

Project 3: Fast Fourier Transform

1. Introduction

The goal of this lab is to design architectures that implement the Fast Fourier Transform (FFT). The FFT is a more efficient version of the Discrete Fourier Transform (DFT). The FFT utilizes symmetry in the DFT coefficients to provide a recursive implementation that reduces the runtime from $O(N^2)$ to $O(N \log N)$ where N is the number of samples in the input signal.

2. Materials

You are given a zip file with three folders `0_Initial`, `1_Subcomponents`, and `2_Skeleton_Restructured`. Folder `0_Initial` contains the files corresponding to the “software” version of the FFT. Folder `2_Skeleton_Restructured` provides some framework for a more optimized FFT implementation. And folder `1_Subcomponents` has a number of subfolders that allow you to create projects for individual functions that you will develop over the project. This is largely for your convenience for testing and development. All of the code developed here will eventually be placed in to `0_Initial` and `2_Skeleton_Restructured`.

The structure of each of these folders is largely the same.

- `*.cpp` → The place where you write your synthesizable code.
- `*.h` → header file with various definitions that may be useful for developing your code.
- `*_test.cpp` → test bench
- `out.gold.dat` → “Golden” output. The testbench (`*_test.cpp`) generates a sample input and calls the function `*` in `*.cpp` with that sample input. This output of the function is compared to the expected output. This will indicate PASS or FAIL. If it fails, then the code in `*.cpp` is incorrect.
- `script.tcl` and `directive.tcl` → These allow you to easily create a project. To do this, open the Vivado HLS Command Prompt tool (Start-->Xilinx Design Tools-->Vivado HLS-->Vivado HLS Command Prompt) and go to the directory where source files and script files reside by using `cd` command. Then type `vivado_hls script.tcl`. This will create a HLS project automatically, so you do not have to add source files, set the top function, select the target device, etc. This will create a folder called `hls` in that directory with the project. It will also synthesize the project.
- `project3-fft.pdf` → project instructions: this file

3. Project Goal

You will create a 1024 FFT. The project is divided into stages. Unlike the previous projects, you are expected to write much of the C code yourself. The first part of the project is to perform the bit reversal of the input data. Then you optimize a “software” version of the code which we have given you (minus the bit reversal portion). Finally, you will create a more hardware friendly FFT architecture. We have provided a testbenches for the individual functions in addition to the testbenches for the overall FFT.

While the major goal of this project is to start writing code by yourself, you will also perform optimizations on the code. You should modify the code to create a number of different architectures that tradeoff between performance and area. You will create a report describing how you generated these different architectures (code restructuring, pragmas utilized, etc.). For each architecture you should provide its results including the resource utilization (BRAMs, DSP48, LUT, FF), and performance in terms of throughput (number of FFT operations/second), latency, clock cycles, clock frequency (which is fixed to 10 ns).

4. Bit Reversing the Input Data

The first step in most optimized FFT implementation is to reorder the input data by performing “bit reversed” swapping. This allows for in-place computation of the FFT, i.e., the resulting “frequency domain” data (as well as the intermediate results) can be stored into the same locations as the input “time domain” data. In addition, the output frequency domain data will be in the “correct order” at the end of the computation.

An example of the bit reversed data for an 8 point FFT is as follows:

Input Decimal (time domain)	Input Binary	Reversed Binary (frequency domain)	Reversed Decimal
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

In other words, the input data that was initially stored in the array at location 1 is stored in location 4 after the bit reversal is completed. The input data stored in the array at location 4 will be put in array location 1. The input data stored in locations 0, 2, 5 and 7 stay in those locations. Note that this is only true for an 8 point FFT. Other sizes of FFT will have different reordering of the data though it is still based on the bit reversed pattern. For example, in a 16 point FFT, the input data stored in location 1 (binary 0001) will be relocated into location 8 (binary 1000).

You should create an architecture that, efficiently as possible, transforms the input data into a bit reversed order. Note that there are many “software” implementations of this that will not effectively map to “hardware”. While the first goal is to get a working function, you should also consider the performance of the architecture.

We have given you a set of files that allows you to develop and test this bit reversal code in isolation. This includes a simple testbench that exercises this function directly. You should develop and optimize your bit reversed code here. You will later copy this code into the FFT code.

This code is in subfolder `1_bit_reverse` in the folder `1_Subcomponents`. You should develop your code here to insure that it matches the expected result. Note that this testbench is exercising only one input/output result. In other words, even if it passes this, it may not pass all results. Feel free to add additional testbenches to insure your code is correct.

The bit reverse function has the following prototype:

```
void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE])
```

You should perform the swapping “in place” on the data in both of the real and imaginary portions of the data. That is the input data in both `x_r` and `x_i` will be reordered when the function completes.

Your report should describe your code and its performance and area results. Focus on how you modified your code in order to make it more “hardware friendly”.

Hint: logical operations map well to hardware. Bit reversal can be done using logical operations.

5. Optimizing the “Software” Version of the FFT

The next portion of this lab performs optimization on a typical software implementation of the FFT. You are given typical three nested loop implementation of the FFT in the folder `0_initial`. First, you should understand in detail what this code is doing. It is worth spending time on this now as you will have to rewrite the FFT in a more hardware friendly manner in the next steps. You can reuse some of this code in those steps.

You should optimize this code as much as possible and detail these optimizations and their performance area results in your report. The results of the code will be poor; it will likely have > 250 million cycles. The throughput here is likely much worse than running this in software on a microprocessor. This often happens when we put the initial software versions of an application into a high level synthesis tool. And it should not be all that surprising. The code is optimized to run quickly in software, which runs largely in a sequential model of computation. The code must typically be carefully optimized with the final hardware architecture in mind to get good results. This involves exploiting parallelism and pipelining.

You will also notice that the first loop has function calls to `sin` and `cos`. Fortunately, unlike in the DFT project, this code will synthesize quickly despite these function calls. However, you may wish to replace these function calls (which will synthesize into CORDIC cores), into table lookups. We have provided two tables in the `.h` file, `w_real` and `w_imag` which contain the precomputed twiddle factors for our 1024 FFT, i.e., `w_real[i] = cos(2.0*pi*i/SIZE)` and `w_imag[i] = sin(2.0*pi*i/SIZE)` where `i = [0,512)`.

Some potential optimizations include:

- Using the `w_real` and `w_imag` tables
- Pipelining
- Loop unrolling
- Memory partitioning

In the report, you should discuss the performance and area numbers of the various optimizations that you apply. This must describe the effect of these optimizations on the architecture, and not just state the optimization that you used.

6. Hardware Friendly FFT Implementation

A good architecture will selectively expose and take advantage of parallelism, and allow for pipelining. Your final FFT architecture will restructure the code such that each stage is computed in a separate function or module. There will be one module for bit reversal that you

have already developed, and then $\log_2 N$ stages (10 in our case) for the butterfly computations corresponding to the 2-point, 4-point, 8-point, 16-point, ... FFT stages.

The skeleton code for this final FFT implementation can be found in the `2_Skeleton_Restructured` folder. This code connects a number of functions in a staged fashion with arrays acting as buffers between the stages. Figure 1 provides a graphical depiction of this process.

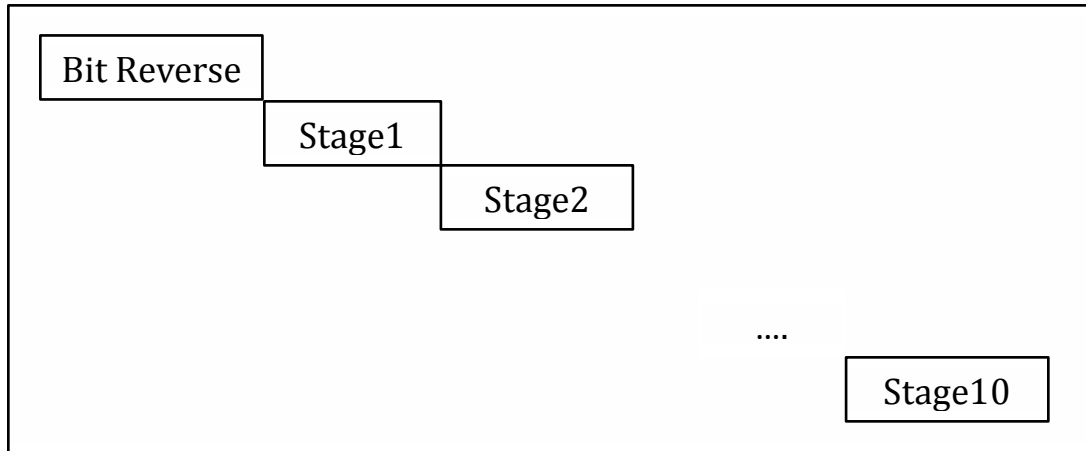


Figure 1: A staged implementation of a 1024 FFT. Bit reversal is followed by 10 stages of butterfly computations. This architecture is capable of pipeline both within the stages and across the stages.

The first step in this process is to create code that computes the first and last stages of the FFT. The hope is that this will allow you to get a better understanding of exactly how memory accesses and the butterfly computations are performed in a general case. You can develop these two functions `fft_stage_first` and `fft_stage_last` in isolation. They both have subfolders in the `1_Subcomponents` folder. Once these are working correctly, you can copy and paste the code directly in the same functions in the `2_Skeleton_Restructured` project.

The next task is to create code that can implement “generic” function, i.e., one that can compute any stage of the FFT. This is the function `fft_stages` which also has its own project in the `1_Subcomponents` folder. Note that this function prototype is similar to `fft_stage_first` and `fft_stage_last` with one major difference: it has a `stage` argument. This code will be used to implement stages 2 through 9 in the `2_Skeleton_Restructured` project.

Hints:

- These stages are performing the same calculation as one iteration of the outer for loop in the `0_Initial` project.
- The major difference between the stages is what data elements you are performing the butterfly functions on, i.e., in what order do you pull data from `x_R` and `x_I`.
- Test each of the functions in isolation with the provided projects. Make sure that the code compiles and passes the testbench before attempting any optimizations.

Once you have a correctly functioning set of functions, you should copy and paste them in the `2_Skeleton_Restructured` project and make sure that it passes the testbench. Since our testbenches only perform one check, which is far from comprehensive, it is possible, though

hopefully unlikely, that you have some error that the `2_Skeleton_Restructured` testbench exposes and was not exercised in the individual testbench. If your code passes the `2_Skeleton_Restructured` project you can assume it is correct (though again since it is only one test, it may be wrong; you would need to perform significantly more testing in any “non-class” situation).

Now on to the final part of the project, optimizing of this restructured. You should perform the typical tricks here: pipelining, memory partitioning, unrolling, etc. Some of these may not make sense depending on how you wrote your code. This final architecture should be orders of magnitude better than the `0_Initial` project. Highly optimized FFT architectures can easily have less than 10000 cycles.

You should describe in your report the baseline results and the effects of any optimizations that you performed. By this point, you should be starting to understand what optimizations will work and which will not work, and most importantly why. Describe these insights in your report.

7. Optimization Hints and Guidelines

- You must always use a clock period of 10 ns.
- The output of the various architectures that you generate must match the golden output. We have broken down the project into subcomponents to allow you to develop and test them individually. You would be wise to do it in such a manner.
- You should not change the data types as given to you. You do not need to perform bitwidth optimization of this project.
- There are some variables defined in the .h files for your convenience. These include `SIZE = 1024`, `SIZE2 = 512` and `M = 10 (log SIZE)`. Feel free to use these in your code. They are defined in every .h file across all of the different folders.
- The major goal of this project is to get a fully functional version of all the code. You should focus more of your time on getting everything to work rather than getting some but not all of the components to work and optimizing them.
- Your report should describe your code at a high level. Since each of you will likely write different code anyone that reads your report should be able to fully understand what each line of your code does. However, this does not mean that you need to explain each line of code in excruciating detail. It is possible to be succinct and thorough at the same time.
- Comment your code.

8. Submission Procedure

You must also submit your code (and only your code, not project files). Your code should have everything in it so that we can synthesize it directly. This means that you should use pragmas in your code, and not use the GUI to insert optimization directives. Each folder should have the necessary files to synthesize and test the code, e.g., `fft_test.cpp`, `fft.h`, etc.

The folder should be zipped with a similar file structure as below:

FFT_lab_YOUR_LAST_NAME.zip

Contents:

- “report.pdf” – A document describing all of your optimizations.

- Folder “0_Initial”: original unmodified dft.cpp
 - Subfolder “architecture1”: the set of files required to synthesize your first 0_Initial architecture
 - Subfolder “architecture2”: the set of files required to synthesize your second 0_Initial architecture
 - ...
 - Subfolder “architectureN”: the set of files required to synthesize your Nth 0_Initial architecture
- Folder “2_Skeleton_Restructured”: original unmodified dft.cpp
 - Subfolder “architecture1”: the set of files required to synthesize your first 2_Skeleton_Restructured architecture
 - Subfolder “architecture2”: the set of files required to synthesize your second 2_Skeleton_Restructured architecture
 - ...
 - Subfolder “architectureN”: the set of files required to synthesize your Nth 2_Skeleton_Restructured architecture

You should not send anything related to the 1_Subcomponents projects.

Email this to Janarbek Matai <jmatai@cs.ucsd.edu> with the title “CSE 237C Project 3”.