# Stratus User's Guide

## Contents

# Introduction

## Stratus

**Name** Stratus – Procedural design language based upon *Python*

**Description** *Stratus* is a set of *Python* methods/functions dedicated to procedural generation purposes. From a user point of view, *Stratus* is a circuit's description language that allows *Python* programming flow control, variable use, and specialized functions in order to handle vlsi objects.

Based upon the *Hurricane* data structures, the *Stratus* language gives the user the ability to describe netlist and layout views.

**Configuration** A configuration file can be used to direct the generation process of Stratus. With this file, the user can choose the output format (vst, vhdl...), the simulator (asimut, ghdl...), the standard cell library... This configuration file named `.st_config.py` must be placed either in the HOME directory or in the current directory. This file contains a set of variables used in the process generation of Stratus, as for example :

```
format = 'vhdl'
simulator = 'ghdl'
```

The default configuration of Stratus uses the Alliance CAD system, ie `'vst'` as `format` and `'asimut'` as `simulator`.

**Description of a cell** A cell is a hierachical structural description of a circuit in terms of ports (I/Os), signals (nets) and instances.

The description of a cell is done by creating a new class, derivating for class `Model`, with different methods :

- Method `Interface` : Description of the external ports of the cell :
  - SignalIn, SignalOut, ...
- Method `Netlist` : Description of the netlist of the cell :
  - Inst, Signal
- Method `Layout` : Description of the layout of the cell :
  - Place, PlaceTop, PlaceBottom, PlaceRight, PlaceLeft ...
- Method `Stimuli` : Description of the simulation stimuli of the cell :
  - affect, addd ...

**Creation of the cell** After the description of a cell as a sub-class of `Model`, the cell has to be instantiated. The different methods described before have to be called.

Then different methods are provided :

- Method `View` : Opens/Refreshes the editor in order to see the created layout
- Method `Save` : Saves the created cell in the desired format thanks to the configuration file
  - no argument : creation of a netlist file
  - PHYSICAL : creation of a netlist file AND a layout file
  - STRATUS : creation of a python/stratus file
    - FileName : optionnal argument when using Save(STRATUS) in order to choose the name of the file to be generated

    - Be careful : if one wants to create a stratus file AND a netlist, always use Save(STRATUS) before Save() !

- Method `Testbench` : Creates the testbench of the cell using the `Stimuli` method to compute the stimuli. The output format depends of the `format` variable given in the configuration file
- Method `Simul` : Runs the simulation using the simulator named in the configuration file

**Syntax** A *Stratus* file must have a .py extension and must begin as follow :

```
#!/usr/bin/env python

from stratus import *
```

The description of a cell as a sub-class of `Model` is done as follow :

```
class myClass ( Model ) :
    ...
```

The creation of the cell is done by instantiating the previous class as follow :

```
exemple = myClass ( name, param )
```

After the different methods can be called as follow :

```
exemple.Interface()
exemple.Netlist()
exemple.Save()
    ...
```

In order to execute a *Stratus* file (named `file` for example), one has two choices :

```
python file.py
```

Or :

```
chmod u+x file.py
./file.py
```

The names used in *Stratus*, as arguments to *Stratus* functions, should be alphanumerical, including the underscore. The arguments of *Stratus* are case sensitive, so VDD is not equivalent to vdd.

Vectorized connectors or signal can be used using the [n:m] construct.

**Syntax highlighting** When using vi, it's possible to have the right syntax highlighting :

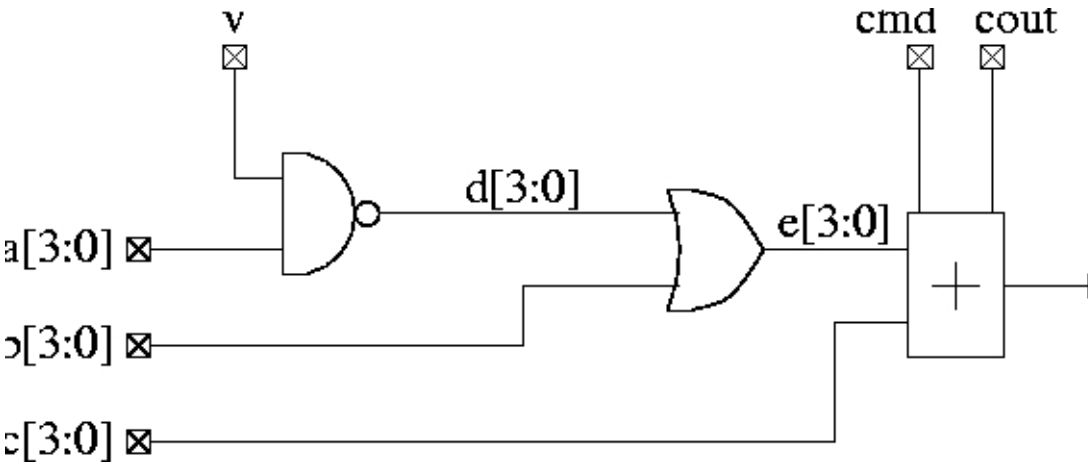
- Commands to do when you want to change once the coloration of your file :

```
:syntax off
:source /asim/coriolis/share/etc/stratus.vim
```

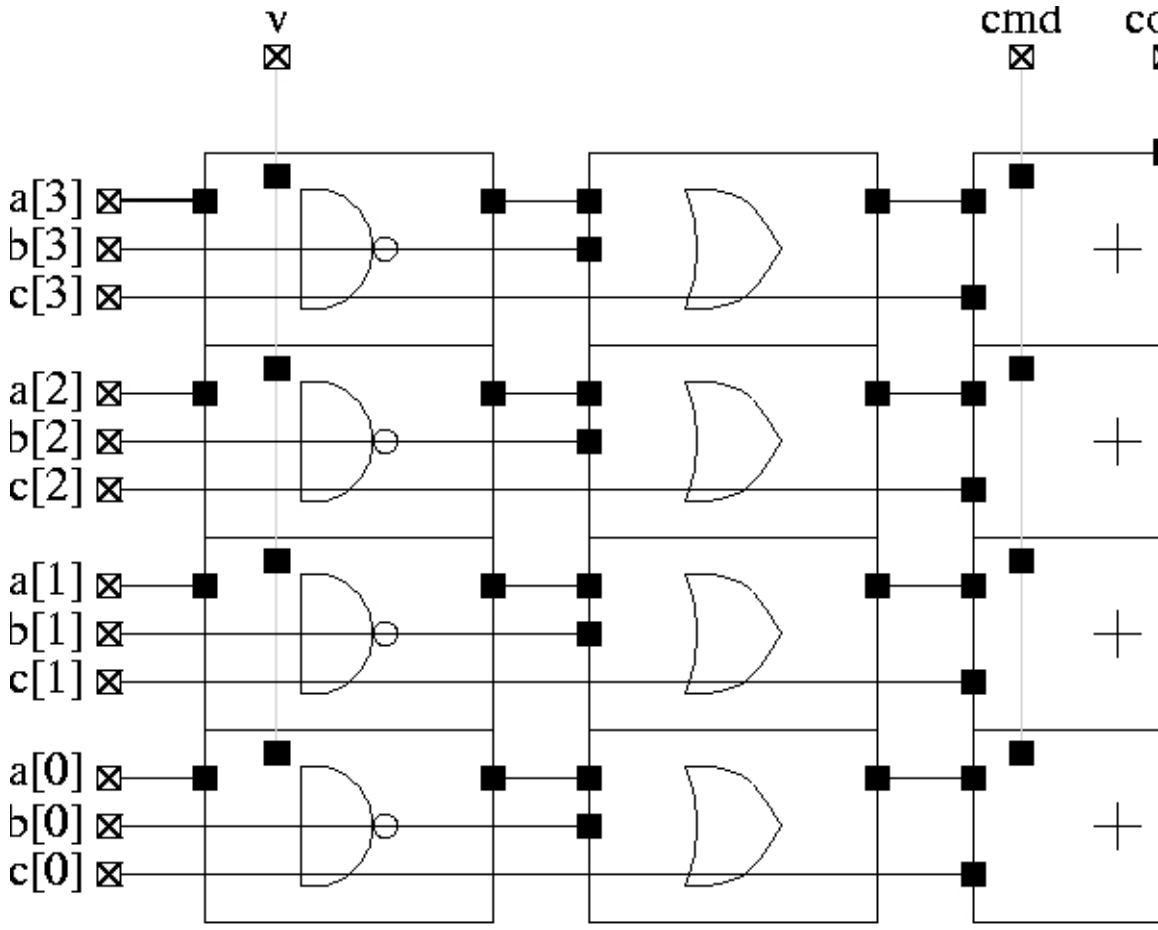
- Modification of your .vimrc in order to have the syntax highlighting each time you open a file :

```
syntax off
autocmd BufRead,BufNewfile *.py so /asim/coriolis/share/etc/stratus.vim
syntax on
```

**Example**




**The addaccu circuit**




**The data-path**

```
#!/usr/bin/env python

from stratus import *

class addaccu ( Model ) :

  def Interface ( self ) :
    self.nbit = self._param

    self.a    = LogicIn  (
    self.b    = LogicIn  (
    self.c    = LogicIn  (
    self.v    = LogicIn  (
    self.cmd  = LogicIn  (

    self.cout = LogicOut (
    self.s    = LogicOut (

    self.vdd  = VddIn    (
    self.vss  = VssIn    (

  def Netlist ( self ) :
    d_aux = Signal ( "d_aux
    e_aux = Signal ( "e_aux
    ovr   = Signal ( "ovr"

    self.instNand2 = Inst (
                           '
                           '
                           )
    self.instOr2   = Inst (
                           '
                           '
                           )
```

**Creation of the circuit : file test.py**

```
#!/usr/bin/env python

from stratus import *
from addaccu import addaccu

nbit = Param ( "n" )

dict = { 'nbit' : nbit }

inst_addaccu = addaccu ( "inst_addaccu", dict )

inst_addaccu.Interface()
inst_addaccu.Netlist()

inst_addaccu.Layout()

inst_addaccu.View()
inst_addaccu.Save()
```

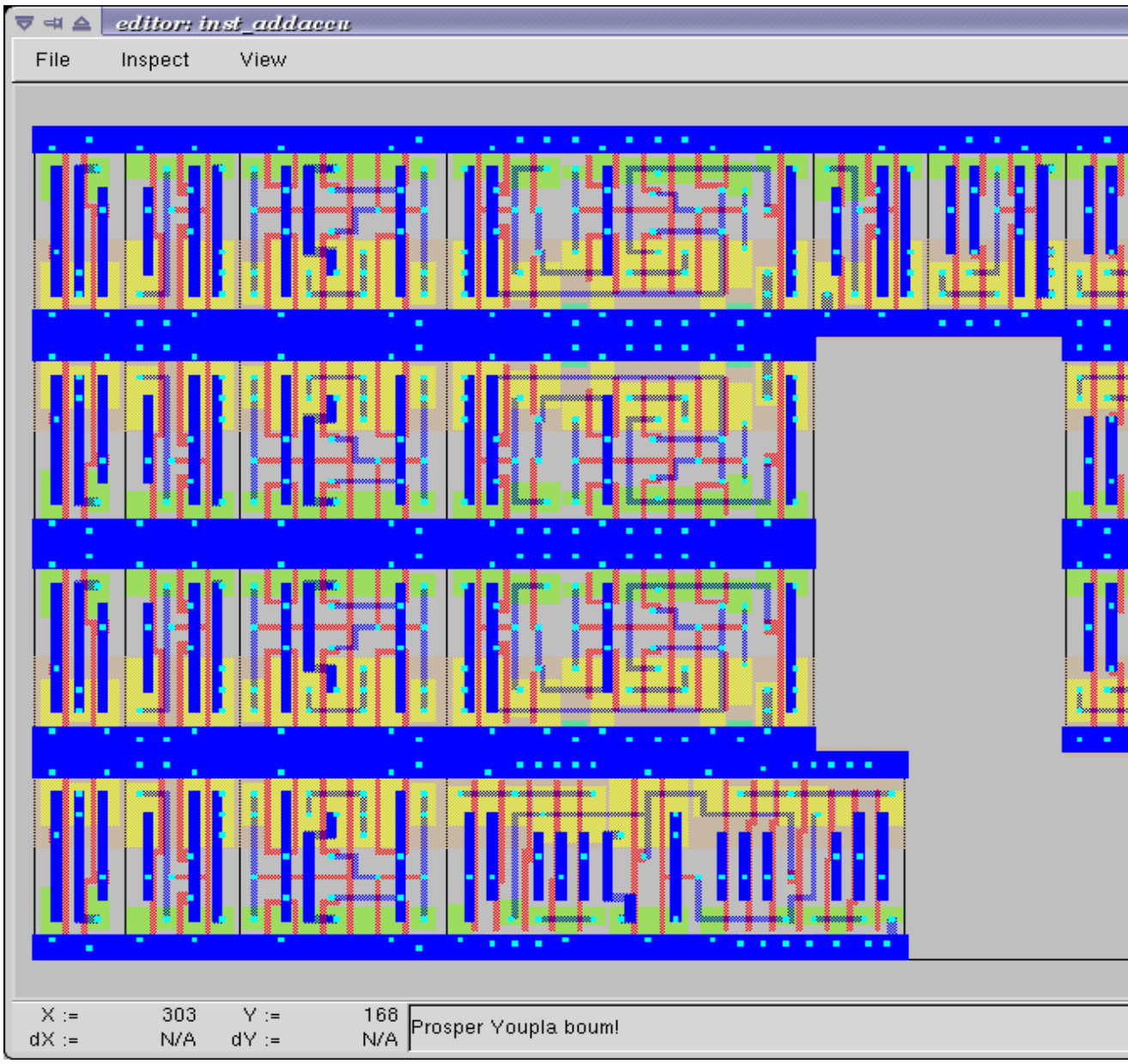
**How to execute the file**

```
python test.py -n 4
```

or :

```
chmod u+x test.py
./test -n 4
```

**The editor** The method `View` permits to open an editor in which one can see the cell being created as shown in the picture below.




**Function Param** This function allows the user to give parameters when creating a cell. When one wants to give values to two parameters, one can type on the shell :

```
python test.py -n 4 -w 8
```

The file `test.py` has then to contain :

```
nbit, nword = Param ( "n", "w" )
```

The letters typed on the shell must be the ones given as parameters of function `Param`.

**How to instanciate your generator in another generator** One can create a generator and instantiate it in another generator. To do that, the model name of the generator must have the form : "file_name.class_name". Note that if the two generators are not in the same directory, the directory of the generator to be instantiated has to be added in the CRL_CATA_LIB environment variable.

For example, in order to instanciate the addaccu created above in a cell :

```
n = 4
Generate ( "addaccu.addaccu", "my_addaccu_%dbits" % n
         , param = { 'nbit' : n } )

Inst ( "my_addaccu_%dbits" % n
     , map = { 'a'    : self.netA
             , 'b'    : self.netB
             , 'c'    : self.netC
             , 'v'    : self.netV
             , 'cmd'  : self.netCmd
             , 'cout' : self.netCout
             , 's'    : self.netS
             , 'vdd'  : self.vdd
             , 'vss'  : self.vss
             }
     )
```

## Description of a netlist

### Nets

**Name** SignalIn, SignalOut ... – Creation of nets

### Synopsys

```
netA = SignalIn ( "a", 4 )
```

**Description** How to create and use nets.

**Nets** Differents kind of nets are listed below :

- `SignalIn` : Creation of an input port
- `SignalOut` : Creation of an output port
- `SignalInOut` : Creation of an inout port
- `SignalUnknown` : Creation of an input/output port which direction is not defined
- `TriState` : Creation of a tristate port
- `CkIn` : Creation of a clock port
- `VddIn` : Creation of the vdd alimentation
- `VssIn` : Creation of the vss alimentation
- `Signal` : Creation of an internal net

**Parameters** All kind of constructors have the same parameters :

- `name` : the name of the net (mandatory argument)
- `arity` : the arity of the net (mandatory argument)
- `indice` : for bit vectors only : the LSB bit (optional argument : set to 0 by default)

Only `CkIn`, `VddIn` and `VssIn` do not have the same parameters : there is only the `name` parameter (they are 1 bit nets).

**Functions and methods** Some functions/methods are provided in order to handle nets :

- function `Cat` : Concatenation of nets, beginning with the MSB

```
Inst ( 'DpgenInv'
     , map = { 'i0'  : Cat ( A, B )
             , 'nq'  : S
             , 'vdd' : vdd
             , 'vss' : vss
             }
     )
```

  Or :

```
tab = []
tab.append ( A )
tab.append ( B )

Inst ( 'DpgenInv'
     , map = { 'i0'  : Cat ( tab )
             , 'nq'  : S
             , 'vdd' : vdd
             , 'vss' : vss
             }
     )
```

  If A and B are 2 bits nets, the net `myNet` will be such as :

```
myNet[3] = A[1]
myNet[2] = A[0]
myNet[1] = B[1]
myNet[0] = B[0]
```

- function `Extend` : Creation of a net which is an extension of the net which it is applied to

```
temp    = Signal (     "temp", 5 )
tempExt = Signal ( "temp_ext", 8 )

tempExt <= temp.Extand ( 8, 'one' )
```

- method `Alias` : Creation of an alias name for a net

```
cin.Alias  ( c_temp[0] )
cout.Alias ( c_temp[4] )
for i in range ( 4 ) :
  Inst ( "Fulladder"
       , map = { 'a'    : a[i]
               , 'b'    : b[i]
               , 'cin'  : c_temp[i]
               , 'sout' : sout[i]
               , 'cout' : c_temp[i+1]
               , 'vdd'  : vdd
               , 'vss'  : vss
               }
       )
```

**Errors** Some errors may occur :

- `Error in SignalIn :`
  `the lenght of the net must be a positive value.`
  One can not create a net with a negative lenght.

### Instances

**Name** Inst – Creation of instances

### Synopsys

```
Inst ( model
     , name
     , map = connectmap
     )
```

**Description** Instantiation of an instance. The type of the instance is given by the `model` parameter. The connexions are made thanks to the `connectmap` parameters.

### Parameters

- `Model` : Name of the mastercell of the instance to create (mandatory argument)
- `name` : Name of the instance (optional) When this argument is not defined, the instance has a name created by default. This argument is usefull when one wants to create a layout as well. Indeed, the placement of the instances is much easier when the conceptor has chosen himself the name f the instances.</para>
- `connectmap` : Connexions in order to make the netlist

`param` and `map` are dictionnaries as shown in the example below.

### Example

```
Inst ( 'a2_x2'
     , map = { 'i0'  : in0
             , 'i1'  : in1
             , 'q'   : out
             , 'vdd' : vdd
             , 'vss' : vss
             }
     )
```

You can see a concrete example at :

**Errors** Some errors may occur :

- `Error in Inst : the model Model does not exist. Check CRL_CATA_LIB.` Either one has made a mistake in the name of the model, either the environment variable is not correct.
- `Error in Inst : port does not exist in model Model.` One port in map is not correct.
- `Error in Inst : one input net is not dimensionned.` The size of the output nets is automatically calculated bus the input nets must be dimensionned before being connected.

### Generators

**Name** Generate – Interface with the generators

#### Synopsys

```
Generate ( model, modelname, param = dict )
```

**Description** The `Generate` function call is the generic interface to all generators.

#### Arguments

- `model` : Specifies which generator is to be invoked
  - If the generator belongs to the Dpgen library provided by Stratus, the model name of the generator is simply the name of the class of the generator.
  - If the generator is created by the user, the model name of the generator must have the form : "file_name.class_name". (Note that if the the generator is not in the working directory, the directory of the generator to be instantiated has to be added in the CRL_CATA_LIB environment variable)
- `modelname` : Specifies the name of the model to be generated
- `dict` : Specifies the parameters of the generator

**Parameters** Every generator has it's own parameters. They must be described in the map `dict`. Every generator provides a netlist view. Two other views can be generated, if they are provided by the generator. Two parameters have to be given, in order to choose those views :

- 'physical' : True/False, generation of the physical view (optionnal, False by default)
- 'behavioral' : True/False, generation of the behavioral view (optionnal, False by default)

**Errors** Some errors may occur :

- `[Stratus ERROR] Generate : the model must be described in a string.`

## Description of a layout

### Place

**Name** Place – Places an instance

#### Synopsys

```
Place ( ins, sym, point )
```

**Description** Placement of an instance. The instance has to be instantiated in the method `Netlist`, in order to use the `Place` function.

**Parameters**

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed. The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `point` : coordinates of the lower left corner of the abutment box of the instance in the current figure.

**Example**

```
Place ( myInst, NOSYM, XY ( 0, 0 ) )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.`
  The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.`
  One can not place an instance twice
- `[Stratus ERROR] Place : wrong argument for placement type.`
  The symetry given as argument is not correct.
- `[Stratus ERROR] Place : wrong argument for placement,`
  " the coordinates must be put in a XY object."
  The coordinates are not descrobed the bood way.

**PlaceTop**

**Name** PlaceTop – Places an instance at the top of the "reference instance"

**Synopsys**

```
PlaceTop ( ins, sym, offsetX, offsetY )
```

**Description** Placement of an instance. The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the top left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

**Parameters**

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed. The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

**Example**

```
Place    ( myInst1, NOSYM, 0, 0 )
PlaceTop ( myInst2, SYM_Y )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.` One can not place an instance twice
- `[Stratus ERROR] PlaceTop : no previous instance.` One can use `PlaceTop` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceTop : wrong argument for placement type.` The symetry given as argument is not correct.

**PlaceBottom**

**Name** PlaceBottom – Places an instance below the "reference instance"

**Synopsys**

```
PlaceBottom ( ins, sym, offsetX, offsetY )
```

**Description** Placement of an instance. The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The top left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

**Parameters**

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed. The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

**Example**

```
Place       ( myInst1, NOSYM, 0, 0 )
PlaceBottom ( myInst2, SYM_Y       )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.` One can not place an instance twice
- `[Stratus ERROR] PlaceBottom : no previous instance.` One can use `PlaceBottom` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceBottom : wrong argument for placement type.` The symetry given as argument is not correct.

**PlaceRight**

**Name** PlaceRight – Places an instance at the right of the "reference instance"

**Synopsys**

```
PlaceRight ( ins, sym, offsetX, offsetY )
```

**Description** Placement of an instance. The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom left corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom right corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

**Parameters**

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed. The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

**Example**

```
Place      ( myInst1, NOSYM, 0, 0 )
PlaceRight ( myInst2, NOSYM )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.` One can not place an instance twice
- `[Stratus ERROR] PlaceRight : no previous instance.` One can use `PlaceRight` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceRight : wrong argument for placement type.` The symetry given as argument is not correct.

**PlaceLeft**

**Name** PlaceLeft – Places an instance at the left of the "reference instance"

**Synopsys**

```
PlaceLeft ( ins, sym, offsetX, offsetY )
```

**Description** Placement of an instance. The instance has to be instantiated in the method `Netlist` in order to use the `PlaceTop` function.

The bottom right corner of the abutment box of the instance is placed, after beeing symetrized and/or rotated, toward the bottom left corner of the abutment box of the "reference instance". The newly placed instance becomes the "reference instance".

**Parameters**

- `ins` : Instance to place.
- `sym` : Geometrical operation to be performed on the instance before beeing placed. The `sym` argument can take eight legal values :
  - `NOSYM` : no geometrical operation is performed
  - `SYM_Y` : Y becomes -Y, that means toward X axe symetry
  - `SYM_X` : X becomes -X, that means toward Y axe symetry
  - `SYMXY` : X becomes -X, Y becomes -Y
  - `ROT_P` : a positive 90 degrees rotation takes place
  - `ROT_M` : a negative 90 degrees rotation takes place
  - `SY_RP` : Y becomes -Y, and then a positive 90 degrees rotation takes place
  - `SY_RM` : Y becomes -Y, and then a negative 90 degrees rotation takes place
- `offsetX` (optionnal) : An offset is put horizontally. The value given as argument must be a multiple of PITCH
- `offsetY` (optionnal) : An offset is put vertically. The value given as argument must be a multiple of SLICE

**Example**

```
Place     ( myInst1, NOSYM, 0, 0 )
PlaceLeft ( myInst2, NOSYM )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Placement : the instance doesn't exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] Placement : the first argument is not an instance.`
- `[Stratus ERROR] Placement : the instance is already placed.` One can not place an instance twice
- `[Stratus ERROR] PlaceLeft : no previous instance.` One can use `PlaceLeft` only if a reference instance exist. Use a `Place` call before.
- `[Stratus ERROR] PlaceLeft : wrong argument for placement type.` The symetry given as argument is not correct.

**SetRefIns**

**Name** SetRefIns – Defines the new "reference instance" for placement

**Synopsys**

```
SetRefIns ( ins )
```

**Description** This function defines the new "reference instance", used as starting point in the relative placement functions. It's regarding the abutmentbox of the instance `ins` that the next instance is going to be placed, if using the appropriate functions.

Note that the more recently placed instance becomes automaticaly the "reference instance", if SetRefIns isn't called.

**Parameters**

- `ins` : defines the new "reference instance"

**Example**

```
Place      ( myInst1, NOSYM, 0, 0 )
PlaceRight ( myInst2, NOSYM       )

SetRefIns  ( myInst1 )
PlaceTop   ( myInst3, SYM_Y       )
```

`myInst3` is on top of `myInst1` instead of `myInst2`.

**Errors** Some errors may occur :

- `[Stratus ERROR] SetRefIns : the instance doesn't exist.` If the instance has not been instanciated, it is impossible do to any placement from it.
- `[Stratus ERROR] SetRefIns : the instance ...is not placed.` If the instance has not been placed, it is impossible do to any placement from it.

**DefAb**

**Name** DefAb – Creates the abutment box of the current cell

**Synopsys**

```
DefAb ( point1, point2 )
```

**Description** This function creates the abutment box of the current cell.
Note that one does not have to call this function before saving in order to create the abutment box. The abutment box is created nevertheless (given to placed instances). This function is usefull if one wants to create an abutment before placing the instances.

**Parameters**

- `point1` : coordinates of the bottom left corner of the created abutment box.
- `point2` : coordinates of the top right corner of the created abutment box.

**Example**

```
DefAb ( XY(0, 0), XY(500, 100) )

Place ( self.inst, NOSYM, XY(0, 0) )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] DefAb : an abutment box already exists.` " Maybe you should use ResizeAb function." One has called DefAb but the current cell already has an abutment box. In order to modify the current abutment box, the function to call is ResizeAb.
- `[Stratus ERROR] DefAb : wrong argument,` " the coordinates must be put in a XY object." The type of one of the arguments is not correct. Coordinates must be put in a `XY` object.
- `[Stratus ERROR] DefAb : Coordinates of an abutment Box in y must be multiple of the slice. Coordinates of an abutment Box in x must be multiple of the pitch.` One has called DefAb with non authorized values.

### ResizeAb

**Name** ResizeAb – Modifies the abutment box of the current cell

### Synopsys

```
ResizeAb ( dx1, dy1, dx2, dy2 )
```

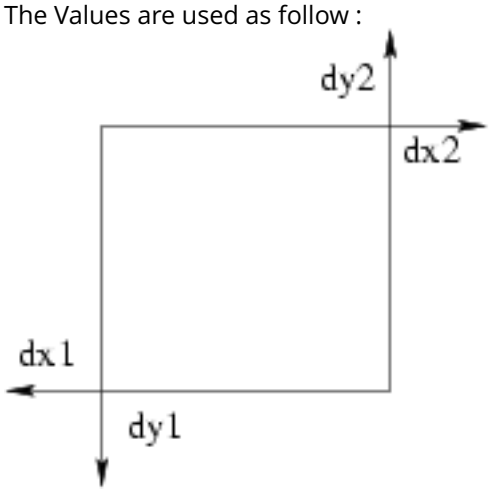
**Description** This function modifies the abutment box of the current cell. The coordinates of the abutment box are the coordinates of the envelop of the abutment boxes of each instance plus the delta values given as argument.

Note that one can not call this function in order to create the abutment box. This fonction only modifies the already created abutment box.

### Parameters

- `(dx1, dy1)` : Values to be substracted to the lower left corner of the previous abutment box.
- `(dx2, dy2)` : Values to be added to the upper right corner of the previous abutment box.

The Values are used as follow :




### Example

```
% Expansion of the abutment box at the top and the bottom
ResizeAb ( 0, 100, 0, 100 )
```

**Errors** Some errors may occur :

- " [Stratus ERROR] ResizeAb :" `Coordinates of an abutment Box in y must be multiple of the slice. Coordinates of an abutment Box in x must be multiple of the pitch.` One has called ResizeAb with non authorized values
- " [Stratus ERROR] ResizeAb :" `one of the values of dx1 or dx2 (dy1 or dy2) is incompatible with the size of the abutment box. Coordinates of an abutment Box in x must be multiple of the pitch.` One has called ResizeAb with a value which deteriorates the abtument box

## Patterns generation extension

### Description of the stimuli

The stimuli used for the simulation are described in a `Stimuli` method. This method is a Python function generator that is automatically called by the `Testbench` method to generate all the stimuli. As a Python function generator, the `yield` instruction have to be used at the end of each stimuli computation.

**Affect value to signals** The method `affect` permits to affect a value to a given signal as follow

```
self._stim.affect(self.Ck,0)
```

**Add stimuli** The method `add` permits to finish a step of simulation by add all the values to the current stimuli

```
self._stim.add()
```

## Place and Route

### PlaceSegment

**Name** PlaceSegment – Places a segment

### Synopsys

```
PlaceSegment ( net, layer, point1, point2, width )
```

**Description** Placement of a segment. The segment is created between `point1` and `point2` on the layer `layer` and with width `width`. It belongs to the net `net`. Note that the segment must be horizontal or vertival.

### Parameters

- `net` : Net which the segment belongs to
- `layer` : Layer of the segment. The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `point1`, `point2` : The segment is created between those two points

### Example

```
PlaceSegment ( myNet, "ALU3", XY (10, 0), XY (10, 100), 2 )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PlaceSegment : Argument layer must be a string.`
- `[Stratus ERROR] PlaceSegment : Wrong argument, the coordinates of the segment must be put in XY objects.`
- `[Stratus ERROR] PlaceSegment : Segments are vertical or horizontal.` The two references given as argument do not describe a vertical or horizontal segment. Wether coordinate x or y of the references must be identical.

]*CopyUpSegment*CopyUpSegmentseccopy

### PlaceContact

**Name** PlaceContact – Places a contact

#### Synopsys

```
PlaceContact ( net, layer, point, width, height )
```

**Description** Placement of a contact. The contact is located at the coodinates of `point`, on the layer `layer` and has a size of 1 per 1. It belongs to the net `net`. Note that the segment must be horizontal or vertival.

#### Parameters

- `net` : Net which the contact belongs to
- `layer` : Layer of the segment. The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `point` : Coodinates of the contact
- `width` : Width of the contact
- `height` : Height of the contact

#### Example

```
PlaceContact ( myNet, "ALU2", XY (10, 0), 2, 2 )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PlaceContact : Argument layer must be a string.`
- `[Stratus ERROR] PlaceContact : Wrong argument, the coordinates of the contact must be put in a XY object.`

### PlacePin

**Name** PlacePin – Places a pin

**Synopsys**

```
PlacePin ( net, layer, direction, point, width, height )
```

**Description** Placement of a pin. The pin is located at the coodinates of `point`, on the layer `layer`, has a a direction of `direction` and size of 1 per 1. It belongs to the net `net`.

**Parameters**

- `net` : Net which the pin belongs to
- `layer` : Layer of the segment. The `layer` argument is a string wich can take different values, thanks to the technology (file described in HUR_TECHNO_NAME)
  - NWELL, PWELL, ptie, ntie, pdif, ndif, ntrans, ptrans, poly, ALU1, ALU2, ALU3, ALU4, ALU5, ALU6, VIA1, VIA2, VIA3, VIA4, VIA5, TEXT, UNDEF, SPL1, TALU1, TALU2, TALU3, TALU4, TALU5, TALU6, POLY, NTIE, PTIE, NDIF, PDIF, PTRANS, NTRANS, CALU1, CALU2, CALU3, CALU4, CALU5, CALU6, CONT_POLY, CONT_DIF_N, CONT_DIF_P, CONT_BODY_N, CONT_BODY_P, via12, via23, via34, via45, via56, via24, via25, via26, via35, via36, via46, CONT_TURN1, CONT_TURN2, CONT_TURN3, CONT_TURN4, CONT_TURN5, CONT_TURN6
- `direction` : Direction of the pin
  - UNDEFINED, NORTH, SOUTH, EAST, WEST
- `point` : Coodinates of the pin
- `width` : Width of the pin
- `height` : Height of the pin

**Example**

```
PlacePin ( myNet, "ALU2", NORTH, XY (10, 0), 2, 2 )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PlacePin : Argument layer must be a string.`
- `[Stratus ERROR] PlacePin : Illegal pin access direction. The values are : UNDEFINED, NORTH, SOUTH, EAST, WEST.`
- `[Stratus ERROR] PlacePin : Wrong argument, the coordinates of the pin must be put in a XY object.`

**PlaceRef**

**Name** PlaceRef – Places a reference

**Synopsys**

```
PlaceRef ( point, name )
```

**Description** Placement of a reference. The reference is located at the coordinates of `point`, with name `name`.

**Parameters**

- `point` : Coodinates of the reference
- `name` : Name of the reference

**Example**

```
PlaceRef ( XY (10, 0), "myref" )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PlaceRef : Wrong argument, the coordinates of the reference must be put in a XY object.`
- `[Stratus ERROR] PlaceRef : Argument layer must be a string.`

**GetRefXY**

**Name** GetRefXY – Returns the coordinates of a reference

**Synopsys**

```
GetRefXY ( pathname, refname )
```

**Description** Computation of coordinates. The point returned (object XY) represents the location of the reference of name `refname` within the coodinates system of the top cell. The reference `refname` is instanciated in an instance found thanks to `pathname` which represents an ordered sequence of instances through the hierarchy.

**Parameters**

- `pathname` : The path in order to obtain, from the top cell, the instance the reference `refname` belongs to
- `refname` : The name of the reference

**Example** The cell which is being created (the top cell), instanciates a generator with instance name “my_dpgen_and2”. This generator instanciates an instance called “cell_1” which the reference “i0_20” belongs to.

```
GetRefXY ( "my_dpgen_and2.cell_1", "i0_20" )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] GetRefXY : The instance’s path must be put with a string.`
- `[Stratus ERROR] GetRefXY : The reference must be done with it’s name : a string.`
- `[Stratus ERROR] GetRefXY : No reference found with name ... in masterCell ...`

**CopyUpSegment**

**Name** CopyUpSegment – Copies the segment of an instance in the current cell

**Synopsys**

```
CopyUpSegment ( pathname, netname, newnet  )
```

**Description** Duplication of a segment. The segment is created with the same cordinates and layer as the segment corresponding to the net `netname` in the instance found thanks to `pathname`. It belongs to the net `newnet`. Note that if several segments correspond to the net, they are all going to be copied.

**Parameters**

- `pathname` : The path in order to obtain, from the top cell, the instance the net `netname` belongs to
- `netname` : The name of the net which the segment belongs to
- `net` : The net which the top cell segment os going to belong to

**Example**

```
CopuUpSegment ( "my_dpgen_and2.cell_1", "i0", myNet )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] CopyUpSegment : The instance's path must be put with a string.`
- `[Stratus ERROR] CopyUpSegment : The segment must be done with it's name : a string.`
- `[Stratus ERROR] CopyUpSegment : No net found with name ... in masterCell ...` There is no net with name `netname` in the instance found thanks to the path `pathname`.
- `[Stratus ERROR] CopyUpSegment : No segment found with net ... in masterCell ...` The net with name `netname` has no segment. So the copy of segment can not be done.
- `[Stratus ERROR] CopyUpSegment : the segment of net ... are not of type CALU.` In other words, the net is not an external net. The copy can be done only with external nets.

**PlaceCentric**

**Name** PlaceCentric – Placement of an instance in the middle of an abutment box

**Synopsys**

```
PlaceCentric ( ins )
```

**Description** This function places an instance in the middle of and abutment box. The instance has to be instantiated in the method `Netlist` in order to use this function.

**Parameters**

- `ins` : Instance to place

**Errors** Some errors may occur :

- `[Stratus ERROR] PlaceCentric: the instance does not exist.` The instance must be instanciated in order to be placed.
- `[Stratus ERROR] PlaceCentric : the instance's size is greater than this model.` The instance must fit in the abutment box. The abutment box may not be big enough.

**PlaceGlu**

**Name** PlaceGlue – Automatic placement of non placed instances

**Synopsys**

```
PlaceGlue ( cell )
```

**Description** This function places, thanks to the automatic placer Mistral of Coriolis, all the non placed instances of the cell.

**Parameters**

- `cell` : the cell which the fonction is applied to

**FillCell**

**Name** FillCell – Automatic placement of ties.

**Synopsys**

```
FillCell ( cell )
```

**Description** This function places automatically ties.

**Parameters**

- `cell` : the cell which the fonction is applied to

**Errors** Some errors may occur :

- `[Stratus ERROR] FillCell : Given cell doesn't exist.` The argument is wrong. Check if one has created the cell correctly.

**Pads**

**Name** PadNorth, PadSouth, PadEast, PasWest – Placement of pads at the periphery of the cell

**Synopsys**

```
PadNorth ( args )
```

**Description** These functions place the pads given as arguments at the given side of the cell (PadNorth : up north, PadSouth : down south ...). Pads are placed from bottom to top for PadNorth and PadSouth and from left to right for PadWest and PasEast.

**Parameters**

- `args` : List of pads to be placed

**Example**

```
PadSouth ( self.p_cin, self.p_np, self.p_ng, self.p_vssick0
         , self.p_vddeck0, self.p_vsseck1, self.p_vddeck1, self.p_cout
         , self.p_y[0], self.p_y[1], self.p_y[2]
         )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PadNorth : not enough space for all pads`. The abutment box is not big enough in order to place all the pads. Maybe one could put pads on other faces of the cell.
- `[Stratus ERROR] PadNorth : one instance doesn't exist`. One of the pads given as arguments does not exist
- `[Stratus ERROR] PadNorth : one argument is not an instance`. One of the pads is not one of the pads of the cell.
- `[Stratus ERROR] PadNorth : the instance ins is already placed`. One is trying to place a pad twice.
- `[Stratus ERROR] PadNorth : pad ins must be closer to the center`. The pad name ins must be put closer to the center in order to route the cell

### Alimentation rails

**Name** AlimVerticalRail, AlimHorizontalRail – Placement of a vertical/horizontal alimentation call back

### Synopsys

```
AlimVerticalRail ( nb )
```

**Description** These functions place a vertical/horizontal alimentation call back. It's position is given by the parameter given.

### Parameters

- `nb` : coordinate of the rail
  - For AlimVerticalRail, `nb` is in pitches i.e. 5 lambdas
  - For AlimHorizontalRail, `nb` is in slices i.e. 50 lambdas

### Example

```
AlimVerticalRail   (  50 )
AlimVerticalRail   ( 150 )

AlimHorizontalRail (  10 )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] AlimHorizontalRail : Illegal argument y, y must be between ... and ...` The argument given is wrong : the call back would not be in the abutment box.
- `[Stratus ERROR] Placement of cells : please check your file of layout with DRUC`. The placement of the cell needs to be correct in order to place a call back. Check the errors of placement.

### Alimentation connectors

**Name** AlimConnectors – Creation of connectors at the periphery of the core of a circuit

**Synopsys**

```
AlimConnectors()
```

**Description** This function creates the connectors in Alu 1 at the periphery of the core.

**PowerRing**

**Name** PowerRing – Placement of power rings.

**Synopsys**

```
PowerRing ( nb )
```

**Description** This function places power rings around the core and around the plots.

**Parameters**

- `nb` : Number of pair of rings vdd/vss

**Example**

```
PowerRing ( 3 )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] PowerRing : Pads in the north haven't been placed.` The pads of the 4 sides of the chip must be placed before calling function PowerRing.
- `[Stratus ERROR] PowerRing : too many rings, not enough space.` Wether The argument of PowerRing is to big, or the abutment box of the chip is to small. There's no space to put the rings.

**RouteCk**

**Name** RouteCk – Routing of signal Ck to standard cells

**Synopsys**

```
RouteCk ( net )
```

**Description** This function routes signal Ck to standard cells.

**Parameters**

- `net` : the net which the fonction is applied to

**Errors** Some errors may occur :

- `[Stratus ERROR] RouteCk : Pads in the north haven't been placed` The pads must be placed before calling RoutageCk.

## Instanciation facilities

**Buffer**

**Name** Buffer – Easy way to instantiate a buffer

### Synopsys

```
netOut <= netIn.Buffer()
```

**Description** This method is a method of net. The net which this method is applied to is the input net of the buffer. The method returns a net : the output net. Note that it is possible to change the generator instanciated with the `SetBuff` method.

### Example

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn  ( "a", 4 )

    self.S = SignalOut ( "s", 4 )

    self.Vdd = VddIn  ( "vdd" )
    self.Vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S <= self.A.Buffer()
```

### Multiplexor

**Name** Mux – Easy way to instantiate a multiplexor

### Synopsys

```
netOut <= netCmd.Mux ( arg )
```

**Description** This method is a method of net. The net which this method is applied to is the command of the multiplexor. The nets given as parameters are all the input nets. This method returns a net : the output net. There are two ways to describe the multiplexor : the argument `arg` can be a list or a dictionnary. Note that it is possible to change the generator instanciated with the `SetMux` method.

### Parameters

- List : For each value of the command, the corresponding net is specified. All values must be specified. For example :

  ```
  out <= cmd.Mux ( [in0, in1, in2, in3] )
  ```

  The net out is then initialised like this :

  ```
  if cmd == 0 : out <= in0
  if cmd == 1 : out <= in1
  if cmd == 2 : out <= in2
  if cmd == 3 : out <= in3
  ```

- Dictionnary : A dictionnary makes the correspondance between a value of the command and the corresponding net. For example :

  ```
  out <= cmd.Mux ( {"0" : in0, "1" : in1, "2" : in2, "3" : in3} )
  ```

  This initialisation corresponds to the one before. Thanks to the use of a dictionnary, the connections can be clearer :

- `'default'`: This key of the dictionnary corresponds to all the nets that are not specified For example :

  ```
  out <= cmd.Mux ( {"0" : in0, "default" : in1} )
  ```

  This notation corresponds to :

  ```
  if cmd == 0 : out <= in0
  else        : out <= in1
  ```

  Note that if there is no `'default'` key specified and that not all the nets are specified, the non specified nets are set to 0.

- `#` and `?` : When a key of the dictionnary begins with `#`, the number after the `#` has to be binary and each ? in the number means that this bit is not precised For example :

  ```
  out <= cmd.Mux ( {"#01?" : in0, "default" : in1} )
  ```

  This notation corresponds to :

  ```
  if cmd in ( 2, 3 ) : out <= in0
  else               : out <= in1
  ```

- `,` and `-` : When keys contains thoses symbols, it permits to enumerate intervals For example :

  ```
  out <= cmd.Mux ( {"0,4" : in0, "1-3,5" : in1} )
  ```

  This notation corresponds to :

  ```
  if   cmd in ( 0, 4 )       : out <= in0
  elif cmd in ( 1, 2, 3, 5) : out <= in1
  else                       : out <= 0
  ```

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A    = SignalIn  (    "a", 4 )
    self.B    = SignalIn  (    "b", 4 )
    self.C    = SignalIn  (    "c", 4 )
    self.D    = SignalIn  (    "d", 4 )

    self.Cmd1 = SignalIn  ( "cmd1", 2 )
    self.Cmd2 = SignalIn  ( "cmd2", 4 )

    self.S1   = SignalOut (   "s1", 4 )
    self.S2   = SignalOut (   "s2", 4 )

    self.Vdd = VddIn  ( "vdd" )
    self.Vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S1 <= self.Cmd1.Mux ( [sefl.A, self.B, self.C, self.D] )

    self.S2 <= self.Cmd2.Mux ( { "0"       : self.A
                               , "1,5-7"   : self.B
                               , "#1?1?"   : self.C
                               , "default" : self.D
                               } )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Mux : all the nets must have the same lenght.` All the input nets pust have the same lenght.
- `[Stratus ERROR] Mux : there are no input nets.` The input nets seem to have been forgotten.
- `[Stratus ERROR] Mux : wrong argument type.` The connections of the buses are not described by a list nor a dictionnary.
- `[Stratus ERROR] Mux :the number of nets does not match with the lenght of the command.` When using a list, the number of nets has to correspond to the number of possible values of the command.
- `[Stratus ERROR] Mux : wrong key.` One of the key of the dictionnary is not un number, neither a list or an interval.
- `[Stratus ERROR] Mux :when an interval is specified, the second number of the interval must be greater than the first one.` When creating an interval with "-", the second number has to be greater than the first one.
- `[Stratus ERROR] Mux :the binary number does not match with the lenght of the command.` When using the `#` notation, each digit of the binary number corresponds to a wire of the cmd. The leghts have to correspond.
- `[Stratus ERROR] Mux : after #, the number has to be binary.` When using the `#` notation, the number has to be binary : one can use 0, 1 or ?.

### Shifter

**Name** Shift – Easy way to instantiate a shifter

### Synopsys

```
netOut <= netCmd.Shift ( netIn, direction, type )
```

**Description** This method is a method of net. The net which this method is applied to is the command of the shifter, it's the one which defines the number of bits to shift. The net given as parameter is the input net. The other arguments set the different patameters. The method returns a net : the output net. Note that it is possible to change the generator instanciated with the `SetShift` method.

### Parameters

- `netIn` : the net which is going to be shifted
- `direction` : this string represents the direction of the shift :
  - "left"
  - "right"
- `type` : this string represents the type of the shift :
  - "logical" : only "zeros" are put in the net
  - "arith" : meaningful for "right" shift, the values put in the nets are an extension of the MSB
  - "circular" : the values put in the nets are the ones which have just been taken off

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn ( "a", 4 )

    self.Cmd = SignalIn ( "cmd", 2 )

    self.S1 = SignalOut ( "s1", 4 )
    self.S2 = SignalOut ( "s2", 4 )
    self.S3 = SignalOut ( "s3", 4 )

    self.Vdd = VddIn  ( "vdd" )
    self.Vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S1 <= self.Cmd.Shift ( self.A, "right", "logical" )
    self.S2 <= self.Cmd.Shift ( self.A, "right", "arith"   )

    self.S3 <= self.Cmd.Shift ( self.A, "left", "circular" )
```

If the value of "a" is "0b1001" and the value of "cmd" is "0b10", we will have :

- "s1" : "0b0010"
- "s2" : "0b1110"
- "s3" : "0b0110"

**Errors** Some errors may occur :

- `[Stratus ERROR] Shift : The input net does not have a positive arity.` The net which is going to be shifted must have a positive arity.
- `[Stratus ERROR] Shift : The direction parameter must be "left" or "right".` The "direction" argument is not correct.
- `[Stratus ERROR] Shift : The type parameter must be "logical" or "arith" or "circular".` The "type" argument is not correct.

**Register**

**Name** Reg – Easy way to instantiate a register

**Synopsys**

```
netOut <= netCk.Reg ( netIn )
```

**Description** This method is a method of net. The net which this method is applied to is the clock of the register. The net given as parameter is the input net. The method returns a net : the output net. Note that it is possible to change the generator instanciated with the `SetReg` method.

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A  = SignalIn  (  "a", 4 )
    self.S  = SignalOut (  "s", 4 )

    self.Ck = CkIn ( "ck" )

    self.Vdd = VddIn  ( "vdd" )
    self.Vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S <= self.Ck.Reg ( self.A )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Reg : The input net does not have a positive arity.` The input net must have a positive arity.
- `[Stratus ERROR] Reg : The clock does not have a positive arity.` The clock must have a positive arity.

**Constants**

**Name** Constant – Easy way to instantiate constants

**Synopsys**

```
netOne  <=  One ( 2 )

net8    <= "8"
```

**Description** These functions simplify the way to instanciate constants.

- The functions `One` and`Zero` permits to initialise all the bits of a net to 'one' or 'zero'.
- The instanciation of a constant thanks to a string can be done in decimal, hecadecimal or binary.

**Parameters**

- For `One` and `Zero` :
  - `n` : the arity of the net
- For the instanciation of a constant :
  - the constant given must be a string representing :
    - A decimal number
    - A binary number : the string must begin with "ob"
    - An hexadecimal number : the string must begin with "ox"

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.Ones   = SignalOut (   "ones", 2 )
    self.Zeros  = SignalOut (  "zeros", 4 )

    self.Eight  = SignalOut (  "eight", 4 )
    self.Twentu = SignalOut ( "twenty", 5 )
    self.Two    = SignalOut (    "two", 5 )

    self.Vdd = VddIn  ( "vdd" )
    self.Vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.Ones  <=  One ( 2 )
    self.Zero  <= Zero ( 4 )

    self.Eight   <= "8"
    self.Twenty  <= "0x14"
    self.Two     <= "0b10"
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Const : the argument must be a string representing a number in decimal, binary (0b) or hexa (0x).` The string given as argument does not have the right form.

**Boolean operations**

**Description** Most common boolean operators can be instantiated without the `Inst` constructor.

**List** Boolean operators are listed below :

- `And2 : q <= i0 & i1`
- `Or2 : q <= i0 | i1`
- `Xor2 : q <= i0 ^ i1`
- `Inv : q <= ~i0`

**Generators to instantiate** One can choose the generator to be used. Some methods are applied to the cell and set the generator used when using `&`, `|`, `^` and `~`. The generators used by default are the ones from the virtual library.

Methods are :

- `SetAnd`
- `SetOr`
- `SetXor`
- `SetNot`

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn  ( "a", 4 )
    self.B = SignalIn  ( "b", 4 )
    self.B = SignalIn  ( "c", 4 )

    self.S = SignalOut ( "s", 4 )

    self.vdd = VddIn  ( "vdd" )
    self.vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S <= ( ~self.A & self.B ) | self.C
```

**Errors** Some errors may occur :

- `[Stratus ERROR] & : the nets must have the same lenght`. When one uses boolean expressions, one has to check that the sizes of both nets are equivalent.
- `[Stratus ERROR] : there is no alim`. The cell being created does not have the alimentation nets. The instanciation is impossible.

**Arithmetical operations**

**Description** Most common arithmetic operators can be instantiated without the `Inst` constructor.

**List** Arithmetical operators are listed below :

- `Addition : q <= i0 + i1`
- `Substraction : q <= i0 - i1`
- `Multiplication : q <= i0 * i1`
- `Division : q <= i0 / i1`

**Generators to instantiate** One can choose the generator to be used. Some methods are applied to the cell and set the generator used when using overload. Methods are :

- `SetAdd` (for addition and substraction)
- `SetMult`
- `SetDiv`

The generators used by default are :

- `Addition` : Slansky adder
- `Substraction` : Slansky adder + inversor + cin = '1'
- `Multiplication` : CA2 multiplier (signed, modified booth/Wallace tree)
- `Division` : not available yet

**Example**

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn  ( "a", 4 )
    self.B = SignalIn  ( "b", 4 )

    self.S = SignalOut ( "s", 4 )

    self.T = SignalOut ( "t", 8 )

    self.vdd = VddIn  ( "vdd" )
    self.vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S <= self.A + self.B

    self.T <= self.A * self.B
```

**Errors** Some errors may occur :

- `[Stratus ERROR] + : the nets must have the same lenght`. When one uses arithmetic expressions, one has to check that the sizes of both nets are equivalent.
- `[Stratus ERROR] : there is no alim`. The cell being created does not have the alimentation nets. The instanciation is impossible.

**Comparison operations**

**Name** Eq/Ne : Easy way to test the value of the nets

**Synopsys**

```
netOut <= net.Eq ( "n" )
```

**Description** Comparaison functions are listed below :

- `Eq` : returns `true` if the value of the net is equal to `n`.
- `Ne` : returns `true` if the value of the net is different from `n`.

Note that it is possible to change the generator instanciated with the `SetComp` method.

**Parameters** The constant given as argument must be a string representing :

- A decimal number
- A binary number : the string must begin with “ob”
- An hexadecimal number : the string must begin with “ox”

### Example

```
class essai ( Model ) :

  def Interface ( self ) :
    self.A = SignalIn  ( "a", 4 )

    self.S = SignalOut ( "s", 1 )
    self.T = SignalOut ( "t", 1 )

    self.vdd = VddIn  ( "vdd" )
    self.vss = VssIn  ( "vss" )

  def Netlist ( self ) :

    self.S <= self.A.Eq ( "4" )

    self.T <= self.A.Ne ( "1" )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Eq : the number does not match with the net's lenght.` When one uses comparaison functions on one net, one has to check that the number corresponds to the size of the net.
- `[Stratus ERROR] Eq : the argument must be a string representing a number in decimal, binary (0b) or hexa (0x).` The string given as argument does not have the right form.

### Virtual library

**Description** The virtual library permits to create a cell and map it to different libraries without having to change it.

### List of the generators provided

- `a2 : q <= i0 & i1`
- `a3 : q <= i0 & i1 & i2`
- `a4 : q <= i0 & i1 & i2 & i3`
- `na2 : nq <= ~ ( i0 & i1 )`
- `na3 : nq <= ~ ( i0 & i1 & i2 )`
- `na4 : nq <= ~ ( i0 & i1 & i2 & i3 )`
- `o2 : q <= i0 & i1`
- `o3 : q <= i0 & i1 & i2`
- `o4 : q <= i0 & i1 & i2 & i3`
- `no2 : nq <= ~ ( i0 & i1 )`
- `no3 : nq <= ~ ( i0 & i1 & i2 )`
- `no4 : nq <= ~ ( i0 & i1 & i2 & i3 )`
- `inv : nq <= ~ i`

- `buf:q <= i`
- `xr2:q <= i0 ^ i1`
- `nxr2:nq <= ~ ( i0 ^ i1 )`
- `zero:nq <= '0'`
- `one:q <= '1'`
- `halfadder:sout <= a ^ b` and `cout <= a & b`
- `fulladder:sout <= a ^ b ^ cin` and `cout <= ( a & b ) | ( a & cin ) | ( b & cin )`
- `mx2:q <= (i0 & ~cmd) | (i1 & cmd)`
- `nmx2:nq <= ~( (i0 & ~cmd) | (i1 & cmd) )`
- `sff:if RISE ( ck ) : q <= i`
- `sff2:if RISE ( ck ) : q <= (i0 & ~cmd) | (i1 & cmd)`
- `sff3:if RISE ( ck )` : " q <= (io & ~cmdo) | (((i1 & cmd1)|(i2&~cmd1)) & cmdo)"
- `ts:if cmd : q <= i`
- `nts:if cmd : nq <= ~i`

**Mapping file** The virtual library is mapped to the sxlib library. A piece of the corresponding mapping file is shown below. In order to map the virtual library to another library, on has to write a .xml file which makes correspond models and interfaces. Note that the interfaces of the cells must be the same (except for the names of the ports). Otherwise, one has to create .vst file in order to make the interfaces match.

The environment variable used to point the right file is `STRATUS_MAPPING_NAME`.

```
1  <?xml version="1.0" encoding='us-ascii'?>
2
3  <technology name="sxlib">
4      <model name="And2"  realcell="a2_x2"   i0="i0" i1="i1"  q="q"  vdd="vdd" vss
5      <model name="Nand2" realcell="na2_x1"  i0="i0" i1="i1" nq="nq" vdd="vdd" vss
6      <model name="Or2"   realcell="o2_x2"   i0="i0" i1="i1"  q="q"  vdd="vdd" vss
7      <model name="Nor2"  realcell="no2_x1"  i0="i0" i1="i1" nq="nq" vdd="vdd" vss
8      <model name="Xor2"  realcell="xr2_x1"  i0="i0" i1="i1"  q="q"  vdd="vdd" vss
9      <model name="Nxor2" realcell="nxr2_x1" i0="i0" i1="i1" nq="nq" vdd="vdd" vss
10     <model name="Inv"   realcell="inv_x1"   i="i"          nq="nq" vdd="vdd" vss
11     <model name="Buff"  realcell="buf_x2"   i="i"           q="q"  vdd="vdd" vss
12 </technology>
```

**Generators** Some generators are also provided in order to use the cells of the library with nets of more than 1 bit. One has to upper the first letter of the model name in order to user those generators. What is simply done is a for loop with the bits of the nets. The parameter `'nbit'` gives the size of the generator.

**Example**

- Direct instanciation of a cell

```
for i in range ( 4 ) :
  Inst ( 'a2'
       , map = { 'i0'  : neti0[i]
               , 'i1'  : neti1[i]
               , 'q'   : netq[i]
               , 'vdd' : netvdd
               , 'vss' : netvss
               }
       )
```

- Instanciation of a generator

```
Generate ( 'A2', "my_and2_4bits", param = { 'nbit' : 4 } )
Inst ( 'my_and2_4bits'
     , map  = { 'i0'  : neti0
              , 'i1'  : neti1
              , 'q'   : netq
              , 'vdd' : vdd
              , 'vss' : vss
              }
     )
```

**Errors** Some errors may occur :

- `[Stratus ERROR] Inst : the model ... does not exist. Check CRL_CATA_LIB.` The model of the cell has not been found. One has to check the environment variable.
- `[Stratus ERROR] Virtual library : No file found in order to parse. Check STRATUS_MAPPING_NAME.` The mapping file is not given in the environment variable.

## Useful links

### DpGen generators

You can find the documentation of the DPGEN library at : ./DpGen.html

### Arithmetic package of stratus

You can find the documentation of the arithmetic stratus's package at: `file:////users/outil/arith/latest/modules_stratus/arithmetic/doc/arith/index.html`

### Arithmetic generators and some stratus packages

You can find the documentation of the arithmetic library at : `file:////users/outil/arith/latest/doc/index.html`

### Patterns module

You can find the documentation of the patterns module : {filename}Patterns_HTML.rst

# Patterns Generation Extension

## Description

The patterns module of *Stratus* is a set of *Python* classes and methods that allows a procedural description of input pattern file for the logic simulator. The *Stratus* `Pattern` method produces a pattern description file as output. The file generated by `Pattern` method is in pat format, so IT IS STRONGLY RECOMMENDED TO SEE pat(5) manual BEFORE TO USE IT.

## Syntax

From a user point of view, `Pattern` method is a pattern description language using all standard *Python* facilities. Here follows the description of the `Pattern` method. A pat format file can be divided in two parts : declaration and description part. The declaration part is the list of inputs, outputs, internal signals and registers. Inputs are to be forced to a certain value and all the others are to be observed during simulation. The description part is a set of patterns, where each pattern defines the value of inputs and outputs. The pattern number represents actually the absolute time for the simulator. Similarly, a `Pattern` method can be divided in two parts : declaration and description part. Methods related to the declaration must be called before any function related to the description part.

### Declaration part

The first thing you should do in this part is to instantiate the class `Patwrite` to have access to all patterns declaration and description methods. The constructor of this class take as parameters the name of pattern output file and the *Stratus* cell that is described (see `PatWrite` [patwrite]). Then, this part allows you to declare the inputs, the outputs, and internal observing points (see `declar`[declar] and `declar_interface` [declar:sub:*i*nterface]).

### Description part

After all signals are declared, you can begin the description part (see `pattern_begin` [pattern:sub:*b*egin]). In this part you have to define input values which are to be applied to the inputs of the circuit or output values which are to be compare with the values produced during the simulation. (see `affect` [affect], `affect_any` [affect:sub:*a*ny], `affect_int` [affect:sub:*i*nt] and `affect_fix` [affect:sub:*f*ix]). `Pattern` method describes the stimulus by event : only signal transitions are described. After each event there is a new input in the pattern file (see `addpat` [addpat]). Last thing you should do in this part is to generate the output file (see `pattern_end` [pattern:sub:*e*nd]).

## Methods

### PatWrite

This class is used to create patterns for *Stratus* models. Currently it only supports Alliance ".pat" pattern format. Patterns time stamps are in the "absolute date" format, "relative date" isn't allowed. Legal time unit are ps (default), ns, us and ms. The constructor takes as parameters the pattern output filename and an optional reference to Stratus cell.

### declar

Adds a connector from a Stratus model to the pattern interface. Writes the corresponding connector declaration in the pattern file with name, arity and direction automatically extracted from the connector properties. Supported Stratus connectors are:

- SignalIn,
- SignalOut (only supported if used as an output),
- VddIn,
- VssIn,
- CkIn,
- SignalInOut,
- TriState (always an output),
- Signals.

**Parameters**

- connector : can either be a reference to a stratus net or a string containing the name of the stratus net.
- format : optional format for the connectors values into the pattern file, accepted values are :
  - 'B': binary (default),
  - 'X': hexadecimal,
  - 'O': octal.

**declar_interface**

Adds all the connectors from a Stratus model to the pattern interface. Write the corresponding connector declaration in the pattern file with name, arity and direction directly taken from the connector proprieties.

**Parameters**

- cell : the tested Stratus model reference. Optional if a reference to the tested Stratus model was given during instanciation[patwrite].
- format : optional format for the connectors values into the pattern file, accepted values are :
  - 'B': binary (default),
  - 'X': hexadecimal,
  - 'O': octal.

**declar**

Affect a string value to a connector.

**Parameters**

- connector : *Stratus* connector
- value : string to affect to connector

**affect_int**

Affect an integer (CA2) value to a connector. Convert the 2's complement value to the corresponding binary value. The binary size is taken from the connector arity. If the connector is an output, the binary value is preceded by "?".

**Parameters**

- connector : *Stratus* connector.
- value : 2's complement value to affect to the connector.

**affect_fix**

Affect a fixed point value to a connector. Convert the floating point input value to the corresponding fixed point value with word_length=connector.arity() and integer_word_length=iwl. If the connector is an output, the binary value is preceded by "?".

**Parameters**

- connector : *Stratus* connector.
- value : floating point value to convert and asign to connector.
- iwl : integer word length

**affect_any**

Disable comparison between this connector value and the one calculated during simulation.

**Parameters**

- connector : *Stratus* connector.

**addpat**

Adds a pattern in the pattern file.

**pattern_begin**

Mark the end of the interface declaration and the beginning of the test vectors.

**pattern_end**

Mark the end of the test vectors and of the patterns file.

## Example

`Pattern` method for an addaccu

```
    def Pattern(self):
        # initialisation
        pat = PatWrite(self._name+'.pat',self)

        # declaration of ports
        pat.declar(self.ck, 'B')
        pat.declar(self.load, 'B')
        pat.declar(self.input, 'X')
        pat.declar(self.output, 'X')
        pat.declar(self.vdd, 'B')
        pat.declar(self.vss, 'B')

        # use of pat.declar_interface(self) has the same effect

        # description beginning
        pat.pattern_begin()

        # affect vdd and vss values
        pat.affect_int(self.vdd,1)
        pat.affect_int(self.vss,0)

        # first pattern : load an initial value
        pat.affect_int(self.input,5)
        pat.affect_int(self.load,1)
        pat.affect_int(self.ck,0)
        # add the pattern in the pattern file
```

```
pat.addpat()
# compute next event
pat.affect_int(self.ck,1)
pat.addpat()

# compute 22 cycle of accumulation
pat.affect_int(self.load,0)
for i in range(1,22):
   pat.affect_int(self.ck,0)
   pat.addpat()
   pat.affect_int(self.ck,1)
   pat.affect_int(self.output,i+5)
   pat.addpat()

# end of the description
pat.pattern_end()
```

## Datapath Operator Generator

### DpgenInv

- **Name** : DpgenInv – Inverter Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenInv', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits inverter with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1, 2, 4 or 8
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 )
```

- **Example** :

```
from stratus import *

class inst_inv ( Model ) :

  def Interface ( self ) :
    self.i = SignalIn  ( "i", 54 )
    self.o = SignalOut ( "o", 54 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenInv', 'inv_54'
             , param = { 'nbit'     : 54
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'inv_54', 'inst'
                  , map = { 'i0'  : self.i
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenBuff

- **Name** : DpgenBuff – Buffer Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenBuff', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits inverter with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2, 4 or 8

    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0
```

- **Example** :

```
from stratus import *

class inst_buff ( Model ) :

  def Interface ( self ) :
    self.i = SignalIn  ( "i", 32 )
    self.o = SignalOut ( "o", 32 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenBuff', 'buff_32'
             , param = { 'nbit'     : 32
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'buff_32', 'inst'
                  , map = { 'i0'  : self.i
                          , 'q'   : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNand2

- **Name** : DpgenNand2 – Nand2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNand2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs NAND with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)

  - **i1** : input (n bits)
  - **nq** : output (n bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 and i1 )
```

- **Example** :

```
from stratus import *

class inst_nand2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNand2', 'nand2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nand2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNand3

- **Name** : DpgenNand3 – Nand3 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNand3', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits three inputs NAND with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 and i1 and i2 )
```

- **Example** :

```
from stratus import *

class inst_nand3 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 20 )
    self.in2 = SignalIn  ( "in2", 20 )
    self.in3 = SignalIn  ( "in3", 20 )
    self.o   = SignalOut (   "o", 20 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNand3', 'nand3_20'
             , param = { 'nbit'     : 20
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nand3_20', 'inst'
```

```
                    , map = { 'i0'  : self.in1
                            , 'i1'  : self.in2
                            , 'i2'  : self.in3
                            , 'nq'  : self.o
                            , 'vdd' : self.vdd
                            , 'vss' : self.vss
                            }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## Dpgennand4

- **Name** : DpgenNand4 – Nand4 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNand4', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits four inputs NAND with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **i3** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 and i1 and i2 and i3 )
```

- **Example** :

```
from stratus import *

class inst_nand4 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 9 )
    self.in2 = SignalIn  ( "in2", 9 )
    self.in3 = SignalIn  ( "in3", 9 )
    self.in4 = SignalIn  ( "in4", 9 )
    self.o   = SignalOut (   "o", 9 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNand4', 'nand4_9'
             , param = { 'nbit'     : 9
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nand4_9', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'i2'  : self.in3
                          , 'i3'  : self.in4
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenAnd2

- **Name** : DpgenAnd2 – And2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenAnd2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs AND with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **q** : output (`n` bits)

  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 and i1
```

- **Example** :

```
from stratus import *

class inst_and2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.out = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenAnd2', 'and2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'and2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'q'   : self.out
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenAnd3

- **Name** : DpgenAnd3 – And3 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenAnd3', modelname
         , param = { 'nbit'       : n
```

```
                        , 'drive'      : d
                        , 'physical'   : True
                        , 'behavioral' : True
                        }
            )
```

- **Description** : Generates a `n` bits three inputs AND with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional): Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False): In order to generate a layout
  - **behavioral** (optional, default value : False): In order to generate a behavior
- **Behavior** :

```
nq <= i0 and i1 and i2
```

- **Example** :

```
from stratus import *

class inst_and3 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 16 )
    self.in2 = SignalIn  ( "in2", 16 )
    self.in3 = SignalIn  ( "in3", 16 )
    self.out = SignalOut (   "o", 16 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenAnd3', "and3_16"
             , param = { 'nbit'     : 16
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'and3_16', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
```

```
                                , 'i2'  : self.in3
                                , 'q'   : self.out
                                , 'vdd' : self.vdd
                                , 'vss' : self.vss
                                }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref (0, 0) )
```

## DpgenAnd4

- **Name** : DpgenAnd4 – And4 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenAnd4', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits four inputs AND with an output power of `d` named `modelname`.
- Terminal Names :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **i3** : input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 and i1 and i2 and i3
```

- **Example** :

```
from stratus import *

class inst_and4 ( Model ) :

  def Interface ( self ) :
    self.in1   = SignalIn  ( "in1", 2 )
    self.in2   = SignalIn  ( "in2", 2 )
    self.in3   = SignalIn  ( "in3", 2 )
    self.in4   = SignalIn  ( "in4", 2 )
    self.out   = SignalOut (   "o", 2 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenAnd4', 'and4_2'
             , param = { 'nbit'     : 2
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'and4_2', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'i2'  : self.in3
                          , 'i3'  : self.in4
                          , 'q'   : self.out
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNor2

- **Name** : DpgenNor2 – Nor2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNor2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs NOR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **nq** : output (`n` bits)

  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 or i1 )
```

- **Example** :

```
from stratus import *

class inst_nor2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNor2', 'nor2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nor2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNor3

- **Name** : DpgenNor3 – Nor3 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNor3', modelname
         , param = { 'nbit'       : n
```

```
                        , 'drive'      : d
                        , 'physical'   : True
                        , 'behavioral' : True
                        }
            )
```

- **Description** : Generates a `n` bits three inputs NOR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 or i1 or i2 )
```

- **Example** :

```
from stratus import *

class inst_nor3 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 3 )
    self.in2 = SignalIn  ( "in2", 3 )
    self.in3 = SignalIn  ( "in3", 3 )
    self.o   = SignalOut ( "out", 3 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNor3', 'nor3_3'
             , param = { 'nbit'     : 3
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nor3_3', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
```

```
                                , 'i2'  : self.in3
                                , 'nq'  : self.o
                                , 'vdd' : self.vdd
                                , 'vss' : self.vss
                                }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNor4

- **Name** : DpgenNor4 – Nor4 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNor4', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits four inputs NOR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **i3** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 or i1 or i2 or i3 )
```

- **Example** :

```
from stratus import *

class inst_nor4 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 15 )
    self.in2 = SignalIn  ( "in2", 15 )
    self.in3 = SignalIn  ( "in3", 15 )
    self.in4 = SignalIn  ( "in4", 15 )
    self.out = SignalOut (   "o", 15 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNor4', 'nor4_15'
             , param = { 'nbit'     : 15
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nor4_15', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'i2'  : self.in3
                          , 'i3'  : self.in4
                          , 'nq'  : self.out
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )


  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenOr2

- **Name** : DpgenOr2 – Or2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenOr2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs OR with an output power of `drive` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **q** : output (`n` bits)

  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the a map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, the `drive` is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 or i1
```

- **Example** :

```
from stratus import *

class inst_or2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenOr2', 'or2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'or2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'q'   : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )


  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenOr3

- **Name** : DpgenOr3 – Or3 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenOr3', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits three inputs OR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 or i1 or i2
```

- **Example** :

```
from stratus import *

class inst_or3 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 5 )
    self.in2 = SignalIn  ( "in2", 5 )
    self.in3 = SignalIn  ( "in3", 5 )
    self.o   = SignalOut (   "o", 5 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenOr3', 'or3_5'
             , param = { 'nbit'     : 5
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'or3_5', 'inst'
```

```
                    , map = { 'i0'   : self.in1
                            , 'i1'   : self.in2
                            , 'i2'   : self.in3
                            , 'q'    : self.o
                            , 'vdd'  : self.vdd
                            , 'vss'  : self.vss
                            }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenOr4

- **Name** : DpgenOr4 – Or4 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenOr4', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits four inputs OR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **i2** : input (`n` bits)
  - **i3** : input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 or i1 or i2 or i3
```

- **Example** :

```
from stratus import *

class inst_or4 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 16 )
    self.in2 = SignalIn  ( "in2", 16 )
    self.in3 = SignalIn  ( "in3", 16 )
    self.in4 = SignalIn  ( "in4", 16 )
    self.out = SignalOut (   "o", 16 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenOr4', 'or4_16'
             , param = { 'nbit'     : 16
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'or4_16', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'i2'  : self.in3
                          , 'i3'  : self.in4
                          , 'q'   : self.out
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenXor2

- **Name** : DpgenXor2 – Xor2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenXor2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs XOR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **q** : output (`n` bits)

  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optionnal, default value : False) : In order to generate a layout
  - **behavioral** (optionnal, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= i0 xor i1
```

- **Example** :

```
from stratus import *

class inst_xor2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenXor2', 'xor2_8'
             , param = { 'nbit' : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'xor2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'q'   : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenXnor2

- **Name** : DpgenXnor2 – Xnor2 Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenXnor2', modelname
         , param = { 'nbit'       : n
```

```
                        , 'drive'      : d
                        , 'physical'   : True
                        , 'behavioral' : True
                        }
            )
```

- **Description** : Generates a `n` bits two inputs XNOR with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **io** : input (`n` bits)
  - **i1** : input (`n` bits)
  - **nq** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 1 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= not ( i0 xor i1 )
```

- **Example** :

```
from stratus import *

class inst_xnor2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenXnor2', 'xnor2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'xnor2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
```

```
                                    }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNmux2

- **Name** : DpgenNmux2 – Multiplexer Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNmux2', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs multiplexer named `modelname`.
- **Terminal Names** :
  - **cmd** : select ( 1 bit )
  - **io** : input ( `n` bits )
  - **i1** : input ( `n` bits )
  - **nq** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= WITH cmd SELECT not i0 WHEN '0',
                      not i1 WHEN '1';
```

- **Example** :

```
from stratus import *

class inst_nmux2 ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  (  "in1", 5 )
    self.in2 = SignalIn  (  "in2", 5 )
    self.cmd = SignalIn  (  "cmd", 1 )
    self.o   = SignalOut (    "o", 5 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
```

```
    Generate ( 'DpgenNmux2', 'nmux2_5'
             , param = { 'nbit'     : 5
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nmux2_5', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'cmd' : self.cmd
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

def Layout ( self ) :
  Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenMux2

- **Name** : DpgenMux2 – Multiplexer Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenMux2', modelname
         , param = { 'nbit'       : n
                   , 'drive'      : d
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits two inputs multiplexer with an output power of `d` named `modelname`.
- **Terminal Names** :
  - **cmd** : select ( 1 bit )
  - **io** : input ( `n` bits )
  - **i1** : input ( `n` bits )
  - **q** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **nbit_cmd** (mandatory) : Defines the size of the generator
  - **drive** (optional) : Defines the output power of the gates
    - Valid drive are : 2 or 4
    - If this parameter is not defined, it's value is the smallest one permitted
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nq <= WITH cmd SELECT i0 WHEN '0',
                      i1 WHEN '1';
```

- **Example** :

```
from stratus import *

class inst_mux2 ( Model ) :

  def Interface ( self ) :
    self.in1  = SignalIn  ( "in1", 8 )
    self.in2  = SignalIn  ( "in2", 8 )
    self.cmd  = SignalIn  ( "cmd", 1 )
    self.o    = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenMux2', 'mux2_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'mux2_8', 'inst'
                  , map = { 'i0'  : self.in1
                          , 'i1'  : self.in2
                          , 'cmd' : self.cmd
                          , 'q'   : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNbuse

- **Name** : DpgenNbuse – Tristate Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNbuse', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : true
                   , 'behavioral' : true
                   }
         )
```

- **Description** : Generates a `n` bits tristate with an complemented output named `modelname`.
- **Terminal Names** :
  - **cmd** : select ( 1 bit )
  - **io** : input ( `n` bits )
  - **nq** : output ( `n` bits )

  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nts:BLOCK(cmd = '1') BEGIN
    nq <= GUARDED not(i0);
END
```

- **Example** :

```
from stratus import *

class inst_nbuse ( Model ) :

  def Interface ( self ) :
    self.i   = SignalIn  (   "i", 29 )
    self.cmd = SignalIn  ( "cmd", 1  )
    self.o   = SignalOut (   "o", 29 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNbuse', 'nbuse29'
             , param = { 'nbit'     : 29
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nbuse29', 'inst'
                  , map = { 'i0'  : self.i
                          , 'cmd' : self.cmd
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenBuse

- **Name** : DpgenBuse – Tristate Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenBuse', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
```

```
                        }
            )
```

- **Description** : Generates a `n` bits tristate named `modelname`.
- **Terminal Names** :
  - **cmd** : select ( 1 bit )
  - **io** : input ( `n` bits )
  - **q** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
nts:BLOCK(cmd = '1') BEGIN
    q <= GUARDED i0;
END
```

- **Example** :

```
from stratus import *

class inst_buse ( Model ) :

  def Interface ( self ) :
    self.i   = SignalIn  (   "i", 8 )
    self.cmd = SignalIn  ( "cmd", 1 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenBuse', 'buse_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'buse_8', 'inst'
                  , map = { 'i0'  : self.i
                          , 'cmd' : self.cmd
                          , 'q'   : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNand2mask

- **Name** : DpgenNand2mask – Programmable Mask Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNand2mask', modelname
         , param = { 'nbit'       : n
                   , 'const'      : constVal
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits conditionnal NAND mask named `modelname`.
- **Terminal Names** :
  - **cmd** : mask control ( 1 bit )
  - **io** : input ( `n` bits )
  - **nq** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **const** (mandatory) : Defines the constant (string beginning with ob, ox or oo functions of the basis)
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - If the `cmd` signal is set to `zero`, the mask is NOT applied, so the whole operator behaves like an inverter.
  - If the `cmd` signal is set to `one`, the mask is applied, the output is the *complemented* result of the input value *ANDed* with the mask (suplied by `constVal`).
  - The constant `constVal` is given to the macro-generator call, therefore the value cannot be changed afterward : it's hard wired in the operator.
  - A common error is to give a real constant for the `constVal` argument. Be aware that it is a character string.
- **Behavior** :

```
nq <= WITH cmd SELECT not(i0)              WHEN '0',
                      not(i0 and constVal) WHEN '1';
```

- **Example** :

```
from stratus import *

class inst_nand2mask ( Model ) :

  def Interface ( self ) :
    self.i   = SignalIn  (   "i", 32 )
    self.cmd = SignalIn  ( "cmd",  1 )
```

```
    self.o   = SignalOut (   "o", 32 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNand2mask', 'nand2mask_0x0000ffff'
             , param = { 'nbit'     : 32
                       , 'const'    : "0x0000FFFF"
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nand2mask_0x0000ffff', 'inst'
                  , map = { 'i0'  : self.i
                          , 'cmd' : self.cmd
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNor2mask

- **Name** : DpgenNor2mask – Programmable Mask Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNor2mask', modelname
         , param = { 'nbit'       : n
                   , 'const'      : constVal
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits conditionnal NOR mask named `modelname`.
- **Terminal Names** :
  - **cmd** : mask control ( 1 bit )
  - **io** : input ( `n` bits )
  - **nq** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **const** (mandatory) : Defines the constant (string beginning with ob, ox or oo functions of the basis)
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior

- **How it works** :
  - If the `cmd` signal is set to `zero`, the mask is NOT applied, so the whole operator behaves like an inverter.
  - If the `cmd` signal is set to `one`, the mask is applied, the output is the *complemented* result of the input value *ORed* with the mask (suplied by `constVal`).
  - The constant `constVal` is given to the macro-generator call, therefore the value cannot be changed afterward : it's hard wired in the operator.
  - A common error is to give a real constant for the `constVal` argument. Be aware that it is a character string.
- **Behavior** :

```
nq <= WITH cmd SELECT not(i0)             WHEN '0',
                      not(i0 or constVal) WHEN '1';
```

- **Example** :

```
from stratus import *

class inst_nor2mask ( Model ) :

  def Interface ( self ) :
    self.i   = SignalIn  (   "i", 8 )
    self.cmd = SignalIn  ( "cmd", 1 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNor2mask', 'nor2mask_000111'
             , param = { 'nbit'     : 8
                       , 'const'    : "0b000111"
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'nor2mask_000111', 'inst'
                  , map = { 'i0'  : self.i
                          , 'cmd' : self.cmd
                          , 'nq'  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenXnor2mask

- **Name** : DpgenXnor2mask – Programmable Mask Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenXnor2mask', modelname
         , param = { 'nbit'       : n
```

```
                        , 'const'      : constVal
                        , 'physical'   : True
                        , 'behavioral' : True
                        }
            )
```

- **Description** : Generates a `n` bits conditionnal XNOR mask named `modelname`.
- **Terminal Names** :
  - **cmd** : mask control ( 1 bit )
  - **io** : input ( `n` bits )
  - **nq** : output ( `n` bits )
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **const** (mandatory) : Defines the constant (string beginning with ob, ox or oo functions of the basis)
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - If the `cmd` signal is set to `zero`, the mask is NOT applied, so the whole operator behaves like an inverter.
  - If the `cmd` signal is set to `one`, the mask is applied, the output is the *complemented* result of the input value *XORed* with the mask (suplied by `constVal`).
  - The constant `constVal` is given to the macro-generator call, therefore the value cannot be changed afterward : it's hard wired in the operator.
  - A common error is to give a real constant for the `constVal` argument. Be aware that it is a character string.
- **Behavior** :

```
nq <= WITH cmd SELECT not(i0)              WHEN '0',
                      not(i0 xor constVal) WHEN '1';
```

- **Example** :

```
from stratus import *

class inst_xnor2mask ( Model ) :

  def Interface ( self ) :
    self.i   = SignalIn  (   "i", 8 )
    self.cmd = SignalIn  ( "cmd", 1 )
    self.o   = SignalOut (   "o", 8 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenXnor2mask', 'xnor2mask_0b000111'
```

```
                , param = { 'nbit'     : 8
                          , 'const'    : "0b000111"
                          , 'physical' : True
                          }
                )
        self.I = Inst ( 'xnor2mask_0b000111', 'inst'
                      , map = { 'i0'  : self.i
                              , 'cmd' : self.cmd
                              , 'nq'  : self.o
                              , 'vdd' : self.vdd
                              , 'vss' : self.vss
                              }
                      )

    def Layout ( self ) :
      Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenAdsb2f

- **Name** : DpgenAdsb2f – Adder/Substractor Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenAdsb2f', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits adder/substractor named `modelname`.
- **Terminal Names** :
  - **io** : First operand (input, `n` bits)
  - **i1** : Second operand (input, `n` bits)
  - **q** : Output operand (ouput, `n` bits)
  - **add_sub** : Select addition or substraction (input, 1 bit)
  - **c31** : Sarry out. In unsigned mode, this is the overflow (output, 1 bit)
  - **c30** : Used to compute overflow in signed mode : `overflow = c31 xor c30` (output, 1 bit)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - If the `add_sub` signal is set to `zero`, an addition is performed, otherwise it's a substraction.
  - Operation can be either signed or unsigned. In unsigned mode `c31` is the overflow ; in signed mode you have to compute overflow by *XORing* `c31` and `c30`

- **Example** :

```
from stratus import *

class inst_ADSB2F ( Model ) :

  def Interface ( self ) :
    self.in1 = SignalIn  ( "in1", 8 )
    self.in2 = SignalIn  ( "in2", 8 )
    self.out = SignalOut (   "o", 8 )
    self.as  = SignalIn  (  "as", 1 )
    self.c0  = SignalOut (  "c0", 1 )
    self.c1  = SignalOut (  "c1", 1 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenAdsb2f', 'adder_8'
             , param = { 'nbit'     : 8
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'adder_8', 'inst'
                  , map = { 'i0'      : self.in1
                          , 'i1'      : self.in2
                          , 'add_sub' : self.as
                          , 'q'       : self.out
                          , 'c30'     : self.c0
                          , 'c31'     : self.c1
                          , 'vdd'     : self.vdd
                          , 'vss'     : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenShift

- **Name** : DpgenShift – Shifter Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenShift', modelname
         , param = { 'nbit'     : n
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a `n` bits shifter named `modelname`.
- **Terminal Names** :
  - **op** : select the kind of shift (input, 2 bits)
  - **shamt** : the shift amount (input, `Y` bits)
  - **i** : value to shift (input, `n` bits)

- **o** : output (`n` bits)
- **vdd** : power
- **vss** : ground

- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout

- **How it works** :
  - If the `op[0]` signal is set to `one`, performs a right shift, performs a left shift otherwise.
  - If the `op[1]` signal is set to `one`, performs an arithmetic shift (only meaningful in case of a right shift).
  - shamt : specifies the shift amount. The width of this signal (`Y`) is computed from the operator's width : `Y = ceil(log2(n)) - 1`

- **Example** :

```
from stratus import *

class inst_shifter ( Model ) :

  def Interface ( self ) :
    self.instop    = SignalIn  (    "instop", 2 )
    self.instshamt = SignalIn  ( "instshamt", 2 )
    self.insti     = SignalIn  (     "insti", 4 )
    self.insto     = SignalOut (     "insto", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenShifter', 'shifter_4'
             , param = { 'nbit'     : 4
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'shifter_4', 'inst'
                  , map = { 'op'    : self.instop
                          , 'shamt' : self.instshamt
                          , 'i'     : self.insti
                          , 'o'     : self.insto
                          , 'vdd'   : self.vdd
                          , 'vss'   : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenShrot

- **Name** : DpgenShrot – Shift/Rotation Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenShrot', modelname
         , param = { 'nbit'     : n
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a `n` bits shift/rotation operator named `modelname`.
- **Terminal Names** :
  - **op** : select the kind of shift/rotation (input, 3 bits)
  - **shamt** : the shift amount (input, `Y` bits)
  - **i** : value to shift (input, `n` bits)
  - **o** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
- **How it works** :
  - If the `op[0]` signal is set to `one`, performs a right shift/rotation , otherwise left shift/rotation occurs.
  - If the `op[1]` signal is set to `one`, performs an arithmetic shift (only meaningful in case of a right shift).
  - If the `op[2]` signal is set to `one`, performs a rotation, otherwise performs a shift..
  - `shamt` specifies the shift amount. The width of this signal (`Y`) is computed from the operator's width : `Y = ceil(log2(n)) - 1`
- **Example** :

```
from stratus import *

class inst_shrot ( Model ) :

  def Interface ( self ) :
    self.rotop     = SignalIn  (     "rotop", 3 )
    self.instshamt = SignalIn  ( "instshamt", 2 )
    self.insti     = SignalIn  (     "insti", 4 )
    self.insto     = SignalOut (     "insto", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenShrot', 'shrot_4'
             , param = { 'nbit'     : 4
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'shrot_4', 'inst'
                  , map = { 'op'    : self.rotop
                          , 'shamt' : self.instshamt
```

```
                                , 'i'     : self.insti
                                , 'o'     : self.insto
                                , 'vdd'   : self.vdd
                                , 'vss'   : self.vss
                                }
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenNul

- **Name** : DpgenNul – Zero Detector Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenNul', modelname
         , param = { 'nbit'     : n
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a `n` bits zero detector named `modelname`.
- **Terminal Names** :
  - **io** : value to check (input, `n` bits)
  - **q** : null flag (1 bit)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
- **Behavior** :

```
q <= '1' WHEN ( i0 = X"00000000" ) ELSE '0';
```

- **Example** :

```
from stratus import *

class inst_nul ( Model ) :

  def Interface ( self ) :
    self.i = SignalIn  ( "i", 4 )
    self.o = SignalOut ( "o", 1 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenNul', 'nul_4'
             , param = { 'nbit'     : 4
                       , 'physical' : True
                       }
```

```
            )
    self.I = Inst ( 'nul_4', 'inst'
                  , map = { 'i0'  : self.i
                          , 'nul' : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenConst

- **Name** : DpgenConst – Constant Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenConst', modelname
         , param = { 'nbit'       : n
                   , 'const'      : constVal
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a `n` bits constant named `modelname`.
- **Terminal Names** :
  - **q** : the constant (output, `n` bit)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **const** (mandatory) : Defines the constant (string beginning with ob, ox or oo functions of the basis)
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **Behavior** :

```
q <= constVal
```

- **Example** :

```
from stratus import *

class inst_const ( Model ) :

  def Interface ( self ) :
    self.o = SignalOut ( "o", 32 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )
```

```
def Netlist ( self ) :
  Generate ( 'DpgenConst', 'const_0x0000ffff'
           , param = { 'nbit'     : 32
                     , 'const'    : "0x0000FFFF"
                     , 'physical' : True
                     }
           )
  self.I = Inst ( 'const_0x0000ffff', 'inst'
                , map = { 'q'   : self.o
                        , 'vdd' : self.vdd
                        , 'vss' : self.vss
                        }
                )

def Layout ( self ) :
  Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenRom2

- **Name** : DpgenRom2 – 2 words ROM Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenRom2', modelname
         , param = { 'nbit'     : n
                   , 'val0'     : constVal0
                   , 'val1'     : constVal1
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a `n` bits 2 words optimized ROM named `modelname`.
- **Terminal Names** :
  - **selo** : address of the value (input, 1 bit)
  - **q** : the selected word (output, `n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **valo** (mandatory) : Defines the first word
  - **val1** (mandatory) : Defines the second word
  - **physical** (optional, default value : False) : In order to generate a layout
- **Behavior** :

```
q <= WITH sel0 SELECT
     constVal0  WHEN B"0",
     constVal1  WHEN B"1";
```

- **Example** :

```
from stratus import *

class inst_rom2 ( Model ) :

  def Interface ( self ) :
    self.sel0 = SignalIn  (    "sel0", 1 )
    self.q    = SignalOut ( "dataout", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenRom2', 'rom2_0b1010_0b1100'
             , param = { 'nbit'     : 4
                       , 'val0'     : "0b1010"
                       , 'val1'     : "0b1100"
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'rom2_0b1010_0b1100', 'inst'
                  , map = { 'sel0' : self.sel0
                          , 'q'    : self.q
                          , 'vdd'  : self.vdd
                          , 'vss'  : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenRom4

- **Name** : DpgenRom4 – 4 words ROM Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenRom4', modelname
         , param = { 'nbit'     : n
                   , 'val0'     : constVal0
                   , 'val1'     : constVal1
                   , 'val2'     : constVal2
                   , 'val3'     : constVal3
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a `n` bits 4 words optimized ROM named `modelname`.
- **Terminal Names** :
  - **sel1** : upper bit of the address of the value (input, 1 bit)
  - **sel0** : lower bit of the address of the value (input, 1 bit)
  - **q** : the selected word (output, `n` bits)
  - **vdd** : power
  - **vss** : ground

- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **val0** (mandatory) : Defines the first word
  - **val1** (mandatory) : Defines the second word
  - **val2** (mandatory) : Defines the third word
  - **val3** (mandatory) : Defines the fourth word
  - **physical** (optional, default value : False) : In order to generate a layout
- **Behavior** :

```
q <= WITH sel1 & sel0 SELECT constVal0  WHEN B"00",
                             constVal1  WHEN B"01",
                             constVal2  WHEN B"10",
                             constVal3  WHEN B"11";
```

- **Example** :

```
from stratus import *

class inst_rom4 ( Model ) :

  def Interface ( self ) :
    self.sel0 = SignalIn  (    "sel0", 1 )
    self.sel1 = SignalIn  (    "sel1", 1 )
    self.q    = SignalOut ( "dataout", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenRom4', 'rom4_0b1010_0b1100_0b1111_0b0001'
             , param = { 'nbit'     : 4
                       , 'val0'     : "0b1010"
                       , 'val1'     : "0b1100"
                       , 'val2'     : "0b1111"
                       , 'val3'     : "0b0001"
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'rom4_0b1010_0b1100_0b1111_0b0001', 'inst'
                  , map = { 'sel0' : self.sel0
                          , 'sel1' : self.sel1
                          , 'q'    : self.q
                          , 'vdd'  : self.vdd
                          , 'vss'  : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenRam

- **Name** : DpgenRam – RAM Macro-Generator

- **Synopsys** :

```
Generate ( 'DpgenRam', modelname
         , param = { 'nbit'     : n
                   , 'nword'    : regNumber
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a RAM of `regNumber` words of `n` bits named `modelname`.
- **Terminal Names** :
  - **ck** : clock signal (input, 1 bit)
  - **w** : write requested (input, 1 bit)
  - **selram** : select the write bus (input, 1 bit)
  - **ad** : the address (input, `Y` bits)
  - **datain** : write bus (input, `n` bits)
  - **dataout** : read bus (output, `n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **nword** (mandatory) : Defines the size of the words
  - **physical** (optional, default value : False) : In order to generate a layout
- **Example** :

```
from stratus import *

class inst_ram ( Model ) :

  def Interface ( self ) :
    self.ck      = SignalIn  (      "ck",  1 )
    self.w       = SignalIn  (       "w",  1 )
    self.selram  = SignalIn  (  "selram",  1 )
    self.ad      = SignalIn  (      "ad",  5 )
    self.datain  = SignalIn  (  "datain", 32 )
    self.dataout = TriState ( "dataout", 32 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenRam', 'ram_32_32'
             , param = { 'nbit'     : 32
                       , 'nword'    : 32
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'ram_32_32', 'inst'
                  , map = { 'ck'      : self.ck
                          , 'w'       : self.w
```

```
                                , 'selram'   : self.selram
                                , 'ad'       : self.ad
                                , 'datain'   : self.datain
                                , 'dataout'  : self.dataout
                                , 'vdd'      : self.vdd
                                , 'vss'      : self.vss
                                }
                    )

    def Layout ( self ) :
      Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenRf1

- **Name** : DpgenRf1, DpgenRf1ro – Register File Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenRf1', modelname
         , param = { 'nbit'     : n
                   , 'nword'    : regNumber
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a register file of `regNumber` words of `n` bits without decoder named `modelname`.
- **Terminal Names** :
  - **ckok** : clock signal (input, 1 bit)
  - **sel** : select the write bus (input, 1 bit)
  - **selr** : the decoded read address (input, `regNumber` bits)
  - **selw** : the decoded write address (input, `regNumber` bits)
  - **dataino** : first write bus (input, `n` bits)
  - **datain1** : second write bus (input, `n` bits)
  - **dataout** : read bus (output, `n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the words (even, between 2 and 64)
  - **nword** (mandatory) : Defines the number of the words (even, between 4 and 32)
  - **physical** (optional, default value : False) : In order to generate a layout
- **How it works** :
  - dataino and datain1 are the two write busses. Only one is used to actually write the register word, it is selected by the sel signal.
  - When sel is set to zero dataino is used to write the register word, otherwise it will be datain1
  - selr, selw : this register file have no decoder, so selr have a bus width equal to `regNumber`. One bit for each word

- The DpgenRf1ro variant differs from the DpgenRf1 in that the register of address zero is stuck to zero. You can write into it, it will not change the value. When read, it will always return zero

- **Example** :

```
from stratus import *

class inst_rf1 ( Model ) :

  def Interface ( self ) :
    self.ck      = SignalIn    (       "ck",  1 )
    self.sel     = SignalIn    (      "sel",  1 )
    self.selr    = SignalIn    (     "selr", 16 )
    self.selw    = SignalIn    (     "selw", 16 )
    self.datain0 = SignalIn    (  "datain0",  4 )
    self.datain1 = SignalIn    (  "datain1",  4 )
    self.dataout = SignalOut   (  "dataout",  4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenRf1', 'rf1_4_16'
             , param = { 'nbit'     : 4
                       , 'nword'    : 16
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'rf1_4_16', 'inst'
                  , map =  { 'ck'      : self.ck
                           , 'sel'     : self.sel
                           , 'selr'    : self.selr
                           , 'selw'    : self.selw
                           , 'datain0' : self.datain0
                           , 'datain1' : self.datain1
                           , 'dataout' : self.dataout
                           , 'vdd'     : self.vdd
                           , 'vss'     : self.vss
                           }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenRf1d

- **Name** : DpgenRf1d, DpgenRf1dro – Register File with Decoder Macro-Generator

- **Synopsys** :

```
Generate ( 'DpgenRf1d', modelname
         , param = { 'nbit'     : n
                   , 'nword'    : regNumber
                   , 'physical' : True
                   }
         )
```

- **Description** : Generates a register file of `regNumber` words of `n` bits with decoder named `modelname`.
- **Terminal Names** :
  - **ck** : clock signal (input, 1 bit)
  - **sel** : select the write bus (input, 1 bit)
  - **wen** : write enable (input, 1 bit)
  - **ren** : read enable (input, 1 bit)
  - **adr** : the read address (input, `Y` bits)
  - **adw** : the write address (input, `Y` bits)
  - **dataino** : first write bus (input, `n` bits)
  - **datain1** : second write bus (input, `n` bits)
  - **dataout** : read bus (output, `n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the words (even, between 2 and 64)
  - **nword** (mandatory) : Defines the number of the words (even, between 6 and 32)
  - **physical** (optional, default value : False) : In order to generate a layout
- **How it works** :
  - dataino and datain1 are the two write busses. Only one is used to actually write the register word, it is selected by the sel signal.
  - When sel is set to zero dataino is used to write the register word, otherwise it will be datain1
  - adr, adw : the width (Y) of those signals is computed from regNumber : `Y = log2(regNumber)`
  - wen and ren : write enable and read enable, allows reading and writing when sets to `one`
  - The DpgenRf1dro variant differs from the DpgenRf1d in that the register of address zero is stuck to zero. You can write into it, it will not change the value. When read, it will always return zero
- **Example** :

```
from stratus import *

class inst_rf1d ( Model ) :

  def Interface ( self ) :
    self.ck      = SignalIn  (       "ck", 1 )
    self.sel     = SignalIn  (      "sel", 1 )
    self.wen     = SignalIn  (      "wen", 1 )
    self.ren     = SignalIn  (      "ren", 1 )
    self.adr     = SignalIn  (      "adr", 4 )
    self.adw     = SignalIn  (      "adw", 4 )
    self.datain0 = SignalIn  (  "datain0", 4 )
    self.datain1 = SignalIn  (  "datain1", 4 )
    self.dataout = SignalOut (  "dataout", 4 )
```

```
    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenRf1d', 'rf1d_4_16'
             , param = { 'nbit'     : 4
                       , 'nword'    : 16
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'rf1d_4_16', 'inst'
                  , map = { 'ck'      : self.ck
                          , 'sel'     : self.sel
                          , 'wen'     : self.wen
                          , 'ren'     : self.ren
                          , 'adr'     : self.adr
                          , 'adw'     : self.adw
                          , 'datain0' : self.datain0
                          , 'datain1' : self.datain1
                          , 'dataout' : self.dataout
                          , 'vdd'     : self.vdd
                          , 'vss'     : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenFifo

- **Name** : DpgenFifo – Fifo Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenFifo', modelname
         , param = { 'nbit'       : n
                   , 'nword'      : regNumber
                   , 'physical'   : True
                   }
         )
```

- **Description** : Generates a FIFO of `regNumber` words of `n` bits named `modelname`.
- **Terminal Names** :
  - **ck** : clock signal (input, 1 bit)
  - **reset** : reset signal (input, 1 bit)
  - **r** : read requested (input, 1 bit)
  - **w** : write requested (input, 1 bit)
  - **rok** : read acknowledge (output, 1 bit)
  - **wok** : write acknowledge (output, 1 bit)
  - **sel** : select the write bus (input, 1 bit)
  - **datain0** : first write bus (input, `n` bits)
  - **datain1** : second write bus (input, `n` bits)

  - **dataout** : read bus (output, `n` bits)
  - **vdd** : power
  - **vss** : ground

- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the words (even, between 2 and 64)
  - **nword** (mandatory) : Defines the number of words (even, between 4 and 32)
  - **physical** (optional, default value : False) : In order to generate a layout

- **How it works** :
  - dataino and datain1 : the two write busses. Only one is used to actually write the FIFO, it is selected by the sel signal.
  - sel : when set to `zero` the dataino is used to write the register word, otherwise it will be datain1.
  - r, rok : set r when a word is requested, rok tells that a word has effectively been popped (rok == not empty).
  - w, wok : set w when a word is pushed, wok tells that the word has effectively been pushed (wok == not full).

- **Example** :

```
from stratus import *

class inst_fifo ( Model ) :

  def Interface ( self ) :
    self.ck      = SignalIn    (       "ck", 1 )
    self.reset   = SignalIn    (    "reset", 1 )
    self.r       = SignalIn    (        "r", 1 )
    self.w       = SignalIn    (        "w", 1 )
    self.rok     = SignalInOut (      "rok", 1 )
    self.wok     = SignalInOut (      "wok", 1 )
    self.sel     = SignalIn    (      "sel", 1 )
    self.datain0 = SignalIn    (  "datain0", 4 )
    self.datain1 = SignalIn    (  "datain1", 4 )
    self.dataout = SignalOut   (  "dataout", 4 )

    self.vdd   = VddIn ( "vdd" )
    self.vss   = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenFifo', 'fifo_4_16'
             , param = { 'nbit'     : 4
                       , 'nword'    : 16
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'fifo_4_16', 'inst'
                  , map = { 'ck'      : self.ck
                          , 'reset'   : self.reset
                          , 'r'       : self.r
                          , 'w'       : self.w
                          , 'rok'     : self.rok
```

```
                                , 'wok'      : self.wok
                                , 'sel'      : self.sel
                                , 'datain0'  : self.datain0
                                , 'datain1'  : self.datain1
                                , 'dataout'  : self.dataout
                                , 'vdd'      : self.vdd
                                , 'vss'      : self.vss
                                }
                    )

    def Layout ( self ) :
      Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenDff

- **Name** : DpgenDff – Dynamic Flip-Flop Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenDff', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a n bits dynamic flip-flop named `modelname`. The two latches of this flip-flop are dynamic, i.e. the data is stored in a capacitor.
- **Terminal Names** :
  - **wen** : write enable (1 bit)
  - **ck** : clock signal (1 bit)
  - **io** : data input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - When wen is set to `one`, enables the writing of the flip-flop
- **Example** :

```
from stratus import *

class inst_dff ( Model ) :

  def Interface ( self ) :
    self.ck  = SignalIn  (  "ck", 1 )
    self.wen = SignalIn  ( "wen", 1 )
```

```
    self.i   = SignalIn  (   "i", 4 )
    self.o   = SignalOut (   "o", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenDff', 'dff_4'
             , param = { 'nbit'     : 4
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'dff_4', 'inst'
                  , map = { "wen" : self.wen
                          , "ck"  : self.ck
                          , "i0"  : self.i
                          ,  "q"  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenDfft

- **Name** : DpgenDfft – Dynamic Flip-Flop with Scan-Path Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenDfft', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a n bits dynamic flip-flop with scan-path named `modelname`. The two latches of this flip-flop are dynamic, i.e. the data is stored in a capacitor.
- **Terminal Names** :
  - **scan** : scan-path mode (input, 1 bit)
  - **scin** : scan path in (input, 1 bit)
  - **wen** : write enable (1 bit)
  - **ck** : clock signal (1 bit)
  - **io** : data input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator

- **physical** (optional, default value : False) : In order to generate a layout
- **behavioral** (optional, default value : False) : In order to generate a behavior

- **How it works** :
  - When scan is set to `one`, it enables the scan-path mode. Note that in scan-path mode, the wen signal is not effective
  - scin is the input of the scan-path. This terminal is different from `i0[0]`. The scout is q[N-1] (in the following example this is `q[31]`)
  - When wen is set to `one` enables the writing of the flip-flop

- **Example** :

```
from stratus import *

class inst_dfft ( Model ) :

  def Interface ( self ) :
    self.scan = SignalIn  ( "scin", 1 )
    self.scin = SignalIn  ( "scan", 1 )
    self.ck   = SignalIn  (   "ck", 1 )
    self.wen  = SignalIn  (  "wen", 1 )
    self.i    = SignalIn  (    "i", 4 )
    self.o    = SignalOut (    "o", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenDfft', 'dfft_4'
             , param = { 'nbit' : 4
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'dfft_4', 'inst'
                  , map = { "wen"  : self.wen
                          , "ck"   : self.ck
                          , "scan" : self.scan
                          , "scin" : self.scin
                          , "i0"   : self.i
                          ,  "q"   : self.o
                          , 'vdd'  : self.vdd
                          , 'vss'  : self.vss
                          }
                  )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenSff

- **Name** : DpgenSff – Static Flip-Flop Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenSff', modelname
         , param = { 'nbit'       : n
```

```
                        , 'physical'   : True
                        , 'behavioral' : True
                        }
            )
```

- **Description** : Generates a n bits static flip-flop named `modelname`. The two latches of this flip-flop are static, i.e. each one is made of two interters looped together.
- **Terminal Names** :
  - **wen** : write enable (1 bit)
  - **ck** : clock signal (1 bit)
  - **io** : data input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - When wen is set to `one`, enables the writing of the flip-flop
- **Example** :

```
from stratus import *

class inst_sff ( Model ) :

  def Interface ( self ) :
    self.ck  = SignalIn  (  "ck", 1 )
    self.wen = SignalIn  ( "wen", 1 )
    self.i   = SignalIn  (   "i", 4 )
    self.o   = SignalOut (   "o", 4 )

    self.vdd = VddIn ( "vdd" )
    self.vss = VssIn ( "vss" )

  def Netlist ( self ) :
    Generate ( 'DpgenSff', 'sff_4'
             , param = { 'nbit'     : 4
                       , 'physical' : True
                       }
             )
    self.I = Inst ( 'sff_4', 'inst'
                  , map = { "wen" : self.wen
                          , "ck"  : self.ck
                          , "i0"  : self.i
                          ,  "q"  : self.o
                          , 'vdd' : self.vdd
                          , 'vss' : self.vss
                          }
```

```
                    )

  def Layout ( self ) :
    Place ( self.I, NOSYM, Ref(0, 0) )
```

## DpgenSfft

- **Name** : DpgenSfft – Static Flip-Flop with Scan-Path Macro-Generator
- **Synopsys** :

```
Generate ( 'DpgenSfft', modelname
         , param = { 'nbit'       : n
                   , 'physical'   : True
                   , 'behavioral' : True
                   }
         )
```

- **Description** : Generates a n bits static flip-flop with scan-path named `modelname`. The two latches of this flip-flop are static i.e. each one is made of two interters looped togethers.
- **Terminal Names** :
  - **scan** : scan-path mode (input, 1 bit)
  - **scin** : scan path in (input, 1 bit)
  - **wen** : write enable (1 bit)
  - **ck** : clock signal (1 bit)
  - **io** : data input (`n` bits)
  - **q** : output (`n` bits)
  - **vdd** : power
  - **vss** : ground
- **Parameters** : Parameters are given in the a map `param`.
  - **nbit** (mandatory) : Defines the size of the generator
  - **physical** (optional, default value : False) : In order to generate a layout
  - **behavioral** (optional, default value : False) : In order to generate a behavior
- **How it works** :
  - When scan is set to `one`, it enables the scan-path mode. Note that in scan-path mode, the wen signal is not effective
  - scin : the input of the scan-path. This terminal is different from `i0[0]`. The scout is `q[N-1]` (in the following example this is `q[3]`)
  - When wen is set to `one`, it enables the writing of the flip-flop
- **Example** :

```
from stratus import *

class inst_sfft ( Model ) :

  def Interface ( self ) :
    self.scan = SignalIn  ( "scin", 1 )
    self.scin = SignalIn  ( "scan", 1 )
```

```
        self.ck   = SignalIn  (   "ck", 1 )
        self.wen  = SignalIn  (  "wen", 1 )
        self.i    = SignalIn  (   "in", 4 )
        self.o    = SignalOut (  "out", 4 )

        self.vdd = VddIn ( "vdd" )
        self.vss = VssIn ( "vss" )

      def Netlist ( self ) :
        Generate ( 'DpgenSfft', 'sfft_4'
                 , param = { 'nbit'     : 4
                           , 'physical' : True
                           }
                 )
        self.I = Inst ( 'sfft_4', 'inst'
                      , map = { "wen"  : self.wen
                              , "ck"   : self.ck
                              , "scan" : self.scan
                              , "scin" : self.scin
                              , "i0"   : self.i
                              ,  "q"   : self.o
                              , 'vdd'  : self.vdd
                              , 'vss'  : self.vss
                              }
                      )

      def Layout ( self ) :
        Place ( self.I, NOSYM, Ref(0, 0) )
```

# For Developpers

## Class Model

### Synopsys

```
class myClass ( Model ) :
    ...

exemple = myClass ( name, param )
```

### Description

Every cell made is a class herited from class `Model`. Some methods have to be created, like `Interface`, `Netlist` ... Some methods are inherited from the class `Model`.

### Parameters

- `name` : The name of the cell (which is the name of the files which will be created)
- `param` : A dictionnary which gives all the parameters useful in order to create the cell

### Attributes

- `_name` : Name of the cell
- `_st_insts` : List of all the instances of the cell
- `_st_ports` : List of all the external nets of the cell (except for alimentations and clock)

- `_st_sigs` : List of all the internal nets of the cell
- `_st_vdds`, `_st_vsss` : Two tabs of the nets which are instancied as `VddIn` and `VssIn`
- `_st_cks` : List of all the nets which are instancied as `CkIn`
- `_st_merge` : List of all the internal nets which have to be merged
- `_param` : The map given as argument at the creation of the cell
- `_underCells` : List of all the instances which are cells that have to be created
- `_and`, `_or`, `_xor`, `_not`, `_buff`, `_mux`, `_reg`, `_shift`, `_comp`, `_add`, `_mult`, `_div` : tells which generator to use when using overload
- `_NB_INST` : The number of instances of the cell (useful in order to automatically give a name to the instances)
- `_TAB_NETS_OUT` and `_TAB_NETS_CAT` : Lists of all the nets automatically created
- `_insref` : The reference instance (for placement)

And, in connection with Hurricane :

- `_hur_cell` : The hurricane cell (None by default)
- `_db` : The database
- `_lib0` : `self._db.Get_CATA_LIB ( 0 )`
- `_nb_alims_verticales`, `_nb_pins`, `_nb_vdd_pins`, `_nb_vss_pins`, `standard_instances_list`, `pad_north`, `pad_south`, `pad_east`, `pad_west` : all place and route stuffs ...

#### Methods

Methods of class `Model` are listed below :

- `HurricanePlug` : Creates the Hurricane cell thanks to the stratus cell. Before calling this method, only the stratus cell is created, after this method, both cells are created. This method has to be called before View and Save, and before Layout.
- `View` : Opens/Refreshes the editor in order to see the created layout
- `Quit` : Finishes a cell without saving
- `Save` : Saves the created cell If several cells have been created, they are all going to be saved in separated files

Some of those methods have to be defined in order to create a new cell :

- `Interface` : Description of the external ports of the cell
- `Netlist` : Description of the netlist of the cell
- `Layout` : Description of the layout of the cell
- `Vbe` : Description of the behavior of the cell
- `Pattern` : Description of the patterns in order to test the cell

### Nets

#### Synopsys

```
netInput = LogicIn ( name, arity )
```

**Description**

Instanciation of net. Differents kind of nets are listed below :

- `LogicIn` : Creation of an input port
- `LogicOut` : Creation of an output port
- `LogicInOut` : Creation of an inout port
- `LogicUnknown` : Creation of an input/output port which direction is not defined
- `TriState` : Creation of a tristate port
- `CkIn` : Creation of a clock port
- `VddIn` : Creation of the vdd alimentation
- `VssIn` : Creation of the vss alimentation
- `Signal` : Creation of an internal net

**Parameters**

- `name` : Name of the net (mandatory argument)
- `arity` : Arity of the net (mandatory argument)
- `indice` : For buses only : the LSB bit (optional argument : set to 0 by default)

Only `CkIn, ``\ ``VddIn` and `VssIn` do not have the same parameters: there is only the `name` parameter (they are 1 bit nets).

**Attributes**

- `_name` : Name of the net
- `_arity` : Arity of the net (by default set to 0)
- `_ind` : LSB of the net
- `_ext` : Tells if the net is external or not (True/False)
- `_direct` : If the net is external, tells the direction (“IN”, “OUT”, “INOUT”, “TRISTATE”, “UNKNOWN”)
- `_h_type` : If the net is an alimentation or a clock, tells the type (“POWER”, “GROUND”, “CLOCK”)
- `_type` : The arithmetic type of the net ( “nr” )
- `_st_cell` : The stratus cell which the net is instanciated in
- `_real_net` : If the net is a part of a net (Sig) it is the real net corresponding
- `_alias` : [] by default. When the net has an alias, it’s a tab. Each element of the tab correspond to a bit of the net (from the LSB to the MSB), it’a a dictionnary : the only key is the net which this net is an alias from, the value is the bit of the net
- `_to_merge` : [] by default. The same as _alias
- `_to_cat` : [] by default. The same as _alias

And, in connection with Hurricane :

- `_hur_net` : A tab with all the hurricane nets corresponding to the stratus net ; From the LSB to the MSB (for example, with a 1 bit net, one gets the hurricane net by doing : `net._hur_net[0]` ).

### Methods

- `Buffer` : Instanciation of a Buffer
- `Shift` : Instanciation of a shifter
- `Mux` : Instanciation of a multiplexor
- `Reg` : Instanciation of a register
- `Eq/Ne` : Instanciation of comparison generator
- `Extend` : A net is extended
- `Alias` : A net is an alias of another net
- `Delete` : Deletion of the Hurricane nets

And the overloards :

- __init__ : Initialisation of nets
- __le__ : initialisation of a net thanks to <= notation
- __getitem__, __geslice__ : Creation of "Sig" nets : which are part of nets (use of `[]` and `[:]`)
- __and__, __or__, __xor__, __invert__ : boolean operation with &, |, ^ ,
- __add__, __mul__, __div__ : arithmetic operators with +, * and /

## Instances

### Synopsys

```
Inst ( model
     , name
     , param = myParam
     , map = myMap
     )
```

### Description

Instantiation of an instance. The type of the instance is given by the `model` parameter. The connexions are made thanks to the `map` parameters.

### Parameters

- `model` : Name of the mastercell of the instance to create (mandatory argument)
- `name` : Name of the instance (optional) When this argument is not defined, the instance has a name created by default. This argument is usefull when one wants to create a layout as well. Indeed, the placement of the instances is much easier when the conceptor has chosen himself the name f the instances.
- `param` : Dictionnary for parameters of the mastercell (optional : only for mastercells which require it)
- `map` : Dictionnary for connexions in order to make the netlist

**Attributes**

- `_name` : Name of the instance (the name given as parameter if there's one, a name created otherwise)
- `_model` : Name of the model given as argument
- `_real_model` : Name of the model created thanks to `_model` and all the parameters
- `_map` : Dictionnary `map` given at the instanciation
- `_param` : Dictionnary `param` given at the instanciation
- `_st_cell` : The stratus cell which the instance is instanciated in
- `_st_masterCell` : The stratus master cell of the instance

For placement :

- `_plac` : tells if the instance is placed or not (UNPLACED by default)
- `_x`, `_y` : the coordinates of the instance (only for placed instances)
- `_sym` : the symetry of the instance (only for placed instances)

And, in connection with Hurricane :

- `_hur_instance` : The hurricane instance (None by default)
- `_hur_masterCell` : The Hurricane master cell of the instance (None by default)

**Methods**

- Delete : Deletion of the Hurricane instance