Chips Documentation

Release 0.1

Jonathan P Dawson

March 19, 2011

CONTENTS

Contents:

CHAPTER

ONE

CHIPS LANGUAGE REFERENCE MANUAL

1.1 Chip

A Stream based concurrent programming library for embedded systems

class System (**args*)

A Chip device containing streams, sinks and processes.

Typically a System is used to describe a single device. You need to provide the System object with a list of all the sinks (devie outputs). You don't need to include any process, variables or streams. By analysing the sinks, the system 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 *System* that *leds* and *serial_out* are part of
#the device. The *System* can work out for itself that *switches* and
#*serial_in* are part of the device.

s = System(
    leds,
    serial_out,
)

s.write_code(plugin)
```

1.2 Processes

class Process (*bits, *instructions*)

class Variable (*initial*)

1.3 Streams

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

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*)

class **Resizer** (*source, bits*)

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

class **Fifo** (*data_in, 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 shothand for: Repeater(5)*Repeater(2)
```

class **Sequence** ()

class **Stimulus** (*bits*)

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

A Simulus stream allows a transparent method to pass data from the Python envrinment 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 = System(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 strean 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 = System(SerialOut(Printer(dip_switches)))
```

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

A *SerialIn* yields 8-bit data from a serial uart input.

class Output ()

class Printer (*source*)

class HexPrinter (*source*)

class Scanner ()

1.4 Sinks

class Response (*a*)

A Response block allows data to be read from a stream in the python design environment. A similar interface can be used in native python simulations and also co-simulations using external tools.

class OutPort (*a, name*)

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

class Asserter (*a*)

class Console (*a*)

1.5 Instructions

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:

```
intialise = 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:

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

Example:

```
#equivilent 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 contataining loop.

Example:

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

class If(*condition, *instructions*)

The *If* statement conditionaly 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 intructions in the loop to be skipped. Execution then repeats from the begining 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 the process is stalled for less than 1 microsecond. This behaviour is usefull 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

TWO

AUTOMATIC CODE GENERATION

2.1 VHDL Code Generation

2.2 C++ Code Generation

C++ code generator for streams library

2.3 Visualisation Code Generation

CHAPTER

THREE

INDICES AND TABLES

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

MODULE INDEX

INDEX