Chapter 6

Larger and complex test cases

In this chapter, we explore two significantly larger and more complex designs than the earlier discussed wireless decoders. These test cases comprise of a million-point sparse Fourier transform design and a Reduced Instruction Set Computing (RISC) processor capable of booting Linux. Through these examples, we show how our technique scales and the benefits it provides for complex and heterogeneous hardware designs.

6.1 Million-point SFFT design

The first complex design chosen is a high-throughput implementation of a sparse Fourier transform that operates on a million (2^{20}) inputs that are frequency sparse, *i.e.,*only a few (in this case up to 500) frequency coefficients are non-zero. We first describe the design of the SFFT hardware and then provide various activity metrics for it.

6.1.1 Design overview

Processing million-point Fourier Transforms in real time can enable numerous applications ranging from GHz-wide spectrum sensing and radar signal processing to high resolution computational photography and medical imaging. Currently, million-point FFTs are not practical. Hardware implementations of such large FFTs are prohibitively expensive in terms of high energy consumption and large area requirements. However, for most of the above applications the Fourier transform is sparse which means that only few of the output frequencies have energy and the rest are noise. Recent work [34] in the field of algorithms has shown how to compute these sparse FFTs (SFFT) in sub-linear time more efficiently than standard FFTs and using only a sub-linear number of samples.

At a high level, the SFFT algorithm has two main steps:

1. *Bucketization:* In this step, the algorithm maps the million (2^{20}) frequencies into 4096 buckets such that the value of each bucket is the sum of the values of 256 consecutive frequencies mapped to it. This is done by multiplying the input samples by a Gaussian filter and performing a 4096-point FFT. This bucketization is repeated for several iterations but in each iteration a permuted set of samples of the input is chosen. This permutation of time samples results in a permutation of the frequencies and randomizes the mapping of frequencies to buckets as described in [34].

2. *Estimation:* The algorithm then estimates the locations and values of the large frequency coefficients. To estimate the locations, the algorithm uses a voting based approach. At the output of the 4096-point FFT, it picks the buckets with the largest values. These buckets vote for the frequencies that map to them. A large frequency coefficient will get a vote in every iteration as the values of the buckets they map to are proportional to their own large value. A negligible frequency coefficient will not always get a vote due to the random mapping of frequencies to buckets. Thus, the frequencies with the most votes are the large frequency coefficients. The values of these frequencies are estimated from the values of the buckets they map to.

The SFFT algorithm enables computing a million-point Fourier transform faster than standard FFT if the output is sparse. However, published software implementations [33, 65] of SFFT algorithms are unable to achieve high input data rates, nor are they efficient from the perspectives of power, energy, unit cost or form factor. We

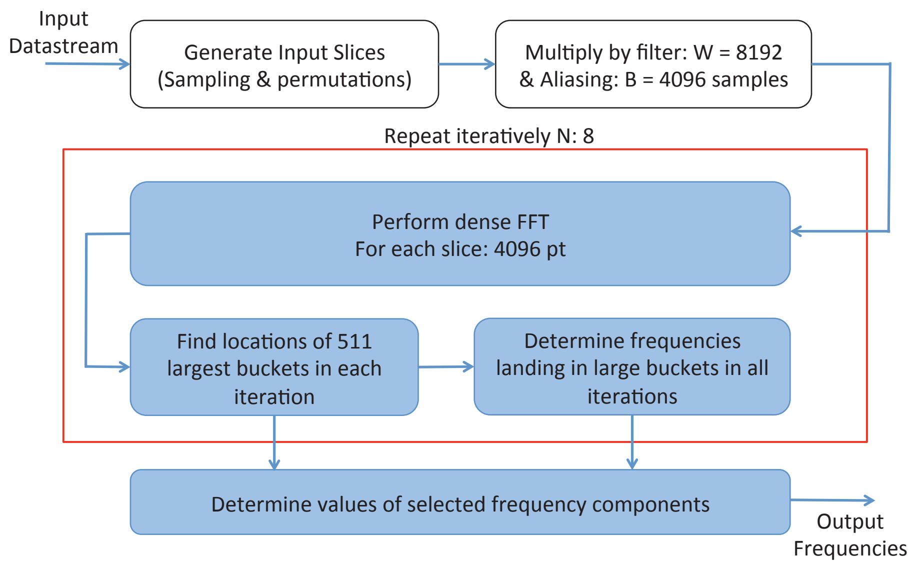
Input Datastream
Generate Input Slices (Sampling & permutations)
Multiply by filter: W = 8192 & Aliasing: B = 4096 samples
Repeat iteratively N: 8
Perform dense FFT For each slice: 4096 pt
Find locations of 511 largest buckets in each iteration
Determine frequencies landing in large buckets in all iterations
Determine values of selected frequency components
Output Frequencies

Figure 6-1: Stages of the SFFT algorithm. Core stages are highlighted in blue.

present the first hardware implementation of a million-point SFFT. We use Bluespec SystemVerilog [10], a high-level hardware design language for the design. Our design works in a streaming manner on 24-bit input samples to generate the locations and values of the largest 500 frequency coefficients in 4.49 milliseconds and hence can support input data rates of 2.23×10^8 samples per second. The SFFT algorithm version implemented in this work is robust with respect to the noise-level in the input. Our design is also reconfigurable for various sparse applications.

6.1.2 Design Architecture

The SFFT implementation consists of several modules, each implementing a distinct stage of the algorithm. Figure 6-1 shows the various stages involved in the SFFT algorithm.

In this section, we describe the implementation of four main stages of the algorithm. We have termed these stages collectively the SFFT Core, because they are responsible for the bulk of the computation and resource usage in the algorithm. Figure 6-2 shows the main modules used in our implementation of the SFFT Core and their input-output semantics. Our design has been parameterized to allow design exploration and to generate optimized results for the desired specifications. The

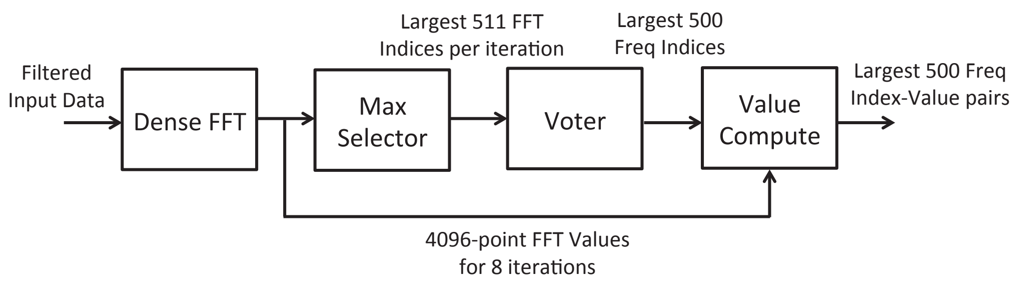
Filtered Input Data
Dense FFT
Largest 511 FFT Indices per iteration
Max Selector
Largest 500 Freq Indices
Voter
Value Compute
Largest 500 Freq Index-Value pairs
4096-point FFT Values for 8 iterations

Figure 6-2: Modules implementing the SFFT Core

Table 6.1: Parameters for SFFT Core implementation

Parameter	**Value**
Input data type	Complex fixed-point
Fractional bits for fixed-point data	24
Total number of input data values	2^{20}
Maximum non-zero input frequency	500
Number of iterations in algorithm	8
Size of FFT in each iteration	4096

parameters chosen for the discussed implementation are given in Table 6.1.We chose the input data type to be complex fixed-point with 24 fractional bits for each of the real and imaginary components. The high number of fractional bits ensures that we have sufficient accuracy for various applications. The number of input samples is 2^{20}, thus each input sample has a 20-bit location index and a 64-bit value (accounting for real and imaginary sign bit, integral bit and six overflow bits). The input data is constrained to have a maximum of 500 non-zero frequency coefficients. Increasing the number of iterations and size of FFT in each iteration, increases the accuracy of the probabilistic SFFT algorithm. But it also increases the resource usage and time required for completion. We chose 8 iterations with a 4096-point FFT in each iteration, as this choice gave sufficient accuracy while still providing an achievable hardware target. Each module was targeted to achieve a minimum operating frequency of 100 MHz. We next describe the architecture of the SFFT Core modules in detail.

4096-point FFT

The SFFT algorithm requires taking a standard FFT of filtered input data slices, which we refer to as dense FFT. Our design of the dense FFT module was required to have a high throughput, low area and maximum size of the FFT possible. The larger the size of the FFT, the lower is the chance of collisions occurring due to non-zero frequency components being mapped to the same bucket. Initial attempts to use folded in-place FFT designs [25] failed, as they did not scale to a size beyond 512 points for the 24-bit input data. Instead, this implementation uses a fully pipelined streaming FFT architecture [35], utilizing a Radix-2^2 Single Delay Feedback. Each internal block of the FFT architecture is designed as shown in Figure 6-3, where N is a parameter that varies from 1 to 1024. The definition of the butterfly structures is shown in Figure 6-4.

Figure 6-5 shows how the internal blocks are instantiated with appropriate parameter values to generate the pipelined 4096-point dense FFT implementation. The use of pipelined multipliers for complex fixed-point input, and adequate buffering in FIFO queues between blocks allows the design to continuously stream data across iterations without any stalls. The twiddle factors W_i used in the computation were generated as a look-up table that each block can independently query for values. The indices for the twiddle factors are determined by the collective value of the 2-bit counters present in each block. Each block has two shift registers that map directly to FPGA shift registers. Absence of large multiplexers, which are usually present in folded in-place FFT designs, allows this implementation to be highly efficient in FPGA resource usage. The design is parameterized for the input data type, FFT size and amount of pipelining in the complex multipliers.

Max Selector

In the previous stage, by performing the 4096-point FFT on the input data we have mapped the 2^{20} input frequencies into 4096 buckets. This stage of the sparse FFT algorithm requires determining which of these buckets have a large magnitude,

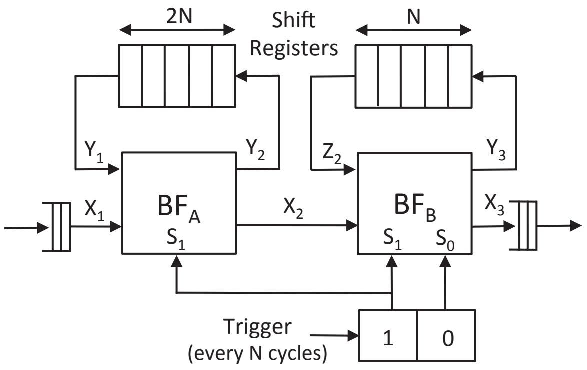

Figure 6-3: Single parameterized block of streaming FFT

```
if $S_1 == 1$
        $X_2 = X_1 + Y_1$
        $Y_2 = X_1 - Y_1$
else
        $X_2 = Y_1$
        $Y_2 = X_1$
```

(a) BF_A

```
if $S_0 == 1$
    if $S_1 == 1$
        $X_3 = X_2 + Z_2$
        $Y_3 = X_2 - Z_2$
    else
        $X_3 = X_2 - jZ_2$
        $Y_3 = X_2 + jZ_2$
else
    $X_3 = Z_2$
    $Y_3 = X_2$
```

(b) BF_B

Figure 6-4: Definition of butterfly structures in R2^2SDF design

indicating that one or more of the frequencies mapped are non-zero. Selecting buckets by setting a threshold would have been sensitive to noise levels in input and hence, not robust. Sorting all the FFT outputs to generate the ordered magnitudes was observed to be highly resource intensive and time consuming, as well as overkill since the algorithm does not require them to be ordered. Instead, we implement this step by selecting the largest (but unordered) 511 magnitudes of the 4096-point FFT output for each iteration. The chosen selector architecture operates on $2^n - 1$ entries, hence the number of entries being 511. Since the input data has a maximum of 500 non-zero frequency coefficients, selecting top 511 buckets by magnitude was sufficient to collect