Python Streams User Manual

Jonathan P Dawson

December 29, 2010

Contents

Chapter 1

Introduction

1.1 A new approach to device design

Traditionaly, the tool of choice for digital devices is a Hardware Description Language (HDL), the most common being Verilog and VHDL. These languages provide a reasonably rich environment for modeling and simulating hardware, but only a limitied subset of the langauge can be realised in a digital device (synthesised).

A software designer would typically implement a function in an imperative style using loops, branches and subprocedures, but hardware models written in an imperative style cannot be synthesised.

Synthesisable designs require a different approach. Digital device designers must work at the Register Transfer Level (RTL). The primitive elements of an RTL design are clocked memory elements (registers) and combinational logic elements. A typical synthesis tool would be able to infer boolean logic, addition, subtraction, multiplexing and bit manipulation from HDL code written in a very specific style.

An RTL designer has to work at a low level of abstraction. In practical terms this means that a designer has to do more of the work themselves.

1. A designer is reponsible for designing their own interfaces to the outside world.
2. The designer is responsible for clock to clock timing, manualy balancing propogation delays between clocked elements to achieve high performance.
3. A designer has to provide their own mechanism to synchronise and pass data bwetween concurrent computational elements (by implementing a bus with control and handshaking signals).
4. A designer has to provide their own mechanism to control the flow of execution within a computational element (usually by manually coding a finite state machine).
5. The primitive elements are primitive.

This is where Python Streams comes in. In Python Streams, there is no synthesiseable subset, but a standalone synthesisable language built on top of

Python. Python streams allows designers to work at a higher level of abstraction. It does a lot more of the work for you.

1. Python streams provides a suite of device interfaces including portio, uart, usb and ethernet.
2. Synthesisable RTL code is generated automatically by the tool. Clocks, resets, and clock to clock timing are all taken care of behind the scenes.
3. Python Streams provides a simple method to synchronise concurrent elements, and to pass data between them - streams. The tool automatically generates interconnect busses and handshaking signals behind the scenes.
4. Python streams provides imperative style sequeneces branches and loop. The tool automatically generates state machines, or highly optimized soft-core processors behind the scenes.
5. The primitive elements are not so primitive. Common constructs such as counters, lookup tables, ROMS and RAMS are invoked with a single keyword and a few parameters. Python Streams also provides a richer set of arithmetic operators including fully synthesiseable division and multiplication.

1.2 A language within a language

Python Streams is a python library, just an add-on to Python which is no more or less than a programming language. The Python Streams library provides an Application Programmers Interface (API) to a suite of hardware design functions.

The Python Streams library can also be considered a language in its own right, The Python language itself provides statements which are executed on your own computer. The Python Streams provides an alternative language, statements which are executed on the target device.

Chapter 2

The Concepts

Chapter 3

The Language

3.1 Systems

3.2 Streams

3.2.1 Counter(Start, Stop, Step)

This versatile Stream emits numbers in the sequence $[start, start+step, start+2*step, ..., stop, start, ...]$.

```
Example
Counter(1, 3, 1)
=> [1, 2, 3, 1, 2, 3, 1, 2, 3, ...,]
Counter(3, 1, -1)
=> [3, 2, 1, 3, 2, 1 ...,]
```

3.2.2 InPort(name, bits)

The `InPort` Stream emits numbers aquired from a physical io port in a target device.

`name` is a string parameter specifying a name to identify the port in the target implementation.

Since it is not possible to automatically determine the number of bits needed, the width of the port must be explicitly set using the `bits` parameter.

No handshaking is implemented, the value emited by an `InPort` Stream is the value present on the io port at the time the stream transaction completes. The `InPort` Stream will automatically generate logic to synchronise the io port to the local clock domain.

Example:

```
Example
dip_switch = InPort("sw0", 8)
minimal_system = System(OutPort("led0", dip_switch))
```

3.2.3 Output()

While inputs to a process are created implicitly by reading from a stream, process outputs must be explicitly created by creating an `Output`.

```
Example
output1 = Output()
output2 = Output()
data = Variable(0)
Process(bits,
  Loop(
    stream.read(data),
    stream.write(output1),
    stream.write(output2),
  )
)
```

3.2.4 Repeater(value)

The `Repeater` Stream emits `value` repeatedly.

```
Example
a = Repeater(1) + Repeater(1)
=> [2, 2, 2, 2, 2, 2, 2 ...]
```

3.2.5 Scanner()

3.2.6 Sequence(seqeunce)

The `Sequence` Stream emits each item in `sequence` in turn. When the end of the sequence is reached, the whole process repeates starting again from the first item.

```
Example
Sequence(0, 1, 1, 3, 4)
=> [0, 1, 1, 3, 4, 0, 1, 1 ...]
```

3.2.7 SerialIn

3.2.8 Stimulus

3.3 Stream Combinators

3.3.1 Array

3.3.2 Decoupler

3.3.3 HexPrinter

3.3.4 Lookup

3.3.5 Printer

3.3.6 Process

3.3.7 Resizer

3.4 Stream Expressions

A Stream Expression can be formed by combining Streams or Stream Expressions with the following ooperators:

```
+, -, *, \/, %, &, |, ^, <<, >>, ==, !=, <, <=, >, >=
```

Each data item in the resulting Stream Expression will be evaluated by removing a data item from each of the operand streams, and applying the operator function to these data items.

Generaly speaking a Stream Expression will have enough bits to contain any possible result without any arithmetic overflow. The one exception to this is the left shift operator where the result is always truncated to the size of the left hand operand. Stream expressions may be explicitly truncated or sign extended using the `Resizer` (section 3.3.6).

If one of the operands of a binary operator is not a Stream, Python Streams will attempt to convert this operand into an integer. If the conversion is successfull, a `Repeater` (section 3.2.3) stream will be created using the integer value. The repeater stream will be used in place of the non-stream operand. This allows constructs such as `a = 47+InPort(12, 8)` to be used as a shorthand for `a = Repeater(47)+InPort(12, 8)` or `count = Counter(1, 10, 1)+3*2` to be used as a shorthand for `count = Counter(1, 10, 1)+Repeater(5)`. Of course `a=1+1` still yields the integer 2 rather than a stream.

The operators provided in the Python Streams library are summarised in the table below. The bit width field specifies how many bits are used for the result based on the number of bits in the left and right hand operands.

3.4.1 operator precedence

The operator precedence is inherited from the python language. The following table summarizes the operator precedences, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same row have the same precedence.

Operator	**Function**	**Bit Width**
`+`	Signed Add	$max(left, right) + 1$
`-`	Signed Subtract	$max(left, right) + 1$
`*`	Signed Multiply	$left + right$
`//`	Signed Floor Division	$max(left, right) + 1$
`%`	Signed Modulo	$max(left, right)$
`&`	Bitwise AND	$max(left, right)$
`|`	Bitwise OR	$max(left, right)$
`^`	Bitwise XOR	$max(left, right)$
`<<`	Arithmetic Left Shift	$left$
`>>`	Arithmetic Right Shift	$left$
`==`	Equality Comparison	1
`!=`	Inequality Comparison	1
`<`	Signed Less Than Comparison	1
`<=`	Signed Less Than or Equal Comparison	1
`>`	Signed Greater Than Comparison	1
`>=`	Signed Greater Than or Equal Comparison	1

Table 3.1: Operator summary

Operator	**Description**
`==, !=, <, <=, >, >=`	Comparisons
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<, >>`	Shifts
`+, -`	Addition and subtraction
`*, //, %`	multiplication, division and modulo

Table 3.2: Operator precedence

3.5 Sinks

3.5.1 Asserter

3.5.2 Console

3.5.3 OutPort

3.5.4 Response

3.5.5 SerialOut

3.5.6 SVGA

3.6 Processes

3.6.1 Output

3.6.2 Variable

3.6.3 VariableArray

3.7 Process Statements

3.7.1 Block

3.7.2 Break

3.7.3 Constant

3.7.4 Continue

3.7.5 DoUntil

3.7.6 DoWhile

3.7.7 Evaluate

3.7.8 If

3.7.9 Loop

3.7.10 Print

3.7.11 Scan

3.7.12 Until

3.7.13 Value

3.7.14 Variable

3.7.15 WaitUs

3.7.16 While

3.7.17 Expressions

3.8 Process Expressions

3.9 Patterns of Use