Accumulator Tutorial

SRC Deliverables Draft, Fall 2009

1. Introduction

In this tutorial, we will be creating a PyTDL design that is capable of counting tokens

The goal of this tutorial is show how to create persistent data and the associated reset pattern.

1.1. Take-Away

PyTDL does not support explicit persistence of data, but it does provide enough mechanism via token links to introduce loops. Our aim is to demonstrate this ability.

2. The Design Stage

The requirements for the design are:

- Wait for an input token and keep a running sum of the token's payload.
- Output the sum everytime it is updated.

2.1. The First Steps

The first step is to describe the problem in terms of actions that need to be performed. We will map each action into a rule:

- Rule to dispatch input token/data
- Rule to accept input token/data and a current sum, add the two together

The first pass will be:

```
from Int import GlobalIn, GlobalOut

# This will not synthesize correctly

class Accumulator:
      def __init__(self):
            self.sum = Int(32)
            self.sum_out = GlobalOut(32)
            self.value = GlobalIn(32)

      def accumulate(self, trigger="tInput and tSum"):
            self.sum = self.sum + self.value
            create(tSum)

      def create_output(self, trigger="tSum"):
            self.sum_out = self.sum
            create(tOutput)
```

We have translated the rule descriptions above into PyTDL. You can see that there is a token loop here – the accumulate rule both creates `tSum` and accepts it as a trigger.

Note that we follow the convention requirement that outputs have their own variable – in this case, we use `sum_out`. If we didn't, we'd have to declare sum as both input and output (as you will see later in this tutorial). For now, we assert it as a sound design practice.

There are a few issues with this design: first, by setting the rule to contain the conjunction of `tInput` and `tSum`, we have introduced a Join arbiter. This in and of itself is unremarkable and is expected behavior. However, further analysis will show that we are attempting to use only parts of the data payloads from each token – the sum comes from `tSum` and the value from `tInput`. As of now, PyTDL does not support this type of disambiguation, so the user needs to modify the output Verilog.

Before we do that, however, we have a second issue to consider: reset semantics. There is no way to initialize the `tSum` value. This is covered in the next sub-section.

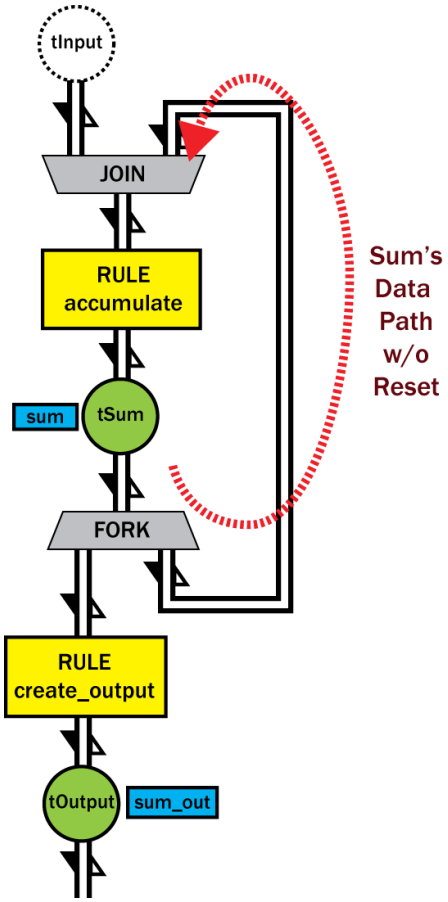

Figure 1: This is the basic design without any reset.

2.2. Reset Semantics

In order to reset the running sum, we need some way of interrupting the loop. That is, we need a mechanism to insert a token which contains the reset value of the sum after we have asserted the circuit's generic reset. We do this by introducing a rule `merge_sum`, which will accept either a new token `tReset`, or the running sum token `tSum`. We then create an intermediate token `tSumReady` which essentially lets `accumulate` know which value of `sum` to add to.

The modified code is:

```
from Int import GlobalIn, GlobalOut
class Accumulator:
      def __init__(self):
            self.sum = GlobalIn(32)
            self.sum_out = GlobalOut(32)
            self.value = GlobalIn(32)

      def merge_sum(self, trigger="tReset or
tSum"):
            create(tSumReady)

      def accumulate(self, trigger="tInput
and tSumReady"):
            self.sum = self.sum + self.value
            create(tSum)

      def create_output(self,
trigger="tSum"):
            self.sum_out = self.sum
            create(tOutput)
```

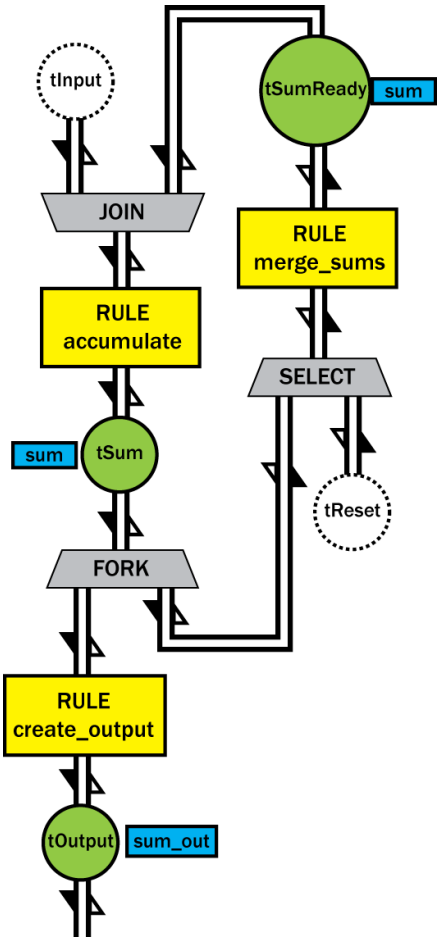

Figure 2: The accumulator with reset. All data attached to tokens has been annotated.

This code will now synthesize in PyTDL correctly, but ***will not generate working Verilog***.

We will now go back to the first design issue mentioned – the data selection of the Join arbiter.

First, look at the source code for the top-level Verilog module `Accumulator.v`. If you scroll down towards the bottom, you should see a section labeled "`DATA NETWORK`." Here, you should see something similar to the following:

```
/* -- JOIN -- */
// tInput -> join0x0
assign join0x0_sum_i0 = sum;
assign join0x0_value_i0 = value;


// tSumReady -> join0x0
assign join0x0_sum_i1 = tSumReady_sum;
assign join0x0_value_i1 = tSumReady_value;
```

This is the Join arbiter inputs. We can see that it assigns the dummy variable sum to its first input, attached to token tInput, while the second input's value is tied to tSumReady. What we want instead is the sum result to come from tSumReady and the value result from the tInput token. We simply make the following change:

```
/* -- JOIN -- */
// tInput -> join0x0
assign join0x0_sum_i0 = tSumReady_sum;
assign join0x0_value_i0 = value;
```

```
// tSumReady -> join0x0
assign join0x0_sum_i1 = tSumReady_sum;
assign join0x0_value_i1 = value;
```

Now, no matter which data set it decides to choose (doing so randomly), it will be correct.

We can now test the design with the packaged test bench to verify that it correctly accumulates the tokens.

2.3. Constraints File

The final component before we can synthesize the design is the constraints file, which lets PyTDL know which data I/O is associated with which token.

The syntax is detailed in the Reference Guide, so we defer any details to that document.

Even without reading the syntax rules, the constraints file is straightforward and easy to understand:

```
input = (tInput[value], tReset[value, sum])
output = (tOutput[sum_out])
```

We associate value to `tInput`. Token `tReset` also accepts a value and sum result, but these can simply be tied to zero externally and optimized out. If we exclude the value data on the `tReset` token, then the tools will not arbitrate correctly between `tReset` and the internal `tSumReady` token.

Output value `sum_out` is attached to token `tOutput`.