FFT - OFDM Receiver

Name: Will Zegers
PID: A09291938
Email: wzegers@eng.ucsd.edu

Description

My main change from the "software" FFT implementation to a more hardware-friendly design was targeted at removing variable for-loop bounds, particularly in the outer and inner loops of the `fft_stages()` function. The motivation for this was so that it would be possible to extract latency and interval metrics, which was not possible for variable loop bounds without the use of a tripcount pragma (which, in retrospect, did not end up giving accurate values anyway). Moreover, upon successfully flattening the two-tier for-loop structure in `fft_stages()` into one, this had the (not unexpected) result of significantly reducing resource usage (FF and LUT, specifically) by several factors.

Possibly the only real drawback to the new `fft_stages()` architecture was in making control flow slightly more obscure, as I ended up using separate variables for array indexing. These statements can be found on lines 106 and 107 in *fft1024_best/fft.cpp*:

```
theta = ((j+1) == theta + PHASE) ? theta + PHASE : theta;
i = (i >= UBOUND) ? i - (UBOUND - 1) : i + DFTPTS;
```

Here, `theta` is the index used to iterate over the `W_real` and `W_imag` lookup-tables, while `i` controls iteration over the input and output vectors; both values depend on not only the stage number, but also the progression of the loop itself, using functions of these values to "switch" or jump to a new index. For this, I used the ternary operator (implemented in hardware as a MUX) to control the values of the control variables, avoiding more costly if/else statements.

Additionally and for the purposes of maintainability and code reuse, I removed the `fft_stage_first()` and `fft_stage_last()` functions, which are easily replicated with calls to `fft_stages()` function. This didn't offer any significant boost to performance or area reduction, but it did simplify implementation and work flow, since all FFT stages could now be performed by a single function.

I also included some of the standard optimization pragmas now that the new architecture was in place. This included using the dataflow pragma on the top-level `fft()` function — which boosted performance by a factor of 10 by pipelining execution through each stage — as well as more fine-grained pipelining in the `bit_reverse()` and `fft_stages()` loop bodies.

Lastly, the design described above was unfeasible due to >100% DSP48E utilization; however, using the `ARRAY_MAP` pramga I was able to merge each pair of `StageN_R` and `StageN_I` arrays used to pass data between functions and significantly reduce BRAM18K and DSP48E use. This optimization brought resource utilization for the problem components (DSP48E) *far* below 100% utilization.

A few other code optimizations helped to reduce the execution time of more minor components of the FFT. For example, the `bit_reverse()` uses a bit-manipulation macro to map values from an input array to the bit-reversed indices of an added output array, all in a single line of code (per element). The `qpsk_decode()` function uses nested ternary operators to quickly translate input R and I values to the output D array, again avoiding if/else control statements.

As a final note, I experimented with additional optimizations that either offered little to no gain in regards to the throughput/area tradeoff, or took *far* too long to synthesize. For this reason, optimizations such as loop unrolling and the HLS stream library were not used in the final design.

The end result for interval cycles (1043) fell well below the 2000 cycle target, giving a final throughput of about 96 KHz.