FPGA IMPLEMENTATION OF METAHEURISTIC OPTIMIZATION ALGORITHM

PHUAH SOON EU

B.ENG (HONS.) ELECTRICAL ENGINEERING (ELECTRONICS)

UNIVERSITI MALAYSIA PAHANG

UNIVERSITI MALAYSIA PAHANG

DECLARATION OF THESIS AND COPYRIGHT

Author's Full Name : PHUAH SOON EU

Date of Birth : 05/05/1998

Title : FPGA IMPLEMENTATION OF METAHEURISTIC OPTIMIZATION ALGORITHM

Academic Session : SEMESTER II 2021/2022

I declare that this thesis is classified as:

☐	CONFIDENTIAL	(Contains confidential information under the Official Secret Act 1997)*
☐	RESTRICTED	(Contains restricted information as specified by the organization where research was done)*
☑	OPEN ACCESS	I agree that my thesis to be published as online open access (Full Text)

I acknowledge that Universiti Malaysia Pahang reserves the following rights:

1. The Thesis is the Property of Universiti Malaysia Pahang
2. The Library of Universiti Malaysia Pahang has the right to make copies of the thesis for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

Certified by:

(Student's Signature)

980505-08-5481
New IC/Passport Number
Date: 21/6/2022

(Supervisor's Signature)

Nurul Hazlina Noordin
Name of Supervisor
Date: 24 June 2022

THESIS DECLARATION LETTER (OPTIONAL)

Librarian,
Perpustakaan Universiti Malaysia Pahang,
Universiti Malaysia Pahang,
Lebuhraya Tun Razak,
26300, Gambang, Kuantan.

Dear Sir,

CLASSIFICATION OF THESIS AS RESTRICTED

Please be informed that the following thesis is classified as RESTRICTED for a period of three (3) years from the date of this letter. The reasons for this classification are as listed below.

Author's Name
Thesis Title

Reasons (i)

(ii)

(iii)

Thank you.

Yours faithfully,

____
(Supervisor's Signature)

Date:

Stamp:

Note: This letter should be written by the supervisor, addressed to the Librarian, *Perpustakaan Universiti Malaysia Pahang* with its copy attached to the thesis.

SUPERVISOR’S DECLARATION

I hereby declare that I have checked this thesis and in my opinion, this thesis is adequate in terms of scope and quality for the award of the degree of the Bachelor of Electrical Engineering (Electronics) with Honours

(Supervisor’s Signature)

Full Name : Nurul Hazlina Noordin
Position : Assosiate Professor
Date : 24 June 2022

(Co-supervisor’s Signature)

Full Name :
Position :
Date :

STUDENT’S DECLARATION

I hereby declare that the work in this thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at Universiti Malaysia Pahang or any other institutions.

(Student’s Signature)

Full Name : PHUAH SOON EU
ID Number : EA18096
Date : 21/6/2022

FPGA IMPLEMENTATION OF METAHEURISTIC OPTIMIZATION ALGORITHM

PHUAH SOON EU

Thesis submitted in fulfillment of the requirements
for the award of the
Bachelor of Electrical Engineering (Electronics) with Honours

College of Engineering
UNIVERSITI MALAYSIA PAHANG

JUNE 2022

ACKNOWLEDGEMENTS

I would like to use this opportunity to express with great pleasure to thank all those who have helped to complete this project. I am thankful for the invaluable advice and guidance from Ir. Dr. Nurul Hazlina binti Noordin during the implementation of this project and also providing spiritual and moral support during the research into this project. Moreover, I also like to thank everyone directly and indirectly for their support, help and ideas in completing this project progress report. I sincerely thank all who share their knowledge, idea, and experience on helping me to complete the project on time Lastly, I would like to thank my family for being considerate and supportive towards me to allow me to successfully pursue and complete the project.

ABSTRAK

Algoritma metaheuristik semakin popular di kalangan penyelidik kerana keupayaannya untuk menyelesaikan masalah pengoptimuman bukan linear serta keupayaan untuk disesuaikan untuk menyelesaikan pelbagai masalah.Terdapat lonjakan metaeuristik novel yang dicadangkan baru-baru ini, namun tidak pasti sama ada ia sesuai untuk pelaksanaan FPGA. Di samping itu, terdapat pelbagai metodologi reka bentuk perlaksanaan metaheuristik FPGA yang boleh meningkatkan keupayaannya. Projek ini dimulakan dengan meneliti dan mengenal pasti metaheuristik yang sesuai untuk pelaksanaan FPGA. Metaeuristik yang dipilih ialah Simulated Kalman Filter (SKF) yang mencadangkan algoritma yang rendah kerumitan dan menggunakan bilang langkah yang kecil. Kemudian SKF Diskret telah disesuaikan daripada metaheuristik asal dengan membundarkan semua nilai titik terapung kepada nombor bulat serta menetapkan keuntungan Kalman tetap 0.5. SKF Diskret kemudiannya dimodelkan menggunakan pemodelan tingkah laku untuk menghasilkan SKF Binari yang kemudiannya dilaksanakan pada FPGA. Reka bentuk telah dibuat secara modular dengan menghasilkan modul berasingan yang menguruskan bahagian metaheuristik yang berbeza dan juga melaksanakan konfigurasi port Selari-Dalam-Selari-Keluar yang meningkatkan keupayaannya. SKF Diskret kemudiannya disimulasikan pada MATLAB manakala SKF Binari dilaksanakan pada FPGA dan keupayaannya diukur berdasarkan penggunaan cip, kelajuan pemprosesan dan ketepatan keputusan. SKF Binari menghasilkan peningkatan kelajuan sehingga 69 kali lebih pantas berbanding dengan simulasi SKF Diskret.

ABSTRACT

Metaheuristic algorithms are gaining popularity amongst researchers due to their ability to solve nonlinear optimization problems as well as the ability to be adapted to solve a variety of problems. There is a surge of novel metaheuristics proposed recently, however it is uncertain whether they are suitable for FPGA implementation. In addition, there exists a variety of design methodologies to implement metaheuristics upon FPGA which may improve the performance of the implementation. The project begins by researching and identifying metaheuristics which are suitable for FPGA implementation. The selected metaheuristic was the Simulated Kalman Filter (SKF) which proposed an algorithm that was low in complexity and used a small number of steps. Then the Discrete SKF was adapted from the original metaheuristic by rounding all floating-point values to integers as well as setting a fixed Kalman gain of 0.5. The Discrete SKF was then modelled using behavioural modelling to produce the Binary SKF which was then implemented onto FPGA. The design was made modular by producing separate modules that managed different parts of the metaheuristic and also implemented Parallel-In-Parallel-Out configuration of ports. The Discrete SKF was then simulated on MATLAB meanwhile the Binary SKF was implemented onto FPGA and their performance were measured based on chip utilization, processing speed, and accuracy of results. The Binary SKF produced speed increment of up to 69 times faster than the Discrete SKF simulation.

TABLE OF CONTENT

DECLARATION

TITLE PAGE

ACKNOWLEDGEMENTS ii

ABSTRAK iii

ABSTRACT iv

TABLE OF CONTENT v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

CHAPTER 1 INTRODUCTION 1

1.1 Project Background 1

1.2 Problem Statement 1

1.3 Objectives 2

1.4 Scope of Project 2

1.5 Thesis Outline 3

CHAPTER 2 LITERATURE REVIEW 4

2.1 Introduction 4

2.2 Novel Optimization Algorithms 4

- 2.2.1 Single-Agent Finite Impulse Response Optimizer 4
- 2.2.2 Barnacles Mating Optimization 5
- 2.2.3 Orca Predation Algorithm 5

2.2.4 Simulated Kalman Filter 6

2.2.5 Single-solution Simulated Kalman Filter 7

2.2.6 Particle Swarm Optimization 7

2.2.7 Variants of Particle Swarm Optimization 8

2.2.8 Binary Ant Colony Optimization 9

2.2.9 Hybrid Binary Bat Enhanced Particle Swarm Optimization Algorithm 9

2.3 FPGA Implementation Optimization Techniques 10

2.3.1 Parallel Implementation of Particle Swarm Optimization 10

2.3.2 Pipeline Architecture of Particle Swarm Optimization 11

2.3.3 FPGA Realization of Particle Swarm Optimization Algorithm using Floating Point Arithmetic 12

CHAPTER 3 METHODOLOGY 13

3.1 Introduction 13

3.2 Simulated Kalman Filter 13

3.3 Discrete Simulated Kalman Filter Adaptation 17

3.4 Binary Simulated Kalman Filter Behavioural Modelling 19

3.4.1 Parallel-In-Parallel-Out Configuration of Modules 19

3.4.2 Pseudo Random Number Generator (RNG) 20

3.4.3 Random-Access Memory (RAM) 21

3.4.4 Activation Function 22

3.4.5 2-bit Pseudo Random Number Generator 22

3.4.6 Measure 23

3.4.7 Estimate 24

3.4.8 Finite State Machine Controller 25

3.4.9 Testbench Verification 29

3.5 Hardware Components 30

3.6 Software Components 31

3.7 Performance Parameters 31

3.7.1 Chip Utilization 32

3.7.2 Processing Speed 32

3.7.3 Accuracy of Result 32

CHAPTER 4 RESULTS & DISCUSSION 33

4.1 Introduction 33

4.2 Experimental Setup 33

4.3 Simulation and Implementation Results 33

4.3.1 MATLAB Simulation of Discrete Simulated Kalman Filter 34

4.3.2 FPGA Implementation of Binary Simulated Kalman Filter 35

4.4 Chip Utilization 37

4.5 Processing Speed 38

4.6 Accuracy of Result 39

4.7 Approximating the Time Taken by FPGA to Complete Run 40

CHAPTER 5 CONCLUSION 41

5.1 Discrete Simulated Kalman Filter 41

5.2 Implementation of Binary Simulated Kalman Filter 41

5.3 Project Limitations 42

5.4 Suggestions and Future Work 42

5.4.1 Improving Accuracy of Results 42

5.4.2 Implementation of Pipeline Structure 42

5.4.3 Implementation of CEC2014 Benchmark Functions 43

5.4.4 Physical Synthesis to Improve Timing and Accuracy of Results 43

5.4.5 Utilizing Phase-Locked Loop to Increase Clock Frequency 43

REFERENCES 44

LIST OF TABLES

Table 1 Compiled result of simulation and FPGA implementation 33

Table 2 Comparison of chip utilization 37

Table 3 Comparison of processing speed 38

Table 4 Comparison of accuracy of result 39

Table 5 Approximating time taken by FPGA to complete run 40

LIST OF FIGURES

Figure 1 Flowchart for Single-agent Finite Impulse Response Optimizer 4
Figure 2 Flowchart for Barnacles Mating Optimizer 5
Figure 3 Flowchart for Orca Predation Algorithm 6
Figure 4 Flowchart for Simulated Kalman Filter 6
Figure 5 Flowchart for Single-solution Simulated Kalman Filter 7
Figure 6 Flowchart for Particle Swarm Optimization 8
Figure 7 Parallel Implementation of Particle Modules 10
Figure 8 Data path between registers 11
Figure 9 Pipeline structure 12
Figure 10 Simulated Kalman Filter flowchart 14
Figure 11 Generation of population and fitness evaluation of agent for SKF 15
Figure 12 Generation of population and fitness evaluation of agent for Discrete SKF 17
Figure 13 Progression of Kalman gain in original SKF 18
Figure 14 2s complement binary representation of Binary SKF 18
Figure 15 Stacked random number generator modules 20
Figure 16 Random-Access Memory module 21
Figure 17 Activation Function module 22
Figure 18 2-bit Pseudo Random Number Generator 22
Figure 19 Measure module 23
Figure 20 Estimate module 24
Figure 21 Simplified state diagram of finite state machine controller 25
Figure 22 State s0 Reset 26
Figure 23 State s1 Generate population 26
Figure 24 State s2 Fitness evaluation and comparison 27
Figure 25 State s3 Measure 27
Figure 26 State s4 Estimate 28
Figure 27 Waveform generated by a testbench with parameters N = 50, D = 10, maxrun = 2, t_{max} = 50. 29
Figure 28 DE10 – Lite System Builder application 30
Figure 29 Quartus Prime Lite Edition 18.1 31
Figure 30 Performance parameters 32
Figure 31 MATLAB simulation result for D = 5 34
Figure 32 MATLAB simulation result for D = 10 34

Figure 33	MATLAB simulation result for D = 20	34
Figure 34	FPGA implementation result for D = 5	35
Figure 35	FPGA implementation result for D = 10	35
Figure 36	FPGA implementation result for D = 20	35
Figure 37	FPGA implementation chip utilization for D = 5	36
Figure 38	FPGA implementation chip utilization for D = 10	36
Figure 39	FPGA implementation chip utilization for D = 20	36

LIST OF ABBREVIATIONS

FPGA	Field Programmable Gate Array
SKF	Simulated Kalman Filter
FSM	Finite State Machine
RNG	Random Number Generator
RAM	Random-Access Memory
PIPO	Parallel-In-Parallel-Out
SISO	Serial-In-Serial-Out
CPU	Central Processing Unit
LFSR	Linear Feedback Shift Register
PLL	Phase-Locked Loop

CHAPTER 1

INTRODUCTION

1.1 Project Background

A metaheuristic algorithm is an optimization method that is used to solve complex nonlinear and multimodal problems [1] .In addition, metaheuristic algorithms are flexible due to their ability to be adapted to solve a wide range of optimization problems. In recent times, some metaheuristics have risen to popularity due to their flexibility and capability in solving a wide scale and variety of optimization problems. A few examples of these metaheuristics are such as Genetic Algorithm, Grey Wolf Optimizer, Particle Swarm Optimization, and Simulated Kalman Filter.

Implementation of a metaheuristic onto a field programmable gate array (FPGA) is a straightforward process. The metaheuristic equations are implemented according to the flowchart and with the assistance of the pseudocode into a description hardware language such as System Verilog which will then be programmed onto an FPGA. However, the design methodology of the implementation can significantly affect the performance of the metaheuristics on the hardware.

1.2 Problem Statement

Lately, a large influx of novel metaheuristic algorithms has been proposed such as Barnacles Mating Optimizer, Orca Predation Algorithm, and Single-Agent Finite Impulse Response Optimizer. Furthermore, researchers have also worked on existing metaheuristics to produce variants of the original work such as Transitional Particle Swarm Optimization, Binary Particle Swarm Optimization, and Single-solution Simulated Kalman Filter.

The increasing interest of developing new or variants of metaheuristic algorithms is because each metaheuristic can achieve a different performance in terms of their exploration and exploitation process [2]. This would lead to varying performance difference in solving different kind of optimization problems. However, these novel metaheuristics are typically designed with numerous and complex steps that may be difficult to implement on a FPGA.

In recent times, edge computing has been a growing trend as it contends against its alternative cloud computing. Instead of transmitting raw data from the source to a central data centre for analysis and processing, some of the work is shifted to the source itself. However, this method of computing requires reliable yet cost effective hardware to be equipped at the front end.

1.3 Objectives

The objectives of this project are listed below:

i. To study, explore and modify Simulated Kalman Filter algorithm for hardware implementation

ii. To investigate, design & implement binary Simulated Kalman Filter

iii. To evaluate the performance of the structure in terms of accuracy, speed, and cost

1.4 Scope of Project

This project will only consider metaheuristic algorithms for numerical optimization. The hardware used for the implementation will be the DE10 - Lite board (10M50DAF474G) and the performance of the project will be dependent on its capabilities. In addition, the performance of the implementation will be measured using one activation function only.

1.5 Thesis Outline

This thesis is composed of five main chapters which are the introduction, literature review, methodology, results and discussion, and conclusion.

Chapter 1 introduces the background of project, the problem statement, objectives, and scope of the project. This chapter is dedicated to enable the reader to grasp the essential topics of the project such as the definition of metaheuristic and its utilizations. It also elaborates the aim and limitations of the project so that readers are more aware of what the project tries to achieve within the given scope.

Chapter 2 is the literature review where articles written by experts of the field of metaheuristics and the implementation of metaheuristics into FPGA are reviewed. In the literature review, the thesis looks into a variety of novel optimization algorithms that have been recently introduced. Then a list of design methodologies of FPGA implementation is also reviewed to understand the effects of different design methodologies can have upon the performance of the metaheuristic.

Chapter 3 describes the adaptation of the original Simulated Kalman Filter into the Discrete Simulated Kalman Filter that is more suitable for implementation into digital systems. Then the Discrete Simulated Kalman Filter is then modelled using System Verilog through behavioural modelling to generate the necessary modules to operate the Binary Simulated Kalman Filter. The design of the finite state machine controller is also described in detail in this chapter. Next, the software and hardware components are briefly described. Lastly, the performance parameters which will be used to verify the performance of the implementation is discussed.

Chapter 4 presents the results of the FPGA implementation of the Binary Simulated Kalman Filter and the MATLAB simulation of the Discrete Simulated Kalman Filter. The chapter then continues to analyse the performance of both methods in terms of the performance parameters.

Chapter 5 concludes the project by summing up results of the project. Then the limitations of the project were identified. This is followed by the suggestions and future work that should be implemented to further improve the performance of the implementation.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

The first objective of the project is to study different metaheuristic algorithms and to identify suitable metaheuristics for hardware implementation. The project begins by studying a variety of novel optimization algorithms.

2.2 Novel Optimization Algorithms

2.2.1 Single-Agent Finite Impulse Response Optimizer

The metaheuristic proposed is a single-agent metaheuristics that is inspired by the unbiased finite impulse response filter. It proposes an algorithm that optimizes a single solution iteratively until a stopping condition is met. It boasts a great performance in exploration and exploitation which enables it to search a wide range of possible solutions and lastly produce a near optimum solution [1]. Its process is described in the flowchart as shown in Figure 1.

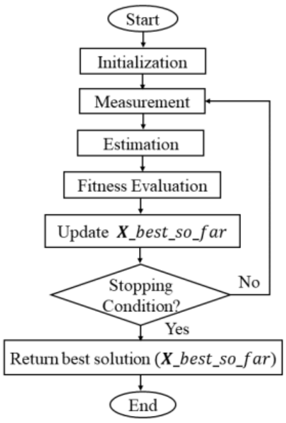


Figure 1 Flowchart for Single-agent Finite Impulse Response Optimizer

2.2.2 Barnacles Mating Optimization

The proposed algorithm is a novel multi-agent optimization algorithm which mimics the mating behaviours of barnacles as described in Figure 2. It involves a sequence where barnacles are randomly selected, and the reproduction process occurs to a set population of barnacles. Then, the barnacles may only mate with the surrounding barnacles based on the length of their penis which is set prior to simulation. The offspring of the barnacles will then inherit the characteristics from its parents [2].

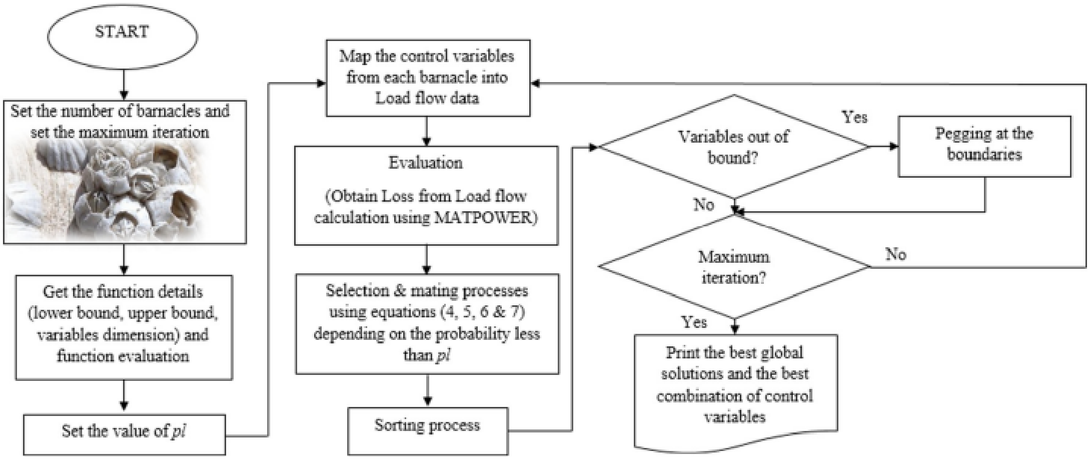

START
Set the number of barnacles and set the maximum iteration
Get the function details (lower bound, upper bound, variables dimension) and function evaluation
Set the value of pl
Map the control variables from each barnacle into Load flow data
Evaluation
(Obtain Loss from Load flow calculation using MATPOWER)
Selection & mating processes using equations (4, 5, 6 & 7) depending on the probability less than pl
Sorting process
Variables out of bound?
Yes
Pegging at the boundaries
No
Maximum iteration?
No
Yes
Print the best global solutions and the best combination of control variables

Figure 2 Flowchart for Barnacles Mating Optimizer

2.2.3 Orca Predation Algorithm

A novel multi-agent bio-inspired metaheuristic which mimics the hunting behaviour of orcas. The metaheuristic introduces a sequence where orcas drive, encircle, and attack a school of fish. The algorithm emphasises on different stages of the sequence such as driving and encircling to effectively adjust its exploration and exploitation respectively. This enables the algorithm to solve a large variety of problems as it was implemented onto several engineering optimization problems which showed great performance [3]. The metaheuristic is described using a complex flowchart as seen in Figure 3.

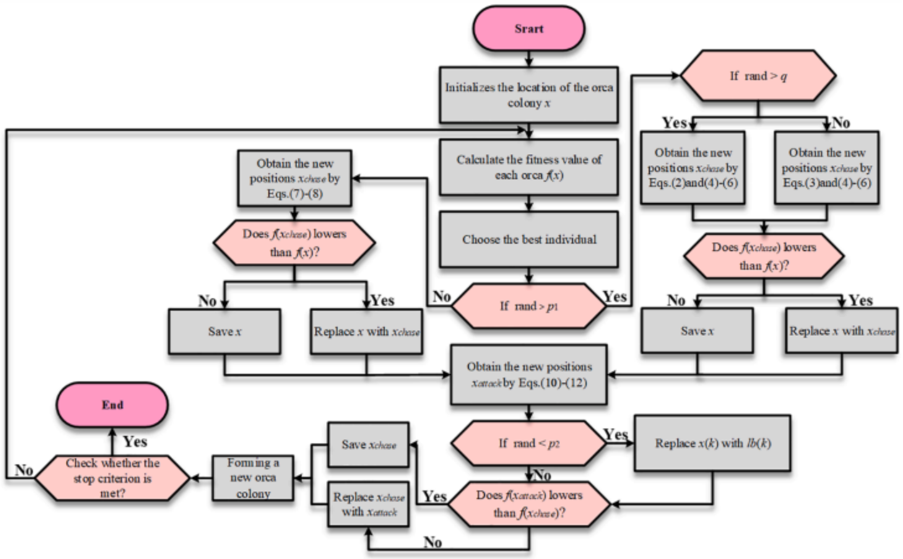

Srart
Initializes the location of the orca colony x
Calculate the fitness value of each orca $f(x)$
Choose the best individual
If rand > p_1
No
Yes
Obtain the new positions x_{chase} by Eqs.(7)-(8)
Does $f(x_{chase})$ lowers than $f(x)$?
No
Yes
Save x
Replace x with x_{chase}
If rand > q
Yes
No
Obtain the new positions x_{chase} by Eqs.(2)and(4)-(6)
Obtain the new positions x_{chase} by Eqs.(3)and(4)-(6)
Does $f(x_{chase})$ lowers than $f(x)$?
No
Yes
Save x
Replace x with x_{chase}
Obtain the new positions x_{attack} by Eqs.(10)-(12)
If rand < p_2
Yes
Replace $x(k)$ with $lb(k)$
No
Does $f(x_{attack})$ lowers than $f(x_{chase})$?
Yes
No
Save x_{chase}
Replace x_{chase} with x_{attack}
Forming a new orca colony
Check whether the stop criterion is met?
No
Yes
End

Figure 3 Flowchart for Orca Predation Algorithm

2.2.4 Simulated Kalman Filter

A multi-agent metaheuristic where each search agent acts as a Kalman Filter which is a state estimation method popularized in the year 1960. Each search agent then estimates the optimum solution to the fitness function through several steps such as predict, measure, and estimate to consequently produce the best-so-far solution as shown in Figure 4 [4].

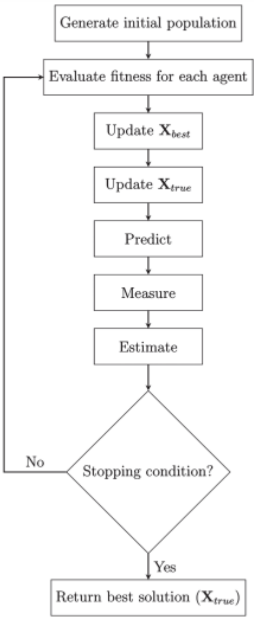

Generate initial population
Evaluate fitness for each agent
Update $\mathbf{X}_{best}$
Update $\mathbf{X}_{true}$
Predict
Measure
Estimate
Stopping condition?
No
Yes
Return best solution ($\mathbf{X}_{true}$)

Figure 4 Flowchart for Simulated Kalman Filter

2.2.5 Single-solution Simulated Kalman Filter

This algorithm is a variant of the original Simulated Kalman Filter metaheuristic which is a multi-agent metaheuristic. This variant is adapted to become a single-agent metaheuristic where there is only one search agent acting as a Kalman Filter. The proposed algorithm boasts simplified equations since it only uses one agent [5]. Its flowchart is described in Figure 5.

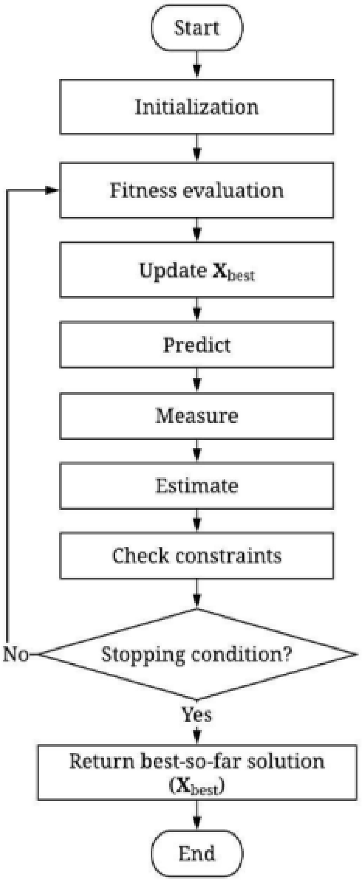


Figure 5 Flowchart for Single-solution Simulated Kalman Filter

2.2.6 Particle Swarm Optimization

A multi-agent metaheuristic inspired by the movement of flock of birds such as scattering and regrouping in search of food. Each agent also known as a particle is a candidate solution which moves around the search space during each iteration in search of improvements to the solution. The position and velocity are influenced by each particle’s best-known position as well as the best-known position of other particles as well [6]. The flowchart for Particle Swarm Optimization is illustrated in Figure 6.

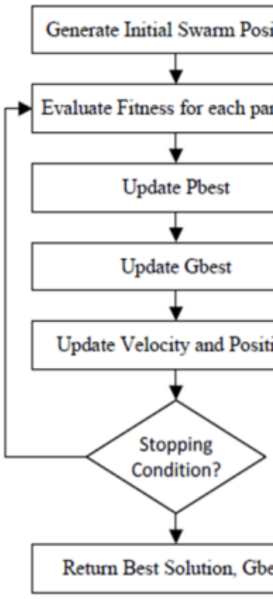

Generate Initial Swarm Position
Evaluate Fitness for each particle
Update Pbest
Update Gbest
Update Velocity and Position
Stopping Condition?
Return Best Solution, Gbest

Figure 6 Flowchart for Particle Swarm Optimization

2.2.7 Variants of Particle Swarm Optimization

Due to the popularity of the original Particle Swarm Optimization algorithm, a large variety of its variants were proposed to improve upon the algorithm in different aspects. The Binary Particle Swarm Optimization was introduced to adapt the original algorithm into a discrete search space which overcomes the problems faced by the original algorithm which was designed to be used in a continuous search space. In this variant, the particles represent its position in binary meanwhile its velocity is defined as the probability that it will change its state. The structure of the variant is like the original algorithm however it utilizes a separate set of equations since it is adapted to work in binary [7].

The Random Time-Varying Particle Swarm Optimization algorithm was introduced to enable the algorithm to reduce its processing time and evaluate positions of real-time locating systems with reasonable accuracy. The variant employs a smaller number of particles and fewer iterations to reduce the processing time to fulfil the rigid real-time conditions. In addition, the variant was adapted to produce quick and accurate solutions from a dynamic search space. The variant was also implemented on hardware which further improved its performance through simultaneous computations [8].

The Transitional Particle Swarm Optimization implements a transition in the original algorithm from asynchronous update at the start of the search and transitions to synchronous iteration towards the end of the run. This is because asynchronous iteration enables a better exploration meanwhile synchronous iteration enables a better exploitation of the search space. Synchronous iteration is done by evaluating the entire population then identifying the individual particle's and the population's best solution. In comparison, asynchronous iteration is done by immediately updating the particle's and population's best solution immediately upon completing its own fitness evaluation [9].

2.2.8 Binary Ant Colony Optimization

In the year 1991, the original Ant Colony Optimization metaheuristic was introduced. This paper implements the original metaheuristic into a binary solution domain such that the solution search space is represented in a binary format. Then its performance was verified through a binary function optimization problem opposed to the typical continuous function optimization problem [10].

2.2.9 Hybrid Binary Bat Enhanced Particle Swarm Optimization Algorithm

This metaheuristic combines two variants of the original Bat optimization and Particle Swarm Optimization to form a hybrid metaheuristic namely a combination of binary Bat metaheuristic and binary Particle Swarm Optimization to form the Hybrid Binary Bat Enhanced Particle Swarm Optimization Algorithm. It is claimed that the binary variants of metaheuristics are capable of producing superior results.

The binary Bat metaheuristic applies a binary map onto the solution found since the solution search space is continuous. Meanwhile the Binary Particle Swarm Optimization converts continuous values into binary values. The results shows that the hybrid metaheuristic is capable of producing better results than other binary variants of other metaheuristics such as binary Genetic Algorithm, binary Particle Swarm Optimization, binary Greywolf, binary Bat, and binary Dragonfly. The hybrid combination of both binary Bat and binary Particle Swarm Optimization produced better solutions than the individual metaheuristics [11].

2.3 FPGA Implementation Optimization Techniques

The performance of a metaheuristics on FPGA is dependent upon its design methodology. A typical implementation of a metaheuristic may vary in comparison to the implementation with different design methodology such as parallel implementations, pipeline architectures, and implementation of floating-point arithmetic modules.

2.3.1 Parallel Implementation of Particle Swarm Optimization

This implementation optimization technique was implemented to accelerate the processing speed of the algorithm. It was designed to process large volumes of data from processes such as Big Data and Mining of Massive Datasets. This was achieved through the implementation of several particle modules in parallel to concurrently compare the best fitness value of all particles as shown in Figure 7. Despite the increment of particle modules to enable parallel computing of the particles, the results of the hardware costs showed that it used less registers and Look Up Tables in comparison to similar works from the literature [12].

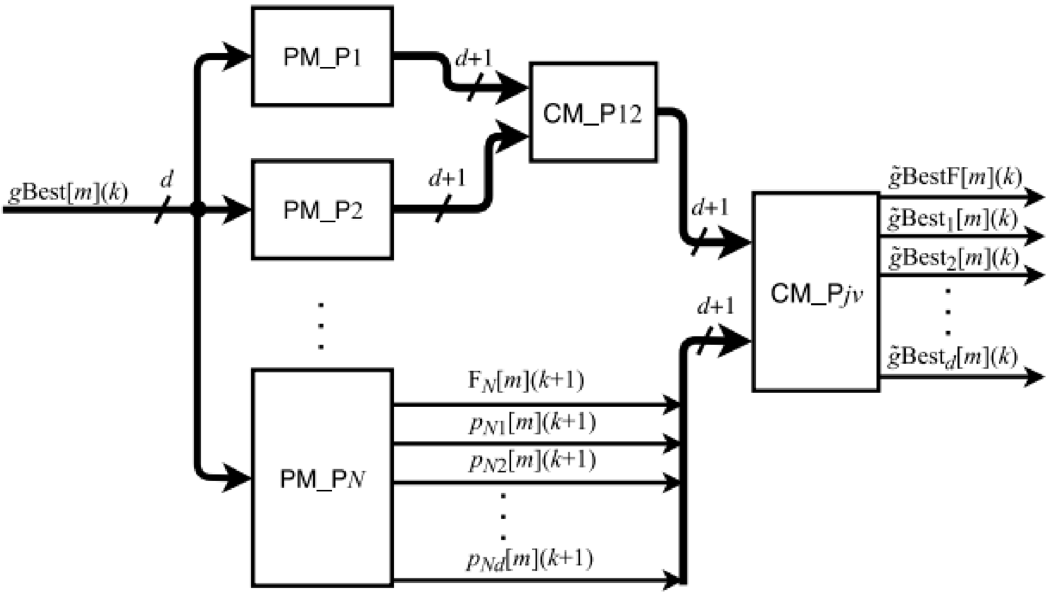


Figure 7 Parallel Implementation of Particle Modules

2.3.2 Pipeline Architecture of Particle Swarm Optimization

The paper works upon the previously mentioned variant of the Particle Swarm Optimization with random time-varying inertia weight and acceleration coefficients and its existing serial architecture implemented on hardware. The improvements proposed in this paper includes simplifications of the hardware architecture such as changing the control mechanism from a complicated distributed style into a simple centralized style to improve the stability of the hardware system as shown in Figure 8. This was done to overcome the calculation error that occurred in the previous implementation. Next, the performance of the implementation was improved by introducing registers and reconfiguration of the state transitions.

The implementation was able to successfully run 6 concurrent operations through a pipeline architecture as shown in Figure 9. In addition, the simplification of hardware architecture was able to successfully reduce errors and improve the rate of available results. The results reported showed a significant improvement in terms of performance. However, an increment in chip usage was observed [13].

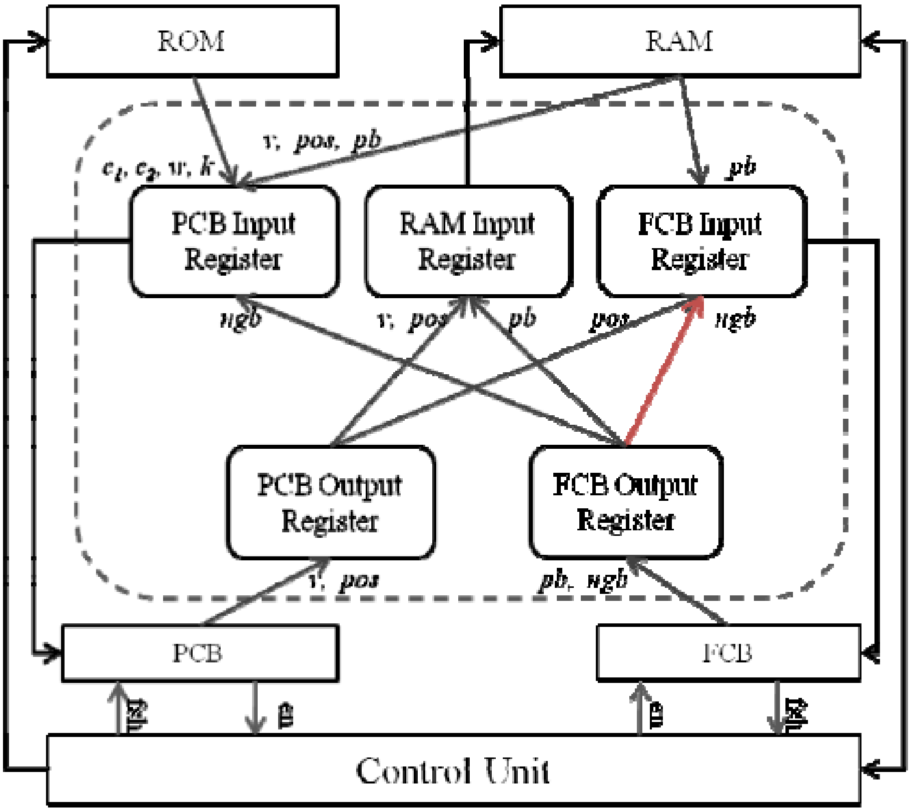


Figure 8 Data path between registers

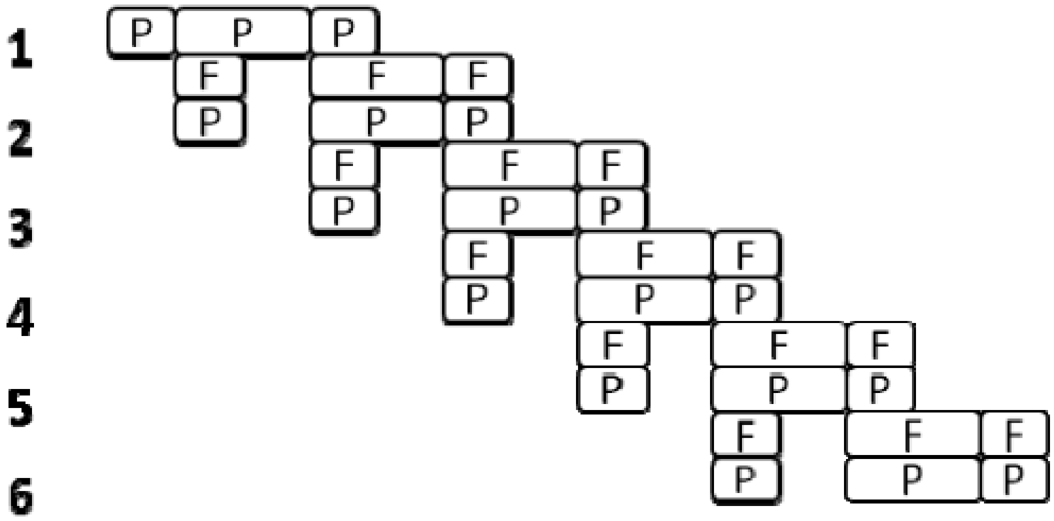


Figure 9 Pipeline structure

2.3.3 FPGA Realization of Particle Swarm Optimization Algorithm using Floating Point Arithmetic

The paper introduces the implementation of the original Particle Swarm Optimization algorithm with a modification upon its arithmetic modules. The arithmetic modules were replaced with floating-point arithmetic modules to further improve the accuracy of each particle results. This is because floating-point arithmetic used such as the single precision and double precision floating point arithmetic module enables the implementation to calculate to more decimal points. However, this resulted in extremely high chip usage. The implementation of the double precision floating point arithmetic module exceeded the total hardware available. The hardware cost was reduced by further implementing resource sharing for multiplication and addition operations which managed to free up some Look Up Tables [14].

During the review of the paper, it was noted that the paper did not reveal any results in the terms of accuracy despite the implementation of the floating-point arithmetic module was intended to improve the accuracy of results. However, it did successfully evaluate the hardware costs of implementing the floating-point arithmetic module and the effect of swarm size upon hardware costs.

CHAPTER 3

METHODOLOGY

3.1 Introduction

In this chapter, the process of adapting the original Simulated Kalman Filter metaheuristic into Discrete Simulated Kalman Filter is described. Then the Discrete Simulated Kalman Filter is then modelled using System Verilog through behavioural modelling to produce the Binary Simulated Kalman Filter which is then implemented into the FPGA. The individual modules are initially designed modularly and then integrated into a finite state machine to handle timing and interconnection of modules. Test benches were also produced to test the functionality of each individual module as well as the finite state machine prior to implementation into FPGA. Lastly, the performance parameter which is used to verify the performance of the implementation is discussed.

3.2 Simulated Kalman Filter

Through the process of elimination, the metaheuristic chosen for FPGA implementation was the Simulated Kalman Filter (SKF). The bio-inspired metaheuristics Barnacles Mating Optimizer and Orca Predation Algorithm proposed steps that were too complex and numerous. This would result in a difficult implementation of the algorithm as it may incur high costs to implement all the different states and modules of the algorithm. The Single-Agent Finite Impulse Response Optimizer proposes an iterative process with a sub-iterative process which introduces the same problem as mentioned above.

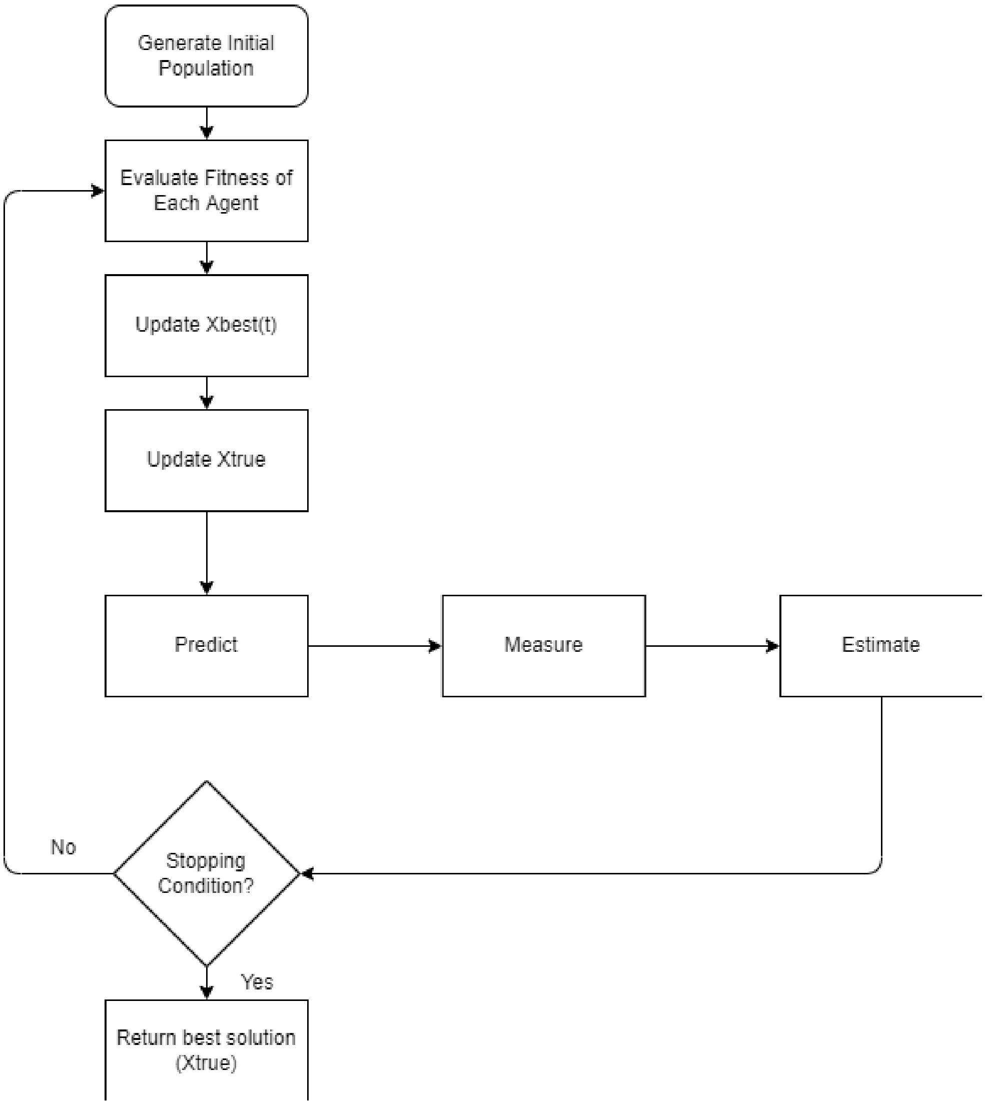
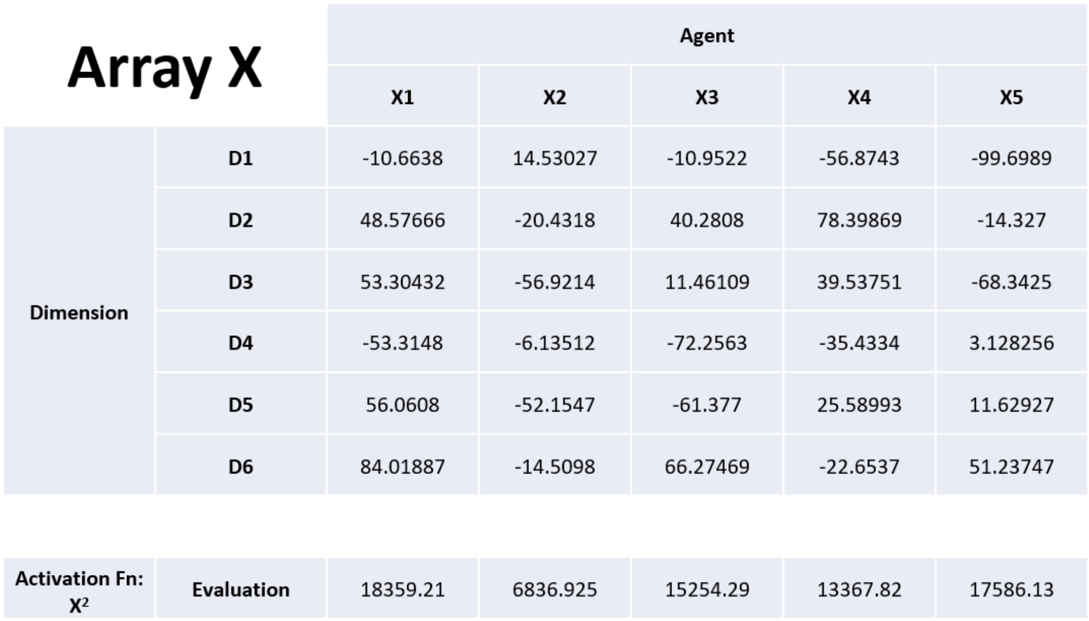

Generate Initial Population
Evaluate Fitness of Each Agent
Update Xbest(t)
Update Xtrue
Predict
Measure
Estimate
Stopping Condition?
No
Yes
Return best solution (Xtrue)

Figure 10 Simulated Kalman Filter flowchart

The SKF metaheuristic is composed of 4 major components namely the population generation, fitness evaluation of agents, fitness evaluation comparison and storage, and the Kalman Filter components which are the predict, measure, and estimate steps as shown in Figure 10. These steps are carried out iteratively until a stopping condition is met which is typically the maximum number of iterations. There are several variables of the Kalman Filter components which are significant such as the initial error covariance estimate, P(0), process noise, Q, measurement noise, R, and the Kalman gain, K.



Array X		**Agent**				
		X1	**X2**	**X3**	**X4**	**X5**
Dimension	**D1**	-10.6638	14.53027	-10.9522	-56.8743	-99.6989
	D2	48.57666	-20.4318	40.2808	78.39869	-14.327
	D3	53.30432	-56.9214	11.46109	39.53751	-68.3425
	D4	-53.3148	-6.13512	-72.2563	-35.4334	3.128256
	D5	56.0608	-52.1547	-61.377	25.58993	11.62927
	D6	84.01887	-14.5098	66.27469	-22.6537	51.23747

Activation Fn: X^2	**Evaluation**	18359.21	6836.925	15254.29	13367.82	17586.13

Figure 11 Generation of population and fitness evaluation of agent for SKF

Before the first iteration starts, several parameters need to be defined such as the number of agents, N, number of dimensions, D, and maximum number of iterations, t_{max}. Then the initial generation of population is done by generating a random floating-point value between the range of -100 to 100 and loading it into a multidimensional array X which has N columns and D rows.

Then the evaluation step is proceeded by evaluating the fitness of the agents through the activation function. The activation function of the original SKF metaheuristic utilizes the CEC2014 benchmark function which contains a set of 30 activation functions. Figure 11 illustrates the generation of population and fitness evaluation using the sphere activation function. The sphere activation function is described in Equation 3.1.

$$Fitness(t) = \sum_{n=1}^{D} X(t)^2 \qquad 3.1$$

After all agents have their fitness evaluated, all the fitness evaluation values are compared and the agent with the best fitness evaluation for the current iteration is selected as the $X_{best}(t)$. The selection process for $X_{best}(t)$ depends on whether it is a minimization problem or maximization problem. Equation 3.2 is used to determine $X_{best}(t)$ if it is a minimization problem and Equation 3.3 is used if it is a maximization problem. The $X_{best}(t)$ is then used to update the best-so-far solution of the run, X_{true} by replacing the best-so-far solution with the $X_{best}(t)$ value if $X_{best}(t) < X_{true}$ for minimization problems and $X_{best}(t) > X_{true}$ for maximization problems.

$$X_{best} = min_{i \in 1,\ldots,n} fitness_i\left(X(t)\right) \tag{3.2}$$

$$X_{best} = max_{i \in 1,\ldots,n} fitness_i\left(X(t)\right) \tag{3.3}$$

In the first iteration, the predict step is carried out by setting the initial error covariance to 1000, P(0) = 1000, the process noise is set to 0.5, Q = 0.5, and the measurement noise is set to 0.5, R = 0.5. Then the error covariance is calculated using Equation 3.4.

$$P = P + Q \tag{3.4}$$

Then the measurement step is carried out using Equation 3.5. where the X is the position of the agents and Y is the measured value of the agents.

$$\mathrm{Y} = X + \left(sin(rand(n,1) \times 2 \times \pi)\right) \times abs(X - X\_true) \tag{3.5}$$

Then the estimate step is proceeded to calculate the Kalman gain value as shown in Equation 3.6. Then the position of agents is updated using the Equation 3.7 and lastly the new error covariance is updated through the Equation 3.8.

$$K = \frac{P}{(P + R)} \tag{3.6}$$

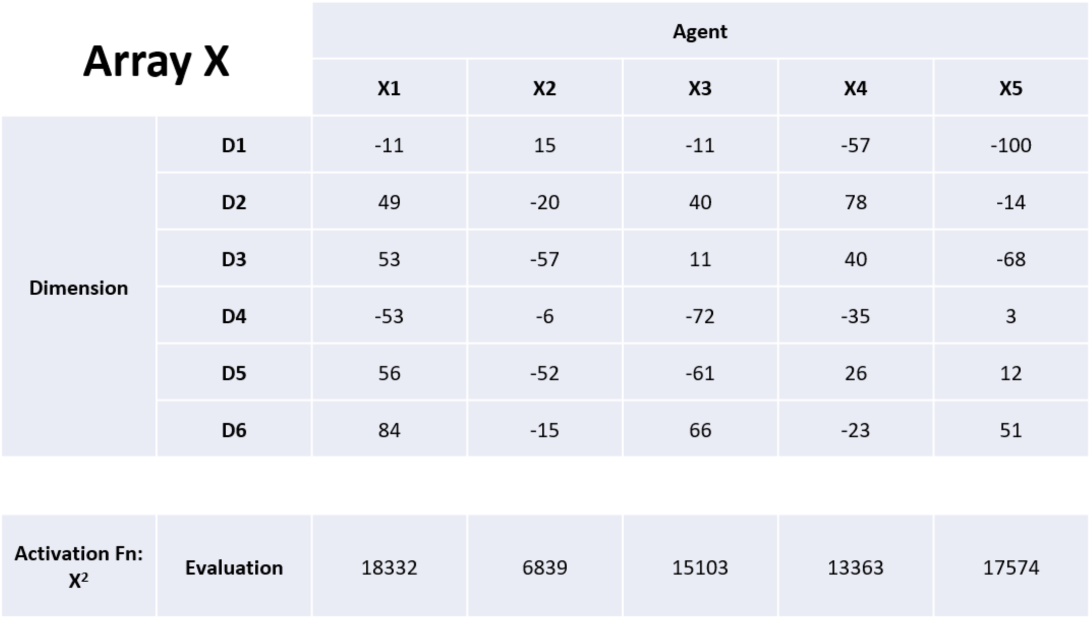
$$X = X + K \times (Y - X) \tag{3.7}$$

$$P = (1 - K) \times P \tag{3.8}$$

At the end of the estimate step of the Kalman Filter components, the stopping condition is checked and the run stops if the maximum number of iterations have been achieved. Else the run continues on to the next iteration and the steps are repeated from the evaluation of fitness agent.

3.3 Discrete Simulated Kalman Filter Adaptation

The original SKF metaheuristic was simulated using MATLAB utilizing floating-point values. The operation of this simulation is represented in Figure 11 where the agent positions as well as its fitness evaluation are represented by whole numbers, decimal point, and its corresponding fractional part. This introduced complications into the design of the FPGA implementation as it is difficult to represent the fractional parts of a number in a digital system. Hence, the original SKF was adapted into a Discrete SKF to remove this complication.



Array X

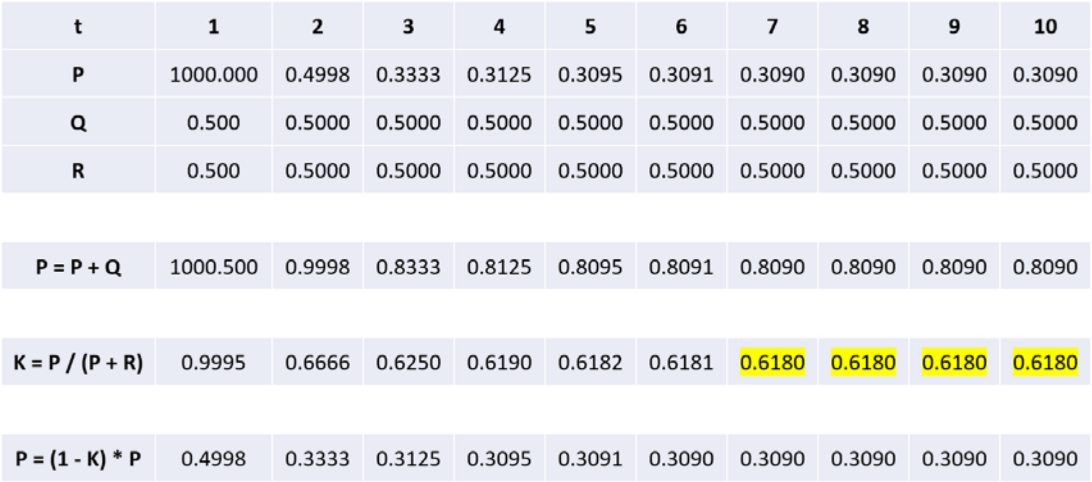
		Agent				
		X1	**X2**	**X3**	**X4**	**X5**
Dimension	**D1**	-11	15	-11	-57	-100
	D2	49	-20	40	78	-14
	D3	53	-57	11	40	-68
	D4	-53	-6	-72	-35	3
	D5	56	-52	-61	26	12
	D6	84	-15	66	-23	51

Activation Fn: X^2	**Evaluation**	18332	6839	15103	13363	17574

Figure 12 Generation of population and fitness evaluation of agent for Discrete SKF

The Discrete SKF utilizes the same minimization sphere function but utilizes only integer values. This is done by rounding all the floating-point agent position values to the nearest integer. This would consequently produce fitness evaluation values in whole number as well. The generation of population and fitness evaluation by Discrete SKF is illustrated in Figure 12.

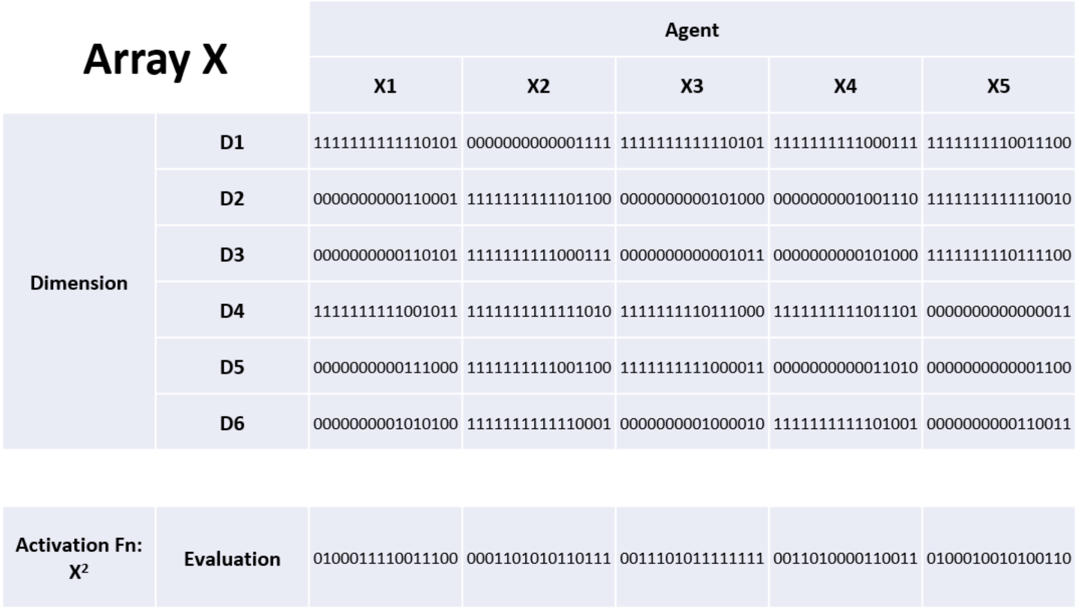
In addition to rounding up the floating-point values, the Kalman gain which was previously calculated using Equation 3.6 is set to a fixed value of K = 0.5. This is done because a multiplication of 0.5 can be translated into a division by 2 which can be operated in binary as a logical right shift of 1 bit. The deliberate value of 0.5 is set after thorough investigation of the transition of the Kalman gain in the original SKF MATLAB simulation. The Kalman gain can be observed to slowly decrease from 0.9995 at the first iteration to 0.6180 at the seventh iteration. The Kalman gain value then stagnates at 0.6180 until the end of the run as seen in Figure 13.



t	1	2	3	4	5	6	7	8	9	10
P	1000.000	0.4998	0.3333	0.3125	0.3095	0.3091	0.3090	0.3090	0.3090	0.3090
Q	0.500	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000
R	0.500	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000
P = P + Q	1000.500	0.9998	0.8333	0.8125	0.8095	0.8091	0.8090	0.8090	0.8090	0.8090
K = P / (P + R)	0.9995	0.6666	0.6250	0.6190	0.6182	0.6181	0.6180	0.6180	0.6180	0.6180
P = (1 - K) * P	0.4998	0.3333	0.3125	0.3095	0.3091	0.3090	0.3090	0.3090	0.3090	0.3090

Figure 13 Progression of Kalman gain in original SKF

This Discrete SKF was then simulated in MATLAB to verify its results. The significance of the adaptation from the original to discrete version of SKF is that the discrete version can be readily implemented in a digital system by representing the integer values as binary values. The implementation of Discrete SKF as Binary SKF is illustrated in Figure 14 where the integer values are represented in 2s complement binary values.



Array X

		Agent				
		X1	**X2**	**X3**	**X4**	**X5**
Dimension	**D1**	1111111111110101	0000000000001111	1111111111110101	1111111111000111	1111111110011100
	D2	0000000000110001	1111111111101100	0000000000101000	0000000001001110	1111111111110010
	D3	0000000000110101	1111111111000111	0000000000001011	0000000000101000	1111111110111100
	D4	1111111111001011	1111111111111010	1111111110111000	1111111111011101	0000000000000011
	D5	0000000000111000	1111111111001100	1111111111000011	0000000000011010	0000000000001100
	D6	0000000001010100	1111111111110001	0000000001000010	1111111111101001	0000000000110011

Activation Fn: X^2	**Evaluation**	0100011110011100	0001101010110111	0011101011111111	0011010000110011	0100010010100110

Figure 14 2s complement binary representation of Binary SKF

3.4 Binary Simulated Kalman Filter Behavioural Modelling

Each section of the Binary SKF metaheuristic was segmented and individually designed into separate modules using System Verilog. The major components that needed to be designed were the random number generator to generate the initial population of agents, Random-Access Memory (RAM) to store the agent position and measurement values, Activation Function module to evaluate the fitness of each agent according to Equation 3.1, Measure module to carry out measurement calculations as shown in Equation 3.5, and the Estimate module to update the position of agents according to Equation 3.7. The initial design produced is capable of processing 10 dimensions. This design is repurposed to produce 3 different variants which are the 5-dimension, 10-dimension and 20-dimension Binary SKF for FPGA implementation.

3.4.1 Parallel-In-Parallel-Out Configuration of Modules

During the behavioural modelling of the required modules, it was discovered that it was possible to produce modules with multiple input and output ports. This was significant as it enabled a module to receive multiple inputs, process the inputs, and produce multiple outputs at once. This Parallel-In-Parallel-Out (PIPO) configuration was exploited to maximise the number of inputs and outputs of each module and consequently improve their performance. The number of ports is dependent on the number of dimensions for that implementation. The rest of the chapter describes the design of the 10-dimension implementation.

3.4.2 Pseudo Random Number Generator (RNG)

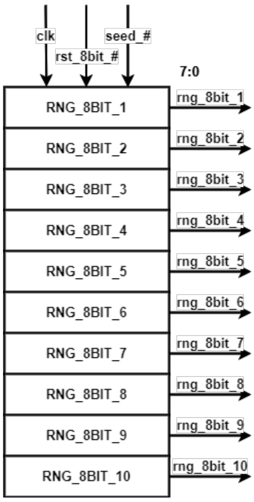


Figure 15 Stacked random number generator modules

The random number generator is designed upon the concept of Linear Feedback Shift Register (LFSR) which is a long chain of flip-flop with XOR gates at specific outputs of the flip-flop chain. The LFSR then produces a sequence of pseudo random numbers. Since the numbers generated are a sequence, the number will repeat themselves after $2^N - 1$ outputs where N is the polynomial factor or number of flip-flops used in the module. In this implementation, the module is designed with a polynomial factor of 16 which results in a sequence that repeats itself after 65535 number of outputs. In practical terms, this would be sufficiently long enough to be considered random. In addition, the sequence is determined by the input seed value. A different seed value would result in a different sequence of numbers. This is useful because it enables the same module to be reused to generate separate set of randomly generated numbers.

The bus width of the output is set to 8-bits to sufficiently represent the position of the agents which is between -127 to 127. To produce a value between -100 to 100, an internal boundary function is implemented to ensure the values produced are in the allowed range.

A single random number generator module is capable of outputting a single random number in a single clock cycle. This is sufficient to generate the inputs for the RAM holding the agent positions. To further improve the speed of the generator, it was decided to stack 10 modules together to generate 10 random values as shown in Figure 15. Each random number generator then outputs the value for a single dimension.

3.4.3 Random-Access Memory (RAM)

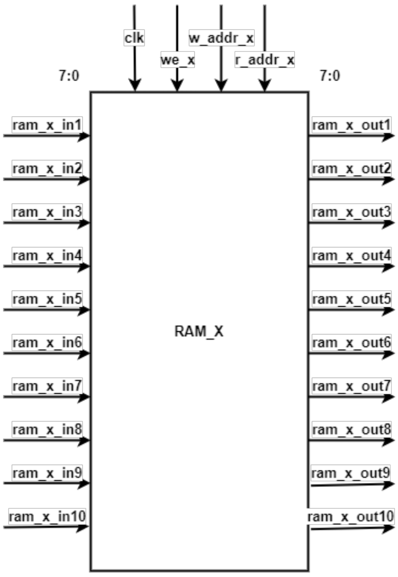


Figure 16 Random-Access Memory module

The Random-Access Memory (RAM) module was designed to temporarily store the position value of the all the dimensions of the agents as illustrated in Figure 16. The RAM module is a multidimensional register with N number of columns and D number of rows. The input and output ports are similarly designed in PIPO arrangement which allows all dimension of a single agent to read and written to in a single clock cycle.

The we_x is the write enable port. When pulled high, it enables writing to the module and when pulled low, it disables writing to the module. The w_addr_x and r_addr_x ports are the write to and read from address ports, respectively. The writing operation is done by pulling we_x high, inputting a specific address between value of 0 to N-1 and inputting the value at the input ports. To read from the RAM, sending an address value between 0 to N-1 to the r_addr_x port will result in the value of the address produced at the output ports. The bus width of all the input and outputs are 8-bits wide similarly to the RNG module. The instantiation of this module to store the position of agents is called RAM_X.

The module is also instantiated with a larger input and output bus width of 9 bits called RAM_Y. This separate module is used to store the output values from the Measure module. The enhanced resolution is to facilitate the arithmetic operations upon the agent's original position which is conducted in the Measure module.

3.4.4 Activation Function

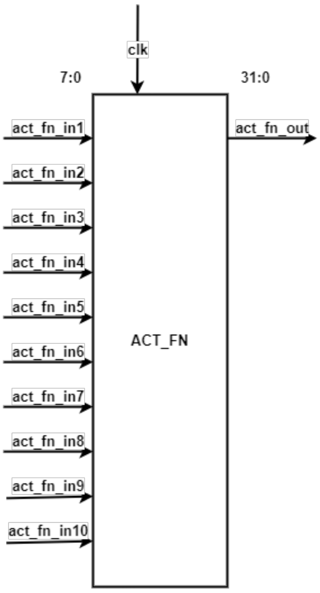


Figure 17 Activation Function module

The Activation Function module facilitates the calculations required to evaluate the fitness of agent using the activation function described in Equation 3.1. The module receives input from the RAM_X module and then computes and outputs the evaluation of the agent as shown in Figure 17. In a single clock cycle, every dimension of an agent is loaded into the module and its fitness evaluation is outputted from its output port.

3.4.5 2-bit Pseudo Random Number Generator

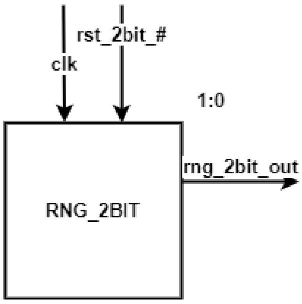


Figure 18 2-bit Pseudo Random Number Generator

This RNG module is a modified version of the 8-bit pseudo random number generator that produces an output with only 2-bits and further bounded to 3 specific outputs which are -1, 0 and 1. This is done to simulate the $\left(sin(rand(n,1)\ \times 2\ \times\ \pi)\right)$ component of the measure step as described in Equation 3.5 which originally generates a floating-point value ranging between -1 to 1. The module is represented in Figure 18.

3.4.6 Measure

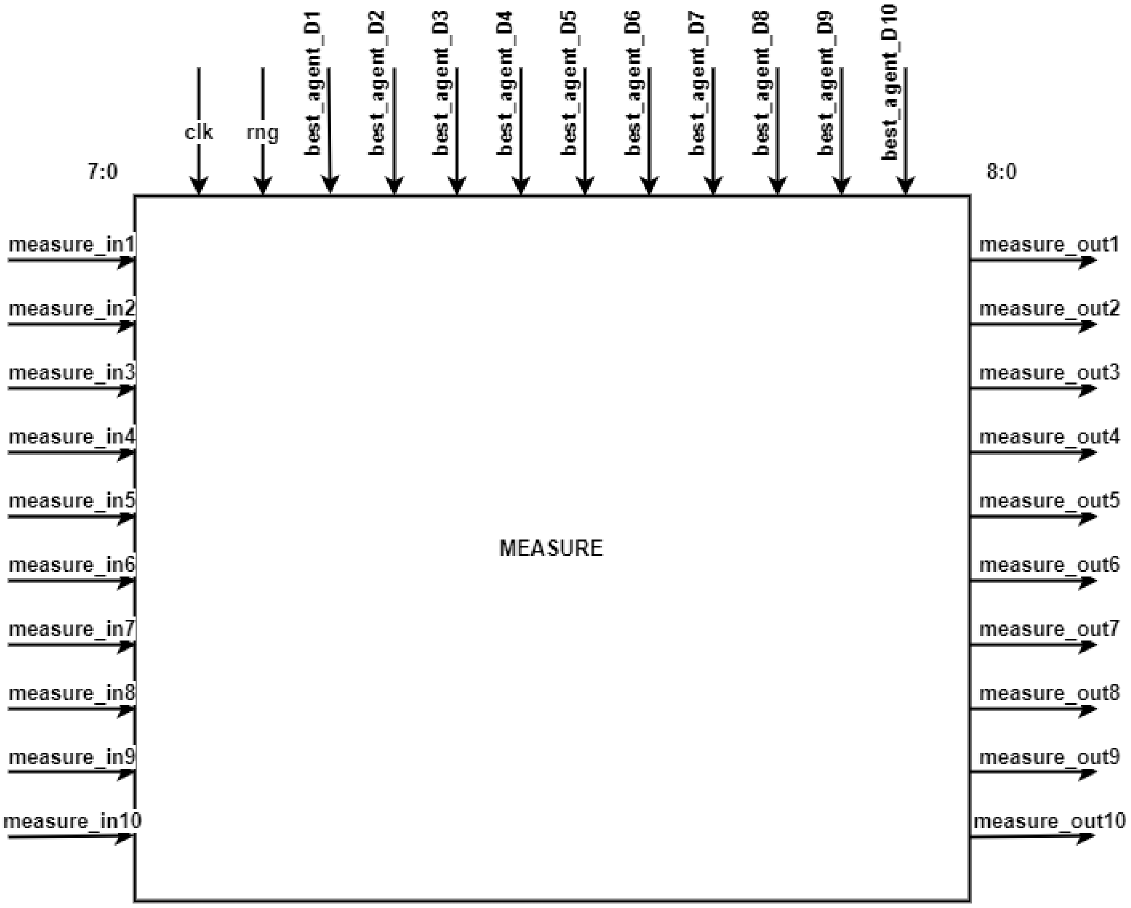


Figure 19 Measure module

The Measure module facilitates the calculations required to carry out the measure step as described in Equation 3.5. It receives a random number of either -1, 0, or 1 from its rng port which is then used for the calculation of the outputs. The best_agent_D# ports are the input ports of the position value of the X_{true} which are used for the $abs(X - X\_true)$ component of Equation 3.5. The input ports receive the agent position values from the RAM_X output ports which is then processed and produced at the output ports as shown in Figure 19. The output ports are 9-bits wide to facilitate the measurement arithmetic operations and enable an output value ranging between -511 to 511.

3.4.7 Estimate

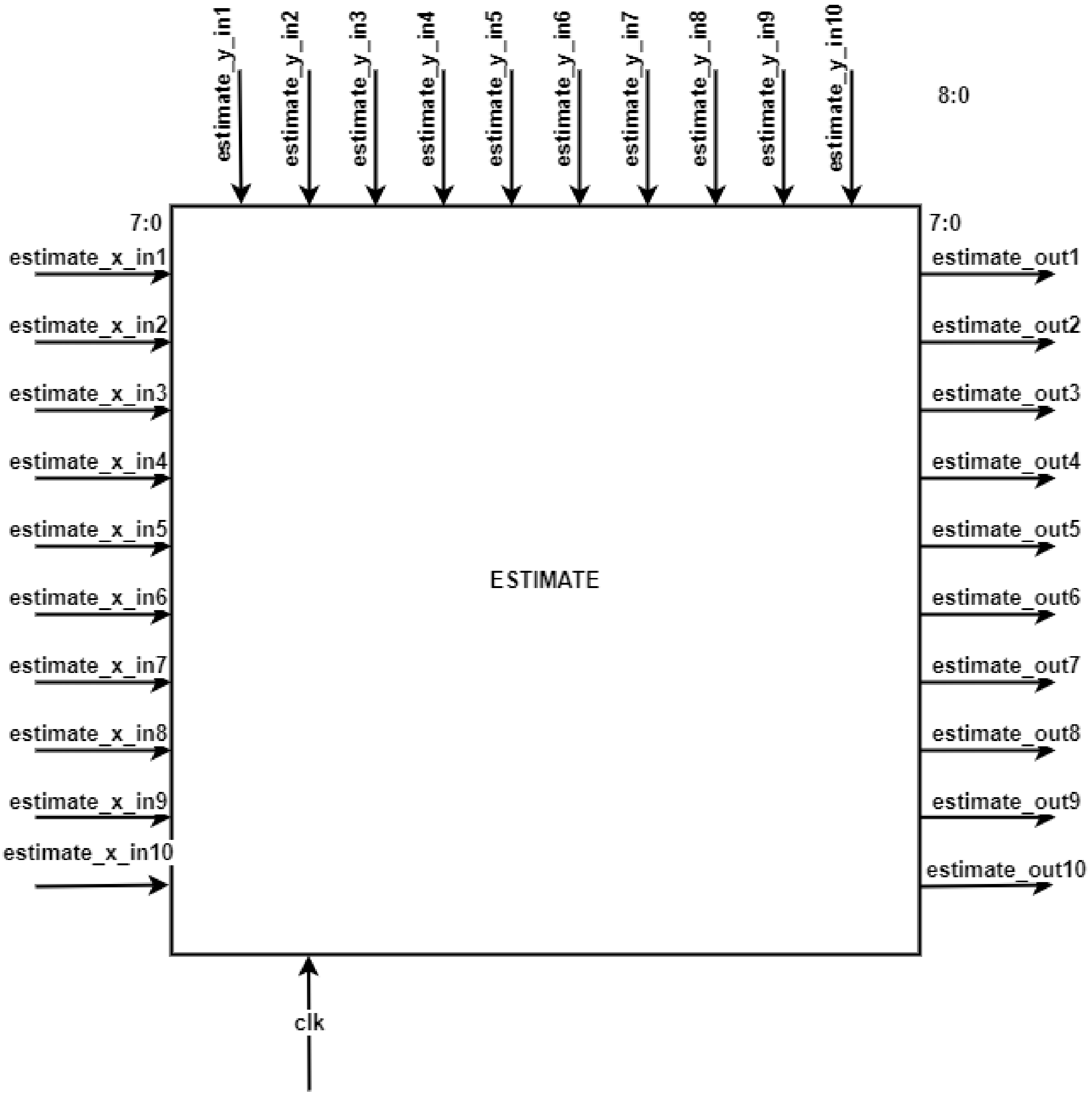


Figure 20 Estimate module

The Estimate module facilitates the calculations required to carry out the estimate step as described in Equation 3.7. The module receives inputs from both RAM_X and RAM_Y which it then processes and outputs in its output ports as shown in Figure 20. The output value is internally bounded to ensure that the output values are within the acceptable range of -100 to 100. The output values are then loaded to the input ports of the RAM_X to update the new value of agent position.

3.4.8 Finite State Machine Controller

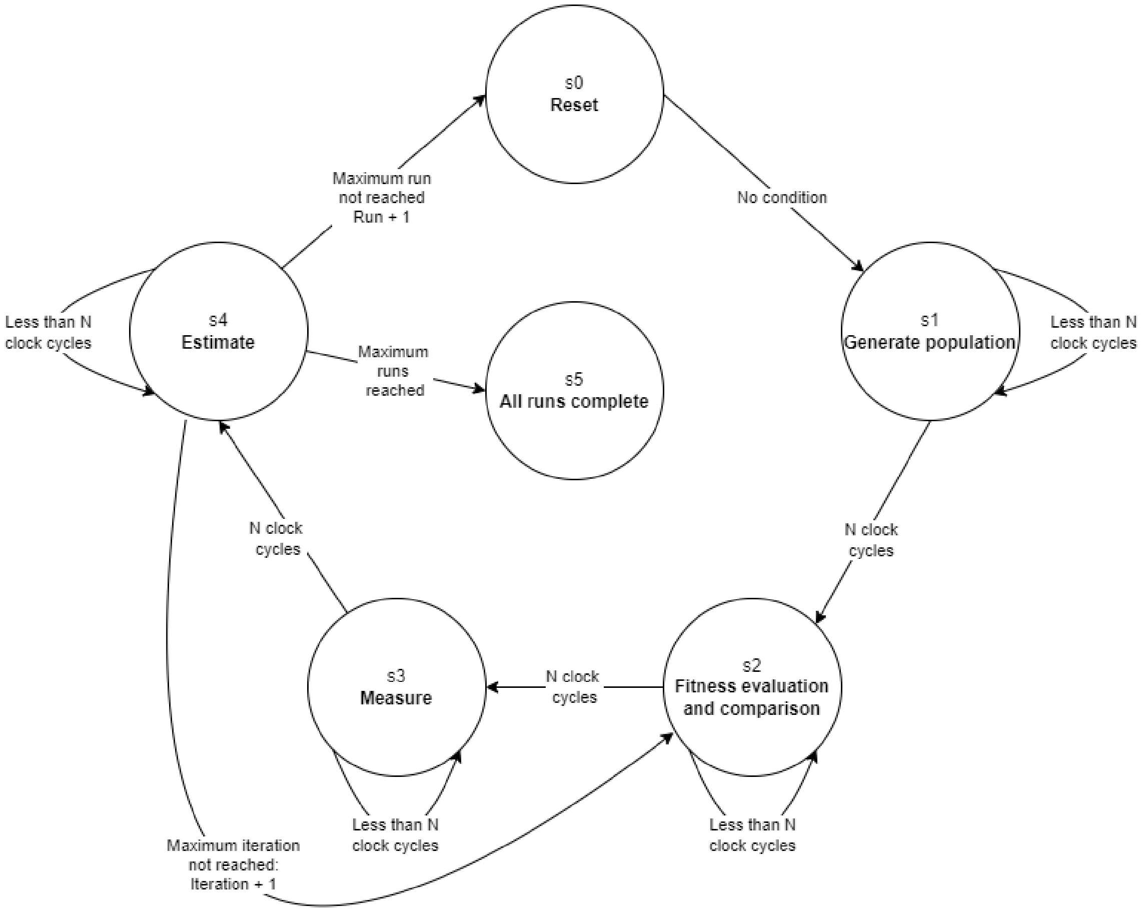


Figure 21 Simplified state diagram of finite state machine controller

To ensure proper timing and flow of data between modules, a finite state machine (FSM) was modelled using System Verilog. The FSM instantiates all the necessary modules and the necessary wires for interconnection between modules. The FSM has 6 states in sequence of s0 Reset, s1 Generate population, s2 Fitness evaluation and comparison, s3 Measure, s4 Estimate, and s5 Complete. A simplified state diagram of the FSM is illustrated in Figure 21.

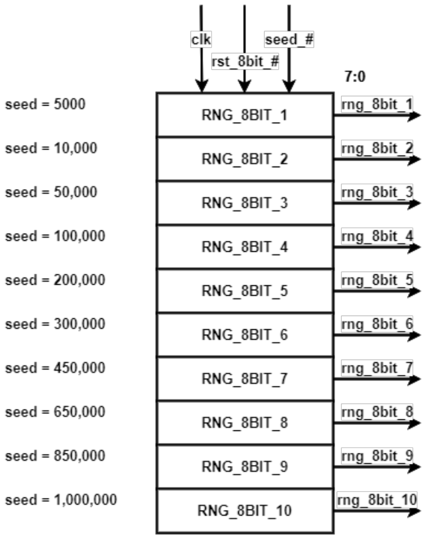


Figure 22 State s0 Reset

In state s0, all modules are reset. In this state, the initial seed values are input into the RNG modules, all comparison and storage registers are cleared except for X_{true}, and write enable for RAM_X is disabled a shown in Figure 22. There is no condition for moving onto the next state, so it will proceed to state s1 upon the next clock cycle.

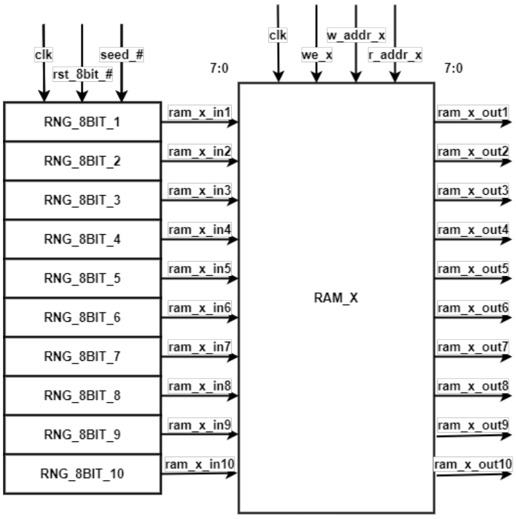


Figure 23 State s1 Generate population

In state s1, all modules have their reset ports pulled low to leave reset state. The RNG stack output is then connected to the input ports of the RAM_X as shown in Figure 23. At the same time, the write enable for RAM_X is enabled. On the first clock cycle, the write address is directed to the address 0 and increases by 1 on each positive edge of the clock signal. When the write address reaches the value N-1, the write enable of RAM_X is disabled, the RNG reset is pulled high, and the FSM enters the next state.

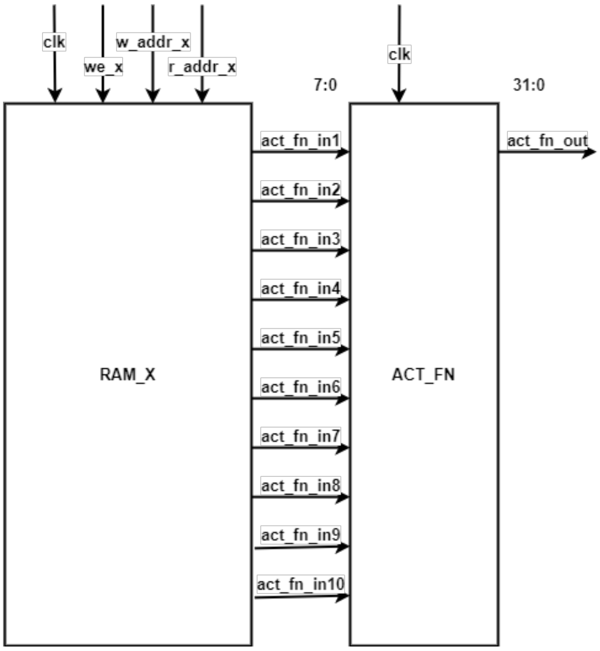


Figure 24 State s2 Fitness evaluation and comparison

In state s2, the output ports of RAM_X is connected to the input ports of the Activation Function module as shown in Figure 24. On the first clock cycle, the read address of RAM_X is directed to address 0 and increases by 1 on each positive edge of the clock signal. The Activation Function module then processes and outputs the agent evaluation at its output port. The FSM controller then compares the current evaluation with the overall best evaluation for the current iteration to update the X_{best} register. At the same time, it updates the X_{true} register and lastly moves onto the next state when the read address of RAM_X reaches the value N.

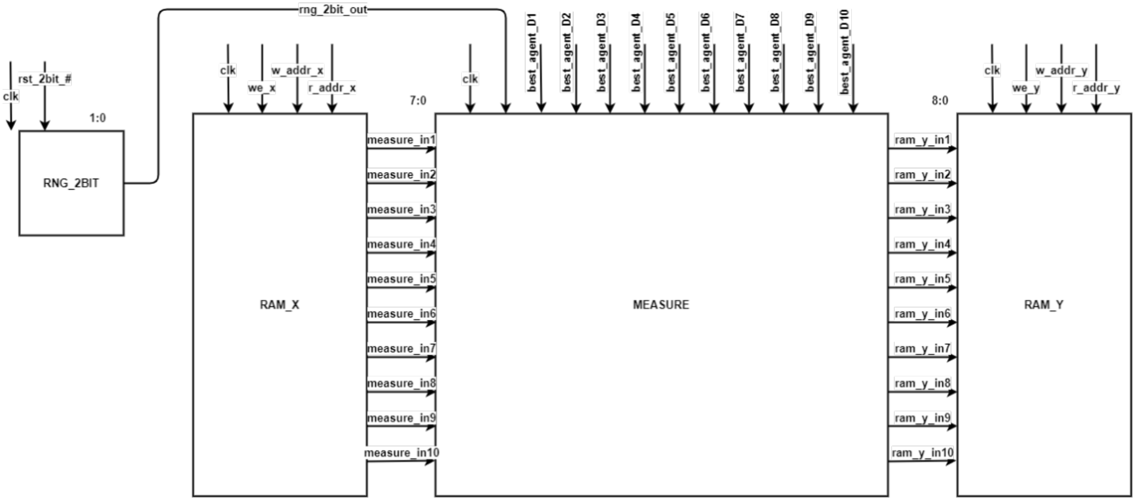


Figure 25 State s3 Measure

In state s3, the output ports of RAM_X is connected to the input ports of the Measure module. The 2-bit RNG module is connected to the rng port of the Measure module and the best_agent_D# ports are driven by the FSM controller registers as shown in Figure 25. The output ports of the Measure module are connected to the input ports of the RAM_Y module and the write enable for RAM_Y is enabled. On the first clock cycle, the read address of the RAM_X is directed to address 0 meanwhile the write address of the RAM_Y is directed to address 0. Both addresses increase on each clock cycle until they reach the value of N-1 which then triggers the condition to enter the next state.

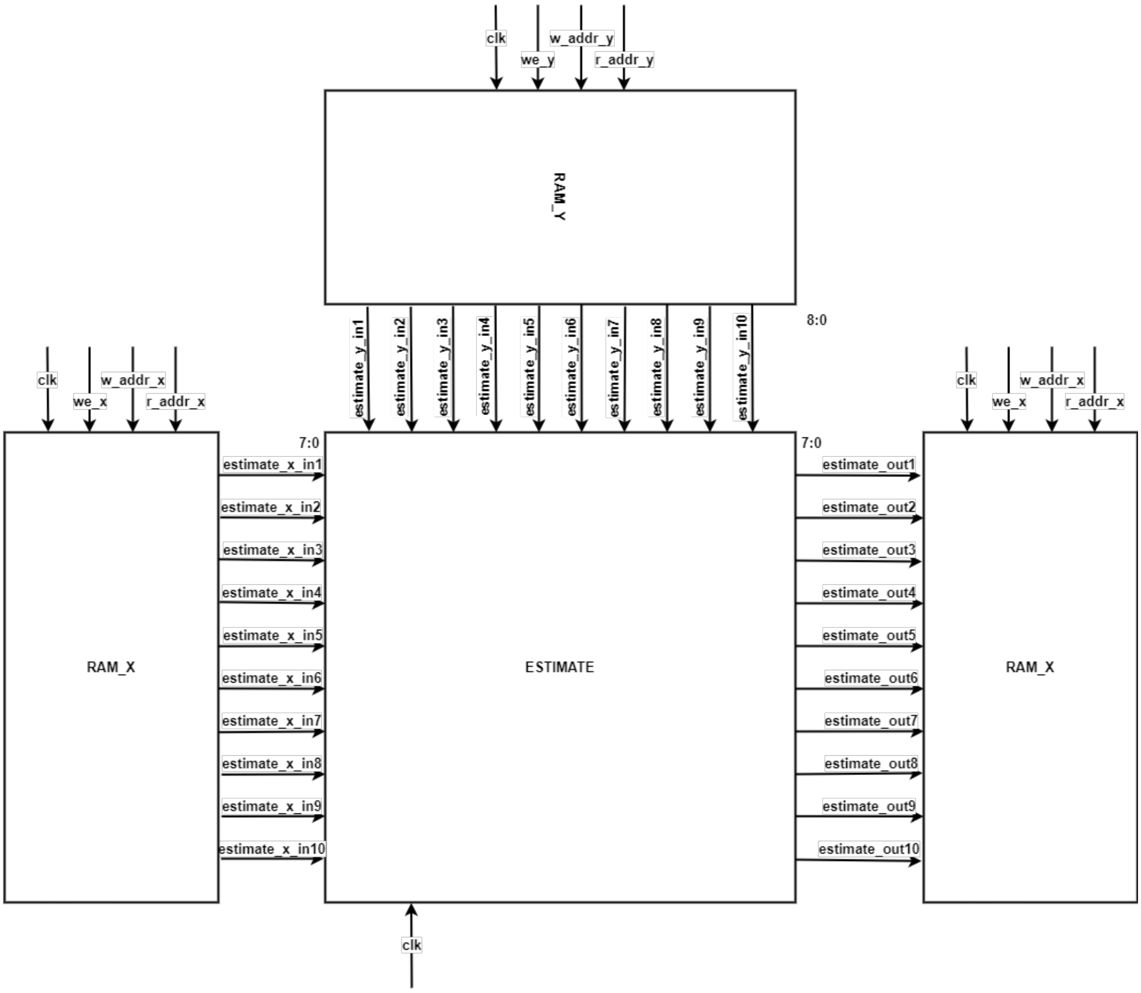


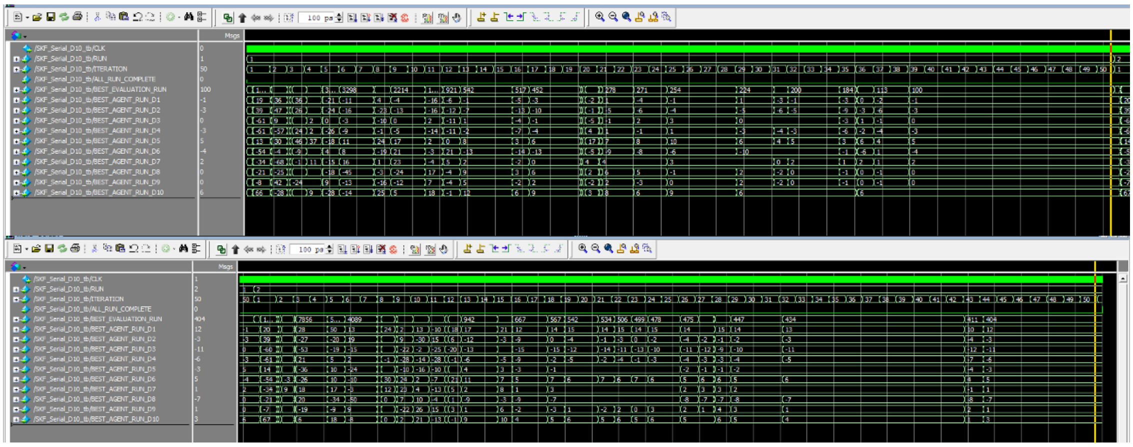
Figure 26 State s4 Estimate

In state s4, the output ports of RAM_X and RAM_Y are connected to the input ports of the Estimate module as shown in Figure 26. The output ports of the Estimate are then connected to the input ports of the RAM_X module. The write enable for RAM_Y is disabled meanwhile the write enable for RAM_X is enabled to allow the Estimate module to update the positions of the agents. On the first clock cycle, the read address of both RAM_X and RAM_Y is directed to address 0 and increases by 1 on each clock cycle. On the second clock cycle, the write address on RAM_X is directed to address 0 so that the output of the Estimate module can overwrite the value of the agent in the address. Similarly, the RAM_X write address increases by 1 on each clock cycle. When the write address reaches the value of N-1, the state machine transits to the next state depending on the current number of runs, and the current iteration number.

If the number of iterations have not reached t_{max}, the FSM will enter state s2 to move onto the next iteration of the run. If the maximum number of iterations have been reached but the maximum number of runs have not been reached, the FSM will enter state s0 and move onto the next run. Lastly, if the maximum number of iterations and maximum number of runs have been achieved, the FSM enters state s5 which marks the completion of all operations. In this state, the FSM displays the result of $X_{true,}$ and the time interval taken to complete all iterations and runs.

3.4.9 Testbench Verification

Several testbenches were designed using System Verilog to test the functionality of each module and the FSM prior to implementation into FPGA. The individual modules were tested to ensure the calculations produce the expected results prior to integration into the FSM. Then the FSM itself was simulated on a testbench to ensure proper timing as well as module interconnection was achieved. The waveform produced by the testbench is as shown in Figure 27 and is used to troubleshoot any errors that may occur.

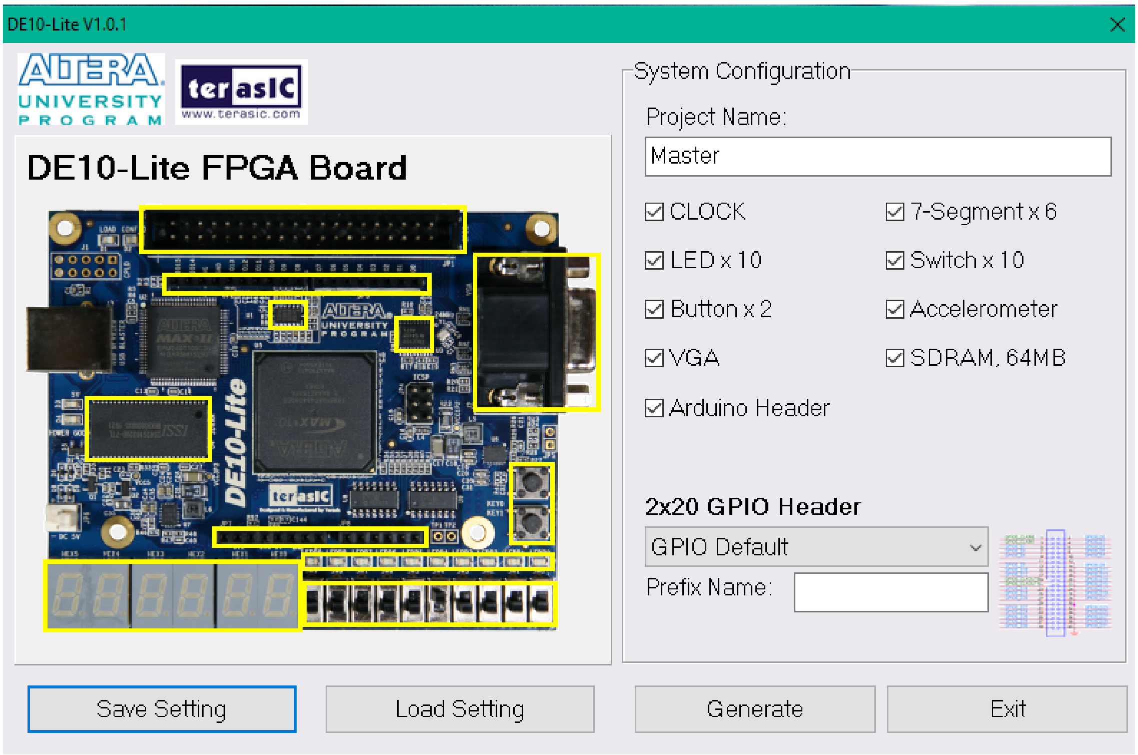


```
/SKF_Serial_D10_tb/CLK                    0
/SKF_Serial_D10_tb/RUN                    1
/SKF_Serial_D10_tb/ITERATION              50
/SKF_Serial_D10_tb/ALL_RUN_COMPLETE       0
/SKF_Serial_D10_tb/BEST_EVALUATION_RUN    100
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D1      -1
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D2      -3
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D3      0
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D4      -3
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D5      5
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D6      -4
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D7      2
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D8      0
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D9      0
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D10     6
```

```
/SKF_Serial_D10_tb/CLK                    1
/SKF_Serial_D10_tb/RUN                    2
/SKF_Serial_D10_tb/ITERATION              50
/SKF_Serial_D10_tb/ALL_RUN_COMPLETE       0
/SKF_Serial_D10_tb/BEST_EVALUATION_RUN    404
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D1      12
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D2      -3
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D3      -11
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D4      -6
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D5      -3
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D6      5
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D7      1
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D8      -7
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D9      1
/SKF_Serial_D10_tb/BEST_AGENT_RUN_D10     3
```

Figure 27 Waveform generated by a testbench with parameters N = 50, D = 10, maxrun = 2, t_{max} = 50.

3.5 Hardware Components



DE10-Lite V1.0.1

DE10-Lite FPGA Board

System Configuration

Project Name: Master

☑ CLOCK
☑ 7-Segment x 6
☑ LED x 10
☑ Switch x 10
☑ Button x 2
☑ Accelerometer
☑ VGA
☑ SDRAM, 64MB
☑ Arduino Header

2x20 GPIO Header: GPIO Default

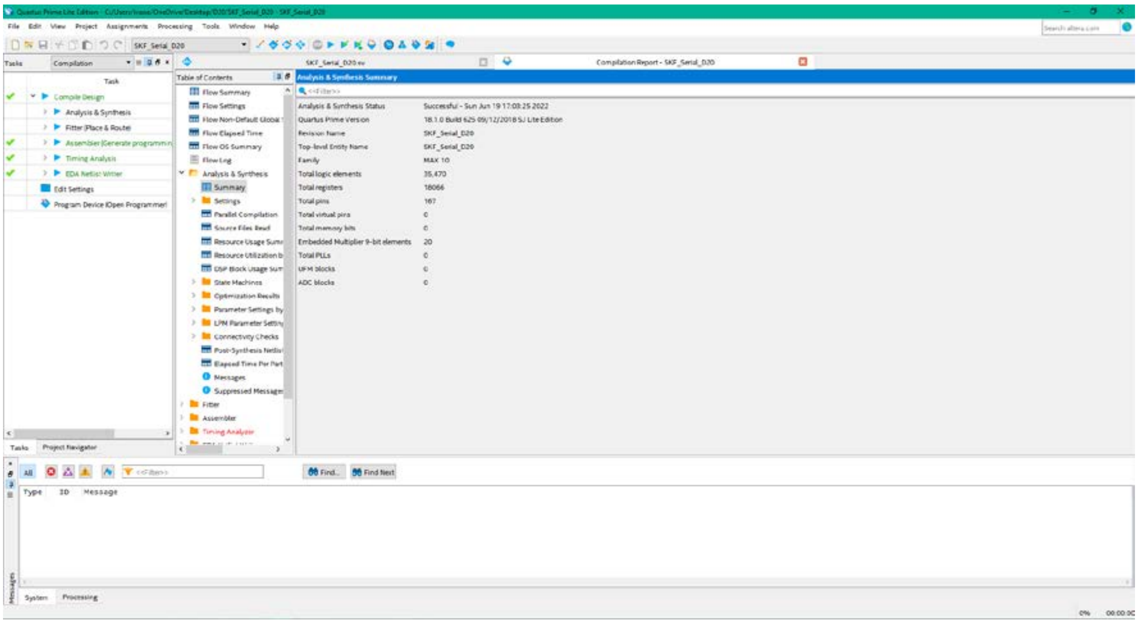
Prefix Name:

Figure 28 DE10 – Lite System Builder application

The board chosen for the implementation of the Binary SKF was the DE10 - Lite which is powered by the 10M50DAF474G chip produced by Altera. The board was a provided by the Intel FPGA University Program. The board provides sufficient peripheral ports and interaction devices such as LEDs, slide switches, temporary pushbuttons, and seven – segment displays. Included on the board is a clock generation chip which outputs a stable 50MHz clock cycle to drive the 10M50DAF474G chip. The FPGA chip is sufficiently powerful to power the design as it contains 50 thousand logical elements, 1638 Kb of M9K memory, 5888 Kb of user flash memory, and 144 units of 18 x 18 multiplier.

Meanwhile, the MATLAB simulation of Discrete SKF is ran on a HP ProBook 440 G7 which is equipped with a 10th generation Intel processor namely the Intel Core i5 – 10210U CPU and is paired with 16 gigabytes of RAM.

3.6 Software Components


Analysis & Synthesis Summary
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
Timing Analysis
EDA Netlist Writer
Edit Settings
Program Device (Open Programmer)
Compilation Report - SKF_Serial_D20
Find...
Find Next

Analysis & Synthesis Status	Successful - Sun Jun 19 17:03:25 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	SKF_Serial_D20
Top-level Entity Name	SKF_Serial_D20
Family	MAX 10
Total logic elements	35,470
Total registers	18066
Total pins	167
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	20
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 29 Quartus Prime Lite Edition 18.1

The software used to design and implement the Binary SKF onto FPGA is the Quartus Prime Lite Edition 18.1. It is a proprietary software by Intel that is used to compile, synthesize, place, route and analyse designs for implementation onto Altera FPGA chips. The software also enables access to the IP Catalog which contains a library of parametric modules which can be easily integrated into existing designs. The software is available for free on the Intel website and does not require license to operate. However, the performance of the software is limited and can be unlocked with the purchase of a license.

3.7 Performance Parameters

After the Binary SKF has been programmed onto the FPGA, its performance was verified based on several metrics such as the chip utilization, processing speed and the accuracy of the result. The design was also altered to produce 3 variants, each handling different number of dimensions. The 3 variants are set to handle 5-dimensions, 10-dimensions and 20-dimensions, respectively. The processing speed and accuracy of result produced by the variant implementations were then compared against the Discrete SKF metaheuristic simulated in MATLAB. Figure 30 illustrates the performance parameters that will be evaluated.

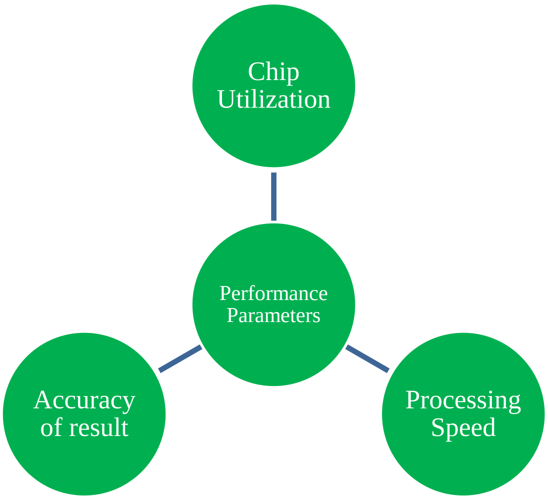


Figure 30 Performance parameters

3.7.1 Chip Utilization

The resources on the 10M50D474CG is limited to 50 thousand logical elements. As a design increases in complexity, more logical elements are used to synthesize the design. The chip utilization of the designs can be analysed using the Analysis and Synthesis Summary report generated by the Quartus Prime software.

3.7.2 Processing Speed

The processing speed of the Binary SKF is defined as the time taken for the design to complete all calculations and display the results. The time taken for the FPGA implementation is set to be displayed on the seven-segment display during the operation and stop when the calculation processes are done. Similarly, the MATLAB simulation of the Discrete SKF is set to display the time taken to complete all the iterations and runs. A faster processing speed would mean a shorter time period taken to complete all calculations.

3.7.3 Accuracy of Result

Since the problem is a minimization of the sphere function, the expected result would be X = 0 for all dimensions since a value of 0 would produce the smallest possible fitness evaluation. The results produced by the Discrete SKF simulation, and the Binary SKF would then be compared to the expected value to verify the accuracy of their outputs. The closer the output result is to the expected result, the more accurate the output.

CHAPTER 4

Results & Discussion

4.1 Introduction

In this chapter the results of the FPGA implementation of Binary SKF metaheuristic and MATLAB simulation of Discrete SKF metaheuristic is presented. The results then analysed and discussed based on the performance parameters.

4.2 Experimental Setup

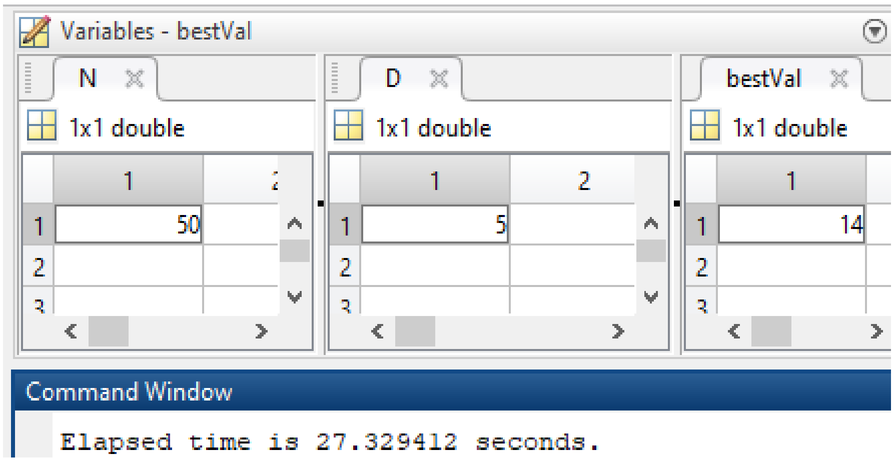
The run parameter set for both the MATLAB simulation and FPGA implementation is set to 50 maximum runs, 5000 maximum iterations, 50 agents, and fixed Kalman gain of 0.5. The runs are conducted 3 times on MATLAB using 3 different dimension values which are 5-dimensions, followed by 10-dimensions, and lastly 20-dimensions. The FPGA implementation is programmed with 3 different designs and ran 3 separate times to facilitate the run of the 3 different variants namely the 5-dimension, 10-dimension, and 20-dimension implementations. The time taken to complete all calculations, the value of X_{true} and chip utilization of each design is noted is compiled into Table 1.

4.3 Simulation and Implementation Results

Table 1 Compiled result of simulation and FPGA implementation

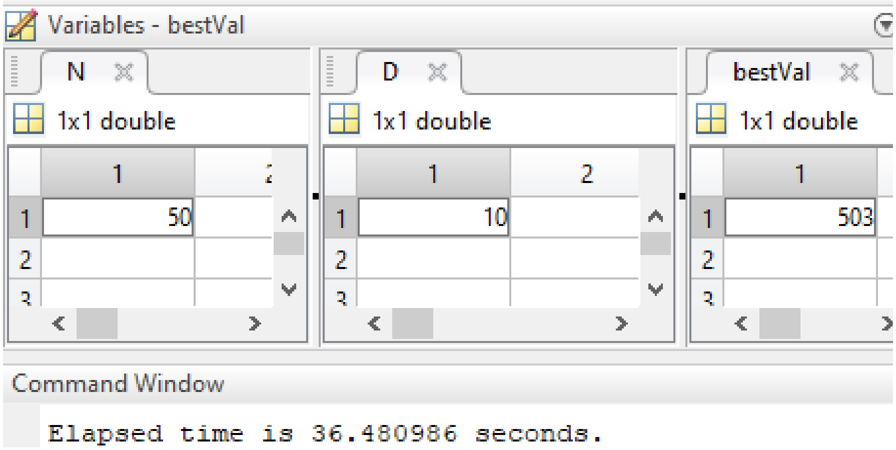
	Agents, N	Dimensions, D	X_{true}	Time taken (s)	Logical Elements	Registers	Multiplier 9-bit elements
Simulation	**50**	**5**	**14**	**27.329**	**-**	**-**	**-**
	50	**10**	**503**	**36.481**	**-**	**-**	**-**
	50	**20**	**3936**	**54.571**	**-**	**-**	**-**
FPGA	**50**	**5**	**32**	**0.780**	**8220**	**4686**	**5**
	50	**10**	**253**	**0.780**	**14637**	**9146**	**10**
	50	**20**	**4749**	**0.780**	**27413**	**18066**	**20**

4.3.1 MATLAB Simulation of Discrete Simulated Kalman Filter



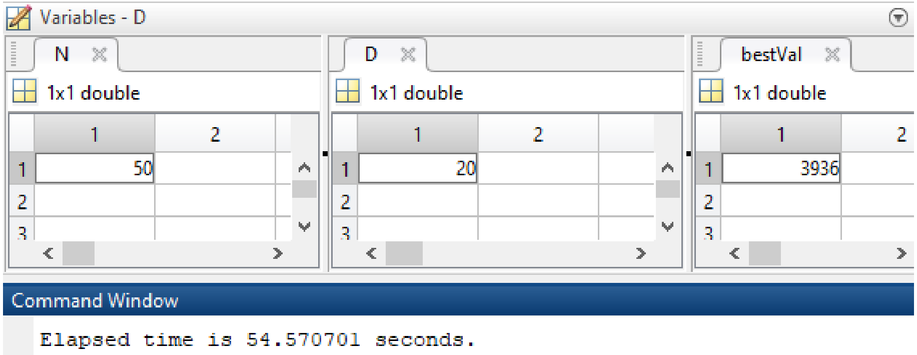
```
Elapsed time is 27.329412 seconds.
```

Figure 31 MATLAB simulation result for D = 5



```
Elapsed time is 36.480986 seconds.
```

Figure 32 MATLAB simulation result for D = 10



```
Elapsed time is 54.570701 seconds.
```

Figure 33 MATLAB simulation result for D = 20

4.3.2 FPGA Implementation of Binary Simulated Kalman Filter

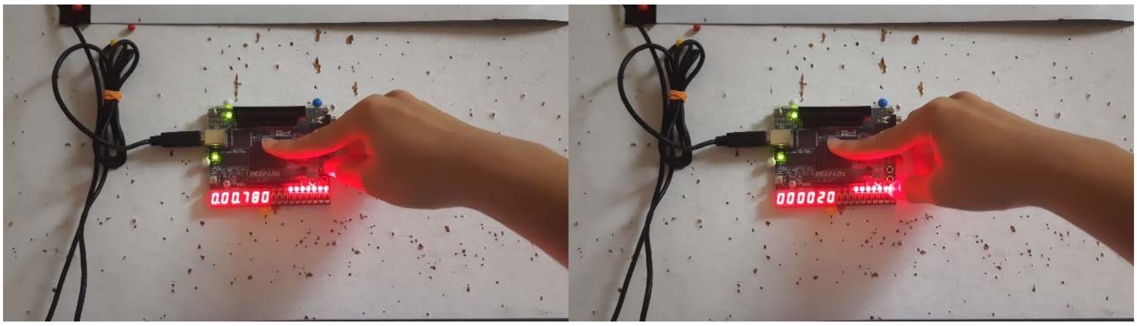


Figure 34 FPGA implementation result for D = 5

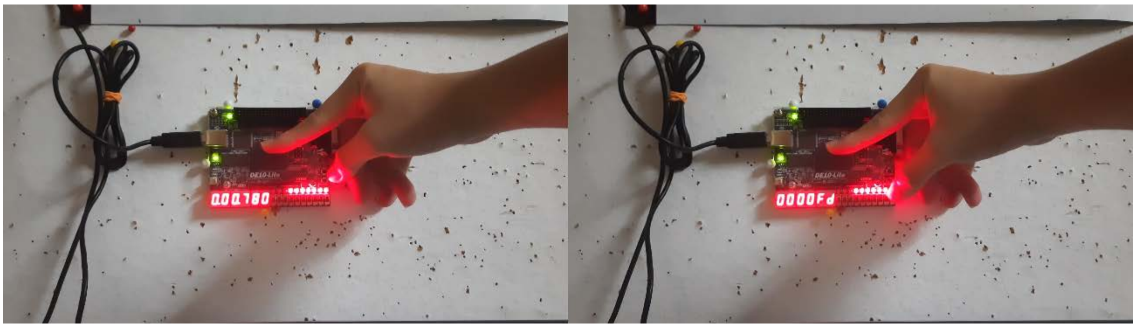


Figure 35 FPGA implementation result for D = 10

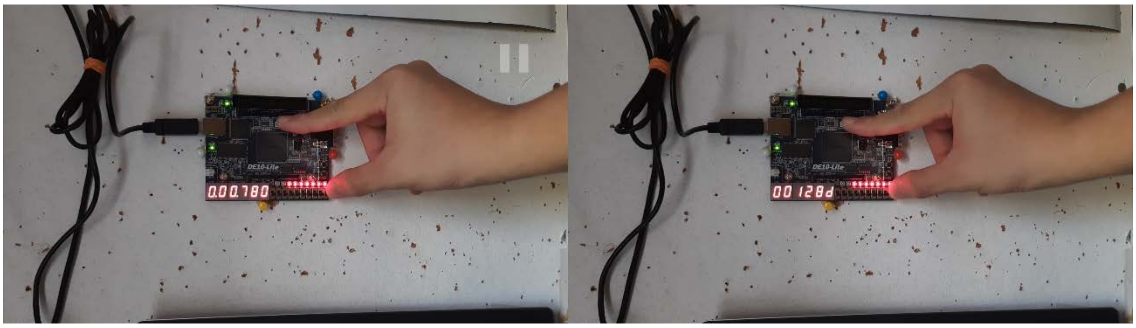
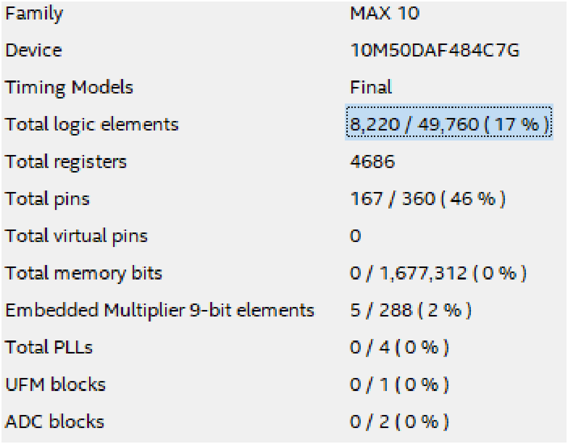
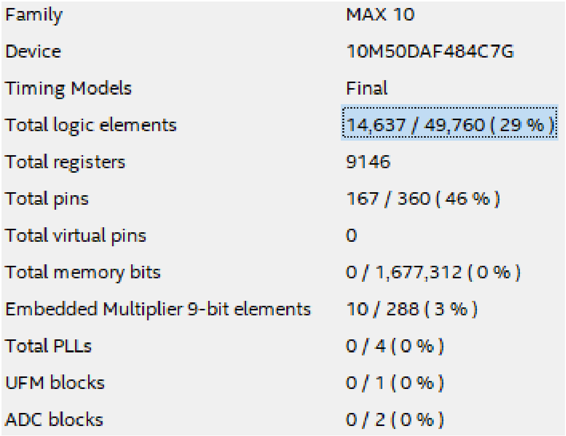


Figure 36 FPGA implementation result for D = 20



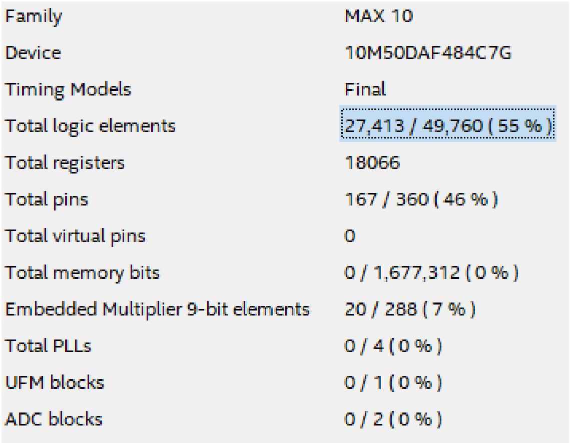
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	8,220 / 49,760 (17 %)
Total registers	4686
Total pins	167 / 360 (46 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	5 / 288 (2 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 37 FPGA implementation chip utilization for D = 5



Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	14,637 / 49,760 (29 %)
Total registers	9146
Total pins	167 / 360 (46 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	10 / 288 (3 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 38 FPGA implementation chip utilization for D = 10



Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	27,413 / 49,760 (55 %)
Total registers	18066
Total pins	167 / 360 (46 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	20 / 288 (7 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 39 FPGA implementation chip utilization for D = 20

4.4 Chip Utilization

Table 2 Comparison of chip utilization

	Agents, N	**Dimensions, D**	**Logical Elements**	**Registers**	**Multiplier 9-bit elements**
FPGA	**50**	**5**	**8220**	**4686**	**5**
	50	**10**	**14637**	**9146**	**10**
	50	**20**	**27413**	**18066**	**20**

For the 5-dimension FPGA implementation, 8220 logical elements, 4686 registers, and 5 units of 9-bit multiplier elements were utilized to synthesize the design. For the 10-dimension FPGA implementation, 14637 logical elements, 9146 registers, and 10 units of 9-bit multiplier elements were utilized to synthesize the design. Lastly, 27413 logical elements, 18066 registers, and 20 units of 9-bit multiplier elements were utilized to synthesize the 20-dimension FPGA implementation. The chip utilization of the 5 dimension, 10 dimension, and 20 dimension designs were tabulated in Table 2.

It can be observed that as the number of dimension of the FPGA implementation increases the resources used to synthesize the design significantly increases as well. The number of registers roughly doubles whenever the number of dimensions is doubled. This is because the multidimensional array X and Y doubles in depth whenever dimension is doubled as observed in Figure 12.

The logical elements utilization increases by 43% when the number of dimensions increases from 5 to 10. Meanwhile, an increment of 46% is observed when the number of dimensions increases from 10 to 20. Extrapolating this information, the 10M50DAF474G chip may only be able to handle a similar design with 50 dimensions before running out of logical elements to successfully synthesize the design.

4.5 Processing Speed

Table 3 Comparison of processing speed

	Agents, N	Dimensions, D	Time taken (s)
Simulation	**50**	**5**	**27.329**
	50	**10**	**36.481**
	50	**20**	**54.571**
FPGA	**50**	**5**	**0.780**
	50	**10**	**0.780**
	50	**20**	**0.780**

The MATLAB simulation of Discrete SKF is ran 3 times using 3 different dimension parameters of D = 5, D = 10, and D = 20. The 5-dimension, 10-dimension and 20-dimension run were completed in 27.329 seconds, 36.481 seconds, and 54.571 seconds respectively.

Meanwhile, the FPGA implementation of Binary SKF was programmed with 3 separate designs for 3 different dimension parameters of D = 5, D = 10, and D = 20. Each run on the FPGA implementation took exactly 0.780 seconds to complete. It can be observed that the 5-dimension, 10-dimension, and 20-dimension run on the FPGA implementation introduced a time reduction by a factor of 35.04, 46.77, and 69.96 respectively compared to the MATLAB simulation. The results of the simulation and FPGA implementation were tabulated on Table 3.

The time taken to complete the simulation on MATLAB increases by 25% when the number of dimensions increased from 5 to 10. Meanwhile increasing the dimensions from 10 to 20 further increased the time taken by 33%. This shows that the time taken to complete the simulation increases as the number of dimensions increases for the MATLAB simulation

However, there is no change in time taken to complete the run on the FPGA implementation. It is observed that the time taken to complete the run remains the same despite increasing the number of dimensions for the FPGA implementation. This is because the modules of the FPGA implementation are designed using the PIPO configuration which enables all dimensions of a single agent to be loaded and processed in a single clock cycle. Consequently, the PIPO configuration of modules will result in the same processing time despite an increment or decrement of number of dimensions.

4.6 Accuracy of Result

Table 4 Comparison of accuracy of result

	Agents, N	Dimensions, D	X_{true}	Expected Value
Simulation	**50**	**5**	**14**	**0**
	50	**10**	**503**	**0**
	50	**20**	**3936**	**0**
FPGA	**50**	**5**	**32**	**0**
	50	**10**	**253**	**0**
	50	**20**	**4749**	**0**

The X_{true} value produced by the MATLAB simulation is 14, 503, and 3936 for the 5-dimension, 10-dimension, and 20-dimension run respectively. None of the X_{true} was able to reach the expected value of 0. It can also be observed that as the number of dimensions increase for the simulation, the X_{true} value strays further away from the expected value.

The output of the FPGA implementation is presented on the seven-segment display of the board which is encoded in Binary Coded Hexadecimal. The output of the FPGA is then decoded and inserted into Table 1 and Table 4. Similar to the MATLAB simulation, the FPGA implementation is not able to achieve the expected value of 0 in any of the runs. The FPGA implementation produced X_{true} values of 32, 253, and 4749 which follows a trend similar to the MATLAB simulation result. As the dimension increases, the X_{true} value strays further away from the expected value. The results of the MATLAB simulation and FPGA implementation are tabulated under Table 4.

The MATLAB simulated Discrete SKF and the FPGA implemented Binary SKF are both unable to produce an accurate result. For the MATLAB simulation, the culprit behind this diminished accuracy is because of the rounding of floating-point values to integers. This reduces the resolution and the accuracy of the results. In addition, a fixed Kalman gain of 0.5 further diminishes the accuracy as it reduces the ability of the metaheuristic to conduct the exploitation process which is the process that enables the metaheuristic to narrow down the solution to the optimal solution. This error is then carried forward to the FPGA implementation which uses binary format to represent the integer values.

4.7 Approximating the Time Taken by FPGA to Complete Run

Table 5 Approximating time taken by FPGA to complete run

Iteration	**1st**	**2nd to 4999th**	**5000th**
States	**s0, s1, s2, s3, s4**	**s2, s3, s4**	**s2, s3, s4, s5**
Clock cycles	**250**	**749,700**	**200**
Total clock cycles per run	**750,150**		
Total clock cycles	**37,507,500**		
Approximated time (s)	**0.750**		
Actual time (s)	**0.780**		

The FSM controls the timing of the implementation which iterates through the 6 states from state s0 through to state s5. In the first iteration, the state machine goes through state s0, s1, s2, s3, and s4. Then from the 2nd to 4999th iteration, the FSM goes through states s2, s3, and S4. Lastly, on the 5000th iteration, the FSM goes through state s2, s3, s4, and completes in s5. The number of clock cycles required to transit from one state to another is roughly N number of clock cycles. In this implementation it would mean 50 clock cycles is required to transit from one state to another. In conclusion, 5 runs at 5000 iterations would require 37,507,500 clock cycles to complete. Since the clock is running at 50MHz, the time taken to complete the run would be 0.750 seconds. This is a close approximation to the actual time taken to complete the run which is 0.780 seconds. The calculation of approximating the time taken is tabulated on Table 5.

CHAPTER 5

CONCLUSION

5.1 Discrete Simulated Kalman Filter

The original SKF metaheuristic was successfully adapted into the Discrete SKF where all of its elements are represented in whole numbers. The adaptation removes all decimal points and fractional part of all elements as well as uses a fixed value Kalman gain of 0.5. This is important because it enables a direct implementation into a digital system where all elements are represented in a binary format.

5.2 Implementation of Binary Simulated Kalman Filter

The Binary SKF was successfully designed using System Verilog to produce a behavioural model of the Discrete SKF. The design was then modified into 3 different variants namely the 5-dimension, 10-dimension, and 20-dimension versions. The designs were then implemented and evaluated based on the performance parameters. The chip cost of the 3 designs were observed to increase as the number of dimensions of the implementation increased. The FPGA implementation was able to increase processing speed by a factor of 35.04, 46.77, and 69.96 for the 5-dimension, 10-dimension, and 20-dimension respectively compared to the MATLAB simulation. In addition, the accuracy of the FPGA implementation of Binary SKF was not able produce accurate results. However, the results follow a trend similar to the Discrete SKF simulation in MATLAB. As the number of dimensions increased, the X_{true} value produced strays further away from the expected value for both Discrete SKF and Binary SKF. Lastly, the configuration of ports of modules in PIPO arrangements enabled processing time to be independent from the number of dimensions.

5.3 Project Limitations

The Discrete SKF is an adaptation of the original SKF metaheuristic which originally operates using floating-point arithmetic. The adaptation removes all floating-point values and converts them into integers. This severely reduces the resolution and accuracy of the result. In addition, the fixed Kalman gain of 0.5 reduces the exploitation performance of the Discrete SKF as it struggles to find the optimum solution.

This adaptation into Discrete SKF is done so that the metaheuristic can be translated into Binary SKF using System Verilog without the implementation of floating-point arithmetic. This would significantly reduce the scope of the project as a significant number of modifications would be required to each module to facilitate arithmetic in floating-point. This includes representing all numbers in all registers of the design to be represented in the IEEE 754 format. The implementation of floating-point arithmetic would amend the problem of diminished accuracy discussed in Chapter 4.6.

5.4 Suggestions and Future Work

5.4.1 Improving Accuracy of Results

To address the project limitations described in Chapter 5.3, it is recommended to implement the floating-point arithmetic module available in the IP Catalog of the Quartus Prime software. The module would be implemented inside the Activation Function module, Measure module, and Estimate module to facilitate all floating-point calculations.

In addition, the availability of floating-point calculations would eliminate the need for a fixed Kalman gain. The original SKF metaheuristic utilizes the error covariance estimate, process noise value, and measurement noise value to calculate the Kalman gain in each iteration which would then be used in the measurement step. This would restore the exploitation ability of the SKF metaheuristic to produce accurate results.

5.4.2 Implementation of Pipeline Structure

Furthermore, the processing speed of the implementation can be further improved by introducing a pipeline structure by tweaking the FSM controller. This implementation should produce a significant increase in processing speed as it removes the waiting time between states of the FSM.

5.4.3 Implementation of CEC2014 Benchmark Functions

To verify the performance of the FPGA implementation, it would be necessary to introduce a method to equally compare the performance of the original SKF metaheuristic and the FPGA implementation. The implementation of all 30 functions in the CEC2014 benchmarking functions would enable the FPGA implementation to be directly compared with the original SKF metaheuristic in terms of accuracy of results as well as processing speed.

5.4.4 Physical Synthesis to Improve Timing and Accuracy of Results

The Quartus Prime software provides users with the ability to set the synthesis effort, physical synthesis effort, and fitter effort to various levels of efforts. The fastest methods were used to implement the designs for this project as it reduces the compilation time of the design. However, this may affect the result of the implementation by reducing the performance of the implementation such as timing errors and other inefficiencies. This can be improved by using a higher effort to compile the designs which would consequently increase the time taken for compilation.

In addition, Quartus Prime includes the ability to use the Chip Planner functionality which allows users to manually do fitting operations. This could enhance the performance of the implementation by improving the timing delays, power consumption, and accuracy of results.

5.4.5 Utilizing Phase-Locked Loop to Increase Clock Frequency

The DE10 – Lite board is powered by an on-board chip that generates a 50MHz clock signal. This clock signal is then fed onto the FPGA and directly affects the processing speed as described in Chapter 4.7. It is possible to use the Phase-Locked Loop (PLL) available on the 10M50DAF484C7G chip to generate an internal clock of 100MHz based on the 50MHz external clock. The application of PLL which is available in the IP Catalog into the design could significantly increase the processing speed of the implementation on FPGA.

REFERENCES

[1] T. Ab Rahman, Z. Ibrahim, N. A. Ab Aziz, S. Zhao, and N. H. Abdul Aziz, "Single-Agent Finite Impulse Response Optimizer for Numerical Optimization Problems," *IEEE Access*, vol. 6, pp. 9358–9374, Jan. 2018, doi: 10.1109/ACCESS.2017.2777894.

[2] M. H. Sulaiman, Z. Mustaffa, M. M. Saari, and H. Daniyal, "Barnacles Mating Optimizer: A new bio-inspired algorithm for solving engineering optimization problems," *Engineering Applications of Artificial Intelligence*, vol. 87, Jan. 2020, doi: 10.1016/j.engappai.2019.103330.

[3] Y. Jiang, Q. Wu, S. Zhu, and L. Zhang, "Orca predation algorithm: A novel bio-inspired algorithm for global optimization problems," *Expert Systems with Applications*, vol. 188, Feb. 2022, doi: 10.1016/j.eswa.2021.116026.

[4] Z. Ibrahim, N. H. Abdul Aziz, N. A. Nor, S. Razali, and M. S. Mohamad, "Simulated Kalman Filter: A novel estimation-based metaheuristic optimization algorithm," *Advanced Science Letters*, vol. 22, no. 10, pp. 2941–2946, Oct. 2016, doi: 10.1166/asl.2016.7083.

[5] N. H. Abdul Aziz *et al.*, "A Tutorial on Single-solution Simulated Kalman Filter," *MEKATRONIKA*, vol. 1, no. 2, pp. 33–44, Jul. 2019, doi: 10.15282/mekatronika.v1i2.4895.

[6] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948. doi: 10.1109/ICNN.1995.488968.

[7] M. A. Khanesar, M. Teshnehlab, M. Aliyari, and S. K. N. Toosi, "A Novel Binary Particle Swarm Optimization," 2007.

[8] H. Zhu Member Yuji Tanabe Non-member Takaaki Baba Non-member Waseda, "A Random Time-Varying Particle Swarm Optimization for the Real Time Location Systems," 2008.

[9] N. A. A. Aziz, Z. Ibrahim, M. Mubin, S. W. Nawawi, and N. H. A. Aziz, "Transitional particle swarm optimization," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 3, pp. 1611–1619, Jun. 2017, doi: 10.11591/ijece.v7i3.pp1611-1619.

[10] M. Kong and P. Tian, "Introducing a Binary Ant Colony Optimization."

[11] M. A. Tawhid and K. B. Dsouza, “Hybrid binary bat enhanced particle swarm optimization algorithm for solving feature selection problems,” *Applied Computing and Informatics*, vol. 16, no. 1–2, pp. 117–136, Apr. 2018, doi: 10.1016/j.aci.2018.04.001.

[12] A. L. X. da Costa, C. A. D. Silva, M. F. Torquato, and M. A. C. Fernandes, “Parallel Implementation of Particle Swarm Optimization on FPGA,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 11, pp. 1875–1879, Nov. 2019, doi: 10.1109/TCSII.2019.2895343.

[13] X. Cai, S. Ngah, H. Zhu, Y. Tanabe, and T. Baba, “Pipeline architecture of particle swarm optimization,” in *Proceedings - 9th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2010*, 2010, pp. 3–8. doi: 10.1109/ICIS.2010.42.

[14] A. Rathod and R. A. Thakker, “FPGA realization of Particle Swarm Optimization algorithm using floating point arithmetic,” Feb. 2015. doi: 10.1109/ICHPCA.2014.7045338.