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

$$\begin{array}{r|l}
 & 32x^2 + 52x + 61 \\
x^4 + 1 & 32x^6 + 52x^5 + 61x^4 + 60x^3 + 34x^2 + 16x + 5 \\
 & 32x^6 + 0x^5 + 0x^4 + 0x^3 + 32x^2 \quad - \\
 & 52x^5 + 61x^4 + 60x^3 + 2x^2 + 16x + 5 \\
 & 52x^5 + 0x^4 + 0x^3 + 0x^2 + 52x \quad - \\
 & 61x^4 + 60x^3 + 2x^2 - 36x + 5 \\
 & 61x^4 + 0x^3 + 0x^2 + 0x + 61 \quad - \\
 & 60x^3 + 2x^2 - 36x - 56
\end{array}$$

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 n^{th} 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.

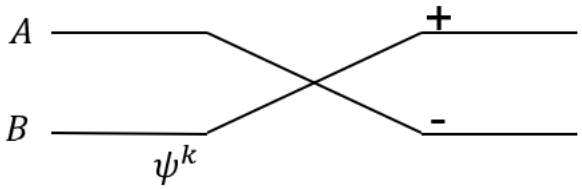

Figure 2.4: Cooley-Tukey (CT) butterfly unit for calculating NTT

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:

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

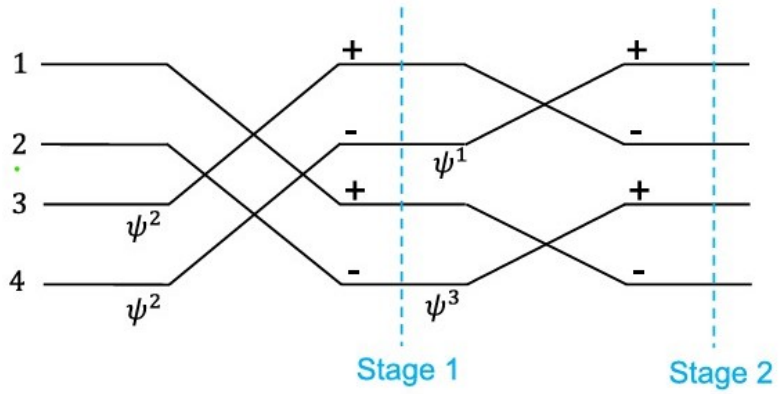

Figure 2.5: CT Butterflies for n = 4 and [1, 2, 3, 4] as its input

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.

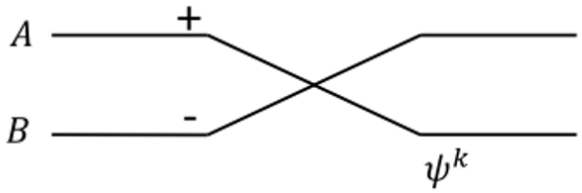

Figure 2.6: Gentleman-Sande (GS) butterfly unit for calculating INTT

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:

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

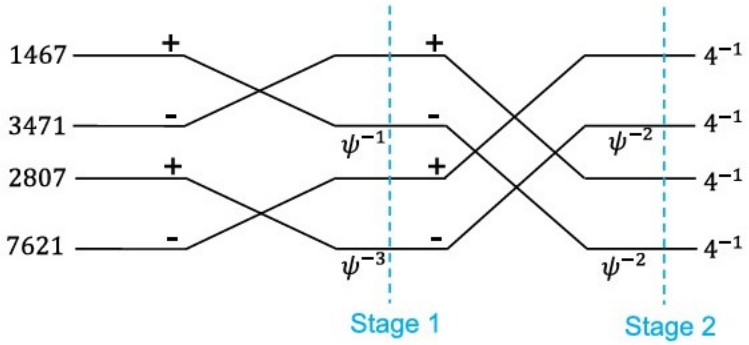

Figure 2.7: GS Butterflies for n = 4 and [1467, 2807, 3471, 7621] as its input

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.

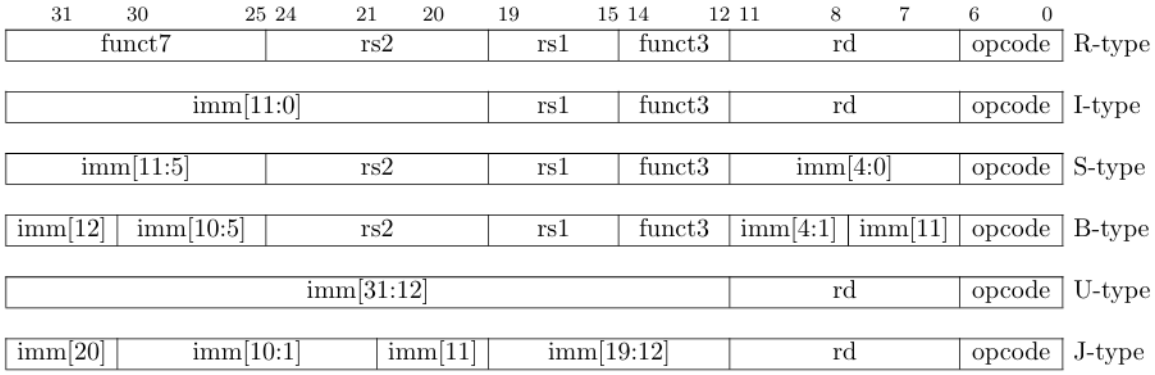

Figure 3.1: RISC-V Base Instruction Formats

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 SIMD & SIMT Execution

4.1.1 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.

4.1.2 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)

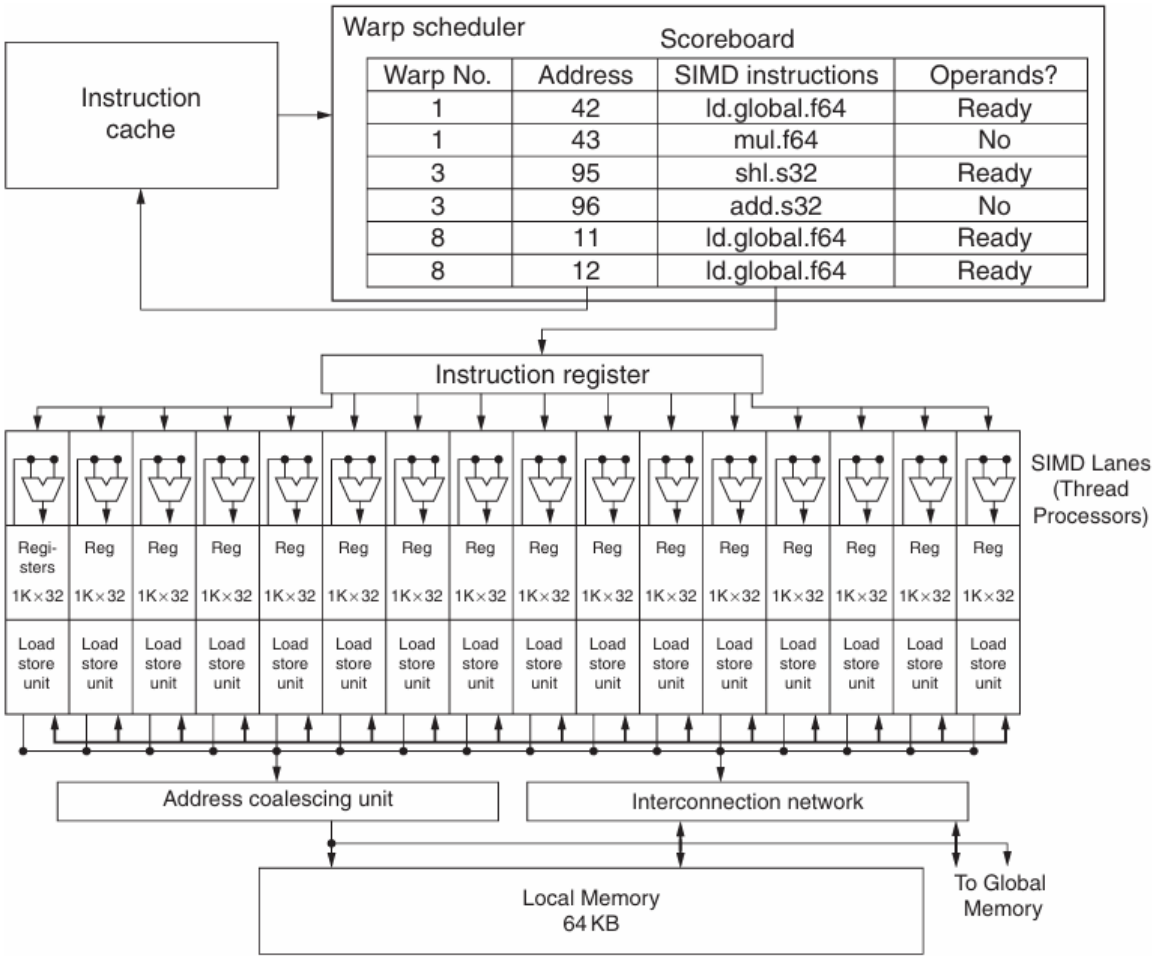

Figure 4.1: Simplified block diagram of a Multithreaded SIMD Processor

4.2 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.

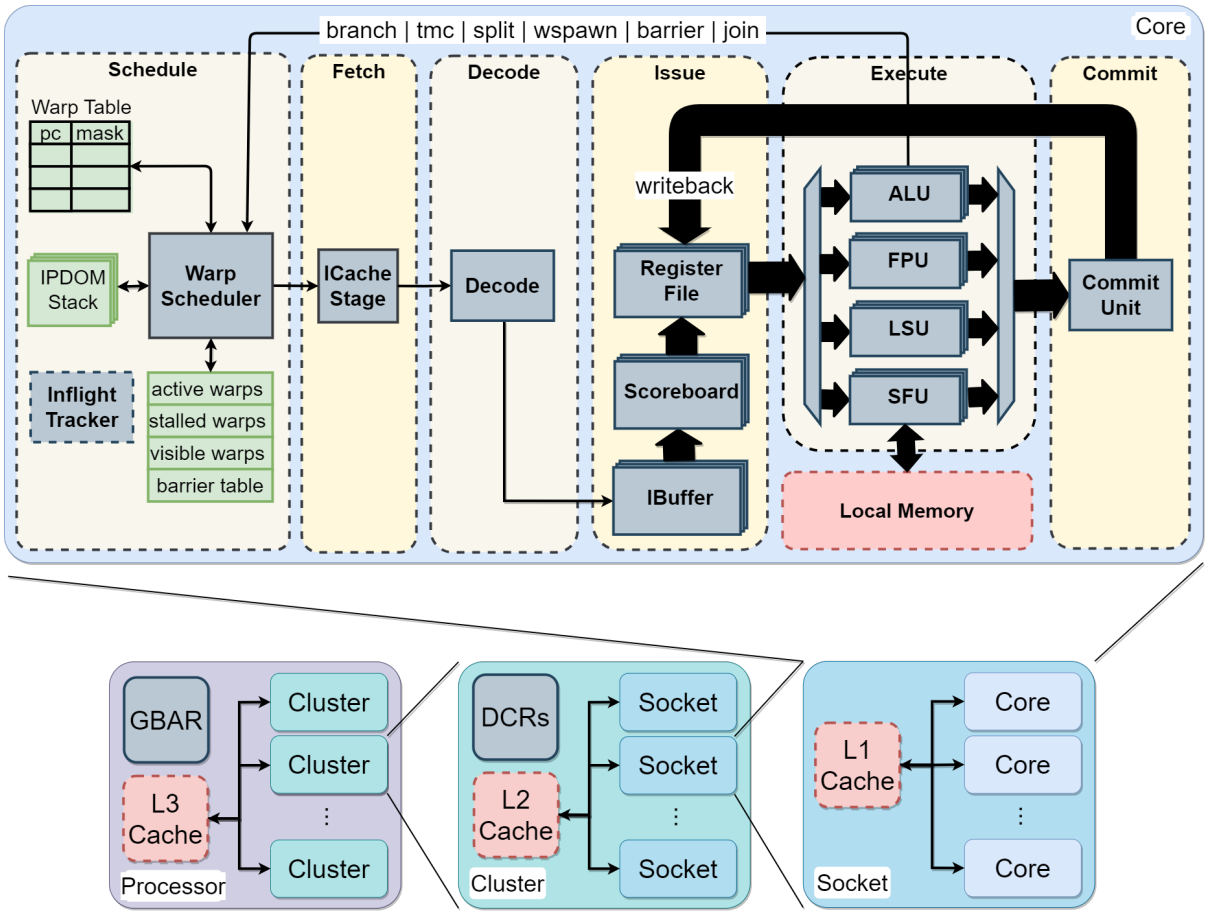

Figure 4.2: Vortex Microarchitecture

4.3 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.

4.3.1 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.

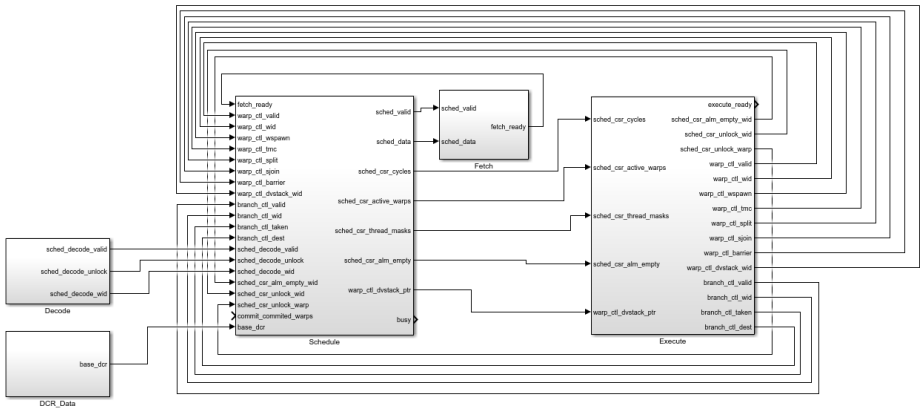

Figure 4.3: Block Diagram of the Schedule Stage

4.3.2 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.

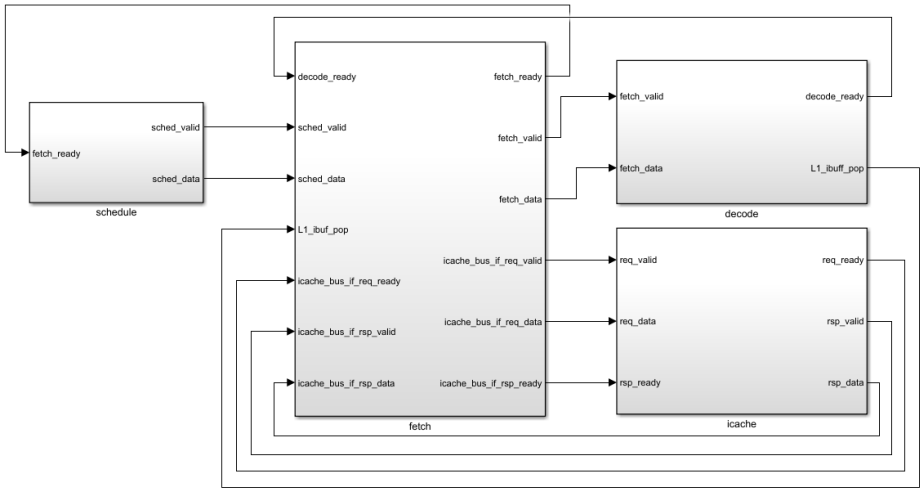

Figure 4.4: Block Diagram of the Fetch Stage

4.3.3 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.

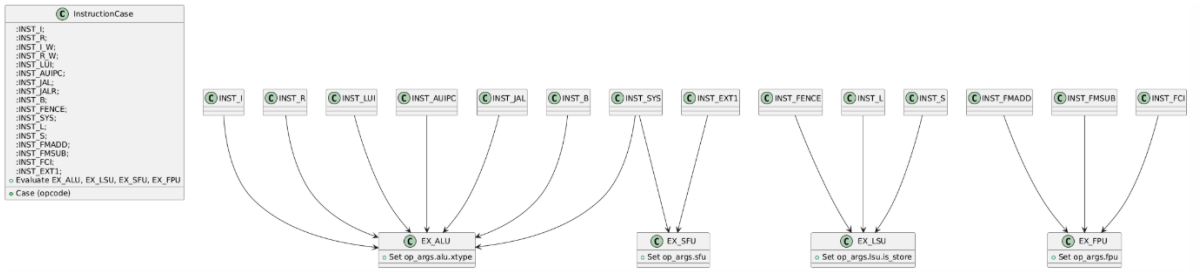

Figure 4.5: Mapping of different instruction types to different execution units

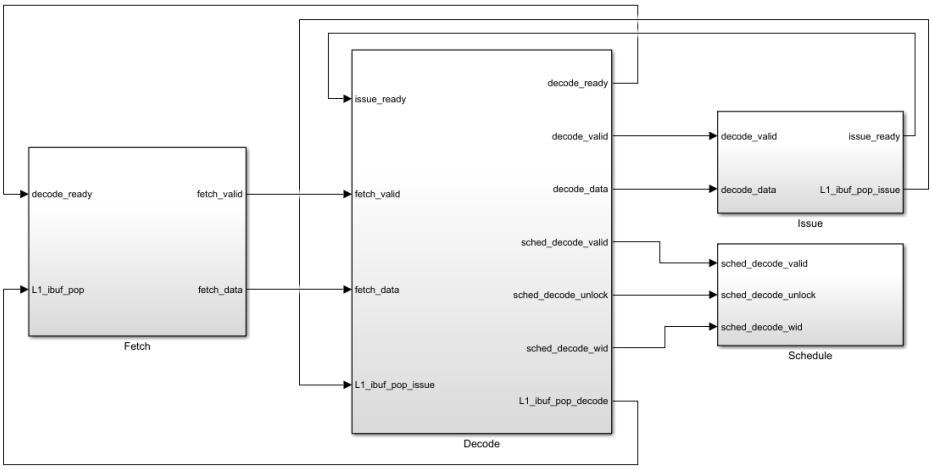

Figure 4.6: Block Diagram of the Decode Stage

4.3.4 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.

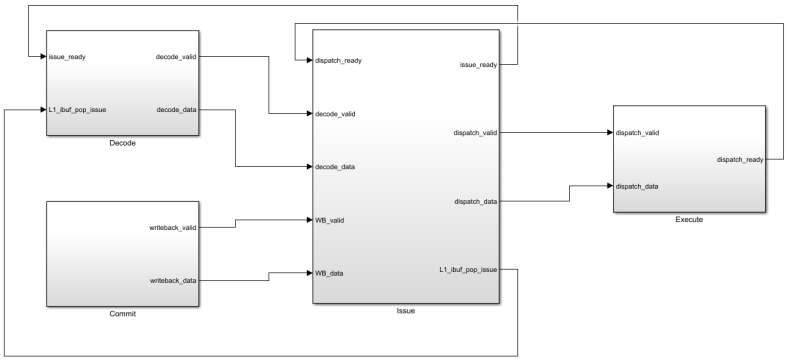

Figure 4.7: Block Diagram of the Issue Stage

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.

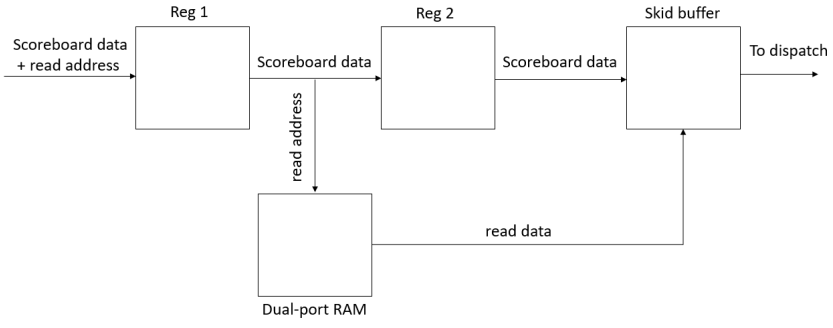

Figure 4.8: Block Diagram of Operand Collector

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.

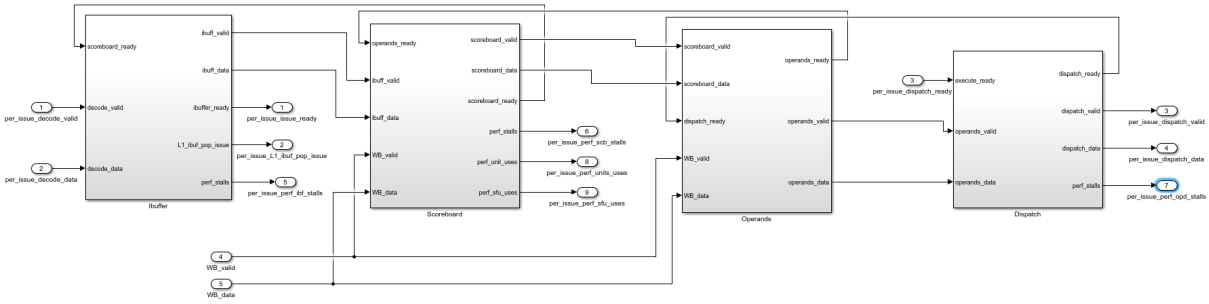

Figure 4.9: Block Diagram of Issue Slice

4.3.5 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.

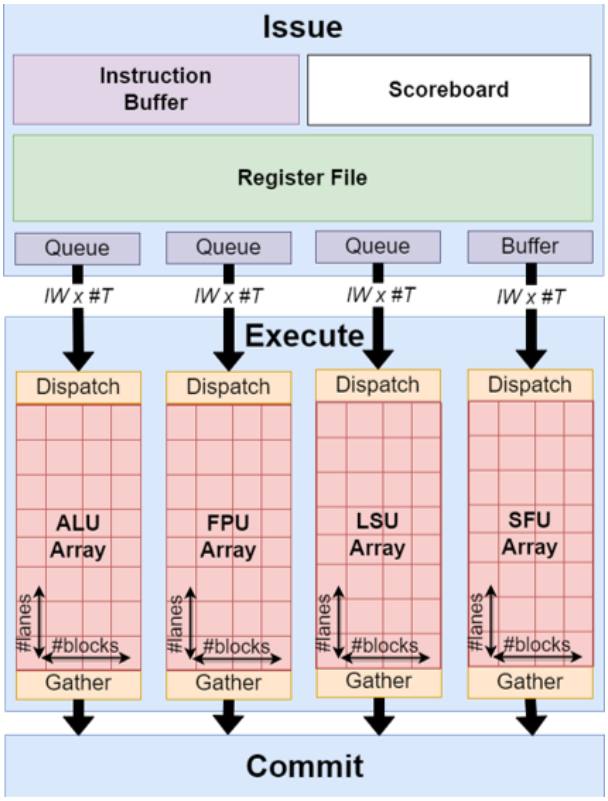

Figure 4.10: Dispatch and Gather of Threads

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

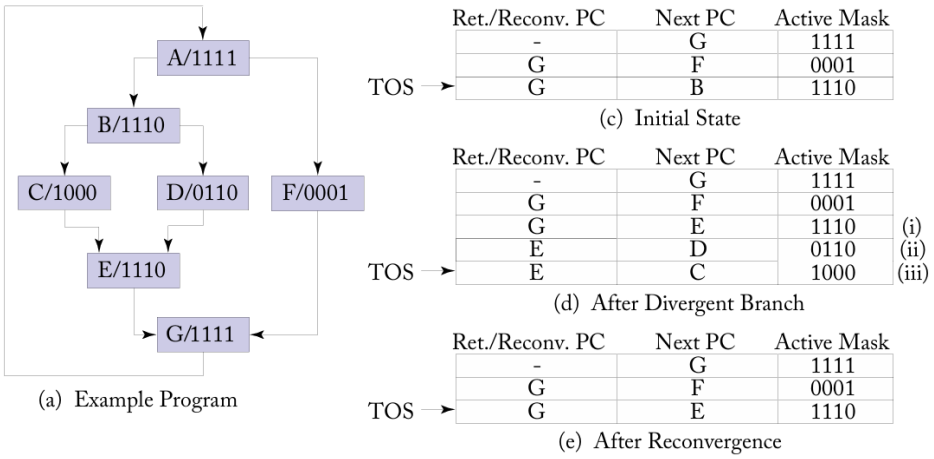

Figure 4.11: Branch Divergence and IPDOM Stack

  - **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.

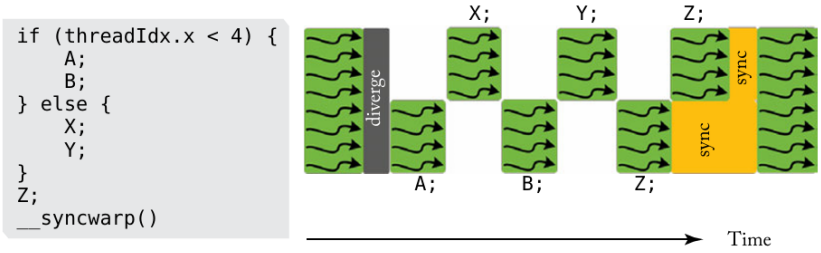

Figure 4.12: Barrier Handling in SIMT Execution

- **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.

4.3.6 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.

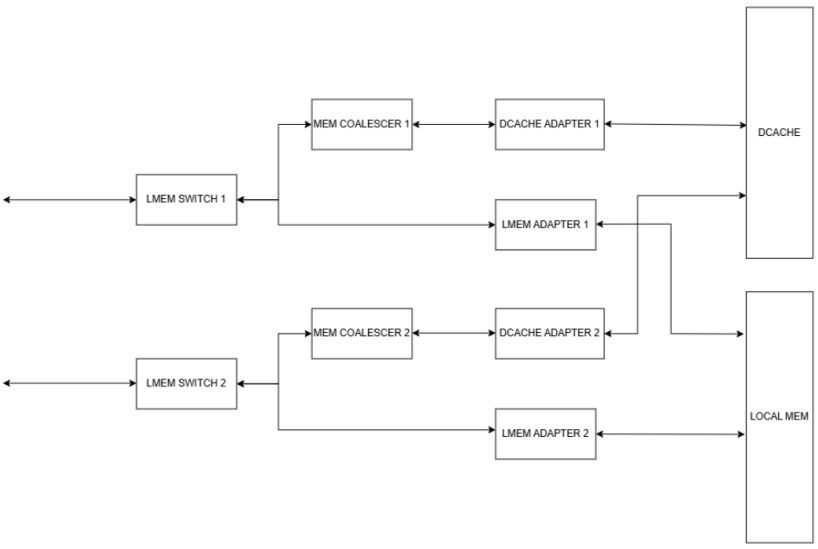

Figure 4.13: Memory Unit Block Diagram

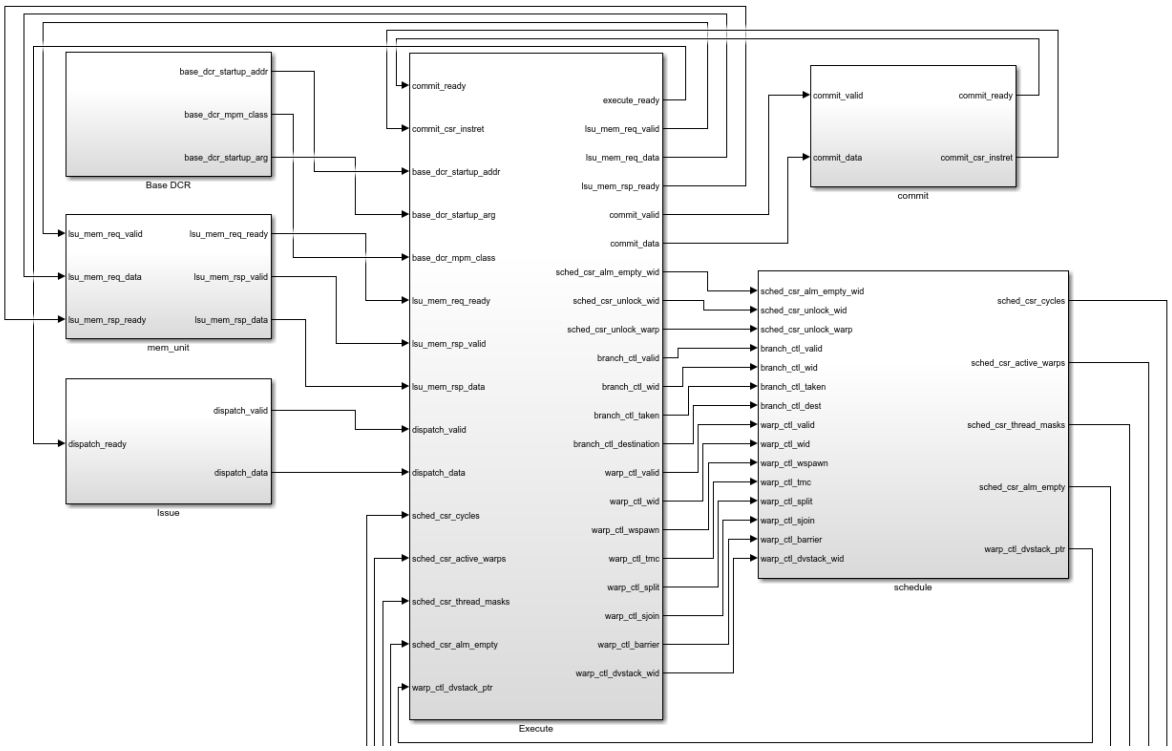

Figure 4.14: Block Diagram of the Execute Stage

4.3.7 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.

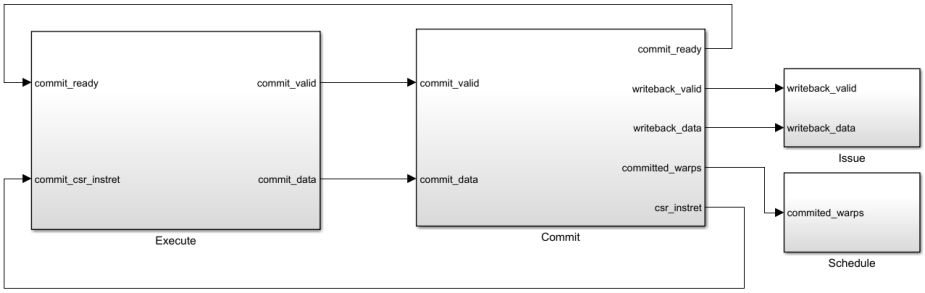

Figure 4.15: Block Diagram of the Commit Stage

4.3.8 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.

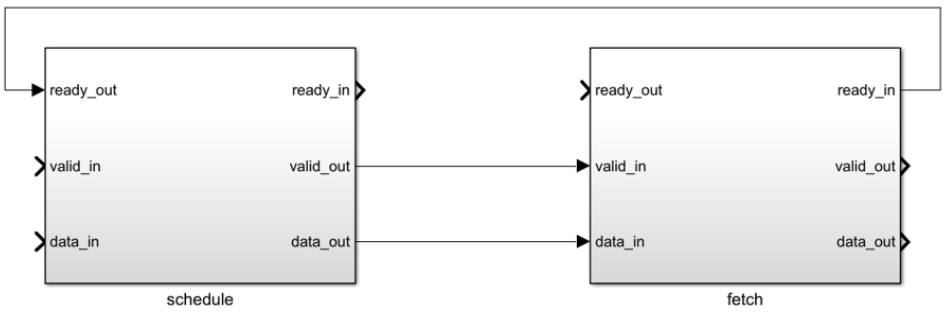

Figure 4.16: Skid Buffer Handshake Protocol between Schedule & Fetch Stages

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

4.4 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.

4.4.1 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**

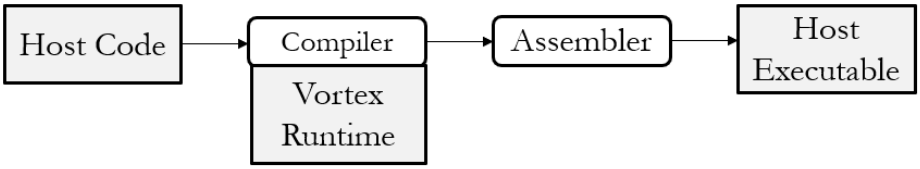

Figure 4.17: Host Compilation Flow

4.4.2 GPU Compilation Flow

The flow goes through the following steps:

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

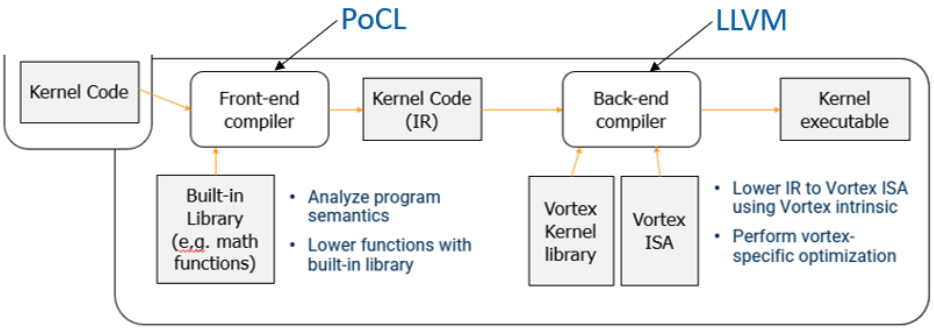

Figure 4.18: GPU Compilation Flow

4.4.3 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.

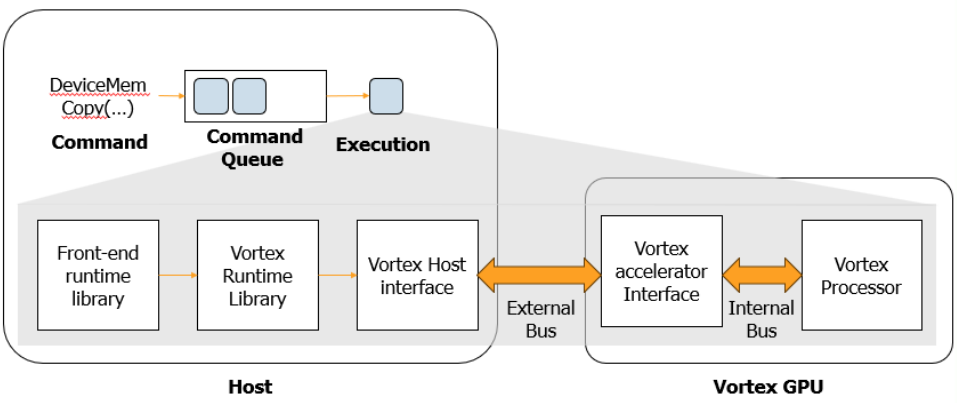

Figure 4.19: Execution Flow

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] 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.

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