POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Enhancing Onboard Orbit Propagation: A Deep Dive into FPGA Capabilities

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Davide Giacomini**

Student ID: 965126
Advisor: Dr. Alessandro Morselli
Co-advisor: Gianfranco Di Domenico
Academic Year: 2022-2023

Abstract

Deep-space missions heavily rely on ground stations and human involvement to determine the spacecraft position via radiometric tracking and plan the maneuvers execution to allow the satellite to reach its target orbit. These operations are usually defined as Guidance, Navigation, and Control (GNC). However, this traditional approach presents scalability challenges as the number of deep-space assets increases rapidly. The delays and costs associated with ground control became overwhelming, necessitating a shift towards autonomous GNC operations, where the operations are performed directly on board, hence limiting the need for communications with ground and hence mission costs. In the realm of space exploration and deep-space missions, where precise trajectory predictions are essential, the ability to propagate an object's orbit quickly and accurately becomes indispensable. Rapid and reliable orbit propagation is key for performing precise orbit determination and to predict the spacecraft future trajectory, aiding in navigation and ensuring mission success. Power efficiency is also a vital consideration in space missions, where resources are often scarce. Optimizing the orbit propagation process to consume minimal power is fundamental as the available power is limited and hence an efficient utilization of onboard resources is mandatory. Efforts are therefore focused on developing specialized hardware, such as Field-Programmable Gate Arrays (FPGAs), that can efficiently handle the demanding computations involved in orbit propagation. FPGAs offer the potential for high-speed processing with reduced power consumption, making them well-suited for onboard satellite computing. This thesis uses the PYNQ-Z2 as a showcase to explore the possibilities of orbit propagation acceleration onto FPGAs. Different orbit scenario are analyzed, to validate and estimate the performance of FPGA-based orbit propagation, showing a gain of 76% wrt to CPU-based computation.

Keywords: Deep Space, FPGA, Orbit Propagation, Computational Efficiency, Power Consumption, Hardware, Space Missions

Abstract in lingua italiana

Le missioni nello spazio profondo si basano principalmente su stazioni terrestri e coinvolgimento umano per determinare la posizione del veicolo spaziale tramite il tracciamento radiometrico e pianificare l'esecuzione delle manovre per consentire al satellite di raggiungere la sua orbita di destinazione. Queste operazioni sono generalmente definite come Guida, Navigazione e Controllo (GNC). Tuttavia, questo approccio tradizionale presenta sfide di scalabilità a causa dell'aumento rapido del numero di risorse nello spazio profondo. I ritardi e i costi associati al controllo da terra stanno diventando significativi, rendendo necessaria una transizione verso sistemi GNC autonomi, dove le operazioni vengono svolte direttamente a bordo, limitando quindi la necessità di comunicazioni con la terra e riducendo i costi della missione. Nell'ambito dell'esplorazione spaziale e delle missioni nello spazio profondo, dove la previsione accurata delle traiettorie è fondamentale, diventa indispensabile avere la capacità di propagare rapidamente e accuratamente l'orbita di un oggetto. La propagazione rapida e affidabile dell'orbita è fondamentale per eseguire una determinazione dell'orbita precisa e per predire la futura traiettoria del veicolo spaziale, agevolando la navigazione e garantendo il successo delle missioni. L'efficienza energetica rappresenta anche un fattore cruciale nelle missioni spaziali, dove le risorse sono spesso limitate. L'ottimizzazione del processo di propagazione dell'orbita per ridurre il consumo energetico a bordo è fondamentale. A tal fine, sono in corso diversi sforzi nello sviluppo di hardware specializzato, come le FPGA (Field-Programmable Gate Arrays), che sono in grado di gestire in modo efficiente i calcoli complessi necessari per la propagazione dell'orbita. Le FPGA offrono l'opportunità di elaborare i dati ad alta velocità con un consumo energetico ridotto, rendendole adatte per l'elaborazione a bordo dei satelliti. La tesi in questione fa uso della PYNQ-Z2 come esempio per esplorare le possibilità di accelerazione della propagazione dell'orbita utilizzando le FPGA. Vengono analizzati diversi scenari orbitali per convalidare e stimare le prestazioni della propagazione dell'orbita basata su FPGA, mostrando un guadagno del 76% rispetto al calcolo basato su CPU.

Parole chiave: Spazio Profondo, FPGA, Propagazione dell'Orbita, Efficienza Computazionale, Consumo di Potenza, Hardware, Missioni Spaziali

Contents

1 | Introduction

Deep-space Guidance, Navigation, and Control (GNC) in present times heavily relies on ground-based operations, where ground antennas are required to acquire radiometric observations for orbit determination and powerful workstations are employed for trajectory optimization to determine the most effective propulsion system actuation, necessitating ground stations and human involvement. However, this traditional approach introduces significant delays and costs, posing scalability challenges for space missions. As the number of deep-space assets increases rapidly, reliance on ground control will overwhelm existing facilities. In Figure 1.1, the number of missions planned and launched in the past two decades is depicted[1]. It is evident that the number of missions has been growing

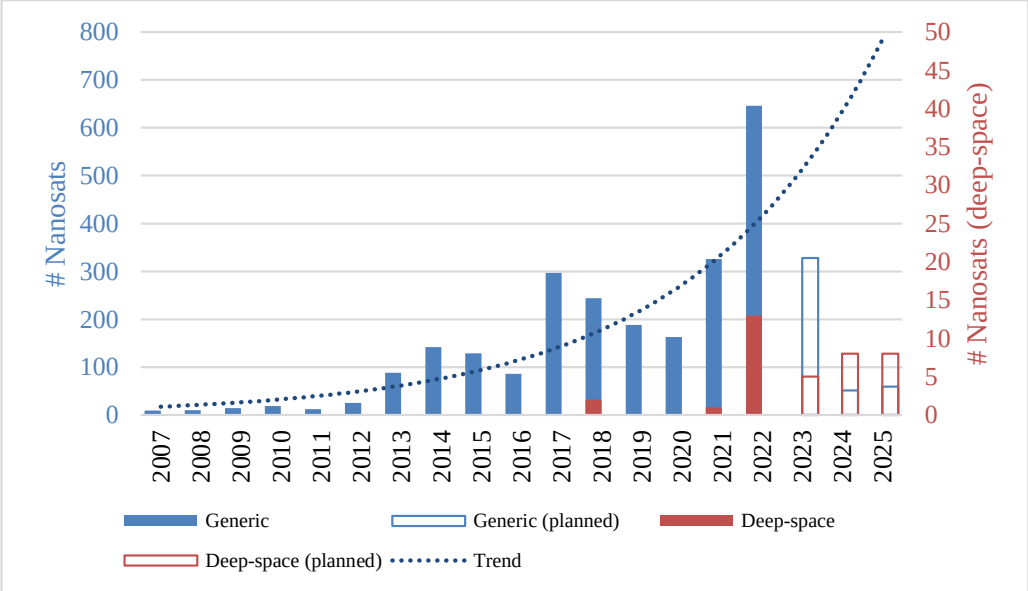


Figure 1.1: Small Satellite missions (<500 kg) past and predicted launches per year [Estimation based on data available on nanosats.eu/tables# (last accessed on July 2, 2023)]

exponentially, with a notable peak in deep-space missions occurring in 2022, mainly due

[1] nanosats.eu/tables#, last accessed: July 2, 2023

to the release of 10 CubeSats via during the NASA mission Artemis I. This increase is mainly due to the deployment of constellations such as OneWeb, Starlink which are taking advantage of these cheaper satellite platforms, which are standardized and suitable for mass production. However, without a shift in approach and the implementation of efficient onboard computation, the capacity for deep-space missions will soon become saturated. To address these limitations and enable the future exploration of deep-space probes, autonomous GNC operations without human intervention are crucial [3].

Autonomous GNC systems require efficient propagation and prediction of the probe's trajectory while consuming minimal energy. Orbit propagation plays a vital role in the Orbit Determination (OD) process, requiring fast and reliable propagation forward in time.

Orbit propagation is the process of propagating an object's state forward in time, based on a mathematical model and on an initial state value. Autonomous spacecraft systems rely on efficient determination and prediction of the probe's trajectory in various contexts.

On-Board Computer (OBC) systems on modern satellites have several key performance criteria that need to be considered. These include memory capacity, software reliability, hardware resilience to the space environment, and low power consumption compared to general computers [31].

Typically, the power budget for CubeSats in Low-Earth-Orbit (LEO) ranges from 2 to 8 $Watts$, which makes integration with FPGAs a challenging task. For example, the Virtex4QV Radiation Tolerant FPGA family's average power consumption ranges from 1.25 to 12.5 $Watts$ [5]. Generally speaking, the IPC-7000 from Space Micro is a specific example of OBC, with a power consumption of $40W$[2].

The objective of this thesis is to study the efficiency of an FPGA for onboard satellite computing in LEO and Deep-Space missions. It aims at comparing against the CPU counterpart computational speed and power consumption in a cost-effective system like the FPGA PYNQ-Z2. Additionally, the thesis explores the complexity of designing optimized systems without writing Hardware Description Language (HDL) code, rather implementing the IP module in High-Level Synthesis (HLS) using C language.

This exploration is guided by the research question highlighted in the box below. Through this investigation, the thesis aims to contribute valuable insights into the development of future autonomous orbit propagation systems on-board of satellites.

[2]satnow.com/products/on-board-computers/space-micro/116-1219-ipc-7000, Last accessed: July 2, 2023

What are the advantages and potential improvements in accuracy and efficiency of orbit propagation in satellites when employing FPGA technology?

1.1. Thesis Organization

In the next chapter of this thesis I will start by giving some introduction about the use of GNC systems on board of spacecrafts, then I will proceed introducing the two-body problem in astrodynamics, describing its mathematical properties. I will also delve into numerical solutions used for mathematical integrations, as they are the main ingredient of orbit propagation. I will conclude with some background information about previous works in similar topics.

Then, I will proceed with the following chapter by describing the FPGA PYNQ-Z2 architecture and its functionalities, and how the Vivado Design Suite IDE can be used to program its Programmable Logic (PL). In the fourth chapter, I will go in details about the work flow for the design, concluding in the fifth chapter with some valid results.

2 | Background

Guidance, Navigation and Control (GNC) is a fundamental aspect for estimating spacecrafts position, predicting their orbit and correcting their trajectory. GNC systems may operate both onboard or from ground stations. Guidance refers to the process of determining the desired path or trajectory for a vehicle or system to achieve a specific objective. It involves making decisions on how the system should move or maneuver to reach a target or follow a desired trajectory. The guidance system provides high-level instructions or commands to steer the vehicle towards the intended goal. Navigation, or Orbit Determination (OD), involves the determination of a vehicle's position, orientation, and velocity in a given reference frame. It is the process of estimating the current state of the vehicle, including its position, velocity, and attitude, with respect to a known reference frame or coordinate system. Navigation systems utilize sensors, such as GNSS, inertial measurement units (IMUs), and other external measurements, in order to determine and update the vehicle's state. Control involves the manipulation of actuators or Reaction Control Systems (RCS) to steer the vehicle and maintain desired attitude. It focuses on the execution of commands generated by the guidance and navigation systems to ensure the vehicle follows the intended trajectory and achieves its objectives [26].

As mentioned in Chapter 1, this thesis focuses on the propagation aspect of OD. OD is fundamental for determining the position and velocity of an object in space, typically a satellite or a spacecraft, based on observations of its position over time and on mathematical models [25]. Once the initial object's state (i.e. position and velocity) is estimated, the orbit propagation involves numerically integrating a mathematical model to predict the object's future state(s) in time. The mathematical model typically includes the gravitational forces acting on the object from celestial bodies (such as the Earth, Moon, and other planets), as well as atmospheric drag, solar radiation pressure, and other relevant perturbations.

To assess the exact location of a probe, both at the beginning of the OD and during the algorithm to make some corrections, some observations need to be done, and are nowadays usually operated from ground stations [3]. In deep-space, Earth-based radiometric

tracking techniques are typically used, in order to provide highly accurate orbit information [35]. Although this technique yields extreme accuracy in position and velocity, it still requires ground stations and human-in-the-loop operations. As mentioned in Chapter 1, this paradigm will soon become unsustainable, and autonomous GNC are the key for the next missions

In the next sections I will introduce a mathematical model used for describing the motion of an object in space, its numerical solutions, and finally some techniques usually employed for orbit propagation.

2.1. Astrodynamics models

Mathematical models are used for reconstructing the initial state vector of a probe and propagating its orbit through time. By iteratively integrating the mathematical models, the probe's trajectory can be estimated with precision as it travels through space.

2.1.1. Two-Body Problem

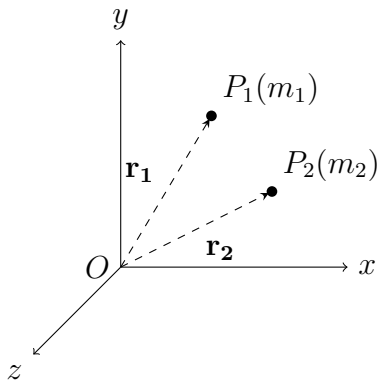


Figure 2.1: Two-body problem illustration

The mathematical model chosen for this work comprises the equations of motion of the two-body problem. The 2-body problem is a fundamental concept in celestial mechanics that involves the gravitational interaction between two bodies, typically a large primary body (such as a planet or star) and a smaller secondary body (such as a satellite or spacecraft). The problem assumes that the masses of the bodies are significant compared to other forces acting upon them, and that the bodies move in isolation without any significant influence from other celestial bodies [25].

In the 2-body problem, the primary and secondary bodies are considered point masses, meaning their sizes and shapes are neglected, and only their masses and positions are

considered. The gravitational force between two bodies is described by Newton's law of universal gravitation:

$$\mathbf{F} = G \cdot \frac{m_1 \cdot m_2}{\|\mathbf{r}\|^3} \cdot \mathbf{r} \tag{2.1}$$

The key objective of the 2-body problem is to determine the motion of the secondary body under the influence of the gravitational force exerted by the primary body. Given Equation 2.1 and the illustration at Figure 2.1, we can derive the equations of motion of the bodies:

$$\begin{cases} m_1 \ddot{\mathbf{r}}_1 = G \dfrac{m_1 m_2}{\|\mathbf{r}_2 - \mathbf{r}_1\|^3} (\mathbf{r}_2 - \mathbf{r}_1) \\ m_2 \ddot{\mathbf{r}}_2 = G \dfrac{m_1 m_2}{\|\mathbf{r}_1 - \mathbf{r}_2\|^3} (\mathbf{r}_1 - \mathbf{r}_2) \end{cases}$$

Defining the relative position $\mathbf{r} := \mathbf{r}_2 - \mathbf{r}_1$ and the gravitational parameter $\mu := G(m_1 + m_2)$, the motion of P_1 with respect to P_2 can be described as:

$$\ddot{\mathbf{r}} + \mu \frac{\mathbf{r}}{\|\mathbf{r}\|^3} = 0 \tag{2.2}$$

Combining Equation 2.2 with a spacecraft state vector $\mathbf{x} = (\mathbf{r}, \mathbf{v})^T$, where $\mathbf{r}$ represents the position vector, and $\mathbf{v}$ represents the velocity vector, we obtain the equations of motion of a satellite:

$$\begin{cases} \dot{\mathbf{r}} = \mathbf{v} \\ \dot{\mathbf{v}} = -\dfrac{\mu}{\|\mathbf{r}\|^3} \mathbf{r} \end{cases} \tag{2.3}$$

Equation 2.3 is a system of first-order Ordinary Differential Equations (ODEs). It models the ideal case, without atmospheric drag, solar radiation pressure, or perturbations from other celestial bodies. These factors may be incorporated into the ODEs to improve the fidelity of the dynamic model.

2.2. First-Order ODEs

A differential equation is an equation that comprises one or more unknown functions, their derivatives and a given function. Solving a differential equation involves finding the unknown function.

In Ordinary Differential Equations (ODEs), the unknown function has only one real or complex independent variable, often denoted as x. The order of the differential equation is defined by the highest derivative order of the equation. Therefore, a first-order ODE

can be generally indicated as:

$$\frac{dy}{dx} = f(x, y)$$

where x is the independent variable, $y(x)$ is the unknown function and $f(x, y)$ is the given function.

To solve a first-order ODE, it is necessary to determine the antiderivative of the unknown function, which involves the process of integration. For example, the solution of:

$$\frac{dy}{dx}(x) = cos(x)$$

is the integral of $cos(x)$:

$$y(x) = sin(x) + C$$

The equations of motion of Equation 2.3 are said to be explicit, because the derivative of the unknown function appears on the left-hand side of the equations, while the functions and their independent variables only appear on the right-hand side.

2.2.1. Initial-Value Problem (IVP)

As seen in the previous example, the solution of a first-order ODEs includes the unknown integral constant C. When an ODE is defined along with an initial condition which specifies the value of the unknown function at a given point in the domain, the exact solution of the ODE can be found. A system with an ODE and an initial condition is called Initial-Value Problem (IVP), and it is in general defined as:

$$\begin{cases} \frac{dy}{dx} = f(y, x) \\ y(x_0) = y_0 \end{cases}$$

If an initial condition is indeed added to the previous example:

$$\begin{cases} \frac{dy}{dx}(x) = cos(x) \\ y(0) = 1 \end{cases}$$

replacing $y(0) = 1$ to the solution we can find the value of C:

$$y(0) = sin(0) + C = 1 \quad \Rightarrow \quad C = 1$$

Therefore, the final solution to the IVP problem of the example is indicated as:

$$y(x) = sin(x) + 1$$

In this example, f was depending only on x and not on $y(x)$, hence the problem could be solved with a simple integration. However, when the integral becomes too complex, approximate numerical solutions are usually employed, such as for example the Runge-Kutta method.

2.3. IVP Numerical Solutions

Initial-value problems can be approximated with iterative solutions, solving the unknown function step-by-step starting from the initial value.

Below, I will briefly address three numerical solutions: the Euler method, the linear multistep method, and lastly the Runge-Kutta method. Single-step methods (such as Euler method) refer to only one previous point and its derivative to determine the next value. Methods such as Runge-Kutta take some intermediate steps (for example, a half-step) to obtain a higher order method, but then discard all previous information before taking a second step. Multistep methods instead attempt to gain efficiency by keeping and using the information from previous steps rather than discarding them. Consequently, multistep methods refer to several previous points and derivative values.

2.3.1. Euler Method

The Euler Method is the easiest and most straightforward method to solve IVPs. The idea behind it is that, for each time step, the derivative of the function is calculated and it is propagated until the next time step, starting from the previous solution. Hence, the error will depend on the length of the time step, as shown at Figure 2.2. The shortest is the time step, the more precise is the approximation.

In the Euler method the next epoch is obtained by the equation:

$$y_{n+1} = y_n + hf(y_n, t_n) \quad \wedge \quad f(y_n, t_n) = \frac{dy}{dt}(t_n)$$

where $h = \Delta t$.

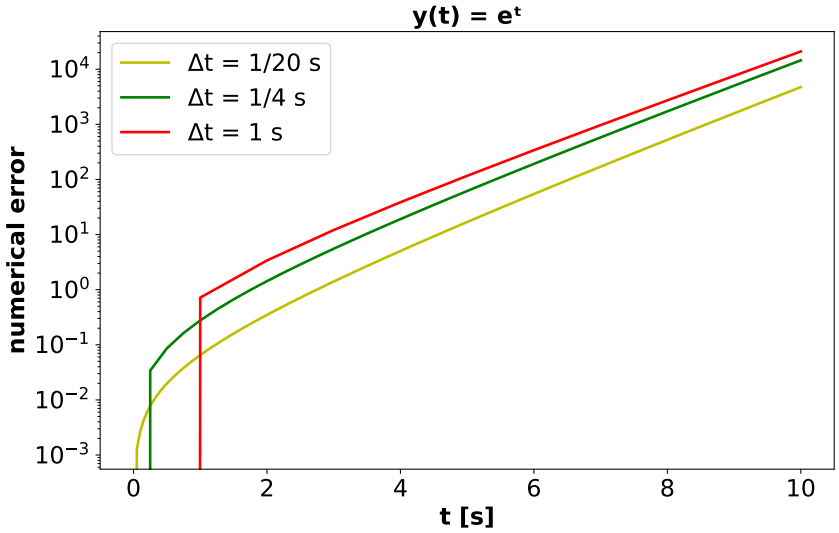


Figure 2.2: Error using the Euler method for `y(t) = e^t`

2.3.2. Linear Multistep Method

As mentioned, the linear multistep method is a linear combination of the previous points and derivative values. An example of linear multistep is the Adams-Moulton method, which equations are reported below:

$$
\begin{aligned}
y_n &= y_{n-1} + hf(y_{n-1}, t_{n-1}) \\
y_{n+1} &= y_n + \frac{1}{2}h\left(f(y_{n+1}, t_{n+1}) + f(y_n, t_n)\right) \\
y_{n+2} &= y_{n+1} + h\left(\frac{5}{12}f(y_{n+2}, t_{n+2}) + \frac{8}{12}f(y_{n+1}, t_{n+1}) - \frac{1}{12}f(y_n, t_n)\right) \\
y_{n+3} &= y_{n+2} + h\left(\frac{9}{24}f(y_{n+3}, t_{n+3}) + \frac{19}{24}f(y_{n+2}, t_{n+2})\right. \\
&\qquad \left. -\frac{5}{24}f(y_{n+1}, t_{n+1}) + \frac{1}{24}f(y_n, t_n)\right) \\
y_{n+4} &= y_{n+3} + h\left(\frac{251}{720}f(y_{n+4}, t_{n+4}) + \frac{646}{720}f(y_{n+3}, t_{n+3})\right. \\
&\qquad \left. -\frac{264}{720}f(y_{n+2}, t_{n+2}) + \frac{106}{720}f(y_{n+1}, t_{n+1}) - \frac{19}{720}f(y_n, t_n)\right)
\end{aligned}
$$

where, as before, $h = \Delta t$ and $f(y_k, t_k) = \frac{dy}{dt}(t_n)$.

2.3.3. Runge-Kutta Methods

The Runge-Kutta method achieves higher accuracy compared to simpler methods by considering multiple intermediate steps within each interval. It is known as a "multi-stage" method because it evaluates the derivative function at various points within a single step to estimate the slope and update the solution accordingly.

The most commonly used variant of the Runge-Kutta method is the fourth-order Runge-Kutta (RK4) method. The RK4 method uses four intermediate stages to calculate the solution at each interval. However, there are other variants such as the second-order Runge-Kutta (RK2) method and the higher-order Runge-Kutta methods [9].

It is important to note that the Runge-Kutta method is a numerical approximation technique, and the accuracy of the solution depends on the step size chosen and the complexity of the ODE being solved. Selecting an appropriate step size and considering error control mechanisms can help ensure accurate and reliable results.

In Runge-Kutta, the next epoch is obtained by:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \tag{2.4}$$

where the k_i elements are the intermediate stages, and they are evaluated as:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + c_2 h, y_n + (a_{21} k_1) h) \\ k_3 &= f(t_n + c_3 h, y_n + (a_{31} k_1 + a_{32} k_2) h) \\ &\vdots \\ k_s &= f(t_n + c_s h, y_n + (a_{s1} k_1 + a_{s2} k_2 + \ldots + a_{s,s-1} k_{s-1}) h) \end{aligned} \tag{2.5}$$

The constants b_i, a_i and c_i presented in the Equations 2.4 and 2.5 are coefficients defined in the Butcher tableau. Depending on the precision and the order of the algorithm, the coefficients of the Butcher tableau may be different. The c-values, also known as the stage time nodes, determine the internal sub-intervals within the main integration interval. The a-values indicate the weights or contributions of the derivative evaluations at different stages. The b-values represent the contribution of each k_i stage to the final solution update. As an example, the Butcher tableau of the classic fourth-order Runge-Kutta (RK4) method is reported at Table 2.1.

$\mathbf{c_i}$	$\mathbf{a_{ij}}$			$\mathbf{b_i}$
0				$\frac{1}{6}$
$\frac{1}{2}$	$\frac{1}{2}$			$\frac{1}{3}$
$\frac{1}{2}$	0	$\frac{1}{2}$		$\frac{1}{3}$
1	0	0	1	$\frac{1}{6}$

Table 2.1: Butcher tableau of RK4

Adaptive Runge-Kutta Methods

Adaptive methods are advanced numerical approaches that dynamically adjust the step size used in the numerical integration process to achieve a balance between accuracy and computational efficiency. Unlike fixed-step methods, where the step size remains constant throughout the integration, adaptive methods modify the step size as needed based on the local behavior of the solution.

The primary motivation behind adaptive methods is to handle situations where the solution of the ODE exhibits varying behaviors across different regions of the integration interval. In some regions, the solution may vary rapidly, requiring smaller steps to capture the fine details accurately. In other regions, the solution may change slowly, allowing for larger steps without sacrificing accuracy.

Adaptive methods employ an error estimation technique to monitor the accuracy of the numerical solution. The error estimation is typically based on the difference between two approximations obtained using different step sizes. By comparing the local error estimate to a predefined tolerance, the adaptive algorithm determines whether the step size should be increased or decreased.

Implementing an adaptive method involves additional computational overhead due to the error estimation and step size control mechanisms. However, the benefits of improved accuracy and efficiency often outweigh the additional computational cost, especially for problems with complex and rapidly changing solutions.

In adaptive methods, two different orders of the Runge-Kutta method denoted as p and q ($q > p$) are calculated, typically with $q = p + 1$. To ensure computational efficiency, the two orders share the same function evaluations, indicated by the coefficients a_{ij} [13]. This approach allows for an efficient calculation of the error without incurring significant additional computational cost. Runge-Kutta $q(p)$ methods use the higher order as primal and the lower order only to compute the Local Truncation Error (LTE), instead Runge-

Kutta $p(q)$ methods use the higher order to compute the LTE.

The order step calculation for determining the LTE is given by:

$$\hat{y}_{n+1} = y_n + h \sum_{i=1}^{s} \hat{b}_i k_i$$

Then, the LTE of the order scheme is calculated as:

$$e_{n+1} = y_{n+1} - \hat{y}_{n+1} = h \sum_{i=1}^{s} (b_i - \hat{b}_i) k_i$$

The optimal step size, which is a step size that allows the tolerance to be respected while avoiding too many computational steps, can be determined using the Runge-Kutta orders and the calculated error between order p and q:

$$h_{n+1} = 0.9 h_n \left[\frac{\delta}{\|e_{n+1}\|} \right]^{\frac{1}{p+1}} \tag{2.6}$$

where δ is the maximum allowable local error.

For this thesis, the Butcher tableau of RK5(4)7M has been used [13], which is also used in the Matlab function `ode45` [33]. The corresponding tableau is reported in Table 2.2.

$\mathbf{c_i}$	$\mathbf{a_{ij}}$						$\mathbf{b_i}$	$\mathbf{b^*_i}$
0							$\frac{35}{384}$	$\frac{5179}{57600}$
$\frac{1}{5}$	$\frac{1}{5}$						0	0
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					$\frac{500}{1113}$	$\frac{7571}{16695}$
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				$\frac{125}{192}$	$\frac{393}{640}$
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			$-\frac{2187}{6784}$	$-\frac{92097}{339200}$
1	$-\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		$\frac{11}{84}$	$\frac{187}{2100}$
1	$\frac{35}{84}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	$\frac{1}{40}$

Table 2.2: Butcher tableau of RK5(4)7M

2.4. State Estimation Techniques

In the field of autonomous navigation, the Kalman filter and the batch filter are powerful estimation tools that have been extensively used in navigation of robotic systems operating

in unknown environments [26]. Both the batch filter and the Kalman filter are two methods that can be applied to nonlinear systems. The main difference to keep in mind between batch filter and Kalman filter is that the former is non-recursive and the latter is sequential.

2.4.1. Kalman Filter

The Kalman filter [22] is a powerful recursive algorithm that serves as an effective tool for the fusion of information from a measurement system with the dynamics of a model. This fusion process results in a more precise estimation of the system state than what could be achieved by relying solely on the measurements or dynamics. The measurement system, the dynamics model, and the fusion algorithms selected within the filter each play a crucial role in shaping the final navigation solution.

Moreover, the standard Extended Kalman filter (EKF) operates under the assumption of fixed uncertainties in stochastic error parameters within the dynamics and measurement models. These uncertainties are known to propagate through the filter, influencing the accuracy of the output. However, the impact of these uncertainties can typically be mitigated to a reasonable extent through the careful process of tuning the EKF. This tuning process involves adjusting the parameters of the filter to optimize its performance, thereby enhancing the reliability and accuracy of the navigation solution it provides.

In essence, the Kalman filter is a sophisticated tool that leverages the power of recursive algorithms and information fusion to deliver accurate system state estimations. Its effectiveness, however, is contingent on the quality of the measurement system, the dynamics model, and the chosen fusion algorithms, as well as the successful mitigation of uncertainties through the tuning of the EKF.

The efficacy of the Kalman filter is demonstrated by advanced GNC systems used in recent missions [15], such as PRISMA [11], CanX-4/5 [8], and AVANTI [16].

The Unscented Transformation (UT) is a crucial component in the implementation of Kalman Filters, specifically the Unscented Kalman Filter (UKF). The UKF leverages the UT to predict means and covariances in nonlinear systems, which is particularly useful in on-board orbit reconstruction. This process is essential in various applications, such as relative dynamics and orbit determination. This method uses a set of weighted sigma points that match the properties of the prior distribution. This approach resembles a Monte Carlo method but does not use random sampling, requiring only a small number of points [21].

2.4.2. Batch Filter

Batch filters provide a robust method for estimating the state of a system at a specific epoch using a set of measurement data. The batch filter works by minimizing the sum of the squares of measurement residuals. As for the Kalman filter, it adjusts the estimated state to minimize the difference between the predicted and actual measurements. This process often involves iterative methods and requires a good initial estimate to ensure convergence.

Instead of what it is done by the Kalman filter, the batch filter considers all the measurements and not only the last one. This yields to more precise but also much more expensive computations [27].

The work in [27] discusses two types of batch filters: the batch least squares filter and the non-recursive unscented batch filter. The batch least squares filter is a traditional method that has been widely used in orbit determination. However, the non-recursive unscented batch filter can be more stable and may require fewer iterations for convergence, especially when dealing with highly nonlinear systems or when considering large initial errors, long sampling periods, and large measurement noises. The unscented batch filter has the advantage of being able to handle strong nonlinearity better than the traditional batch least squares filter. This is particularly important in the context of satellite orbit determination, where the motion of the satellite is influenced by various nonlinear factors such as gravitational forces, atmospheric drag, and solar radiation pressure. However, the unscented batch filter requires a good initial estimate to ensure convergence. Moreover, it can be computationally intensive, especially for large systems.

2.5. Onboard GNC Solutions

One of the first onboard autonomous navigation systems (AutoNav) implemented in deep-space was for the Deep-Space 1 (DS1) mission. Launched by NASA in 1998, DS1 was a technology demonstration mission aimed at testing advanced technologies for future space exploration. The AutoNav system on DS1 utilized a combination of imaging and image processing techniques to autonomously navigate and guide the spacecraft. It employed a technique known as "image matching," where the spacecraft compared the real-time images of celestial objects with the pre-stored reference images to determine its position and orientation [6, 7].

[41] introduces an innovative architecture for a robust and reconfigurable FPGA-based computer, specifically designed for use in critical Guidance, Navigation, and Control

(GNC) systems. The architecture is engineered to counteract Single Event Upsets (SEUs), a common source of malfunction in space applications. The architecture employs a combination of multi-layer reconfiguration and multi-layer Triple Modular Redundancy (TMR) techniques, providing a comprehensive solution to SEU-related issues. The architecture has been successfully implemented using FPGA technology and has been integrated into the GNC system of a circumlunar return and reentry flight vehicle, demonstrating its practical applicability. The study does not focus on a specific piece of hardware but rather emphasizes the broad application of FPGA-based computers within GNC systems. The research underscores the potential of such systems in enhancing the reliability and adaptability of deep space exploration missions.

In the context of autonomous spacecraft operations, the work of Hao et al. [19] presents a significant advancement. They propose an intelligent Guidance, Navigation, and Control (GNC) system that incorporates state-of-the-art AI components for on-orbit manipulation. The system leverages a monocular camera and a keypoint-based Deep Learning (DL) framework for relative navigation, a critical aspect for autonomous close proximity operations in orbital robotic missions. Furthermore, the system includes onboard computational hardware suitable for Computer Vision (CV), underscoring the potential for using onboard technology such as Field-Programmable Gate Arrays (FPGAs) or microcontrollers in space applications. This research highlights the potential of integrating advanced AI components and onboard technology to enhance the autonomy and efficiency of GNC systems for deep space exploration.

[34] discusses the process of orbit propagation and determination for Low Earth Orbit (LEO) satellites. The paper emphasizes the importance of accurate orbit determination, which is achieved through the analysis of position and velocity measurements from a space-based GPS receiver located on the satellite itself. The paper also highlights the use of a nonlinear filtering method for immediate orbit tasks, which requires more precise satellite orbit state parameters in a short time.

Segret et al. [32] discuss the development of an autonomous Guidance, Navigation, and Control (GNC) technology called BIRDY-T for a CubeSat. The authors present an On-Board Orbit Determination (OD) system that uses asynchronous optical measurements for triangulation, which then feeds into a Kalman filter. The system is designed to handle various mission profiles, including cruise and proximity operations. The authors also discuss the importance of a tailor-made ground segment, developed with experts in deep space dynamics and on-board software architecture, for mission preparation. The study emphasizes the potential of using advanced algorithms and onboard technology, such as FPGAs or microcontrollers, for autonomous OD in deep space missions.

[1] demonstrates the practical application of advanced algorithms and hardware acceleration in the field of space exploration. Specifically, the use of the SSRLCV library for 3D terrain information generation, and the implementation of this library on the MOCI CubeSat equipped with a Nvidia TX2i GPU, exemplifies the potential of using advanced algorithms for autonomous onboard execution.

The work of Pitonak et al. [28] presents the development of an FPGA-based hardware-accelerated quantized Convolutional Neural Network (CNN) for on-board cloud cover classification. The authors propose a workflow that includes training the classification model, observing the impact of quantization on model accuracy, and exploring hardware architecture design space to identify configurations with the highest throughput and minimal FPGA resource utilization. The proposed method, CloudSatNet-1, is implemented on FPGA and demonstrates high accuracy and low false-positive rate in cloud cover classification.

2.6. Onboard Processing for Spacecrafts

In the context of developing future spacecrafts, especially SmallSat platforms for deep-space missions, the challenge lies in balancing the demand for onboard sensors and science processing with the constraints of reduced power, size, weight, and cost [17]. The harsh space environment necessitates the use of radiation-hardened (rad-hard) hardware, which is often costly. Companies like Xilinx have been producing rad-hard FPGAs for space applications, such as the RT Kintex UltraScale FPGA, the Virtex 5QV FPGA, and the Virtex 4QV FPGA[1]. However, rad-hard processors often cannot match the computational capabilities of common processors due to their inherent redundancy and other protective features [18]. The paper by George and Wilson [17] presents a solution in the form of hybrid computing, which combines rad-hard devices and Commercial Off-The-Shelf (COTS) devices. This approach aims to achieve both high reliability and performance, making it particularly relevant for the goal of accelerating the orbit determination (OD) algorithm with an FPGA for autonomous execution onboard a spacecraft.

In the challenging environment of space, electronic components encounter particles from various sources such as Earth's magnetic field, galactic cosmic rays, and solar-weather events. The interaction of these particles with electronics can lead to two broad categories of effects: long-term cumulative effects and short-term transient effects. Cumulative effects encompass phenomena like Total Ionizing Dose (TID) levels, circuit ionization, enhanced Low-Dose-Rate Sensitivity (ELDRS), and Displacement-Damage Dose (DDD).

[1] xilinx.com/applications/aerospace-and-defense/space.html#products

On the other hand, transient effects, often referred to as Single-Event Effects (SEEs), include Single-Event Upsets (SEUs), Single-Event Transients (SETs), Single-Event Latchups (SELs), Single-Event Functional Interrupts (SEFIs), and others. These effects are primarily due to undesired alterations in voltage levels, consequently affecting current flow in electronic components [14]. The NASA Electronic Parts and Packaging (NEPP) program provides guidance for understanding and mitigating these radiation effects, emphasizing the importance of radiation hardness assurance (RHA) for reliable space system design [17].

In the realm of custom circuit devices, there exists a balance to be struck between the use of high-cost, cutting-edge devices known as Application-Specific Integrated Circuits (ASICs) and the use of less efficient, cost-effective devices known as Commercial Off-The-Shelf (COTS) products. A promising solution lies in the utilization of reconfigurable devices, which are characterized by their adaptive designs that can be programmed to create a variety of architectures and circuits. Field-Programmable Gate Arrays (FPGAs) are the most commonly associated with reconfigurable computing. These devices offer several advantages over general-purpose CPUs or microprocessors. Firstly, they allow designers to create custom, application-specific architectures that can exploit algorithmic parallelism. Secondly, FPGAs are typically more energy-efficient than general-purpose processors, enabling designers to achieve significant computational speedup on an application while consuming less energy. Furthermore, they are able to apply a circuit reconfiguration at runtime by applying the so-called partial reconfiguration, which allows the device to adapt to the best possible solution [24].

An hybrid system are often the most suitable for complex applications. For instance, a GPU, which excels in graphical computations, is often paired with a CPU for more general computations. Similarly, an FPGA is typically interfaced with a host device, often a CPU, to handle tasks that are less suited for parallelism or pipelining, or tasks for which the FPGA has not been configured. This is particularly relevant in the context of Xilinx's Zynq FPGAs, a family of devices that integrate both a CPU and an FPGA in the same System on a Chip (SoC), primarily interfaced through an AXI interface. These hybrid systems, like the Xilinx Zynq-7020 SoC, combine fixed (dual ARM Cortex-A9/NEON cores) and reconfigurable (Artix-7 FPGA fabric) logic[2].

[2]xilinx.com/products/silicon-devices/soc/zynq-7000.html

2.7. Uncertainties Propagation (Monte Carlo)

The Monte Carlo method is a powerful statistical technique that enables numerical solutions to complex problems through random sampling [23]. The Monte Carlo method plays a crucial role in simulating uncertainties associated with the initial condition.

The first step entails framing the problem in probabilistic terms. After OD is performed, the covariance (uncertainty) associated to the navigation solution is obtained and sampled to generate N initial conditions. Subsequently, each initial condition gets propagated forward in time for each set of random inputs using deterministic physics, solving the equations of motion mentioned in Section 2.1. The outcomes of the deterministic calculations for each set of random inputs are subsequently analyzed to provide a probabilistic description of the spacecraft's future state.

In the environmental modelling process, Monte Carlo analysis is one of the methods commonly used in uncertainty assessment and characterisation. It is applicable for different purposes of application, stages of the modelling process, and sources and types of uncertainty addressed. This method is particularly useful when dealing with complex systems with multiple sources of uncertainty [29].

In the field of dark matter indirect detection, Monte Carlo methods are used to compute signals of TeV-scale Dark Matter annihilations and decays in the Galaxy and beyond. The method is used to estimate uncertainties in the computation of energy spectra of various particles [10].

In risk assessment, an Advanced Monte Carlo Method based on interval analysis approach and Monte Carlo simulation is proposed to propagate uncertainties in an atmospheric dispersion model. The purpose is to compute with accuracy the geographical region in which the concentration of the considered toxic gas is less than the threshold of irreversible effects [30].

Finally, in [12] the Monte Carlo method is used in the context of testing the proposed method for initial orbit determination and subsequent follow-on tracking. Two test cases have been considered: a single orbit case with a Monte Carlo sampling of the measurement data, and a grid of multiple orbits with a single set of data per orbit.

2.8. Adimensionalization

Adimensionalization involves transforming the variables and parameters of a system of equations into dimensionless quantities. This can help to reduce the number of parameters

in the system, simplify the analysis, and reveal scaling laws and similarities between different systems.

In case memory constraints are present in the design, adimensionalization can be used to optimize the number of bits necessary for representing a value.

In this case, the adimensionalization involves defining two constants: length L and time T. Considering the equations of motion (Equation 2.3), the two constants can be initialized as:

$$\begin{cases} \dfrac{L^3}{T^2} = \mu \\ \dfrac{L}{T} = \|\mathbf{v_0}\| \end{cases}$$

Once chosen the parameters length and time, each value must be transformed according to the new normalization:

$$\begin{cases} \mathbf{r} = L \cdot \mathbf{r}^* \\ \mathbf{v} = \dfrac{L}{T} \cdot \mathbf{v}^* \\ \mu = \dfrac{L^3}{T^2} \cdot \mu^* \\ h = T \cdot h^* \end{cases}$$

If chosen accordingly to these rules, the normalization factors L and T cancel out in the implementation of Runge Kutta. For example, when calculating an intermediate stage $\mathbf{k_i} = f(\mathbf{x} + h\sum_j^{i-1} a_{ij}\mathbf{k_j})$, the normalized transformation would be:

$$\begin{cases} \dfrac{L}{T}\mathbf{k}^*_{\mathbf{r_i}} = \dfrac{L}{T}\mathbf{v}^* + (Th^*)\left(\dfrac{L}{T^2}\displaystyle\sum_j^{i-1} a_{ij}\mathbf{k}^*_{\mathbf{v_j}}\right) \\ \dfrac{L}{T^2}\mathbf{k}^*_{\mathbf{v_i}} = -\left(\dfrac{L^3}{T^2}\mu^*\right)\dfrac{L\mathbf{r}^* + (Th^*)\left(\frac{L}{T}\sum_j^{i-1} a_{ij}\mathbf{k}^*_{\mathbf{r_j}}\right)}{\left\|L\mathbf{r}^* + (Th^*)\left(\frac{L}{T}\sum_j^{i-1} a_{ij}\mathbf{k}^*_{\mathbf{r_j}}\right)\right\|^3} \end{cases}$$

It is straightforward to see that the normalized factors cancel out. This property is proved in any passage of the algorithm, hence, adimensionalized inputs will not affect the outputs.

3 | FPGA Overview

In the ever-evolving landscape of digital system design, the demand for flexible and efficient solutions has grown exponentially. Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling option, offering unparalleled versatility and performance compared to traditional fixed-function integrated circuits.

FPGAs present a unique proposition by allowing users to dynamically configure the behavior of digital circuits, even after the manufacturing process. This characteristic grants designers the ability to adapt and modify their circuits, making FPGAs suitable for a wide range of applications that require flexibility and customization. By leveraging configurable logic blocks interconnected by programmable routing channels, FPGAs enable the realization of complex digital systems, ranging from small-scale embedded designs to large-scale computational architectures.

One of the primary advantages of FPGAs lies in their ability to achieve high-performance computing through parallel processing. By harnessing the parallelism of multiple logic blocks, FPGAs can execute tasks in a concurrent and efficient manner, thereby accelerating computationally intensive operations. This makes FPGAs an appealing choice for applications that demand real-time processing, high-speed data handling, and hardware acceleration.

The programming or configuration of FPGAs is typically accomplished using hardware description languages (HDLs) such as VHDL or Verilog. These languages provide a means to describe the desired behavior of the digital circuit in a textual or schematic form. With the synthesis of HDL code into a configuration bitstream, the FPGA can be programmed to execute the specified functionality, further enhancing its adaptability and versatility.

As FPGA technology continues to advance, it has become more accessible to a broader range of engineers and developers. Integrated development environments (IDEs) and higher-level synthesis tools have simplified the design process, allowing users to work at higher levels of abstraction. This shift enables students, researchers, and practitioners to explore and exploit the capabilities of FPGAs without being burdened by the intricacies of low-level hardware design.

3.1. Xilinx Software

Xilinx offers a comprehensive suite of software tools and development environments that complement their programmable logic devices. These software tools are designed to enable efficient and streamlined FPGA development, helping designers unleash the full potential of Xilinx devices.

One of the key software offerings from Xilinx is the Vivado Design Suite. Vivado provides a complete development ecosystem for designing, implementing, and optimizing FPGA designs. It includes various modules and features such as IP integrator, high-level synthesis, system-level design, and advanced verification capabilities. Vivado allows designers to work at different levels of abstraction, from algorithmic design to register-transfer level (RTL) coding, making the development process more intuitive and efficient.

Xilinx Vitis High-Level Synthesis (HLS) is a powerful tool that enables designers to accelerate their development process by raising the level of abstraction. Vitis HLS allows designers to write algorithms and complex data processing tasks in high-level languages such as C, C++, or OpenCL, and then automatically transforms them into highly optimized hardware designs for Xilinx FPGAs. It provides a range of advanced features to assist in the optimization process, such as directive-based pragmas, dataflow control, pipelining, loop unrolling, resource sharing, and memory optimization techniques. These features enable designers to explore different architectural trade-offs and fine-tune their designs to achieve the desired performance, area, and power efficiency. By leveraging Xilinx Vitis HLS, designers can significantly reduce development time and effort, accelerate time-to-market, and achieve high-performance implementations of complex algorithms on Xilinx FPGAs. It bridges the gap between software and hardware design, enabling a more efficient and productive development experience for FPGA-based systems.

3.2. Xilinx Products

Xilinx offers a wide range of Field-Programmable Gate Arrays (FPGAs) that cater to diverse application requirements. Here is an overview of some of the key Xilinx FPGA product families:

- Virtex UltraScale+: The Virtex UltraScale+ family is Xilinx's flagship FPGA product line. These FPGAs deliver exceptional performance and integration capabilities, making them suitable for high-end applications such as data centers, networking, and Advanced Driver-Assistance Systems (ADAS). They offer a scalable architecture with advanced DSP, memory, and connectivity resources.

- Kintex UltraScale+: The Kintex UltraScale+ FPGAs provide a balance of performance, power efficiency, and cost-effectiveness. They are designed for a wide range of applications, including wireless communications, video processing, and industrial automation. These FPGAs offer high-speed serial transceivers, ample Digital Signal Processing (DSP) resources, and on-chip memory options.
- Artix-7: The Artix-7 family offers a cost-optimized FPGA solution ideal for applications that require moderate performance and low power consumption. These FPGAs are suitable for applications such as motor control, medical devices, and consumer electronics. Artix-7 devices provide a good balance of capabilities and affordability, making them popular for cost-sensitive projects.
- Spartan-7: The Spartan-7 family is focused on providing low-cost, entry-level FPGAs with sufficient performance for a range of applications. These FPGAs are suitable for IoT devices, automotive electronics, and consumer applications. Spartan-7 devices offer a good combination of low power consumption, small form factor, and ease of integration.
- Zynq UltraScale+: The Zynq UltraScale+ family combines FPGA fabric with high-performance ARM processors in a single device. These devices are known as System-on-Chip (SoC) FPGAs and offer the benefits of both programmable logic and embedded processing. Zynq UltraScale+ devices are suitable for applications that require high flexibility and real-time processing, such as embedded vision, industrial control, and automotive systems.

3.2.1. Xilinx PYNQ-Z2 Overview

The Xilinx PYNQ-Z2 is an embedded development board designed for applications in the field of embedded systems and programmable logic. It combines the power of Xilinx's Zynq-7000 SoC (System-on-Chip) with the Python programming language and an accessible development environment.

At the core of the PYNQ-Z2 board lies the Xilinx Zynq-7000 SoC, which integrates a dual-core ARM Cortex-A9 processor and an FPGA fabric. This combination allows developers to leverage the flexibility and performance of FPGA-based hardware acceleration alongside the convenience of a traditional processor.

One of the standout features of the PYNQ-Z2 board is its support for the PYNQ (Python + Zynq) framework. PYNQ enables users to rapidly design and deploy embedded systems using Python, a widely adopted and user-friendly programming language. With PYNQ,

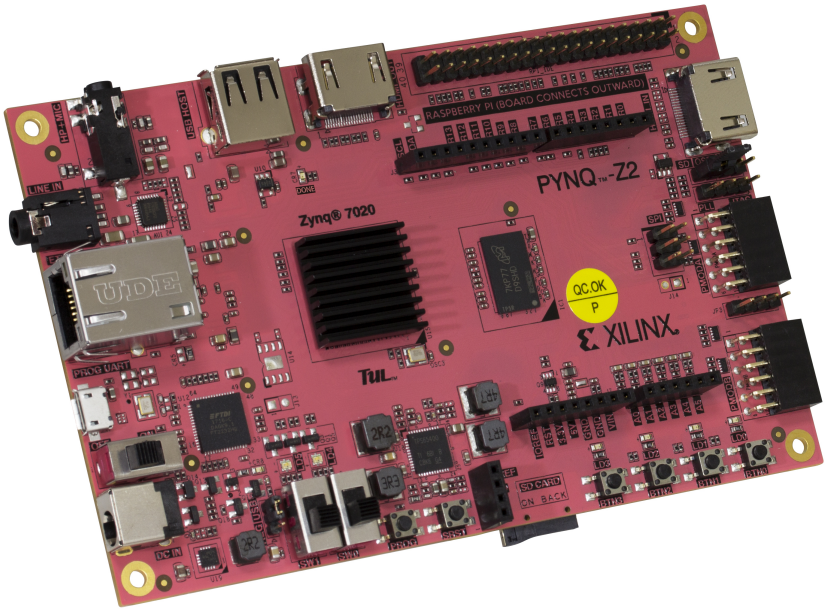


Figure 3.1: PYNQ-Z2 board [Resource from tulembedded.com/FPGA/ProductsPYNQ-Z2.html (Last accessed: July 2, 2023)]

developers can take advantage of pre-built overlays and libraries, making it easier to develop and accelerate applications with FPGA-based hardware.

The PYNQ-Z2 board provides a range of peripheral interfaces, including HDMI, USB, Ethernet, and GPIOs, making it versatile and suitable for a wide range of projects. It also offers expansion connectors, such as Arduino and Raspberry Pi headers, allowing for compatibility with various add-on boards and modules.

The development workflow for the PYNQ-Z2 board involves using the PYNQ framework, which provides Jupyter notebooks as an interactive development environment. These notebooks allow developers to write and execute Python code while accessing the underlying FPGA fabric and peripherals of the board. This streamlined development process enables rapid prototyping, debugging, and system integration.

The PYNQ-Z2 board offers an exceptional value proposition, being remarkably affordable with a price tag of approximately $150[1]. This affordable price, however, does come with certain resource limitations. Despite these constraints, demonstrating the applicability of my work on this FPGA serves as a strong foundation, especially when considering future missions that may have a more generous budget. By showcasing the effectiveness of my work on a cost-effective platform like the PYNQ-Z2, it highlights the potential for even greater achievements and possibilities when equipped with additional resources

[1] newark.com/search?st=tul-corporation, last accessed: July 2, 2023

and funding. This approach serves as a prudent and strategic starting point, allowing for scalability and adaptability as projects progress and budgets expand.

3.2.2. PYNQ-Z2 Architecture

The PYNQ-Z2 is divided into Processing System (PS) and Programmable Logic (PL) (Figure 3.2). The former refers to the ARM-based processing unit integrated into the

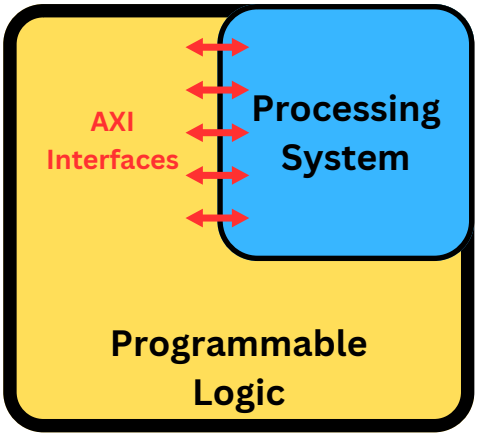


Figure 3.2: PYNQ-Z2 architecture overview

Zynq-7000 SoC. It consists of dual-core ARM Cortex-A9 processors running at a specified clock frequency. The processing system handles the execution of software tasks, such as running operating systems (e.g., Linux) and applications. It manages system resources, interfaces with external peripherals, and provides a familiar programming environment for software development. The PL refers to the FPGA fabric within the Zynq-7000 SoC (Artix-7 FPGA). The FPGA fabric can be programmed using Hardware Description Languages (HDLs) or High-Level Synthesis (HLS) tools, enabling hardware customization and acceleration for specific applications.

PS-PL Interface

Usually, one of the starting points when programming an FPGA is to manage the interface Host-FPGA. FPGAs often need to be paired with a Host, either for performing more general computations or even just to manage the control flow of the FPGA used. As already mentioned in previous paragraphs, the PYNQ-Z2 is a System on Chip (SoC) and the paradigm Host-FPGA is already embedded into the board.

In order to connect CPU and FPGA, this SoC employs employs the Advanced eXtensible Interface protocol (AXI), which is very common in ARM systems. There are three types of AXI Interfaces that connect the FPGA to the CPU [39]:

- AXI Coherent Port (ACP): This interface is used for coherent access to the PS's L2 cache and DDR memory. It operates asynchronously and has a data width of 64 bits.
- AXI General Purpose (GP): These interfaces connect slave peripherals to the PS and are used for general-purpose communication between the PS and PL. They operate asynchronously and have a data width of 32 bits.
- AXI High Performance (HP): These interfaces are used for high-performance data transfer between the PS and PL. They operate asynchronously and can have a data width of 32 or 64 bits.

Figure 3.3 reports the block diagram of the interconnections of the Zynq PS, which is also shown in [39]. The AXI GP interfaces are directly linked to the ports of the master interconnect and the slave interconnect, without any extra FIFO buffering. Unlike the AXI HP interfaces, which utilize advanced FIFO buffering to enhance performance and throughput, these interfaces do not have that feature. As a result, the performance of these interfaces is determined by the capabilities of the master and slave interconnect ports. It is important to note that these interfaces are primarily intended for general-purpose use and are not designed to achieve high performance.

The four AXI HP interfaces are designed to offer high-speed data paths to the DDR and OCM memories for PL bus masters. Each interface consists of two FIFO buffers, one for reading and one for writing traffic. The PL to memory interconnect is responsible for connecting the AXI HP ports to either two DDR memory ports or the OCM. The AXI HP interfaces are also commonly referred to as AFI (AXI FIFO interface), highlighting their buffering capabilities.

One of the two AXI GP ports has been used for control signals, and two of the HP ports for high-performance memory-mapped interconnection. Using the AXI Memory-Mapped protocol for the AXI HP ports, bursts can be used to smooth out long-latency transfers (more details about bursts in Section 3.4.1). To avoid stalls during the transactions, this architecture comprises FIFO buffers of 1kB for both reads and writes.

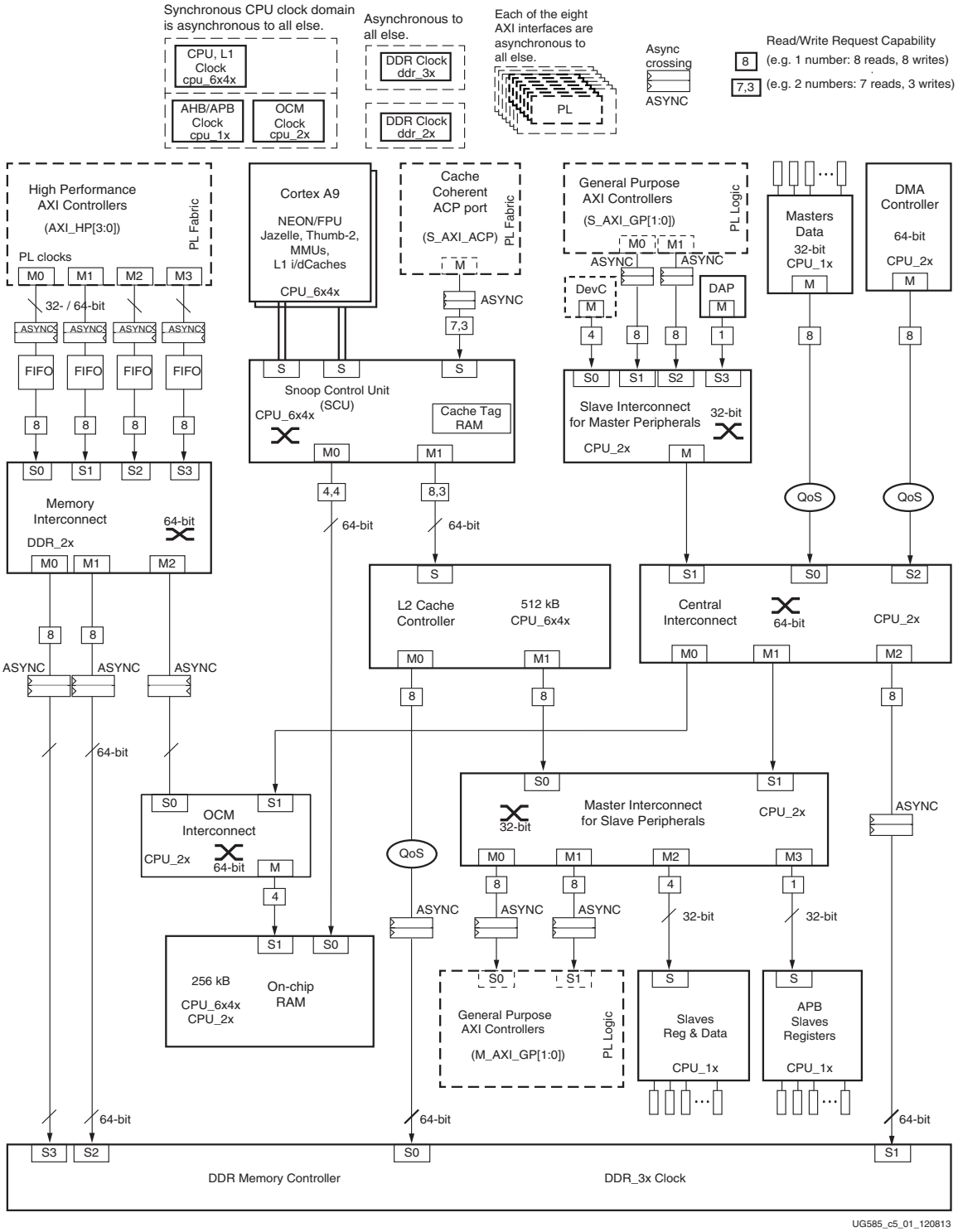


Figure 3.3: Interconnect block diagram [39]

DSP Blocks

Digital Signal Processing (DSP) blocks are optimized for multiply and add (MAC) operations, and they are a lot used for vector or matrix multiplication, expecially in Neural Networks. The PYNQ-Z2 has 220 DSP48E1 blocks, reported in [37] and at Figure 3.4. One DSP block can multiply two's complement digits of 18 and 25 bits each. For multi-

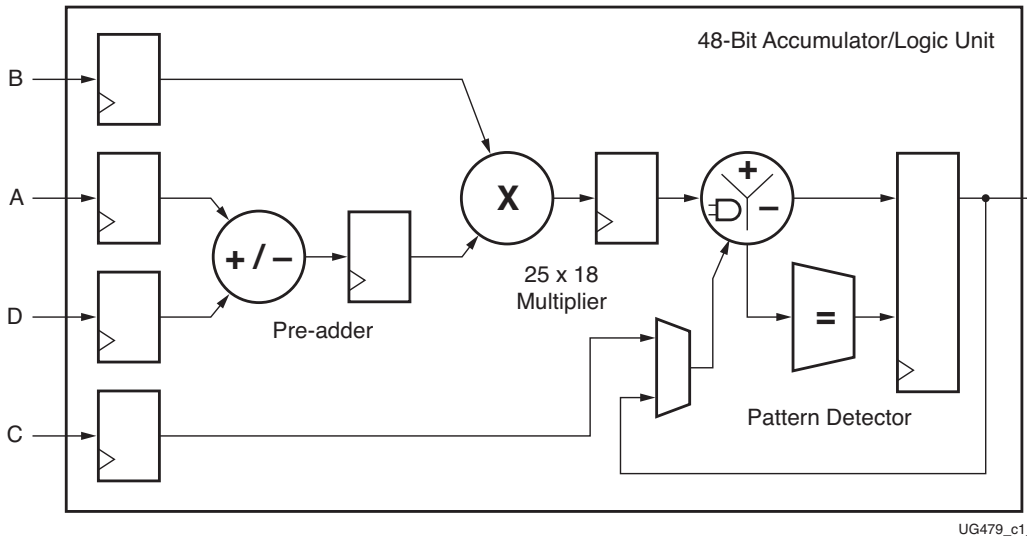


Figure 3.4: DSP48E1 functionality [37]

plying more bit-width digits, more than one DSP have to be cascaded together.

The pre-adders in the DSP blocks make them very convenient and fast for MAC operations. In addition, DSP48E1 blocks has some registers in critical points of the data flow that allow pipelined operations up to three stages (3 clock cycles). Therefore, in case of a limited number of resources, the same DSP can be re-utilized in a pipelined manner with an Initiation Interval $II = 1$ clock cycle.

Storage

The PYNQ-Z2 board memory capabilities can be divided into three main components:

- DDR: This is the global memory, which is depicted at Figure 3.3. It is directly connected to both the FPGA and CPU for performance purposes.
- Block RAM (BRAM): This memory is internal to the FPGA. The information stored in BRAMs can be immediately accessible.
- Distributed RAM: This type of memory is actually fabricated with FFs and LUTs. The storage capability depends on the number of LUTs and FFs used, but for depths

greater than 128 bits BRAM should be used [36].

The BRAM in true dual-port consists of 36kb of storage area and two completely independent access ports, A and B, reported at Figure 3.5 from [38]. The RAM itself is divided

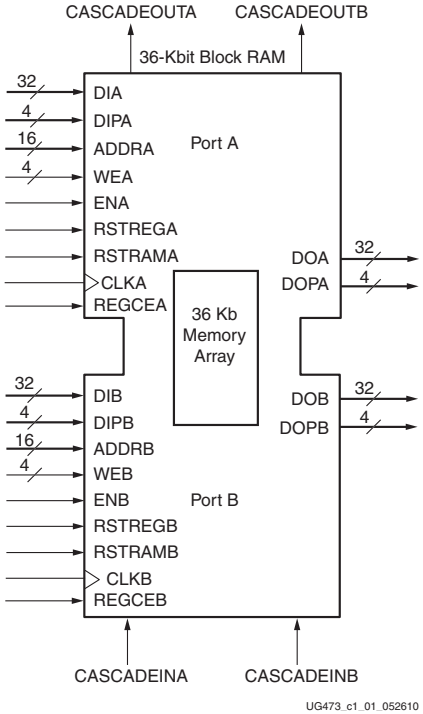


Figure 3.5: True dual-port RAMB36 [38]

into two identical blocks of 18kb each, with structure and behaviour fully symmetrical. More blocks can be cascaded to augment the ports capability. For example, the RAMB36 in simple dual-port mode comprises one port of 72 bits and a 36kb storage array. The PYNQ-Z2 is composed by a total of 630kb of block RAMs, therefore counting 280 blocks 18kb of storage.

3.3. Xilinx Vitis HLS Workflow

The Vitis High-Level Synthesis (HLS) tool is a high-level synthesis tool that allows C, C++, and OpenCL functions to become hardwired onto the device logic fabric and RAM/DSP blocks. Vitis HLS implements hardware kernels in the Vitis application acceleration development flow and uses C/C++ code for developing RTL IP for Xilinx device designs in the Vivado Design Suite.

HLS code gives the developer the ability to specify some of the low-level details of an FPGA implementation, like the amount of resources to use for a particular operation, or the protocol used for the interfaces, or again the clock frequency of the generated IP.

Vitis HLS supports two design flows: the Vitis Kernel flow or the Vivado IP flow. The former is more restrictive than the Vivado IP flow, and the kernels produced by the HLS tool must meet the specific requirements of the platforms and Xilinx Runtime (XRT) [40]. The latter is more flexible and less structured than the Vitis Kernel flow. It supports a wide variety of interface specifications and data transfer protocols, and does not naturally have the XRT requirements of the Vitis system. The Vivado IP flow provides much greater discretion in your design choices, however, leaves the integration and management of the IP up to you as well. I employed Vivado IP flow, which includes the following steps:

1. Compile, simulate, and debug the C/C++ algorithm.
2. View reports to analyze and optimize the design.
3. Synthesize the C algorithm into an RTL design.
4. Verify the RTL implementation using RTL co-simulation.
5. Package the RTL implementation into an IP and export it.
6. Integrate the IP into the IP Integrator of Vivado and build a block design.

3.3.1. Write Code for Simulation and Testing

The HLS code implementation comprises a source code and a test bench. The test bench is pure software and can be written in C++ without any particular constraint, whereas the source code, where the kernel is implemented, must respect some implementation constraints, like not using dynamic memory allocation or functional programming.

The source code is composed by a top-level function (the kernel function), serving as the accelerated function on the FPGA, along with optional lower-level functions for enhanced code readability and maintainability. The arguments of the top-level function define the interface signals between the FPGA and the Host. Pragmas can be used to specify which ports the interface arguments must be implemented into. For example, a pointer interface could both be implemented with an AXI Memory-Mapped protocol or an AXI Lite protocol, depending on the design choices. In general, the source code can be enriched with pragmas to optimize the throughput or reduce resource consumption, depending on the objective of the application.

As mentioned earlier, the source code can be thought to be divided into levels. The kernel function is the top-level, and will be synthesized as an RTL module by the tool. If, after the synthesis, the RTL module calls other submodules, they are said to be lower-level modules. An illustration of this concept is available at Figure 3.6. In the example, the

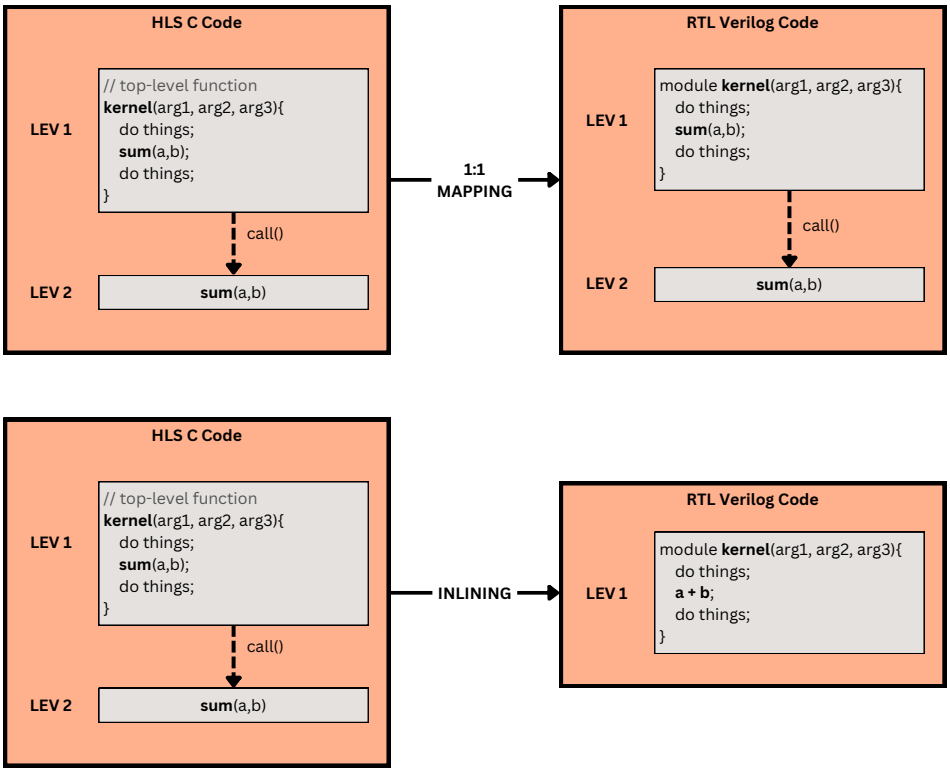


Figure 3.6: High-Level-Synthesis hierarchy

HLS code gets compiled in two different RTL implementations. In the top one, the levels are mapped as they are coded in HLS, resulting in two different RTL modules mapped into FPGA. The bottom one is instead a process called "inlining", i.e. the function called gets inlined into the upper-level callee and results in only one level implemented in RTL. Vitis HLS default behaviour is to inline functions when possible, as it usually results in a more optimized hardware behaviour. However, at the same time, it may result in more resource consumption and less control over the implementation part. It is therefore important to understand the concept of hierarchy level before starting the design.

When using C code to develop RTL Intellectual Property (IP) for FPGA implementation, certain inherent limitations arise with respect to the supported constructs. To ensure successful synthesis, the code written must pre-determine the precise amount of resources required for FPGA implementation. I will delve into some of the limitations taken into account later on in this manuscript.

Constructs Limitations

In HLS implementation, the resources have to be known before the execution of the application. Therefore, dynamic memory allocation, a common practice in software devel-

opment, is prohibited when implementing source code in HLS. Consequently, alternative strategies must be employed to manage data and memory efficiently within the constraints of the RTL IP development process.

Another significant limitation pertains to the use of pointers in Vitis HLS. Firstly, Vitis HLS does not support general pointer casting, restricting casting capabilities solely to native data types. Secondly, while Vitis HLS does allow the use of pointer arrays for synthesis, it imposes specific conditions on their usage. Each pointer within the pointer array must point to a scalar or an array of scalars. However, arrays of pointers are not permitted to point to additional pointers.

Finally, the use of functional programming is not admitted in HLS. Although it is a very convenient construct for Runge-Kutta algorithms, it is not synthesizable in HLS. Hence, when calling the Runge-Kutta function, instead of passing as argument the ODE function like it's done for C++ or Python, every single parameter has to be singularly forwarded through the ODE callee. In Algorithm 3.1 and 3.2 the difference between the two methods is shown. Functional programming is much easier to maintain and extend: without it, for each modification to the f arguments, the kernel must be also modified.

Algorithm 3.1 Functional programming

```
1: def f(x, f_args);          ▷ The arguments args are the same for each call of f.
2:
3: f_args ← initial values.
4: lambda(x) = f(x, f_args);
5:
6: call kernel(lambda(x), kernel_args):
7:    ...
8:    x ← update value
9:    call lambda(x);                  ▷ args is implicitly passed through lambda.
10:   ...
```

Algorithm 3.2 Traditional programming

```
1: def f(x, f_args);          ▷ The arguments args are the same for each call of f.
2:
3: f_args ← initial values.
4:
5: call kernel(lambda(x), f_args, kernel_args):
6:    ...
7:    x ← update value
8:    call f(x, f_args);                ▷ args must be explicitly passed to f.
9:    ...
```

Code Optimization

Vitis HLS strives to optimize code performance by leveraging parallelism and pipelining techniques, whenever applicable. The software incorporates default optimization options, while also allowing developers to specify additional optimizations using pragmas. Pragmas serve as directives to guide the tool's behavior, enabling developers to tailor optimizations to their specific needs.

When optimizing code in Vitis HLS, two primary directions can be pursued. The first direction focuses on throughput and latency, aiming to maximize throughput and minimize latency to the greatest extent possible. This approach is geared towards achieving high-performance execution.

The second direction emphasizes resource consumption and power efficiency. By minimizing resource utilization, this optimization direction seeks to reduce power consumption, making it particularly beneficial in power-constrained environments.

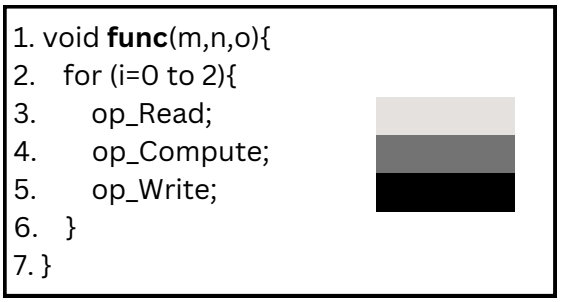
The optimal trade-off between throughput, latency, and resource consumption depends on the specific application being accelerated. Developers must carefully consider the requirements of the application and strike a balance that satisfies the desired goals. By effectively managing these trade-offs, developers can ensure that the optimized code meets the required performance, resource utilization, and power consumption targets.

Pipelining code helps reducing the Initiation Interval (II) between one iteration and the next one, consequently reducing the latency for the whole loop. Pipelining involves cutting the computation in independent stages (like read, compute, write), and overlap subsequent iterations. I reported an example at Figure 3.7. The function `func`, without pipelining, would have to wait for 3 clock cycles at each iteration. It is easy to see that the more iterations are employed in a loop, the more gain pipelining creates.

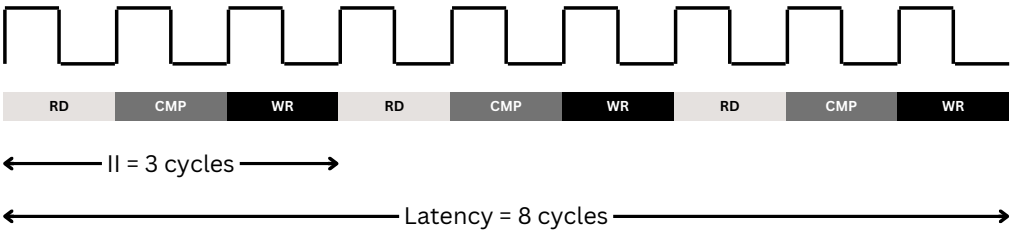
While pipelining can be achieved without increasing the number of resources, parallelization inherently requires more resources to compute. A very common process of parallelization is loop unrolling, where the computations of each iteration happen at the same time. Loop unrolling is not possible if the next iteration is dependent on the result of the previous one. At Figure 3.7d unrolling is shown, assuming that each iteration of the loop is independent from one another.

Arbitrary Precision Data Types

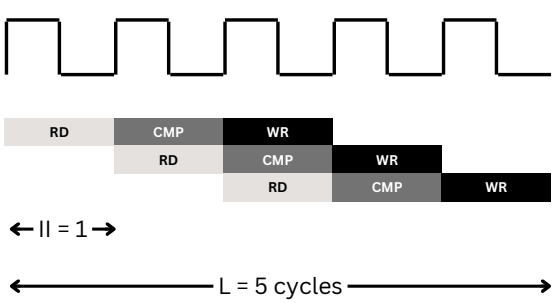
Arbitrary precision (AP) data types allow your code to customize the bit-width of the variables, without necessarily using the standard types ($double, float, int \ldots$). The smaller



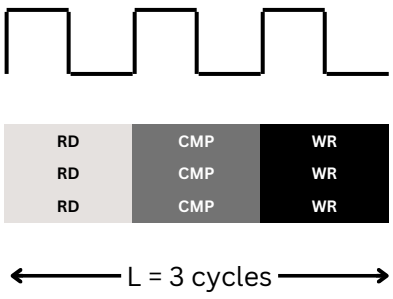
(a) Loop example



(b) Without loop pipelining



(c) With loop pipelining



(d) With loop unrolling

Figure 3.7: Loop optimizations [40]

bit-widths result in hardware operators which are in turn smaller and run faster. This allows more logic to be placed in the FPGA, and for the logic to execute at higher clock frequencies. AP data types are provided in an HLS library and allow the developer to model data types of any width from 1 to 1024 *bits*. This default maximum bit-width may be overridden manually by the developer. AP data types can also be used to implement fixed-point data types. Fixed-point types are a useful replacement for floating point types, because floating points are complex and resource-hungry to implement on FPGA. Unless, unlikely, the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type. Moreover, hardware integer operators, like `add` or `mul` can be also used for fixed-point, without the need of instantiating different modules.

Vitis HLS provides a library with AP data types operations, including the square root function. However, these operations have limited bit-width requirements. Specifically, considering I the integer number of bits, F the fractional number of bits and $W = I + F$ the total number of bits, they must adhere to the following bit-width specifications:

1. `ap_fixed<W,I>` where $I \leq 33$ and $F \leq 32$
2. `ap_ufixed<W,I>` where $I \leq 32$ and $F \leq 32$
3. `ap_int<W,I>` where $I \leq 33$
4. `ap_uint<W,I>` where $I \leq 32$

As explained in Section 4.3.1, these limitations have been a problem for implementing the fixed-point square root operation, hence the need to implement a custom function.

3.4. AXI Protocols

The AMBA (Advanced Microcontroller Bus Architecture) AXI (Advanced eXtensible Interface) protocol is a part of ARM's system IP. This protocol is designed for high-speed, high-frequency system designs and includes features that make it suitable for high-speed sub-micron interconnect.

AXI is a point-to-point interconnect that connects one master (initiator) to one slave (target). Data is transferred between the master and slave using a write channel to the slave or a read channel to the master.

There are 3 types of AXI4-Interfaces (AMBA 4.0)[2]:

[2]https://support.xilinx.com/s/article/1053914?language=en_US, Last accessed: July 2, 2023

- AXI4 (Full AXI4): For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: For high-speed streaming data.

The AXI protocol defines 5 transaction channels:

- Read Address Channel: This channel is used to send the address from which data is to be read.
- Read Data Channel: This channel is used to receive the data that has been read from the memory. It also carries information about whether the read operation succeeded or failed.
- Write Address Channel: This channel is used to send the address to which data is to be written.
- Write Data Channel: This channel is used to send the data that is to be written to the memory.
- Write Response Channel: This channel is used to receive information about whether the write operation succeeded or failed. This is necessary because in the AXI protocol, writes can fail, unlike in the standard five-port SRAM interface.

A channel is an independent collection of AXI signals associated with the VALID and READY signals.

To read content from the memory, the master sends the address to read from through the read address channel, and when the memory is ready to send back data it reads the data from the read data channel. For the writing, the master tells the memory where to write data through the write address channel, waits for the memory to be ready for receiving the data, and then sends the data through the write data channel. The write response channel communicates the success or failure of the data written [4].

3.4.1. AXI4 Memory Mapped

The Full AXI4 supports burst transfers and out-of-order transactions. Out-of-order transactions enable the system to continue processing other transactions while waiting for a response of a particular transaction. Bursting involves transferring multiple transfers, or *beats*, in only one transaction. Bursting transactions is necessary for high-performance data flows.

The number of beats for each burst is called *burst length*, and it can go up to 16 for the AXI3 protocol and 256 for the AXI4 protocol. Considering that the Zynq PS utilizes the AXI3 protocol, a maximum burst length of 16 is considered in this case. The beat size of the protocol is defined as 128 bits, but the HP ports in the PYNQ-Z2 between the DDR and the PL can go up to 64 bits. Hence, a length of maximum 64 bits can be considered in this case.

The concept of a burst transaction is very similar to a pipelined loop. An illustration is shown at Figure 3.8. Being the DDR an external memory, the latency to fetch data

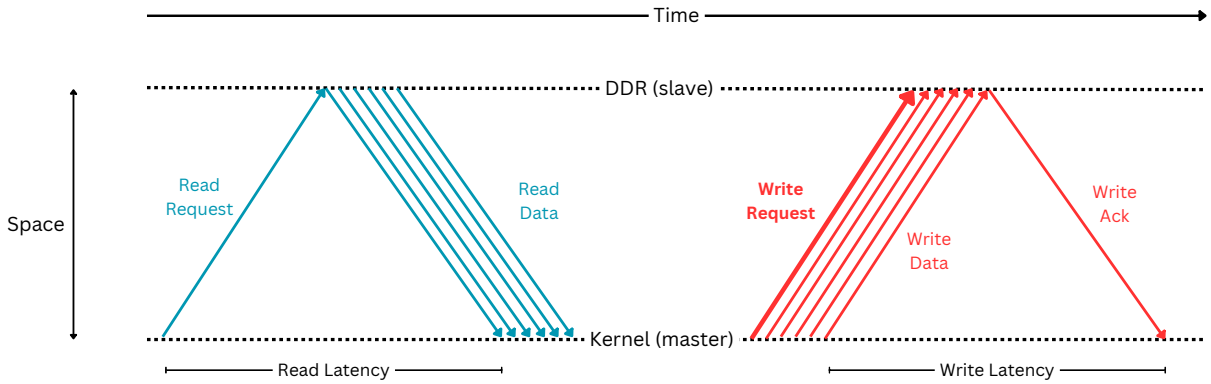


Figure 3.8: AXI burst transaction [40]

and write data is in the order of dozens of clock cycles, which is huge if compared to 1 clock cycle needed to read from BRAM in the FPGA. Hence, instead of loading each single element independently from external memory, the idea is to load the most possible amount of elements in one transaction, store them to the internal BRAMs, and proceed the computation loading and storing from and to BRAMs. In the same way, writing to DDR can be optimized by storing the most possible amount of results inside internal BRAMs and then writing them to external memory in one transaction. From the master, only the initial address and the burst length is indicated, so that there is no need to wait for the latency round-trip each time.

AXI-MM in Vitis HLS

In Vitis HLS, the AXI Memory-Mapped protocol can be configured in three different modes:

- `offset = off`: The base address where to write to or read from of the memory is sets at the HLS level. In this mode, the base address cannot be changed at run-time.

- `offset = direct`: The address can be changed at run-time, while the HLS module is computing
- `offset = slave`: The address is chosen by the Host and is set to the HLS through an AXI Lite connection. After setting the address, the HLS can read from global memory.

I chose the `slave` mode for my implementation, because it is more flexible than the `off` mode and less complex than the `direct`. At Figure 3.9 the data flow for this mode is shown. The Host starts by writing the content that the HLS module will have to read

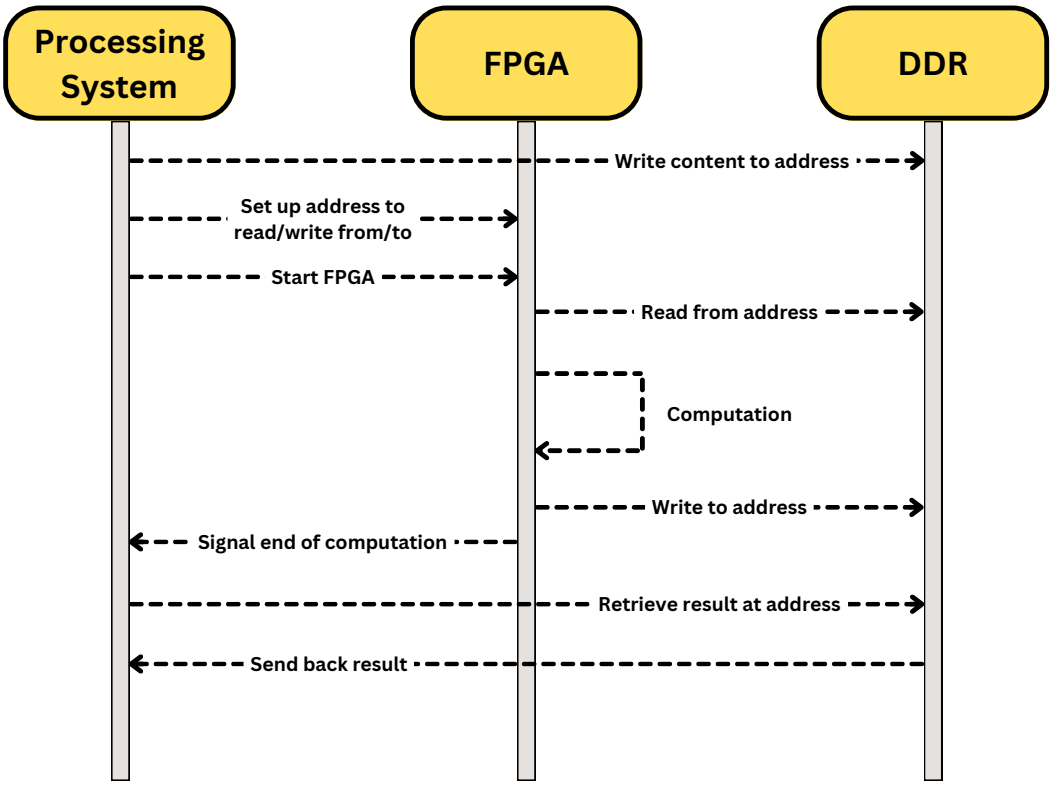


Figure 3.9: AXI Master slave mode data flow

from DDR, and proceeds later on to communicating to the FPGA the starting address of the DDR. This communication happens through the AXI Lite protocol, where the signal exchanged is a 64-bit address. This operation may be repeated for different AXI HP ports, up to 4. After setting the addresses, the Host starts the HLS module, which reads from memory, computes and writes to memory. In the end, the HLS module signals the Host, which can proceed to retrieve the information from the DDR, which is now updated.

It is worth noticing that between the top-level arguments of the HLS module and the DDR, Vitis HLS employs the so called "AXI Adapter", which works as bridge between the IP

and the Zynq PS. The AXI Adapter supports ports up to 512-bit wide PL-side, converting them to normal 64-bit ports PS-side. A 512-port is broken down into 8 blocks of 64 bits each, and the adapter will manage the transfer of data over a number of clock cycles[3]. With this configuration, there is the theoretical possibility to transfer 512 bits per clock cycle, but it is very unlikely that happens. As shown in Section 3.2.2, the FIFO buffers between the Memory Interconnect and the PL allow for asynchronous interconnection, as the ones between the MI and the DDR. However, to ensure a 512 bits/cc transfer the MI and DDR should have a frequency 8 times higher than the PL. In addition, PL-side, the information must be stored inside BRAMs, which have a limited number of ports. Finally, synthesizing a 512-bit wide port occupies a lot of resources in terms of LUTs and FFs. As a consequence, it is rarely worth it widening the port up to 512 bits.

3.4.2. AXI4 Lite

AXI4 Lite is a simplified variant of the AXI protocol, designed for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).

The key functionality of AXI4-Lite operation encompasses several important aspects. Transactions within the AXI4-Lite protocol have a burst length of 1, ensuring each transaction involves a single data transfer. Furthermore, the protocol always utilizes the full width of the data bus, differently from Full AXI4. It is worth noting that AXI4-Lite supports a data bus width of only either 32-bit or 64-bit configurations.

Finally, all accesses within the AXI4-Lite protocol are non-modifiable and non-bufferable, and the protocol does not support out-of-order transactions.

3.5. Synthesis Analysis in Vitis HLS

During synthesis analysis, the tool shows the amount of resources estimated for the HLS module, its estimated slack time (if negative, the module does not process all the information at each clock cycle), the operational units utilized and the correspondent lines in the source code.

In addition to these pieces of information, the schedule viewer can show an estimated schedule for each operation of the module. For example, in case of a loop pipelined, it will show if the Initiation Interval required is respected or if some limitations arose, such as a dependency limitation. When parallelization is required but it cannot be applied for

[3]discuss.pynq.io/t/axi-burst-size-in-pynq-z2-vs-vitis-hls/5448/15?u=davide-giacomini, Last accessed: July 2, 2023

resource limitations, the synthesis analyzer and in particular the schedule viewer show where and why the parallelization has not been applicable.

The analyzer can also show the mapping between C functions hierarchy and RTL modules hierarchy.

3.6. RTL Co-simulation

After generating the Register Transfer Level (RTL) design through synthesis and analysis, it can be co-simulated using the same test function employed for software simulation. During co-simulation, the resources required for FPGA implementation and the clock frequency imposed during synthesis are taken into consideration. Co-simulation offers a more accurate analysis of timing behavior, allowing for a detailed examination of the design's performance.

It is worth noting that co-simulation results may differ from software simulation outcomes, even if the software simulation was successful. This disparity can occur due to various factors, such as the inclusion of FPGA-specific constraints and the impact of physical implementation on timing. Hence, co-simulation provides a valuable means to identify potential timing issues and verify the design's behavior under more realistic hardware conditions.

3.7. RTL IP Export

Once the design is generated, it can be exported as an RTL IP. This exportation enables seamless integration of the accelerated function into the block design using the Vivado Design Suite. The exported RTL IP encompasses all the essential information required for smooth integration with the PS during block design.

The IP generated includes vital components, such as the Verilog and VHDL implementations, ensuring compatibility with different design languages. Additionally, it incorporates constraints specified in an `xdc` (Xilinx Design Constraints) file. These constraints define critical aspects of the IP's behavior and interface, facilitating proper integration with the PS in the block design.

3.8. IP Block Design Integration

The IP Block Design Integrator in Vivado is a powerful tool that enables seamless integration and configuration of Intellectual Property (IP) blocks within a larger system

design. It is a key component of the Vivado Design Suite, which is widely used in the field of digital design and FPGA development.

The IP Block Design Integrator simplifies the process of creating complex designs by providing a graphical environment for block-level design and integration. It allows designers to combine IP cores, customized logic, and other design elements into a cohesive system, facilitating efficient hardware-software co-design.

One of the main advantages of the IP Block Design Integrator is its user-friendly interface, which allows designers to drag and drop IP blocks from a predefined catalog and interconnect them using a graphical canvas. This visual representation of the design simplifies the creation of connections between different modules, reducing the complexity and potential for errors.

Furthermore, the IP Block Design Integrator offers advanced features for configuring and customizing IP blocks. Designers can modify parameters, set constraints, and define interfaces for each IP block to tailor its behavior to the specific requirements of the system. This flexibility allows for integration of both Xilinx-provided and user-defined IP blocks into the overall design.

3.9. Python Productivity for Zynq

PYNQ, which stands for "Python productivity for Zynq", refers to the Python interface that enables smooth communication between the Processing System (PS) and the Programmable Logic (PL) components of the PYNQ board. As mentioned in Section 3.2.2, the PYNQ board consists of an ARM CPU and an Artix-7 FPGA, with the CPU acting as the Host which controls the FPGA. The PS runs on an Ubuntu operating system and provides Jupyter notebooks as a convenient interface to interact with the FPGA.

The PS of the PYNQ-Z2 board can be easily accessed through SSH and it appears as a normal Ubuntu-based operating system[4]. The system has a home page and a Jupyter notebook folder accessible through browser. This approach simplifies the development process by providing a familiar and accessible programming environment. Through a Jupyter notebook file, Python can be leveraged to run any kind of application on the board CPU. Moreover, the system comes with the library `pynq` incorporated in Python, offering extensive functionality to program interface the CPU with DDR and FPGA. It also contains a bunch of IPs ready to be imported and harnessed.

By leveraging the capabilities of the PYNQ framework, users can easily combine the power

[4]pynq.readthedocs.io/en/v3.0.0/getting_started/pynq_z2_setup.html, Last accessed: July 2, 2023

of Python programming with the flexibility of FPGA-based hardware acceleration. This integration eases development tasks and facilitates rapid prototyping and experimentation.

4 | Methodology

In this chapter, I will thoroughly explore the intricate steps taken to enhance the efficiency of the orbit propagation algorithm on the FPGA PYNQ-Z2. The fundamental goal of this implementation is to showcase the viability of accelerating the orbit propagation problem on devices with limited resources, such as the PYNQ-Z2.

4.1. Problem Definition

As a fundamental benchmark to highlight the capabilities of accelerated computations for orbit propagation, particular emphasis was placed on the two-body problem, extensively discussed in Section 2.1.1. The equations of motion, in the absence of non-conservative forces, have been represented in Equation 2.3.

In this context, the orbit propagation involves solving the Initial-Value Problem (IVP) given by the aforementioned set of Ordinary Differential Equations (ODEs) and the initial state vector $\mathbf{x_0} = (\mathbf{r_0}, \mathbf{v_0})^T$:

$$\begin{cases} \dot{\mathbf{x}} = A\mathbf{x} \\ \mathbf{x}(t_0) = (\mathbf{r_0}, \mathbf{v_0})^T \end{cases} \quad \text{with} \quad \mathbf{x} = \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}, \quad A = \begin{bmatrix} O & I \\ -\frac{\mu}{\|\mathbf{r}\|^3} I & O \end{bmatrix} \tag{4.1}$$

I discussed some possible numerical solution to an IVP in Section 2.3. While the Euler method is the simplest and least computationally intensive approach, its Local Truncation Error (LTE) is too high for Orbit Determination (OD) applications. Linear-multistep methods may achieve the LTE desired, but their steps are fixed in size and cannot adapt to the orbit shapes. On the other hand, the Runge-Kutta methods have proven to be a more suitable alternative, offering a manageable computational complexity and a variable step size depending on the LTE tolerance set in advance by the user.

4.2. OD Implementation

As discussed in Section 2.3.3, the Runge-Kutta method is renowned for its high accuracy, which can vary depending on the order of the method and its adaptability. Adaptive Runge-Kutta methods, in particular, offer the flexibility of adjusting the step size based on the maximum allowable Local Truncation Error (LTE) for a given application. In the context of Low Earth Orbit (LEO), an LTE in the order of micrometers (10^{-9} km) is typically considered acceptable, whereas for interplanetary orbit, an LTE in the order of meters (10^{-11} AU) may be deemed acceptable.

For the purpose of this work, the RK5(4)7M method has been implemented, aiming for better comparability with the Matlab implementation. The equations described in Section 2.3.3 has been applied, utilizing the Butcher tableau provided in Table 2.2. Due to the constrained resources on the FPGA, the calculation of the optimal step size has been implemented differently than the one outlined in [13] (Equation 2.6), because the equation of the paper squares and roots, which are generally extremely inefficient. Instead, it has been employed a fixed scale factor which determines whether the time step should be extended or reduced at each iteration. Specifically, I utilized a scale factor of 1.11 to extend the time step and a scale factor of 0.99 to reduce the time step. By using these scale factors, the goal was to achieve a sub-optimal yet effective time stepping strategy. This ensured that the propagation algorithm could effectively handle orbit variations while maintaining the desired level of accuracy throughout the simulation.

In the context of OD, adaptability in the step-size is fundamental expecially for eccentric orbits. The point farthest from the object around which another object is orbiting is referred to as the "apocenter", while the point closest is known as the "pericenter" (Figure 4.1). Near the pericenter, the speed of the orbiting object increases as the gravitational

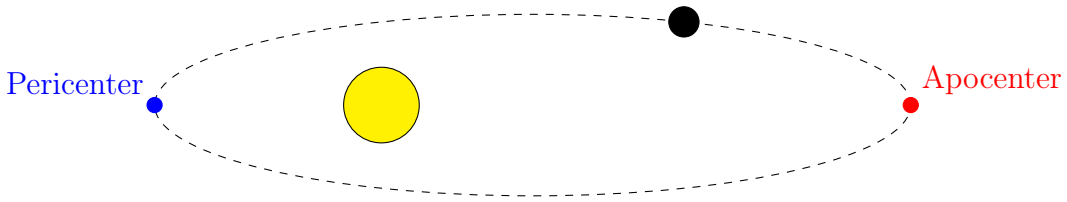


Figure 4.1: Apocenter and pericenter illustration

attraction increases, hence the position and velocity change rapidly. To accurately capture these rapid changes, a smaller step size is needed to ensure that the numerical integration accurately represents the orbit's trajectory. On the other hand, near the apocenter, the object's velocity is lower, and the position and velocity changes are less pronounced

compared to the pericenter. Therefore, a larger step size can be used without sacrificing much accuracy. This is why it was so important to employ an adaptive step size based on the accuracy required.

I implemented the algorithm using three different programming languages: Python, C++, and HLS C. Python was essential for its ease of use and allowed me to quickly compare my implementation with MATLAB. C++ was used for extensive testing and served as a bridge to validate the HLS C code. Finally, HLS C was necessary for Vitis HLS to synthesize the RTL counterpart specifically for the FPGA. By leveraging the strengths of each language, I could ensure comprehensive testing, efficient development, and successful synthesis for the FPGA implementation.

Based on the reference provided in [13] for the RK5(4)7M method, my algorithm employs 7 intermediate stages denoted as k_i. To begin, I will present a high-level pseudo-code that can be universally applied to all implementations. Later, I will delve into specific variations among the implementations. The high-level pseudo-code is outlined in Algorithm 4.1.

There are several minor implementation adjustments to the RK5(4)7M algorithm that have been made. Firstly, the time step is now bounded by a maximum and minimum value. If the minimum time step, denoted as $h_{\min}$, is set too high to maintain the local truncation error (LTE) within the specified tolerance $atol$, a flag is raised to indicate that the LTE exceeded the tolerance at some point during the computation. This flag serves as a notification to the interface that the LTE may not have remained within the desired tolerance throughout the algorithm execution.

Moreover, the two-body problem equations of motion do not depend explicitly on time, as shown in Equation 2.3. The ODE function $f(t_n, \mathbf{x}_n)$ therefore does not really need the time instant as input. Hence, the function indicated in the algorithm only depends on the state vector: $f(\mathbf{x}_n)$.

Additionally, a refinement has been introduced by combining the computation of $\mathbf{x}_{n+1}$ with the calculation for the seventh intermediate stage $\mathbf{k_7}$. This optimization is possible because the values of b_i and a_{i7} in Table 2.2 are identical. As a result, it becomes feasible to pre-calculate the next state vector, and use it to derive the value of $\mathbf{k_7}$. This improvement helps to reduce redundant calculations and streamline the algorithm's efficiency.

Lastly, a specific scenario has been addressed to ensure precise alignment with the final time stamp provided as an input. If the sub-optimal predicted time step exceeds the specified final time, an adjustment is made to ensure that the algorithm reaches the final

Algorithm 4.1 Runge-Kutta 5(4)7M Algorithm

Inputs: ODE function $f(\mathbf{x})$, initial and final time t_0 and t_f, initial state vector $\mathbf{x}_0$, initial time step h_0, maximum and minimum allowable time step $h_{\max}$ and $h_{\min}$, and absolute tolerance $atol$.

Initialize matrix $\mathbf{x}$ and vectors $\mathbf{t}$ and $\mathbf{h}$ with their initial values.
$h_n \leftarrow h_0$, $t_n \leftarrow t_0$, $\mathbf{x}_n \leftarrow \mathbf{x}_0$, $flag \leftarrow$ `true` ▷ Values initialization.

while $t_n < t_f$ **do**
  if $t_n + h_n > t_f$ **then**
    $h_n = t_f - t_n$ ▷ Adjust the last step to end exactly at t_f if necessary.
  end if

  Update $\mathbf{k}_i$:
    $\mathbf{k}_1 = f(\mathbf{x}_n)$
    $\vdots$ ▷ Intermediate stages as mentioned in Equation 2.5
    $\mathbf{k}_6 = f(\mathbf{x}_n + (a_{61}\mathbf{k}_1 + a_{62}\mathbf{k}_2 + \ldots + a_{65}\mathbf{k}_5)h)$
  End Update

  $t_{n+1} \leftarrow t_n + h$ ▷ Prepare next time stamp t_{n+1}.
  $\mathbf{x}_{n+1} \leftarrow \mathbf{x}_n + h\sum_{i=1}^{6} b_i\mathbf{k}_i$ ▷ Prepare next state vector $\mathbf{x}_{n+1}$.
  $\mathbf{k_7} \leftarrow f(\mathbf{x}_{n+1})$ ▷ Calculate last stage $\mathbf{k}_7$.
  $LTE \leftarrow \|h\sum_{i=1}^{7}(b_i - b_i^*)\mathbf{k}_i\|$ ▷ Calculate LTE with norm.

  if $LTE \leq atol$ **or** $h_n \leq h_{\min}$ **then**
    Append $\mathbf{x}_{n+1}$, t_{n+1}, and h_n to $\mathbf{x}$, $\mathbf{t}$, and $\mathbf{h}$ ▷ Store $\mathbf{x}_{n+1}$ and t_{n+1}.
    $\mathbf{x}_n \leftarrow \mathbf{x}_{n+1}$, $t_n \leftarrow t_{n+1}$ ▷ Prepare next iteration.
    $h_{n+1} \leftarrow 1.11 \cdot h_n$ ▷ Increase time step.

    if $LTE > atol$ **and** $h_n \leq h_{\min}$ **then**
      // If $atol$ is not respected and h can't be decreased, raise a flag and continue
      $flag \leftarrow$ `false`
    end if
  else
    // If $atol$ is not respected and h can be decreased, decrease h and repeat
    $h_{n+1} \leftarrow 0.99 \cdot h_n$
  end if

  $h_n \leftarrow \max\big(\min(h_{n+1}, h_{\max}), h_{\min}\big)$ ▷ Update time step.

end while
return $\mathbf{x}, \mathbf{t}, \mathbf{h}$, flag

time exactly. Consequently, in this particular situation, the time step might become smaller than the minimum time step. By implementing this adjustment, the algorithm guarantees that the computation concludes precisely at the intended final time while maintaining consistency with the defined time step constraints.

In this context, the function $f(\mathbf{x})$ used for the intermediate stages is defined by the set of ODEs of Equation 4.1:

$$\dot{\mathbf{x}} = A\mathbf{x} =: f(\mathbf{x})$$

The implementation of this function is straightforward and can be found in Algorithm 4.2. In the current implementation, only conservative forces have been considered, reflecting the ideal case scenario without atmospheric drag, solar radiation pressure, or perturbations from other celestial bodies.

Algorithm 4.2 ODE Algorithm

1: **Inputs:** state vector $\mathbf{x}_n = (\mathbf{r}_n, \mathbf{v}_n)^T$
2:
3: $\mathbf{r}_{n+1} \leftarrow \mathbf{v}_n$ ▷ Calculate new position vector
4: $\mathbf{v}_{n+1} \leftarrow -\mu \cdot \mathbf{r}_n \|\mathbf{r}_n\|^{-3}$ ▷ Calculate new velocity vector
5:
6: **return** $\mathbf{x}_{n+1} \leftarrow (\mathbf{r}_{n+1}, \mathbf{v}_{n+1})^T$

4.3. High Level Synthesis Implementation

Before delving into the High Level Synthesis (HLS) implementation, it is essential to provide a comprehensive overview of the workflow associated with Xilinx Vitis HLS. As discussed in Section 3.3, Vitis HLS offers two distinct flows: the Vivado IP flow and the Vitis kernel flow. For this specific use case, it has been leveraged the Vivado IP Flow, as the objective was to accelerate the orbit propagation algorithm, generate its IP, and interface it with the Zynq Processing System (PS) using Vivado's block design capabilities.

It has been employed a visual comparison strategy to verify the accuracy of the computed orbit, as outlined in the results (Chapter 5). Initially, the computed orbit has been compared against a test function and then against a reference orbit. The objective of the reference orbit comparison was to assess the accumulated error between the precise orbit and the computed one. Additionally, it has been conducted a comparison between the kernel and the test function to examine the variation in precision between double floating points and fixed points. As expected, in this specific case, there were no inconsistencies between the two, since the chosen fixed-point configuration adequately preserved precision by utilizing a sufficient number of bits.

The exact orbit was calculated using a MATLAB function called `keplerUniversal`[1]. This implementation represents our reference trajectory, since it provides the analytical solution of the keplerian motion at the desired time instant from a set of initial conditions. The computed time stamps and their corresponding state vectors of the orbit have been stored in a `csv` file. Afterwards, the saved state vectors have been compared with the state vectors obtained as output from the aforementioned MATLAB function.

The test function has been implemented in C++, which bears resemblances to the Python implementation. Similar to Python, it has been utilized functional programming techniques[2] for a neater coding style. Standard vectors and arrays were employed to facilitate dynamic memory allocation. However, it is worth mentioning that these methodologies are incompatible with HLS programming. Further analysis will be done in the subsequent discussions.

In the next sections more details will be seen about the HLS implementation, starting by assessing the problem of precision, and continuing with how to implement interfaces between the Programmable Logic (PL) and the Processing System (PS), finally completing with several ways of optimizing the code for better synthesis.

4.3.1. Arbitrary Precision Implementation

One of the first things to consider when implementing complex calculus for FPGA implementation is to choose whether to employ floating point precision or fixed point precision. Floating point precision representations use a sign bit (S), E exponent bits and M mantissa bits. Figure 4.2 shows a visual representation of the double floating point precision. The formula for converting binary representation to decimal format involves calculating

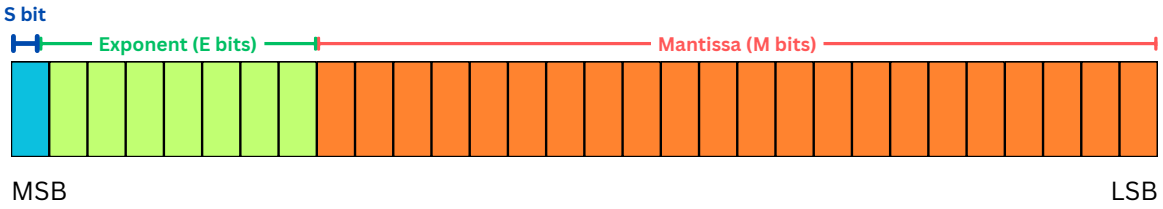


Figure 4.2: Floating point representation

$(-1)^s(1+m) \times 2^{(e-bias)}$, where s is the sign bit, m is the value of the mantissa, and e is the value of the exponent. The bias is needed to represent negative exponents and it's defined as $2^{(E-1)} - 1$, where E is the number of bits for the exponent part. Illustratively,

[1] mathworks.com/matlabcentral/fileexchange/35566-vectorized-analytic-two-body-propagator-kepler-universal-variables?s_tid=prof_contriblnk, Last accessed: July 2, 2023

[2] en.cppreference.com/w/cpp/header/functional, Last accessed: July 2, 2023

an instance of double precision floating point representation can be observed in the binary number 1 10000000001 001010...0_b, which can be translated to its binary equivalent as $-1.00101_b \times 2^2$, resulting in the decimal value of -4.625_d.

At Table 4.1 are shown the characteristic values of single and double floating point precision. It is worth noticing that the floating point precisions have a huge range. These

Type	Size	S	E	M	Precision Range
single	32 b	1 b	8 b	22 b	$\approx 10^{\pm 38}$
double	64 b	1 b	11 b	52 b	$\approx 10^{\pm 308}$

Table 4.1: Precision values for floating point representation

values, particularly for the `double` type, are very convenient considering the required order of magnitude tolerance for this specific application. As discussed in Section 4.2, the magnitude can range from a tolerance requirement of 10^{-9} to 10^{-11}. Nevertheless, the implementation of floating point operations such as `dmul` (double-precision floating-point multiplication) or `dadd` (double-precision floating-point addition) necessitates significantly more resources and power consumption compared to their fixed-point counterparts.

In contrast, the utilization of fixed point representation presents notable advantages, primarily in terms of implementation simplicity and the ability to employ the same operational units as integers for addition and multiplication. Within the HLS, the fixed point representation follows a specific structure, utilizing I bits for the integer part and F bits for the fractional part. For unsigned representation, the sign bit is not required, while the signed representation allocates the MSB of the integer part for sign representation. When defining a fixed point data type in HLS, the developer has the option to specify the quantization mode (Q) and the overflow mode (O). By default, the quantization mode truncates the number towards negative infinity, and the overflow mode does not incorporate any control or saturation mechanisms for overflow handling. These default modes are less computational complex to implement. The declaration of a fixed point data type is straightforward and can be implemented using the syntax `ap_[u]fixed<W, I, Q, O>`, where "W" denotes the total number of bits allocated for the representation ($W = I + F$).

The formula for converting fixed-point binary representation to decimal format involves calculating $(-1)^s I.F$. For example, the previous value -4.625_d is now represented by 1100.101_b, assuming the fixed point is signed and $I = 3, F = 3$. Of course, depending on the number of bits chosen for the representation at design phase, the application will

be able to bear only a certain amount of precision. If for example the tolerance is set to 10^{-9}, the algorithm must be able to have a precision of at least the same as the tolerance, with the truncation set to the thirty-first fractional bit ($log_2(10^{-9}) = -29.90$).

In my implementation, the fixed-point configuration has been set to 40 integer bits I and 60 fractional bits F. At Table 4.2 there are the precision and range illustrated. It is

Type	Size	S	I	F	Range	Precision
fixed point	85 b	1 b	30 b	55 b	$\approx \pm 5.369 \times 10^8$	$2^{-55} \approx 2.78 \times 10^{-17}$

Table 4.2: Precision values for Fixed point representation

notable how the precision is still enough for representing the tolerance required by the application, but the range of values is much stricter of the floating points. Moreover, the fixed-point occupy more space than the floating point, which require at most 64 bits of storage. The absence of an exponent part in the fixed-point representation makes it impossible to keep up with the floating point.

As mentioned in Section 3.3.1, the HLS library provides the square root function for unsigned fixed value with a fractional part less than or equal to 32 bits. The maximum precision employable with this bit-width is $2^{-32} \approx 2.33 \times 10^{-10}$ in *input* to the square root function. This is not bearable, considering how usually a square root is the consequence of a previous square operation. Considering, for example, an error in the order of $\approx 10^{-9}$, as soon as the error gets squared, the number of bits to maintain the same precision should double ($\approx 10^{-18}$), yielding to the impracticability of the library.

To fix this problem, I had to come up with a custom implementation, inspired from a pseudo-code thought for HDL code[3]. I implemented that code in HLS C, and I later on used that function to call the square root of a number with any bit-width. The pseudo-code of the custom square root can be found at Algorithm 4.3. In my implementation, the bit-width of the result is given in advance as input of the function, as long as the integer part of the result is $I_{res} \geq ceil(I_{inp}/2)$. Even though the fractional part may be of any length, for avoiding a drop in accuracy, it should follow the same rule of the integer part ($F_{res} \geq ceil(F_{inp}/2)$). In the HLS implementation there are other controls and some optimizations pragmas, but the pseudo-code provided should cover the idea behind the square root of a fixed point number.

[3]projectf.io/posts/square-root-in-verilog/, last accessed: July 2, 2023

Algorithm 4.3 Custom Square Root

Inputs: input radicand X

$A \leftarrow 0$, $T \leftarrow 0$, $Q \leftarrow 0$ ▷ Initialize values

while Q width is less than the expected result width **do**
  Left shift X by two places into A
  Set $T = A - \{Q, 01\}$
  Left shift Q

  if $T \geq 0$ **then**
    $A \leftarrow T$
    $Q[0] \leftarrow 1$
  end if
end while

return Q

4.4. Interface Arguments

The FPGA can interface the Processing System (PS) through the AXI protocol. The arguments of the top-level function of the HLS code are synthesized as AXI ports in input or output. For control signals, AXI Lite is the best choice, as it is less complex to implement. On the other hand, for directly accessing the memory to load large vectors or matrices, the AXI Memory-Mapped interface is the best choice.

The AXI-MM interface is called AXI Master in Vitis HLS (`m_axi`), and it sets a direct connection between the FPGA and the external DDR memory. I implementd the so called "slave mode" (`offset=slave`), introduced in Section 3.4.1, to control the address of the memory-mapped interface.

As discussed in Section 3.4.1, the burst capability of AXI-MM is fundamental for high-performance data transfers. Moreover, although Vitis HLS may widen up the ports until 512 bits, it would have been a waste of resources, given that they would not have been exploited. At Table 4.3 are illustrated all the AXI parameters set in my implementation. I widened the port to 128 bits to keep up with the DDR frequency and I set the burst length at 16. Given the Zynq FIFO buffers capability, I set the write outstanding write requests to 8. The outstanding read requests are set to 1 because the HLS module only reads the first element in memory.

Two of the top-level arguments are array: one stores the state vectors and the other one

offset	data width	r/w burst len	read outs.	write outs
slave	128 *b*	16	1	8

Table 4.3: AXI master parameters

the time stamps. I used one AXI HP port for each, and I used the AXI-MM protocol for direct memory access. Moreover, I saved the elements of the arrays in the DDR as double floating points, as they occupy less storage and are much easier to manage when interfacing the DDR to the CPU in Python. The resources taken to convert the numbers in fixed-point while storing them into the BRAM is negligible.

The other top-level arguments are control signals, either in input or output which are implemented with the same AXI GP port using the AXI Lite protocol. To implement the AXI Master in offset slave, I also added the addresses of the arrays as control signals to the GP port.

The idea behind this burst optimization is to bring the information the closest possible to the computational source. As better explained later on in Section 4.8, the latency required to read from DDR is dozens of clock cycles, while the latency for reading from BRAM is only 1 clock cycle. Burst optimization is able to amortize the latency of the first read into a series of sequential immediate reads, or writes.

4.5. Pipeline Optimization

In the HLS implementation there a lot of multiply and accumulation (MAC) operations inside loops, in order to calculate the intermediate stages k_i and the error LTE of the Runge-Kutta algorithm. The number of DSP used for one MAC operation depends on the bit-width of each value. In my implementation, each value has 100 bits, which translates in 15 DSPs needed for each MAC operation. Therefore, one operation results already in the 14.7% of the total DSPs available on the board. This is the reason why, when possible, I opted for pipelining rather than parallelize.

I will take as an example the HLS implementation of the next state vector update (line 18 of Algorithm 4.1):

$$\mathbf{x}_{step+1} = \mathbf{x}_{step} + h \sum_{i=1}^{6} b_i \mathbf{k}_i$$

In C, each element of vectors like $\mathbf{x}$ or $\mathbf{k}$ must be explicitly updated, resulting in the simple for loop at Algorithm 4.4. In this case, the best implementation for the nested

Algorithm 4.4 State Vector Update

```
// State vector has N elements
for n = 0 to N − 1 do
    mac ← 0
    for i = 0 to 6 do
        mac ← B[i] · k[i][n]
    end for
    y_next[n] ← y_curr[n] + h · mac
end for
```

loops could be different depending on the application. The easiest one is the sequential implementation, which requires each iteration to end before starting the next one (Figure 3.7b). In this case, the initiation interval (II), measured in clock cycles (CC), depends on the number of stages required at each iteration. The overall latency will result in the single iteration latency multiplied by the number of iterations. Another way to implement this nested loop is to pipeline only the inner loop (Figure 3.7c). In this case, the inner loop would have $II = 1$, and the total latency for it would be $N_{it} + (Latency_{it} - 1)$ clock cycles. In this case though the outer loop would still not be pipelined.

If we pipeline instead only the outer loop, Vitis HLS will unroll the inner loop by default (Figure 3.7d). This is a very throughput efficient way of dealing with nested loops but it is extremely inefficient in terms of resource utilization. Moreover, if we unroll a 6-iterations loop with each MAC being 15 DSPs, we are basically occupying the 41% of the DSP resources of the whole board in one single loop. This approach is better with most expensive devices. To avoid unrolling the inner loop, the key word "`rewind`" can be added to the pipeline pragma, and in this case the outer loop doesn't wait for the last iteration of the inner loop to finish. Algorithm 4.5 shows the same example with the added `pragmas`.

Algorithm 4.5 State Vector Update Pipelined

```
for n = 0 to N − 1 do
    #pragma HLS PIPELINE rewind

    mac ← 0
    for i = 0 to 6 do
        #pragma HLS PIPELINE

        mac ← B[i] · k[i][n]
    end for
    y_next[n] ← y_curr[n] + h · mac
end for
```

4.6. Function Hierarchy

The HLS code comprises a hierarchy of functions, and the RTL code is generated depending on it. The top-level function can be considered such as the `main()` function in a C program. It is in the top-level and its arguments are implemented as port interfaces. I covered this concept in details in Section 3.3.1.

Vitis HLS tries to optimize throughput by merging different C functions together in the RTL implementation. When this happens, the code of the function called is said to be "inlined" at the same hierarchy of the callee. This behaviour usually increases the number of resources used for the operations. The developer may need just one sub-optimal implementation to be used repeatedly, such as one module built for a multiplication, say, 100x100 bits. In this case the module can be isolated through a different function in C, and to avoid the inlining process, a specific pragma can be added to the code. At Algorithm 4.6 an example of the MUL implementation is shown. In normal circumstances,

Algorithm 4.6 Function "`multiply(x,y)`"

1: **Inputs:** inputs x and y with a specific bit-width pre-determined
2: `#pragma HLS INLINE off`
3: **return** $x \cdot y$

isolating a multiplication inside a different function is a pattern mistake that unnecessarily complicates the readability and maintainability of the code. In this case, though, inlining the multiplications into the HLS code may lead to implementations of MUL with different bit-widths (even 1 bit of difference), leading to the useless duplication of resources. On the other hand, with this isolation, and choosing the maximum possible bit-width necessary, the same module can be used as needed without any resource duplication.

Even with inlining disabled and loops pipelined with rewind, the tool could duplicate the number of instances of the function isolated for throughput efficiency. To avoid unwanted behaviour, Vitis HLS lets the developer specify how many instances of a function can be allocated inside the design. In case of the function `multiply`, this limitation can be indicated with the pragma:

```
#pragma HLS ALLOCATION function instances=multiply limit=1
```

This pragma doesn't work if the aforementioned precautions are not taken: if the outer loop gets pipelined without the rewind enabled, the inner loop gets unrolled, hence resources must be duplicated; if the outer loop does not get pipelined, the resources do not get duplicated, but the algorithm uselessly stalls to wait for the end of the inner loop;

finally, if the modules are not isolated through with inline disabled, unwanted duplications could be triggered for reasons like bit-width differences.

Now we can re-write the small loop analyzed in Algorithm 4.5 with all the pragmas and the functions inlined. The new optimized code is shown in Algorithm 4.7.

Algorithm 4.7 State Vector Update Optimized

```
1: #pragma HLS ALLOCATION function instances=multiply limit=1
2: #pragma HLS ALLOCATION function instances=macply limit=1
3:
4: for n = 0 to N − 1 do
5:   #pragma HLS PIPELINE rewind
6:
7:   mac ← 0
8:   for i = 0 to 6 do
9:     #pragma HLS PIPELINE
10:
11:    macply(mac, B[i], k[i][n])                ▷ mac ← B[i] · k[i][n]
12:  end for
13:  y_next[n] ← y_curr[n] + multiply(h, mac)     ▷ y_next[n] ← y_curr[n] + h · mac
14: end for
```

4.7. Operation Order Optimization

A way to optimize the code is to change the order of the operands without changing the final results. Consider the velocity derivative introduced in Equation 2.3, which I report here:

$$\dot{\mathbf{v}} = -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r}$$

This equation contains the cube of the norm of the position vector $\mathbf{r}$. If $\mathbf{r}$ is a three-dimensional vector, the explicit form of the cube of the norm is:

$$\|\mathbf{r}\|^3 = \left(\sqrt{r_1^2 + r_2^2 + r_3^2}\right)^3$$

For each operation, in general, to avoid truncation, the result bit-width is different than the inputs bit-width. When multiplying two values $z = x \cdot y$, the bit-width of the result z is the sum of the input's integer parts and their fractional part. Instead, the result of an addition $z = x + y$ will have the same configuration plus one bit in the integer part to store all possible values. Hence, the cube of the norm will need three times the original bit-width in the result to conserve the same precision.

To alleviate the precision problem, the order of the operations used for the derivative can be changed:

$$\dot{\mathbf{v}} = -\left(\frac{\mu}{\|\mathbf{r}\|^2}\right)\left(\frac{\mathbf{r}}{\|\mathbf{r}\|}\right)$$

In this case, the maximum needed number of bits to avoid truncating the result is given by the square norm $\|\mathbf{r}\|^2$. Say the vector $\mathbf{r}$ has I integer bits and F fractional bits, the medium-step bit-width needed will be $(2+2I+2F)$, since it is composed by a three-input addition of squared values.

4.8. Memory Constraints

The FPGA alone does not have the BRAM necessary to store all the information that will be given as output to the host. This board has 630kB of BRAM available in the FPGA and $512MB$ of external memory DDR3, connected with the FPGA and the CPU through the AXI protocol. As mentioned in Section 3.4.1, external memory requests are extremely inefficient, but they can be bursted. Hence, the ideal flow of data is to store on the BRAM the maximum storable information, work with it inside the FPGA (BRAM has indeed only 1cc latency), and then burst write the computed information back to the external memory.

Therefore, in my implementation I added a control to write back to global memory each time I reached the storable limit of the BRAM. A small pseudo-code snippet is shown in Algorithm 4.8

Algorithm 4.8 State Vector Update Optimized

```
1: while t_n < t_f do
2:     if BRAM full then
3:         Copy local content to external DDR
4:         Clean internal BRAM
5:     end if
6:     Store new state vector in BRAM
7: end while
```

4.9. Vivado Block Diagram

The Vivado IDE is a tool for Register Transfer Level (RTL) development using Hardware Description Languages (HDLs). The general flow of a development with Vivado follows these steps: it starts with a source code written in an HDL language (like Verilog or VHDL) simulated with a test bench code; if the simulation succeeds, the developer can

synthesize the design, where the resources consumption and an estimate of the timing get defined; during implementation the synthesized code gets placed and routed into the FPGA, highlighting any last possible issue; finally, the bitstream to be uploaded is generated from the implemented design.

The tool has also an IP Integrator section, where a block design can be visually generated dragging and dropping IP modules and connecting them to each other. At Figure 4.3 I am showing the final block design for my implementation. As seen in the HLS development

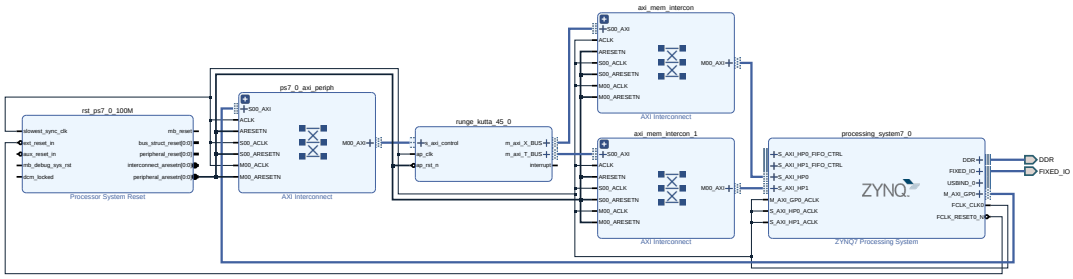
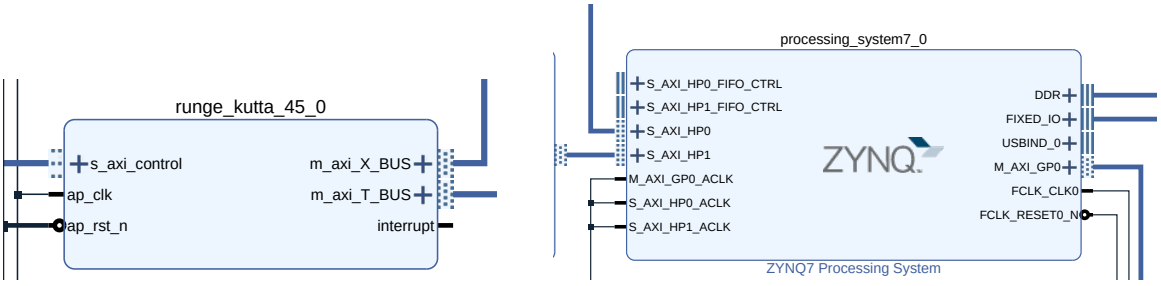


Figure 4.3: Block design

in Section 4.4, my design interfaces the CPU of the PYNQ-Z2 with two AXI HP ports and one AXI GP port. At Figure 4.4a you can see indeed three ports: `m_axi_X_BUS`, `m_axi_T_BUS` and `s_axi_control`. The first two ports are directly connected to the DDR through the HP ports of the PYNQ-Z2 (`s_axi_HP0` and `s_axi_HP1` of Figure 4.4b), whereas the last port is connected to the GP port of the PYNQ-Z2 (`m_axi_GP0` of Figure 4.4b).



(a) Module IP (b) Zynq PS IP

Figure 4.4: IP modules

Moreover, depending on the frequency indicated during synthesis, the IP Integrator tool lets the developer manually modify the `ap_clk` signal and the Phase-Locked Loop (PPL) of the PS which manages the interface with the PL. To avoid mismatching, the general clock and the PLL have to be set so that the PS interface runs at the same clock cycle of the PL module.

Once built the block design, Vivado auto-manages the wrapping of the code and the generation of new HDL code. Depending on the frequency set, the synthesis generates a design more or less rich of LUTs and FFs.

In this stage, the Vivado tool may generate a synthesis with a resource consumption much different than the one estimated with the HLS tool (usually higher), and sometimes the amount of resources chosen after the Vivado synthesis exceeds the limits of the device. In this case, the implementation is impossible and the design must be re-implemented starting from the HLS code. This is why it is very important to understand what is more or less happening under the hood already at HLS level. The more a developer knows what they are doing, the less time is wasted coming back and forth from and to this stage of the synthesis.

The last stage before generating the bitstream is the actual P&R (Place and Route). Here the amount of resources has already been chosen and the design is placed on the board. In this stage, the timing reports estimated during synthesis may change (usually they get worse), leading to a negative slack. A negative slack is the consequence of a design where the signals do not propagate properly in a clock cycle, leading to incorrect results. To tackle this problem without changing the design, the PL frequency could be lowered, but the entire process, starting from the IP generation in HLS, must be redone.

4.10. Python Interface

Once the bitstream gets generated, it is ready to be loaded into the board. As mentioned in Section 3.9, the PYNQ-Z2 does not need an external Host to be interfaced with, as the paradigm Host-FPGA is already hardwired into the board through the connection CPU-FPGA.

Once accessed the board, a small program in Python can be written in Jupyter notebook to test that the bitstream of the FPGA works properly. To upload the bitstream, a `bit` file and a `hwh` file must be imported to the notebook. If the desing is called for eaxmple "`design_1`", the bitstream and the description file are usually located here:

- `<project_name>/<project_name>.gen/sources_1/bd/design_1/hw_handoff/design_1.hwh`
- `<project_name>/<project_name>.runs/impl_1/design_1_wrapper.bit`

Once moved in a known location, obviously reachable from the application, the two files have to be renamed in the same way.

Once the bitstream is uploaded, the overlay library of the `pynq` module lets you get the IP object for further manipulation. To interface the FPGA with the DDR, the processor must firstly allocate some memory to the DDR, as depicted at Figure 3.9. In the Algorithm 4.9 there is a pseudo-code of this phase illustrated.

Algorithm 4.9 Jupyter pseudo-code

```
1: from pynq import Overlay
2: from pynq import allocate
3:
4: // Upload bitstream to the board
5: overlay ← Overlay("./design_1_wrapper.bit")
6: rk45_ip ← overlay.runge_kutta_45_0
7:
8: // Allocate space in the DDR and initialize it
9: buffer_y_FPGA ← allocate(( max_rows, N ), np.float64)
10: buffer_t_FPGA ← allocate(( max_rows, ), np.float64)
11:
12: buffer_y_FPGA[0] ← y_0                    ▷ y_0 = Initial state vector
13: buffer_t_FPGA[0] ← t_0
14:
15: // Write control signals to the AXI Lite input port
16: for each port address and its value do
17:   rk45_ip.write(address_port , LSB_value)
18:   rk45_ip.write(address_port + 0x04, MSB_value)
19: end for
20:
21: // Start IP run
22: rk45_ip.write(0x00, 1)
23: while rk45_ip.read(0x00) & 0x04 != 0x04 do
24:   wait
25: end while
26: buffer_y_FPGA.invalidate()
27: buffer_t_FPGA.invalidate()
28:
29: // Read from AXI Lite output ports
30: size = rk45_ip.mmio.read(address_size, length=4)
31: flag = rk45_ip.mmio.read(address_flag, length=4)
```

5 | Results

For analysis purposes, the algorithm was tested in Low Earth Orbit (LEO), Geostationary Transfer Orbit (GTO) and Interplanetary Orbit.

LEO refers to an orbit that is relatively close to Earth's surface, typically ranging from a few hundred kilometers up to about two thousands kilometers. Satellites in LEO complete orbits around the Earth relatively quickly, typically within a couple of hours, depending on the orbit semi-major axis, and have an eccentricity typically less than 0.25, resulting in an almost circular orbit. This is mainly due to the scientific and commercial objectives of such missions, which are mainly intended for Earth observations or communication purposes (e.g., Starlink constellation). In such cases circular orbits guarantee constant and stable observations and communication conditions. Currently, the majority of the satellites are in LEO, such as the International Space Station (ISS), which altitude oscillates between $413km$ (pericenter) and $422km$ (apocenter). The ISS orbit has a period of around one hour and thirty minutes.

GTO is an elliptical orbit that is used to transfer satellites from the initial launch stage to a geostationary orbit. Geostationary orbits are positioned around $35,786km$ above the equator and have the same rotational period as the Earth, making the satellites appear stationary relative to the Earth's surface. The inclination of such orbit is very low, and in most cases is within few degrees from the equatorial plane. GTO is a highly elliptical orbit, with eccentricity around 0.7. Its pericenter is typically around LEO and its apocenter as high as geostationary orbit [20]. Therefore, this kind of orbit is mainly populated by depleted launchers upper stages or transfer modules, even though also some telescope missions might exploit this kind of orbits.

Interplanetary orbits refer to the paths followed by objects traveling between planets within our solar system. These orbits are highly elliptical and have an orbital period in the order of years. I will test the system under these conditions using the orbit of the comet *67P/Churyumov-Gerasimenko* (abbreviated as 67P), seen by ESA's Rosetta spacecraft in 2014. Its orbit lasts 6.45 years and oscillates between a perihelion of 1.24AU and an aphelion of 5.68AU.

The eccentricities of the orbit influence the choice of the time-step during an adaptive propagation algorithm. They are defined in a range between 0 and 1, where 0 represents a circular orbit and the closer to 1, the more eccentric the orbit appears (Figure 5.1).

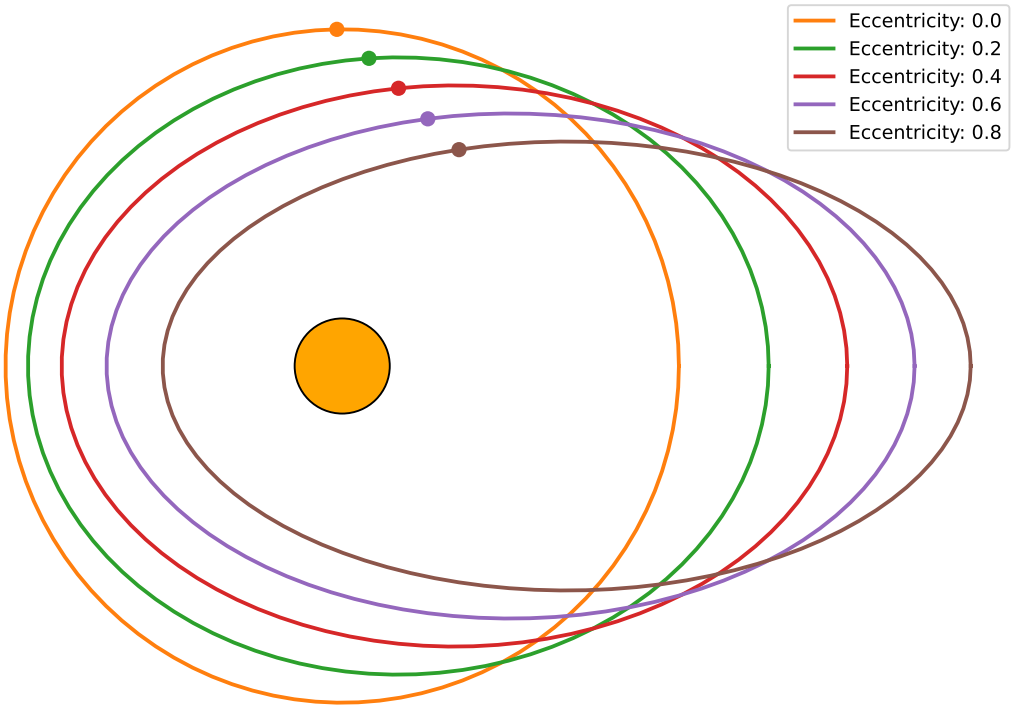


Figure 5.1: Elliptic orbit by eccentricity

LEO and GTO have been tested in the Earth-Centered Inertial (ECI) reference frame, and 67P in the Sun-Centered Inertial (SCI) reference frame. The initial values fed to the orbit propagator are the values at the pericenters. They are reported in Table 5.1.

To obtain the exact orbit of an object given the initial conditions and the gravitational parameter, it has been employed the MATLAB function `keplerUniversal`, mentioned in Section 4.3.

To compare the implemented algorithm with an already existent and well-tested baseline, the orbit has been propagated using the MATLAB function `ode45`[1]. The implemented algorithm is based on the same paper which the MATLAB implementation gets inspiration from [13]. The Butcher tableau used in the MATLAB implementation refers to the RK5(4)7M variant of the paper, (reported at Table 2.2). More implementation details are accessible in Section 4.2.

[1] mathworks.com/help/matlab/ref/ode45.html, Last accessed: July 2, 2023

Initial Values	LEO	GTO	67P
$\mathbf{r}_x$	$6,893.65\ km$	$6,054.31km$	$3.94 \times 10^{-6}\ AU$
$\mathbf{r}_y$	$607.77\ km$	$-3,072.04\ km$	$1.28\ AU$
$\mathbf{r}_z$	$1,052.69\ km$	$-133.12\ km$	$7.00 \times 10^{-2}\ AU$
$\mathbf{v}_x$	$-1.31\ km/s$	$4.65\ km/s$	$-6.90\ AU/year$
$\mathbf{v}_y$	$3.72\ km/s$	$9.19\ km/s$	$1.49\ AU/year$
$\mathbf{v}_z$	$6.44\ km/s$	$-0.62\ km/s$	$0.36\ AU/year$
Semi-major Axis	$7,000.00\ km$	$36,130.24\ km$	$3.46\ AU$
Eccentricity	1.00×10^{-8}	0.81	0.65
Inclination	$1.05\ rad$	$0.06\ rad$	$0.07\ rad$
Long. of Ascending Node	$0.00\ rad$	$2.36\ rad$	$0.63\ rad$
Argument of Pericenter	$6.66 \times 10^{-8}\ rad$	$3.46\ rad$	$0.39\ rad$
True Anomaly	$0.17\ rad$	$6.28\ rad$	$0.55\ rad$

Table 5.1: Initial state values

To compare two orbits I calculated the positions and velocities Euclidean distances for each epoch and I plotted them in a logarithmic graph. Given a target state vector $\mathbf{x} = (\mathbf{r}, \mathbf{v})^T$ and an estimated state vector $\hat{\mathbf{x}} = (\hat{\mathbf{r}}, \hat{\mathbf{v}})^T$, the Euclidean distances are defined as:

$$\begin{aligned} d(\mathbf{r}, \hat{\mathbf{r}}) &= \sqrt{\sum_{i=1}^{3} (r_i - \hat{r}_i)^2} \\ d(\mathbf{v}, \hat{\mathbf{v}}) &= \sqrt{\sum_{i=1}^{3} (v_i - \hat{v}_i)^2} \end{aligned} \tag{5.1}$$

5.1. Algorithm Reliability

As mentioned above, to assess the correctness of the general algorithm, which corresponds to the pseudo-code illustrated in Algorithm 4.1, I compared its distance from the exact orbit against the distance from the exact orbit of the MATLAB implementation.

In the next pages, I am reporting the distance error for each of the three orbits. At Figure 5.2 you can see the propagation error of the LEO orbit, at Figure 5.3 you can see the error of the GTO orbit, and at Figure 5.4 the one of the interplanetary orbit.

The Python implementation is indicated in red, and the MATLAB implementation is

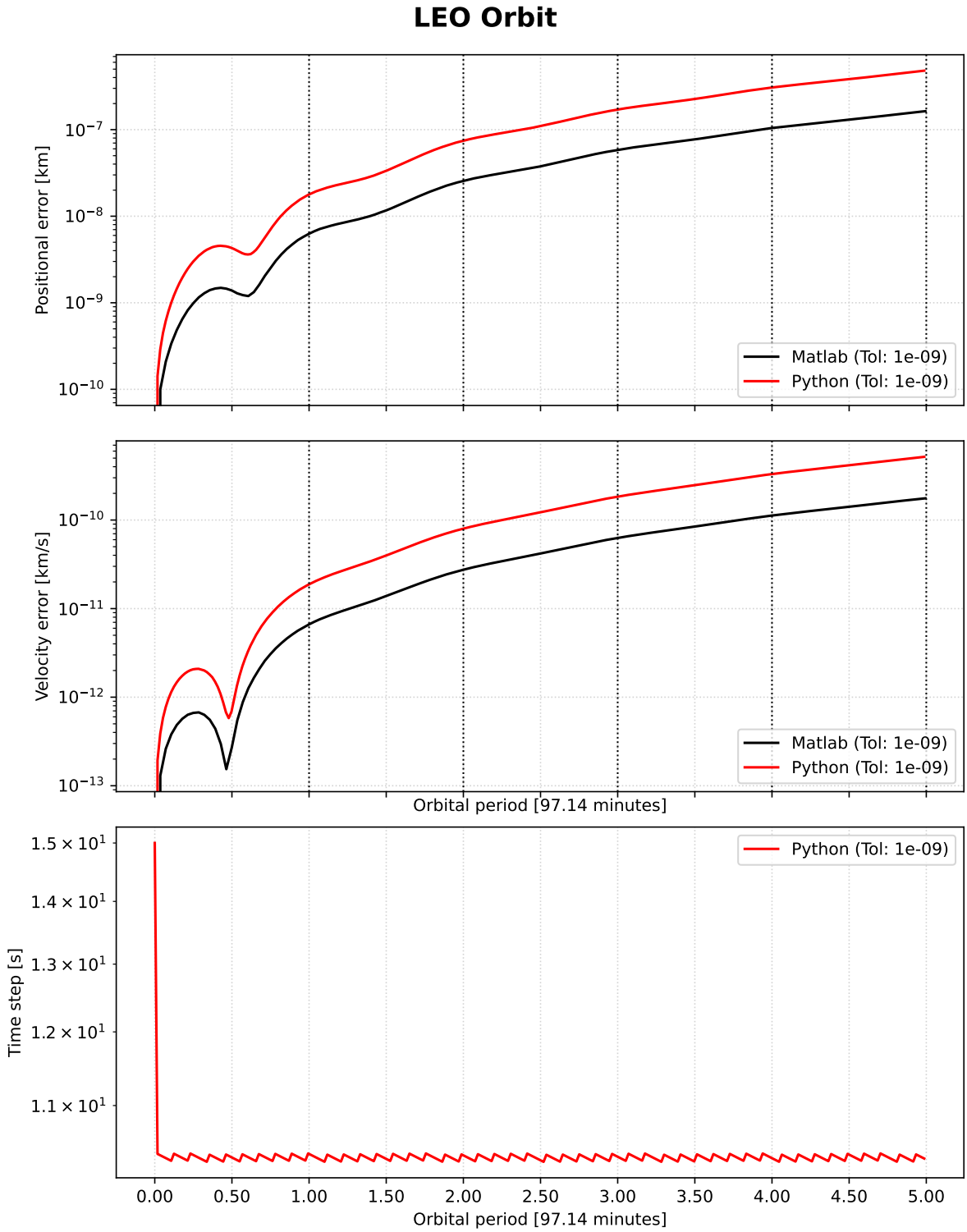


Figure 5.2: LEO propagation error with respect to `keplerUniversal`. The red line represents the error of the custom Python implementation, the black line represents the error of the MATLAB function `ode45`. The bottom graph represents the time step variation per time in the Python implementation.

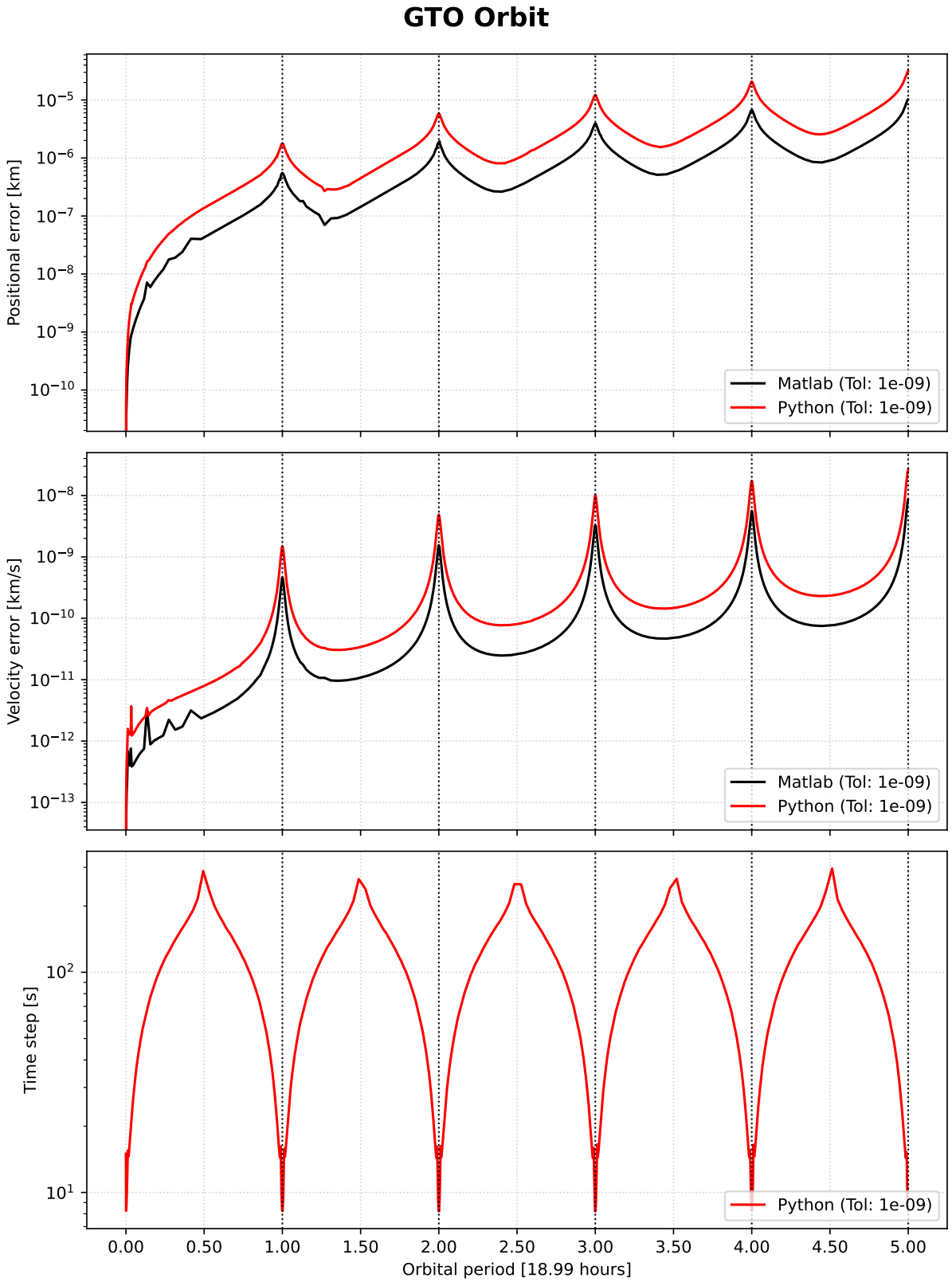


Figure 5.3: GTO propagation error with respect to `keplerUniversal`. The red line represents the error of the custom Python implementation, the black line represents the error of the MATLAB function `ode45`. The bottom graph represents the time step variation per time in the Python implementation.

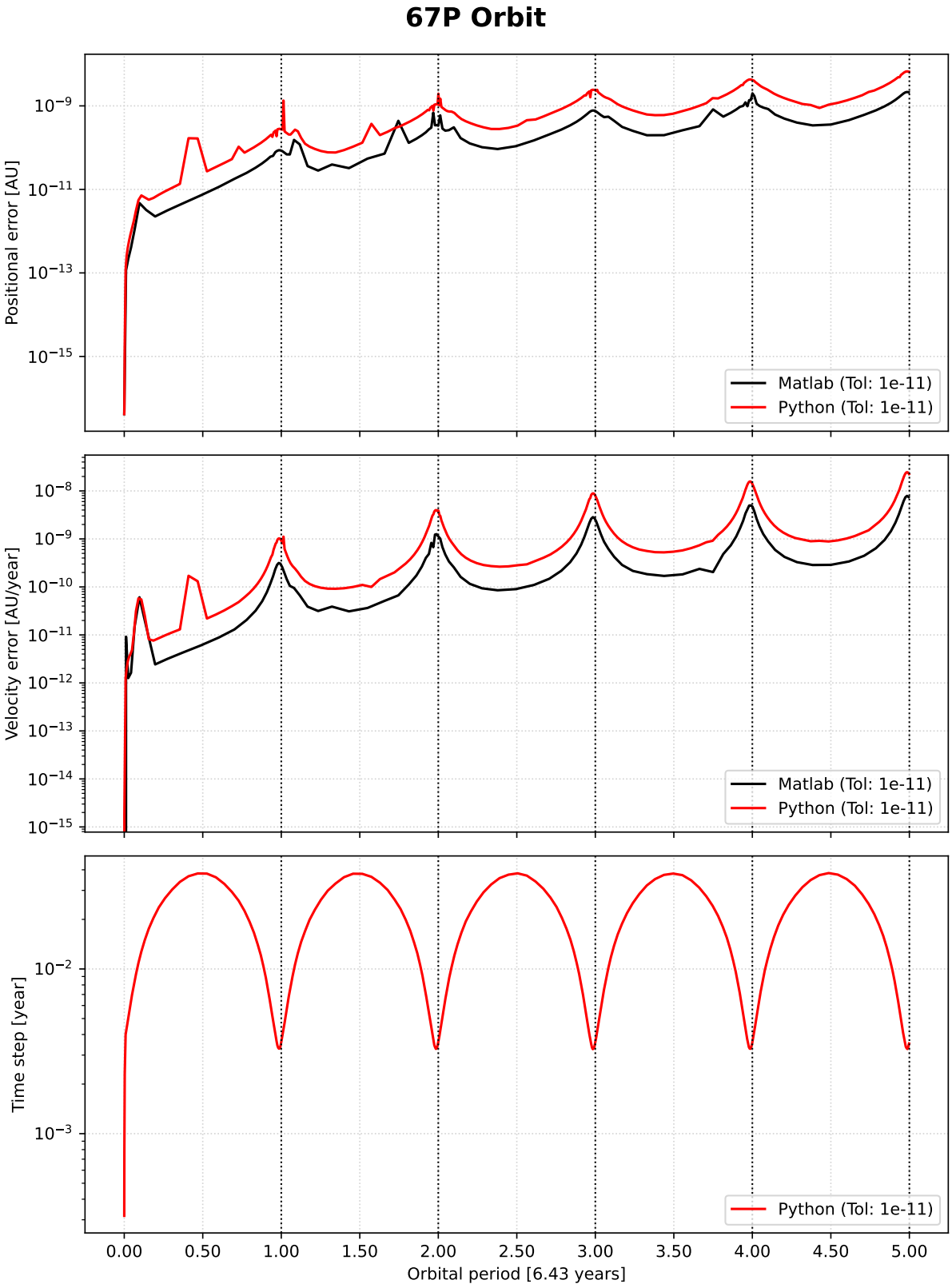


Figure 5.4: 67P propagation error with respect to `keplerUniversal`. The red line represents the error of the custom Python implementation, the black line represents the error of the MATLAB function `ode45`. The bottom graph represents the time step variation per time in the Python implementation.

indicated in black. The horizontal axis is normalized with the orbital period, for a better visualization of pericenter and apocenter, and the vertical axes are normalized with the unit of measure used in the propagation. Finally, the tolerance given in input to the LEO and GTO orbits is in the order of micrometers, instead for the last orbit is in the order of meters.

Both the error shapes of Python and MATLAB are similar, and they are both in the same order of magnitude. The visible difference in accuracy resides in a difference in the way the Local Truncation Error (LTE) gets calculated in the two implementations. As shown in Algorithm 4.1, the custom algorithm calculates the LTE as:

$$LTE = \left|\left| h \sum_{i=1}^{7} (b_i - b_i^*)\mathbf{k}_i \right|\right|$$

and then it compares it to the absolute tolerance given in input so that:

$$LTE \leq atol$$

Moreover, it is worth mentioning that the intermediate stage vectors $\mathbf{k}_i$ are composed by both position and velocity components. The two components are usually not of the same magnitude, hence, inside a norm, the smaller one will give a negligible weight to the LTE. To avoid this problem, it is important to normalize the inputs so that the two components will have similar weights in the norm calculation, as mentioned in Section 2.8.

In the graphs, it is also reported how the value of the time step h varies with time. As mentioned in Section 4.2, the orbital velocity is very high at the pericenter and smaller at the apocenter, hence the time step is larger at the apocenter and smaller at the pericenter. The automatic step-size control is therefore tuning the integration step to preserve the target step tolerance. Knowing that the starting point in these simulations is the pericenter, the time step behaviour mirrors the expectations. The LEO orbit is known for being almost circular, with very small eccentricity. In fact, the time step stays constant during the whole orbital period, oscillating around 10 seconds.

5.2. HLS Synthesis

One of the most complex functions to optimize was the acceleration in the equations of motion (Equation 2.3):

$$\dot{\mathbf{v}} = -\frac{\mu}{\|\mathbf{r}\|^3}\mathbf{r} := f(\mathbf{r})$$

In Runge-Kutta, this equation is applied when calculating the velocity components of the intermediate stage $\mathbf{k}_i$. The intermediate step update for the velocity components of the two body problem, as explicitly shown in Algorithm 4.1, is implemented as:

$$\mathbf{k}_{\mathbf{v},i} = f\left(\mathbf{r} + h\sum_{j}^{i-1} a_{ij}\mathbf{k}_{\mathbf{v},j}\right)$$

where $\mathbf{r}$ is the vector with the position components of the state vector, and $\mathbf{k}_{\mathbf{v},i}$ is the vector with the velocity components of the intermediate stages.

For implementation purposes, the sum can just be considered a vector of constants, called:

$$\mathbf{c}_\mathbf{r} = h\sum_{j}^{i-1} a_{ij}\mathbf{k}_{\mathbf{v},j} \tag{5.2}$$

As a result, the velocity components of the intermediate stage are obtained by:

$$\mathbf{k}_{\mathbf{v},i} = -\mu\frac{\mathbf{r} + \mathbf{c}_\mathbf{r}}{||\mathbf{r} + \mathbf{c}_\mathbf{r}||^3} := f(\mathbf{x} + \mathbf{c}_\mathbf{r})$$

As explained in Section 4.7, for accuracy purposes, this equation is implemented as:

$$\mathbf{k}_{\mathbf{v},i} = -\left(\frac{\mu}{\|\mathbf{r} + \mathbf{c}_\mathbf{r}\|^2}\right)\left(\frac{\mathbf{r} + \mathbf{c}_\mathbf{r}}{\|\mathbf{r} + \mathbf{c}_\mathbf{r}\|}\right) \tag{5.3}$$

Here, the loops for calculating the squared sums of the norms have been pipelined, and only one external module has been created to be used by both the divisions. As shown later on in Table 5.2, the division consumes indeed a lot of resources, and it also takes a huge amount of clock cycles to complete.

As mentioned in Section 4.6, HLS transforms code into RTL modules with certain hierarchies. By disabling the inlining feature and controlling the number of allocations for each function, the compiler can be commanded to not duplicate some resources. In the design, three total single modules have been created: `multiply`, `macply` and `division`. As I just mentioned aforementioned, divisions take a lot of resources and a lot of time, hence the dedicated module. Multiplications and MAC operations, instead, are very convenient to implement with Digital Signal Processing (DSP) blocks, which are a limited number inside the PYNQ-Z2. These operations are very frequent inside the kernel function, but only one big module has been created to accommodate any precision, and it gets called multiple times in a pipelined fashion. Overall, the hierarchy level of the RTL modules is shown at Figure 5.5.

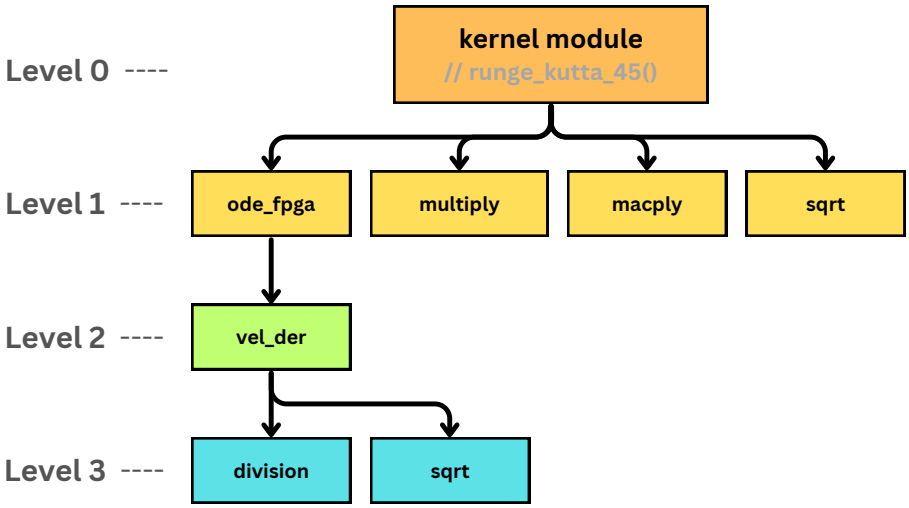


Figure 5.5: RTL modules hierarchy

At Table 5.2 are reported the estimations of resource utilization of the most relevant modules after HLS synthesis, and their estimated clock cycle latency. First of all, this table

Module	LUT	FF	DSP	Latency (cc)
macply	294	0	15	–
multiply	14,833	0	46	–
ode_fpga	18,347	4,082	61	1,293
vel_der	17,456	1,998	61	1,251
division	1,549	1,101	0	199
vel_der_sqrt	530	453	0	91
kernel_sqrt	530	453	0	91
total (kernel)	**55,170**	**11,320**	**122**	–

Table 5.2: Vitis HLS resource utilization estimation

reports the resources of the code only after some optimizations have been applied. For instance, when a single division module was not yet created, the compiler used to synthesize the two divisions separately, duplicating RTL code. This resulted in approximately 6000 LUTs employed for each division, leading to roughly 12k LUTs necessary, which are more than the 20% of the available resources on the PYNQ-Z2.

The square root implementation was not isolated, as it consumes a negligible amount of

resources. In this case, the compiler generated two instances of a module `sqrt`.

Lastly, the MAC operations and the multiplications were so redundant which it was a great waste of resources duplicating them. Moreover, given the large bit-width used in this application, the implementation of `multiply` costs 46 DSP blocks, and the implementation of `macply` costs 15 DSPs. This is why one module had to be big enough to handle the worst case scenario. In case of a multiplication, both the integer parts and the fractional parts of the result should be equal to the sum of both the inputs ($I_R = I_1 + I_2, F_R = F_1 + F_2$). Instead, for an addition, the result must have an integer bit-width as the largest input integer part plus 1 bit ($I_R = \max\{I_1, I_2\} + 1$).

Given the limited bit-width, the user has to be aware of the following constraints:

- At any moment during the computation, the intermediate stages $\mathbf{k}_i$, the constants $\mathbf{c}$ defined in Equation 5.2, and the single elements of the local truncation error vector $\mathbf{e} = h\sum_{i=1}^{7}(b_i - b_i^*)\mathbf{k}_i$ must be representable with a signed fixed point configured with $I = 30$ and $F = 55$.
- The minimum representable value is $2^{-F} = 2^{-55} \approx 2.78 \times 10^{-17}$. I tested a minimum tolerance of 10^{-11}.

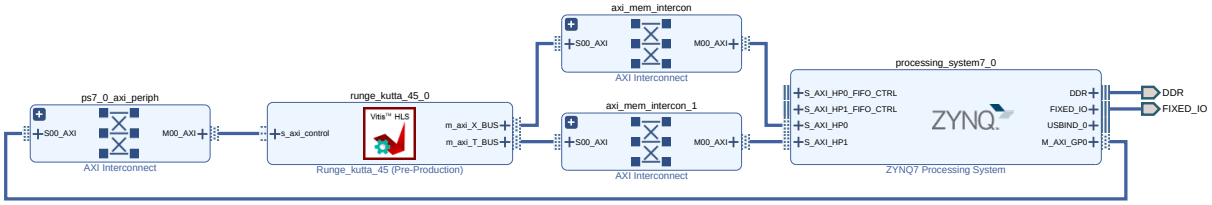
Because of these constraints, it is better to normalize the values so that position and velocity vectors are reasonably comparable, and so that all the possible values of the orbit can be represented in the system.

5.3. IP Block Design Integration

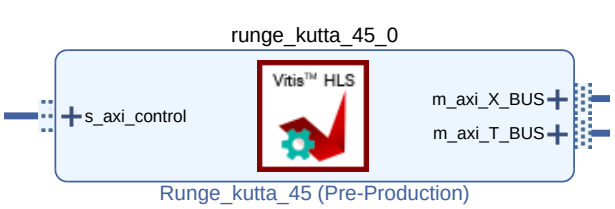
After exporting the RTL design, it has to be integrated with the Zynq Processing System (PS) of the board. This step is done in Vivado, which I introduced in Section 4.9.

Vivado is an IDE for writing RTL instructions in an HDL language like Verilog or VHDL, but it offers several automation flows that can be used when integrating an HLS code exported from Vitis HLS. In the Flow Navigator of Vivado there is a section called "IP Integrator", which makes easy for a designer to create a block design.

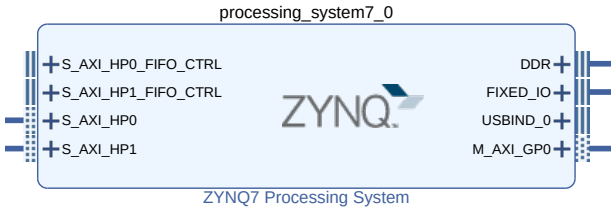
In this case, it was only needed to interface the IP with the Zynq PS through the AXI interface, which has been thoroughly declared and customized at the HLS level. An overview of the final block design is depicted at Figure 5.6a. In this configuration, three ports must be connected to the PS: the slave AXI port and the two master AXI ports. The generated IP uses AXI4, instead the Zynq PS uses the AXI3 protocol. This is one of the reasons why we need a built-in Vitis HLS IP called AXI Interconnect between the



(a) Bock Design Overview



(b) Kernel Module



(c) Zynq PS IP

Figure 5.6: Block design using Vivado IP Integrator

ports. As mentioned in Section 3.4.1, the AXI Interconnect has also tha ability to modify the ports bit-width PL-side, yielding to a more flexible behaviour. In the kernel module, the slave port is the control port, which is piloted by the Host, instead the master ports are data buses dedicated to the array of state vectors $\mathbf{X}$ and the array of time instants $\mathbf{t}$.

I connected the control port to the first master General Purpose port of the PS, and the two IP master ports to the slave High-Performance ports of the PS. As shown in the architecture depicted at Figure 3.3, the Memory Interconnect inside of the Zynq has direct access to the DDR through two master ports.

After connecting the interfaces, the clock of the system must be correctly set to the same frequency as the one set for the kernel module, changing the value of the IO PLL.

Once these precautions are taken into account, the synthesis and the implementation can be run, and finally the bitstream can be generate. Then, after the bitstream has been generated, Vivado shows in a report the estimated timings and the resource utilization. At Table 5.3 is shown the total resource utilization of the design after implementation. Even though in the design there was almost nothing parallelized, the resource utilization in this board is still pretty heavy, mostly due to the large bit-width of the operands.

Vivado also reports an estimate of the power consumption of the chip, reported here in Table 5.3, and it also checks if the timing constraints defined in the design are met. In particular, the Worst Negative Slack (WNS) reported in Table 5.3 is the smallest difference between the end of the needed computation in one clock cycle and the end of the clock

Utilization	LUT	FF	BRAM	DSP
Resources	25,862	15,357	225	141
Percentage	48.61%	14.43%	80.36%	64.09%
Power	Tot. Est. On-Chip Power: 1.473 W			
Timing	Worst Ngative Slack: 27.162 ns			

Table 5.3: Resource utilization after design implementation.

cycle itself. If the WNS is negative, it means that the design takes more time than the clock cycle to finish propagating all the signals.

A second configuration has been synthesized, changing the module hierarchy and generating therefore the least amount of duplications possible, hence the least amount of resources. Although, the timings after implementation were not met. The results are reported at Table 5.4. A failure for timing could be solved either changing the HLS logic,

Utilization	LUT	FF	BRAM	DSP
Resources	34,979	27,111	220	122
Percentage	65.75%	25.48%	78.57%	55.45%
Power	Tot. Est. On-Chip Power: 1.464 W			
Timing	Worst Negative Slack: $-6.511\ ns$			

Table 5.4: Failed synthesis implementation. The timing constraints are too stringent for the complexity of the logic.

or reducing the clock frequency of the PL. Considering that the gain in resources and power consumption was negligible, the chosen solution has been to change the HLS logic rather than the PL frequency.

5.4. Power Consumption

To check the real power consumption of the board, a USB Power Meter[2] has been used. Given that the board power supply is a USB cable, the USB Power Meter can be put in series to the power supply, as it uses a very low resistance to measure the current drained by the board. The USB Power Meter employed here has a resistance of $R = 970\ \mu\Omega$.

[2]https://www.amazon.com/gp/product/B07FMQZVW2/ref=ppx_yo_dt_b_asin_title_o07_s00?ie=UTF8&psc=1, Last accessed: July 2, 2023

In Table 5.5 the measures taken while running the board are reported. It has been reported

Application Status	Time	Current	Power	Energy
Idle State	–	280 mA	1.393 W	–
LEO CPU	125.39 s	303 mA	1.511 W	189.46 J
LEO FPGA	30.03 s	303 mA	1.511 W	45.38 J
GTO CPU	251.19 s	303 mA	1.511 W	379.55 J
GTO FPGA	56.57 s	303 mA	1.511 W	85.48 J
67P CPU	98.34 s	305 mA	1.511 W	148.59 J
67P FPGA	23.98 s	305 mA	1.511 W	36.23 J

Table 5.5: Board power consumption for 67P orbit

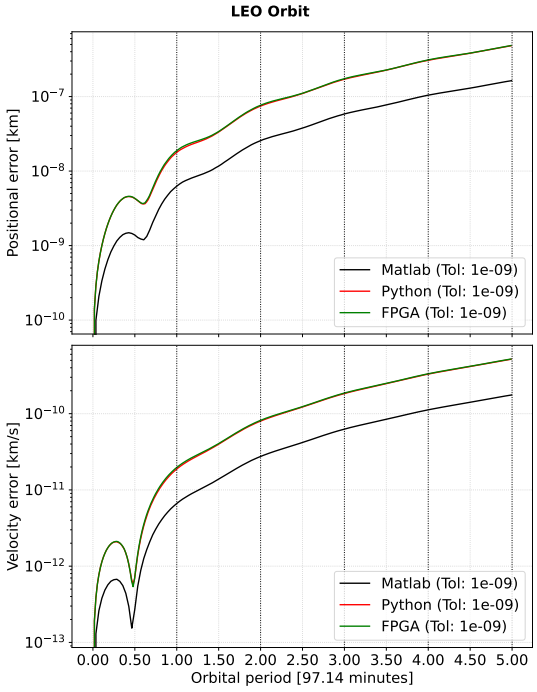
the power drained by the whole board, as the PYNQ-Z2 does not have the PMBus, which is a chip often included on development boards to measure power[3]. Although the power drained to run the ARM CPU or the Artix FPGA is the same, the time taken by the FPGA is much shorter than its CPU counterpart. As a result, the total energy consumed by the FPGA is less than the energy employed by the CPU.

5.5. FPGA Results

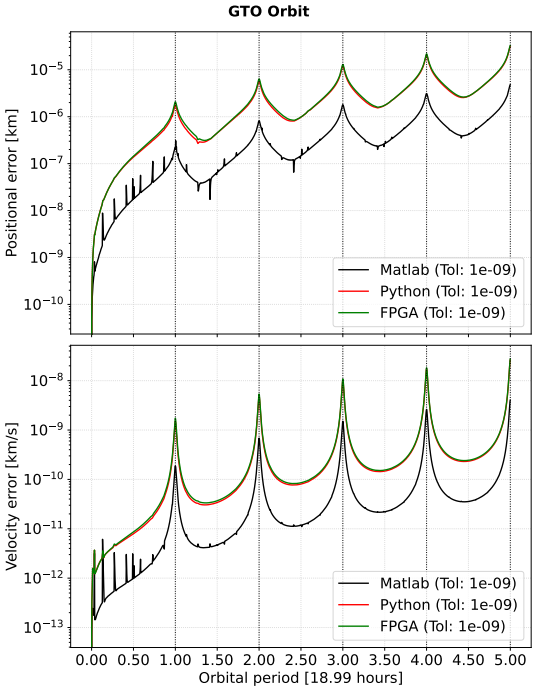
Once run all the three testing orbits, the euclidean distances from the MATLAB orbit and from the Python-computed orbit for both the position and velocity vectors have been plotted, as explained in the introduction of Section 5.

At Figure 5.7a there are the distances from the baseline for each orbit, starting from LEO in the top-left and finishing with the 67P orbit at the bottom. The Python orbit is indicated in red, but it cannot be seen as the FPGA orbit perfectly overlaps the Python one. The three results show the adaptability of the code in both low and high eccentric orbits, denoting a high precision even with very low tolerances (10^{-11} for the interplanetary orbit).

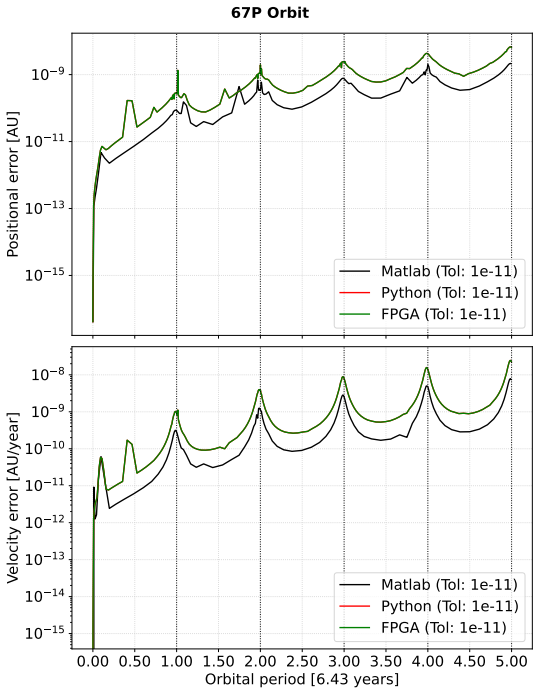
[3]discuss.pynq.io/t/how-to-measure-power-consumption/523/6, Last accessed: July 2, 2023



(a) LEO error



(b) GTO error



(c) 67P error

Figure 5.7: FPGA error from baseline orbit. On the top-left corner, the results for LEO are shown. On the top-right corner, the GTO results are indicated, and on the bottom the 67P orbit error is depicted. The Python orbit is indicated in red, but it cannot be seen as the FPGA orbit (in green) perfectly overlaps the Python one. The black orbit represents the MATLAB implementation.

6 | Conclusion

In this thesis, it has been analyzed the efficiency of implementing in the FPGA PYNQ-Z2 the numerical method Runge-Kutta 5(4)7M. This algorithm can be used for orbit propagation on board of spacecrafts in LEO or deep-space environments.

In the preceding description, a comprehensive outline has been provided regarding the work flow necessary to write high-level synthesis code with the Vitis HLS IDE, highlighting advantages and disadvantages with respect to direct High-Description Language (HDL) programming. It has also been shown how relatively fast it is to design a system with HLS, without the need of caring too much about low-level details. In fact, there was no need to write AXI interfaces, and complex operations have been automatically synthesized by the software in the correct RTL modules. For example, the DSP blocks of this board have multipliers 18×25, which is not enough for the bit-width used. Hence, several DSPs have to be cascaded together, which is is a task completely automated in the HLS compilation. On the other hand, designers do not have much control on what is happening under the hood, leading to some sub-optimized constructs.

It has been proved negligible difference between this implementation and MATLAB's `ode45` implementation, and a comparison between the final FPGA propagation and the baseline orbit proved correctness and consistency after several revolutions.

Finally, the final resource and power consumption of the implemented design have been shown, comparing the ARM CPU of the board against its FPGA. Although the power drained is the same, the overall energy consumption of the FPGA is one order of magnitude less than its CPU counterpart. Moreover, the idle power consumption of the board is around 1.393 W, which is less than the CubeSats power consumption range. However, it should be taken into account that this board is not resistant to space environments, and that resistant FPGAs may have different results.

6.1. Future Work

As seen in the results, the resource constraints of this FPGA make it hard to parallelize more than one propagation at a time, as it is a very affordable board, with a price around \$133[1]. Although, there are FPGAs on the market with a lot more resources, where more advanced applications could apply.

The RT PolarFire FPGA[2] is an example of a product with more resources and resistant in space. A comparison is shown at Table 6.1.

FPGA Name	LE (LUT+DFF)	DSP	RAM	Price
PYNQ-Z2	85,000	220	630 kB	≈ \$130
RT PolarFire	481,000	1,480	33 MB	≈ \$20,000[3]

Table 6.1: RT PolarFire vs PYNQ-Z2

Using more performing FPGAs could enable parallel computations, reducing the time needed for a single propagation or allowing multiple propagations, building a pool of different results at the same time. For example, by selecting an FPGA capable of performing at least seven simultaneous propagations - one for the nominal solution and the others for the propagation of the sigma points - it becomes feasible to implement an Unscented Transform scheme. As mentioned in Section 2.4.1, the Unscented Transform is a method for predicting means and covariances in nonlinear systems, which is crucial in applications such as orbit reconstruction and determination.

Moreover, new reference frame systems may be employed to simplify internal operations and reduce the resource-hungry ones like divisions and square roots.

For analysis purposes, initial value perturbations can simulate the inaccuracy of the measurement systems when assessing the initial state of the spacecraft. With more resources, several different initial states can be propagated in parallel, and the propagated inaccuracy can be derived from them.

If it is necessary a lower-level control over the operations of the algorithm, new designs using the CORDIC (COordinate Rotation DIgital Computer) may be implemented. CORDIC is an iterative algorithm that uses simple shift and add operations to approx-

[1] uk.farnell.com/search?st=tul-corporation, Last accessed: July 2, 2023

[2] microchip.com/en-us/products/fpgas-and-plds/radiation-tolerant-fpgas/rt-polarfire-fpgas, Last accessed: July 2, 2023

imate mathematical functions like trigonometric functions, vector rotations, and logarithmic calculations. One of the significant advantages of the CORDIC algorithm is its simplicity and regularity, which makes it well-suited for hardware implementations. It is widely used in applications where hardware resources are constrained or where high-speed computation is required [2].

Another way to speed up computation may be to implement difficult operations such as squares and roots in a designated hardware, like an external calculator, and the easiest but repetitive computations in FPGA. In this case, the interface between external device and FPGA is usually the bottleneck, hence, a thorough analysis of the advantages and disadvantages is needed.

With Vitis HLS, developers can also program some specific functions in Verilog, interfacing the resulting IP blocks with the HLS code. This may be useful when low-level control is needed in small local parts of the algorithm.

Finally, new rad-hard and space-graded FPGAs can be used to simulate the algorithm and assess its correctness and accuracy. The use of other types of devices is mostly important for analysing the difference in efficiency with different hardware features, such as more resources or a radiation hardened device.

To conclude, let's recall the fundamental research question:

What are the advantages and potential improvements in accuracy and efficiency of orbit propagation in satellites when employing FPGA technology?

We have seen how the FPGAs are extremely flexible in terms of programmability and they can be tuned with respect to the goal of the mission. In general, they exhibit superior performance compared to CPUs while maintaining equivalent power consumption, and they may be programmed without writing HDL code, with the help of advanced HLS tools. In general, FPGAs consume less power while achieving very high performance, and can be easily programmed without too many low-level details.

Bibliography

[1] C. Adams, J. Parker, and D. Cotten. A hardware accelerated computer vision library for 3d reconstruction onboard small satellites. In 2021 IEEE Aerospace Conference (50100), pages 1–14. IEEE, 2021.

[2] R. Andraka. A survey of cordic algorithms for fpga based computers. In Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA '98, page 191–200, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919785. doi: 10.1145/275107.275139. URL https://doi.org/10.1145/275107.275139.

[3] E. Andreis, V. Franzese, and F. Topputo. Onboard orbit determination for deep-space cubesats. Journal of Guidance Control and Dynamics, 2022. doi: 10.2514/1.g006294.

[4] ARM. Amba axi and ace protocol specification, February 2013. URL https://developer.arm.com/documentation/ihi0022/latest/. Last accessed: July 2, 2023.

[5] S. S. Arnold, R. Nuzzaci, and A. Gordon-Ross. Energy budgeting for cubesats with an integrated fpga. In 2012 IEEE Aerospace Conference, pages 1–14. IEEE, 2012.

[6] S. Bhaskaran. Autonomous navigation for deep space missions. In SpaceOps 2012, page 1267135. 2012.

[7] S. Bhaskaran, J. Riedel, S. Synnott, and T. Wang. The deep space 1 autonomous navigation system-a post-flight analysis. In Astrodynamics Specialist Conference, page 3935, 2000.

[8] G. Bonin, N. Roth, S. Armitage, J. Newman, B. Risi, and R. E. Zee. Canx–4 and canx–5 precision formation flight: Mission accomplished! 2015.

[9] J. C. Butcher. A history of runge-kutta methods. Applied numerical mathematics, 20(3):247–260, 1996.

[10] M. Cirelli, G. Corcella, A. Hektor, G. Hütsi, M. Kadastik, P. Panci, M. Raidal, F. Sala, and A. Strumia. Pppc 4 dm id: a poor particle physicist cookbook for dark

matter indirect detection. Journal of Cosmology and Astroparticle Physics, 2011(03): 051, mar 2011. doi: 10.1088/1475-7516/2011/03/051. URL https://dx.doi.org/10.1088/1475-7516/2011/03/051.

[11] S. D'Amico, J.-S. Ardaens, and R. Larsson. Spaceborne autonomous formation flying experiment on the prisma mission. Journal of Guidance Control and Dynamics, 2011. doi: 10.2514/1.55638.

[12] K. J. DeMars and M. K. Jah. Probabilistic initial orbit determination using gaussian mixture models. Journal of Guidance, Control, and Dynamics, 36(5):1324–1335, 2013.

[13] J. Dormand and P. Prince. A family of embedded runge-kutta formulae. Journal of Computational and Applied Mathematics, 6(1):19–26, 1980. ISSN 0377-0427. doi: https://doi.org/10.1016/0771-050X(80)90013-3. URL https://www.sciencedirect.com/science/article/pii/0771050X80900133.

[14] D. M. Fleetwood, P. S. Winokur, and P. E. Dodd. An overview of radiation effects on electronics in the space telecommunications environment. Microelectronics Reliability, 40(1):17–26, 2000.

[15] C. T. Fraser and S. Ulrich. Adaptive extended kalman filtering strategies for spacecraft formation relative navigation. Acta Astronautica, 178:700–721, 2021. ISSN 0094-5765. doi: https://doi.org/10.1016/j.actaastro.2020.10.016. URL https://www.sciencedirect.com/science/article/pii/S009457652030610X.

[16] G. Gaias and J.-S. Ardaens. Flight demonstration of autonomous noncooperative rendezvous in low earth orbit. Journal of Guidance Control and Dynamics, 2017. doi: 10.2514/1.g003239.

[17] A. D. George and C. M. Wilson. Onboard processing with hybrid and reconfigurable computing on small satellites. Proceedings of the IEEE, 106(3):458–470, 2018. doi: 10.1109/JPROC.2018.2802438.

[18] R. Ginosar. Survey of processors for space. Data Systems in Aerospace (DASIA). Eurospace, pages 1–5, 2012.

[19] Z. Hao, R. A. Shyam, A. Rathinam, and Y. Gao. Intelligent spacecraft visual gnc architecture with the state-of-the-art ai components for on-orbit manipulation. Frontiers in Robotics and AI, 8:639327, 2021.

[20] G. Janin. Lifetime of objects in geostationary transfer orbit. In B. Battrick, editor, Proceedings of the 2nd International Space Debris Re-entry Workshop. Euro-

pean Space Agency (ESA), 1991. URL https://conference.sdo.esoc.esa.int/proceedings/isdrw02/paper/10.

[21] S. Julier. The scaled unscented transformation. In Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301), volume 6, pages 4555–4559 vol.6, 2002. doi: 10.1109/ACC.2002.1025369.

[22] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering, 82(1):35–45, 03 1960. ISSN 0021-9223. doi: 10.1115/1.3662552. URL https://doi.org/10.1115/1.3662552.

[23] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev. Why the monte carlo method is so important today. Wiley Interdisciplinary Reviews: Computational Statistics, 6 (6):386–392, 2014.

[24] W. Lie and W. Feng-Yan. Dynamic partial reconfiguration in fpgas. In 2009 Third International Symposium on Intelligent Information Technology Application, volume 2, pages 445–448. IEEE, 2009.

[25] A. Milani and G. Gronchi. Theory of orbit determination. Cambridge University Press, 2010.

[26] B. M. Moghaddam and R. Chhabra. On the guidance, navigation and control of in-orbit space robotic missions: A survey and prospective vision. Acta Astronautica, 184:70–100, 2021.

[27] E.-S. Park, S.-Y. Park, K.-M. Roh, and K.-H. Choi. Satellite orbit determination using a batch filter based on the unscented transformation. Aerospace Science and Technology, 14(6):387–396, 2010. ISSN 1270-9638. doi: https://doi.org/10.1016/j.ast.2010.03.007. URL https://www.sciencedirect.com/science/article/pii/S1270963810000428.

[28] R. Pitonak, J. Mucha, L. Dobis, M. Javorka, and M. Marusin. Cloudsatnet-1: Fpga-based hardware-accelerated quantized cnn for satellite on-board cloud coverage classification. Remote Sensing, 14(13):3180, 2022.

[29] J. C. Refsgaard, J. P. van der Sluijs, A. L. Højberg, and P. A. Vanrolleghem. Uncertainty in the environmental modelling process–a framework and guidance. Environmental modelling & software, 22(11):1543–1556, 2007.

[30] E. A. E. Safadi, O. Adrot, and J.-M. Flaus. Advanced monte carlo method for model uncertainty propagation in risk assessment. IFAC-PapersOnLine, 48(3):529–534, 2015. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.

2015.06.135. URL https://www.sciencedirect.com/science/article/pii/S2405896315003742. 15th IFAC Symposium onInformation Control Problems in-Manufacturing.

[31] satsearch. An overview of on-board computer (obc) systems available on the global space marketplace. Blog post, 2020. URL https://blog.satsearch.co/2020-03-11-an-overview-of-on-board-computer-obc-systems-available\-on-the-global-space-marketplace. Last Accessed: July 2, 2023.

[32] B. Segret, D. Hestroffer, G. Quinsac, M. Agnan, and J. Vannitsen. On-board orbit determination for a deep space cubesat. In International Symposium on Space Flight Dynamics, 2017.

[33] L. F. Shampine and M. W. Reichelt. The matlab ode suite. SIAM journal on scientific computing, 18(1):1–22, 1997.

[34] H.-N. Shou. Orbit propagation and determination of low earth orbit satellites. International Journal of Antennas and Propagation, 2014, 2014.

[35] C. L. Thornton and J. S. Border. Radiometric tracking techniques for deep-space navigation. John Wiley & Sons, 2003.

[36] Xilinx Inc. Ug474 (v1.8) - 7 series fpgas configurable logic block, September 2016. URL https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB. Last accessed: July 2, 2023.

[37] Xilinx Inc. Ug479 (v1.10) - 7 series dsp48e1 user guide, March 2018. URL https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1. Last accessed: July 2, 2023.

[38] Xilinx Inc. Ug473 (v1.14) - 7 series fpgas memory resources, July 2019. URL https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources. Last accessed: July 2, 2023.

[39] Xilinx Inc. UG585 (v1.13) - Zynq-7000 SoC Technical Reference Manual, April 2021. URL https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Last accessed: July 2, 2023.

[40] Xilinx Inc. Ug1399 (v2022.1) - vitis high-level synthesis user guide, June 2022. URL https://docs.xilinx.com/r/en-US/ug1399-vitis-hls. Last accessed: July 2, 2023.

[41] M. Yang, B. Liu, J. Gong, H. Liu, H. Hu, Y. Dong, L. Shi, Y. Zhao, and Z. Miao.

Architecture design for reliable and reconfigurable fpga-based gnc computer for deep space exploration. Science China Technological Sciences, 59:289–300, 2016.

List of Figures

List of Tables

Acknowledgements

Vorrei riservare questo spazio finale della mia tesi di laurea ai ringraziamenti verso tutti coloro che hanno contribuito, con il loro instancabile supporto, alla realizzazione della stessa.

Ringrazio il mio relatore Dr. Morselli e il mio correlatore Di Domenico, che in questi mesi mi hanno dedicato tanto tempo e pazienza. Ringrazio anche il Professor Topputo per avermi dato l'opportunità di collaborare con il suo dipartimento e approfondire pertanto le mie conoscenze da un diverso punto di vista.

Ringrazio anche tutti coloro che in questi anni mi hanno sostenuto durante il percorso di laurea, dentro e fuori le mura universitarie.

Uno speciale ringraziamento va ai miei genitori, che vorrei potessero comprendere quanto importanti e di supporto sono stati per me in questi anni. Sono consapevole dei loro numerosi sacrifici, e spero che un giorno io possa a mia volta dimostrare tale generosità al prossimo.

Mia sorella si merita un paragrafo tutto per lei, considerando quanto per me sia stato un pilastro. Soprattutto in questi ultimi due anni di lontananza, lei ha saputo dimostrarmi molto più amore di quanto io sia anche solo in grado di esprimere, quindi davvero grazie.

Impossibile dimenticarsi dei miei più cari amici Richi, Fil e Paffa. Fanno parte della mia famiglia ormai. Sono davvero stato fortunato con voi. La vostra mera esistenza per me è stata fondamentale in questo percorso.

Ci tengo a ringraziare anche i miei coinquilini e amici Pietro e Giacomo. Sono stati testimoni delle varie fasi di delirio durante la stesura, hanno cucinato per me quando non avevo tempo e mi hanno aiutato con degli aggiustamenti quando non sapevo che pesci pigliare. Direi che possano essere definiti dei coinquilini modello.

Infine vorrei ringraziare me stesso. Mi ringrazio per aver creduto in me, per non aver mai mollato, neanche nei momenti peggiori. Mi ringrazio per il duro lavoro, per le notti in bianco, per i weekend a studiare, e per il mio carattere particolarmente testardo.