ROM Max Tutorial

SRC Deliverables Draft, Fall 2009

1. Introduction

In this tutorial, we will be creating a PyTDL design that is capable of continuously scanning a range of addresses from some memory (in our case, a simulation model of a ROM) and outputting tokens letting you know what the highest memory value is reads is.

The goal of this tutorial is demonstrate when the nondeterministic nature of arbiter results in an arbitrary selection that will not work. This is a property of the language and toolset, and occurs because of the inherently ambiguous nature of data flow and arbitration (that is, PyTDL by design allows the developer to create an ambiguous flow). On the plus side, it is relatively easy to find via simulation and easy to fix.

1.1. Take-Away

We hope that this tutorial clarifies how sometimes the designer needs to manually intervene with some of the nondeterministic elements of the design by modifying the output Verilog.

We additionally will cover reset semantics in the context of keeping persistent data via a loop that must contain at least one nonshorted SELF buffer.

2. The Design Stage

The requirements for the design are:

- Continuously sweep through the addresses of an external memory and read the data
- Maintain a persistent piece of information called “max” that holds the largest integral data value in the memory.

By default, the design must be tolerant to variable latencies, especially from the external memory.

2.1. The First Steps

First, we will establish a high-level overview. We'll need some way of sending a token out to the memory, and receiving data from that memory. We will also need to maintain a “max” variable and a current address variable; the former will hold the largest integer we've seen, and the latter will cycle through the available addresses.

2.1.1. Elastic ROM

We arbitrarily choose a 32-bit wide, 256-deep (and hence 8-bit address) elastic ROM – by elastic, we mean that it looks from the outside world like a token. It has valid inputs and outputs, and stall inputs and outputs.

The elastic ROM model is listed here. This is not directly synthesizable, as it is only a simulation model verified in ModelSim:

```
module elasticROM(valid_in, stall_out, valid_out, stall_in, addr,
data_out);
   parameter ADDRESS_WIDTH = 8;
   parameter DATA_WIDTH = 8;
   parameter DEPTH = 256;
   parameter FILE = "romdata.hex";

   input valid_in;
   output valid_out;
   input stall_in;
   output stall_out;

   input [ADDRESS_WIDTH-1:0] addr;
   output [DATA_WIDTH-1:0] data_out;

   reg [DATA_WIDTH-1:0] data [DEPTH-1:0];
   reg [DATA_WIDTH-1:0] data_out = 0;

   reg valid_out;
   reg stall_out;
   always @(addr or valid_in) begin
      data_out = #1 data[addr];
      valid_out = #1 valid_in;
      stall_out = 0;
   end

   // load file
   initial
      $readmemh(FILE, data);
endmodule
```

This is straight-forward ROM with added elastic control signals.

2.1.2. The Control

We start with a basic control design:

```
from Int import Int, GlobalIn, GlobalOut
class RomMax:
      # will not synthesize yet
      def __init__(self):
            self.max = Int(32)
            self.addr = Int(8)
            self.mem_data_in = GlobalIn(32)
      def nextAddr(self, trigger="tNewAddress"):
            self.addr = self.addr + 1
            create(tReadMemory)
      def checkMax(self, trigger="tMemoryValid"):
            if self.mem_data_in > self.max:
                  self.max = self.mem_data_in
            create(tNewAddress)
```

The `nextAddr` rule increments the address, then creates an external token called `tReadMemory` for the ROM. It then waits for the ROM to respond using the `tMemoryValid` token (which will have `mem_data_in` attached to it). Once the ROM looks up the given address, it fires this token, and hence triggers the `checkMax` rule.

[[AddressFlowDiagram]]

There are a few issues with this: there is no data path to "store" the current address. Rule `checkMax` changes the max variable, stores it onto `tNewAddress`, but does not have any knowledge of the address.

There are two cases here:

1. **Case I:** We can have `checkMax` wait for both `tMemoryValid` (containing the memory data payload) and `tReadMemory` (containing the current address). This presents an issue with the Join that will be automatically inserted: namely, the developer will need to manually change the Verilog output to route the various data into the input of the `checkMax` rule. See the section below for this approach.
2. **Case II:** We can pass both the address and max back into the design. That is, rule `nextAddr` outputs both max and address, which we then manually loop back into `checkMax` by attaching it to the `tMemoryValid` token.

Before we cover both cases' implementations and solutions, we'll cover two important pieces: how to deal with I/Os and how to handle resets.

2.2. Inputs and Outputs

Our design needs an interface to the external memory. We introduce the I/Os and a constraint file. Note that we follow the required convention of introducing *new variables* for I/O. This clears up ambiguous references when synthesizing:

```
from Int import Int, GlobalIn, GlobalOut
class RomMax:
      # will not synthesize yet
      def __init__(self):
            self.max = Int(32)
            self.addr = Int(8)
            self.mem_addr_out = GlobalOut(32)
            self.mem_data_in = GlobalIn(32)
            self.max_out = GlobalOut(32)
      def nextAddr(self, trigger="tNewAddress"):
            self.addr = self.addr + 1
            self.mem_addr_out = self.addr
            create(tReadMemory)
      def checkMax(self, trigger="tMemoryValid"):
            if self.mem_data_in > self.max:
                  self.max = self.mem_data_in
            create(tNewAddress)

            self.max_out = self.max
            create(tMax)
```

We've added a signal output for the address which will be attached to `tReadMemory`. We will not attach `addr` to this token, because it will introduce an ambiguity – the prototype compiler does not yet have a mechanism to resolve internal and external signal conflicts.

Now we create a constraints file to let PyTDL know the input and output token/data sets:

```
input = (tMemoryValid[mem_data_in])
output = (tReadMemory[mem_addr_out])
output = (tMax[max_out])
```

The design will output the `tReadMemory` token with address in tow, and wait for `tMemoryValid` with the data associated with that address. Token `tMax` will contain the highest integer observed.

2.3. Reset

Both max and address are persistent through the token loops, and in order to correctly initialize the data, we need some way of resetting them. This is where the Select/Or arbiter comes into play. The input to `nextAddr` is currently the token `tNewAddress`, carrying both the current address and current max; if we instead tell the rule to accept the `tNewAddress` token from either the `checkMax` rule *or* a new reset rule, then we can essentially "interrupt" the loop and inject new values.

```
from Int import Int, GlobalIn, GlobalOut
class RomMax:
      # will not synthesize yet
      def __init__(self):
            self.max = Int(32)
            self.reset_addr = GlobalIn(8)
            self.addr = Int(8)
            self.mem_addr_out = GlobalOut(32)
            self.mem_data_in = GlobalIn(32)
            self.max_out = GlobalOut(32)
      def resetAddr(self, trigger="tReset"):
            self.addr = self.reset_addr
            self.max = 0
            self.mem_data_in = 0
            create(tNewAddress)
      def nextAddr(self, trigger="tNewAddress"):
            self.addr = self.addr + 1
            self.mem_addr_out = self.addr
            create(tReadMemory)
      def checkMax(self, trigger="tMemoryValid"):
            if self.mem_data_in > self.max:
                  self.max = self.mem_data_in
            create(tNewAddress)

            self.max_out = self.max
            create(tMax)
```

We have now added an extra rule that accepts a "reset address" from the outside world. PyTDL will automatically insert a select arbiter to choose between the token generated from resetAddr and checkMax, nondeterministically choosing one. If both tokens arrive at the same clock edge, it will choose one, then the other – thus, the reset may not be accepted; it is recommended that you verify the Verilog implementation to prioritize the reset input on the arbiter.

We need to include the external token and associated reset data by adding the following to the constraints file:

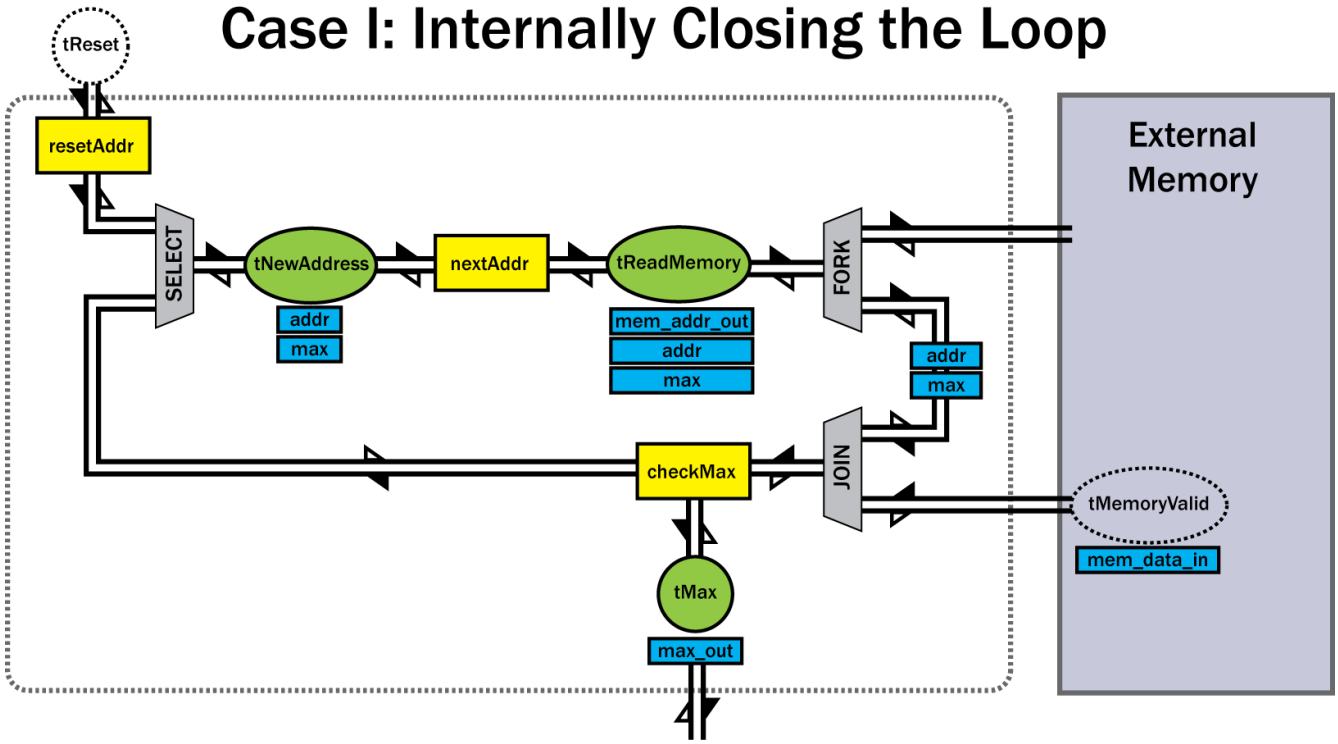
```
input = (tReset[reset_addr,max,mem_data_in])
```

A note on resource cost – although we have introduced a separate input, we can simply tie the reset address to ground/zero and the synthesis tools will replace the register portion of the token with wires. Thus, the overhead is whatever combinational control signals remain.

3. Addressing the Two Cases

3.1. Case I: Internally close the loop

To close the data loop and make the max and address data persistent, we can make `checkMax` wait for both `tMemoryValid` from the memory, and `tReadMemory` from `nextAddr`. The point of adding the latter is it will ship with it both the current address and current max, making the design functionally correct. However, in order to make this work, we need to make modifications to the top-level Verilog module to ensure that the right data source is selected by the Join arbiter.



3.1.1. Synthesize the Design

Now we have a full constraints file, working reset semantic, and data loop. Synthesize the design with this command:

```
python PyTDL.py -o constraints.tc RomMax.py
```

(See the User's Guide for details.)

If you now try to synthesize this, you will get error messages. This is where we need to make changes to the Verilog module, `RomMax.v` located in the `build` subdirectory.

WARNING: PyTDL will **recursively delete and replace** the `build` directory each time! We strongly recommend you copy the synthesized files into another directory.

3.1.2. Fixing The Verilog

After we have discovered that the resulting Verilog is not syntactically or functionally correct, we need to discover why. It turns out that the Join arbiter that waits for both `tMemoryValid` and `tReadMemory` expects data from one of them to be used in the input rule `checkMax`.

By default, Join will randomly select one of its input tokens' data payload. However, we did not specify the address and maximum number as part of the data payload for the input token `tMemoryValid`. PyTDL assumes this data exists somewhere; this is not the case, since the address and maximum number are not relevant inputs.

There is a second issue: we designed the rules such that different pieces of data come from different tokens. The address and max number come from the `tReadMemory` token, while the actual data from

the memory comes from the `tMemoryValid` token. The tools do not currently check for this type of use case (since it is, again, difficult to determine which token's data the developer wanted). In the future, we may support compiler pragmas to disambiguate situations like this one.

In any case, the following output is observed near the data network of the top-level Verilog file `RomMax.v`:

```
/* -- JOIN -- */
// tMemoryValid -> join0x0
assign join0x0_addr_i0 = addr;
assign join0x0_max_i0 = max;
assign join0x0_mem_data_in_i0 = mem_data_in;

// tReadMemory -> join0x0
assign join0x0_addr_i1 = tReadMemory_addr;
assign join0x0_max_i1 = tReadMemory_max;
assign join0x0_mem_data_in_i1 = tReadMemory_mem_data_in;
```

This piece of code assigns the token inputs to the join.

We can see that the payload inputs for the first set of inputs (attached to `tMemoryValid`) are fixed to non-existent Verilog signals `addr` and `max`. To fix this, we simply zero those values out.

The second problem manifests itself on the `tReadMemory` input of the join: `tReadMemory_mem_data_in` is not a valid data signal. To solve this issue, we simply grab the signal from the interface (coming from the memory) and assign it to the signal at the bottom:

```
/* -- JOIN -- */
// tMemoryValid -> join0x0
assign join0x0_addr_i0 = 0;
assign join0x0_max_i0 = 0;
assign join0x0_mem_data_in_i0 = 0;

// tReadMemory -> join0x0
assign join0x0_addr_i1 = tReadMemory_addr;
assign join0x0_max_i1 = tReadMemory_max;
assign join0x0_mem_data_in_i1 = mem_data_in;
```

This covers the *inputs* to the join arbiter, but what about the *output*? You will have to scroll up in the file to search for the instantiation of the `join0x0` module.

The actual output is listed as:

```
assign join0x0_max = join0x0_max_i1;
assign join0x0_addr = join0x0_addr_i1;
```

Here, we lucked out in that it selected its 2nd input (indicated by the "i1" suffix). If we had reordered the rules, it may have worked out oppositely, thereby assigning the "i0" signals to the data output.

3.1.3. Generalizing the Problem and Solution

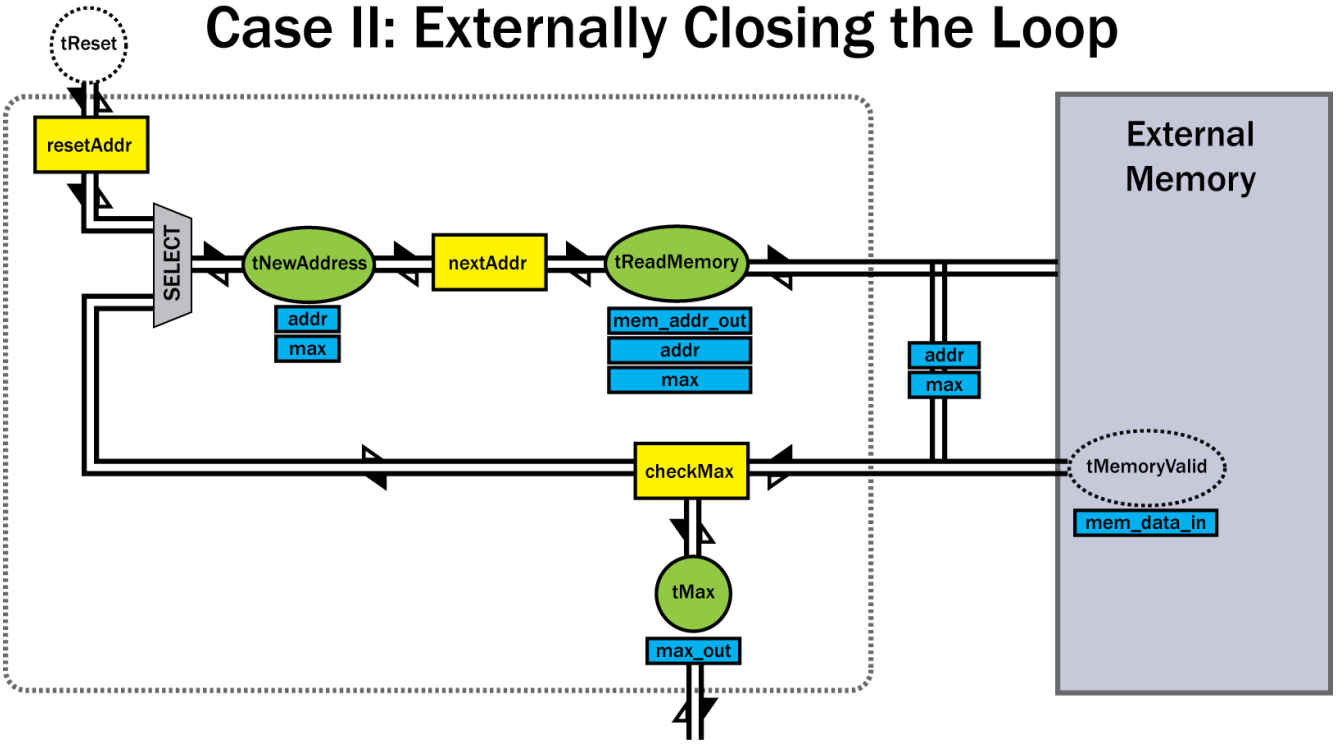
Now that we can demonstrate a situation where this problem arises, we can generalize what can happen and how to approach a solution.

The issue is that there is an inferred data network, the tools have no knowledge of exactly where the designer wanted data to flow. This, combined with the nondeterministic nature of the semantic model

means that the tool will make decisions to resolve the nondeterminism that may affect the end behavior. The Verilog code is structured in a way that it should be easy for the developer to find these fixed behaviors and replace them with whatever they desire.

3.2. Case II: Externally close the loop

The second, and arguably easier, case is to close the loop outside of the design.



This is accomplished in the instantiation of the top-level module. Only the relevant pieces are shown:

```
RomMax rommax(
    ...
    .o_addr( address_loop ),
    .i_addr( address_loop ),
    ...
);
```

The included files contain all of the remaining code, including a full testbench.