6.S195: Lab 5
N-Element Conflict-Free Fifo

Andy Wright

October 15, 2013

Due: Tuesday October 22, 2013

Note: There is an n-element conflict-free fifo implementation in the course locker (mkCFFifo). Do not look at the code for this fifo implementation when working on this lab. It will be more of a distraction than a benefit because it uses a more complicated way of tracking if the fifo is full or empty, and it only works with fifo sizes that are powers of 2.

1 Introduction

This lab focuses on the design of an n-element conflict-free fifos. 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, along with a canonicalize rule, are used to create non-conflicting enqueue and dequeue methods in conflict-free fifos.

To begin this lab, check out the skeleton code from the course locker by running:

```
$ git clone $GITROOT/lab5.git
```

This repo contains skeleton code for two fifo implementations, a test bench file, a makefile, and a text file for written responses to lab questions.

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

index	0	1	2	3	4	5
data	-	-	1	2	3	-
pointers			↑ deqP			↑ enqP

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

index	0	1	2	3	4	5
data	-	-	1	2	3	4
pointers	↑ enqP		↑ deqP			

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

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 if you use it to create larger fifos.

To implement larger fifos, you can use a circular buffer as seen in http://en.wikipedia.org/wiki/Circular_buffer. Using a circular buffer simplifies the canonicalize rule because you no longer need to shift data around; you are only moving pointers to the enqueue position and dequeue position and updating full and empty flags.

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.

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 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

index	0	1	2	3	4	5
data	-	-	1	2	3	4
	↑			↑		
pointers	enqP			deqP		

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

index	0	1	2	3	4	5
data	3	9	6	2	0	3
			↑ ↑			
pointers			enqP deqP			

Figure 4: Full or empty 6-element fifo.

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

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.

index	0	1	2	3	4	5
data	3	9	6	2	0	3
			↑ ↑			
pointers			enqP deqP			
size	full:	True		empty:	False	

Figure 5: Full 6-element fifo.

3 Implementation Details

This section goes into the details required to implement a conflict-free version of this fifo in Bluespec.

3.1 Data Structure

The conflict-free 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 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.

```
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
```

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 not be implemented in the fifo for Exercise 1. For the later fifo implementation that includes a functional `clear` method, this method will set the enqueue and dequeue pointers to 0, and it will set the state of the fifo to empty by setting the internal full and empty signals to their appropriate values.

3.5 Conflict-Free

In order to make a conflict-free fifo, `enq` and `deq` must be able to fire in any order. In addition, 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 the methods `notEmpty` and `first` should be able to fire with `deq`. In order for the two groups of methods `{enq, notFull}` and `{deq, notEmpty, first}` to be able to fire in any order, each method in the first set must be conflict free with each element in the second set (written `{enq, notFull} CF {deq, notEmpty, first}`).

Potential conflicts can be tracked by resource usage. For example, looking at the data registers in the fifo, you will see that only two methods use those resources: `first` and `enq`. The method `first` reads from one of the data registers and `enq` writes to a data register [1]. If both of these actions are done in their respective method instead of a canonicalize rule, you would have the constraint

[1] As the fifo designers, we know that in this type of fifo the two methods will never use the same register in the same cycle, but the Bluespec compiler does not know that.

that `first < enq`. Since `first` is expected to return a value immediately, the register read has to be done within the method. Therefore the register write in `enq` needs to be performed in the canonicalize rule.

To push the register write from the method `enq` to the canonicalize rule, you need to create an EHR, or a group of EHRs to signal that the method `enq` was fired, and you need to send what data went along with the enqueue request. `enq` will write to port 0 of the EHR(s). The canonicalize rule will read from port 1 of the EHR(s), and it will reset the EHR(s) to some state that does not represent an enqueue. This will allow `enq` and `first` to fire in any order.

To find all the other potential conflicts, look at which methods read and write to `enqP`, `deqP`, `full`, and `empty`. If there is a conflict between two methods that should be conflict free, then EHRs need to be used to signal to the canonicalize rule to do the work.

When you are done, your canonicalize rule will be doing most of the logic associated with your fifo. It will be working with port 1 of all of the EHRs, and the methods will be working with port 0 of the EHRs.

3.6 Testing

Two test benches are provided to test your n-element conflict-free fifo. One test bench tests the functionality of your fifo, and the other tests the scheduling constraints of your fifo.

The first test bench tests the functionality of your fifo against a test fifo. The test fifo is `mkCFFifo` from Fifo.bsv. The test bench performs a random series of enqueues, dequeues, and, if applicable, clears. At each step, the output of notFull, notEmpty, and, when available, first are all compared between the test fifo and the reference fifo. The test bench is a good way to tell you if there is a problem with your fifo, but it is not possible to detect all problems with it. If the test makes it to the end, and you can see the output various fifo operations (not just the cycle counter), then the fifo probably works. This test can be run by running the following command:

```
$ make Functional.tb
$ ./simFunctional
```

The second test bench tests the scheduling constraints introduced by your Fifo. This test bench works by forcing all combination of scheduling constraints between the three groups of methods: `{enq, notFull}`, `{deq, notEmpty, first}`, and `{clear}`. This test bench is not meant to be run; it is designed to give compiler warnings or errors if these three groups of methods are not conflict free. This test can be run by running the following command and reading the output from the compiler:

```
$ make Scheduling.tb
```

If the compiler creates the executable `simScheduling` without any errors or warnings, then the fifo is conflict-free.

The `TestBench.bsv` file can be edited to test different sizes of fifos by changing the interface of `mkSpecificFifo`. In the second part of this lab, `mkSpecificFifo` can be modified to create an insteance of the fifo with the reset instead of the fifo from this section.

Planning Questions (10 Points): These are some questions to help you plan your design of an n-element conflict-free fifo. These questions will help you learn about the conflicts you are trying to avoid when designing your fifo. Answer these questions in planning.txt.

1. List the registers read by each interface method.

2. List the registers updated by each interface method.

3. What methods conflict with each other because they are trying to update the same registers?

4. What methods conflict with each other because one reads what another one writes?

5. Which conflicts from 3 and 4 prevent `{enq, notFull}` from being conflict free with `{deq, notEmpty, first}`?

6. What internal fifo storage elements (data, pointers, flags) will you need to turn into EHRs? What new EHRs do you need to introduce?

Exercise 1 (30 Points): Implement `mkCFFifoBasic` as described above without a clear method. Your fifo should have the following scheduling characteristics:

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

4 Adding a Clear Method

Now you are going to add a clear method to your conflict-free fifo. In the fifos used in the previous labs, the clear method conflicted with enq and deq. Because of the order of the EHR ports used, clear had to happen after both enq and deq. This makes sense conceptually since clear should have priority over enq and deq, but in some cases, you need clear to be scheduled before enq or deq while still having priority over the two. This can be done using an EHR to signal if the clear method fired, and use the canonicalize rule to clear the fifo if the EHR says the clear method fired.

4.1 SMIPS Example

The two stage pipelined SMIPS processor can use fifos to communicate between fetch and execute stages. If you want to reduce the misprediction penalty in the processor, you would do two things: make the fifo from execute to fetch a bypass fifo, and clear the fetch to execute fifo on misprediction. These two things seem simple enough, but as you will see soon, your scheduling constraints on your fetch to decode fifo need to be considered.

By implementing the misprediction fifo (the fifo that goes from execute to fetch) as a bypass fifo, you are requiring the execute rule to come before the fetch rule. That is because writes are scheduled before reads when using bypass fifos, and execute is writing and fetch is reading.

This now requires your dequeue on your fetch to execute fifo to be scheduled before your enqueue. If you also clear the fifo in the execute stage, your clear needs to come before your enqueue, but its action should shadow the later enqueue to the fifo. These specific constraints (`{deq, clear} < enq`) can be met by using a fifo that has conflict-free enq, deq, and clear. A fully conflict-free fifo gives you the flexibility to use it in any situation to satisfy existing scheduling constraints without introducing new ones.

4.2 Implementation

The clear method needs to write to `enqP`, `deqP`, `full`, and `empty`. Both enq and deq already modify these registers, so the canonicalize rule needs to be used to do the actual work to clear the fifo. The clear method will just set port 0 on a new EHR to true when it is fired. The canonicalize rule will be modified to read port 1 of this newly added EHR to see if the fifo should be cleared before doing anything else.

Exercise 2 (10 pts): Implement `mkCFFifoClear` by copying your code from `mkCFFifoBasic` and adding a clear method. This clear method should be conflict free with enq and deq, but it should still takes priority over the two. That is, if the fifo is cleared in the same cycle as an enq or a deq, the end result will be an empty fifo. `TestBench.bsv` can be modified to test `mkCFFifoClear` by changing the fifo implemented in `mkSpecificFifo` and setting `specific_fifo_has_clear` to true.

When you are done with this lab, commit all your changes and push your changes back to be graded by running the following commands:

```
$ git commit -am "Done with lab 5"
$ git push
```