Stratus: A procedural circuit description language based upon Python

Sophie Belloeil, Damien Dupuis, Christian Masson, Jean-Paul Chaput and Habib Mehrez
University Paris VI, LIP6/SOC Laboratory,
4, place Jussieu,
75252 Paris Cedex 05, France
Email: sophie.belloeil@lip6.fr

***Abstract*— In this paper we present the language Stratus dedicated to the parametrized generation of VLSI modules. Stratus extends the Python language with a set of methods and functions for the procedural generation of netlist and layout views of structured cell based designs. It also provides a programming framework for the development of various optimization techniques that can be applied during module generation. From the designer's point of view, Stratus takes full advantage of Python: a portable, interpretative, easy to learn and object-oriented language. Stratus is the design capture component of the open-source academic Physical Synthesis platform Coriolis, based upon the Hurricane C++ integrated data-base, which provides both C++ and Python high level APIs. Stratus extends this Python API, and allows the designer to use both low level and high level placement, global routing and detailed routing directives.**

I. Introduction

The pace of technological evolution has allowed to reach very high levels of integration, this has fostered the development of hardware specification and modeling languages of higher and higher abstraction level in order to describe System-on-Chip. This follows the trend for designers to focus mainly on system architecture, behaviour modeling and functional validation, while becoming less concerned with physical implementation and performance issues which are completely defered to standard EDA flows, pre-characterized standard cell libraries and soft or hard IPs. However there are multiple design domains where a tight control of the netlist and layout assembly is mandatory to achieve density, speed and power consumption objectives: the design of parametrized regular structures like RAMs, ROMs, data-paths, signal or image processing VLSI components.

Stratus deals more specifically with this last domain, and has been developped as a support framework for our research team on VLSI architectures specific to arithmetic processing. Stratus is part of an open academic EDA Physical Synthesis platform: Coriolis. It extends the high level Python API encapsulating the underlying Hurricane C++ data-base and the CAD tools, providing a fully integrated Physical Synthesis platform.

In sections 2 and 3, we present the Coriolis platform and the Hurricane data-base. In section 4, we explain the choice of the Python language and the mechanisms used to extend the Hurricane C++ API into a Python API. In section 5 we present the main features of Stratus, which are illustrated with a design example in section 6. At last, some results are presented in section 7, and then we conclude.

II. The Coriolis platform

Coriolis [1] is an experimental open-source integrated platform dedicated to the research, development and experimentation of System-on-Chip physical synthesis algorithms and design flows (downloads under the GPL license are available through the project homepage at [2]).

In the Coriolis back-end platform, tools act as algorithmic engines operating on an integrated C++ data-base around which they consistently interact and collaborate. Coriolis offers a set of core functionalities (such as Lef/Def and Alliance [3] interfaces and a graphical user interface) and a progressively enhanced suite of physical synthesis tools intended to support progressive refinement design flows on hierarchical designs. The flow developed and under experimentation is a top-down refinement process which proceeds by a succession of interleaved phases of quadri-partitioning, global routing and net-list timing optimizations:

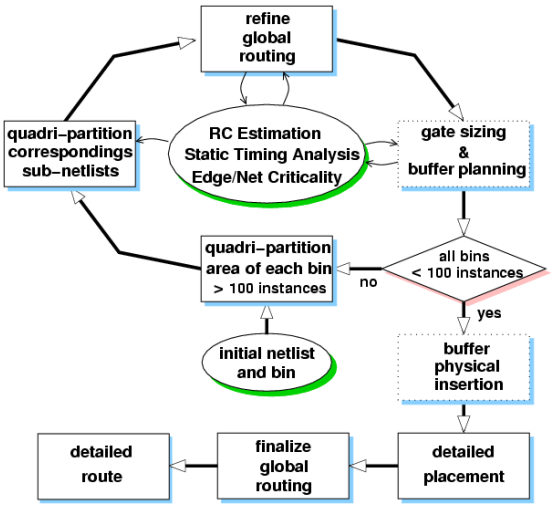

Fig. 1. Overview of the Coriolis flow

Are currently available a standard cell global and detailed placer, a global router, a detailed router, a parasitics estimator

and a static timing analyser [4], a connectivity extractor and a layout versus netlist verifier (with shortcut and disconnect graphical localization capabilities).

III. The Hurricane data-base

Central to the Coriolis project, Hurricane is a lightweight C++ object-oriented data structure and programming platform which provides a unified and consistent modeling of hierarchical VLSI layouts through all the design steps from logic description down to detailed layout. It is outside the scope of this paper to detail Hurricane and interested readers will find documentation along with Coriolis. Here, we will summarize its main features:

- Hurricane provides a powerful object-oriented API for fast access, incremental update and consistent management of the logic and layout views of a hierarchical VLSI design. It fully relieves the application programmer from memory management issues.
- It models in a unified view both the netlist and the routing (global or detailed) through "*hooking*" mechanisms which allow the seamless forward or backward transformation of a netlist into a global routing or a detailed layout (or a mix of those states), ensuring built-in consistency.
- It embeds a built-in high speed graphical display engine of the current state of the design, a feature very useful for designing and debugging layout synthesis algorithms.
- It provides extensibility mechanisms and offers a rich (and extensible) set of powerful and generic query objects named *Collections*. *Collections* are not containers but "*set descriptors*" which provide an associated *Locator* for tracing through the corresponding set of elements. *Collections* can hide a fairly complex algorithmic trace process, visiting huge sets, but with very low memory foot print.
- Like all modern design data-bases, Hurricane represents hierarchical layout as a *folded* memory data model, but provides a **virtually unfolded** view to the tools tracing, annotating or displaying its content. For that purpose it manages the concept of *Occurrences* which can refer any logical or physical item anywhere within the **virtually unfolded** design hierarchy.

IV. A Python-based language

A. Justification of the Python language choice

Module generator languages differ from high level hardware description languages like VHDL or Verilog, as they focus more specifically on the programmability of parametrized descriptions at structural and layout assembly levels. There are for both classes of languages two implementation choices: either develop a specialized language and compiler (like VHDL, Verilog or **Skill** [5]) or extend a general purpose language such as C/C++/Python by enriching it with specialized functions (like SystemC or **Genlib** [6]). As we wanted to easily implement an object-oriented circuits description language providing a rapid protyping and development cycle, the choice of enriching the open-source Python language became natural. Our choice is strengthened by the emergence of languages such as **MyHDL** [7] which has challenged conventional wisdom by making possible the use of Python as a high-level general-purpose language for hardware design. For circuit designers, Stratus is a descripton language which handles VLSI objects while permitting the use of all Python programming capabilities, so they fully benefit from a well-designed, interpretative, easy to learn and widely used open-source language.

B. The Python API

Hurricane provides a python API which is in fact a wrapper around the C++ API. This API: (1) mimics as closely as possible the C++ class hierarchy, (2) maintains a strict one-to-one object pairing between Python and C++ and (3) manages cleanly deletion process. Similar approaches have been developped in generic wrappers such as **swig** [8], but hierarchy and deletetion process control make their use very difficult, therefore a specific wrapper was implemented.

V. Features of Stratus

A. General points

A cell is a hierachical structural description of a circuit in terms of instances and signals (external or internal). It is described as a class with different methods defined, depending on the needs:

- `Interface`: Description of the external ports
- `Netlist`: Description of the logical view
- `Layout`: Description of the physical view
- `Pattern`: Description of stimuli

The created class is derivated from class `Model` which provides useful methods such as `View` (visualization of the layout thanks to Coriolis editor), `Save` (save to disk thanks to Coriolis interface) and `Simul` (call of a simulation tool). The main frame description of the class and its methods is:

```
class myCell ( Model ):
    def Interface ( self ):
        ...
    def Netlist ( self ):
        ...
    def Layout ( self ):
        ...
```

A script is then used in order to create the circuit by calling these methods:

```
Ex = myCell ( "my_cell" )
Ex.Interface()
Ex.Netlist()
Ex.Layout()
Ex.View()
Ex.Save()
```

B. Interface

All the cell's external ports are described in this method. Stratus provides the commonly used types of external ports: input/output (`SignalIn`, `SignalOut`, `SignalInOut`), clock (`CkIn`), supply (`VddIn`, `VssIn`). A signal's instanciation has always the same frame: `signal = SignalType ( "name", arity )`. Besides, clock and supplies are marked as special types in order to identify them for specific treatments.

C. Netlist

In this method instances and their connections are specified.

1) Instanciation: Two ways of doing this description exist:

- By creating explicitly the instances and their connections
- By considering the operators as simple mnemonics

The following examples show how to implement a AND gate:

```
Inst ( "a2", "inst0"
     , map = { 'i0'  : in0
             , 'i1'  : in1
             , 'q'   : out
             , 'vdd' : vdd
             , 'vss' : vss } )
```

or:

```
out <= in0 & in1
```

In the explicit notation, the instance's model is given ("a2") as well as its name ("inst0") and a list of pairs (`map` dictionnary) consisting in an instance port name and the net to which that port is connected. With the mnemonic notation, the same instanciation is automatically done by Stratus. The aim of this notation is basically to ease designers' work. Obviously this notation is available for boolean and arithmetic operators.

In order to make designers' work even easier, several methods have been implemented. For example let's consider a multiplexor with the following behavior (VHDL like description):

```
q <= i0 when ( cmd in (0, 4) )     else
     i1 when ( cmd in (1, 2, 3, 5) else
     0;
```

Stratus offers the possibility to write a concise and powerful description instead of an explicit instanciation:

```
q <= cmd.Mux ({"0,4":i0, "1-3,5":i1, "default":0})
```

Such improvement also exists for:

- Buffer: `out <= in.Buffer()`
- Register: `out <= ck.Reg(in)`
- Shifter: `out <= cmd.Shift(in, direction, type)`

2) Libraries: As for every circuit description language, Stratus relies on cell libraries. Various kind of static libraries are provided: standard cells, symbolic pad cells and full custom cells, as well as a virtual library, permitting to map a circuit to different standard cell libraries without having to modify it. Two other libraries exist, called dynamic libraries, as their blocs are produced by running customizable generators (written in Stratus) with fixed parameters:

- Dpgen: a library of pre-placed regular operators (such as RAM, Fifo, Shifter...) dedicated for data-path generation
- ArithLib: a library of arithmetic operators (such as Sklansky adder, Booth multiplier...)

For example, the generation of a `RAM` of the Dpgen library with 4 words (parameter `nword`) of 8 bits (parameter `nbit`) named "ram_4x8" is done as follow:

```
Generate ( "DpgenRam", "ram_4x8"
         , param = { 'nbit' : 8, 'nword' : 4 } )
```

The instanciation is then done as above, with the name of the generated operator as model.

D. Layout

The physical view is created in this method. This view may contain cells placement as well as routing directives.

1) Placement: The placement can be automatic or handmade depending on what the designer wants to do. The automatic placement, commonly used for irregular control logic, is performed by `PlaceGlue` method, which uses one of the placement tools provided by the Coriolis platform (an argument specifies which tool to use). The handmade placement, commonly used for data-path description, is done with the abutment technique, with the following methods:

- `Place`: exact placement with x,y coordinates
- `PlaceTop`, `PlaceBottom`, `PlaceRight`, `PlaceLeft`: placement relative to current reference instance (with optionnal offset in both directions)
- `SetRefIns`: change of reference instance
- `DefAb`, `ResizeAb`: definition / resize of the abutment box of the cell being described
- `FillCell`: automatic placement of tie cells
- `PadNorth`, `PadSouth`, `PadEast`, `PadWest` : pads placement at the periphery of the cell (automatic repartition on routing grid)

2) Routing: Handmade routing can not be easily exploited in Stratus because there are no functions to describe routing directives and/or routing constraints. Designers can nevertheless create segments (`PlaceSegment`, `CopyUpSegment`) and vias (`PlaceContact`) (using x,y coordinates). As for automatic routing, it is performed, thanks to Coriolis tools, with different methods:

- `AlimVerticalRail`, `AlimHorizontalRail`: generation of the power / ground rail network
- `PowerRing`: generation of a supply ring
- `RouteCk` : generation of a simple grid clock
- `Route` : automatic routing of all signals of a cell

E. Pattern

In addition of creating a cell, Stratus offers the possibility to create test patterns compliant with Alliance (pat) and Synopsys (VHDL IEEE). An object of the `PatWrite` class is created and its methods are used in order to: create the interface (`declar_interface`), assign the values (`assign`) and create a stimulus with the chosen values (`addpat`).

Considering a `And` gate the `Pattern` method would be:

```
def Pattern ( self ) :
  pat = PatWrite ( filename )
  pat.declar_interface ( self )
  pat.pattern_begin()
  pat.assign ( self.vdd, 1 )
  pat.assign ( self.vss, 0 )
  for vala in range(2) :
    for valb in range(2) :
      pat.assign ( self.a, vala )
      pat.assign ( self.b, valb )
      pat.assign ( self.s, vala & valb )
      pat.addpat()
```

VI. A DESIGN EXAMPLE

In this section, we present a circuit description made with Stratus. The design and the corresponding data-path of the circuit are shown in figure 2.

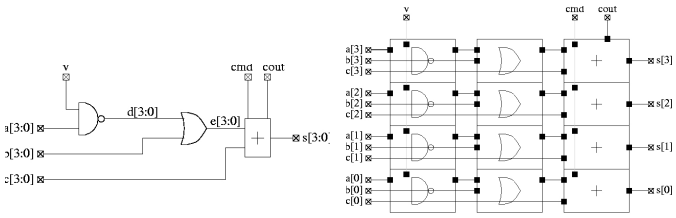

Fig. 2. Circuit

```
class myCell ( Model ) :
  def Interface ( self ) :
    self.a    = SignalIn  (    "a", 4 )
    self.b    = SignalIn  (    "b", 4 )
    self.c    = SignalIn  (    "c", 4 )
    self.v    = SignalIn  (    "v", 1 )
    self.cmd  = SignalIn  (  "cmd", 1 )
    self.cout = SignalOut ( "cout", 1 )
    self.s    = SignalOut (    "s", 4 )
    self.vdd  = VddIn ( "vdd" )
    self.vss  = VssIn ( "vss" )
  def Netlist ( self ) :
    d_aux = Signal ( "d_aux", 4 )
    e_aux = Signal ( "e_aux", 4 )
    ovr   = Signal (  "ovr",  1 )
    temp = Cat ( self.v, self.v, self.v, self.v )
    Generate ( "DpgenNand2", "nand2_4bits"
             , param = { 'nbit' : 4 } )
    Generate ( "DpgenOr2", "or2_4bits"
             , param = { 'nbit' : 4 } )
    Generate ( "DpgenAdsb2f", "adder_4bits"
             , param = { 'nbit' : 4 } )
    self.I1 = Inst ( "nand2_4bits", "inst_nand2"
                   , map = { 'i0'  : temp
                           , 'i1'  : self.a
                           , 'nq'  : d_aux
                           , 'vdd' : self.vdd
                           , 'vss' : self.vss } )
    self.I2 = Inst ( "or2_4bits", "inst_or2"
                   , map = { 'i0'  : d_aux
                           , 'i1'  : self.b
                           , 'q'   : e_aux
                           , 'vdd' : self.vdd
                           , 'vss' : self.vss } )
    self.I3 = Inst ( "adder_4bits", "inst_add2"
                   , map = { 'i0'      : e_aux
                           , 'i1'      : self.c
                           , 'q'       : self.s
                           , 'add_sub' : self.cmd
                           , 'c31'     : self.cout
                           , 'c30'     : ovr
                           , 'vdd'     : self.vdd
                           , 'vss'     : self.vss } )
  def Layout ( self ) :
    Place      ( self.I1, NOSYM, XY(0,0) )
    PlaceRight ( self.I2, NOSYM )
    PlaceRight ( self.I3, NOSYM )
```

VII. PRACTICAL RESULTS

Stratus contains about 50 documented classes, functions and methods. The Dpgen library contains nearly 40 generators whereas the ArithLib, still being developped, contains about 20 generators using both classical and redundant arithmetics. Stratus has successfully been used for 2 years in postgraduate projects as well as for undergraduate course on vlsi design: during Master class, students create a pipeline version of a RISC 32 bits processor, the Mips R3000. The Mips is divided into two parts (each provided in behavioral description): (1) the data-path: hand-made synthesis (students have to choose which operators of the Dpgen library to use) and hand-made placement (2) the controler : automatic synthesis (using a mapper of a behavioral description onto a predefined standard cell library). A hierarchical design is created with these two parts. On the top level cell, automatic placement is applied (method `PlaceGlue`: placement of unplaced instances of the controler taking into account the pre-placed cells of the data-path), as well as pads placement and several routing directives which leads to a layout assembly (figure 3).

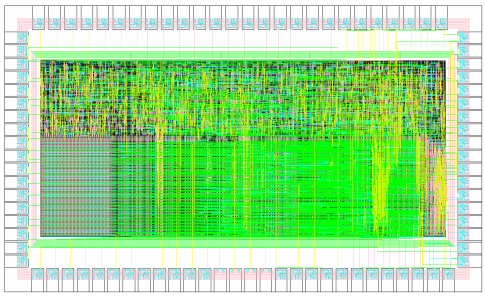

Fig. 3. Mips R3000

VIII. CONCLUSION

We have presented Stratus, a description language of structural and physical views of circuits. More specifically it can easily deal with the design of parametrized regular structures and has been developped for research on VLSI architectures specific to arithmetic processing. Stratus is part of the open academic EDA Physical Synthesis platform Coriolis which provides several back-end tools (placement, global routing, detailled routing, parasistic estimator, static timing analyser). It takes therefore advantage of all those tools and provides a powerful API with several functions specifying structural and layout directives as well as routing directives.

REFERENCES

[1] C. Alexandre, H. Clement, J.-P. Chaput, M. Sroka, C. Masson, and R. Escassut, "Tsunami: An integrated timing-driven place and route research platform," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005, pp. 920–921.

[2] LIP6, "http://www-asim.lip6.fr/recherche/coriolis."

[3] A. Greiner and F. Pecheux, "Alliance: A complete set of cad tools for teaching vlsi design," in *Proceedings of the Third EuroChip Workshop*, 1992.

[4] H. C. Christophe Alexandre, Marek Sroka and C. Masson, "Zephyr: a static timing analyzer integrated in a trans-hierarchical refinement design flow," in *PATMOS'2006: Proceedings of the Power and Timing Modeling Optimization and Simulation conference*, 2006, pp. 319–328.

[5] T. J. Barnes, "Skill: A cad system extension language," in *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, 1990, pp. 266–271.

[6] A. Greiner and F. Pétrot, "Using c to write portable cmos vlsi module generators," in *EURO-DAC '94: Proceedings of the conference on European design automation*, 1994, pp. 676–681.

[7] J. Decaluwe, "Myhdl: a python-based hardware description language," *Linux J.*, vol. 2004, no. 127, p. 5, 2004.

[8] W. Hassan, "Simplified wrapper and interface generator," *Linux J.*, vol. 2000, no. 71es, p. 6, 2000.