FPGA-Based Hardware Acceleration of Numerical Theoretic Transform with RISC-V ISA Customization

Abanoub Emad Hanna

Abdelrhman Emad Fathy

Ahmed Nader Ahmed

Mahmoud Mohamed Soliman

Omar Mahmoud Gabr

Omar Mohamed Afifi

Supervisors: Dr. Hesham Omran & Dr. Hossam Hassan

Graduation Project Thesis
Electronics and Electrical Communication Engineering
Faculty of Engineering, Ain Shams University

Acknowledgements

We would like to express our deepest gratitude to Dr. Hossam Hassan and Dr. Hesham Omran for their invaluable supervision and guidance throughout our graduation project. Their expertise, patience, and dedication have been instrumental in the successful completion of our work.
We are also immensely grateful to our sponsors, Silicon Arts and L2 Iterative Ventures, for their generous support and belief in our project. Their contributions have played a vital role in turning our vision into reality.
We would like to acknowledge the support and contributions of all our friends, colleagues, and family members who have stood by us throughout this project. Their encouragement, understanding, and patience have been instrumental in keeping us motivated and focused. We extend our heartfelt appreciation to all those who have supported us in various ways, and we are truly grateful for their contributions.The collective contributions have not only made our project possible but also provided us with a rich learning experience that we will carry forward into our future endeavors.
Thank you all!

Abstract

This thesis presents a hardware accelerator for the Number Theoretic Transform (NTT), a key operation in post-quantum cryptography, integrated with a RISC-V open-source GPU to enhance computational efficiency. The accelerator leverages parallelism and custom hardware design to reduce computation time and energy consumption, addressing the growing demands of real-time cryptographic workloads.

A comprehensive evaluation of the NTT accelerator is conducted, focusing on metrics such as latency, throughput, and resource utilization. Experimental simulations and synthesis results demonstrate significant improvements over software-only and traditional hardware implementations. The pipelined and parameterized design highlights the trade-offs between computation speed and resource efficiency, offering a scalable solution for cryptographic applications.

The study also explores the integration of the NTT accelerator into a RISC-V-based GPGPU architecture, utilizing SIMT (Single Instruction, Multiple Threads) execution for parallel processing. Custom instructions and optimized memory access patterns are implemented to maximize data throughput and computational efficiency. Additionally, an FPGA-based implementation is evaluated, showcasing the flexibility and scalability of the design for prototyping and real-world applications.

The findings of this thesis advance NTT acceleration techniques for cryptography and transform-based applications. The work concludes by discussing the implications for modern cryptographic systems and identifying future research directions, such as extending the accelerator to support PLONK systems and other post-quantum cryptographic schemes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rise of post-quantum cryptography has intensified the demand for efficient computational techniques, particularly for polynomial arithmetic. The Number Theoretic Transform (NTT) is a cornerstone of modern cryptographic systems, enabling fast polynomial multiplication by reducing complexity from $O(n^2)$ to $O(n \log_2 n)$. By transforming polynomials from coefficient form to evaluation form, the NTT facilitates efficient pointwise multiplication, making it indispensable for applications such as fully homomorphic encryption (FHE), post-quantum cryptography (PQC), and zero-knowledge proofs (ZKP).

1.1 Problem Statement

Despite its theoretical efficiency, the practical implementation of NTT faces significant challenges. Software-based approaches often fail to meet the throughput and latency requirements of real-time cryptographic systems, particularly for large polynomials. Hardware acceleration offers a promising solution, but integrating NTT into scalable and energy-efficient architectures remains an open challenge. This thesis addresses this gap by exploring the design and implementation of an NTT accelerator using RISC-V-based GPGPU architectures.

1.2 Motivation

The RISC-V instruction set architecture, with its open-source nature and scalability, provides a compelling platform for developing customizable hardware accelerators. By leveraging the parallel processing capabilities of RISC-V GPGPUs, this work aims to accelerate NTT computations while minimizing trade-offs between performance, resource utilization, and power consumption. The resulting accelerator has the potential to enhance the efficiency of cryptographic systems, particularly in the context of post-quantum cryptography, where traditional methods may be vulnerable to quantum attacks.

1.3 Scope and Objectives

This thesis focuses on designing and implementing a pipelined NTT accelerator integrated into a RISC-V GPGPU. Key objectives include:

- Developing a parameterized NTT unit optimized for hardware acceleration.

- Integrating the accelerator into the RISC-V pipeline for seamless hardware-software interaction.
- Evaluating the design on an FPGA platform, with a focus on throughput, latency, and power efficiency.
- Comparing hardware-software co-design strategies to identify optimal implementation approaches.

1.4 Thesis Contribution

This work makes several key contributions:

- A novel NTT accelerator design optimized for RISC-V GPGPUs, leveraging SIMT execution and custom instructions for parallel processing.
- A comprehensive evaluation of the accelerator on an FPGA platform, demonstrating significant performance improvements over software implementations.
- Insights into the trade-offs and challenges of hardware-software co-design for NTT acceleration, paving the way for future research in post-quantum cryptographic systems.

Through these contributions, the thesis provides a practical solution for accelerating NTT operations in real-world cryptographic systems and signal processing applications, paving the way for more efficient and secure post-quantum cryptographic protocols.

Part I

Literature Review

Chapter 2

Numerical Theoretic Transform (NTT)

2.1 Linear, Cyclic, and Negacyclic Convolution

This section briefly explains the definition of linear, cyclic, and negacyclic convolutions between polynomials with integer coefficients to show their basic concepts and differences. We also provide simple and consistent examples throughout the section to clarify how different concepts work.

2.1.1 Polynomial Multiplication and Linear Convolution

Suppose that $G(x)$ and $H(x)$ are polynomials of degree $n-1$ in the ring $\mathbb{Z}_q[x]$ where $q \in \mathbb{Z}$ and x is the polynomial variable, a polynomial multiplication of $G(x)$ and $H(x)$ is defined as:

$$Y(x) = G(x) \cdot H(x) = \sum_{k=0}^{2(n-1)} y_k x^k$$

where $y_k = \sum_{i=0}^{k} g_i h_{k-i} \mod q$, g and h are the polynomial coefficients of $G(x)$ and $H(x)$ respectively. Polynomial multiplication is equivalent to a discrete **linear convolution** between the coefficients' vectors g and h.

Example 2.1: Let $G(x) = 1 + 2x + 3x^2 + 4x^3$ and $H(x) = 5 + 6x + 7x^2 + 8x^3$ or in vector notation: $g = [1, 2, 3, 4]$ and $h = [5, 6, 7, 8]$.

$$
\begin{array}{r}
1 + 2x + 3x^2 + 4x^3 \\
5 + 6x + 7x^2 + 8x^3 \\
\hline
8x^3 + 16x^4 + 24x^5 + 32x^6 \quad \times \\
7x^2 + 14x^3 + 21x^4 + 28x^5 \\
6x + 12x^2 + 18x^3 + 24x^4 \\
5 + 10x + 15x^2 + 20x^3 \\
\hline
5 + 16x + 34x^2 + 60x^3 + 61x^4 + 52x^5 + 32x^6 \quad +
\end{array}
$$

Figure 2.1: Schoolbook method for polynomial multiplication or linear convolution with $O(n^2)$ complexity

2.1.2 Cyclic Convolution

Suppose that $G(x)$ and $H(x)$ are polynomials of degree $n-1$ in the quotient ring $\frac{\mathbb{Z}_q[x]}{(x^n-1)}$ where $q \in \mathbb{Z}$. A cyclic convolution or positive wrapped convolution, $PWC(x)$ is defined as:

$$PWC(x) = \sum_{k=0}^{n-1} c_k x^k$$

where $c_k = \sum_{i=0}^{k} g_i h_{k-i} + \sum_{i=k+1}^{n-1} g_i h_{k+n-i} \mod q$. If $Y(x)$ is the result of their linear convolution in the ring $\mathbb{Z}_q[x]$, it also can be defined as:

$$PWC(x) = Y(x) \mod (x^n - 1)$$

Traditional or schoolbook methods to calculate a cyclic convolution through a polynomial multiplication are shown in Example 2.1, followed by a long division.

Example 2.2: Let $G(x) = 1 + 2x + 3x^2 + 4x^3$ and $H(x) = 5 + 6x + 7x^2 + 8x^3$ or in vector notation: $g = [1, 2, 3, 4]$, and $h = [5, 6, 7, 8]$.

$$
\begin{array}{r|l}
 & 32x^2 + 52x + 61 \\
x^4 - 1 & \overline{32x^6 + 52x^5 + 61x^4 + 60x^3 + 34x^2 + 16x + 5} \\
 & 32x^6 + 0x^5 + 0x^4 + 0x^3 - 32x^2 \quad - \\
\hline
 & 52x^5 + 61x^4 + 60x^3 + 66x^2 + 16x + 5 \\
 & 52x^5 + 0x^4 + 0x^3 + 0x^2 - 52x \quad - \\
\hline
 & 61x^4 + 60x^3 + 66x^2 + 68x + 5 \\
 & 61x^4 + 0x^3 + 0x^2 + 0x - 61 \quad - \\
\hline
 & 60x^3 + 66x^2 + 68x + 66
\end{array}
$$

Figure 2.2: Schoolbook method for positively wrapped modular polynomial multiplication or cyclic convolution with $O(n^2)$ complexity

We have calculated $Y(x)$ in Example 2.1, thus we only need to do a long division by $x^n - 1$

2.1.3 Negacyclic Convolution

Suppose that $G(x)$ and $H(x)$ are polynomials of degree $n-1$ in the quotient ring $\mathbb{Z}[x]/(x^n+1)$ where $q \in \mathbb{Z}$. A negacyclic convolution or negative wrapped convolution, $NWC(x)$ is defined as:

$$NWC(x) = \sum_{k=0}^{n-1} c_k x^k$$

where $c_k = \sum_{i=0}^{k} g_i h_{k-i} - \sum_{i=k+1}^{n-1} g_i h_{k+n-i} \mod q$. If $Y(x)$ is the result of their linear convolution in the ring $\mathbb{Z}[x]$, it also can be defined as

$$NWC(x) = Y(x) \mod (x^n+1)$$

Example 2.3: Let $G(x) = 1+2x+3x^2+4x^3$ and $H(x) = 5+6x+7x^2+8x^3$ or in vector notation: $g = [1,2,3,4]$ and $h = [5,6,7,8]$.

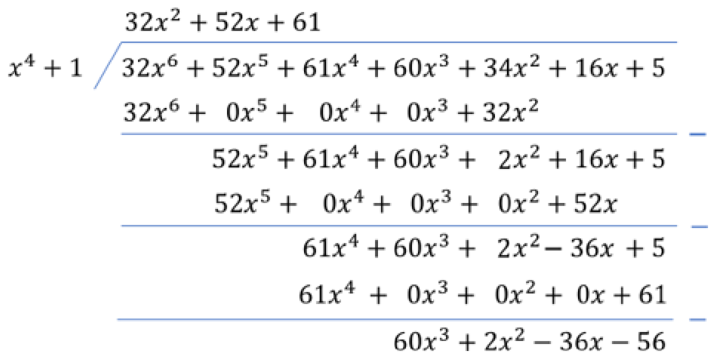


Figure 2.3: Schoolbook method for negatively wrapped modular polynomial multiplication or negacyclic convolution with $O(n^2)$ complexity

We have calculated $Y(x)$ in Example 2.1, thus we only need to do a long division by x^n+1.

Note that the only difference between cyclic and negacyclic convolution is the divisor. The cyclic convolution uses x^n-1 while the negacyclic convolution uses x^n+1. Those schoolbook algorithms have $O(n^2)$ complexity. Many efforts have been made to reduce their complexities by dividing the multiplier and multiplicand into several parts or by parallelizing the algorithm on the implementation side However, those efforts are not scalable as the polynomial degree grows higher.

2.2 NTT-Based Convolution

Many researchers do not differentiate the term NTT and FFT-based algorithms to calculate NTT, which creates confusion when understanding the topic. This report refers to the transformation itself as NTT and the FFT-like algorithms as fast-NTT. The classical NTT has a quadratic complexity of $O(n^2)$ when computed directly, while fast-NTT algorithms have a more efficient quasi-linear complexity $O(n \log n)$.

2.2.1 Primitive nth Root of Unity

Let $\mathbb{Z}_q$ be an integer ring modulo q, and $n-1$ is the polynomial degree of $G(x)$ and $H(x)$. Such rings have a multiplicative identity (unity) of 1. Define ω as a primitive n-th root of unity in $\mathbb{Z}_q$ if and only if:

$$\omega^n \equiv 1 \pmod{q}$$

and

$$\omega^k \not\equiv 1 \pmod{q}$$

for $k < n$.
One thing to note is that the primitive n^{th} root of unity in a ring $\mathbb{Z}_q$ might not be unique. We show the following example for $q = 7681$ and $n = 4$.

Example 3.1: In a ring $\mathbb{Z}_{7681}$ and $n = 4$, the 4-th roots of unity which satisfy the condition $\omega^4 \equiv 1 \pmod{7681}$ are $\{3383, 4298, 7680\}$. Out of three roots, 7680 is not a primitive n-th root of unity, as there exist $k = 2 < n$ that satisfy $\omega^2 \equiv 1 \pmod{7681}$. Therefore $\omega = 3383$ or $\omega = 4298$ are the primitive 4-th roots of unity in $\mathbb{Z}_{7681}$.
The value of ω will be important in calculating NTT and positive wrapped convolution. Calculating the ω of a ring with a large number modulus q is tricky and tedious.

2.2.2 NTT-Based Positive-Wrapped Convolution

This section explains the definition of Number Theoretic Transform (NTT) and its inverse (INTT) based on n-th root of unity, *omega*. The NTT of a polynomial does not have any physical meaning, unlike Discrete Fourier Transform (DFT) which represents a signal in the frequency domain. However, NTT preserves one of the important properties of DFT: the convolution theorem, which is valuable in calculating polynomial multiplication.

Number Theoretic Transform Based on ω

The Number Theoretic Transform (NTT) of a vector of polynomial coefficients a is defined as

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i \mod q$$

where $j = 0, 1, 2, ..., n-1$.

Example 3.2: Let $G(x) = 1 + 2x + 3x^2 + 4x^3$ or in vector notation $g = [1, 2, 3, 4]$. We can infer that $n = 4$. Suppose we work in the ring $\mathbb{Z}_{7681}$ and ω is its primitive n-th root of unity. The NTT of g, $\hat{g}$, can be calculated by the following matrix multiplication:

$$\hat{g} = \begin{bmatrix} \omega^{0\times0} & \omega^{0\times1} & \omega^{0\times2} & \omega^{0\times3} \\ \omega^{1\times0} & \omega^{1\times1} & \omega^{1\times2} & \omega^{1\times3} \\ \omega^{2\times0} & \omega^{2\times1} & \omega^{2\times2} & \omega^{2\times3} \\ \omega^{3\times0} & \omega^{3\times1} & \omega^{3\times2} & \omega^{3\times3} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Notice that the power of ω is the multiplication between the row and column numbers. As ω is the n-root of unity, $\omega^k = \omega^{[k \mod n]}$ for $k > n$. Thus:

$$\hat{g} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^0 & \omega^2 \\ \omega^0 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

From Example 3.1 we obtained one of the n-th roots of unity in $\mathbb{Z}_{7681}$ is $\omega = 3383$. Substituting into the equation:

$$\hat{g} = \begin{bmatrix} 3383^0 & 3383^0 & 3383^0 & 3383^0 \\ 3383^0 & 3383^1 & 3383^2 & 3383^3 \\ 3383^0 & 3383^2 & 3383^0 & 3383^2 \\ 3383^0 & 3383^3 & 3383^2 & 3383^1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3383 & 7680 & 4298 \\ 1 & 7680 & 1 & 7680 \\ 1 & 4298 & 7680 & 3383 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

Therefore, the NTT$(g) = [10, 913, 7679, 6764]$ in $\mathbb{Z}_{7681}$.

Example 3.3: Let $H(x) = 5 + 6x + 7x^2 + 8x^3$ or in vector notation $h = [5, 6, 7, 8]$ in the ring $\mathbb{Z}_{7681}$ and $\omega = 3383$. Using the same principle as Example 3.2, the NTT of h is:

$$\hat{h} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3383 & 7680 & 4298 \\ 1 & 7680 & 1 & 7680 \\ 1 & 4298 & 7680 & 3383 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 26 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

Therefore, the NTT$(h) = [26, 913, 7679, 6764]$ in $\mathbb{Z}_{7681}$.

Note that the NTT of a particular polynomial is not always unique. It depends on the choice of ω. The NTT result of Example 3.2 and 3.3 will differ if one uses $\omega = 4298$ instead of $\omega = 3383$.

Inverse Number Theoretic Transform Based on ω

The Inverse of Number Theoretic Transform (INTT) of an NTT defined as

$$a_i = n^{-1} \sum_{j=0}^{n-1} \omega^{-ij} \hat{a}_j \mod q$$

and $j = 0, 1, 2, ..., n-1$.
Note that the INTT has a very similar formula to NTT. The only differences are ω replaced by its inverse in $\mathbb{Z}_q$ and a n^{-1} scaling factor. It always holds that $a = \text{INTT}(\text{NTT}(a))$.

Example 3.4: Given $\text{NTT}(g) = \hat{g} = [10, 913, 7679, 6764]$ in $\mathbb{Z}_{7681}$ and $\omega = 3383$. We can calculate the inverse of ω, which is $\omega^{-1} = 4298$ and the scaling factor $n^{-1} = 5761$. One can calculate the $\text{INTT}(\text{NTT}(g))$ by the following matrix multiplication:

$$g = n^{-1} \begin{bmatrix} \omega^{-0\times0} & \omega^{-0\times1} & \omega^{-0\times2} & \omega^{-0\times3} \\ \omega^{-1\times0} & \omega^{-1\times1} & \omega^{-1\times2} & \omega^{-1\times3} \\ \omega^{-2\times0} & \omega^{-2\times1} & \omega^{-2\times2} & \omega^{-2\times3} \\ \omega^{-3\times0} & \omega^{-3\times1} & \omega^{-3\times2} & \omega^{-3\times3} \end{bmatrix} \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

$$g = n^{-1} \begin{bmatrix} \omega^{0} & \omega^{0} & \omega^{0} & \omega^{0} \\ \omega^{0} & \omega^{-1} & \omega^{-2} & \omega^{-3} \\ \omega^{0} & \omega^{-2} & \omega^{-4} & \omega^{-6} \\ \omega^{0} & \omega^{-3} & \omega^{-6} & \omega^{-9} \end{bmatrix} \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

$$g = n^{-1} \begin{bmatrix} \omega^{0} & \omega^{0} & \omega^{0} & \omega^{0} \\ \omega^{0} & \omega^{-1} & \omega^{-2} & \omega^{-3} \\ \omega^{0} & \omega^{-2} & \omega^{0} & \omega^{-2} \\ \omega^{0} & \omega^{-3} & \omega^{-2} & \omega^{-1} \end{bmatrix} \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

$$g = 5761 \begin{bmatrix} 4298^{0} & 4298^{0} & 4298^{0} & 4298^{0} \\ 4298^{0} & 4298^{1} & 4298^{2} & 4298^{3} \\ 4298^{0} & 4298^{2} & 4298^{0} & 4298^{2} \\ 4298^{0} & 4298^{3} & 4298^{2} & 4298^{1} \end{bmatrix} \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix}$$

$$g = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4298 & 7680 & 3383 \\ 1 & 7680 & 1 & 7680 \\ 1 & 3383 & 7680 & 4298 \end{bmatrix} \begin{bmatrix} 10 \\ 913 \\ 7679 \\ 6764 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Therefore, the $g = [1, 2, 3, 4]$, which is the initial polynomial coefficients given in Example 3.2

Example 3.5: Given $\text{NTT}(h) = \hat{h} = [26, 913, 7679, 6764]$ in $\mathbb{Z}_{7681}$ and $\omega = 3383$. We can similarly calculate the INTT to the previous example:

$$h = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4298 & 7680 & 3383 \\ 1 & 7680 & 1 & 7680 \\ 1 & 3383 & 7680 & 4298 \end{bmatrix} \begin{bmatrix} 26 \\ 913 \\ 7679 \\ 6764 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

Therefore, the $h = [5, 6, 7, 8]$ which is the initial polynomial coefficients given in Example 3.3

Using NTT to Calculate Positive-Wrapped Convolutions

Because NTT is a variant of DFT in the polynomial ring. One can apply DFT's convolution theorem to calculate positive-wrapped convolution.

Let a and b are the multiplicands' vectors of polynomial coefficients. The positive-wrapped convolution of a and b, c can be calculated by:

$$c = \text{INTT}(\text{NTT}(a) \circ \text{NTT}(b))$$

where $\circ$ is an element-wise vector multiplication in $\mathbb{Z}_q$.

Example 3.6: Let $g = [1, 2, 3, 4]$ and $h = [5, 6, 7, 8]$. From Example 3.2 and 3.3, we know that the NTT of them in $\mathbb{Z}_{7681}$ are $g^\circ = [10, 913, 7679, 6764]$ and $\hat{h} = [26, 913, 7679, 6764]$ when $\omega = 3383$. We can calculate their positive-wrapped convolution by:

$$\begin{aligned}
\text{INTT}\left(\begin{bmatrix}10\\913\\7679\\6764\end{bmatrix} \circ \begin{bmatrix}26\\913\\7679\\6764\end{bmatrix}\right) &= \text{INTT}\begin{bmatrix}260\\4021\\4\\3660\end{bmatrix} \\
&= 5761\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 4298 & 7680 & 3383\\ 1 & 7680 & 1 & 7680\\ 1 & 3383 & 7680 & 4298\end{bmatrix}\begin{bmatrix}260\\4021\\4\\3660\end{bmatrix} = \begin{bmatrix}66\\68\\66\\60\end{bmatrix}
\end{aligned}$$

Therefore, their positive-wrapped convolution is $[66, 68, 66, 60]$, the same result as calculated by schoolbook multiplication and long division in Example 2.2.

While positive-wrapped convolution, commonly known as cyclic convolution, is useful, its implementation is primarily outside the cryptography domain. However, in the context of PQC and HE, the chosen ring is mostly $\frac{\mathbb{Z}_q[x]}{(x^n+1)}$ instead of $\frac{\mathbb{Z}_q[x]}{(x^n-1)}$. One must calculate the polynomial multiplications via the negative-wrapped convolution in such rings.

2.2.3 NTT-Based Negative-Wrapped Convolution

This section explains the definition of Number Theoretic Transform (NTT) and its inverse (INTT) based on $2n$-th root of unity, ψ, and how to utilize them to calculate negative-wrapped or negacyclic convolution.

Primitive $2n$-th Root of Unity

To calculate negative-wrapped convolution, one needs the primitive $2n^{th}$ root of unity, ψ.

Let $\mathbb{Z}_q$ be an integer ring modulo q, and $n-1$ is the polynomial degree of $G(x)$ and $H(x)$. ω is its primitive n-th root of unity. Define ψ as the primitive $2n$-th root of unity if and only if:

$$\psi^2 \equiv \omega \pmod{q} \quad \text{and} \quad \psi^n \equiv -1 \pmod{q}$$

Example 3.7: In a ring $\mathbb{Z}_{7681}$ and $n = 4$, when $\omega = 3383$, the value of ψ can be 1925 or 5756 as $1925^2 \equiv 5756^2 \equiv 3383 \pmod{7681}$ and $1925^4 \equiv 5756^4 \equiv 7680 \equiv -1 \pmod{7681}$. Therefore, one can choose the value $\psi = 1925$ or $\psi = 5756$.

Number Theoretic Transform Based on ψ

The Negative Wrapped Number Theoretic Transform (NTT) of a vector of polynomial coefficients a is defined as $a^{\psi} = \text{NTT}^{\psi}\{a\}$, where:

$$\hat{a}_j = \sum_{i=0}^{n-1} \psi^i \omega^{ij} a_i \pmod{q}$$

and $j = 0, 1, 2, ..., n-1$. As $\psi^2 \equiv \omega \pmod{q}$, we can substitute ω to equation:

$$\hat{a}_j = \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \pmod{q}$$

Example 3.8: Let $g = [1, 2, 3, 4]$, $n = 4$ and $\psi = 1925$ in the ring $\mathbb{Z}_{7681}$. The $\text{NTT}^{\psi}(g) = \hat{g}$, can be calculated by the following matrix multiplication:

$$\hat{g} = \begin{bmatrix} \psi^{2(0\times0)+0} & \psi^{2(0\times1)+1} & \psi^{2(0\times2)+2} & \psi^{2(0\times3)+3} \\ \psi^{2(1\times0)+0} & \psi^{2(1\times1)+1} & \psi^{2(1\times2)+2} & \psi^{2(1\times3)+3} \\ \psi^{2(2\times0)+0} & \psi^{2(2\times1)+1} & \psi^{2(2\times2)+2} & \psi^{2(2\times3)+3} \\ \psi^{2(3\times0)+0} & \psi^{2(3\times1)+1} & \psi^{2(3\times2)+2} & \psi^{2(3\times3)+3} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 \\ \psi^0 & \psi^3 & \psi^6 & \psi^9 \\ \psi^0 & \psi^5 & \psi^{10} & \psi^{15} \\ \psi^0 & \psi^7 & \psi^{14} & \psi^{21} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} 1925^0 & 1925^1 & 1925^2 & 1925^3 \\ 1925^0 & 1925^3 & 1925^6 & 1925^9 \\ 1925^0 & 1925^5 & 1925^{10} & 1925^{15} \\ 1925^0 & 1925^7 & 1925^{14} & 1925^{21} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} 1 & 1925 & 3383 & 6468 \\ 1 & 6468 & 4298 & 1925 \\ 1 & 5756 & 3383 & 1213 \\ 1 & 1213 & 4298 & 5756 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\hat{g} = \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix}$$

Therefore, the $\text{NTT}^{\psi}(g) = [1467, 2807, 3471, 7621]$ when $\psi = 1925$ in $\mathbb{Z}_{7681}$.

Example 3.9: Let $h = [5, 6, 7, 8]$, $n = 4$ and $\psi = 1925$ in the ring $\mathbb{Z}_{7681}$. The $\text{NTT}^{\psi}(h) = \hat{h}$, can be calculated similarly by the following matrix multiplication:

$$\hat{h} = \begin{bmatrix} 1 & 1925 & 3383 & 6468 \\ 1 & 6468 & 4298 & 1925 \\ 1 & 5756 & 3383 & 1213 \\ 1 & 1213 & 4298 & 5756 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix}$$

Therefore, the $\text{NTT}^{\psi}(h) = [2489, 7489, 6478, 6607]$.

Inverse Number Theoretic Transform Based on ψ

The Negative-Wrapped Inverse of Number Theoretic Transform (INTT) of an NTT vector $\hat{a}$ is defined as $a = \text{INTT}^{\psi}(\hat{a})$, where:

$$a_i = n^{-1} \sum_{j=0}^{n-1} \psi^{-j} \omega^{-ij} \hat{a}_j \pmod{q}$$

and $i = 0, 1, 2, ..., n-1$. Substituting $\omega = \psi^2$ yields:

$$a_i = n^{-1} \sum_{j=0}^{n-1} \psi^{-(2ij+j)} \hat{a}_j \pmod{q}$$

Note that the differences between NTT and INTT are the scaling factor n^{-1}, the replacement of ψ by ψ^{-1}, and the transpose of the exponents of the ψ matrix.

Example 3.10: Let $\text{NTT}^{\psi}(g) = \hat{g} = [1467, 2807, 3471, 7621]$ and $\psi = 1925$ in the ring $\mathbb{Z}_{7681}$. Note that $\psi^{-1} = 1213$ and $n^{-1} = 5761$. The vector g can be calculated by the following matrix multiplication:

$$g = n^{-1} \begin{bmatrix} \psi^{-0} & \psi^{-1} & \psi^{-2} & \psi^{-3} \\ \psi^{-0} & \psi^{-3} & \psi^{-6} & \psi^{-9} \\ \psi^{-0} & \psi^{-5} & \psi^{-10} & \psi^{-15} \\ \psi^{-0} & \psi^{-7} & \psi^{-14} & \psi^{-21} \end{bmatrix} \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix}$$

$$g = n^{-1} \begin{bmatrix} \psi^{0} & \psi^{-1} & \psi^{-2} & \psi^{-3} \\ \psi^{0} & \psi^{-3} & \psi^{-2} & \psi^{-1} \\ \psi^{0} & \psi^{-2} & \psi^{0} & \psi^{-2} \\ \psi^{0} & \psi^{-1} & \psi^{-2} & \psi^{-3} \end{bmatrix} \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix}$$

$$g = 5761 \begin{bmatrix} 1213^{0} & 1213^{1} & 1213^{2} & 1213^{3} \\ 1213^{0} & 1213^{3} & 1213^{6} & 1213^{9} \\ 1213^{0} & 1213^{5} & 1213^{10} & 1213^{15} \\ 1213^{0} & 1213^{7} & 1213^{14} & 1213^{21} \end{bmatrix} \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix}$$

$$g = 5761 \begin{bmatrix} 1 & 1213 & 1467 & 1925 \\ 1 & 1925 & 1467 & 1213 \\ 1 & 1467 & 1 & 1467 \\ 1 & 1213 & 1467 & 1925 \end{bmatrix} \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Therefore $g = [1, 2, 3, 4]$.

Example 3.11: Let $\text{NTT}^{\psi}(h) = \hat{h} = [2489, 7489, 6478, 6607]$ and $\psi = 1925$ in the ring $\mathbb{Z}_{7681}$. The vector h can be calculated by the following matrix multiplication:

$$h = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1213 & 5756 & 6468 & 1925 \\ 4298 & 3383 & 4298 & 3383 \\ 5756 & 1213 & 1925 & 6468 \end{bmatrix} \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

Therefore, the $h = [5, 6, 7, 8]$.

Using NTT$^{\psi}$ to Calculate Negative-Wrapped Convolutions

Like its positive-wrapped version, the negative-wrapped NTT can evaluate the negative-wrapped convolutions, commonly referred to as negacyclic convolutions.

Let a and b be the multiplicands' vectors of polynomial coefficients. The negative-wrapped convolution of a and b, c can be calculated by:

$$c = \text{INTT}^{\psi^{-1}}(\text{NTT}^{\psi}(a) \circ \text{NTT}^{\psi}(b))$$

Example 3.12: Let $g = [1, 2, 3, 4]$ and $h = [5, 6, 7, 8]$. From Example 3.8 and 3.9, we know that the NTT$^{\psi}$ of them in $\mathbb{Z}_{7681}$ are $\hat{g} = [1467, 2807, 3471, 7621]$ and $\hat{h} = [2489, 7489, 6478, 6607]$ when $\psi = 1925$. We can calculate their negative-wrapped convolution by:

$$\text{INTT}^{\psi}\left(\begin{bmatrix}1467\\2807\\3471\\7621\end{bmatrix} \circ \begin{bmatrix}2489\\7489\\6478\\6607\end{bmatrix}\right) = \text{INTT}^{\psi}\begin{bmatrix}2888\\6407\\2851\\2992\end{bmatrix}$$

$$= 5761\begin{bmatrix}1 & 1 & 1 & 1\\1213 & 5756 & 6468 & 1925\\4298 & 3383 & 4298 & 3383\\5756 & 1213 & 1925 & 6468\end{bmatrix}\begin{bmatrix}2888\\6407\\2851\\2992\end{bmatrix} = \begin{bmatrix}7625\\7645\\2\\60\end{bmatrix}$$

Therefore, $[7625, 7645, 2, 60]$ or when written with negative numbers $[-56, -36, 2, 60]$ is their negacyclic convolution, the same result as calculated by schoolbook multiplication and long division in Example 2.3.

2.3 Fast NTT

In the previous sections, the presented NTT and INTT transformation pairs have $O(n^2)$ complexity, thus making no difference from the traditional method of negacyclic convolution. However, the NTT is the Discrete Fourier Transform in another ring. Therefore, the DFT optimization techniques can be applied to NTT. The well-known technique of DFT optimization is called the Fast-Fourier Transform (FFT), proposed independently by Cooley-Tukey and Gentleman-Sande. Both using similar butterflies divide-and-conquer technique to reduce the complexity to $O(n \log n)$.
To reduce the complexity and fasten the process of the matrix multiplication needed for the NTT transformation, one can use "divide and conquer" techniques by utilizing the periodicity and symmetry property of ψ:

$$\text{periodicity} : \psi^{k+2n} \equiv \psi^{k} \tag{2.1}$$

$$\text{symmetry} : \psi^{k+n} \equiv -\psi^{k} \tag{2.2}$$

where k is a non-negative integer. The calculation of a n point NTT and INTT can be divided into two $n/2$ points. However, the dividing techniques for NTT and INTT are slightly different.

2.3.1 Cooley-Tukey (CT) Algorithm for Fast-NTT

$$\begin{aligned}\hat{a}_j &= \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \pmod q \\ &= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \sum_{i=0}^{n/2-1} \psi^{4ij+2j+2i+1} a_{2i+1} \pmod q \\ &= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \psi^{2j+1} \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod q\end{aligned}$$

Based on the ψ's symmetry properties:

$$\hat{a}_{j+n/2} = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} - \psi^{2j+1} \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod q$$

Let $A_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i}$ and $B_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1}$, equations become:

$$\begin{aligned}\hat{a}_j &= A_j + \psi^{2j+1} B_j \pmod q \\ \hat{a}_{j+n/2} &= A_j - \psi^{2j+1} B_j \pmod q\end{aligned}$$

Notice that A_j and B_j can be obtained as $n/2$ points NTT. If n is power-of-two.

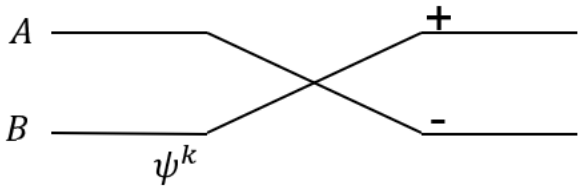


Figure 2.4: Cooley-Tukey (CT) butterfly unit for calculating NTT

One can configure several butterfly units to calculate the entire n length of NTT. The idea is to calculate similar terms in the brackets once and then distribute the results instead of calculating them multiple times. The order of the results of CT-Butterfly is called bit-reversed order (BO), while the correct order of the NTT is called normal order (NO).

Example 3.13:

$$\hat{g} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 \\ \psi^0 & \psi^3 & \psi^6 & \psi^9 \\ \psi^0 & \psi^5 & \psi^{10} & \psi^{15} \\ \psi^0 & \psi^7 & \psi^{14} & \psi^{21} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\begin{aligned} g^0 &= 1\psi^0 + 2\psi^1 + 3\psi^2 + 4\psi^3 \\ g^1 &= 1\psi^0 + 2\psi^3 + 3\psi^6 + 4\psi^9 \\ g^2 &= 1\psi^0 + 2\psi^5 + 3\psi^{10} + 4\psi^{15} \\ g^3 &= 1\psi^0 + 2\psi^7 + 3\psi^{14} + 4\psi^{21} \end{aligned}$$

Factoring:

$$
\begin{aligned}
g'_0 &= \psi^0(1+3\psi^2) + \psi^1(2+4\psi^2) \\
g'_1 &= \psi^0(1+3\psi^6) + \psi^3(2+4\psi^6) \\
g'_2 &= \psi^0(1+3\psi^{10}) + \psi^5(2+4\psi^{10}) \\
g'_3 &= \psi^0(1+3\psi^{14}) + \psi^7(2+4\psi^{14})
\end{aligned}
$$

Based on the ψ periodicity:

$$
\begin{aligned}
g'_0 &= \psi^0(1+3\psi^2) + \psi^1(2+4\psi^2) \\
g'_1 &= \psi^0(1+3\psi^6) + \psi^3(2+4\psi^6) \\
g'_2 &= \psi^0(1+3\psi^2) + \psi^5(2+4\psi^2) \\
g'_3 &= \psi^0(1+3\psi^6) + \psi^7(2+4\psi^6)
\end{aligned}
$$

Based on the ψ symmetry:

$$
\begin{aligned}
g'_0 &= \psi^0(1+3\psi^2) + \psi^1(2+4\psi^2) \\
g'_1 &= \psi^0(1-3\psi^2) + \psi^3(2-4\psi^2) \\
g'_2 &= \psi^0(1+3\psi^2) - \psi^1(2+4\psi^2) \\
g'_3 &= \psi^0(1-3\psi^2) - \psi^3(2-4\psi^2)
\end{aligned}
$$

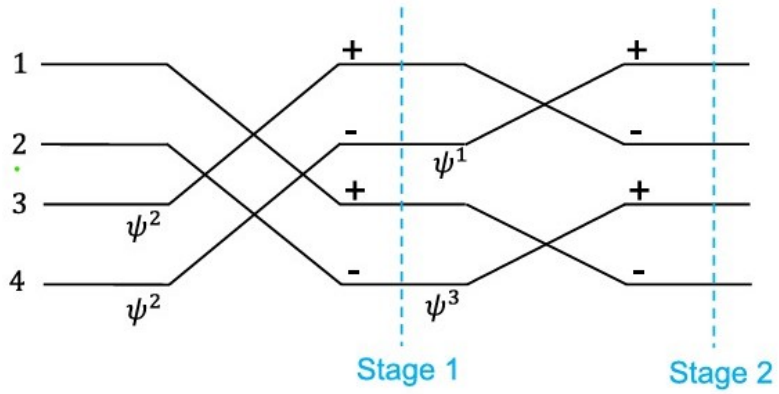


Figure 2.5: CT Butterflies for n = 4 and [1, 2, 3, 4] as its input

2.3.2 Gentleman-Sande (GS) Algorithm for Fast-INTT

To calculate INTT, one will need another but similar "divide and conquer" approach. For the INTT, instead of dividing the summation by its index parity, it is separated by the lower and upper half of the summation. From equation (16) and ignoring n^{-1} term:

$$
\begin{aligned}
a_i &= \sum_{j=0}^{n-1} \psi^{-(2i+1)j} \hat{a}_j \pmod q \\
&= \left[\sum_{j=0}^{\frac{n}{2}-1} \psi^{-(2i+1)j} \hat{a}_j + \sum_{j=0}^{\frac{n}{2}-1} \psi^{-(2i+1)(j+\frac{n}{2})} \hat{a}_{(j+\frac{n}{2})} \right] \pmod q \\
&= \psi^{-i} \left[\sum_{j=0}^{\frac{n}{2}-1} \psi^{-2ij} \hat{a}_j + \sum_{j=0}^{\frac{n}{2}-1} \psi^{-2i(j+\frac{n}{2})} \hat{a}_{(j+\frac{n}{2})} \right] \pmod q
\end{aligned}
$$

Based on the periodicity and symmetry of ψ^{-1}, for the even term:

$$
\begin{aligned}
a_{2i} &= \psi^{-2i} \left[\sum_{j=0}^{\frac{n}{2}-1} \psi^{-4ij} \hat{a}_j + \sum_{j=0}^{\frac{n}{2}-1} \psi^{-4i(j+\frac{n}{2})} \hat{a}_{(j+\frac{n}{2})} \right] \pmod q \\
a_{2i} &= \psi^{-2i} \sum_{j=0}^{\frac{n}{2}-1} \left[\hat{a}_j + \hat{a}_{(j+\frac{n}{2})} \right] \psi^{-4ij} \pmod q
\end{aligned}
$$

Doing the same derivation for the odd term:

$$
a_{2i+1} = \psi^{-2i} \sum_{j=0}^{\frac{n}{2}-1} \left[\hat{a}_j - \hat{a}_{(j+\frac{n}{2})} \right] \psi^{-4ij} \pmod q
$$

Let $A_i = \sum_{j=0}^{\frac{n}{2}-1} \hat{a}_j \psi^{-4ij}$ and $B_i = \sum_{j=0}^{\frac{n}{2}-1} \hat{a}_{j+\frac{n}{2}} \psi^{-4ij}$,

$$
\begin{aligned}
a_{2i} &= (A_i + B_i)\psi^{-2i} \pmod q \\
a_{2i+1} &= (A_i - B_i)\psi^{-2i} \pmod q
\end{aligned}
$$

Notice that A_i and B_i can be obtained as $\frac{n}{2}$ points INTT.

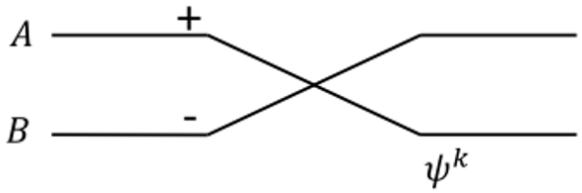


Figure 2.6: Gentleman-Sande (GS) butterfly unit for calculating INTT

Example 3.14:

$$
g = n^{-1} \begin{bmatrix} \psi^{-0} & \psi^{-0} & \psi^{-0} & \psi^{-0} \\ \psi^{-1} & \psi^{-3} & \psi^{-5} & \psi^{-7} \\ \psi^{-2} & \psi^{-6} & \psi^{-10} & \psi^{-14} \\ \psi^{-3} & \psi^{-9} & \psi^{-15} & \psi^{-21} \end{bmatrix} \begin{bmatrix} 1467 \\ 2807 \\ 3471 \\ 7621 \end{bmatrix}
$$

$$
\begin{aligned}
g_0 &= n^{-1}(1467\psi^{-0} + 2807\psi^{-0} + 3471\psi^{-0} + 7621\psi^{-0}) \\
g_1 &= n^{-1}(1467\psi^{-1} + 2807\psi^{-3} + 3471\psi^{-5} + 7621\psi^{-7}) \\
g_2 &= n^{-1}(1467\psi^{-2} + 2807\psi^{-6} + 3471\psi^{-10} + 7621\psi^{-14}) \\
g_3 &= n^{-1}(1467\psi^{-3} + 2807\psi^{-9} + 3471\psi^{-15} + 7621\psi^{-21})
\end{aligned}
$$

Factoring:

$$
\begin{aligned}
g_0 &= n^{-1}(\psi^{-0}(1467 + 3471) + \psi^{-0}(2807 + 7621)) \\
g_1 &= n^{-1}(\psi^{-1}(1467 + 3471\psi^{-4}) + \psi^{-3}(2807 + 7621\psi^{-4})) \\
g_2 &= n^{-1}(\psi^{-2}(1467 + 3471\psi^{-8}) + \psi^{-6}(2807 + 7621\psi^{-8})) \\
g_3 &= n^{-1}(\psi^{-3}(1467 + 3471\psi^{-12}) + \psi^{-9}(2807 + 7621\psi^{-12}))
\end{aligned}
$$

Based on the ψ periodicity:

$$
\begin{aligned}
g_0 &= n^{-1}(\psi^{-0}(1467 + 3471) + \psi^{-0}(2807 + 7621)) \\
g_1 &= n^{-1}(\psi^{-1}(1467 + 3471\psi^{-4}) + \psi^{-3}(2807 + 7621\psi^{-4})) \\
g_2 &= n^{-1}(\psi^{-2}(1467 + 3471\psi^{0}) + \psi^{-5}(2807 + 7621\psi^{0})) \\
g_3 &= n^{-1}(\psi^{-3}(1467 + 3471\psi^{4}) + \psi^{-1}(2807 + 7621\psi^{4}))
\end{aligned}
$$

Based on the ψ symmetry:

$$
\begin{aligned}
g_0 &= n^{-1}(\psi^{-0}(1467 + 3471) + \psi^{-0}(2807 + 7621)) \\
&= \psi^{-0}(\psi^{-0}(1467 + 3471) + \psi^{-0}(2807 + 7621)) \\
g_1 &= n^{-1}(\psi^{-1}(1467 - 3471\psi^{0}) + \psi^{-3}(2807 - 7621\psi^{0})) \\
&= \psi^{-0}(\psi^{-1}(1467 - 3471\psi^{0}) + \psi^{-3}(2807 - 7621\psi^{0})) \\
g_2 &= n^{-1}(\psi^{-2}(1467 + 3471\psi^{0}) - \psi^{-2}(2807 + 7621\psi^{0})) \\
&= \psi^{-2}((1467 + 3471\psi^{0}) - (2807 + 7621\psi^{0})) \\
g_3 &= n^{-1}(\psi^{-3}(1467 - 3471\psi^{0}) + \psi^{-1}(2807 - 7621\psi^{0})) \\
&= \psi^{-2}(\psi^{-1}(1467 - 3471\psi^{0}) - \psi^{-3}(2807 - 7621\psi^{0}))
\end{aligned}
$$

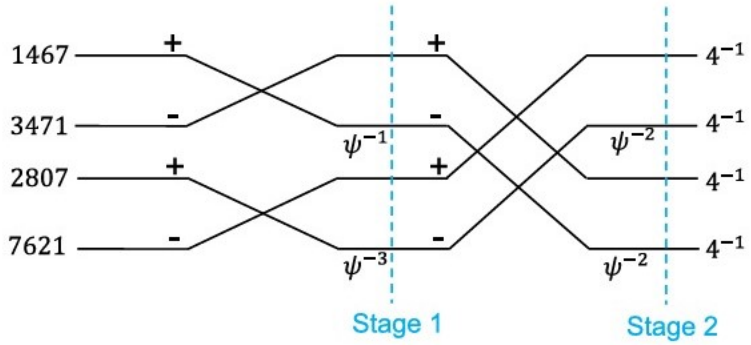


Figure 2.7: GS Butterflies for n = 4 and [1467, 2807, 3471, 7621] as its input

Chapter 3

RISC-V

RISC-V is a modern, open-source Instruction Set Architecture (ISA) that has gained significant traction in academia and industry for its simplicity, modularity, and flexibility. Unlike proprietary ISAs, RISC-V is freely available under a permissive license, enabling researchers and engineers to use, modify, and extend the architecture without incurring licensing costs or restrictions. This open nature makes it particularly attractive for developing hardware accelerators and custom systems tailored to specific applications.

3.1 Overview of RISC-V Architecture

RISC-V follows the Reduced Instruction Set Computing (RISC) principles, which emphasize simplicity and efficiency in instruction design. It provides a base integer instruction set (RV32I or RV64I) that includes essential operations, while optional extensions, such as floating-point (F), atomic (A), and vector (V) instructions, allow developers to adapt the ISA to different workloads. This modularity ensures that systems can be designed with only the necessary features, minimizing hardware complexity and power consumption.
The adoption of RISC-V offers several key advantages over traditional ISAs:

- **Open-Source Flexibility**: RISC-V's open-source nature allows unrestricted access to the ISA specifications, fostering innovation and collaboration among researchers and companies.
- **Customizability**: Unlike fixed ISAs, RISC-V enables developers to add custom instructions and extensions to suit specific workloads, improving performance for specialized tasks.
- **Scalability**: RISC-V supports designs ranging from simple microcontrollers to high-performance processors, making it versatile for a wide range of applications.
- **Ecosystem and Toolchain**: RISC-V has a growing ecosystem, including compilers, simulators, and hardware development tools, which simplifies the development of custom systems.

3.2 RISC-V Registers

RISC-V architecture defines a set of general-purpose registers (GPRs) that are used for storing data and addresses during program execution. The base integer ISA (RV32I or RV64I) includes 32 GPRs, labeled x0 to x31, where x0 is hardwired to zero and serves as the constant zero register. The remaining registers are used for general-purpose computation and data manipulation.

3.3 Instruction Fields

RISC-V instructions are composed of several fields, each serving a specific purpose. These fields define the operation to be performed, the source and destination registers, and any immediate values required for the instruction. Table 3.1 provides an overview of these fields and their roles.

Table 3.1: RISC-V Instruction Fields

Field	Bit Width	Purpose
`Opcode`	7	Specifies the operation type and determines the instruction format.
`rd`	5	Destination register that stores the result of the operation.
`funct3`	3	Encodes the operation's sub-type within the opcode. It refines the instruction's functionality.
`rs1`	5	First source register used in the operation.
`rs2`	5	Second source register used in two-operand operations.
`funct7`	7	Provides additional differentiation for the instruction, particularly for operations like shifts and arithmetic.
`Immediate`	Variable	Encodes constants or offsets required for specific instructions. Immediate values are always sign-extended.

3.4 Base ISA

The RISC-V base ISA includes four primary instruction formats (R, I, S, U) and two additional formats (B, J) for handling different types of immediates. All base instructions are 32 bits in length and must be aligned to a four-byte boundary in memory. Misalignment exceptions are triggered for taken branches or jumps targeting non-aligned addresses. Extensions, such as compressed instructions, allow for two-byte alignment by reducing instruction lengths to 16 bits. Key aspects include:

- Immediate values are sign-extended for simplicity and efficient hardware implementation.
- Register fields (rs1, rs2, rd) are fixed across formats to reduce decoding complexity.

- Immediate fields vary across formats but are packed and aligned to reduce hardware complexity and maintain efficient sign extension.

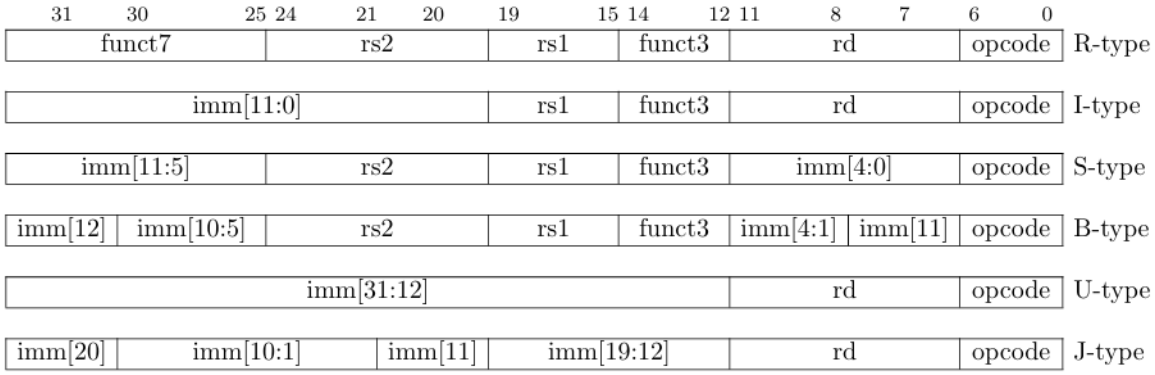


Figure 3.1: RISC-V Base Instruction Formats

Instruction formats serve different purposes:

- **R-Type**: Used for register-register operations, where two source registers are specified along with a destination register.
- **I-Type**: Used for immediate operations, where one source register is combined with an immediate value to produce a result.
- **S-Type**: Used for store operations, where a source register is stored in memory using an immediate offset.
- **U-Type**: Used for upper immediate operations, where an immediate value is loaded into a register.
- **B-Type**: Used for branch operations, where two source registers are compared, and a branch is taken based on the result.

Chapter 4

Vortex GPGPU

The Vortex GPGPU is a RISC-V-based GPU architecture designed for high-throughput parallel processing of data-parallel workloads. It features a SIMT execution model, where multiple threads execute the same instruction stream in lockstep, enabling efficient processing of vectorized computations.

4.1 CPUs vs GPUs

In cryptographic schemes, especially those relying on the Number Theoretic Transform (NTT) and its inverse (INTT), it is typical to operate on a massive number of independent points. These algorithms involve large-scale modular arithmetic that follows a predictable and highly parallel structure. As such, they are well-suited to hardware architectures capable of efficiently executing large numbers of concurrent operations.

Central Processing Units (CPUs) are traditionally optimized for general-purpose computing and sequential workloads. They typically consist of a small number of complex, out-of-order execution cores designed to maximize single-thread performance and minimize latency. This makes CPUs effective for tasks with complex control flows, deep branching, or tight data dependencies. However, these strengths come at the cost of limited throughput when handling wide data-parallel workloads like NTTs, where the same operation must be applied to thousands of data elements independently.

While recent developments such as the RISC-V vector extension (V-extension) aim to bridge this gap by introducing native support for vectorized operations within CPU architectures, they are still inherently limited in the degree of parallelism they can expose. The V-extension can significantly accelerate moderate-scale data-parallel tasks by executing a single instruction over vector registers, but its performance is constrained by the width of the vector unit and the number of available functional units. It is well-suited for improving the performance of embedded or low-power systems where resource efficiency is critical, but it does not offer the same scale of parallel execution as a GPU.

Graphics Processing Units (GPUs), in contrast, are architected specifically for throughput-oriented workloads. They contain hundreds to thousands of simple cores organized into streaming multiprocessors, each capable of managing multiple threads simultaneously. GPUs follow the Single Instruction, Multiple Threads (SIMT) execution model, which allows a single instruction to be applied across many threads, each processing different

data. This model perfectly matches the computational pattern of NTTs and other cryptographic primitives that involve regular, stateless operations across large input vectors.

By offloading NTT computations to the GPU, we are able to exploit this massive thread-level parallelism to achieve significantly higher throughput compared to a CPU-only implementation. This is particularly important in cryptographic applications, such as zero-knowledge proofs and homomorphic encryption, where processing thousands of field elements in parallel directly impacts performance and scalability.

4.2 SIMD & SIMT Execution

4.2.1 SIMD

Single Instruction, Multiple Data (SIMD) enables a single instruction to operate on multiple data elements simultaneously, ideal for vectorized computations with regular data patterns. It is widely used in GPUs and multimedia units for accelerating parallelizable workloads.

4.2.2 SIMT

Single Instruction, Multiple Threads (SIMT) is a GPU execution model where multiple threads execute the same instruction path concurrently, processing independent data elements. It efficiently handles data-parallel workloads by leveraging the GPU's thread-level parallelism. Each 32 or 64 threads are grouped into a **warp**, which executes the same instruction in lockstep. This model allows for divergence in thread execution, where threads can follow different paths but still execute the same instruction stream.

Table 4.1: Key Differences Between SIMD and SIMT

Aspect	SIMD	SIMT
Execution Unit	Operates on multiple data elements with a single instruction	Operates multiple threads in parallel on different data
Independence	All data elements must follow the same operation (1 PC)	Threads can diverge but still execute the same instruction stream (Multiple PCs)

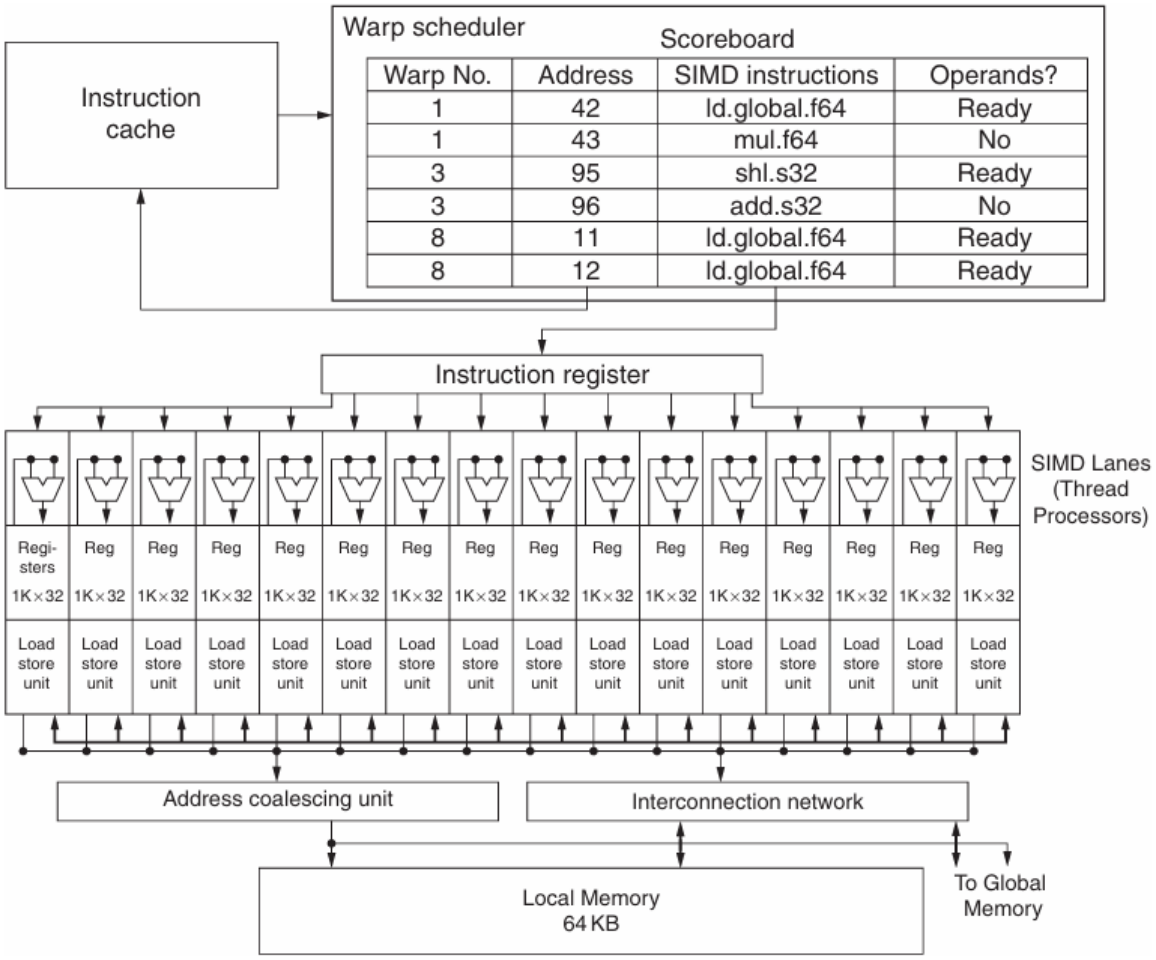


Figure 4.1: Simplified block diagram of a Multithreaded SIMD Processor

4.3 Microarchitecture

The vortex microarchitecture is a hierarchical design designed to maximize data throughput and minimize latency by leveraging parallelism at multiple levels.

- **Processor**: The entire processing unit, which contains groups of cluster sharing L3 cache.
- **Cluster**: A group of sockets that share L2 cache.
- **Socket**: A physical unit that contains multiple cores and share L1 cache.
- **Core**: An individual processing unit within a socket, which contains the execution pipeline.

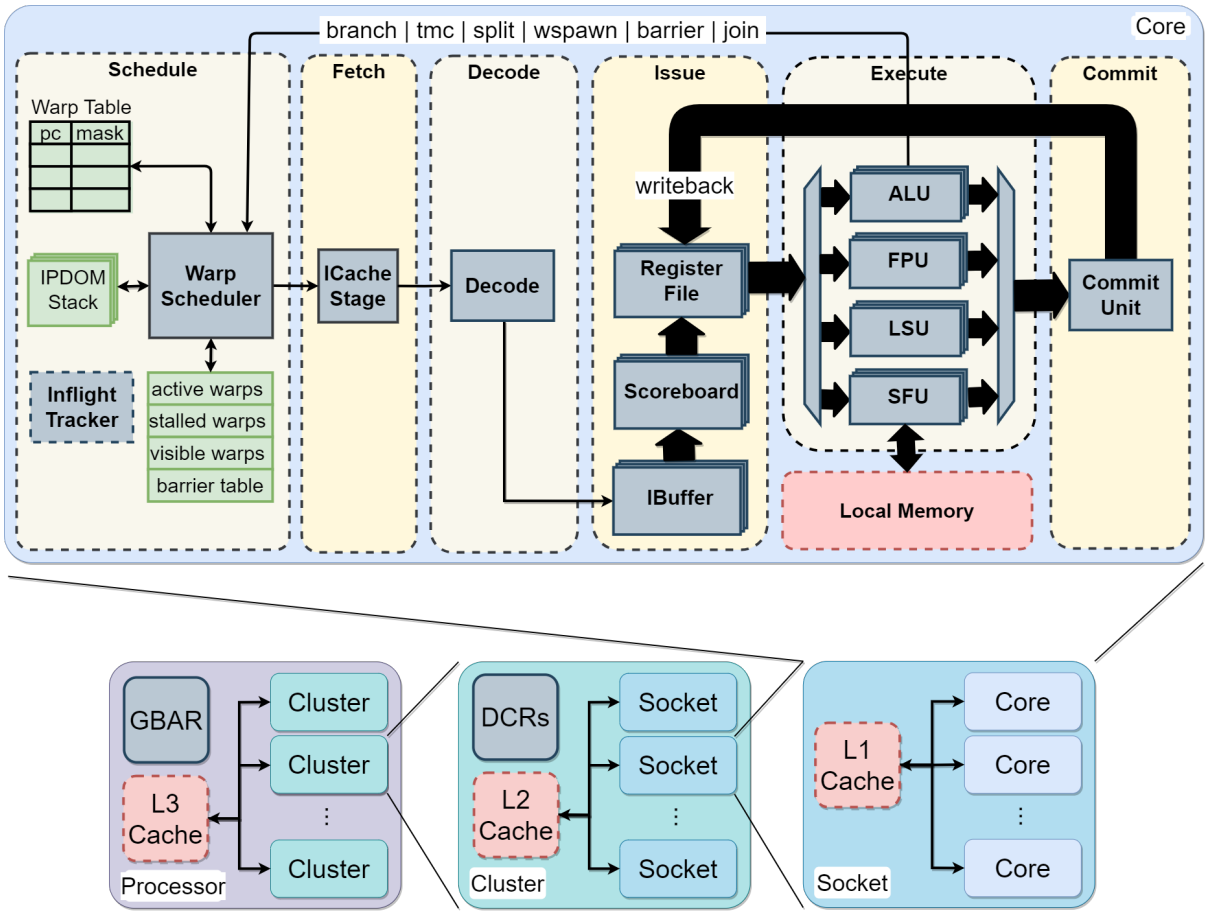


Figure 4.2: Vortex Microarchitecture

4.4 Core Pipeline

The core pipeline consists of 6 stages, each responsible for a specific operation in the instruction execution process. Pipelining helps improve throughput by allowing multiple instructions to be processed simultaneously.

4.4.1 Schedule Stage

The Schedule Stage is the initial stage in the pipeline, playing a pivotal role in the execution of warps. It ensures the smooth progression of instructions through the pipeline by carefully selecting and managing warps. key responsiblities include:

- **Warp Selection:** Chooses a warp to propagate to subsequent stages in the pipeline, prioritizing based on specific criteria such as readiness and resource availability.
- **Thread Mask Management:** Thread masks are used to track active threads within a warp, enabling efficient execution of instructions across multiple threads.
- **Program Counter (PC) Updates:** Adjusts the program counters (PC) of warps to point to the correct instruction for execution, ensuring accurate control flow.
- **Stall Handling:** Identifies and resolves stalls by unlocking warps that were previously blocked due to resource conflicts, dependencies, or other pipeline constraints.

The schedule stage operates as follows:

- Using the signals from warp control unit in the execute stage, new warps are spawned using `WSPAWN` instruction, so the schdule stage updates the active warps tracker.
- Schedule stage updates the active thread masks using branch divergence instructions like `JOIN` & `SPLIT` as well as barrier synchronization instructions like `BAR` are handled.
- The schedule stage also updates the program counter (PC) when encountering branch instructions like `BRANCH` or `JUMP`.
- The schedule stage updates the CSR with information like number of cycles, the active warps, and thread masks.
- A leading zero counter selects the the warp to be scheduled from the active warps.

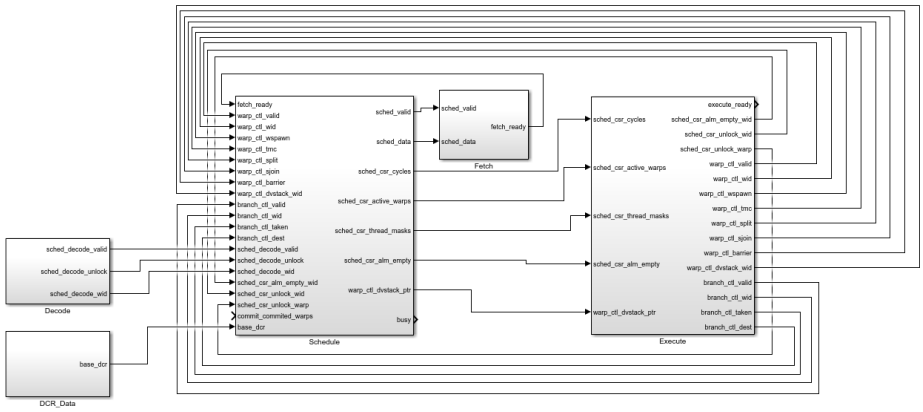


Figure 4.3: Block Diagram of the Schedule Stage

4.4.2 Fetch Stage

The Fetch Stage is responsible for retrieving instructions from the instruction cache, ensuring that the correct instructions are delivered to the pipeline for active warps. Key responsibilities include:

- **Instruction Fetching:** Requests instructions from the instruction cache (Icache) based on the Program Counter (PC) provided by the schedule stage.
- **Warp Context Management:** Maintains context information such as thread masks and PCs for each warp to ensure accurate handling of instruction data.

Fetch stage prevents deadlocks in case of cacheless memory access with the help of signals from issue stage indicating that the instruction buffer is not full yet.

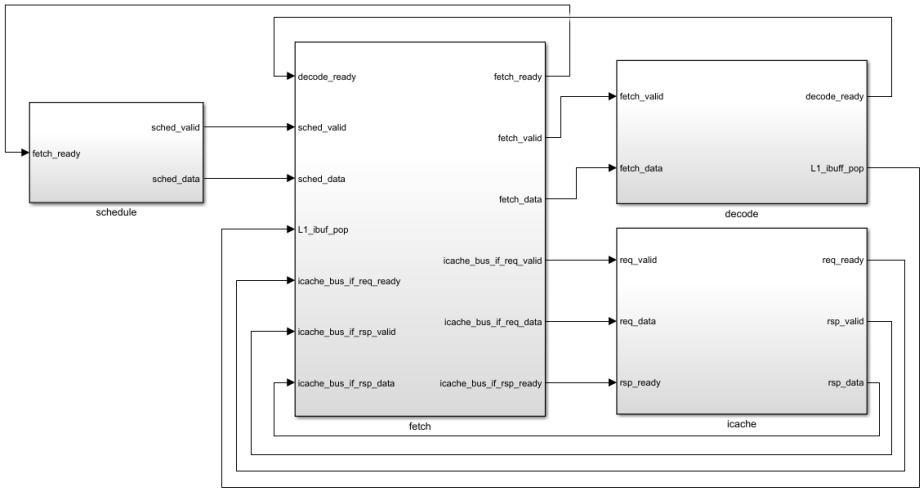


Figure 4.4: Block Diagram of the Fetch Stage

4.4.3 Decode Stage

The Decode Stage is responsible for decoding fetched instructions and preparing them for execution. It extracts relevant information from the instruction stream, such as operation type, register operands, and immediate values, to facilitate efficient processing. The instruction is broken down to:

- **Operation Type:** Determines the type of operation to be performed (e.g., arithmetic, load/store, branch).
- **Register Operands:** Identifies the source and destination registers for the instruction.
- **Operand Arguments:** Extracts immediate values or offsets and other flags or control bits required for the instruction.
- **Execution Type:** Specifies the execution unit or functional unit required to execute the instruction.
- **Writeback Flag:** Indicates whether the instruction result should be written back to the register file.

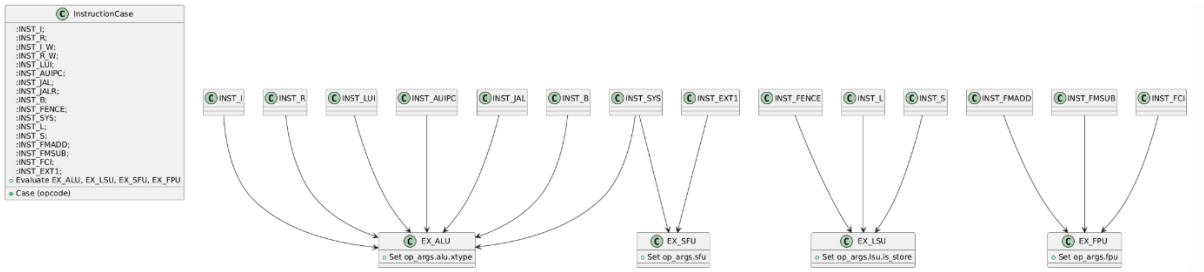


Figure 4.5: Mapping of different instruction types to different execution units

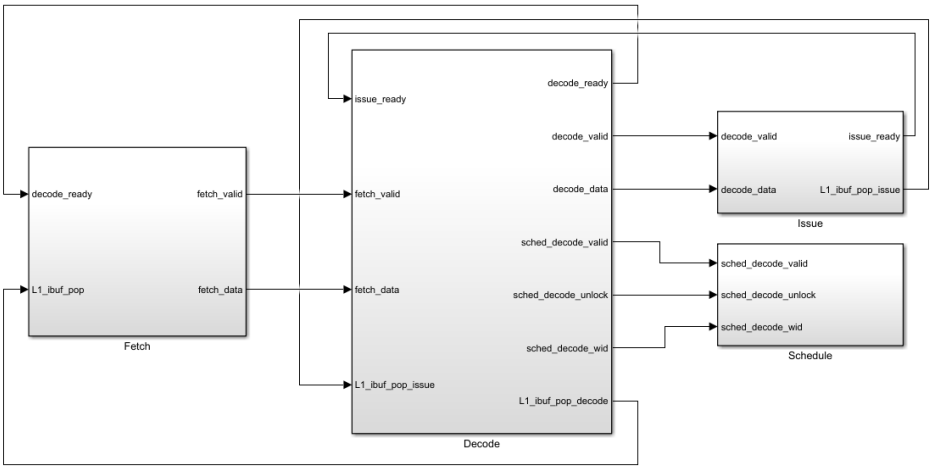


Figure 4.6: Block Diagram of the Decode Stage

4.4.4 Issue Stage

The Issue Stage introduces a second and third scheduling mechanism to efficiently manage instruction dependencies while optimizing data access. These mechanisms work to ensure that instructions are issued in an order that minimizes stalls and maximizes parallelism, enhancing overall performance. Additionally, they aim to maximize bank hits during data fetching, reducing memory access latency and improving throughput. Key responsibilities include:

- **Instruction Buffer Management:** Stores multiple instructions fetched from memory to prevent stalling when there is a dependency.
- **Dependency Tracking:** Utilizes the scoreboard to identify and manage data and structural hazards between instructions, ensuring that issued instructions do not introduce pipeline hazards. This tracking allows overlapping the execution of instructions within the same warp.
- **Operand Access Optimization:** Relies on the operand collector to increase parallelism by maximizing register bank hits. This reduces access contention and improves throughput for source operands during instruction execution.

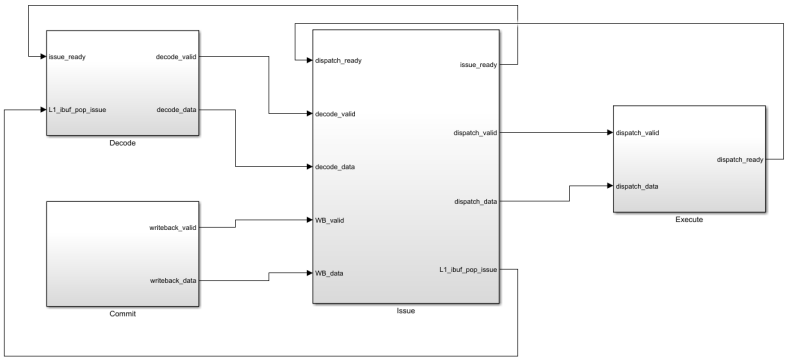


Figure 4.7: Block Diagram of the Issue Stage

The total number of warps is divided into slices, where each slice is equal to 8 warps and share the following:

Instruction Buffer

The instruction buffer is a FIFO (First-In, First-Out) structure that holds multiple decoded instructions, allowing multiple instruction to be stored in case if the instruction will cause hazards. This helps improve the flow of instructions through the pipeline. The instruction buffer also helps reduce delays caused by instruction cache misses by working with instruction miss-status holding registers (MSHRs), making sure that memory delays don't slow down the pipeline too much.

Scoreboard

The scoreboard manages data dependencies between instructions in a GPU core. It uses a simple in-order design to track the readiness of operands and registers for each warp. When an instruction enters the instruction buffer, the scoreboard is accessed to check for dependencies between operands and previously issued instructions. If dependencies are detected, the instruction is stalled until the required operands are available. This design prevents:

- **RAW hazards:** They are prevented by checking if an instruction is trying to read a register that is currently being written to by a previous instruction. If the register is being written, the instruction will be stalled until the write completes.
- **WAW hazards:** They are avoided by ensuring that two instructions do not write to the same register simultaneously. The scoreboard tracks which instructions are writing to each register and stalls any subsequent instructions that attempt to write to the same register before the previous write is completed.

Operand Collector

The operand collector retrieves source operands for instructions from the instruction buffer by interacting with the scoreboard and general-purpose registers (GPRs). It ensures that operands are fetched only when their data is ready, preventing unnecessary delays. To support parallel accesses, the collector uses a multi-bank GPR design and incorporates arbitration to resolve bank conflicts. Each source operand is connected to each register file bank by a crossbar connection. The bank access is pipelined as shown in the figure below.

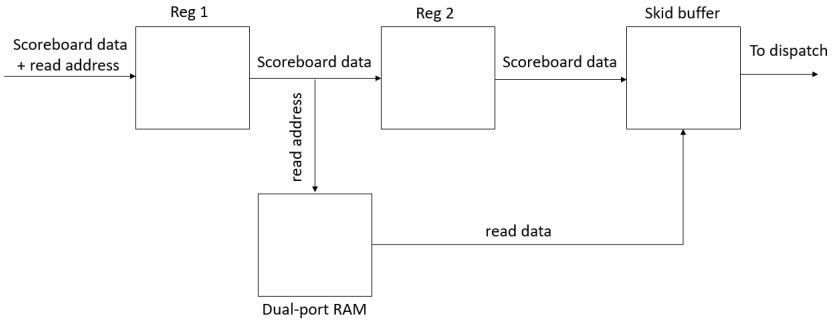


Figure 4.8: Block Diagram of Operand Collector

Dispatch

It is responsible for dispatching operands to execution units in a GPU pipeline. It interfaces with the operand collector to fetch data and ensures that operands are ready before forwarding them to the appropriate execution unit. Each execution unit has an associated skid buffer, which temporarily holds operands to resolve pipeline hazards and maintain a steady flow of data.

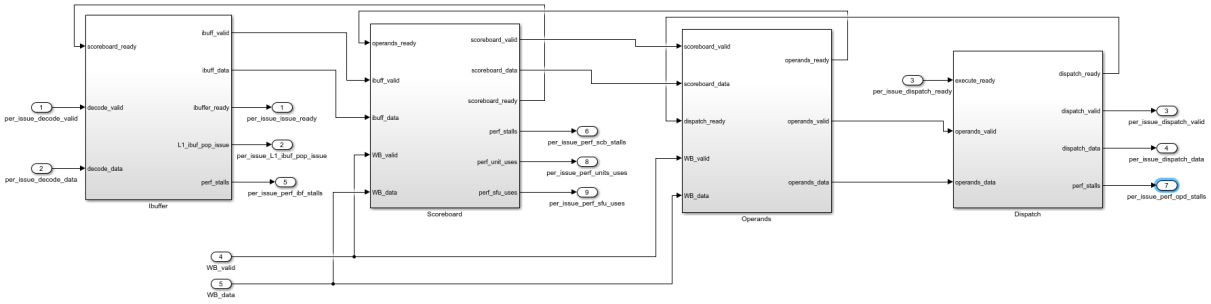


Figure 4.9: Block Diagram of Issue Slice

4.4.5 Execute Stage

Execute stage is responsible for managing instruction execution in a GPU architecture. It integrates multiple execution units, including ALU, LSU, SFU, and optionally, FPU. Each unit interacts with its respective dispatch, commit, and memory interfaces, as well as the overall control logic. Key responsibilities include:

- **Dispatch:** Dispatches the data of each thread to the appropriate execution unit based on the instruction type. It supports configurable block sizes, lane counts, and output buffering to handle diverse workloads.
- **Gather:** Recombines each thread data from the execution units into a single output, ensuring that the results are correctly aligned and formatted for further processing.
- **Local Memory Access:** Manages data access to the local memory, including load/store operations and data transfers between the execution units and memory banks.

Dispatch Unit

Recall that 1 issue slice is equal to 8 warps, and each warp has 32 threads. In 1 clock cycle, only 1 warp is selected, but due to stalling, we can have multiple warps executing in parallel in the same cycle if the warps are from different issue slices. Each issue slices is connected to an execution block, and the number of execution blocks is configurable.

- **Batch Dispatch Logic**
  If the number of issue slices is greater than the number of blocks, the slices are divided to batches. The unit uses an arbiter to select the next batch of valid dispatch instructions. The selected batch index is updated on each cycle. It keeps track of batch indices for dispatching instructions in a round-robin or priority-based manner.

- **Partial Thread Handling**
  If the number of threads is greater than the number of lanes, the unit splits threads into smaller groups (packets) based on the number of lanes. It handles the start (sop), end (eop), and intermediate packets for partially dispatched threads.

- **Data Routing**
  It maps dispatch instructions to the appropriate execution blocks based on the size of the block and batch index. It uses elastic buffers to synchronize data flow between the dispatch and execution stages.

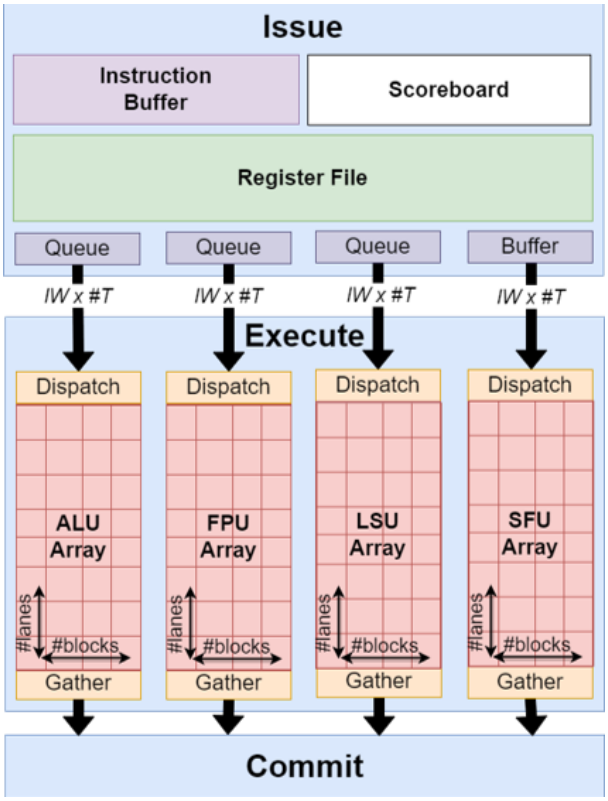


Figure 4.10: Dispatch and Gather of Threads

ALU

It handles the following:

- **Arithmetic Operations:**
  - Performs integer addition (`ADD`), subtraction (`SUB`), and comparison (`SLT, SLTU`).
  - Supports wide and narrow operations (e.g., `ADDW, SUBW`).
- **Logical Operations:**
  - Implements logical operations such as AND, OR, XOR, and SLL (shift left logical).
  - Supports sign-extension or zero-extension for operands based on the instruction type.

- **Branch Operations:**
  - Handles branch instructions by comparing results based on the operation type.
  - Generates the branch destination address and determines whether the branch is taken or not.

FPU

Floating Point Unit (FPU) is responsible for handling floating-point operations, including single-precision (32-bit) and double-precision (64-bit) arithmetic. It supports a wide range of operations, such as addition, subtraction, multiplication, division, and square root, ensuring accurate and efficient processing of floating-point data. It is optional and can be included based on the application requirements.

SFU

Special Function Unit (SFU) contains:

- **Warp Control Unit**
  Warp Control Unit is responsible for handling GPU custom instructions such as:
  - **Branch Divergence/Convergence:** Threads in a warp move in a lockstep executing the same instruction. However, Branch Divergence happens when there is a condition on a certain thread within a warp so that each thread can follow different execution paths. The approach used is to serialize execution of threads following different paths within a given warp. To achieve this serialization of divergent code paths, a SIMT stack is used. Each entry on this stack contains three entries:
    * A reconvergence program counter (RPC)
    * The address of the next instruction to execute (Next PC)
    * An active mask

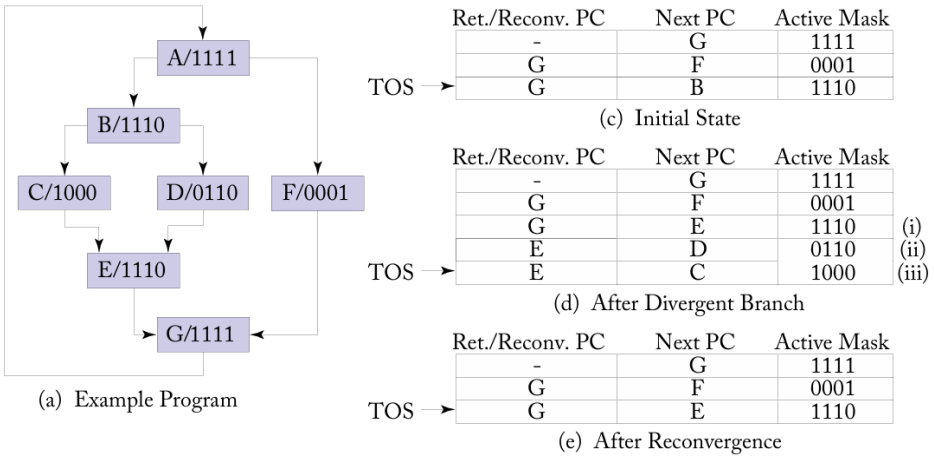


Figure 4.11: Branch Divergence and IPDOM Stack

  - **Barriers:** Branch divergence can potentially lead to deadlocks. When a branch divergence occurs, we must choose whether to execute the if or else

path first. This can cause issues if there is a dependency between the statements inside the if block and those inside the else block. The solution is to alternate execution between the if and else statements. To ensure proper synchronization and avoid deadlocks, we introduce barriers at the reconvergence point, allowing warps to synchronize before proceeding.

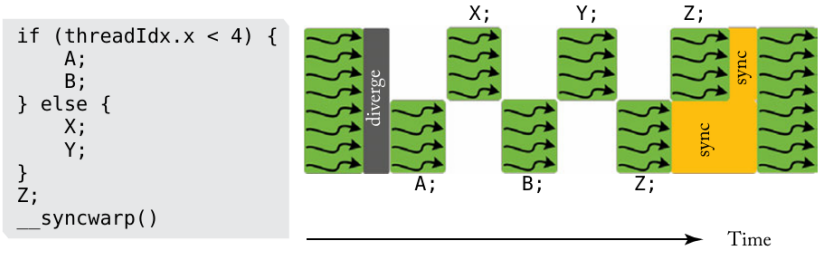


Figure 4.12: Barrier Handling in SIMT Execution

- **TMC / PRED:** Responsible for thread mask control (based on a predicate or not).
- **WSPAWN :** Responsible for spawning warps.

- **CSR Unit**
  The CSR (Control and Status Register) unit is a critical component in RISC-V processors, responsible for managing various system-level and performance-related control registers. It handles the read and write operations for these registers, supporting architectural extensions such as floating-point operations. The CSR unit also facilitates interaction with performance counters, providing insights into metrics like active warps, thread masks, and memory performance. By enabling programmable control over processor behavior, it plays a vital role in both functional and performance aspects of the system.

LSU

The Load Store Unit (LSU) is a critical module in GPU architectures designed for efficient memory management. It is responsible for handling memory operations, including loads, stores, and atomic operations, ensuring data integrity and maximizing memory throughput. Key responsibilities include:

- **Address Calculation:** Combines source operand data and an immediate offset to compute the full memory address for each active lane.
- **Byte Enable Handling:** Dynamically generates byte-enable signals based on operation size (e.g., 8-bit, 16-bit, 32-bit, etc.).
- **Memory Scheduler Integration:** Sends memory requests through the memory scheduler for optimized arbitration and access to memory channels.
- **Misaligned Access Detection:** Asserts an error if an instruction attempts a misaligned memory access based on its operation size.
- **Response Formatting:** Formats data from memory for each thread, applying alignment and sign-extension (or zero-extension) as needed.

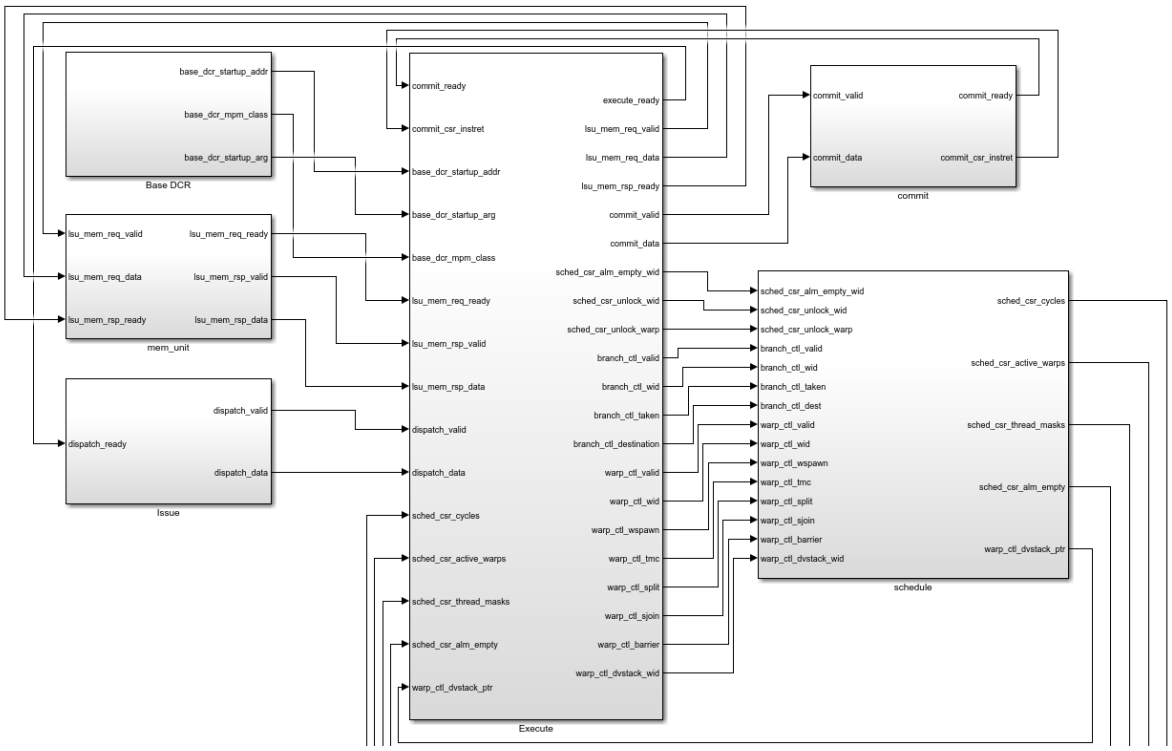


Figure 4.13: Block Diagram of the Execute Stage

4.4.6 Memory Unit

The memory unit is designed to handle a wide range of memory operations for GPUs. It manages memory interactions across local memory (LMEM) and data cache (DCACHE) while supporting multi-lane and multi-block configurations. The module ensures high performance through arbitration, coalescing, and efficient memory access strategies. Key responsibilities include:

- **Local Memory Management (Optional)**
  - Implements a local memory system for faster data access and reduced latency.
  - Supports multi-bank memory architectures with configurable sizes and buffering.
- **Data Cache Interaction**
  - Interfaces with a data cache to handle global memory requests efficiently.
  - Implements coalescing mechanisms to optimize memory traffic.
- **Arbitration Between Memory Systems**
  - Uses configurable arbitration schemes to prioritize requests between local and global memory.
  - Includes switches and adapters for efficient communication.

Local Memory Switch

The local memory switch module is a key component that arbitrates between global and local memory requests in a GPU's memory subsystem. This module ensures that memory requests are correctly routed to either local memory or global memory and handles responses accordingly.

- Decides whether a memory request is intended for local or global memory.
- Buffers requests and responses using elastic buffers to decouple the pipeline stages.
- Uses an arbiter to prioritize and merge memory responses from local and global memory.

Memory Coalescer

The memory coalescer module aggregates smaller memory requests into larger ones to optimize memory access patterns. It handles input requests and responses, coalesces them, and forwards them to the memory subsystem.

- **Input Request Handling**
  - The module accepts multiple parallel input requests, validates them, and identifies overlapping or coalescable requests based on their addresses. This reduces redundant requests to downstream memory units.
- **Output Request Generation**
  - Input requests are aggregated and merged to form larger output requests. Each output request represents multiple coalesced input requests, optimizing memory bandwidth.

Adapter

The adapter (Dcache/Lmem) module bridges the LSU memory interface and the memory bus interface. It facilitates memory requests and responses across multiple lanes by packing and unpacking data streams efficiently.

- **Unpacking Submodule**
  - This submodule unpacks incoming LSU memory requests into individual lane-specific requests, distributing them across Number of Lanes.
- **Packing Submodule**
  - This submodule packs incoming memory responses from individual lanes into a single LSU memory response.

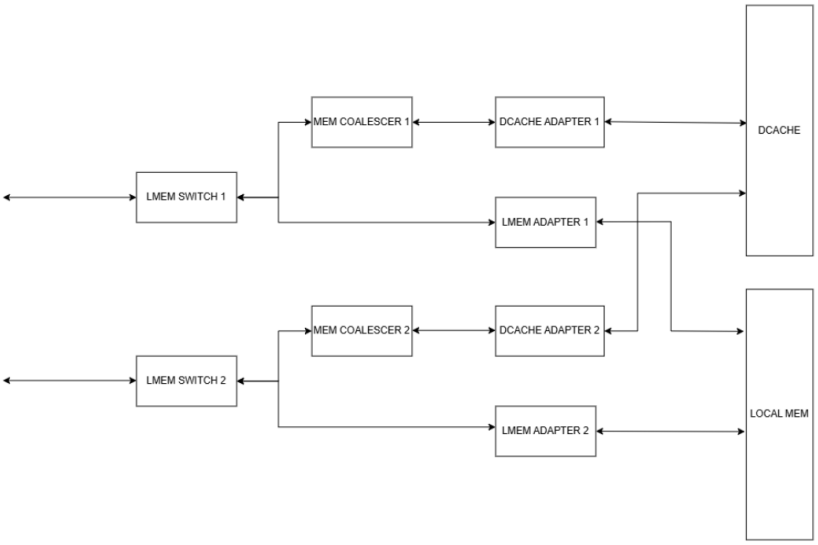


Figure 4.14: Memory Unit Block Diagram

4.4.7 Commit Stage

The Commit Stage is responsible for finalizing the execution of instructions, updating architectural state, and tracking performance metrics. It plays a crucial role in ensuring instructions are correctly retired and results are committed to the register file. Key responsibilities include:

- **Arbitration and Prioritization:** Resolves contention among execution units by arbitrating between valid instructions from multiple sources.
- **Writeback Data Handling:** Manages the transfer of execution results to the register file in the issue stage.
- **Instruction Retirement:** Marks instructions as completed and retires them from the pipeline, and updates the CSR with such performance counters.
- **Warp Management:** Signals the schedule stage about completed warps, enabling efficient management of pipeline resources.

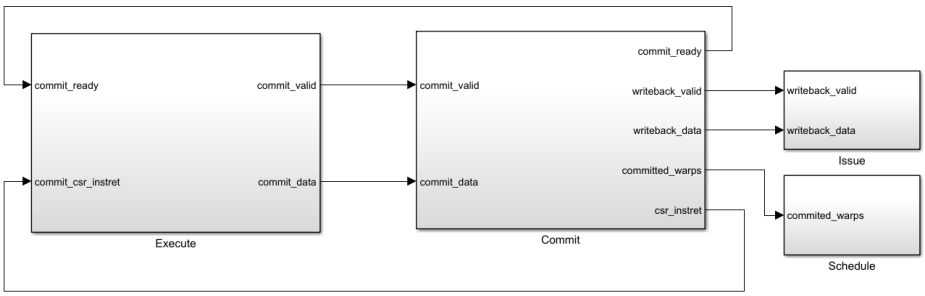


Figure 4.15: Block Diagram of the Commit Stage

4.4.8 Interfaces between Stages

Handshake Protocol Using Skid Buffers

Skid buffers are used to manage data flow between different stages. The handshake protocol ensures that data is transferred correctly and efficiently between these stages even when there is a downstream stage is stalled.

- **Fire In**: When the upstream stage has valid data to send, it asserts the "Fire In" signal. This indicates that the data is ready to be stored in the skid buffer.
- **Store Data**: If the downstream stage is stalled (i.e., it cannot accept new data), the skid buffer temporarily stores the incoming data. This prevents data loss and allows the upstream stage to continue processing without waiting for the downstream stage to become ready.
- **Fire Out**: Once the downstream stage is ready to accept new data, the skid buffer asserts the "Fire Out" signal. This indicates that the stored data is now being forwarded to the downstream stage.

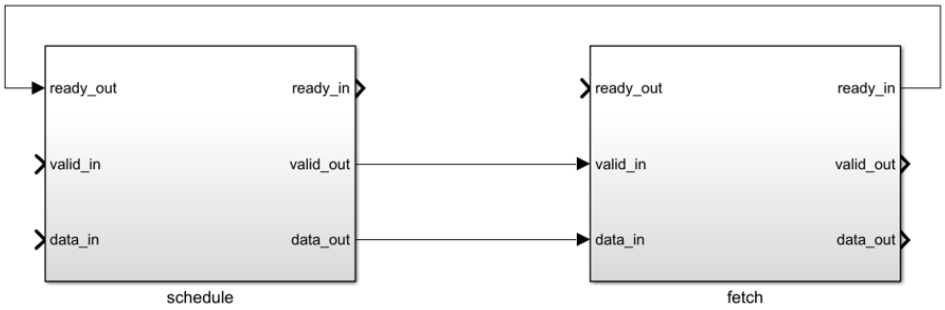


Figure 4.16: Skid Buffer Handshake Protocol between Schedule & Fetch Stages

Stage	**Output Data**
Schedule Stage	PC, thread masks, warp ID
Fetch Stage	Schedule data, instruction
Decode	Schedule data, execution type, op type, op arguments, write-back flag, source and destination registers
Issue	Schedule data, op type, op arguments, writeback flag, source and destination registers
Execute	Schedule data, writeback flag, result, destination register, packet ID, eop, sop
Commit	Schedule data, result, destination register, packet ID, eop, sop

Table 4.2: Data Propagation through the Pipeline Stages

4.5 Software Stack

The Vortex software stack is designed to facilitate the seamless execution of both host and GPU code, enabling efficient communication and task scheduling between the host and the Vortex GPGPU. The stack is divided into two main components: the host compilation flow and the GPU compilation flow. Each component plays a critical role in ensuring that the system can effectively manage memory operations, kernel launches, and hardware-specific optimizations.

4.5.1 Host Compilation Flow

The flow goes through the following steps:

1. **Host Code Development:**
   - Host code is written in C (.c files).
   - The host code manages overall execution flow, memory operations (e.g., data transfers between host and device), and kernel launches (e.g., initiating GPU kernel execution).
2. **Host Compilation:**
   - The host code is compiled using a standard C compiler (e.g., GCC, Clang).
   - The compiler generates object files (.o files) from the .c files, and an executable file that interacts with the Vortex runtime library.
3. **Linking:**
   - The object files are linked with the Vortex runtime library.
   - The runtime library provides functions for memory management and kernel execution (e.g., `vx_start` & `vx_copy_to_dev`).
4. **Executable Generation**

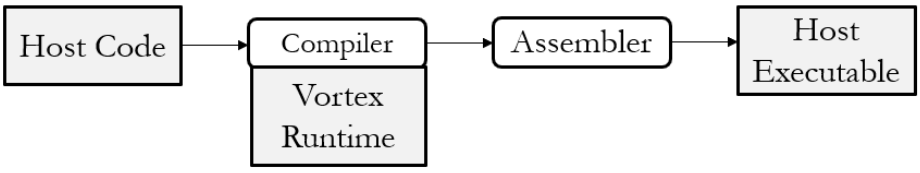


Figure 4.17: Host Compilation Flow

4.5.2 GPU Compilation Flow

The flow goes through the following steps:

1. **Kernel Code Development:**
   - Kernel code is written in OpenCL or CUDA.

- It can be written in C++ by adding Vortex custom library.

2. **Front-end Compilation (PoCL):**
   - The kernel code is processed by a front-end compiler.
   - The front-end compiler generates Intermediate Representation (IR) from the kernel code, which can help improve optimization.
   - Uses built-in libraries for math functions.
3. **Back-end Compilation (LLVM):**
   - The IR is passed to a back-end compiler.
   - The back-end compiler generates the kernel executable.
   - Uses Vortex Kernel Library for accessing runtime information and controlling program flow (e.g., `vx_spawn_tasks` & `vx_num_threads`).
   - Uses Vortex ISA which inlcudes custom RISC-V instructions for the Vortex GPGPU.

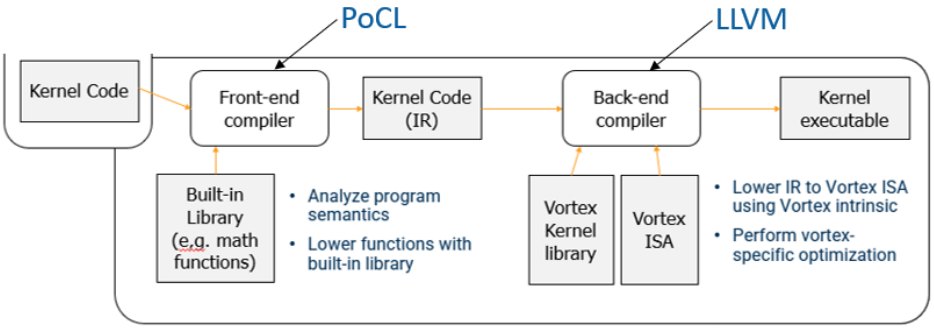


Figure 4.18: GPU Compilation Flow

4.5.3 Vortex Execution flow

The flow for the commands required for device communication, as outlined in the host code, adhere to the following execution sequence:

1. **They are linked with the frontend runtime library.**
   - The frontend runtime library outlines how frontend methods can be executed using the Vortex runtime library.
2. **The Vortex library carries out the operation using the Vortex host interface.**
   - Vortex host interface, in turn, performs the operation on the Vortex processor, which is connected to the external bus via the Vortex host interface.

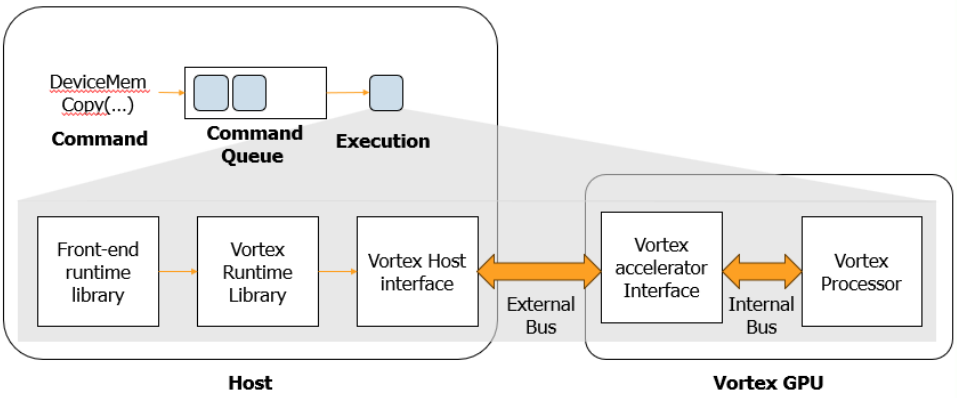


Figure 4.19: Execution Flow

4.5.4 Parallel Programming Model

Threads & Warps

- **Thread**: The smallest execution unit, running one instance of a kernel with its own registers and local memory.
- **Warp**: A group of 32 threads that launch together from the same program counter. The SM's warp scheduler interleaves ready warps to hide latencies (e.g., memory fetches).
- **Divergence**: If threads in a warp take different branches, execution is serialized into separate passes, reducing throughput.

Thread Blocks

- **Definition**: A block is a collection of up to 1,024 threads (hardware-dependent) that execute on the same SM.
- **Shared Memory**: Threads within a block share user-managed on-chip memory (backed by the SM's L1 cache), enabling low-latency data exchange.
- **Synchronization**: Threads coordinate via `__syncthreads()`, acting as a barrier without terminating the kernel.
- **Dimensionality & Indexing**: Blocks can be 1D, 2D, or 3D (e.g., `dim3 blockDim(16,16,1)`), with each thread addressed by `threadIdx.x`, `threadIdx.y`, `threadIdx.z`.

Grids

- **Definition**: The grid is the full set of blocks for one kernel launch, also arranged in 1D-3D (e.g., `dim3 gridDim(64,64,1)`).
- **Scalability**: Independent blocks can be scheduled across multiple SMs, enabling millions of threads to run in parallel.
- **Indexing**: Each thread's global index is computed as

$$\texttt{gid\_x} = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}.$$

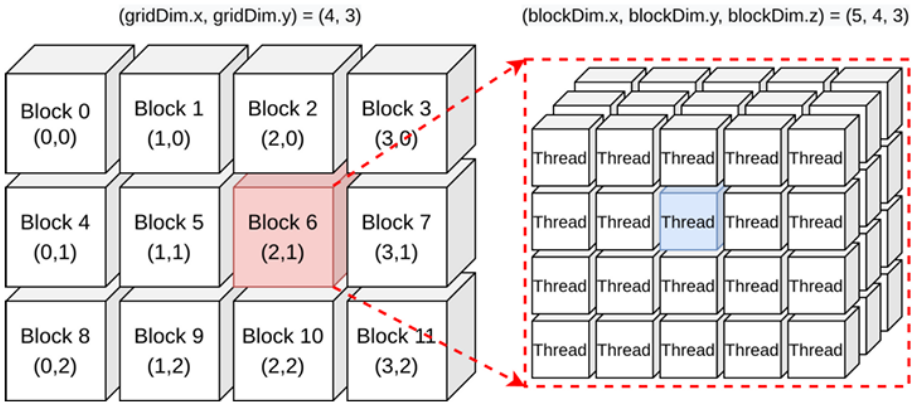


Figure 4.20: Thread Blocks and Grids

Kernel Launch & Streams

- **Kernel**: A device function marked `__global__`, invoked by the host:

```
kernel<<<gridDim, blockDim, sharedMemBytes, stream>>>(...);
```

- **Streams**: Independent command queues. Kernels in different streams can execute concurrently (hardware permitting), enabling overlap of computation and data transfer.
- **Launch Overhead**: Starting or ending a kernel has nontrivial latency; maximize per-launch work and use intra-block synchronization.

Synchronization & Communication

- **Intra-block**: Efficient via `__syncthreads()` and shared memory.
- **Inter-block**: Requires global memory, multiple kernel launches, or cooperative groups—no direct on-chip sync.
- **Best Practice**: Minimize inter-block dependencies and global barriers to keep SMs fully utilized.

Chapter 5

AXI 4 Protocol Overview

5.0.1 overview

In modern System-on-Chip (SoC) designs, interoperability and standardization are essential to managing communication between various intellectual property (IP) cores. All custom and third-party IP cores used in this project communicate using the AXI (Advanced eXtensible Interface) protocol, a widely adopted communication standard developed by ARM as part of the AMBA (Advanced Microcontroller Bus Architecture) family. AXI provides a robust and scalable solution for high-performance interconnects, offering

low-latency, high-bandwidth data transfers with support for burst transactions. This protocol forms the backbone of internal communication in the system, ensuring that all cores—whether they are memory-mapped, control-based, or high-speed data streaming blocks—can interact through a common and predictable interface.

5.0.2 AXI Read and Write Operations

The AXI4 protocol enables multiple data transfers to occur within a single transaction, significantly enhancing data throughput in applications where large amounts of data must be exchanged between modules. This form of communication is commonly known as a burst transfer. All AXI-based communication relies on memory addresses, where

each address is assigned a specific function as defined by the RTL and the top-level design. These addresses dictate how and where data transactions occur across the system. AXI4 supports three types of burst transactions: FIXED, INCR, and WRAP. Each burst

type handles address progression differently, allowing designers to optimize data transfers depending on the access pattern:

- **FIXED**: Maintains a constant address for all data beats. This is ideal for scenarios where the same memory location must be accessed repeatedly.
- **INCR**: Increments the address with each successive beat, making it well-suited for linear memory accesses.
- **WRAP**: After a predetermined number of beats, the address wraps back to a base address. This is commonly used for cache line transfers.

The AXI4-Lite interface, a simplified version of AXI4, does not support burst transfers and instead handles only single data transactions. In contrast, the AXI4-Stream interface facilitates continuous, unidirectional data flow between a master and a slave without involving address signals, making it highly efficient for streaming data applications.

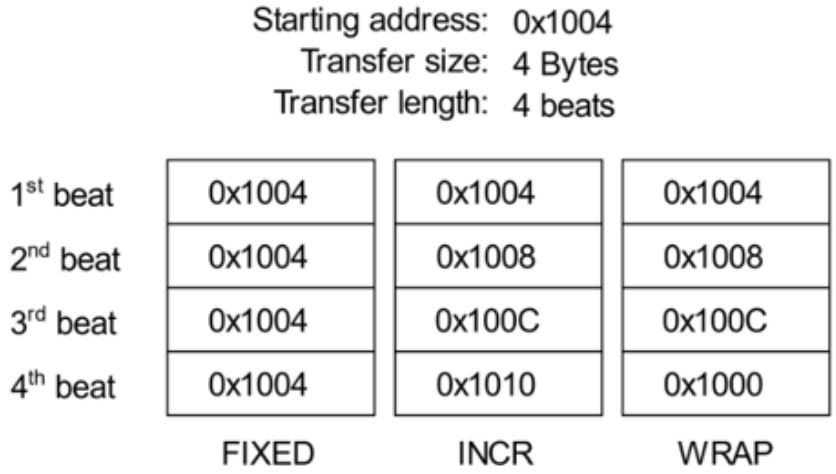


Figure 5.1: AXI_4_brusts

5.0.3 AXI Channel Architecture

The AXI protocol operates through five independent channels, allowing multiple transactions to occur simultaneously:

- **Write Address Channel (AW)**
  Sends the target address and control information for a write.
- **Write Data Channel (W)**
  Transfers actual data to be written.
- **Write Response Channel (B)**
  Returns the status of the write operation.
- **Read Address Channel (AR)**
  Sends the target address and control information for a read.
- **Read Data Channel (R)**
  Returns the requested data and read status.

Each channel uses a handshake mechanism (`VALID` and `READY` signals) to synchronize data flow, allowing for decoupling and pipelining of requests and responses.

Figure 5.2: AXI4_channels

5.0.4 Basic AXI4 Signals

- A **VALID** signal is asserted when valid information is driven by the information transmitter.
- A **READY** signal is asserted when the information receiver is ready to receive.
- A **LAST** signal indicates the transfer of the final data item in a transaction (used in data channels).

5.0.5 AXI4-Lite

AXI4-Lite is a simplified subset of the AXI4 protocol, tailored for use in low-throughput control and register-mapped interfaces. It is often employed for configuration and status register accesses within IP blocks, where high-performance data transfers are not required.

- **Burstless Transactions**: AXI4-Lite only supports single-beat transactions, meaning that every read or write transfer consists of exactly one data item. As a result, control signals such as `WLAST` or `RLAST` are not required.

Table 5.1: AXI Protocol Signal Overview

Signals	Read Address	Read Data	Write Address	Write Data	Write Response
Handshake	ARVALID, ARREADY	RVALID, RREADY	AWVALID, AWREADY	WVALID, WREADY	BVALID, BREADY
Information	ARADDR	RDATA, RLAST	AWADDR	WDATA, WLAST	BRESP
Global	ACLK, ARESETn				

- **Fixed Data Width**: The interface supports either a 32-bit or 64-bit data bus. All transactions must align with the full width of the data bus, ensuring predictable and consistent access patterns.
- **Non-Modifiable and Non-Bufferable Accesses**: All accesses are non-modifiable and non-bufferable, meaning they cannot be altered or delayed in the pipeline, which is essential for deterministic behavior in control logic.
- **No Exclusive Access Support**: AXI4-Lite does not support exclusive accesses, which are used in atomic operations in more complex systems. This simplifies its implementation and use.

AXI4Lite Signals

Global	Read Address	Read Data	Write Address	Write Data	Write Response
ACLK	ARVALID	RVALID	AWVALID	WVALID	BVALID
ARESETn	ARREADY	RREADY	AWREADY	WREADY	BREADY
	ARADDR	RDATA	AWADDR	WDATA	BRESP
	ARPROT		AWPROT	WSTRB	

Part II

Implementation

Chapter 6

NTT Hardware Design

1. Introduction

The Radix-2 Single-Path Delay Feedback (SDF) architecture provides an efficient and hardware-friendly design for implementing the Number Theoretic Transform (NTT). This architecture processes one input coefficient per clock cycle and begins generating output coefficients after the internal pipeline stages are filled. Each computational stage includes a butterfly unit paired with a memory block that stores intermediate values. Figure 1 provides a high-level overview of the proposed SDF-NTT architecture.

2. SDF Architecture Operation

A butterfly operation in NTT requires two input coefficients spaced by a stage-dependent offset. For a 256-point Decimation-In-Frequency (DIF) NTT, the first stage processes coefficient pairs such as (a_i, a_{i+128}) for $i = 0$ to 127. In the second stage, the offset becomes 64, and this pattern continues until the final stage, where the offset is 1.

Each stage operates in two distinct phases:

1. **Input Phase:** The first half of the coefficients is stored in memory.
2. **Computation Phase:** While the second half of the coefficients is streamed in, the first half is simultaneously read from memory to form butterfly input pairs.

The butterfly output is split into two streams:

- One stream is directly forwarded to the next stage
- The other stream is stored for reuse in subsequent operations

This pattern of operation continues across stages with progressively halving offsets.

3. System-Level Architecture

3.1 SDF-NTT Before Optimization

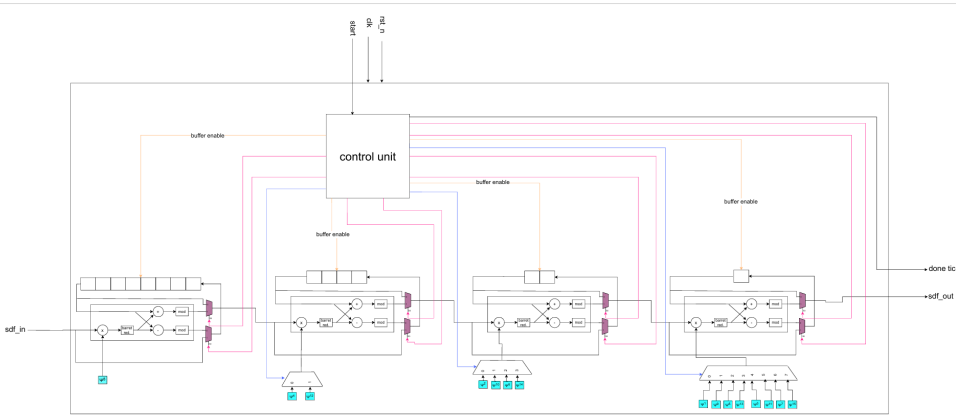


Figure 6.1: SDF-NTT Architecture Before Optimization

Signal	**Description**	**Width**
clk	System clock	1 bit
rst_n	Negative edge reset signal	1 bit
start	Active high enable signal that must be asserted before passing the input data; must remain high until the last output is generated	1 bit
sdf_in	Input data port	64 bits (parameterized)
sdf_out	Output data port	64 bits (parameterized)
done_tick	Flag that becomes high with the last output data	1 bit

Table 6.1: Signals for SDF-NTT Before Optimization

3.2 SDF-NTT After Optimization

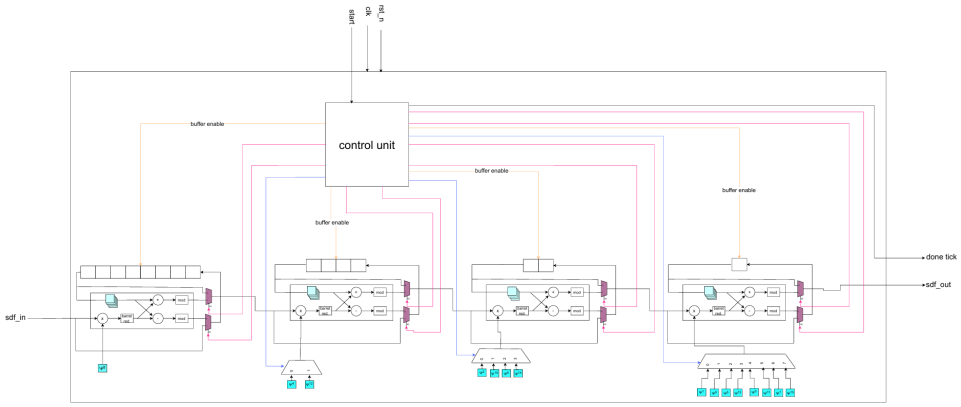


Figure 6.2: SDF-NTT Architecture After Optimization

Signal	Description	Width
clk	System clock	1 bit
rst_n	Negative edge reset signal	1 bit
start	Active high enable signal that must be asserted before passing the input data; must remain high until the last output is generated	1 bit
sdf_in	Input data port	64 bits (parameterized)
sdf_out	Output data port	64 bits (parameterized)
done_tick	Flag that becomes high with the last output data	1 bit

Table 6.2: Signals for SDF-NTT After Optimization

Optimization Details

To improve throughput, we applied a 4-stage pipelining technique to the Barrett Reduction unit. However, this introduced a data hazard due to timing mismatches in the pipeline.

Resolution Strategy

- A 4-stage register chain was inserted into the path of the first input of the butterfly unit. This aligned the timing between the two butterfly inputs, allowing the first stage to operate correctly.
- After implementing this fix, new data hazards were observed in the subsequent stages due to overlapping data dependencies.
- To resolve this issue, we introduced a stalling mechanism for the enable signal of the first-stage buffer. Specifically:

  - When the first-stage buffer is filled for the second time, the enable signal is temporarily de-asserted.
  - This stall is maintained until the second stage completes its first fill cycle, after which the enable signal of the first stage is reasserted again.

This control logic ensures proper synchronization across all stages, allowing the pipeline to operate without data hazards.

4. Extending to INTT Support

4.1 Reducing Memory for Twiddle Factors

An NTT-Based Nega Cyclic operation requires n different powers of the twiddle factor ψ, and similarly, an INTT requires n different powers of ψ^{-1}. Each stage of the NTT or INTT uses only a subset of these powers:

For 16-point NTT:

- Stage 1 uses $n/16$ powers
- Stage 2 uses $n/8$ powers
- Stage 3 uses $n/4$ powers
- Stage 4 uses $n/2$ powers

For 16-point INTT:

- Stage 1 uses $n/2$ powers
- Stage 2 uses $n/4$ powers
- Stage 3 uses $n/8$ powers
- Stage 4 uses $n/16$ powers

Therefore, an NTT/INTT implementation must store at least $2n$ twiddle factor powers (n for NTT and n for INTT). To reduce this memory requirement, we exploit the mathematical properties of twiddle factors:

$$\psi^{2n} \equiv 1 \pmod{q}, \quad \text{and} \quad \psi^{n} \equiv -1 \pmod{q}$$

Using these properties, we can express:

$$\psi^{-i} = \psi^{2n} \cdot \psi^{-i} = \psi^{n} \cdot \psi^{n-i} = -\psi^{n-i}$$

This means that ψ^{-i} can be derived using $-\psi^{n-i}$, which is computationally inexpensive in hardware. It only requires a small subtraction circuit to compute the modular inverse of a twiddle factor during the INTT. This optimization reduces twiddle factor storage by 50% at the cost of $\log_2(n)$ additional subtraction units.

4.2 Eliminating the Multiplication by n^{-1}

In INTT, the final coefficients must be scaled by $n^{-1} \mod q$. Although straightforward, this adds n extra operations and increases latency.

To eliminate this overhead, we merge the scaling with the post-processing step using the Number Theoretic Transform with Constant (NWC) method. A technique replaces the final multiplication by n^{-1} with 2^{-1} in each butterfly operation. This is efficient for odd primes q, as division by 2 in $\mathbb{Z}_q$ can be computed as:

$$a/2 \equiv (a \gg 1) + a[0] \cdot \left(\frac{q+1}{2}\right) \pmod{q}$$

This avoids the need for an extra full-width multiplier by replacing it with two adders in the butterfly unit. We adopted this technique in our implementation to reduce the latency of INTT operations.

5. Integrated NTT/INTT Architecture

To extend the functionality of the SDF to support inverse NTT operations in addition to forward NTT, we integrate the NTT butterfly unit with the INTT butterfly unit as shown in Figure 3. This unified approach allows the same architecture to perform both operations efficiently.

Important Note: This SDF implementation requires the input data to be in normal order for both NTT and INTT operations, and the output data is generated in bit-reversed order. To enable calculation of INTT after NTT for the same data points using the same architecture, we use an output address mechanism to rearrange the output points in normal order before feeding them back to the input.

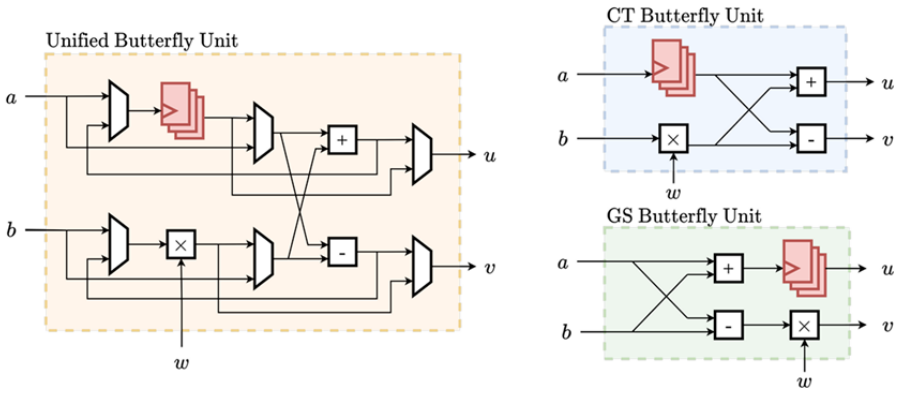


Figure 6.3: Integrated NTT/INTT Butterfly Unit

6. Complete System Overview

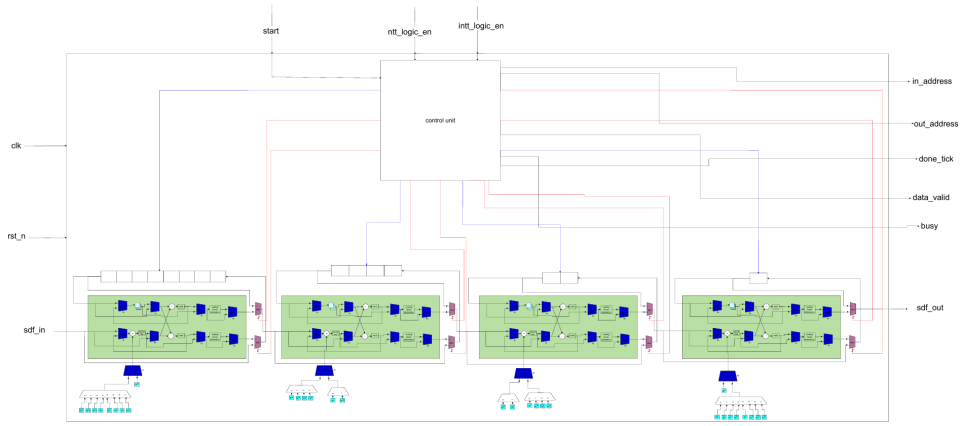


Figure 6.4: Complete SDF-NTT/INTT System Architecture

Signal	**Description**	**Width**
clk	System clock	1 bit
rst_n	Negative edge reset signal	1 bit
ntt_logic_enable	Active high enable signal to enable the data path for the NTT operation that must be asserted before passing the input data; must remain high until the last output is generated	1 bit
intt_logic_enable	Active high enable signal to enable the data path for the INTT operation that must be asserted before passing the input data; must remain high until the last output is generated	1 bit
start	An active high signal that must be asserted when the first input data is ready to be sampled. When it is high at a certain clk edge the system captures it and starts its operation and the system does not listen to it any more until the operation is finished	1 bit
sdf_in	Input data port	32 bits (parameterized)
sdf_out	Output data port	32 bits (parameterized)
output_address	Specifies the order of each output point to be stored in memory	$\log_2(n)$ bits, where n is the polynomial size
in_address	Specifies the address of each input point if they are read from a memory	$\log_2(n)$ bits, where n is the polynomial size
done_tick	Flag that becomes high with the last output data	1 bit

Signal	Description	Width
data_valid	Flag that becomes high with each output data point	1 bit
busy	Flag that becomes high when the system samples the first input until the last output comes out	1 bit

Table 6.3: Signals for Complete SDF-NTT/INTT System

7. Simulation Results

7.1 Python Model Validation

The architectural correctness was confirmed through Python model validation, ensuring that the theoretical framework matches the practical implementation.

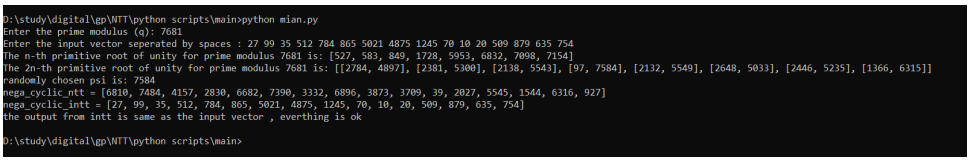


Figure 6.5: Python Model Validation Results

7.2 Pre-Optimization SDF-NTT Performance

The basic operation characteristics, including latency and throughput measurements, were evaluated for the initial implementation.

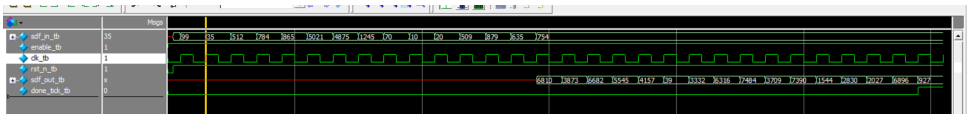


Figure 6.6: Pre-Optimization SDF-NTT Performance Metrics

Note: The hardware implementation outputs data in bit-reversed order, whereas the Python reference model produces results in natural order. This will be handled in the final version of the system in sections 7.3 and 7.4. As shown in the waveform, the first output comes out with feeding in the last input and no latency between the output points.

7.3 Post-Optimization Pipelining Validation

The effectiveness of the 4-stage pipelining optimization was validated through comprehensive simulation tests.

Note: We added an extra flip-flop at the input of each butterfly unit stage to increase frequency.

NTT Test

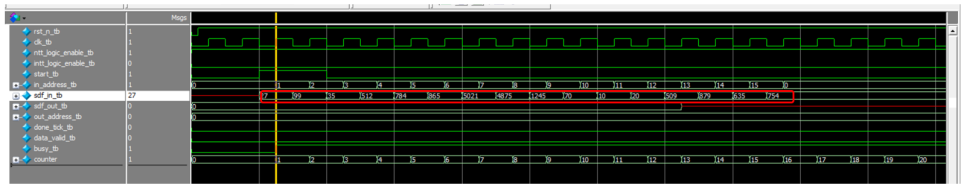


Figure 6.7: NTT Input Vector Snapshot

```
#          Using alternate file: ./wlftjjrsn2
#
# test case           1 succeeded : expected ==>       6810 , got==>       6810
# test case           2 succeeded : expected ==>       7484 , got==>       7484
# test case           3 succeeded : expected ==>       4157 , got==>       4157
# test case           4 succeeded : expected ==>       2830 , got==>       2830
# test case           5 succeeded : expected ==>       6682 , got==>       6682
# test case           6 succeeded : expected ==>       7390 , got==>       7390
# test case           7 succeeded : expected ==>       3332 , got==>       3332
# test case           8 succeeded : expected ==>       6896 , got==>       6896
# test case           9 succeeded : expected ==>       3873 , got==>       3873
# test case          10 succeeded : expected ==>       3709 , got==>       3709
# test case          11 succeeded : expected ==>         39 , got==>         39
# test case          12 succeeded : expected ==>       2027 , got==>       2027
# test case          13 succeeded : expected ==>       5545 , got==>       5545
# test case          14 succeeded : expected ==>       1544 , got==>       1544
# test case          15 succeeded : expected ==>       6316 , got==>       6316
# test case          16 succeeded : expected ==>        927 , got==>        927
# Break in Module sdf_tb at sdf_top_module_tb.v line 65
# Simulation Breakpoint: Break in Module sdf_tb at sdf_top_module_tb.v line 65
```

Figure 6.8: Post-Optimization Pipeline NTT Validation Results

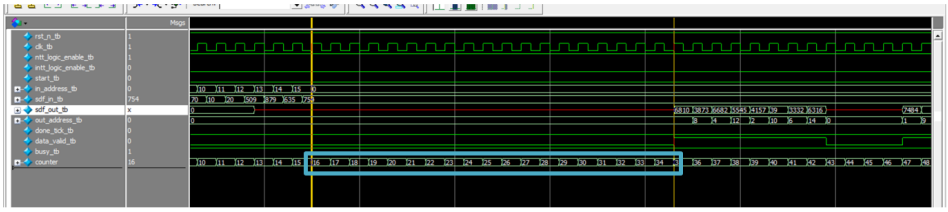


Figure 6.9: First Output Latency

As shown in this snapshot there is a resulted latency between the last input and first output.

The delay till the first output comes out = number of pipeline stages + number of stalling cycles + buffer size of 1st stage + size of polynomial + added latecy due to the flip-flop at the input -1 .

= 4 + 4 + 8 + 16 + 4 -1 = 35.

Since the output port is in the last stage (4th stage) so this latency is 4 cycles

So the latency= 35 – the delay of the last input = 35 -16 = 19 cycles.

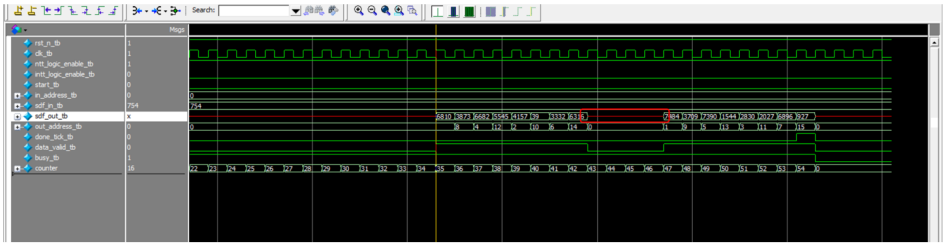


Figure 6.10: Second Bunch of Data Latency

Also, we have an added latency between the first half of the output data and the second half of the output data. This latency equals the number of pipeline stages:

$$= 4 \text{ cycles}$$

INTT Test

Same behavior as the NTT but with different inputs and outputs.

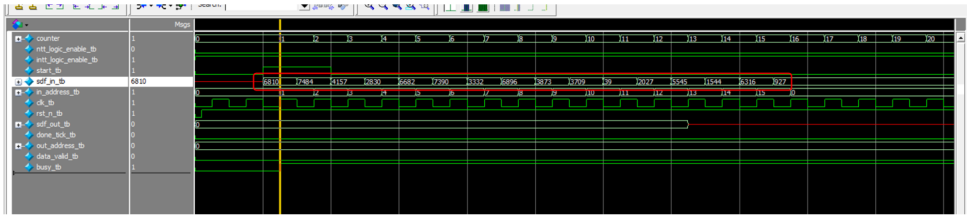


Figure 6.11: INTT Input Vector Snapshot

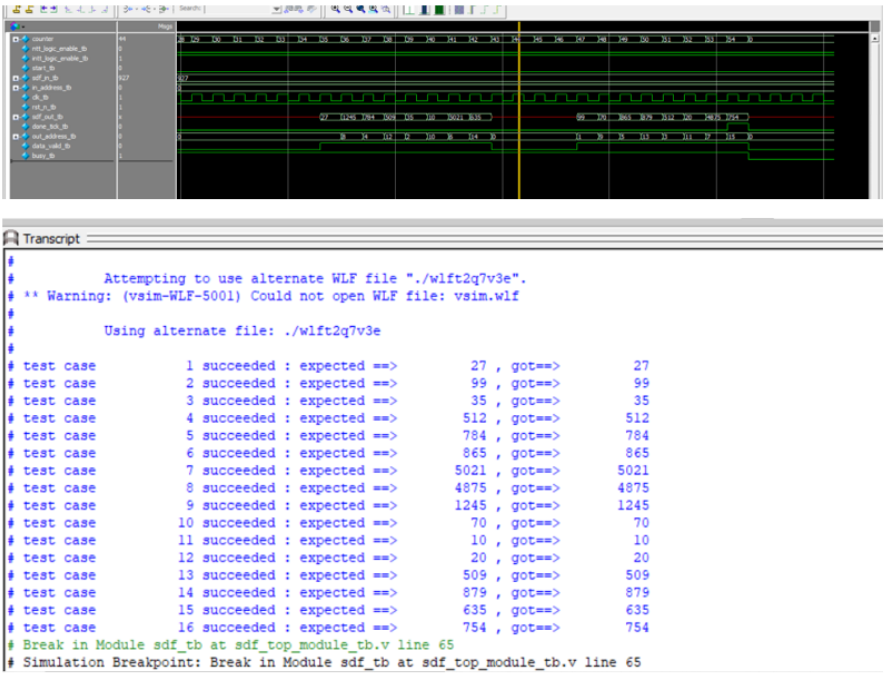


Figure 6.12: Post-Optimization Pipeline INTT Validation Results

Chapter 7

Integration with Vortex GPGPU

7.1 Integration Types

In the pursuit of enhanced computational performance and energy efficiency, hardware accelerators have become an integral part of modern processor architectures. These accelerators are tailored to execute specific tasks significantly faster than general-purpose cores, making them ideal for domains such as signal processing, cryptography, and machine learning.

When integrating accelerators into a system, two primary architectural paradigms are commonly adopted: **tightly-coupled** and **loosely-coupled** integration. Tightly-coupled accelerators are directly embedded within the processor's execution pipeline, often as custom instruction extensions to the Instruction Set Architecture (ISA). This approach ensures minimal communication overhead and low latency, making it ideal for lightweight computations and time-sensitive operations. On the other hand, loosely-coupled accelerators operate as standalone modules connected to the processor through the system bus or interconnect. These units typically handle computationally intensive workloads in bulk and are accessed via memory-mapped I/O, trading off some communication latency for greater processing throughput.

7.1.1 Loosely-coupled Integration

GPUs themselves are, by definition, standalone accelerators loosely coupled to the host processor. In a typical system, a discrete GPU resides on a separate card and interfaces with the CPU over a PCI Express (PCIe) bus; it maintains its own dedicated memory and address space, communicating with the CPU via explicit data transfers and DMA operations. PCIe, while high-bandwidth relative to many interconnects, is still an order of magnitude slower than on-chip memory buses, and its latency and protocol overheads can become a significant bottleneck in GPU-accelerated workloads.

Similarly, studies in GPU-accelerated networking have demonstrated that PCIe transfer overhead can substantially limit performance gains, especially when GPU kernels require frequent host interaction to fetch or store data.

Introducing an additional loosely-coupled accelerator "beyond" the GPU—i.e., connecting a second standalone unit to the GPU over another bus layer—essentially compounds

these communication overheads. Each layer of off-chip transfer incurs setup latency, bus arbitration delays, and potential cache-coherence traffic, meaning that any compute speed-up must be large enough to amortize the extra data-movement costs. In practice, loosely-coupled accelerators only deliver net performance improvements when their workload granularity is sufficiently large to hide these overheads; for smaller or more interactive tasks, the cumulative latency of back-and-forth transfers can actually degrade overall throughput.

Consequently, accelerating an already loosely-coupled device like a GPU with another loosely-coupled accelerator tends to yield diminishing returns, so we have opted for a tightly-coupled approach. By integrating the NTT accelerator directly into the Vortex GPGPU pipeline, we can leverage the existing execution resources and memory hierarchy to minimize communication overhead and maximize performance. This tight integration allows us to treat the NTT accelerator as a custom instruction within the Vortex ISA, enabling seamless execution alongside other GPU workloads.

7.1.2 Tightly-coupled Integration

Tightly-coupled accelerators are designed by decomposing functionality into smaller, fine-grained units that align well with the RISC-V instruction model. This approach allows for direct integration into the pipeline, enabling each custom instruction to trigger a specific operation and accept input data within a single cycle. By breaking down the overall functionality into lightweight, instruction-friendly steps, the accelerator becomes tightly woven into the processor's execution flow. In contrast, using a monolithic or black-box design would categorize the accelerator as loosely coupled, since such designs might operate on a serial input or do very complex tasks that require huge number of cycles to complete.

7.2 NTT Kernel

For tightly-coupled accelerator integration, the best approach is to start with the kernel code and analyze the most time-consuming operations that takes many instructions or many cycles to complete. For NTT, there are mainly 2 algorithms for the kernel discussed in [6]: **Merge-NTT** and **4Step-NTT**.

7.2.1 Merge-NTT

When to Use

The Merge-NTT algorithm is ideal for moderate problem sizes (e.g. $n \leq 2^{18}$) where one can afford multiple global-memory kernel launches without overwhelming launch overhead. It fits scenarios in which:

- The host can precompute and load all twiddle factors in bit-reversed order once, deferring final reordering.
- The degree n is not so large that memory-bound behavior dominates performance.
- Simplicity of synchronization is desirable.

Performance Gains

Compared to a naive DFT, Merge-NTT achieves $O(n \log n)$ arithmetic complexity and minimizes bit-reversal overhead by producing bit-reversed output. On GPUs, carefully tuned kernel partitions can yield up to 20–30% lower latency for $n \approx 2^{16}$–2^{20} relative to four-step variants, due to fewer transposes and simpler memory access patterns.

Advantages

- **Simplicity:** Single kernel per major iteration or two kernels total, easy to implement with Cooley-Tukey butterflies.
- **Low overhead:** No explicit matrix transpose or correction step apart from deferred bit reversal.
- **Flexible partitioning:** Kernel count and block size can be tuned to balance occupancy and global-memory traffic.

Disadvantages

- **Memory-bound at large n:** As n grows, uncoalesced accesses and multiple kernel launches can degrade throughput.
- **Limited parallelism:** Only the inner loop is parallel; outer-loop sequential dependence requires serialized kernels when $n/2 > b_{\text{dim}}$.
- **Bit-reversal cost:** Although deferred, the final reordering may still incur overhead if explicitly needed later.

Algorithm 1 Merge Forward NTT (bit-reversed output)

Require: $a(x) \in \mathbb{Z}_q[x]/(x^n + 1)$, $n = 2^l$, precomputed twiddles $\Psi_{\text{br}}[k] = \psi^{\text{br}(k)} \bmod q$ for $0 < k < n$
Ensure: $a \leftarrow \text{NTT}(a)$ in bit-reversed order

```
t ← n,   m ← 1
repeat
    t ← t/2
    for i = 0 to m − 1 do
        j1 ← 2^i · t,   j2 ← j1 + t − 1
        for j = j1 to j2 do
            (a_j, a_{j+t}) ← CT(a_j, a_{j+t}, Ψ_br[m + i], q)
        end for
    end for
    m ← 2m
until m ≥ n
return a
```

7.2.2 Four-Step NTT

When to Use

The Four-Step NTT is preferable for very large transforms ($n \geq 2^{20}$), when spatial locality and massive parallelism offset extra transpose cost. It excels when:

- One can define an $n_1 \times n_2$ split such that n_1 and n_2 fit shared-memory capacity and yield coalesced loads.
- The application already tolerates (or can hide) one matrix transpose via a second kernel or within shared memory.
- High throughput of many independent small NTTs is required (e.g. batching in homomorphic encryption).

Performance Gains

By decomposing into n_2 n_1-point and n_1 n_2-point transforms, Four-Step NTT can improve arithmetic throughput by 15-25% over Merge-NTT at $n = 2^{22}$–2^{24}, thanks to enhanced coalescing and reduced global-memory stalls during butterfly stages.

Advantages

- **Better locality:** Most butterfly operations occur on contiguous chunks of size n_1 or n_2, improving cache/share-memory reuse.
- **Massive parallelism:** Many small transforms let the GPU saturate all SMs without sequential outer loops.
- **Embedded correction:** Twiddle-factor multiplication can be fused into NTT kernels, reducing extra passes.

Disadvantages

- **Transpose overhead:** One or more transposes (global or shared-memory) add kernel launches or extra memory passes.
- **Complex tuning:** Optimal (n_1, n_2) split and block dimensions require empirical search per device.
- **Implementation complexity:** More intricate indexing and kernel orchestration, plus handling non-square splits.

Algorithm 2 Four-Step Forward NTT

Require: $n = n_1 \cdot n_2$, input vector $a[0..n-1]$, twiddle factors Ω
Ensure: $a \leftarrow \mathrm{NTT}(a)$ in bit-reversed order

Step 1: Vector-to-matrix mapping
for $i = 0$ to $n_1 - 1$ **do**
  for $j = 0$ to $n_2 - 1$ **do**
    $B_{i,j} \leftarrow a[i \cdot n_2 + j]$
  end for
end for
Step 2: Transpose
$B \leftarrow B^\top$
Step 3: n_2 row-wise n_1-point NTTs
for $j = 0$ to $n_2 - 1$ **do**
  $B_{:,j} \leftarrow \mathrm{NTT}(B_{:,j}, \Omega_0^{\mathrm{br}}, n_1, q)$
end for
Step 4: Transpose
$B \leftarrow B^\top$
Step 5: Twiddle correction
for $i = 0$ to $n_1 - 1$ **do**
  for $j = 0$ to $n_2 - 1$ **do**
    $B_{i,j} \leftarrow B_{i,j} \cdot \Omega[i \cdot n_2 + j] \bmod q$
  end for
end for
Step 6: n_1 row-wise n_2-point NTTs
for $i = 0$ to $n_1 - 1$ **do**
  $B_{i,:} \leftarrow \mathrm{NTT}(B_{i,:}, \Omega_1^{\mathrm{br}}, n_2, q)$
end for
Step 7: Final transpose
$B \leftarrow B^\top$
Step 8: Matrix-to-vector mapping
for $j = 0$ to $n_2 - 1$ **do**
  for $i = 0$ to $n_1 - 1$ **do**
    $a[j \cdot n_1 + i] \leftarrow B_{j,i}$
  end for
end for
return a

7.2.3 Optimizations for Merge NTT

To optimize the execution of Merge NTT on GPU architectures, it is crucial to schedule threads in a way that maximizes parallelism and resource utilization. The key idea is to assign a single thread to compute each butterfly operation, effectively processing two elements of the array a at once. This design ensures balanced workload across threads, which is essential for achieving high throughput on SIMT-based GPUs.

Algorithm 3, proposed in [3], demonstrates the scheduling strategy used to map butterfly computations to threads. Variables such as `step`, `length`, `target_idx`, and `step_group` are used to derive the indices of the two array elements and the corresponding twiddle

factor for each butterfly. During execution, the threads work on independent butterfly pairs in-place, reusing the array a for both input and output.

To further improve memory efficiency, powers of the primitive root are precomputed and stored in a lookup table `psis` in shared (local) memory, enabling fast access during butterfly operations. Figure 7.1 visualizes the thread assignment for $n = 8$, highlighting the role of each scheduling variable and the data grouping across iterations. After each stage, the number of active step groups doubles, and the twiddle factors are adjusted accordingly to reflect the higher granularity of computation.

For practical GPU deployment, a constraint exists in the maximum number of threads per block (commonly 1024), which limits the directly addressable input size to 2048 elements. For larger arrays, a hierarchical approach is adopted, where the input is partitioned into blocks of 2048 elements and processed across multiple thread blocks. Each block operates independently, maintaining the same scheduling pattern as in Algorithm 3.

These optimizations—loop restructuring, thread mapping, and shared memory usage—not only reduce redundant computations but also improve memory coalescing and bandwidth utilization, leading to a more efficient Merge NTT implementation on modern GPUs.

Algorithm 3 Scheduling of Array Elements to Threads in GPU for NTT

for `length` $= 1$ to n step `length`$* = 2$ **do**
    `tid` $\leftarrow$ Global index of thread in GPU
    `step` $\leftarrow (n/$`length`$)/2$
    `psi_step` $\leftarrow$ `tid`$/$`step`
    `target_idx` $\leftarrow$ (`psi_step` $\cdot$ `step` $\cdot\, 2) +$ (`tid` mod `step`)
    `step_group` $\leftarrow$ `length` $+$ `psi_step`
    $\psi \leftarrow$ `psis[step_step]`
    $U \leftarrow a$[`target_idx`]
    $V \leftarrow a$[`target_idx` $+$ `step`]
    `// Thread assignment is completed at this point`
    `// Each thread has its corresponding U, V, and` ψ `values`
end for

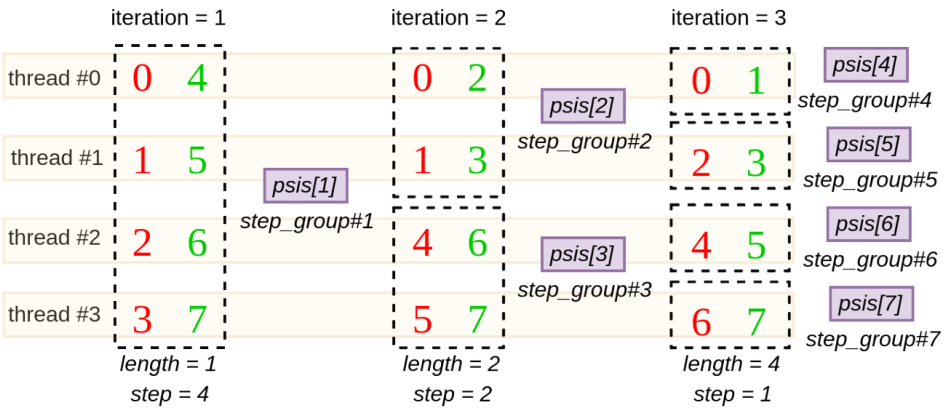


Figure 7.1: Thread Assignment for $n = 8$ in GPU Scheduling

In our kernel implementation, we follow Algorithm 3 to schedule the threads for the butterfly operations and chose the Merge NTT algorithm for its simplicity and efficiency. We also utilize the shared memory to store the twiddle factors, which allows for faster access during the butterfly computations.

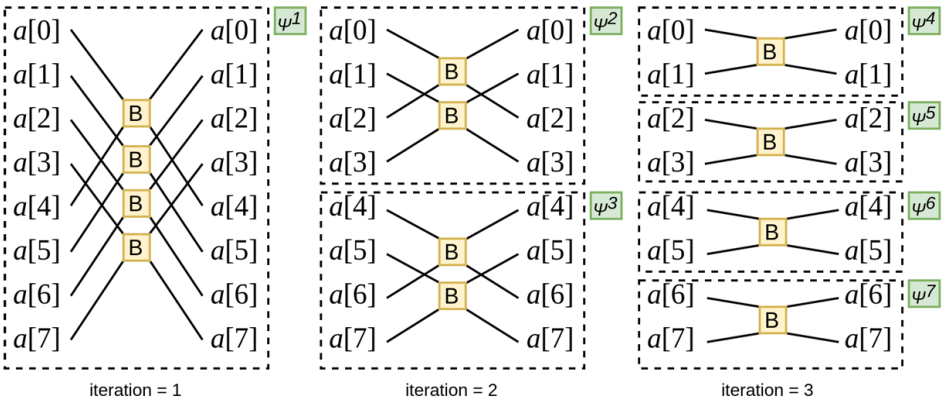


Figure 7.2: Visualization of NTT operation for n=8

7.3 ISA Extension

Let's discuss different instructions that can be used to implement the NTT accelerator in the Vortex GPGPU.

7.3.1 NTT Instruction

At first glance, implementing a single *NTT* instruction that performs the full butterfly computation across threads may seem like an efficient approach. However, this idea quickly breaks down when considering the parallel architecture and execution model of GPGPUs.

Threads in a GPGPU are inherently independent and do not have direct communication pathways between them. In the context of the Number Theoretic Transform (NTT), each butterfly operation often involves a data exchange between two threads (e.g., thread i needs data from thread j and vice versa). This type of cross-thread dependency is fundamentally incompatible with the GPGPU execution model, where threads operate in lockstep but maintain their own register files and execution states.

Because of this, any attempt to implement a full NTT butterfly as a single instruction would require **inter-thread communication**, which is not supported at the instruction level. The only way to facilitate this kind of data exchange is to first **write back to the register file**, and then use higher-level mechanisms such as:

- **Shuffle instructions** (e.g., `SHFL` in NVIDIA's PTX), if supported by the ISA. These allow limited data exchange between threads in the same warp.
- **Shared memory**, where each thread writes its data after computing, and then synchronizes to allow other threads to read the needed values.

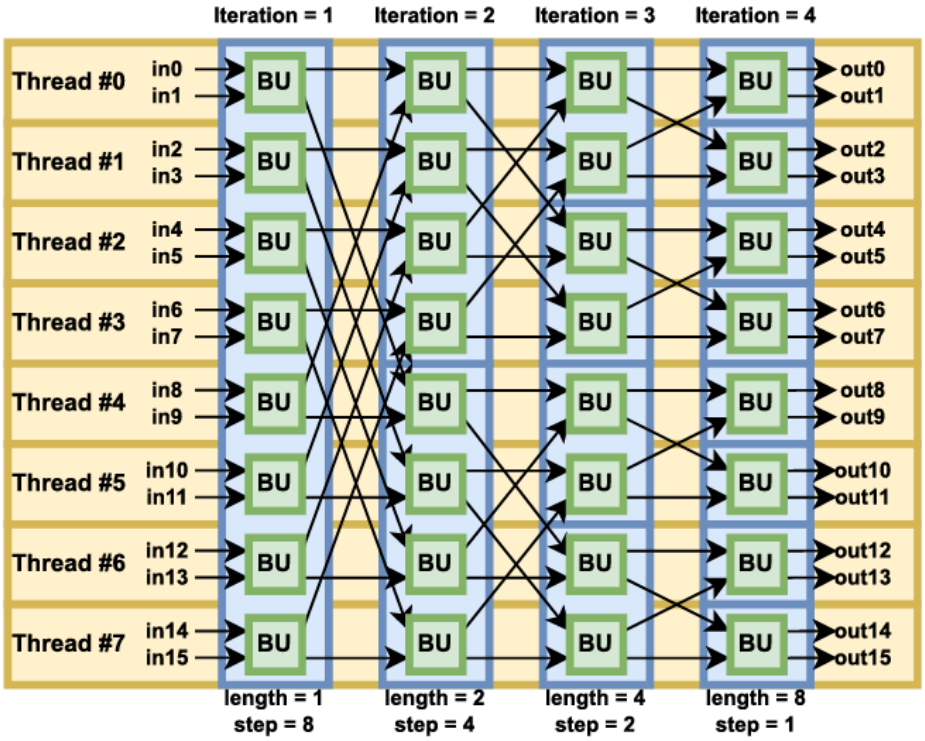


Figure 7.3: NTT Instruction

7.3.2 Butterfly Instruction

Although implementing a dedicated *butterfly instruction* may seem like a natural fit for accelerating the Number Theoretic Transform (NTT), it introduces several architectural challenges that undermine its practicality in a GPGPU setting.

First, the butterfly operation inherently produces two outputs from two inputs, yet the register file interface typically supports a single write-back port per instruction. If we attempt to concatenate the two outputs into a single wide register, we effectively lose half the precision, since only a subset of the result can be used in downstream computations without additional unpacking logic.

Second, butterfly operations require simultaneous access to two registers—one for each input (and potentially for both outputs). This leads to **bank conflicts in the register file**, especially in multi-threaded environments where many threads perform butterfly operations in lockstep. The inability to service multiple read or write ports efficiently becomes a significant bottleneck.

Moreover, the twiddle factor ψ^k used in the butterfly is not constant across threads, as shown in figure 7.2, meaning each thread must store its own ψ^k value in registers or load it dynamically. This adds to *register pressure*, which can negatively impact occupancy and overall performance.

In general, a butterfly instruction is only viable under two very specific conditions:

- **No register file write-back**: If the instruction only performs the computation and

forwards the result internally (e.g., into a temporary buffer or scratchpad memory), it avoids the dual-write constraint.

- **Dual-port access to the register file**: If each thread can issue two concurrent register read/write requests to separate banks, then register-level dataflow bottlenecks and bank conflicts could be avoided.

7.3.3 Proposed Instructions

Table 7.1: Proposed ISA Extensions for NTT Operations

Instruction	Operands	Operation Description
ADD_MOD	rd, rs1, rs2	rd = (rs1 + rs2) mod q
SUB_MOD	rd, rs1, rs2	rd = (rs1 − rs2) mod q
BARRETT	rd, rs1, rs2	rd = (rs1 × rs2) mod q
BIT_REV	rd, rs1, rs2	rd = BitReverse(rs1), rs2 = $\log_2(N)$

Note that:

- In all cases, q is the modulus and is hardcoded into the hardware design. This is justified by the fact that q remains constant throughout the entire NTT computation and is typically fixed in cryptographic applications.
- For bit reversal, the instruction takes two operands: `rs1` (the input value) and `rs2` (the number of bits to reverse). The output is stored in `rd`.

7.4 Hardware Implementation

7.4.1 Crypto Unit

We designed a dedicated *Crypto Unit* responsible for executing cryptographic operations within the GPU pipeline. While the current implementation only integrates the Number Theoretic Transform (NTT) unit, the architectural design is general enough to support future extensions and protocols. Our work motivates the integration of additional cryptographic units such as modular exponentiation, elliptic curve arithmetic, or zero-knowledge proof primitives.

The Crypto Unit is structured around two main components:

- **Dispatch Logic:** This component ensures that each crypto unit in the array receives its own input data correctly and independently. It parses and routes operands based on thread identifiers and instruction type.
- **Gather Logic:** After the cryptographic operation is completed, the gather logic collects and forwards the result back to the appropriate thread or pipeline stage. This ensures that data consistency is maintained across the thread group.

This modular structure not only facilitates clean integration of the NTT unit but also enables the Crypto Unit to scale and accommodate a variety of cryptographic functions in future designs.

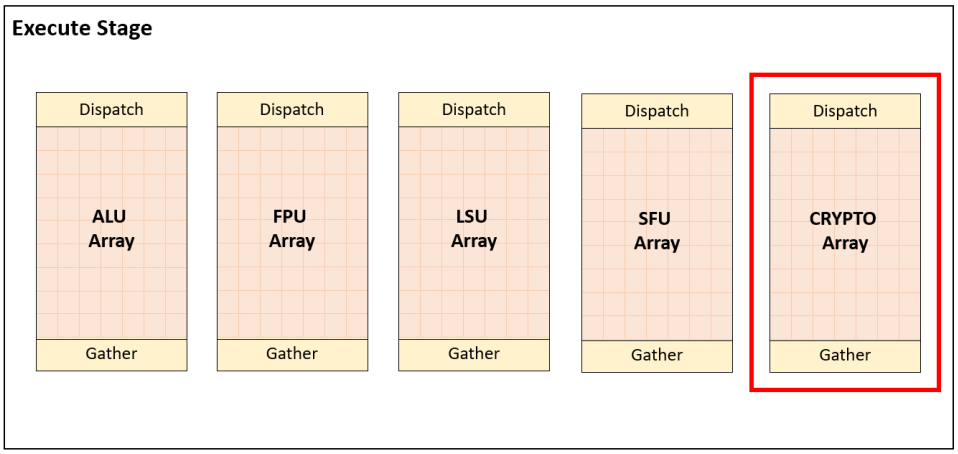


Figure 7.4: Crypto Unit Architecture

7.4.2 NTT Unit

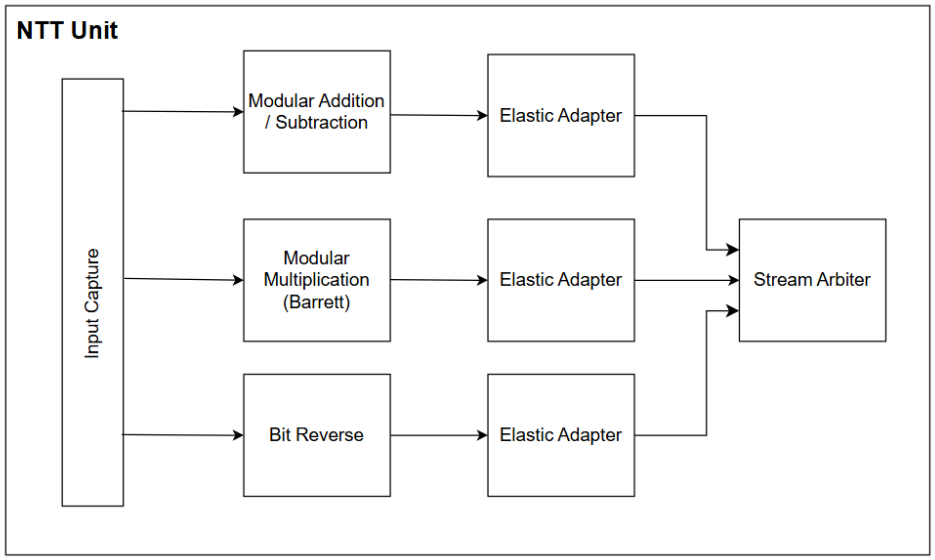


Figure 7.5: NTT Unit Architecture

Input Capture

In a pipelined and parallel crypto execution unit, it is crucial to ensure that the input operands remain stable throughout the execution of an operation. The execute stage provides input data, which changes over time as new instructions are issued. To address this, we capture and store the input operands (e.g., `rs1_data` and `rs2_data`) into internal registers at the beginning of each operation, specifically when both `valid` and `ready` signals are asserted. Once latched, the crypto units use these stable copies throughout

the execution phase. If the modular operations were not pipelined, the input capture would be unnecessary.

Elastic Adapter

The elastic adapter module implements a simple elastic buffer to decouple the upstream and downstream handshaking interfaces. It introduces a single-entry buffer with internal state tracking to manage the flow of data between modules using valid/ready signaling. The module allows data to be accepted from the upstream component when it is not currently holding any data (i.e., not loaded), and it asserts `valid_out` to the downstream only when data is available and the downstream component is not marked as `busy`. The `strobe` signal is asserted when new data is successfully accepted. This module is particularly useful in pipelined designs where decoupling the handshakes helps mitigate stalls and improves overall throughput by enabling elastic flow control.

Stream Arbiter

The stream arbiter manages data flow between three parallel computational units (modular addition/subtraction, modular multiplication, and bit-reversal) and the single output commit interface. Its usage is motivated by several architectural requirements:

- **Operation Multiplexing**: The NTT module contains three distinct arithmetic units that cannot be active simultaneously. The arbiter:
  - Selects between the modular adder/subtractor, modular multiplier, and bit-reversal unit outputs
  - Ensures only one result stream progresses to commit stage per cycle
  - Preserves operation ordering through proper arbitration
- **Metadata Preservation**: The arbiter correctly routes:
  - Instruction metadata (UUID, warp ID, thread mask, PC)
  - Result destination (register file index)
  - Pipeline control signals (writeback enable, start/end of packet)
- **Backpressure Propagation**: Implements correct flow control by:
  - Stalling upstream units when downstream cannot accept data
  - Maintaining independent ready signals for each arithmetic unit
  - Preventing result loss during pipeline stalls

7.5 Conclusion

This section summarizes the impact of parallelism and ISA customization on accelerating the Number Theoretic Transform (NTT) using a GPU-based architecture. All results are obtained under a single-core configuration to isolate architectural effects. Table 7.2 presents the raw cycle counts for three configurations: a single-lane uncustomized GPU

(baseline), a 32-lane uncustomized GPU (parallelism), and a 32-lane customized GPU (parallelism + ISA support).

Table 7.2: Cycle Count Comparison for Different GPU Configurations (Single Core)

# Points	Single-lane Uncustomized	32-lane Uncustomized	32-lane Customized
64	45,953	14,895	13,366
128	82,143	21,783	20,132
256	171,922	35,107	32,790
1024	421,472	116,842	111,065
2048	574,093	229,459	224,352

Speedup Analysis

To quantify the benefits of parallelism and ISA customization, the following two tables present speedup metrics:

Table 7.3: Speedup of 32-lane Customized GPU vs. Single-lane Uncustomized GPU

# Points	Speedup (×)
64	3.44
128	4.08
256	5.24
1024	3.80
2048	2.56

Table 7.4: Speedup of Customized RISC-V GPU (32) vs. RISC-V CPU (1)

# Points	Speedup (×)
64	1.11
128	1.08
256	1.07
1024	1.05
2048	1.02

Discussion

- **Effect of Parallelism:** Comparing the single-lane to the 32-lane GPU highlights the importance of parallel execution. Speedups exceed 5× in some cases, emphasizing that SIMD-style designs are well-suited for the NTT algorithm, which exhibits high data-level parallelism.
- **Effect of Custom Instructions:** The additional performance gain from ISA customization is moderate, maxing at 1.11× speedup. This is mainly due to an architectural bottleneck: after executing each custom instruction, the result is written to the register file, stored in global memory, and then reloaded into the register file for the next iteration. This pattern introduces significant memory latency, limiting the benefits of instruction-level acceleration.
- **GPU vs CPU Comparison Caveat:** Although the single-lane GPU configuration is used as a proxy for CPU-like behavior, it does not provide an exact match.

Differences in clock speed, pipeline depth, and overall architecture make direct comparisons imprecise. Furthermore, GPU programs incur latency overhead due to data transfer from CPU host memory to GPU device memory—something absent in standard CPU execution. These factors must be considered when interpreting performance metrics.

- **Scalability with Multiple Cores:** All measurements were taken using a single-core configuration to isolate per-core performance. In practice, multi-core execution would allow different threads or warps to independently process partitions of the input, enabling better overlap of computation and memory access. This would further amortize memory latency and lead to greater performance improvements, especially for larger input sizes.

In conclusion, the 32-lane customized GPU offers measurable benefits over uncustomized architectures, particularly when combined with parallelism. However, addressing the memory access bottleneck is essential to fully realize the potential of custom instructions in throughput-optimized GPU designs.

Part III

Evaluation

Chapter 8

FPGA Prototyping

8.1 System Architecture

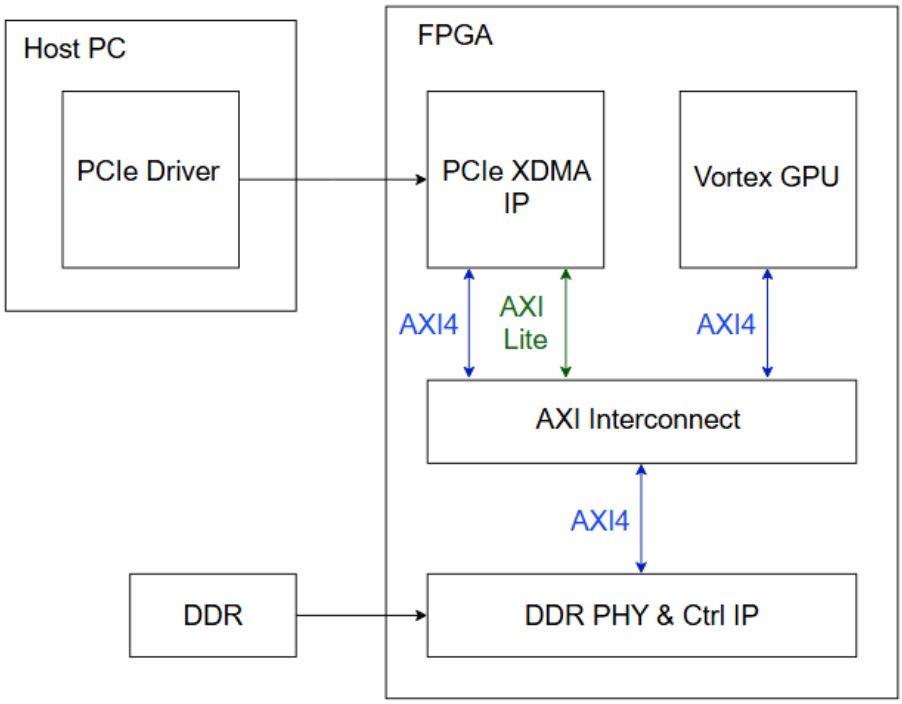


Figure 8.1: System Architecture

8.1.1 System Components Overview

The architecture in Figure 8.1 comprises the following key elements:

- **Host PC**: The primary computing system running the PCIe driver software to manage communication with the FPGA board.
- **PCIe XDMA IP Core**: Xilinx's Direct Memory Access IP that handles:
  - High-bandwidth host-to-FPGA data transfers

  - Protocol conversion between PCIe and AXI buses
  - DMA engine for efficient bulk transfers

- **AXI Interconnect Fabric**:
  - AXI4 (64/128/256-bit) for high-performance data paths
  - AXI4-Lite (32-bit) for control/status registers
  - Provides address decoding and routing between masters and slaves

- **DDR Memory Subsystem**:
  - DDR PHY handles physical layer signaling
  - Memory controller manages timing, refresh, and access scheduling
  - Provides shared memory space for host and FPGA processing elements

8.1.2 Data Transfer Mechanism

Primary Data Pathways

1. **Host-to-FPGA Transfer**:
   - Host writes to XDMA through PCIe
   - XDMA converts to AXI4 bursts
   - Data routed to DDR or processing logic
2. **FPGA-to-Host Transfer**:
   - Processing elements write results to DDR
   - XDMA reads via AXI4 and converts to PCIe
   - Data transferred to host memory space

Control Plane Operation

- **Configuration**:
  - Host configures XDMA parameters (buffer sizes, interrupts)
  - Processing element registers mapped via AXI4-Lite
- **Status Monitoring**:
  - XDMA provides transfer status via PCIe
  - Processing elements report status via AXI4-Lite
- **Synchronization**:
  - Doorbell registers for operation triggering
  - Interrupts for event notification

8.2 Vortex IP

To integrate the Vortex GPGPU into a System-on-Chip (SoC), we need to implement Vortex as an Intellectual Property (IP) block. This IP includes the NTT and integrates Device Control Registers (DCRs) in `vx_dcr_data.sv`, which are assigned to constant values:

- `Startup_address0` = 0x7000
- `Startup_args0` = 0x120000

```
/*
    always @(posedge clk) begin
        if (dcr_bus_if.write_valid) begin
            case (dcr_bus_if.write_addr)
            `VX_DCR_BASE_STARTUP_ADDR0 : dcrs.startup_addr[31:0] <= dcr_bus_if.write_data;
        `ifdef XLEN_64
            `VX_DCR_BASE_STARTUP_ADDR1 : dcrs.startup_addr[63:32] <= dcr_bus_if.write_data;
        `endif
            `VX_DCR_BASE_STARTUP_ARG0 : dcrs.startup_arg[31:0] <= dcr_bus_if.write_data;
        `ifdef XLEN_64
            `VX_DCR_BASE_STARTUP_ARG1 : dcrs.startup_arg[63:32] <= dcr_bus_if.write_data;
        `endif
            `VX_DCR_BASE_MPM_CLASS : dcrs.mpm_class <= dcr_bus_if.write_data[7:0];
            default:;
            endcase
        end
    end
    */
always @(posedge clk  ) begin
            dcrs.startup_addr[31:0] <= 32'h7000 ;
        `ifdef XLEN_64
            dcrs.startup_addr[63:32] <= 32'h0;
        `endif
            dcrs.startup_arg[31:0] <=32'h12000;
        `ifdef XLEN_64
            dcrs.startup_arg[63:32] <=32'h0;
        `endif
             dcrs.mpm_class <= 32'h0;

end
    assign base_dcrs = dcrs;
```

Figure 8.2: DCRs hard coded

To simplify system integration and operation, the DCR interface inputs—namely `dcr_wr`, `dcr_wr_address`, and `dcr_wr_data`—are set to constant values. This effectively disables dynamic DCR configuration during runtime. Consequently, the Vortex GPGPU requires only a reset signal to initiate its operation, streamlining the overall system control and configuration process.
The `VX_CONFIG.vh` file controls Vortex behavior and configurations, such as the number of cores and clusters, and allows enabling or disabling specific modules. Added lines for synthesis include:

- `define FPU_FPNEW`
- `define SYNTHESIS`
- `define DPI_DISABLE`

These settings disable DPI and ensure all Vortex modules use RTL files.

- `define XLEN_64`

- `define XLEN 64`
- `define UM_CLUSTERS 1`
- `define NUM_CORES 1`
- `define NUM_WARPS 1`
- `define L1_DISABLE`

Define the XLEN (32 or 64 bits) that define the and cores would be used

8.2.1 Synthesis Script

TCL Script

Creating a TCL script in the correct order is essential for managing large, complex designs like Vortex. The process includes:

1. Including all directories of header files (`.h`, `.svh`)
2. Using `read_verilog` for all header `.svh` files
3. Using `read_verilog` for `.sv` package files
4. Reading RTL files
5. Setting the Top module

Synthesis Issues

The first issue arises because not all SystemVerilog syntax is synthesizable. A common problem is the use of string parameters like `unsynthesizable`. A script is required to convert these string parameters in each RTL file as follows:

```
parameter string INSTANCE_ID = "" => parameter INSTANCE_ID = ""
```

The second issue involves Vivado, which can create an IP block for a SystemVerilog file as the top module. This is addressed by wrapping Vortex with a `vortex_xl_wrapper.v` file, converting all unpacked arrays to packed arrays.

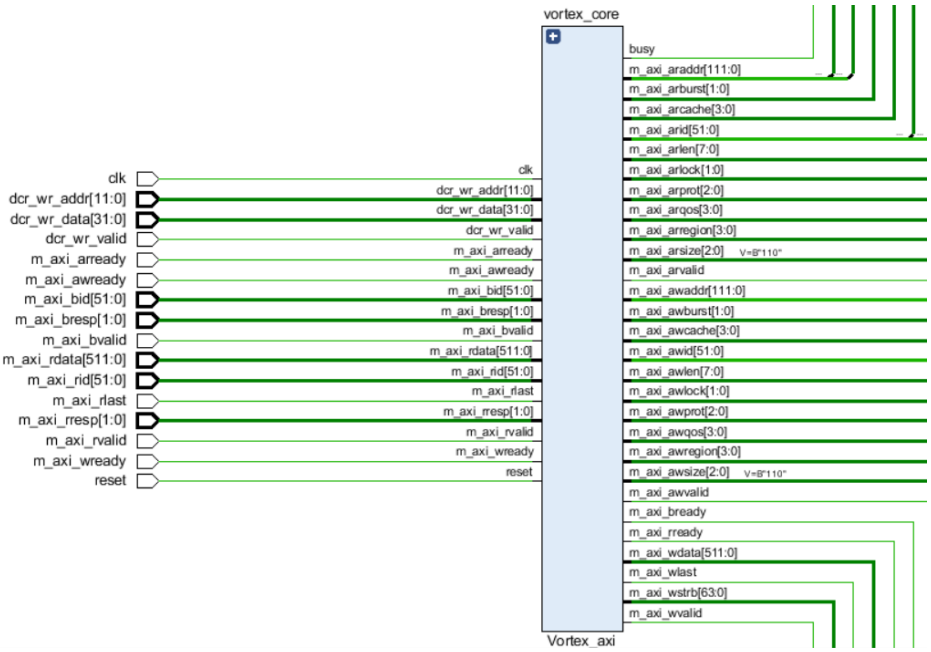


Figure 8.3: Vortex synthesized

8.3 System-on-Chip (SoC)

A System-on-Chip (SoC) is an integrated circuit (IC) that consolidates all the essential components of a computing system onto a single chip. These components typically include one or more processor cores, memory blocks, input/output interfaces, communication protocols, and often specialized accelerators such as GPUs or DSPs. The goal of an SoC is to achieve high performance and functionality while minimizing physical footprint, power consumption, and system complexity. In the context of Field Programmable Gate Arrays

(FPGAs), SoC designs offer the flexibility of integrating custom logic with embedded processing capabilities. This allows for rapid prototyping and hardware/software co-design. For our graduation project, we implement a custom SoC architecture on an

FPGA that integrates Xilinx PCIe XDMA with DDR and `VORTEX_GPGPU` with NTT unit integrated into it, utilizing the AXI4 interface.

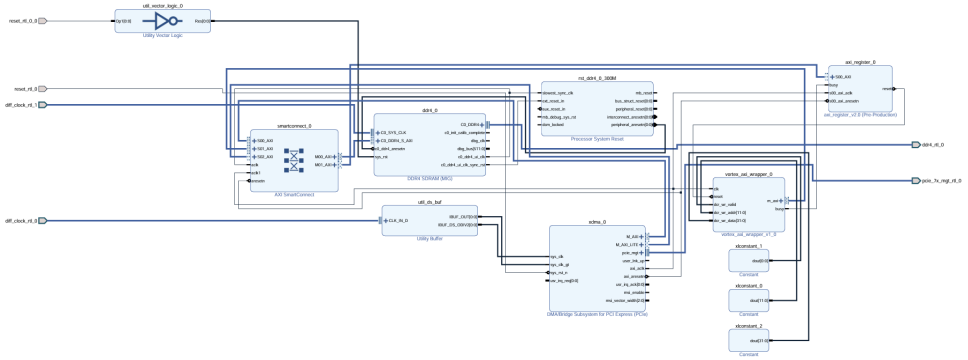


Figure 8.4: Vortex synthesized

8.4 XDMA IP

8.4.1 Overview

PCI Express (PCIe) is a high-speed serial interface standard used to connect peripheral components to a host processor. It provides high bandwidth, low latency, and scalable connectivity, making it ideal for data-intensive applications.

To simplify the process of transferring data between the host system and an FPGA, Xilinx provides the XDMA IP core—a high-performance PCIe Direct Memory Access (DMA) engine designed specifically for Xilinx FPGAs.

The Xilinx PCIe XDMA IP core acts as a bridge between the PCIe interface and user logic inside the FPGA. It supports direct memory transfers between host memory and the FPGA without requiring intervention from the host CPU. This offloading greatly enhances system performance, especially in data-heavy applications such as GPGPU computing and real-time processing.

XDMA IP have several configurations that gives designers the ability and flexibility to control several designs.

8.4.2 XDMA Ports

XDMA IP have several configurations that gives designers the ability and flexibility to control several designs.

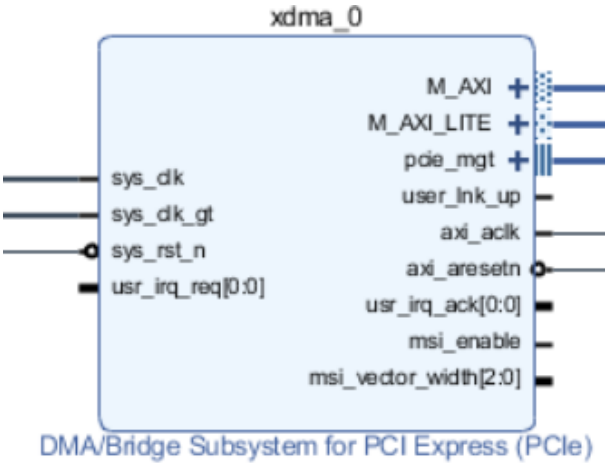


Figure 8.5

The following table describes the port direction and functionality:

Signal name	Direction	Description
sys_clk	I	PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE2. Connected to 125 MHz.
sys_clk_gt	I	PCIe reference clock. Should be driven from the O port of reference clock IBUFDS_GTE3. Also connected to 125 MHZ.
sys_rst_n	I	Reset from the PCIe edge connector reset signal.
axi_aclk	I	PCIe derived clock output for m_axi* and s_axi* interfaces. axi_aclk is a derived clock from the TXOUTCLK pin from the GT block; it is not expected to run continuously while axi_aresetn is asserted. Connected to Vortex IP and AXI_LITE Register IP.
axi_aresetn	O	Connected to 125 MHz.
pcie_mgt	O	MGT instance used by the PCIe IP core for physical layer I/O.

MGT PCIe Transceiver Signals

Signal name	Direction	Description
pci_exp_rxp[1:0]	I	PCIe RX serial interface.
pci_exp_rxn[1:0]	I	PCIe RX serial interface.
pci_exp_txp[1:0]	I	PCIe TX serial interface.
pci_exp_txn[1:0]	I	PCIe TX serial interface .

8.4.3 XDMA Configurations

The DMA/Bridge Subsystem for PCI Express® enables high-throughput data movement between host memory and the device-side DMA subsystem. It utilizes descriptors that specify the source, destination, and the size of the transfer. These direct memory transfers support both host-to-card (H2C) and card-to-host (C2H) directions [**?**].

BAR0

When a PCIe device (like your XDMA IP core inside the FPGA) is connected to a host system (like a Linux PC), the host operating system assigns memory regions to communicate with that device. These memory regions are called **Base Address Registers (BARs)**.

BAR0 is the first Base Address Register exposed by the PCIe device. It typically points to a **memory-mapped I/O (MMIO)** region that the host software can access. This Memory-mapped is **AXI lite registers** typically 4 registers.

The **host memory base address of PCIe:BAR0** is dynamically assigned by the PCIe Root Complex (the CPU) during the system boot or device enumeration process.so The physical memory address where the host OS can access BAR0 is not fixed, and Your software or driver must query the system to discover this address at runtime.

Those registers provide the control of **VORTEX input/ output ports**, handling the enable signal which is the **reset** port and observing the **busy** signal which indicates the start of operation when high, end of operation when fall to 0 again. BAR0 connected to

the AXI_LITE registers via smart connect:

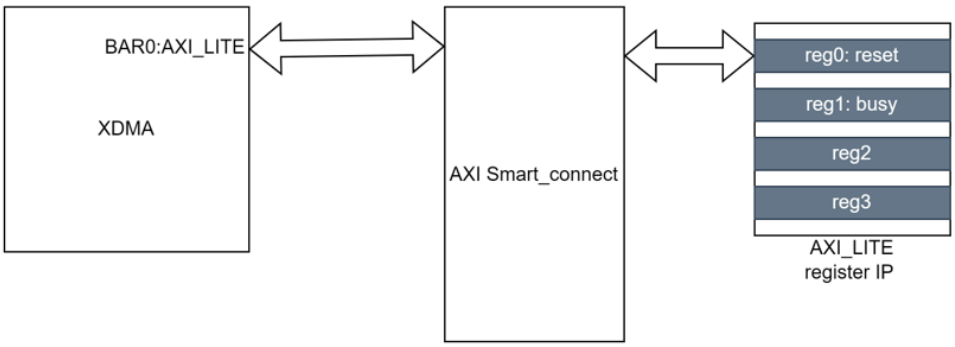


Figure 8.6: DMA Subsystem for PCI Express (PCIe)

The base address of the `AXI_LITE` register block is `0x0000000044A00000`. Similarly, the

host may need to read from or write to the FPGA's memory or internal buffers. In both directions, the PCIe Root Complex and the XDMA engine handle these operations using either Memory-Mapped I/O (MMIO) or DMA descriptor tables. To access a specific location inside `BAR0` from the host software, the physical address is computed as:

$$\text{Physical Address} = \langle \text{PCIe:BAR0 Host Base Address} \rangle + \langle \text{Offset} \rangle$$

Example: Assume `BAR0` is mapped to `0xC0400000` in the host address space, and you want to access a register located at offset `0x100`. Then the physical address is:

$$\texttt{0xC0400000} + \texttt{0x100} = \texttt{0xC0400100}$$

H2C and C2H Channels

A Host-to-Card (H2C) channel generates read requests to PCIe and provides the data or generates a write request to the user application.

A Card-to-Host (C2H) channel either waits for data on the user side or generates a

read request on the user side and then generates a write request containing the data received to PCIe . When multiple channels for H2C and C2H are enabled, transactions

on the AXI4 Master interface are interleaved between all selected channels. Simple round robin protocol is used to service all channels. Transactions granularity depends on host

Max Payload Size (MPS), page size, and other host settings. In the PCIe XDMA block

configuration, the design enables two Host-to-Card (H2C) channels and two Card-to-Host (C2H) channels. This selection strikes a balance between system performance and logic

resource utilization, particularly with respect to Look-Up Tables (LUTs) and routing complexity. By limiting the number of active DMA channels to two in each direction, the design supports concurrent data streams while avoiding the overhead of unused channels, thereby optimizing the FPGA's area and improving timing closure margins.

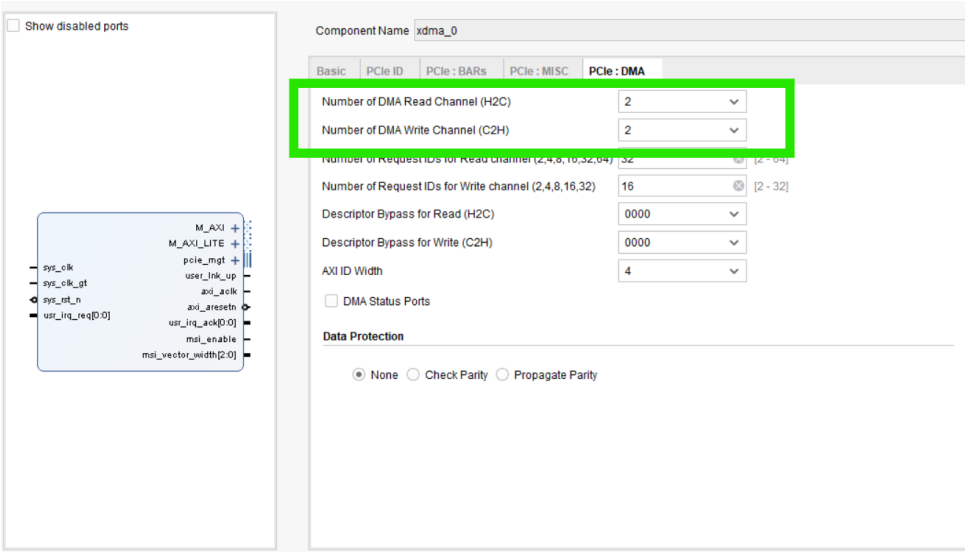


Figure 8.7: Number of Head Channel (H2C)

8.4.4 Advanced Configurations

Lane Width

Refers to the number of independent data lanes used for communication between the host and the device. Common options are x1, x2, x4, x8, x16, where each "x" represents one pair of differential TX and RX lines. Higher lane widths = more bandwidth. In this

design, a `x2` lane width was selected for the PCIe interface to balance performance and FPGA resource utilization, particularly LUT usage. Compared to wider configurations

(like `x4` or `x8`), using 2 lanes reduces the number of active transceivers and supports logic, leading to a more area-efficient implementation while still offering sufficient PCIe bandwidth for the target application.

Link speed

Link speed selected to be 5GT/s The PCIe interface is configured with a `x2` lane width

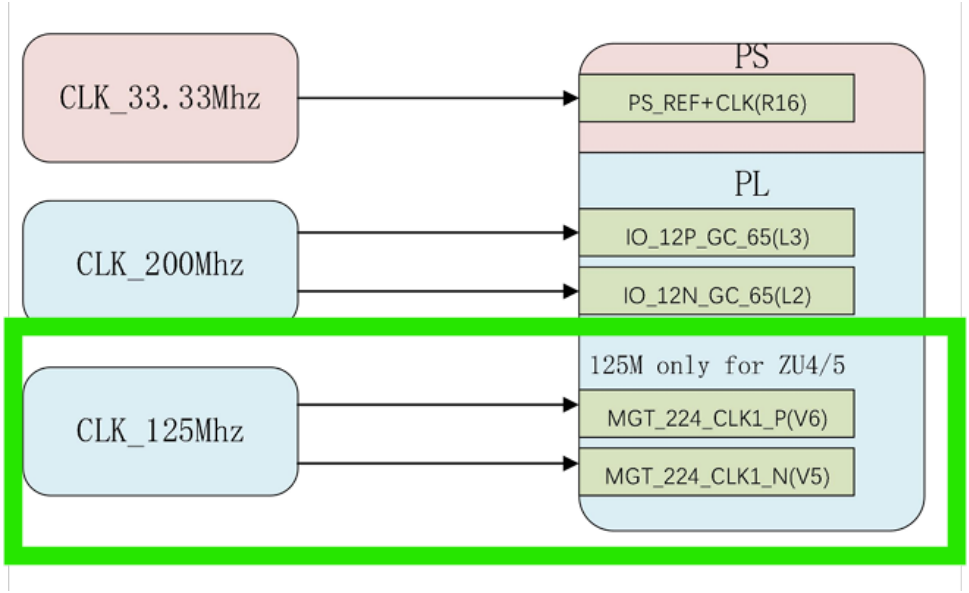
and a maximum link speed of 5.0 GT/s (PCIe Gen2). This configuration offers a theoretical raw bandwidth of 10 GT/s (5 GT/s per lane x 2 lanes), which translates to

approximately 8 Gbps effective throughput after accounting for 8b/10b encoding overhead. This setup provides a balanced trade-off between performance and logic resource

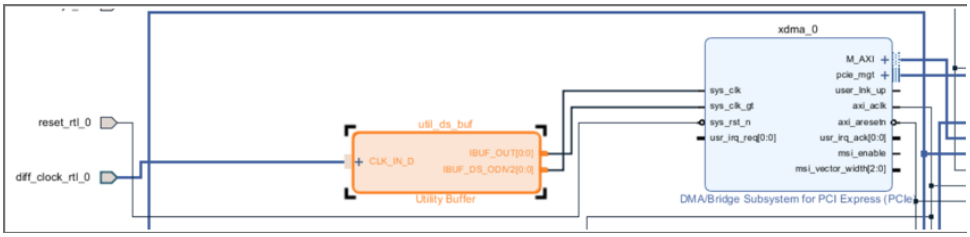
usage. By limiting the lane width to `x2`, the design reduces the number of transceiver resources and associated logic, such as PCS/PMA blocks and protocol handling circuits, thereby optimizing LUT and routing utilization on the FPGA while maintaining sufficient data transfer capability for the system's needs.

Reference clock frequency

The PCIe reference clock frequency is a critical signal used by the FPGA's transceivers (MGTs) to establish and maintain the correct data rate over the PCIe link. vspace1em



125M clock provides the differential clock for the GTX interface. As shown in the figure below.



1. **Differential Clock Input (diff_clock_rtl_0) – 125 MHz**
   - This signal is a differential pair input, coming from an onboard oscillator.
   - It is connected to a Utility Buffer block (`util_ds_buf`) which converts the differential signal into a single-ended clock using internal primitives.
2. **Utility Buffer (`util_ds_buf`)**
   - **Input:** `CLK_IN_D` is the differential clock input.
   - **Outputs:**

  - `IBUF_OUT[0:0]`: The single-ended system clock (`sys_clk`) derived from the differential input.
  - `IBUF_DS_ODIV2[0:0]`: A divided version of the clock (`sys_clk_gt`), used for clocking the transceiver or high-speed logic if needed.
- These outputs feed directly into the XDMA IP block.

8.4.5 Constriants

```
# Reset signal (Using user reset)
set_property PACKAGE_PIN AE12 [get_ports reset_rtl_0]
set_property IOSTANDARD LVCMOS33 [get_ports reset_rtl_0]

set_property PULLUP true [get_ports reset_rtl_0]

## pins

## PCIe

set_property PACKAGE_PIN Y5 [get_ports {diff_clock_rtl_0_clk_n[0]}]
set_property PACKAGE_PIN Y6 [get_ports {diff_clock_rtl_0_clk_p[0]}]

## TXN & TXP
# R0 P
set_property PACKAGE_PIN U4 [get_ports {pcie_7x_mgt_rtl_0_rxp[0]}]
# R0 N
set_property PACKAGE_PIN U3 [get_ports {pcie_7x_mgt_rtl_0_rxn[0]}]
# R1 P
set_property PACKAGE_PIN W4 [get_ports {pcie_7x_mgt_rtl_0_rxp[1]}]
# R1 N
set_property PACKAGE_PIN W3 [get_ports {pcie_7x_mgt_rtl_0_rxn[1]}]
# T0 P
set_property PACKAGE_PIN V2 [get_ports {pcie_7x_mgt_rtl_0_txp[0]}]
# T0 N
set_property PACKAGE_PIN V1 [get_ports {pcie_7x_mgt_rtl_0_txn[0]}]
# T1 P
set_property PACKAGE_PIN Y2 [get_ports {pcie_7x_mgt_rtl_0_txp[1]}]
# T1 N
set_property PACKAGE_PIN Y1 [get_ports {pcie_7x_mgt_rtl_0_txn[1]}]
########################################################################
```

Configuration Management Interface

This option is a check box in PCIE:MISC. In this design, the Configuration Management Interface was intentionally disabled to reduce complexity and save FPGA resources. Since the XDMA core already provides access to required PCIe configuration and status registers through its builtin logic, and the system does not require custom or dynamic

manipulation of PCIe configuration space, enabling this interface would introduce unnecessary logic overhead. Disabling the Configuration Management Interface results in lower LUT and register usage, contributes to simpler integration, and eliminates unused logic paths, which is beneficial for timing closure and overall resource efficiency.

8.5 Xilinx DDR4 (MIG)

DDR (Double Data Rate) memory is a widely used memory technology due to its high data transfer rates and low power consumption. DDR memory often serves as a buffer or storage for kernel offload from CPU (host), crypto unit computed results, and parameters for the crypto unit. The Memory Interface Generator (MIG) is a configurable IP core

provided by Xilinx to facilitate the integration of DDR4 SDRAM into FPGA designs. MIG abstracts the complexity of DDR4 signaling, timing, and protocol management, providing a fully functional and verified memory controller that connects the FPGA fabric to external DDR4 memory devices.

8.5.1 MIG IP ports

Port	Direction	Description
C0_SYS_CLK	I	Input differential clock, connected to 200 MHz clock on PL side IO_12P_GC_65 (L3)/IO_12N_GC_65 (L2).
C0_DDR4_S_AXI	I/O	Slave AXI interface for memory.
sys_rst	I	This is the external system-level reset. It's typically active-low, and it resets the MIG controller logic, clocking logic, and internal calibration engine.
c0_ddr4_aresetn	I	AXI interface reset for controller 0 of the MIG. It resets only the user-facing AXI interface logic, not the full MIG controller or PHY. Must be synchronized to the UI clock (c0_ddr4_ui_clk) and often comes from a processor system reset block.
C0_DDR4	–	Physical Interface signals for DDR4.
C0_DDR4_UI_CLK	O	User Interface Clock output generated by the DDR4 MIG IP for controller 0.
C0_DDR4_UI_CLK_SYNC_RST	O	Active-high, synchronous reset signal generated by the DDR4 MIG IP for Controller 0. Used to reset logic operating in the UI clock domain under c0_ddr4_ui_clk.

8.5.2 MIG IP Configuration

8.5.3 Reference clock

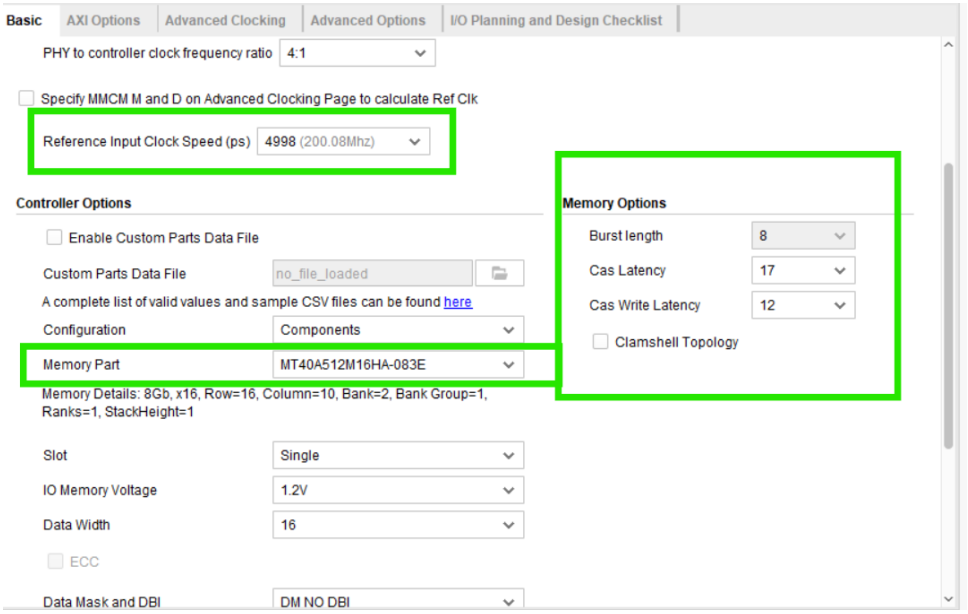
The reference input clock is selected to be 200 MHz to ensure compatibility with the clock source available on the PL side, specifically at IO_12P_GC_65 (L3)/IO_12N_GC_65 (L2).

8.5.4 DDR4 Component selection

The DDR4 memory model used for the xczu4ev-sfvc784-2-i FPGA is MT40A512M16LY-062E IT:E.

MIG Field	Value for MT40A512M16LY-062E IT:E
Memory Type	DDR4 SDRAM
Data Width	16
Number of Ranks	1
Row Address	16
Column Address	10
Bank Group Address	2
Bank Address	2
DRAM Bus Width	16
Memory Voltage	1.2 V

The closest memory component is MT40A512M16HA-083E



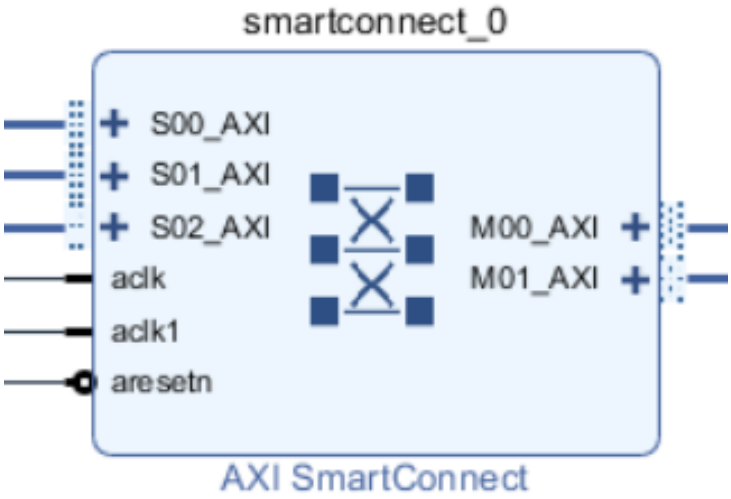
8.5.5 constraints

```
# DDR4 clock pins
set_property IOSTANDARD DIFF_SSTL12 [get_ports diff_clock_rtl_1_clk_p]
set_property PACKAGE_PIN L3 [get_ports diff_clock_rtl_1_clk_p]
set_property PACKAGE_PIN L2 [get_ports diff_clock_rtl_1_clk_n]
set_property IOSTANDARD DIFF_SSTL12 [get_ports diff_clock_rtl_1_clk_n]

# DDR reset
set_property -dict {PACKAGE_PIN D2 IOSTANDARD LVCMOS18} [get_ports reset_rtl_0_0]

# pins constraints
set_property PACKAGE_PIN L6 [get_ports ddr4_rtl_0_act_n]
set_property PACKAGE_PIN H1 [get_ports {ddr4_rtl_0_adr[0]}]
```

```
set_property PACKAGE_PIN U8 [get_ports {ddr4_rtl_0_adr[1]}]
set_property PACKAGE_PIN J1 [get_ports {ddr4_rtl_0_adr[2]}]
set_property PACKAGE_PIN V8 [get_ports {ddr4_rtl_0_adr[3]}]
set_property PACKAGE_PIN K4 [get_ports {ddr4_rtl_0_adr[4]}]
set_property PACKAGE_PIN R8 [get_ports {ddr4_rtl_0_adr[5]}]
set_property PACKAGE_PIN J2 [get_ports {ddr4_rtl_0_adr[6]}]
set_property PACKAGE_PIN U9 [get_ports {ddr4_rtl_0_adr[7]}]
set_property PACKAGE_PIN K2 [get_ports {ddr4_rtl_0_adr[8]}]
set_property PACKAGE_PIN W8 [get_ports {ddr4_rtl_0_adr[9]}]
set_property PACKAGE_PIN T8 [get_ports {ddr4_rtl_0_adr[10]}]
set_property PACKAGE_PIN K1 [get_ports {ddr4_rtl_0_adr[11]}]
set_property PACKAGE_PIN T6 [get_ports {ddr4_rtl_0_adr[12]}]
set_property PACKAGE_PIN Y8 [get_ports {ddr4_rtl_0_adr[13]}]
set_property PACKAGE_PIN K3 [get_ports {ddr4_rtl_0_adr[14]}]
set_property PACKAGE_PIN R7 [get_ports {ddr4_rtl_0_adr[15]}]
set_property PACKAGE_PIN P9 [get_ports {ddr4_rtl_0_adr[16]}]
set_property PACKAGE_PIN H2 [get_ports {ddr4_rtl_0_ba[0]}]
set_property PACKAGE_PIN T7 [get_ports {ddr4_rtl_0_ba[1]}]
set_property PACKAGE_PIN V9 [get_ports {ddr4_rtl_0_bg[0]}]
set_property PACKAGE_PIN H4 [get_ports {ddr4_rtl_0_ck_t[0]}]
set_property PACKAGE_PIN H3 [get_ports {ddr4_rtl_0_ck_c[0]}]
set_property PACKAGE_PIN R6 [get_ports {ddr4_rtl_0_cs_n[0]}]
set_property PACKAGE_PIN K5 [get_ports {ddr4_rtl_0_cke[0]}]
set_property PACKAGE_PIN L7 [get_ports {ddr4_rtl_0_dm_n[1]}]
set_property PACKAGE_PIN J5 [get_ports {ddr4_rtl_0_dm_n[0]}]
set_property PACKAGE_PIN J6 [get_ports {ddr4_rtl_0_dq[0]}]
set_property PACKAGE_PIN H9 [get_ports {ddr4_rtl_0_dq[1]}]
set_property PACKAGE_PIN J7 [get_ports {ddr4_rtl_0_dq[2]}]
set_property PACKAGE_PIN K9 [get_ports {ddr4_rtl_0_dq[3]}]
set_property PACKAGE_PIN H7 [get_ports {ddr4_rtl_0_dq[4]}]
set_property PACKAGE_PIN J9 [get_ports {ddr4_rtl_0_dq[5]}]
set_property PACKAGE_PIN H6 [get_ports {ddr4_rtl_0_dq[6]}]
set_property PACKAGE_PIN H8 [get_ports {ddr4_rtl_0_dq[7]}]
set_property PACKAGE_PIN M6 [get_ports {ddr4_rtl_0_dq[8]}]
set_property PACKAGE_PIN N8 [get_ports {ddr4_rtl_0_dq[9]}]
set_property PACKAGE_PIN N6 [get_ports {ddr4_rtl_0_dq[10]}]
set_property PACKAGE_PIN N7 [get_ports {ddr4_rtl_0_dq[11]}]
set_property PACKAGE_PIN L8 [get_ports {ddr4_rtl_0_dq[12]}]
set_property PACKAGE_PIN N9 [get_ports {ddr4_rtl_0_dq[13]}]
set_property PACKAGE_PIN L5 [get_ports {ddr4_rtl_0_dq[14]}]
set_property PACKAGE_PIN M8 [get_ports {ddr4_rtl_0_dq[15]}]
set_property PACKAGE_PIN K7 [get_ports {ddr4_rtl_0_dqs_c[0]}]
set_property PACKAGE_PIN K8 [get_ports {ddr4_rtl_0_dqs_t[0]}]
set_property PACKAGE_PIN P6 [get_ports {ddr4_rtl_0_dqs_c[1]}]
set_property PACKAGE_PIN P7 [get_ports {ddr4_rtl_0_dqs_t[1]}]
set_property PACKAGE_PIN J4 [get_ports {ddr4_rtl_0_odt[0]}]
set_property PACKAGE_PIN R8 [get_ports {ddr4_rtl_0_dqs_t[2]}]
set_property PACKAGE_PIN T8 [get_ports {ddr4_rtl_0_dqs_c[2]}]
```

```
set_property PACKAGE_PIN H4 [get_ports {ddr4_rtl_0_dqs_t[3]}]
set_property PACKAGE_PIN H3 [get_ports {ddr4_rtl_0_dqs_c[3]}]
set_property PACKAGE_PIN P7 [get_ports {ddr4_rtl_0_dqs_t[4]}]
set_property PACKAGE_PIN P6 [get_ports {ddr4_rtl_0_dqs_c[4]}]
set_property PACKAGE_PIN K8 [get_ports {ddr4_rtl_0_dqs_t[5]}]
set_property PACKAGE_PIN K7 [get_ports {ddr4_rtl_0_dqs_c[5]}]
set_property PACKAGE_PIN G3 [get_ports {ddr4_rtl_0_dqs_t[6]}]
set_property PACKAGE_PIN F3 [get_ports {ddr4_rtl_0_dqs_c[6]}]
set_property PACKAGE_PIN B4 [get_ports {ddr4_rtl_0_dqs_t[7]}]
set_property PACKAGE_PIN A4 [get_ports {ddr4_rtl_0_dqs_c[7]}]
set_property PACKAGE_PIN G8 [get_ports {ddr4_rtl_0_dqs_t[8]}]
set_property PACKAGE_PIN F7 [get_ports {ddr4_rtl_0_dqs_c[8]}]
set_property PACKAGE_PIN L1 [get_ports ddr4_rtl_0_reset_n]
```

8.6 Smart Connect IP



The AXI SmartConnect IP serves as the primary interconnection fabric in the system, linking all AXI-compliant components, including the PCIe XDMA interface, DDR4 memory controller, Vortex GPGPU core, and DMA engines. With features such as automatic address decoding, protocol conversion support, and optimized routing capabilities, SmartConnect significantly simplifies SoC integration and enhances resource efficiency. It enables low-latency and high-throughput communication between processing and memory subsystems, which is essential for performance-critical applications executed on the Vortex accelerator. AXI SmartConnect plays a central role in managing the connections between AXI master and slave interfaces of various system components. Notable connections include:

- Host (XDMA) → DDR4
- Vortex → DDR4
- Host (XDMA) → `AXI_LITE` register

The SmartConnect ensures correct routing of transactions among these modules while handling protocol compliance, handshaking, clock domain crossing (if applicable), and transaction interleaving.

8.6.1 Multi clock handling

Clock Domain Crossing (CDC) Management with AXI SmartConnect

The AXI SmartConnect IP can internally manage Clock Domain Crossings (CDCs) between AXI master and slave interfaces that operate on different clock domains. It achieves this by automatically inserting the necessary clock conversion logic, including asynchronous FIFOs, skid buffers, and handshake synchronizers. These mechanisms ensure data integrity and AXI protocol compliance when transferring data across differing clock domains. By default, all interfaces connected to SmartConnect are assumed to op-

erate in the same clock domain, driven by the `aclk` input. However, the SmartConnect IP can be configured to support multiple clock domains by enabling additional clock input ports such as `aclk1`, `aclk2`, etc. The Vivado tools automatically detect the clock domain for each AXI interface by tracing the source of the clock signal driving the connected master or slave. SmartConnect then applies the appropriate clock input to each Slave Interface (SI) and Master Interface (MI) accordingly. An error will be reported during synthesis or implementation if a clock domain used by any SI or MI is not connected to one of the enabled clock inputs on SmartConnect. It is unnecessary to connect the same

clock signal to multiple SmartConnect clock inputs. Instead, it is recommended to enable only as many clock inputs as there are unique clock sources in the design. Each distinct clock source should be connected to a single enabled input port. In the implemented SoC

design, the following SmartConnect clock ports are used:

- `aclk` — 300 MHz clock for the DDR4 MIG interface.
- `aclk1` — 125 MHz clock shared between XDMA, Vortex GPGPU, and AXI_LITE register interfaces.

8.6.2 Resets

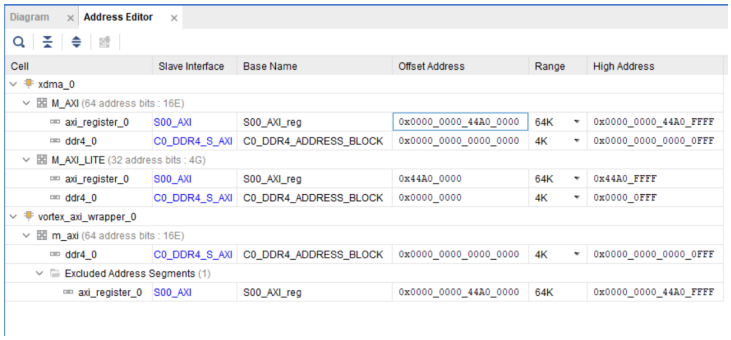
Reset Strategy for AXI SmartConnect

The AXI SmartConnect IP core includes a single active-low reset input signal, `aresetn`. This reset input is internally resynchronized to each of the individual clock domains connected to the core, ensuring consistent and reliable behavior across all timing domains. Upon power-up or when `aresetn` is asserted, the SmartConnect core deasserts all AXI

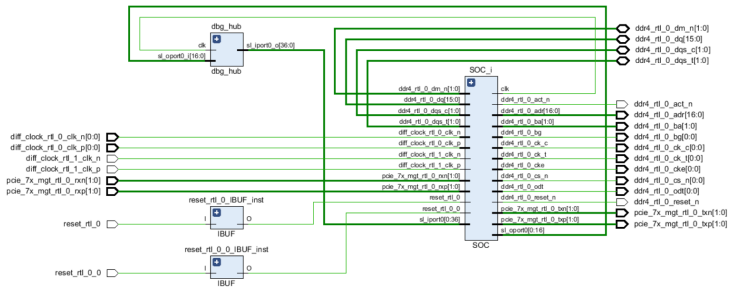
valid and ready output signals. This guarantees that no transactions occur until all parts of the system are correctly initialized. The outputs remain deasserted for the duration of the reset pulse and a short stabilization period afterward. In the implemented SoC design,

the `aresetn` input of SmartConnect is connected to the `axi_aresetn` signal provided by the XDMA interface, ensuring coordinated reset control across the PCIe, Vortex GPGPU, AXI_LITE register interface, and SmartConnect interconnect.

8.6.3 Address Mapping



8.7 SOC implementation

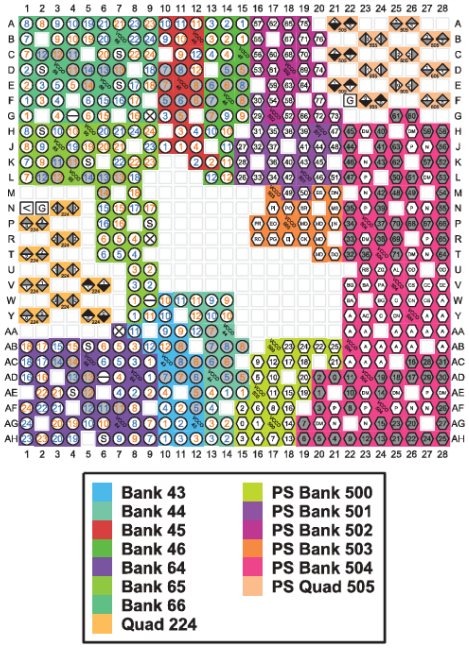


8.7.1 Bank definition

I/O Banks in Xilinx FPGAs

In Xilinx FPGAs, an **I/O bank** is a group of I/O pins that share common resources such as a power supply voltage or output current reference. This architectural grouping simplifies the FPGA's internal routing and manufacturing process, thereby reducing overall device cost and pin count. However, the use of shared resources within a bank

imposes certain constraints. Specifically, the selection of programmable I/O standards (e.g., voltage levels, drive strength, termination) is limited by the configuration of the entire bank. As a result, careful planning is required during the pin assignment phase to ensure that all I/O standards used within a bank are compatible.



8.7.2 Pins and Banks

The PCIe 3.0 x2 interface is implemented on the FPGA carrier board, with the interface signals routed to **Bank 44/43** and **Bank 224** of the FPGA device. Bank 224 is a Gigabit

Transceiver (GTX/GTH) bank, we don't assign IOSTANDD to MGT pins.

Table 8.1: PCIe Signal to FPGA Pin Mapping

RTL Port Name	Signal Description	FPGA Pin	Bank
`diff_clock_rtl_0_clk_p[0]`	PCIe Reference Clock (P)	V6	224
`diff_clock_rtl_0_clk_n[0]`	PCIe Reference Clock (N)	V5	224
`pcie_7x_mgt_rtl_0_rxp[0]`	PCIe RX Lane 0 (P)	V2	224
`pcie_7x_mgt_rtl_0_rxn[0]`	PCIe RX Lane 0 (N)	V1	224
`pcie_7x_mgt_rtl_0_txp[0]`	PCIe TX Lane 0 (P)	U4	224
`pcie_7x_mgt_rtl_0_txn[0]`	PCIe TX Lane 0 (N)	U3	224
`pcie_7x_mgt_rtl_0_rxp[1]`	PCIe RX Lane 1 (P)	Y2	224
`pcie_7x_mgt_rtl_0_rxn[1]`	PCIe RX Lane 1 (N)	Y1	224
`pcie_7x_mgt_rtl_0_txp[1]`	PCIe TX Lane 1 (P)	W4	224
`pcie_7x_mgt_rtl_0_txn[1]`	PCIe TX Lane 1 (N)	W3	224
`reset_rtl_0`	PCIe Reset (PERST#)	AE12	43

8.7.3 Port delays

Note: `set_input_delay` and `set_output_delay` constraints are not required for the external memory interface pins in this design, as the calibration process automatically runs at start-up. Any related warnings observed during implementation can be safely ignored.

8.7.4 I/o ports

A single industrial-grade DDR4 memory is integrated on the Programmable Logic (PL) side. The DDR4 interface pins on the Processing System (PS) side are dedicated and can be selected via software. In this SoC design, only the PL-side DDR4 RAM

is utilized. Pin assignments are handled through an `.xdc` constraints file, while the remaining I/O settings are automatically configured by the MIG IP and Vivado tool. Manual editing of these parameters is avoided due to the risk of misconfiguration. The

PL-side DDR4 is connected to Bank 65, which is a High-Speed I/O (HSIO) bank with full DDR4 support. This bank satisfies all electrical, timing, and routing constraints required for DDR4 operation.

Table 8.2: DDR4 RTL Signal to FPGA Pin Mapping

RTL Port Name	**Pin**	**RTL Port Name**	**Pin**
ddr4_rtl_0_adr[0]	H1	ddr4_rtl_0_adr[1]	U8
ddr4_rtl_0_adr[2]	J1	ddr4_rtl_0_adr[3]	V8
ddr4_rtl_0_adr[4]	K4	ddr4_rtl_0_adr[5]	R8
ddr4_rtl_0_adr[6]	J2	ddr4_rtl_0_adr[7]	U9
ddr4_rtl_0_adr[8]	K2	ddr4_rtl_0_adr[9]	W8
ddr4_rtl_0_adr[10]	T8	ddr4_rtl_0_adr[11]	K1
ddr4_rtl_0_adr[12]	T6	ddr4_rtl_0_adr[13]	Y8
ddr4_rtl_0_adr[14]	K3	ddr4_rtl_0_adr[15]	R7
ddr4_rtl_0_adr[16]	P9	ddr4_rtl_0_ba[0]	H2
ddr4_rtl_0_ba[1]	T7	ddr4_rtl_0_bg[0]	V9
ddr4_rtl_0_ck_t[0]	H4	ddr4_rtl_0_ck_c[0]	H3
ddr4_rtl_0_cs_n[0]	R6	ddr4_rtl_0_cke[0]	K5
ddr4_rtl_0_dm_n[1]	L7	ddr4_rtl_0_dm_n[0]	J5
ddr4_rtl_0_dq[0]	J6	ddr4_rtl_0_dq[1]	H9
ddr4_rtl_0_dq[2]	J7	ddr4_rtl_0_dq[3]	K9
ddr4_rtl_0_dq[4]	H7	ddr4_rtl_0_dq[5]	J9
ddr4_rtl_0_dq[6]	H6	ddr4_rtl_0_dq[7]	H8
ddr4_rtl_0_dq[8]	M6	ddr4_rtl_0_dq[9]	N8
ddr4_rtl_0_dq[10]	N6	ddr4_rtl_0_dq[11]	N7
ddr4_rtl_0_dq[12]	L8	ddr4_rtl_0_dq[13]	N9
ddr4_rtl_0_dq[14]	L5	ddr4_rtl_0_dq[15]	M8
ddr4_rtl_0_dqs_t[0]	K8	ddr4_rtl_0_dqs_c[0]	K7
ddr4_rtl_0_dqs_t[1]	P7	ddr4_rtl_0_dqs_c[1]	P6
ddr4_rtl_0_odt[0]	J4	ddr4_rtl_0_reset_n	—

8.7.5 I/O settings

Table 8.3: DDR4 IO Standard and Electrical Constraints

#	Constraint	Description
1	`DIFF_POD12_DCI`	**DIFF**: Differential signal, **POD12**: Pseudo Open Drain with 1.2V, required by DDR4, **DCI**: Digitally Controlled Impedance to match internal FPGA drivers with trace impedance.
2	`SLEW: FAST`	Controls the signal transition speed from logic low to high and vice versa. `FAST` is required for high-speed interfaces such as DDR4.
3	`OUTPUT_IMPEDANCE: RDRV_40_40`	Defines the driver impedance. `RDRV_40_40` corresponds to 40 Ohms for both P and N of the differential pair.
4	`IBUF_LOW_PWR: FALSE`	Disables the low-power input buffer mode. Setting this to `FALSE` enables full-strength input buffers for improved signal integrity at high speeds.

8.8 Reset Architecture

This part discusses the reset system for the SOC

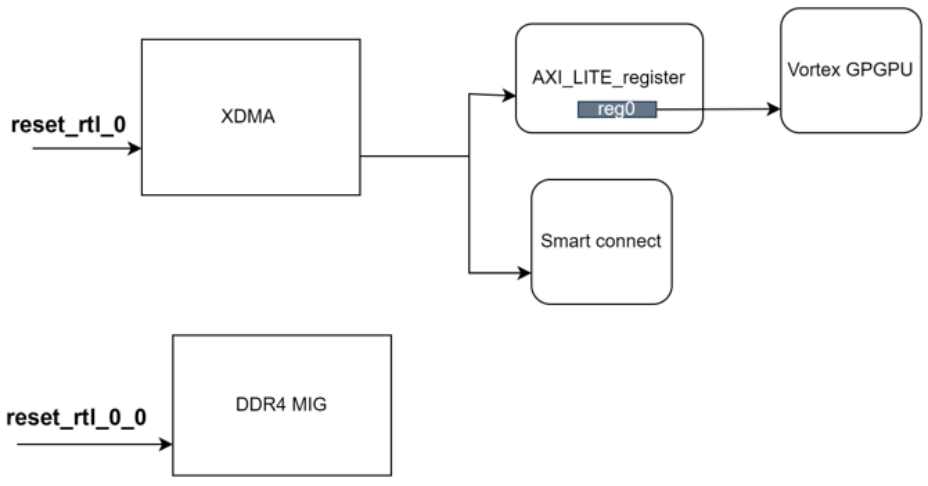


Figure 8.8: DCRs hard coded

8.8.1 PCIe reset vs Global reset

In the implemented SoC, two separate reset signals are used: one for the PCIe subsystem and another as a global reset for the rest of the system.

- **PCIe Reset:** `reset_rtl_0` is assigned to pin `AE12`. This is a dedicated reset signal provided by the host (CPU) that specifically resets the PCIe endpoint. It

must comply with the PCI Express specification and originates from the external environment.

- **Global Reset:** `reset_rtl_0_0` is assigned to pin `D2` and is connected to the DDR4 MIG controller. This reset signal is intended to initialize or restart system-level peripherals within the PL.

It is incorrect to use a single reset signal for both PCIe and the rest of the system, as the PCIe reset must follow strict timing and sequencing rules defined by the PCIe standard, which differ from those of general system peripherals.

8.8.2 Peripherals reset

Other peripheral resets are driven by the active-low reset signal `axi_aresetn`, which is an output from the XDMA PCIe IP core. This reset signal ensures that all downstream logic,

including peripheral IPs, remains in reset until the PCIe link is successfully established and stable. This sequencing guarantees proper initialization and avoids unpredictable behavior during PCIe negotiation and configuration phases.

8.9 Post Synthesis & Implementation Reports

8.9.1 Implementation on board

8.9.2 Utalization report

8.9.3 Power report

8.9.4 timing report

8.9.5 Execution Flow

The complete execution flow of the system is initiated and orchestrated by the host via PCIe and proceeds as follows:

1. **Kernel Deployment:** The host system transfers the compiled Vortex kernel binary through the PCIe interface into a known startup address located in DDR4 memory.
2. **Kernel Boot:** The Vortex IP is then triggered to begin execution by the host through PCIe BAR0 that controls the AXI_LITE_reg IP to trigger the reset signal for Vortex.
3. **Execution Phase:** Vortex starts reading the kernel from DDR, line by line, and executes the kernel.

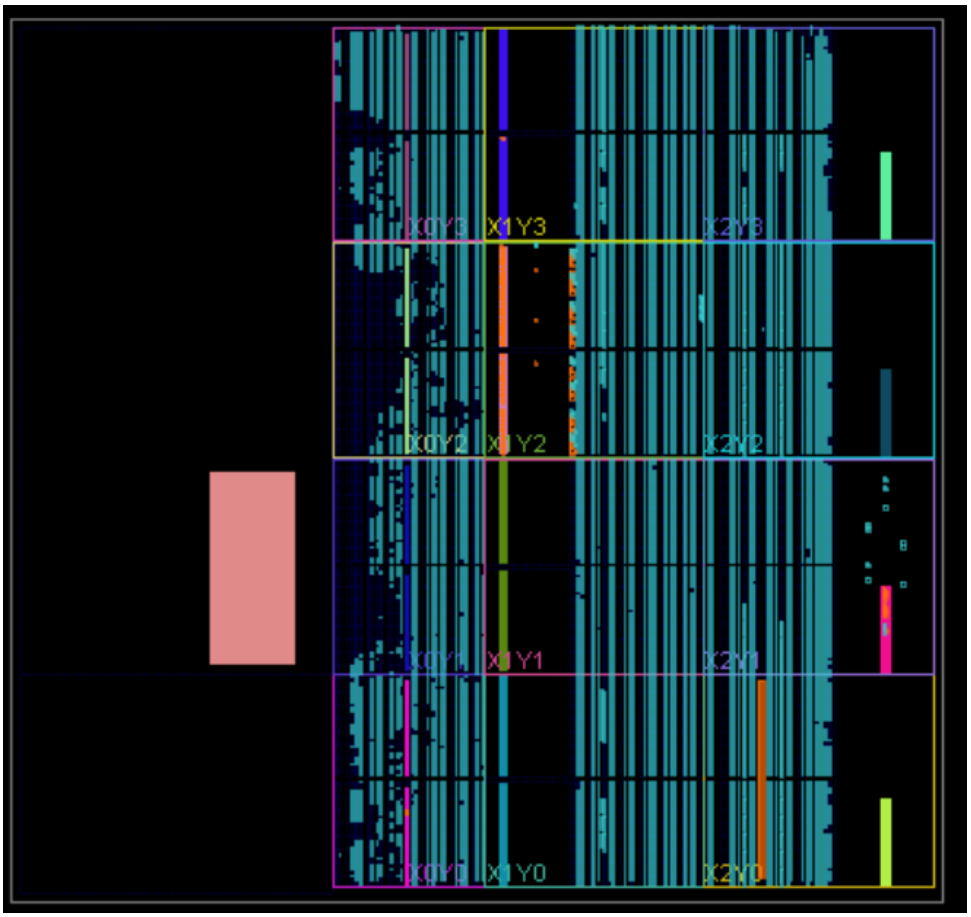


Figure 8.9: LUts and CLBs used after implementation

4. **Finishing Phase:** After Vortex finishes and uploads results to the destination address in DDR4, the busy signal is triggered to be low again.

5. **Completion Phase:** The host can read the busy signal through the AXI_LITE_reg IP via BAR0.

6. **Hardware Accelerator Trigger:** The host downloads results from the DDR4 destination address.

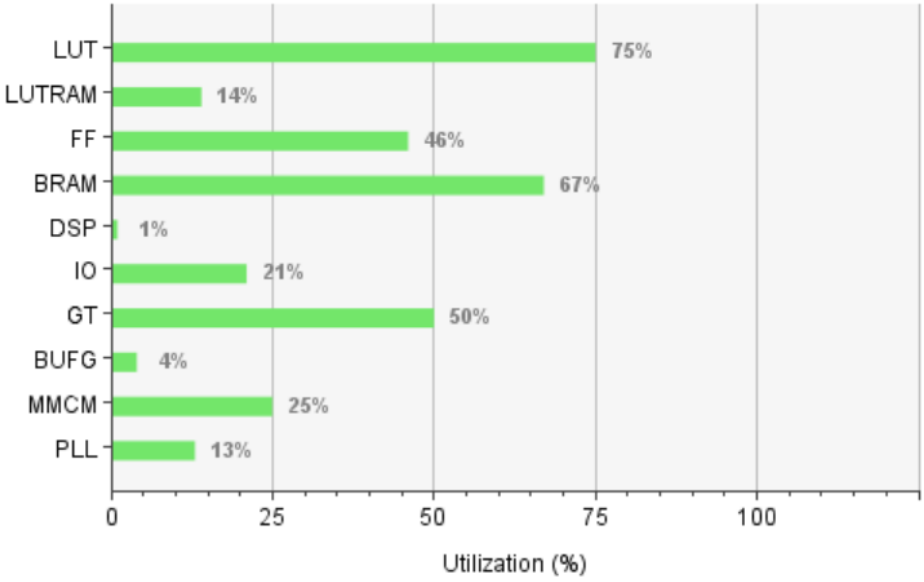


Figure 8.10: Utalization report

Name	CLB LUTs (87840)	CLB Registers (175680)
SOC_wrapper	74.98%	46.38%
dbg_hub (dbg_hub)	0.49%	0.41%
SOC_i (SOC)	74.50%	45.97%
axi_register_0 (SO...	0.06%	0.08%
ddr4_0 (SOC_ddr4...	9.62%	5.45%
rst_ddr4_0_300M (...	0.01%	0.02%
smartconnect_0 (S...	18.91%	14.04%
util_ds_buf (SOC_u...	0.00%	0.00%
util_vector_logic_0 (...	0.00%	0.00%
vortex_axi_wrapper...	26.13%	14.42%
xdma_0 (SOC_xdm...	19.76%	11.96%

Figure 8.11: UtalizationHairarchy

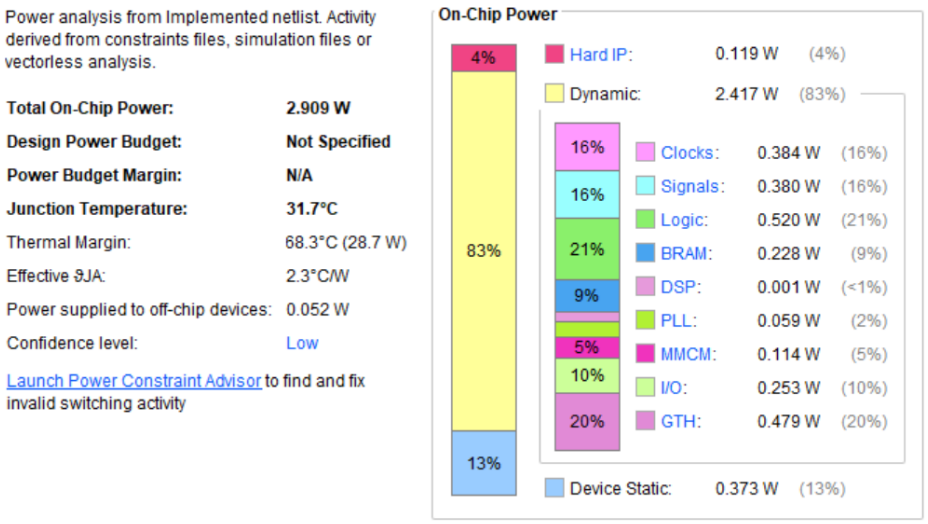


Figure 8.12: Power report

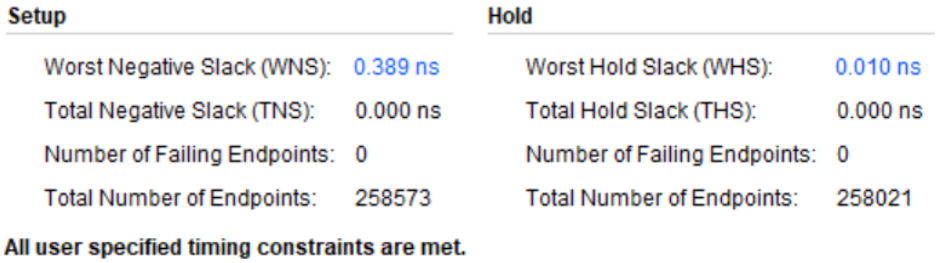


Figure 8.13: timitng report

Chapter 9

Test Bench

9.1 DCRs

9.1.1 Device Configurable Registers (DCRs)

DCR stands for Device Configurable Registers. Vortex has three inputs for DCRs, as shown in Figure **??**, which provide a control interface between the host processor (or driver) and the Vortex hardware. DCRs are memory-mapped registers in the `vx_dcr_data` unit. Initializing the startup address DCR register tells Vortex where to go in the connected memory model to find the first line of the kernel off-loaded by the host to start execution. An example of a startup address is `0x80000000`.

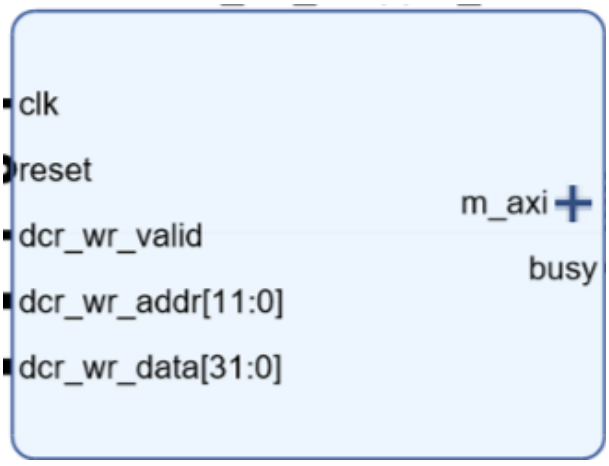


Figure 9.1: Vortex IP on Vivado

9.1.2 Kernel Argument Initialization

Initializing the kernel argument address informs Vortex of the location of constant arguments used during kernel execution. For example, in the basic test, the number of points is copied and moved to another location in memory. The kernel argument address specifies both the source location of the data and the destination to which it should be moved. For the NTT kernel, several arguments are used to configure execution. These

include the number of NTT points (n), the modulus value (q), the ψ value, and memory addresses for various data. Specifically, the input address indicates where input values are stored in memory, while the output address specifies where Vortex should write the results. Additionally, the `twiddle_address` refers to the memory location of the twiddle factors required for computation. The configuration also includes `block_dim` and `grid_dim`, which define how threads are organized and executed in the GPGPU architecture. The reason for uploading some arguments as memory addresses is due to a

limitation in the kernel argument passing mechanism. All `kernel_args` must be packed into a single 512-bit wide memory line at a specific argument location. To meet this constraint, large or complex arguments—such as input/output arrays or twiddle factors—are passed as addresses pointing to their actual locations in memory. These arguments are then fetched and managed internally by the kernel program during execution. Therefore,

the memory model must be initialized by the host, as shown in Figure **??**, according to the selected DCR values.

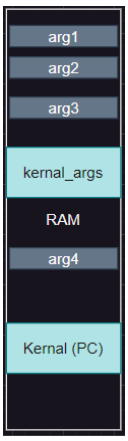


Figure 9.2: memory model sections

9.2 Testing on tool

9.2.1 RTLsim: runtime folder

The Vortex environment supports three test methods: **simx**, which uses the C++ model of Vortex; **RTLSim**, which relies on RTL simulation through Verilator; and an **external testbench** using QuestaSim, which enables deeper debugging capabilities and more advanced testing control. In the VORTEX GPGPU project, the source files are organized

into multiple directories, notably the `runtime` and `sim` folders. The `runtime` folder contains the host-side runtime system. The `vortex.cpp` file is responsible for setting up the interface that communicates with the RTL-level implementation, including the connection to co-simulation interfaces via Verilator. The `rtlsim` component handles running the

application by instantiating the Vortex core and initializing the core configuration parameters such as `NUM_THREADS`, `NUM_WARPS`, `NUM_CORES`, `NUM_CLUSTERS`, `CACHE_BLOCK_SIZE`, and `GLOBAL_MEM_SIZE`, which can be passed through BlackBox. It also instantiates and

builds the memory model for the simulated Vortex GPGPU core. This involves allocating a large block of memory that represents the global memory space shared across all GPU cores, where global variables, textures, and buffers are stored. This memory buffer is passed to the RTL simulator, enabling read/write operations during kernel execution.

9.2.2 RTLsim: sim folder

The host-side driver program is responsible for initializing the simulation environment. It creates a RAM model that starts at address `0x0`, using a page size of 4 KB. The `RAM` class provides the read and write functionality for memory access during simulation. The processor instance is attached to this RAM, enabling it to access the memory space throughout the simulation.

The compiled kernel is loaded into the RAM at a specified `startup_addr` using:

```
ram.loadBinImage(program, startup_addr);
```

To configure the Vortex core before execution, DCR (Device Configuration Register) inputs are written using:

```
processor.dcr_write();
```

Once the memory and registers are initialized, the simulation begins with:

```
processor.run();
```

9.2.3 Simulation in `rtlsim`

The `rtlsim` environment supports waveform-level debugging by generating a trace file in VCD (`.vcd`) format. This trace file, named `trace.vcd`, contains detailed signal-level activity of the RTL simulation and can be visualized using tools such as GTKWave.

To enable VCD generation, the simulation must be run with the debug level set to 3 by passing the flag `--debug=3` to Blackbox. Once generated, the `trace.vcd` file can be opened with GTKWave for signal inspection and in-depth debugging of RTL behavior.

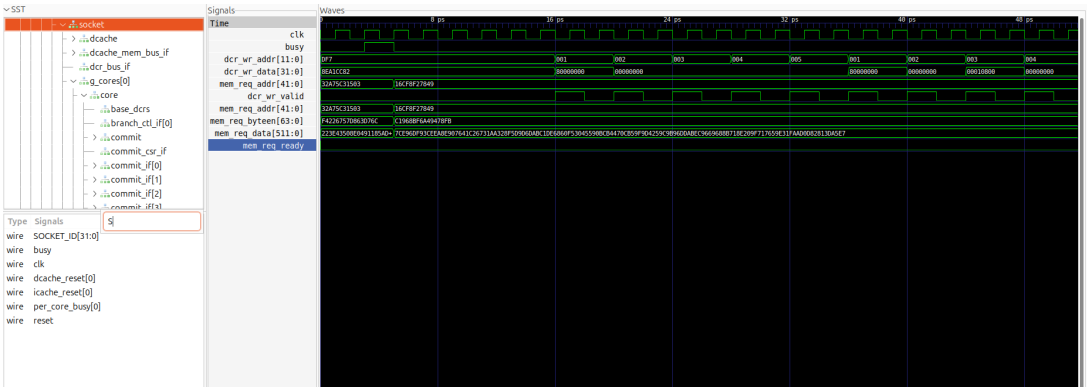


Figure 9.3: GTK wave form

Vortex GPGPU connects to the memory model through the AXI4 protocol. Therefore, understanding AXI4 signaling is essential for debugging. Observing AXI4 transactions is a crucial step during the creation of the test bench.

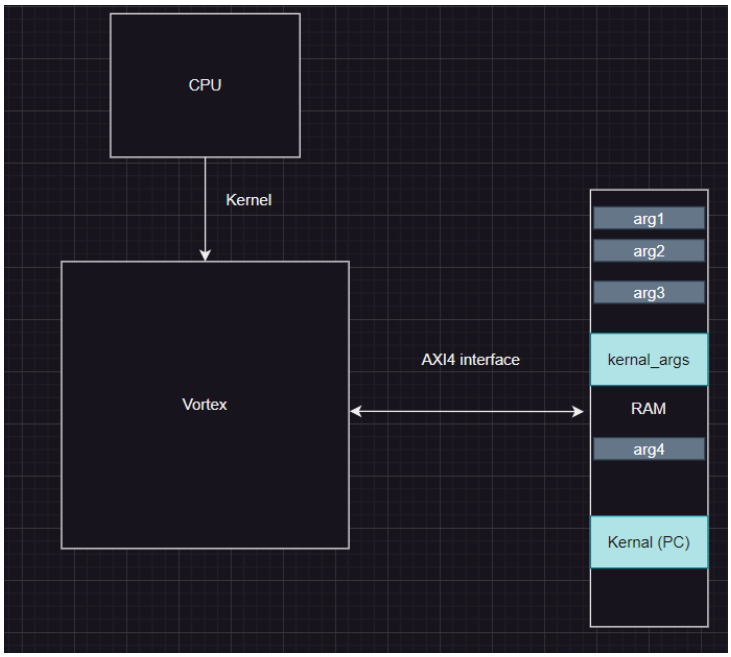


Figure 9.4: TB block diagram

9.2.4 CSV file

The CSV file contains all kernel instructions executed by the Vortex core. It provides detailed visibility into the instruction execution sequence, including the values written, destination registers, source register contents, and the occurrence of custom instructions. This enhances debugging and analysis capabilities. The file can be generated using the following command:

```
./ci/trace_csv.py -trtlsim run.log -otrace_rtlsim.csv
```

```
uuid,PC,opcode,instr,core_id,warp_id,tmask,destination,operands
0,0x80000000,JAL,0xc000ef,0,0,1000,"x1={0x80000004, -, -, -}",
1,0x8000000c,CSRRS,0x340027f3,0,0,1000,"x15={0x10800, -, -, -}","x0={0x0, -, -, -}"
2,0x80000010,LW,0x7a703,0,0,1000,"x14={0x100, -, -, -}","x15={0x10800, -, -, -}"
3,0x80000014,LW,0x87a683,0,0,1000,"x13={0x10000, -, -, -}","x15={0x10800, -, -, -}"
4,0x80000018,LW,0x107a583,0,0,1000,"x11={0x10400, -, -, -}","x15={0x10800, -, -, -}"
5,0x8000001c,CSRRS,0xcc2027f3,0,0,1000,"x15={0x0, -, -, -}","x0={0x0, -, -, -}"
6,0x80000020,MUL,0x2e787b3,0,0,1000,"x15={0x0, -, -, -}","x15={0x0, -, -, -}, x14={0x100, -, -, -}"
7,0x80000024,BEQ,0x2070863,0,0,1000,,"x14={0x100, -, -, -}, x0={0x0, -, -, -}"
8,0x80000028,ADD,0xf70733,0,0,1000,"x14={0x100, -, -, -}","x14={0x100, -, -, -}, x15={0x0, -, -, -}"
9,0x8000002c,SLLI,0x271713,0,0,1000,"x14={0x400, -, -, -}","x14={0x100, -, -, -}"
10,0x80000030,SLLI,0x279793,0,0,1000,"x15={0x0, -, -, -}","x15={0x0, -, -, -}"
11,0x80000034,ADD,0xd787b3,0,0,1000,"x15={0x10000, -, -, -}","x15={0x0, -, -, -}, x13={0x10000, -, -, -}"
12,0x80000038,ADD,0xd70733,0,0,1000,"x14={0x10400, -, -, -}","x14={0x400, -, -, -}, x13={0x10000, -, -, -}"
13,0x8000003c,SUB,0x40d585b3,0,0,1000,"x11={0x400, -, -, -}","x11={0x10400, -, -, -}, x13={0x10000, -, -, -}"
14,0x80000040,LW,0x7a603,0,0,1000,"x12={0xdeadbeef, -, -, -}","x15={0x10000, -, -, -}"
15,0x80000044,ADD,0xf586b3,0,0,1000,"x13={0x10400, -, -, -}","x11={0x400, -, -, -}, x15={0x10000, -, -, -}"
16,0x80000048,ADDI,0x478793,0,0,1000,"x15={0x10004, -, -, -}","x15={0x10000, -, -, -}"
17,0x8000004c,SW,0xc6a023,0,0,1000,,"x13={0x10400, -, -, -}, x12={0xdeadbeef, -, -, -}"
18,0x80000050,BNE,0xfee798e3,0,0,1000,,"x15={0x10004, -, -, -}, x14={0x10400, -, -, -}"
19,0x80000040,LW,0x7a603,0,0,1000,"x12={0xbd5b7ddf, -, -, -}","x15={0x10004, -, -, -}"
20,0x80000044,ADD,0xf586b3,0,0,1000,"x13={0x10404, -, -, -}","x11={0x400, -, -, -}, x15={0x10004, -, -, -}"
21,0x80000048,ADDI,0x478793,0,0,1000,"x15={0x10008, -, -, -}","x15={0x10004, -, -, -}"
22,0x8000004c,SW,0xc6a023,0,0,1000,,"x13={0x10404, -, -, -}, x12={0xbd5b7ddf, -, -, -}"
23,0x80000050,BNE,0xfee798e3,0,0,1000,,"x15={0x10008, -, -, -}, x14={0x10400, -, -, -}"
24,0x80000040,LW,0x7a603,0,0,1000,"x12={0x7ab6fbbf, -, -, -}","x15={0x10008, -, -, -}"
```

Figure 9.5: CSV file

Here, we can clearly observe that the `startup_address` is set to `0x80000000`, and the `kernel_args` address is `0x00010800`.

9.2.5 TB in Questasim

Compilation Order

When compiling complex designs, it is essential to use a script file (e.g., `.do`) to enforce the correct compilation order and manage file inclusion properly. This ensures that the design is compiled without errors. The recommended compilation steps are:

1. Include directory paths for all header files.
2. Compile the header files.
3. Compile all required packages.
4. Compile the RTL (Register Transfer Level) files.

Core - RAM Interface

In our project, only one core, one socket, and one cluster are used, so the number of AXI banks is defined as `AXI_NUM_BANKS = 1`.

All Vortex AXI signals are defined as unpacked arrays. To interface them with the memory model, they must be converted into packed arrays.

Example 1: Signals without processing For example, the signal `m_axi_araddr` is an output from Vortex as a master:

```
[AXI_ADDR_WIDTH-1:0]  m_axi_araddr  [AXI_NUM_BANKS];
```

To map this port directly (when no processing is needed), select the first bank index:

```
.s_axi_araddr(m_axi_araddr[0]),
```

Example 2: Signals requiring processing For signals such as `m_axi_bvalid` that need processing:

```
[AXI_NUM_BANKS]
```

Use the following steps:

1. Define a new packed signal:

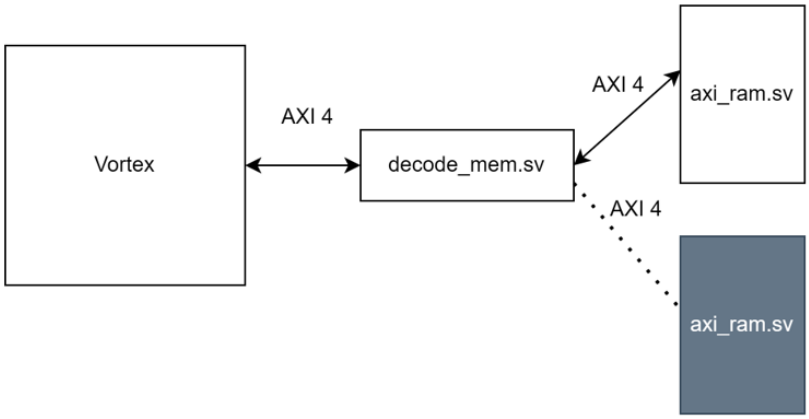
   ```
   logic packed_bvalid0;
   ```

2. Assign the packed signal to the appropriate bank index:

   ```
   m_axi_bvalid[0] = packed_bvalid0;
   ```

3. Use the packed signal in the port mapping:

   ```
   .s_axi_bvalid(packed_bvalid0),
   ```



`decode_mem.sv` is used as an interface to manage the conversion of unpacked AXI signal arrays into packed arrays. It also handles decoding logic when multiple memory models are used in the simulation. When a high startup address is used (e.g., `0x80000000`) and

the kernel arguments are placed at `0x12000`, the required memory depth becomes significantly large—on the order of gigabytes. Due to simulation memory limitations in tools like Questasim, this scenario requires instantiating two separate memory models: one for the kernel file upload and another for kernel arguments and output data. Alternatively,

the startup address can be changed to a lower value (e.g., `0x7000`), which allows the use of a single memory model. This simplifies the setup and eliminates the need for decoding logic in the `decode_mem.sv` file.

9.2.6 Testbench Construction

Address Translation Note Note that the actual memory location to which read and write operations are performed is shifted right by 6 bits (i.e., `>>6`). For example, if the `startup_address` is set to `0x7000`, the kernel is actually uploaded to address `0x1C0` in the memory model.

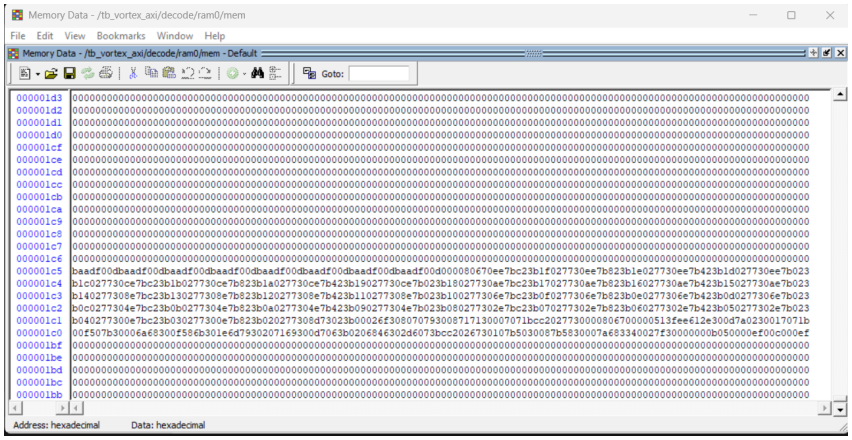


Figure 9.6: ploaded kernel for Basic test

9.2.7 Kernal_args upload

Kernal arguments depend on how the test or kernel program works, so it varies from test to test. An easy way to find and extract the kernel is to go to the test `common.h` file (e.g., Basic test, NTT test, ...) and look for the `kernel_args_t` struct.

Example from basic test:

```
typedef struct {
    uint32_t count;
    uint64_t src_addr;
    uint64_t dst_addr;
} kernel_arg_t;
```

Example from NTT test:

```
typedef struct {
    uint32_t N;            // Size of the vector
    uint32_t q;            // Modulus
    uint32_t psi;          // 2N-th root of unity
    uint64_t input_addr;   // Address of input vector
    uint64_t output_addr;  // Address of output vector
    uint64_t twiddle_addr; // Address of twiddle factors
    uint32_t grid_dim;     // 1D grid dimension
    uint32_t block_dim;    // 1D block dimension
} kernel_arg_t;
```

Uploading the struct uses descending order to concatenate the whole struct into one line of 512 bits width, padding the unused bits with zeros. For basic tests:

```
decode.ram0.mem[(kernel_arg >> 6)] = {{(320){1'b0}}, dst_addr, src_addr, count};
```

For NTT test:

```
decode.ram0.mem[kernel_arg >> 6] =
    {block_dim, grid_dim, twiddle_addr, output_addr, input_addr, psi, q, n};
```

9.2.8 VortexAXI signals configurations

Vortex uses an AXI adapter module (`VX_axi_adapter.sv`) that manages the conversion of Vortex request signals (`mem_req_valid`, `mem_req_rw`, `mem_req_byteen`, `mem_req_addr`, `mem_req_data`, `mem_req_tag`, `mem_req_ready`) and response signals (`mem_rsp_valid`, `mem_rsp_data`, `mem_rsp_tag`, `mem_rsp_ready`) to 39 AXI signals, with 4 signals assigned to default values. RAM models can ignore these unused signals, as done in the testbench used in this project.

Ignored signals in ram model

Signal	Width	Direction	Default	Description
`m_axi_arregion`	[3:0]	Vortex→RAM	4'b0000	AXI4 Region signaling lets one slave interface act as multiple logical interfaces using `ARREGION`.
`m_axi_awqos`	[3:0]	Vortex→RAM	4'b0000	Used to prioritize transactions; this value indicates the lowest priority.
`m_axi_awregion`	[3:0]	Vortex→RAM	4'b0000	Optional feature used to indicate the region of address space for a write transaction.
`m_axi_arqos`	[3:0]	Vortex→RAM	4'b0000	Same concept as `m_axi_awqos`; applies to read transactions.

Table 9.1: Default AXI4 signals set by the Vortex AXI Adapter

Note that default values are assigned from VX_axi_adapter.sv

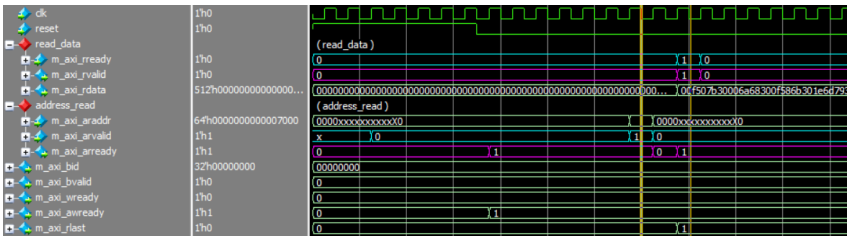
9.2.9 Tracing Basic test

Basic test instantiates the memory RAM model and uploads some number of words (e.g., `count = 100`) to the source address `0x10000`, then copies the words and loads them to the destination address `0x10400`. For simplicity in our testbench, `count = 1`, and only

one word was uploaded with the value `512'hdeadbeaf`.

Operation tracing

Tracing the waveform is done primarily by observing the AXI connections along with the `busy` and `reset` signals. This approach helps monitor the interaction between the core and memory model during different phases of execution, such as data read/write and idle periods.

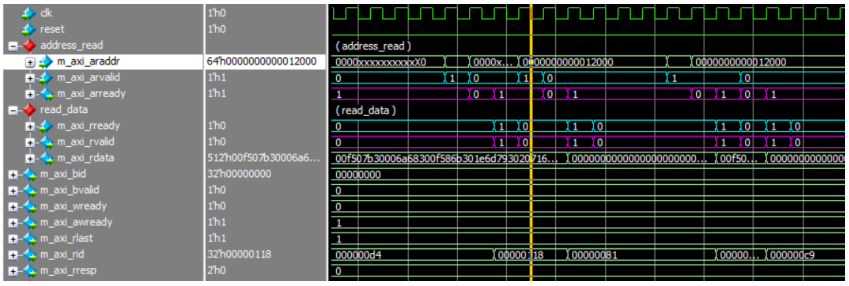


1. After resetting, the Vortex core starts by loading the first line of the kernel. It sends a read request to the memory with the address equal to `startup_addr` (0x7000) and waits for the memory to assert the `ARREADY` (address read ready) signal.

2. Once the memory loads the first line of the kernel, it asserts the `RVALID` (read data valid) signal and waits for the Vortex to assert the `RREADY` (read data ready) signal. When both signals are high, the kernel line is received by Vortex and the read handshake is completed.

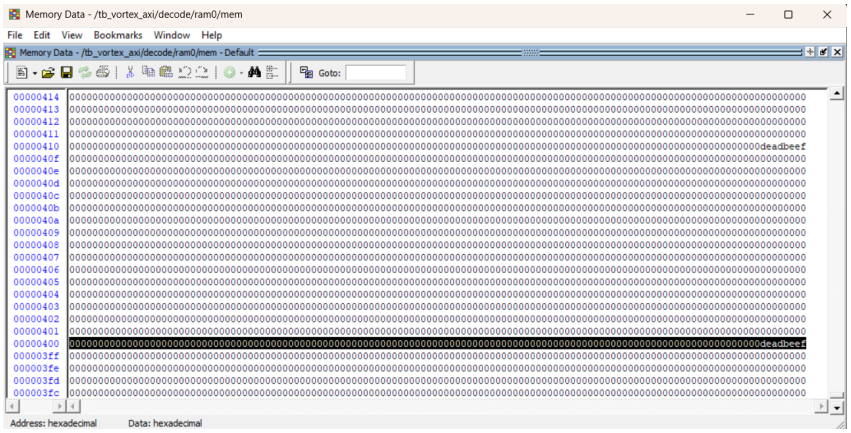
3. After some cycles, depending on the kernel line execution, Vortex initiates a second memory read to load `kernel_args` from address `0x12000`, using the same AXI handshaking protocol.

4. The same process continues, with Vortex issuing memory read or write requests and waiting for appropriate handshakes, until the kernel execution reaches its end.



9.2.10 End of simulation

After completing the kernel execution, the simulation confirms that the basic test has passed. The word was successfully moved from the source address `0x10000` (i.e., `0x10000 >> 6 = 0x400`) to the destination address `0x10400` (i.e., `0x10400 >> 6 = 0x410`).



And busy signal goes low.

```
# Source buffer written.
# Running kernel...
# Kernel executed with startup address=7000, src address=10000, dst address=10400.
# Busy went low after 897 cycles
# Reading destination buffer from address 66560...
# Data[0] = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000deadbeef
# Destination buffer read completed.
# Verifying results...
# ********************************
# ********************************
# Verification PASSED.
# ********************************
# ********************************
# ** Note: $stop    : Vortex_tb.sv(262)
#    Time: 9075 ps  Iteration: 1  Instance: /tb_vortex_axi
# Break in Module tb_vortex_axi at Vortex_tb.sv line 262
```

Chapter 10

ASIC Design & Implementation

10.1 ASIC Flow

This chapter describes the ASIC implementation of the Vortex GPGPU architecture enhanced with a custom NTT accelerator. Our goal was to explore area and performance tradeoffs of moving from an FPGA prototype to an ASIC flow. Due to licensing limitations of proprietary EDA tools, the entire RTL-to-GDSII process was performed using the open-source OpenLane framework and the SkyWater Sky130 Process Design Kit (PDK).

10.1.1 Overview and Tool Selection

The Vortex GPGPU is an open-source, RISC-V-based architecture designed for highly parallel applications such as cryptography and signal processing. To extend its capabilities, we integrated hardware blocks for the Number Theoretic Transform (NTT), a core primitive in lattice-based cryptography.

The design was initially validated on FPGA with promising performance results. To further improve power and area efficiency, an ASIC implementation was pursued using OpenLane. OpenLane provides a complete ASIC flow using tools like Yosys for synthesis, OpenROAD for physical design, and Magic for verification. The target PDK was Sky130—a 130nm technology node suitable for academic prototyping.

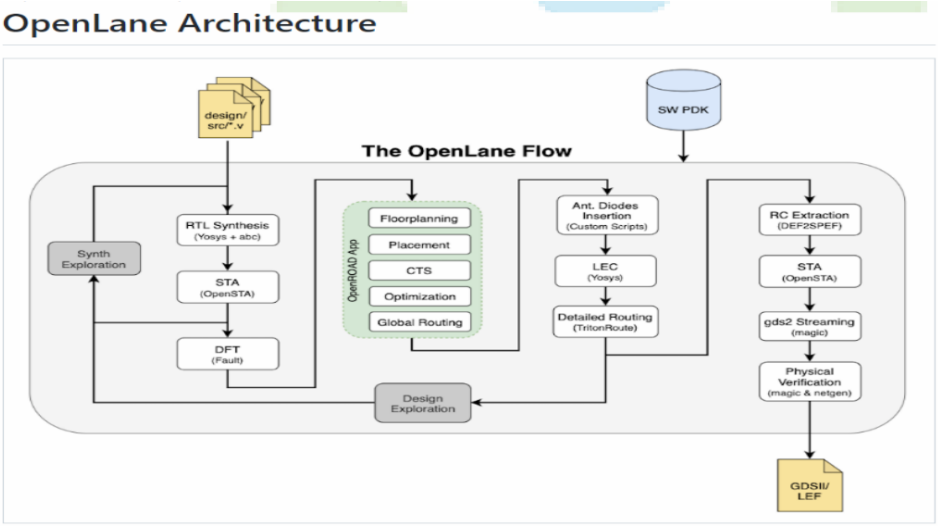


Figure 10.1: Overview of the OpenLane ASIC flow

10.1.2 SystemVerilog Compatibility and Conversion

The Vortex+NTT source code was written in advanced SystemVerilog, utilizing constructs such as ‘`interfaces`‘, ‘`structs`‘, and ‘`always_comb`‘ blocks, which are not fully supported in the OpenLane flow (ENABLE_SYSTEM_VERILOG = 0). To enable synthesis, we used the `sv2v` tool to convert SystemVerilog to synthesizable Verilog.

This conversion step introduced significant manual effort:

- AXI interfaces had to be flattened into explicit port-level connections.
- Structs and parameterized modules required simplification or rewriting.
- Several post-processing passes were required to ensure compatibility with Yosys.

Although the resulting Verilog code passed synthesis, the translation may have introduced subtle functional mismatches, as some structural constructs were abstracted away.

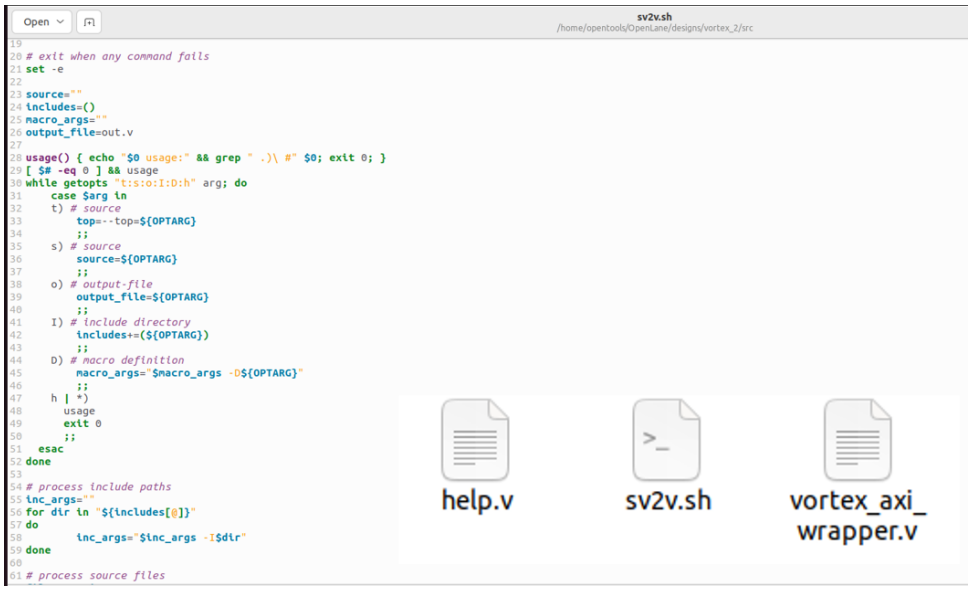


Figure 10.2: SystemVerilog to Verilog conversion via `sv2v`

10.1.3 Synthesis

Synthesis was performed using Yosys with the following key parameters:

- **Clock period:** 20 ns
- **Strategy:** Area-optimized (‘SYNTH_STRATEGY = AREA 0‘)
- **Max fanout:** 25 (‘SYNTH_MAX_FANOUT = 25‘)

The resulting netlist included 1,399,964 standard cells and 242,938 nets. These values indicate high design complexity due to the combination of GPU pipelines, instruction decoding logic, and the NTT arithmetic blocks.

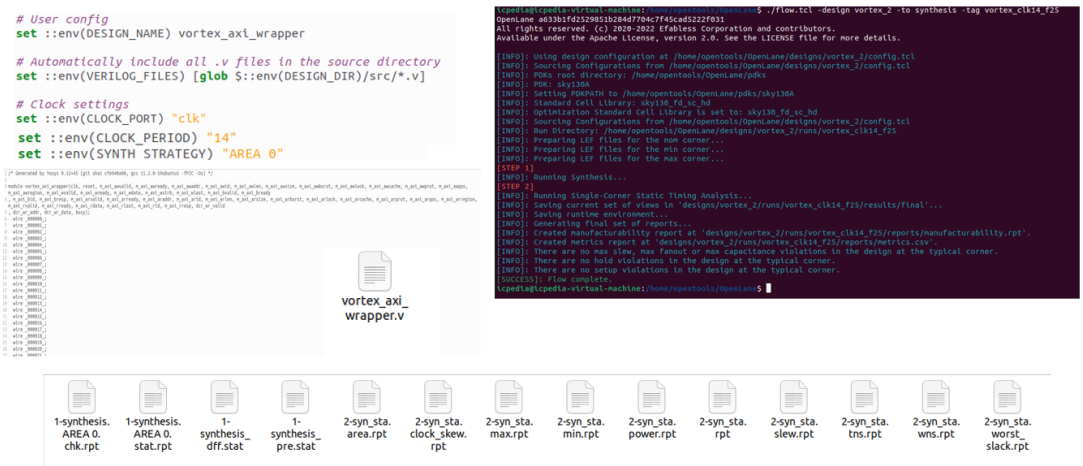


Figure 10.3: RTL-to-Gate-Level Synthesis Result

10.1.4 Floorplanning and Power Planning

After synthesis, floorplanning was initiated. The die area was initialized at 7000×7000 μm^2 with a core area of 6990×6990 μm^2. The following configurations were applied:

- **Core utilization:** 30% (`FP_CORE_UTIL = 30`)
- **Placement density:** 35% (`PL_TARGET_DENSITY = 0.35`)
- **IO Mode:** Random (`FP_IO_MODE = 1`)
- **Power straps:** Across all metal layers (met1 to met5) with 150 µm pitch

This configuration was chosen to prevent congestion and reduce memory usage during placement.

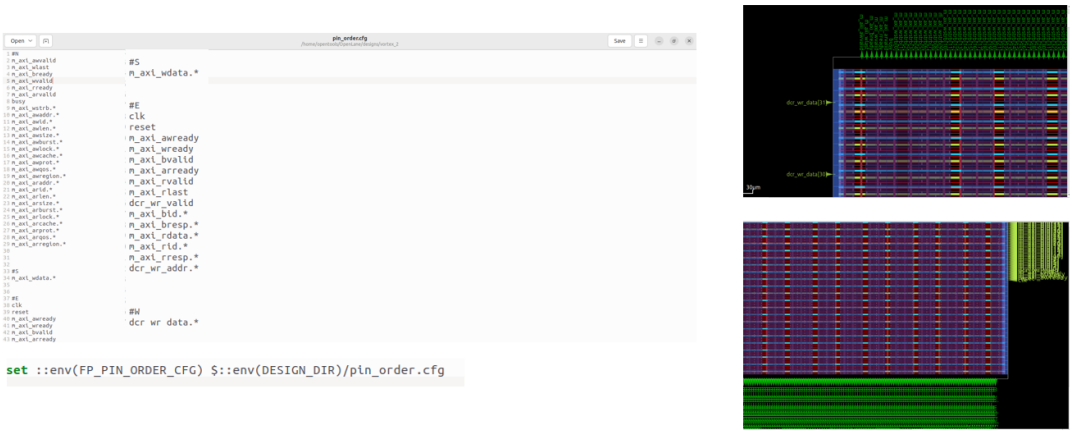


Figure 10.4: Initial floorplanning with IO pin assignment

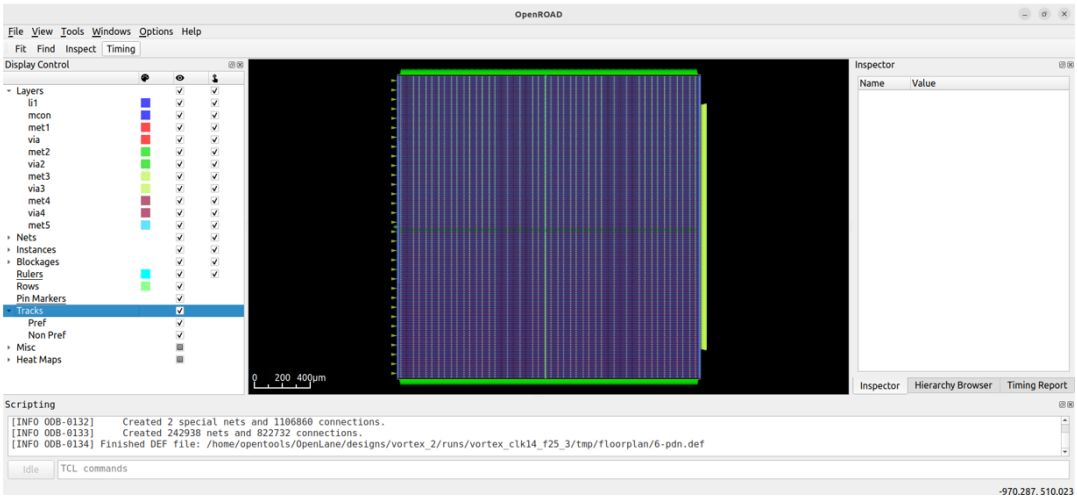


Figure 10.5: Expanded chip outline view

Figure 10.6: Power grid structure with full metal coverage (met1–met5)

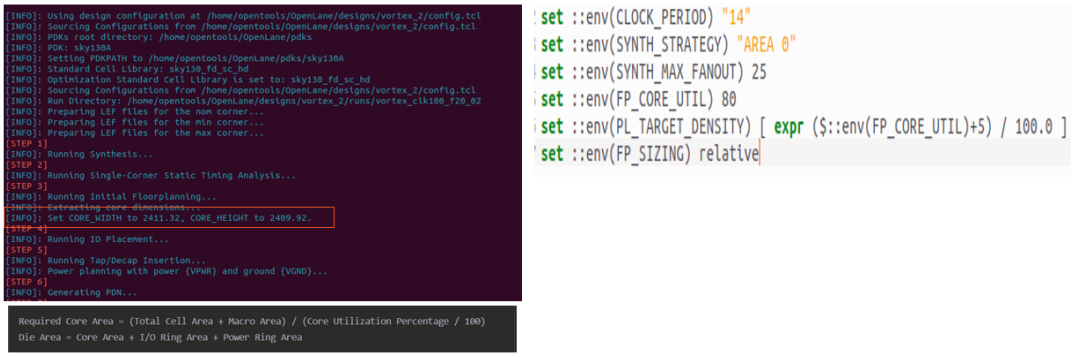


Figure 10.7: Finalized power planning view

10.1.5 Placement

Global placement was performed with the following parameters:

- `GPL_MAX_ITER = 600`
- `GPL_TIMING_DRIVEN = 1`
- `GPL_ROUTABILITY_DRIVEN = 1`
- `GPL_CELL_PADDING = 0`

These values optimized for routability without over-padding, which had previously worsened congestion.

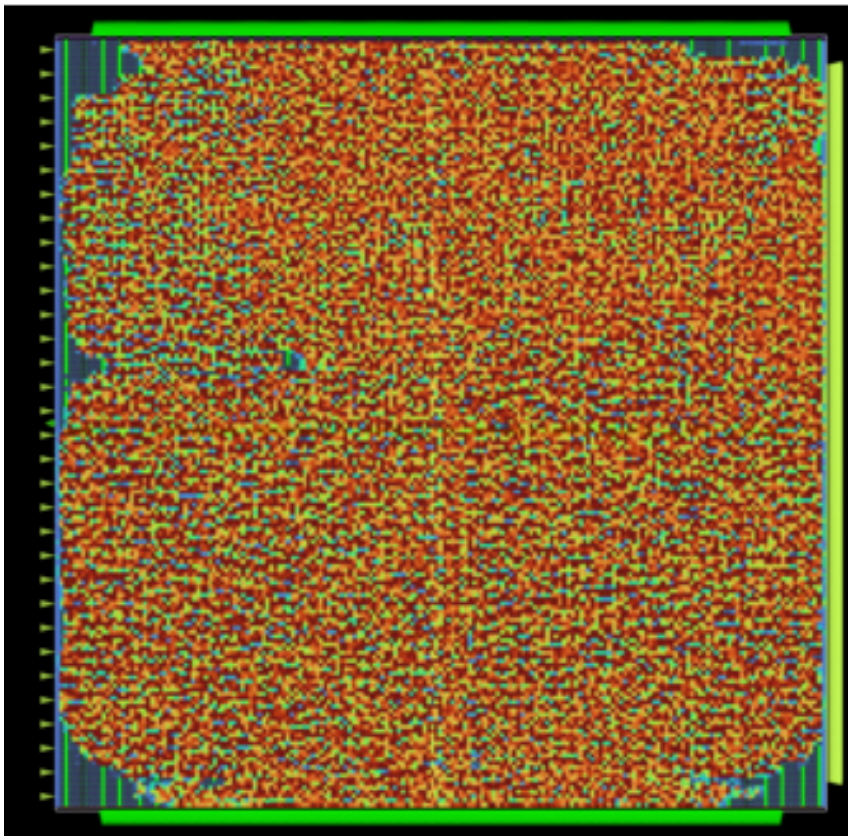


Figure 10.8: Global placement of standard cells

10.1.6 Detailed Placement and Limitations

Detailed placement failed due to memory exhaustion, producing error ‘DPL-0036‘, indicating that cells (e.g., _388707_, _390242_) could not be legalized due to overlap. Attempts to mitigate this included:

- Reducing padding to zero.
- Increasing utilization to 30%.
- Reducing the die area to 7000×7000 µm^2.

However, the issue persisted, likely due to high cell density near memory controllers or NTT logic, combined with the lack of macro placement constraints. The estimated memory requirement exceeded 128 GB of RAM, which was unavailable on the current system.

Figure 10.9: Detailed placement failure due to overlap and memory limitations

10.1.7 Results and Analysis

- Synthesis and floorplanning were successfully completed.
- The design produced a netlist with nearly 1.4 million components.
- Power planning used straps across met1–met5 to ensure current delivery.
- Global placement succeeded, but detailed placement could not legalize all cells.
- Key bottlenecks included memory capacity, congestion in high-density blocks, and lack of native support for SystemVerilog constructs.

10.1.8 Lessons Learned and Future Work

1. **Tool Limitations:** OpenLane provided a viable open-source path, but scalability remains a challenge for designs of this complexity.
2. **SystemVerilog Handling:** sv2v conversion enabled basic synthesis but required heavy manual intervention. Native support (e.g., via Surelog and UHDM) is highly recommended.
3. **Resource Constraints:** Designs of this size require systems with 128+ GB RAM and multi-core CPUs for successful placement and routing.
4. **Configuration Tuning:** Careful adjustment of core utilization, placement density, and die size is essential to balance congestion and legal placement.
5. **Future Enhancements:**
    - Partition the design into subsystems (e.g., separate NTT accelerator).
    - Use macro blocks and physical constraints to guide placement.
    - Explore commercial-grade EDA tools and more advanced PDKs (e.g., TSMC 28nm, GF12LP) for better performance and yield.

References

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. *General-Purpose Graphics Processor Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 5 edition, 2011.

[3] Ozgun Ozerk, Can Elgezen, Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. Efficient number theoretic transform implementation on GPU for homomorphic encryption. Cryptology ePrint Archive, Paper 2021/124, 2021.

[4] Ardianto Satriawan, Infall Syafalni, Rella Mareta, Isa Anshori, Wervyan Shalannanda, and Aleams Barra. Conceptual review on number theoretic transform and comprehensive review on its implementations. *School of Electrical Engineering and Informatics, Institut Teknologi Bandung*, 2023.

[5] Georgia Tech University. A general-purpose gpu accelerator. https://github.com/vortexgpgpu/vortex, 2023.

[6] Ali Şah Özcan and Erkay Savaş. Two algorithms for fast GPU implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410, 2023.