Lab 7: IFFT

Due: 11:59:59pm, Sunday June 4, 2017

1 Introduction

The Inverse Fast Fourier Transform (IFFT) is widely used in signal processing circuits, including the ubiquitous WiFi wireless networks. In this lab, you will build circuits for three different implementations of this operation, combinational, folded, and pipelined, and compare their performance.

2 Butterfly Unit and Stages

Figure 1 shows the construction of the IFFT operation. You do NOT need to understand the IFFT algorithm to complete this lab, but briefly, the input is a set of 64 complex numbers; each 'butterfly 4' block (Bfly4) performs some complex arithmetic function on 4 complex numbers, producing 4 output complex numbers; and each Permute block permutes its 64 input complex numbers into its output 64 complex numbers.

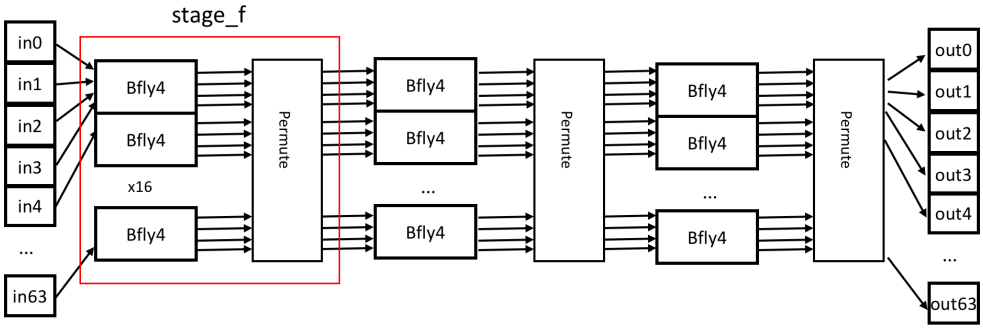

Figure 1: Construction of IFFT Operation

The 64 point IFFT operation could be divided into three stages. A function called `stage_f` has been provided in all modules. The input of the IFFT module should pass through `stage_f` three times. This function has another parameter called `stage` which should indicate which of the three stages the current function call is implementing. In other words, this parameter should be set to 0 for the first, 1 for the second, and 2 for the third stage, in order to produce the correct output. You can use the datatype `StageIdx` for this parameter to access a particular stage which is basically a Bit(3) and can take the value of 0, 1, or 2. You can see this and other datatypes of this lab in `IFFTCommon.bsv`.

3 Interface

You will be implementing modules with the following interface. All modules contain two FIFOs, `inFIFO` and `outFIFO`. The `enq` method writes the inputs to `inFIFO` and the `deq` method reads the outputs from `outFIFO`.

```
interface IFFT;
    method Action enq(DataType in);
    method ActionValue#(DataType) deq();
endinterface
```

4 Exercises

First, make a copy of the source code using the following command

```
$ cp -r  /home/linux/ieng6/cs140g/public/lab7 .
```

The modules `mkIFFTCombinational`, `mkIFFTFolded`, and `mkIFFTElasticPipeline` in `IFFT.bsv` should implement functionally-equivalent versions of the 64-point IFFT. Your job is to complete them by implementing the rules. The provided testbenches verify the correct functionality of your modules and report the number of cycles they took to process all their 128 inputs. You should make sure that this number is correct for each implementation.

Exercise 1 (20 pts): In `mkIFFTCombinational`, connect the three stages serially to complete a combinational IFFT implementation. This implementation should finish the overall IFFT algorithm (starting from dequeuing the input FIFO to enqueuing the output FIFO) in one cycle. Compile and test using:

```
$ make comb
$ ./simComb
```

Exercise 2 (30 pts): In `mkIFFTFolded`, create a folded IFFT implementation that makes use of just one stage. You can to create additional registers to hold the intermediate values. This implementation should finish the overall IFFT algorithm (starting from dequeuing the input FIFO to enqueuing the output FIFO) in exactly 3 cycles. Compile and test using:

```
$ make fold
$ ./simFold
```

Exercise 3 (20 pts): In `mIkFFTElasticPipeline`, create an elastic pipeline IFFT implementation. This implementation should make use of three stages and two FIFOs for pipelining. This implementation should finish the overall IFFT algorithm (starting from dequeuing the input FIFO to enqueuing the output FIFO) in exactly 3 cycles. Compile and test using:

```
$ make pipe
$ ./simPipe
```

Exercise 4 (30 pts): In `Lab7.txt`, write how you expect the maximum clock frequency, area, and throughput of the three different implementations relate to each other. Note that throughput is measured in IFFT *operations per second*, maximum clock frequency is the inverse of the maximum combinational logic propagation delay, and area is the total area of combinational logic and state elements (registers). Write **C** for combinational, **F** for folded and **P** for pipelined and use less-than operator ($<$) to show the relationship. For example, you can write $\mathbf{C} < \mathbf{F} < \mathbf{P}$ to indicate that a property is minimum for the combinational implementation and maximum for the pipelined implementation. Remember that your submissions are graded by a script that expects this format. If you write your answers in any other way, you will lose the points of this exercise.

5 Submission

Write down your name, PID and e-mail address in `Lab7.txt` located in `lab7/` directory. If you worked in a group, write down your partner's information as well. While your solutions can be identical, each group member must make their own submission. You will be scored based on your own submission. To submit your assignment, simply run the following command from `lab7/` directory:

```
$ bundleP7 <your-email>
```