SimpMask
Programmer's Manual

— Version 1.0 —

A Simple API for Silicon Compilation
Tech Report TR-02-01-03
January, 2002

Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, ON M5S 3G4, Canada
jzhu@eecg.toronto.edu

Contents

1 Introduction

Names

Layouts of integrated circuits are usually created using two approaches: custom design using a layout editor, or automated design using placer and router.

The full-custom approach makes the best use of human expertise. While experienced designer can create area and speed efficent custom layout, this approach is extremely laborious. Furthermore, custom layouts are not portable to processes with different design rules. It requires major effort to migrate a legacy layout design to new technology even with automation tools. Another disadvantage of this approach is that the layout created using this approach is not parameterizable. Whenever a similar design with slightly different parameter, for example, bit width of an adder, new layout needs to be created even though one can use the same layout architecture, design style, and leaf cells.

The fully-automated approach treats every design as a random network of cells. A “brute-forced” approach is used to find the ”optimal” layout. This approach has become dominant for ASIC design due to its “simplicity” as compared to the full-custom approach. However, as the placers and routers make little effort in preserving the structure inherent in the design, the created layout is often unpredictable, leading to unpredictable, uncontrollable performance.

A large class of VLSI circuits have well-defined, regular layout structures which render the automated placers and routers unsuitable. Such circuits include ROMs, PLAs, SRAMs, FIFOs, and datapaths. For these circuits, designer often has an algorithm in mind to combine basic components, usually by **tiling**, into a large hard intellectual-property (IP) components. Such algorithms are designed to work with different parameters. Since each hard IP has its own specific algorithm, which is hard to generalize, it is undesirable to write a tool that can generate them all. A better, extensible approach is to implement each hard IP in the form of a software component, which can generate the layout when instantiated, possibly with parameters. This software component can either be run locally as a dynamically loadable object module, as is usually the case; or remotely as a CORBA object, if the IP is out-sourced from an IP vendor.

The software components, however different they are, need a common set of services, for example, the query of technology information, the manipulation of geometric objects, and the generation of mask material. SimpMask is an API which provides such service. It is designed in the form of C++ class, which contains a rich set of methods. Hard IPs can be created by subclassing the SimpMask class and encapsulating its own layout generation algorithm.

1.1 Technology Information Query

This first set of methods provided by SimpMask is for technology information query.

The layout of a circuit can be considered as a hierarchical set of polygons, called the masks. Masks in SimpMask are exclusively rectangles, called **tiles**. Each mask is of a different **mask material type**. Such material type is uniquely identified by an integer, which can be accessed by the methods `space`, `poly`, `well`, `metal`, `ndiff`, `pdiff`. Each material type is associated with a particular **mask layer**, called a **plane**. Adding some complication to this simple concept, contacts are naturally associated with two planes since each contact connects two mask material of different planes. We use two identifiers for each contact type, each of which sits in the same plane (called the home plane) as one material it connects, and serves as the mirror of the other identifier. The identifier of a contact can be accessed by the method `contact`, with its first argument being the material on its home plane, and the second argument being the other material.

To ensure the generated layout be portable to different processes, the layout generation needs to know the design rules in order to make decision on the location and shape of the masks. The method `minwidth` returns the minimum width of a mask material, and the method `minspace` returns the minimum space between two materials (can be the same).

1.2 Plain Mask Generation

Masks are generated by calling the `paint` methods within the pair of `beginPaint` and `endPaint` method calls. To paint each mask, one needs to specify the material type identifier, as well as a rectangle, detailing the position and the shape of the mask.

It is often useful to attach **labels** to the layout, which can be used to associate layout masks with high-level circuit or netlist nodes. A label is generated using the `paintLabel` method. A label is usually of rectangular shape, and attached to a particular mask material type. A point or edge shape, which is a degenerated zero width and/or zero height rectangle, is also allowed. The label name `obox`, or the overlapping box, is reserved by the SimpMask's tiling service. This region identifies the box according to which the layout of this cell can be abutted with others to form larger cells.

1.3 Dealing with Hierarchy

Complex layout cannot be created without hierarchy. The basic unit of hierarchy in SimpMask is called a **type**, represented by `SimpType`. Type is traditionally the basic encapsulation unit of programming languages. Here we use that as the basic encapsulation unit of an IP. An IP type contains many different **facets**, among which is the layout facet that SimpMask helps to create. One may find it convenient to consider a type as the concept of cell found in traditional layout systems. However a type in SimpMask can be polymorphic, or can be parameterized. The parameter can be either data values, such as integer constants or string constants, or other types. The layout generation algorithm can access the type parameter using the methods `intParameter`, `stringParameter` and `typeParameter` for integer, string or type parameters respectively.

One can load the layout of a cell by using the `loadMask` method, which expects a type name as its argument. For example, `loadMask( "Inverter" )` loads an inverter cell. Types can be defined under different name spaces (packages) the same way as Java does. For example, one can create the inverter type using `loadMask( "edu.toronto.eecg.Inverter" )`. If the inverter type is a polymorphic and takes one integer value as a parameter value to specify the desired driving strength, one can load the corresponding layout by using `loadMask( "edu.toronto.eecg.Inverter[4]" )`.

One can create **instances** of types (or cells) by using the `paintInst` method. Other than specifying the type and name of the instance, the geometric information needs to be specified by using a transform object, which defines an **affine transformation** of the instantiated cell layout. The transformation not only captures the position of the cell, but also allows one to rotate or mirror the instantiated cell in different directions.

1.4 Geometric Manipulation

The creation of both masks and instances needs to use geometric objects such as rectangles and transforms. SimpMask provides methods to create and manipulate these objects.

One can create a null rectangle or a rectangle with specified position and dimension using the `rect` method. The binding box of a particular type (cell) can be obtained with the `bbox` method. The overlapping box of a type can be obtained with `obox` method. One can also obtain a rectangle by applying a transform on an existing rectangle using the `transformRect` method.

SimpMask has a set of predefined transforms. The most frequently used is the identity transform, accessed by the `identityTransform` method. Other predefined transformations performs rotation or mirroring: `mirrorX`, `mirrorY`, `rotate90`, `rotate90`, `rotate270`, `ref45`, `ref135`. One can obtain a composite transformation of two transforms by using the `transform`. One can translate an existing transform by a specified amount by using the `translate` method.

1.5 Slicing-tree Based Tiling

A large class of VLSI layout can be created using the so-called **tiling methodology**. Here a layout is created by the composition of a set of carefully crafted components, called the leaf cells. The leaf cells are planed in such a way that when they are abutted appropriately, no additional wiring or routing is ever needed. Therefore, a floorplan of the layout is good enough to complete the entire layout.

SimpMask provides the `SimpSlice` class to help create such a floorplan based on the notion of **slicing tree**. At each level, a slicing tree partitions the geometric space it commands by performing a **cut** in a particular direction. In SimpMask, the four directions are identified as `GEO_NORTH` for up, `GEO_SOUTH` for down, `GEO_WEST` for right, and `GEO_EAST` for left.

Each SimpSlice object represents a slicing tree node, and the key service it provides is the `expand` method. A SimpSlice object claims a zero-sized rectangular geometric space from its parent at the current position when it is initially created. The geometric space can then be expanded according to a supplied rectangle and a transform. The rectangle is usually the binding box or the overlapping box of a cell instance. It returns a composite transform that can be used to instantiate the cell at the desired location. Alternatively, the rectangle can be the geometric space claimed by a child slicing tree node, therefore, the expansion can be carried out hierarchically. For a particular tree node, the expansion can be performed only at the direction specified when the node is created.

This hierarchical procedural expansion process has to be performed in a bottom-up fashion along the tree. Without explicitly building the tree, it is best implemented with a stack. SimpSlice goes one more step by eliminating the need of explicitly creating a stack: it directly uses the program calling stack! To make this work, one needs to embed each SimpSlice object either into a C++ scope (a block statement) or a separate method. When the scope is entered, or a method is called, the constructor of the SimpSlice object is called, which initializes the geometric space it commands into a zero-sized rectangle at the current available position of its parent. When the algorithm leaves the scope, or the method is returned, the destructor of the object will be called, which expands its parent with the geometric space it commands.

1.6 Channel Connected Component

Under construction, stay tuned.

1.7 Gridless Routing

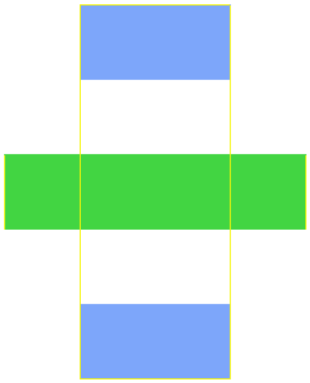
Under construction, stay tuned.

1.8 A Simple Example

This example shows how SimpMask can be used to create layout facets.

For each cell, one needs to create a separate C++ file. The `simpmask.h` header file must be included in the beginning.

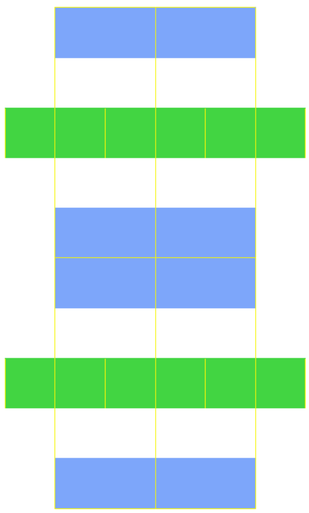
Each cell corresponds to a C++ class derived from the SimpMask class, or its derived classes. The cell **Leaf** is created as follows. The layout generation algorithm is implemented in the `go` method. The algorithm uses SimpMask's plain mask generation service to create the layout. The generated layout is shown as follows.



```
class Leaf : public SimpMask {
  public:
    Leaf( IWorld iwld, IType ti ) : SimpMask( iwld, ti ) {}

    Jerror go( void ) {
        begin();

        // paint masks
        beginPaint();
        paint( metal(1), rect( 2, 0, 4, 2 ) );
        paint( metal(1), rect( 2, 8, 4, 2 ) );
```

```
        paint( ndiff(),  rect( 0, 4, 8, 2 ) );
        endPaint();

        // paint labels
        paintLabel( ndiff(), rect( 0, 4, 0, 2 ), GEO_WEST, "left" );
        paintLabel( ndiff(), rect( 8, 4, 0, 2 ), GEO_WEST, "right" );
        paintLabel( space(), rect( 2, 0, 4, 10 ), GEO_WEST, "obox" );

        end();
        return 0;
        }
    };
```

The `Composite` cell consists of a two dimensional tiling of `Leaf` instances. To create the layout, the layout generation algorithm first obtain the `Leaf` type by using the `parseType` method of `SimpType`. It then obtains its the binding/overlapping box. It then using the `SimpSlice` class to help find the instantiation transform of each instance. The top-level tree is responsible for expanding from left to right. And the leaf-level tree is responsible for expanding from bottom to top.

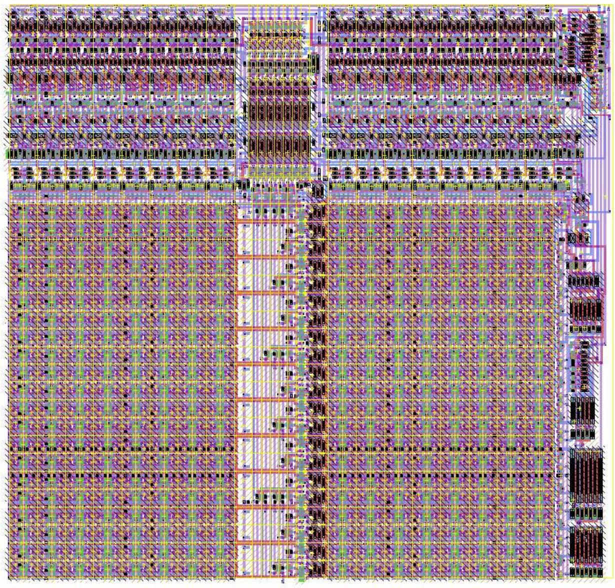


```
class Composite : public SimpMask {
  public:
    Composite( IWorld iwld, IType ti ) : SimpMask( iwld, ti ) {}

    Jerror go( void ) {
```

```
        SimpType    leafType = loadMask( "Leaf" );
//      Rect        box = bbox( leafType );
        Rect        box = obox( leafType );
        SimpSlice   root( NULL, GEO_EAST ); // the roof tree expanding
                                            // from left to right

        begin();
        for( int i = 0; i < 2; i ++ ) {
            // the child slicing tree expanding up
            SimpSlice   cur( &root, GEO_NORTH );
            char        buffer[80];

            for( int j = 0; j < 2; j ++ ) {
                sprintf( buffer, "inst_%d_%d", i, j );
                if( j & 0x1 )
                    paintInst( leafType, buffer, cur.expand(box) );
                else
                    paintInst( leafType, buffer, cur.expand(box, mirrorX()) );
                }
            }
        end();
        return 0;
        }
    };
```

1.9 An SRAM Show Case

As a more ambitious application of SimpMask API, a parameterized static RAM example is created. One can create its layout by specifying the number of addressable words, the data width of each word, and the number of columns. The layout is created by the hierarchical tiling of about 50 leaf cells, which are in turn custom-designed.



2 SimpMask API

simpmask.h

Names

Author: Jianwen Zhu
Version: 1.0

2.1 class **SimpMask**

SimpMask Name Space

Names

2.1.1

SimpType **typeParameter** (Jint i)

parameter access

This method obtains the ith parameter. The parameter has to be a type.

Return Value: a SimpType object
Parameters: `i` — the parameter number

2.1.2

Jint **intParameter** (Jint i)

parameter access

This method obtains the ith parameter. The parameter has to be an integer.

Return Value: the integer value of the parameter
Parameters: `i` — the parameter number

2.1.3

char* **stringParameter** (Jint i)

parameter access

This method obtains the ith parameter. The parameter has to be a string.

Return Value: the string value of the parameter
Parameters: `i` — the parameter number

2.1.4

Jint **space** ()

material type identifier query

This method obtains a special material type called space, which stands for empty material.

Return Value: the identifier
Parameters: `none` —

2.1.5

Jint **poly** (Jint i)

material type identifier query

This method obtains the identifier for polysilicon.

Return Value: the identifier
Parameters: `i` — the ith layer of the poly

2.1.6

Jint **metal** (Jint i)

material type identifier query

This method obtains the identifier for metal.

Return Value: the identifier
Parameters: `i` — the ith layer of the metal

2.1.7

Jint **pwell** (void)

material type identifier query

This method obtains the identifier for P type well.

Return Value: the identifier
Parameters: `none` —

2.1.8

Jint **nwell** (void)

material type identifier query

This method obtains the identifier for N type well.

Return Value: the identifier
Parameters: `none` —

2.1.9

Jint **pdiff** (void)

material type identifier query

This method obtains the identifier for P type diffusion.

Return Value: the identifier
Parameters: `none` —

2.1.10

Jint **ndiff** (void)

material type identifier query

This method obtains the identifier for N type diffusion.

Return Value: the identifier
Parameters: `none` —

2.1.11

Jint **contact** (Jint i, Jint j)

material type identifier query

This method obtains the identifier for contact or via.

Return Value: the identifier
Parameters: `i` — the first material (in the home plane of the contact) the contact connects
`j` — the second material (in the home plane of the contact) the contact connects

2.1.12

Jint **minwidth** (Jint i)

design rule query

This method obtains the minimum width of a material type.

Return Value: the width value in units of λ
Parameters: `i` — the material type

2.1.13

Jint **minspace** (Jint i, Jint j)

design rule query

This method obtains the minimum space between two material types.

Return Value: the space value in units of λ
Parameters: `i` — the first material type
`j` — the second material type

2.1.14

Rect **rect** (Jint xbot, Jint ybot, Jint width, Jint height)

geometric manipulation

This method creates a rectangle with specified position and dimension.

Return Value: the created rectangle
Parameters: `xbot` — the x coordinate of the lower left corner
`ybot` — the y coordinate of the lower left corner
`width` — the width of the rectangle
`height` — the height of the rectangle

2.1.15

Rect **rect** (void)

geometric manipulation

This method creates a null rectangle.

Return Value: the created rectangle
Parameters: `none` —

2.1.16

Transform **identity** (void)

geometric manipulation

This method creates an identity affine transform.

Return Value: the created transform
Parameters: `none` —

2.1.17

Transform **mirrorX** (void)

geometric manipulation

This method creates an mirror affine transform in the horizontal direction.

Return Value: the created transform
Parameters: `none` —

2.1.18

Transform **mirrorX** (Transform t)

geometric manipulation

This method creates a composite transform by applying the horizontal mirror transform on a transform.

Return Value: the created transform
Parameters: `t` — the transform to be mirrored

2.1.19

Transform **mirrorY** (void)

geometric manipulation

This method creates an vertical mirror affine transform.

Return Value: the created transform
Parameters: `none` —

2.1.20

Transform **mirrorY** (Transform t)

geometric manipulation

This method creates a composite transform by applying the vertical mirror transform on a transform.

Return Value: the created transform
Parameters: `t` — the transform to be mirrored

2.1.21

Transform **rotate90** (void)

geometric manipulation

This method creates a 90 degree rotation transform.

Return Value: the created transform
Parameters: `none` —

2.1.22

Transform **rotate90** (Transform t)

geometric manipulation

This method creates a composite transform by applying the 90-degree rotation transform on a transform.

Return Value:	the created transform
Parameters:	`t` — the transform to be mirrored

2.1.23

Transform **rotate180** (void)

geometric manipulation

This method creates a 180 degree rotation transform.

Return Value:	the created transform
Parameters:	`none` —

2.1.24

Transform **rotate180** (Transform t)

geometric manipulation

This method creates a composite transform by applying the 180-degree rotation transform on a transform.

Return Value: the created transform
Parameters: `t` — the transform to be mirrored

2.1.25

Transform **rotate270** (void)

geometric manipulation

This method creates a 270 degree rotation transform.

Return Value: the created transform
Parameters: `none` —

2.1.26

Transform **rotate270** (Transform t)

geometric manipulation

This method creates a composite transform by applying the 270-degree rotation transform on a transform.

Return Value: the created transform
Parameters: `t` — the transform to be mirrored

2.1.27

Transform **translate** (Transform from, Jint x, Jint y)

geometric manipulation

This method creates a composite transform by translating a transform by a specified amount.

Return Value: the created transform
Parameters: `from` — the transform to be translated
`x` — the horizontal amount to translate
`y` — the vertical amount to translate

2.1.28

Transform **transform** (Transform t1, Transform t2)

geometric manipulation

This method creates a composite transform by applying transform t1 to transform t2.

Return Value: the created transform
Parameters: `t1` — the transform to apply
`t2` — the transform to be applied

2.1.29

Rect **transformRect** (Transform t, Rect r)

geometric manipulation

This method transforms a rectangle r using the affine transform t. transform t1 to transform t2.

Return Value: the created transform
Parameters: `t` — the transform to apply
`r` — the rectangle ti be transformed

2.1.30

void **beginPaint** (void)

mask painting

This method starts a painting session.

Return Value: none
Parameters: `none` —

2.1.31

void **paint** (Jint tt, Rect rect)

mask painting

This method paints a material of type tt at the given rectangular box rect.

Return Value: none
Parameters: `tt` — the material to be painted
`rect` — the geometric shape to be painted

2.1.32

void **endPaint** (void)

mask painting

This method ends a painting session.

Return Value: none
Parameters: `none` —

2.1.33

void **paintLabel** (Jint tt, Rect rect, Jint align, char* text)

mask painting

This method paints a label attached to material type tt at the given rectangular box rect.

Return Value:	none
Parameters:	`tt` — the material the label is to be attached `rect` — the shape of the label `align` — how the label is aligned with rect, can be one of GEO_WEST, GEO_EAST, GEO_NORTH, GEO_SOUTH, or GEO_CENTER `text` — the label text

2.1.34

void **paintInst** (SimpType type, char* name, Transform pos)

mask painting

This method paints an instance of a particular type and particular name.

Return Value:	none
Parameters:	`type` — the type of the instance `name` — the name of the instance. The name can be NULL, in which SimpMask will automatically assign a name `pos` — the transform to apply on the instance

2.1.35

Rect **bbox** (SimpType type)

cell information query

This method obtains the binding box of a cell type.

Return Value: a rectangle defining the box.
Parameters: `type` — the type of the instance

2.1.36

Rect **obox** (SimpType type)

cell information query

This method obtains the overlapping box of a cell type.

Return Value: a rectangle defining the box.
Parameters: `type` — the type of the instance

2.2

class **SimpSlice**

A helper class for tiling

Names

2.2.1

SimpSlice (SimpSlice* parent, Jint direction)

SimpSlice constructor

This constructor constructs a slicing tree node with specified direction and the identity transform.

Return Value: none

Parameters: `parent` — the parent of the tree node. can be NULL if root of the tree.
`direction` — the direction for expansion. can be one of GEO_WEST, GEO_EAST, GEO_NORTH or GEO_SOUTH.

2.2.2

SimpSlice (SimpSlice* parent)

SimpSlice constructor

This constructor constructs a slicing tree node with specified the GEO_NORTH direction and the identity transform.

Return Value: none

Parameters: `parent` — the parent of the tree node. can be NULL if root of the tree.

2.2.3

~**SimpSlice** (void)

SimpSlice destructor

The destructor expands the slicing tree node into its parent.

Return Value:	none
Parameters:	`none` —

2.2.4

Transform **expand** (Rect bbox, Transform xform)

SimpSlice expansion

This method expands the geometric region of a slicing tree node with the rectangle bbox, which needs to be first transformed by the transform xform.

Return Value:	the net transform to position the rectangle
Parameters:	`bbox` — the rectangle
	`xform` — the transform

2.2.5

Transform **expand** (Rect rect)

SimpSlice expansion

This method expands the geometric region of a slicing tree node with the rectangle bbox.

Return Value: the net transform to position the rectangle
Parameters: `bbox` — the rectangle
`xform` — the transform

Class Graph