Lab 5: FIFOs

Due: 11:59:59pm, Sunday May 21, 2017

1 Introduction

FIFOs are extremely useful in streaming applications where the system works on a first-come-first-served basis. When the system has multiple modules, FIFOs can be used as buffers to store next requests to a module while the module is busy processing the current request. Similarly, the module results can be en-queued to a FIFO so that the module can proceed with the next request. In this assignment, you will use FIFOs to implement a simple streaming integer arithmetic system shown in Figure 1.

System Specification. Figure 1 shows the block diagram of the system as implemented in `mkTop` module. `Requests` to the system, which are composed of the operation type (add or GCD) and two operands (you can see the definitions in `Processors.bsv`), are first en-queued to `Input Request FIFO`. They are then transferred to `Adder Request FIFO` or `GCD Request FIFO` based on their `Operation` field. At this point, the system also updates the number of addition and GCD `Requests` it has received in `countADDOpers` and `countGCDOpers` registers. The `adder` module takes `Requests` from `Adder Request FIFO` and writes the results in the form of `Responses` to `Adder Response FIFO`. Similarly, `GCD` module takes `Requests` from `GCD Request FIFO` and writes the results in the form of `Responses` to `GCD Response FIFO`. Finally, `Responses` are transferred from `Adder Response FIFO` and `GCD Response FIFO` to `Output Response FIFO`. Note that the `Responses` do not need to be en-queued to the `Output Response FIFO` in the same order as the corresponding `Requests` were de-queued from `Input Request FIFO`.

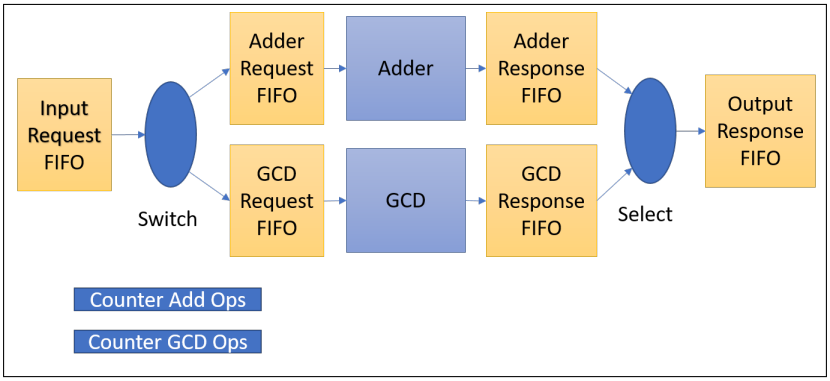

Figure 1: Block diagram of `mkTop` module.

2 Lab Assignment

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

```
$ cp -r /home/linux/ieng6/cs140g/public/lab5 .
```

2.1 Exercise 1: 20 points

Complete the methods of `mkAdder` and `mkGCD` in `Processors.bsv`. Both modules provide an interface called `Processor` which is composed of two methods. Method `request` is used to send a request (with the two operands) to a module and method `response` is used to retrieve the result. Do NOT make any changes to the rules in either module. You can test your implementations for correctness using the following testbenches. For the `adder`, type:

```
$ make adder
$ ./simAdder
```

For `GCD`, type:

```
$ make gcd
$ ./simGCD
```

2.2 Exercise 2: 80 points

Complete the rules and methods in `Top.bsv` so that the module `mkTop` operates according to the given specifications. The following is a series of suggested steps for completing `mkTop` module.

1. Connect `inputRequestFIFO` to `adderRequestFIFO` and `gcdRequestFIFO` by writing rule(s) that transfer requests based on their operation.
2. Write a rule to connect `adderRequestFIFO` to `mkAdder` module.
3. Write a rule to connect `gcdRequestFIFO` to `mkGCD` module.
4. Write a rule to connect `mkAdder` module to `adderResponseFIFO`.
5. Write a rule to connect `mkGCD` module to `gcdResponseFIFO`.
6. Connect `adderResponseFIFO` and `gcdResponseFIFO` to `outputResponseFIFO` by writing rule(s) that transfer `Response`s and handle conflicts (*Hint: You can use the compiler attribute* `descending_urgency` *to give explicit priority to certain rules. [BSV by Example Section 5.3.1]*).
7. Create the module interface by completing the methods `request`, `response` and `getCounts`.

You can test your implementation for `mkTop` module using:

```
$ make top
$ ./simTop
```

The testbench makes a series of `Request`s and reports both the correctness of your implementation (by printing `ALL PASSED!`) and the number of clock cycles it took your module to process and return all `Response`s. The lower the number of clock cycles, the faster your implementation is. Note that this is just a sample test for you to verify the correctness of your code and evaluate different approaches as you work. We will use a similar test, but with different operations/operators, to grade your submissions. The faster your implementation passes the test (i.e. the less the number of cycles reported by the testbench), the higher your score will be.

You can further improve the performance of your module by increasing the depths of the FIFO(s). FIFO depths are defined using `typedef` statements at the top of `Top.bsv` in lines 20-25. With a depth of one, the FIFOs work like simple registers, but a larger FIFO can speed up the design by avoiding stalls due to full FIFOs or busy modules. Assume that **you can only increase the depth of *one* of the FIFOs from 1 to 4**. Which one would you choose? Experiment with one FIFO at a time and find increasing the depth of which one results in the fastest operation. We will use this improved implementation for grading. *Note that in the end, the implementation you submit should have five FIFOs with depth 1 and one FIFO with depth 4.*

3 Submission

Write down your name, PID and e-mail address in `Lab5.txt` located in `lab5/` 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 `lab5/` directory:

```
$ bundleP5 <your-email>
```