Lab 4: N-Element FIFOs

Due: 11:59:59pm, Wednesday October 14, 2015

1 Introduction

This lab focuses on the design of various N-element FIFOs including a conflict-free FIFO. Conflict-free FIFOs are an essential tool for pipelined designs because they allow for pipeline stages to be connected without introducing additional scheduling constraints.

Creating a FIFO that is conflict-free is difficult because you have to create enqueue and dequeue methods that don't conflict with each other. FIFOs that are not conflict free, such as pipeline and bypass FIFOs, make an assumption about the ordering of enqueue and dequeue. Pipeline FIFOs assume dequeue is done before enqueue, and bypass FIFOs assume enqueue is done before dequeue. EHRs alone are used to implement pipeline and bypass FIFOS, and EHRs along with a canonicalize rule are used to create conflict-free FIFOs.

2 Parameterizable Sized FIFO Functionality

In lecture you have seen an implementation for a two element conflict-free FIFO. This module leveraged EHRs and a canonicalize rule to achieve conflict-free enqueue and dequeue methods. Dequeue only read from the first register, and enqueue only wrote into the second register. The canonicalize rule would move the contents of the second register to the first register if necessary. This structure works well for a small FIFO such as a two element FIFO, but it is too complicated to use for larger FIFOs.

To implement larger FIFOs, you can use a circular buffer as seen in http://en.wikipedia.org/wiki/Circular_buffer.

Figure 1 shows a FIFO implemented in a circular buffer. This FIFO contains the data `{1,2,3}` with 1 at the front and 3 at the back. The pointer `deqP` points to the front of the FIFO, and `enqP` points to the first free location past the FIFO.

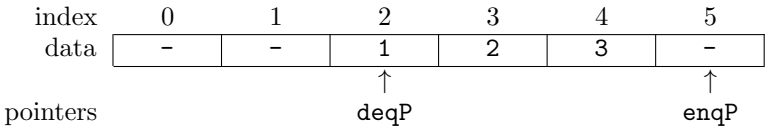

Figure 1: Example 6-element FIFO implemented in a circular buffer. This FIFO contains {1,2,3}.

Enqueues into a FIFO implemented in a circular buffer are simply a write to the location `enqP` and incrementing `enqP` by one. The result of enqueuing the value 4 into the example FIFO can be seen in Figure 2.

Dequeues are even simpler. To dequeue, all you need to do is increment `deqP` by one. The result of dequeuing a value from the example FIFO can be seen in Figure 3. Notice the data is not removed. The value 1 is still stored in registers for the FIFO, but it is in invalid space so it will never be seen by the user again. All of the `-`'s in the FIFO figures refer to old data that used to be in the FIFO, but they are no longer

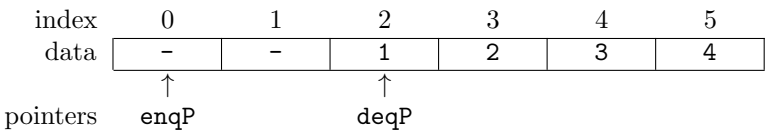

Figure 2: 6-element FIFO after enqueuing 4. This FIFO contains {1,2,3,4}.

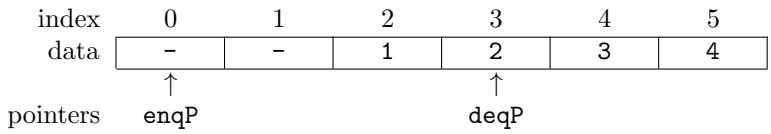

Figure 3: 6-element FIFO after dequeuing an element. This FIFO contains {2,3,4}.

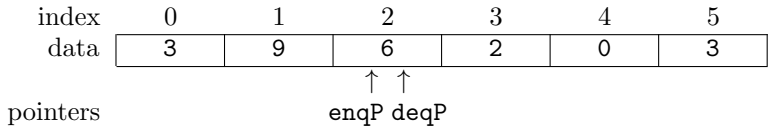

Figure 4: Full or empty 6-element FIFO.

valid. There are no valid bits in this FIFO structure. Locations are valid if they are at or after the dequeue pointer but before the enqueue pointer. This adds some complexity to figuring out if a FIFO is full or empty.

Consider the FIFO state in Figure 4. This figure shows a FIFO with `enqP` and `deqP` pointers pointing to the same element. Is this FIFO full or empty? You can not tell unless you have more information. To keep track of the state of FIFOs when pointers overlap, we will have a register saying if the FIFO is full and another one saying if it is empty. A full FIFO with the additional registers keeping track of full and empty can be seen in Figure 5

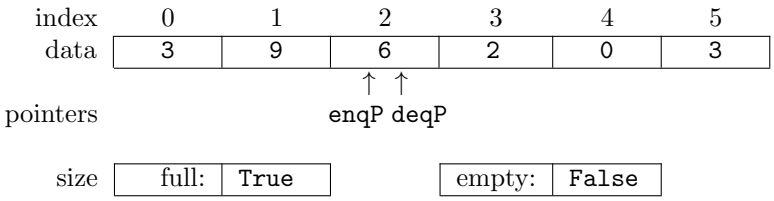

Figure 5: Full 6-element FIFO.

A cleared FIFO will have `enqP` and `deqP` pointing to the same location with `empty` being `True` and `full` being `False`.

If `enqP` or `deqP` are pointing to the same location, one of `empty` or `full` should be true. When one pointer is moved to the same position as the other pointer, the FIFO needs to set the `empty` or `full` signal depending on what method moved the pointer. If an enqueue operation was performed, `full` should be true. If a dequeue operation was performed, `empty` should be true.

3 N-Element FIFO Implementation Details

This section goes into the details required to implement an N-element FIFO as a circular buffer in Bluespec System Verilog.

3.1 Data Structure

The FIFO will have an `n` element vector of registers to store the data in the FIFO. This FIFO should be designed to work with a parametric type `t`, so the registers will be of type `Reg#(t)`.

3.2 Pointers

The FIFO will have pointers for both enqueue and dequeue operations. These pointers, `enqP` and `deqP`, point to the locations where the operations will happen next. The enqueue pointer points to the next element just past all the valid data, and the dequeue pointer points to the front of the valid data. These pointers will be registers with values of type `Bit#(TLog#(n))`. `TLog#(n)` is the numeric type corresponding to the ceiling of the base-2 logarithm of the value of the numeric type `n`. In short, `TLog#(n)` is the number of bits required to count from `0` to `n-1`.

3.3 State Flags

There are also two state flags for the FIFO to go along with the enqueue and dequeue pointers: `full` and `empty`. These registers are both false when `enqP` is not equal to `deqP`, but when `enqP` and `deqP` are equal, either `full` or `empty` is true expressing the state of the FIFO.

3.4 Interface Methods

This FIFO will keep the same interface as the previous FIFOs introduced in class.

```
interface Fifo#(numeric type n, type t);
  method Bool notFull;
  method Action enq(t x);
  method Bool notEmpty;
  method Action deq;
  method t first;
  method Action clear;
endinterface
```

The data type is `t` and the size is the numeric type `n`.

3.4.1 NotFull

The `notFull` method returns the negation of the internal full signal.

3.4.2 Enq

The `enq` method writes data to the location that the enqueue pointer points to, increments the enqueue pointer, and updates empty and full values if necessary. This method should be blocked with a guard if an enqueue is not possible.

3.4.3 NotEmpty

The `notEmpty` method returns the negation of the internal empty signal.

3.4.4 Deq

The `deq` method increments the dequeue pointer, and it updates the empty and full values if necessary. This method should be blocked with a guard if a dequeue is not possible.

3.4.5 First

The `first` method returns the element that the dequeue pointer points to, as long as the FIFO is not empty. This method should be blocked with a guard if the FIFO is empty.

3.4.6 Clear

The `clear` method will set the enqueue and dequeue pointers to 0, and it will set th state of the FIFO to empty by setting the internal full and empty signals to their appropriate values.

3.5 Method Ordering

Depending on the type of FIFO implemented, `enq` and `deq` may be able to fire in any order, a set order, or they may not be able to fire in the same cycle. The methods that are commonly associated with `enq` and `deq` should be able to fire with their respective method. That is, `notFull` should be able to fire with `enq`, and likewise `notEmpty` and `first` should be able to fire with `deq`. In all cases, the `clear` method should have priority over all other methods, and therefore it will appear to happen last.

3.6 Testing Infrastructure

There are two sets of testbenches for this lab: functional testbenches and scheduling testbenches.

The functional testbenches compare your FIFO implementation against a reference FIFO. The testbenches randomly enqueue and dequeue data and make sure all the outputs of the two FIFOs give the same results. These reference FIFOs are implemented as wrappers to a built-in BSV FIFO.

The scheduling testbenches works differently than all the other testbenches so far. The scheduling testbenches aren't meant to be run, they are only supposed to be compiled. These testbenches force schedules that your FIFOs should be able to meet. If the testbenches compile without warnings, then your FIFOs are able to meet those schedules, and they pass the tests. If your FIFOs are unable to meet the schedules, there will be compiler warnings or errors produced during compilation. That message will either be that two rules in the testbench cannot fire together, or that the condition of some rule depends on the firing of that rule.

When looking at the compiler output, make sure to look at what module is causing the errors by finding the lines that say `code generation for <module_name> starts`. Because of the way the Bluespec compiler is used, all the testbenches are partially compiled whenever you build one testbench so you may see warnings from modules you are not focusing on.

4 Implementing N-Element FIFOs

4.1 Conflicting FIFO

To start, you will implement an N-Element FIFO with only registers. This will cause `enq` and `deq` to conflict, but it will provide a starting point for all further FIFO designs.

Exercise 1 (5 Points): Implement `mkMyConflictFifo` in `MyFifo.bsv`. You can build and run the functional testbench by running

```
$ make conflict
$ ./simConflictFunctional
```

There is no scheduling testbench for this module because `enq` and `deq` are expected to conflict.

Now that we have an initial conflicting FIFO, we will explore the conflicts a bit and construct its conflict matrix.

Discussion Question 1 (5 Points): What registers are read from and written to in each of the interface methods? Remember that register reads performed in guards count.

Discussion Question 2 (5 Points): Fill out the conflict matrix for `mkMyConflictFifo`. For simplicity, treat writes to the same register as conflicting (not just conflicting within a single rule).

4.2 Pipeline and Bypass FIFOs

The pipeline and bypass FIFOs are a step past the conflicting FIFO. The pipeline and bypass FIFOs enable concurrent enqueues and dequeues by declaring a set ordering between them and their associated methods.

The pipeline FIFO has the following scheduling annotations.

```
{notEmpty, first, deq} < {notFull, enq} < clear
```

The bypass FIFO has the following scheduling annotations.

```
{notFull, enq} < {notEmpty, first, deq} < clear
```

There is a structural procedure to get these scheduling annotations from a conflicting design using EHRs.

1. Replace conflicting registers with EHRs.
2. Assign ports of the EHRs to match the desired schedule. First set of methods get port 0, second set gets port 1, etc.

For example, to get the scheduling annotation

```
{notEmpty, first, deq} < {notFull, enq} < clear
```

first replace the registers that prevent the above scheduling annotation with EHRs. In this case, that includes `enqP`, `deqP`, `full`, and `empty`. Now assign ports of the EHRs to match the desired schedule. `{notEmpty, first, deq}` all get port 0, `{notFull, enq}` get port 1, and `clear` gets port 2. You can optimize this design slightly by reducing the size of EHRs that have unused ports, but that is not necessary.

Exercise 2 (10 Points): Implement `mkMyPipelineFifo` and `mkMyBypassFifo` in `MyFifo.bsv` using EHRs and the method mentioned above. You can build the functional and scheduling testbenches for the pipeline FIFO and the bypass FIFO by running `make pipeline` and `make bypass` respectively. If these compile with no scheduling warning, then the scheduling testbench passed and the two FIFOs have the expected scheduling behavior. To test their functionality against reference implementations you can run `./simPipelineFunctional` and `./simBypassFunctional`. If you are having trouble implementing clear with the correct schedule and functionality, you can remove it from the tests temporarily by setting `has_clear` to false in the associated modules in `TestBench.bsv`.

4.3 Conflict-Free FIFO

The conflict-free is the most flexible FIFO. It can be placed in a processor pipeline without adding additional scheduling constraints between stages. The desired scheduling annotation for a conflict-free FIFO is shown below.

```
{notFull, enq} CF {notEmpty, first, deq}
{notFull, enq, notEmpty, first, deq} < clear
```

The `clear` method was chosen not to be conflict-free with `enq` and `deq` because it is given priority over the other methods. If `clear` and `enq` happen in the same cycle, the `clear` method will have priority and the FIFO will be empty in the next cycle. To match the behavior using method ordering, `clear` comes after `enq` and `deq`.

Just like the procedure for pipeline and bypass fifos, there is a procedure to get the desired conflict-free scheduling annotation using EHRs.

1. For each conflicting `Action` and `ActionValue` method that needs to be conflict-free with another method, add an EHR to represent a request to call that method. If the method takes no arguments, the data type in the EHR should be `Bool` (True for a request, False for no request). If the method takes one argument of type `t`, the data type in the EHR should be `Maybe#(t)` (`tagged Valid x` for a request with argument `x`, `tagged Invalid` for no request). If the method takes arguments of type `t1`, `t2`, etc., the data type in the EHR should be `Maybe#(TupleN#(t1,t2,...))`.
2. Replace the actions in each conflicting `Action` and `ActionValue` method with a write to the newly added EHR corresponding to the method.
3. Create a canonicalize rule to take requests from the EHRs and perform the actions that used to be in each of the methods. This canonicalize rule should fire at the end of each cycle after all of the other methods.

BSV does not have a way to force the canonicalize rule to fire every cycle, but it does have a way to statically check that it will fire every cycle at compile time. By using *compiler attributes*, you can add additional information about a module, method, rule, or function to the Bluespec compiler. You've already seen the `(* synthesize *)` attribute, now you will learn about two more for rules.

As you know, the guard for a rule or a method is the combination of the explicit guard and the implicit guard. The attribute `(* no_implicit_conditions *)` is placed right before a rule to tell the compiler that you don't expect there to be any implicit guards (the compiler calls guards conditions) from the body of the rule. If you are wrong and there are implicit guards in the rule, the compiler will throw an error at compile time. This guard acts as an assertion that `CAN_FIRE` is equal to the explicit guard.

Another thing that can prevent a rule from firing is conflicts with other rules and methods. The attribute `(* fire_when_enabled *)` is placed right before a rule to tell the compiler that whenever the guards for the rule are met, the rule should fire. If there is a way the guards can be met without the rule firing, then the compiler will throw an error at compile time. This guard acts as an assertion that `WILL_FIRE` is equal to `CAN_FIRE`.

Using these two attributes together will assert that the rule will fire whenever your explicit guard is true. If your explicit guard is true (or empty), then it is asserting that the rule will fire every cycle. Below is an example of the two attributes used together:

```
(* no_implicit_conditions *)
(* fire_when_enabled *)
rule firesEveryCycle;
    // body of rule
endrule
```

If the rule `fireEveryCycle` cannot actually fire every cycle, the Bluespec compiler will throw an error. You should have these attributes above your canonicalize rule to make sure it is firing every cycle.

Discussion Question 3 (5 Points): Using your conflict matrix for `mkMyConflictFifo`, which conflicts do not match the conflict-free FIFO scheduling constraints shown above?

Exercise 3 (30 Points): Implement `mkMyCFFifo` as described above *without* the clear method. You can build the functional and scheduling testbenches by running `make cfnc`. If these compile with no scheduling warning, then the scheduling testbench passed and the `enq` and `deq` methods of the FIFO can be scheduled in any order. (It is fine to have a warning saying that rule "m_maybe_clear" has no action and will be removed.) You can run the functional testbench by running `./simCFNCFunctional`.

4.4 Adding the clear method to the Conflict-Free FIFO

The clear method adds some complexity to the design. It needs scheduling constraints that prevent it from being scheduled before `enq` and `deq`, but it can't conflict with the canonicalize rule.

One of the easiest ways to create a scheduling constraint between to methods is have one method write to an EHR, and the other method read from a later port of the EHR. In this case, you should be able to use existing EHRs to force this scheduling constraint.

Exercise 4 (10 Points): Add the clear method to `mkMyCFFifo`. It should come after all other interface methods, and it should come before the canonicalize rule. You can build the functional and scheduling testbenches by running `make cf`. If these compile with no scheduling warning, then the scheduling testbench passed and the FIFO has the expected scheduling behavior. You can run the functional testbench by running `./simCFFunctional`.

Discussion Question 4 (5 Points): In your design of the clear method, how did you force the scheduling constraint `{enq, deq} < clear`?