NTT 201 - Foundations of NTT Hardware Design

Yuval Domb
yuval@ingonyama.com

Chapter 1

Theory

1.1 Introduction

NTT, or Number Theoretic Transform. is the term used to describe a Discrete Fourier Transform (DFT) over finite fields. In general, there's nothing unique about NTT that does not apply to DFT and visa-versa. However, the uses of NTT, its typical operating point, and some subtleties make it a very different beast. This note is twofold. In this chapter, we describe NTT from the use-case point-of-view, emphasizing what it is good for and what it is not. In the next chapter, we show how an NTT of a specific operating point can be optimally implemented in hardware.

1.2 From FT to DFT

1.2.1 FT

Fourier Transform (FT) [1] converts a continuous signal[1] from the time domain to the frequency domain and is defined as

$$X(\varepsilon) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi\varepsilon t} dt \tag{1.1}$$

FT exists under complicated necessary conditions, but a generally sufficient condition is that $x(t) \in \mathcal{L}_1$ where $\mathcal{L}_1$ is the set of all absolutely integrable functions.[2] Other than that, $x(t)$ is an infinite, non-periodic function of time. FT is a unitary linear transformation and obeys Parseval's theorem [2]. One can think of $X(\varepsilon_0)$ as the quantity of signal energy at frequency ε_0 since it is exactly the correlation between the signal and the discrete tone $e^{-i2\pi\varepsilon_0 t}$.

1.2.2 DTFT

The Discrete-Time FT (DTFT) [3] is an FT for discrete-time signals and is defined as

$$X(f) = \sum_{-\infty}^{\infty} x[n] e^{-i2\pi f n} \tag{1.2}$$

[1]We tend to refer to the time-domain function as a signal in the context of FT.

[2]a.k.a. Lebesgue integrable functions.

The sampled discrete tone $e^{-i2\pi fn}$ is periodic, leading to a periodic frequency-domain function $X(f)$ with maximum frequency at $f = 0.5$. The DTFT can be thought of as a folded version of FT where the domain $\varepsilon \in (-\infty, \infty)$ is folded onto $f \in (-0.5, 0.5]$. DTFT can still be thought of in the sense of signal energy per frequency, but now the frequency f contains the accumulated energy of $X(\varepsilon)$ at frequencies $\{..., \varepsilon-2, \varepsilon-1, \varepsilon, \varepsilon+1, \varepsilon+2, ...\}$. The folding phenomenon in the frequency domain, often referred to as aliasing, is associated with time-domain sampling. In that sense, we can think of $x[n]$ as a sampled version of $x(t)$ (i.e. $x[n] = x(nT)$ where T is the sampling period).[3]

1.2.3 DFT

But what about periodic signals? On the one hand, a periodic signal has infinite energy. On the other hand, a single period is deterministically sufficient to represent it. So how about a transformation that represents a single period of a periodic signal? As it turns out, this is possible. For continuous-time periodic signals, this is called a Fourier Series (FS) [4]. For discrete-time periodic signals, it is called a DFT [5] and is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi \frac{k}{N} n} \tag{1.3}$$

The frequency equivalent for DFT is $\frac{k}{N}$ for $k \in [0, N-1]$. DFT can be viewed as a normalized DTFT of a periodic signal. It is for that reason that for DFT we typically refer to signal power rather than energy.

The following table summarises the relationship between different transforms:

	FT	DTFT	FS	DFT
Periodic TD (Sampled FD)			✓	✓
Periodic FD (Sampled TD)		✓		✓

Note that it may seem that DFT is finite-time and finite-frequency. In fact, that is not the case but it is rather periodic in both time and frequency. This, as we shall see, has much influence on some of its main properties.

1.3 Properties of DFT

The set of discrete tones used in the DFT can be rewritten as

$$e^{-i2\pi \frac{k}{N} n} = \left(e^{-i\frac{2\pi}{N}}\right)^{kn} = \omega^{kn} \tag{1.4}$$

where ω is termed the N'th root-of-unity, since N is the smallest integer such that $\omega^N = 1$. We can now rewrite the DFT in its generalized form

$$X[k] = \sum_{n=0}^{N-1} x[n] \omega^{kn} \tag{1.5}$$

where ω^{kn} is a generalized discrete tone.

[3]An exact comparison between FT and DTFT requires normalization by the sampling period $T = f^{-1}$.

Replacing $x[n]$ with coefficients x_n of and substituting z for ω^k, results in the z-transform of the sequence $\{x_n\}_{n=0}^{N-1}$

$$X(z) = \sum_{n=0}^{N-1} x_n z^n \tag{1.6}$$

which is simply a polynomial of order $N-1$ in z.[4] With that, we can think of the transform $X[k]$ as polynomial evaluations of $X(z)$ over the root-of-unity powers[5] (i.e. $X[k] = X(\omega^k)$). This perspective is extremely useful in understanding some of the most useful properties of DFT.

1.3.1 Convolution

Linear convolution of two length-N sequences f_n and g_n is defined as

$$(f * g)_n = \sum_{m=0}^{N-1} f_m g_{n-m} \tag{1.7}$$

where both sequences are set zero for all $n \notin [0, N-1]$. In the polynomial form, this is described as

$$\begin{aligned}
(F \cdot G)(z) &= F(z)G(z) && (1.8)\\
&= \sum_{m=0}^{N-1} f_m z^m \cdot \sum_{k=0}^{N-1} g_k z^k && (1.9)\\
&= \sum_{m=0}^{N-1} \sum_{k=0}^{N-1} f_m \cdot g_k z^{m+k} && (1.10)\\
&= \sum_{m=0}^{N-1} \sum_{n=m}^{m+N-1} f_m \cdot g_{n-m} z^n && (1.11)\\
&= \sum_{n=0}^{2N-1} \left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m} \right) z^n && (1.12)
\end{aligned}$$

where equality (1.12) is obtained by using the previous definition that $g_n \equiv 0$ for all $n \notin [0, N-1]$. Note that the term in parenthesis is equivalent to our previous definition of linear convolution (1.7).

So how can we use this? Imagine you have two length-N sequences and you would like to calculate their linear convolution of length $2N-1$, or you have coefficients of two degree-$(N-1)$ univariate polynomials and you want to calculate the coefficients of their product polynomial of degree-$(2N-2)$. The above claims that the point-wise multiplication of two polynomials' evaluations results in the product polynomial. The nice thing is that the product of evaluations can be used to calculate the corresponding coefficients by Inverse DFT (IDFT).[6] This is nearly sufficient and only missing one crucial ingredient. This

[4]The Region of Convergence (RoC) is obviously $z \leqslant 1$ so it consists of $[\omega]$

[5]Note how the root-of-unity powers are an equidistant partition of the unit circle.

[6]DFT is a unitary transformation and is thus bijective (i.e. invertible). This property is discussed hereafter.

ingredient is an outcome of the *Fundamental Theorem of Algebra* [6] which states that any polynomial of degree-$(L-1)$ is uniquely defined by any L distinct evaluations. This means that any $2N-1$ evaluations of the product polynomial $(F \cdot G)(z)$ suffice to uniquely define the coefficients $(f * g)_n$. With that, let us define the following convolution algorithm for two sequences of length-N

$$(f * g)_{n=0}^{2N-1} = \mathrm{IDFT}_{2N-1}\left(\mathrm{DFT}_{2N-1}(f_n) \circ \mathrm{DFT}_{2N-1}(g_n)\right) \tag{1.13}$$

where $\circ$ denotes the Hadamard element-wise product, subscript $\square_{2N-1}$ depicts a transform's length, and all sequences are assumed to be zero-padded to the required length as needed. The advantage of the r.h.s. of (1.13), as discussed later in this work, is that its typical computational complexity is $\mathcal{O}(N \log N)$, whereas the complexity of the naive linear convolution is $\mathcal{O}(N^2)$.

Finally, one could ask what would result from the following algorithm with all transforms shortened to length N

$$(f \circledast g)_{n=0}^{N-1} = \mathrm{IDFT}_{N}\left(\mathrm{DFT}_{N}(f_n) \circ \mathrm{DFT}_{N}(g_n)\right) \tag{1.14}$$

As it turns out, and somewhat hinted by the l.h.s. notation of (1.14), the above is an algorithm for calculating the cyclic convolution of two length-N sequences or the polynomial product of two polynomials in a polynomial ring $P[z]/(z^N-1)$. To see why this is so let us reexamine the polynomial product in the context of the ring $P[z]/(z^N-1)$ (i.e. $F(z)G(z)/(z^N-1)$). Clearly, the modulo restriction adds the cyclic constraint $z^{k+N} = z^k$ which when plugged into (1.12) leads to the following

$$\begin{aligned}
F(z)G(z)/(z^N-1) &= \sum_{n=0}^{2N-1}\left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m}\right) z^n && (1.15)\\
&= \sum_{n=0}^{N-1}\left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m}\right) z^n + \sum_{n=N}^{2N-1}\left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m}\right) z^n && (1.16)\\
&= \sum_{n=0}^{N-1}\left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m}\right) z^n + \left(\sum_{m=0}^{N-1} f_m \cdot g_{n-m+N}\right) z^n && (1.17)\\
&= \sum_{n=0}^{N-1}\left(\sum_{m=0}^{N-1} f_m \cdot (g_{n-m} + g_{n-m+N})\right) z^n && (1.18)\\
&= \sum_{n=0}^{N-1}\left(\sum_{m=0}^{N-1} f_m \cdot \widetilde{g}_{n-m}\right) z^n && (1.19)
\end{aligned}$$

where $\widetilde{g}_n$ is the N-periodic concatenation of g_n. The resulting polynomial (1.19) is of maximum degree $N-1$ and is thus uniquely defined by N distinct evaluations of the product $F(z)G(z)$ which immediately leads to algorithm (1.14).

1.3.2 Interpolation

Interpolation is the process of calculating the value of an evaluation of a polynomial for some point in its domain. Since the polynomial coefficients (or time domain signal) are well-defined, the interpolated value is unique. For a single sample, this can be achieved by directly evaluating the polynomial from its coefficients or by using a form of *Lagrange Interpolation* such as the *Barycentric Formula* [7] over its available evaluations. In both cases, this is achievable with computational complexity $\mathcal{O}(N)$. It is often the case in data processing that we are interested in interpolating $L \geqslant N$ points located on a regularly-spaced grid. The naive implementation would require $\mathcal{O}(LN)$ complexity, but as it turns out, this can be performed by simple manipulation of the DFT, lowering the complexity to at most $\mathcal{O}(L \log L)$.

Suppose we have a sequence f_n of length N. One can easily use (1.6) to evaluate $F(z)$ anywhere in its domain. With a DFT of length N one can evaluate $\Omega = \{F(z) : z \in [\omega]\}$ where ω is the N'th root-of-unity and $[\omega]$ is the cyclic multiplicative subgroup generated by ω. With a DFT of length $L > N$ one can evaluate $\Upsilon = \{F(z) : z \in [\upsilon]\}$ where υ is the L'th root-of-unity. One may ask how a length-L DFT can be performed over the sequence f_n of length N. Since we treat the time domain sequence as polynomial coefficients, the only requirement is to zero-pad it to length-L and proceed as if it were a length-L sequence. This is equivalent to an order-$L-1$ univariate polynomial whose $L-N$ most significant monomials are zero. This leads to the following algorithm for arbitrary interpolation from subdomain $[\omega] \to [\upsilon]$

$$F([\upsilon]) = \mathrm{DFT}_L(\mathrm{IDFT}_N(F([\omega]))) \tag{1.20}$$

where $F([\omega]) \equiv \{F(z) : z \in [\omega]\}$ and zero-padding is assumed implicitly.

When $L = \lambda N$ for some $\lambda \in \mathbb{N}$ then $[\omega] = [\upsilon^\lambda] \leqslant [\upsilon]$ which means that

$$[\upsilon] = \bigcup_{l=0}^{\lambda-1} \upsilon^l[\omega] \tag{1.21}$$

where $\upsilon^l[\omega]$ are cosets of $[\omega]$. The key observation is that $F([\omega]) = F([\upsilon^\lambda]) \subset F([\upsilon])$ and it seems that in algorithm (1.20) the evaluations $F([\omega])$ would appear in both the input and output (i.e. some computational effort would be spent on recomputing $F([\omega])$). Can we somehow avoid this unnecessary effort? As it turns out, we can and this is particularly beneficial when λ is small (e.g. $\lambda = 2$ as is often the case). To see how this can be done, let us plug the k'th element from coset $\upsilon^l[\omega]$ into (1.6)

$$F(\upsilon^l\omega^k) = \sum_{n=0}^{N-1} f_n(\upsilon^l\omega^k)^n \tag{1.22}$$

$$= \sum_{n=0}^{N-1} \upsilon^{ln} f_n \omega^{kn} \tag{1.23}$$

In the context of the transform of length L (i.e. with υ as the root of unity) the above is exactly the formula for $F[\lambda k + l]$, the evaluations of coset l. This leads to the following algorithm for evaluating coset l from an original sequence of N evaluations

$$F(\upsilon^l[\omega]) = \mathrm{DFT}_N([\upsilon^l] \circ \mathrm{IDFT}_N(F([\omega]))) \tag{1.24}$$

where the Hadamard multiplication by $[\upsilon^l]$ is called modulation. Note how modulation of the coefficients by $[\upsilon^l]$ translates to a shift of the evaluations by υ^l (i.e. $F([\omega]) \to F(\upsilon^l[\omega])$). As an example, for $\lambda = 2$ the cost of interpolation reduces from a length-N IDFT and a length-$2N$ DFT to two length-N transforms.

Note that we used the frequency-shift/time-modulation property. The dual of this property (i.e. time-shift/frequency-modulation) holds in very much the same way and is extremely common in signal processing applications.

1.4 Fast DFT (FFT)

Fast Fourier Transform (FFT) is the common terminology for efficient methods for calculating the DFT. There are two unique methods.

The less common method, that can be shown to achieve near-linear complexity, is due to Winograd [8]. The problem with the Winograd method is that it is difficult to construct for arbitrary sizes. Constructions are known for fairly small-size transforms and as a result, it is not widely used. Although the Winograd FFT is utilized in Part 2 of this work, we will not explore it further here.

The common method for constructing FFT is typically associated with a 1965 paper due to Cooley and Tukey [9] even though it incorporates ideas that date back to Gauss as early as 1805 [10]. The method is often referred to as CT-FFT. The general idea is that when the transform length is a composite integer, it can be partitioned into smaller transforms whose results are combined to provide the desired calculation. Imagine we are interested in a length-N transformation where $N = LM$ with $L, M \in \mathbb{N}$. With CT-FFT the complexity can be reduced from $\mathcal{O}(N^2) = \mathcal{O}(L^2M^2)$ to approximately $\mathcal{O}(LM^2 + ML^2)$. The recursive application of the above method reduces the overall complexity to $\mathcal{O}(N \log N)$.

There are two main methods to construct recursive partitioning, Decimation-In-Time (DIT) and Decimation-In-Frequency (DIF). Both methods are essentially one and it can be shown that they both produce identical results up to permutation. Starting with the following DFT for a length-N sequence f_n

$$F(\omega^k) = \sum_{n=0}^{N-1} f_n \omega^{kn} \tag{1.25}$$

we can construct a DIT by partitioning the sequence to L sets $\left\{ \{f_{mL+l}\}_{m=0}^{M-1} \right\}_{l=0}^{L-1}$ and proceeding to calculate the partial sums of (1.25) accordingly.[7]

$$\begin{aligned}
F(\omega^k) &= \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} f_{mL+l} \omega^{k(mL+l)} && (1.26)\\
&= \sum_{l=0}^{L-1} \omega^{kl} \sum_{m=0}^{M-1} f_{mL+l} (\omega^L)^{km} && (1.27)\\
&= \sum_{l=0}^{L-1} (\omega^M)^{ql} \omega^{\rho l} \sum_{m=0}^{M-1} f_{mL+l} (\omega^L)^{\rho m} && (1.28)\\
&= \sum_{l=0}^{L-1} \mu_L^{ql} \omega^{\rho l} \sum_{m=0}^{M-1} f_{mL+l} \cdot \mu_M^{\rho m} && (1.29)
\end{aligned}$$

[7]The interleaved manner of the inner sets in the partition is often referred to as reverse-bit-ordering for DFTs of lengths that are powers of two.

$$= \sum_{l=0}^{L-1} \omega^{\rho l} \cdot \left(\sum_{m=0}^{M-1} f_{mL+l} \cdot \mu_M^{\rho m} \right) \mu_L^{ql} \tag{1.30}$$

where $\mu_L = \omega^M$ and $\mu_M = \omega^L$ are the L-th and M-th roots of unity respectively and $k = qM + \rho$ where $\rho < M$. Note that the internal sum in parenthesis represents L length-M DFTs. The outputs of these DFTs are multiplied element-wise by the twiddle-factors $\omega^{\rho l}$ and the products are fed into M length-L DFTs. An intuitive way to think about this algorithm is as follows:

1. Organise the sequence f_n of length N in an $L \times M$ matrix, where the construction is done column-wise.
2. Perform an independent length-M DFT per each row of the matrix.
3. Multiply all matrix elements such that element (l, ρ) is multiplied by the twiddle-factor $\omega^{\rho l}$.
4. Perform an independent length-L DFT per each column of the matrix.

The DIF alternative is to partition the transform $\{F(\omega^k)\}_{k=0}^{N-1}$ to $\left\{\{F(\omega^{mL+l})\}_{m=0}^{M-1}\right\}_{l=0}^{L-1}$ and proceed to calculate the partial transforms. Recursing the FFT partitioning is easy and will not be discussed here. As mentioned above, CT-FFT partitioning only relies on the ability to factorize N to integral factors. As such, the finest partition is limited by the prime factors of N.[8] It is worth noting that DFTs with lengths that are powers of two are extremely popular in practice.

1.5 NTT

NTT is a DFT over a finite field and is typically used in cryptography. Although one can give a notion of frequency and power to the transform, it is typically used as a mathematical tool for efficiently operating with polynomials in polynomial rings. An NTT for a sequence will always exist, provided a root-of-unity of appropriate order exists. An interesting property of finite fields is that the elements of its multiplicative group all lie on the unit circle and are all roots of unity.

A question that often arises is, when is NTT a beneficial tool? In general, NTT should be used for efficiently performing convolutions and for interpolating many points that lie on regularly-spaced grids. As a rule of thumb, one should be concerned when the number of output values being calculated is smaller than the size of the NTTs being used. In those cases, there are usually more efficient methods for calculation that often do not require NTT at all.

1.5.1 Example: Groth16

Groth16 [11] is a Zero Knowledge Proving (ZKP) system [12]. As part of the system, the prover is required to calculate the following quotient

$$Q(x) = \frac{A(x)B(x) - C(x)}{x^N - 1} \tag{1.31}$$

[8] This is not the case for Winograd FFT.

where $A(x)$, $B(x)$, and $C(x)$ are of order $N-1$, and the denominator factorizes as

$$x^N - 1 = \prod_{n=0}^{N-1} (x - \omega^n) \tag{1.32}$$

where ω is an N'th root of unity. When the prover is honest, $x^N - 1$ divides the nominator, and $Q(x)$ is a polynomial of order $N-1$. The prover begins with the following evaluation sequences $A([\omega])$, $B([\omega])$, and $C([\omega])$. Since $[\omega]$ are all roots of both the nominator and denominator, simply plugging those evaluations in (1.31) would result in $\frac{0}{0}$ for evaluations of $Q([\omega])$ which is practically useless.

We may deduce that since the nominator is of maximum degree $2N-2$, we would be required to interpolate $A(x)$, $B(x)$, and $C(x)$ to at least $2N-1$ distinct evaluations before proceeding. As it turns out, we can do a little better by recognizing that the result polynomial $Q(x)$ is of order $N-1$. This means that N distinct evaluations suffice to represent it. An efficient method to get these N evaluations is by sampling all polynomials in N non-zero locations. This sampling is achievable by algorithm (1.24) using two length-N NTTs per source polynomial. Altogether the complexity is six length-N NTTs and some linear-time processing.

Chapter 2

Practice

2.1 Introduction

NTT implementation is cumbersome. The main challenge is data movement between memories and processors. For small NTTs, whose data completely fits in on-chip random access memory, it is usually easy to build efficient solutions. The complications arise when NTT sizes are too large and data must be handled via off-chip memory such as DDR or HBM [13, 14]. To illustrate this we chose to implement a hardware accelerator for the maximum-size NTT used in the Filecoin protocol [15]. This NTT is of length 2^{27} over elements of the scalar field of BLS12-381 [16] with an appropriate root-of-unity. An element of the scalar field is 255 bits long, but we will use 256 bits or 32 Bytes for our calculations hereafter. The platform we chose to use for this implementation is Xilinx's C1100 card which houses the Ultrascale+ FPGA device VU35P [17, 18]. We chose this platform since it is readily available and houses an HBM device. As it turns out, most of the challenge is in matching the NTT's parameters to the platform's restrictions such that the trade-offs between interface, memory, and processing are optimized. Although our design choices are fairly customized to the selected operating point, we paid careful attention to architect the solution in an extendable manner allowing future migration to other operating points. The main purpose of this chapter is to describe in fair detail the architecture and design process for building a large NTT efficiently. Although this chapter was written after the fact, it attempts to describe the architecture and design process as a chronological decision-making process similar to the one that took place in practice.

2.2 Solution Landscape

Our hardware accelerator resides in a CPU-PCIe system. This means that the C1100 card is installed on a PCIe bus as a companion card in a CPU environment. NTT tasks are initiated by the CPU by streaming data from the PC memory to the FPGA. Once completed, the output is streamed back from the FPGA to the PC memory. The complete sequence can be broken down into the following stages:

1. PC streams input data to HBM

2. HBM data is read to FPGA fabric

3. Data is processed in FPGA fabric
4. Processed data is written back to HBM
5. PC streams output data from HBM

Note that steps 2 to 4 may repeat multiple times.

We can break down the above list into three distinct mechanisms that can run concurrently. Lines 1 and 5 involve PCIe transactions for moving data in and out of the accelerator. Lines 2 and 4 involve HBM transactions for moving data in and out of the processing core. Finally, line 3 involves the processing itself. The following three subsections describe limiting factors for the three mechanisms that greatly influenced our design choices, followed by a fourth subsection that provides our initial throughput analysis, given these choices.

2.2.1 PCIe to HBM

A single NTT's input size in Bytes is $2^{27} \cdot 32 = 2^{32}$. That is 4GB which is exactly half of the 8GB HBM capacity in VU35P. Assuming in-place NTT processing, this allows storing the data for two complete NTTs concurrently. The first design choice we made was to use the HBM as a double buffer. This allows processing the current NTT in one buffer while using the other buffer to stream the previous NTT out and the next NTT in. The double buffer structure allows further extensions discussed later in the chapter. Figure 2.1 illustrates the double buffer mechanism.

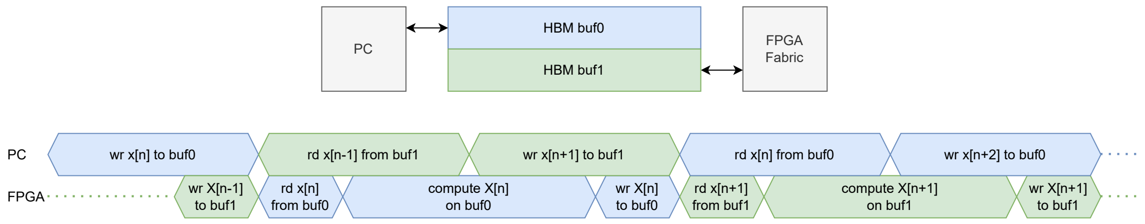


Figure 2.1: HBM Double Buffer

The choice of PCIe type (generation and number of lanes) was made to provide reasonable throughput while keeping the PCIe controller sufficiently small. This is important since for VU35P the PCIe controller is instantiated on FPGA fabric and consumes resources that can otherwise be used for processing. Our initial choice was PCIe Gen3x8 with optional future upgrades.

2.2.2 HBM to Fabric

In order to prevent the HBM interface from becoming a bottleneck, it is important to structure it to provide sufficient parallel throughput. The main trade-off is again the area occupancy of the HBM control logic in FPGA fabric. The HBM in VU35P is subdivided into 8 banks of 1GB each. Each bank is attached to a 4×4 switch. Each switch connects to its 1GB HBM bank with four 64 bits, independently addressable, full-duplex interfaces, each running at 1.8GSps. Each of the four outputs of each switch can be connected to an AXI master with configurable width. In order to match our requirements, we configured the switch's outputs to be 256 bits wide. This lowered the interface speed to 450MSps, maintaining the throughput. As we shall see, The logic required to translate the interface from 64 to 256 bits limited the total number of 256-bit AXI interfaces per switch to two, reducing the total possible throughput to half of its maximum capacity. As we shall see, this is still sufficiently high to not become a bottleneck. An additional mechanism offered in the VU35P device is an interconnecting network between the switches. This allows any AXI master to address any of the eight HBM banks, regardless of the switch it is connected to. Since the NTT access pattern is quite regular, we decided to disable this logic and only allow each master to access the 1GB bank directly attached to its switch. In fact, we further partitioned the 1GB banks into two 0.5GB banks such that each AXI master has its own distinct address space.

The resulting HBM interface is 16 AXI masters, each 256-bit wide at a maximum speed of 450MSps. going forward let us use *Word* to describe a 256-bit data word. For efficiency, we would like to read and write the HBM in whole pages only. The HBM page size is 1KB which is 32 Words, so each AXI master's 0.5GB is an address space consisting of 2^{19} pages, which is 2^{18} pages per buffer in the double-buffer scheme described before. Each AXI master is equipped with a Memory Management Unit (MMU) that is used to control the internal data-flow as described later in the chapter. Figure 2.2 illustrates the resulting HBM interface.

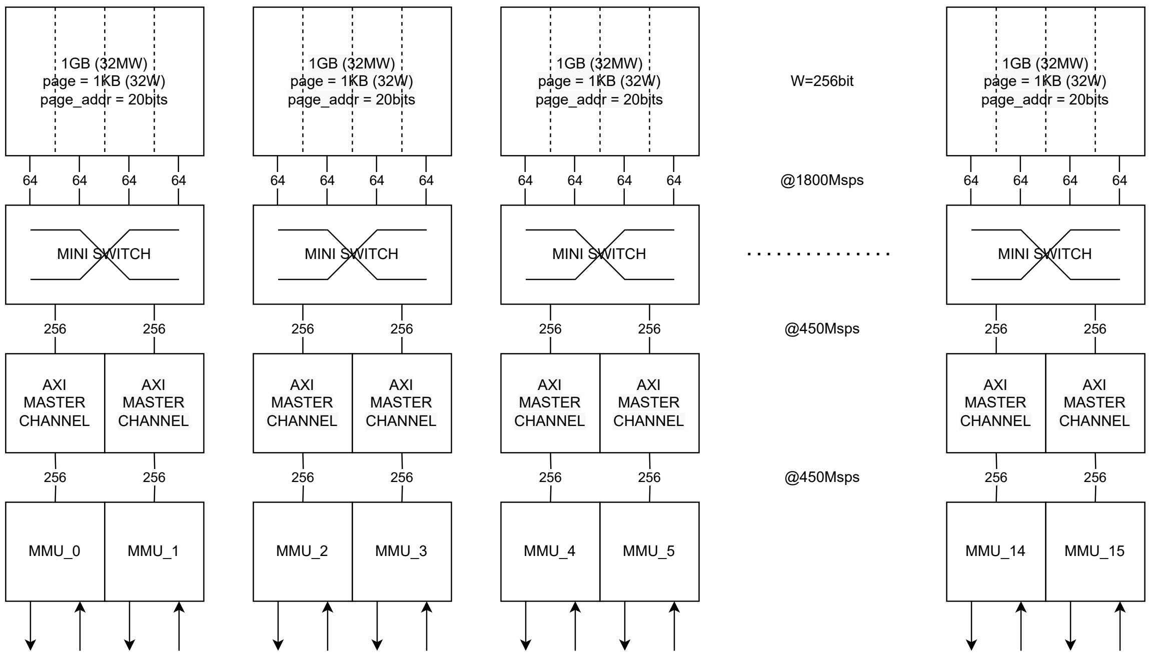


Figure 2.2: HBM Interface

Notably, the connection of the PCIe controller to the HBM is also routed via the switches. Focusing mainly on the internal interaction, our initial design elected the simplest solution of connecting the PCIe via a single AXI master interface connected to the physically closest switch. For this master, we allowed the cross-connecting feature between switches such that it can access the whole HBM space. This design choice is obviously limiting as it provides limited bandwidth for the PCIe and potentially causes congestion due to the interconnect activity. It is our intention to improve this in the future along with additional extensions for the PCIe MMU as is described later in this chapter.

2.2.3 Processing in Fabric

The core data processor is perhaps the main topic for much of this chapter. Since it significantly affects the rest of the design, its early (and accurate) selection is crucial.

Understanding that most of the processing involves 256-bit modular multiplications, we opted to estimate how many such multipliers we would be able to instantiate without pushing the design beyond the physical boundaries of the device. We chose to use our single precision Domb-Barrett multipliers [19] and estimated that for this particular field, we would be able to fit approximately 12 such multipliers. A less connected design could perhaps fit a little more but we estimated that the required data-flow, and particularly the calculation of what we will soon term *subNTT*, will restrict us to fit the main processor logic within a single FPGA SLR[1] leading to the more conservative limit.

Another aspect of selecting the processor core involves the number of round trips required for the NTT data from and to the HBM. Using the Cooley-Tukey (CT) algorithm for Fast Fourier Transform (FFT) leads to performing a number of stages (or cycles) of processing over the whole NTT data. The processing per stage is very similar across stages and involves passing the data through a processor called a *Butterfly* in equally sized chunks that completely partition the data. The size of the chunk is usually called the radix of the butterfly. An FFT can be performed such that all stages use the same radix or by utilizing different radixes in which case it is termed mixed-radix. The number of stages is directly determined by the radix (or radixes). For a single-radix design, the number of stages is the logarithm of the NTT size with the radix used as the base. As such, a larger radix results in fewer stages, thus fewer round trips from and to the HBM. Another advantage of a larger radix is the ability to use the Winograd FFT algorithm, potentially saving multipliers [8].

The downside of a large radix is that it complicates the element ordering and transposition requirements for FFT processing much more than for the ordinary radix-2, as we shall see later in this chapter.

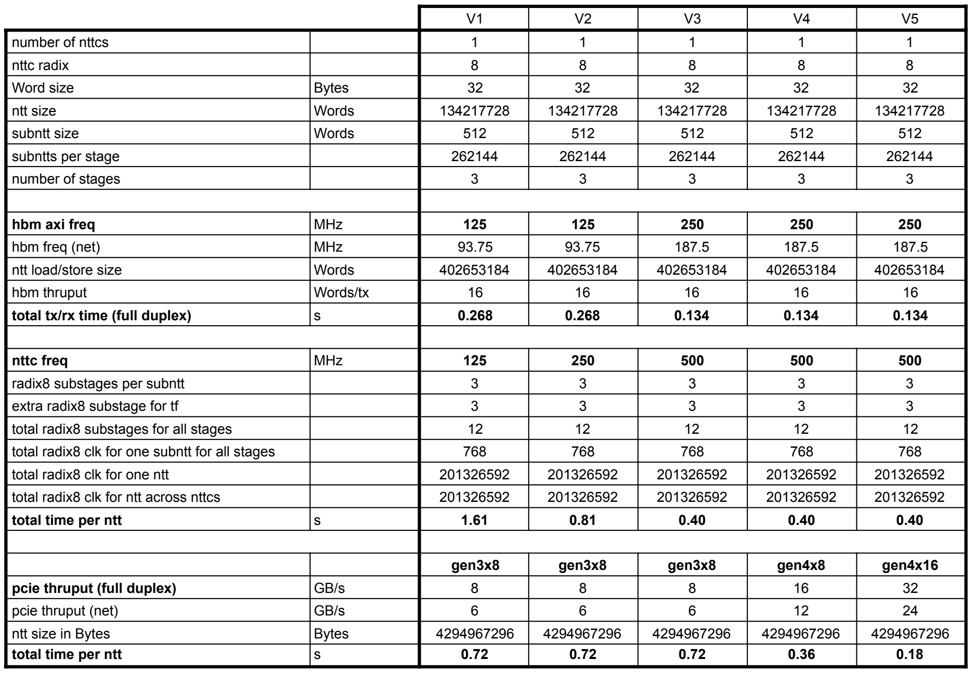
After careful consideration, we opted to work with a radix-8 core. As we shall see, the cost of this core in the Winograd implementation is eight full multipliers and four constant multipliers. Our analysis shows that it should fit into a single SLR and enable constructing our subNTT on top of it without over-complication. Other alternatives that were considered and discarded were parallel radix-2 engines and parallel radix-4 engines. A single radix-16 engine was very suitable but unfortunately not feasible due to its size. The smaller radix options proved to not match well with the 16 HBM MMU design or over-complicated the transposition logic.

The simple way to work with 16 HBM MMUs and radix-8 cores is to split the data between the left 8 and right 8 MMUs and have a radix-8 core per side. Since only a single radix-8 core is available we decided to double-clock it to serve both HBM sides.

[1]VU35P has two SLRs and SLR crossings across wide data-flow interfaces greatly limits the maximum working frequency.

2.2.4 Throughput Estimation

The table in Figure 2.3 shows the throughput analysis for different versions of the NTT accelerator. The difference between the versions is the operating frequencies of various parts in the design. Note how the PCIe interface and fabric processing dominate the total time, while the HBM is less significant. Although this was not a prior consideration, this is very desirable from a power dissipation perspective.



		V1	V2	V3	V4	V5
number of nttcs		1	1	1	1	1
nttc radix		8	8	8	8	8
Word size	Bytes	32	32	32	32	32
ntt size	Words	134217728	134217728	134217728	134217728	134217728
subntt size	Words	512	512	512	512	512
subntts per stage		262144	262144	262144	262144	262144
number of stages		3	3	3	3	3
hbm axi freq	MHz	**125**	**125**	**250**	**250**	**250**
hbm freq (net)	MHz	93.75	93.75	187.5	187.5	187.5
ntt load/store size	Words	402653184	402653184	402653184	402653184	402653184
hbm thruput	Words/tx	16	16	16	16	16
total tx/rx time (full duplex)	s	**0.268**	**0.268**	**0.134**	**0.134**	**0.134**
nttc freq	MHz	**125**	**250**	**500**	**500**	**500**
radix8 substages per subntt		3	3	3	3	3
extra radix8 substage for tf		3	3	3	3	3
total radix8 substages for all stages		12	12	12	12	12
total radix8 clk for one subntt for all stages		768	768	768	768	768
total radix8 clk for one ntt		201326592	201326592	201326592	201326592	201326592
total radix8 clk for ntt across nttcs		201326592	201326592	201326592	201326592	201326592
total time per ntt	s	**1.61**	**0.81**	**0.40**	**0.40**	**0.40**
		gen3x8	**gen3x8**	**gen3x8**	**gen4x8**	**gen4x16**
pcie thruput (full duplex)	GB/s	8	8	8	16	32
pcie thruput (net)	GB/s	6	6	6	12	24
ntt size in Bytes	Bytes	4294967296	4294967296	4294967296	4294967296	4294967296
total time per ntt	s	**0.72**	**0.72**	**0.72**	**0.36**	**0.18**

Figure 2.3: Throughput Estimation

2.3 Solution Scheme

One way to illustrate the single-radix CT scheme is by placing the NTT data on a hypercube[2]. The dimension and edge-length of the hypercube are the number of required stages and the radix, respectively. For the simple case of two stages, the reader is referred back to Section 1.4. In this case, we decided to split the 2^{27} into three stages of radix-512, where each 512 NTT is termed *subNTT*. The scheme follows the illustration in Figure 2.4.

We initially organize the data over a three-dimensional cube. The data organization can be done in one of two arrangements (more on this soon). At the first stage, we perform

[2]Or hyper-rectangle for mixed-radix schemes.

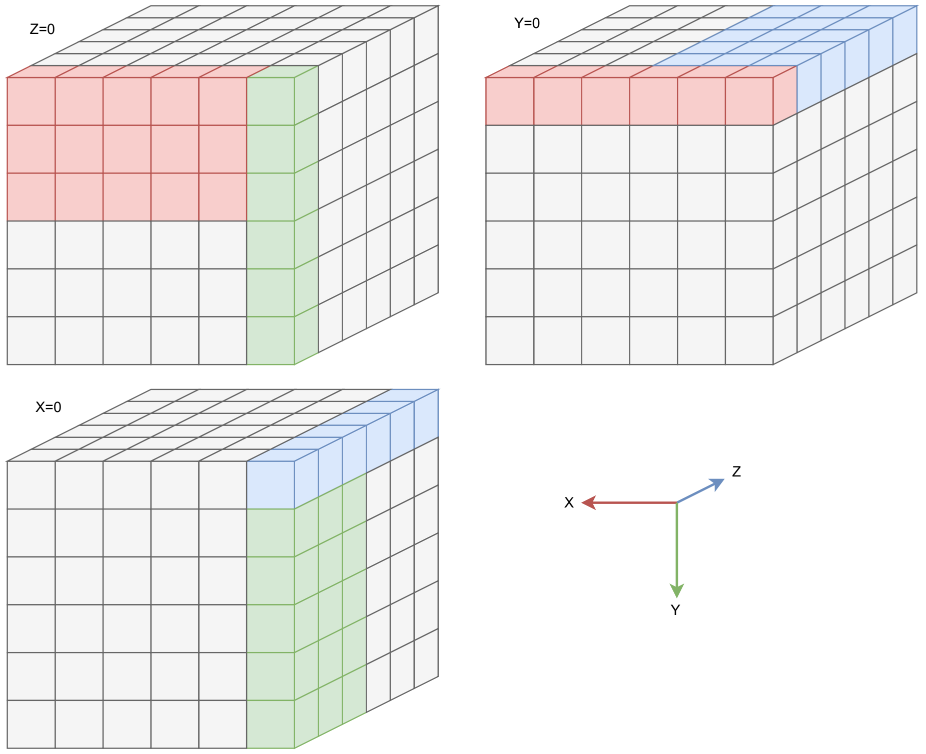


Figure 2.4: CT FFT Cube Scheme

all subNTTs along the x-axis, then at the second stage along the y-axis, and finally at the third stage along the z-axis. Between stages, we multiply by the appropriate twiddle-factors. Note that from a data-flow perspective, switching between two axes, as is done between stages, is equivalent to a transpose operation between those two axes.

Figure 2.5 is a visualization of the same scheme from a slightly different perspective. In this visualization, we tried to number the samples in the same way as they appear in memory for the in-place implementation. The ordering is quite cumbersome but one important thing to notice is how the order of subNTTs at the output is strided relative to its sequential order at the input. The striding is by a factor of 512 and is actually equivalent to reversing the bit order of the 9 least significant bits in the subNTT 18-bit indexes. This is a characteristic of the in-place CT FFT. The subNTT level ordering is either strided at the input for DIF or output for DIT. Note that this is independent of the ordering of inputs and outputs of the subNTT itself.

We noted, through experimentation, that the simplest way (for us) to systematically map the transposed process, without requiring bit-reverse ordering of the data at every stage was to keep the data reverse-bit-ordered throughout the whole process. The outcome of that was that we constructed every NTT level, including radix-8, subNTT, and the full NTT to be reverse-bit-ordered at the input and output. The only outcome is that it required calculating the twiddle-factors in a reverse-bit-ordered fashion as $\omega^{\mathrm{rbo}(\rho)\mathrm{rbo}(l)}$ rather than $\omega^{\rho l}$. In conclusion, the input to the NTT accelerator must be reverse-bit-ordered by the PC prior to streaming to the device. When received back it must be subNTT block reordered to remove the striding and then reverse-bit-ordered. Clearly, all reordering can be done

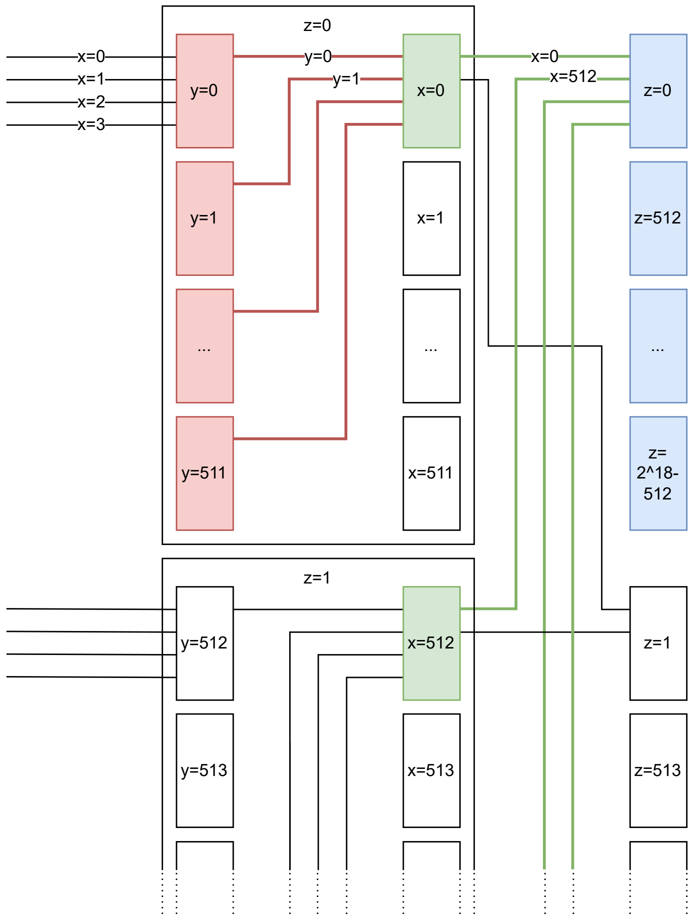


Figure 2.5: CT FFT Butterfly Scheme

in $\mathcal{O}(n)$. Moving some of the reordering to hardware is a topic that is currently being explored by us.

2.4 Architecture

We mentioned in a previous section that the number of MMUs is 16 while the internal processor is a single radix-8 machine (i.e. it has eight inputs). As mentioned there, the way to make this work is by time-sharing the radix-8 processor between the left and right MMUs. To simplify the presentation in this section, let us ignore the time-sharing and pretend that we have two radix-8 processors.

Let us begin by presenting the top-level architecture (see Figure 2.6). The presentation uses a data grouping terminology that was not yet defined but will be clarified as the section proceeds. The design is split into four subsystems, as follows:

- SHELL - Consists of the PCIe (and DMA) interface to the PC and the HBM AXI interfaces to the fabric.

- HBM-SS - Consists of the HBM MMUs and the super-program state-machine.

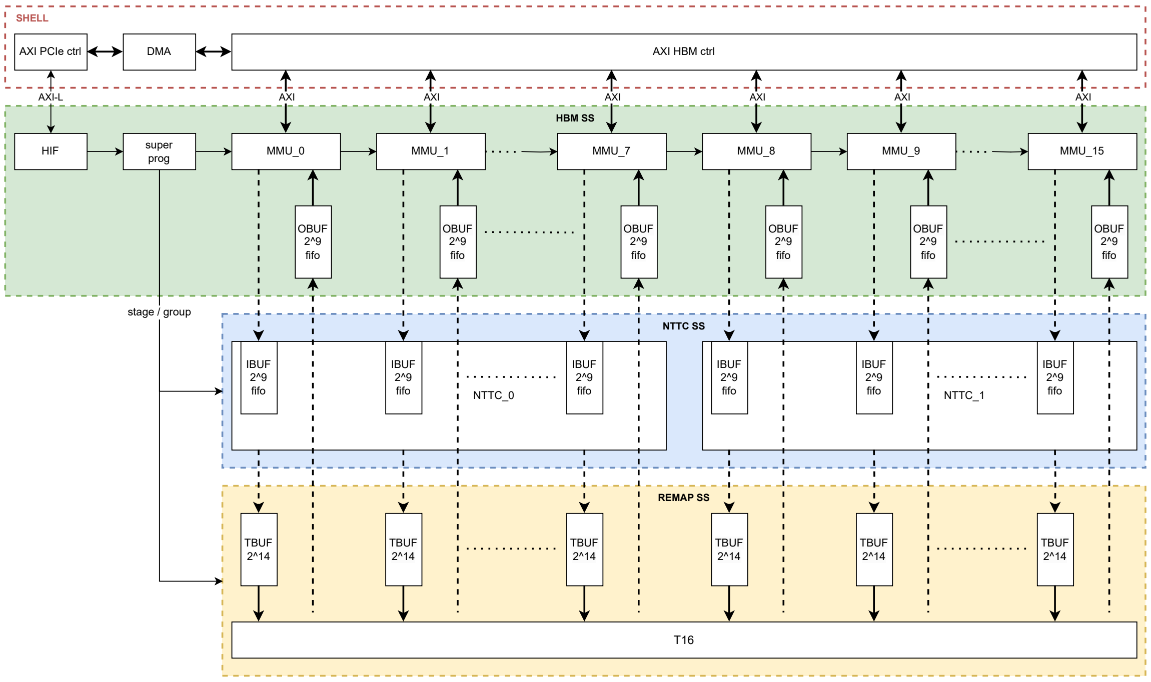


Figure 2.6: Top Level Architecture

- NTTC-SS - Consists of the logic required to compute a *Batch* of subNTTs, including multiplication by external twiddle-factors required for the full NTT.
- REMAP-SS - Consists of the logic required to transpose a *Slice* of subNTTs.

The design is controlled by the super-program state-machine that essentially manages the data-flow and processing across all subsystems. The general flow of the super-program, as depicted in Figure 2.7, is as follows:

1. Once data is available in HBM, the HBM-SS commences the super-program.
2. MMUs move a Batch of data from HBM to IBUFs (a.k.a. input buffers). A Batch is 16 subNTTs, sub-grouped according to even and odd indices. The eight even indexed subNTTs are at the left MMUs (i.e. 0 to 7) and the eight odd subNTTs at the right MMUs (i.e. 8 to 15). The data is written to IBUFs sequentially.
3. The two NTTCs within NTTC-SS process the data Batch, reading from IBUFs and writing to TBUFs (a.k.a. transpose buffers). The data is read from IBUFs and written to TBUFs sequentially.
4. Once TBUFs contain at least one Slice, which is 16 Batches, T16 within the REMAP-SS transposes the data in TBUFs and writes the transposed data to OBUFs (a.k.a. output buffers). The data is written to OBUFs sequentially. The data to an OBUF has to be in a minimal quant of an HBM page (32 Words). This assures that the HBM transaction overheads are minimal. Slice is the minimum amount of data required for transposing into whole pages at the output.
5. MMUs write pages of data from OBUFs back to HBM.

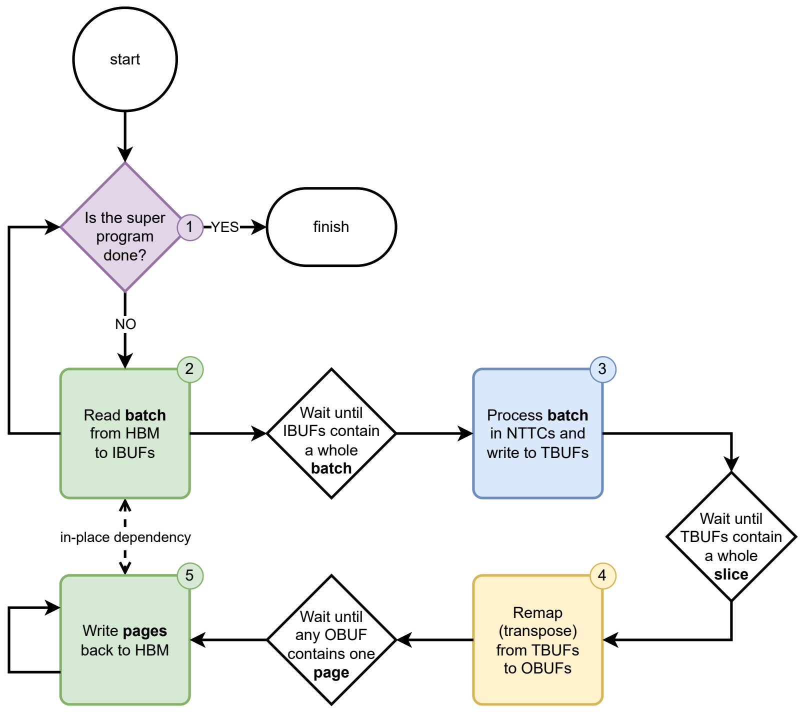

start
Is the super program done?
1
YES
finish
NO
2
Read **batch** from HBM to IBUFs
Wait until IBUFs contain a whole **batch**
3
Process **batch** in NTTCs and write to TBUFs
Wait until TBUFs contain a whole **slice**
in-place dependency
5
Write **pages** back to HBM
Wait until any OBUF contains one **page**
4
Remap (transpose) from TBUFs to OBUFs

Figure 2.7: Super Program State Machine

6. This process repeats for all three stages and all Batches per stage. The subsystems' functional configuration changes across stages and Batches.

For all intents and purposes, a single instruction of the super-program is used to process a *Group* which amounts to two consecutive Slices. This minimizes internal stalls that are not compute-induced. A complete 2^{27} NTT will run a super-program going through all stages and all Groups in order, but running individual Groups in random order or any sub-programs or combinations thereof is possible. This is often helpful for debugging purposes.

In order to keep the design as modular as possible without sacrificing increased delays due to a lack of synchronization between subsystems, the design data-flow is controlled using back-pressure techniques. Once a super-program has commenced, the data will continuously flow through the design until the program is complete. Each block will digest incoming data or stall it if it is currently busy. This assures no unnecessary stalls. The back-pressure scheme is portrayed in Figure 2.8.

The next subsections outline the data organization and architectural details for the main functions in the design.

2.4.1 Data Organization

The data is organized in the HBM in the order in which it is read into the design. The general organization is depicted in Figure 2.9.

In essence, the HBM is subdivided into two rows and two columns. The two rows account for the HBM double buffer and the two columns account for the left and right NTTC sides. The NTT data (corresponding to a single buffer) consists of 512 Groups,

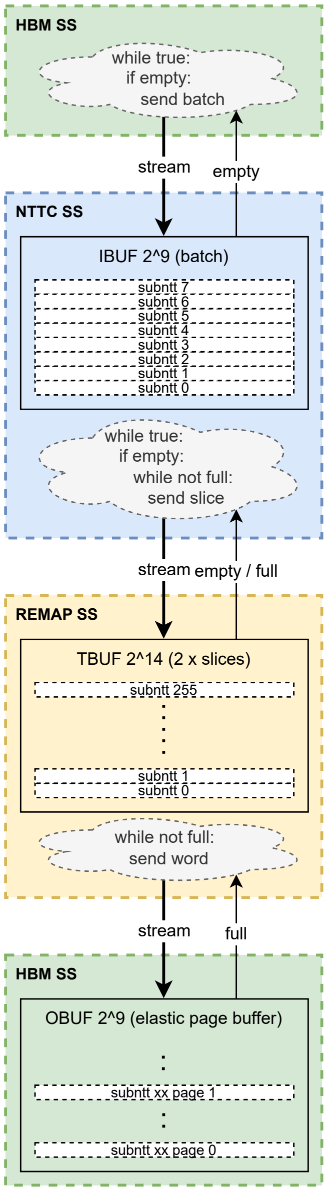


Figure 2.8: Back-Pressured Data-Flow Design

each Group consisting of two Slices, each Slice consisting of 16 Batches, and each Batch consisting of 16 subNTTs. The even indexed subNTTs in a Batch are placed on the left NTTC side and the odd on the right. The lower part of Figure 2.9 shows how a single subNTT of 512 Words is distributed across the left eight MMUs. Note that this results in two pages per MMU (out of 8 MMUs of a particular NTTC side) for storage of each subNTT.

An alternative visualization of the data organization is presented in Figure 2.10 where the data is shown in a pyramid structure for a single complete NTT. The left pyramid in the figure presents the organization from the subNTT perspective while the right pyramid presents it from the page perspective.

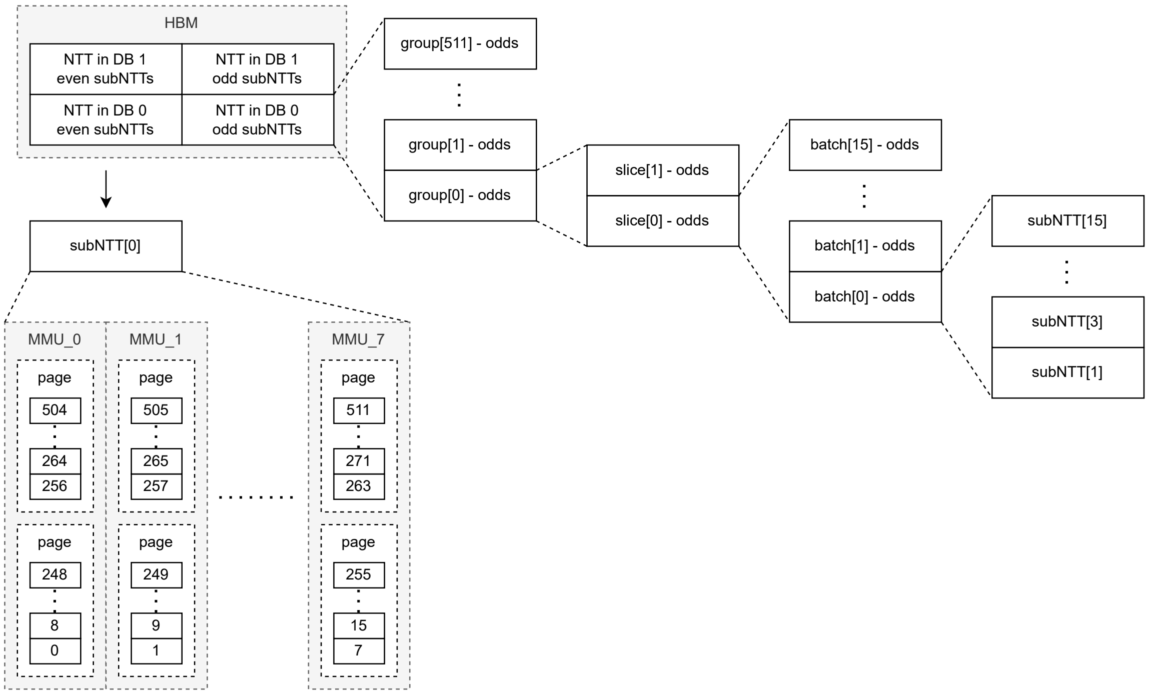


Figure 2.9: Data Organization in HBM

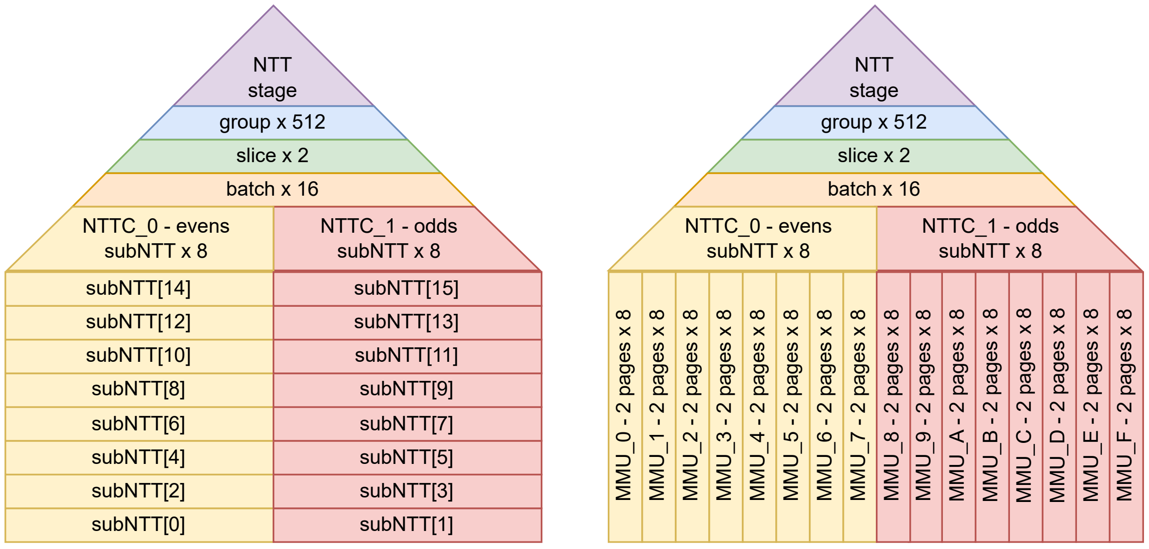
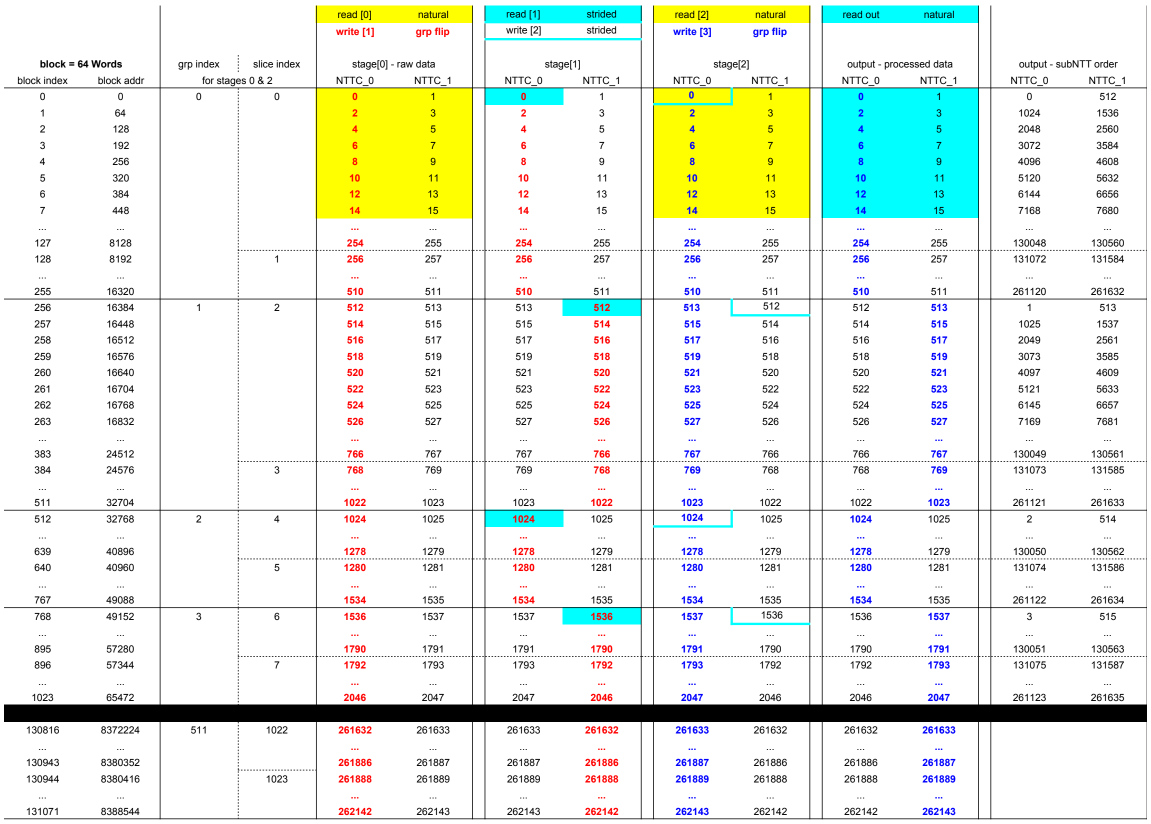


Figure 2.10: Data Organization in HBM - subNTT (left) vs. pages (right) perspectives

2.4.2 MMU (HBM-SS)

The MMUs are primarily in charge of selecting the groups flowing from HBM to the processor and back to HBM. If we refer back to Figure 2.5, the MMUs are in charge of selecting the red group at stage 0, the green group at stage 1, and the blue group at stage 2. Per stage, the groups are selected according to the transpose input requirements of T16. It is clear from the figure that T16 performs the FFT wiring between stages (i.e. green wiring between stages 0 and 1 and red wiring between 1 and 2). The table in Figure 2.11 provides a more detailed view of the MMUs read/write ordering.



				read [0]	natural	read [1]	strided	read [2]	natural	read out	natural		
				write [1]	**grp flip**	write [2]	strided	**write [3]**	**grp flip**				
block = 64 Words		grp index	slice index	stage[0] - raw data		stage[1]		stage[2]		output - processed data		output - subNTT order	
block index	block addr	for stages 0 & 2		NTTC_0	NTTC_1	NTTC_0	NTTC_1	NTTC_0	NTTC_1	NTTC_0	NTTC_1	NTTC_0	NTTC_1
0	0	0	0	**0**	1	**0**	1	**0**	1	**0**	1	0	512
1	64			**2**	3	**2**	3	**2**	3	**2**	3	1024	1536
2	128			**4**	5	**4**	5	**4**	5	**4**	5	2048	2560
3	192			**6**	7	**6**	7	**6**	7	**6**	7	3072	3584
4	256			**8**	9	**8**	9	**8**	9	**8**	9	4096	4608
5	320			**10**	11	**10**	11	**10**	11	**10**	11	5120	5632
6	384			**12**	13	**12**	13	**12**	13	**12**	13	6144	6656
7	448			**14**	15	**14**	15	**14**	15	**14**	15	7168	7680
...	...			**...**	...	**...**	...	**...**	...	**...**	...	...	...
127	8128			**254**	255	**254**	255	**254**	255	**254**	255	130048	130560
128	8192		1	**256**	257	**256**	257	**256**	257	**256**	257	131072	131584
...	...			**...**	...	**...**	...	**...**	...	**...**	...	...	...
255	16320			**510**	511	**510**	511	**510**	511	**510**	511	261120	261632
256	16384	1	2	**512**	513	513	**512**	**513**	512	512	**513**	1	513
257	16448			**514**	515	515	**514**	**515**	514	514	**515**	1025	1537
258	16512			**516**	517	517	**516**	**517**	516	516	**517**	2049	2561
259	16576			**518**	519	519	**518**	**519**	518	518	**519**	3073	3585
260	16640			**520**	521	521	**520**	**521**	520	520	**521**	4097	4609
261	16704			**522**	523	523	**522**	**523**	522	522	**523**	5121	5633
262	16768			**524**	525	525	**524**	**525**	524	524	**525**	6145	6657
263	16832			**526**	527	527	**526**	**527**	526	526	**527**	7169	7681
...	...			**...**	...	...	**...**	**...**	...	...	**...**	...	...
383	24512			**766**	767	767	**766**	**767**	766	766	**767**	130049	130561
384	24576		3	**768**	769	769	**768**	**769**	768	768	**769**	131073	131585
...	...			**...**	...	...	**...**	**...**	...	...	**...**	...	...
511	32704			**1022**	1023	1023	**1022**	**1023**	1022	1022	**1023**	261121	261633
512	32768	2	4	**1024**	1025	**1024**	1025	**1024**	1025	**1024**	1025	2	514
...	...			**...**	...	**...**	...	**...**	...	**...**	...	...	...
639	40896			**1278**	1279	**1278**	1279	**1278**	1279	**1278**	1279	130050	130562
640	40960		5	**1280**	1281	**1280**	1281	**1280**	1281	**1280**	1281	131074	131586
...	...			**...**	...	**...**	...	**...**	...	**...**	...	...	...
767	49088			**1534**	1535	**1534**	1535	**1534**	1535	**1534**	1535	261122	261634
768	49152	3	6	**1536**	1537	1537	**1536**	**1537**	1536	1536	**1537**	3	515
...	...			**...**	...	...	**...**	**...**	...	...	**...**	...	...
895	57280			**1790**	1791	1791	**1790**	**1791**	1790	1790	**1791**	130051	130563
896	57344		7	**1792**	1793	1793	**1792**	**1793**	1792	1792	**1793**	131075	131587
...	...			**...**	...	...	**...**	**...**	...	...	**...**	...	...
1023	65472			**2046**	2047	2047	**2046**	**2047**	2046	2046	**2047**	261123	261635
130816	8372224	511	1022	**261632**	261633	261633	**261632**	**261633**	261632	261632	**261633**		
...	...			**...**	...	...	**...**	**...**	...	...	**...**		
130943	8380352			**261886**	261887	261887	**261886**	**261887**	261886	261886	**261887**		
130944	8380416		1023	**261888**	261889	261889	**261888**	**261889**	261888	261888	**261889**		
...	...			**...**	...	...	**...**	**...**	...	...	**...**		
131071	8388544			**262142**	262143	262143	**262142**	**262143**	262142	262142	**262143**		

Figure 2.11: MMUs Data-Flow

The table presents the order of subNTTs on and between stages for the whole super-program. The table is at the subNTT granularity and distinguishes between the left and right NTTC sides. It doesn't however detail individual MMU lanes. Note that the right-most column in the table is the subNTT order resulting in the output, where the second to last column is used to index the data movement internally.

An interesting outcome of the wiring (transposing) is that the reading order between stages 1 and 2 requires Groups that are constructed from all even or all odd subNTTs. For instance, the first Group consists of subNTTs $\{0, 512, 1024, \ldots\}$. By default, since they are even, these subNTTs are all on the left NTTC side. Doing nothing would result in a 50% utilization for stage 1 processing since only a single side can be used concurrently. To prevent this from happening, we group flipped the data between the left and right sides for odd Groups, as is marked by bold numbers in the table. The hardware implication of this is insignificant and it is easily handled by T16.

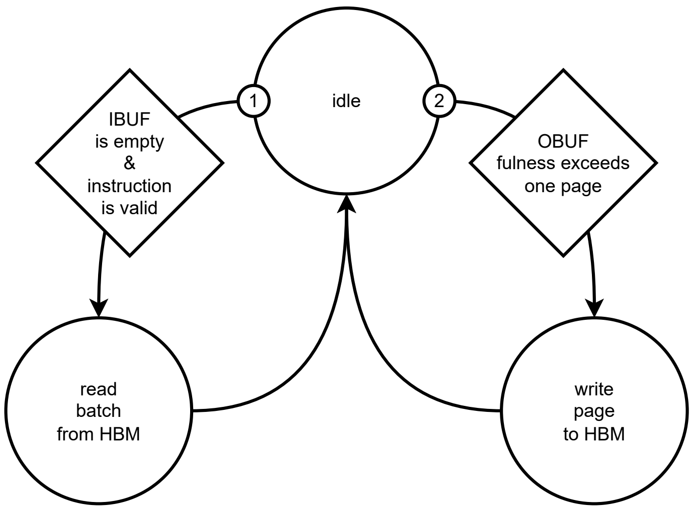

idle
1
2
IBUF is empty & instruction is valid
OBUF fulness exceeds one page
read batch from HBM
write page to HBM

Figure 2.12: MMU State-Machine

2.4.3 T16 (REMAP-SS)

The purpose of the T16 module is to transpose the data in accordance with the CT FFT scheme from Figure 2.5. Achieving this along with the constraint of providing whole HBM pages at the output OBUFs leads to the minimal required input size of a Slice. The reason for requiring that T16 be able to handle two consecutive Slices (i.e. one Group) is due to the in-place requirements of the data being written back to the HBM. As it happens, the first Slice always transposes to page 0 of all output subNTTs in the Group, and the second Slice to page 1. This means that writing back the first Slice's output before reading the second Slice from the HBM would lead to writing before reading, a violation of the in-place assumption, and corruption of the data.

Since the storage requirements for T16's TBUFs are already high (i.e. a whole Group), the challenge is to suffice with that and read directly from there to the OBUFs (i.e. without requiring additional intermediate memory). The difficulty is that this requires storing data from all 16 lanes in their arrival order and organizing it into 16 TBUFs such that the relevant data to be read from the 16 TBUFs to the OBUFs is all exposed (i.e. that all required samples for the next read are stored on mutually exclusive TBUFs). As it happens, this is possible but requires input and output staggering of the data within the TBUFs. The three columns of the table in Figure 2.13 show the organization of the input data, the data as it is stored on the TBUFs, and the output data. To understand the table, examine the first sample, marked 0, of the first four subNTTs on the left and the first four subNTTs on the right, in the input column. Clearly, making a single output row from them is not possible in their input arrangement, since each four occupy a single MMU lane. To enable this, we use a rotating mux to stagger the data as it enters the TBUFs, as can be seen in the middle column. Finally, when reading, all 0 marked samples can be read concurrently by simply manipulating the TBUFs addresses since they are already present on different lanes.

Other functions performed by T16 are the handling of data-flipping between the left and right NTTC sides to expose the required subNTTs for stage 1, as discussed previously, and a bypass function that allows skipping the transpose function at the final stage.

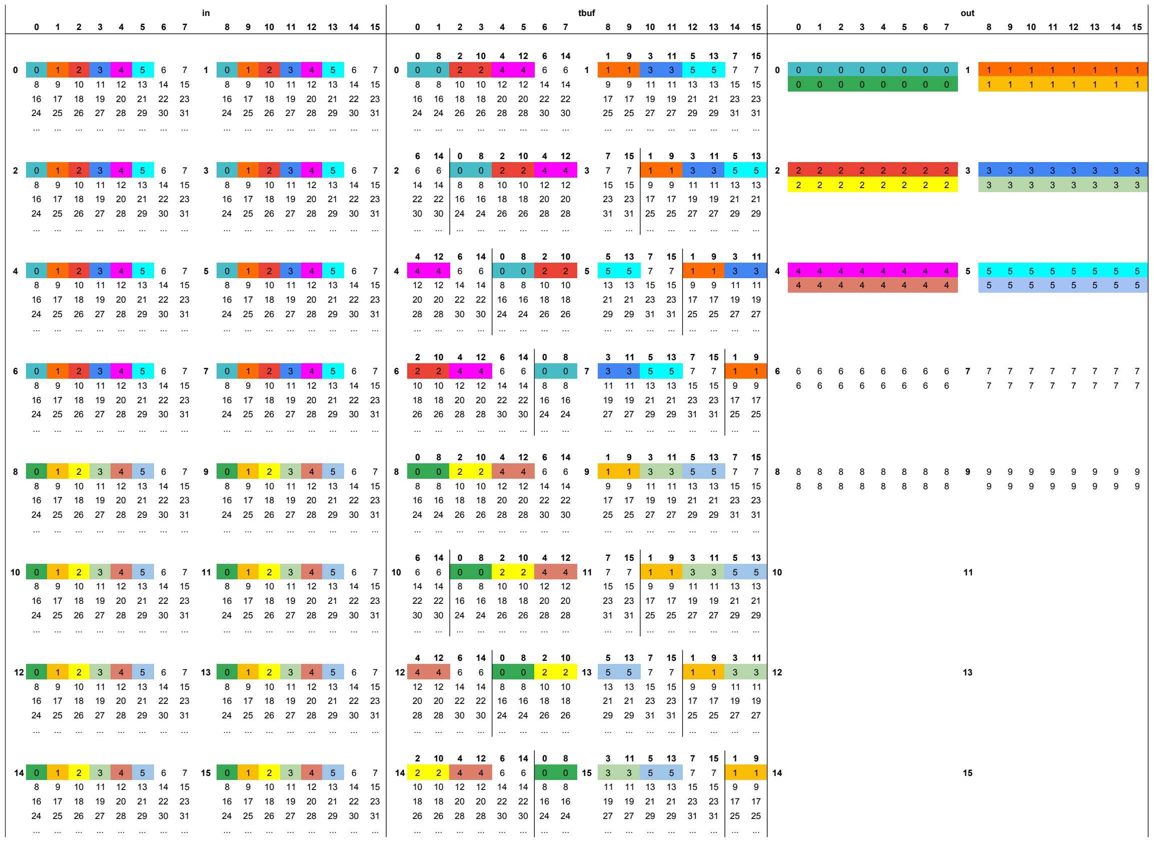


Figure 2.13: T16 Data Manipulation

2.4.4 NTTC (NTTC-SS)

NTTC is the core processor of the NTT design. In its heart, it relies on a radix-8 processor. Figure 2.14 illustrates the CT radix-8 implementation. This implementation requires three radix-2 stages consisting of 12 butterflies and five non-trivial internal twiddle-factors. Note that the internal twiddle-factors are constant. The external twiddle-factors, used to construct larger NTTs based on this radix-8 core are handled via the input multipliers m0 to m7.

To optimize the implementation we chose a Winograd-type implementation for the circuit based on [20]. This function-equivalent circuit is presented in Figure 2.15. Besides requiring only four non-trivial internal twiddle-factors, this implementation has a shorter critical path and is significantly more delay-balanced.

For maximum flexibility in constructing larger NTTs, we needed to understand how to use the radix-8 core in three additional operating points, radix-4, radix-2, and multiply-only. The prior two are necessary in order to construct NTTs whose length is not a power of eight[3]. The multiply-only option is used when external twiddle-factors, that are unavailable in memory, need to be constructed from the existing tables using multiplication (more on this soon). Radix-2 and multiply-only can be easily supported by exiting the pipeline

[3]Our assumption is that all lengths of interest are a power-of-two.

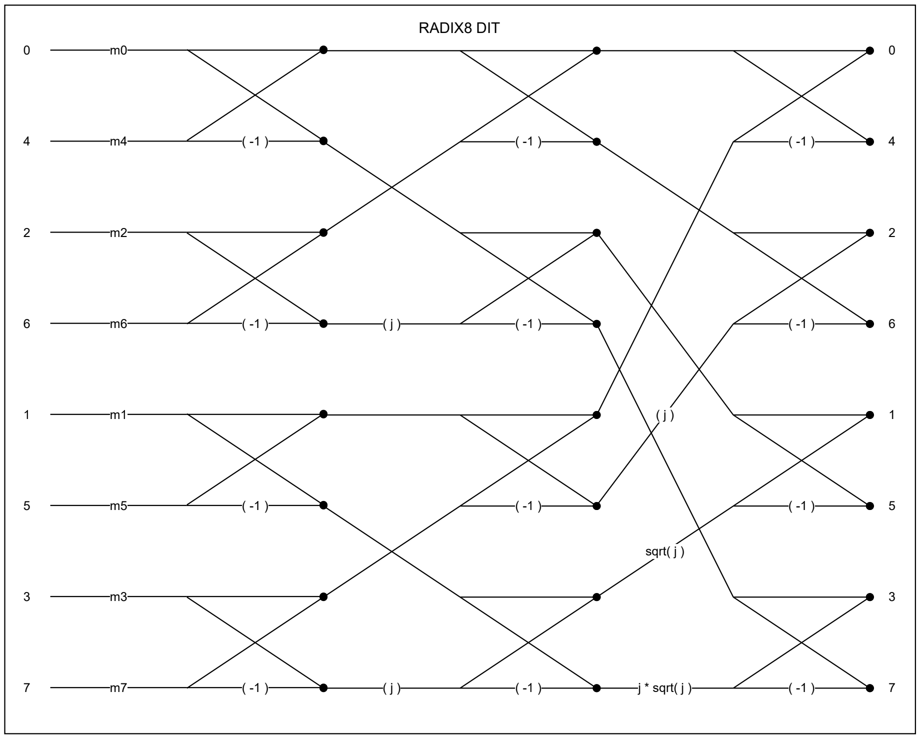


Figure 2.14: Radix-8 Winograd

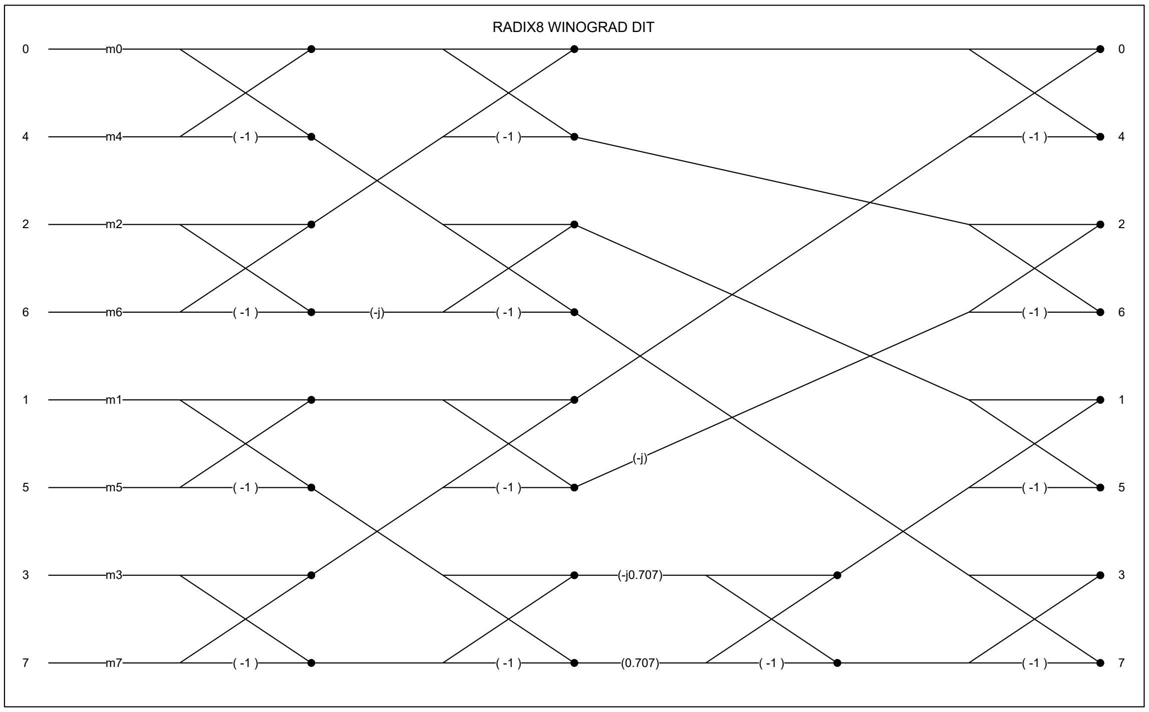


Figure 2.15: Radix-8 Winograd

prematurely. However, in the Winograd implementation, radix-4 does not seem to have a convenient exiting point[4]. To mend this we manipulated the Winograd circuit and came up with the alternative one as presented in Figure 2.16.

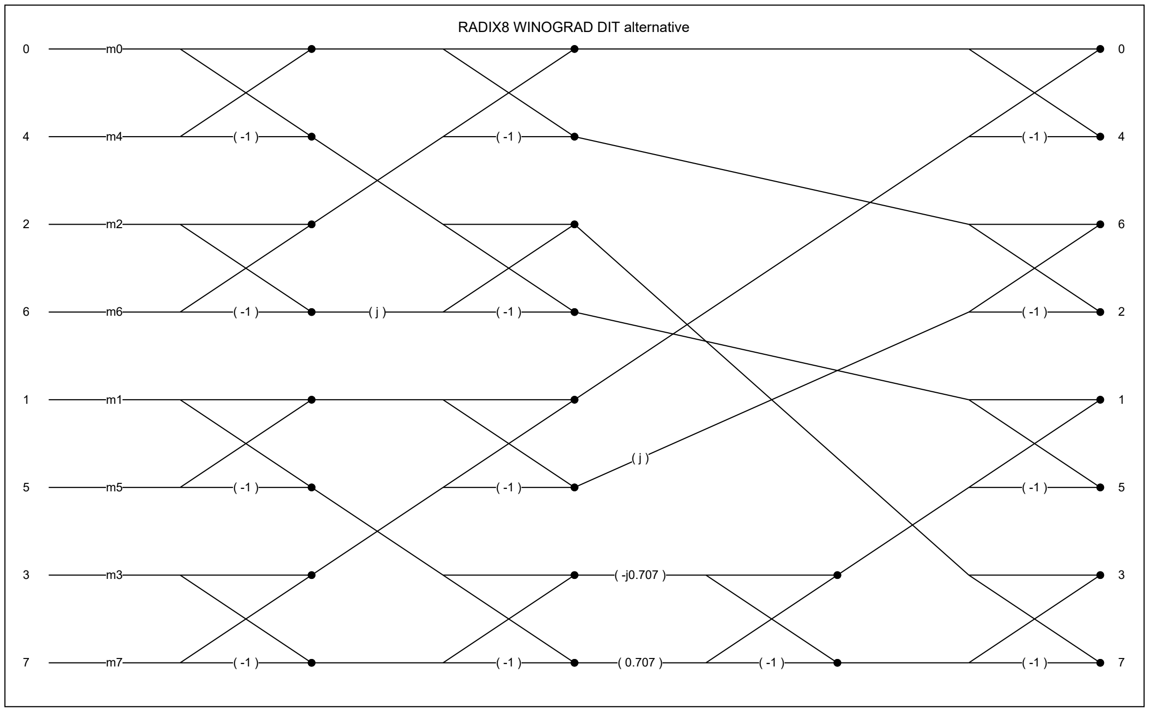


Figure 2.16: Radix-8 Winograd Alternative

The main difference is the exchange of multiplication by $-j$ for multiplication by j and other small rewiring changes required to maintain the original functionality. Note how the new circuit has one radix-4 available trivially at the top half and a second radix-4 available by multiplexing smartly to use the second j multiplier. The resulting fully-pipelined hardware circuit is depicted in Figure 2.17. Note that the circuit delay is constant for all modes.

The NTTC core is constructed around the radix-8 processor to perform subNTT processing over a Batch of subNTTs. Since NTTC works on a single side (left or right MMUs), the number of subNTTs it processes in one Batch is eight. The processing follows a CT scheme, in a similar manner to the full NTT scheme, but with out-of-place memory. We chose to allocate double the space for the intermediate NTTC memory, SBUF, in order to simplify the design and easily enable a zero-stall machine. Additionally, processing a Batch of subNTTs allowed the loosening of the CT dependencies between stages and permitted working with long latencies for the radix-8 engine. Figure 2.18 illustrates the general structure of the NTTC engine along with the input and output muxes for operating it for both the left and right NTTC sides. Note that the NTTC includes its own transpose-8 module T8 at the input to the SBUFs. The T8 module is responsible for the data transposition between stages and is built to support all previously discussed modes supported by the

[4]Note that we would like to support the following modes: a single radix-8, two parallel radix-4, or four parallel radix-2

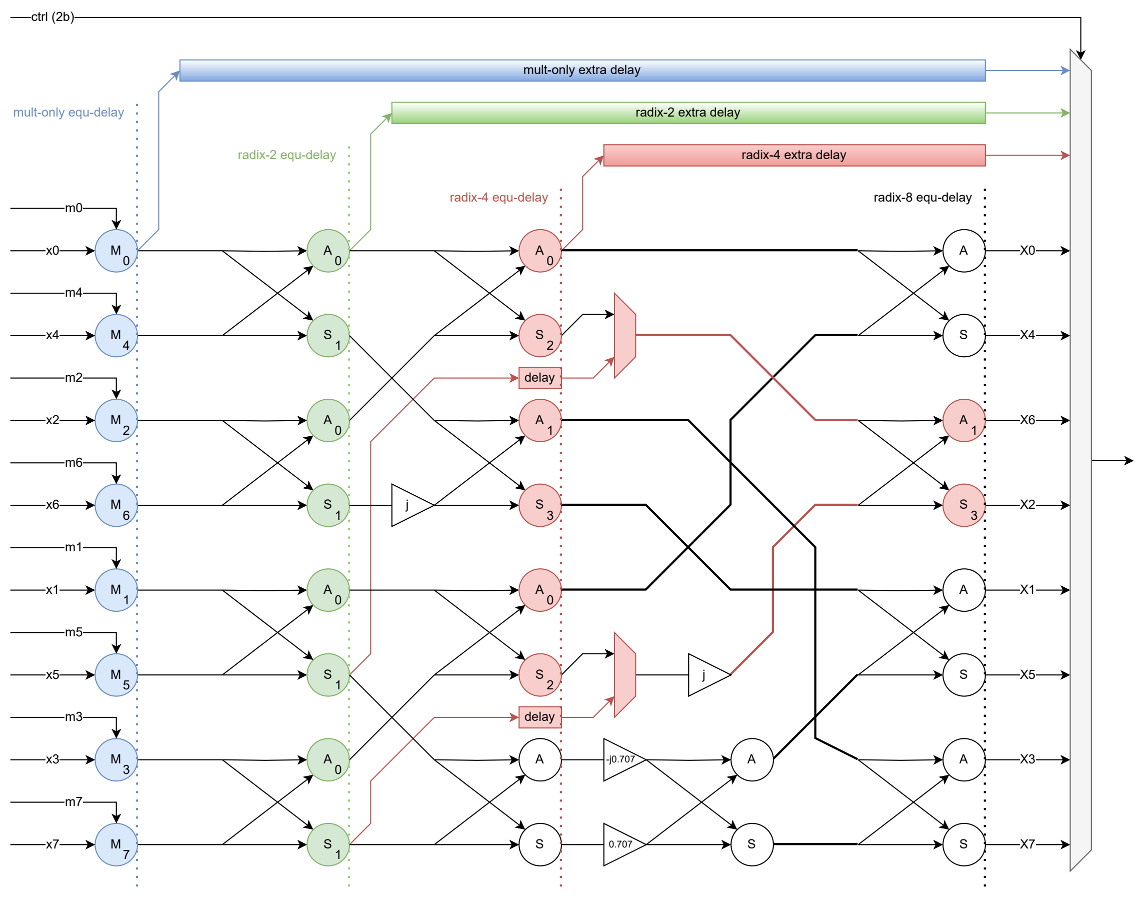


Figure 2.17: Radix-8 Circuit

radix-8 processor.

Supporting all required external twiddle-factors was achieved by a three-level memory structure. The smallest required root-of-unity is of order 2^{27}. Storing all precalculated powers of this root-of-unity is wasteful and practically impossible. Simple analysis shows that the NTTC needs a table of 512 powers of the 512th root-of-unity powers for its subNTT processing. For the construction of the larger NTT, it requires the powers of roots-of-unity of orders 2^{18} and 2^{27} for stages 1 and 2, respectively. The required powers can be constructed from smaller tables by simple manipulation. Assume ω is a 2^{27} root-of-unity and we need the twiddle-factor $\omega^{\rho l}$ in stage 2, where $\rho \in [0, 2^{18})$ and $l \in [0, 2^9)$ (see Section 1.4 for more detail). We can multiply the two integers $r = \rho l$ resulting in a 27-bit integer that can be represented by three 9-bit digits $r = a2^{18} + b2^9 + c$. The required twiddle-factor can thus be represented as $\omega^r = (\omega^{2^{18}})^a \cdot (\omega^{2^9})^b \cdot \omega^c$. It is easy to see that $\omega^{2^{18}}$, ω^{2^9} are the roots-of-unity of orders 2^9 and 2^{18}, respectively, and since a, b, and c are 9-bit digits, all required twiddle-factors can be constructed using three 512-entry look-up tables. An illustration of the NTTC, detailing its memory structure is presented in Figure 2.19

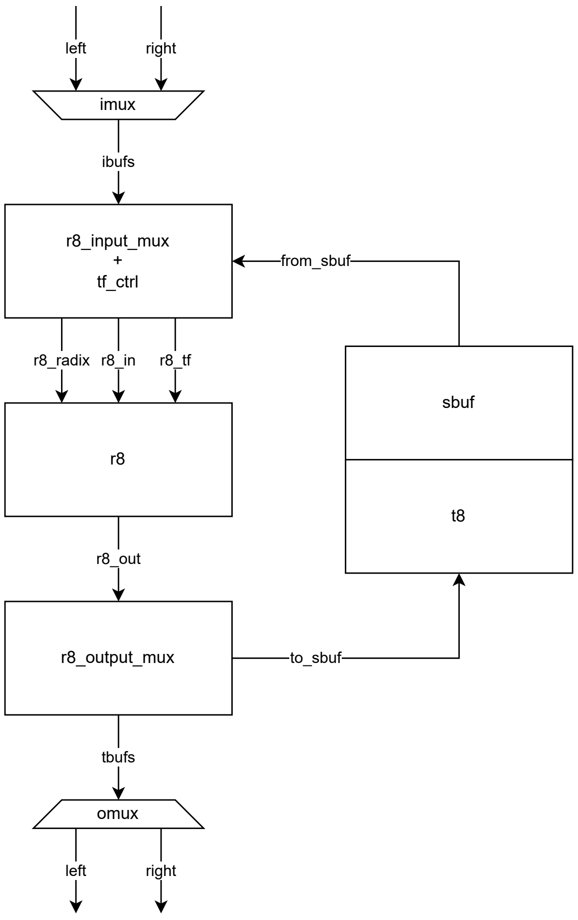


Figure 2.18: NTTC Architecture

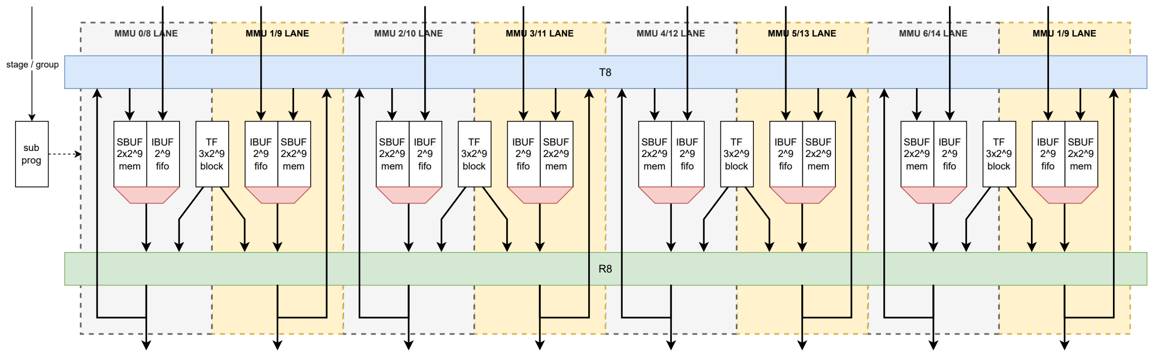
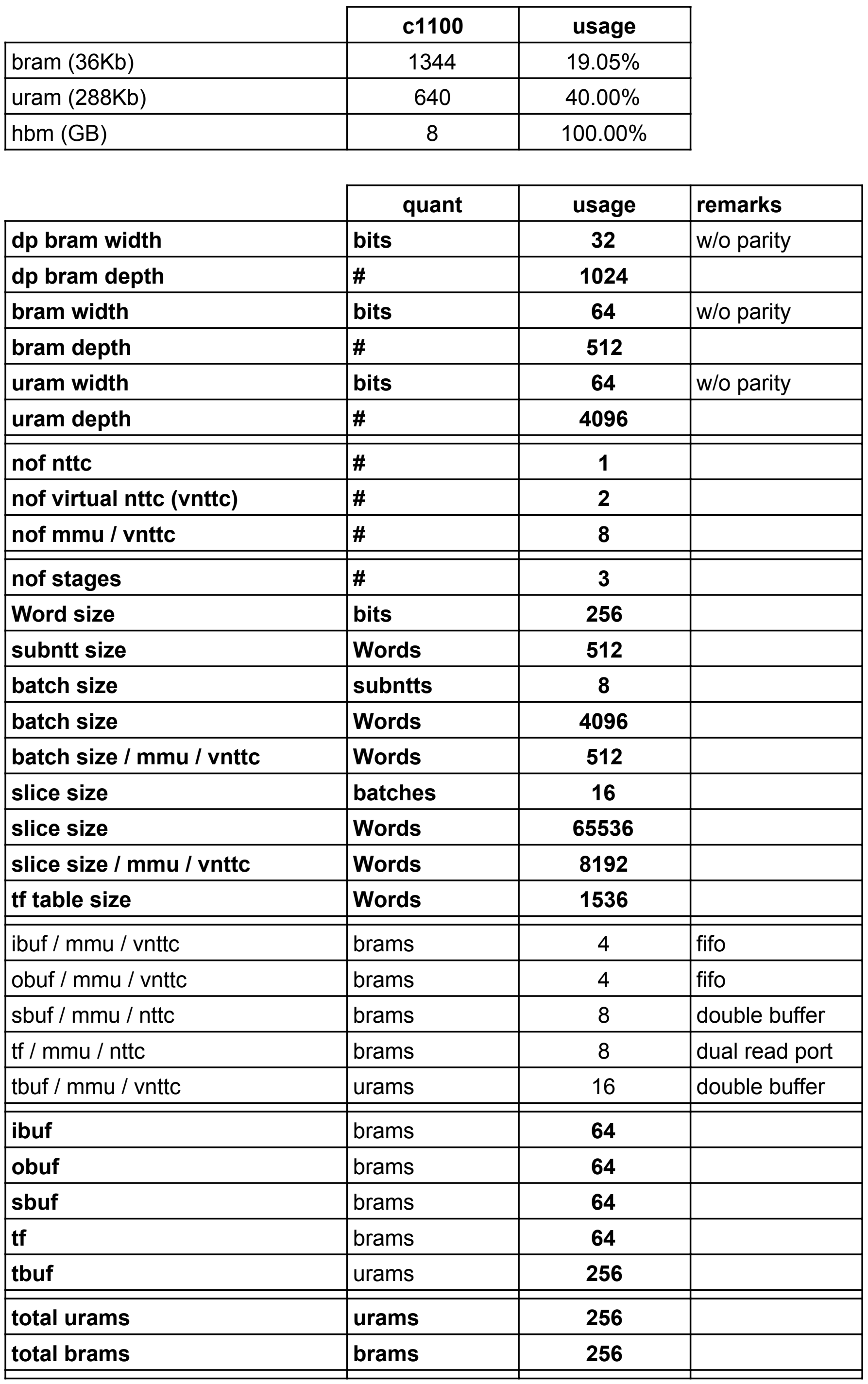


Figure 2.19: NTTC Structure

2.4.5 Memory Estimation

Given the planned architecture, as presented in this section, we estimated the memory usage of the design. To simplify the placement and routing of the design, along with previous constraints like limiting NTTC-SS to a single SLR, we attempted to use no more than 50% of the existing memory resources. The largest required memory is in REMAP-SS and we chose to use URAMs for it. The rest of the memories were defined to use BRAMs. The table in Figure 2.20 details our estimation.



	c1100	usage
bram (36Kb)	1344	19.05%
uram (288Kb)	640	40.00%
hbm (GB)	8	100.00%

	quant	usage	remarks
dp bram width	**bits**	**32**	w/o parity
dp bram depth	**#**	**1024**	
bram width	**bits**	**64**	w/o parity
bram depth	**#**	**512**	
uram width	**bits**	**64**	w/o parity
uram depth	**#**	**4096**	
nof nttc	**#**	**1**	
nof virtual nttc (vnttc)	**#**	**2**	
nof mmu / vnttc	**#**	**8**	
nof stages	**#**	**3**	
Word size	**bits**	**256**	
subntt size	**Words**	**512**	
batch size	**subntts**	**8**	
batch size	**Words**	**4096**	
batch size / mmu / vnttc	**Words**	**512**	
slice size	**batches**	**16**	
slice size	**Words**	**65536**	
slice size / mmu / vnttc	**Words**	**8192**	
tf table size	**Words**	**1536**	
ibuf / mmu / vnttc	brams	4	fifo
obuf / mmu / vnttc	brams	4	fifo
sbuf / mmu / nttc	brams	8	double buffer
tf / mmu / nttc	brams	8	dual read port
tbuf / mmu / vnttc	urams	16	double buffer
ibuf	brams	**64**	
obuf	brams	**64**	
sbuf	brams	**64**	
tf	brams	**64**	
tbuf	urams	**256**	
total urams	**urams**	**256**	
total brams	**brams**	**256**	

Figure 2.20: Memory Estimation

Bibliography

[1] Fourier transform. `https://en.wikipedia.org/wiki/Fourier_transform`.

[2] Parseval's theorem. `https://en.wikipedia.org/wiki/Parseval%27s_theorem`.

[3] Discrete-time fourier transform. `https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform`.

[4] Fourier series. `https://en.wikipedia.org/wiki/Fourier_series`.

[5] Discrete fourier transform. `https://en.wikipedia.org/wiki/Discrete_Fourier_transform`.

[6] Fundamental theorem of algebra. `https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra`.

[7] Lagrange polynomial. `https://en.wikipedia.org/wiki/Lagrange_polynomial`.

[8] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.

[9] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965. URL: `http://cr.yp.to/bib/entries.html#1965/cooley`.

[10] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[11] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. `https://eprint.iacr.org/2016/260`.

[12] Zero-knowledge proof. `https://en.wikipedia.org/wiki/Zero-knowledge_proof`.

[13] Ddr sdram. `https://en.wikipedia.org/wiki/DDR_SDRAM`.

[14] High bandwidth memory. `https://en.wikipedia.org/wiki/High_Bandwidth_Memory`.

[15] Filecoin. `https://filecoin.io/`.

[16] Ben Edgington. Bls12-381 for the rest of us. `https://hackmd.io/@benjaminion/bls12-381`.

[17] Xilinx c1100. `https://www.xilinx.com/products/accelerators/varium/c1100.html#overview`.

[18] Xilinx ultrascale+. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html`.

[19] Ben Edgington. Fast modular mutiplication. `https://github.com/ingonyama-zk/papers/blob/main/modular_multiplication.pdf`.

[20] Mateusz Raciborski and Aleksandr Cariow. On the derivation of winograd-type dft algorithms for input sequences whose length is a power of two. *Electronics*, 11(9), 2022.