance with respect to the origin) can be seen in Figure 4.8, split into the two relevant dimensions (the UAV was flown level, at the same altitude). The true path and the SLAM UKF estimated path are nearly indistinguishable, but the dead reckoning path clearly deviates further and further as the simulation proceeds particularly in the (world) y-axis as the UAV makes many more turns. Figure 4.9 shows the position error of the UAV over the whole simulation. This

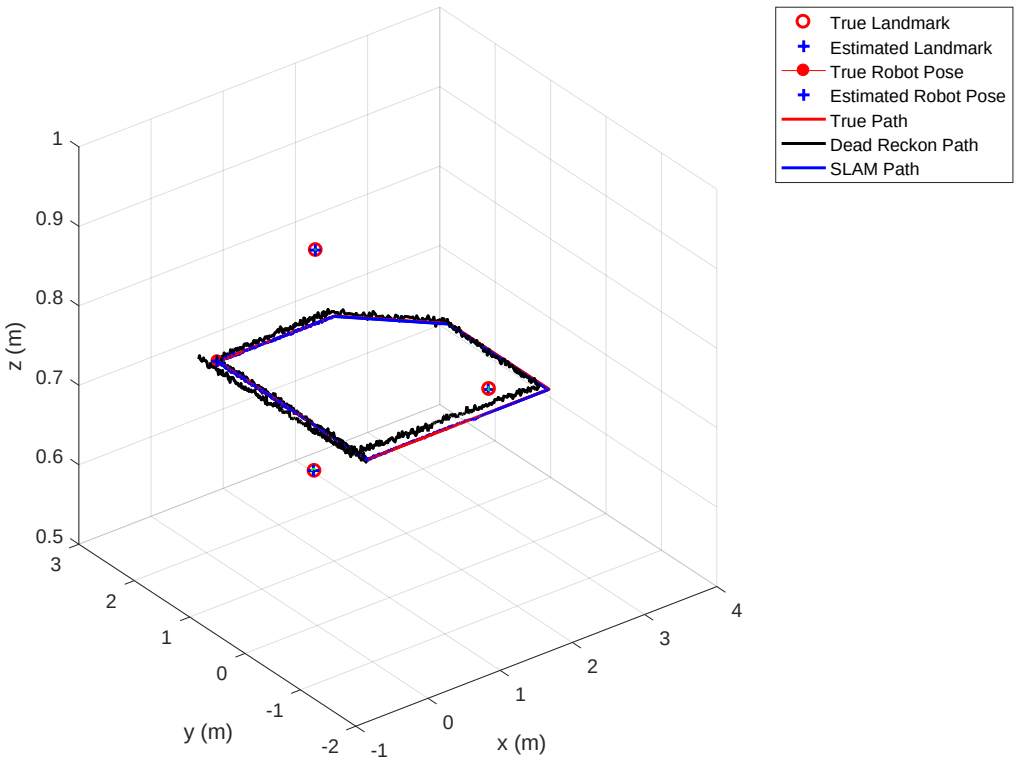


Figure 4.7: The final UAV and landmark position estimates for the SLAM simulation. The green ellipses are the covariance (representing the uncertainty) of the various position estimates.

is the difference between the UAV's true path and the UKF's estimate of the path. The dead reckoning estimate rapidly becomes poor while the SLAM UKF estimate is able to maintain a high level ($<$ 50 mm) of accuracy for the whole simulation. However, there is some instability on the accuracy of the SLAM UKF estimate because the number of landmarks in view is not static. The UAV loses sight of the landmarks during some parts of the simulation leading to a worsening position estimate.

The measurement and analysis of the latency for the SLAM solution is conducted slightly differently than the nanosatellite application. Hardware or HW/SW implementations of the UKF are rare to begin with (see Section 2.5.3) and the author is not aware of any targeting a SLAM system. Because there are a number of different ways to implement a SLAM system, literature in the field often focuses on algorithmic performance rather than runtime performance or execution time (e.g. Wang,

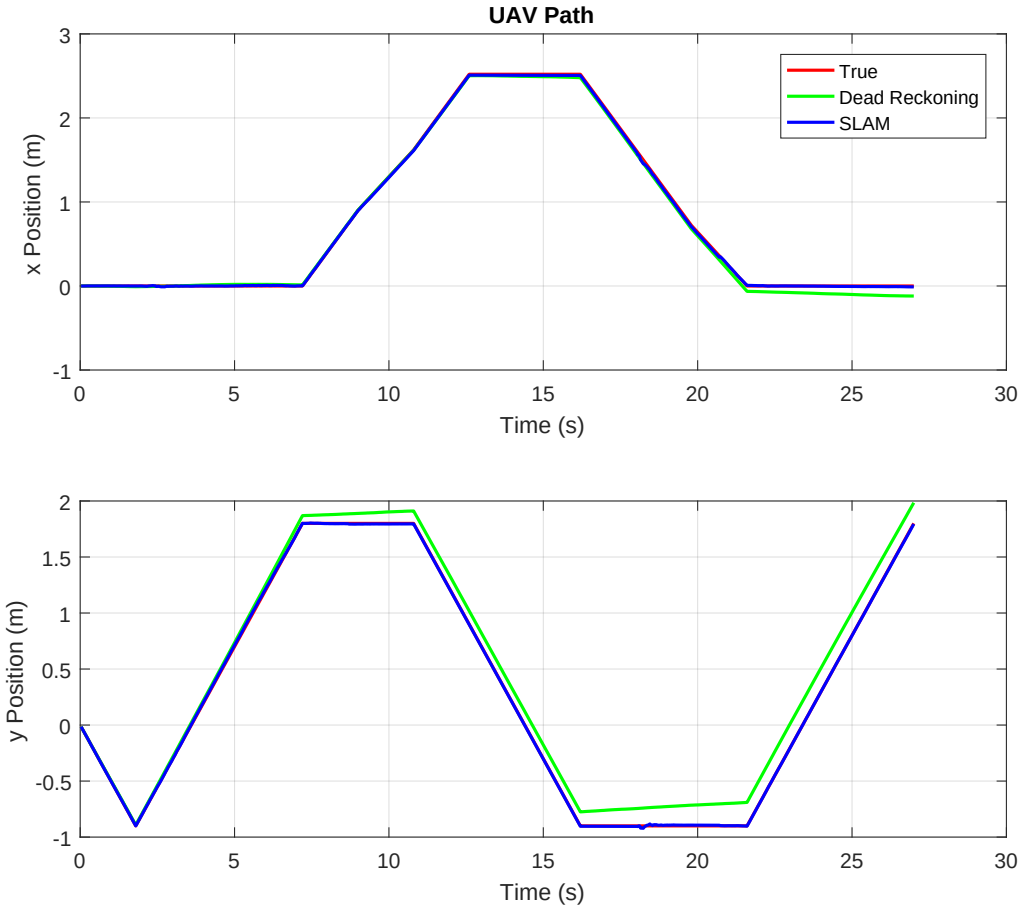


Figure 4.8: The path flown by the UAV for the SLAM simulation. The figure shows the distance of the UAV from the origin. The z-axis is omitted as the UAV was flown level, at the same altitude.

Fu, et al. (2013) or Huang et al. (2013)). For literature that does factor in runtime performance (e.g. Holmes et al. (2009) or Tuna et al. (2012)), the SLAM system is usually implemented on a powerful PC microprocessor; examples of the SLAM system being implemented on an embedded system are rare and examples of an embedded UKF-SLAM system are even rarer. Furthermore, the presented example application is a simplistic SLAM system aimed at demonstrating the flexibility of the HW/SW codesign rather than attempting to provide a competitive SLAM implementation. Factors such as data association are not handled and there are only a small number of landmarks (realistic SLAM systems can utilise thousands of landmarks/features). This makes comparing the presented example application to existing implementations

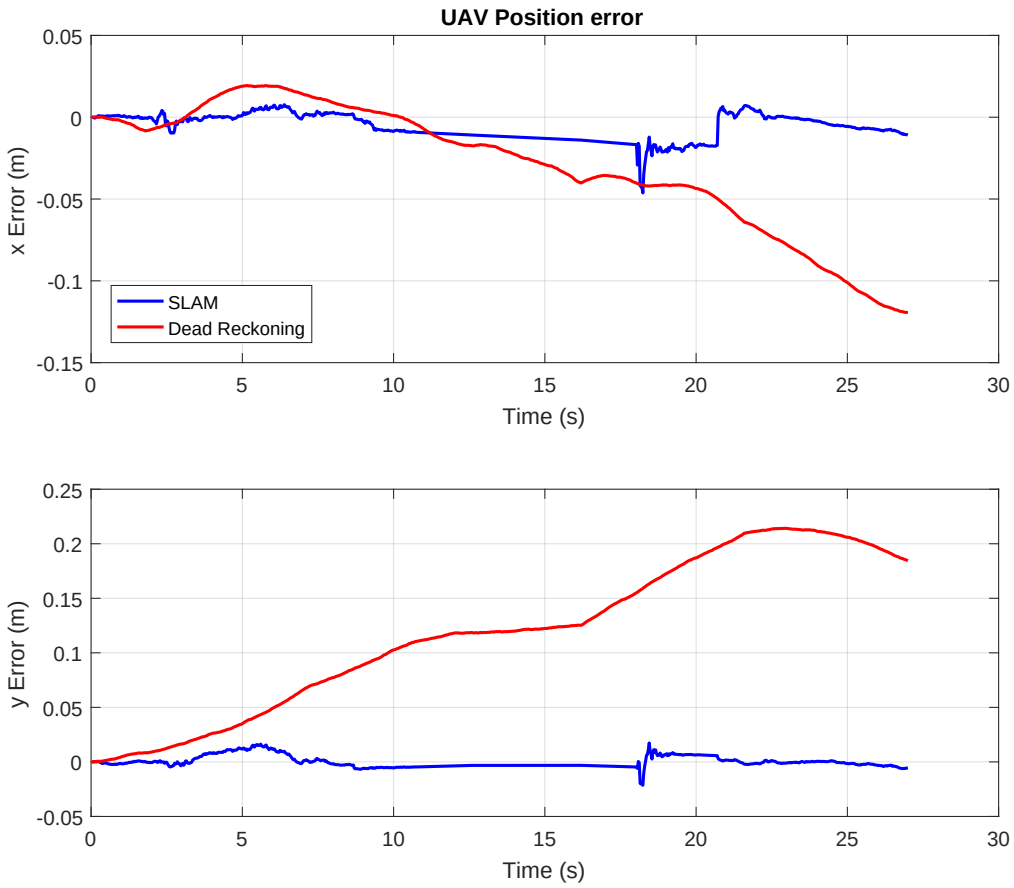


Figure 4.9: The UAV position error for the SLAM simulation. The UAV loses sight of all three of the landmarks in the 10-18 second range leading to a small deviation in the x-axis estimate. The instability in the SLAM estimate around 18 - 21 seconds is when the UAV regains sight of these landmarks and incorporates them into the position estimate again.

somewhat difficult and not necessarily meaningful.

For these reasons, runtime performance of the Matlab implementation, done on a PC, is reported rather than an embedded implementation such as the C (SW) implementation in the nanosatellite application. The Matlab simulation was performed on a PC featuring an Intel Core i7-4770K @ 3.5 GHz; each step was run at least 10 times and the average value is reported here. Though the performance of the hardware part of the codesign, the IP core, was measured using a behavioural simulation in the Vivado Simulator, the software part of the codesign was still implemented in C onto the Zedboard which allows for a slightly more accurate performance estimate.

This means the performance of the hardware part and software part was measured independently then combined later; this is unlike the nanosatellite application where the entire codesign was implemented, tested and measured as a whole.

	Matlab	Serial	Parallel
Sig. Gen.	76	-	334
Predict	902	944	281
Update	819	220	169
Total:			
- 0	978	944	615
- 1	1873	1572	1118
- 2	2768	2200	1621
- 3	3663	2828	2124

Table 4.5: Overall latency for the SLAM application. All values in μs. Total values are for the listed number of observed features at the current time step. Matlab's Cholesky Decomposition implementation is heavily optimised and use of vectorised arithmetic for the matrix multiplication is why the `sig_gen` step is so quick.

The overall latency of the SLAM solution for the Matlab, Serial and Parallel (2 PE) implementations can be seen in Table 4.5. The latency of the algorithm depends on the number of landmarks that are visible at any given time step. The structure of the algorithm is also slightly different compared to the nanosatellite application. In the previous application, the UKF at a single time step involves a `sig_gen` step followed by the `predict` step followed by an `update` step. For the SLAM solution, however, multiple `update` steps needs to be performed depending on how many observations were made in a single time step; in some cases, no observations of landmarks were made and so no `update` step was performed. In addition to this, because the `update` step updates the augmented state and covariance, subsequent `update` steps require the sigma points to be re-sampled. The full process is:

for time step k **do**
  `sig_gen` step
  `predict` step
  for i = observation **do**
    if i = new landmark **then**

```
            Initialise new landmark with (4.23)
        else
            if i > 1 then
                sig_gen step to re-sample sigma points
            end if
            update step
        end if
    end for
end for
```

Thus at each time step, one `sig_gen` and `predict` step is performed, but n `update` and $n-1$ `sig_gen` steps are required where n is the number of observations made of *known* landmarks; here, the maximum number of observations is three (there are three landmarks) so $n = 0-3$ at each time step. No `update` is performed for new landmarks or known landmarks that have no observations. The Serial design provides a modest 1.04 - 1.3× speed-up over the Matlab implementation, depending on the number of observed features, while the Parallel design for the 2 processing element case provides a 1.6 - 1.7× speed-up; though the performance benefit is small, the working clock frequency for the FPGA is a mere fraction of the PC. The faster `update` step means the two HW/SW codesign implementations will not scale as poorly as the Matlab implementation for larger numbers of landmarks.

The main concern for visual SLAM implementations, or visual navigation systems in general, is the ability to perform online calculations in 'real-time'. For vision-based systems 'real-time' usually means the frame rate of the camera (or cameras), commonly 30 Hz. Each of the implementations here would more than meet the real-time requirements for low numbers of landmarks. The example presented here is only the state estimation part of a full vision-based navigation system which would also include the actual image processing, data association, path planning etc. but the faster the state estimation part is able to run, the more time is available to be performing more complex, and intelligent, calculations.

It should also be noted that the implementation presented here is a 'naive' approach

to the SLAM problem. This is because each observed landmark adds variables to the state vector. Clearly, as the number of observed landmarks increases, potentially into the hundreds or thousands depending on the exact application, the state vector increases to a point where the UKF is no longer feasible. A quirk of the SLAM problem, however, is that observations of landmarks are independent of each other – i.e. for any given observation, the only state variables affected are the UAV's, or in general the robot/autonomous system, and the landmark observed. Given this independence between landmarks, it is not necessary to formulate one UKF with all the landmarks but instead multiple UKFs can be formulated, each of which track one landmark; Huang et al. (2009), for example, detail one such approach. This approach still requires n `update` steps and re-sampling of the sigma points each time but with much smaller state vectors. In this example, the `predict` state and augmented state vector would only need to have 7 and 13 variables respectively (only including the UAV pose and control noise but not the measurement noise) while the `update` state and augmented state vector would have 13 and 15 variables respectively (including the UAV pose, landmark position and measurement noise). Segmenting the UKF in this way benefits the hardware/software approach more so than microprocessor-based approaches since with appropriate choices of processing elements, the time complexity of the each UKF instance could be reduced even further. Particle filter based solutions to the SLAM problem, one, for example, described by Kim et al. (2008), could go another additional step and run UKF instances for each particle in parallel (since each particle is also independent of other particles), either with multiple instantiations of the Serial/Parallel design, or perhaps by using the Pipeline design.

4.3 Summary

In this chapter, implementations of the HW/SW UKF for two example applications were presented. The first application utilised the UKF as part of an attitude determination subsystem for a nanosatellite. Two situations were modelled: a single uncontrolled nanosatellite and a small constellation of five nanosatellites, one of which was

the 'lead' satellite that performed attitude determination for the whole constellation. The HW/SW codesign implementation was compared to a Matlab implementation of the UKF for this application and was found to generate exactly the same state estimation results i.e. no functionality issues arose when taking the UKF to a HW/SW implementation. The Serial and Parallel (2 PE) variants were used in the single nanosatellite case and were found to offer a 1.8× and 2.4× speedup, respectively, over a purely C (SW) implementation of the UKF. The Pipeline (2 PE) design was found to offer a 2.75× speedup in the nanosatellite constellation case.

The second application utilised the UKF as the state estimator within a SLAM system on a small UAV. The focus for the second application was runtime performance of the HW/SW codesign. The Serial and Parallel (2 PE) variants were found to offer a 1.3× and 1.7× speedup over a Matlab implementation of the UKF, respectively, with three landmarks present.

The two example applications presented were representative of the applications the proposed generic state estimation library is targeted at. Although both feature aerospace systems, the differences between the example applications are sufficient to demonstrate the ease of which a system designer can apply the HW/SW codesign to their system. All three variants of the codesign are viable and, though modest, provide performance gains over purely software implementations of the UKF. The example applications show that the HW/SW codesign of the UKF retains the flexibility and portability of software while enjoying the performance benefits of hardware.

Chapter 5

Implementation Analysis of the Hardware/Software Codesign

In this chapter, the physical implementation of the HW/SW codesign is analysed in much greater detail. The HW/SW codesign, all three variants, was implemented for a wide range of parameters in order to demonstrate the flexibility and effectiveness of the design. In particular, the Parallel and Pipeline variants allow for scaling of the processing elements to suit the application and the hardware efficiency of either variant changes dramatically depending on the size of various UKF parameters; for example, the length of the augmented state vector, number of state variables, number of observation variables etc. This chapter presents implementations for three example applications, giving synthesis results, power usage and a timing analysis for each application. Two further analyses are given focussing on the latency of each submodule in the IP core and the effect of scaling the augmented state vector on the latency of the IP core.

5.1 Analysis overview

For all implementations described in this chapter, synthesis and implementation runs were targeted at the Zynq-7000 XC7Z045 (Xilinx, 2016) at a target frequency of

100 MHz. Though the implementations of the example applications presented in Chapter 4 was for the Zynq-7000 XC7Z020, the Parallel and Pipeline design do not fit on this device for larger numbers of processing elements. In order to still compare implementation details, this larger device in the Zynq-7000 family is used instead. All the devices in the Zynq-7000 family feature the same processing system; the only difference for larger devices is the amount of programmable logic available.

Resources utilisation of the device by the IP core is reported by Vivado (Xilinx, 2017a) post-implementation. The power analysis is done via the Xilinx Power Estimator (XPE) (Xilinx, 2017c) post-implementation; all power estimates exclude the device static power dissipation and the processing system power draw.

The execution time (latency) for any hardware part is measured via behavioural simulation in Vivado Simulator (Xilinx, 2017b), assuming a clock frequency of 100 MHz; this assumption was validated post-implementation for all designs. Though behavioural simulations are usually used for only functional verification, Vivado Simulator provides cycle accurate execution times as long as timing assumptions made in the simulation are verified post-implementation. The entire IP core utilises synchronous logic and is on a single clock domain which makes confirming the proper distribution of the assumed clock signals, in this case 100 MHz, relatively straightforward.

The execution time (latency) of any software part is measured via the ARMv7 Performance Monitor Unit (PMU) which counts processor clock cycles between two epochs; because the number of processor clock cycles to perform a given task can vary, each measurement was conducted at least 10 times and the average latency measured is reported here.

Implementations for the three example applications are done to explore the effect certain UKF parameters (see Section 3.1.1) have on the IP core. The first example application is simply an expanded implementation of the design first presented in Section 4.1, the attitude determination subsystem for a nanosatellite; this further reinforces the suitability of the HW/SW codesign for these type of applications. The second example application explores the impact a UKF with larger numbers of obser-

vation variables may have on the hardware. For each of the three variants presented in Chapter 3, the `update` step was the most complex part of the designs. Because the `update` step updates the state estimate with the current observation, increasing the number of observations may have a disproportionately detrimental effect on performance. The third example application explores the effect of fine-tuning the number of processing elements (PEs) used for each module; all the other implementations so far have assumed the same number of PEs for each module but this finer level of tuning is available to the system designer if they wish to use it.

The final two sections explore the scalability of the design further. The impact on the latency of the IP core as the number of PEs is scaled is analysed for each module in the IP core. This allows the system designer to intelligently make decisions about adjusting the number of PEs further. The impact on the latency of the IP core as the augmented state variables is scaled, is also analysed. This is of importance because some applications can have very large numbers of state variables (e.g. thousands for visual SLAM systems); a system designer will want to use the least number of PEs, and thus resources, to achieve an adequate level of performance.

5.2 Example application: Nanosatellites

First, expanded implementation results for the nanosatellite example application described in the previous chapter (Section 4.1) are presented and analysed. Recall that this application, involving the attitude determination of a nanosatellite, had 7 state variables, 6 observation variables and 20 augmented state variables. This section is based on results first presented in Soh and Wu (2014) for the Serial design and Soh and Wu (2017b) for the Pipeline design.

5.2.1 Synthesis results

Synthesis results for the Serial design and a selection of different numbers of processing elements for the Parallel design can be seen in Table 5.1. These results do not include

the processor but do include the logic necessary for the AXI4 interface ports. The initial numbers of PEs were chosen to be multiples of the number of augmented state variables so that the major datapaths remained data efficient. Recall, for example, the Parallel design matrix multiply-add datapath (Section 3.3.2.2); each PE calculates an entire row in the result matrix. If the number of PEs is not a multiple of the size of the matrix, then the last iteration of the calculations will not have enough data to fill all the PEs making the datapath slightly inefficient.

Resource	Serial	2 PE	5 PE	10 PE
FF	7668 (2)	14286 (3)	27311 (6)	48714 (11)
LUT	5764 (3)	15158 (7)	29500 (13)	53427 (24)
BRAM	16.5 (3)	36.5 (7)	62 (11)	109.5 (20)
DSP48	35 (4)	62 (7)	104 (12)	182 (20)

Table 5.1: Resource utilisation (% Total) for the Serial and Parallel designs on the XC7Z045

The resources required by the Serial design are relatively low only needing a small percentage of the available resources. Similarly, the Parallel design, with low numbers of processing elements, utilises a relatively small percentage of the available resources. The XC7Z045 is a mid-range device in the Zynq-7000 series which means even the 10 PE case for the Parallel design still only uses a quarter of the available LUTs. For comparison, synthesis results of the Serial design and the Parallel design for the 2 and 5 PE cases for the XC7Z020 are presented in Table 5.2.

Resource	Serial	2 PE	5 PE
FF	7401 (7)	15813 (15)	30712 (29)
LUT	5941 (11)	13635 (26)	26377 (50)
BRAM	16.5 (12)	36.5 (26)	62 (44)
DSP48	18 (8)	36 (16)	78 (36)

Table 5.2: Resource utilisation (% Total) for the Serial and Parallel designs on the XC7Z020

The XC7Z020 is a low-end device in the Zynq-7000 family but the Serial design only utilises a quarter of available resources and even the Parallel design up to the 3 PE

case uses a reasonably small amount of resources. This means that the Serial design, at the very least, will likely have very few issues being integrated into a full SoC, even on smaller, low-end devices. This is important because these low-end devices are likely to be favoured by cost (monetary, power consumption, physical space) conscious designers as nanosatellite systems are generally severely constrained. The Parallel design, even for low numbers of PEs, uses enough resources that integration into a SoC may be infeasible for low-end devices; use of the Parallel design as a coprocessor instead has greater viability. For nano/microsatellite applications, where a mid-range device such as the XC7Z045 can be used, it may be possible for even the 10 PE case to be integrated into a SoC.

Neither the Serial nor Parallel design require a proportionally large amount of any one resource. This will allow easier integration into a full SoC, particularly if partially-reconfigurable regions are used. Requiring too much of any one resource type can lead to placement and routing issues since resource on-chip locations are fixed by the manufacturer. The Parallel design, however, uses a disproportionately smaller amount of FFs than other resources. This implies additional register stages could be added to major datapaths which would increase the overall latency but could allow an increase in clock frequency as well. If the increase in clock frequency was greater than the increase in latency, the overall performance of the design would benefit.

Synthesis results for the Pipeline design can be seen in Table 5.3. The Pipeline design uses a huge amount of resources compared to the Serial and Parallel designs. The Pipeline 2 PE implementation uses nearly the same amount of resources as the Parallel 10 PE implementation. This most likely makes the Pipeline infeasible on low-end devices, although in mid-range devices the design could still potentially be part of a SoC for low numbers of processing elements. Alternatively, the performance gain of being able to calculate multiple instances of the UKF at once for multiple satellites could be a worthwhile trade-off for needing to use mid-range, or even high-end, devices.

Though the usage of LUTs and DSPs in the Pipeline design has increased massively, the usage of BRAMs stayed roughly the same. This is likely because the data is

Resource	1 PE	2 PE	5 PE	10 PE
FF	29178 (7)	47022 (11)	101549 (23)	192958 (44)
LUT	20823 (10)	33938 (16)	73798 (34)	138907 (64)
BRAM	32.5 (6)	46 (8)	71.5 (13)	129 (24)
DSP48	64 (7)	114 (13)	264 (29)	514 (57)

Table 5.3: Resource utilisation (% Total) for the Pipeline design

being streamed constantly though the design; intermediate results are no longer being stored and so do not need to use additional BRAM resources. Once again, the FF usage is disproportionally smaller than LUT usage, implying that increasing the clock frequency could be possible.

5.2.2 Power consumption

A power consumption breakdown for the hardware IP core (i.e. excluding the processor) of the Serial and Parallel designs can be seen in Table 5.4. The power consumption of the Serial design is reasonably low, due to the area efficiency design goals and the heavy utilisation of the FPGA clock enable resources to disable modules that are not currently in use. For reference, the device static power consumption (@ 25° C) is $\approx$ 245 mW and the rough power consumption of the processing system is $\approx$ 1.5 W. A conservative estimate of the electrical power available to a CubeSat is in the order of 1-2 W per unit (Selva and Krejci, 2012); larger 2-3U or more CubeSats have a greater surface area to cover in solar panels. The Serial design could be incorporated into a 1U or larger CubeSat with relative ease, but the Parallel design looks to be feasible only for 2U CubeSats or larger, even for just the 2 PE case.

A power consumption breakdown of the IP core for the Pipeline design can be seen in Table 5.5. As might be expected, the power consumption of the Pipeline design is much larger than the Serial and Parallel designs. The smaller PE cases (1-2) may be feasible on 2-3U CubeSats or larger but realistically the Pipeline design may only be appropriate for micro-satellites or larger. On the other hand, it is possible that the performance gains of the Pipeline design may outweigh the downsides in power

Resource	Serial	2 PE	5 PE	10 PE
Clocks	38	74	136	234
Signals	24	83	144	261
Logic	23	76	126	219
BRAM	51	82	112	209
DSP	4	6	21	52
Total	140	336	549	975

Table 5.4: Power consumption of the Serial and Parallel designs. All values in mW

consumption, especially for a constellation.

	1 PE	2 PE	5 PE	10 PE
Clocks	131	196	408	754
Signals	130	206	448	836
Logic	116	175	364	666
BRAM	84	111	190	308
DSP	30	53	124	238
Total	491	741	1534	2802

Table 5.5: Power consumption of the Pipeline design. All values in mW

5.2.3 Timing analysis

A breakdown of the execution time (latency) of different modules for the Serial and Parallel designs can be seen in Table 5.6. The design spends a large amount of the time propagating the sigma points through the two system models. In the Parallel design, the majority of the time spent by the design is actually in these system models, making the software part the main bottleneck. Looking at the sigma point propagation process a little closer, however, the latency of reading the sigma points from the memory buffer and writing the transformed points back to the memory buffer was 116 μs. The actual calculation of the system models took a mere 21 μs. So the bottleneck is actually the speed of the AXI4 port in transferring data between the processor and the memory buffer. Using a higher performance communication bus or other techniques such as Direct Memory Access (DMA) ports may alleviate

this issue but intra-chip communication methods are beyond the scope of this thesis.

	SW	Serial	2 PE	5 PE	10 PE
Sig. Gen.	-	-	92	61	51
System model	52	137	137	137	137
Predict	522	170	13	8.5	6.5
Update	87	56	30	21	17
Total	660	363	272	228	212

Table 5.6: Latency of each stage for the Serial and Parallel designs. System models encompasses propagation through both the `predict` and the `update` model on the processor. All values in μs.

For the hardware part, the majority of time is spent in the `sig_gen` step. The two modules in the `sig_gen` step, the triangular linear equations solver and the matrix multiply-add, are both large matrix operations which scale with the number of augmented state variables. Operations in the `predict` and `update` step tend to scale with the number of state or observation variables respectively which are always necessarily smaller than the number of augmented state variables. It should be noted that the hardware part appears to suffer from diminishing returns with regards to decreasing the latency as the number of processing elements increases.

A breakdown of the time spent in different modules for the Pipeline design can be seen in Table 5.7. Each stage of the Pipeline design, as well as the overall latency, is roughly in-line with the Serial and Parallel designs.

	1 PE	2 PE	5 PE	10 PE
Sig. Gen. (a)	64	59	56	55
Sig. Gen. (b)	93	47	19	10
Models	137	137	137	137
Predict	17	11	6.7	4.7
Update	29	21	16	14
Total	340	275	235	221

Table 5.7: Latency of each stage for the Pipeline design. All values in μs.

Comparing with Table 4.4, since the speed of each stage in the pipeline is limited by the PS stage, the overall latency for the HW/SW codesign would normally be

estimated to be roughly 5 times the PS stage. However, there exists further additional overhead when writing the augmented state/covariance into the memory buffer at the start of each UKF instance, and when reading the current state estimate from the memory buffer at the end of each UKF instance so the overall latency ends up being roughly 10 times the longest stage. A timing diagram for the whole pipeline can be seen in Figure 5.1. Given that the software part limits the pipeline, using a larger number of processing elements is mostly unnecessary since the pipeline is already inefficient. Using lower numbers of processing elements may be able to maintain performance while saving resources. Given the inefficiencies caused by the software stage, the system models could also be implemented in hardware for a proper, full hardware pipeline but this would obviously relinquish the portability advantage.

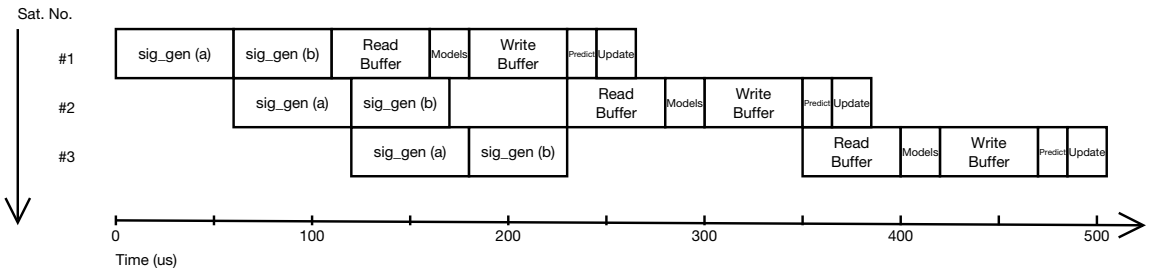


Figure 5.1: Timing of the pipeline for the 2 PE case

5.3 Example application: Large number of observation variables

A second example application is presented to explore what happens when there are a greater number of observation variables than state variables, i.e for $M_{obs} > M_{state}$. The number of observation variables predominantly affects the `update` step as it is the `update` step that uses the observations to update the state estimate and covariance. Because the `update` step is generally the most complex sub-module in any variant of the HW/SW codesign (compare, for example, Figure 3.23 with Figure 3.16), increasing the number of observation variables may have a disproportionate impact on the implementation of the IP core.

Since the primary focus here is the performance of the IP core, rather than the UKF itself, in this example, details of only the hardware IP core are presented. To set some of the relevant parameters, consider the example of: an application using 3-axis attitude and angular velocity (6 state variables), two sets of sensor measurements for both (12 observation variables, double that of the previous section, Section 5.2) and ideal system models f and h, i.e. no errors; the length of the augmented state vector, M, is then 18. Once again, the same number of processing elements is instantiated for each module. The results in this section are based on results that were first presented in Soh and Wu (2017b).

5.3.1 Synthesis results

Synthesis results for the Serial design and a range of processing elements for the Parallel design is given in Table 5.8. The resource usage is roughly the same as in the previous example application. Resource usage seems to be dominated by the number of processing elements rather than changes in the number of state or observation variables.

Resource	Serial	2 PE	5 PE	10 PE
FF	8307 (2)	14392 (3)	27502 (6)	49061 (11)
LUT	6299 (3)	14824 (7)	29026 (13)	52554 (24)
BRAM	16.5 (3)	39 (7)	67.5 (12)	120 (22)
DSP48	35 (4)	66 (7)	108 (12)	184 (20)

Table 5.8: Resource utilisation (% Total) for the Serial and Parallel designs.

Synthesis results for the Pipeline design can be seen in Table 5.9; again the resource usage is virtually the same as the previous example.

5.3.2 Power consumption

A power estimate for the Serial and Parallel designs can be seen in Table 5.10. Both the Serial and Parallel designs utilise slightly more power than in the nanosatellite

Resource	1 PE	2 PE	5 PE	10 PE
FF	29248 (7)	47055 (11)	100996 (23)	191032 (44)
LUT	20811 (10)	33932 (16)	73376 (34)	138949 (64)
BRAM	33 (6)	46.5 (9)	82 (15)	129.5 (24)
DSP48	64 (7)	114 (13)	264 (29)	514 (57)

Table 5.9: Resource utilisation (% Total) for the Pipeline design.

application (Section 5.2). The additional power usage, in this example, appears to be entirely from the BRAMs. The `update` step does use more memory than either the `sig_gen` or the `predict` steps which means that increasing the number of observation variables leads to these memories being larger and could be why this implementation has a slightly higher power consumption.

	Serial	2 PE	5 PE	10 PE
Clocks	44	74	134	233
Signals	29	83	152	249
Logic	28	75	128	208
BRAM	25	101	180	338
DSP	5	22	34	56
Total	131	355	628	1084

Table 5.10: Power consumption for the Serial and Parallel designs. All values in mW

The power estimate for the Pipeline design can be seen in Table 5.11. Unlike the Parallel design, the Pipeline design only shows increases in power consumption for the 5+ processing element cases; however, the majority of increases are in the signals, logic and DSPs rather than the BRAMs. The Pipeline design does not use as much memory as the Serial/Parallel designs because many of the intermediate products need not be stored and so the increase in power consumption may simply be from the increase in activity in the `update` step.

	1 PE	2 PE	5 PE	10 PE
Clocks	127	197	408	721
Signals	135	205	418	920
Logic	121	172	346	720
BRAM	78	105	194	309
DSP	32	54	124	252
Total	493	733	1490	2922

Table 5.11: Power consumption for the Pipeline design. All values in mW

5.3.3 Timing analysis

The latency across each step for the Serial and Parallel designs can be seen in Table 5.12. The IP core now spends roughly the same amount of time in the `sig_gen` and `update` steps, likely because of the `trisolve` module. Overall, the IP core for this implementation is slightly slower than the IP core in the nanosatellite implementation (recall no software stage is used here). The number of augmented state variables decreased by 2 compared to the nanosatellite implementation which affects the slowest step but this appears to be more than offset by the large increase in observation variables.

	SW	Serial	2 PE	5 PE	10 PE
Sig. Gen.	402	116	72	52	43
Predict	31	15	9	8	5
Update	174	115	76	52	44
Total	606	246	157	112	92

Table 5.12: Latency of each step for the Serial and Parallel designs. All values in μs.

The latency across each step for the Pipeline design can be seen in Table 5.13 where, as with the Serial and Parallel designs, the increase in observation variables causes the `update` step to outweigh the reduction in augmented state variables.

	1 PE	2 PE	5 PE	10 PE
Sig. Gen. (a)	51	47	45	44
Sig. Gen. (b)	69	35	16	8
Predict	12	7	6	4
Update	84	55	42	37
Total	216	144	109	93

Table 5.13: Latency of each stage for the Pipeline design. All values in μs.

5.4 Example application: Varied PEs

In the two previous example applications (Sections 5.2 and 5.3), the Parallel and Pipeline designs were synthesised with the same number of processing elements for each module; however, this need not be the case. For system designers who want to quickly and easily implement the UKF with reasonable performance for an application, leaving this 'default' parameterisation of the processing elements may be suitable. However for designers looking to optimise the design further, additional options regarding the parameterisation scheme are available. The nanosatellite application, for example, may be one where heavy optimisation to squeeze every bit of performance from the onboard computing is highly desired. Possible alternative parameterisation schemes for this application (with $M = 20$, $M_{state} = 7$, $M_{obs} = 6$) are listed in Table 5.14; a set of illustrative examples are chosen where one of the major datapaths uses a different number of PEs to the rest of the design.

Scheme No.	Description
1	Parallel: 5 PE, `trisolve` = 2 PE
2	Parallel: 10 PE, `trisolve` = 5 PE
3	Parallel: 2 PE, Matrix multiply-add = 10 PE
4	Pipeline: 1 PE, `sig_gen` = 2 PE

Table 5.14: Possible schemes where the number of processing elements varies between modules.

Synthesis results, power consumption estimates and timing breakdowns can be seen in Tables 5.15, 5.16 and 5.17 respectively; it should be noted that only details of the

hardware part are listed here. Listed in each of these tables are results for all of the alternative parameterisation schemes with the number in the header row corresponding to the scheme in Table 5.14.

Resource	#1	#2	#3	#4
FF	24431 (6)	44640 (10)	14488 (3)	32918 (8)
LUT	25798 (12)	47554 (22)	15297 (7)	23724 (11)
BRAM	60.5 (11)	94.5 (17)	36.5(7)	35.5 (7)
DSP48	82 (9)	154 (17)	46 (5)	72 (8)

Table 5.15: Resource utilisation (% Total) the IP core when the number of processing elements is varied between modules.

Resource	#1	#2	#3	#4
Clocks	126	221	78	144
Signals	126	198	89	144
Logic	106	159	81	129
BRAM	131	174	72	89
DSP	20	29	16	33
Total	509	781	347	539

Table 5.16: Power consumption of the IP core when the number of processing elements is varied between modules. All values in mW

	#1	#2	#3	#4
Sig. Gen. (a)	64	52	55	59
Sig. Gen. (b)	-	-	-	47
Predict	11	11	12	11
Update	24	17	27	21
Total	99	80	94	138

Table 5.17: Latency of the IP core when the number of processing elements is varied between modules. All values in μs.

The first two schemes, where the `trisolve` module is instantiated with less processing elements than the rest of the design, show a roughly 10% reduction in resource usage; this despite very little to negligible timing performance loss when compared to the results presented in Section 5.2 (#1 compared to the 5 PE implementation

and #2 compared to the 10 PE implementation). The similar timing performance is largely due to the fact that greater numbers of processing elements do not necessarily accelerate the Cholesky Decomposition (a closer look at this effect is given in Section 5.5), but also because the number of observation variables is small (e.g. in the 10 PE implementation, since $M_{obs} < N_{PE}$, the `trisolve` datapath during the `update` step is already inefficient). The number of observation variables primarily affects the `update` step calculations, while the number of state variables affects the `predict` variables and, of course, the number of augmented state variables mostly affects the `sig_gen` step. The power consumption savings for these first two schemes, especially for scheme #2, are also quite significant owing to the removal of unnecessary hardware.

Scheme #3 shows an implementation where one module is given a disproportionately greater number of processing elements, in this case, the matrix multiply-add module. Interestingly, the resource usage doesn't actually change compared to the 2 PE implementation in Section 5.2 despite an $\approx 30\%$ reduction in runtime performance. Repeating elements of the comparatively simple matrix multiply-add datapath may have allowed the synthesis or implementation tool in Vivado to use existing hardware resources more efficiently. The LUTs featured by the Zynq-7000 family, for example, are capable of implementing two 4-input logic functions OR one 5-input logic function OR one 6-input logic function (Xilinx, 2016). In this case, it is possible that many more LUTs are being used to their full capacity in implementing 6-input logic functions, which would allow an increase in functionality but not necessarily in resource usage. The power consumption of this scheme is actually slightly higher as well, pointing to greater activity in the resources used.

Scheme #4 shows a case that attempts to even out the latencies of each stage in the Pipeline design. Unlike the Parallel design, where any increase in performance in any module will increase the performance of the whole design, the Pipeline design is limited by the latency of the longest stage. Increasing the processing elements to benefit some stages while another stage still has a longer latency is largely useless. The `predict` and `update` steps are much quicker than the `sig_gen` step so there is

not much point using additional processing elements in either step. Scheme #4 uses two processing elements for the `sig_gen` step to try and equalise the two `sig_gen` stages but leaves only one processing element for the rest of the design. Compared to the 2 PE implementation listed in Section 5.2, scheme #4 has the exact same hardware performance but uses ≈ 30 % less resources and power.

Though only a small selection of possible parameterisation schemes have been presented here, it is clear that plenty of options exist to customise the design for interested designers. System designers can increase the amount of development effort in order to optimise performance in their desired given application or settle for reasonable performance but low development effort and ease of integration with existing systems.

5.5 Latency: UKF steps

A closer look at the latency of each of the `sig_gen`, `predict` and `update` steps is presented in this section. It can be seen in previous implementations that the IP core as a whole suffers from diminishing returns as the number of processing elements increases. For example, in the nanosatellite application (see Table 5.6) going from the Serial design to the 2 PE Parallel design reduces the execution time by $\approx 90\ \mu$s but adding another 8 PEs to implement the 10 PE case only reduces the execution time by $\approx 60\ \mu$s. Although the latency *is* lower, it may not be enough to justify the additional resource usage.

Consider an application with 20 augmented state variables, an even split between the number of state and observation variables and perfect system models (i.e. $M_{state} = M/2$, $M_{obs} = M/2$), and only implementing the Parallel design. Using 20 augmented state variables allows comparison of the latencies here with the nanosatellite application (see Section 5.2) which also features 20 augmented state variables. Splitting the variables evenly between state variables and observation variables eliminates any potential bias they have on the `predict` and `update` steps (since the state variables

predominantly affects the `predict` step while the observation variables predominantly affects the `update` step).

5.5.1 Sigma points generation

Figure 5.2 shows a graph of the latency of the `sig_gen` step versus the number of processing elements. The first thing to note is that the latency of the `trisolve` module barely changes with increasing numbers of processing elements. As alluded to in Section 3.3.2, the Cholesky Decomposition cannot be effectively parallelised. Instantiating additional processing elements for this module appears to be a waste of resources in the `sig_gen` step. Conversely, the other module, the matrix multiply-add, greatly benefits from the additional processing elements. Therefore the `trisolve` module will remain the main hindrance in the `sig_gen` datapath regardless of instantiated processing elements while the matrix multiply-add greatly benefits from the same.

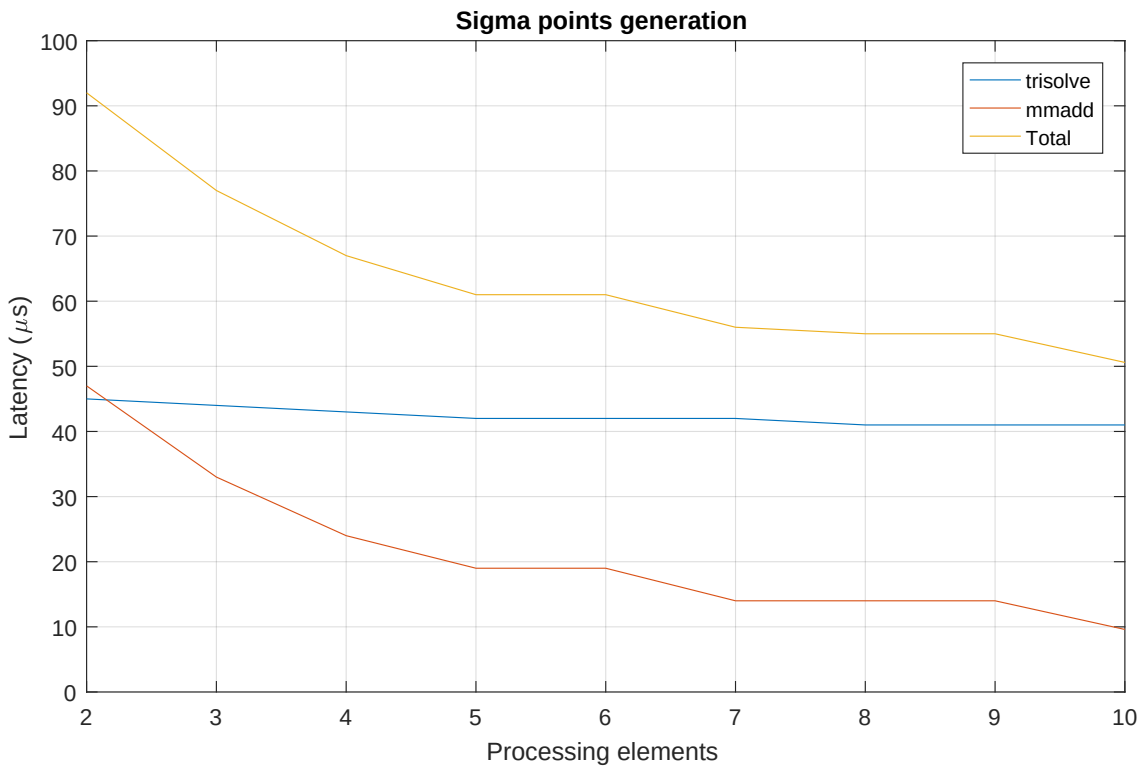


Figure 5.2: Latency vs. processing elements for the `sig_gen` step

5.5.2 Predict step

Figure 5.3 shows a graph of the latency of the `predict` step versus the number of processing elements. It can be seen that none of the modules in the `predict` datapath disproportionately cause any congestion; furthermore, all modules appear to benefit from additional processing elements. For inefficient processing element numbers, i.e. a non-multiple of the state variables, additional processing elements actually slightly increase the latency. However, the total latency of the `predict` step is much lower than the other two steps. Even though the `predict` step benefits from additional processing elements, it may not be necessary to use them since the other steps take much longer in terms of overall latency anyway.

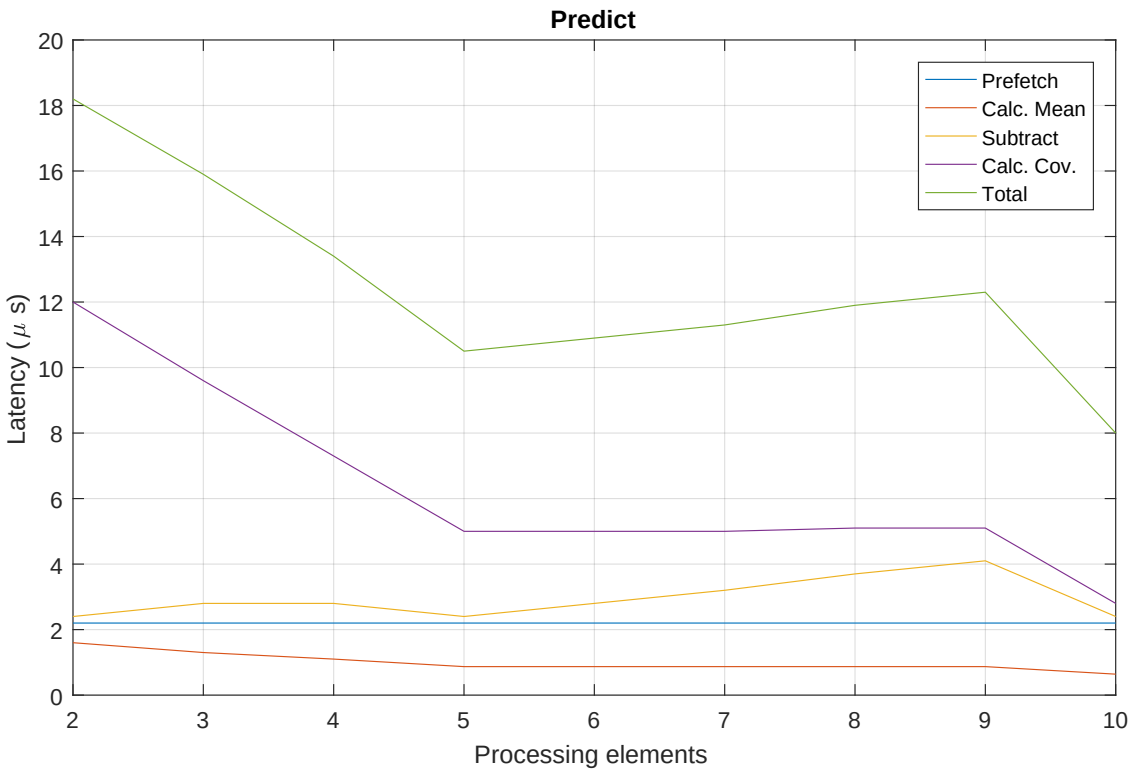


Figure 5.3: Latency vs. processing elements for the `predict` step

5.5.3 Update step

Figure 5.4 shows a graph of the latency of the `update` step versus the number of processing elements. As with the `predict` step, additional processing elements reduce the latency of every module in the `update` step. Unlike the `sig_gen` step, the `trisolve` module here actually does decrease in latency when additional processing elements are used. This is most likely due to the fact that the `trisolve` module here is used for the matrix right 'divide'; i.e. the Cholesky Decomposition followed by forward elimination then back substitution. Although the Cholesky Decomposition cannot be effectively parallelised, the forward elimination and back substitution can be, meaning those operations benefit from additional processing elements. Despite this, the Cholesky Decomposition is still necessary and so, as the number of observation variables increases the `trisolve` will likely become the limiting factor again.

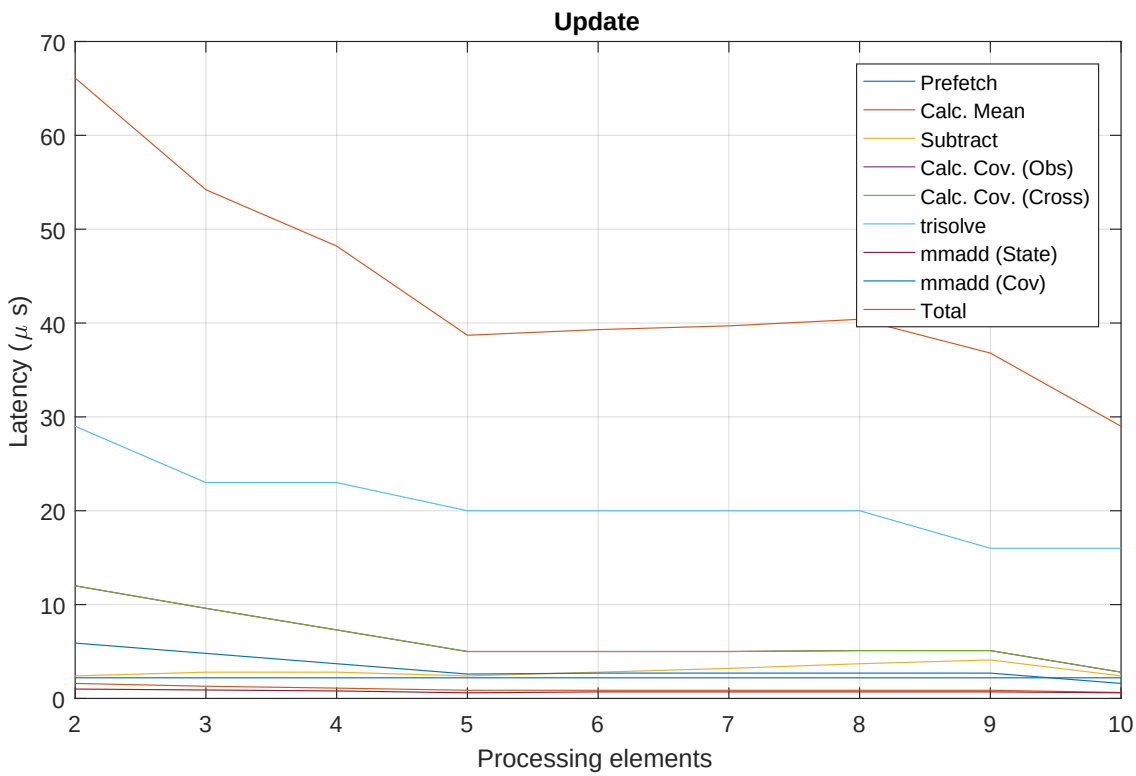


Figure 5.4: Latency vs. processing elements for the `update` step

5.6 Latency: Augmented state variables

The increase in augmented state variables, state variables and observation variables have different impacts on each of the steps for the IP core. In this section, an implementation exploring the effect these variables have on the latency of the design is examined. Consider an application with an even split between the number of state and observation variables and perfect system models (i.e. $M_{state} = M/2$, $M_{obs} = M/2$). The even split between state and observation variables has been chosen to remove any potential bias that might be introduced by having one larger than the other; a small amount of this bias is seen in Section 5.3. Results for only the Serial and Parallel designs are presented; although the Pipeline design can calculate multiple instances of the UKF very quickly, for a single instance of the UKF, the calculation is very similar to the Parallel design.

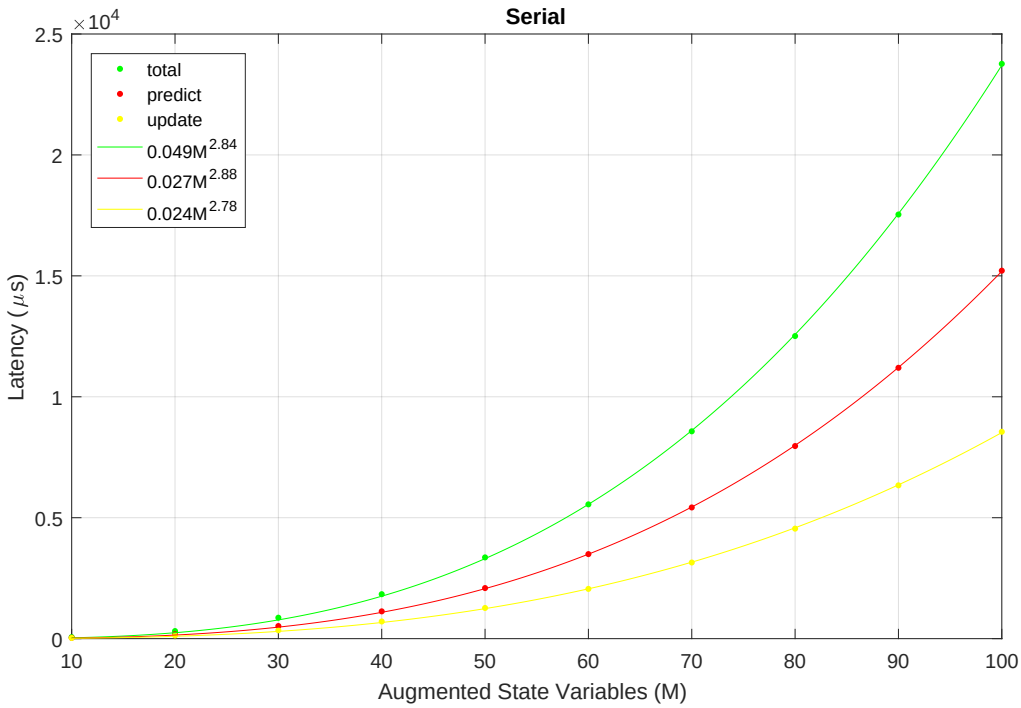


Figure 5.5: Latency vs. augmented state variables for the Serial design

A graph of the latency versus the number of augmented state variables for the Serial design can be seen in Figure 5.5. Power series fits for each of the parts can also be

seen, each of which are roughly $\mathcal{O}(M^{2.8})$. The Cholesky Decomposition in both steps, as well as the large matrix multiplication for sigma points generation, dominate the execution time, especially as the state vector gets larger. The Serial design makes no attempt to really accelerate the UKF over sequential microprocessor-based implementations as it aims to be used within fault-tolerant SoC systems rather than for performance computing; an analysis of the time complexity for microprocessor-based UKFs by Holmes et al. (2009) (with thousands of state variables in fact) notes a time complexity of $\mathcal{O}(M^{2.8})$ which is in agreement with the results presented here.

A graph of the latency of each step versus the number of augmented state variables for the 5 PE case can been seen in Figure 5.6 and for the 10 PE case in Figure 5.7. The number of augmented state variables was capped at a much lower level compared to Figure 5.5 in order to show some of the small effects of the processing elements more clearly. In both cases, and as seen in the previous implementations, the `sig_gen` step takes the longest out of the three steps. The increase in the `sig_gen` step's latency also rises faster than the other two steps.

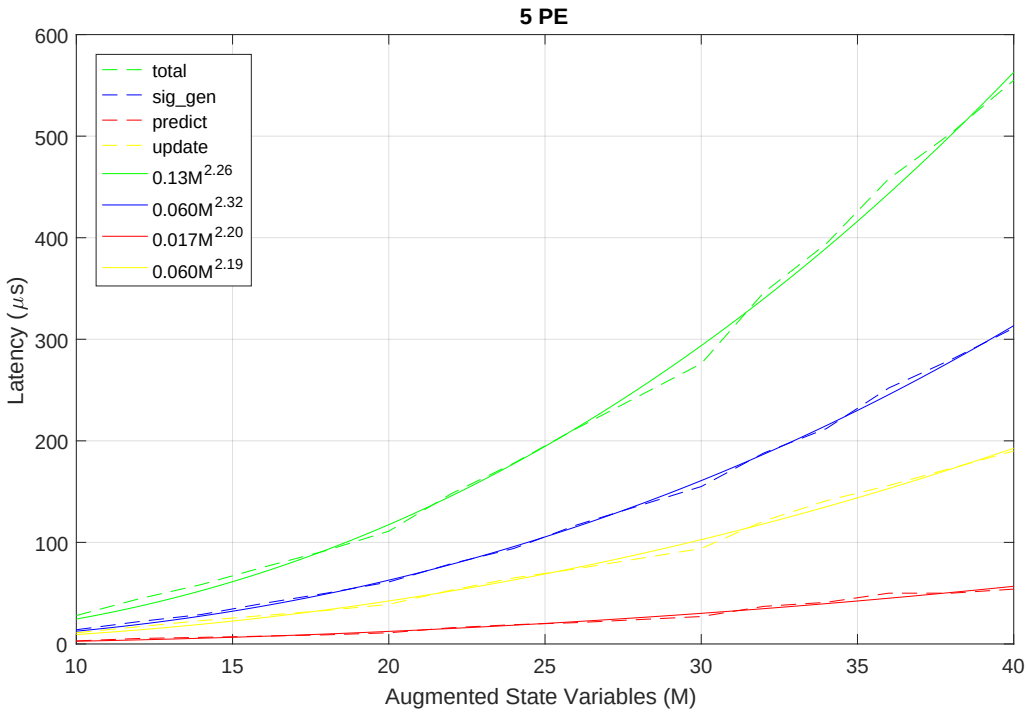


Figure 5.6: Latency vs. augmented state variables for the Parallel design (5 PE)

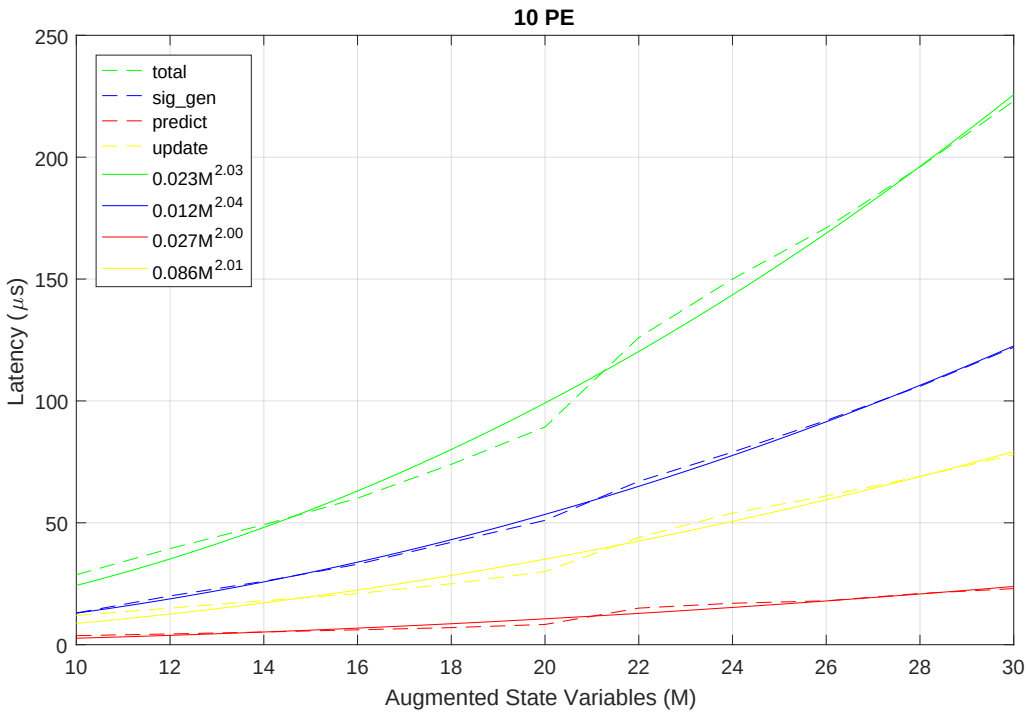


Figure 5.7: Latency vs. augmented state variables for the Parallel design (10 PE)

Small dips in the overall latency can be seen in both cases. This is because the parallelisation scheme of many of the modules discussed in Section 3.3 is most efficient when the number of processing elements is some multiple of the size of the matrix being calculated. For example, consider the matrix multiply-add module described in Section 3.3.2; if the row size of the matrix to be multiplied is 10, and 10 processing elements are used, then the calculation only required one iteration as each processing element calculates one row. If the row size of the matrix to be multiplied is 11-20, then the number of iterations necessary is 2. Thus, for matrices of size 11-19, the module is now somewhat inefficient, since not all processing elements are used every iteration. Going back to Figure 5.6, the small dips can be seen at every multiple of 5 for the total latency and the `sig_gen` curves. This is likely because of the large matrix multiply-add during the `sig_gen` step. The dips are less pronounced in the `update` step and negligible in the `predict` step; however, the `predict` step already has comparatively low latency. In Figure 5.7, it can be seen that although there is a very obvious dip at $M = 20$, after that the curves are more or less smooth; for larger

numbers of processing elements this effect seems to become negligible.

For lower numbers of the augmented state variables, the extra processing elements are able to reduce the time complexity of the UKF. The 10 processing element case is reduced to quadratic complexity, but the 5 processing element case is only reduced to $\approx \mathcal{O}(M^{2.3})$. As the augmented state vector grows much larger than the number of processing elements, the impact of the parallelisation becomes smaller. Figure 5.8 shows the 10 processing element case for much larger augmented state vectors where the complexity at $\mathcal{O}(M^{2.5})$ is not quite as poor as the Serial design. Figure 5.9 shows the latency for the 20 processing element case with two power series fits for augmented state variables lower and higher than the number of processing elements. There is an increase in complexity as the augmented state vector passes the 20 mark and at these low numbers of augmented state variables (compared to the number of processing elements), the complexity itself is even less than quadratic.

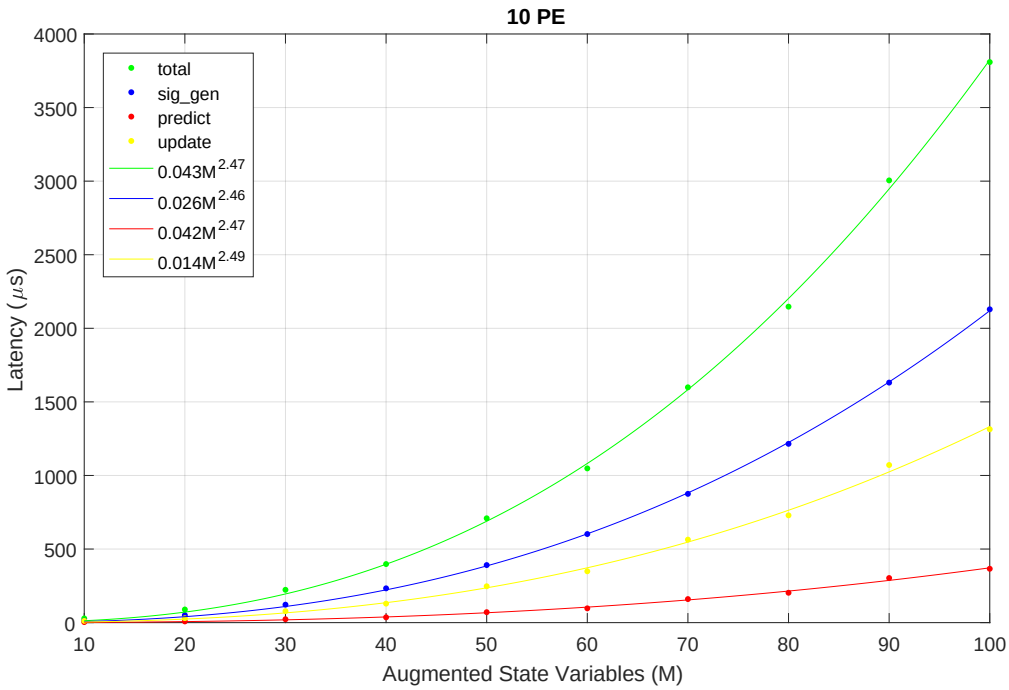


Figure 5.8: Latency vs. augmented state variables for the Parallel design (10 PE) (Large)

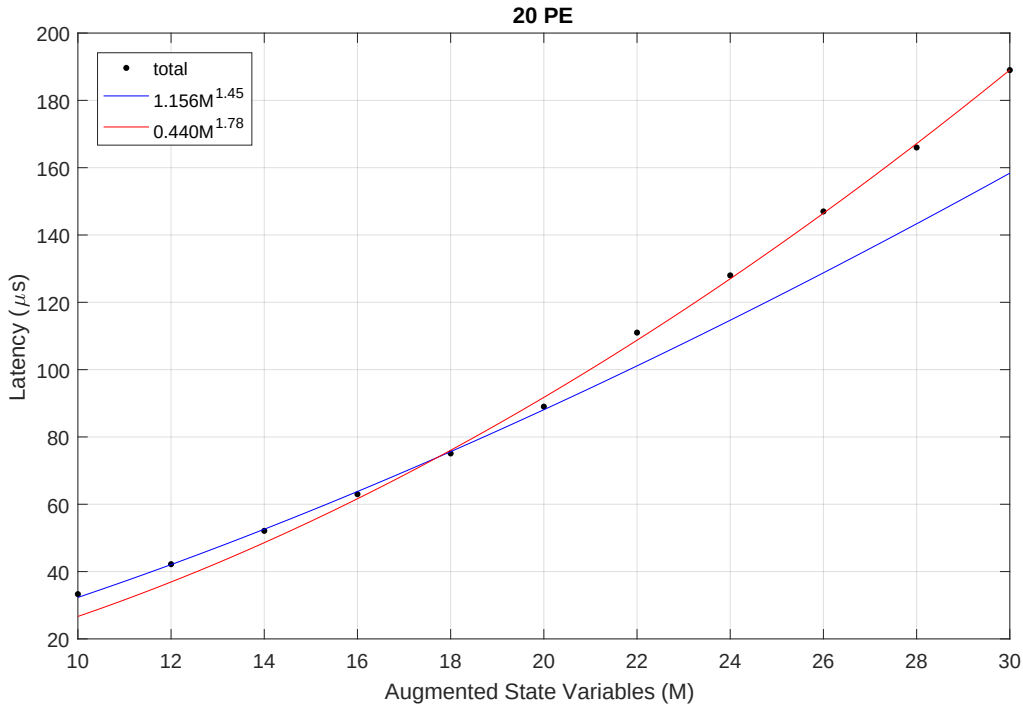


Figure 5.9: Latency vs. augmented state variables for the Parallel design (20 PE)

5.7 Summary

This chapter analysed the implementation of the HW/SW codesign in detail. Implementations for three example applications were presented: an expanded implementation of the nanosatellite application first described in Section 4.1, a theoretical implementation which featured a large number of observation variables and an implementation where alternative parameterisation schemes for the number of PEs were explored. Two additional analyses of the effect of the number of processing elements and the effect of the number of augmented state variables on the latency of the IP core were also given.

Implementation details, i.e. resource utilisation, power consumption and a timing breakdown, were presented for each of the three example applications. The results demonstrate the HW/SW codesign's flexibility by showing how a system designer can trade resources for additional performance as desired. The two latency analyses reinforce this idea, showing how the HW/SW codesign can be optimised further if

the system designer wishes to spend the extra time and effort. The final analysis in particular shows that the HW/SW codesign scales no worse than a traditional software UKF and potentially, if sufficient resources are spent, scales much better. Thus the proposed HW/SW codesign has been shown to be a flexible and scalable implementation of the UKF, suitable for a generic state estimation library.

Chapter 6

Conclusion

6.1 Summary

In this thesis, a scalable, portable FPGA-based implementation of the Unscented Kalman Filter was presented. The proposed design balances development effort/complexity with performance, combining the advantages of both the traditional software approach and hardware approaches to create a library that system designers can easily use in a potentially wide variety of applications.

Chapter 1 describes the issue at hand: an eagerness to accelerate the development of autonomous systems creates a large demand for fast and accurate state estimation, especially for aerospace applications, but a simultaneous desire to miniaturise aerospace systems and minimise development effort, primarily driven by cost concerns, leads to heavy constraints on computing power, electrical power and physical space available on said systems. The traditional software approach has great portability between different applications and comparatively simple development processes but can have lacklustre performance; while the specialised hardware approach trades a large increase in performance for long and expensive development. Instead, a hardware/software codesign utilises the strengths of both.

Chapter 2 gives background into the Field Programmable Gate Array, the technologies

it is based on, its common development process and its viability in many applications, including space (astronautic) applications. The potential to use the FPGA to implement a System-on-Chip, where multiple 'black boxes' or IP cores are implemented together on a single chip to form a computing system with diverse functionality, is explored including a brief introduction to intra-chip communication and, of course, the idea of hardware/software codesign. The 'workhorse' of state estimation, the Extended Kalman Filter, and its shortcomings is described before introducing a potential replacement in the Unscented Kalman Filter. Finally, existing attempts at hardware or hardware/software Kalman filters and its variants are detailed.

Chapter 3 presents the three variants of the proposed codesign. The Serial variant is a straight translation of the UKF into hardware, forgoing the main benefit of hardware – namely, parallelism – in order to minimise resource usage and power consumption such that it is an attractive option for inclusion in fault tolerance reconfigurable systems or other SoCs. The Parallel variant does leverage the main benefit of hardware, using multiple parallel instances of the critical datapaths to accelerate performance with the intent of being used as a coprocessor in high performance computing systems. The last variant, Pipeline, adapts a common hardware abstraction to create a high throughput IP core capable of calculating multiple independent UKFs extremely quickly.

Chapter 4 presents simulations of two example applications demonstrating the effectiveness of the UKF and the proposed hardware/software codesign. The first application simulates the attitude determination of a singular uncontrolled nanosatellite as well as a constellation of five uncontrolled nanosatellites. The UKF is able to converge quickly and maintain an accurate state estimate for the duration of the simulation. The hardware/software codesign, using the same simulated datasets is shown to be capable of completely replicating the UKF with no functionality issues. The Serial and Parallel (2 PE) designs offer a 1.8× and 2.4× speedup respectively, over a similar, purely software, implementation when simulating the singular nanosatellite. The Pipeline (2 PE) design offers a 2.75× speedup when simulating the nanosatellite constellation. The second application simulates the state estimation part of a monocular Simultaneous Localisation and Mapping system on a small UAV. The UKF is able

to accurately estimate the UAV position as well as the maximum of three observed landmarks for the duration of the simulation. The Serial design offers up to a 1.3× speedup and the Parallel (2 PE) design offers up to a 1.7× speedup over a Matlab implementation when three landmarks are being observed.

Chapter 5 presents implementation results for a variety of situations, including resource usage, power consumption and timing latency, demonstrating the flexibility of the codesign library which allows for different parameterisation schemes. The first example application is the same nanosatellite application presented earlier with results for the Serial design and the 1, 2, 5, 10 PE cases for the Parallel and Pipeline designs. The second application is a theoretical application where the number of observation variables are greater than the number of state variables in order to explore any potential biases; results for the Serial design and the 1, 2, 5, 10 PE cases of the Parallel and Pipeline designs are given. The final application is another theoretical application where the number of processing elements per module is varied; four different parametermisation schemes for the Parallel design only are explored. A closer look at the impact of the number of processing elements on the `sig_gen`, `predict` and `update` steps in only the Parallel design is given; it is seen that the Cholesky Decomposition largely does not benefit from more processing elements and acts as a drag on performance. Finally, the impact of the number of augmented state variables on the latency of the design is examined. The Serial design is shown to have similar time complexity, $\mathcal{O}(M^{2.8})$, to microprocessor-based implementations of the UKF. The Parallel design reduces to quadratic complexity for numbers of augmented state variables comparable to the number of processing elements but tends to $\mathcal{O}(M^{2.5})$ for larger numbers. For numbers of augmented state variables equal to the number of processing elements or less, the complexity is below quadratic.

6.2 Main contributions

The main contribution of this thesis is the hardware/software (HW/SW) codesign of the Unscented Kalman Filter (UKF). A need for fast and accurate state estimation

for small aerospace systems was identified. The need for high performance in these systems is offset by the desire to limit overall costs which leads to a reduction in available physical space, computing power and electrical power; it is strongly desired to simplify development processes as well. Hardware approaches, such as using a Field Programmable Gate Array (FPGA), can provide the level of performance required and, if using a System-on-Chip, can adhere to severe physical and electrical power constraints; however, FPGAs increase development complexity compared to software approaches and so do not necessarily reduce costs.

A HW/SW codesign takes the performance gains of a hardware approach and combines it with the flexibility and portability of a software approach. The portability means the development costs of subsequent aerospace systems are reduced, potentially back down to feasible levels. When the HW/SW codesign methodology is applied to a prolific state estimation algorithm in the UKF, the result is a high performance state estimation implementation that is also widely applicable and could be used in a generic state estimation library.

The proposed HW/SW codesign of the UKF described in this thesis splits the application-specific and the non-application specific parts of the UKF algorithm and implements the application-specific parts in software while implementing the non-application-specific parts in hardware as a parameterisable IP core. This allows the HW/SW codesign to make use of the simpler software development processes when moving to a new application, while still enjoying hardware acceleration for the remainder of the algorithm. The proposed HW/SW codesign includes three variations: the Serial design, the Parallel design and the Pipeline design. The Serial design is the most basic and only provides a direct implementation of the UKF; the Serial design uses the least amount of resources. The Parallel design makes use of parallelism in its major datapaths to provide performance boosts; the Parallel design can use a low or high amount of resources depending on the exact parameterisation scheme. The Pipeline design makes use of top-level parallelism, in addition to parallelised datapaths, to calculate multiple instances of the UKF at once; the Pipeline design uses the most amount of resources. The overall theme of these variants is that a system designer

can choose the balance between resources used and performance as they desire. Thus, the proposed HW/SW codesign is both a portable and scalable implementation of the UKF.

The proposed HW/SW codesign is implemented in two illustrative example applications for validation. A nanosatellite application with two related situations, a single nanosatellite and a nanosatellite constellation, is presented. Here, the UKF is part of the attitude determination subsystem of the nanosatellite. The HW/SW codesign is found to completely replicate the UKF with no functionality issues and provides modest performance boosts over similar purely software implementations. The second example application is the state estimation part within a Simultaneous Localisation and Mapping (SLAM) system on a small Unmanned Aerial Vehicle (UAV). The HW/SW codesign once again provides modest performance boosts over purely software implementations. These two example applications are representative of the aerospace systems the HW/SW codesign is targeted at and they show the HW/SW codesign does indeed boost performance while retaining portability.

A series of deeper analyses of the HW/SW codesign's physical implementation is also presented. The HW/SW codesign is implemented for a variety of parameterisation schemes in three example applications. The implementation of the nanosatellite application used for validation is expanded, a theoretical application with a large number of observation variables is implemented before, finally, an application where the number of processing elements (PEs) varies between modules is implemented. Two further analyses on the effect of the number of processing elements and augmented state variables on the latency of the IP core are given. These example applications and analyses show the flexibility of the IP core, allowing the system designer to optimise the performance of the IP core if they desire, but still providing adequate performance if they don't. They also show the HW/SW codesign, at worst, scales as well as an ordinary software implementation of the UKF but, at best, scales far better; the choice is up to the system designer to use resources to gain additional performance.

Thus, this thesis describes a scalable, portable, FPGA-based implementation of the UKF which makes use of HW/SW codesign techniques to provide a foundation for a

generic state estimation library.

6.3 Future work

The first area of future work should explore intra-chip communication technologies and how to integrate the proposed design into a full SoC. As noted in Chapter 5, one of the main areas of congestion in the design is the interaction between the processing system and the IP core. Although the propagation of the sigma points through the system models is not necessarily a long process, the writing and reading of the sigma points and transformed sigma points to and from the IP core memory buffer is obviously highly dependant on the communication interface. In the examples presented here, the general-purpose AXI4 port acted as somewhat of a hindrance, resulting in the software part being the longest latency aspect of the design. However, the communication method used may be dictated by other requirements in a SoC implementation, so integration into a proper SoC with the goal of exploring the effects of those requirements on the performance of the codesign is another potential area of focus.

Whether within a SoC or standalone embedded system, implementation of the codesign for a 'real' system, as opposed to the simulated systems presented here, could be tested. Though care was taken to produce high fidelity simulations, the nanosatellite and SLAM applications presented are obviously only a starting point toward more realistic implementations. Usage of the design within a predominantly COTS nanosatellite computing system or trying to integrate an FPGA and the codesign onto a small UAV to attempt hardware-in-loop simulations could be the next step for future work.

Taking the system models and translating them into hardware is also a potential option. Though this contradicts the design philosophy used throughout this thesis, and forgoes many of its benefits, for certain applications where performance demands are high, in particular where the Pipeline design may be useful, an interested designer may be willing to spend the additional development effort. The IP core functions as

described as long as the appropriate control bits are set and valid data is placed in the memory buffer, regardless of what is on the other end of the buffer. If, instead of a communication interface, a secondary IP core implementing the system models was attached, as long as that IP core also managed the control register, the UKF codesign would still function as is. This means a designer would still save on development effort overall, since most of the UKF is already implemented, but could squeeze even more performance out of the hardware and potentially negate the largest source of latency.

References

Actel (2012). *ProASIC3 FPGA Fabric User's Guide*. Revision 4.

Altera (2016). *Cyclone IV FPGA Device Family Overview*. CYIV-51001-2.0.

Amiri, Kiarash, Joseph R. Cavallaro, Chris Dick, and Raghu Mysore Rao (2011). "A High Throughput Configurable SDR Detector for Multi-user MIMO Wireless Systems". English. In: *Journal of Signal Processing Systems* 62.2, pp. 233–245. ISSN: 1939-8018. DOI: 10.1007/s11265-009-0360-5. URL: http://dx.doi.org/10.1007/s11265-009-0360-5.

ARM (2013). *ARM AMBA AXI and ACE Protocol Specification*. Issue E.

Aung, Yan Lin, Siew-Kei Lam, and T. Srikanthan (2013). "Hardware-Software Codesign of EKF-Based Motor Control for Domain-Specific Reconfigurable Platform". In: *Electronic System Design (ISED), 2013 International Symposium on*, pp. 93–97. DOI: 10.1109/ISED.2013.25.

AVNet (2014). *Zedboard Hardware User's Guide*. v2.2.

Aysu, A., C. Patterson, and P. Schaumont (2013). "Low-cost and area-efficient FPGA implementations of lattice-based cryptography". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 81–86. DOI: 10.1109/HST.2013.6581570.

Azarderakhsh, R. and A. Reyhani-Masoleh (2015). "Parallel and High-Speed Computations of Elliptic Curve Cryptography Using Hybrid-Double Multipliers". In: *IEEE Transactions on Parallel and Distributed Systems* 26.6, pp. 1668–1677. ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.2323062.

Bacon, David F, Rodric Rabbah, and Sunil Shukla (2013). "FPGA programming for the masses". In: *Communications of the ACM* 56.4, pp. 56–63.

Bahri, I., L. Idkhajine, E. Monmasson, and M.E.A. Benkhelifa (2013). "Optimal hardware/software partitioning of a system on chip FPGA-based sensorless AC drive current controller". In: *Mathematics and Computers in Simulation* 90. ELECTRIMACS 2011 - PART I, pp. 145–161. ISSN: 0378-4754. DOI: http://dx.doi.org/10.1016/j.matcom.2012.06.008. URL: http://www.sciencedirect.com/science/article/pii/S0378475412001437.

Baklouti, M., Ph. Marquet, J.L. Dekeyser, and M. Abid (2015). "FPGA-based manycore System-on-Chip design". In: *Microprocessors and Microsystems* 39.4, pp. 302–312. ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2015.

03.007. URL: http://www.sciencedirect.com/science/article/pii/S0141933115000320.

Becker, J., M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka (2007). "Dynamic and Partial FPGA Exploitation". In: *Proceedings of the IEEE* 95.2, pp. 438–452. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.888404.

Bergsman, P. (2003). "Xilinx FPGA blasted into orbit". In: *Xcell Journal* 46, pp. 86–88.

Biradar, R. G., A. Chatterjee, P. Mishra, and K. George (2015). "FPGA implementation of a multilayer Artificial Neural Network using System-on-Chip design methodology". In: *2015 International Conference on Cognitive Computing and Information Processing(CCIP)*, pp. 1–6. DOI: 10.1109/CCIP.2015.7100683.

Bonato, V., R. Peron, D.F. Wolf, J.A.M. de Holanda, E. Marques, and J.M.P. Cardoso (2007). "An FPGA Implementation for a Kalman Filter with Application to Mobile Robotics". In: *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pp. 148–155. DOI: 10.1109/SIES.2007.4297329.

Bonato, Vanderlei, Eduardo Marques, and George A. Constantinides (2009). "A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots". English. In: *Journal of Signal Processing Systems* 56.1, pp. 41–50. ISSN: 1939-8018. DOI: 10.1007/s11265-008-0257-8. URL: http://dx.doi.org/10.1007/s11265-008-0257-8.

Bossuet, Lilian, Michael Grand, Lubos Gaspar, Viktor Fischer, and Guy Gogniat (2013). "Architectures of Flexible Symmetric Key Crypto Engines&Mdash;a Survey: From Hardware Coprocessor to Multi-crypto-processor System on Chip". In: *ACM Comput. Surv.* 45.4, 41:1–41:32. ISSN: 0360-0300. DOI: 10.1145/2501654.2501655. URL: http://doi.acm.org/10.1145/2501654.2501655.

Bouwmeester, J. and J. Guo (2010). "Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology". In: *Acta Astronautica* 67.7-8, pp. 854–862. ISSN: 0094-5765. DOI: http://dx.doi.org/10.1016/j.actaastro.2010.06.004. URL: http://www.sciencedirect.com/science/article/pii/S0094576510001955.

Brzoza-Woch, R. and P. Nawrocki (2016). "FPGA-Based Web Services – Infinite Potential or a Road to Nowhere?" In: *IEEE Internet Computing* 20.1, pp. 44–51. ISSN: 1089-7801. DOI: 10.1109/MIC.2015.23.

Caffrey, M., K. Morgan, D. Roussel-Dupre, S. Robinson, A. Nelson, A. Salazar, M. Wirthlin, W. Howes, and D. Richins (2009). "On-orbit flight results from the reconfigurable cibola flight experiment satellite (CFESat)". In: *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on.* IEEE, pp. 3–10.

Chekired, F., A. Mellit, S.A. Kalogirou, and C. Larbes (2014). "Intelligent maximum power point trackers for photovoltaic applications using FPGA chip: A comparative study". In: *Solar Energy* 101.Supplement C, pp. 83–99. ISSN: 0038-092X. DOI: https://doi.org/10.1016/j.solener.2013.12.026. URL: http://www.sciencedirect.com/science/article/pii/S0038092X13005513.

Chen, R. and V. K. Prasanna (2016). "Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 212–219. DOI: 10.1109/FCCM.2016.62.

Chodowiec, Pawel and Kris Gaj (2003). "Very Compact FPGA Implementation of the AES Algorithm". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Ed. by Colin D. Walter, ÇetinK. Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 319–333. ISBN: 978-3-540-40833-8. DOI: 10.1007/978-3-540-45238-6_26. URL: http://dx.doi.org/10.1007/978-3-540-45238-6_26.

Chu, P.P. (2008). *FPGA prototyping by Verilog examples: Xilinx Spartan-3 version*. Wiley-Interscience.

Civera, J., A. J. Davison, and J. M. M. Montiel (2008). "Inverse Depth Parametrization for Monocular SLAM". In: *IEEE Transactions on Robotics* 24.5, pp. 932–945. ISSN: 1552-3098. DOI: 10.1109/TRO.2008.2003276.

Compton, K. and S. Hauck (2002). "Reconfigurable computing: a survey of systems and software". In: *ACM Computing Surveys (csuR)* 34.2, pp. 171–210.

Contreras, L., S. Cruz, J. M. S. T. Motta, and C. H. Llanos (2015a). "Hardware Architecture of the EKF Prediction Stage applied to mobile robot localization". In: *2015 IEEE 6th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4. DOI: 10.1109/LASCAS.2015.7250446.

Contreras, Luis, Sérgio Cruz, J. M. S. T. Motta, and Carlos H. Llanos (2015b). "Hardware and Software Co-design for the EKF Applied to the Mobile Robotics Localization Problem". In: *International Journal of Machine Learning and Computing* 5.2, p. 101.

Crassidis, J. L., F. L. Markley, and Y. Cheng (2007). "Survey of nonlinear attitude estimation methods". In: *Journal of Guidance Control and Dynamics* 30.1, p. 12.

Crassidis, John L. and F. Landis Markley (2003). "Unscented filtering for spacecraft attitude estimation". In: *Journal of guidance, control, and dynamics* 26.4, pp. 536–542.

Cruz, S., D.M. Munoz, M. Conde, C.H. Llanos, and G.A. Borges (2013). "FPGA implementation of a sequential Extended Kalman Filter algorithm applied to mobile robotics localization problem". In: *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*, pp. 1–4. DOI: 10.1109/LASCAS.2013.6519021.

Cummings, M. and S. Haruyama (1999). "FPGA in the software radio". In: *Communications Magazine, IEEE* 37.2, pp. 108–112. ISSN: 0163-6804. DOI: 10.1109/35.747258.

Daoud, Luka, Dawid Zydek, and Henry Selvaraj (2014). "A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing". In: *Advances in Systems Science: Proceedings of the International Conference on Systems Science 2013 (ICSS 2013)*. Ed. by Jerzy Swiątek, Adam Grzech, Paweł Swiątek, and Jakub M. Tomczak. Cham: Springer International

Publishing, pp. 483–492. ISBN: 978-3-319-01857-7. DOI: 10.1007/978-3-319-01857-7_47. URL: https://doi.org/10.1007/978-3-319-01857-7_47.

Davidson, J. (1993). "FPGA implementation of a reconfigurable microprocessor". In: *Custom Integrated Circuits Conference, 1993., Proceedings of the IEEE 1993*, pp. 3.2.1–3.2.4. DOI: 10.1109/CICC.1993.590366.

Dawood, A. S., S. J. Visser, and J. A. Williams (2002). "Reconfigurable FPGAS for real time image processing in space". In: *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*. Vol. 2. IEEE, pp. 845–848.

Dimond, R., S. Racaniere, and O. Pell (2011). "Accelerating Large-Scale HPC Applications Using FPGAs". In: *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pp. 191–192. DOI: 10.1109/ARITH.2011.34.

Dumitriu, V., L. Kirischian, and V. Kirischain (2012). "A framework for adaptive reconfigurable space-borne computing platforms for run-time self-recovery from transient and permanent hardware faults". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pp. 280–287. DOI: 10.1109/AHS.2012.6268663.

Durrant-Whyte, H. and Tim Bailey (2006). "Simultaneous localization and mapping: part I". In: *Robotics Automation Magazine, IEEE* 13.2, pp. 99–110. ISSN: 1070-9932. DOI: 10.1109/MRA.2006.1638022.

Dyken, Jason Van and José G. Delgado-Frias (2010). "FPGA schemes for minimizing the power-throughput trade-off in executing the Advanced Encryption Standard algorithm". In: *Journal of Systems Architecture* 56.2–3, pp. 116–123. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2009.12.001. URL: http://www.sciencedirect.com/science/article/pii/S1383762109000800.

Fiethe, B., F. Bubenhagen, T. Lange, H. Michalik, H. Michel, J. Woch, and J. Hirzberger (2012). "Adaptive hardware by dynamic reconfiguration for the Solar Orbiter PHI instrument". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pp. 31–37. DOI: 10.1109/AHS.2012.6268666.

Flesch, G., D. Keymeulen, D. Dolman, C. Holyoake, and D. McKee (2017). "A System-On-Chip platform for Earth and Planetary Laser Spectrometers". In: *2017 IEEE Aerospace Conference*, pp. 1–12. DOI: 10.1109/AERO.2017.7943935.

Fossati, L. and J. Ilstad (2011). "The future of embedded systems at ESA: Towards adaptability and reconfigurability". In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pp. 113–120. DOI: 10.1109/AHS.2011.5963924.

Fraser, B.J, C.T Russell, J.D Means, F.W Menk, and C.L Waters (2000). "FedSat — An Australian research microsatellite". In: *Advances in Space Research* 25.7–8. Proceedings of the DO.1 Symposium of COSPAR Scientific Commission D, pp. 1325–1336. ISSN: 0273-1177. DOI: 10.1016/S0273-1177(99)00641-9. URL: http://www.sciencedirect.com/science/article/pii/S0273117799006419.

Fritz, Michael, Sebastian Winter, Juergen Freund, Stefan Pflueger, Oliver Zeile, Jens Eickhoff, and Hans-Peter Roeser (2015). "Hardware-in-the-loop environment for verification of a small satellite's on-board software". In: *Aerospace Science and*

Technology 47, pp. 388–395. ISSN: 1270-9638. DOI: http://dx.doi.org/10.1016/j.ast.2015.09.020. URL: http://www.sciencedirect.com/science/article/pii/S1270963815002783.

García, Gabriel J., Carlos A. Jara, Jorge Pomares, Aiman Alabdo, Lucas M. Poggi, and Fernando Torres (2014). "A Survey on FPGA-Based Sensor Systems: Towards Intelligent and Reconfigurable Low-Power Sensors for Computer Vision, Control and Signal Processing". In: *Sensors* 14.4, pp. 6247–6278. ISSN: 1424-8220. DOI: 10.3390/s140406247. URL: http://www.mdpi.com/1424-8220/14/4/6247.

Gelb, Arthur (1974). *Applied optimal estimation*. MIT press.

Ghallab, Y. H. and Y. Ismail (2014). "CMOS Based Lab-on-a-Chip: Applications, Challenges and Future Trends". In: *IEEE Circuits and Systems Magazine* 14.2, pp. 27–47. ISSN: 1531-636X. DOI: 10.1109/MCAS.2014.2314264.

El-Ghazawi, T., E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell (2008). "The promise of high-performance reconfigurable computing". In: *Computer* 41.2, pp. 69–76.

Giannitrapani, A., N. Ceccarelli, F. Scortecci, and A. Garulli (2011). "Comparison of EKF and UKF for Spacecraft Localization via Angle Measurements". In: *IEEE Transactions on Aerospace and Electronic Systems* 47.1, pp. 75–84. ISSN: 0018-9251. DOI: 10.1109/TAES.2011.5705660.

Golub, Gene H. and Charles F. Van Loan (1996). *Matrix computations*. Third. Baltimore: Johns Hopkins University Press.

Gomperts, A., A. Ukil, and F. Zurfluh (2011). "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications". In: *Industrial Informatics, IEEE Transactions on* 7.1, pp. 78–89. ISSN: 1551-3203. DOI: 10.1109/TII.2010.2085006.

Grillmayer, Georg, Albert Falke, and Hans-Peter Roeser (2005). "Technology Demonstration with the Micro-satellite Flying Laptop". In: *Small Satellites for Earth Observation: Selected Proceedings of the 5th International Symposium of the International Academy of Astronautics, Berlin, April 4-8 2005*. De Gruyter, p. 419.

Guo, H., H. Chen, F. Xu, F. Wang, and G. Lu (2012). "Implementation of EKF for Vehicle Velocities Estimation on FPGA". In: *Industrial Electronics, IEEE Transactions on* PP.99, p. 1. ISSN: 0278-0046. DOI: 10.1109/TIE.2012.2208436.

Guo, Shuxiang, Shaowu Pan, Xiaoqiong Li, Liwei Shi, Pengyi Zhang, Ping Guo, and Yanlin He (2017). "A system on chip-based real-time tracking system for amphibious spherical robots". In: *International Journal of Advanced Robotic Systems* 14.4, p. 1729881417716559. DOI: 10.1177/1729881417716559. eprint: https://doi.org/10.1177/1729881417716559. URL: https://doi.org/10.1177/1729881417716559.

Haddow, P.C. and A.M. Tyrrell (2011). "Challenges of evolvable hardware: past, present and the path to a promising future". In: *Genetic Programming and Evolvable Machines* 12.3, pp. 183–215.

Hamada, Tsuyoshi and Yuichiro Shibata (2013). "FPGA-Based HPRC Systems for Scientific Applications". In: *High-Performance Computing Using FPGAs*. Ed. by Wim Vanderbauwhede and Khaled Benkrid. New York, NY: Springer New York, pp. 367–387. ISBN: 978-1-4614-1791-0. DOI: 10.1007/978-1-4614-1791-0_12. URL: https://doi.org/10.1007/978-1-4614-1791-0_12.

Hartley, E. N., J. L. Jerez, A. Suardi, J. M. Maciejowski, E. C. Kerrigan, and G. A. Constantinides (2014). "Predictive Control Using an FPGA With Application to Aircraft Control". In: *IEEE Transactions on Control Systems Technology* 22.3, pp. 1006–1017. ISSN: 1063-6536. DOI: 10.1109/TCST.2013.2271791.

He, Chun, A. Papakonstantinou, and Deming Chen (2009). "A novel SoC architecture on FPGA for ultra fast face detection". In: *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pp. 412–418. DOI: 10.1109/ICCD.2009.5413122.

Herbordt, M.C., T. VanCourt, Yongfeng Gu, B. Sukhwani, A. Conti, J. Model, and D. Di Sabello (2007). "Achieving High Performance with FPGA-Based Computing". In: *Computer* 40.3, pp. 50–57. ISSN: 0018-9162. DOI: 10.1109/MC.2007.79.

Hoang, Trang and Van Loi Nguyen (2012). "An Efficient FPGA Implementation of the Advanced Encryption Standard Algorithm". In: *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on*, pp. 1–4. DOI: 10.1109/rivf.2012.6169845.

Hodjat, A. and I. Verbauwhede (2004). "A 21.54 Gbits/s fully pipelined AES processor on FPGA". In: *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 308–309. DOI: 10.1109/FCCM.2004.1.

Holmes, S.A., G. Klein, and D.W. Murray (2009). "An O(N^2) Square Root Unscented Kalman Filter for Visual Simultaneous Localization and Mapping". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31.7, pp. 1251–1263. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2008.189.

Honegger, D., H. Oleynikova, and M. Pollefeys (2014). "Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4930–4935. DOI: 10.1109/IROS.2014.6943263.

Hopson, B., K. Benkrid, D. Keymeulen, and N. Aranki (2012). "Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU amp; multicore processor systems". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pp. 107–114. DOI: 10.1109/AHS.2012.6268637.

Huang, G.P., AI Mourikis, and S.I Roumeliotis (2013). "A Quadratic-Complexity Observability-Constrained Unscented Kalman Filter for SLAM". In: *Robotics, IEEE Transactions on* 29.5, pp. 1226–1243. ISSN: 1552-3098. DOI: 10.1109/TRO.2013.2267991.

Huang, Guoquan P., AI Mourikis, and S.I Roumeliotis (2009). "On the complexity and consistency of UKF-based SLAM". In: *Robotics and Automation, 2009. ICRA*

'09. IEEE International Conference on, pp. 4401–4408. DOI: 10.1109/ROBOT.2009.5152793.

Huber, F., P. Behr, HP Röser, and S. Pletner (2007). "FPGA based on-board computer system for the Flying Laptop micro-satellite". In: *Proceedings of the Data System in Aerospace Conference, SP-638, ESA, Naples.*

Huebner, M., T. Becker, and J. Becker (2004). "Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration". In: *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, pp. 28–32. DOI: 10.1109/SBCCI.2004.240972.

Idkhajine, L., E. Monmasson, and A. Maalouf (2012). "Fully FPGA-Based Sensorless Control for Synchronous AC Drive Using an Extended Kalman Filter". In: *Industrial Electronics, IEEE Transactions on* 59.10, pp. 3908–3918. ISSN: 0278-0046. DOI: 10.1109/TIE.2012.2189533.

Iturbe, X., K. Benkrid, T. Arslan, Chuan Hong, A.T. Erdogan, and I. Martinez (2011). "Enabling FPGAs for future deep space exploration missions: Improving fault-tolerance and computation density with R3TOS". In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pp. 104–112. DOI: 10.1109/AHS.2011.5963923.

Jafarzadeh, S., C. Lascu, and M.S. Fadali (2012). "State Estimation of Induction Motor Drives Using the Unscented Kalman Filter". In: *Industrial Electronics, IEEE Transactions on* 59.11, pp. 4207–4216. ISSN: 0278-0046. DOI: 10.1109/TIE.2011.2174533.

Jones, D.H., A. Powell, C. Bouganis, and P. Y K Cheung (2010). "GPU Versus FPGA for High Productivity Computing". In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 119–124. DOI: 10.1109/FPL.2010.32.

Julier, S.J. (2003). "The spherical simplex unscented transformation". In: *American Control Conference, 2003. Proceedings of the 2003.* Vol. 3, 2430–2434 vol.3. DOI: 10.1109/ACC.2003.1243439.

Julier, S.J. and J.K. Uhlmann (1997). "A new extension of the Kalman filter to nonlinear systems". In: *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls.* Vol. 3, p. 26.

— (2004). "Unscented filtering and nonlinear estimation". In: *Proceedings of the IEEE* 92.3, pp. 401–422.

Jung, S.L., M.Y. Chang, J.Y. Jyang, L.C. Yeh, and Y.Y. Tzou (1999). "Design and implementation of an FPGA-based control IC for AC-voltage regulation". In: *Power Electronics, IEEE Transactions on* 14.3, pp. 522–532.

Kalman, Rudolph Emil (1960). "A new approach to linear filtering and prediction problems". In: *Journal of Basic Engineering* 82.1, pp. 35–45.

Kandepu, Rambabu, Bjarne Foss, and Lars Imsland (2008). "Applying the unscented Kalman filter for nonlinear state estimation". In: *Journal of Process Control* 18.7–8, pp. 753–768. ISSN: 0959-1524. DOI: 10.1016/j.jprocont.2007.11.004. URL: http://www.sciencedirect.com/science/article/pii/S0959152407001655.

Kestur, S., J. D. Davis, and O. Williams (2010). "BLAS Comparison on FPGA, CPU and GPU". In: *2010 IEEE Computer Society Annual Symposium on VLSI*, pp. 288–293. DOI: 10.1109/ISVLSI.2010.84.

Kim, Chanki, R. Sakthivel, and Wan Kyun Chung (2008). "Unscented FastSLAM: A Robust and Efficient Solution to the SLAM Problem". In: *Robotics, IEEE Transactions on* 24.4, pp. 808–820. ISSN: 1552-3098. DOI: 10.1109/TRO.2008.924946.

Kim, Daijin (2000). "An implementation of fuzzy logic controller on the reconfigurable FPGA system". In: *Industrial Electronics, IEEE Transactions on* 47.3, pp. 703–715. ISSN: 0278-0046. DOI: 10.1109/41.847911.

Kish, F., V. Lal, P. Evans, S. W. Corzine, M. Ziari, T. Butrie, M. Reffle, H. S. Tsai, A. Dentai, J. Pleumeekers, M. Missey, M. Fisher, S. Murthy, R. Salvatore, P. Samra, S. Demars, N. Kim, A. James, A. Hosseini, P. Studenkov, M. Lauermann, R. Going, M. Lu, J. Zhang, J. Tang, J. Bostak, T. Vallaitis, M. Kuntz, D. Pavinski, A. Karanicolas, B. Behnia, D. Engel, O. Khayam, N. Modi, M. R. Chitgarha, P. Mertz, W. Ko, R. Maher, J. Osenbach, J. T. Rahn, H. Sun, K. T. Wu, M. Mitchell, and D. Welch (2018). "System-on-Chip Photonic Integrated Circuits". In: *IEEE Journal of Selected Topics in Quantum Electronics* 24.1, pp. 1–20. ISSN: 1077-260X. DOI: 10.1109/JSTQE.2017.2717863.

Kowalczyk, Marcin and Tomasz Kryjak (2017). "Object Tracking With the Use of a Moving Camera Implemented in Heterogeneous Zynq System on Chip". In: *Trends in Advanced Intelligent Control, Optimization and Automation: Proceedings of KKA 2017—The 19th Polish Control Conference, Kraków, Poland, June 18–21, 2017*. Ed. by Wojciech Mitkowski, Janusz Kacprzyk, Krzysztof Oprzędkiewicz, and Paweł Skruch. Cham: Springer International Publishing, pp. 354–363. ISBN: 978-3-319-60699-6. DOI: 10.1007/978-3-319-60699-6_34. URL: https://doi.org/10.1007/978-3-319-60699-6_34.

Krach, F., B. Frackelton, J. Carletta, and R. Veillette (2003). "FPGA-based implementation of digital control for a magnetic bearing". In: *2003 American Control Conference*, pp. 1080–1085.

Kumar, S., A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani (2002). "A network on chip architecture and design methodology". In: *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 105–112. DOI: 10.1109/ISVLSI.2002.1016885.

Kumar, Vinay B.Y., Siddharth Joshi, Sachin B. Patkar, and H. Narayanan (2010). "FPGA Based High Performance Double-Precision Matrix Multiplication". English. In: *International Journal of Parallel Programming* 38.3-4, pp. 322–338. ISSN: 0885-7458. DOI: 10.1007/s10766-010-0131-8. URL: http://dx.doi.org/10.1007/s10766-010-0131-8.

Kuon, I., R. Tessier, and J. Rose (2008). "Fpga architecture: Survey and challenges". In: *Foundations and Trends ® in Electronic Design Automation* 2.2, pp. 135–253.

Kurt-Yavuz, Z. and S. Yavuz (2012). "A comparison of EKF, UKF, FastSLAM2.0, and UKF-based FastSLAM algorithms". In: *Intelligent Engineering Systems (INES),*

2012 IEEE 16th International Conference on, pp. 37–43. DOI: 10.1109/INES.2012.6249866.

Kuwahara, T., F. Böhringer, A. Falke, J. Eickhoff, F. Huber, and H.P. Röser (2009). "FPGA-based operational concept and payload data processing for the Flying Laptop satellite". In: *Acta Astronautica* 65.11, pp. 1616–1627.

Lacey, G., G. W. Taylor, and S. Areibi (2016). "Deep Learning on FPGAs: Past, Present, and Future". In: *ArXiv e-prints*. arXiv: 1602.04283 [cs.DC].

Lambert, C., T. Kalganova, and E. Stomeo (2009). "FPGA-based systems for evolvable hardware". In: *International Journal of Electrical, Computer, and Systems Engineering* 3.1, pp. 62–68.

Ledesma-Carrillo, L.M., E. Cabal-Yepez, R. de J Romero-Troncoso, A. Garcia-Perez, R.A. Osornio-Rios, and T.D. Carozzi (2011). "Reconfigurable FPGA-Based Unit for Singular Value Decomposition of Large m x n Matrices". In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 345–350. DOI: 10.1109/ReConFig.2011.77.

Lehtoranta, I., E. Salminen, A. Kulmala, M. Hannikainen, and T.D. Hamalainen (2005). "A parallel MPEG-4 encoder for FPGA based multiprocessor SoC". In: *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 380–385. DOI: 10.1109/FPL.2005.1515751.

López, B., J. Valverde, E. de la Torre, and T. Riesgo (2014). "Power-aware multi-objective evolvable hardware system on an FPGA". In: *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 61–68. DOI: 10.1109/AHS.2014.6880159.

Maheshwarappa, M. R., M. D. J. Bowyer, and C. P. Bridges (2017). "Improvements in CPU FPGA Performance for Small Satellite SDR Applications". In: *IEEE Transactions on Aerospace and Electronic Systems* 53.1, pp. 310–322. ISSN: 0018-9251. DOI: 10.1109/TAES.2017.2650320.

Al-Mahmood, Ali and Michael Opoku Agyeman (2017). "A Study of FPGA-based System-on-Chip Designs for Real-Time Industrial Application". In: *International Journal of Computer Applications* 163.6, pp. 9–19.

Michell, G. De and R. K. Gupta (1997). "Hardware/software co-design". In: *Proceedings of the IEEE* 85.3, pp. 349–365. ISSN: 0018-9219. DOI: 10.1109/5.558708.

Monmasson, E. and M.N. Cirstea (2007). "FPGA Design Methodology for Industrial Control Systems – A Review". In: *Industrial Electronics, IEEE Transactions on* 54.4, pp. 1824–1842. ISSN: 0278-0046. DOI: 10.1109/TIE.2007.898281.

Nørgaard, Magnus, Niels K. Poulsen, and Ole Ravn (2000). "New developments in state estimation for nonlinear systems". In: *Automatica* 36.11, pp. 1627–1638. ISSN: 0005-1098. DOI: http://dx.doi.org/10.1016/S0005-1098(00)00089-3. URL: http://www.sciencedirect.com/science/article/pii/S0005109800000893.

Oetken, A., S. Wildermann, J. Teich, and D. Koch (2010). "A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs". In: *Field*

Programmable Logic and Applications (FPL), 2010 International Conference on, pp. 234–239. DOI: 10.1109/FPL.2010.54.

Pang, L., M. Zhao, and Y. d. Luo (2014). "A high performance system-on-chip architecture for digital wideband radar receiver". In: *2014 12th International Conference on Signal Processing (ICSP)*, pp. 2106–2109. DOI: 10.1109/ICOSP.2014.7015366.

Patwardhan, Sachin C., Shankar Narasimhan, Prakash Jagadeesan, Bhushan Gopaluni, and Sirish L. Shah (2012). "Nonlinear Bayesian state estimation: A review of recent developments". In: *Control Engineering Practice* 20.10. 4th Symposium on Advanced Control of Industrial Processes (ADCONIP), pp. 933–953. ISSN: 0967-0661. DOI: http://dx.doi.org/10.1016/j.conengprac.2012.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0967066112000871.

Perea, Laura, Jonathan How, Louis Breger, and Pedro Elosegui (2007). "Nonlinearity in sensor fusion: divergence issues in EKF, modified truncated GSF, and UKF". In: *AIAA Guidance, Navigation and Control Conference and Exhibit*, p. 6514.

St-Pierre, M. and D. Gingras (2004). "Comparison between the unscented Kalman filter and the extended Kalman filter for the position estimation module of an integrated navigation information system". In: *IEEE Intelligent Vehicles Symposium, 2004*, pp. 831–835. DOI: 10.1109/IVS.2004.1336492.

Prasad, E. L., A. R. Reddy, and M. N. G. Prasad (2016). "Performance comparison of Network on Chip methods". In: *2016 Online International Conference on Green Engineering and Technologies (IC-GET)*, pp. 1–8. DOI: 10.1109/GET.2016.7916861.

Qiu, Jiantao, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang (2016). "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: ACM, pp. 26–35. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847265. URL: http://doi.acm.org/10.1145/2847263.2847265.

Quinchia, A.G. and C. Ferrer (2011). "A low-cost GPS&INS integrated system based on a FPGA platform". In: *Localization and GNSS (ICL-GNSS), 2011 International Conference on*, pp. 152–157. DOI: 10.1109/ICL-GNSS.2011.5955277.

Ramchandani, Varun, Kranthi Pamarthi, and Shubhajit Roy Chowdhury (2012). "Comparative Study of Maximum Power Point Tracking Using Linear Kalman Filter & Unscented Kalman Filter for Solar Photovoltaic Array on Field Programmable Gate Array". In: *International Journal on Smart Sensing & Intelligent Systems* 5.3.

Rhudy, Matthew, Yu Gu, Jason Gross, and Marcello R Napolitano (2012). "Evaluation of matrix square root operations for UKF within a UAV GPS/INS sensor fusion application". In: *International Journal of Navigation and Observation* 2011.

Rossi, D.L., V. Bonato, E. Marques, and J.M. Gago Pontes de Brito Lima (2011). "A PID Controller Applied to the Gain Control of a CMOS Camera Using Reconfig-

urable Computing". In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 141–145. DOI: 10.1109/ReConFig.2011.2.

Saggese, G.P., A. Mazzeo, N. Mazzocca, and A.G.M. Strollo (2003). "An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm". In: *Field Programmable Logic and Application*. Ed. by Peter Cheung and GeorgeA. Constantinides. Vol. 2778. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 292–302. ISBN: 978-3-540-40822-2. DOI: 10.1007/978-3-540-45234-8_29. URL: http://dx.doi.org/10.1007/978-3-540-45234-8_29.

Salvador, R., A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina (2013). "Self-Reconfigurable Evolvable Hardware System for Adaptive Image Processing". In: *IEEE Transactions on Computers* 62.8, pp. 1481–1493. ISSN: 0018-9340. DOI: 10.1109/TC.2013.78.

Schaeferling, M. and G. Kiefer (2011). "Object Recognition on a Chip: A Complete SURF-Based System on a Single FPGA". In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 49–54. DOI: 10.1109/ReConFig.2011.65.

Selva, Daniel and David Krejci (2012). "A survey and assessment of the capabilities of Cubesats for Earth observation". In: *Acta Astronautica* 74, pp. 50–68. ISSN: 0094-5765. DOI: http://dx.doi.org/10.1016/j.actaastro.2011.12.014. URL: http://www.sciencedirect.com/science/article/pii/S0094576511003742.

Sharma, M. and D. Kumar (2012). "Wishbone bus Architecture – A Survey and Comparison". In: *arXiv preprint arXiv:1205.1860*.

Sharma, S., S. Kulkarn, V. Pujari, M. Vanitha, and P. Lakshminarsimhan (2010). "FPGA Implementation of M-PSK Modulators for Satellite Communication". In: *Advances in Recent Technologies in Communication and Computing (ARTCom), 2010 International Conference on*. IEEE, pp. 136–139.

Simon, Dan (2006). *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons.

Singh, Amit Kumar, Muhammad Shafique, Akash Kumar, and Jörg Henkel (2013). "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: ACM, 1:1–1:10. ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488734. URL: http://doi.acm.org/10.1145/2463209.2488734.

Siozios, K. and D. Soudris (2012). "A low-cost fault tolerant solution targeting to commercial FPGA devices". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pp. 46–53. DOI: 10.1109/AHS.2012.6268668.

Sitz, A., U. Schwarz, J. Kurths, and H. U. Voss (2002). "Estimation of parameters and unobserved components for nonlinear systems from noisy time series". In: *Phys. Rev. E* 66 (1), p. 016210. DOI: 10.1103/PhysRevE.66.016210. URL: http://link.aps.org/doi/10.1103/PhysRevE.66.016210.

Soh, J. and X. Wu (2012). "A FPGA-based approach to attitude determination for nanosatellites". In: *Industrial Electronics and Applications (ICIEA), 2012 7th IEEE Conference on*, pp. 1700–1704. DOI: 10.1109/ICIEA.2012.6360999.

— (2014). "A Modular FPGA-based Implementation of the Unscented Kalman Filter". In: *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pp. 127–134. DOI: 10.1109/AHS.2014.6880168.

— (2017a). "A Five-Stage Pipeline Architecture of the Unscented Kalman Filter for System-on-Chip Applications". In: *IEEE Transactions on Industrial Electronics* PP.99, pp. 1–1. ISSN: 0278-0046. DOI: 10.1109/TIE.2017.2740844.

— (2017b). "An FPGA-Based Unscented Kalman Filter for System-On-Chip Applications". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.4, pp. 447–451. ISSN: 1549-7747. DOI: 10.1109/TCSII.2016.2565730.

Stratikopoulos, A., G. Chrysos, I. Papaefstathiou, and A. Dollas (2014). "HPC-gSpan: An FPGA-based parallel system for frequent subgraph mining". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4. DOI: 10.1109/FPL.2014.6927441.

Teich, J. (2012). "Hardware/Software Codesign: The Past, the Present, and Predicting the Future". In: *Proceedings of the IEEE* 100.Special Centennial Issue, pp. 1411–1430. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2182009.

Tertei, D. T., J. Piat, and M. Devy (2014). "FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM". In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032523.

Todman, T.J., G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung (2005). "Reconfigurable computing: architectures and design methods". In: *Computers and Digital Techniques, IEE Proceedings-* 152.2, pp. 193–207.

Tumeo, A., M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto (2007). "A Pipelined Fast 2D-DCT Accelerator for FPGA-based SoCs". In: *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pp. 331–336. DOI: 10.1109/ISVLSI.2007.13.

Tuna, G., K. Gulez, V.C. Gungor, and T. Veli Mumcu (2012). "Evaluations of different Simultaneous Localization and Mapping (SLAM) algorithms". In: *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pp. 2693–2698. DOI: 10.1109/IECON.2012.6389151.

Van Dyke, Matthew C., Jana L. Schwartz, and Christopher D. Hall (2004). "Unscented Kalman filtering for spacecraft attitude state and parameter estimation". In: *Advances in the Astronautical Sciences* 118.1, pp. 217–228.

Visser, S.J., A.S. Dawood, and J.A. Williams (2002). "FPGA based real-time adaptive filtering for space applications". In: *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pp. 322–326. DOI: 10.1109/FPT.2002.1188702.

Vladimirova, Tanya and Xiaofeng Wu (2007). "A Reconfigurable System-on-Chip Architecture for Pico-Satellite Missions". In: *CPA*. Ed. by Alistair A. McEwan,

Steve A. Schneider, Wilson Ifill, and Peter H. Welch. Vol. 65. Concurrent Systems Engineering Series. IOS Press, pp. 493–502. ISBN: 978-1-58603-767-3.

Wan, E.A. and R. Van Der Merwe (2000). "The unscented Kalman filter for nonlinear estimation". In: *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*. IEEE, pp. 153–158.

Wang, Hongjian, Guixia Fu, Juan Li, Zheping Yan, and Xinqian Bian (2013). "An adaptive UKF based SLAM method for unmanned underwater vehicle". In: *Mathematical Problems in Engineering* 2013.

Wang, J., S. Zhong, L. Yan, and Z. Cao (2014). "An Embedded System-on-Chip Architecture for Real-time Visual Detection and Matching". In: *IEEE Transactions on Circuits and Systems for Video Technology* 24.3, pp. 525–538. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2013.2280040.

Wang, Y., Q. Liu, and A. E. Fathy (2013). "CW and Pulse – Doppler Radar Processing Based on FPGA for Human Sensing Applications". In: *IEEE Transactions on Geoscience and Remote Sensing* 51.5, pp. 3097–3107. ISSN: 0196-2892. DOI: 10.1109/TGRS.2012.2217975.

Wang, Yue, K. Cunningham, P. Nagvajara, and J. Johnson (2010). "Singular Value Decomposition Hardware for MIMO: State of the Art and Custom Design". In: *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pp. 400–405. DOI: 10.1109/ReConFig.2010.62.

Wiklund, D. and Dake Liu (2003). "SoCBUS: switched network on chip for hard real time embedded systems". In: *Proceedings International Parallel and Distributed Processing Symposium*. DOI: 10.1109/IPDPS.2003.1213180.

Williams, J.A., A.S. Dawood, and S.J. Visser (2002). "FPGA-based cloud detection for real-time onboard remote sensing". In: *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pp. 110–116. DOI: 10.1109/FPT.2002.1188671.

Wittig, R.D. and P. Chow (1996). "OneChip: an FPGA processor with reconfigurable logic". In: *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pp. 126–135. DOI: 10.1109/FPGA.1996.564773.

Wolf, W. (2003). "A decade of hardware/software codesign". In: *Computer* 36.4, pp. 38–43. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1193227.

Wu, Haoyang, Tao Wang, Zhiwei Li, Boyan Ding, Xiaoguang Li, Tianfu Jiang, Jun Liu, and Songwu Lu (2017). "GRT 2.0: An FPGA-based SDR Platform for Cognitive Radio Networks". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, pp. 294–295. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021798. URL: http://doi.acm.org/10.1145/3020078.3021798.

Xilinx (2010). *Virtex-4 Family Overview*. DS112 (v3.1).

— (2016). *Zynq-7000 All Programmable SoC Technical Reference Manual*. UG585 (v1.11).

— (2017a). *Vivado Design Suite User Guide: Getting Started*. UG910 (v2017.2).

— (2017b). *Vivado Design Suite User Guide: Logic Simulation*. UG900 (v2017.2).

Xilinx (2017c). *Vivado Design Suite User Guide: Power Analysis and Optimisation.* UG907 (v2017.2).

Xiong, K., C. W. Chan, and H.Y. Zhang (2007). "Detection of satellite attitude sensor faults using the UKF". In: *Aerospace and Electronic Systems, IEEE Transactions on* 43.2, pp. 480–491. ISSN: 0018-9251. DOI: 10.1109/TAES.2007.4285348.

Yang, Depeng, G.D. Peterson, Husheng Li, and Junqing Sun (2009). "An FPGA Implementation for Solving Least Square Problem". In: *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pp. 303–306. DOI: 10.1109/FCCM.2009.47.

Yang, G., L. Xie, M. Mäntysalo, X. Zhou, Z. Pang, L. D. Xu, S. Kao-Walter, Q. Chen, and L. R. Zheng (2014). "A Health-IoT Platform Based on the Integration of Intelligent Packaging, Unobtrusive Bio-Sensor, and Intelligent Medicine Box". In: *IEEE Transactions on Industrial Informatics* 10.4, pp. 2180–2191. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2307795.

Yang, Shuangming, Bin Deng, Jiang Wang, Huiyan Li, Chen Liu, Chris Fietkiewicz, and Kenneth A Loparo (2017). "Efficient implementation of a real-time estimation system for thalamocortical hidden Parkinsonian properties". In: *Scientific reports* 7, p. 40152.

Ye, Zhuan, J. Grosspietsch, and G. Memik (2007). "An FPGA Based All-Digital Transmitter with Radio Frequency Output for Software Defined Radio". In: *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pp. 1–6. DOI: 10.1109/DATE.2007.364561.

Zheng, D., T. Vladimirova, H. Tiggeler, and M. Sweeting (2001). "Reconfigurable Single-Chip On-Board Computer for a Small Satellite". In: *52nd International Astronautical Congress, Toulouse, France.*

Zhigang, Luo, Li Wei, Zhang Yan, and Guan Wei (2003). "A multi-standard SDR base band platform". In: *Computer Networks and Mobile Computing, 2003. ICCNMC 2003. 2003 International Conference on*, pp. 461–464. DOI: 10.1109/ICCNMC.2003.1243091.

Zhong, G., S. Niar, A. Prakash, and T. Mitra (2016). "Design of Multiple-Target Tracking System on Heterogeneous System-on-Chip Devices". In: *IEEE Transactions on Vehicular Technology* 65.6, pp. 4802–4812. ISSN: 0018-9545. DOI: 10.1109/TVT.2016.2546957.

Zhou, Junchuan, Yuhong Yang, Jieying Zhang, Ezzaldeen Edwan, Otmar Loffeld, and Stefan Knedlik (2011). "Tightly-coupled INS/GPS using Quaternion-based Unscented Kalman filter". In: *AIAA Guidance, Navigation and Control Conference.*

Zhu, Jihan and Peter Sutton (2003). "FPGA Implementations of Neural Networks – A Survey of a Decade of Progress". In: *Field Programmable Logic and Application.* Ed. by Peter Cheung and George A. Constantinides. Vol. 2778. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1062–1066. ISBN: 978-3-540-40822-2. DOI: 10.1007/978-3-540-45234-8_120. URL: http://dx.doi.org/10.1007/978-3-540-45234-8_120.

刘仙, 朱波, 刘会军, and 高庆 (2014). *FPGA (field programmable gate array)-based UKF (unscented Kalman filter) algorithm and filtering on brain dynamics model by FPGA-based UKF algorithm*. CN Patent CN104143017 A. URL: http://www.google.com/patents/CN104143017A?cl=en.

姬红兵, 李倩, 王玮, and 闫家铭 (2013). *FPGA (Field Programmable Gata Array)-based unscented kalman filter system and parallel implementation method*. CN Patent CN101777887 B. URL: http://www.google.com.na/patents/CN101777887B?cl=en.