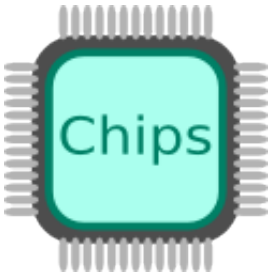


Chips Documentation

Release 2.0.1

Jonathan P Dawson

May 26, 2018

Contents

CHAPTER 1

Introduction

Chips is a high level, FPGA design tool inspired by *Python*.

1.1 Design components in C, design FPGAs in Python

A Chips design resembles a network of computers implemented in a single chip. A chip consists of many interconnected components operating in parallel. Each component acts like a computer running a C program.

Components communicate with each other sending messages across buses. The design of a chip - the components and the connections between them - is carried out in Python.

Chips come in three parts:

1. A Python library to build and simulate chips by connecting together digital components using high speed buses.
2. A collection of ready made digital components.
3. A C-to-hardware compiler to make new digital components in the C programming language.

1.2 A quick example

```
from chips.api.api import *

#create a new chip
chip = Chip("knight_rider")

#define a component in C
scanner = Component(C_file = """

    /* Knight Rider */
    unsigned leds = output("leds");
    void main (){
```

(continues on next page)

(continued from previous page)

```
        unsigned i;
        while(1)
        {
            for(i=1; i<=0x80; i<<=1) fputc(i, leds);
            for(i=0x80; i>=1; i>>=1) fputc(i, leds);
        }
    }

""", inline=True)

#capture simulation output in Python
scanner_output = Response(chip, "scanner", "int")

#add scanner to chip and connect
scanner(chip, inputs = {}, outputs = {"leds":scanner_output})

#generate synthesisable verilog code
chip.generate_verilog()

#run simulation in Python
chip.simulation_reset()
while len(scanner_output) < 16:
    chip.simulation_step()

#check the results
print list(scanner_output)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 128, 64, 32, 16, 8, 4, 2, 1]
```

1.3 Work at a higher level of abstraction

In Chips, the details of gates, clocks, resets, finite-state machines and flow-control are handled by the tool, this leaves the designer free to think about the architecture and the algorithms. This has some benefits:

- Designs are simpler.
- Simpler designs take much less time to get working.
- Simpler designs are much less likely to have bugs.

1.4 With Chips the batteries *are* included

With traditional Hardware Description Languages, there are many restrictions on what can be translated into hardware and implemented in a chip.

With Chips almost all legal code can be translated into hardware. This includes division, single and double precision IEEE floating point, maths functions, trig-functions, timed waits, pseudo-random numbers and recursive function calls.

1.5 Python is a rich verification environment

Chips provides the ability to simulate designs natively in Python. Python is an excellent programming language with extensive libraries covering many application domains. This makes it the perfect environment to verify a chip.

NumPy , SciPy and MatPlotLib will be of interest to engineers, but thats just the start .

1.6 Under the hood

Behind the scenes, Chips uses some novel techniques to generate compact and efficient logic - a hybrid of software and hardware.

Not only does the compiler translate the C code into CPU instructions, it also generates a customised pipelined RISC CPU on the fly. The CPU provides the optimal instruction set for any particular C program.

By minimising the logic required to perform each concurrent task, designers can reduce power and area or cost. Performance gains can be achieved by increasing the number of concurrent tasks in a single device (tens in a small device to around a thousand or more large device).

While the code generated by chips is compact and efficient, die hard FPGA designers will be pleased to know that they can still hand craft performance critical data paths if they need to. There are even a few hand crafted components thrown in!

1.7 Try it out

Why not try the Chips web app.

CHAPTER 2

Installation

2.1 Download (github)

You can download the source from the Git Hub homepage. Alternatively clone the project using git:

```
~$ git clone --recursive https://github.com/dawsonjon/Chips-2.0.git
```

2.2 Install from github

```
$ cd Chips-2.0
$ sudo python setup install
```

2.3 Install from PyPi

```
$ pip-install chips
```

2.4 Icarus Verilog

Chips can automatically simulate the verilog it generates, to simulate verilog you will need the Icarus Verilog simulator. This will need to be installed in your command path.

2.5 C Preprocessor

Chips uses an external C processor. Make sure you have a c preprocessor *cpp* installed in your path

2.6 Other packages

While not strictly speaking dependencies, you may want to install the following packages to use all the libraries and examples:

- numpy
- scipy
- matplotlib
- pil
- wxpython

CHAPTER 3

Examples

3.1 Calculate Square Root using Newton's Method

In this example, we calculate the sqrt of a number using Newton's method. The problem of finding the square root can be expressed as:

$$n = x^2$$

Which can be rearranged as:

$$f(x) = x^2 - n$$

Using Newton's method, we can find numerically the approximate point at which $f(x) = 0$. Repeated applications of the following expression yield increasingly accurate approximations of the Square root:

$$f(x_k) = x_{k-1} - \frac{{x_{k-1}}^2 - n}{2x_{k-1}}$$

Turning this into a practical solution, the following code calculates the square root of a floating point number. An initial approximation is refined using Newton's method until further refinements agree to within a small degree.

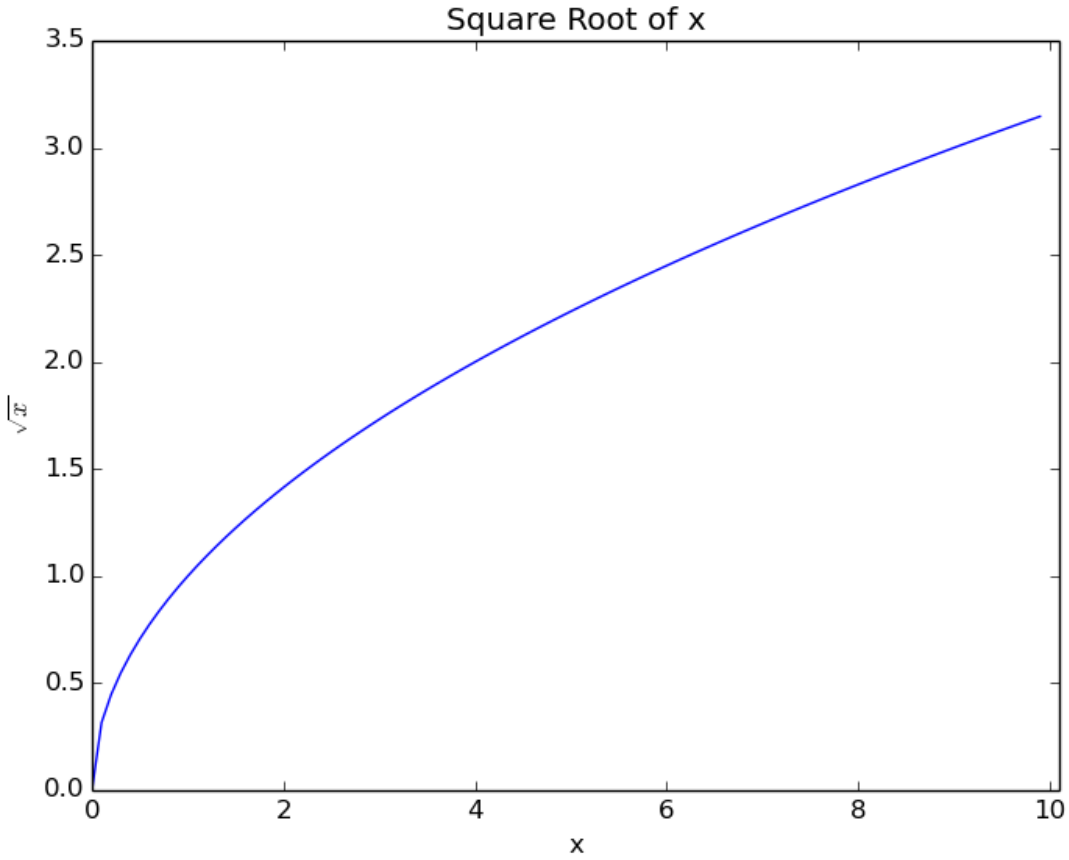
```
/* sqrt.c */
/* Jonathan P Dawson */
/* 2013-12-23 */

#include <stdio.h>

/* approximate sqrt using newton's method*/
double sqrt(double n){
    double square, x, old;
    x = n;
    old = 0.0;
    while(old != x){
        old = x;
```

(continues on next page)

(continued from previous page)

```
        x = (x + n/x)*0.5;
    }
    return x;
}

/* test sqrt function*/
const int x_in = input("x");
const int sqrt_x_out = output("sqrt_x");
void main(){
    double x;
    while(1){
        x = fget_float(x_in);
        fput_float(sqrt(x), sqrt_x_out);
    }
}
```

Note that the code isn't entirely robust, and cannot handle special cases such as Nans, infinities or negative numbers. A simple test calculates $\sqrt{x}$ where $-10 < x < 10$.



3.2 Approximating Sine and Cosine functions using Taylor Series

In this example, we calculate an approximation of the cosine functions using the Taylor series:

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

The following example uses the Taylor Series approximation to generate the Sine and Cosine functions. Successive terms of the taylor series are calculated until successive approximations agree to within a small degree. A Sine function is also synthesised using the identity $sin(x) \equiv cos(x - \pi/2)$

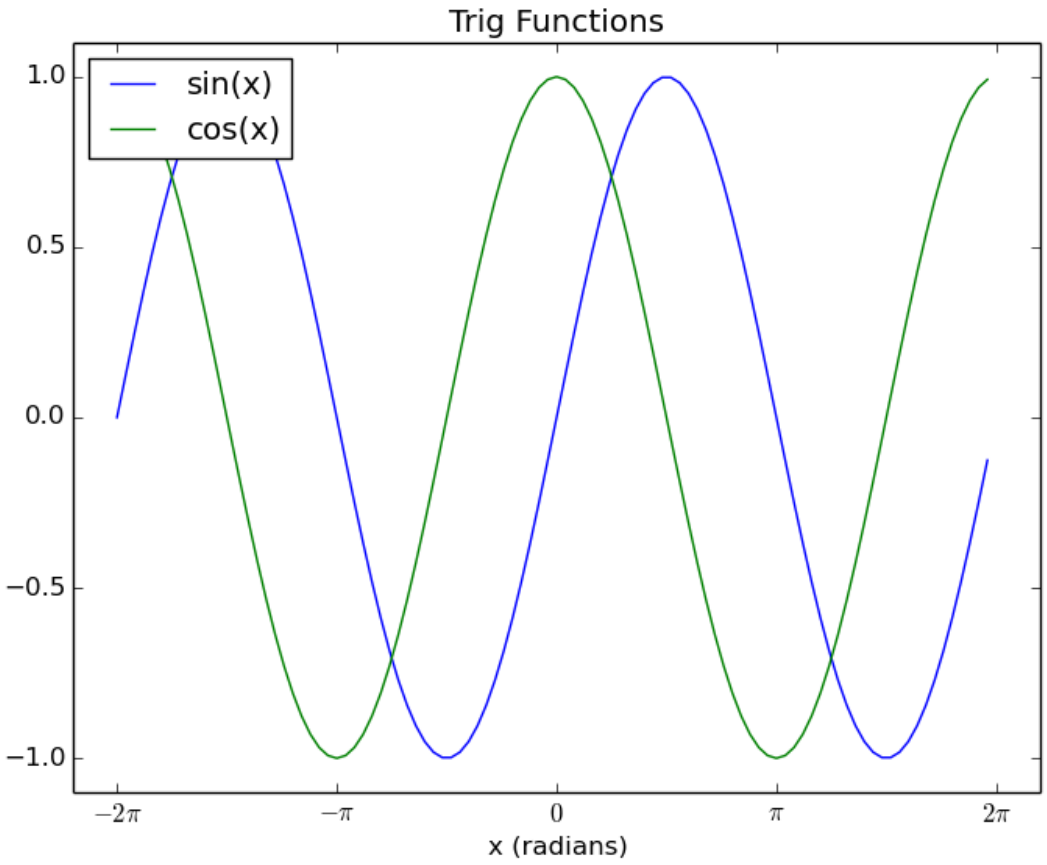
```
/* taylor.c */
/* Jonathan P Dawson */
/* 2013-12-23 */

#include <stdio.h>

/* globals */
double pi=3.14159265359;

/* approximate the cosine function using Taylor series */

double taylor(double angle){

    double old, approximation, sign, power, fact;
    unsigned count, i;

    approximation = angle;
    old = 0.0;
    sign = -1.0;
    count = 1;
    power = 1.0;
    fact = 1.0;

    for(i=3; approximation!=old; i+=2){
        old = approximation;

        while(count<=i){
            power*=angle;
            fact*=count;
            count++;
        }

        approximation += sign*(power/fact);
        sign = -sign;

    }
    return approximation;
}


/* return the sine of angle in radians */

double sin(double angle){

    return taylor(angle);
```

(continues on next page)

(continued from previous page)

```
}

/* return the cosine of angle in radians */

double cos(double angle){

    return sin(angle+(pi/2));

}


/* test routine */
const int x_in = input("x");
const int sin_x_out = output("sin_x");
const int cos_x_out = output("cos_x");

void main(){
    double x;

    while(1){
        x = fget_double(x_in);
        fput_double(sin(x), sin_x_out);
        fput_double(cos(x), cos_x_out);
    }

}
```

A simple test calculates Sine and Cosine for the range $-2\pi <= x <= 2\pi$.



3.3 Implement Quicksort

This example sorts an array of data using the Quick Sort algorithm

The quick-sort algorithm is a recursive algorithm, but *Chips* does not support recursive functions. Since the level of recursion is bounded, it is possible to implement the function using an explicitly created stack.

```
/* sort.c */
/* Jonathan P Dawson */
/* 2013-12-23 */

/* Based on the in-place algorithm on the Wikipedia page */
/* http://en.wikipedia.org/wiki/Quicksort#In-place_version */

/*globals*/
const unsigned length = 32;

/* partition subarray */
unsigned partition(
        int array[],
        unsigned left,
        unsigned right,
        unsigned pivot_index)
```

(continues on next page)

(continued from previous page)

```
{

    int temp, pivot_value;
    unsigned i, store_index;

    store_index = left;
    pivot_value = array[pivot_index];

    temp = array[pivot_index];
    array[pivot_index] = array[right];
    array[right] = temp;

    for(i=left; i<right; i++){
        if(array[i] <= pivot_value){
            temp = array[store_index];
            array[store_index] = array[i];
            array[i] = temp;
            store_index++;
        }
    }

    temp = array[store_index];
    array[store_index] = array[right];
    array[right] = temp;

    return store_index;
}


/* recursive sort */
void quick_sort(int array[], unsigned left, unsigned right){
    unsigned pivot;

    /* if the subarray has two or more elements */
    if (left < right){

        /* partition sub array into two further sub arrays */
        pivot = (left + right) >> 1;
        pivot = partition(array, left, right, pivot);

        /* push both subarrays onto stack */
        quick_sort(array, left, pivot-1);
        quick_sort(array, pivot+1, right);
    }

}

void main(){
    int array[length];
    unsigned i;

    /* Fill array with zeros */
    for(i=0; i<length; i++){
        array[i] = 0;
    }

    /* Add unsorted data to the array */
```

(continues on next page)

(continued from previous page)

```
    array[0] = 16;
    array[1] = 15;
    array[2] = 14;
    array[3] = 13;
    array[4] = 12;
    array[5] = 11;
    array[6] = 10;
    array[7] = 9;
    array[8] = 8;
    array[9] = 7;
    array[10] = 6;
    array[11] = 5;
    array[12] = 4;
    array[13] = 3;
    array[14] = 2;
    array[15] = 1;

    /* Sort the array */
    quick_sort(array, 0, length-1);

    for(i=0; i<length; i++){
        report(array[i]);
    }

}
```

The algorithm is tested using an array containing out of order values. The program correctly sorts the array:

```
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 0 (report at line: 96 in file: sort.c)
 1 (report at line: 96 in file: sort.c)
 2 (report at line: 96 in file: sort.c)
 3 (report at line: 96 in file: sort.c)
 4 (report at line: 96 in file: sort.c)
 5 (report at line: 96 in file: sort.c)
 6 (report at line: 96 in file: sort.c)
 7 (report at line: 96 in file: sort.c)
 8 (report at line: 96 in file: sort.c)
 9 (report at line: 96 in file: sort.c)
10 (report at line: 96 in file: sort.c)
11 (report at line: 96 in file: sort.c)
12 (report at line: 96 in file: sort.c)
13 (report at line: 96 in file: sort.c)
```

(continues on next page)

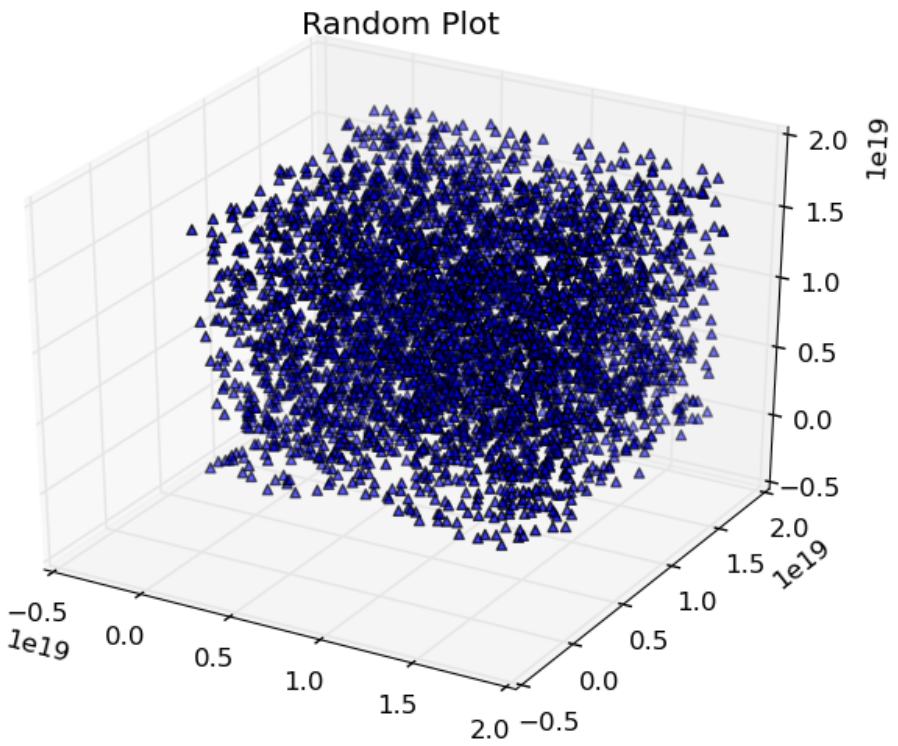
(continued from previous page)

```
14 (report at line: 96 in file: sort.c)
15 (report at line: 96 in file: sort.c)
16 (report at line: 96 in file: sort.c)
```

3.4 Pseudo Random Number Generator

This example uses a Linear Congruential Generator (LCG) to generate Pseudo Random Numbers.

```
/*globals*/
unsigned long int seed;

void srand(unsigned int s){
    seed = s;
}

unsigned long rand(){
    const unsigned a = 1103515245u;
    const unsigned c = 12345u;
    seed = (a*seed+c);
    return seed;
}

void main(){
    unsigned i;
    for (i=0; i<4096; i++){
        file_write(rand(), "x");
        file_write(rand(), "y");
        file_write(rand(), "z");
    }
}
```



3.5 Fast Fourier Transform

This example builds on the Taylor series example. We assume that the sin and cos routines have been placed into a library of math functions math.h, along with the definitions of π, M_PI.

The Fast Fourier Transform (FFT) is an efficient method of decomposing discretely sampled signals into a frequency spectrum, it is one of the most important algorithms in Digital Signal Processing (DSP). The Scientist and Engineer's Guide to Digital Signal Processing gives a straight forward introduction, and can be viewed on-line for free.

The example shows a practical method of calculating the FFT using the Cooley-Tukey algorithm.

```
/* fft.c */
/* Jonathan P Dawson */
/* 2013-12-23 */

#include <math.h>
#include <stdio.h>

/*globals*/
const int n = 1024;
const int m = 10;
double twiddle_step_real[m];
double twiddle_step_imaginary[m];
```

(continues on next page)

(continued from previous page)

```
/*calculate twiddle factors and store them*/
void calculate_twiddles(){
    unsigned stage, span;
    for(stage=0; stage<m; stage++){
        span = 1 << stage;
        twiddle_step_real[stage] = cos(M_PI/span);
        twiddle_step_imaginary[stage] = -sin(M_PI/span);
    }
}

/*bit reverse*/
unsigned bit_reverse(unsigned forward){
    unsigned reversed=0;
    unsigned i;
    for(i=0; i<m; i++){
        reversed <<= 1;
        reversed |= forward & 1;
        forward >>= 1;
    }
    return reversed;
}

/*calculate fft*/
void fft(double reals[], double imaginaries[]){

    int stage, subdft_size, span, i, ip, j;
    double sr, si, temp_real, temp_imaginary, imaginary_twiddle, real_twiddle;


    //read data into array
    for(i=0; i<n; i++){
        ip = bit_reverse(i);
        if(i < ip){
            temp_real = reals[i];
            temp_imaginary = imaginaries[i];
            reals[i] = reals[ip];
            imaginaries[i] = imaginaries[ip];
            reals[ip] = temp_real;
            imaginaries[ip] = temp_imaginary;
        }
    }

    //butterfly multiplies
    for(stage=0; stage<m; stage++){
        subdft_size = 2 << stage;
        span = subdft_size >> 1;

        //initialize trigonometric recurrence

        real_twiddle=1.0;
        imaginary_twiddle=0.0;

        sr = twiddle_step_real[stage];
        si = twiddle_step_imaginary[stage];
```

(continues on next page)

(continued from previous page)

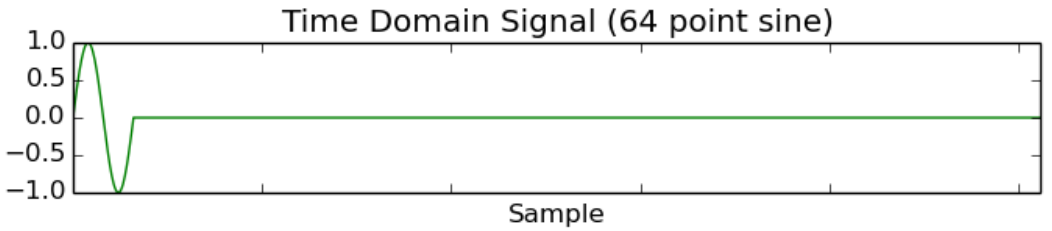
```
        report(stage);

        for(j=0; j<span; j++){
            for(i=j; i<n; i+=subdft_size){
                ip=i+span;

                temp_real      = reals[ip]*real_twiddle      -␣
↪imaginaries[ip]*imaginary_twiddle;
                temp_imaginary = reals[ip]*imaginary_twiddle + imaginaries[ip]*real_
↪twiddle;

                reals[ip]       = reals[i]-temp_real;
                imaginaries[ip] = imaginaries[i]-temp_imaginary;

                reals[i]       = reals[i]+temp_real;
                imaginaries[i] = imaginaries[i]+temp_imaginary;

            }
            //trigonometric recurrence
            temp_real=real_twiddle;
            real_twiddle      = temp_real*sr - imaginary_twiddle*si;
            imaginary_twiddle = temp_real*si + imaginary_twiddle*sr;
        }

    }

}

const int x_re_in = input("x_re");
const int x_im_in = input("x_im");
const int fft_x_re_out = output("fft_x_re");
const int fft_x_im_out = output("fft_x_im");

void main(){
    unsigned i;
    double reals[n];
    double imaginaries[n];

    /* pre-calculate sine and cosine*/
    calculate_twiddles();

    while(1){
        /* read time domain signal */
        for(i=0; i<n; i++){
            reals[i] = fget_double(x_re_in);
            imaginaries[i] = fget_double(x_im_in);
        }

        /* transform into frequency domain */
        fft(reals, imaginaries);

        /* output frequency domain signal*/
        for(i=0; i<n; i++){
            fput_double(reals[i], fft_x_re_out);
            fput_double(imaginaries[i], fft_x_im_out);
        }
    }
```

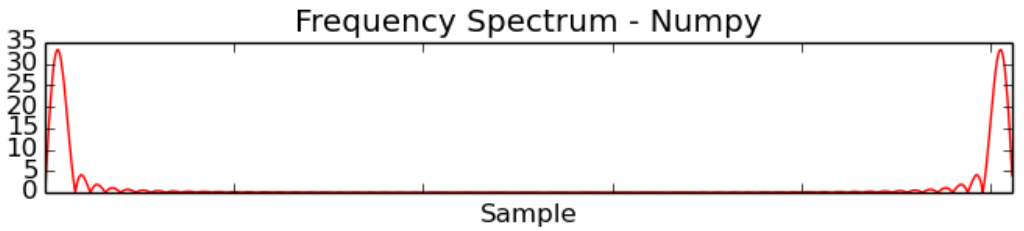
(continues on next page)

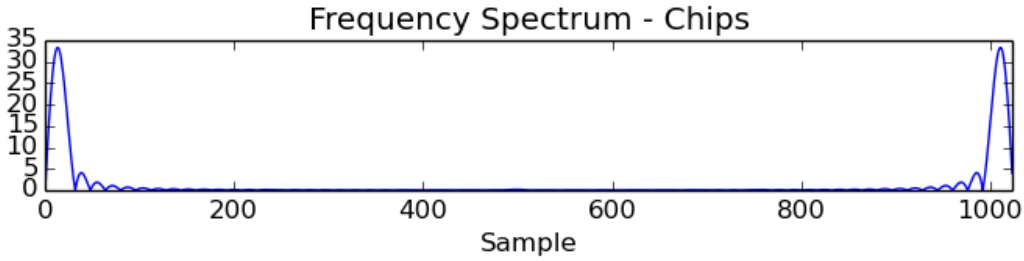
(continued from previous page)

```
}
```

The C code includes a simple test routine that calculates the frequency spectrum of a 64 point sine wave.







3.6 FIR Filter

An FIR filter contains a tapped delay line. By applying a weighting to each tap, and summing the results we can create a filter. The coefficients of the filter are critical. Here we create the coefficients using the firwin function from the SciPy package. In this example we create a low pass filter using a Blackman window. The Blackman window gives good attenuation in the stop band.

```
    from math import pi
from numpy import abs
from scipy import fft
from scipy.signal import firwin
from matplotlib import pyplot
from chips.api.api import Chip, Stimulus, Response, Wire, Component

#create a chip
chip = Chip("filter_example")
```

(continues on next page)

(continued from previous page)

```
#low pass filter half nyquist 50 tap
kernel = Stimulus(chip, "kernel", "float", firwin(50, 0.5, window="blackman"))

#impulse response
input_ = Stimulus(chip, "input", "float", [1.0] + [0.0 for i in range(1024)])
output = Response(chip, "output", "float")

#create a filter component using the C code
fir_comp = Component("fir.c")

#add an instance to the chip
fir_inst_1 = fir_comp(
    chip,
    inputs = {
        "a":input_,
        "k":kernel,
    },
    outputs = {
        "z":output,
    },
    parameters = {
        "N":len(kernel)-1,
    },
)

#run the simulation
chip.simulation_reset()
while len(output) < 1024:
    chip.simulation_step()

#plot the result
pyplot.semilogy(abs(fft(list(output)))[0:len(output)/2])
pyplot.title("Magnitude of Impulse Response")
pyplot.xlim(0, 512)
pyplot.xlabel("X Sample")
pyplot.savefig("../docs/source/examples/images/example_6.png")
pyplot.show()
```

The C code includes a simple test routine that calculates the frequency spectrum of a 64 point sine wave.

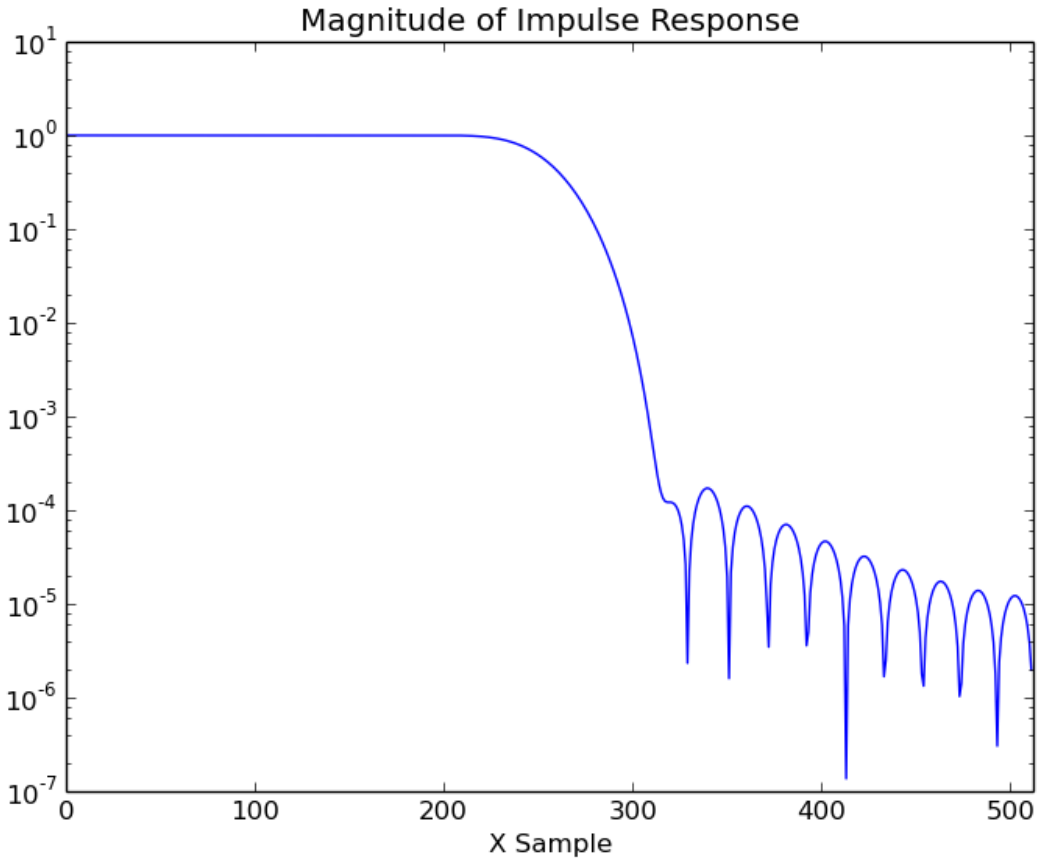
```
/* Chips-2.0 FIR Filter Example */
/* Jonathan P Dawson 2014-07-05 */

#include <stdio.h>

unsigned in = input("a");
unsigned out = output("z");
unsigned kernel_in = input("k");

void main(){
    unsigned i = 0;
    unsigned inp = 0;
    float delay[N];
    float kernel[N];
    float data_out;

    /* read in filter kernel */
```

(continues on next page)

(continued from previous page)

```
    for(i=0; i<N; i++){
       kernel[i] = fget_float(kernel_in);
    }


    /* execute filter on input stream */
    while(1){
        delay[inp] = fget_float(in);
        data_out=0.0; i=0;
        while(1){
            data_out += delay[inp] * kernel[i];
            if(i == N-1) break;
            i++;
            if(inp == N-1){
                inp=0;
            }else{
                inp++;
            }
        }
        fput_float(data_out, out);
    }
}
```

Increasing the length of the filter kernel results in a faster roll-off and greater attenuation.



While in this example, we calculate all the coefficients inside a single process, it is possible to generate a pipelined implementation, and allow the work to be carried out by multiple processes resulting in an increase in the throughput rate.

The Scientist and Engineer's Guide to Digital Signal Processing gives a straight forward introduction, and can be viewed on-line for free.

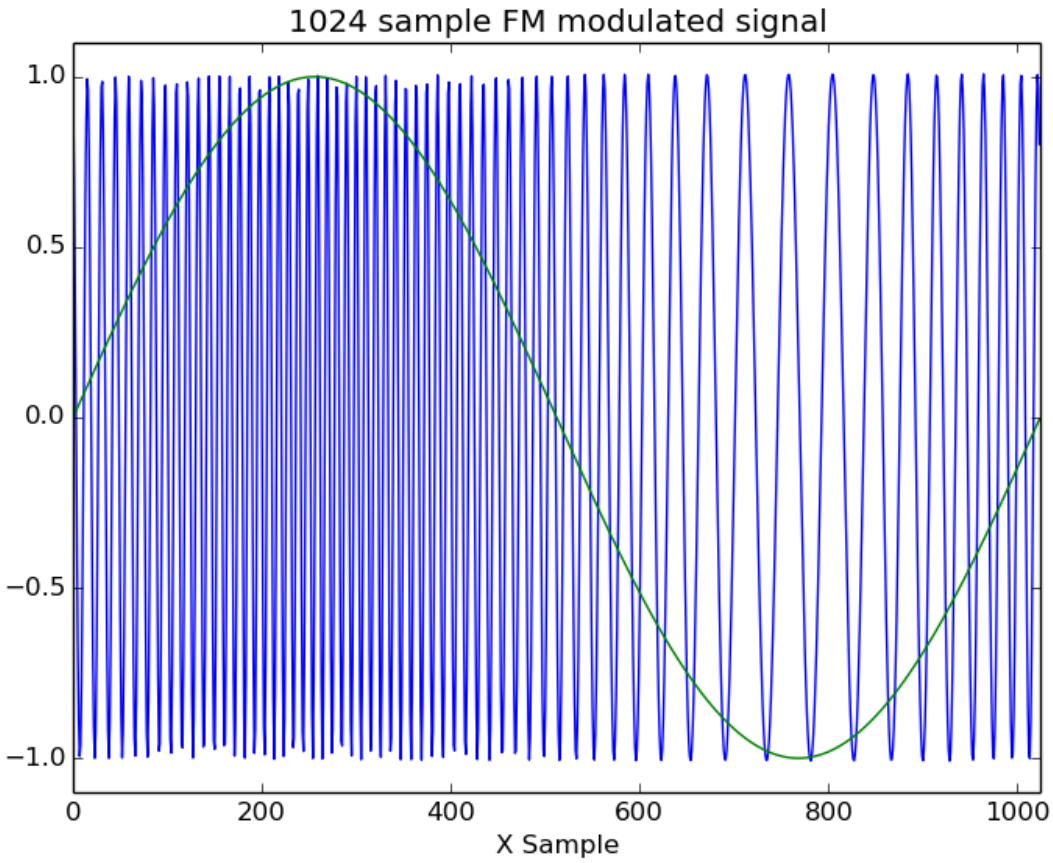
3.7 FM Modulation

It is often useful in digital hardware to simulate a sin wave numerically. It is possible to implements a sinusoidal oscillator, without having to calculate the value of the sinusoid for each sample. A typical approach to this in hardware is to store within a lookup table a series of values, and to sweep through those values at a programmable rate. This method relies on a large amount of memory, and the memory requirements increase rapidly for high resolutions. It is possible to improve the resolution using techniques such as interpolation.

In this example however, an alternative method is employed, trigonometric recurrence allows us to calculate the sin and cosine of a small angle just once. From there, subsequent samples can be found using multipliers.

```
#include <stdio.h>
#include <math.h>

unsigned frequency_in = input("frequency");
unsigned sin_out = output("sin");
unsigned cos_out = output("cos");

void main(){
    float sin_x, cos_x, new_sin, new_cos, si, sr, frequency;
    int i;

    cos_x = 1.0;
    sin_x = 0.0;
    sr = cos(2.0 * M_PI/N);
    si = sin(2.0 * M_PI/N);

    while(1){
        frequency = fget_float(frequency_in);
        for(i=0; i<frequency; i++){
            new_cos = cos_x*sr - sin_x*si;
            new_sin = cos_x*si + sin_x*sr;
            cos_x = new_cos;
            sin_x = new_sin;
        }
        fput_float(cos_x, sin_out);
        fput_float(sin_x, cos_out);
    }

}
```

Conveniently, using this method, both a sin and cosine wave are generated. This is useful in complex mixers which require a coherent sin and cosine wave. We can control the frequency of the generated wave by stepping through the waveform more quickly. If the step rate is received from an input, this can be used to achieve frequency modulation.



3.8 Edge Detection

This simple example shows how a simple 3x3 convolution matrix can be used to perform an *edge detect* operation on a grey-scale image. The convolution matrix is the "quick mask" matrix presented in Image Processing in C which also gives a straight forward introduction to edge detection algorithms.

The Python Imaging Library allows real images to be used in the simulation.

```
/*Edge Detection*/
/*Jonathan P Dawson 2014-07-06*/

void set_xy(int image[], int x, int y, int pixel){
    if(x<0) return;
    if(x>=WIDTH) return;
    image[x+y*WIDTH] = pixel;
}

int get_xy(int image[], int x, int y){
    if(x<0) return 0;
    if(x>=WIDTH) return 0;
    return image[x+y*WIDTH];
}
```

(continues on next page)

(continued from previous page)

```
void main()
{

    unsigned image_in = input("image_in");
    unsigned image_out = output("image_out");

    unsigned image[SIZE];
    unsigned new_image[SIZE];

    int x, y, pixel;

    while(1){

        /* read in image */
        for(y=0; y<HEIGHT; y++){
            for(x=0; x<WIDTH; x++){
                set_xy(image, x, y, fgetc(image_in));
            }
            report(y);
        }

        /* apply edge detect */
        for(y=0; y<HEIGHT; y++){
            for(x=0; x<WIDTH; x++){

                pixel =  get_xy(image, x,   y  ) << 2;
                pixel -= get_xy(image, x-1, y+1);
                pixel -= get_xy(image, x+1, y-1);
                pixel -= get_xy(image, x-1, y-1);
                pixel -= get_xy(image, x+1, y+1);
                set_xy(new_image, x, y, pixel);
            }
            report(y);
        }

        /* write out image */
        for(y=0; y<HEIGHT; y++){
            for(x=0; x<WIDTH; x++){
                fputc(get_xy(new_image, x, y), image_out);
            }
            report(y);
        }

    }
}
```

3.9 LZSS Compression

LZSS is a simple form of of run length compression that exploits repeated sequences in a block of data. The encoder scans a block of data, and sends literal characters. However if the encoder encounters a sequence of characters that have already been sent, it will substitute the sequence with a reference to the earlier data. The encoder will always

select the longest matching sequence that it has already sent. To achieve this the encoder needs to store a number of previously sent characters in a buffer. This buffer is referred to as the window.

```
/*LZSS Compression Component*/
/*Jonathan P Dawson 2014-07.10*/

unsigned raw_in = input("raw_in");
unsigned compressed_out = output("compressed_out");

/*Send data of an arbitrary bit length*/
unsigned packed, stored = 0;
void send_bits(unsigned data, unsigned bits){
    unsigned i;
    for(i=0; i<bits; i++){
        packed >>= 1;
        packed |= (data & 1) << 31;
        data >>= 1;
        stored++;
        if(stored == 32){
            fputc(packed, compressed_out);
            stored = 0;
        }
    }
}

/*A function that reads a stream of uncompressed data,
and creates a stream of compressed data*/

void main(){

    unsigned pointer, match, match_length, longest_match, longest_match_length;
    unsigned buffer[N];

    unsigned new_size;

    while(1){
        for(pointer=0; pointer<N; pointer++){
            buffer[pointer] = fgetc(raw_in);
        }

        pointer=0;
        new_size = 0;
        while(pointer<N){

            /*Find the longest matching string already sent*/
            longest_match = 0;
            longest_match_length = 0;
            for(match=0; match<pointer; match++){

                /*match length of 0 indicates no match*/
                match_length = 0;

                /*search through buffer to find a match*/
                while(buffer[match+match_length] == buffer[pointer+match_length]){
                    match_length++;
                }

                /*If this is the longest match, remember it*/
```

(continues on next page)

(continued from previous page)

```
                if(match_length > longest_match_length){
                    longest_match = match;
                    longest_match_length = match_length;
                }

            }

            /*send data*/
            if(longest_match_length >= 3){
                send_bits(0, 1);
                send_bits(longest_match_length, LOG2N);
                send_bits(pointer - longest_match, LOG2N);
                pointer += longest_match_length;
                new_size += LOG2N + LOG2N + 1;
            }
            else{
                send_bits(1, 1);
                send_bits(buffer[pointer], 8);
                pointer++;
                new_size += 9;
            }

            report(pointer);
        }
        /*report the compression ratio of this block in simulation*/
        report(new_size / (8.0*N));
    }
}
```

The encoding is simple. A bit is sent to indicate whether a raw character or a reference continues. A reference consists of a distance length pair. The distance tells the decoder how many characters ago the matching sequence was sent, and the distance indicates the length of the matching sequence. The size of the distance and length pointers will depend on the size of the window, for example a window size of 1024 requires the pointers to be 10 bits each.

```
/*LZSS Decmpression Component*/
/*Jonathan P Dawson 2014-07-10*/

unsigned raw_out = output("raw_out");
unsigned compressed_in = input("compressed_in");

/*A function to get data of an arbitrary bit length*/

unsigned stored = 0;
unsigned packed;
unsigned get_bits(unsigned bits){
    unsigned i, value = 0;
    for(i=0; i<bits; i++){
        if(!stored){
            stored = 32;
            packed = fgetc(compressed_in);
        }
        value >>= 1;
        value |= (packed & 1) << 31;
        packed >>= 1;
        stored--;
    }
```

(continues on next page)

(continued from previous page)

```
    return value >> (32 - bits);
}


/*Decompress a stream of lzss compressed data,
and generate a stream of raw data*/

void main(){
    unsigned i, pointer, distance, length, data;
    unsigned buffer[N];

    while(1){

        /*get distance length*/
        if(get_bits(1)){
            data = get_bits(8);
            buffer[pointer] = data;
            pointer++;
            fputc(data, raw_out);
        }
        else{
            length = get_bits(LOG2N);
            distance = get_bits(LOG2N);
            for(i=0; i<length; i++){
                data = buffer[pointer-distance];
                buffer[pointer] = data;
                pointer++;
                fputc(data, raw_out);
            }
        }
    }
}
```

In the simulation, a short passage of text is compressed by the encoder component, sent to the decoder component, decompressed and recovered. A fuller explanation may be found on wikipedia.

CHAPTER 4

Reference Manual

4.1 Python API

The Python API provides the ability to build systems from C components. Designs can be simulated using Python as a rich verification environment. Designs can be converted into Verilog and targeted to FPGAs using the FPGA vendors synthesis tools.

```
from chips.api.api import *
```

class chips.api.api.**Chip**(*name*)

A chip is a canvas to which you can add inputs outputs, components and wires. When you create a chips all you need to give it is a name.

```
mychip = Chip("mychip")
```

The interface to a *Chip* may defined by calling *Input* and *Output* objects. Each input and output object is given a name. The name will be used in the generated Verilog. While any string can be used for the name, if you intend to generate Verilog output, the name should be a valid Verilog identifier.

```
Input(mychip, "input_1")
Input(mychip, "input_2")
Output(mychip, "output_1")
```

The implementation of a chip is defined by creating and instancing components. For example a simple adder can be created as follows:

```
#define a component
adder = Component(C_file = """
        input_1 = input("input_1");
        input_2 = input("input_2");
        output_1 = output("output_1");
        void main(){
            while(1){
```

(continues on next page)

(continued from previous page)

```
                fputc(fgetc(input_1)+fgetc(input_2), output_1);
            }
        }
""", inline=True)
```

The adder can then be instanced, and connected up by calling the adder. When an added in instanced, the *inputs* and *outputs* arguments should be supplied. These dictionaries specify how the inputs and outputs of the component should be connected. The dictionary key should be the input/output name, and the value should be an *Input*, *Output* or *Wire* instance:

```
#instance a component
my_adder = adder(
    inputs = {
        "input_1" : input_1,
        "input_2" : input_2,
    },
    outputs = {
        "output_1" : output_1,
    },
)
```

HDLs provide the ability define new components by connecting components together. Chips doesn't provide a means to do this. There's no need. A Python function does the job nicely. A function can be used to build a four input adder out of 2 input adders for example:

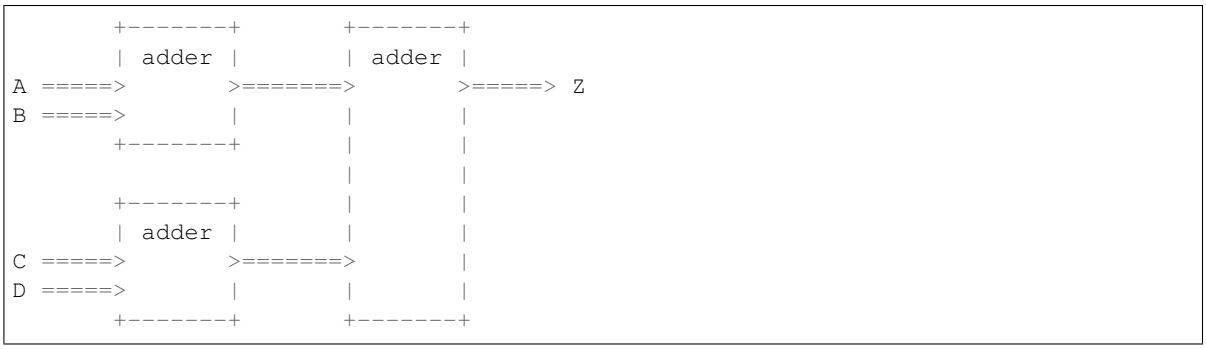
```
def four_input_adder(chip, input_a, input_b, input_c, input_d, output_z):

    adder = Component(C_file = """
            a = input("a");
            b = input("b");
            z = output("z");
            void main(){
                while(1){
                    fputc(fgetc(a)+fgetc(b), z);
                }
            }
    """, inline=True)

    wire_a = Wire(chip)
    wire_b = Wire(chip)

    adder(mychip,
        inputs = {"a" : input_a, "b" : input_b},
        outputs = {"z" : wire_a})

    adder(mychip,
        inputs = {"a" : input_c, "b" : input_d},
        outputs = {"z" : wire_b})

    adder(mychip,
        inputs = {"a" : wire_a, "b" : wire_b},
        outputs = {"z" : output_z})

mychip = Chip("mychip")
input_a = Input(mychip, "a")
input_b = Input(mychip, "b")
```

(continues on next page)

(continued from previous page)

```
input_c = Input(mychip, "c")
input_d = Input(mychip, "d")
output_z = Output(mychip, "z")
four_input_adder(mychip, input_a, input_b, input_c, input_d, output_z)
```

A diagrammatic representation of the *Chip* is shown below.



Functions provide a means to build more complex components out of simple ones, but it doesn't stop there. By providing the basic building blocks, you can use all the features of the Python language to build chips.

Ideas:

- Create multiple instances using loops.
- Use tuples, arrays or dictionaries to group wires into more complex structures.
- Use a GUI interface to customise a components or chips.
- Build libraries of components using modules or packages.
- Document designs with docutils or sphinx.

There are two ways to transfer data between the python environment, and the *Chip* simulation.

The first and most flexible method is to subclass the *Input* and *Output* classes, overriding the data_source and data_sink methods. By defining your own data_source and data_sink methods, you can interface to other Python code. Using this method allows the simulation to interact with its environment on the fly.

The second simpler method is to employ the *Stimulus* or *Response* classes. These classes are themselves inherited from *Input* and *Output*. The *Stimulus* class is provided with a Python sequence object for example a list, and iterator or a generator at the time it is created created. The *Response* class store data as the simulation progresses, and is itself a sequence object.

It is simple to run the simulation, which should be initiated with a reset:

```
mychip.simulation_reset()
```

The simulation can be run for a single cycle:

```
mychip.simulation_step()
```

The *simulation_run* method executes the simulation until all processes complete (which may not happen):

```
mychip.simulation_run()
```

There are a couple of methods to terminate the simulation, by waiting for simulation time to elapse, or for a certain amount of output data to be accumulated.

```
#run simulation for 1000 cycles
while mychip.time < 1000:
    mychip.simulation_step()


#run simulation until 1000 data items are collected
response = Response(chip, "output", "int")
while len(response) < 1000:
    mychip.simulation_step()
```

Chips designs can be programmed into FPGAs. Chips uses Verilog as its output because it is supported by FPGA vendors build tools. Chips output almost will be compatible with any FPGA family. Synthesisable Verilog code s generated by calling the *generate_verilog* method.

```
mychip.generate_verilog()
```

You can also generate a matching testbench using the *generate_testbench* method. You can also specify the simulation run time in clock cycles.

```
mychip.generate_testbench(1000) #1000 clocks
```

To compile the design in Icarus Verilog, use the *compile_iverilog* method. You can also run the code directly if you pass *True* to the *compile_iverilog* function. This is most useful to verify that chips components match their native python simulations. In most cases Verilog simulations will only be needed to by *Chips* developers.

```
mychip.compile_iverilog(True)
```

The generated Verilog code is dependent on the chips_lib.v file which is output alongside the synthesisable Verilog.

compile_iverilog(*run=False*)

Synopsis:

```
chip.compile_iverilog(run=False)
```

Description:

Compile using the external iverilog simulator.

Arguments:

run: (optional) run the simulation.

Returns:

None

cosim()

Synopsis:

```
chip.generate_testbench(stop_clocks=None)
```

Description:

Generate a Verilog testbench.

Arguments:

stop_clocks: The number of clock cycles for the simulation to run.

Returns:

None

cosim_step()

Synopsis:

```
chip.cosim_step()
```

Description:

Run cosim for one cycle.

Arguments:

None

Returns:

None

generate_testbench(*stop_clocks=None*)

Synopsis:

```
chip.generate_testbench(stop_clocks=None)
```

Description:

Generate a Verilog testbench.

Arguments:

stop_clocks: The number of clock cycles for the simulation to run.

Returns:

None

generate_verilog()

Synopsis:

```
chip.generate_verilog(name)
```

Description:

Generate synthesisable Verilog output.

Arguments:

None

Returns:

None

simulation_reset()

Synopsis:

```
chip.simulation_reset()
```

Description:

Reset the simulation.

Arguments:

None

Returns:

None

simulation_run()

Synopsis:

```
chip.simulation_run()
```

Description:

Run the simulation until all processes terminate.

Arguments:

None

Returns:

None

simulation_step()

Synopsis:

```
chip.simulation_step()
```

Description:

Run the simulation for one cycle.

Arguments:

None

Returns:

None

class chips.api.api.**Component**(*C_file*, *options={}*, *inline=False*)

The Component class defines a new type of component.

Components are written in C. You can supply the C code as a file name, or directly as a string:

```
#Call an external C file
my_component = Component(C_file="adder.c")

#Supply C code directly
my_component = Adder(C_file="""
    unsigned in1 = input("in1");
    unsigned in2 = input("in2");
    unsigned out = input("out");
    void main(){
        while(1){
            fputc(fgetc(in1) + fgetc(in2), out);
        }
    }
""", inline=True)
```

Once you have defined a component you can use the __call__ method to create an instance of the component:

```
...

adder_1 = Adder(
    chip,
    inputs = {"in1":a, "in2":b},
```

(continues on next page)

(continued from previous page)

```
    outputs = {"out":z},
    parameters = {}
)
```

You can make many instances of a component by "calling" the component. Each time you make an instance, you must specify the *Chip* it belongs to, and connect up the inputs and outputs of the *Component*.

class chips.api.api.**Input**(*chip*, *name*)

An *Input* takes data from outside the *Chip*, and feeds it into the input of a *Component*. When you create an *Input*, you need to specify the *Chip* it belongs to, and the name it will be given.

```
input_a = Input(mychip, "A")
input_b = Input(mychip, "B")
input_c = Input(mychip, "C")
input_d = Input(mychip, "D")
```

In simulation, the Input calls the *data_source* member function, to model an input in simulation, subclass the Input and override the *data_source* method:

```
from chips.api.api import Input

stimulus = iter([1, 2, 3, 4, 5])

class MyInput(Input):

    def data_source(self):
        return next(stimulus)
```

data_source()

Override this function in your application

simulation_reset()

This is a private function, you shouldn't need to call this directly. Use Chip.simulation_reset() instead

simulation_step()

This is a private function, you shouldn't need to call this directly. Use Chip.simulation_step() instead

simulation_update()

This is a private function, you shouldn't need to call this directly. Use Chip.simulation_update() instead

class chips.api.api.**Output**(*chip*, *name*)

An *Output* takes data from a *Component* output, and sends it outside the *Chip*. When you create an *Output* you must tell it which *Chip* it belongs to, and the name it will be given.

In simulation, the *Output* calls the *data_source* member function, to model an output in simulation, subclass the Output and override the *data_sink* method:

```
from chips.api.api import Output

class MyOutput(Output):

    def data_sink(self, data):
        print data
```

data_sink()

override this function in your application

simulation_reset()

This is a private function, you shouldn't need to call this directly. Use Chip.simulation_reset() instead

simulation_step()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_step() instead

simulation_update()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_update() instead

class chips.api.api.**Wire**(*chip*)
A *Wire* is a point to point connection, a stream, that connects an output from one component to the input of another. A *Wire* can only have one source of data, and one data sink. When you create a *Wire*, you must tell it which *Chip* it belongs to:

```
wire_a = Wire(mychip)
wire_b = Wire(mychip)
```

simulation_reset()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_reset() instead

simulation_update()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_update() instead

class chips.api.api.**Stimulus**(*chip*, *name*, *type_*, *sequence*)
Stimulus is a subclass of Input. A Stimulus input provides a convenient means to supply data to the Chips simulation using any python sequence object for example a, list, an iterator or a generator.

```
from chips.api.api import Sequence

mychip = Chip("a chip")
...

Sequence(mychip, "counter", "int", range(1, 100))

mychip.simulation_reset()
mychip.simulation_run()
```

data_source()
This is a private function, you shouldn't need to call this directly.

simulation_reset()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_reset() instead

class chips.api.api.**Response**(*chip*, *name*, *type_*)
Response is a subclass of Output. A Response output provides a convenient means to extract data to the Chips simulation. A response behaves as a Python iterator.

```
from chips.api.api import Response

mychip = Chip("a chip")
...

sinx = Response(mychip, "sinx", "int")

mychip.simulation_reset()
mychip.simulation_run()

plot(sinx)
```

data_sink(*value*)
This is a private function, you shouldn't need to call this directly.

simulation_reset()
This is a private function, you shouldn't need to call this directly. Use Chip.simulation_reset() instead

class chips.api.api.**VerilogComponent**(*C_file*, *V_file*, *options={}*, *inline=False*)
The VerilogComponent is derived from Component. The VerilogComponent does not use the C Compiler to generate the Verilog implementation, but allows the user to supply Verilog directly. This is useful on occasions when hand-crafted Verilog is needed in a performance critical section, or if some pre-existing Verilog code needs to be employed.

```
my_component = Adder(C_file="adder.c", V_file="adder.v")
```

4.2 C Compiler

The heart of Chips is the C compiler and simulator. The C compiler allows you to define new hardware components using the C language. Components written in C can be simulated, or converted automatically into Verilog components. These Verilog components may then be implemented in an FPGA using tools provided by FPGA vendors.

The C dialect used in Chips has been made as standard as possible. This section of the manual describes the subset of the C language that is available in *Chips*.

4.2.1 Types

The following types are available in chips:

- *char*
- *int*
- *long*
- *unsigned char*
- *unsigned int*
- *unsigned long*
- *float*
- *double*

A *char* is at least 8 bits wide. An *int* is at least 32 bits wide. A *long* is at least 64 bits wide.

The *float* type is implemented as an IEEE 754 single precision floating point number. The *double* and *long double* types are implemented as IEEE 754 double precision floating point numbers. Round-to-zero (ties to even) is the only supported rounding mode.

Any type may be used to form an array, including *struct* s and arrays. Arrays may be nested arbitrarily to form arrays of arrays. *struct* s may be assigned, passed to and returned from functions.

Arrays may be passed to functions, in this case the array is not copied, but a reference is passed. Any modification made to the array within the called function will also be reflected within the calling function.

4.2.2 Missing Features

Chips is getting close to supporting the whole C language. The following features have not been done yet.

- *union* s

- *enum* s
- Variadic Functions

4.2.3 Functions

Recursion is permitted, but it should be remembered that memory is at a premium in FPGAs. It may be better to avoid recursive functions so that the memory usage can be predicted at compile time. Only a fixed number of arguments is supported, optional arguments are not permitted.

4.2.4 Control Structures

The usual control structures are supported.

4.2.5 Operators

The usual operators are supported.

4.2.6 Stream I/O

The language has been extended to allow components to communicate by sending data through streams.

To open an input or an output stream use the built in *input* and *output* functions. These functions accept a string argument identifying the name of the input or output. They return a file handle that can be passed to the *fgetc*, and *fputc* functions to send or receive data. You can also use the file I/O functions defined in *print.h*, *scan.h* and *stdio.h*.

```
unsigned spam = input("spam");
unsigned eggs = input("eggs");
unsigned fish = input("fish");
int temp;
temp = fgetc(spam); //reads from an input called spam
temp = fgetc(eggs); //reads from an input called eggs
fputc(temp, fish);   //writes to an output called fish
```

Reading or writing from inputs and outputs causes program execution to block until data is available. If you don't want to commit yourself to reading and input and blocking execution, you can check if data is ready.

```
unsigned spam = input("spam");
int temp;
if(ready(spam)){
   temp = fgetc(spam);
}
```

There is an equivilent *output_ready* function to check whether an output, is waiting for data. Care should be taken to avoid deadlocks which might arise if both the sender and receiver are waiting for the other to be waiting.

4.2.7 Timed Waits

Timed waits can be achieved using the built-in *wait-clocks* function. The wait_clocks function accepts a single argument, the numbers of clock cycles to wait.

```
wait_clocks(100); //wait for 1 us with 100MHz clock
```

4.2.8 Debug and Test

The built in *report* function displays the value of an expression in the simulation console. *This will have no effect in a synthesised design.*

```
int temp = 4;
report(temp); //prints 4 to console
report(10); //prints 10 to the console
```

The built in function assert causes a simulation error if it is passed a zero value. *The assert function has no effect in a synthesised design.*

```
int temp = 5;
assert(temp); //does not cause an error
int temp = 0;
assert(temp); //will cause a simulation error
assert(2+2==5); //will cause a simulation error
```

In simulation, you can write values to a file using the built-in *file_write* function. The first argument is the value to write, and the second argument is the file to write to. The file will be overwritten when the simulation starts, and subsequent calls will append a new vale to the end of the file. Each value will appear in decimal format on a separate line. A file write has no effect in a synthesised design.

```
file_write(1, "simulation_log.txt");
file_write(2, "simulation_log.txt");
file_write(3, "simulation_log.txt");
file_write(4, "simulation_log.txt");
```

You can also read values from a file during simulation. A simulation error will occur if there are no more value in the file.

```
assert(file_read("simulation_log.txt") == 1);
assert(file_read("simulation_log.txt") == 2);
assert(file_read("simulation_log.txt") == 3);
assert(file_read("simulation_log.txt") == 4);
```

4.2.9 C Preprocessor

Chips uses an external C pre-processor, you will need to make sure that Chips can see the *cpp* command in its command path.

4.3 C Libraries

4.3.1 ctype.h

The isalnum function

Synopsis:

```
#include <ctype.h>
int isalnum(int c);
```

Description:

The isalnum function tests for any character for which isalpha or isdigit is true.

The isalpha function

Synopsis:

```
#include <ctype.h>
int isalpha(int c);
```

Description:

The isalpha function tests for any character for which isupper or islower is true, or any of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, or isspace is true. In the C locale, isalpha returns true only for the characters for which isupper or islower is true.

The iscntrl function

Synopsis:

```
#include <ctype.h>
int iscntrl(int c);
```

Description:

The iscntrl function tests for any control character.

The isdigit function

Synopsis:

```
#include <ctype.h>
int isdigit(int c);
```

Description:

The isdigit function tests for any decimal-digit character.

The isgraph function

Synopsis:

```
#include <ctype.h>
int isgraph(int c);
```

Description:

The isgraph function tests for any printing character except space (‘ ‘).

The islower function

Synopsis:

```
#include <ctype.h>
int islower(int c);
```

Description:

The islower function tests for any lower-case letter or any of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, or isspace is true. In the C locale, islower returns true only for the characters defined as lower-case letters.

The isprint function

Synopsis:

```
#include <ctype.h>
int isprint(int c);
```

Description:

The isprint function tests for any printing character including space (‘ ‘).

The ispunct function

Synopsis:

```
#include <ctype.h>
int ispunct(int c);
```

Description:

The ispunct function tests for any printing character except space (‘ ‘) or a character for which isalnum is true.

The isspace function

Synopsis:

```
#include <ctype.h>
int isspace(int c);
```

Description:

The isspace function tests for the standard white-space characters or for any of an implementation-defined set of characters for which isalnum is false. The standard white-space characters are the following: space (‘ ‘), form feed (‘f’), new-line (‘n’), carriage return (‘r’), horizontal tab (‘t’), and vertical tab (‘v’). In the C locale, isspace returns true only for the standard white-space characters.

The isupper function

Synopsis:

```
#include <ctype.h>
int isupper(int c);
```

Description:

The isupper function tests for any upper-case letter or any of an implementation-defined set of characters for which none of iscntrl, isdigit, ispunct, or isspace is true. In the C locale, isupper returns true only for the characters defined as upper-case letters.

The isxdigit function

Synopsis:

```
#include <ctype.h>
int isxdigit(int c);
```

Description:

The isxdigit function tests for any hexadecimal-digit character.

The tolower function

Synopsis:

```
#include <ctype.h>
int tolower(int c);
```

Description:

The tolower function converts an upper-case letter to the corresponding lower-case letter.

Returns:

If the argument is an upper-case letter, the tolower function returns the corresponding lower-case letter if there is one; otherwise the argument is returned unchanged. In the C locale, tolower maps only the characters for which isupper is true to the corresponding characters for which islower is true.

The toupper function

Synopsis:

```
#include <ctype.h>
int toupper(int c);
```

Description:

The toupper function converts a lower-case letter to the corresponding upper-case letter.

Returns:

If the argument is a lower-case letter, the toupper function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged. In the C locale, toupper maps only the characters for which islower is true to the corresponding characters for which isupper is true.

4.3.2 math.h

The isfinite macro

Synopsis:

```
#include <math.h>
int isfinite(real-floating x);
```

Description:

The isfinite macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument. Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal long double value might become subnormal when converted to double, and zero when converted to float.

Returns:

The isfinite macro returns a nonzero value if and only if its argument has a finite value.

The isinf macro

Synopsis:

```
#include <math.h>
int isinf(real-floating x);
```

Description:

The isinf macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns:

The isinf macro returns a nonzero value if and only if its argument has an infinite value.

The isnan macro

Synopsis:

```
#include <math.h>
int isnan(real-floating x);
```

Description:

The isnan macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns:

The isnan macro returns a nonzero value if and only if its argument has a NaN value.

The isnormal macro

Synopsis:

```
#include <math.h>
int isnormal(real-floating x);
```

For the isnan macro, the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

Description:

The isnormal macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns:

The isnormal macro returns a nonzero value if and only if its argument has a normal value.

The signbit macro (not in C89)

Synopsis:

```
#include <math.h>
int signbit(real-floating x);
```

Description:

The signbit macro determines whether the sign of its argument value is negative.

Returns:

The signbit macro returns a nonzero value if and only if the sign of its argument value is negative.

The fabs function

Synopsis:

```
#include <math.h>
double fabs(double x);
```

Description:

The fabs function computes the absolute value of a floating-point number x.

Returns:

The fabs function returns the absolute value of x.

The modf function

Synopsis:

```
#include <math.h>
double modf(double value, double *iptr);
```

Description:

The modf function breaks the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by iptr.

Returns:

The modf function returns the signed fractional part of value.

The fmod function

Synopsis:

```
#include <math.h>
double fmod(double x, double y);
```

Description:

The fmod function computes the floating-point remainder of x/y.

Returns:

The fmod function returns the value x i y , for some integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the fmod function returns zero is implementation-defined.

The exp function

Synopsis:

```
#include <math.h>
double exp(double x);
```

Description:

The exp function computes the exponential function of x. A range error occurs if the magnitude of x is too large.

Returns:

The exp function returns the exponential value.

The sqrt function

Synopsis:

```
#include <math.h>
double sqrt(double x);
```

Description:

The sqrt function computes the nonnegative square root of x. A domain error occurs if the argument is negative.

Returns:

The sqrt function returns the value of the square root.

The pow function

Synopsis:

```
#include <math.h>
double pow(double x, double y);
```

Description:

The pow function computes x raised to the power y. A domain error occurs if x is negative and y is not an integer. A domain error occurs if the result cannot be represented when x is zero and y is less than or equal to zero. A range error may occur.

Returns:

The pow function returns the value of x raised to the power y.

The ldexp function

Synopsis:

```
#include <math.h>
double ldexp(double x, int exp);
```

Description:

The ldexp function multiplies a floating-point number by an integral power of 2. A range error may occur.

Returns:

The ldexp function returns the value of x times 2 raised to the power exp.

The frexp function

Synopsis:

```
#include <math.h>
double frexp(double value, int *exp);
```

Description:

The frexp function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the int object pointed to by exp.

Returns:

The frexp function returns the value x , such that x is a double with magnitude in the interval [1/2, 1) or zero, and value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.

The floor function

Synopsis:

```
#include <math.h>
double floor(double x);
```

Description:

The floor function computes the largest integral value not greater

than x.

Returns:

The floor function returns the largest integral value not greater than x , expressed as a double.

The ceil function

Synopsis:

```
#include <math.h>
double ceil(double x);
```

Description:

The ceil function computes the smallest integral value not less than x.

Returns:

The ceil function returns the smallest integral value not less than x , expressed as a double.

The cos function

Synopsis:

```
#include <math.h>
double cos(double x);
```

Description:

The cos function computes the cosine of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns:

The cos function returns the cosine value.

The sin function

Synopsis:

```
#include <math.h>
double sin(double x);
```

Description:

The sin function computes the sine of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns:

The sin function returns the sine value.

The tan function

Synopsis:

```
#include <math.h>
double tan(double x);
```

Description:

The tan function returns the tangent of x (measured in radians). A large magnitude argument may yield a result with little or no significance.

Returns:

The tan function returns the tangent value.

The atan function

Synopsis:

```
#include <math.h>
double atan(double x);
```

Description:

The atan function computes the principal value of the arc tangent of x.

Returns:

The atan function returns the arc tangent in the range [-PI/2, +PI/2] radians.

The atan2 function

Synopsis:

```
#include <math.h>
double atan2(double y, double x);
```

Description:

The atan2 function computes the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns:

The atan2 function returns the arc tangent of y/x , in the range [-PI, +PI] radians.

The asin function

Synopsis:

```
#include <math.h>
double asin(double x);
```

Description:

The asin function computes the principal value of the arc sine of x. A domain error occurs for arguments not in the range [-1, +1].

Returns:

The asin function returns the arc sine in the range [-PI/2, +PI/2] radians.

The acos function

Synopsis:

```
#include <math.h>
double acos(double x);
```

Description:

The acos function computes the principal value of the arc cosine of x. A domain error occurs for arguments not in the range [-1, +1].

Returns:

The acos function returns the arc cosine in the range [0, PI] radians.

The sinh function

Synopsis:

```
#include <math.h>
double sinh(double x);
```

Description:

The sinh function computes the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

Returns:

The sinh function returns the hyperbolic sine value.

The cosh function

Synopsis:

```
#include <math.h>
double cosh(double x);
```

Description:

The cosh function computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Returns:

The cosh function returns the hyperbolic cosine value.

The tanh function

Synopsis:

```
#include <math.h>
double tanh(double x);
```

Description:

The tanh function computes the hyperbolic tangent of x.

Returns:

The tanh function returns the hyperbolic tangent value.

The log function

Synopsis:

```
#include <math.h>
double log(double x);
```

Description:

The log function computes the natural logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero and the logarithm of zero cannot be represented.

Returns:

The log function returns the natural logarithm.

The log10 function

Synopsis:

```
#include <math.h>
double log10(double x);
```

Description:

The log10 function computes the base-ten logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero and the logarithm of zero cannot be represented.

Returns:

The log10 function returns the base-ten logarithm.

The log2 function (Not in C89 standard)

Synopsis:

```
#include <math.h>
double log2(double x);
```

Description:

The log2 function computes the base-two logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero and the logarithm of zero cannot be represented.

Returns:

The log2 function returns the base-two logarithm.

4.3.3 stdio.h

In contrast to the C standard, *fputc* and *fgetc* are built-in functions, you do not need to include *stdio.h* to use them.

The globals *stdin* and *stdout* should be set to an input or output by the user.

The *fputs* function prints *string* to the output *handle*.

```
void fputs(unsigned string[], unsigned handle);
```

The *fgets* function reads a line, up to *maxlength* characters, or a line end from the input *handle*. The string will be null terminated. *maxlength* includes the null character.

```
void fgets(unsigned string[], unsigned maxlength, unsigned handle);
```

The *puts* function prints *string* to stdout.

```
void puts(unsigned string[]);
```

The *gets* function reads a line, up to *maxlength* characters, or a line end from stdin. The string will be null terminated. *maxlength* includes the null character.

```
void gets(unsigned string[], unsigned maxlength);
```

The *getc* returns a single character from stdin.

```
unsigned long getc();
```

The *putc* writes a single character to stdout.

```
void putc(unsigned c);
```

4.3.4 <stdlib.h>

macros

The header <stdlib.h> defines the following macros:

- NULL
- RAND_MAX
- MB_CUR_MAX
- MB_LEN_MAX

Note: The EXIT_FAILURE and EXIT_SUCCESS macros are not defined.

RAND_MAX expands to an integral constant expression, the value of which is the maximum value returned by the rand function. *MB_CUR_MAX* expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE), and whose value is never greater than *MB_LEN_MAX*.

Note: The EXIT_FAILURE and EXIT_SUCCESS macros are not defined.

RAND_MAX expands to an integral constant expression, the value of which is the maximum value returned by the rand function.

MB_CUR_MAX expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE), and whose value is never greater than *MB_LEN_MAX*.

types

The header <stdlib.h> defines the following types:

- div_t

The atof function

Synopsis:

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description:

The atof function converts the initial portion of the string pointed to by nptr to double representation. Except for the behavior on error, it is equivalent to

strtod(nptr, (char **)NULL)

Returns:

The atof function returns the converted value.

Note: This function is not implemented!!!

The atoi function

Synopsis:

Description:

The atoi function converts the initial portion of the string

pointed to by nptr to int representation. Except for the behavior on error, it is equivalent to

(int)strtol(nptr, (char **)NULL, 10)

Returns:

The atoi function returns the converted value.

Note: This function is not implemented!!!

The atol function

Synopsis:

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Description:

The atol function converts the initial portion of the string

pointed to by nptr to long int representation. Except for the behavior on error, it is equivalent to

strtol(nptr, (char **)NULL, 10)

Returns:

The atol function returns the converted value.

Note: This function is not implemented!!!

The strtod function

Synopsis:

Note: This function is not implemented!!!

The strtol function

Synopsis:

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Note: This function is not implemented!!!

The strtoul function

Synopsis:

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr,
        int base);
```

Note: This function is not implemented!!!

The rand function

Synopsis:

```
#include <stdlib.h>
int rand(void);
```

Description:

The rand function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX.

The implementation shall behave as if no library function calls the rand function.

Returns:

The rand function returns a pseudo-random integer.

The srand function

Synopsis:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description:

The srand function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If rand is called before any calls to srand have been made, the same sequence shall be generated as when srand is first called with a seed value of 1.

Returns:

The srand function returns no value.

The malloc function

Synopsis:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description:

The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate.

Returns:

The malloc function returns either a null pointer or a pointer to the allocated space.

The calloc function

Synopsis:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description:

The calloc function allocates space for an array of nmemb objects, each of whose size is size. The space is initialized to all bits zero.

Returns:

The calloc function returns either a null pointer or a pointer to the allocated space.

The realloc function

Synopsis:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

The realloc function changes the size of the object pointed to by ptr to the size specified by size. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If ptr is a null pointer, the realloc function behaves like the malloc function for the specified size. Otherwise, if ptr does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to the free or realloc function, the behavior is undefined. If the space cannot be allocated, the object pointed to by ptr is unchanged. If size is zero and ptr is not a null pointer, the object it points to is freed.

Returns:

The realloc function returns either a null pointer or a pointer to the possibly moved allocated space.

The free function

Synopsis:

```
#include <stdlib.h>
void free(void *ptr);
```

Description:

The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to free or realloc, the behavior is undefined.

Returns:

The free function returns no value.

The abort function

Synopsis:

```
#include <stdlib.h>
void abort(void);
```

Note: this function is not implemented!!!

The atexit function

Synopsis:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Note: this function is not implemented!!!

The exit function

Synopsis:

```
#include <stdlib.h>
void exit(int status);
```

Note: This function is not implemented!!!

The getenv function

Synopsis:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Note: this function is not implemented!!!

The system function

Synopsis:

```
#include <stdlib.h>
int system(const char *string);
```

Note: This function is not implemented!!!

The bsearch function

Synopsis:

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
         size_t nmemb, size_t size,
         int (*compar)(const void *, const void *));
```

Note: This function is not implemented!!!

The qsort function

Synopsis:

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
          int (*compar)(const void *, const void *));
```

Note: This function is not implemented!!!

The abs function

Synopsis:

```
#include <stdlib.h>
int abs(int j);
```

Description:

The abs function computes the absolute value of an integer j. If the result cannot be represented, the behavior is undefined.

Returns:

The abs function returns the absolute value.

The div function

Synopsis:

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Description:

The div function computes the quotient and remainder of the division of the numerator numer by the denominator denom . If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, quot * denom + rem shall equal numer .

Returns:

The div function returns a structure of type div_t, comprising both the quotient and the remainder. The structure shall contain the following members, in either order.

```
int quot;   /*  quotient */
int rem;    /*  remainder */
```

The labs function

Synopsis:

..code-block:

```
#include <stdlib.h>
long int labs(long int j);
```

Description:

The labs function is similar to the abs function, except that the argument and the returned value each have type long int.

The ldiv function

Synopsis:

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description:

The ldiv function is similar to the div function, except that the arguments and the members of the returned structure (which has type ldiv_t) all have type long int.

The mblen function

Synopsis:

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Note: This function is not implemented!!!

The mbtowc function

Synopsis:

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Note: This function is not implemented!!!

The wctomb function

Synopsis:

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Note: This function is not implemented!!!

The mbstowcs function

Synopsis:

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Note: This function is not implemented!!!

The wcstombs function

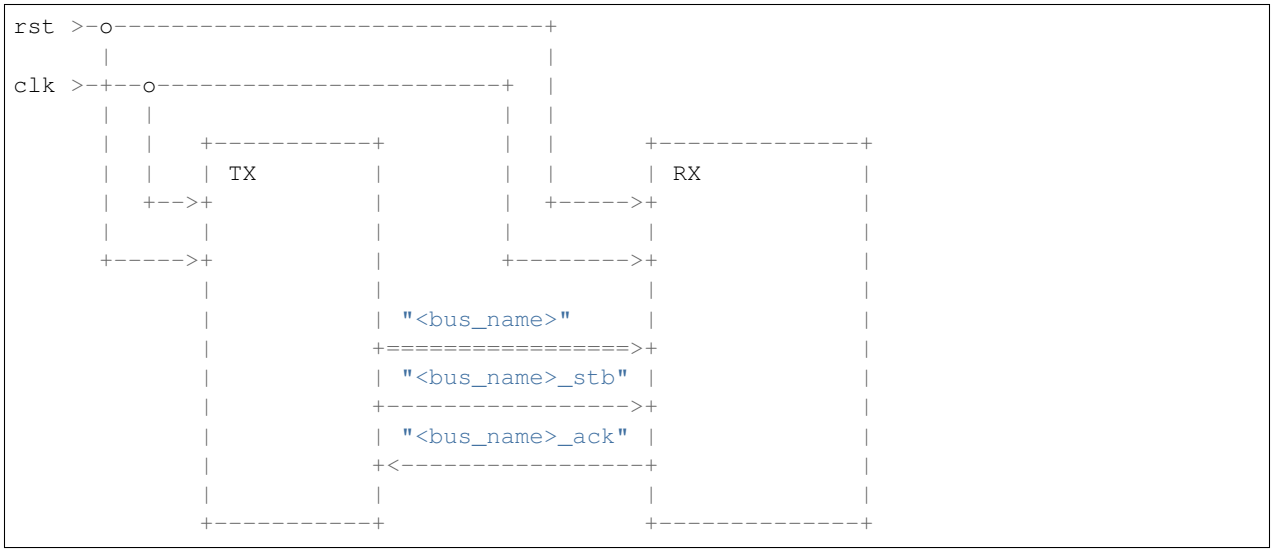
Synopsis:

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Note: This function is not implemented!!!

4.4 Physical Interface

Input, *Output* and *Wire* objects within a chip are implemented using a synchronous interconnect bus. The details of the interconnect bus are described here. This section will be of most use to developers who want to integrate a *Chips* design into a larger design, or to generate an HDL wrapper to support a *Chips* design in new hardware.



4.4.1 Global Signals

Name	Direction	Type	Description
clk	input	bit	Clock
rst	input	bit	Reset

4.4.2 Interconnect Signals

Name	Direction	Type	Description
<bus_name>	TX to RX	bus	Payload Data
<bus_name>_stb	TX to RX	bit	'1' indicates that payload data is valid and TX is ready.
<bus_name>_ack	TX to RX	bit	'1' indicates that RX is ready.

4.4.3 Interconnect Bus Transaction

1. Both transmitter and receiver **shall** be synchronised to the 0 to 1 transition of *clk*.
2. If *rst* is set to 1, upon the 0 to 1 transition of *clk* the transmitter **shall** terminate any active bus transaction and set <*bus_name*>*_stb* to 0.
3. If *rst* is set to 1, upon the 0 to 1 transition of *clk* the receiver **shall** terminate any active bus transaction and set <*bus_name*>*_ack* to 0.
4. If *rst* is set to 0, normal operation **shall** commence.
5. The transmitter **may** insert wait states on the bus by setting <*bus_name*>*_stb* to 0.
6. The transmitter **shall** set <*bus_name*>*_stb* to 1 to signify that data is valid.
7. Once <*bus_name*>*_stb* has been set to 1, it **shall** remain at 1 until the transaction completes.
8. The transmitter **shall** ensure that <*bus_name*> contains valid data for the entire period that <*bus_name*>*_stb* is 1.
9. The transmitter **may** set <*bus_name*> to any value when <*bus_name*>*_stb* is 0.
10. The receiver **may** insert wait states on the bus by setting <*bus_name*>*_ack* to 0.
11. The receiver **shall** set <*bus_name*>*_ack* to 1 to signify that it is ready to receive data.
12. Once <*bus_name*>*_ack* has been set to 1, it **shall** remain at 1 until the transaction completes.
13. Whenever <*bus_name*>*_stb* is 1 and <*bus_name*>*_ack* are 1, a bus transaction **shall** complete on the following 0 to 1 transition of *clk*.
14. Both the transmitter and receiver **may** commence a new transaction without inserting any wait states.
15. The receiver **may** delay a transaction by inserting wait states until the transmitter indicates that data is available.
16. The transmitter **shall** not delay a transaction by inserting wait states until the receiver is ready to accept data. Deadlock would occur if both the transmitter and receiver delayed a transaction until the other was ready.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index