Chips Documentation

Release 0.1

Jonathan P Dawson

April 14, 2011

CONTENTS

1 Introduction **3**
- 1.1 A new approach to device design . . . 3
- 1.2 A language within a language . . . 4

2 Tutorial **5**
- 2.1 Learn Python . . . 5
- 2.2 Install Chips . . . 5
- 2.3 First Simulations . . . 5
- 2.4 Hello World . . . 6
- 2.5 Generating VHDL . . . 6
- 2.6 More Streams . . . 7
- 2.7 Introducing Processes . . . 8

3 Chips Language Reference Manual **9**
- 3.1 Chip . . . 9
- 3.2 Process . . . 9
- 3.3 Streams . . . 12
- 3.4 Sinks . . . 18
- 3.5 Instructions . . . 20

4 Automatic Code Generation **25**
- 4.1 VHDL Code Generation . . . 25
- 4.2 C++ Code Generation . . . 25
- 4.3 Visualisation Code Generation . . . 25

5 IP library **27**

6 Extending the Chips Library **29**

7 Indices and tables **31**

Module Index **33**

Index **35**

Contents:

CHAPTER

ONE

INTRODUCTION

The Chips library gives Python the ability to design, simulate and realise digital devices such as FPGAs. Chips provides a simple yet powerful suite of primitive components, *Streams*, *Processes* and *Sinks* that can be succinctly combined to form *Chips*. The *Chips* library can automatically convert *Streams*, *Processes* and *Sinks* into a Hardware Description Language, which can be synthesised into real hardware.

Python programs cannot themselves be converted into real hardware, but it is possible to programmatically generate which construct *Chips*, which can in-turn be converted into hardware. When combined with the extensive libraries already supported by Python, such as NumPy and SciPy, Python and Chips make the ideal design and verification environment.

1.1 A new approach to device design

Traditionally, 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 limited subset of the language can be realised in a digital device (synthesised).

While a software designer would typically implement a function in an imperative style using loops, branches and sub procedures; a hardware model written in an imperative style cannot be synthesised.

Synthesizable 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 responsible for designing their own interfaces to the outside world.
2. The designer is responsible for clock to clock timing, manually balancing propagation delays between clocked elements to achieve high performance.
3. A designer has to provide their own mechanism to synchronise and pass data between 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. Synthesis tools provide limited support for multiplication, and division is not usually supported at all.

This is where *Python Chips* comes in. In *Python Chips*, there is no synthesizable subset, but a standalone synthesizable language built on top of Python. *Python Chips* allows designers to work at a higher level of abstraction. It does a lot more of the work for you.

1. *Python Chips* provides a suite of device interfaces including I/O ports and USARTs.
2. Synthesizable 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 Chips* provides a simple method to synchronise concurrent elements, and to pass data between them - streams. The tool automatically generates interconnect buses and handshaking signals behind the scenes.
4. *Python Chips* provides processes with imperative style sequences 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 Chips* also provides a richer set of arithmetic operators including fully synthesizable division and multiplication.

1.2 A language within a language

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

The *Python Chips* 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 Chips* provides an alternative language, statements which are executed on the target device.

CHAPTER

TWO

TUTORIAL

2.1 Learn Python

In order to make any real use of the *Chips* library you will need to be familiar with the basics of Python. The python tutorial is a good place to start.

2.2 Install Chips

2.2.1 Windows

1. First install python. You need Python 2.6 or later, but not Python 3.
2. Then install the chips library from the windows installer.

2.2.2 Linux

1. First install python. You need Python 2.6 or later, but not Python 3.
2. Then install the chips library from the source distribution:

```
desktop:~$ tar -zxf Chips-0.1.tar.gz
desktop:~$ cd Chips-0.1
desktop:~$ python setup.py install #run as root
```

2.3 First Simulations

Once you have python and chips all set up, you can start with some simple examples. This one counts to 10 repeatedly:

```
from streams import *

#create a chip model
my_chip = Chip(
  Console(
    Printer(
        Counter(0, 10, 1),
    ),
  ),
```

```
)

#run a simulation
my_chip.reset()
my_chip.execute(100)
```

The example can be broken down as follows:

- `from stream import *` adds the basic features of the streams library to the local namespace.
- A *Chip* models a target device. You need to tell it what the outputs (*sinks*) are, but it will work out what the inputs are by itself. In this case the only *sink* is the *Console*.
- A *Console* is a sink that outputs a stream of data to the console. The only argument it needs is the data stream, *Printer*.
- A *Printer* is a *stream* object that represents a stream of data in decimal format as a string of ASCII characters. A *Printer* is not a source of data in itself, it transforms a stream of data that you supply, the *Counter*.
- The *Counter* is a fundamental data stream. It accepts three arguments: start, stop and step. The *Counter* will yield a stream of data counting from *start* to *stop* in *step* increments.

2.4 Hello World

No language would be complete without a "hello world" example:

```
from chips import *

#convert string into a sequence of characters
hello_world = tuple((ord(i) for i in "hello world\n"))

my_chip = Chip(
    Console(
        Sequence(*hello_world),
    )
)

#run a simulation
my_chip.reset()
my_chip.execute(100)
```

In this example we have made only a few changes:

- `hello_world = tuple((ord(i) for i in "hello world\n"))` creates a tuple containing the numeric values of the ascii characters in a string.
- This example introduces a new stream, the *Sequence*. The *Sequence* stream outputs each of its arguments in turn, when the arguments are exhausted, the process repeats.
- A *Printer* is *stream* is not needed in this example since the stream is already a sequence of ASCII values.

2.5 Generating VHDL

Now lets consider how the "hello world" example could be implemented in an actual device. A first step to implementing a device would be to generate a VHDL model:

```
from chips import *
from chips.VHDL_plugin import Plugin

#convert string into a sequence of characters
hello_world = tuple((ord(i) for i in "hello world\n"))

my_chip = Chip(
    Console(
        Sequence(*hello_world),
    )
)

#generate a VHDL model
code_generator = Plugin(project_name="hello world")
my_chip.write_code(code_generator)
```

The *Chips* library uses plugins to generate output code from models. This means that new code generators can be added to Chips without having to change the way that hardware is designed and simulated. At present, chips supports C++ and VHDL code generation, but it is VHDL code that allows *Chips" to be synthesised.

The VHDL code generation plugin is found in `chips.VHDL_plugin` If you run this example you should find hello_world.vhd has been generated. You can run this code in an external vhdl simulator, but you won't be able to synthesise it into a device because real hardware devices don't have a concept of a *Console*.

To make this example synthesise, we need to write the characters to some realisable hardware interface. The *Chips* library provides a *SerialOut* sink, this provides a simple way to direct the stream of characters to a serial port:

```
from chips import *
from chips.VHDL_plugin import Plugin

#convert string into a sequence of characters
hello_world = tuple((ord(i) for i in "hello world\n"))

my_chip = Chip(
    SerialOut(
        Sequence(*hello_world),
    )
)

#generate a VHDL model
code_generator = Plugin(project_name="hello world")
my_chip.write_code(code_generator)
```

Now you should have a hello_world.vhd file that you can synthesise in a real device. By default, SerialOut will assume that you are using a 50 MHz clock and a baud rate of 115200. If you need something else you can use the clock_rate and baud_rate arguments to specify what you need.

2.6 More Streams

So far we have seen three types of streams, *Counter*, *Sequence* and *Printer*. Chips provides a few more basic streams which you can read about in the Language Reference Manual. It is also possible to combine streams using arithmetic operators : `+, -, *, //, %, <<, >>, &, |, ^, ==, !=, <, <=, >, >=` on the whole they have the same (or very similar) meaning as they do in Python except that they operate on streams of data.

2.7 Introducing Processes

CHAPTER

THREE

CHIPS LANGUAGE REFERENCE MANUAL

3.1 Chip

class **Chip** (**args*)

A Chip device containing streams, sinks and processes.

Typically a Chip is used to describe a single device. You need to provide the Chip object with a list of all the sinks (device outputs). You don't need to include any process, variables or streams. By analysing the sinks, the chip can work out which processes and streams need to be included in the device.

Example:

```
switches = InPort("SWITCHES", 8)
serial_in = SerialIn("RX")
leds = OutPort(switches, "LEDS")
serial_out = SerialOut("TX", serial_in)

#We need to tell the *Chip* that *leds* and *serial_out* are part of
#the device. The *Chip* can work out for itself that *switches* and
#*serial_in* are part of the device.

s = Chip(
    leds,
    serial_out,
)

s.write_code(plugin)
```

3.2 Process

Processes are used to define the programs that will be executed in the target *Chip*. Each *Process* contains a single program made up of instructions. When a *Chip* is simulated, or run in real hardware, the program within each process will be run concurrently.

3.2.1 Process Inputs

Any *Stream* may be used as the input to a *Process*. Only one process may read from any particular stream. A *Process* may read from a *Stream* using the *read* method. The *read* method accepts a *Variable* as its argument. A *read* from a *Stream* will stall execution of the *Process* until data is available. Similarly, the stream will be stalled, until data is read from it. This provides a handy way to synchronise processes together, and simplifies the design of concurrent systems.

Example:

```
#sending process
theoutput = Output()
count = Variable(0)
Process(16,
    #wait for 1 second
    count.set(1000),
    While(count,
        count.set(count-1),
        WaitUs()
    ),
    #send some data
    theoutput.write(123),
)

#receiving process
target_variable = Variable(100)
Process(16,
    #This instruction will stall the process until data is available
    theOutput.read(target_variable),
    #This instruction will not be run for 1 second
    #..
)
```

3.2.2 Process Outputs

An *Output* is a special *Stream* that can be written to by a *Process*. Only one *Process* may write to any particular stream. Like any other *Stream*, an *Output* may be:

- Read by a *Process*.
- Consumed by a *Sink*.
- Modified to form another *Stream*.

A *Process* may write to an *Output* stream using the *write* method. The *write* method accepts an expression as its argument. A *write* to an output will stall the process until the receiver is ready to receive data.

Example:

```
#sending process
theoutput = Output()
Process(16,
    #This instruction will stall the process until data is available
    theoutput.write(123),
    #This instruction will not be run for 1 second
    #..
)

#receiving process
```

```
target_variable = Variable(0)
count = Variable(0)
Process(16,
    #wait for 1 second
    count.set(1000),
    While(count,
        count.set(count-1),
        WaitUs(),
    ),
    #get some data
    theoutput.read(target_variable),
)
```

3.2.3 Variables

Data is stored and manipulated within a process using *Variables*. A *Variable* may only be accessed by one process. When a *Variable* an initial value must be supplied. A variable will be reset to its initial value before any process instructions are executed. A *Variable* may be assigned a value using the *set* method. The *set method accepts an expression as its argument.

It is important to understand that a *Variable* object created like this:

```
a = Variable(12)
```

is very different from a normal Python variable created like this:

```
a = 12
```

The key is to understand that a *Variable* will exist in the target *Chip*, and may be assigned and referenced as the *Process* executes. A Python variable can exist only in the Python environment, and not in a *Chip*. While a Python variable may be converted into a *Constant* in the target *Chip*, a *Process* has no way to change its value when it executes.

3.2.4 Constants

Like a *Variable*, a constant must be supplied with an initial value when it is created. Unlike a *Variable*, the value of a *Constant* can never be changed.

3.2.5 Expressions

Variables and *Constants* are the most basic form of expressions. More complex expressions can be formed by combining *Constants*, *Variables* and other expressions using following operators:

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

If one of the operands of a binary operator is not an expression, the Chips library will attempt to convert this operand into an integer. If the conversion is successful, a *Constant* object will be created using the integer value. The *Constant* object will be used in place of the non-expression operand. This allows constructs such as `a = 47+Constant(10)` to be used as a shorthand for `a = Constant(47)+Constant(10)` or `count.set(Constant(15)+3*2` to be used as a shorthand for `count.set(Constant(15)+Constant(6)`. Of course `a=1+1` still yields the integer 2 rather than an expression.

An expression within a process will always inherit the data width in bits of the *Process* in which it is evaluated. A *Stream* expression such as `Repeater(255) + 1` will automatically yield a 10-bit *Stream* so that the value 256 can

be represented. A similar expression Constant(255)+1 will give an 9-bit result in a 9-bit process yielding the value -1. If the same expression is evaluated in a 10-bit process, the result will be 256.

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

class `Process` (*bits, *instructions*)

class `Variable` (*initial*)

class `VariableArray` (*size*)

3.3 Streams

Streams are a fundamental component of the *Chips* library.

A stream is used to represent a flow of data. A stream can act as a:

- An input to a *Chip* such as an *InPort* or a *SerialIn*.
- A source of data in its own right such as a *Repeater* or a *Counter*.
- A means of performing some operation on a stream of data to form another stream such as a *Printer* or a *Lookup*.
- A means of transferring data from one process to another, an *Output*.

3.3.1 Stream Expressions

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

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

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.

Generally 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*.

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 successful, a *Repeater* 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("in", 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.

Operator	Function	Data Width (bits)
+	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 Comparison	1

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

3.3.3 Streams Reference

class `Array`(*address_in, data_in, address_out, depth*)

An *Array* is a stream yields values from a writeable lookup table.

Like a *Lookup*, an *Array* looks up each data item in the *address_in* stream, and yields the value in the lookup table. In an *Array*, the lookup table is set up dynamically using data items from the *address_in* and *data_in* streams. An *Array* is equivalent to a Random Access Memory (RAM) with independent read, and write ports.

A *Lookup* accepts *address_in*, *data_in* and *address_out* arguments as source streams. The *depth* argument specifies the size of the lookup table.

Example:

```
def video_raster_stream(width, height, row_stream, col_stream,
                        pixel_intensity):

    pixel_clock = Counter(0, width*height, 1)
    red_intensity, green_intensity, blue_intensity = pixel_intensity

    red = Array(
        address_in = row_stream * width_stream + col_stream,
        data_in = red_intensity,
        address_out = pixel_clock,
        depth = width * height,
    )

    green = Array(
        address_in = row_stream * width_stream + col_stream,
        data_in = green_intensity,
        address_out = pixel_clock,
        depth = width * height,
    )

    blue = Array(
        address_in = row_stream * width_stream + col_stream,
        data_in = blue_intensity,
        address_out = pixel_clock,
        depth = width * height,
    )

    return red, green, blue
```

class `Counter` (*start, stop, step*)

A Stream which yields numbers from *start* to *stop* in *step* increments.

A *Counter* is a versatile, and commonly used construct in device design, they can be used to number samples, index memories and so on.

Example:

```
Counter(0, 10, 2) # --> 0 2 4 6 8 10 0 \..

Counter(10, 0, -2) # --> 10 8 7 6 4 2 0 10 \..
```

class `Decoupler` (*source*)

A *Decoupler* removes stream handshaking.

Usually, data is transfered though streams using blocking transfers. When a process writes to a stream, execution will be halted until the receiving process reads the data. While this behaviour greatly simplifies the design of parallel processes, sometimes Non-blocking transfers are needed. When a data item is written to a *Decoupler*, it is stored. When a *Decoupler* is read from, the value of the last stored value is yielded. Neither the sending or the receiving process ever blocks. This also means that the number of data items written into the *Decoupler* and the number read out do not have to be the same.

A *Decoupler* accepts only one argument, the source stream.

Example:

```
def time_stamp_data(data_stream):

    us_time = Output()
```

```
    time = Variable(0)
    Process(8,
        Loop(
            WaitUs,
            time.set(time + 1),
            us_time.write(time),
        ),
    )

    output_stream = Output()
    temp = Variable(0)
    Process(8,
        Loop(
            data_stream.read(temp),
            output_stream.write(temp),
            us_time.read(temp),
            output_stream.write(temp),
        ),
    )

    return output_stream
```

class `Resizer` (*source, bits*)

A *Resizer* changes the width, in bits, of the source stream.

The *Resizer* takes two arguments, the source stream, and the *width* in bits. The *Resizer* will truncate data if it is reducing the width, ans sign extend if it is increasing the width.

Example:

```
a = InPort(name="din", bits=8) a has a width of 8 bits
b = Inport + 1 #b has a width of 9 bits
c = Resizer(b, 8) #c is truncated to 8 bits
Chip(OutPort(name="dout"))
```

class `Lookup` (*source, *args*)

A *Lookup* is a stream yields values from a read-only look up table.

For each data item in the source stream, a *Lookup* will yield the addressed value in the lookup table. A *Lookup* is basically a Read Only Memory(ROM) with the source stream forming the address, and the *Lookup* itself forming the data output.

Example:

```
def binary_2_gray(input_stream):
    return Lookup(input_stream, 0, 1, 3, 2, 6, 7, 5, 4)
```

The first argument to a *Lookup* is the source stream, all additional arguments form the lookup table. If you want to use a Python sequence object such as a tuple or a list to form the lookup table use the following syntax:

```
my_list = [0, 1, 3, 2, 6, 7, 5, 4]
my_sequence = Sequence(Counter(0, 7, 1), *my_list)
```

class `Fifo` (*data_in, depth*)

A *Fifo* stores a buffer of data items.

A *Fifo* contains a fixed size buffer of objects obtained from the source stream. A *Fifo* yields the data items in the same order in which they were stored.

The first argument to a *Fifo*, is the source stream, the *depth* argument determines the size of the Fifo buffer.

Example:

```
def digital_oscilloscope(ADC_stream, trigger_level):
    temp = Variable(0)
    count = Variable(0)

    Process(16,
        Loop(
            ADC_stream.read(temp),
            If(temp > trigger_level,
                count.set(buffer_depth),
                While(count,
                    ADC_stream.read(temp),
                    buffer.write(temp),
                    count.set(count-1),
                ),
            ),
        ),
    )

    return SerialOut(Printer(Fifo(buffer, buffer_depth)))
```

class Repeater(*value*)

A stream which repeatedly yields the specified *value*.

The *Repeater* stream is one of the most fundamental streams available.

The width of the stream in bits is calculated automatically. The smallest number of bits that can represent *value* in twos-complement format will be used.

Examples:

```
Repeater(5) #--> 5 5 5 5 \..
#creates a 4 bit stream.

Repeater(10) #--> 10 10 10 10 \..
#creates a 5 bit stream.

Repeater(5)*2 #--> 10 10 10 10 \..
#This is shorthand for: Repeater(5)*Repeater(2)
```

class Sequence()

class Stimulus(*bits*)

A Stream that allows a Python iterable to be used as a stream.

A Stimulus stream allows a transparent method to pass data from the Python environment into the simulation environment. The sequence object is set at run time using the set_simulation_data() method. The sequence object can be any iterable Python sequence such as a list, tuple, or even a generator.

Example:

```
import PIL

picture = Stimulus()
s = Chip(Console(Printer(picture)))

im = PIL.open("test.bmp")
```

```
image_data = list(im.getdata())
picture.set_simulation_data(image_data)

picture.reset()
picture.execute(1000)
```

class InPort(*name, bits*)

A device input port stream.

An *InPort* allows a port pins of the target device to be used as a data stream. There is no handshaking on the input port. The port pins are sampled at the point when data is transfered by the stream. When implemented in VHDL, the *InPort* provides double registers on the port pins to synchronise data to the local clock domain.

Since it is not possible to determine the width of the stream in bits automatically, this must be specified using the *bits* argument.

The *name* parameter allows a string to be associated with the input port. In a VHDL implementation, *name* will be used as the port name in the top level entity.

Example:

```
dip_switches = Inport("dip_switches", 8)
s = Chip(SerialOut(Printer(dip_switches)))
```

class SerialIn(*name='RX', clock_rate=50000000, baud_rate=115200*)

A *SerialIn* yields data from a serial UART port.

SerialIn yields one data item from the serial input port for each character read from the source stream. The stream is always 8 bits wide.

A *SerialIn* accepts an optional *name* argument which is used as the name for the serial RX line in generated VHDL. The clock rate of the target device in MHz can be specified using the *clock_rate* argument. The baud rate of the serial input can be specified using the *baud_rate* argument.

Example:

```
#echo typed characters
my_chip = Chip(SerialOut(SerialIn())
```

class Output()

An *Output* is a stream that can be written to by a process.

Any stream can be read from by a process. Only an *Output* stream can be written to by a process. A process can be written to by using the *read* method. The read method accepts one argument, an expression to write.

Example:

```
def tee(input_stream):
    output_stream_1 = Output()
    output_stream_2 = Output()
    temp = Variable(0)
    Process(input_stream.get_bits,
        Loop(
            input_stream.read(temp),
            output_stream_1.write(temp),
            output_stream_2.write(temp),
        )
    )
    return input_stream_1, input_stream_2
```

class `Printer`(*source*)

A *Printer* turns data into decimal ASCII characters.

Each each data item is turned into the ASCII representation of its decimal value, terminated with a newline character. Each character then forms a data item in the *Printer* stream.

A *Printer* accepts a single argument, the source stream. A *Printer* stream is always 8 bits wide.

Example:

```
#print the numbers 0-10 to the console repeatedly
Chip(
    Console(
        Printer(
            Counter(0, 10, 1),
        ),
    ),
)
```

class `HexPrinter`(*source*)

A *HexPrinter* turns data into hexadecimal ASCII characters.

Each each data item is turned into the ASCII representation of its hexadecimal value, terminated with a newline character. Each character then forms a data item in the *HexPrinter* stream.

A *HexPrinter* accepts a single argument, the source stream. A *HexPrinter* stream is always 8 bits wide.

Example:

```
#print the numbers 0x0-0x10 to the console repeatedly
Chip(
    Console(
        Printer(
            Counter(0x0, 0x10, 1),
        ),
    ),
)
```

class `Scanner`()

3.4 Sinks

Sinks are a fundamental component of the *Chips* library.

A sink is used to terminate a stream. A sink may act as:

- An output of a *Chip* such as an *OutPort* or *SerialOut*.
- A consumer of data in its own right such as an *Asserter*.

3.4.1 Sinks Reference

class `Response`(*a*)

A *Response* sink allows data to be transfered into Python.

As a simulation is run, the *Response* sink accumulates data. After a simulation is run, you can retrieve a python iterable using the get_simulation_data method. Using a *Response* sink allows you to seamlessly integrate your

Chips simulation into a wider Python simulation. This works for simulations using an external simulator as well, in this case you also need to pass the code generation plugin to get_simulation_data.

A *Response* sink accepts a single stream argument as its source.

Example:

```
import PIL

def image_processor():
    #some image processing algorithm
    pass

response = Response(image_processor)
chip = Chip(response)

chip.reset()
chip.execute(10000)


image_data = list(response.get_simulation_data(plugin))
im = PIL.Image.new("L", (64, 64))
im.putdata(image_data)
im.show()
```

class **OutPort** (*a, name*)

An *OutPort* sink outputs a stream of data to I/O port pins.

No handshaking is performed on the output port, data will appear at the time when the source stream transfers data.

An output port take two arguments, the source stream *a* and a string *name*. Name is used as the port name in generated VHDL.

Example:

```
dip_switches = Inport("dip_switches", 8)
led_array = OutPort(dip_switched, "led_array")
s = Chip(led_array)
```

class **SerialOut** (*a, name='TX', clock_rate=50000000, baud_rate=115200*)

A *SerialOut* outputs data to a serial UART port.

SerialOut outputs one character to the serial output port for each item of data in the source stream. At present only 8 data bits are supported, so the source stream must be 8 bits wide. The source stream could be truncated to 8 bits using a *Resizer*, but it is usually more convenient to use a *Printer* as the source stream. The will allow a stream of any width to be represented as a decimal string.

A SerialOut accepts a source stream argument *a*. An optional *name* argument is used as the name for the serial TX line in generated VHDL. The clock rate of the target device in MHz can be specified using the *clock_rate* argument. The baud rate of the serial output can be specified using the *baud_rate* argument.

Example:

```
#convert string into a sequence of characters
hello_world = tuple((ord(i) for i in "hello world\n"))

my_chip = Chip(
    SerialOut(
        Sequence(*hello_world),
```

```
        )
    )
```

class **Asserter**(*a*)

An *Asserter* causes an exception if any data in the source stream is zero.

An *Asserter* is particularly useful in automated tests, as it causes a simulation to fail is a condition is not met. In generated VHDL code, an asserter is represented by a VHDL assert statement. In practice this means that an *Asserter* will function correctly in a VHDL simulation, but will have no effect when synthesized.

The *Asserter* sink accepts a source stream argument, *a*.

Example:

```
a = Sequence(1, 2, 3, 4)
Chip(Asserter((a+1) == Sequence(2, 3, 4, 5)))
```

Look at the Chips test suite for more examples of the Asserter being used for automated testing.

class **Console**(*a*)

A *Console* outputs data to the simulation console.

Console stores characters for output to the console in a buffer. When an end of line character is seen, the buffer is written to the console. A *Console* interprets a stream of numbers as ASCII characters. The source stream must be 8 bits wide. The source stream could be truncated to 8 bits using a *Resizer*, but it is usually more convenient to use a *Printer* as the source stream. The will allow a stream of any width to be represented as a decimal string.

A *Console* accepts a source stream argument *a*.

Example:

```
#convert string into a sequence of characters
hello_world = tuple((ord(i) for i in "hello world\n"))

my_chip = Chip(
    Console(
        Sequence(*hello_world),
    )
)
```

3.5 Instructions

The instructions provided here form the basis of the software that can be run inside *Processes*.

3.5.1 Instructions Reference

class **Block**(*instructions*)

The *Block* statement allows instructions to be nested into a single statement. Using a *Block* allows a group of instructions to be stored as a single object.

Example:

```
Initialise = Block(a.set(0), b.set(0), c.set(0))
Process(8,
    initialise,
```

```
        a.set(a+1), b.set(b+1), c.set(c+1),
        initialise,
    )
```

class **Break**()

The *Break* statement causes the flow of control to immediately exit the loop.

Example:

```
#equivalent to a While loop
Loop(
    If(condition == 0,
        Break(),
    ),
    #do stuff here
),
```

Example:

```
#equivalent to a DoWhile loop
Loop(
    #do stuff here
    If(condition == 0,
        Break(),
    ),
),
```

class **Continue**()

The *Continue* statement causes the flow of control to immediately jump to the next iteration of the containing loop.

Example:

```
Process(12,
    Loop(
        in_stream.read(a),
        If(a&1,
            Continue(),
        ),
        out_stream.write(a),
    ),
)
```

class **If**(*condition, *instructions*)

The *If* statement conditionally executes instructions.

The condition of the *If* branch is evaluated, followed by the condition of each of the optional *ElsIf* branches. If one of the conditions evaluates to non-zero then the corresponding instructions will be executed. If the *If* condition, and all of the *ElsIf* conditions evaluate to zero, then the instructions in the optional *Else* branch will be evaluated.

Example:

```
If(condition,
    #do something
).ElsIf(condition,
    #do something else
).Else(
```

```
    #if all else fails do this
)
```

class **Loop**(**instructions*)

The *Loop* statement executes instructions repeatedly.

A *Loop* can be exited using the *Break* instruction. A *Continue* instruction causes the remainder of instructions in the loop to be skipped. Execution then repeats from the beginning of the *Loop*.

Example:

```
#filter filter values over 50 out of a stream
Loop(
    in_stream.read(a),
    If(a > 50, Continue()),
    out_stream.write(a),
),
```

Example:

```
#initialise an array
Loop(
    If(index == 100,
        Break(),
    ),
    myarray.write(index, 0),
),
```

class **Value**(*expression*)

The *Value* statement gives a value to the surrounding *Evaluate* construct.

An *Evaluate* expression allows a block of statements to be used as an expression. When a *Value* is encountered, the supplied expression becomes the value of the whole evaluate statement.

Example:

```
#provide a And expression similar to Pythons and expression
def LogicalAnd(a, b):
    return Evaluate(
        If(a,
            Value(b),
        ).Else(
            0,
        ),
    )
```

class **WaitUs**()

WaitUs causes execution to halt until the next tick of the microsecond timer.

In practice, this means that the process is stalled for less than 1 microsecond. This behaviour is useful when implementing a real-time counter function because the execution time of statements does not affect the time between *WaitUs* statements (Providing the statements do not take more than 1 microsecond to execute of course!).

Example:

```
seconds = Variable(0)
count = Variable(0)
Process(12,
```

```
        seconds.set(0),
        Loop(
            count.set(1000),
            While(count,
                WaitUs(),
                count.set(count-1),
            ),
            seconds.set(seconds + 1),
            out_stream.write(seconds),
        ),
    )
```

class **While**()

class **Scan**(*stream, variable*)

class **Print**(*stream, exp, minimum_number_of_digits=None*)

class **Evaluate**(**instructions*)

CHAPTER

FOUR

AUTOMATIC CODE GENERATION

4.1 VHDL Code Generation

VHDL Code Generation for streams library

4.2 C++ Code Generation

C++ code generator for streams library

4.3 Visualisation Code Generation

Visualisation for streams library

CHAPTER

FIVE

IP LIBRARY

CHAPTER

SIX

EXTENDING THE CHIPS LIBRARY

CHAPTER

SEVEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

C

chips.cpp_plugin, 25
chips.instruction, 20
chips.ip, 27
chips.process, 9
chips.sinks, 18
chips.streams, 12
chips.VHDL_plugin, 25
chips.visual_plugin, 25

INDEX

A

Array (class in chips), 13
Asserter (class in chips), 20

B

Block (class in chips), 20
Break (class in chips), 21

C

Chip (class in chips), 9
chips.cpp_plugin (module), 25
chips.instruction (module), 20
chips.ip (module), 27
chips.process (module), 9
chips.sinks (module), 18
chips.streams (module), 12
chips.VHDL_plugin (module), 25
chips.visual_plugin (module), 25
Console (class in chips), 20
Continue (class in chips), 21
Counter (class in chips), 14

D

Decoupler (class in chips), 14

E

Evaluate (class in chips), 23

F

Fifo (class in chips), 15

H

HexPrinter (class in chips), 18

I

If (class in chips), 21
InPort (class in chips), 17

L

Lookup (class in chips), 15
Loop (class in chips), 22

O

OutPort (class in chips), 19
Output (class in chips), 17

P

Print (class in chips), 23
Printer (class in chips), 17
Process (class in chips), 12

R

Repeater (class in chips), 16
Resizer (class in chips), 15
Response (class in chips), 18

S

Scan (class in chips), 23
Scanner (class in chips), 18
Sequence (class in chips), 16
SerialIn (class in chips), 17
SerialOut (class in chips), 19
Stimulus (class in chips), 16

V

Value (class in chips), 22
Variable (class in chips), 12
VariableArray (class in chips), 12

W

WaitUs (class in chips), 22
While (class in chips), 23