PyTDL Reference Guide

Revision A
SRC Deliverables Draft
Fall 2009

Table of Contents

1. Introduction

1.1. Overview

PyTDL is a language and prototype toolset that piggybacks on Python, borrowing its syntax and applying rule-based semantics to describe designs. The basic elements of these designs are tokens and latency tolerant rules that fire atomically when their associated input tokens arrive. The latency insensitivity is facilitated by automatic back-pressure that can propagate along the links connecting logical rules together; in this way, only rules that need to be stalled are stalled.

The language/tool pair additionally introduce a paradigm for logic design: behavioral correctness is achieved early, without any premature optimization; optimization is done after the behavior has been verified, and can be done *without affecting the correctness of the design*. Further, integration is simplified by inherently designing tolerance to timing variations into the specification. The end result is a relatively efficient, robust, and easy-to-integrate control unit.

1.1.1. The Toolset

The tools currently take PyTDL specifications and a constraints file as input and generate Verilog HDL. It is then left to the user to simulate the design for correctness and synthesize it to the target of the user's choosing.

After the behavior has been described, the designer may then assign control regions (section **4. Control Region Assignment**) to perform a trade-off between tolerance to latency variability which may have a high resource cost versus combinational loops which define timing constraints.

1.2. Structure of this Document

This reference guide is divided into two primary sections. The first, “High-Level Syntax and Semantics,” details the abstraction model presented to the developer, as well as guidelines for correct construction of certain patterns. The second section, “Implementation,” describes how the model's elements are realized in hardware; this gives the developer a starting point for understanding the end result of the tool, an essential aspect to debugging.

1.2.1. Syntax Definitions

Syntax definitions will follow standard documentation conventions. All literal code will be typefaced in a monospaced **`Courier`** font. Variable names will appear *italicized*. Optional blocks will appear in standard-font square brackets. If there is a choice between two rules, such as two keywords, they will appear in parentheses with a pipe (|) symbol separating them. Ellipses indicate lists.

1.2.2. Code Samples

The code samples will appear in a monospaced `Courier` font. For the sake of readability, explicit whitespace markers such as tabs and newlines are omitted, despite the whitespace sensitivity of Python. If appropriate, comments have been added to notate text layout ambiguity.

Variable, rule, and token naming follows the following conventions as *a matter of style only* (any valid

Python identifier that is not also a PyTDL keyword is accepted):

- Tokens will begin with a lowercase 't' followed by the token name capitalized, e.g. tTokenName
- Rules will be named descriptively, with the first letter lowercased and words separated by capitalization, e.g. ruleThatCalculatesSum
- Variables will be lower case, with underscores (_) separating words, e.g. first_multiplicand
- Variable instantiation classes and top-level design classes will be capitalized, e.g. PipelinedProcessor

For instance:

```
class ClassName:
  def fireMemoryWrite(self, trigger="tWrite and tDataToWrite"):
        self.addr_out = self.pc + 4
        create(tMemoryWrite)
```

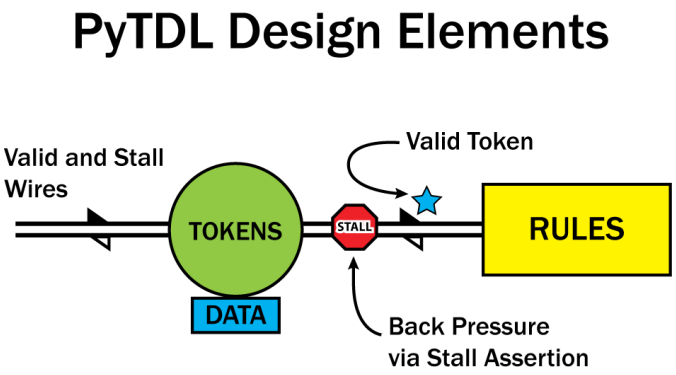
1.3. System Requirements

- Python 2.5.x
- PLY 2.5 or later (given that it is 2.5+ compatible)

PyTDL runs on Python 2.5 ***and is not forward compatible***. The precise revision used in developing this tool is 2.5.2; however, we have no reason to believe that any of the 2.5.x versions will not work. Be warned that we have not tested it in 2.6 or later. Critical introspection components used to compile designs are purported to have changed according to the current documentation (to a different and newer API) or have been deprecated.

Currently, the constraints files are compiled using the **PLY** package, available at: **http://www.dabeaz.com/ply/**. Installation instructions are deferred to the documentation on the source site.

2. High-Level Syntax and Semantics



The four basic design elements are described below, each one subdivided into a description of the semantic model and a definition of the syntax.

2.1. Four Basic Elements

Four basic design elements compose the abstraction model:

1. **Tokens**
2. **State**, which are data elements strictly attached to tokens
3. **Rules** executed atomically when triggered by tokens
4. **Back-Pressure**, which facilitates latency tolerance

2.1.1. Tokens

I. SEMANTICS

Tokens are the basic unit of control. They contain a source, which may be an external input or a rule creating a token, and a destination, which may be a rule or an external output. They trigger the firing of rules, and provide the necessary data that their destination rules then process. Semantically, a token can be thought of as a storage element such as a flip-flop with the following control signals:

- **Valid Input and Output:** Valid input without a stall signifies a write to the token. Valid output specifies that the token contains valid data so that the token's destination can be fired.
- **Stall Input and Output:** Stall comes in from the token's destination and essentially notifies the token that it should not fire. This signal is propagated backwards to the token's source (whatever creates that token) via the last output, notifying it not to create any more tokens until the stall signal has been lowered.

The stall signals implement the back-pressure mechanism explained below in further detail. Note that tokens may also be used without data – that is, for control only. However, we do not fully support this use case at this time, largely because the synthesis of PyTDL designs is based on the flow of data. If there is an ill-defined datapath, the synthesis tools may not correctly elaborate the design.

II. SYNTAX

Tokens do not need to be declared. In order to use a token, simply create it as follows:

`create(`*tokenName*`)`

To link it to a rule, include the token name in that rule's trigger condition. The rest (arbitration and data flow) is handled automatically by PyTDL.

2.1.2. State

I. SEMANTICS

Attached to a given token is any state relevant to the rules that it triggers. Each token has its own copy of data – that is to say, there is no relationship between two tokens' data, nor is there an explicit one between the current data and any past data. That is to say, data exists purely from token-to-token, and there is no coherence or persistence of data outside of the token-rule coupling.

There is no persistence storage or shared state in PyTDL. All of the data exists as payloads on tokens.

When the design is elaborated, a data path network is constructed to determine the assignment of data payloads based on which rules reference variables with the same name.

Designs essentially specify where data is used by accessing them in rules. While there is no persistence beyond the token passing from one rule to another, there is a consistent notion of naming. If two rules reference variable `foo`, and one passes a token to the other, they will refer to the same transient *value*.

From this comes the notion of a data path network. PyTDL infers where data flows based on its use, and does so as intelligently as it can. Consider three rules chained together such that a single token can pass through all three rules. If a variable is only referenced in the first and third rule in the chain, PyTDL will construct a graph that includes this dependency and will likewise include the data in all token links.

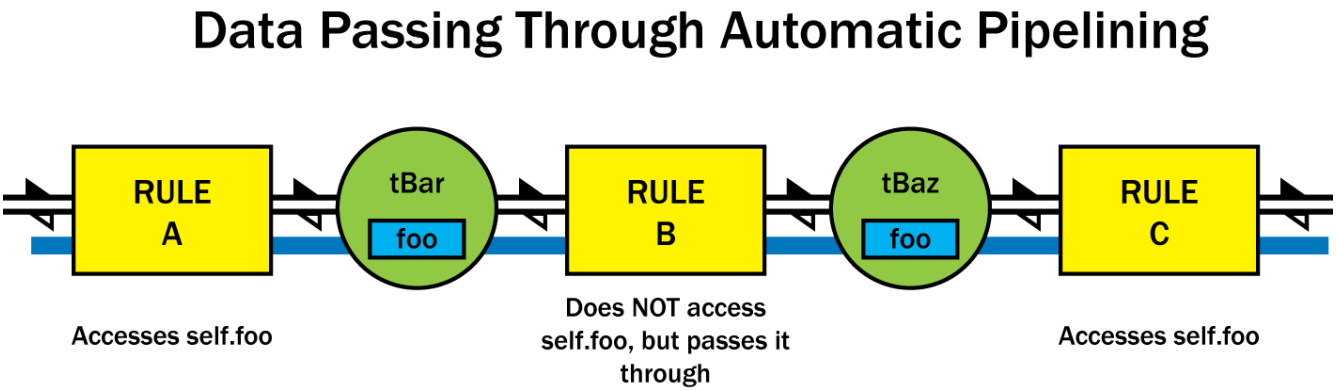


Figure 1: Rules A and C reference self.foo, but rule B doesn't. PyTDL, however, will detect the dependency from the inferred data flow network, and will subsequently attach the foo data to the tBar token.

II. SYNTAX

For information on how to define, declare, and use data, see section **5.6.2. Instantiating/Declaring Data Types**.

2.1.3. Atomic Rules

I. SEMANTICS

Rules are pure functions which take in as input the data attached to its trigger tokens and create as output new tokens (to which any modified data is attached). They may not arbitrarily observe the state of tokens and conditionally accept. That is to say, there is no mechanism to check if a token exists. They simply fire when a token arrives, so long as there is no back-pressure (as described in the next section).

These rules are atomic, meaning the ordering of operations is inconsequential so long as code blocks are consistent (e.g. the true branch of a conditional `if` statement must contain all of its statements within the block), and the stateless variable ordering is functionally preserved. These semantics are borrowed from the output language; rules turn into Verilog functions, as the Implementation section will explain.

II. SYNTAX

Rules look syntactically like Python methods, but instead of passing parameters to the body of the method, the parameters are used as attributes to the rule itself.

Each rule follows this syntax:

`def` *ruleName*`(self, trigger="`*triggerConditionExpression*`"):`

    ...

Both the `self` and the `trigger` portions are required in exactly that form. The designer may then select a rule name and trigger condition (enclosed in quotations).

The rule condition is specified as any valid combination of conjunctions and disjunctions of token names:

tokenName [(**`and`**|**`or`**) *tokenName*[(**`and`**|**`or`**) *tokenName*[...]]]

Example of a valid trigger conditions:

```
def ruleName(self, trigger="tFoo and (tBar or tBaz)"):
  ...
```

The rule does not return anything (therefore the `return` keyword is not valid). It interacts with other rules via creation of tokens.

For details on how multiple tokens are combined, see **2.1.5. Arbitration**.

2.1.4. Back-pressure

Back-pressure is the mechanism that creates chains of stall conditions. If a rule cannot fire (for instance, if it contains a long combinational chain which is still propagating, or it is waiting for some external token to arrive), then it notifies its input tokens of the stall (via the tokens' stall input signals). These tokens will then propagate *their* stall outputs to whatever the token source is.

This continues until the outermost token asserts its stall on the external I/Os.

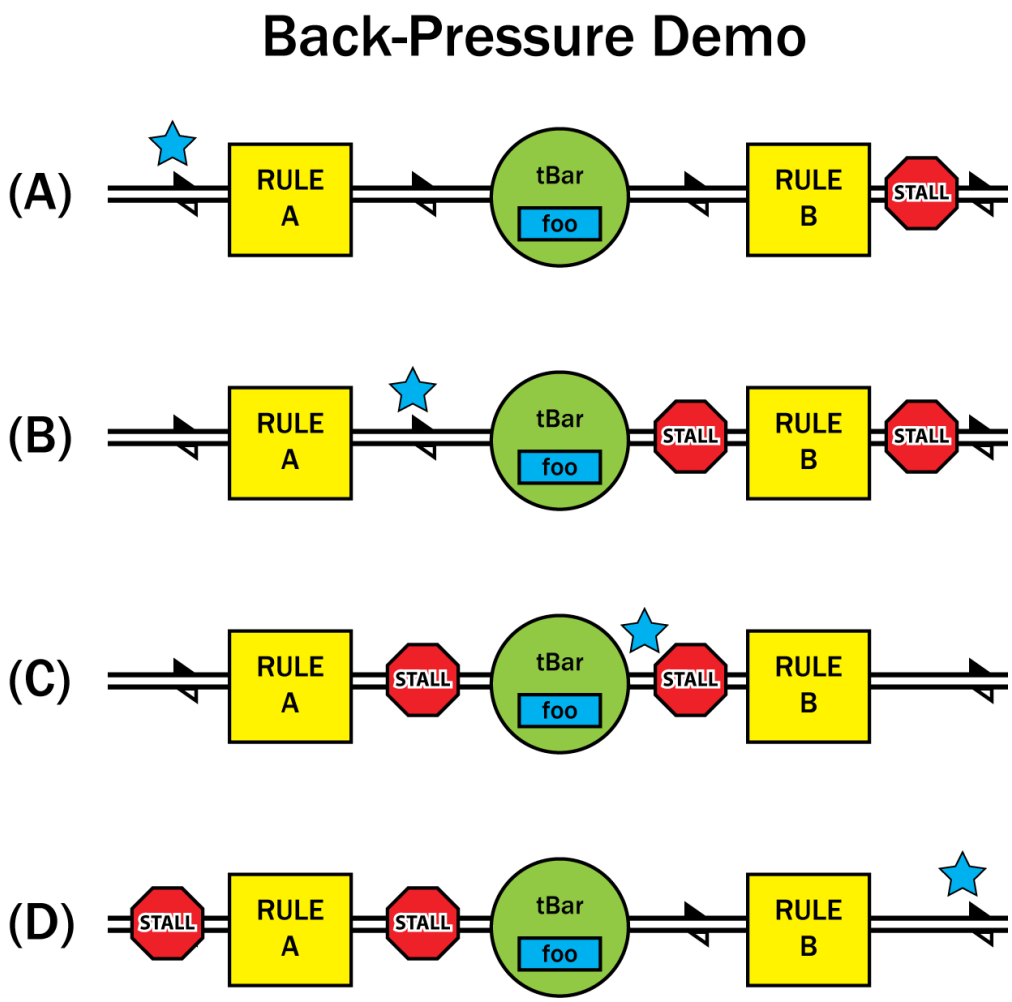


Figure 2: In stage (A) a valid token arrives at rule, while at the same time rule B observes back-pressure; (B) rule B backwards propagates the back-pressure while rule A forward propagates the token; (C) the token can no longer continue and tBar will hold onto the data until the stall is cleared in (D).

2.1.5. Arbitration

Oftentimes, designs require control decisions to be made on the level of tokens. For instance, in order to design a system that branches off into two paths (by having the rule create two tokens) and later combining the two, the designer needs some way of describing the combination of more than one token as the trigger for a rule.

PyTDL offers two nondeterministic arbitration methods, and a fork/token-duplication node – all automatically generated. The former consists of **conjunctive token combination** (the "AND" or "JOIN" of two tokens) and **disjunctive token selection** (the "OR" or "SELECT" of two tokens). Both are automatically inserted based on the rule condition, described by the `trigger` input.

1. **Conjunctive Token Combination (JOIN)**: Join arbiters wait for more than one token to arrive at a single point before firing the output token. They are inserted anywhere a rule's

trigger contain logical conjunctions ala `"tInput and tRead"`. If the conjunction condition is not met – that is, not all of the tokens have arrived – the arbiter asserts back-pressure on those links which have tokens waiting (indicating that the join cannot consume those inputs) until the other tokens arrive.

2. **Disjunctive Token Selection (SELECT):** Select arbiters funnel multiple tokens that arrive at the same time into one output token, nondeterministically choosing which token passes through and forcing the other tokens to wait via back-pressure. The designer can prescribe a certain priority ordering for tokens to help the compiler resolve the nondeterminism by adding specific commands to the constraints file. See **6.4. Setting Select Arbiter Priorities**.
3. **Token Duplication (FORK):** Finally, if more than one rule accepts the same token, a fork node is automatically inserted. The default behavior is an eager fork; if the fork sees back-pressure on one node, but not the other, it will first fire the token it *can* fire before propagating the stall.

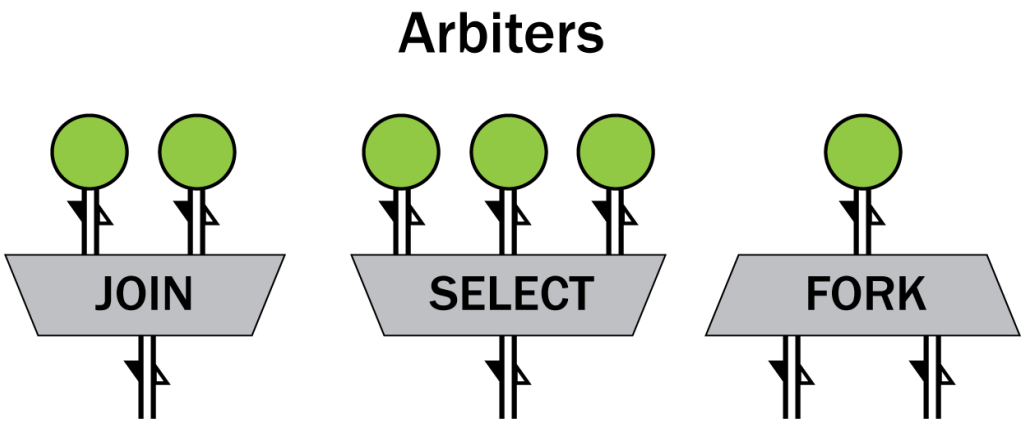


Figure 3: A 2-input Join will wait for both tokens to arrive; a 3-input select will nondeterministically choose one of the tokens to fire and force the others to wait; a 2-output fork will duplicate this token over two links.

2.2. Examples

2.2.1. Trivial Rule Firing

Consider the following code example:

```
from Int import GlobalIn, GlobalOut
class TrivialRuleFiring:
  def __init__(self):
        self.x_in = GlobalIn(32)
        self.x_out = GlobalOut(32)
  def trivialRule(self, trigger="tInput"):
        self.x_out = self.x_in + 1
        create(tOutput)
```

We have defined the top-level design class called `TrivialRuleFiring`. This class contains an initialize method which specifies that the member variable `x_in` is of type `GlobalIn` and has a size of 32 bits. Likewise for `x_out` with type `GlobalOut`. The data types are covered in section **5.6. Data Types**. For now it suffices to say that this sets up `x_in` as a *potential* data payload label that will arrive from the outside world attached to some token, and that `x_out` similarly will be an output attached to a token.

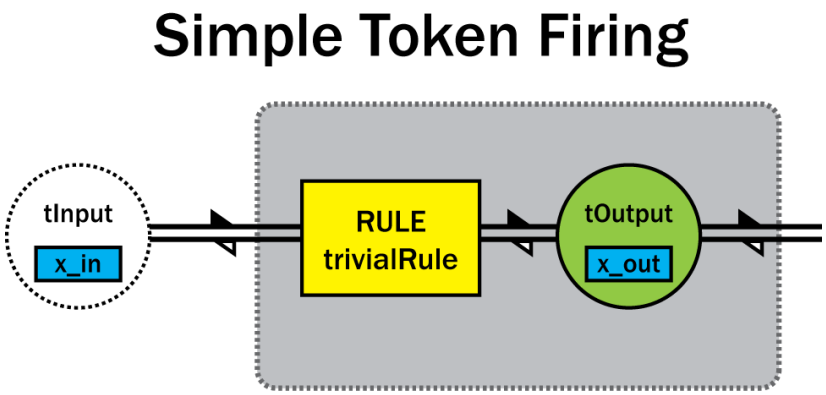
Recall that data has no meaning without an intrinsic attachment to a token, and the design on its own has no notion of input/output – rather, it has only classified two variables as part of the module's interface.

These variables gain meaning depending on their use. When a token is attached to the input of a rule, and that rule then references, for instance, `x_in`, this variable now has state that will be associated with the input token. The same applies to `x_out` – by virtue of creating token `tOutput` in rule `trivialRule`, the variable `x_out` (or, rather, its value) is attached to a token and now has meaning.

The constraints file fills the required token input/output information, as well as data payloads of tokens. The syntax is straightforward – the `input` line lists input tokens with an associated comma-separated list of payload variables in the square brackets. The same goes for the `output` line. Detailed constraints file syntax is explained in section **6. Constraints File**.

```
input = (tInput[x_in])
output = (tOutput[x_out])
```

The design can be graphically represented as:



`tInput` contains the contents of `x_in`, (referred to with the `self` prefix to indicate that this is stateful data passed between rules) so that when rule `trivialRule` fires, it is automatically aware of the token's data payload. The rule increments this value, then attaches it to `tOutput`.

2.2.2. A Small Pipeline

Now we look at a small "pipeline." Of course, this is an extremely trivial and silly example of passing data from one stage to another, but it will highlight one key aspect of how this code is elaborated.

```
from Int import Int,GlobalIn, GlobalOut
class SmallPipeline:
  def __init__(self):
        self.x = GlobalIn(32)
        self.y = GlobalIn(32)
        self.z = GlobalOut(32)

  def addXAndY(self, trigger="tInput"):
        self.z = self.x + self.y
        create(tMulByTwo)

  def mulByTwo(self, trigger="tMulByTwo"):
        self.z = self.z + self.z
        create(tSub)

  def sub(self, trigger="tSub"):
        self.z = self.z - 11
        create(tOutput)
```

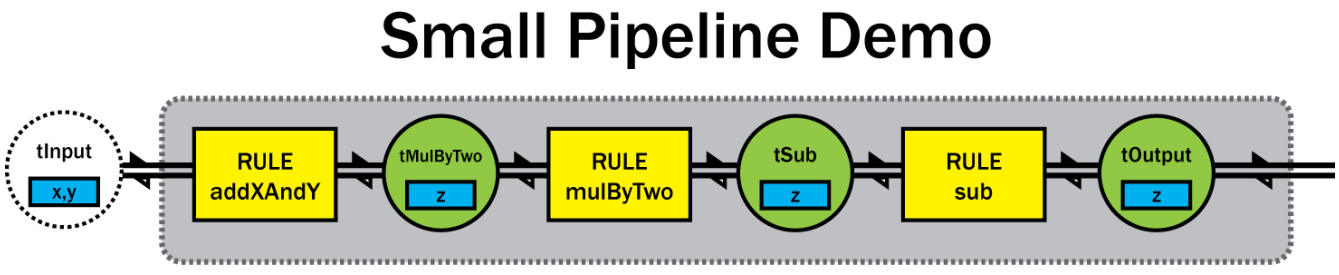
The associated constraints file is (see section **6. Constraints File**):

```
input = (tInput[x, y])
output = (tOutput[z])
```

We can see that `tInput`, with data `x` and `y` in tow, triggers rule `addXAndY`. This rule does what it says – adds `x + y` and stores the result, labeled `z`, into the token `tMulByTwo`. Rule `mulByTwo` is now triggered and multiplies `z` by two, the result of which is stored into token `tSub`. Finally, token `tSub` triggers rule `sub` and its payload `z` is reduced by 11 and attached to token `tOutput`.

This example highlights what separates PyTDL from most other languages – **values of variables in one rule are not the same as values of variables (of the same name) in another. There is no shared or persistent state.** In this case, the value of `z` referenced in rule `sub` is not the same as the value of `z` referenced in `mulByTwo` if observed at the same time. Instead, the latter will pass on its modified variable to the former via the token. Both rules can (and by default will) fire at the same time, since they have distinctly different values and the rule can both fire the token and accept a new input token on the same cycle. In rule `sub`, the assignment operation reads `self.z` from token `tSub`, subtracts 11 from it, and assigns the result to the `z` attached to token `tOutput`.

Oftentimes, it is easy to see what is going on by visualizing the design:



3. Implementation

The four basic elements of PyTDL designs are realized in hardware with SELF buffers and combinational logic. This section introduces both, along with arbiter implementation.

3.1. Tokens, State, and Back-Pressure as SELF Buffers

SELF (Synchronous ELastic Flow) buffers were introduced by Cortadella, *et al.* [Cortadella06] as a pipelining mechanism that allows latency tolerance (on the granularity of clock cycles) in a combinatorially isolated buffer that looks semantically similar to a flip-flop or a FIFO.

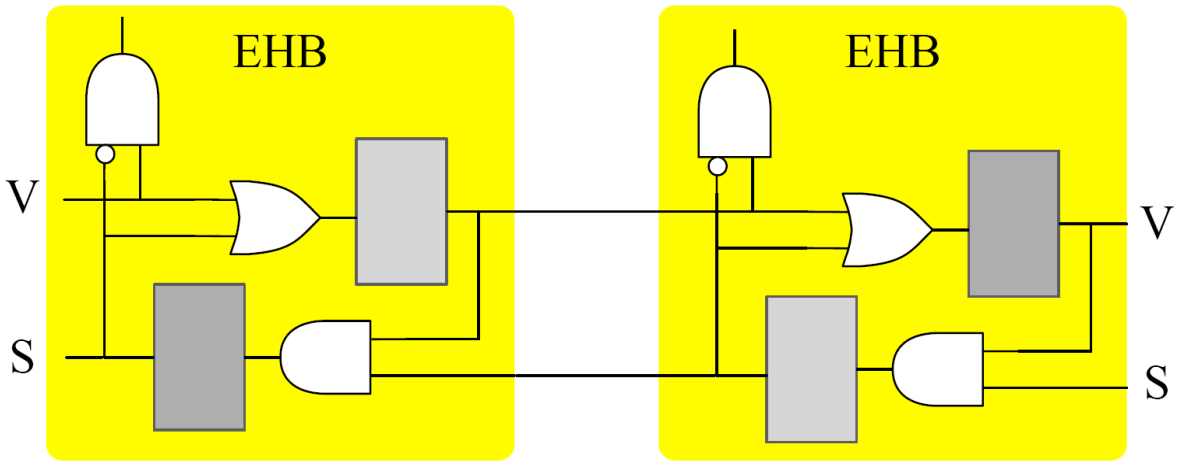


Figure 4: Two SELF Buffers (labeled EHB, Elastic Half-Buffers) chained together. (Source: Hoover06)

The buffers themselves are composed of a pair of master-slave latches and gates dictating valid and stall control signals, along with a register to store the data portion. As demonstrated in Figure 1, these control signals do not introduce chains of combinational logic and can thus be scaled up arbitrarily.

The mapping from SELF buffer to token is, by default, one-to-one. The definition of a token in this context is the assertion of a valid signal and its associated payload data stored in the register. Back-pressure is defined as the assertion of the stall signal; if there is a valid token in the SELF buffer, the data is kept in place until it can pass on to its destination.

The combinational isolation property means rules can live in their own control domains. See **4. Control Region Assignment**.

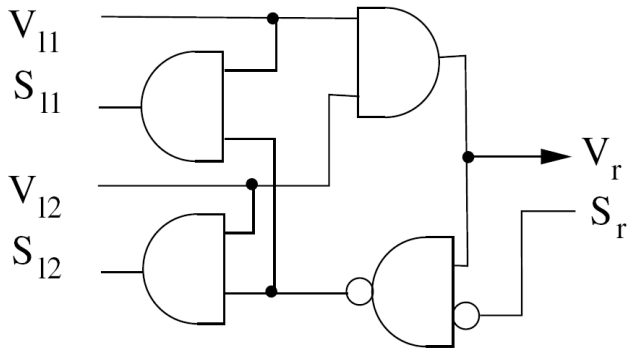


Figure 6: Cortadella's SELF join requires both V11 and V12 to be asserted in order for the output Vr to assert. Source: Cortadella06

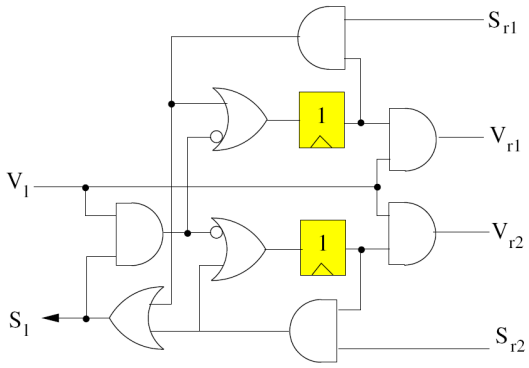


Figure 5: SELF eager fork: V1 and S1 are inputs and are greedily forked into the two valid, stall pairs. Source: Cortadella06

3.2. Arbiter Implementation

Cortadella *et al.* also introduced the aforementioned fork and join arbiters. Figures 5 and 6 show the two, respectively. These are instantiated automatically based on the trigger conditions of rules, as described in section **2.1.5. Arbitration**.

It's important to note that PyTDL implements Select arbiters (which nondeterministically choose one token from potentially many) using a fixed, random priority. This can be overridden via the constraints file. See **6.4. Setting Select Arbiter Priorities** for details.

3.3. Rules as Combinational Logic

Rules are atomic pure functions and have a direct mapping to combinational logic. If a rule, for instance, performs arithmetic on the input data, this elaborates to the appropriate high-level Verilog operation. The resulting HDL is behaviorally described to allow the synthesis toolchain to determine actual implementation. E.g. the type of carry propagation adder may be dependent on the target technology, and thus the most suitable choice may be used. It is left to the designer to make any low-level decisions beyond behavioral descriptions.

3.3.1. Intermediate Stateless Variables

PyTDL borrows one of Verilog's semantics of temporary variables for the sake of readability. As an aside, it is important to note that **it is not advisable to rely on the underlying implementation beyond what is described in this section;** writing platform-dependent code takes away from the robustness of the design.

For the sake of code brevity and neatness, PyTDL allows the designer to use temporary variables within a rule. The following rule does not instantiate any state or SELF buffers for the stateless variables `bar` and `baz`. Note that the declarations are required.

```
rule readableRule(self, trigger="tFoo"):
  bar = Int(32)
  baz = Int(32)
  bar = self.x + self.y
  baz = bar[8:16] + bar[0:8]
  self.z = baz
```

The best way to think about the code behavior is to backwards substitute the variables. Recall that bit slicing is distinct from Verilog bus select (covered in **5.4.2. Bit Slicing**). The above rule then reduces to code that is behaviorally identical to the following:

```
rule unreadableRule(self, trigger="tFoo"):
  self.z = ((self.x + self.y) >> 16) + ((self.x + self.y) & 0xFFFF)
```

It is clear that body of rule `readableRule` is much more readable than the above.

A possible downside to this syntactic shortcut is that assignment operations are now sensitive to ordering; proper division of the problem into small, manageable rules gets past this issue when coupled with control region assignment.

3.4. Inputs and Outputs

From a high level, any token and its payload can be used as either an input or an output. However, due to the tools current limitations, and the ambiguity of dealing with data of the same label in multiple places, it is required (although not enforced by the tools) that outputs and inputs use dedicated variables. They do not add any additional resource cost, since any data used by a rule is going to automatically be buffered.

It is not syntactically correct to use a variable declared as an I/O internally within the design.

4. Control Region Assignment

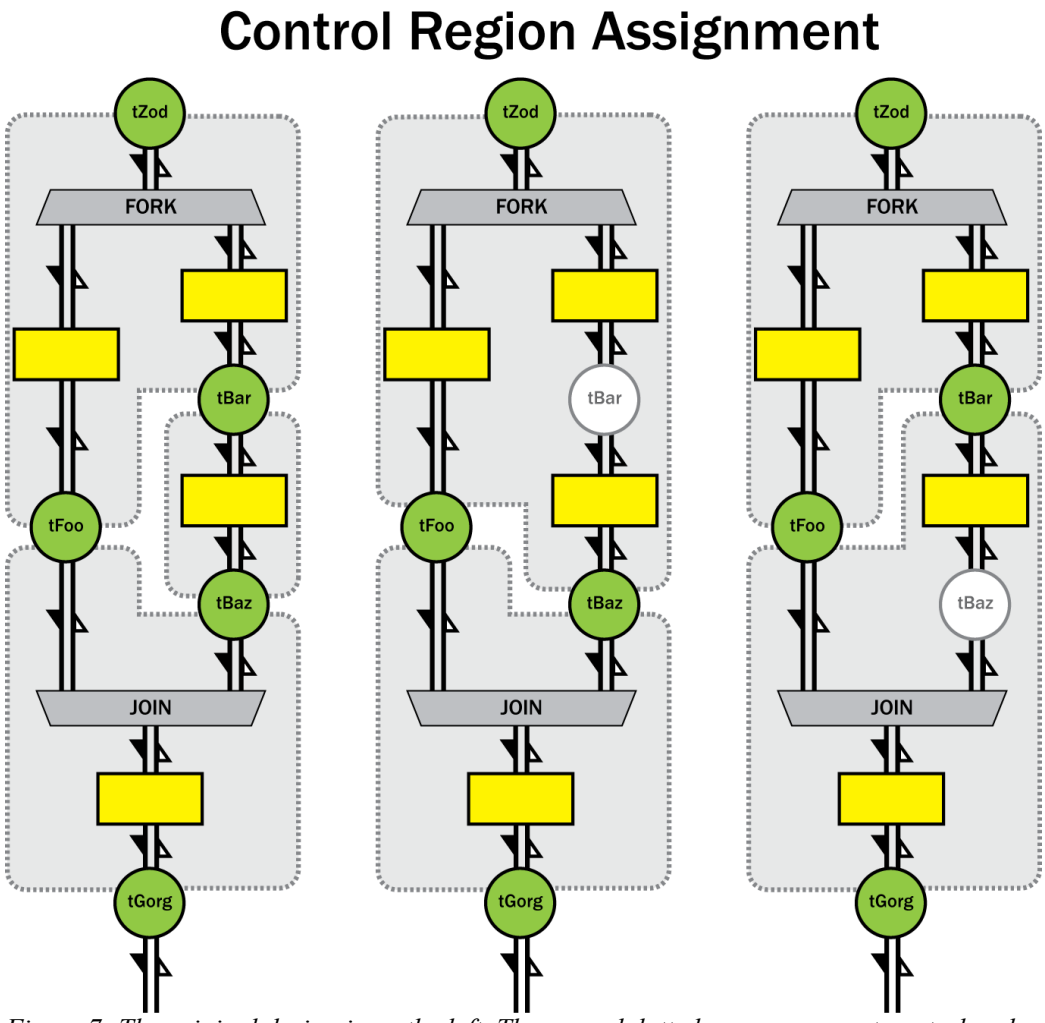


Figure 7: The original design is on the left. The grayed dotted areas represent control and timing domains. No combinational loops cross these boundaries. The middle design demonstrates merging the top two regions by "shorting" token tBar. The right image shows the merger of the lower two regions by shorting tBaz.

The philosophy behind the design flow of PyTDL is to first get the design behaviorally correct and verified, then to focus on incremental optimizations. To this end, the default behavior of PyTDL focuses solely on functionality of a latency tolerant circuit by assigning SELF buffers to every token link. However, this approach has a high resource cost. Since the semantic execution model and language force the designer to make inherently latency tolerant designs, it is possible to get around the resource cost by removing freedom *without affecting behavior*.

This freedom is based on the concept of a **control region**: a control region is a spacial and logical region where interdependencies exist, both timing (sharing the same clock) and wire/gate dependencies (in the form of combinational logic loops).

By default, each rule lives in its own isolated control region. This one-to-one mapping of rules to control domains can lead to situations where the high cost of SELF buffers dominates resources. The designer ends up with each intermediate piece of data attached to tokens, and the associated logic for valid/stall pairs.

If instead, the designer *assigns* rules to specific regions, then an application-appropriate mapping from rules in the design to control regions can lead to a more efficient implementation.

Thus, control region assignment is the post-design process of "shorting" tokens. Shorting tokens means replacing SELF buffers with logic equivalent gates and wires – valid in is wired to valid out, stall in is wired to stall out, and the register containing the data portion of the token is replaced with wires from the D input to the Q output. The result is that rules on either end of the shorted gate now combine into the same timing and control domain.

In essence, shorting provides a mechanism for the designer to trade off between long combinational loops and the resource cost of SELF buffers.

This process can be iteratively continued until there are only a handful of actual SELF buffers; during the entire process, if the designer conformed to the latency-tolerant semantic model in the original design, the circuit will remain functionally identical. The difference would be long logic loops of combined rules and different timing characteristics.

Syntax of how to short tokens is covered in section **6.3. Control Region Assignment Syntax**.

5. Syntax Details

The constraints file syntax and any PyTDL syntax not yet covered by the above sections will be described in this section. For the latter's details, we defer to the official Python documentation (**http://www.python.org/doc/2.5.2/**).

5.1. Whitespace

Python, and thus PyTDL, is whitespace-sensitive; code blocks are specified by the whitespace level of their member statements. For instance, conditional if blocks are grouped together if they have the same whitespace level. Whitespace formally includes spaces and tabs for indentation; newlines separate statements except where the parser can unambiguously determine statement continuity. For more information, see section 2.1 in the Python language reference.

5.2. Comments

Comments begin with a hash symbol (#) and include everything until the next line. They may begin anywhere after a valid statement, after whitespace, or take up the entire line.

5.3. Constants

Constants can be expressed as either decimal literals, which is simply the decimal number, or as hexadecimal literals following standard C-style syntax: `0xABCD0123`, for example.

5.4. Supported Operations

5.4.1. Arithmetic and Bitwise Operations

Table 1 lists the supporting unary and binary operations from highest to lowest precedence.

Operator	Syntax
Highest Precedence	
Bitwise Invert	`~a`
Logical Not	`!a`
Multiplication	`a * b`
Addition	`a + b`
Subtraction	`a - b`
Left Shift	`a << b`
Right Shift	`a >> b`
Comparison Greater Than	`a > b`
Comparison Greater Than or Equal To	`a >= b`
Comparison Less Than	`a < b`
Comparison Less Than or Equal To	`a <= b`
Comparison Equality	`a == b`
Comparison Inequality	`a != b`
Bitwise And	`a & b`
Bitwise Xor	`a ^ b`
Bitwise Or	`a | b`
Logical And	`a && b`
Logical Or	`a || b`
Lowest Precedence	

Table 1: Supported operators with corresponding precedence

5.4.2. Bit Slicing

Bit slicing/selection is a built-in operator, but works according to the Python slice operation and **not** the Verilog semantic. To select bits, use the following syntax form:

variableName`[`*lowerBoundInclusive* `:` *upperBoundExclusive*`]`

where *lowerBoundInclusive* ≤ *upperBoundExclusive*. The Verilog equivalent would be:

```
variableName[upperBoundExclusive-1:lowerBoundInclusive]
```

By default, the Verilog output will instantiate all vectors as directly mapped indices. The highest index corresponds to the most signifant bit.

To illustrate the PyTDL method, the following results in `bar` containing the lower 16 bits of `foo`, and `baz` containing the upper 16 bits of `foo`:

```
foo = Int(32)
bar = foo[0:16]
baz = foo[16:32]
```

PyTDL does not support arbitrary bit index assignment. The indices represent bit position directly – that is, index 0 is the LSB, and the highest numbered index is the MSB. For further clarification, the Verilog equivalent of the above is:

```
bar = foo[15:0]
baz = foo[31:16]
```

5.5. Statements

5.5.1. Conditional Statements

PyTDL rules can contain blocks of nested conditional statements. The general syntax is:

if *expression*:
<whitespace>*statementlist*
[**elif** *expression*:
<whitespace>*statementlist*
[**elif** *expression*:
...]]
[**else**:
<whitespace>*statementlist*]

Conditionals are evaluated atomically. Conflicting, non-mutually exclusive statements will prioritize the last assignment operation:

```
if self.x > 5:
  self.y = 1
if self.x > 10:
  self.y = 2
```

In the case that `self.y` is 11, both conditions evaluate to true and the last assignment to 2 is accepted.

Tokens can be generated conditionally by putting the `create` keyword inside one of the branches. The condition is simply prefixed to the valid input of the token.

5.6. Data Types

Python is a dynamically typed language; in the context of generating hardware, however, this relaxed typing scheme is difficult to realize efficiently. Thus, all data types must be declared by the user as some variation of bit vectors. These are defined using data class types, and are declared using instantiations of these classes.

5.6.1. Defining Basic Properties

Data classes define a base set of methods that return properties of the data type; they are called by the PyTDL tools during synthesis – that is, they are pragmatic, non-synthesizable components of the

design. These methods have names with two underscores surrounding them, mimicking Python's convention (e.g. `__value__`). We will explain these types by showing the implementation of a generic integer used in the tutorials and code examples of this document.

First, in order to use the integer class, the designer must import the appropriate package (defined by filenames of the same name). In this case, the `Int` package is contained in a file named `Int.py`. Hence the statement:

```
from Int import Int, GlobalIn, GlobalOut
```

This states that PyTDL should search the file `Int.py` (the `.py` is appended to the package name), and pull the classes `Int`, `GlobalIn`, and `GlobalOut` into the current namespace.

Put another way, it imports the three comma-separated classes listed from the package `Int`.

More generally, the designer may use one of three methods of importing classes:

1. Import a specified list of classes from a package:

   from *PackageName* **import** *ClassName*[, *ClassName*[,...]]

2. Import all classes from a package:

   from *PackageName* **import ***

3. Import the namespace, prefixing each instantiation with the package name:

   import *PackageName*
   ...
   variableName **=** *PackageName*.*ClassName***()**

See **5.6.2. Instantiating/Declaring Data Types**.

Now we look at the source of the file.

```
class Int:
    def __init__(self, size=1, value=0):
        self.size = size
        self.value = value

    def __size__(self):
       return self.size

    def __value__(self):
        return self.value

    def __coherence__(self):
        return "none"

class GlobalOut(Int):
    def __output__(self):
        return True

class GlobalIn(Int):
    def __input__(self):
        return True
```

There are three simple data type classes here which implement several different built-in methods.

In general, data classes can implement any of the built-ins in Table 2 describing the characteristics of any instantiation of this class. We can see that the above integer example contains at least one class implementing each method.

Method Name	Description
`__value__`	Returns the storage value of this data type.
`__size__`	Returns an integer of the size, in bits, of the data
`__coherence__`	Not supported.
`__input__`	Returns a boolean value indicating that this data is an input.
`__output__`	Returns a boolean value indicating that this data is an output.

Table 2: Data class built-in methods

5.6.2. Instantiating/Declaring Data Types

Once a data type class has been formed, the designer may instantiate the data. The location of the instantiation depends on what the purpose of the data is:

1. Stateful, Token-linked Variables: Declared in the `__init__` routine of the main class.
2. Stateless Temporary Variables: Declared in whatever rule they are used, and must be declared before they are used.

The syntax of the declaration for stateful data is:

`self``.`*variableName* `=` [*PackageName*`.`]*DataTypeClass*`(`[*parameter1*[`,` *parameter2*[`,...`]]]`)`

Where as stateless temporary variables omit the prefixing self object:

variableName `=` [*PackageName*`.`]*DataTypeClass*`(`[*parameter1*[`,` *parameter2*[`,...`]]]`)`

See above **5.6.1. Defining Basic Properties** for details on when the *PackageName* is necessary.

5.6.3. Extending Data Types

In addition to the built-in methods, classes may implement their own pure functions. These are read-only functions that can help structure the code; for instance, the MSP430 example contains an `Instruction` data type containing decoding specific functions such as `single_opcode`. These simply return bit slices or function results dependent on some part of the data.

Extended functions must be of the form:

```
def function_name(self, size=integerSize):
    ...
    return [...]
```

The function name must be any valid Python identifier, and the prototype is required to be in the form shown. The `size` attribute is a required constant (specified syntactically like a Python parameter with a default value, but semantically different) specifying how many bits the result of this function is.

To demonstrate how to implement such a method, below is a very simple 32-bit IEEE 754 floating point data type:

```
class Float32:
  def __init__(self):
        self.size = 32

  def __size__(self):
        return self.size

  def __coherence__(self):
        return "none"

  def sign(self, size=1):
        return self[32]

  def exponent(self, size=8):
        return self[23:31] - 127

  def significand(self, size=23):
        return self[0:23]
```

Beyond the standard built-in property methods, there are three added functions which either return a bit slice of the internal data, or in the case of `exponent`, returns the unbiased exponent. Note that the bit slicing is not the same as Verilog, as covered in the bit-slicing section in section **5.4.2. Bit Slicing**.

6. Constraints File

6.1. Basics

The constraints file is a list of inputs, outputs, and shorted tokens. Each specification must be on its own line. Whitespace (spaces and tabs) is ignored before and after the lines, and blank lines are permitted.

Comments follow the same rules as PyTDL designs – namely, a hash symbol (`#`) followed by the comment that can extend until the next new line character.

6.2. Inputs and Outputs

The constraints file describes input and output tokens and their corresponding data payloads; the former will appear as valid/stall pairs in the top-level Verilog module, while the latter are standard Verilog input and output buses.

The syntax for specifying input tokens and data is:

`input = (`[*tokenName*[**`[`***attachedVariable*[**`,`** *attachedVariable*, ...]]**`]`**`,` ...]**`)`**

The keyword `input` followed by an equals sign and a comma-separated list of token-data combinations. Each token-data combination consists of the token's name along with an *optional* comma-separated variable list within square brackets.

Output tokens and data are specified in the same way, with the output keyword instead:

`output = (`[*tokenName*[**`[`***attachedVariable*[**`,`** *attachedVariable*, ...]]**`]`**`,` ...]**`)`**

The designer may spread out all of the tokens over multiple input and output specifications, so long as each list contains a mutually exclusive set of tokens. For example, if a design had four input tokens `tFoo`, `tBar`, `tBaz`, and `tZod`, each attached to data `fooData`, `barData`, `bazData`, and `zodData`, respectively, the constraints file may be partitioned as follows:

```
input = (tFoo[fooData], tBar[barData])
input = (tBaz[bazData], tZod[zodData])
```

6.3. Control Region Assignment Syntax

Control regions are assigned by “shorting” tokens – this amounts to connecting to the valid input signal to the valid output signal, and the stall input signal to the stall output signal. The data register is then replaced with wires from the D inputs to the Q outputs, hence the term “shorting.”

In order to select which tokens to short, simply add one or more lines in the constraints file similar to the above input and output specifications:

`short = (`[*token1*[**`,`** *token2* [**`,`** ...]]**`)`**

Simply follow the short keyword with an equals sign and a parenthesized list of token names.

Examples:

```
short = (tFoo, tBar, tZod)
```

6.4. Setting Select Arbiter Priorities

Select arbiters are by design semantically nondeterministic. However, when PyTDL generates the output design, it randomly chooses a deterministic ordering – as of this writing, the Select arbiters have a fixed priority such that for the n inputs, input i_0 will have the highest priority, i_1 the second highest, and so on with i_{n-1} having the lowest priority.

While designs should be tolerant to any ordering of tokens on the output, oftentimes the nature of the problem dictates that tokens have a specific ordering. To this end, PyTDL provides the ability to prioritize tokens through the constraints file.

To specify the priorities, insert a list (on its own line) of tokens in decreasing priority surrounded by parentheses:

(*tokenHighest*, *tokenSecondHighest*[, ...])

This will tell the compiler to use this priority ordering if any of those tokens become inputs to Select arbiters.

7. References

1. G. Hoover, "Distributed Control For Embedded System Design," Ph.D. dissertation, University of California, Santa Barbara, CA, 2008
2. J. Cortadella, M. Kishinevsky, and B. Grundmann. "Synthesis of synchronous elastic architectures." In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 657–662, New York, NY, USA, 2006. ACM Press.
3. J. Cortadella and M. Kishinevsky. "Synchronous elastic circuits with early evaluation and token counterflow." In *DAC '07: Proceedings of the 44th annual conference on Design automation*, 2007.
4. J. Cortadella, M. Kishinevsky, and B. Grundmann. "Self: Specification and design of synchronous elastic circuits." In *TAU '06: Proceedings of the ACM/IEEE International Workshop on Timing Issues 2006* , 2006.
5. G. Hoover, T. Sherwood, and F. Brewer. "Towards understanding architectural tradeoffs inmems closed-loop feedback control." In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems,* 2007.
6. F. Brewer, G. Hoover, and C. Gill. "Latency-insensitive hardware/software interfaces." In *MEMOCODE '08: Proceedings of the Sixth ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, 2008.
7. G. Hoover and F. Brewer. "Synthesizing synchronous elastic flow networks." In *Design, Automation and Test in Europe, 2008. Proceedings*, 2008.
8. S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. "Synchronous elastic networks." In *FMCAD '06: Proceedings of the Formal methods in computer aided design, pages 19–30*, 2006.
9. M. Kishinevsky, J. O'Leary, S. Krstic, and J. Cortadella. "Synchronous elastic networks." In *Formal Methods in Computer Aided Design (FMCAD'06)*, 2006.