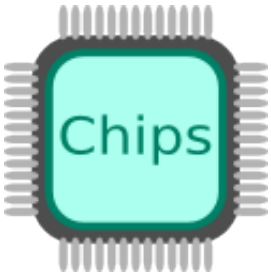

Chips Documentation

Release 0.1

Jonathan P Dawson

April 23, 2011

CONTENTS

CHAPTER

ONE

WHAT IS CHIPS?

Chips is a Python library that provides a language for designing hardware devices.

CHAPTER

TWO

FEATURES

Some of the key features include:

- High level modeling language makes device design simpler and more powerful.
- An open source hardware design environment.
- Provides fast native simulations that integrate with Python.
- Exploit Python extension modules such as Scipy, Numpy, Matplotlib and PIL provide a rich verification environment.
- Automatic generation of synthesizable VHDL.
- Plugin mechanism also allows C++ and graphviz outputs to be generated.
- Existing VHDL IP can be imported.
- Seamless co-simulation of C++ and VHDL outputs.

CHAPTER

THREE

A QUICK TASTER

```
>>> #4 bit linear feedback shift register

>>> from chips import *

>>> new_bit = Variable(0)
>>> shift_register = Variable(1) #initialise to anything but 0
>>> output_stream = Output()

>>> Process(5,
...     Loop(
...
...         #tap off bit 2 and 3
...         new_bit.set((shift_register >> 0) ^ (shift_register >> 1) ^ new_bit),
...
...         #implement shift register
...         shift_register.set(((new_bit & 1) << 3) | (shift_register >> 1)),
...
...         #4 bit mask
...         shift_register.set(shift_register & 0xf),
...
...         #write to stream
...         output_stream.write(shift_register)
...     )
... )
Process(...

>>> device = Chip(Console(Printer(output_stream)))
>>> device.reset()
>>> device.execute(1000)
8
12
14
7
3
1
...
```

CHAPTER

FOUR

DOWNLOAD

You can download a source distribution or a windows self installer from the GitHub homepage.

CHAPTER

FIVE

DOCUMENTATION

5.1 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.

5.1.1 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 UARTs.
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.

5.1.2 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.

5.2 Tutorial

5.2.1 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.

5.2.2 Install Chips

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.

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
```

5.2.3 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 chips import *

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

>>> #run a simulation
>>> my_chip.reset()
>>> my_chip.execute(100)
0
1
2
3
4
5
6
7
8
9
10
0
...
```

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.

5.2.4 Hello World

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

```
>>> from chips import *

>>> my_chip = Chip(
...     Console(
...         Sequence(*map(ord, "hello world\n")),
...     )
```

```
... )

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

In this example we have made only a few changes:

- `map(ord, "hello world\n")` creates a list 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.

5.2.5 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

>>> my_chip = Chip(
...     Console(
...         Sequence(*map(ord, "hello_world\n")),
...     )
... )

>>> #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 that a VHDL file called hello_world.vhd has been generated.

Take a look through this file. you may find that it is difficult to understand what is going on. the file isn't meant to be read by humans, *Chips* treats VHDL as a compatibility layer. *VHDL* is pretty much universally supported by synthesis tools. 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

>>> my_chip = Chip(
```

```
...     SerialOut(
...         Sequence(*map(ord, "hello_world\n")),
...     )
... )

>>> #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.

5.2.6 More Streams and Sinks

So far we have seen three types of streams, *Counter*, *Sequence* and *Printer*. *Chips* provides a range of streams. The full documentation for streams is in the reference manual but a quick summary is included here:

Stream	Description
Array()	An indexable memory with an independent read and write port.
Counter()	A versatile counter with min, max and step parameters
Decoupler()	A Decoupler removes stream handshaking.
Resizer()	A Resizer changes the width, in bits, of the source stream.
Lookup()	An indexable Read Only Memory with a single read port.
Fifo()	Stores data items in a buffer.
Repeater()	Yields the same data item repeatedly.
InPort()	Yields the value of input port pins.
SerialIn()	Yields values from a serial UART.
Output()	A stream that is fed by a *Process* (more on this later)
Printer()	A decimal ASCII representation of the source stream.
HexPrinter()	A hexadecimal ASCII representation of the source stream.
Scanner()	yields the value of the decimal ASCII source stream.

You can also combine streams using the operators : `abs, ~, +, -, *, //, %, <<, >>, &, |, ^, ==, !=, <, <=, >, >=` on the whole they have the same (or very similar) meaning as they do in *Python* except that they operate on streams of data. It is also possible to form an expression from regular integers and streams, *Chips* will automatically transform an integer into an appropriate *Repeater* stream. For example `Counter(0, 9, 1)*2` is a shorthand for `Counter(0, 9, 1)*Repeater(2)`.

The following table summarises the available sinks:

Sink	Description
Response()	A Response sink allows data to be transfered into Python.
OutPort()	An OutPort sink outputs a stream of data to I/O port pins.
SerialOut()	A SerialOut outputs data to a serial UART port.
Asserter()	An Asserter causes an exception if any data in the source stream is zero.
Console()	A Console outputs data to the simulation console.

5.2.7 Types and Bit Width

For convenience, the central numerical type in *Chips* is a signed integer with a fixed number of bits. This is in contrast to *Python*, where integers have a potentially infinite width. *Chips* tries to simplify some of the design issues involved

with limited width numbers by doing a lot of the work for you, but it is not always possible to completely hide these details, so you need to how things are handled behind the scenes.

Chips will automatically determine the width of a stream whenever possible. In a *Repeater*, *Counter* or *Lookup*, *Chips* will chose use the number of bits needed to hold the greatest possible value. This is not possible for *InPort*, or *Array* streams because the maximum possible value is not known at compile time. When it is not possible to determine the maximum value, the width must be specified using the bits parameter.

When streams are combined using operators, the width of the resulting stream will usually be chosen to handle the maximum possible value in the resulting stream, though there are some exceptions. Adding two 8 bit streams will result in a 9 bit stream, multiplying two 8 bit streams will result in a 16 bit stream. The precise handling of bit widths is documented more fully in the reference manual.

You can manually change the width of a stream using the *Resize* stream. Making a streams smaller in width will result in large values being truncated. Making a stream larger in width will result in sign extension.

5.2.8 Introducing Processes

We have seen how the *Chips* library provides quite a few ready made streams out of the box. Sometimes these streams won't suite our needs, sometimes we need to define new operations on streams. Suppose we wanted to double the value of every data item within in an existing stream, a Counter say. Thats easy, just use the multiply operator `Counter(0, 9, 1)*2`. Now suppose that we wanted to square each data item instead. Not so simple, there is no squaring operator, or even a power operator for that matter. Thats where the *Process* comes in:

```
>>> from chips import *

>>> counter = Counter(0, 9, 1)
>>> temp = Variable(0)#create a temporary variable and initialise it to 0.
>>> counter_squared_stream = Output()

>>> p=Process(counter.get_bits()*2,
...     Loop(
...         counter.read(temp),
...         counter_squared_stream.write(temp*temp),
...     )
... )

>>> c = Chip(Console(Printer(counter_squared_stream)))
>>> c.reset()
>>> c.execute(1000)
0
1
4
9
16
25
36
...
```

This example demonstrates some of the key features of the *Process*:

- Put it simply, a *Process* is small computer program which can contain loops and if statements like any other language.
- A *Chip* can contain any number of Process objects, they will all run in parallel.

- Within a *Process*, you can use *Variables* to store data. Each variable can only be used within one *Process*, to communicate with another *Process* you need to use streams.
- A *Process* can read from any type of stream, in this example the process is reading from a *Counter* stream. Only *Output* streams can be written to.
- Streams can only be used for point to point communications. A stream cannot be read by more than one *Process*. Likewise, an *Output* stream can only be written to by one *Process*.

5.2.9 Process Instructions

Instruction	Description
Variable()	A Variable is used within a Process to store data.
Value()	The Value statement gives a value to the surrounding Evaluate construct.
Evaluate()	An Evaluate expression allows a block of statements to be used as an expression.
Loop()	The Loop statement executes instructions repeatedly.
If()	The If statement conditionally executes instructions.
Break()	The Break statement causes the flow of control to immediately exit the loop.
WaitUs()	WaitUs causes execution to halt until the next tick of the microsecond timer.
Continue()	The Continue statement causes the flow of control to immediately jump to the next iteration of the containing loop.
Block()	The Block statement allows instructions to be nested into a single statement.
Output.write()	This method returns a write instruction that writes a single data item to the Output stream.
<stream>.read()	This method returns a read instruction that reads a single data item from a stream.
Variable.set()	This method returns a set instruction that assigns the value of an expression to a variable.

5.2.10 Bit Width Within a Process

We have already seen how streams are usually sized automatically to handle the largest possible data value. Inside a *Process* however things are handled differently. A *Process* has a fixed bit width. The width is the first argument given to a *Process*. Inside a *Process*, the value of any expression will be resized the width of the *Process*. When a *Process* reads from a stream, the value will be truncated or sign extended to the width of the *Process*. It is important to make sure that the width of a *Process* is sufficiently large.

5.2.11 Hierarchical Design

You may be expecting *Chips* to provide some mechanism for hierarchical design. You might expect that *Chips* would provide a means too group items together to form re-usable components or modules. A really good design tool would allow you to parameterise components and modules using generics or templates. *Chips* does not provide any of these things. It doesn't have to.

The *Python* language itself already provides all these things and more. If you want to make a reusable component you can simply write a *Python* function:

```
>>> from chips import *

>>> def double(input_stream):
...     """If you use Python functions to build components you can take
...     advantage of docstrings to document your design."""
...
...     return input_stream * 2
```

```
>>> c = Chip(
...     Console(
...         Printer(
...             double(
...                 Sequence(1, 2, 3)
...             )
...         )
...     )
... )

>>> c.reset()
>>> c.execute(10)
2
4
6
2
...
```

5.2.12 Streams from Multiple Sources

Streams can only be have one source of data and one sink, but it is possible to combine data from more than one source into a single stream using a *Process*. The simplest approach is to read a value from each source, and write it to the destination thus:

```
>>> from chips import *

>>> def simple_arbiter(source_0, source_1):
...     """Combine data from two streams into a single stream"""
...     temp = Variable(0)
...     dest = Output()
...     Process(max([source_0.get_bits(), source_1.get_bits()]),
...         Loop(
...             source_0.read(temp),
...             dest.write(temp),
...             source_1.read(temp),
...             dest.write(temp),
...         ),
...     )
...     return dest

>>> c = Chip(
...    Console(
...        Printer(
...            simple_arbiter(
...                Repeater(1), Repeater(2)
...            )
...        )
...    )
... )

>>> c.reset()
>>> c.execute(100)
1
2
1
2
```

```
1
2
...
```

This type of arbiter will always take an equal number of items from source_0, and source_1. This may be fine in some applications, but if data were not available on source_0, data from source_1 would also be blocked. One solution is to use the *available* method of a stream to test whether data is available before committing to a blocking read:

```
>>> from chips import *

>>> def non_blocking_arbiter(source_0, source_1):
...     """Combine data from two streams into a single stream"""
...     temp = Variable(0)
...     dest = Output()
...     Process(max([source_0.get_bits(), source_1.get_bits()]),
...         Loop(
...             If(source_0.available(),
...                 source_0.read(temp),
...                 dest.write(temp),
...             ),
...             If(source_1.available(),
...                 source_1.read(temp),
...                 dest.write(temp),
...             ),
...         ),
...     )
...     return dest
...

>>> blocked = Output()
>>> p=Process(8,
...     #outputs one value then blocks
...     blocked.write(1),
... )

>>> c = Chip(
...     Console(
...         Printer(
...             non_blocking_arbiter(
...                 blocked, Repeater(2)
...             )
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
2
1
2
2
2
...
```

5.2.13 Streams with Multiple Sinks

Sometimes a stream will need to be used in more than one place. A simple solution is to make a splitter or tee using a *Process*:

```
>>> from chips import *

>>> def tee(source):
...     """split data into two streams"""
...     temp = Variable(0)
...     dest_0 = Output()
...     dest_1 = Output()
...     Process(source.get_bits(),
...         Loop(
...             source.read(temp),
...             dest_0.write(temp),
...             dest_1.write(temp),
...         ),
...     )
...     return dest_0, dest_1

>>> dest_0, dest_1 = tee(Counter(0, 9, 1))

>>> c = Chip(
...     Console(
...         Printer(dest_0),
...     ),
...     Console(
...         Printer(dest_1),
...     )
... )

>>> c.reset()
>>> c.execute(100)
0
0
1
1
2
2
3
3
...
```

5.2.14 A Worked Example

TODO

5.2.15 Further Examples

The source distribution contains a number of more involved examples so that you can see for yourself how more complex hardware designs can be formed from these simple components.

5.3 Chips Language Reference Manual

5.3.1 Chip

class Chip (**args*)

A Chip is 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:

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

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

>>> #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,
... )

>>> plugin = Plugin()
>>> s.write_code(plugin)
```

5.3.2 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.

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:

```
>>> from chips import *

>>> #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),
... )
Process(...

>>> #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
...     #..
... )
Process(...
```

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
...     #..
... )
Process(...

>>> #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),
... )
Process(...
```

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 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.

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 unary operators:

```
~
```

and the folowing binary operators:

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

The function *Not* evaluates to the logical negation of each data item equivalent to `==0`. The function *abs* evaluates to the magnitude of each data item.

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.

Note: The divide `//` operator in *Chips* works differently then the divide operator in Python. While a floor division in Python rounds to -infinite, in *Chips* division rounds to `0`. Thus `-3//2` rounds to `-2` in Python, it rounds to `-1` in *Chips*. This should be more familiar to users of C, C++ and VHDL. The same also applies to the modulo `%` operator.

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.

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
~	bitwise NOT
Not, abs	logical NOT, absolute

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

class **Variable** (*initial*)

A *Variable* is used within a *Process* to store data. A *Variable* can be used in only one *Process*. If you need to communicate with another *Process* you must use a stream.

A *Variable* accepts a single argument, the initial value. A *Variable* will be reset to the initial value when a simulation, or actual device is reset.

A *Variable* can be assigned an expression using the *set* method.

class **VariableArray** (*size*)

A *VariableArray* is an array of variables that can be accessed from within a single *Process*.

When a *VariableArray* is created, it accepts a single argument, the *size*.

A *VariableArray* can be written to using the *write* method, the *write* method accepts two arguments, an expression indicating the *address* to write to, and an expression indicating the *data* to write.

A *VariableArray* can be read to using the *read* method, the *read* method accepts a single argument, an expression indicating the *address* to read from. The *read method returns an expression that evaluates to the value contained at *address*.

Example:

```
>>> from chips import *

>>> def reverse(stream, number_of_items):
...       """Read number_of_items from stream, and reverse them."""
...       temp = Variable(0)
...       index = Variable(0)
...       reversed_stream = Output()
...       data_store = VariableArray(number_of_items)
...       Process(8,
...           index.set(0),
...           While(index < number_of_items,
...               stream.read(temp),
...               data_store.write(index, temp),
...               index.set(index+1),
...           ),
...           index.set(number_of_items - 1),
...           While(index >= 0,
...               reversed_stream.write(data_store.read(index)),
```

```
...                 index.set(index-1),
...             ),
...         )
...
...         return reversed_stream


>>> c = Chip(
...     Console(
...         Printer(
...             reverse(Sequence(0, 1, 2, 3), 4)
...         ),
...     ),
... )

>>> c.reset()

>>> c.execute(1000)
3
2
1
0
```

5.3.3 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*.

Stream Expressions

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

```
~
```

and the folowing binary operators:

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

The function *Not* yields the logical negation of each data item equivalent to `==0`. The function *abs* yields the magnitude of each data item.

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.

Note: The divide `//` operator in *Chips* works differently then the divide operator in Python. While a floor division in Python rounds to -infinite, in *Chips* division rounds to `0`. Thus `-3//2` rounds to `-2` in Python, it rounds to `-1` in *Chips*. This should be more familiar to users of C, C++ and VHDL. The same also applies to the modulo `%` operator.

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)
abs	Logical Not	argument
Not	Logical Not	1
~	Bitwise not	right
+	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

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
~	bitwise NOT
Not, abs	logical NOT, absolute

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,
...                         intensity):
...
...     pixel_clock = Counter(0, width*height, 1)
...
...     pixstream = Array(
...         address_in = (row_stream * width) + col_stream,
...         data_in = intensity,
...         address_out = pixel_clock,
...         depth = width * height,
...     )
...
...     return pixstream

>>> pixstream = video_raster_stream(
...     64,
...     64,
...     Repeater(32),
...     Counter(0, 63, 1),
...     Repeater(255),
... )
```

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:

```
>>> from chips import *

>>> c=Chip(
...     Console(
...         Printer(
...             Counter(0, 10, 2) #creates a 4 bit stream
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
0
2
4
```

```
6
8
10
0
...

>>> c=Chip(
...     Console(
...         Printer(
...             Counter(10, 0, -2) #creates a 4 bit stream
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
10
8
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:

```
>>> from chips import *

>>> 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

>>> time_stamped_stream = time_stamp_data(SerialIn())
```

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:

```
>>> from chips import *

>>> def scope(ADC_stream, trigger_level, buffer_depth):
...     temp = Variable(0)
...     count = Variable(0)
...     buffer = Output()
...
...     Process(16,
...         Loop(
...             ADC_stream.read(temp),
...             If(temp > trigger_level,
...                 buffer.write(temp),
...                 count.set(buffer_depth - 1),
...                 While(count,
...                     ADC_stream.read(temp),
...                     buffer.write(temp),
...                     count.set(count-1),
...                 ),
...             ),
...         ),
...     )
...
...     return Printer(Fifo(buffer, buffer_depth))
...

>>> test_signal = Sequence(0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5, 5)
>>> c = Chip(Console(scope(test_signal, 0, 5)))
>>> c.reset()
>>> c.execute(100)
1
2
3
4
5
```

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:

```
>>> from chips import *

>>> #print the numbers 0x0-0x10 to the console repeatedly
>>> c=Chip(
...     Console(
...         HexPrinter(
...             Counter(0x0, 0x10, 1),
...         ),
...     ),
... )

>>> c.reset()
>>> c.execute(1000)
0
1
2
3
4
5
6
7
8
9
a
b
...
```

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:

```
>>> from chips import *
>>> dip_switches = InPort("dip_switches", 8)
>>> s = Chip(SerialOut(Printer(dip_switches)))
```

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:

```
>>> from chips import *

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

>>> c = Chip(
...     Console(
...         Printer(binary_2_gray(Counter(0, 7, 1)))
...     )
... )

>>> c.reset()
>>> c.execute(100)
0
1
3
2
6
7
5
4
0
...
```

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 = Lookup(Counter(0, 7, 1), *my_list)
```

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:

```
>>> from chips import *

>>> 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 output_stream_1, output_stream_2

>>> os_1, os_2 = tee(Counter(1, 3, 1))

>>> c = Chip(
...     Console(
```

```
...         Printer(os_1),
...     ),
...     Console(
...         Printer(os_2),
...     ),
... )

>>> c.reset()
>>> c.execute(100)
1
1
2
2
3
3
...
```

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:

```
>>> from chips import *

>>> #print the numbers 0-10 to the console repeatedly
>>> c=Chip(
...    Console(
...         Printer(
...             Counter(0, 10, 1),
...         ),
...     ),
... )

>>> c.reset()
>>> c.execute(100)
0
1
2
3
4
...
```

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:

```
>>> from chips import *
```

```
>>> c=Chip(
...     Console(
...         Printer(
...             Repeater(5) #creates a 4 bit stream
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
5
5
5
...

>>> c=Chip(
...     Console(
...         Printer(
...             Repeater(10) #creates a 5 bit stream
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
10
10
10
...

>>> c=Chip(
...     Console(
...         Printer(
...             #This is shorthand for: Repeater(5)*Repeater(2)
...             Repeater(5)*2
...         )
...     )
... )

>>> c.reset()
>>> c.execute(100)
10
10
10
...
```

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:

```
>>> from chips import *
>>> a = InPort(name="din", bits=8) #a has a width of 8 bits
>>> a.get_bits()
8
```

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

Scanner (*stream, bits*)

A *Scanner* converts a stream of decimal ASCII into their integer value.

Numeric characters separated by non-numeric characters are interpreted as numbers. As it is not possible to determine the maximum value of a *Scanner* stream at compile time, the width of the stream must be specified using the bits parameter.

The *Scanner* stream accepts two inputs, the source stream and the number of bits.

Example:

```
>>> from chips import *

>>> #multiply by two and echo
>>> c = Chip(
...     Console(
...         Printer(
...             Scanner(Sequence(*map(ord, "10 20 30 ")), 8)*2,
...         ),
...     ),
... )

>>> c.reset()
>>> c.execute(1000) # doctest: +ELLIPSIS
20
40
60
20
...
```

Sequence (**args*)

A *Sequence* stream yields each of its arguments in turn repeatedly.

A *Sequence* accepts any number of arguments. The bit width of a sequence is determined automatically, using the number of bits necessary to represent the argument with the largest magnitude. A *Sequence* allows Python sequences to be used within a Chips simulation using the `Sequence(*python_sequence)` syntax.

Example:

```
>>> from chips import *

>>> c = Chip(
...     Console(
...         Sequence(*map(ord, "hello world\n")),
...     )
... )

>>> c.reset()
>>> c.execute(50)
hello world
```

```
hello world
hello world
...
```

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:

```
>>> from chips import *
>>> #echo typed characters
>>> c = Chip(SerialOut(SerialIn()))
```

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:

```
>>> from chips import *

>>> stimulus = Stimulus(8)
>>> c = Chip(Console(Printer(stimulus)))

>>> def count():
...     i=0
...     while True:
...         yield i
...         i+=1
...

>>> stimulus.set_simulation_data(count())
>>> c.reset()
>>> c.execute(100)
0
1
2
...
```

5.3.4 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*.

Sinks Reference

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:

```
>>> from chips import *
>>> a = Sequence(1, 2, 3, 4)
>>> c = 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:

```
>>> 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),
...     )
... )
```

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:

```
>>> from chips import *
>>> dip_switches = InPort("dip_switches", 8)
>>> led_array = OutPort(dip_switches, "led_array")
>>> s = Chip(led_array)
```

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:

```
>>> from streams import *
>>> import PIL.Image #You need the Python Imaging Library for this

>>> def image_processor():
...     #black -> white
...     return Counter(0, 63, 1)*4

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

>>> chip.reset()
>>> chip.execute(100000)

>>> image_data = list(response.get_simulation_data())
>>> image_data = image_data[:(64*64)-1]
>>> im = PIL.Image.new("L", (64, 64))
>>> im.putdata(image_data)
>>> im.show()
```

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:

```
>>> from chips import *

>>> #convert string into a sequence of characters
>>> hello_world = map(ord, "hello world\n")

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

5.3.5 Instructions

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

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. A block accepts a single argument, *instructions*, a Python Sequence of instructions

Example:

```
>>> from chips import *

>>> a = Variable(0)
>>> b = Variable(1)
>>> c = Variable(2)

>>> 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),
... )
Process(...
```

class Break ()

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

Example:

```
#equivalent to a While loop
Loop(
    If(Not(condition),
        Break(),
    ),
    #do stuff here
),
```

Example:

```
#equivalent to a DoWhile loop
Loop(
    #do stuff here
    If(Not(condition),
        Break(),
    ),
),
```

class Continue ()

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

Example:

```
>>> from chips import *

>>> in_stream = Counter(0, 100, 1)
>>> out_stream = Output()
>>> a = Variable(0)
>>> #allow only even numbers
>>> Process(12,
```

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

>>> c = Chip(Console(Printer(out_stream)))
>>> c.reset()
>>> c.execute(100)
0
2
4
6
8
...
```

DoUntil(*condition, *instructions*)
A loop in which one iteration will be executed each time the condition is false. The condition is tested after each loop iteration.

Equivalent to:

```
Loop(
     instructions,
     If(condition, Break()),
 )
```

DoWhile(*condition, *instructions*)
A loop in which one iteration will be executed each time the condition is true. The condition is tested after each loop iteration.

Equivalent to:

```
Loop(
     instructions,
     If(Not(condition), Break()),
 )
```

class Evaluate(**instructions*)

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 *Elif* 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 *Elif* conditions evaluate to zero, then the instructions in the optional *Else* branch will be evaluated.

Example:

```
If(condition,
    #do something
).Elif(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:

```
>>> from chips import *

>>> #filter values over 50 out of a stream
>>> in_stream = Sequence(10, 20, 30, 40, 50, 60, 70, 80, 90)
>>> out_stream = Output()
>>> a = Variable(0)
>>> Process(8,
...     Loop(
...         in_stream.read(a),
...         If(a > 50, Continue()),
...         out_stream.write(a),
...     )
... )
Process(...

>>> c = Chip(
...     Console(
...         Printer(out_stream)
...     )
... )

>>> c.reset()
>>> c.execute(100)
10
20
30
40
50
10
...
```

Example:

```
>>> from chips import *

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

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

The *Print* instruction write an integer to a stream in decimal ASCII format.

Print will not add any white space or line ends (in contrast to the *Printer*) The *Print* instruction accepts two arguments, the destination *stream*, which must be an *Output* stream, and a numeric expression, *exp*. An optional third argument specifies the minimum number of digits to print (leading 0 characters are added).

Example:

```
>>> #multiply by 2 and echo
>>> temp = Variable(0)
>>> inp = Sequence(*map(ord, "1 2 3 "))
>>> out_stream = Output()
>>> p=Process(8,
...     Loop(
...         Scan(inp, temp),
...         out_stream.write(temp*2),
...     )
... )

>>> c = Chip(Console(Printer(out_stream)))
>>> c.reset()
>>> c.execute(1000)
2
4
6
2
...
```

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

The *Scan* instruction reads an integer value from a stream of decimal ASCII characters.

Numeric characters separated by non-numeric characters are interpreted as numbers. If *Scan* encounters a number that is too large to represent in a process, the result is undefined.

The *Scan* accepts two arguments, the source stream and a destination variable.

Example:

```
>>> from chips import *

>>> #multiply by 2 and echo
>>> temp = Variable(0)
>>> inp = Sequence(*map(ord, "1 2 3 "))
>>> out = Output()
>>> p=Process(8,
...     Loop(
...         Scan(inp, temp),
...         out.write(temp*2),
...     )
... )

>>> c = Chip(Console(Printer(out)))
>>> c.reset()
>>> c.execute(1000) # doctest: +ELLIPSIS
2
4
6
...
```

Until (*condition, *instructions*)

A loop in which one iteration will be executed each time the condition is false. The condition is tested before each loop iteration.

Equivalent to:

```
Loop(
    If(condition, Break()),
   instructions,
)
```

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:

```
>>> from chips import *

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

>>> check = Output()
>>> Process(8,
...     If(LogicalAnd(1, 4),
...         check.write(-1),#true
...     ).Else(
...         check.write(0),#false
...     )
... )
Process(...

>>> c = Chip(Asserter(check))
>>> c.reset()
>>> c.execute(100)
```

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:

```
>>> from chips import *

>>> seconds = Variable(0)
>>> count = Variable(0)
>>> out_stream = Output()
```

```
>>> Process(12,
...     seconds.set(0),
...     Loop(
...         count.set(1000),
...         While(count,
...             WaitUs(),
...             count.set(count-1),
...         ),
...         seconds.set(seconds + 1),
...         out_stream.write(seconds),
...     ),
... )
Process(...
```

While (*condition, *instructions*)

A loop in which one iteration will be executed each time the condition is true. The condition is tested before each loop iteration.

Equivalent to:

```
Loop(
    If(Not(condition), Break()),
    instructions,
)
```

5.4 Automatic Code Generation

5.4.1 VHDL Code Generation

VHDL Code Generation for streams library

5.4.2 C++ Code Generation

C++ code generator for streams library

5.4.3 Visualisation Code Generation

Visualisation for streams library

5.5 IP library

5.6 Extending the Chips Library

CHAPTER

SIX

NEWS

- 2011-04-09 Chips Library Published on GitHub.

CHAPTER

SEVEN

LINKS

- SciPy Scientific Tools for Python.
- matplotlib 2D plotting library for Python.
- Python Imaging Library (PIL) Python Imaging Library adds image processing capabilities to Python.
- MyHDL A Hardware description language based on Python.

CHAPTER

EIGHT

INDICES AND TABLES

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

MODULE INDEX

C

INDEX