Lab 3: FFT Pipeline

Due: Monday September 29, 2014

1 Introduction

In this lab you will build up different versions of the Fast Fourier Transform (FFT) module, starting with a combinational FFT module. This module is described in detail in the posted presentation from a previous version of this class.

First you will implement a folded 3-stage multi-cycle FFT module. This implementation shares hardware between stages to reduce the area required. Next you will implement an inelastic pipeline implementation of the FFT using registers between each stage. Finally you will implement an elastic pipeline implementation of the FFT using FIFOs between each stage.

2 Guards

The posted FFT presentation assumes guards on all of the FIFOs. Guards on `enq`, `deq`, and `first` prevent the rules enclosing calls of these methods from firing if the guards on the methods are not met. Because of this assumption, the code in the presentation uses `enq`, `deq`, and `first` without checking if the FIFO is `notFull` or `notEmpty`.

The syntax for a guard on a method is shown below:

```
method Action myMethodName(Bit#(8) in) if (myGuardExpression);
    // method body
endmethod
```

`myGuardExpression` is an expression that is `True` if and only if it is valid to call `myMethodName`. If `myMethodName` is going to be used in a rule the next time it is fired, the rule will be blocked from executing until `myGuardExpression` is `True`.

Exercise 1 (5 pts): As a warmup, add guards to the `enq`, `deq`, and `first` methods of the two-element conflict-free FIFO included in `Fifo.bsv`.

3 Data Types

Multiple data types are provided to help with the FFT implementation. The default settings for the provided types describe an FFT implementation that works with an input vector of 64 different 64-bit complex numbers. The type for the 64-bit complex data is defined as `ComplexData`. `FftPoints` defines the number of complex numbers, `FftIdx` defines the data type required for accessing a point in the vector, `NumStages` defines the number of stages, `StageIdx` defines a data type to access a particular stage, and `BflysPerStage` defines the number of butterfly units in each stage. These type parameters are provided for your convenience, feel free to use any of these in your implementations.

It should be noted that the goal of this lab is not to understand the FFT algorithm, but rather to experiment with different control logics in a real-world application. The `getTwiddle` and `permute` functions are provided with the testbench for your convenience. However, their implementations are not strictly adhering to the FFT algorithm, and may even change later. It would be beneficial to focus not on the algorithm, but on changing the control logic of a given datapath in order to enhance its characteristics.

4 Butterfly unit

The module `mkBfly4` implements a 4-way butterfly function which was discussed in the presentation. This module should be instantiated exactly as many times as you use it in your code.

```
interface Bfly4;
    method Vector#(4,ComplexData) bfly4(Vector#(4,ComplexData) t, Vector#(4,ComplexData) x);
endinterface

module mkBfly4(Bfly4);
    method Vector#(4,ComplexData) bfly4(Vector#(4,ComplexData) t, Vector#(4,ComplexData) x);
        // Method body
    endmethod
endmodule
```

5 Different Implementations of the FFT

You will be implementing modules corresponding to the following FFT interface:

```
interface Fft;
    method Action enq(Vector#(FftPoints, ComplexData) in);
    method ActionValue#(Vector#(FftPoints, ComplexData)) deq();
endinterface
```

The modules `mkFftCombinational`, `mkFftFolded`, `mkFftInelasticPipeline`, and `mkFftElasticPipeline` should all implement a 64-point FFT which is functionally equivalent to the combinational model. The module `mkFftCombinational` is given to you. Your job is to implement the other 3 modules, and demonstrate their correctness using the provided combinational implementation as a benchmark.

Each of the modules contain two FIFOs, `inFifo` and `outFifo`, which contain the input complex vector and the output complex vector respectively, as shown below.

```
module mkFftCombinational(Fft);
    Fifo#(2, Vector#(FftPoints, ComplexData)) inFifo <- mkCFFifo;
    Fifo#(2, Vector#(FftPoints, ComplexData)) outFifo <- mkCFFifo;
```

These FIFOs are the two-element conflict-free FIFOs shown in class with guards added in exercise one. Each module also contains a `Vector` or multiple `Vector`s of `mkBfly4`, as shown below.

```
    Vector#(3, Vector#(16, Bfly4)) bfly <- replicateM(mkBfly4);
```

The `doFft` rule should dequeue an input from `inFifo`, perform the FFT algorithm, and finally enqueue the result into `outFifo`. This rule will usually require other functions and modules to function correctly. The elastic pipeline implementation will require multiple rules.

```
    rule doFft;
        // Rule body
    endrule
```

The `Fft` interface provides methods to send data to the FFT module and receive data from it. The interface only enqueues into `inFifo` and dequeues from `outFifo`.

```
    method Action enq(Vector#(FftPoints, ComplexData) in);
        inFifo.enq(in);
    endmethod

    method ActionValue#(Vector#(FftPoints, ComplexData)) deq;
        outFifo.deq;
        return outFifo.first;
    endmethod
endmodule
```

Exercise 2 (5 pts): In `mkFftFolded`, create a folded FFT implementation that makes use of just 16 butterflys overall. This implementation should finish the overall FFT algorithm (starting from dequeuing the input FIFO to enqueuing the output FIFO) in exactly 3 cycles.

The makefile can be used to build `simFold` to test this implementation. Compile and run using

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

Exercise 3 (5 pts): In `mkFftInelasticPipeline`, create an inelastic pipeline FFT implementation. This implementation should make use of 48 butterflys and 2 large registers, each carrying 64 complex numbers. The latency of this pipelined unit must also be exactly 3 cycles, though its throughput would be 1 FFT operation every cycle.

The makefile can be used to build `simInelastic` to test this implementation. Compile and run using

```
$ make inelastic
$ ./simInelastic
```

Exercise 4 (10 pts): In `mkFftElasticPipeline`, create an elastic pipeline FFT implementation. This implementation should make use of 48 butterflys and two large FIFOs. The stages between the FIFOs should be in their own rules that can fire independently. The latency of this pipelined unit must also be exactly 3 cycles, though its throughput would be 1 FFT operation every cycle.

The makefile can be used to build `simElastic` to test this implementation. Compile and run using

```
$ make elastic
$ ./simElastic
```

6 Discussion Questions

Write your answer to this question in the text file `discussion.txt` provided in the lab repository.

Discussion Question 1 (5 Points): Assume you are given a black box module that performs a 10-stage algorithm. You can not look at its internal implementation, but you can test this module by giving it data and looking at the output of the module. You have been told that it is implemented as one of the structures covered in this lab, but you do not know which one.

1. How can you tell whether the implementation of the module is a folded implementation or whether it is a pipeline implementation? (3 Points)
2. Once you know the module has a pipeline structure, how can you tell if it is inelastic or if it is elastic? (2 Points)

Bonus

For an extra challenge, implement the polymorphic super-folded FFT module that was introduced in the last few optional slides of the FFT presentation. This super-folded FFT module performs the FFT operation given a limited number of butterflys (either 1, 2, 4, 8, or 16) butterflys. The parameter for the number of butterflys available is given by `radix`. Since `radix` is a type variable, we have to introduce it in the interface for the module, so we define a new interface called `SuperFoldedFft` as follows:

```
interface SuperFoldedFft#(radix);
    method Action enq(Vector#(64, ComplexData inVec));
    method ActionValue#(Vector#(64, ComplexData)) deq;
endinterface
```

We also have to declare *provisos* in the module `mkFftSuperFolded` in order to inform the Bluespec compiler about the arithmetic constraints between `radix` and `FftPoints` (namely that `radix` is a factor for `FftPoints`/4).

We finally instantiate a super-folded pipeline module with 4 butterflies, which implements a normal `Fft` interface. This module will be used for testing. We also show you the function which converts from a `SuperFoldedFft#(radix, n)` interface to an `Fft` interface.

The makefile can be used to build `simSfol` to test this implementation. Compile and run using

```
$ make sfol
$ ./simSfol
```

In order to do the super-folded FFT module, first try writing a super-folded FFT module with just 2 butterflys, without any type parameters. Then try to extrapolate the design to use any number of butterflys.