Mod2mag
User's Manual

Nicolas Paris [1]

November 8, 1991

[1] Laboratoire d'Informatique de l'Ecole Normale Supérieure URA 1327 du C.N.R.S.

Acknowledgements

I am very grateful to Pr. Bertrand Zavidovique who trusted me for such a large work.

I wish to thank Thierry Bernard, Georges Quenot, Ronan Keryell and Jean-Dominique Gascuel for their help and suggestions.

Last but not least, I give Philippe Hoogvorst special thanks for the time he has spent checking this document.

Chapter 1

Introduction

1.1 Presentation of Mod2mag

Mod2mag is the masterpiece of the VLSI compilation environment developed at the Computer Science Laboratory of the Ecole Normale Supérieure. This document is mostly dedicated to the description of **mod2mag**. However a manual page is available (mod2mag(1)) to summarize the flags of **mod2mag**. This environment is a complement of the Magic Software Distribution of the University of California at Berkeley. File formats are compatible with the formats of the **magic** tools. It is necessary for the reader to read the manual of the Berkeley distribution in order to understand this document.

Mod2mag is a silicon compiler. It takes input files in the syntax of the **model** language and may generate different kinds of output:

- Transistor level netlists for electric simulation,
- Switch-level/logical/behavioral netlists for multi-level simulation,
- Layouts for **magic**,
- Layout netlists for the **magic** router,
- Pcb netlists for the Dedale2000 pcb router.

Mod2mag uses basic layout tiles which are designed with **magic**.

Being very efficient for the development of small cells, the **magic** environment does not provide any efficient facility to convert a schematic description into a layout hierarchy. **Mod2mag** has been developed to tackle this problem. In this environment:

- **Magic** is used to design basic library cells,
- **Mod2mag** is used to describe, simulate and debug schematics and finally to generate the layout, and the PCB netlists.

Mod2mag allows to follow a top-down methodology due to the behavioral description of the cells. It is associated with **msim** for the multi-level simulation. Our **model** syntax can be classified in Hardware Description Language. Furthermore, it provides facilities to develop parametrical libraries in order to generate layout in both full-custom and/or standard cell style.

1.2 Libraries, examples and technologies

Associated with the **Magic** environment, this environment is efficient for the design of VLSI. The last problem is the development of libraries in the designer technology. Tools and methodologies are provided in order to help a designer or a local maintainer to develop custom libraries in a local technology. This distribution is associated with layout libraries in 2 technologies:

- the `mcmos` technology which corresponds to the ECDM20-C 2μm Cmos technology of European Silicon Structure. A specification of the standard cells for this technology is provided in the appendix A. Libraries and examples are provided in the software distribution in the directory `~cad/ulm/mcmos`.
- the `ecpd` technology which corresponds to the ECPD15 1.6μm Cmos technology of European Silicon Structure. A specification of the standard cells for this technology is provided in the appendix B. Libraries are provided in the software distribution in the directory `~cad/ulm/ecpd`. This technology is scalable (unless for pads which must be adapt to the technology). So it is relatively easy to develop a technology in the existing ECPD12 1.2μm Cmos technology of European Silicon Structure and to the ECPD8 0.8μm Cmos technology to come of European Silicon Structure.

The appendix C explains to the local maintainer how to develop a new environment for a new technology.

1.3 Presentation of the global environment

The figure 1.1 shows the data flows between files and program for the whole environment. Programs in bold face belongs to our distribution while programs in italic belongs to the Berkeley environment.

1.3.1 File formats

File formats are identified by the extension of the of each file:

- the `.mod` files are text files which contain the source for **mod2mag**. They follow the syntax of the **model** language. This document precisely describe the syntax and the semantic of this format..`mod` files can be generated by **ext2mod** from `.ext` files.
- the `.mag` files are layout files, which contains the graphical hierarchical description of the layout cells. This format is the magic(5) format. `.mag` files are generated by **mod2mag** and can be edited by **magic**.
- the `.ext` files are hierarchical netlist files. They describe under a hierarchical format the transistor netlist of a circuit. They are generated by **magic** and are use by **ext2mod** and **ext2sim**. This format is described in ext(5).
- the `.net` files are netlist for the **magic** router. It may be generated by **mod2mag**. This format is described in net(5).
- the `.sim` files are flat electrical netlist. This format is described in sim(5). These files are generated by **mod2mag** or by **ext2sim**.

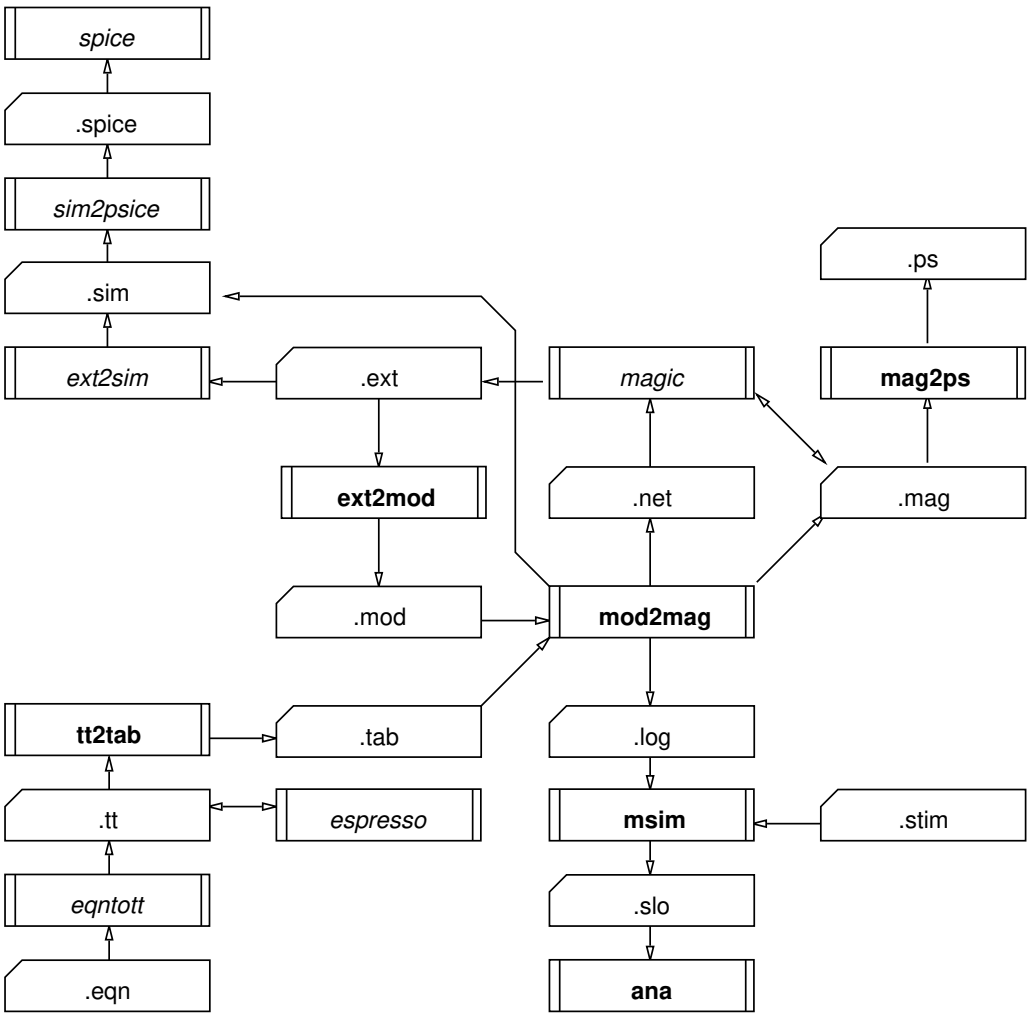


Figure 1.1: software environment

- the `.al` files are netlist alias files. This format is described in sim(5). These files are generated by **mod2mag** or by **ext2sim**. They describe the different names given to the same wires. They are associated with electrical flat netlists (`.sim`) or logical flat netlists (`.log`). These files are used by **msim** and **sim2spice**.

- the `.spice` files are electrical flat netlists for **spice**. These file are generated by **sim2spice**.

- the `.log` files are logical flat netlists. This format is described in log(5). These files are generated by **mod2mag**. They are read by **msim**.

- the `.beh` files are behavioral flat netlists. This format is described in msim(1). These files are generated by **mod2mag**. They are read by **msim**.

- the `.stim` files are stimuli for multilevel simulation. The format is described in msim(1). These are manually edited text files. They are read by **msim**.

- the `.slo` files hold the wave-forms generated by **msim** and displaid by **ana**. The format is described in slo(5).

- `.eqn` files hold the equation of PLAs. They are text files edited manually. This format is described in eqntott(1). They are converted into `.tt` format using **eqntott**.

- the `.tt` files are table description of PLAs. They are generated by `eqntott` from `.eqn` files. They may be optimized by the **espresso** program. The format is described in espresso(1).

- the `.tab` files are PLA files. They can be read by the `Read` command in **model**. It may be generated by **tt2tab** from the `.tt` files.

- the `.ps` files are the Postscript files. The program **mag2ps** converts `.mag` cells into printable Postscript files. **Ana** provide also `.ps` files to plot the wave-forms.

1.3.2 Programs

Our distribution consists of the following programs:

- **mod2mag** is the silicon compiler. It takes as input `.mod` files and `.tab` files and generates layouts (`.mag`), routing netlists for **magic** (`.net`), electrical netlist (`.sim` and `.al`) and multi-level simulation netlists (`.log`, `.beh` and `.al`). This document is dedicated to the description of this program. However a manual page is available (mod2mag(1)).

- **ext2mod** generates a **model** interface and a data sheet for every `.mag` cells. This program is documented in ext2mod(1).

- **msim** is the multi-level simulator. It as input takes a multi-level netlist as input(`.log`,`.beh` and `.al`) and stimuli (`.stim`). This simulators is documented in msim(1).

- **ana** is the wave-form analyser/**model**-debugger. This program is documented in ana(1). It reads `.slo` files and may produce Postscript files.

- **mag2ps** generates Postscript print out of layout cells. This program is documented in mag2ps(1). All the layout figures of this document have been generated by **mag2ps**.

- **tt2tab** converts `.tt` format to a format readable by this program.

The manual pages for all these programs and the new file formats are located in `~cad/ulm/man` and in the appendixF.

1.3.3 Organization of this document

This document follows this plan:

- the next chapter presents the syntax of the language **model**,
- the chapter 3 presents the simulation aspects,
- the chapter 4 presents the layout generation aspects,
- the first 3 appendices presents the specification of cells for two technologies and how to develop a new technology.
- the appendix D presents the data sheet of the library in the `mcmos` technology.
- the appendix E presents small, though non trivial circuits generated by **mod2mag**.
- the last appendix contains the manual pages of the programs and file formats of the distribution.

Chapter 2

Getting Started

This chapter is dedicated to the presentation of the syntax of **model**. The design of different levels of implementation of a *nand* gate will be an opportunity to illustrate the syntax of **model** and the associated semantic.

2.1 Preparing an input file

Files expressed in the **model** syntax possess usually the "`.mod`" extension. An input file is mainly made up of two kinds of statements:

- declarations and definitions of objects,
- instances of these objects.

The order of these declarations/definitions and instances is irrelevant, as long as objects are declared or defined before being used.

2.2 First lesson : the basic syntax

Table 2.1[1] shows the first implementation of our `nand` gate. At the first line, a library file is included. The `Include` statement is the priviliged way of sharing objects and/or libraries between different designs: it is possible to build and share libraries. These included files are read as if their contents take place directly in the calling file. As a single file for each library is included in various designs, the coherence between them is automatically ensured.

Here, the `magicdef.mod` file is assumed to contain the definitions of the `ntrans` and `ptrans` objects. Between line 3 to line 12, the `nand` gate is defined. At line 17, this gate is used.

2.2.1 Definition and instance of Parts

The main purpose of the **model**language is to provide a syntax for the description of pieces of hardware. Circuits are often described with a hierarchy of cells. Cells are built with electronic

[1] `lesson1.mod`

```
1   Include "magicdef.mod"
2       { description of a nand gate with two inputs in(0:1)
3   Part nand[in(0:1)] -> out
4
5       Signal middle
6
7       ntrans(1000,200)[in(0)] -> GND,middle
8       ntrans(1000,200)[in(1)] -> out,middle
9       ptrans(1500,200)[in(0)] -> Vdd,out
10      ptrans(1500,200)[in(1)] -> Vdd,out
11
12  End
13
14  Signal a,b
15  Signal output
16
17  nand[a,b] -> output
```

Table 2.1: the nand gate : first implementation

component like fet transistors and capacitors and with subcells. Every cell is called a *part* in the **model**terminology.

The definition of the `nand` gate looks like the definition of a function. It begins with the `Part` keyword and ends with the `End` keyword. Each enclosed statement belongs to the `nand` definition. The next word is the name of this part, followed the different parameters of this "function":

- an optional list of integer parameters, enclosed between parenthesis. An empty list is illegal and must be omitted;
- a list of input signals surrounded with square bracket "`[ ]`" (here the bus `in(0:1)`);
- a list of output signals (located after an arrow "`->`"). When the output list is empty, the arrow must be suppressed

Mod2mag provides electrical rule checking which is not based on the distinction between inputs and outputs in a part declaration. (the Electrical Rule Checking mechanism is described in chapter 3). This distinction exists only for legibility purpose. Anyway, we must handle signals which are not really well-typed. For instance, the source and the drain of a FET transistor are neither input nor output. By convention, the irresolute input/output signals take place into the part output list, as shown in our first example for the instances of `ntrans` and `ptrans`, which are FET transistors.

These instances show examples of the use of integer parameters (which represent here the width and the length of the transistor channel in hundredths of micron).

2.2.2 Signals and buses

Wires are required for connecting the different parts together. A wire may be known under several names. These names are called *signals* in the **model** syntax. They must be declared before being used. The `Signal` keyword declares signals in the current context:

- local signals in the body of a part declaration (e.g. `middle` at line 5),
- global signals in the top level of the file (e.g. `a`, `b` and `output` at lines 14 and 15)

Parameter signals are implicitly declared in the header of a part definition: so the signals `in(0)`, `in(1)` and `out` are declared in the current part by `in(0:1)` and by `out` in the header of the part definition (at line 3).

An easy way to declare a list of signals is to declare a bus which is an array of signals. An index mechanism allows to extract signals from a bus. The domain of this index must be specified at the declaration of the bus with its limits separated with column (`:`) and enclosed with parenthesis. So,

```
Signal data(0:4)
```

declares a bus consisting of 5 signals: `data(0)`, `data(1)`, `data(2)`, `data(3)` and `data(4)`. A given identifier can take place in only one signal declaration in a given context. For instance, it is illegal to write:

```
Signal data,data(0:4),data(7:11)
```

When using a bus, it may be useful to extract a subbus from this bus, or to build a bus from signals and subbuses.

2.2.3 Construction of signal lists

Different constructors are available to build list of signals:

- the most simple operator is the signal list concatenation. This operator is the comma (","). If `a`, `b` and `c` are signals or buses, then `a, b, c` is a list of signals. The first way to list every signal of the `data` bus is to write directly:

  ```
  data(0), data(1), data(2), data(3), data(4)
  ```

- It is possible to share in a list the identical bus name:

  `data(2),data(1)` ⇔ `data(2,1)`

  It is possible to apply a list of integers to a bus as long as every integer belongs to the range of the bus: it results a signal list. By this mechanism, we can use every integer list constructor for signals list. The second way to list every signals of the `data` bus is to write:

  ```
  data(0,1,2,3,4)
  ```

- It is possible to specify a regular enumeration of integers with the colon operator (":"). `1:5` is a list of those five integers `1, 2, 3, 4, 5`. We may also specify a step. `6 : 1 By -2` is a list of the three integers `6, 4, 2`. The third way to list every signal of the `data` bus is to write:

  ```
  data(0:4)
  ```

- Sometimes, every signals of a bus must be listed in the natural order. The name of the bus without any parenthesis is enough to describe the whole signal list. The most simple

type	*name*	*tension*
ground	`Ground`	0V
	`Gnd`	0V
	`Earth`	0V
	`Zero`	0V
	`Vss`	0V
power	`Power`	5V
	`Vdd`	5V
	`Vcc`	5V
	`One`	5V

Table 2.2: power-supplies special signals

way to list every signals of the `data` bus is:

```
data
```

2.2.4 Special signals: the power-supplies

Power supplies signals are special global signals in **model**. These global signals are known by the software. Their names are keywords. There exist 2 power supplies: the ground (0V) and Vdd (5V). the table 2.2 shows the multiple names given for each power-supplies.

When required, it is possible to built power-supplies buses : `Vdd(1:4)`.

2.2.5 Keywords

We have yet discovered some keywords: `Part`, `End`, `Signal`, `Vdd` and `GND`. A word beginning with an upper-case letter must be a keyword. The rest of the word is not case-sensitive : `Part` and `PART` are equivalent.

2.2.6 Identifiers

A word beginning with a lower-case letter is an identifier. Unlike keywords, the identifiers are case-sensitive: `clock` and `clocK` are not equivalent. Identifier names are made up of characters (with no limitation on the number) chosen among letters, digits and the underscore character ("_"), with the classical convention that the first character is not a digit.

2.2.7 Various lexical elements

Instructions are delimited by one or more carriage returns (<CR>). The carriage return is meaningless after a comma (`,`), an arrow (`->`), or one of these keywords `Then`, `Else`, `And` and `Or`. It is possible to cut a too long line with the character minus (`-`) placed just at the end of the line.

Comments start with a curly bracket ({) and stop at the end of the line.

Strings are surrounded by double quotes (`"`). Double quotes, carriage returns, tabulations, backslashes are expressed in a string by `\"`,`\n`,`\t` and `\\`.

2.2.8 Compiling a model file

Configuration

In order to compile the examples of this manual, you need to add the directory `~cad/bin` to your environment variable `PATH`. It is also better to create your own directory, where to run the examples. You have two files to copy in this directory:

- `~cad/ulm/mcmos/example/.magic`
- `~cad/ulm/mcmos/example/lesson1.mod`

The `.magic` file specifies where to find the layout cells (`.mag`) and the **model** files (`.mod`). This file contains **magic** orders, which are executed at the beginning of a **magic** session (see magic(1)). It is necessary to specify where to find the cells thanks to the `path` command. This file is also interpreted by **mod2mag** only to read eventual `path` command. Programs (**magic** and **mod2mag**) successively read the optional files:

- `~cad/lib/magic/sys/.magic`,
- `~/.magic`,
- `./.magic`.

The last `path` command defines where to find the cells and the **model** files. Our `.magic` file contains the command:

```
path ":.:~cad/ulm/mcmos/example:~cad/ulm/mcmos/lib8:~cad/ulm/mcmos/pad:~cad/ulm/mcmos/pla:"
```

which indicates that **model** files and layout cells are successively searched in :

- `.` the current directory,
- `~cad/ulm/mcmos/example` the example directory,
- `~cad/ulm/mcmos/lib8` the cell library,
- `~cad/ulm/mcmos/pad` the pad library,
- `~cad/ulm/mcmos/pla` the pla library,

As the `example` directory takes place is the `path` command, you need not to copy the `lesson1.mod` file, but it is better for you to have your own copy of the file if you want to modify it.

Running the compiler

To compile the example, run the following command:

```
model lesson1.mod
```

or

```
model lesson1
```

The compiler generates one of those 2 files:

- `lesson1.err` which contains the errors of the file (if any).

```
1   Include "magicdef.mod"
2
3   Part nand(n)[in(0:n-1)] -> out
4
5       Signal middle(0:n)
6       Integer i
7
8       If n > 10 Then
9           Error "too many inputs : ",n," in the nand gate"
10      Endif
11
12      Message "warning nand: ",n," inputs > 5" If n > 5
13
14      middle(0) -> GND
15      For i=0:n-1 Cycle
16          ntrans(1000*n,200)[in(i)] -> middle(i),middle(i+1)
17          ptrans(2000,200)[in(i)] -> Vdd, out
18      Repeat
19      middle(n) -> out
20
21  End
22
23  Constant size = 4
24
25  Signal input(1:size)
26  Signal output
27
28  nand(size)[input] -> output
```

Table 2.3: the nand gate : second implementation

- `lesson1.out` which is the interpreted output of the compiler, if the design description is coherent (if not correct).

When **mod2mag** is called without any flags, only the correction of the input file is checked. We will discuss in the next two chapters the differents that can be set and the different output files generated. In this chapter, we pay attention only the basical syntax of **model**.

2.3 Second lesson : more about the basic syntax

Let us write now a more general *nand* (table 2.3[2]). We want to express in a single part definition how to build `nand`s with 2, 3, 4 and so on inputs. We need integer identifiers, arithmetic operators and parameterized calls. This example is the opportunity to discuss the following points:

[2] file `lesson2.mod`

- Declaration and use of integers,
- Flow control,
- Debugging messages.

2.3.1 Integers

Integers are used in **model** as in any programming language. They can have an integer value assigned to, together with integer constants in arithmetical/logical expressions. Last but not least, they can be passed by value as parameter to functions.

Integer identifiers

Integer identifiers have the same lexical form as the signals identifiers. It is legal for a signal and a integer to have the same name [3]. Arrays of integers can be declared and used as arrays of signals (= buses).

Integer lists

Integer lists can be constructed using the same syntactical constructs as the list of signals:

```
Integer i(0:6),j,k
```

successively declares an array of the 7 integers `i(0)`,...`i(6)` and 2 integers `j` and `k`. The instances of integers is identical to signals ones. `i(5:1 By -2)` means the integer list `i(5),i(3),i(1)`.

In the same manner as a signal, an integer identifier can be either local to a part (i.e. declared between the `Part` header and the corresponding `End` statement) or global (.i.e declared outside of any definition of a part).

Integer initialization

Each global variable is initialized at the null value, unless another value is given in the command line of the **model** program. The line 23 of our example could be changed by:

```
Integer size
size = 4 If size = 0
```

By default, `size` gets the 0 value at its declaration, and is then modified to get the 4 value. If we compile this example with the **model** command:

```
model size=3 lesson2.mod [other options]
```

`size` is initialized to 3. This mechanism is convenient to parameterize the compilation without modifying the **model** source.

Constant integer identifiers

A convenient way to use numerical constants is illustrated at line 23 of the example 2.3. The corresponding identifiers ("`size`" in the example tab. 2.3) cannot be modified after its declaration.

[3] This should be avoided for legibility!

operator	*arity*	*function*
+	2	addition
-	2	subtraction
-	1	unary minus
*	2	multiplication
/	2	euclidean division
%	2	modulo
&	2	logical and
!	2	logical or
!!	2	exclusive or
<<	2	shift left
>>	2	arithmetical shift right
\	1	to 1's complement
^	2	exponentiation

Table 2.4: the integer operators and their arities

Numerical constants

Numerical constants are expressed by default in decimal base or in any other base from 2 to 36. If the base is not decimal, the value of the base in decimal is concatenated to left of the numerical constant with an underscore (_) as separator. When the base is greater than 10, the missing "digit" to build the number are taken from the alphabet (case-unsensitive). So 'a' represents 10, 'b' represents 11 and so on until 'z'. For example `123, 16_C, 2_1101` represent the decimal values `123, 12, 13`.

Integer expressions

Integer expressions are built like in programming languages. The table 2.4 presents the different integer operators with their arities.

The precedence of operators is processed among this order:

1. - (unary) and \,
2. ^,
3. *, / and %,
4. + and -,
5. << and >>,
6. &,
7. ! and !!.

Parenthesis are used to force the precedence. Each binary operator is left associative.

operators	*function*
=	equal
#	different
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

Table 2.5: comparator functionality

2.3.2 Flow control

Flow control is useful to express the construction algorithms of parts. There are 6 kinds of flow controls:

1. conditional control `If`,
2. endless loop control `Cycle`.
3. enumeration control `For`,
4. conditional ended loop control `While`,
5. conditional ended loop control `Until`,
6. case switch control `Switch`.

`Exit` and `Continue` are used to modify the execution of the flow. Some of these flow controls need conditions. They are expressed by boolean expressions.

Boolean expressions

Basic booleans are comparisons between integer expressions. Then, they can be made up of boolean operators to build more complex boolean expressions.

Comparators A comparator admits two integer arguments as input and returns a boolean. The table 2.5 presents these different comparators with their respective functionality.

Boolean operators (`And`, `Or` and `Not`).
The operator precedence follows this order:

- `Not`,
- `And`,
- `Or`.

If : conditional execution of statements

The complete syntax of the `If` statement is:

```
If condition Then
        statement_list_1
Else
        statement_list_2
Endif
```

The boolean expression is first evaluated. If the value is true, the *statement_list_1* is executed and *statement_list_2* is skipped. Otherwise *statement_list_1* is skipped and *statement_list_2* is executed. If the second statement list is empty, the `Else` keyword is not necessary. The example 2.3 shows this construction between lines 8–10.

This heavy construction can be shortened when the first statement body has only one statement and when the second statement body is empty:

```
statement If condition
```

Program 2.3 shows an example this construction at line 12.

Iteration statements

There is a general construction to express the body of some loop:

```
Cycle
        statements
Repeat
```

In the body statement, the `Continue` statement jumps to the next iteration of the loop and the `Exit` statement jumps out of the loop.

The iteration statement described above will never end unless an exit statement is executed. Apart from explicitly programming an exit, there are 3 ways of controlling loops.

- The `For` statement will repeat the execution of the loop body while the control variable is successively assigned the values of a list of integers. The loop is terminated when the list is exhausted. Syntax:

  ```
  For variable = integer list Cycle
          statements
  Repeat
  ```

- The `While` statement evaluates a boolean condition to continue the iteration:

  ```
  While condition Cycle
          statements
  Repeat
  ```

  The condition is first evaluated. Then if the boolean value wass true, the body of the loop is executed[4].

- The `Until` statement also uses a boolean to control the loop. Unlike the `While` statement, the loop body is first executed[5]. Then the condition is evaluated. The body of the loop is executed once more if the boolean is false. Syntax:

  ```
  Cycle
          statements
  ```

[4] It may be executed 0 times.

[5] It will be executed at least once.

`Repeat Until` *condition*

`Exit` and `Continue` are still available with these 3 ways of controlling loops.

Case statements

The `Switch` statement allows to choose between multiple possibilities

For : enumeration control

The `For` statement is classically used in progamming languages to execute a statement body for each elements of a enumeration of an index.

A list of values for the variable is provided, instead of the range of variation:

`For` *variable* `=` *integer list* `Cycle`
statements

Statements are evaluated for each element of the list. At each evaluation *variable* is set to the value of the next integer of the list. The content of the list is evaluated before entering the loop. This mechanism is more flexible thanks to the lists construction. In our example 2.3 lines 15-18, a classical range is used with to the enumeration (`:`) operator for list construction.

As for the `If` statement, it is possible to shorten the notation, when a single statement is involved in the body of the `For` loop:

statement `For` *variable* `=` *integer list*

The `For` loop execution can also be modified by the `Continue` and `Exit` operators.

While : conditional ended loop control

The `Cycle` loop can be legibly written with either the `While` or `Until` constructions. It indicates when to stop the execution of the loop with a boolean expression. The boolean can be evaluated before (`While`) or after (`Until`) the body of statements.

The `While` statement syntax:

`While` *condition* `Cycle`
statements
`Repeat`

The condition is evaluated at every iteration of the statement body. If the test is passed, the execution continues inside the loop, otherwise the execution resumes after the end of loop. The `For` construction between line 15 and 18 can be rewritten:

```
i = 0
While i < n Cycle
    ntrans(1000*n,200)[in(i)] -> middle(i),middle(i+1)
    ptrans(2000,200)[in(i)] -> Vdd, out
    i = i+1
Repeat
```

The `Until` statement syntax:

`Cycle`
statements
`Repeat Until` *condition*

The condition is evaluated after every evaluation of the body statements. If the test is passed, the execution continues to the next statement after the end of loop, otherwise jumps back to the start of the loop.

The `While` and `Until` loops execution can be also modified with the `Continue` and `Exit` operators.

Switch : case switch control

The `Switch` statement is useful to choose a sequence of statements between several sequences depending on the value of an expression. The structure of the `Switch` statement is:

```
Switch expression
        Case expression1
                statement list 1
        Case expression2
                statement list 2
        ...
        Default
                statement list n
End
```

It roughly works as the C `switch`. The *expression* is evaluated at the entrance of the `switch` statement. For each `Case` statement the calculated value of *expression* is compared to the evaluation of the associated expression (*expression1*, *expression2*,...). At the first time the expressions are equal, the statement lists following the `Case` statement are executed until the first `Exit` execution. The statements following the `Default` keywords (if it exists) are executed if no equality has been formerly found.

2.3.3 Signals merging

As it can be observed in line 14 and 19 the arrow operator (`->`) is used to connect signals. Different names can refer to the same wire). This connection can merge directly a signal list (possibly of one element) to another one of same size. The different names for an equipotential are equivalent: the arrow operator is commutative (unlike the assignment operator).

Instance mechanism

The expression `ptrans(2000,200)[in(i)]` (line 17) defines an instance of `ptrans` with the integer parameters 2000 and 200, with `in(i)` as an input bus and returns the list of the output signals of the instance which must be connected. The arrow operator completes this task by connecting the corresponding signals to `Vdd` and `out`. It is only a legibility convention to put the instance at the left of the arrow. When a instance has no output, the right side of the arrow is empty. In this case the arrow must suppressed.

Unconnected signals

Sometimes, some of the outputs are unused. Because of the instance mechanism, the designer must specify a signal for each output. It is tedious to define ghost signals for each unconnected output: the operator "unconnected" `--` allows us to define an unnamed unconnected signal. It

acts like a dummy symbol generator. Because many signals may be unconnected in a signal list, it is possible to define an unconnected bus: `--(0:7)` is a bus of 8 unconnected signals.

2.3.4 Debugging messages

Model provides messages for debugging facilities. There are two kinds of print statements:

- `Message/Messagef` : print and continue compilation,
- `Error/Errorf` : print and abort compilation,

These statements have the following syntax:

`Message` *list of strings and integer value*
`Messagef` *"string format" list of strings and integers*
`Error` *list of strings and integer value*
`Errorf` *"string format" list of strings and integers*

In the `Message/Error` statements, strings are printed literally and integers are converted to their decimal representations. For the `Messagef/Errorf` statements, the first parameter is a format string. The syntax of this format is the same as C's (see printf(3)).

2.4 Third lesson : last refinement about the syntax

To end our fast sight of the **model** syntax, let us consider our third implementation of the nand gate (table 2.6[6]).

```
1   Include "magicdef.mod"
2
3   Part basic_nand[in(1:*)] -> out
4       Integer n,i
5       n = Length[in]
6       Signal middle(1:n+1)
7
8       middle(1) -> GND
9       For i=1:n Cycle
10          ntrans(1000*n,200)[in(i)] -> middle(i),middle(i+1)
11          ptrans(2000,200)[in(i)] -> Vdd, out
12      Repeat
13      middle(n+1) -> out
14
15  End
16
17  Part basic_nor[in(1:*)] -> out
18      Integer n,i
19      n = Length[in]
20      Signal middle(1:n+1)
21
22      middle(1) -> Vdd
23      For i=1:n Cycle
24          ptrans(2000*n,200)[in(i)] -> middle(i),middle(i+1)
25          ntrans(1000,200)[in(i)] -> GND, out
26      Repeat
27      middle(n+1) -> out
28  End
```

[6] in file `lesson3.mod`

```
29
30  Part basic_or[in(1:*)] -> out
31      basic_nor[basic_nor[in]] -> out
32  End
33
34  Part nand[in(1:*)] -> out
35  Integer n,p,j,first,last
36  n = Length[in]
37  Signal middle(1:3)
38
39  If n < 5 Then        { recursion terminaison }
40      basic_nand[in] -> out
41  Else
42      p = Sqrt(n)
43      p = 3 If p > 3
44      first = 1
45      For j=1:p Cycle
46          last = first + (n -1)/p
47          last = n If last > n
48          nand[in(first:last)] -> middle(j)
49          first = last+1
50      Repeat
51      basic_or[middle(1:p)] -> out
52  Endif
53  End
```

Table 2.6: the nand gate : third implementation

This example provides the opportunity to study the next features:

- use of * as implicit parameter,
- built-in functions and procedures,
- recursivity.

2.4.1 Use of star (*) as implicit parameter

At line 3 of the last example, a star (*) appears in the input signal list of a part definition. This star represents a number which is in fact a parameter of the part. The value of the star is dynamically calculated when the part is instanced, so that the length of the list of input formal parameters matches the length of the actual input parameters.

The value of the star is an implicit parameter of the part. It is necessary to get the size of the bus containing the star. The `Length` operator achieves this task. It takes as input a list of signals (typically the bus containing the star) and yields an integer which is the length of the list:

```
Length[a,b(1,3,5),c(0:2),d]
```

returns 8. If an input bus is declared as in tab. 2.6 line3, the length operator applied to the bus will give the value of the *[7].

[7] If the bus is declared `b(0:*)`, `Length[b]` will yield the width of the bus, which is the value of the star plus 1.

2.4.2 Built-in functions and procedures

Built-in functions and procedures (like `Length`) are provided to deal with parameters, to compute maths functions and to read files.

2.4.3 `Log` function

The `Log` function is convenient to calculate the number of buffering stages of an inverter. The `Log` function takes a single parameter p and returns the smallest integer n such that $2^n \geq p$. For instance:

```
Log(15) = 4
Log(16) = 4
Log(17) = 5
```

It is used at line 68 of table 4.4 page 50 for a multistage inverter generator.

2.4.4 `Sqrt` function

The `Sqrt` function is convenient to develop an amplification tree of depth 2 with load repartition. The `Log` function takes a single parameter p and returns the smallest integer n such that $n^2 \geq p$. For instance:

```
Sqrt(3) = 2
Sqrt(4) = 2
Sqrt(5) = 3
```

It is used at line 42 of table 2.6.

2.4.5 Input functions

In order to import a large number of parameters into **model**, which could not be easily expressed through the **model** syntax, file reading mechanisms are provided. As more than one files can be read simultaneously, each opened file is identified by a unique integer number. This number (from 0 to 19) is returned by the `Open` function which takes one parameter: the name of the file.

descriptor `= Open "`*filename*`"`

The `Read` function reads the next integer in the file pointed by a descriptor. `Read` takes one integer parameter (the file descriptor) and returns the value of the next integer read from the file. It is assumed that integers are written in decimal, one by line. A line in the file which begins with a star is treated as a comment and is skipped. If the line start with a star, it is skipped (because it is a comment).

The `Close` procedure releases the file descriptor in order to be used again. It admits the descriptor as unique parameter.

2.4.6 Recursivity

Model parts can be recursive. The recursion is always based on integer parameters. In the example presented in table 2.6, the `nand` part is recursive (on the value of the hidden parameter star). Recursivity is essential to express easily complex structures like trees.

2.4.7 How to debug complex files?

The integer parameters and the recursivity allow to develop very powerful objects. The price of this power is the complexity of the debug. We require some help to debug easily **model** files.

When a **model** file has been created, it must be checked. The command:

`model` *file*

compiles it. Without any flags, the software only checks the input file and evaluates it.

The compilation for **model** consist in fact in the evaluation of integer expressions. Compilation time errors are of two types:

- syntax errors,
- evaluation errors.

The compilation process has two passes.

The first one parses the **model** input file and checks the syntax. An internal representation of each object is then built. In this representation, no evaluation is done. In the first pass, the symbols are checked (declaration before instance). These errors are easy to correct because error messages are explicit and point out the implied lines.

At the second pass, this representation is interpreted. At this stage, errors may occur:

- arithmetical errors: divide by 0, negative arguments for `Sqrt` and `Log`.
- range errors: overflow and underflow of arrays and buses,
- mismatches in signal list connections: wrong number of input signals, bus size mismatch in bus connections.
- non-existence in the upper contexts of inherited signals.

These errors are also easy to correct. Using `Message`, it is possible to trace integer values and to detect what is wrong within the input file.

When the file is apparently correct (there is no more compilation time errors), the file can be observed as correct for the point of view of the software: the input file is consistent; it is correct from the compiler view point. There is no more "`.err`" file and a "`.out`" file is created.

The "`.out`" file contains the interpreted view of the input design in model syntax. No integer and no flow control structure subsist. This view of the design conserves its hierarchical structure. It is the reference of the exact structure of the design.

2.5 Fourth lesson : scope of signals and integers

Chapter 3

Description for the simulation

One of the purposes of an Hardware Description Language like **model**, is to describe the electrical interconnection of a circuit. The language must contain the information required by the software to make some electrical verifications (**E**lectrical **R**ule **C**hecking) and to extract the following different data:

- transistor simulation netlist,
- gate and behavioral simulation netlist,
- routing netlist for chip router
- routing netlist for PCB router.

The last two points concern hardware generation and will be discussed in the next chapter.

The electrical structure of a design is specified with two mechanisms:

- the definition the electrical (and logical) behavior of each basic cells (the leaves of the hierarchy),
- the interconnection of these cells in blocks (the nodes of the hierarchy).

From the designer point of view, programming in **model** consists in taking basic cells from a library to build more complex structures. According to classical design methodologies (derived from standard-cell systems), the designer does not build libraries. In fact, this task is difficult because a layout, an Electrical Rule Checking interface, a transistor level description and a gate level description must be provided for each cell. One purpose of **mod2mag** is to allow the designer to build himself his own cells and libraries. That is why we describe how to specify the electrical structure of imported cells (corresponding to existing layouts). A tool (**ext2mod**) is provided for the definition of this electrical/logical structure. We also describe how to specify a behavioral model in order to follow a top-down approach in the design methodology.

In this chapter, we first explain the Electrical Rule Checking mechanism. Then we describe the generation of the simulation netlists and continue with the description of the behavioral models. We finish with the presentation of **ext2mod**, the tool which automatically derives the electrical structure of a cell from its layout. The description of the generation of router netlists will be described in the next chapter because of the topological aspects of layout.

```
1   Part nand2[i(0:1)] -> out
2       Source i
3       Dest out
4
5       ...
6
7   End
8
9   Part andor[a(0:1),b(0:1)] -> out
10      Signal left,right
11
12      nand2[a] -> left
13      nand2[left,right] -> out
14      nand2[b] -> right
15  End
16
17  Signal select, selectbar, d(0:1), outmux
18  Source select,d(0:1)
19
20  nand2[select,VDD] -> selectbar
21  andor[d(0),selectbar,d(1),select] -> outmux
```

Table 3.1: the nand gate : the electrical specification of the IOs of the nand cell

3.1 Electrical Rule Checking mechanism

The Electrical Rule Checking mechanism checks the consistency of the connections: it detects the non-polarized inputs and the short-circuits.

This mechanism is based on the type-check of the input/output signals of the cells. Each of them cell may be declared as input or output or tristate signal.

The designer may specify these types for some cells (generally the basic ones) and let the system automatically determine the types of the IO signals of composed cells by analysing the already typed subcells.

From a theoretical point of view, only the leaves of the hierarchy need to be typed. Practically, it is better to introduce this typing mechanism at the gate level. At transistor level, source and drain of FET transistors can only be classified as tristate signals. The type of the output of a non-tristate classical gate would be found *tristate*, as the union of 2 tri-state signals. Strictly speaking, this is not false. However, this not accurate enough for the ERC to detect a short-circuit between 2 non-tristate outputs. So the best place to define the type of the IO signals is at gate-level.

The table 3.1[1] shows an example of the ERC interface of the nand gate.

The `Source`, `Dest` and `Tristate` statements follow the same syntax:

`Source` *IO-signal list*

[1] The file `erc.mod`

The `Source` statement (line 2) specifies that each signal of the list is an input of the cell and must be polarized (i.e. connected to a single output or to one or more tristate signals).

The `Dest` statement (line 3) specifies that each signal of the list is an output (it may be connected only to input signals).

The `Tristate` statement specifies that each signal of the list is a tristate signal (it is sometimes a low impedance IO and can polarize input; it can be connected to other tristate signals in order to build multi-speaker buses and is not allowed to be connected to classical outputs).

Power-supplies are considered as output signals.

The check is performed at each level of hierarchy and the reporting message for each error precisely indicates the implied level of hierarchy.

At the top level of the hierarchy, the global inputs of the system are not polarized. They induce a warning report. In order to avoid this, it is possible to declare an external polarization of these global signals by a `Source` statement (line 18).

The Check is performed at the generation of simulation netlists (see below).

3.2 Generation of the simulation netlists

The generation of the transistor netlist is launched by the command:

`model -s` *design*

The flat netlist is written into the *design*`.sim` file. The format of this file is described in sim(5). The gate netlist is obtained in the *design*`.log` file by:

`model -l` *file*

The format (`.log`) of this file is described in log(5).

For both commands, an alias file *design*`.al` is also generated. Its holds the synonyms for the signals.

The generation of the transistor netlist is controlled by the `Output/Outputf` statement, which writes into the *design*.sim file. The syntax of this statement is the same as the syntax of the `Message/Messagef` statement:

`Output` *list of strings and integers*

or

`Outputf` *"format string" list of strings and integers*

The generation of the gate netlist is controlled by the `Logic/Logicf` statement, which writes into the *design*.log file. The syntax of this statement is the same as the syntax of the `Message/Messagef` statement:

`Logic` *list of strings and integers*

or

`Logicf` *"format string" list of strings and integers*

These commands (`Output/Outputf` and `Logic/Logicf`) are very similar; their only functional difference lies in the command line flags which control them. However, the `Output/Outputf` command will usually be inserted in the definition of transistors while the `Logic/Logicf` command will lie in the definition of gates.

The table 3.2 shows a part of the file `magicdef.mod` (which is located in the directory `~cad/ulm/mcmos/lib8`).

A signal may be known under various names; however, in the transistor and gate netlists, all the names of a given signal must be changed into an unique global name for this signal. This is done by the `Net` function, which takes a list as input and returns a list of strings representing the unique names of each signal of the list.

```
1   Part ntrans(w,l)[g] -> s,d
2   Output "n ",Net[g],Net[s],Net[d],l/50," " ,w/50," 0 0"
3   End
4
5   Part ptrans(w,l)[g] -> s,d
6   Output "p ",Net[g],Net[s],Net[d],l/50," " ,w/50," 0 0"
7   End
```

Table 3.2: definitions of `nmos` and `pmos`

At each instance of the *nmos* part, the netlister writes in the netlist file the line:

`n` *gate source drain channel-length channel-width* 0 0

which complies to the format of the transistor netlist.

3.3 Behavioral cell description

3.3.1 The aim of behavioral modeling

Behavioral modeling is necessary:

- to follow a top-down methodology. First develop the global behavior of the circuit, and step by step refine each block in sub-blocks until the gate level (which are basic cells).
- the modeling of external devices (for instance, existing integrated circuits, buses, memories,...).

Behavioral modeling is the way to explain how to simulate a circuit. The behavioral description capabilities of model is compatible with the **msim** simulator (see msim(1)). It is a multimode simulator which operates at the following level :

- switch level for pass transistors.
- gate level,
- behavioral level.

During the generation of the logical netlist (the `-l` option), a description of switches and gates is written into the `.log` by the `Logic/Logicf` statements, just described above. Behavioral models are written in the `.beh` file (see msim(1)). These two files are read by **msim** in order to perform the simulation.

The table 3.3[2] shows an example of the model for a *nand* gate. Even with a behavioral model, it is necessary to specify the type of the inputs/outputs (line 2 and 3).

The behavioral description is a list of special statements enclosed by the `Behavior` and `End` keywords (line 5 to 17).

[2] file `behavior.mod`

```
1   Part nand[a,b] -> c
2       Dest c
3       Source a,b
4
5       Behavior
6           Capa 100 a,b,c
7           Slew 100 c
8           Variable va,vb,vc
9
10          Undef c
11          When a,b Change Do
12              va = Value[a]
13              vb = Value[b]
14              At Time + 1 NS Do
15                  vc = Eval(\(va & vb))
16                  Set c = vc
17              Done
18          Done
19      End
20  End
21
22  Signal a,b,c,d
23  nand[a,b] -> c
24  nand[c,c] -> d
25  Behavior
26      Capa 1000 c
27  End
```

Table 3.3: behavioral model of a *nand*

3.3.2 Organization of the behavioral modeling

The modeling of the behavior of a component consists of the following operations:

- the input states are read (line 12-13),
- computing a new state from the inputs states and, possibly, internal states (line 15),
- modifying of the internal state (line 15),
- modifying of the outputs (line 16),
- modeling the impedance of the inputs/outputs (line 6-7).

Flow control can be used to control the execution of the behavioral model during the simulation. This flow control is extended to specify:

- when a execution block must be awaken (line 11);
- the delay of the execution of a block (line 14).

Behavioral description requires the manipulation of expressions and the assignment of variables.

3.3.3 Variables

Behavioral simulation most of the time requires internal variables. These variables are declared in the `Behavior`...`End` block by the statement:

`Variable` *list of identifiers*

They can take any value between -2^31 and 2^31-1 plus a special value `Undef`, which stands for *undefined*.

The difference between the integers and the variables lies in the step in which they are used: the integers are used during the compilation while the variables are used during the simulation.

It is possible to declare an array of variables.

3.3.4 Signal value acquisition

The `Value` function converts a list of signals into a 32 bit integer value. The argument is the list of signals:

```
Variable v
v = Value[inputbus]
```

- If one of the signals is `HIGHZ` or `UNDEF`, v is assigned the `Undef` value.
- Otherwise, the list of signals is transformed into a list of bits by changing `HIGH` into 1 and `LOW` into 0. The list of bits is padded by 0s at the left to 32 bits if necessary. Finally the 32 least significant bits form the integer value which is returned. The first signal corresponds to the least significant bit of the result.

3.3.5 Behavioral expressions

Behavioral expressions consist of basic expression items and operators. The basic expression items are:

- numerical constants, possibly with a time unit `PS`, `NS`, `US` or `MS` (default is `PS`), a capacitance unit `FF`, `PF`,`NF` or `UF` (default is `FF`) or a slew rate unit `PS/FF`, `NS/FF`,... (default is `NS/PF`),
- integers (declared with `Integer` which have been changed into numerical constants),
- variables (declared with `Variable` which remain variable at the simulation),
- `Time` the current date during the simulation,
- utility functions,
- `Eval` function, used to evaluate early an expression (this function is useful with the schedule operator `At` and is described at this occasion;
- `Value` operator, already described in the previous subsection.

The behavioral expression operators are the same as the integer ones. The comparators has been added to these operators. They return either 0 if the comparison is true or 1 if not. The evaluation is the same. If one of the operands is undefined (set to `Undef`) then the results will be undefined (`Undef` is returned).

Utility functions

The utility functions like the variables are evaluated during the simulation. 5 functions are available:

- `Random` requires no argument and returns a different 32-bit value at each call (during the simulation). This function is useful to define random actions on external buses;
- `Log` takes a single parameter and returns the smallest integer n so that $2^n \geq p$. It works like the *integer* function `Log` but is evaluated during the simulation;
- `Sqrt` takes a single parameter and returns the smallest integer n so that $n^2 \geq p$;
- `Open` *("filename")*; this function opens the file *filename* in reading mode and returns a file descriptor. This action is performed at the simulation.
- `Read` *(fdesc)*; this function reads the next integer in the file pointed by the *fdesc* descriptor. It is assumed that integers are written in decimal, one by line. If a line in the file begins with a '*', it is treated as a comment and skipped. `Open` and `Read` are useful for instance for the initialization of of a ROM.

An utility procedure can be added to these function. The `Close` takes a single parameter and closes the file associated with this parameter.

3.3.6 The modification of signals

When the behavior has been calculated, the simulator must diffuse the result in the output signals of the part. This can be performed using the following statements:

1. `Undef` *signal list*. The value of these signals are set to undefined state (example on line 10).
2. `HighZ` *signal list*. The value of these signals are set to high impedance state.
3. `Low` *signal list*. The value of these signals are set to the low level ($0V$).
4. `High` *signal list*. The value of these signals are set to the high level ($5V$).

It is also possible to change the state of a bus according to the binary value of a variable:
There is another way to change the value of a bus:

`Set` *signal list* `=` *expression*

If the expression value is `Undef`, every signal of the list is set to `Undef`. If the expression has an integer value v, the n^{th} signal is set to high (resp. low) if the n^{th} bit of the binary representation of v is 1 (resp. 0). The least significant bit of v corresponds to the first signal of the list.

3.3.7 The modeling of the inputs/outputs

If nothing else is specified, inputs and outputs of the part are supposed to be perfect:

- no parasitical capacitances on inputs/outputs,
- null output impedance.

It is possible to add a capacitance to a signal or a signal list with the statement:

`Capa` *capacitance signal list*

The capacitance is added for every signal of the *signal list*.

It is possible to specify an impedance for each signal statements. For instance,

`Slew` *impedancesignal list*

associates the *signal list* a delay equal to the product of the capacitance of each node with *impedance*. The value of the impedance is expressed in pS/pF.

3.3.8 The modeling of a multitalker bus

We must describe what happens when 2 different modifications of state are applied to the same signal. These modification are never applied simultaneously because of the sequential organization of the simulator. Two desired effects are possible:

- The two modifications belong to the same object. The new state modification must override the old state. For instance, a clock generator sets and resets the clock at each period.
- The two modifications belong to different objects. These objects will be in conflict on the output.

In order to offer naturally the 2 possibilities to the designer, each signal modification statements is associated with the current part instance. Each modification statement changes the state of the output pole associated with the current part instance. Modification statements associated with different part instances modify the states of different output poles. Each output pole has a current state and an impedance. Every Output poles corresponding to the same signal will be used by the simulator to compute the state of the signal, as it is done for electrical gates.

3.3.9 Debugging functions

For debugging purpose at simulation time, it is possible to print a message with the commands `Print` and `Printf`. They work like `Message` and `Messagef`.

3.3.10 Scheduling flow control

Two new statements are used to describe the behavior of a circuit:

- `When` instructs the simulator to execute a statement list on every specified transition on some signals,
- `At` postpones the execution of some statements.

These features are not easy to understand because of the parallelism they may hide. The next example describes the behavior of a Read-Only-Memory.

```
1   Part tms2732A_17[a(0:11),eb,gb] -> d(0:7)
2
3   Source a,eb,gb
4   TriState d
5   Signal m(0:7),on
```

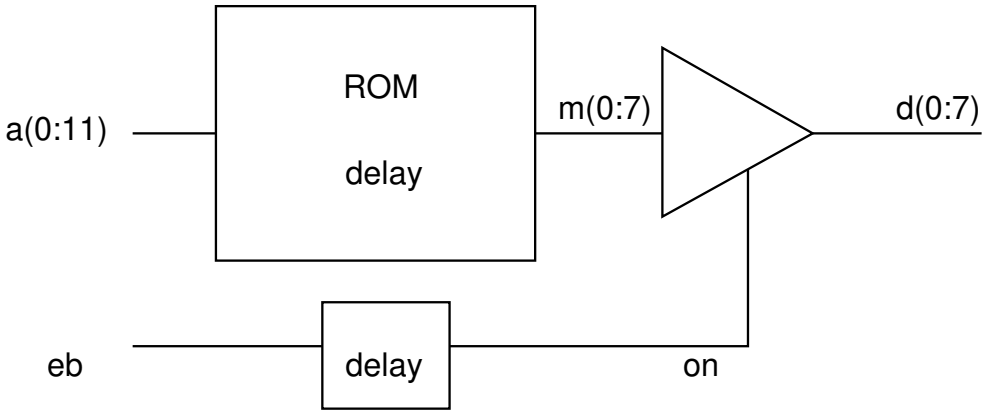


Figure 3.1: organization of behavioral description of the ROM

```
6   Integer ta,ten
7   Constant size = 1 << 12
8
9   Behavior
10      Capa 9  PF a,eb
11      Capa 20 PF gb
12      Capa 12 PF d
13      Slew 60 PS/FF d
14
15      Variable rom(0:size-1)
16      Variable i,f,date
17
18      f = Open("program.rom")
19      For i=0:size-1 Cycle
20          rom(i) = Read(f) & 255
21      Repeat
22      Close(f)
23
24      Undef m
25
26      When a Change Do
27          Undef m
28          date = Time
29
30          At Time + 170 NS Do
31              If Time = date + 170 NS Then
32                  Set m = rom(Eval(Value[a]))
33              Endif
34          Done
35      Done
36
37
```

```
38      When eb Low Do
39          At Time + 60 NS Do
40              Set on = 1
41          Done
42      Done
43      Set on = 0 When eb High
44      Undef on When eb Undef
45
46      When m,on Change Do
47          Switch Value[on]
48          Case 1
49              Set d = Value[m]
50              Exit
51          Case 0
52              HighZ d
53              Exit
54          Default
55              Undef d
56          End
57      Done
58  End
59  End
```

Behavioral statements which are not contained to a `When` block, are evaluated during the initialization phase of the simulation. These statements are the initialization of the behavioral model for the part (lines 10–22).

The array of variables `rom` will contain the values of the words of the ROM. It is loaded from a file during the initialization of the simulation (line 18–22).

We then describe the actions to be taken on given events (with the `When` operator). As it is easier to describe separate automata for separate functionalities, we have divided the description of the ROM is 3 parts (fig. 3.1):

- the ROM itself with its delay (line 26–35),
- a delay on the enable (`eb`) (line 38–48),
- the delayless tristate buffer (line 50–61).

The `When` operator

The `When` operator is used to trigger an action on each occurrence of a given event.

The syntax of this statement is:

`When` *signal list* `<transition> Do`
statement list
`Done`

or the abbreviate one:

statement `When` *signal list* `transition`

`<transition>` is one of the following keywords:

- `Undef` : the *statement list* is executed at every end of transition to the `Undef` state.
- `HighZ` : the *statement list* is executed at every end of transition to the `HighZ` state.

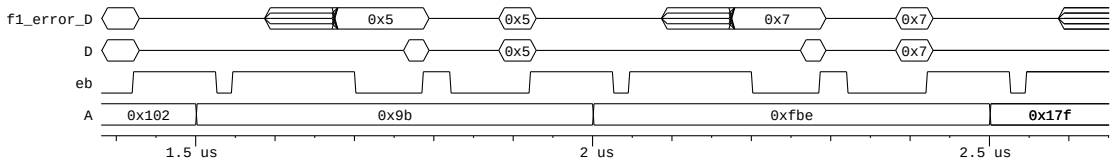


Figure 3.2: wave-forms of the ROM

- `Low` : the *statement list* is executed at every end of transition to the `Low` state.
- `High` : the *statement list* is executed at every end of transition to the `High` state.
- `Change` : the *statement list* is executed at every end of transition of any kind of the *signal list*.

For instance (line 43) the `Set` statement is executed each time the signal *eb* reaches the state `High`.

The `At` operator

To simulate delays, we need to postpone the execution of actions using the `At` operator. The syntax of this statement is:

`At` *expression* `Do`
statement list
`Done`

or the abbreviate one:

statement `At` *expression*

The *statement list* is executed at absolute time given by *expression*, expressed in picoseconds. The `Time` keyword which represents the current time during the simulation is used to transform a delay into an absolute time.

When the delay is elapsed, the action is executed. It generally needs variables and states of signals. A serious problem is to choose whether to take the values of the variables and signals before the delay or after. By default the evaluation occurs after the elapsed delay.

Suppose (fig. 3.2) that *eb* is set to 0 at time t_0 and set at time $t_1 < t_0 + 60$ns. Then at t_1 *on* is set to 0 and later at $t_0 + 60$ns *on* is set to the opposite of the value of *eb* at time $t_0 + 60$ns: 0. So nothing happens.

On the opposite, if we replace the line 40 by

```
Set on = Eval(\Value[eb])
```

the expression is evaluted at time t_0 with the value 1. At time t_1, *on* is set to 0. At time $t_0 + 60$ns, *on* is set to 1.

In this example, the first choice is the good one. But it is not the case if we need to simulate a pipe which replays with a given delay everything that comes in:

```
When in Change Do
       Set out = Eval(Value[in]) At Time + 10 NS
Done
```

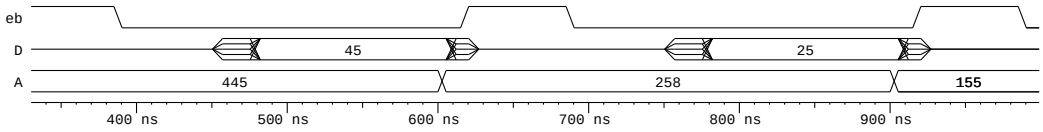


Figure 3.3: wave-forms of the ROM

Here the `Eval` functions is necessary when the delay between two changes may be smaller than 10 ns.

Glitches are the source of our problem. Sometimes we want to specify that the result is available after a delay during which the inputs are steady. The behavior model needs to check that no transition has occurred since a given time (line 31).

The figure 3.3 shows the wave-forms of the simulation.

3.4 Ext2mod: an automatic cell characterization

The major problem of a library is its reliability. To provide a complete interface for a layout library we must provide for each cell the following information:

- the layout,
- the **model** input/output interface,
- the Electrical Rule Checking interface,
- the transistor level description,
- the gate level description,

The description of the layout aspect consists in a `.mag` file (see magic(5)) while the rest of the information is contained in the associated **model** function. The purpose of **ext2mod** is to automatically derived from the extracted netlist file `.ext` associated to the layout file the **model** description of the file. Creating a new cell proceeds in the following steps:

- the design of the layout with **magic** (creation of *cell*`.mag` see layout in figure 3.4 and file `nand2.mag`),
- the extraction of the hierarchical netlist with **magic** (creation of *cell*`.ext` see file `nand2.ext`),
- the generation of the model description by the command:
  `ext2mod` *cell*
  (creation of *cell*`.mod` see table 3.4) and file `nand2.mod`).

Ext2mod detects the input/output signals of the cell (line 1). In order to reserve simple names to generic part of **model**, cells imported from **magic** are identified with their `mag_` prefix.

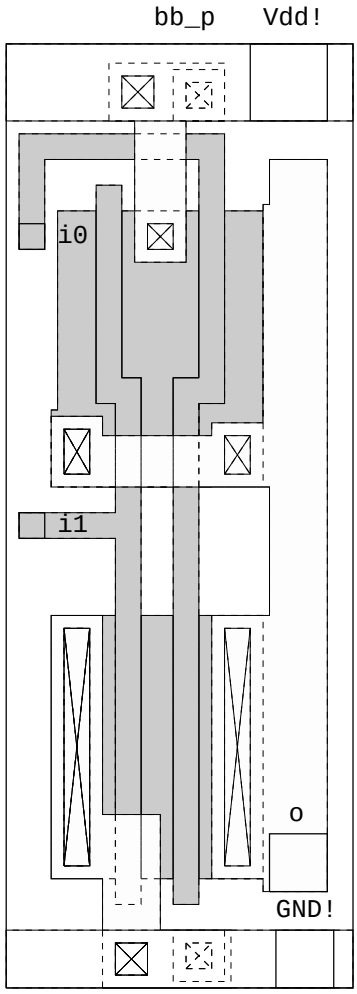


Figure 3.4: layout of the `nand2` gate

The transistor level description of the layout of the nand gate is shown at lines 5–14. Parasitic capacitances and transistors are defined and interconnected.

The Electrical Rule Checking interface is defined for the input/output signals at lines 17–18. The gate level description takes place at lines 19–21. Parasitic capacitances and logical gates are defined and interconnected. A timing model is directly derived from the size of the transistors and the capacitors. Delays are defined in ps and fanout are defined in ps/pF. This *nand* has the following characteristics:

- an intrinsic delay of 586 ps on each input,
- an additional delay of 3138 ps for every 1 pF connected to the output,
- a 0.095 pF load capacitance on each input.

```
1   MagicPart mag_nand2 [i(0:1)] -> o
2       { transistor description }
3       Signal l_0
4
5       Output "C ",Net[Vdd]," GND 35"
6       Output "C ",Net[o]," GND 74"
7       Output "C ",Net[i(0)]," GND 7"
8       Output "C ",Net[i(1)]," GND 5"
9       Output "C ",Net[l_0]," GND 17"
10      Output "C ",Net[GND]," GND 24"
11      Output "p ",Net[i(1),o,Vdd],"2300 200 0 0"
12      Output "p ",Net[i(0),Vdd,o],"2350 200 0 0"
13      Output "n ",Net[i(1),GND,l_0],"2050 200 0 0"
14      Output "n ",Net[i(0),l_0,o],"2050 200 0 0"
15
16      { logic description }
17      Dest o
18      Source i(0), i(1)
19      Logic "cap ",Net[i(0)],95
20      Logic "cap ",Net[i(1)],92
21      Logic "nand ",Net[o]," 3138 ",Net[i(0)]," 586 ",Net[i(1)]," 586 "
22
23  End
24
```

Table 3.4: electrical description of the `nand6` gate generated by **ext2mod**

3.4.1 Layout optimization

The `-f` option of **ext2mod** generates `feed` a **magic** command file which can be used while editing the cell. The command:

```
:source feed
```

executes this file. It defines some feedbacks which indicates some information on the transistors. It helps to scale the n/p-transistors. If the p-transistor path of a gate is to weeak, transistors responsible for this fact will get feedbacks to announce the problem associated with the scale required to adapt it.

3.4.2 Automatic documentation

The `-t` option of **ext2mod** generates a data sheet associated with the cell. The next page shows the data sheet automatically generated by the command:

ext2mod -t nand2; latex nand2; dvips nand2.

The appendix D has been automatically generated by **ext2mod**. For more details about this program, see ext2mod(1).

As we have now defined how to import layouts into the system, it is time to learn how to build bigger layout structures with **mod2mag**. This is the subject of the next chapter.

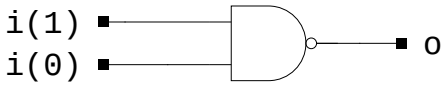
Cell Nand2: a 2-input nand

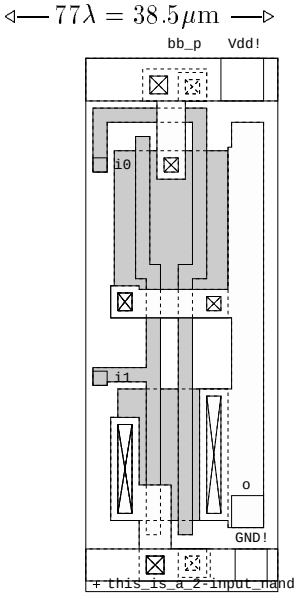
`MagicPart mag_nand2 [i(0:1)] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.82
Inputs	*Capacitance*	Δt ns
i(0)	0.09 pF	0.54
i(1)	0.09 pF	0.54

Description





Chapter 4

Hardware Generation

This chapter is the occasion to discover the different styles of hardware generation. We will discover:

- the use of external layouts,
- the tessellation of layouts,
- the generation of a basic cell,
- the generation of a macro-cell,
- the data-path generation,
- the chip finishing,
- the generation of pcb data.

Every declared part may contain the information on the way to build its layout. The style of this building is specified by replacing the "`Part`" keyword by one of the following:

- `Part`: this is only a method for building the hierarchy. No Layout will be generated: it will be the responsibility of the calling parts. `Part` is a macro with parameters.
- `MagicPart` specifies that the part is an imported cell. Its layout must already exist.
- `PadPart` specifies that the part is an imported pad cell. Its layout must already exist.
- `PavePart` specifies that the layout is built by tessellation of the layouts of subcells.
- `BlockPart` specifies that the part is a compiled data-path.
- `ChipPart` specifies that the part is a whole circuit with pads.
- `PcbPart` specifies that the part is a whole board.

`MagicPart`, `PadPart`, `PavePart`, `BlockPart` are involved in with layout generation (`.mag` files). The command:

`model` *design* `-m`

generates the layout files. `ChipPart` and `PcbPart` are concerned by Pcb data generation. The command:

`model` *design* `-p`

produces theses data.

Some of these styles (`PavePart` and `ChipPart`) are only placement operators: the electric connectivity is not guaranteed and is assumed to be achieved by the placement itself. No routing is performed by **mod2mag**. This is the reason why, an associated netlist is generated in the `.net` format of **magic**(see net(5)). It is possible to use the router of **magic** to draw these interconnection (see the example of the section 4.6, for the chip finishing).

We will discover in this chapter these different layout generation styles.

4.1 Importing external layout

It is necessary to provide a way to import layouts, in order to get:

- basic pieces of layouts to assemble them,
- complex and hierarchical pieces of layouts designed using other CAD-tools (it allows a bridge with external tools).

If the layout of a part is directly imported, it is specified through the keyword `MagicPart` for general cells and `PadPart` for pad cells which must be located at the periphery of the chip. Therefore we can directly use the **model** description given by **ext2mod**.

When the software needs to generate the layout of the part `mag_nand2`, it simply uses the one accessible in the file named `nand2.mag`.

A pleasant way to use the file generated by ext2mod is to include it (the `Include` command). This will be very flexible in case of modification of the layout. Everything can be recompiled thanks to a makefile mechanism.

To avoid the proliferation of very small tile files required to describe the various parts of layout, it is possible to gather them in a layout library. The statement

`Magiclib "`*layoutlib*`"`

declares the name of a layout library located in *layoutlib*`.mag`.

It is also possible to declare zonesin the layout (using the labels `x_`*param* and `y_`*param*) that will be stretched depending on the value of integer parameters (*param*) in the **model** description of the part.

The layout library and the zone stretch mechanisms are detailed in the section describing how to build a basic cell.

4.2 Tessellation of layout

As we know how to call basic pieces of layout, it is time to learn how to build composed layouts.

The most simple way of building layout consists in putting different layout tiles together. The resulting building is a tessellation of layouts. This style of generation does not deal with electrical connections (nonetheless, they can be performed by side effects). It consists in putting

side by side the different pieces. This style is specified while declaring a part. The use of the keyword `PavePart` instead of `Part` indicates that the tessellation style is desired for the generation of the layout of that part.

The structure of a layout building is hierarchically described with constructors. It follows the natural hierarchy given by the `Part` instance mechanism. It is also possible to structure the layout inside a part definition using the following constructors:

- placement constructors (`Xplace`, `Yplace` and `Yplace/Pile`),
- transformation operators (`Xmirror`, `Ymirror`, `Rotate90`, `Rotate180` and `Rotate90`).

4.2.1 Placement constructors

The Placement constructors are used as soon as a topological placement is required for the layout generation or for the PCB generation. The placement constructors follow the syntax:

`Xplace` { *or* `Yplace` *or* `Zplace` *or* `Pile` }
statements
`End`
statements.

Reference rectangle

A geometric support is required to achieve the tessellation. The geometrical support is a rectangle (called *reference rectangle*) which roughly defines the occupied place of the layout. In an imported layout, this rectangle is specified by a label (see `magic` (5)) named `bb_p`. This label can be observed in the layout of the `nand2` cell (figure 3.4, page 37). If this label does not exist in the layout, the effective bounding box is used. Often the desired rectangle fits the bounding box, the label is then optional. In a composed layout (built with the tessellation constructors, or other constructors to be defined), the rectangle is the bounding box of the set of reference rectangles of the constituting layouts.

This rectangle is the only topological information used by the layout generator. The effects of the constructors are directly drawn from these rectangles. It is now possible to describe the constructors.

`Xplace` concatenates the constituting instances in the order of their interpretation, from left to right, in a way that the lower-right corner of the reference rectangle of an instance fits the lower-left corner of the reference rectangle of the next instance. The figure 4.1.a shows an example of an X-axis style concatenation.

`Yplace` concatenates the constituting instances in the order of their interpretation, from the bottom to the top, in a way that the upper-left corner of the reference rectangle of an instance fits the lower-left corner of the reference rectangle of the next instance. The figure 4.1.b shows an example of an Y-axis style concatenation.

`Pile` or `Zplace` piles all the constituting instances on top each other in a way that the lower-left corners of the reference rectangle of each instance are located at the same point. The figure 4.1.c shows an example of piling up style. This operator is very useful for customizing a

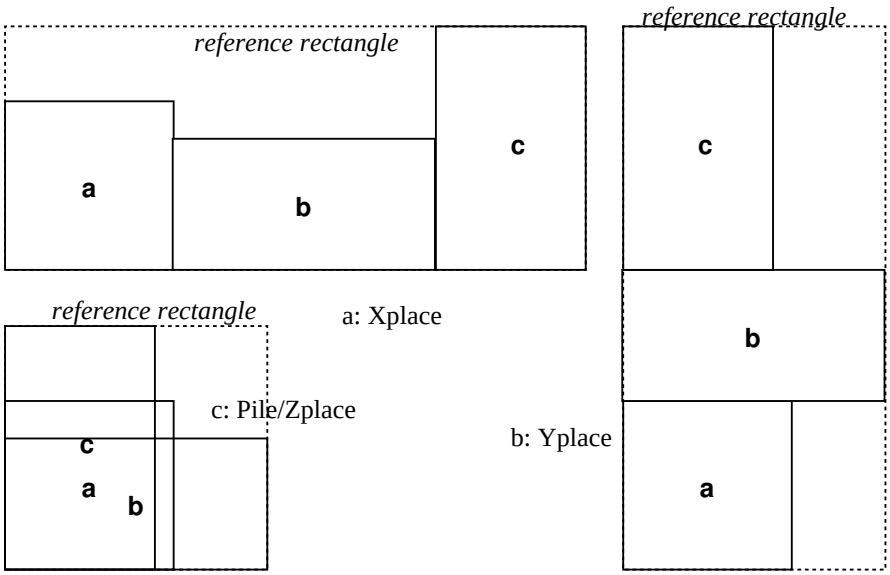


Figure 4.1: examples of the constructors

keyword	*transformations*
`Xmirror`	*symmetry* $-x$
`Ymirror`	*symmetry* $-y$
`Rotate90`	*rotation by* $90°$
`Rotate180`	*rotation by* $180°$
`Rotate270`	*rotation by* $270°$

Table 4.1: transformation operators

cell with vias or contacts. A tile contains the major part of the layout, while other tiles are the personalization of the layout with small connections. For instance, it is useful for decoders.

While the constructors specify the concatenation axis, the transformation operators is useful to change the orientation of the part of a layout.

4.2.2 Transformation operators

The syntax of these operators is the same as the one of the constructors:

```
<Transformation operator>
        statement(s)
End
```

The operators applies to the statement(s) which is(are) between itself and the `End` statement. The table 4.1 summarizes those different operators.

Two examples will illustrate the capabilities of the tessellation style with their operators. The first one will show how to build a basic parametrical cell and second one will show how to

build macro-cell.

These examples will be the subject of the next two sections.

4.3 The building a basic cell

In this section we illustrate the use of the tessellation style for basic cells. As this style is adequate with transistors sizing, we have chosen to design a parametrical inverting buffer. This buffer will be speed optimal. All the files involved in this example are available in the directory `~cad/ulm/mcmos/lib8`.

4.3.1 Speed of a CMOS inverting driver

The purpose of a driver is to amplify a weak signal to distribute it on a heavily loaded wire. A basic inverter can be characterized by:

- its input capacitance C (expressed in pF), proportionnal to the $W \times L$[1] area of the gates of the transistors,
- its output impedance R (expressed in ns/pF), proportionnal to the form factor $\frac{W}{L}$ of the transistor gates.

It results that L must be minimal ($2\mu m = 4\ \lambda$) in our technology. Both the input capacitance and the output impedance are directly related to the transistor sizes (W for the n transistor and $2W$ for the p transistor)[2]. Our time optimization consists in designing the most rapid driver, for an given input capacitance C_i and a given output load capacitance C_o.

The result of this classical optimization follows:

- The inverter driver must be constituted with an odd number of inverters.
- The input capacitance of each inverters must follow a geometrical sequence.
- The ratio of this sequence must approach e and consequently the size of the sequence must be the odd number near to $\log k = \log \frac{C_o}{C_i}$.

The numerical table 4.2 summarizes the boundaries of the k values for different stage numbers of the sequence.

4.3.2 The model file of the driver

Our inverter (`niceinv`) will consist in a geometrical sequence of inverters (`n_inv`).

The parametrical basic inverter

We first define a parametrical basic inverter `n_inv` (see the table 4.3[3]). The parameter n is the width of the channel of the n-transistor; the size of the p-transistor being twice larger.

[1] W is the width of the transistor channel and L is its length.

[2] As the holes are 2.5 times slower than electrons, the p transistor will be 2 times larger than the n transistor in a inverter.

[3] a part of the file `niceinv.mod`.

p	$2p-1$	$2p+1$	k
1	1	3	5.19615
2	3	5	46.1175
3	5	7	360.778
4	7	9	2741.89
5	9	11	20603.2

Table 4.2: optimum stages number for a given k factor

This **model** file calls (on line 1) the library cell `invlib` which contains all the necessary pieces of layout. This mechanism avoids the proliferation of very small layout cells. The **model** statement

`Magiclib "`*magiccell*`"`

reads the layout contained in the file "*magiccell*`.mag`" and identifies the different tiles. Each tile is limited by the label "`c_`*tilename*", which corresponds to the layout of a part known under **model** as `mag_`*magiccelltilename*. The reference rectangle for this tile is delimited by the label "`p_`*tilename*". This label is not required if the reference rectangle fits the `c_` label. The figure 4.2 shows this layout. For instance, the tile `iv` is delimited by the label `c_iv` and its reference rectangle is delimited by the label `p_iv`. This layout is known under **model** as the part `mag_invlibiv` (line 7-8).

All the tiles of this cell (figure 4.2[4] must be declared in the **model** file with the `MagicPart` header (lines 3 to 18). These cells are empty because the electrical characterization of the inverter will be directly given in its definition (lines 22 to 26).

The parametrical basic inverter is built according to different styles depending on the required size. As we want to use this inverter as part of a standard library (see appendices A and D) for the data-path generator, we have some constraints about the height of the cells and about the positions of the inputs/outputs and the power-supplies (see the section 4.5). We shall try to have the smallest width for a given parameter. This leads to 3 different styles:

- a small inverter with one Y-axis pair of transistors (tile `vert`).
- a medium inverter with two Y-axis pairs of transistors (tiles `vert` and `iv`).
- a large inverter with two X-axis pairs of n transistors and four X-axis pairs of p transistors (tiles `hl`,`hm` and `hr`.

Vertical scalable inverter If the n parameter is less than 27 μm (line 32) the vertical style is used. It is composed of a parametrical vertical inverter (`mag_invlibvert` at line 38, to cover the 4–15μm range) and, if necessary, an associated 12μm fixed size inverter (`mag_invlibiv` at line 39 to cover the 16-27μm range). The parametrical vertical inverter is declared at lines 5-6 with the parameters `n`, `en`, `ep` and `p` which must correspond in the layout with the labels `y_n`, `y_en`, `y_ep` and `y_p` to stretch different zones. A given parameter *param* will be automatically associated with one or more labels called `y_`*param* and/or `x_`*param* in the layout. An error is generated if none is found. The resulting layout for a given instance of this cell will be stretched

[4] file `invlib.mag`

```
1   MagicLib "invlib"
2
3   MagicPart mag_alim[]
4   End
5   MagicPart     mag_invlibvert(n,en,ep,p)[in]
6   End
7   MagicPart mag_invlibiv[]
8   End
9   MagicPart mag_invlibhl[in]
10  End
11  MagicPart mag_invlibhm[]
12  End
13  MagicPart mag_invlibhr(adjust)[]
14  End
15  MagicPart mag_invlibpc[]
16  End
17  MagicPart mag_invlibout[] -> out
18  End
19
20  Part n_inv(n)[i] -> o      { n is the channel width in microns }
21          {the transistor description}
22      Output "n ", Net[i], Net[GND], Net[o], 4, " ", 2*n, " 0 0"
23      Output "p ", Net[i], Net[VDD], Net[o], 4, " ", 4*n, " 0 0"
24          {the logical description}
25      Source i
26      Dest o
27      Logic "cap ",Net[i],n*4
28      Logic "not ",Net[o],(58800/n),Net[i], 0
29      Logic "cap ",Net[o],n*5
30          {the topological building}
31      Integer p,i
32      If n < 27 Then
33          { vertical inverter }
34          p = n
35          p = p - 12 If p > 15
36          Xplace
37              p = 4 If p < 4
38              mag_invlibvert(2*p,38-2*p,68-4*p,4*p)[i]
39              mag_invlibiv[] If n > 15
40          End
41      Else
42          { horizontal inverter }
43          Xplace
44              n = n - 27
45              mag_invlibhl[i]
46              While n > 38 Cycle
47                  mag_invlibhm[]
48                  n = n - 38
49              Repeat
50              mag_invlibhr(n) []
51          End
52      Endif
53  End
```

Table 4.3: definition of a one-stage parametrical inverter

so that the width of the `x_`*param* label(s) and the height of the `y_`*param* label(s) will be of *param*

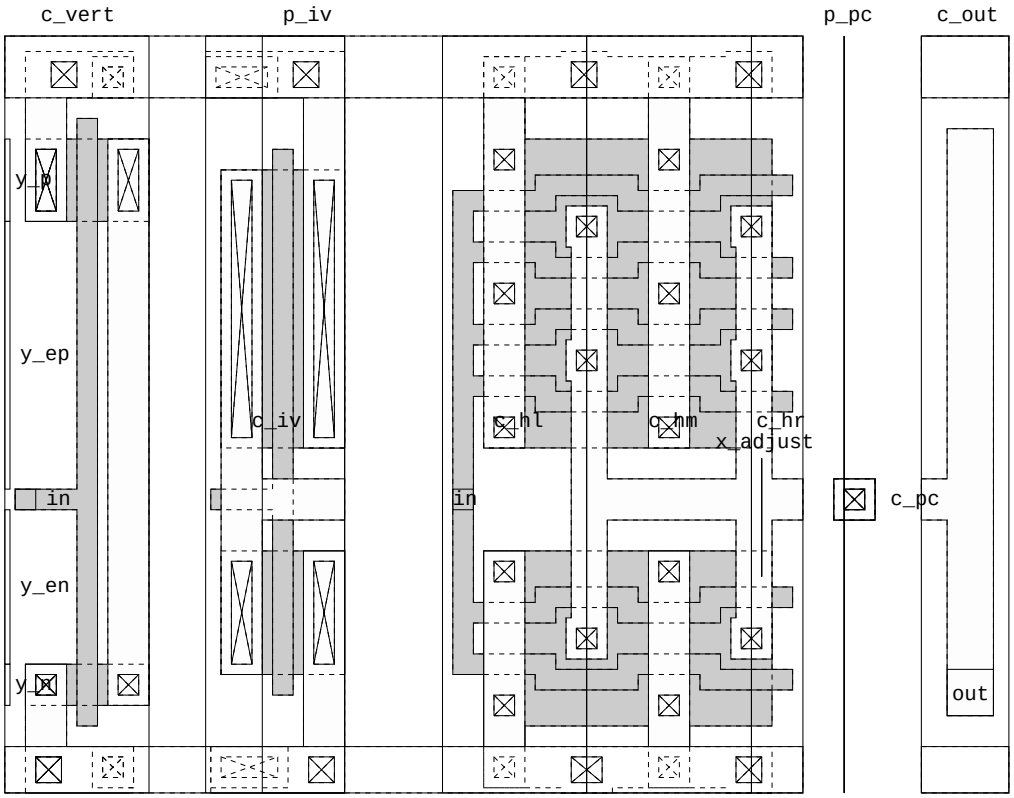


Figure 4.2: layout of the `invlib` library cell

λ.

Horizontal scalable inverter The horizontal scalable inverter is built with 3 tiles:

- the left tile (`mag_invlibhl` at line 45 which contributes for 25μm of inverter gate width),
- the repetitive tile (`mag_invlibhm` at line 47 which contributes for 38μm at each occurrence),
- the scalable right tile `mag_invlibhr` at line 50 which contributes in the range 2–40μm. The 38μm range of this tile is useful to adjust precisely the whole size of the inverter.

Multistage scalable inverter

In the layout library, 5 areas can be identified:

1. a small vertical inverter (`c_vert`),
2. a small fixed-size inverter (`c_iv`),
3. a big horizontal scalable inverter (`c_hl`, `c_hm`, `c_hr`),

4. a *polysilicon* contact `c_pc` to be able to connect multistages drivers. The reference rectangle of `c_pc` is degenerated and allows the inverter interconnections without any space lost.

5. A *metal1* output interface with its label for the driver finishing.

The inverting driver is decomposed in $2p+1$ different basic inverters with their sizes following a geometrical sequence of ratio $\sqrt[2p+1]{\frac{C_o}{C_i}}$. The associated **model** part is described in the table 4.4[5]. Three kinds of driver can be generated:

- the one-stage driver if k is less than 5 (lines 64–65),
- the three-stages driver if k is less than 46 (lines 67–77),
- the five-stages driver if k is greater than 46 (lines (79–92),

The tile `pc`, called `mag_invlibpc` in the **model** file (lines 73, 75, 84,...) connects the successive stages. The output interface (`mag_invlibout`)is added for every styles. It makes the cell compatible with the data-path generator.

4.3.3 Labeling the input/outputs of a generated cell

When a layout is generated, all the tiles are put together into a single block. The labels are dropped unless connected to an input/output signal of the cell. Keeping these labels allows to use the new cell as part of another bigge cell.

Each of these labels is initially located in a sub-tiles with a name local of the context of the model part associated to this subtiles. This label must be renamed with the name in the context of the created cell.

For instance, the tile part `mag_invlibout` is declared with one output called `out` (line 17). This part is used (lines 65, 77 and 92) with its output connected to the signal `o` which is in fact a input/output signal of the part `niceinv` generated by the tessellation generator. So every labels of `c_out` the name of which is "`out`" will be dumped in the resulting layout of `niceinv` under the name `o` of the signal in the context of the generated cell `niceinv`.

Global signals (ended under magic with a `!`) are systematically dumped. It is also possible to force every labels of a subcell to be dumped, if there exists a label call `b_flatlabel` in the subcell.

The figure 4.3 shows the instances of `niceinv` for the parameters: (6, 180), (6, 20), (20, 100) and (6, 60).

4.4 The building of a regular macro-cell

In this section, we show how to develop a regular macro-cell generator. We will illustrate it with a Programmable Logic Array generator.

A **P**rogrammable **L**ogic **A**rray is a piece of hardware useful to implement random combinatorial logic. It is composed in 4 logic levels:

1. buffers and inverters of the input signals,

[5] A part of `niceinv.mod`.

```
   ...
55 PavePart niceinv(cd,cf)[i] -> o
56
57     Integer k,s
58     Signal temp(0:3)
59
60     Xplace
61         cf = 6 If cf < 6
62         k = cf/cd
63         If k <= 5 Then
64             n_inv(cf)[i]->o
65             mag_invlibout[] -> o
66         Else
67             If k<= 46 Then
68                 s = (10 * (Log(k)-1))/3
69                 While(s^3 < k*1000) Cycle
70                     s = s+1
71                 Repeat
72                 n_inv(cd * s/10)[i]-> temp(0)
73                 mag_invlibpc[]
74                 n_inv(cd * s * s/100)[temp(0)]-> temp(1)
75                 mag_invlibpc[]
76                 n_inv(cd *s*s*s/1000)[temp(1)]-> o
77                 mag_invlibout[] -> o
78             Else
79                 s = (10 * (Log(k)-1))/5
80                 While(s^5 < k*100000) Cycle
81                     s = s+1
82                 Repeat
83                 n_inv(cd * s/10)[i]-> temp(0)
84                 mag_invlibpc[]
85                 n_inv(cd * s^2/100)[temp(0)]-> temp(1)
86                 mag_invlibpc[]
87                 n_inv(cd *s^3/1000)[temp(1)]-> temp(2)
88                 mag_invlibpc[]
89                 n_inv(cd *s^4/10000)[temp(2)]-> temp(3)
90                 mag_invlibpc[]
91                 n_inv(cd *s^5/100000)[temp(3)]-> o
92                 mag_invlibout[] -> o
93             Endif
94         Endif
95     End
96 End
```

Table 4.4: the **model** description of the scalable driver `niceinv`

2. a first matrix of *nor* gates (often realized with NMOS technics),
3. a second matrix of *nor* gates,
4. output buffers.

It allows to implement every conjunction of disjunction of the inputs or their opposite. The number of terms of the PLA (the number of connections between the two matrices) limits the application field of the PLA. A logical function can be implemented in different PLA with different number of terms. It is important to minimize the number of terms of the PLA.

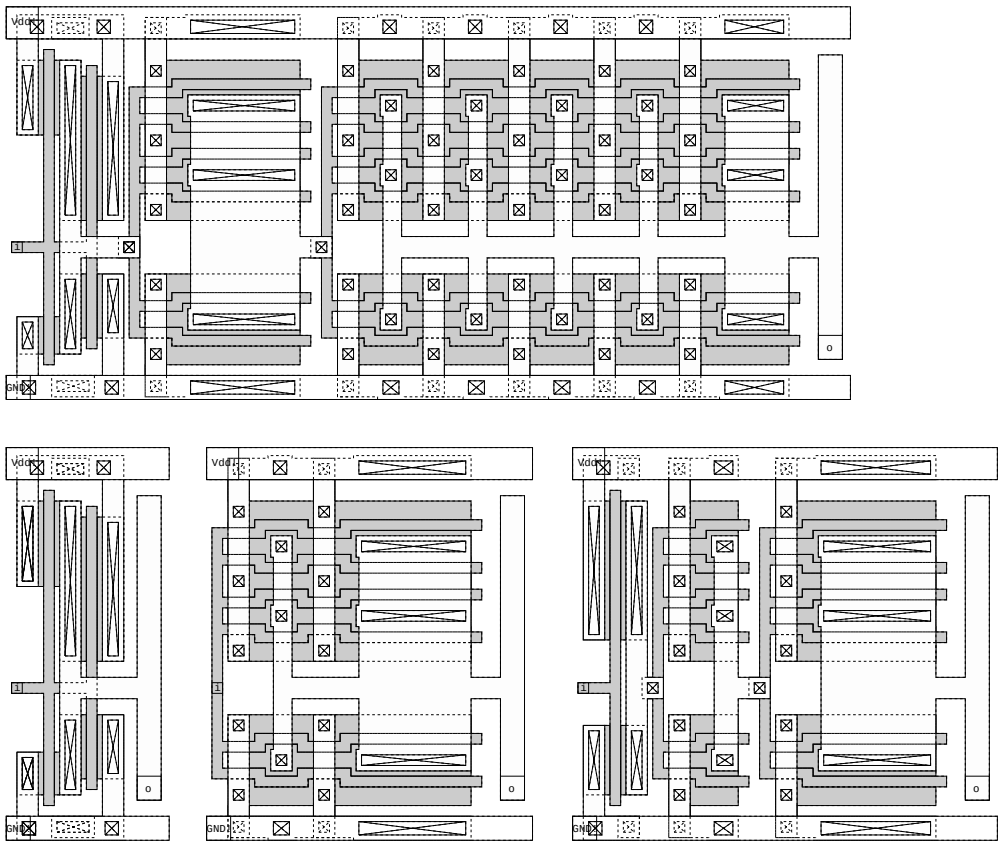


Figure 4.3: layouts of different instances of `niceinv`

Espresso (see espresso (1)) achieves some optimizations. In fact ROMs are unoptimized PLA. The first matrix is then the decoder of the n bit input ROM and the number of terms is 2^n. The files related to this example can be found in the directory `~cad/ulm/mcmos/pal`. We will follow the template approach proposed in MPLA (MPLA (1)) and continue to use the input formats (`.eqn` and `.tt`) of the Berkeley CAD-tools for PLA in order to use espresso. The file `hexa.eqn` contains the equation of an hexadecimal 7 segment decoder (table 4.5).

The following unix command (with the Berkeley tools) converts this file in the `.tt` format (table 4.6):

```
eqntott -l -r hexa.eqn | espresso > hexa.tt
```

The `.tt` format cannot be read by the **model** `Read` statement: the `tt2tab` (see tt2tab(1)) program is provided, which converts the `tt` file to a `.tab` file, which can be read by **model**.

The source of the `tt2tab` is located in the file `~cad/ulm/src/divers/tt2tab.c`. It also generates also on its standard output the **model** interface for the PLA. The command

```
tt2tab hexa > hexa.mod
```

generates the interface shown in the table 4.7 (file `hexa.mod`).

The part `pla` (line 5) is defined in the `pla.mod` model file (located in `pla.mod`). Unfortunately, this file is too big to be printed and detailed in this text. We will

```
1   NAME=hexa;
2   INORDER = i0 i1 i2 i3;
3   OUTORDER = a b c d e f g;
4   a = !i3 & !i2 & (!i0 | i1) | !i3 & i2 & (i0 | i1) |
5       i3 & !i2 & (!i1 | !i0) | i3 & i2 & (!i0 | i1);
6   b = !i3 & !i2 | !i3 & i2 & (!i1 & !i0 | i1 & i0) |
7       i3 & !i2 & (!i1 | !i0) | i3 & i2 & !i1 & i0;
8   c = !i3 & !i2 & (!i1 | i0) | !i3 & i2 |i3 & !i2 |
9       i3 & i2 & !i1 & i0;
10  d = !i3 & !i2 & (!i0 | i1) | !i3 & i2 & (!i1 & i0 | i1 & !i0) |
11      i3 & !i2 & (!i1 | i0) | i3 & i2 & (!i1 | !i0);
12  e = !i3 & !i2 & !i0 | !i3 & i2 & i1 & !i0 |
13      i3 & !i2 & (!i0 | i1) | i3 & i2 ;
14  f = !i3 & !i2 & !i1 & !i0 |!i3 & i2 & (!i1 | !i0) |
15      i3 & !i2 |i3 & i2 & (!i0 | i1);
16  g = !i3 & !i2 & i1 |!i3 & i2 & (!i1 | !i0) |
17      i3 & !i2 |i3 & i2 & (i0 |i1);
```

Table 4.5: the equation describing an hexadecimal 7 segment decoder (file `hexa.eqn`)

```
1   .na hexa
2   .ilb  i0 i1 i2 i3
3   .ob  a b c d e f g
4   .i 4
5   .o 7
6   .p 14
7   1101 0001100
8   -100 0001001
9   11-0 1110000
10  000- 0001010
11  -001 1001000
12  0010 0110011
13  -00- 0110000
14  1011 0111101
15  1010 1011011
16  0-0- 1100100
17  -111 1000111
18  0-11 1001110
19  --01 0010011
20  0110 1011111
21  .e
```

Table 4.6: the table describing hexadecimal 7 segment decoder (file `hexa.tt`)

only comment few features. The template is located in `plalib.mag` (see figure 4.4).

4.4.1 Building the PLA structure from the `.tab` file

The file `hexa.tab` is opened in the `hexa Part` at line 3. The first data of the file is a timestamp number of the PLA to identify it without ambiguity. It is read on line 4 and passed in parameter to the `pla Part` in order to identify the instance of the PLA.

The file contains then the number of inputs, the number of outputs and the number of

```
1   PavePart hexa[i0,i1,i2,i3] -> a,b,c,d,e,f,g
2       Integer fd,t
3       fd = Open ("hexa.tab")
4       t = Read(fd)
5       pla(4,7,fd,t,0)[i0,i1,i2,i3] -> a,b,c,d,e,f,g
6   End
```

Table 4.7: the **model** interface of the hexadecimal 7 segment decoder

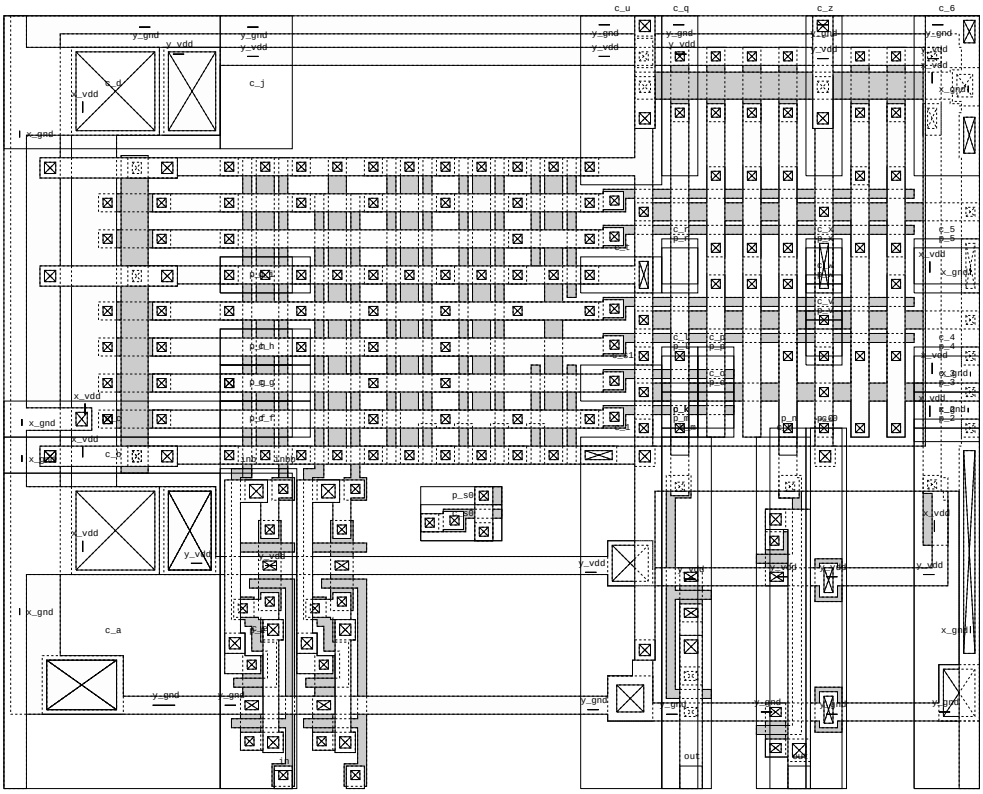


Figure 4.4: PLA template

internal terms. The different zones of the PLA are successively build:

- the left border (with the pullup of the first *nor* matrix),
- the input buffers,
- the first *nor* matrix,
- the intermatrices zone,
- the output buffers,
- the second *nor* matrix.

```
...
202     Yplace
203         For j = 0:nt-1 Cycle         { For each term
204             If j & 3 = 0 Then            { put a body-tie line every 4 lines
205                 Xplace
206                     mag_plalibi[] For i = 0:ni-1
207                 End
208             Endif
209             Xplace
210                 Logicf "nor %s 51477",Net[t(j)]
211                 For i = 0:ni-1 Cycle        { For each input of the current term
212                     k = Read(fd)
213                     Switch k
214                     Case 0                  { this input is not used
215                         mag_plalibf[]
216                         Exit
217                     Case 1                  { this input is used directly
218                         mag_plalibh[r(i)] -> t(j)
219                         Logicf " %s 3850",Net[r(i)]
220                         Exit
221                     Case 2                  { this input is used inverted
222                         mag_plalibg[rb(i)] -> t(j)
223                         Logicf " %s 3850",Net[rb(i)]
224                         Exit
225                     Default
226                         Error "bad value for k :",k If k <0 Or k > 2
227                     End
228                 Repeat
229                 Logicf "\n"
230             End
231         Repeat
232     End
```

Table 4.8: the generation of the first *nor* matrix

The `hexa.tab` contains successively the information to build:

- the first matrix each term after each term (the matrix is built line by line),
- the second matrix each term after each term (the matrix is built column by column),

The table 4.8 shows the generation of the first matrix. `nt` is the number of terms, `ni` the number of inputs, `t(0:nt-1)` is the term bus and `r(0:ni-1)`, `rb(0:ni-1)` are the non-inverter and inverter input bus.

The `For` loops (lines 203–231) enumerates the number of terms and the `For` loops (lines 211–228) enumerates the number of inputs for each terms. For each input of each term, `k` is read (line 212) from the `hexa.tab` file. According to `k` :

- if $k = 0$, the current input is not used by the current term and the tile `mag_plalibf` is laid out (label `c_f` on fig. 4.5),
- if $k = 1$, the current input is used and the tile `mag_plalibh` is laid out (label `c_h` on fig. 4.5),

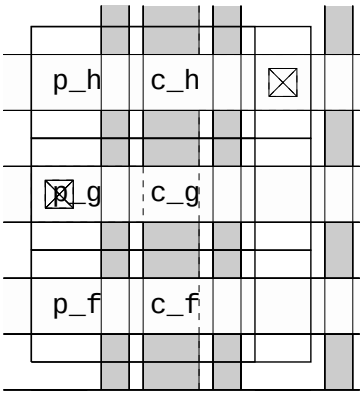


Figure 4.5: the nor matrix tiles

- if $k = 2$, the inversion of the current input is used and the tile `mag_plalibg` is laid out (label `c_g` on fig. 4.5),
- else this is an error which is reported.

The transistors (for the `.sim` netlist) are generated in the calls of the subcells. The *nor* gate is generated input by inputs on lines 210, 219, 223 and 229.

4.4.2 Stretching the power–supplies

Two values `gnd` and `vdd` are calculated from the number of inputs, the number of outputs and the number of terms in order to stretch the power–supplies rails. The tiles located on the boundary accept these two variable as parameters. And the label `x_gnd`, `y_gnd`, `x_vdd` and `y_vdd` specifies in the different tiles to location where to stretch on the different axis.

The resulting layout is shown on figure 4.6.

In the tessellation style, the electrical connection between the tiles must be a consequence of the placement. The designer is responsible of the adequation of the compiled netlist with the resulting layout. It is possible as long the generated layout is rather regular. Unfortunately, only few parts of an integrated circuit verify this condition. That is why we propose another style which offers rather good densities for semiregular layouts like data-paths.

4.5 Data-path generation

The data-path generation style is invoked with the keyword `BlockPart`. The data-path generator is a cell building system. The interconnections in between the cells and between the cells and the sides of the data-path are performed automatically. The placement of the cells is specified by the designer using the placement operators already used for the tessellation style.

It is possible to use this generator to build random logic, but it will be tedious to manually specify the placement. That is why, this style is efficient only for semiregular layout (like data-paths). The figure 4.7 shows the structure of a data-path. There exists an obvious bi-dimensional placement:

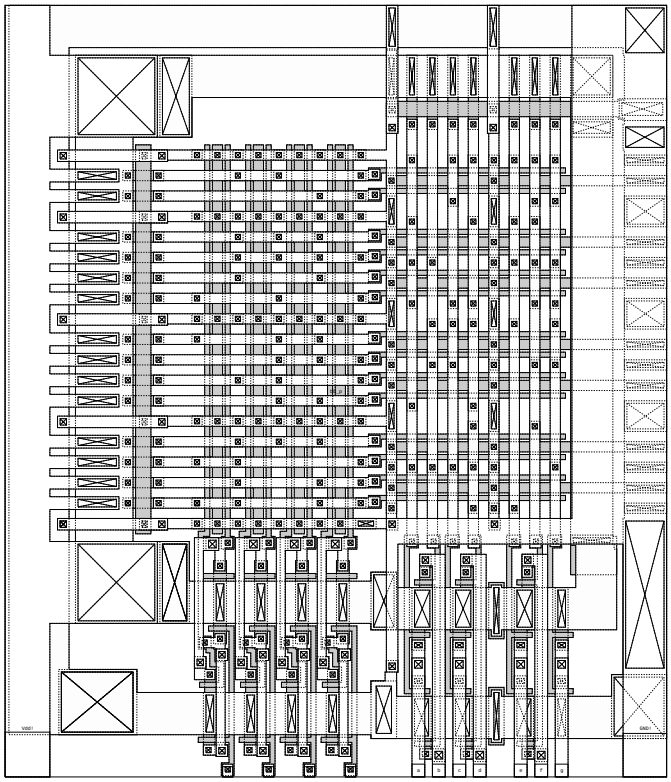


Figure 4.6: the resulting `hexa` PLA

- a whole slice for 1 bit of the data-path is develop along the X axis,
- columns represent basic operators on word width.

Irregularities may be introduced in the structure. Side effect can take place directly in the data-path. The final quality (i.e. the density) will depends of course on the regularity of the layout. Basic cells must comply to a list of constraints in order to be used by the data-path generator. These constraints are detailed for the `mcmos` technology in the appendix A and for the `ecpd` technology in the appendix B. If you want to use the data-path compiler in another technology, inquire at the **mod2mag** maintainer of your site to get the constraint specification for the technology or if you are yourself the local **mod2mag** maintainer, read the appendix C which contains guide lines to build yourself the constraint specification. We will first describe the constraints on the basic cells and then we will explain the building of a data-path.

4.5.1 Use of the data-path generator

The design of a shifter (left, right and rotate) will be an opportunity to discover how to build a data-path. The **model** source is shown on table 4.9[6] and the basic schematic is shown on the figure 4.8

[6] file `shift.mod`

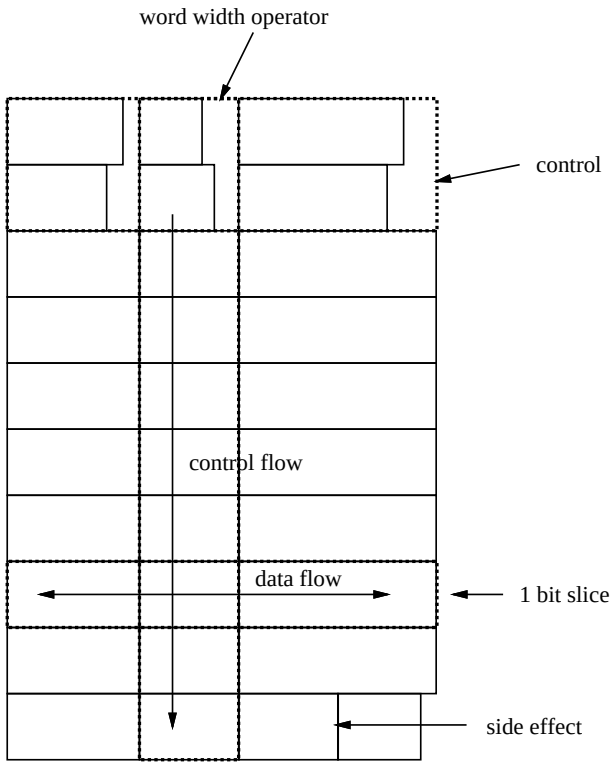


Figure 4.7: the data-path structure

```
1   Include "lib8.mod"
2
3   {    build a step at a given position :
4   {
5   {      Example : n=4
6   {        if in = 0 then out = 11110000000000000
7   {        if in = 1 then out = 11111000000000000
8
9   Part step(n)[in,c(0:n-1)] -> out(0:2^n-1)
10  Signal mid(0:1)
11  Signal outbis(0:2^n-1)
12  Integer i
13  Yplace
14      If n=1 Then
15          Xplace
16              mag_and2[c,in] -> out(0)
17              mag_or2[c,in] -> out(1)
18          End
19      Else
20          step(n-1)[mid(0),c(1:n-1)] -> outbis(0:2^(n-1)-1)
21          step(1)  [in,c(0)] -> mid
22          step(n-1)[mid(1),c(1:n-1)] -> outbis(2^(n-1):2^n-1)
23          out(i) -> outbis((2^(n-1) * (i & 1) + i/2) & 2^n-1) For i=0:2^n-1
24      Endif
```

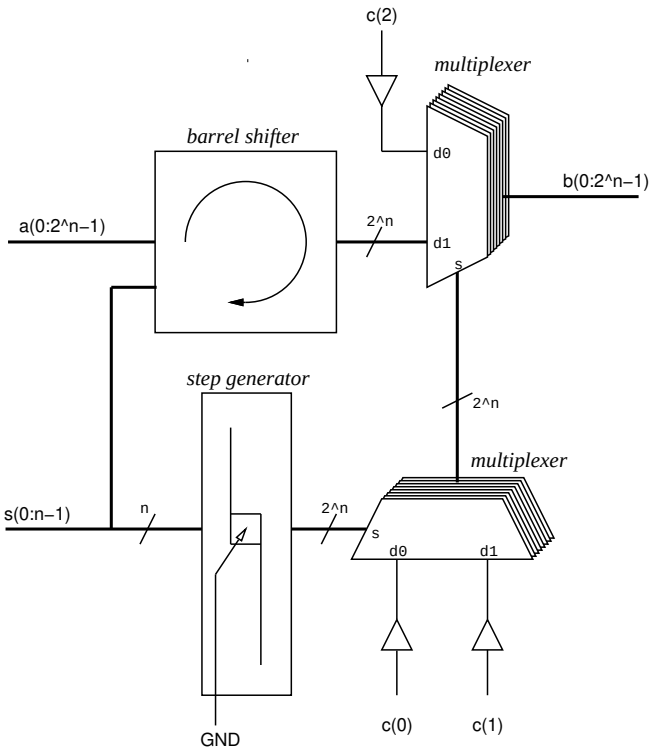


Figure 4.8: the shifter scheme

```
25 End
26 End
27
28 {    oriented step :
29 {        "c" is the position of the stairs
30 {        "in" is the value of the stair's cell
31 {        "m" is the direction of the stairs :
32 {            0 :  000000000000000
33 {            1 :  111111000000000
34 {            2 :  000000111111111
35 {            3 :  111111111111111
36
37 Part ostep(n)[in,c(0:n-1),m(0:1)] -> out(0:2^n-1)
38 Signal outbis(0:2^n-1),mbuf(0:1)
39 Integer i
40 Xplace
41     Yplace
42         step(n)[in,c(0:n-1)] -> outbis(0:2^n-1)
43         mag_buf[m(0)] -> mbuf(0)
44     End
45     Yplace
46         mag_mux21[mbuf,outbis(i)] -> out(i) For i=0:2^n-1
47         mag_buf[m(1)] -> mbuf(1)
48     End
49 End
```

```
50  End
51
52  {    shifter is a general shifter :
53  {        2^n is the size of the shifter
54  {        s is the value of the shift (to the left)
55  {        c is the command :
56  {            c(0:1) is the direction
57  {                1 is Left
58  {                2 is Right
59  {                3 is Rotate
60  {            c(2) is the injected bit in shifts
61
62  BlockPart shifter(n)[a(0:2^n-1),s(0:n-1),c(0:2)] -> b(0:2^n-1)
63  Integer i,j,b,p
64  Top s,c
65  Left a
66  Right b
67  p = 2^n
68  Signal sh(0:(n+1)*p-1),m(0:p-1),sbuf(0:n-1),cbuf(0:2)
69
70  Xplace
71      sh(0:p-1) -> a
72      b = 0
73      For i=0:n-1 Cycle
74          Yplace
75              For j=0:p-1 Cycle
76                  mag_mux21[sh(b+j,b+((j+2^(n-1-i))&(p-1))),sbuf(n-i-1)] -> sh(b+p+j)
77              Repeat
78              mag_buf[s(n-i-1)] -> sbuf(n-i-1)
79              b = b+p
80          End
81      Repeat
82      ostep(n)[GND,sbuf(0:n-1),c(0:1)] -> m(0:p-1)
83      Yplace
84          mag_mux21[cbuf(2),sh(n*p+i),m(i)] -> b(i) For i=0:2^n-1
85          mag_buf[c(2)] -> cbuf(2)
86      End
87  End
88  End
89
90  Constant n=3
91  Signal a(0:2^n-1),sh(0:n-1),com(0:2),out(0:2^n-1)
92
93  Source a,sh,com
94  shifter(n)[a,sh,com] -> out
```

Table 4.9: a general shifter

The shifter is composed of 4 parts:

- a barrel shifter (lines 73-80),
- a step generator (`Parts step/ostep` lines 9-26),
- a multiplexer to compose the mask (line 46),
- a multiplexer to select the final result (line 84).

Barrel shifter

The 2^n barrel shifter is built with n stages of multiplexers (the `For` loop lines 73-86). Each stages i is composed of $p = 2^n$ (line 67) multiplexers (line 76). The j^{th} multiplexer selects between the outputs j and $j + 2^{n-1-i} \bmod 2^n$ of the level $i - 1$.

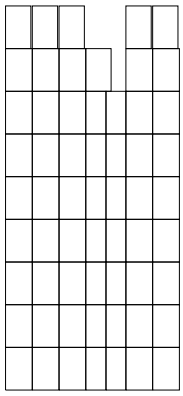


Figure 4.9: the shifter: tessellation of cells

Step generator

The step generator takes as input the n-bit bus `c` and a control signal `in` and returns the 2^n-bit bus `out`. The value of the i^{th} bit of `out` is:

- 1 if i is less than the value of `c(0:n-1)`,
- 0 if i is greater than the value of `c(0:n-1)`,
- the value of `in` if i is equal to the value of `c(0:n-1)`.

The step generator is built recursively. A 2^1-bit step (the terminal recursion case) is described at lines 16-17. A 2^n-bit step is build by the interlacing of two 2^{n-1}-bit steps (line 20 and 22) built from the $n-1$ most significant bits of `c`. `c(0)` is used to particularize each substep (with each `in` input).

4.5.2 Placement and Recursivity

Recursivity is not natural at first sight for the description of hardware. We have said (subsection 2.4.6) that recursivity is essential for the description of trees. When using the layout generator, it is up to the designer to specify both the electrical connectivity and the placement. As a matter of fact recursivity is adequate to describe tree structure for both connectivity and layout placement. The table 4.10 presents a generic tree structure with its placement.

The three important criteria for the placement of an operator in a data-path are:

- the adequation of the height of the operator to the height of the data-path. This is generally easy to fulfill because the operator size is related to the width of the data-buses. In the case of a binary tree, there are $2^h - 1$ nodes and 2^h leaves. As generally, the node

```
1   Part tree(n)[in(0:2^n-1)] -> out
2   Signal left,right
3
4   If n = 1 Then
5       leaf[in] -> out
6   Else
7       Yplace
8           tree(n-1)[in(0:2^(n-1)-1)] -> left
9           node[left,right] -> out
10          tree(n-1)[in(2^(n-1):2^n-1)] -> right
11      End
12  Endif
13
14  End
```

Table 4.10: a generic tree structure with its placement

is able to deal with 2 bits of the data, the whole size of the operator $2^{h+1} - 1$ nearly reaches the size of the data-path 2^{h+1}.

- the minimization of the routing within the operator. This is also easy to reach for the tree (see right of figure 4.10) thanks to the sequence of the lines 8–10.

- the minimization of the routing between the operator and the rest of the data-path. For this example (tab. 4.10) the routing is direct. This is often the case (for every binary–tree reduction). It is not the case for the `step` operator because the location of the cell dealing with the n^{th} bit is at the row p, where p is the inverted binary representation of n. Even in that it is better to use recursivity, because the extra–cost of rearranging the input wires is less than the internal wiring cost for the solution which lays out the cells in the right order. Furthermore the recursive solution is far more understandable than the non-recursive one.

4.5.3 Data-path building

The shifter part declared with the keyword `BlockPart` will be generated by the data-path generator.

The generator first produces the layout as in the tessellation style (fig. 4.9). As every basic cell has the same height, cells are horizontally aligned. Horizontal connections will be done in *metal2* over the cells and, if necessary, in extra channels between rows. Each even row is put up side down in order to connect correctly the Vdd power–supplies of adjacent rows and to share the extra channel (on the GND side).

The generator then detects the vertical alignment between the left side of adjacent cells. Vertical channels are declared and run along different height. In regular Data-path, vertical channels run from the bottom to the top of the data-path. The nesting of the `Xplace` and `Yplace` is important to force some alignment between cells located on top each others.

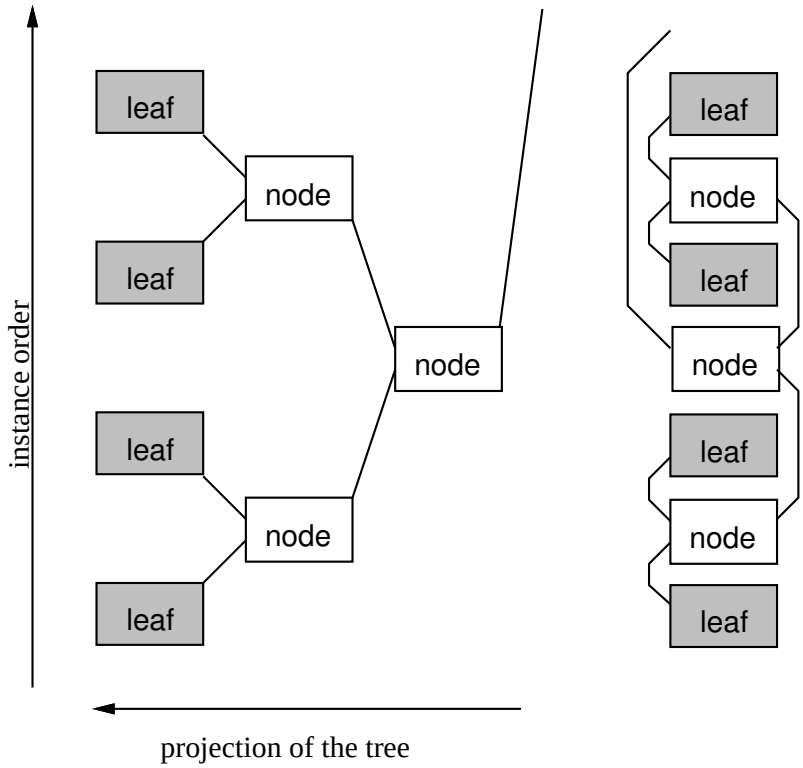


Figure 4.10: placement of a binary–tree

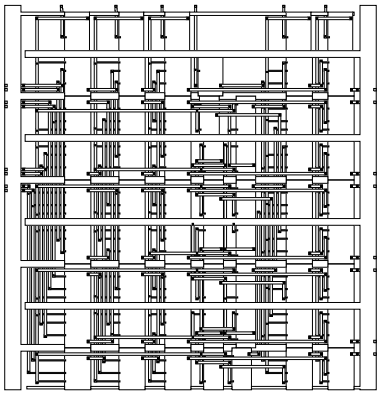


Figure 4.11: the final layout of the shifter

Each signal is a set of n terminals of cells. Each signal is split in $n-1$ terminal-to-terminal connections. The cost of each connection is evaluated on the number of row to cross, the number of cells to run over and the number of vias.

Each connection is then allocated to the different channels (horizontally over the cells and vertically at the left of the cells). From now on, the densities of the channels are known and the columns of cells are move away from each others to make space for the vertical channels and the layout is drawn (fig. 4.11).

Input/Output signals are then extended to a side of the data-path (by default to the right side). It is possible to specify the side for every input/output. The syntax is:

`Right` (or `Left`, `Top`, `Bottom`) *signal list*

The buses `s` and `c` (line 64) will be accessible to the top of the data-path while the bus `a` will be accessible to the left side (line 65) and the bus `b` to the right side (line 66). These inputs/outputs will be labeled in order to finish the cell.

4.6 Chip generation

The use of the chip finishing style will be illustrated on the example shown on the table 4.11[7], which generates the layout of the 74C138 (a 1-of-8 decoder with 3 enables).

```
1   Include "lib8.mod"
2   Include "pad.mod"
3
4   BlockPart decoder8[a(0:2),g,gb(1:2)] -> ob(0:7)
5       Left a(0,1),ob(0)
6       Top ob(1:4)
7       Right ob(5:7)
8       Bottom g,gb,a(2)
9
10      Signal ab(0:2)
11      Signal i(0:7),j(0:7),k(0:7),sel,selb,gbb(1:2),o(0:7)
12      Integer i
13
14      Xplace
15          Yplace
16              mag_biginv[a(0)] -> ab(0)
17              mag_biginv[a(1)] -> ab(1)
18              mag_biginv[a(2)] -> ab(2)
19              mag_inv[gb(1)] -> gbb(1)
20              mag_inv[gb(2)] -> gbb(2)
21              mag_nand3[g,gbb] -> selb
22              mag_biginv[selb] -> sel
23          End
24          Yplace
25              For i=0:7 Cycle
26                  i(i) -> ab(0) If i & 1 = 0
27                  i(i) -> a(0) If i & 1 # 0
28                  j(i) -> ab(1) If i & 2 = 0
29                  j(i) -> a(1) If i & 2 # 0
30                  k(i) -> ab(2) If i & 4 = 0
31                  k(i) -> a(2) If i & 4 # 0
32                  Xplace
33                      mag_and3[i(i),j(i),k(i)] -> o(i)
34                      mag_nand2[o(i),sel] -> ob(i)
35                  End
36              Repeat
37          End
```

[7] file `138.mod`

```
38      End
39  End
40
41  ChipPart ls138[a,b,c,g,gb(1:2)] -> ob(0:7)
42          { PCB interface }
43      Source a,b,c,g,gb
44      Pcb "dil16-supp 8 18 7 1"
45      Pin a,b,c,gb,g,ob(7),Gnd
46      Pin ob(6 : 0 By -1),Vdd
47          { local signal }
48      Signal ina,inb,inc,ing,ingb(1:2),inob(0:7)
49          { core generation }
50      channel(24)[]
51      Yplace
52      channel(6)[] Rotate90
53      decoder8[ina,inb,inc,ing,ingb(1:2)] -> inob(0:7)
54      channel(6)[] Rotate90
55      End
56          {pad ring generation }
57      Left
58      inpad(cmos)[b] -> inb
59      inpad(cmos)[a] -> ina
60      vddpad[]
61      outpad[inob(0)] -> ob(0)
62      Top
63      outpad[inob(1)] -> ob(1)
64      outpad[inob(2)] -> ob(2)
65      outpad[inob(3)] -> ob(3)
66      outpad[inob(4)] -> ob(4)
67      Right
68      outpad[inob(5)] -> ob(5)
69      outpad[inob(6)] -> ob(6)
70      gndpad[]
71      outpad[inob(7)] -> ob(7)
72      Bottom
73      inpad(cmos)[g] -> ing
74      inpad(cmos)[gb(2)] -> ingb(2)
75      inpad(cmos)[gb(1)] -> ingb(1)
76      inpad(cmos)[c] -> inc
77
78  End
79
80  Signal a,b,c,g,gb(1:2),ob(0:7)
81
82  ls138[a,b,c,g,gb(1:2)] -> ob(0:7)
```

Table 4.11: a whole chip: the 74C138

A part declared with the keyword `ChipPart` is generated by the chip generator. It belongs to the 2 worlds:

- Layout world: it is the top of a layout hierarchy,
- Pcb world: it is a leaf of a PCB hierarchy.

4.6.1 Layout finishing

The layout of a chip contains two parts:

- the core,
- the pad ring.

Core generation

The core layout is a tessellation (like the ones performed by `PavePart`) of few big subcells (like ROM, RAM, PLA and data-path) that must be connected with each others and with the input/output pads. The interconnections between these blocks and the pad ring will be performed by a channel router (for instance the **magic** one) thanks to the `.net` file provided by the netlist generator of **mod2mag**. Therefore it is important to manage some space for the channels. The parametrical part `channel` describe in `magicdef.mod` reservs some routing step space for each channels (lines 50, 52 and 54).

Pad ring generation

The pad ring is a ring which runs around the core and which contains the input/output pads (declared with the keyword `PadPart`). Every subcell of the chip (even if it is hidden by a `Part` call) is located at one side of the ring. The context can be changed to specify to which side a pad instance belongs. The statements `Top` (line 62), `Bottom` (line 72), `Left` (line 57) and `Right` (line 67) change this context. The order of pad instances is significant: it follows the trigonometric order around the pad ring.

The directory `~cad/ulm/mcmos/pad` contains all the files related to the pad generation. the `pad.mod` (line 2) provides a generic interface for the pads.

model generates only the differents block of the core and the pad ring as it can be observed on figure 4.12.

Each pad known by **mod2mag** possesses two layouts (see section C.2):

- the slim style, in order to maximize the horizontal pad density of the sides (used for IO–limited circuits),
- the flat style, in order to minimize the height of the circuit (used for core–limited circuits).

The pad ring generator choose independently the style for each side of the chip and select the best configuration in order to minize the global area.

The figure 4.13 shows the same layout with another pad configuration (a wider space has between asked for the routing channel).

We must then run **magic** to finish the circuit.

Chip finishing under magic

Three interventions are required under **magic** to finish the chip:

- adjust the fine position of the different blocks of the core (to avoid routing grid alignment problem). This is facultative if the core is a single data-path.
- to draw manually the power-supply from the pad ring to the different blocks of the core.

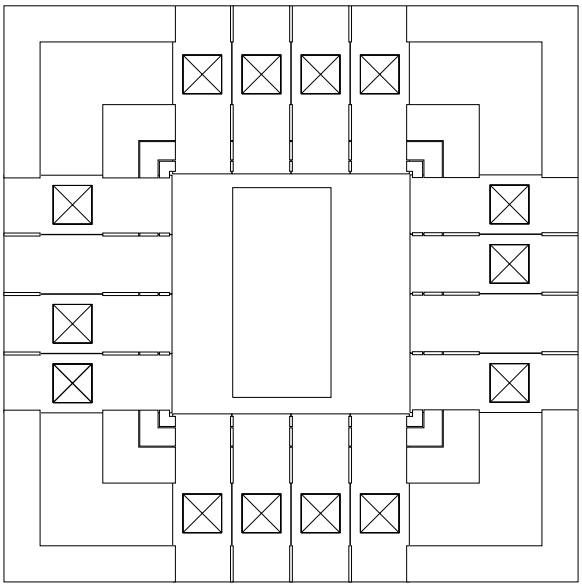


Figure 4.12: the unfinished layout

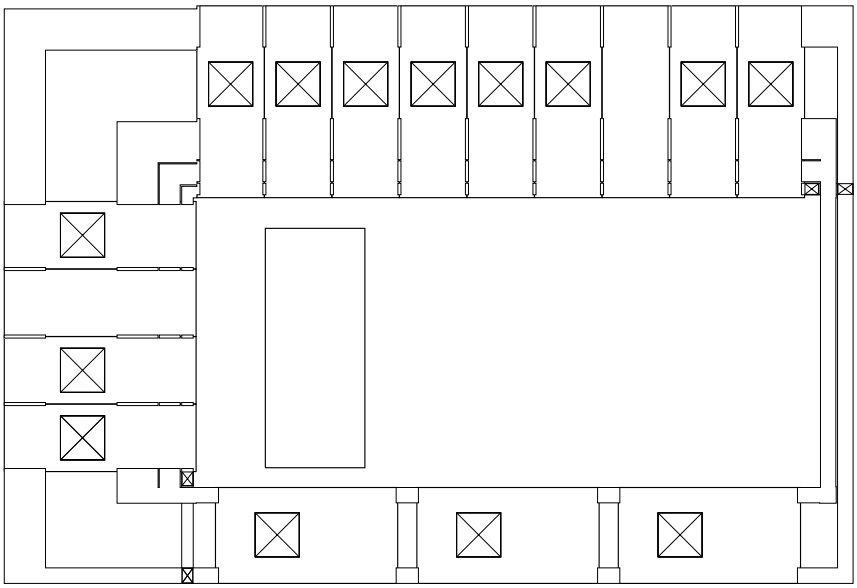


Figure 4.13: a different pad configuration

- to route the circuit:
  - first load the chip layout,
  - select the top cell,

– run the router `:route`.

The figure 4.14 shows the final result after the intervention under **magic**.

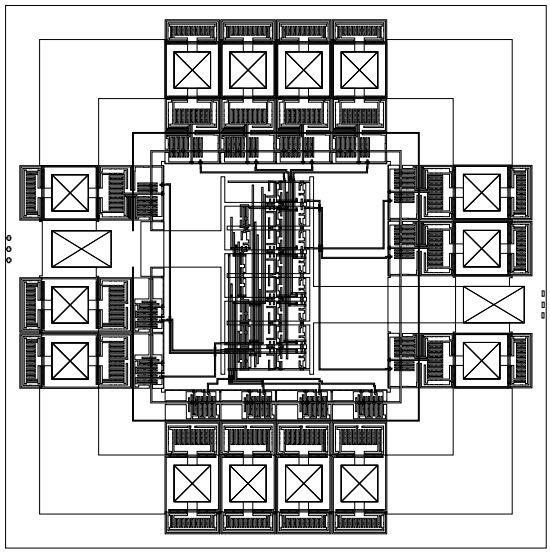


Figure 4.14: the final layout of the chip

4.6.2 PCB interface

The PCB interface describe the space required by a chip or a connector on a board. It must handle the two features (tab. 4.11 lines 44–46):

- The packaging definition,
- mapping between the input/output signals under **model** with the pin numbers of the chip.

Pcb statement

The `Pcb` statement has the following syntax:

`Pcb "`*string*`"`

and defines 5 parameters which are fields of the *string*. The first one represents the name of the package for the PCB system (the Dedale2000 software of Decad society). The four last parameters represents the bounding box (x0, y0, x1, y1) of the circuit relatively to the pin number 1 of the circuit.

Pin statement

The `Pin` statement has the following syntax:

`Pin` *pin list*

It enumerates the pins from the pin number 1 and associates them to a signal (taken from the inputs, the outputs and the power-supplies of the chip).

4.6.3 Declaration of existing circuits

In order to build boards and to simulate them with existing circuits, we have to provide a **model** interface at `ChipPart level` for these chips. Because they already exist, its is not necessary to describe the layout of the circuit. However, we must provide 3 interfaces:

- the ERC interface (`Source`, `Dest` and `Tristate`),
- the PCB interface (`Pcb` and `Pin`),
- the behavioral description of the circuit in order to perform simulation.

The table 4.12[8] shows this description for the TTL 374 circuit (an 8-bit register with output enable).

4.7 Pcb generation

Whole boards can be described thanks to the `PcbPart` keyword. The table 4.13[9] shows an example of board description.

Placement keywords (line 9) are significative to build the initial placement for the Pcb system. The command:

`model` *design* `-p`

generates for each `PcbPart` *name* used in the design the files:

- *name*`.pcbnet`: the input data file for Dedale2000 (tab. 4.14),
- `pcb_`*name*`.mag`: a **magic** file that contains the placement of the chips on the board (fig. 4.15).

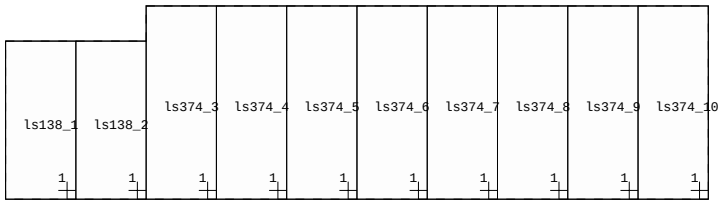


Figure 4.15: the chip placement file (`pcb_register_bank.mag`

[8] file `374.mod`
[9] `bank.mod`

```
1   Chippart c374[d(0:7),clk,enb] -> q(0:7)
2       Signal m(0:7)
3       Source d,clk,enb
4       Tristate q
5       Constant delay = 10
6
7       Pcb "dil20-supp 8 22 7 1"
8       Pin enb,q(0),d(0,1),q(1,2),d(2,3),q(3),Gnd
9       Pin clk,q(4),d(4,5),q(5,6),d(6,7),q(7),Vdd
10
11      Behavior
12          Capa 9 PF d
13          Capa 20 PF en,clk
14          Capa 12 PF q
15          Slew 60 PS/PF q
16          Undef m
17          Undef q
18          When enb Low Do
19              Set q = Value[m(0:7)]
20          Done
21          When enb High Do
22              HighZ q
23          Done
24          When clk High Do
25              At Time + delay * 1  NS Do
26                  Set m = Eval(Value[d(0:7)])
27              Done
28          Done
29          When m Change Do
30              If Value[enb] = 0 Then
31                  Set q = Value[m(0:7)]
32              Endif
33          Done
34          When enb,clk Undef Do
35              Undef q
36          Done
37      End
38  End
```

Table 4.12: the declaration of an existing circuit(`374.mod`)

```
1   Include "138.mod"
2   Include "374.mod"
3
4
5   PcbPart register_bank[d(0:8),ck,w(0:2),wen,wenb,r(0:2),ren,renb] -> out(0:7)
6
7       Signal wck(0:7),rs(0:7)
8       Integer i
9       Xplace
10          ls138[w,wen,wenb,ck] -> wck(0:7)
11          ls138[r,ren,renb,GND] -> rs(0:7)
12          ls374[d(0:7),wck(i),rs(i)] -> out(0:7) For i=0:7
13      End
14  End
```

Table 4.13: a whole board: a register bank(`bank.mod`)

```
/implantation
<P1> mg=<dil16-supp> x=12.700  y=3.810
<P2> mg=<dil16-supp> x=22.860  y=3.810
<P3> mg=<dil20-supp> x=33.020  y=3.810
<P4> mg=<dil20-supp> x=43.180  y=3.810
    ....
<P9> mg=<dil20-supp> x=93.980  y=3.810
<P10> mg=<dil20-supp> x=104.140 y=3.810
/alimentations 0V +5V
pal : P1,7 =0V
pal : P1,15 =+5V
pal : P2,7 =0V
pal : P2,15 =+5V
    ....
pal : P10,19 =+5V
pal : P2,5 =0V
/equipotentielles
<N0> P3,1 P2,15
<N1> P4,1 P2,14
<N3> P6,1 P2,12
....
<N30> P6,17 P5,17 P4,17 P3,17
<N31> P10,18 P9,18 P8,18 P7,18
<N31> P6,18 P5,18 P4,18 P3,18
```

Table 4.14: the input data file for the Pcb router (`register_bank.pcbnet`)

Appendix A

The constraints on cells in mcmos technology

Cells must verify a list of constraints in order to be used by the data-path generator. The constraints belong to the following classes:

- size of the cells,
- organization and reservation of the power-supply layer and the routing layers,
- specification and position of the cell inputs/outputs,
- side effects of the design rule check.

A cell library is available in the `~cad/ulm/mcmos/lib8` directory. The file `lib8.mod` contains all the **model** descriptions of the cells. However, it is important to describe the constraints that a cell must fulfill in order to the designer to develop new cells or to develop cells in another technologies. We are going to describe the constraints of our current technology (`mcmos`).

A.1 Cell size

the figure A.1 is an example of cell complying with the constraints. The height of the cells is constant: $150\lambda = 75.0\mu$m. It is possible to build constant height rows of cells. The length of the cells is variable in order to adapt the area to the need. These dimensions represent the reference rectangle (fig. A.1[1]) which must be defined by the `bb_p` label (for data-path cells, the reference rectangle is always different of the bounding-box because of design rule check reasons).

A.2 Power-supplies and routing layers

In order to facilitate the signal and power-supply interconnections between cells, a global organization of the routing layer *metal1* and *metal2* is required.

[1] `tribuf.mag`

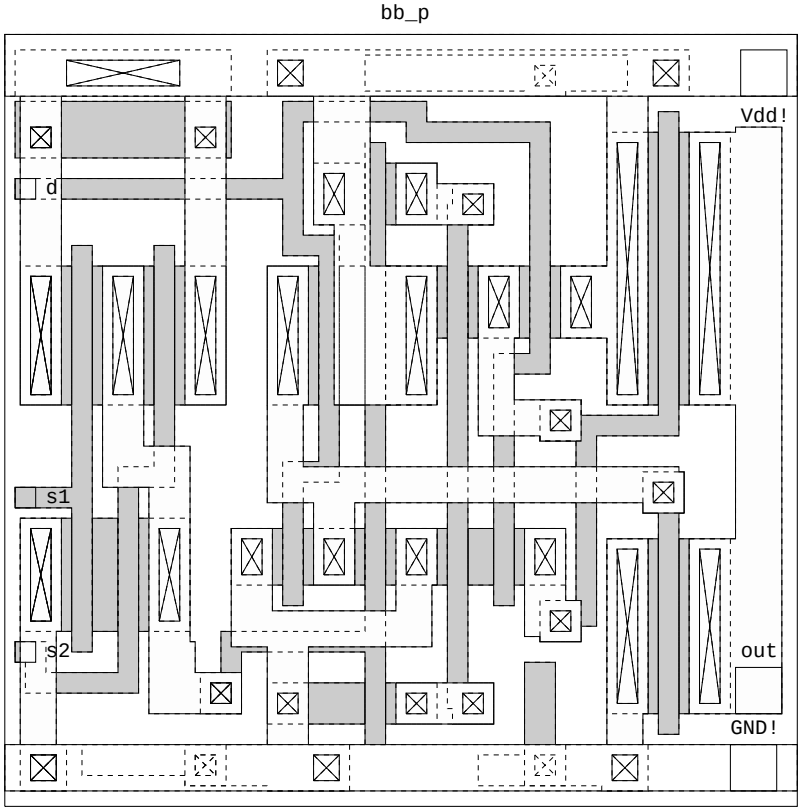


Figure A.1: A basic cell for the data-path generator

A.3 Power-supplies

Metal2 power-supplies are located along the horizontal borders of the cells. The positive one follows the top side on the whole length of the cell. It is a $12\lambda = 6.0\mu$m width *metal2* rail (fig. A.2). The ground power–supply run across the cell in a $9\lambda = 4.5\mu$m width *metal2* rail, located at $3\lambda = 1.5\mu$m from the bottom of the cell.

Cell rows of even rank are used with an X axis symmetry. Power-supply rails can be shared between consecutives rows. The positive rails of adjacent odd/even rows are directly connected because they follow the side of the cells. Ground power-supply rails of the adjacent even/odd rows are not connected, in order to insert some extra horizontal metal2 channels. However, if no extra channels are required, the gap between the two rails is filled with *metal2*. Power–supplies are identified by labels: `Vdd!` for the positive one and `GND!` for the ground (fig A.1).

A.4 Routing organization

The *metal2* power-supply rails of the cells lead to a global organization of the power-supply for the whole data-path: two *metal2* combs (one for each power-supplies) are nested horizontally. *Metal2* layer is reserved for the horizontal connections. In the same way, vertical connections are drawn in *metal1*. *Metal2* wires can run over the cells. The corresponding space must be

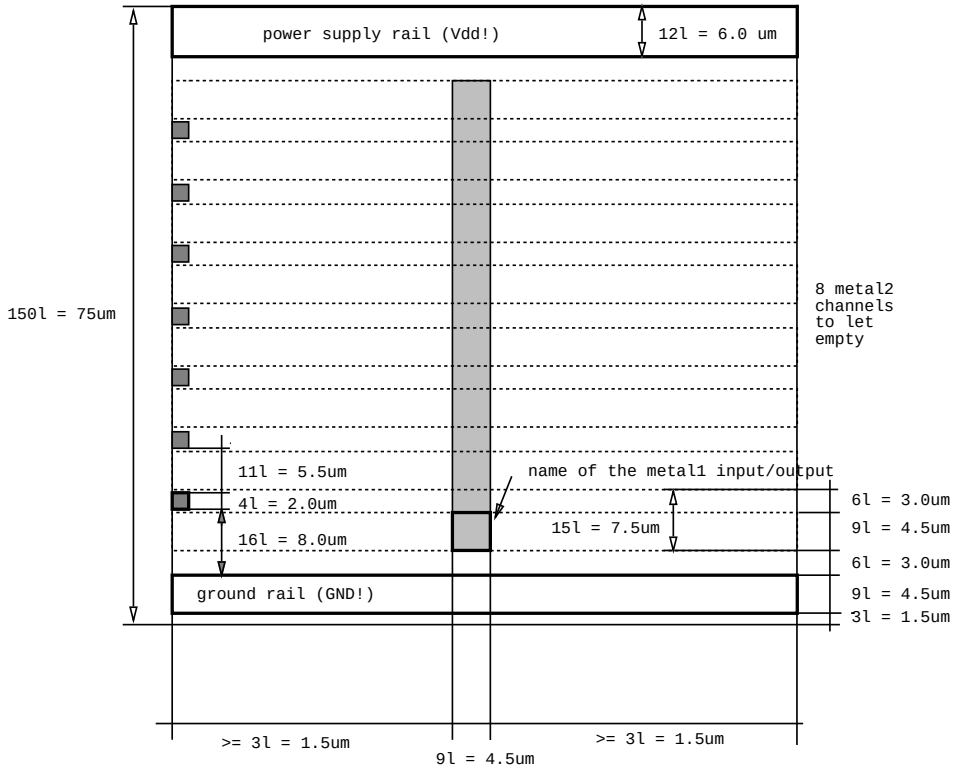


Figure A.2: A basic cell for the data-path generator

number	*lower end*		*upper end*	
	λ	μm	λ	μm
1	18	9.0	27	13.5
2	33	16.5	42	21.0
3	48	24.0	57	28.5
4	63	31.5	72	36.0
5	78	39.0	87	43.5
6	93	46.5	102	51.0
7	108	54.0	117	58.5
8	123	61.5	132	66.0

Table A.1: the location of the *metal2* channels from the bottom of the cell

free of `metal2` in the cells. 8 connections (in the `lib8` library) can run through the cells and be possibly connected to their inputs/outputs. The table A.1 specifies the location of these channels inside the cell. The metal2 channels are drawn by the data-path generator: *metal2* is prohibited inside the cells unless for the power-supplies. The figure A.2 shows the locations of these channels.

number	*lower end*		*upper end*	
	λ	μm	λ	μm
1	13	6.5	17	9.5
2	28	14.0	32	16.0
3	43	21.5	47	23.5
4	58	29.0	62	31.0
5	73	36.5	77	38.5
6	88	44.0	92	46.0
7	103	51.5	107	53.5
8	118	59.0	122	61.0
9	133	66.5	137	68.5

Table A.2: the location of the *polysilicon* inputs/outputs in a cell

Sometimes the 8 channels are not sufficient to route the horizontal connections; the required space is then allocated beyond the ground side of the cell.

Vertical connections are achieved in *metal1* at the left of each cell column: the *metal1* layer is available to achieve internal connections in the cells (constraints on this layers will be defined below).

A.5 Definition of the inputs/outputs of the cells

Two technics are available (fig. A.2):

- *polysilicon* connection on the left side of the cell,
- *metal1* vertical rail.

A.5.1 Polysilicon input/output

They are mainly used mainly for the input of the cells (quite always directly connected to transistor gates). They are defined by a $4\lambda = 2.0\mu$m width square of *polysilicon*. The boundary of the label must located at the left side of the cell. There exists 9 possible locations for these inputs (fig. A.2) and are precisely defined in the table A.2. These locations fit with the 9 gaps between the `metal2` rails in order to avoid stacking rule errors.

A.5.2 Metal1 input/output

Other inputs/outputs are drawn with a $9\lambda = 4.5\mu$m *metal1* vertical rail (fig. A.2). The ends of each rail are located at $12\lambda = 6.0\mu$m away from the horizontal side of the cell and fit the ends of the external *metal2* channels. This rail must be moved away from the different contacts and boundaries of layers in order to prevent every stacking rule errors on the vias which could be drawn at every intersections of the rail and the `metal2` channels. A good test consists in drawing all the vias and check with the DRC.

A $9\lambda = 4.5\mu$m space is required between 2 rails to guarantee a minimum distance between vias.

Lower and upper sides of the cells must be free of `metal1` to allow the extension of every *metal1* rail. This extension is required in the following cases:

- The connection of an input/output to a power–supply. It is achieved by a via on the power–supply rail at one side of the cell.
- The connection of 2 inputs/outputs of adjacent cells if the positions of the rails fit.

Consequently, a $21\lambda = 10.5\mu$m gap free of `metal1` must be centered on the axis of each input/output rail.

An input-output rail is declared by a square label at its lower end. It must be located at a minimum distance of $3\lambda = 1.5\mu$m of one of the vertical sides of the cell.

A.6 Design rule constraints

The cells must fulfill the design rule for the technology for themselves and also for possible side-effect with their neighbor. The data-path compiler makes the assumption that as soon as cells do not overlap, design rules are expected.

For the side effects, only spacing and stacking rules are important. Stacking rules happen when connecting a *metal1* rail to a external *metal2* channel with a via : it is easy to check every combinaison with the **magic** DRC.

The spacing rule can be classified in two kinds:

- between diffusions which are n or p-typed,
- between other layers.

For non-typed layers, spacing rules will be independent of the side type. if n is the distance required between 2 objects of the same layer then the distance between every object of this layer and the sides must be at least $\lceil n/2 \rceil$.

In order to limit the constraints, it is important to adapt the rules to the sides of the cell:

- the top side, under the Vdd power–supply rail. In this area, we will find logically most p-transistor. By the parity inversion of the rows of the data-path, each cell is suppose to touch by his top side, the top side of another cell. The spacing rule for top side-effect is calculated for a n-well environment. Spacing rules for non-typed layer are
- the bottom side, under the GND power–supply rail. In this area, we will find logically most n-transistor. By the parity inversion of the rows of the data-path, each cell is suppose to touch by his bottom side, the bottom side of another cell. The spacing rule for top side-effect is calculated for a p-substrate environment.
- the vertical side which are supposed to be connected to vertical routing channels. The assumption (which is right for the lib8 cell) is that those channel is wide enough(it contains at least one wire) to isolate from the next horizontal cell (as each cell has at least one *poly* input). Side effect are ignore for typed layers.

The table A.3 summarize the distance to observe for every layers and sides.

masks	*lower side*		*upper side*		*vertical side*	
	λ	μm	λ	μm	λ	μm
polysilicon	2	1.0	2	1.0	2	1.0
metal1	3	1.5	3	1.5	3	1.5
n diffusion	4	2.0	16	8.0	–	–
p diffusion	16	8.0	4	2.0	–	–
p body-tie	4	2.0	12	6.0	–	–
n body-tie	10	5.0	4	2.0	–	–

Table A.3: drc constraints between layers depending on the side of the cell

Appendix B

The constraints on cells in ecpd technology

Cells must verify a list of constraints in order to be used by the data-path generator. The constraints belong to the following classes:

- size of the cells,
- organization and reservation of the power-supply layer and the routing layers,
- specification and position of the cell inputs/outputs,
- side effects of the design rule check.

A cell library is available in the `~cad/ulm/ecpd/lib8` directory. The file `lib8.mod` contains all the **model** descriptions of the cells. However, it is important to describe the constraints that a cell must fulfill in order to the designer to develop new cells or to develop cells in another technologies. We are going to describe the constraints of our current technology (`ecpd`).

B.1 Cell size

the figure B.1 is an example of cell complying with the constraints. The height of the cells is constant: $160\lambda = 64.0\mu$m. It is possible to build constant height rows of cells. The length of the cells is variable in order to adapt the area to the need. These dimensions represent the reference rectangle (fig. B.1[1]) which must be defined by the `bb_p` label (for data-path cells, the reference rectangle is always different of the bounding-box because of design rule check reasons).

B.2 Power-supplies and routing layers

In order to facilitate the signal and power-supply interconnections between cells, a global organization of the routing layer *metal1* and *metal2* is required.

[1] `tribuf.mag`

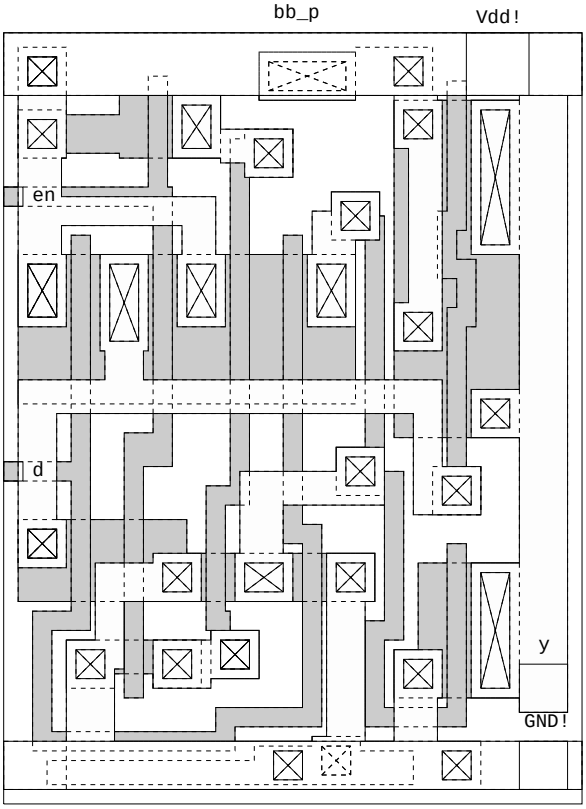


Figure B.1: A basic cell for the data-path generator

B.3 Power-supplies

Metal2 power-supplies are located along the horizontal borders of the cells. The positive one follows the top side on the whole length of the cell. It is a $13\lambda = 5.2\mu$m width *metal2* rail (fig. B.2). The ground power–supply run across the cell in a $10\lambda = 4.0\mu$m width *metal2* rail, located at $3\lambda = 1.2\mu$m from the bottom of the cell.

Cell rows of even rank are used with an X axis symmetry. Power-supply rails can be shared between consecutives rows. The positive rails of adjacent odd/even rows are directly connected because they follow the side of the cells. Ground power-supply rails of the adjacent even/odd rows are not connected, in order to insert some extra horizontal metal2 channels. However, if no extra channels are required, the gap between the two rails is filled with *metal2*. Power–supplies are identified by labels: `Vdd!` for the positive one and `GND!` for the ground (fig B.1).

B.4 Routing organization

The *metal2* power-supply rails of the cells lead to a global organization of the power-supply for the whole data-path: two *metal2* combs (one for each power-supplies) are nested horizontally. *Metal2* layer is reserved for the horizontal connections. In the same way, vertical connections are drawn in *metal1*. *Metal2* wires can run over the cells. The corresponding space must be

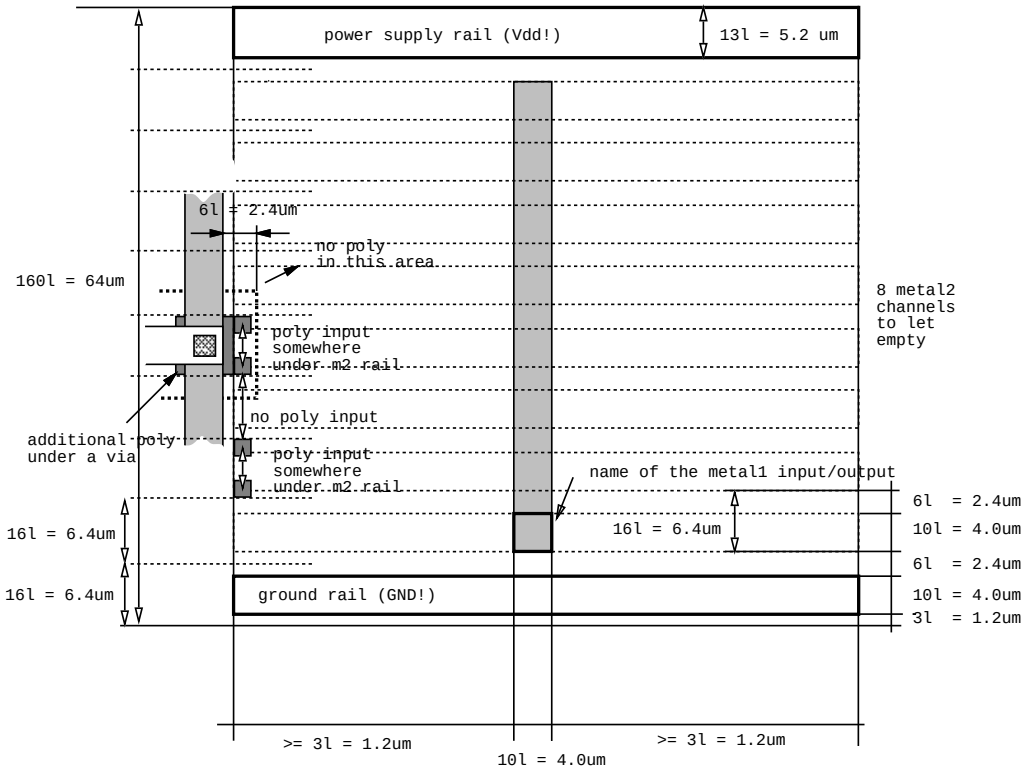


Figure B.2: A basic cell for the data-path generator

number	*lower end*		*upper end*	
	λ	μm	λ	μm
1	19	7.6	29	11.6
2	35	14.0	45	18.0
3	51	20.4	61	24.4
4	67	26.8	77	30.8
5	83	33.2	93	37.2
6	99	39.6	109	43.6
7	115	46.0	125	50.0
8	131	52.4	141	56.4

Table B.1: the location of the *metal2* channels from the bottom of the cell

free of `metal2` in the cells. 8 connections (in the `lib8` library) can run through the cells and be possibly connected to their inputs/outputs. The table B.1 specifies the location of these channels inside the cell. The metal2 channels are drawn by the data-path generator: *metal2* is prohibited inside the cells unless for the power-supplies. The figure B.2 shows the locations of these channels.

number	*location*	
	λ	μm
1	16	6.4
2	32	12.8
3	48	19.2
4	64	25.6
5	80	32
6	96	38.4
7	112	44.8
8	128	51.2
9	144	57.6

Table B.2: the *metal2* pitches

Sometimes the 8 channels are not sufficient to route the horizontal connections; the required space is then allocated beyond the ground side of the cell.

Vertical connections are achieved in *metal1* at the left of each cell column: the *metal1* layer is available to achieve internal connections in the cells (constraints on this layers will be defined below).

B.5 Definition of the inputs/outputs of the cells

Two technics are available (fig. B.2):

- *polysilicon* connection on the left side of the cell,
- *metal1* vertical rail.

B.5.1 Polysilicon input/output

They are mainly used mainly for the input of the cells (quite always directly connected to transistor gates). They are defined by a $4\lambda = 1.6\mu$m width square of *polysilicon*. The boundary of the label must located at the left side of the cell. As the pitch of *vias* is to small to let run a *poly* wire between 2 vias, the only for a *poly* wire to cross a *metal1* rail is to put the *poly* wire on the same axis as the *metal2* wires. The `poly` wire is enlarged if a via is put on the `metal2` wire. At the left of the cell, the vertical channel may enlarged the *poly wire* to run it under vias. In order to avoid side effects, *poly* inputs may float between 2 consecutive `metal2` pitches (see tab. B.2) but are not allowed to overhang over the pitches. If a *poly* input uses the gap between the pitches n and $n+1$ the neighborhood gaps ($n-1$ — n and $n+1$ — $n+2$) must be free of *poly* inputs. Furthermore, *poly* wires inside the cells must be at least $6\lambda = 2.4\mu$m of the left side in those gap (see fig. B.2).

B.5.2 Metal1 input/output

Other inputs/outputs are drawn with a $10\lambda = 4.0\mu$m *metal1* vertical rail (fig. B.2). The ends of each rail are located at $19\lambda = 7.6\mu$m away from the horizontal sides of the cell and fit the

ends of the external *metal2* channels. This rail must be moved away from the different contacts and boundaries of layers in order to prevent every stacking rule errors on the vias which could be drawn at every intersections of the rail and the `metal2` channels. A good test consists in drawing all the vias and check with the DRC.

A $10\lambda = 4.0\mu$m space is required between 2 rails to guarantee a minimum distance between vias.

Lower and upper sides of the cells must be free of `metal1` to allow the extension of every *metal1* rail. This extension is required in the following cases:

- The connection of an input/output to a power–supply. It is achieved by a via on the power–supply rail at one side of the cell.
- The connection of 2 inputs/outputs of adjacent cells if the positions of the rails fit.

Consequently, a $22\lambda = 8.8\mu$m gap free of `metal1` must be centered on the axis of each input/output rail.

An input/output rail is declared by a square label at its lower end. It must be located at a minimum distance of $3\lambda = 1.2\mu$m of one of the vertical sides of the cell.

B.6 Design rule constraints

The cells must fulfill the design rule for the technology for themselves and also for possible side-effect with their neighbor. The data-path compiler makes the assumption that as soon as cells do not overlap, design rules are expected.

For the side effects, only spacing and stacking rules are important. Stacking rules happen when connecting a *metal1* rail to a external *metal2* channel with a via : it is easy to check every combinaison with the **magic** DRC.

The spacing rule can be classified in two kinds:

- between diffusions which are n or p-typed,
- between other layers.

For non-typed layers, spacing rules will be independent of the side type. if n is the distance required between 2 objects of the same layer then the distance between every object of this layer and the sides must be at least $\lceil n/2 \rceil$.

In order to limit the constraints, it is important to adapt the rules to the sides of the cell:

- the top side, under the Vdd power–supply rail. In this area, we will find logically most p-transistor. By the parity inversion of the rows of the data-path, each cell is suppose to touch by his top side, the top side of another cell. The spacing rule for top side-effect is calculated for a n-well environment. Spacing rules for non-typed layer are
- the bottom side, under the GND power–supply rail. In this area, we will find logically most n-transistor. By the parity inversion of the rows of the data-path, each cell is suppose to touch by his bottom side, the bottom side of another cell. The spacing rule for top side-effect is calculated for a p-substrate environment.
- the vertical side which are supposed to be connected to vertical routing channels. The assumption (which is right for the lib8 cell) is that those channel is wide enough(it contains

masks	*lower side*		*upper side*		*vertical side*	
	λ	μm	λ	μm	λ	μm
polysilicon	3	1.2	3	1.2	3	1.2
metal1	3	1.2	3	1.2	3	1.5
n diffusion	4	1.6	20	8.0	–	–
p diffusion	20	8.0	4	2.0	–	–
p body-tie	4	1.6	20	8.0	–	–
n body-tie	8	3.2	4	2.0	–	–

Table B.3: drc constraints between layers depending on the side of the cell

at least one wire) to isolate from the next horizontal cell (as each cell has at least one *poly* input). Side effect are ignore for typed layers.

The table B.3 summarize the distance to observe for every layers and sides.

Appendix C

Customization of model

Unfortunately, the technology rules is site-dependent. All the examples provided in this manual has been developed in the `mcmos` technology (which is the ECDM20 process of the European Silicon Structure company). You probably use another technology. The goal of this chapter is to help the customization of the **model** environment according to your technology.

Using a new technology requires:

- to define the new configuration file for **mod2mag**,
- to design the libraries in the new technology, particularly a pad library and a data-path cell library.

C.1 Configuration file

The configuration file is mainly used by the data-path generator. However, this file is read as soon the `-m` flag is set. This file must be defined even if the data-path compiler is not used.

The configuration file is located in `~cad/lib/magic/`*technology*`/model`. For our technology `mcmos`, our 8-channel style file is shown in the table C.1.

The `metal1 pitch` and `metal2 pitch` respectively represents the horizontal and vertical routing pitches: they must also comply with the via spacement rules (this explains the difference between the `pitch` value and the sum of the `size` and `distance` values.

Furthermore, the `metal1 pitch` and the `metal2 pitch` must be equal to the routing pitch of **magic** (as defined in the `.tech` file), in order to use the **magic** router.

The number of *metal2* channels (included the 2 power–supplies channels) is defined by the ratio between the `cell size` and the `metal2 pitch`:

$$(channel\ number + 2) \times metal2\ pitch = cell\ size$$

In order to connects the polysilicon inputs to vertical metal1 wires, the router have to horizontally extend the polysilicon port. Those polysilicon connection can collide with vias. To avoid this stacking rule constraints, two strategies can be used :

- if the metal2 pitch is wide enough to let a *poly* wire run between 2 vias (fig. C.1.a), then *poly* inputs are interlaced with *metal2* rails. This is specified by the command :
  `polyvia onaxis = 0`

```
*********************************************************
*   8-channel style file for Model in mcmos technology  *
*********************************************************
metal2 name = metal2
metal2 pitch = 15
metal2 size = 7
metal2 distance = 6
metal1 name = metal1
metal1 pitch = 15
metal1 size = 7
metal1 distance = 5
poly name = polysilicon
poly size = 4
polycontact name = polycontact
polycontact size = 8
via name = via
via size = 9
cell size = 150
power size = 12
```

Table C.1: example of *style* file

This leads to a given style for the location of *poly* inputs of the cells (see techno `mcmos` in section A.5.1).

- Else not, via and polysilicon are aligned on the same grid. This is specified by the command :
  `polyvia onaxis = 1`
  When a via must be stacked on a polysilicon wire, polysilicon square is drawn around the via location to move away the polysilicon boundary. The command :
  `polyvia oversize = 3`
  defines the overhang of polysilicon under the vias (fig. C.1.b). This leads to a given style for the location of *poly* inputs of the cells (see techno `ecpd` in section B.5.1).

When a new configuration file has been created, it is important to publish the new constraints in the new technology. The LaTeX source of the two first appendices is available in `~cad/ulm/src/tex/mcmos.tex` and in `~cad/ulm/src/tex/ecpd.tex`. It is up to the local **model** maintainer to copy and modify it in order to describe the new constraints on the cell for the data-path generator for every new technology.

It is also important to develop in the new technology few basic cells complying with the new constraints in order to provide examples for the users.

C.2 Development of a new pad family

If a new technology (*newtechno*) is used, its important to provide a pad library for the user. The convention is to locate this pad library in the `~cad/ulm/`*newtechno*`/pad` directory. This

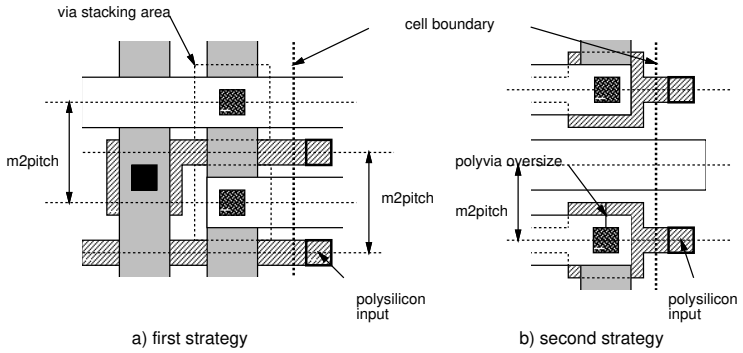


Figure C.1: via/polysilicon stacking rule problem

directory should contain:

- the layouts (`.mag`) of the pads (see below),
- their extracted netlist (`.ext`),
- their **model** description (`.mod`) (automatically generated by **ext2mod**),
- a general **model** pad file which provides a generic interface for the pads (`pad.mod`).

A pad `mag_`*name* (declared by the `PadPart` keyword) must possess two layouts (fig. C.2):

- *name*`.mag`, a slim layout for pad ring limited chips (left of fig. C.2).
- `flat_`*name*`.mag`, a flat layout for core limited chips (right of fig. C.2).

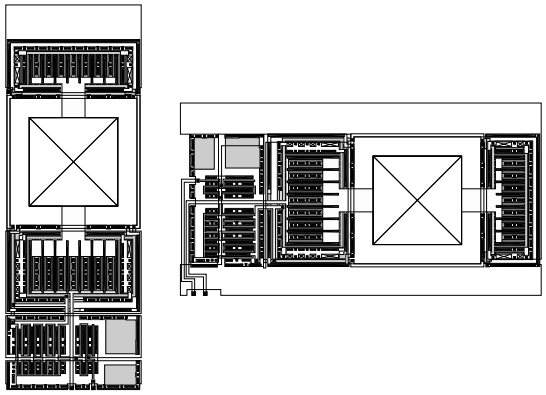


Figure C.2: the two layouts of a pad

The different sides of the pad ring are generated with compliance to the following constraints:

- the size of the hole of the pad ring must greater than the side of the core (there is enough room in the center of the ring to contain the core of the chip);

- the whole circuit occupies the smallest area.

This optimization leads to 3 generation styles for each sides (independently):

- empty : no pad is present,
- flat : there few pads : the circuit is core–limited.
- slim : there numerous pads : the circuit is pad–limited.

empty, flat and slim. Pads are regularly disposed along the side. Connections between the pads are drawn automatically thanks to layout cells defined in the `corner.mag` shown on figure C.3.

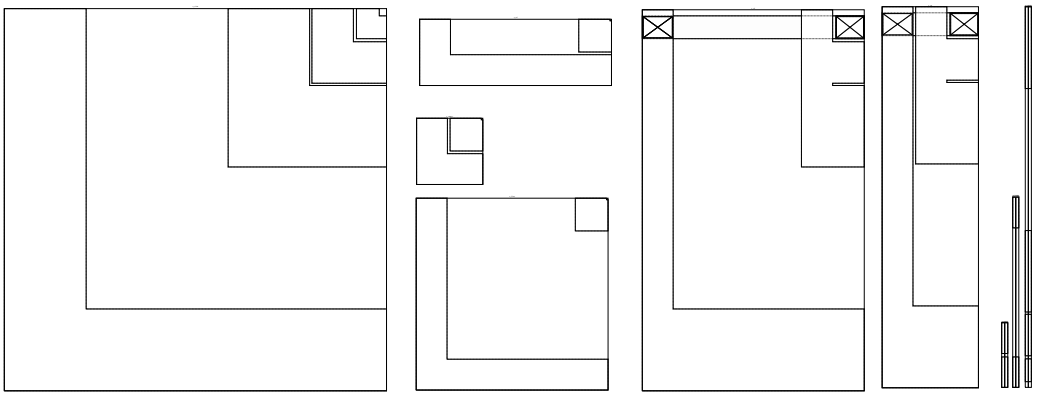


Figure C.3: the corner layouts

3 tiles defines the section of the ring side according to the style:

- `c_eside` is the side section of the empty style.
- `c_fside` is the side section of the flat style.
- `c_sside` is the side section of the slim style.

Each tile is cut by a `stretch` label which indicates the section where to stretch the tile if necessary.

6 other tiles are the corners between the different ring styles:

- `c_empty` between two empty sides.
- `c_flat` between two flat sides.
- `c_slim` between two slim sides.
- `c_ef` between an empty side and a flat side.
- `c_se` between a slim side and an empty side.
- `c_sf` between a slim side and an flat side.

It is up to the local **Mod2mag** maintainer to develop the new `corner` cell and also to provide a minimal library of pads.

Appendix D

Layout Libraries

This appendix presents the layout libraries for the mcmos technology (the ECDM20 process of the ES2 company). Layouts are organized in 4 groups :

- basic cells (logic gates and flip-flops) compatible with the Data-Path compiler. Layouts and **model** descriptions are located in the directory `~cad/ulm/mcmos/lib8`. The first section of this appendix is dedicated to them;
- parametrical fan-in/fanout buffers `~cad/ulm/mcmos/lib8/niceinv.mod`,
- input/output pads. They are located in the directory `~cad/ulm/mcmos/pad`;
- the PLA generator (`~cad/ulm/mcmos/pla`).

D.1 Mcmos Lib8 Library

The `lib8.mod` contains the basic library cell compatible with the data-path compiler. It contains logic gates and flip-flops (table D.1).

D.1.1 The `logic.mod` Library

```
Include "lib8.mod"
Part nand [i(0:*)] -> o
Part and [i(0:*)] -> o
Part nor [i(0:*)] -> o
Part or [i(0:*)] -> o
```

The library `~cad/ulm/mcmos/operators/logic.mod` contains generic cells describing the 4 logic gates `nand`, `nor`, `and` and `or`. These logic gates are recursively built with the basic cells of the `lib8.mod` library. If the number of inputs is large, the cells are built with several gates which are layout along the x-axis.

cell	*function*
	buffers & inverters
`inv`	an inverter
`biginv`	an inverter buffer
`buf`	a buffer
`tribuf`	a tristate buffer
	basic logic gates
`nand2`	a 2-input nand
`nand3`	a 3-input nand
`nand4`	a 4-input nand
`and2`	a 2-input and
`and3`	a 3-input and
`and4`	a 4-input and
`nor2`	a 2-input nor
`nor3`	a 3-input nor
`or2`	a 2-input or
`or3`	a 3-input or
`xor2`	a 2-input xor
`xnor2`	a 2-input xnor
`oax2`	an or and xor 2-input gate
	multiplexer
`mux21`	a 2-to-1 multiplexer
	latches & flip-flops
`gff`	a latch
`gffb`	a latch
`dff`	a D-flipflop
`dffb`	a D-flipflop
	adder basic cells
`pg`	a propagate/generate gate
`composepg`	composition cell for look ahead
`addcell`	an adder cell
`addcell1`	an adder cell (odd position)
`addcarry`	an adder cell
`addcarry1`	an adder cell (odd position)

Table D.1: the cells of the `lib8.mod` library

D.1.2 `Niceinv.mod`: parametrical buffers

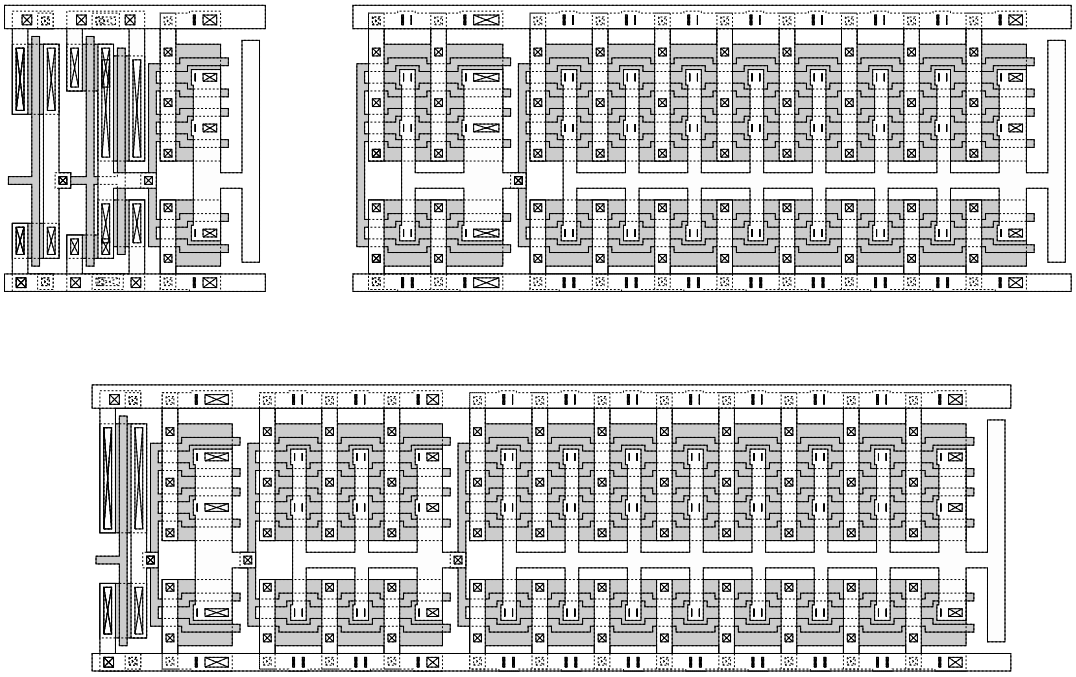
```
Part niceinv(ci,co) [i] -> o
Part nicebuf(ci,co) [i] -> o
```

The library `~cad/ulm/mcmos/lib8/niceinv.mod` contains the definition of the two parts:

- `niceinv`, a parametrical fan-in/fanout inverter;
- `nicebuf`, a parametrical fan-in/fanout buffer;

The parameters c_i and c_o represent the equivalence expected capacitances of the input of the part and of the load of the output. 10 units correspond to the input capacitance of an inverter with a 2μm$\times 10\mu$m n-transistor and a 2μm$\times 20\mu$m p-transistor.

According to the c_i/c_o ratio, an odd (resp. even) number n is chosen as number of basic inverters required to build the final inverter (resp. buffer). The fanout of each basic inverter follow a geometric sequence such that the input/output capacitances complies with the constraints



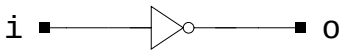
D.1.3 Cell Inv: an inverter

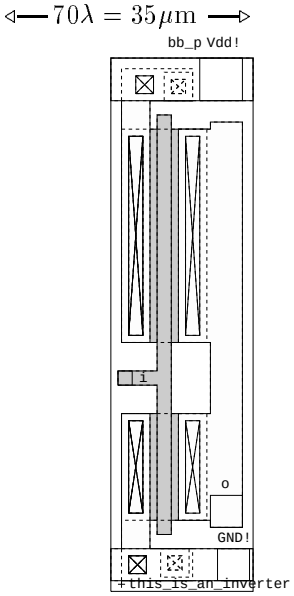
`MagicPart mag_inv [i] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.35
Inputs	*Capacitance*	Δt ns
i	0.09 pF	0.26

Description





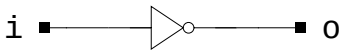
D.1.4 Cell Biginv: an inverter buffer

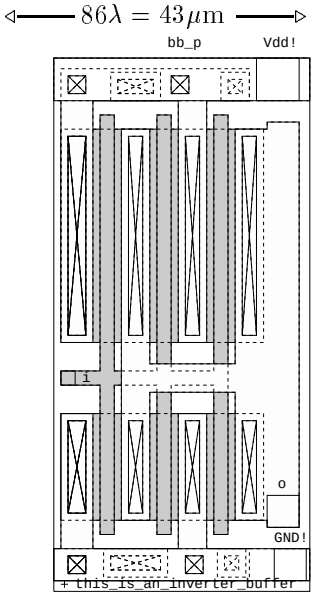
`MagicPart mag_biginv [i] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	0.78
Inputs	*Capacitance*	Δt ns
i	0.28 pF	0.24

Description





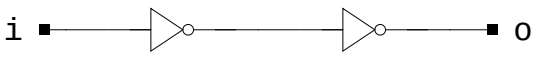
D.1.5 Cell Buf: a buffer

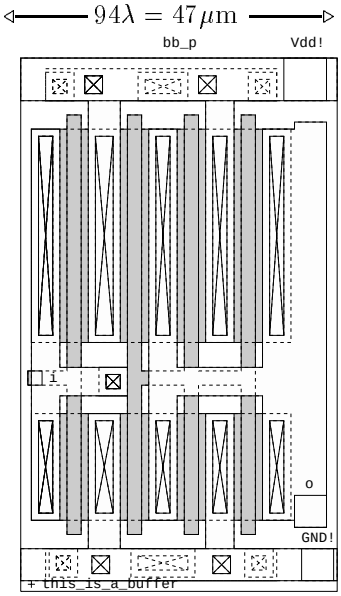
`MagicPart mag_buf [i] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	0.78
Inputs	*Capacitance*	Δt ns
i	0.09 pF	1.17

Description





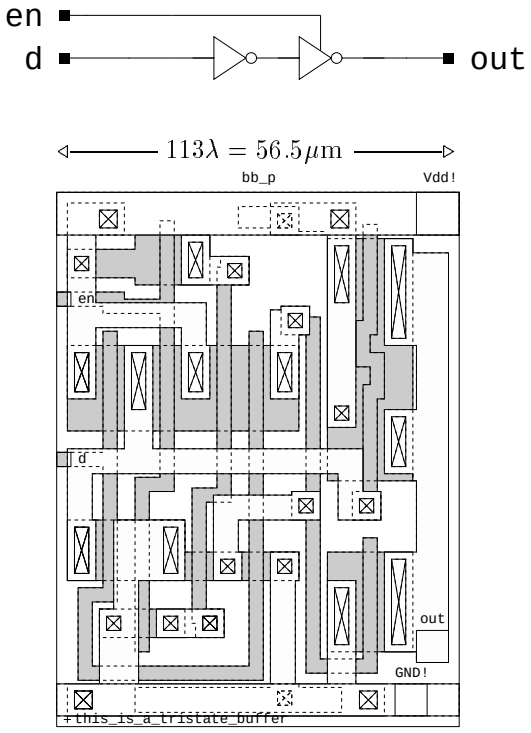
D.1.6 Cell Tribuf: a tristate buffer

`MagicPart mag_tribuf [d,en] -> out`

Inputs and Outputs

	Outputs	`out`
	$\Delta t/\Delta C$ (ns/pF)	1.85
Inputs	*Capacitance*	Δt ns
`d`	0.08 pF	1.39
`en`	0.07 pF	0.24
`out`	0.13 pF	0.00

Description



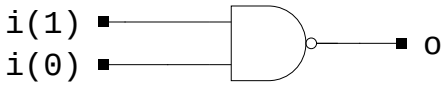
D.1.7 Cell Nand2: a 2-input nand

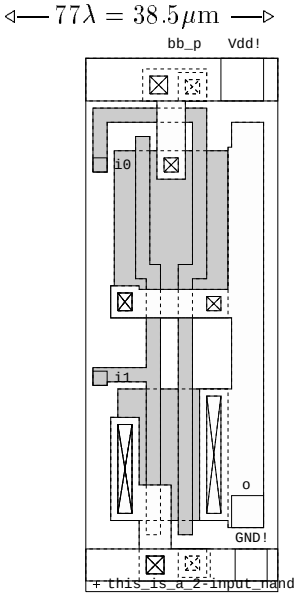
`MagicPart mag_nand2 [i(0:1)] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.82
Inputs	*Capacitance*	Δt ns
i(0)	0.09 pF	0.54
i(1)	0.09 pF	0.54

Description





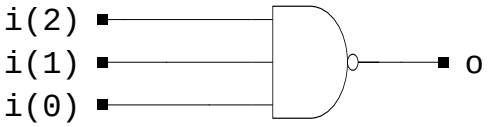
D.1.8 Cell Nand3: a 3-input nand

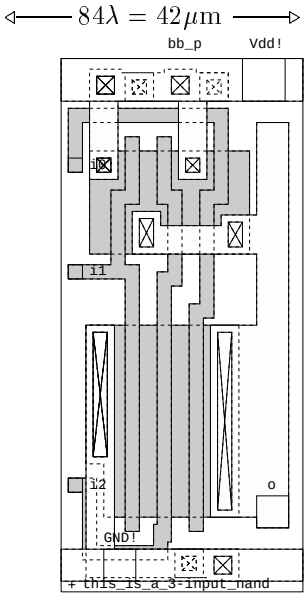
`MagicPart mag_nand3 [i(0:2)] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	4.41
Inputs	*Capacitance*	Δt ns
i(0)	0.09 pF	0.90
i(1)	0.09 pF	0.90
i(2)	0.09 pF	0.90

Description





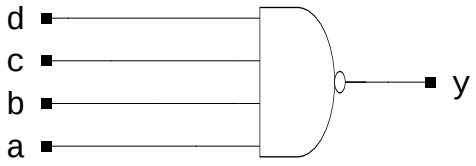
D.1.9 Cell Nand4: a 4-input nand

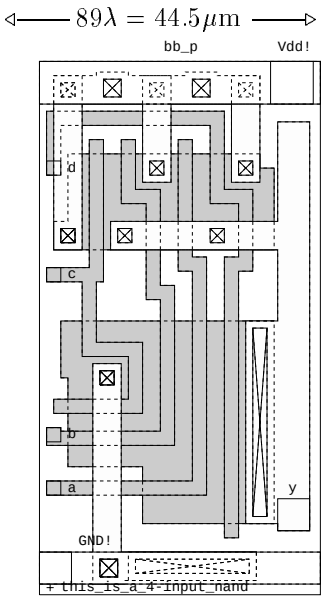
`MagicPart mag_nand4 [a,b,c,d] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	5.22
Inputs	*Capacitance*	Δt ns
a	0.09 pF	1.41
b	0.10 pF	1.41
c	0.08 pF	1.41
d	0.10 pF	1.41

Description





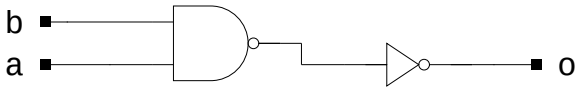
D.1.10 Cell And2: a 2-input and

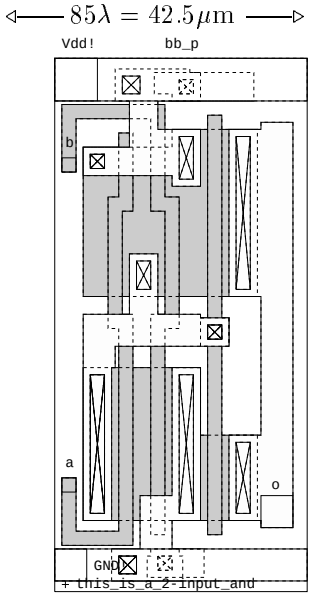
`MagicPart mag_and2 [a,b] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	3.00
Inputs	*Capacitance*	Δt ns
a	0.09 pF	1.01
b	0.09 pF	1.01

Description





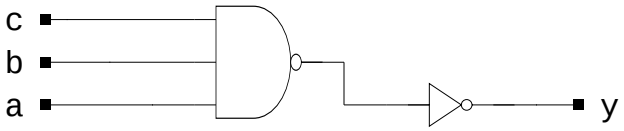
D.1.11 Cell And3: a 3-input and

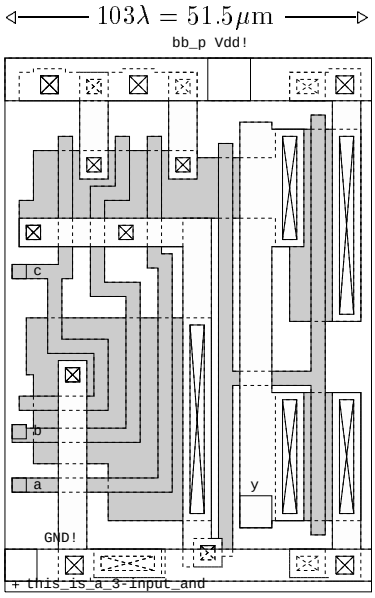
`MagicPart mag_and3 [a,b,c] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	1.98
Inputs	*Capacitance*	Δt ns
a	0.09 pF	2.13
b	0.10 pF	2.13
c	0.08 pF	2.13

Description





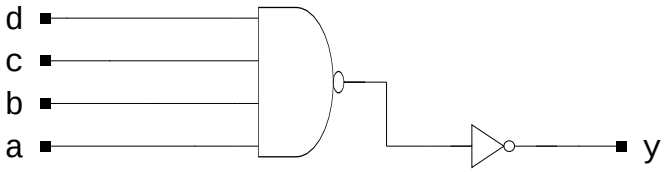
D.1.12 Cell And4: a 4-input and

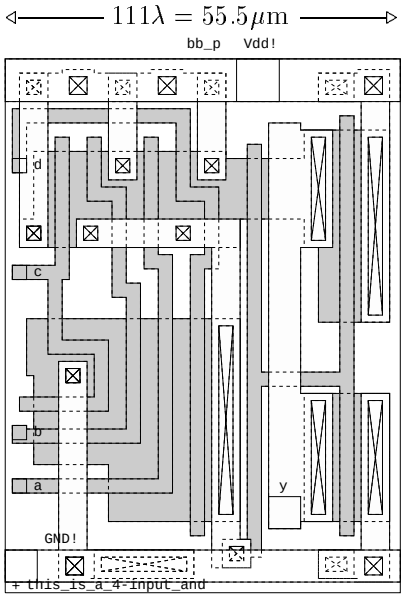
`MagicPart mag_and4 [a,b,c,d] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	1.98
Inputs	*Capacitance*	Δt ns
a	0.09 pF	2.29
b	0.10 pF	2.29
c	0.08 pF	2.29
d	0.10 pF	2.29

Description





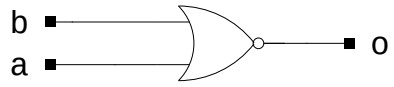
D.1.13 Cell Nor2: a 2-input nor

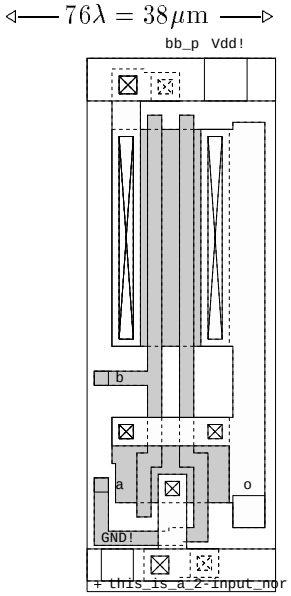
`MagicPart mag_nor2 [a,b] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	4.62
Inputs	*Capacitance*	Δt ns
a	0.09 pF	0.78
b	0.09 pF	0.78

Description





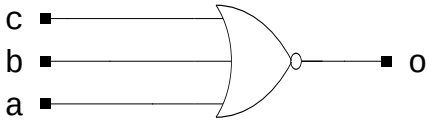
D.1.14 Cell Nor3: a 3-input nor

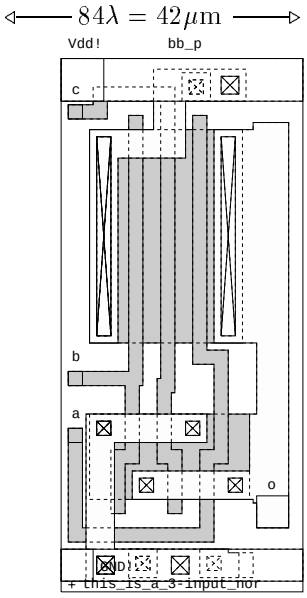
`MagicPart mag_nor3 [a,b,c] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	7.05
Inputs	*Capacitance*	Δt ns
a	0.09 pF	1.39
b	0.09 pF	1.39
c	0.09 pF	1.39

Description





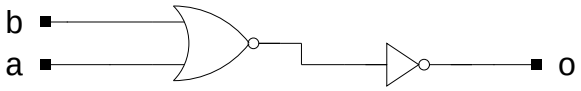
D.1.15 Cell Or2: a 2-input or

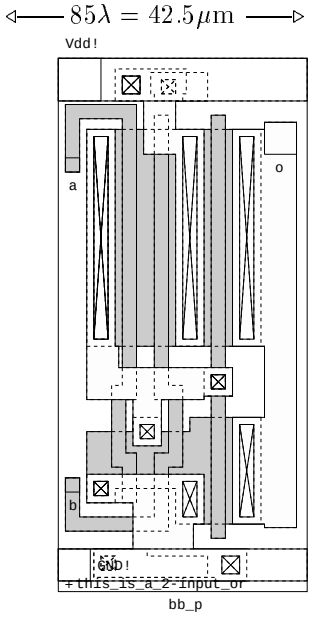
`MagicPart mag_or2 [a,b] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.31
Inputs	*Capacitance*	Δt ns
a	0.08 pF	1.37
b	0.08 pF	1.37

Description





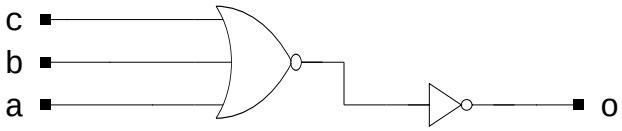
D.1.16 Cell Or3: a 3-input or

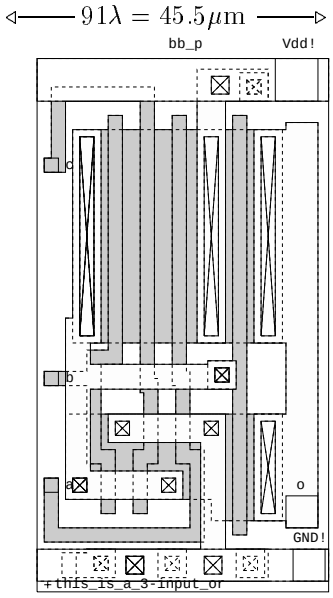
`MagicPart mag_or3 [a,b,c] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.35
Inputs	*Capacitance*	Δt ns
a	0.09 pF	2.52
b	0.09 pF	2.52
c	0.09 pF	2.52

Description





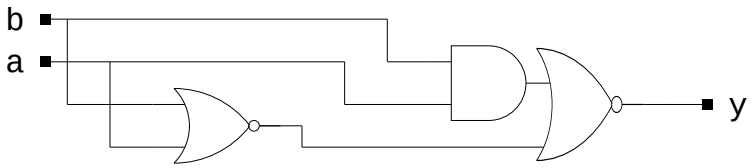
D.1.17 Cell Xor2: a 2-input xor

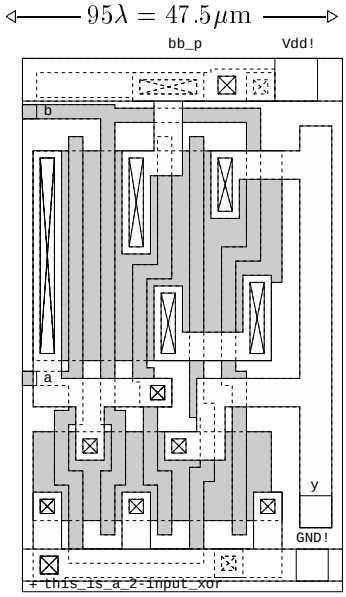
`MagicPart mag_xor2 [a,b] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	4.53
Inputs	*Capacitance*	Δt ns
a	0.18 pF	2.38
b	0.19 pF	2.38

Description





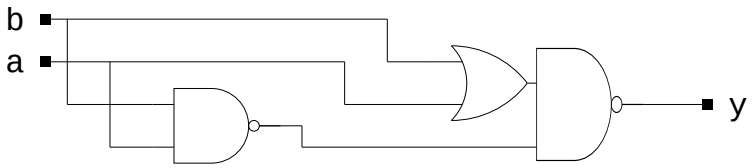
D.1.18 Cell Xnor2: a 2-input xnor

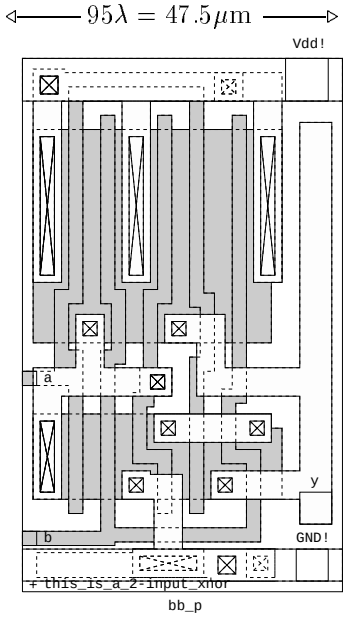
`MagicPart mag_xnor2 [a,b] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	4.59
Inputs	*Capacitance*	Δt ns
a	0.18 pF	2.26
b	0.19 pF	2.26

Description





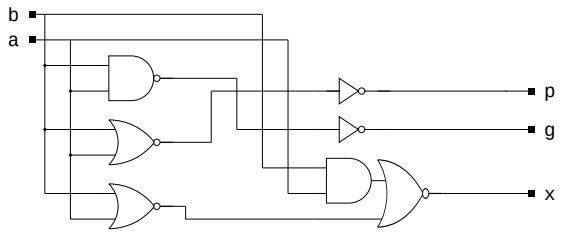
D.1.19 Cell Oax2: an or and xor 2 input gate

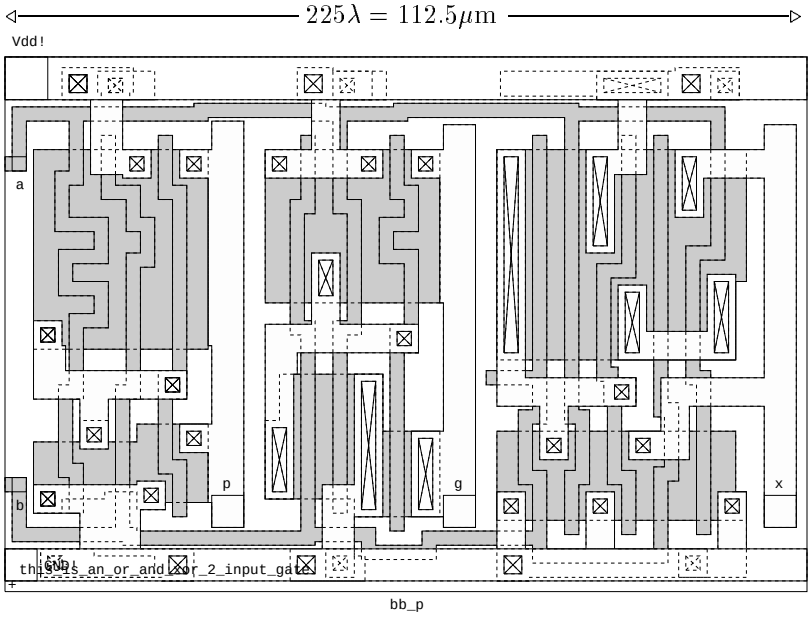
`MagicPart mag_oax2 [a,b] -> g,p,x`

Inputs and Outputs

	Outputs	g	p	x
	$\Delta t/\Delta C$ (ns/pF)	2.31	2.35	4.53
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
a	0.40 pF	0.99	1.32	2.38
b	0.37 pF	0.99	1.32	2.38

Description





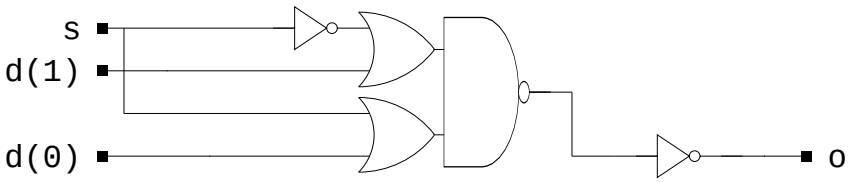
D.1.20 Cell Mux21: a 2-to-1 multiplexer

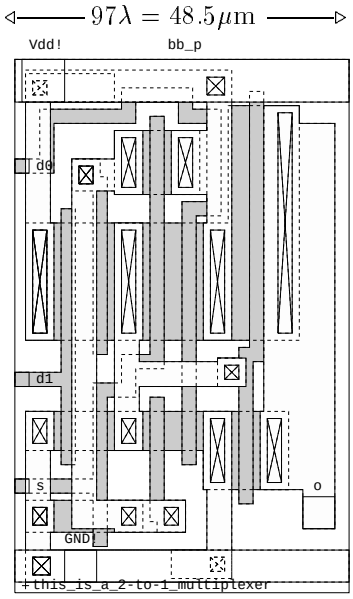
`MagicPart mag_mux21 [d(0:1),s] -> o`

Inputs and Outputs

	Outputs	o
	$\Delta t/\Delta C$ (ns/pF)	2.33
Inputs	*Capacitance*	Δt ns
d(0)	0.05 pF	2.45
d(1)	0.04 pF	2.45
s	0.07 pF	3.12

Description





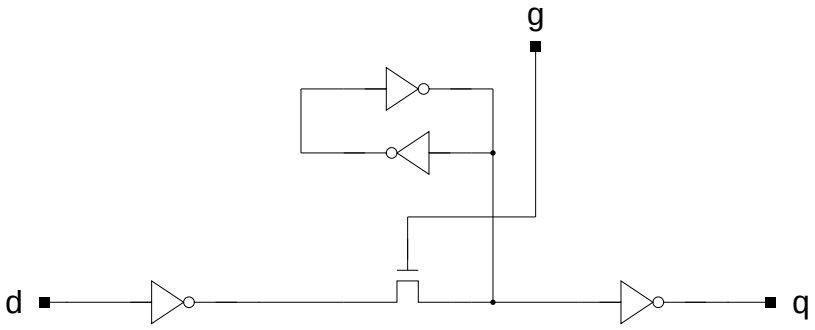
D.1.21 Cell Gff: a latch

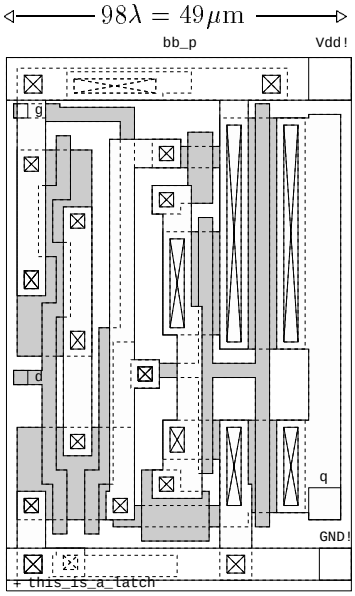
`MagicPart mag_gff [d,g] -> q`

Inputs and Outputs

	Outputs	q
	$\Delta t/\Delta C$ (ns/pF)	2.17
Inputs	*Capacitance*	Δt ns
d	0.10 pF	0.85
g	0.03 pF	0.57

Description





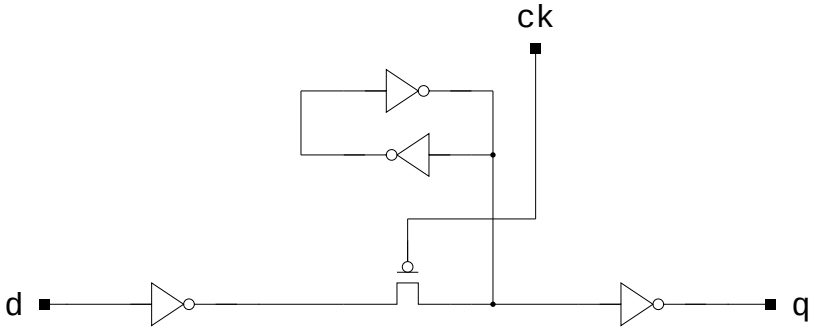
D.1.22 Cell Gffb: a latch

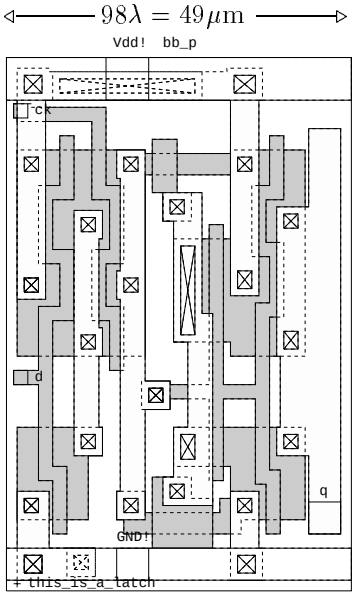
`MagicPart mag_gffb [ck,d] -> q`

Inputs and Outputs

	Outputs	q
	$\Delta t/\Delta C$ (ns/pF)	1.95
Inputs	*Capacitance*	Δt ns
ck	0.07 pF	0.75
d	0.11 pF	1.12

Description





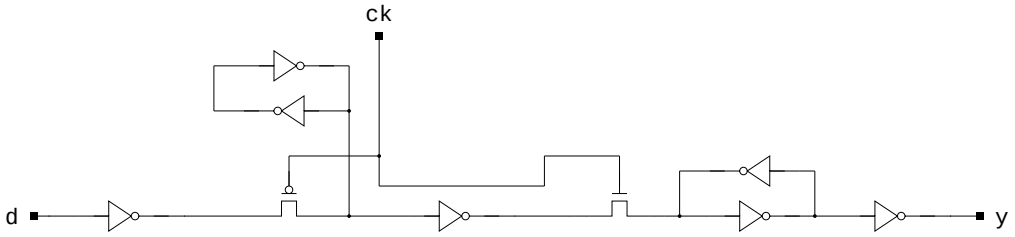
D.1.23 Cell Dff: a D-flipflop

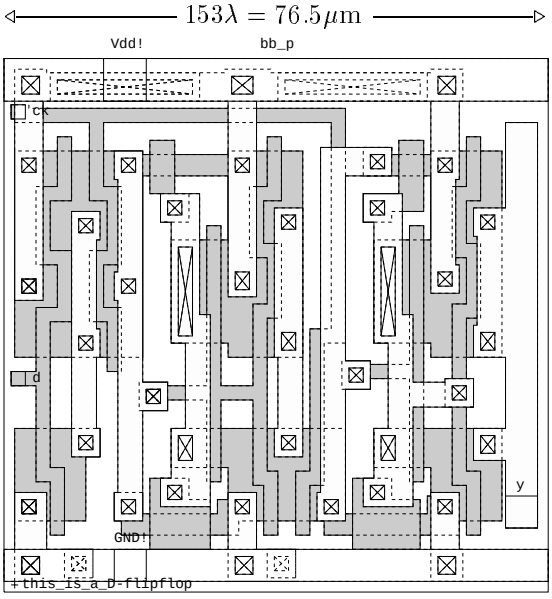
`MagicPart mag_dff [ck,d] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	2.01
Inputs	*Capacitance*	Δt ns
ck	0.11 pF	1.39
d	0.11 pF	—

Description





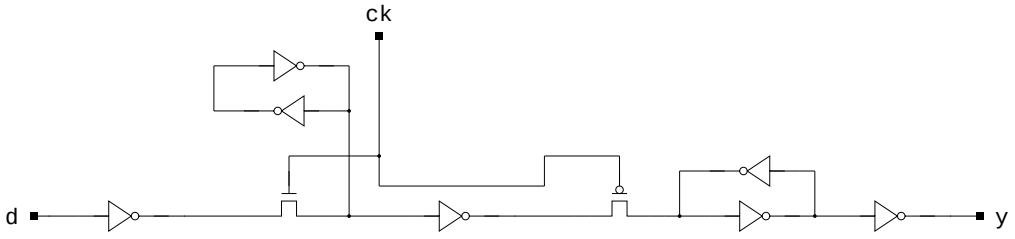
D.1.24 Cell Dffb: a D-flipflop

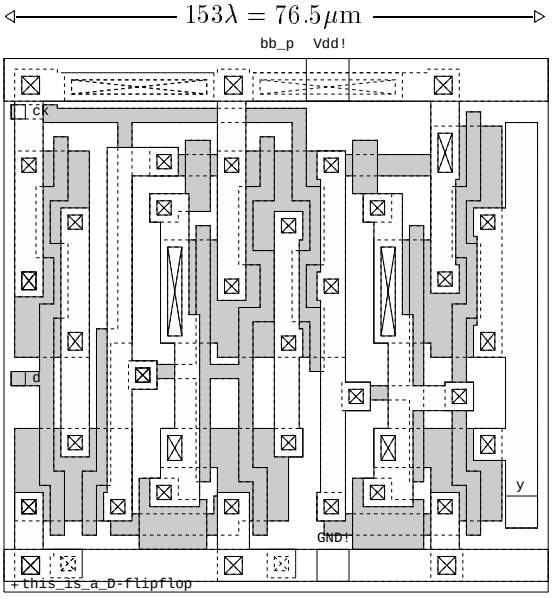
`MagicPart mag_dffb [ck,d] -> y`

Inputs and Outputs

	Outputs	y
	$\Delta t/\Delta C$ (ns/pF)	1.83
Inputs	*Capacitance*	Δt ns
ck	0.11 pF	1.57
d	0.10 pF	—

Description





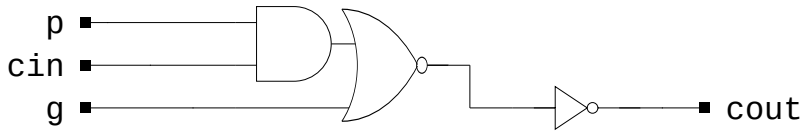
D.1.25 Cell Pg: a propagate/generate gate

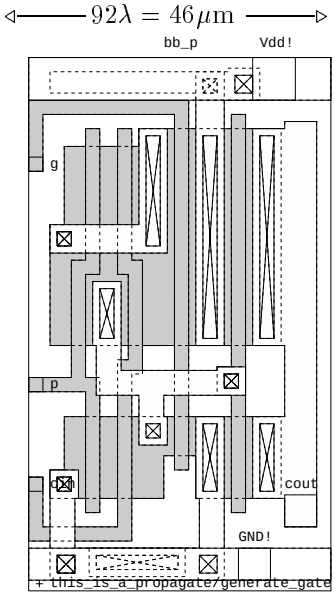
`MagicPart mag_pg [cin,g,p] -> cout`

Inputs and Outputs

	Outputs	cout
	$\Delta t/\Delta C$ (ns/pF)	2.31
Inputs	*Capacitance*	Δt ns
cin	0.08 pF	1.73
g	0.07 pF	1.73
p	0.08 pF	1.73

Description





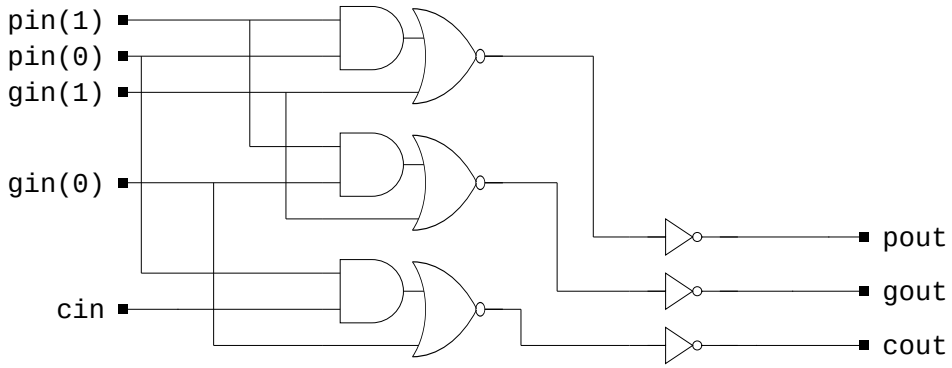
D.1.26 Cell Composepg: composition gate for look ahead

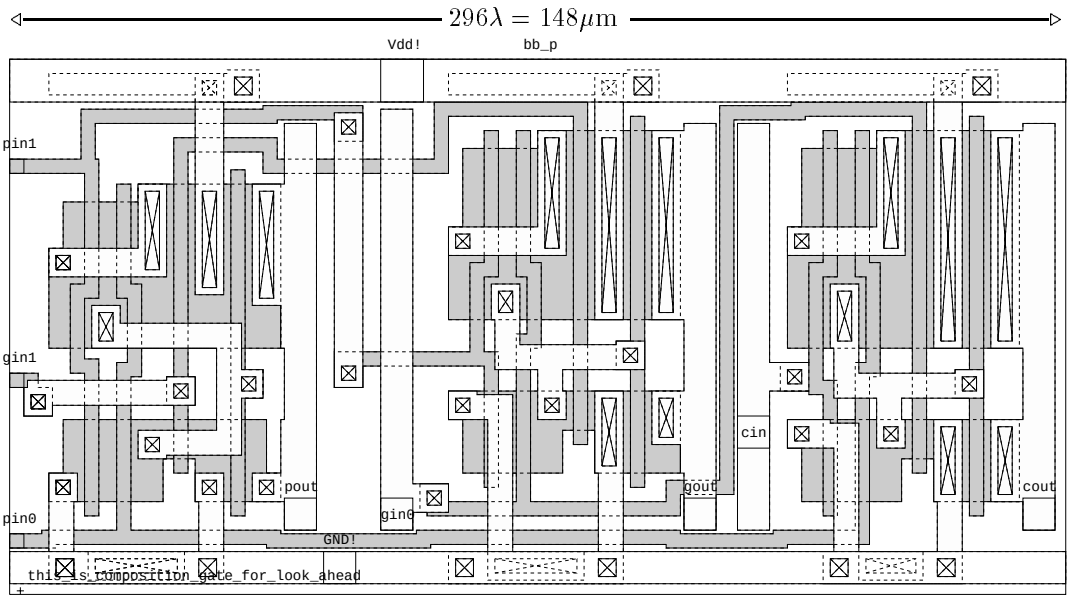
`MagicPart mag_composepg [cin,gin(0:1),pin(0:1)] -> cout,gout,pout`

Inputs and Outputs

	Outputs	cout	gout	pout
	$\Delta t/\Delta C$ (ns/pF)	2.31	2.66	3.06
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
cin	0.09 pF	1.73	—	—
gin(0)	0.17 pF	1.73	1.78	—
gin(1)	0.13 pF	—	1.78	1.90
pin(0)	0.16 pF	1.73	—	1.90
pin(1)	0.16 pF	—	1.78	1.90

Description




296λ = 148μm
Vdd!
bb_p
pin1
gin1
pin0
pout
gin0
GND!
cin
gout
cout
this_is_composition_gate_for_look_ahead

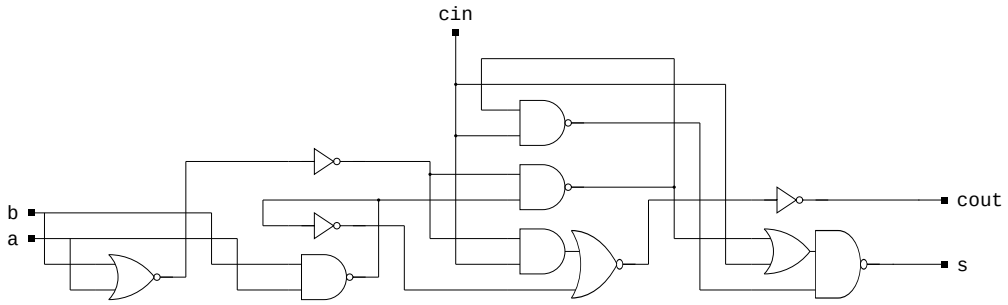
D.1.27 Cell Addcell: an adder cell

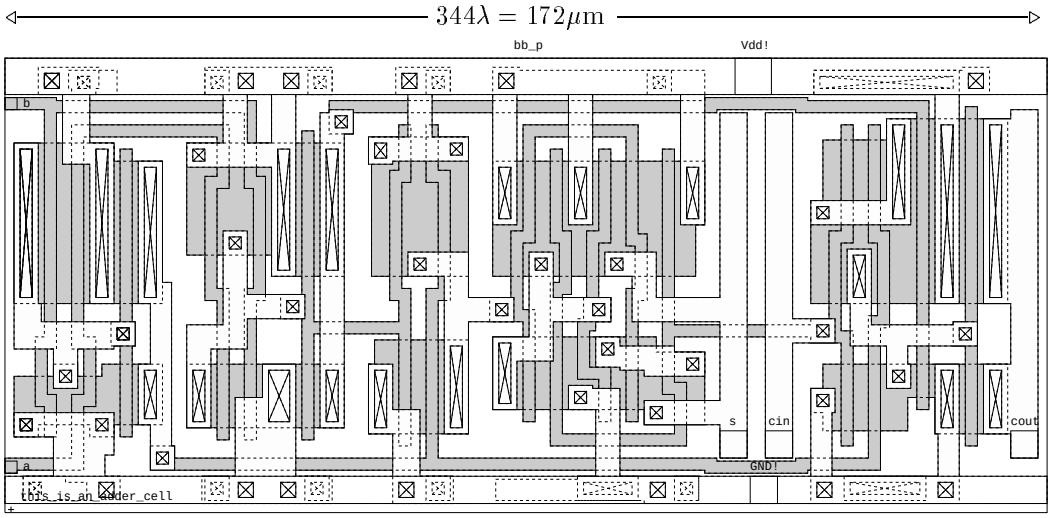
`MagicPart mag_addcell [a,b,cin] -> cout,s`

Inputs and Outputs

	Outputs	cout	s
	$\Delta t/\Delta C$ (ns/pF)	2.31	7.15
Inputs	*Capacitance*	Δt ns	Δt ns
a	0.14 pF	3.61	5.01
b	0.14 pF	3.61	5.01
cin	0.23 pF	1.72	2.16

Description





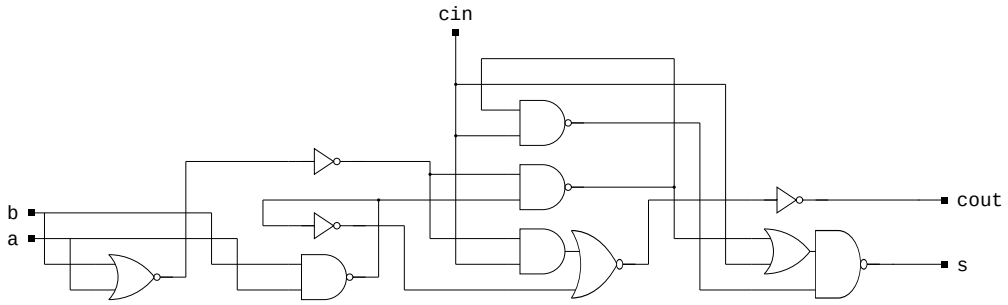
D.1.28 Cell Addcell1: an adder cell

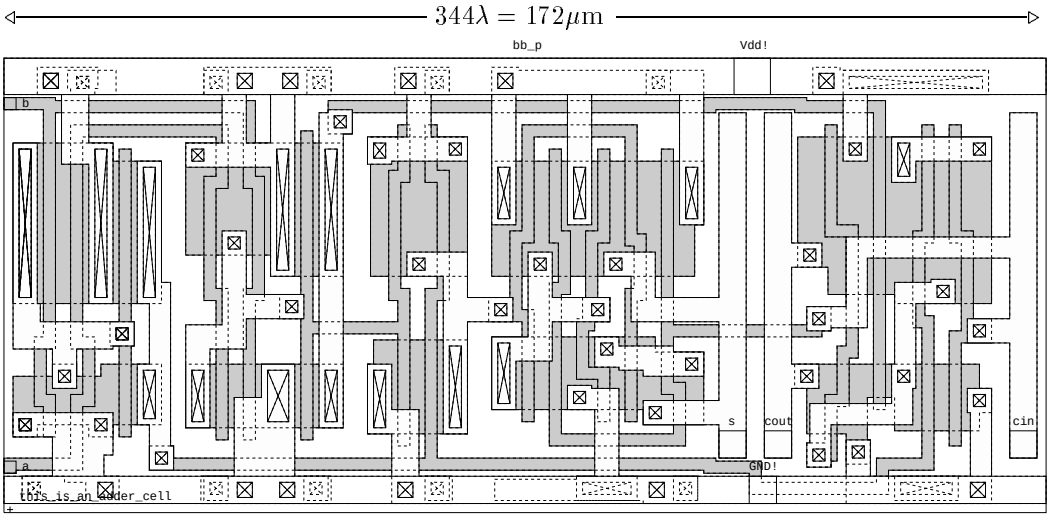
`MagicPart mag_addcell1 [a,b,cin] -> cout,s`

Inputs and Outputs

	Outputs	cout	s
	$\Delta t/\Delta C$ (ns/pF)	2.39	7.15
Inputs	*Capacitance*	Δt ns	Δt ns
a	0.14 pF	3.78	5.02
b	0.14 pF	3.78	5.02
cin	0.24 pF	1.91	2.16

Description





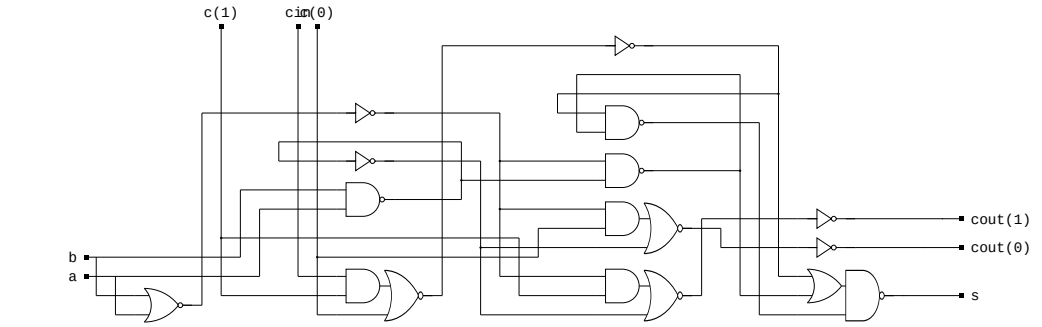
D.1.29 Cell Addcarry: an adder cell

`MagicPart mag_addcarry [a,b,c(0:1),cin] -> cout(0:1),s`

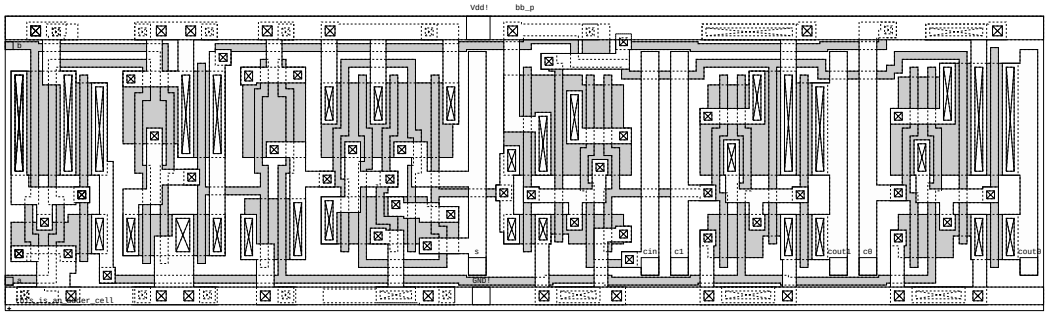
Inputs and Outputs

	Outputs	`cout(0)`	`cout(1)`	`s`
	$\Delta t/\Delta C$ (ns/pF)	2.47	3.00	7.15
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
`a`	0.14 pF	3.88	3.89	5.25
`b`	0.14 pF	3.88	3.89	5.25
`c(0)`	0.16 pF	1.75	—	4.71
`c(1)`	0.15 pF	—	1.77	4.71
`cin`	0.07 pF	—	—	4.71

Description



$529\lambda = 264.5\mu\text{m}$



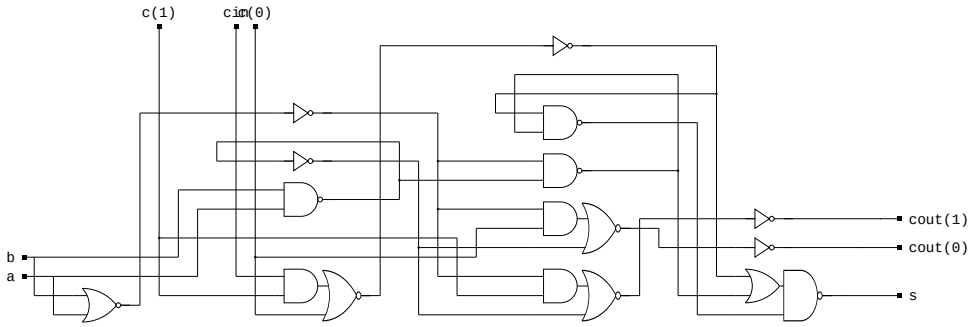
D.1.30 Cell Addcarry1: an adder cell

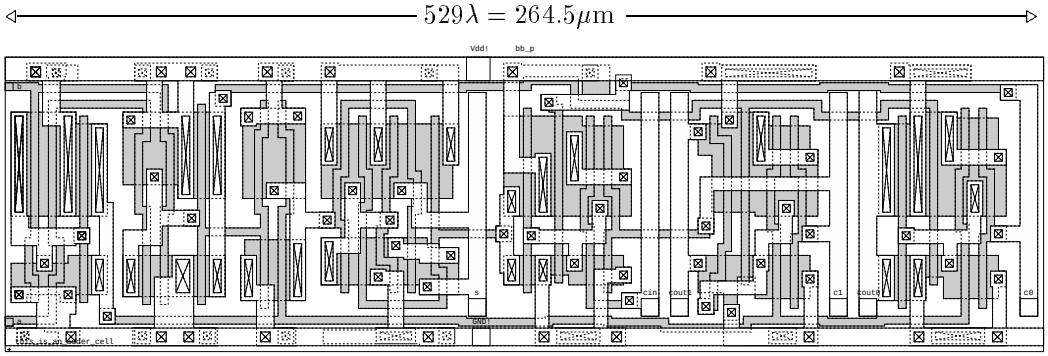
`MagicPart mag_addcarry1 [a,b,c(0:1),cin] -> cout(0:1),s`

Inputs and Outputs

	Outputs	`cout(0)`	`cout(1)`	`s`
	$\Delta t/\Delta C$ (ns/pF)	3.00	2.71	7.15
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
`a`	0.14 pF	3.87	3.95	5.25
`b`	0.14 pF	3.87	3.95	5.25
`c(0)`	0.16 pF	1.76	—	4.71
`c(1)`	0.15 pF	—	1.85	4.71
`cin`	0.07 pF	—	—	4.71

Description





D.2 Mcmos pad library

The `pad.mod` library (`~cad/ulm/mcmos/pad`) contains generic Parts which call layouts (table D.2):

```
Part inpad(type)[pad] -> out
Part outpad[out] -> pad
Part tripad(type)[in,oe] -> out,pad
Part dirpad[] -> pad
Part gndpad[]
Part vddpad []
```

The layouts exists in slim and flat style.

generic cell	*cell*	*function*
`inpad(cmos)`	`padic`	a cmos input pad
`inpad(ttl)`	`padit`	a ttl input pad
`outpad`	`pado`	an output pad
`tripad(cmos)`	`pad3c`	a cmos tristate pad
`tripad(ttl)`	`pad3t`	a ttl tristate pad
`dirpad`	`paddirect`	a direct pad
`gndpad`	`padgnd`	a ground power-supply pad
`vddpad`	`padvdd`	a vdd power-supply pad

Table D.2: the cells of the `pad.mod` library

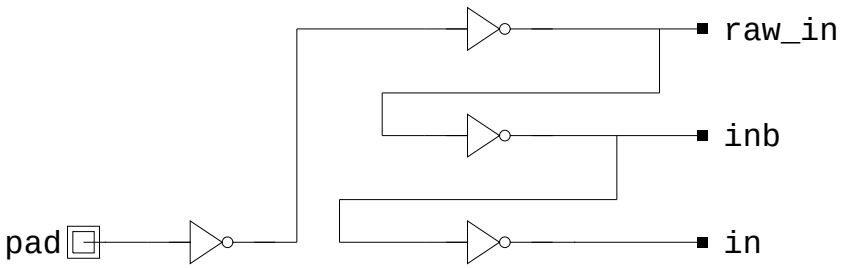
D.2.1 Cell Padic: a cmos input pad

`PadPart mag_padic [pad] -> in,inb,raw_in`

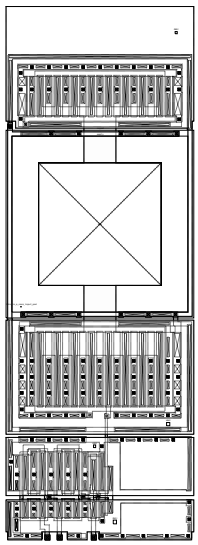
Inputs and Outputs

	Outputs	`in`	`inb`	`raw_in`
	$\Delta t/\Delta C$ (ns/pF)	0.50	0.48	1.05
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
`pad`	4.58 pF	2.33	2.09	1.65

Description



$\triangleleft\!\!-\ 509\lambda = 254.5\mu\text{m}\ -\!\!\triangleright$



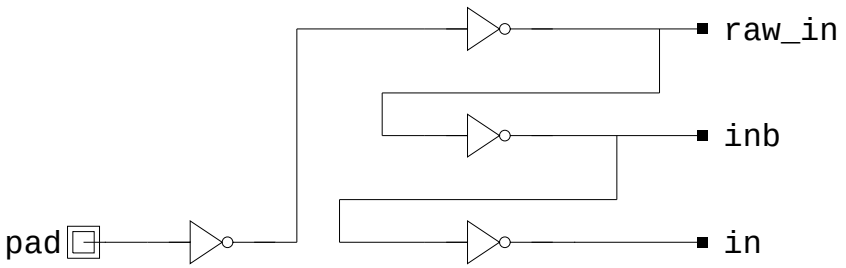
D.2.2 Cell Padit: a ttl input pad

`PadPart mag_padit [pad] -> in,inb,raw_in`

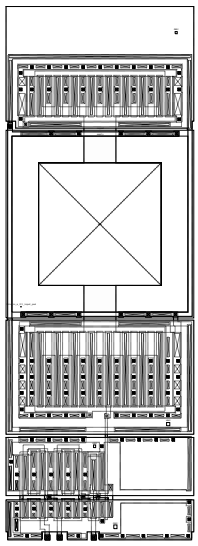
Inputs and Outputs

	Outputs	`in`	`inb`	`raw_in`
	$\Delta t/\Delta C$ (ns/pF)	0.50	0.48	1.05
Inputs	*Capacitance*	Δt ns	Δt ns	Δt ns
`pad`	4.56 pF	4.74	4.50	4.05

Description



$\triangleleft\!\!-\, 509\lambda = 254.5\mu\text{m} \,-\!\!\triangleright$



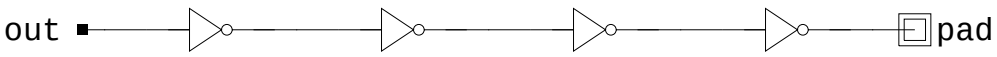
D.2.3 Cell Pado: an output pad

`PadPart mag_pado [out] -> pad`

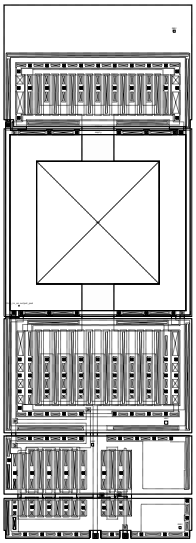
Inputs and Outputs

	Outputs	`pad`
	$\Delta t/\Delta C$ (ns/pF)	0.06
Inputs	*Capacitance*	Δt ns
`out`	0.11 pF	3.23

Description



$\leftarrow 509\lambda = 254.5\mu\text{m} \rightarrow$



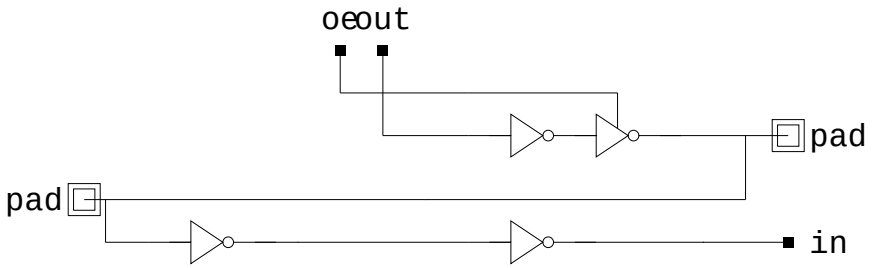
D.2.4 Cell Pad3c: a cmos tristate pad

`PadPart mag_pad3c [oe,out] -> in,pad`

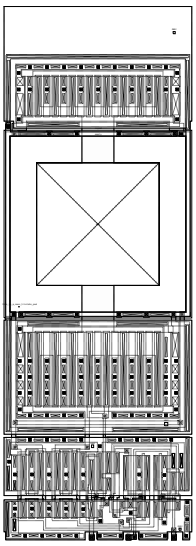
Inputs and Outputs

	Outputs	in	pad
	$\Delta t/\Delta C$ (ns/pF)	1.05	0.06
Inputs	*Capacitance*	Δt ns	Δt ns
oe	0.13 pF	2.15	0.27
out	0.14 pF	3.53	1.66
pad	4.56 pF	1.87	0.00

Description



$\triangleleft\!\!-\ 509\lambda = 254.5\mu\text{m}\ -\!\!\triangleright$



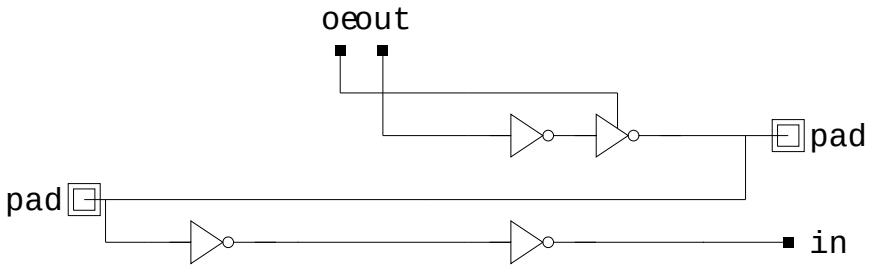
D.2.5 Cell Pad3t: a ttl tristate pad

`PadPart mag_pad3t [oe,out] -> in,pad`

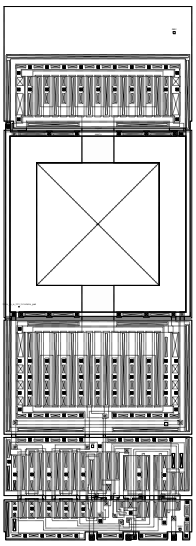
Inputs and Outputs

	Outputs	in	pad
	$\Delta t/\Delta C$ (ns/pF)	1.05	0.06
Inputs	*Capacitance*	Δt ns	Δt ns
oe	0.16 pF	3.90	0.28
out	0.14 pF	5.28	1.66
pad	4.57 pF	3.62	0.00

Description



$\triangleleft\!\!-\ 509\lambda = 254.5\mu\text{m}\ -\!\!\triangleright$

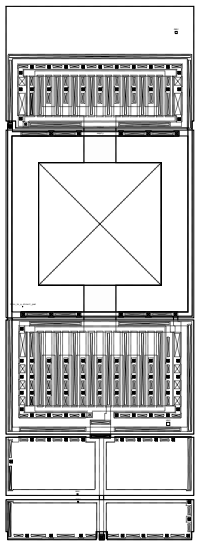


D.2.6 Cell Paddirect: a direct pad

`PadPart mag_paddirect [pad]`

Description

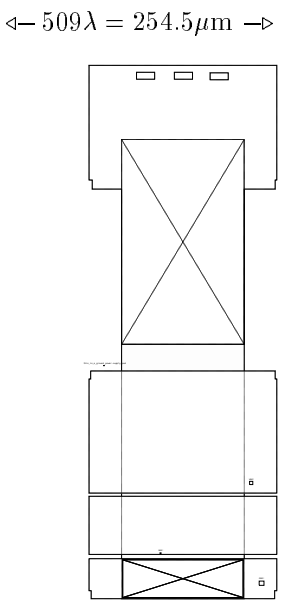
$\triangleleft\!\!-\ 506\lambda = 253\mu\text{m}\ -\!\!\triangleright$



D.2.7 Cell Padgnd: a ground power-supply pad

`PadPart mag_padgnd []`

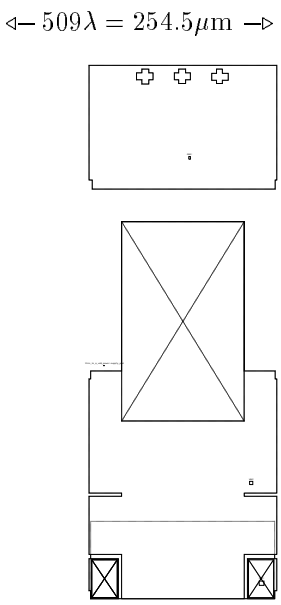
Description



D.2.8 Cell Padvdd: a vdd power-supply pad

`PadPart mag_padvdd []`

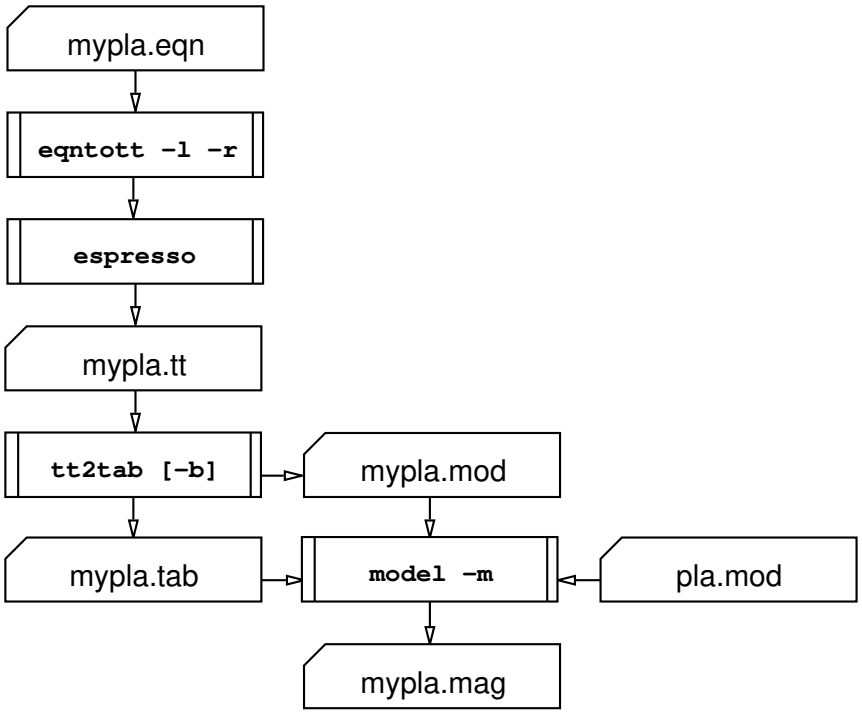
Description



D.3 Mcmos PLA generator

PLA Generator

```
Include "pla.mod"
PavePart mypla[i1,i2,...,in] -> o1,o2,...,op
    Integer fd,t
    fd = Open ("mypla.tab")
    t = Read(fd)
    pla(n,p,fd,t,0)[i1,i2,...,in] -> o1,o2,...,op
{   pla(n,p,fd,t,1)[i1,i2,...,in] -> o1,o2,...,op
End
```



Usage

The PLA generator (see diagram) consists of:

- a model file (`~cad/ulm/mcmos/pla/pla.mod`)
- the classical programs of the Berkeley distribution (**eqntott** and **espresso**) for the generation of the optimized `.tt` format.
- the **tt2tab** program which converts the `.tt` format to the `.tab` format.

The designer provides the *mypla*`.eqn` file (see eqntott(1)). The unix command

`eqntott -l -r` *mypla*`.eqn | espresso >` *mypla*`.tt`

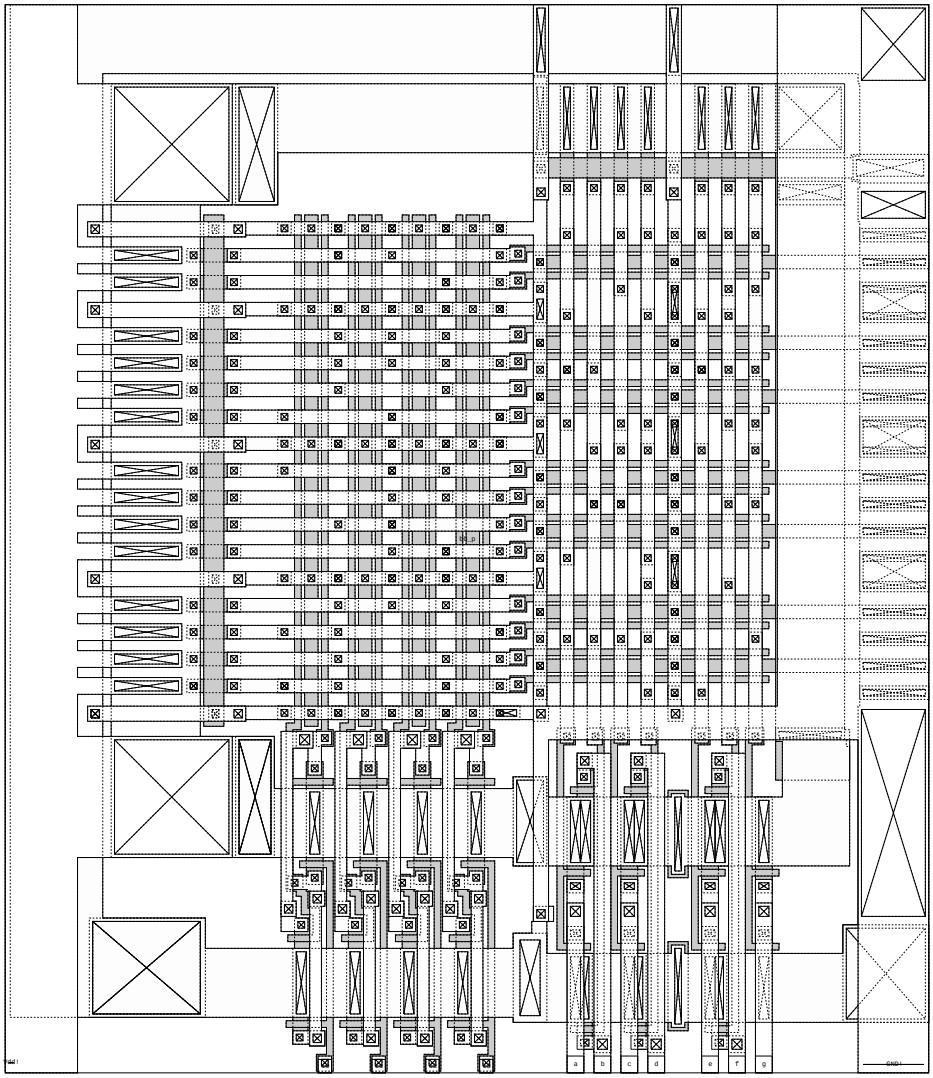
generates the *mypla*`.tt` file which contains the optimized description of the PLA in the table format. The unix command

`tt2tab [-b]` *mypla* `>` *mypla*`.mod`

generates the two files:

- *mypla*`.tab` the description of the pla in a format readable by the `pla.mod` model program.
- *mypla*`.mod` the model interface of the pla.

When the `-b` flag is set in the `tt2tab` command, the 5^{th} argument of the call to the `pla` part is set to 1 and **mod2mag** generates internal buffers for the intermediate terms of the PLA.



Appendix E

Examples

In this appendix, we presents different examples of circuits described in the **model** language. These examples are teaching projects given at the Ecole Nationale Supérieure des Techniques Avancées.

E.1 Temperature and voltage measurement

This example is located in `~cad/ulm/mcmos/more_examples/tvm`.

This circuit has been developed in collaboration with Georges Quenot. Thermal dissipation and power distribution may be critical in large systems. The goal of this circuit is to measure the voltage between the power-supply and the temperature of the circuit. This circuit is a only a test circuit. In a real application this circuit must be a part of a larger circuit. The size of the circuit ($< 2\mu m^2$) is small enough to use it as a part of a large VLSI.

The circuit contains two ring-oscillators:

- an n-transistor delay oscillator,
- an n-diffusion delay oscillator.

The frequency of these oscillators are around 10 MHz and depends of both the temperature and the voltage. Each oscillator drives a counter which can be loaded and downloaded via a common scan-path. Used in another circuit, the scan-path may take place in the scan-path of the whole circuit in order to save pads.

The inputs/outputs of the circuit are:

- the input of the scan-path,
- the output of the scan-path,
- the clock of the scan-path,
- the *enable* of the ring-oscillators.

The directory contains the following files:

- `delayinv.mag` and `delayinv.mod`: the n-transistor delay cell,

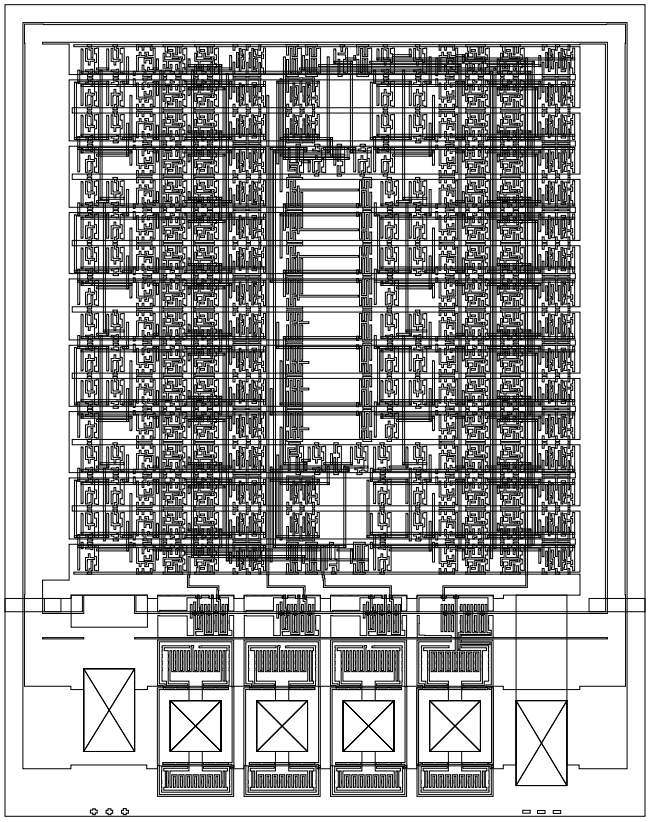


Figure E.1: layout of the TVM chip

- `delayres.mag` and `delayres.mod`: the n-diffusion delay cell,
- `chip.mod`: the model source file of the circuit.
- `chip.stim`: simulation stimuli for msim.

E.1.1 Generation of the layout

The command:

```
model chip -m
```

generates the layout (`chip.mag`, `tvm_2.mag` and `core.mag`), and the netlist file `tvm_2.net`. The circuit `tvm_2` can be finished under **magic** by the command:

```
:route
```

The help of the user is required for the connection of the power-supplies to the core of the circuit. It is simple for this chip. The figure E.1 shows the metal layers of the layout of the tvm chip.

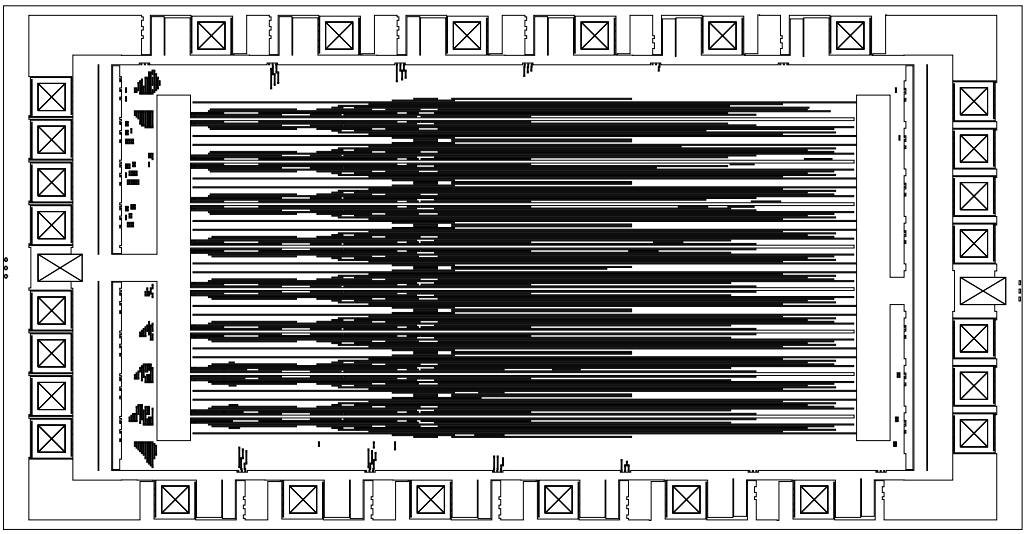


Figure E.2: layout of the cross–bar

E.1.2 Simulation

the command:

```
model chip -l
```

generates the logical netlist (`chip.log`,`chip.beh` and `chip.al`). the command:

```
ana chip -f chip.stim
```

starts the `msim` simulator in its graphical framework.

E.2 A 16 by 16 Cross-bar

This example is located in `~cad/ulm/mcmos/more_examples/cross`.

This circuit allows to interconnect 16 serial links. The circuit is configured through a command bus (`com(0:5)`), an address bus (`a(0:3)`) and a enable (`v`). Each link is successively configured. The address bus selects the current link to configure. A positive pulse on the enable saves the states of the command bus into the internal table of the chip.

- `com(4)` defines whether the current link is an input (1) or an output (0),
- `com(5)` defines if the output is inverted,
- `com(0:3)` defines the number of the selected link if the current link is an output.

This circuit has been developed in collaboration with Thierry Bernard. It contains:

- the control of each link
- a 16×16 crossbar

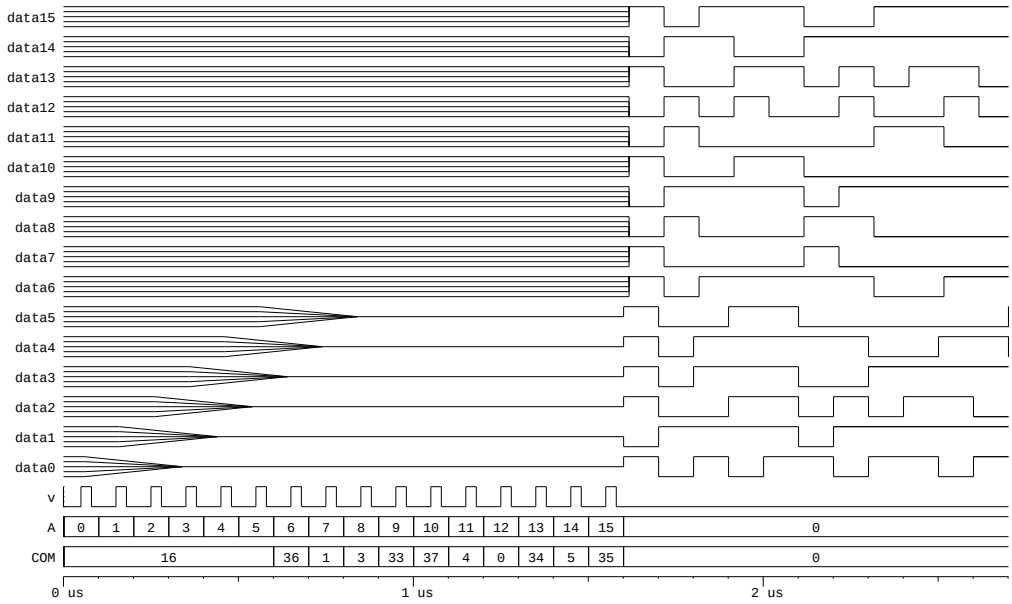


Figure E.3: simulation of the cross–bar

E.2.1 Generation of the layout

The figure E.2 shows the layout of the cross-bar. The command:

```
model chip -m
```

generates the layout (`chip.mag`, `tvm_2.mag` and `core.mag`), and the netlist file `tvm_2.net`. The circuit `tvm_2` can be finished under **magic** by the command:

```
:route
```

The help of the user is required for the connection of the power-supplies to the core of the circuit. It is simple for this chip. The figure E.1 shows the metal layers of the layout of the tvm chip.

E.2.2 Simulation

the command:

```
model chip -l
```

generates the logical netlist (`chip.log`,`chip.beh` and `chip.al`). the command:

```
ana chip -f chip.stim
```

starts the `msim` simulator in its graphical framework. The figure E.2 shows the wave-form of the cross-bar during the initialization and during the use of the links.

E.3 A microcoded divider

This example is located in `~cad/ulm/mcmos/more_examples/div`. The generated circuit is a microcoded divider. The method used is this circuit is not the classical one but the method

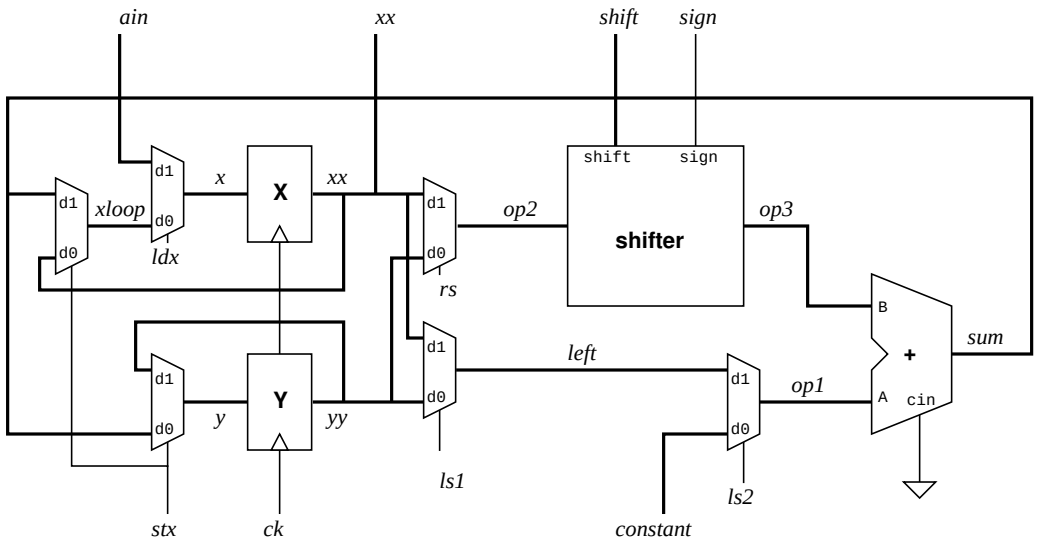


Figure E.4: schematic of the ALU

proposed by [Li85]. Every denominator n can be written:

$$n = 2^k \times p$$

with an odd p. The division by a power of 2 is done by a right shift operation. It remains the problem of dividing by an odd number. As

$$\forall p \text{odd}, \exists n, k, 2^n - 1 = p \times k,$$

the division by p is equivalent to a multiplication by k and a division by $2^n - 1$. But $1/(2^n - 1)$ may be written:

$$0.\overbrace{0\cdots0}^{n-1}1\overbrace{0\cdots0}^{n-1}1\overbrace{0\cdots0}^{n-1}1\cdots$$

If n is big enough, this number contains a lot of 0: it limits the problem of the carry propagation for the multiplication. The division will be performed by addition and shift. The precision problem can be avoided by adding sometimes a constant.

The table E.1 shows the sequence of operations for each odd denominator less than 16.

E.3.1 Organization of the divisor

The ALU of the divisor

The systematic structure of the instructions of the table E.1 allows us to define the basic organization of the ALU E.4. Two variables x and y are used. At each step, one of the variable is modified as the result of an addition. It left operand is either a variable or a constant between 0 and 5. The right operand is the result of a shift (left, right or no shift) of a variable x or y by a constant.

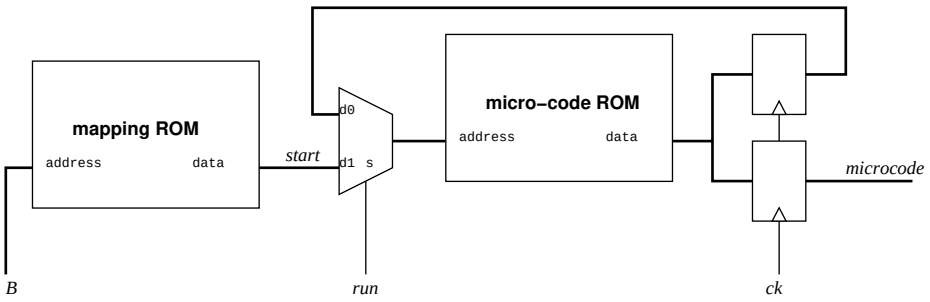


Figure E.5: schematic of the sequencer

The sequencer

The sequencer (fig. E.5) contains 4 parts:

- a microcode ROM with the micronstruction for the control of the ALU.
- a sequencing register which clocks the current state and the microinstruction,
- a multiplexer to force the start of the microinstruction sequence,
- a mapping rom, which converts the B operand into the starting address of the microinstruction sequence.

The sequencer is a finite state automaton (fig. E.6). The value of B (the squares on the figure) are converted by the mapping ROM into starting states. From these states, the microcode is executed until the finite state automaton reaches the idle state 0. Even denominators first execute a shift instruction before jumping to the sequence of an odd denominator.

E.3.2 The layout

The program "micro" reads the file div.c (fig. E.1) and generates the informations for the 2 PLA :

- `map.tn` for the mapping ROM,
- `micro.tn` for the microcode ROM.

These tables are optimized by **espresso** and then converted to the `.tab` format by **tt2tab**.

The **model** source is split in 6 files:

- `chip.mod` the whole circuit,
- `div.mod` the core of the circuit,
- `alu.mod` the operators of the ALU,
- `map.mod` the map ROM,
- `micro.mod` the microcode ROM.

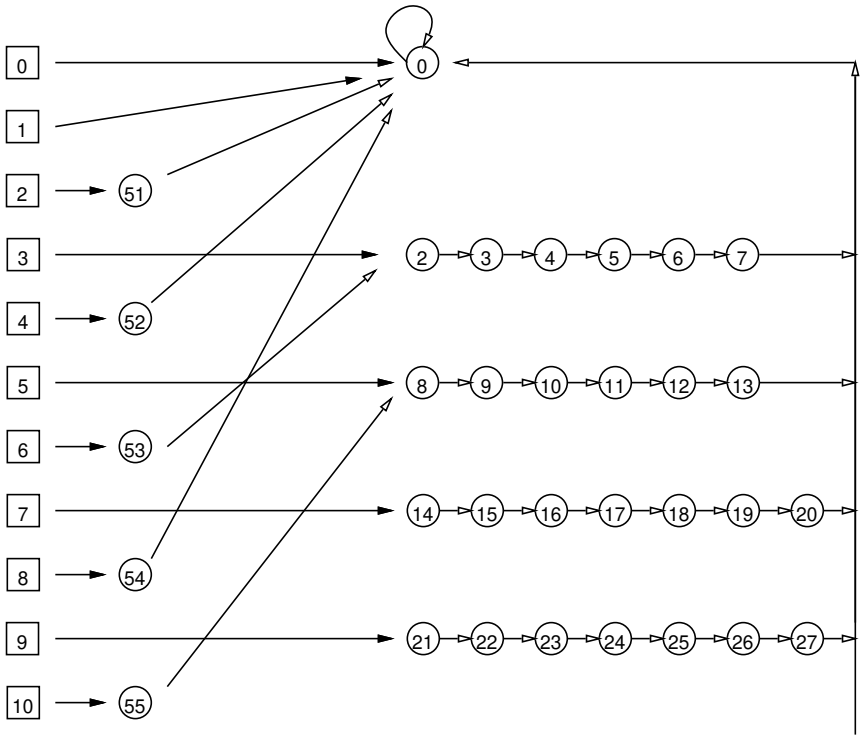


Figure E.6: part of the finite state automaton

The `test.mod` contains an instance of the core of the chip in order to facilitate the simulation of the circuit. At last the file `beh.mod` which contains the behavioral descriptions of the operators. The command

```
make chip.mag
```

generates the layout (fig. E.7).

The core of the circuit contains 4 parts:

- the data-path of the ALU,
- the microcode ROM,
- the mapping ROM,
- the data-path of the sequencer

The circuit is then finished under magic:

- the connections of the power–supplies rails,
- the global routing.

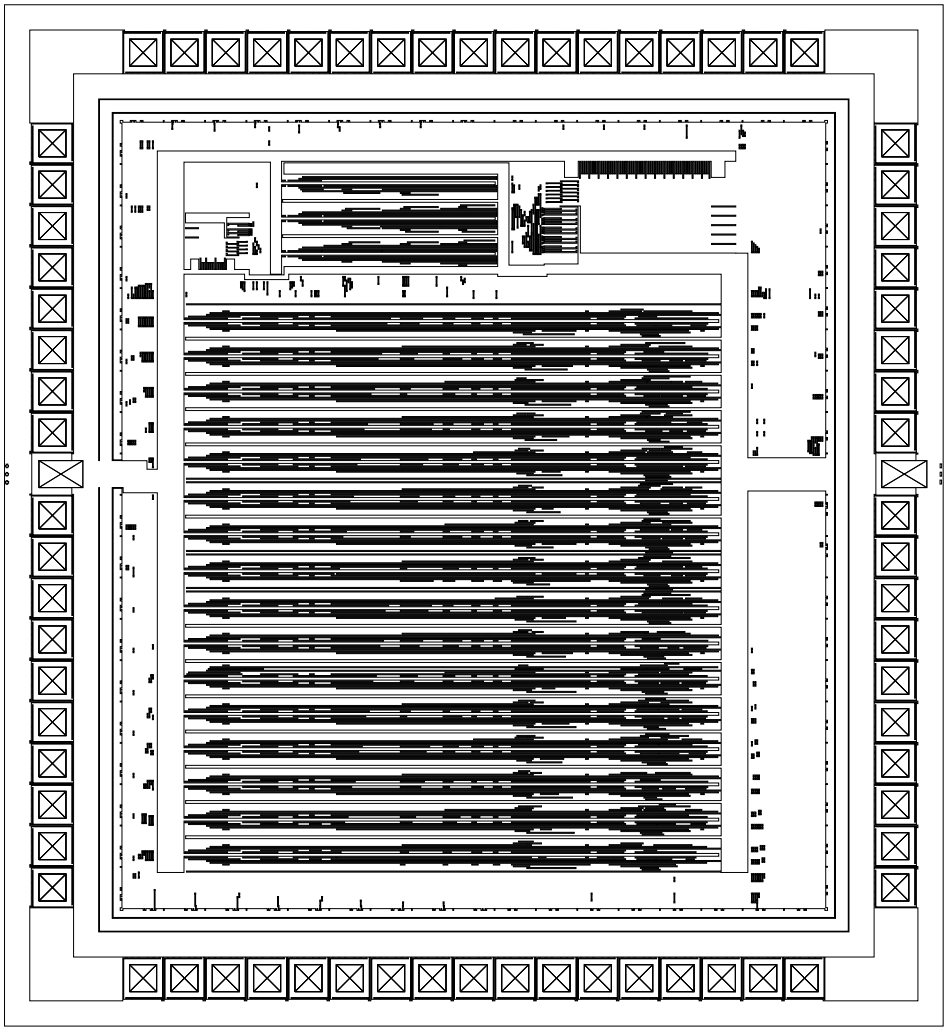


Figure E.7: layout of the divisor chip (*metal2* layer)

E.3.3 Simulation

The command:

```
make test.slo
```

generates the simulation. The figure E.8 shows the 9 step division of 95182059 by 14.

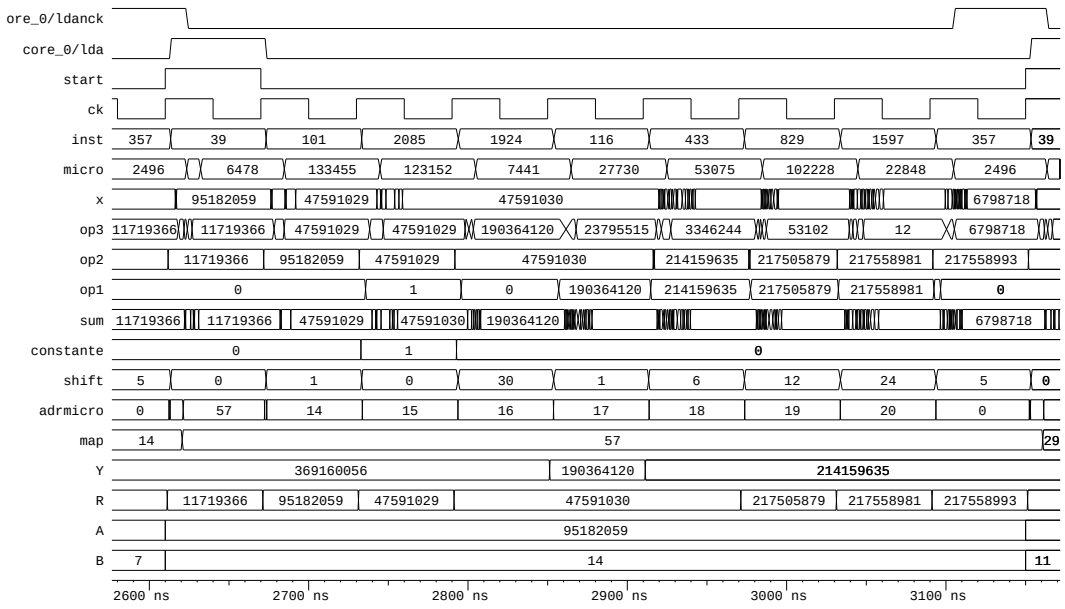


Figure E.8: wave-forms of the divisor

```
case 3 :
        x = x + (x<<2);
        x = 5 + x;
        x = x + (x>>4);
        x = x + (x>>8);
        x = x + (x>>16);
        x = 0 + (x>>4);
        break;
case 5 :
        x = x + (x<<1);
        x = 3 + x;
        x = x + (x>>4);
        x = x + (x>>8);
        x = x + (x>>16);
        x = 0 + (x>>4);
        break;
case 7 :
        x = 1 + x;
        y = 0 + (x<<2);
        y = y + (x>>1);
        x = y + (y>>6);
        x = x + (x>>12);
        x = x + (x>>24);
        x = 0 + (x>>5);
        break;
case 9 :
        x = 1 + x;
        y = 0 + (x<<1);
        y = y + (x>>1);
        x = x + y;
        x = x + (x>>6);
        x = x + (x>>12);
        x = x + (x>>24);
        x = 0 + (x>>5);
        break;
case 11 :
        x = 1 + x;
        y = x + (x<<2);
        y = y + (y>>4);
        x = y + (x>>1);
        x = x + (x>>10);
        x = x + (x>>20);
        x = 0 + (x>>6);
        break;
case 13 :
        x = 1 + x;
        y = 0 + (x<<2);
        y = y + (x>>1);
        x = 0 + (y<<2);
        x = y + (y>>5);
        x = x + (y>>4);
        x = x + (x>>12);
        x = x + (x>>24);
        x = 0 + (x>>6);
        break;
case 15 :
        x = 1 + x;
        y = 0 + (x << 2);
        x = y + (x >> 2);
        x = x + (x >> 8);
        x = x + (x >> 16);
        x = 0 + (x >> 6);
        break;
```

Table E.1: basic operations for odd numbers less than 16

Appendix F

Manuel pages

This appendix contains the manual pages of file formats and programs. In the section 1:

- model,
- msim.
- ana,
- mag2ps,
- ext2mod.

In the section 5:

- log,
- slo.

Bibliography

[Li85] Shuo-Yen Robert Li. Fast Constant Division Routines. *IEEE transactions on computers*, Vol. C–34 No 9, Septembre 1985.

[Mod86] *The MODEL Language Reference Manual.* Wemyss Place Edinburgh EH3 6DH UK, 1986.

[SMHO86] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout. *1986 VLSI Tools: Still More Works by the Original Artists.* Technical Report UCB/CSD 86/272, Computer Science Division (EECS), University of California Berkeley, California 94720, 1986.

Contents

List of Tables

List of Figures

Index