Sintel

Release 0.1

Apr 13, 2020

Software

Sintel is a simulator and modeler of basic digital systems that allows, in addition to visualizing inputs and outputs in real time, to export the model to a hardware description language such as VHDL and verilog.

Sintel is written in python with the QT graphics library, it's an open source project with GPLv3 license or later.

In this manual we hope to give you a guide to the use and operation of the different parts of the software, as well as to invite you to contribute with various elements within it such as the creation of components.

You can find the master code in this link <https://gitlab.com/WilferCiro/sintel>

CHAPTER 1

Installation

1.1 Dependency Icarus Verilog

For windows, download the .exe file in https://iverilog.fandom.com/wiki/Main_Page and follow steps and install, restart your computer and follow the nexts steps

For Ubuntu:

```
sudo apt install iverilog
```

For Arch:

```
sudo pacman -S iverilog
```

1.2 Binary

For install in Windows, download the https://gitlab.com/WilferCiro/sintel/-/raw/master/instals/Sintel.exe and open the Sintel.exe file and continue with the steps

For install in Linux

```
$ wget https://gitlab.com/WilferCiro/sintel/-/raw/master/instals/LinuxInstaller.tar.gz
$ tar xf LinuxInstaller.tar.gz
$ cd LinuxInstaller
$ sh install.sh
```

1.3 For developers

for install in Ubuntu

```
sudo apt install python3-pip qttools5-dev-tools pyqt5-dev-tools iverilog python3-
↪matplotlib
```

for install in arch

```
sudo pacman -S python3-pip pyqt5 iverilog python3-matplotlib
```

Next, exec on all platforms

```
sudo pip3 install shutil sqlite3
```

1.4 Execution

For exec Sintel, cd into Sintel's folder and put in your terminal.

```
git clone https://gitlab.com/WilferCiro/sintel.git
cd sintel/
python3 application.py
```

CHAPTER 2

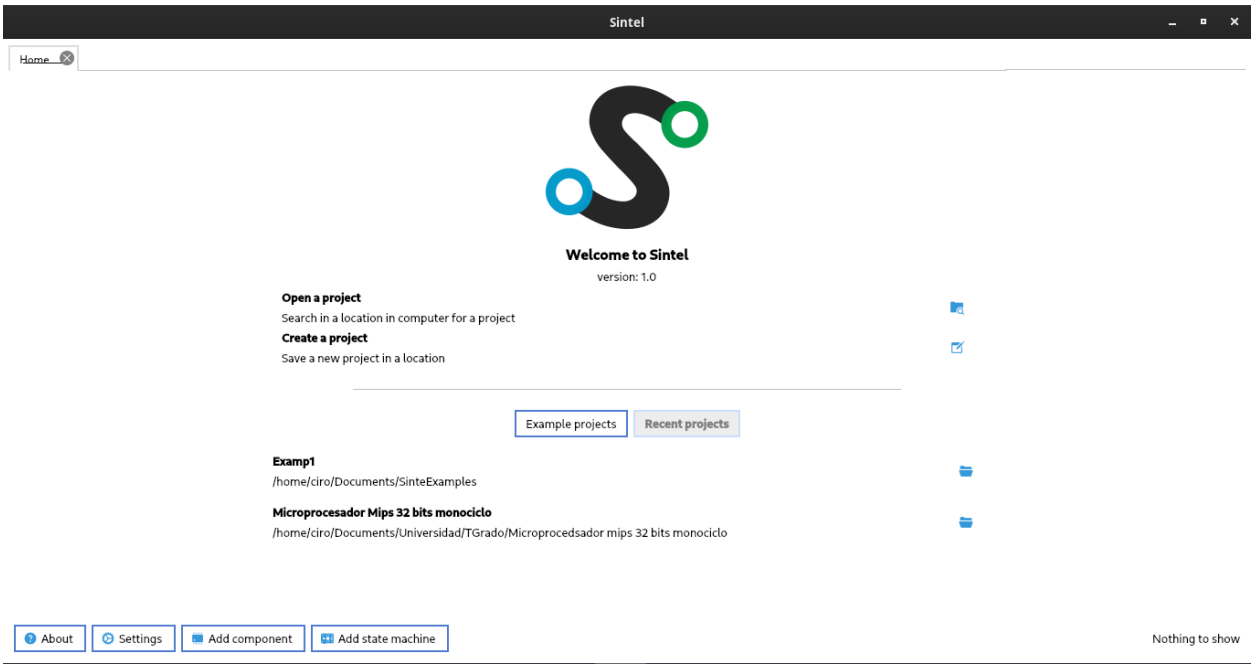
Visual Interface

It has several interfaces throughout the software, among them are

- Home view
- Simulation view
- Settings view
- Add component view
- Add state machine view

2.1 Home view

When you start Sintel, the first view is the home view, this section is very basic, where you can create and open projects, there is a list of 3 recent opened projects ordered by accessed date.



2.1.1 Open and load project

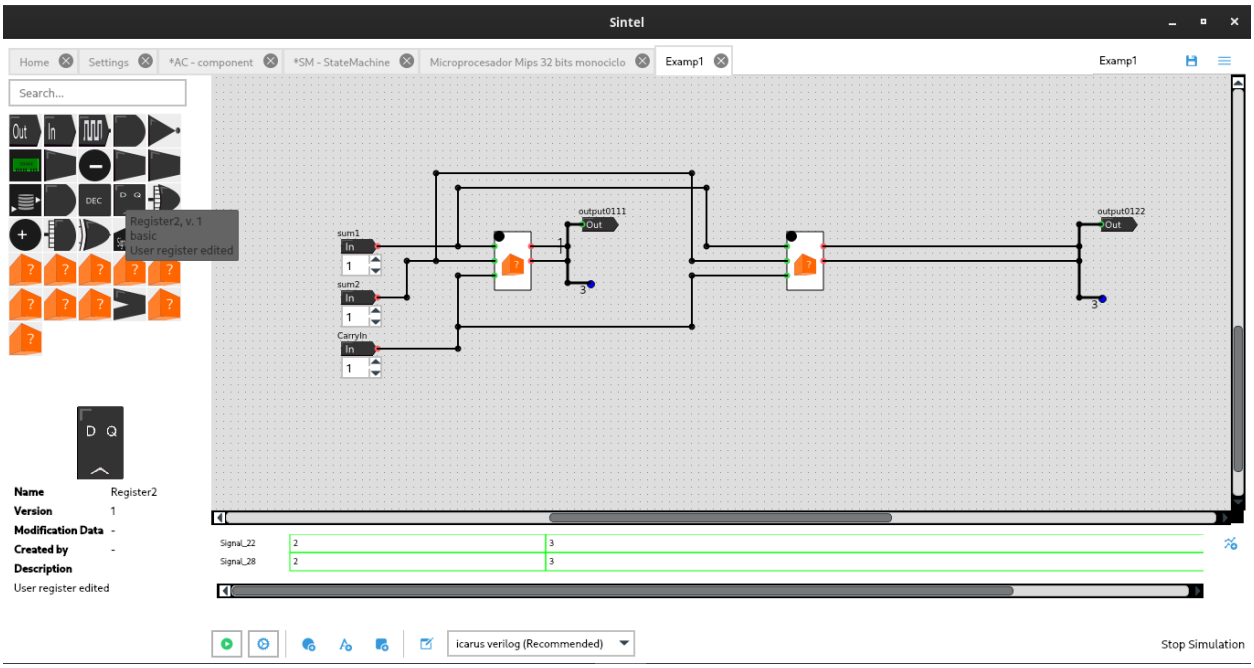
The projects extension is .sintel, you can open projects with this extension.

2.2 Simulation view

The simulation view is the main view in the software, since it's the place where you can join components for simulate your digital system.

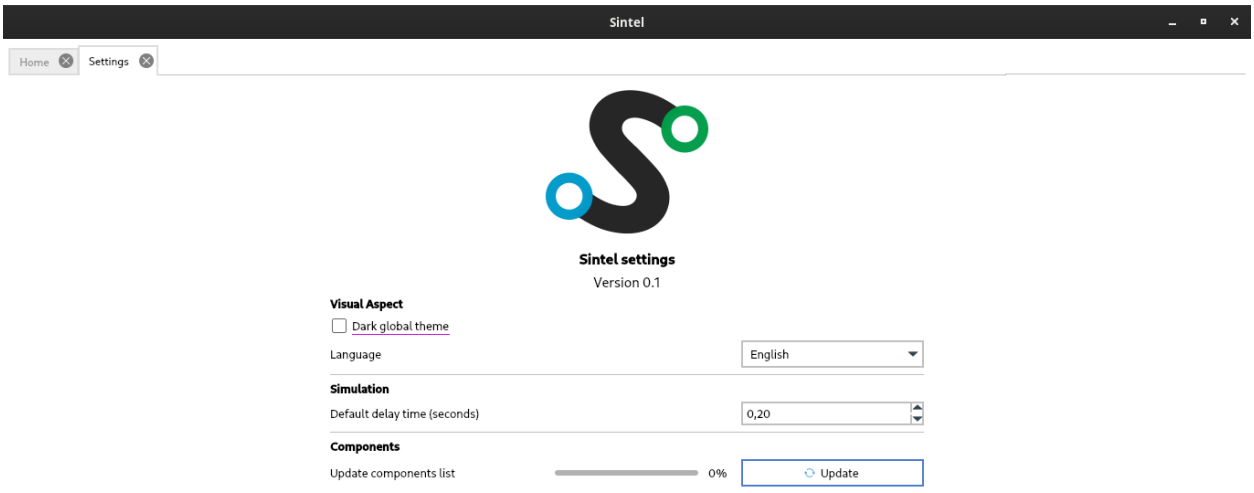
This section is displayed when you create or open a project. This view is divided in several parts like

- **Left side:** there is the components list, you can drag and drop them into the workspace
- **Center side:** there is the workspace, there you can join the components and model the digital system.
- **Bottom side:** there are several buttons, these buttons serve to run the simulation, and configure aspects of this.
- **Top right side:** there are buttons for save and open projects, and another button for open the simulation view menu.
- **Simulation view menu:** this menu has several options that will be described below.



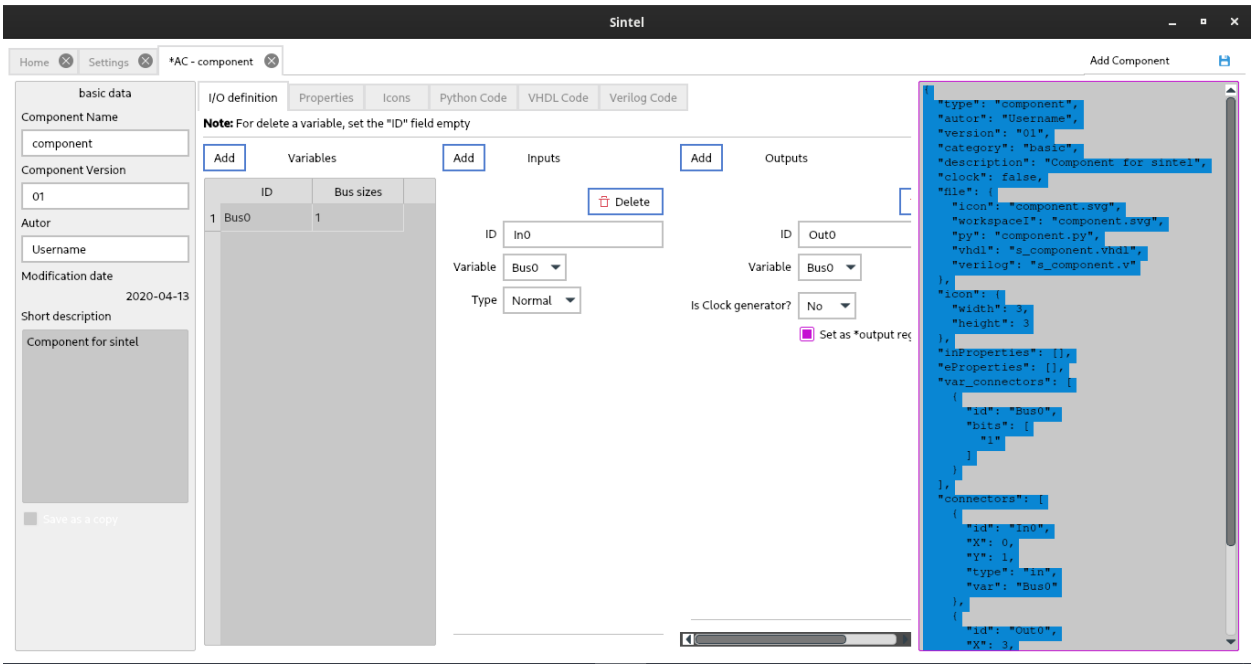
2.3 Settings view

This section is designed for select characteristics important for the experience in the software.



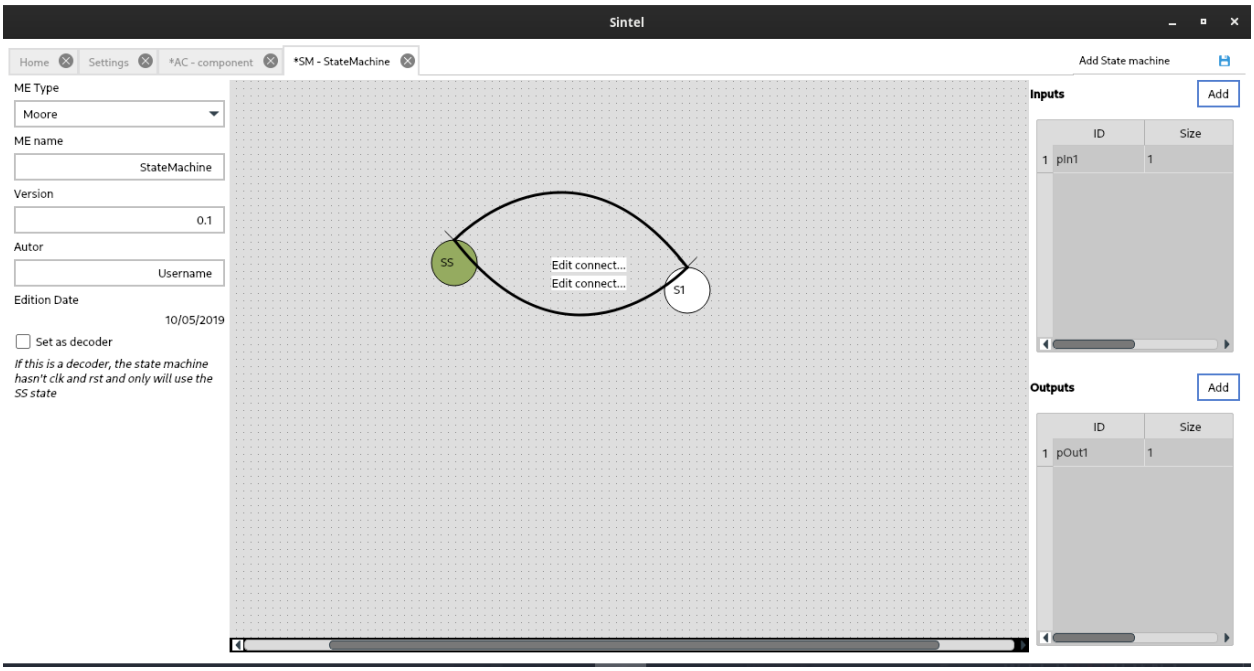
2.4 Add component view

This section is designed for give the possibilities of add new components from zero, adding i/o, code and icons.



2.5 Add state machine view

In this view, you can add and edit a state machine graphically, for more information go to “Add state machine” section



CHAPTER 3

Export the model to HDL

You can export your simulation model to

- VHDL
- Verilog

For export the model you have to press the "Export to HDL" button ubicated in the simulation view menu, this action creates two folders in the project path with the vhdl and verilog files.

The files in the folders are:

- vhdl and verilog component files.
- verilog/main.v and vhdl/main.vhdl, this file contains the connections of all components made visually, and is the top level entity.
- verilog/testbench.v and vhdl/testbench.vhdl, this file contains a simple workbench for simulate the model, you can edit this file to create more complex simulations.
- verilog/Makefile and vhdl/Makefile, this file contains the commands to run the testbench in gtkwave.

3.1 (Optional) Install free simulation tools (Linux)

3.1.1 GTKWave

For simulate the generated HDL model, you can use IDEs like Quartus with the modelsim, and we recommend the GTKWave (http://gtkwave.sourceforge.net/) application, this applicatión is free and can be installed in Windows, Mac OSX and Linux.

For install in linux, it depends of your distribution, for example, for ubuntu you can install it with:

```
sudo apt install gtkwave
```

for arch

```
sudo pacman -S gtkwave
```

3.1.2 GHDL

ghdl is a open source simulator for VHDL language (more info in http://ghdl.free.fr/), for install it you can execute

In Ubuntu

```
sudo apt install ghdl
```

In arch

```
yaourt -S ghdl-llvm-git
```

3.1.3 iverilog

icarus verilog is a open source verilog compiler (mode info in https://iverilog.fandom.com/wiki/Main_Page), for install it you can execute

In Ubuntu

```
sudo apt install iverilog
```

In arch

```
sudo pacman -S iverilog
```

3.2 (Optional) simulate the vhdl model in gtkwave with ghdl

For simulate the generated code, there is a Makefile in the vhdl directory of your project, this file contains the compiler and run commands

```
ghdl -a *.vhdl
ghdl -e testbench
ghdl -r testbench --vcd=testbench.vcd
gtkwave testbench.vcd
```

When these commands are finished, you will have gtkwave open with a default simulation created in the testbench.vhdl, you can edit this file and improve the simulation at your convenience.

3.3 (Optional) simulate the verilog model in gtkwave with iverilog

For simulate the generated code, there is a Makefile in the verilog directory of your project, this file contains the compiler and run commands

```
iverilog -o simulation *.v
vvp simulation
gtkwave test.vcd &
```

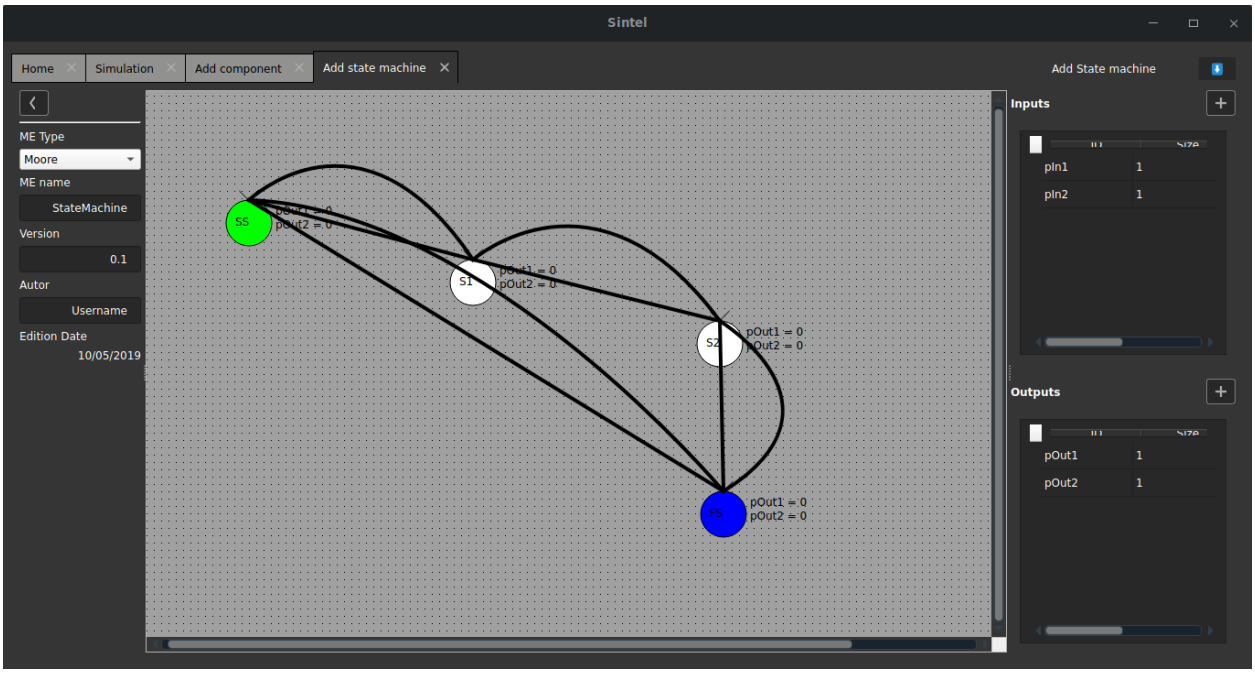
When these commands are finished, you will have gtkwave open with a default simulation created in the testbench.v, you can edit this file and improve the simulation at your convenience.

CHAPTER 4

State machine (sm)

For create a state machine, press the “Add state machine” button in the simulation view menu, there a new tab will be created with the creation interface, this interface is divided in

- Left side: basic data and sm type
- Center: workspace for add and link states
- Right side: for add inputs and outputs



4.1 basic data

This is the information data of the state machine, the edition date is automatically updated.

Warning: Be careful with the name of the state machine, since it defines the files to be saved, if there is already a state machine with the name you enter, it will be replaced by the new machine.

4.2 Add I/O

For add an input or output, press the "Add" button located in the respective input or output section, two fields are automatically added to the table, these are

- ID: is the entry or exit identifier, must be unique
- Size: You can define a connector bus size

For delete a field, you must leave the ID field blank and press enter key.

4.3 Workspace

In this section, you can add states and join them for create a visual state machine. When you add a new state machine, there are two default states, the green state "SS" is the start state, and the blue state "FS" is the final state.

4.3.1 Add state

For add an state, you must press double click in a blank space of the workspace, there a circle identified with the state ID is created, which automatically increases

4.3.2 Delete state

For delete an state, you must select the state and press the Supr key.

4.3.3 Edit output in state

For add outputs to an state, the state machine must be moore type. Double click to state and a modal window will be displayed, there is a table, then you define the output value for all the outputs, this value is limited by

$$2^{Size} - 1$$

where Size is the bus size defined in the output addition

4.3.4 Join states

For join states, you must click on the source state and then click on the destination state, you will see that a line is created that joins them with an arrow identifying the origin and destination

4.3.5 Delete join

For delete a join, click in line and press supr key.

4.3.6 Edit conditions

The conditions are edited in the lines that connect the states, for this, double click in these lines, next a modal window is shown with two tabs

- Conditions tab: you can add the list of conditions, add a line with the button and modify them, if you have more open parentheses than closed ones, the program will automatically close them at the end
- Outputs tab: This section is enabled if the state machine is mealy type and its operation is the same as adding the outputs in the states.

4.4 Edit state machine

To edit a state machine, you must add a component of this state machine type to the simulation workspace, double click on the component and press the edit button located in the upper right part of the properties window.

When you add, edit or delete an input/output of a state machine, we recommend that you close Sintel and open for take the state machine changes.

CHAPTER 5

Add and edit components

There are two ways to add a new component

- Joining components for create “macro components”.
- Adding all from scratch, IO definitions, python code, vhd code, verilog code, svg icon.

5.1 Joining components

You can create components joining another components, we call this union as “maco component”, you can create macro components joining components with components and macro components.

For add a macro component, you must select the components that are part of the macro component and press in the bottom site the button “Add macro component”, and this will create a white box that contains the components inside, you can expand or contract this box clicking the “+” icon or the “-” icon.

This box initially not have inputs and outputs, you can add them double click the white box, and a modal window will appear, there are two tables, you can add the inputs and outputs separated, define the ID, bus size and if is input, a default value (if this no has default value, put “-1” in the field), for delete a I/O, leave blank the ID field.

You can save this macro component, going to second tab of the macro component properties, put a name and press the save button, now this schematic circuit will save in your directory.

If you select a macro component and press “supr” key, the macro is removed but the internal components are conserved, for delete all complete, just right click and press delete all.

The macro components schematic is saved in JSON format in the SintelDir/components_macros, yo can share this file to another people.

5.2 Adding components from scratch

This action is realized in the “add components view”, for show this section you must press the “Add component” button in the menu of simulation view.

This view has 3 main sections

- **Left panel:** there is a form with the basic data of the component.
- **Right panel:** there is a view of the configuration component file, and a resume of the entered data in JSON format.
- **Center:** there are tabs for add the component data

5.2.1 Left Panel:

> **Warning:** Take care, if there is a component with the name that you putted, the old component will be replaced.

> **Danger:** Don't use a reserved python, vhdl or verilog word for name the component.

> **Warning:** There are two fields that are for define if the component is an input or output of all the system, for example a clock generator, don't select them if your component isn't a general IO of the system, since if you select it, the component can consume more computing resources and will not work in the best way.

5.2.2 Tabs of data

I/O Definition

- Variables are created for change define the bus size, you can assign them to Inputs and outputs, and when the user change the size of a connector, all connectors that are associated to this variable will change, this is useful when you have outputs that depends of inputs for example a OR gate, the output or 8 bits is realized by 2 inputs of 8 bits.
- Inputs, define the input ID, the variable for change the connector size in the simulation, and if this is a normal input, a clock input (If you will connect a clock to this connector, for example the register clock input), or a reset input (If you will connect a reset signal, for example the reset input of a state machine).
- Outputs, define the output ID, the variable for change the connector size in the simulation, and if this is a clock generator, select "Yes" in "is clock generator" field.

Properties

You can add two types of properties, checking the "Show Inline?" value

- Inline: are visible in the workspace under the component, use them only for characteristics that can be changed in execution time, it should be noted that the processor usage increments when a property is changed in execution time, this can be used for change system inputs in real time. Those properties won't be exported to HDL.
- Hide: these properties are shown when you double click on a component in workspace, those properties will be exported to HDL.

Regardless of the type of property, you can select different types of user interaction, for example a number input, or a list. These types of input have their own fields for fill.

- **Select list:** add the list of values separated by colon, if you select the "Is Binary?" option, the values that you can add are "1" and "0".

- **Number:** you can add a default value, the min value and the max value, if you prefer, you can limit the maximum value to a variable size, then, when the user changes the bus size, this maximum value will change.
- **Bit:** Only allows select "1" or "0".

For delete a property, press the "Delete button".

Icons

There is a workspace where you can drag and drop the Inputs or Outputs added previous, select the SVG icon clocking the button, and you cand adjust the size with the "range inputs".

Note: When you create the svg file, try to take the measurements respect "25px", for example: Width (50px) height (75px).

Python code

The basic structure for the python file of a component is

```
#!/usr/bin/env python
from common import Component

class component(Component):
        def setup(self, properties):
                {{IO Declarations}}
                self.userSetup(properties)

        def userSetup(self, properties):
                {{your setup code}}

        def update(self):
                {{ your update code }}
```

- "IO Declarations", these lines are added automatically, and defines the inputs and outputs of the component.
- "your update code", you must add the update code, this code is executed when a Signal writes a value in this connector
- "setup user code" is executed in the constructor, in this method you can assign the properties, these properties are sent on a dictionary like (this example applies if you add two properties called "reset" and "myProperty")

```
properties = {"reset_pos" : "0", "myProperty" : "100"}
self._resetPos = int(properties["reset_pos"])
```

Next you will see two examples, one of an OR, and another of a register

```
1 #!/usr/bin/env python
2 # OR example by sintel
3 from common import Component
4
5 class OR1(Component):
6         def setup(self, properties):
7                 self._addInput("pIn1")
8                 self._addInput("pIn2")
9                 self._addOutput("pOut")
```

(continues on next page)

(continued from previous page)

```
                self.userSetup(properties)

        def userSetup(self, properties):
                pass

        def update(self):
                # Read the input values
                value1 = self.read("pIn1", base = 2)
                value2 = self.read("pIn2", base = 2)

                # Checks if the size are equal
                if len(value1) == len(value2):

                        # Realize the OR operation
                        write = ['1' if value1[i] == '1' or value2[i] == '1' else '0' for i in range(len(value1))]

                        # Write the result
                        self.write("pOut", write, base = 2)
                else:
                        # Writes an unknown value
                        self.write("pOut", 'U')
```

```
#!/usr/bin/env python
# Register example by sintel
from common import Component

class Register2(Component):
        def setup(self, properties):
                self._addInput("inD")
                self._addInput("ena")
                self._addInput("rst")
                self._addInput("clk")
                self._addOutput("outQ")
                self.userSetup(properties)

        def userSetup(self, properties):
                self._resetPos = int(properties["reset"])
                self._firstTime = True
                self._lastValue = None

        def update(self):
                # Read the inputs
                resetVal = self.read("rst")
                ena = self.read("ena")

                if resetVal == self._resetPos:

                        # As this component updates with clock, this line prevents that the component writes the same value and reduce the processor usage.
                        if self._lastValue != 0:

                                # Writes 0 to output
                                self.write("outQ", 0)
                                self._lastValue = 0

```

(continues on next page)

(continued from previous page)

```
                elif ena == 1:
                        value = self.read("inD")

                        # As this component updates with clock, this line prevents that the component writes the same value and reduce the processor usage.
                        if self._lastValue != value:

                                # Writes the input value to output
                                self.write("outQ", value)
                                self._lastValue = value
```

The methods that you can use in the python code are:

a = self.read("pInput", base = 10) Reads the input value, this value is in 10 base by default, if you want read in another base, just pass the "base" parameter with the base that you want, if this is base=2, the returned value is a list with strings of "1" or "0", for example ["0", "0", "1", "0"] (this is the number 2).

self.write("pOutput", value, base = 10) Writes a value in the signal connected to this output, base 10 is by default, if you want to write a value in another base, pass the "base" parameter, for base 2, you have to write a list with strings like ["1", "0", "1", "0"] (this is the number 10)

VHDL code

If the component has clock, you have to add two different codes, one for falling edge and another for rising edge, Sintel assign the property of select the clock flank for work, and you must have this present in the vhdl code.

The inputs and generics will be created by default, and you must add the architecture code.

There are two examples for two components in VHDL, an OR gate and a register.

```
--- OR gate example for sintel

library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity OR01 is
        generic(
                Bus_LEN:natural := 1
        );
        port(
                pIn1: in std_logic_vector(Bus_LEN - 1 downto 0);
                pIn2: in std_logic_vector(Bus_LEN - 1 downto 0);
                pOut: out std_logic_vector(Bus_LEN - 1 downto 0)
        );
end entity;
architecture rtl of OR01 is
begin
-- Put the code here
        pOut <= pIn1 or pIn2;
end architecture;
```

```
--- Register example for sintel

library ieee;
```

(continues on next page)

(continued from previous page)

```
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity Register2 is
        generics(
                buses_LEN:natural := 1;
                reset_PAR:std_logic := '0'
        );
        port(
                inD: in std_logic_vector(buses_LEN - 1 downto 0);
                ena: in std_logic;
                rst: in std_logic;
                clk: in std_logic;
                outQ: out std_logic_vector(buses_LEN - 1 downto 0)
        );
end entity;

--- This code is executed if the component is configured with the rising edge flank
architecture rising_edge_arch of Register2 is
begin
        outQ <= (others => '0') when rst = reset_PAR else inD when falling_Edge(clk)
 ↪and ena='1';
end architecture;

--- This code is executed if the component is configured with the falling edge flank
architecture falling_edge_arch of Register2 is
begin
        outQ <= (others => '0') when rst = reset_PAR else inD when rising_Edge(clk)
 ↪and ena='1';
end architecture;
```

As you can see in the register example, there is two architectures that are executed separated depending the component configuration made by user.

Verilog

The verilog code addition is equal to vhdl edition, please read the prev item for understand all steps, next, there are two examples of an OR gate and a Register in verilog.

CHAPTER 6

About the project

Sintel is a simulator and modeler of basic digital systems that allows, in addition to visualizing inputs and outputs in real time, to export the circuit to a hardware description language such as VHDL and verilog.

Sintel is written in python with the QT graphics library, it is an open source project with GPLv3 license or later.