

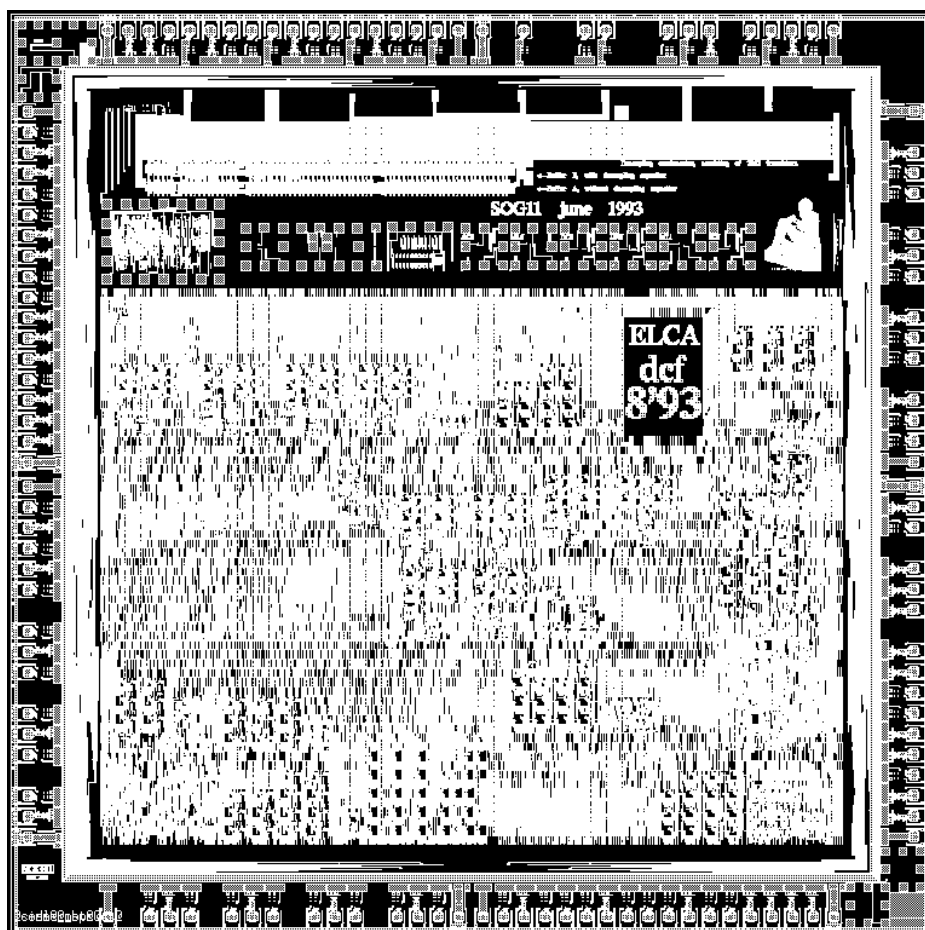
Ocean: the Sea-of-Gates Design System

Patrick Groeneveld and Paul Stravers

Delft University of Technology, faculty of Electrical Engineering
Delft, the Netherlands

e-mail: space-support-ewi@tudelft.nl

September 30, 2019



Picture on cover page:

The 11th chip which was made with the OCEAN sea-of-gates design system. The 200,000 transistor chip contains is configured as a 'multi-project chip', containing a number of independent designs. The top quarter contains a few smaller analog circuits and large on-chip capacitor. The rest of the chip contains a DCF-77 radio clock receiver circuit.



Copyright ©1994 by P. Groeneveld and P. Stravers.

This document was processed with the use of latex.

Contents

1	Introduction	1
1.1	About this manual	1
1.2	Overview of the system	2
1.3	How to retrieve the OCEAN system?	5
1.4	How to run the OCEAN tools?	5
1.5	Additional information and related documents	6
2	The Sea-of-Gates image	7
2.1	Why Sea-of-Gates?	7
2.2	The TU-Delft fishbone chip	9
2.3	The general structure of the fishbone image	11
2.4	Restrictions on placing contacts	13
2.5	Transistor layout on fishbone: gate isolation technique	13
2.6	Power net strategy	14
2.7	Some practical hints	14
3	A quick tutorial	17
3.1	Let's get behind the terminal	17
3.2	Designing a simple library cell	18
3.2.1	Before we start: making a project	18
3.2.2	Creating a circuit description	18
3.2.3	Simulating your circuit	20
3.2.4	Creating a layout	21
3.2.5	Extracting the layout and simulating it	22
3.3	The second example: a hotel switch	23
3.3.1	Functionality	23
3.3.2	A Moore machine	23

3.3.3	A three level hierarchy	23
3.3.4	Layout implementation with the OCEAN tools	25
3.3.5	Verification	26
3.3.6	Extracting the circuit	27
3.3.7	Simulating at various levels of abstraction	28
3.3.8	Graphically editing the stimuli file	28
3.4	The third example: using logic synthesis to make 'hotel'	28
3.5	Some advanced features	30
3.5.1	Playing around with <i>Madonna</i>	30
3.5.2	Using <i>trout</i> in various ways	31
3.5.3	Inserting your own nor in the circuit	32
3.5.4	Making the hotel switch on a different image	32
3.5.5	Plotting your layout	33
3.5.6	Removing cells and projects	33
4	The programs in the OCEAN system: a glossary	35
4.1	Programs to generate netlists	35
4.2	Programs to generate layout	35
4.3	Programs for simulation	37
4.4	Programs to list and manipulate design projects	37
4.5	Programs to list and manipulate cells	37
4.6	Programs to convert to and from the database	38
4.7	Miscellaneous programs	38
5	'Seadali': your gateway to sea-of-gates layout	39
5.1	Setting up the environment: creating a project using 'mkopr', 'mkepr' or 'mkpr'	39
5.2	Starting 'seadali'	40
5.3	Getting on-line help in 'seadali'	41
5.4	Reading and writing layout: database	41
5.5	Running other programs: the automatic tools menu	42
5.6	Importing cells from a library: the instances menu	42
5.7	Editing mask layout: the boxes menu	43
5.7.1	Selecting active masks	43
5.7.2	Making wires using APPEND	43

5.7.3	Making wires with <code>wire</code>	43
5.7.4	Ensuring design-rule correctness	43
5.7.5	How to make contacts between layers?	44
5.7.6	Deleting layout: <code>DELETE</code>	44
5.7.7	Copying layout: <code>yank</code> and <code>put</code>	44
5.7.8	Purifying and checking your layout: <code>fish</code>	45
5.7.9	Automatically routing your design: <code>trout</code>	45
5.7.10	Verifying layout connectivity: <code>Check nets</code>	45
5.8	Adding terminals: the <code>terminals</code> menu	46
5.8.1	How to add a terminal?	46
5.8.2	How can I get rid of a terminal?	46
5.9	Setting the visible layers: <code>visible</code>	46
5.10	Some other nice features: the <code>settings</code> menu	46
5.10.1	Customizing <i>seadali</i> : the <code>.dalirc</code> -file	47
5.10.2	Switching on the Sea-of-Gates mode: <code>Image Mode</code>	47
5.10.3	Setting drawing style for instances: <code>draw hashed</code>	47
5.10.4	Setting dominant or transparent drawing style: <code>draw dominant</code>	47
5.10.5	Setting the print command for the <code>hardcopy</code> -button	48
5.10.6	X-window redrawing strategy: <code>backing store</code>	48
5.10.7	In case you were wondering.....	48
5.11	Interrupting Seadali: keyboard input	48
6	'Fish': the layout purifier and design rule checker	51
6.1	What errors does <i>fish</i> detect?	51
6.2	How does <i>fish</i> handle instances?	52
6.3	How to create an empty array of image?	52
7	Automatic placement with 'Madonna'	53
7.1	What is required before I can call 'Madonna'?	53
7.2	Running 'Madonna'	53
7.2.1	From 'seadali'	54
7.2.2	Calling 'madonna' from the command line using 'sea'	54
8	Automatic routing: 'trout'	57

8.1	What is required before I can click Trout ?	57
8.2	Running <i>trout</i>	58
8.2.1	Calling <i>trout</i> from <i>seadali</i>	58
8.2.2	Running <i>trout</i> from the command line	59
8.3	What does <i>trout</i> do for me?	59
8.4	Guidelines for successful routing	59
8.5	What should I do in the case of incomplete routing?	60
8.5.1	Many nets are unconnected	60
8.5.2	A small number of nets is unconnected	61
8.6	The run-time of <i>trout</i>	61
8.7	Additional options of <i>trout</i>	62
8.8	The final assembly of your layout design on a chip	63
8.8.1	The pad ring	63
8.8.2	Connecting terminals to the pad ring	63
8.8.3	Final touches	64
9	Using OCEAN: tips and tricks	67
9.1	Simple troubleshooting: the seadif database	67
9.2	Assembling cells from various projects	67
9.3	Renaming, copying or moving projects: how to proceed	69
10	Logic Synthesis with 'kissis'	71
10.1	How to use <i>kissis</i> ?	71
10.2	Legal options.	72
10.3	What files are required before I can call <i>kissis</i> ?	72
10.4	More info about <i>SIS</i>	72
11	Layout to circuit extraction: 'space' and 'ghoti'	73
11.1	The extractor <i>space</i>	73
11.2	<i>space</i> : Capitalization of the cell name	73
11.3	Purifying netlists: <i>ghoti</i>	74
12	Simulating circuits with 'simeye'	75
12.1	Description	75
12.2	Commands of <i>simeye</i>	75
12.3	Command file for <i>sls</i>	78

12.4 Startup file for simeye	78
13 The sea-of-gates libraries	81
13.1 Introduction	81
13.2 Information of each library cell	82
13.3 digilib8_93: the basic digital library	84
13.3.1 iv110	84
13.3.2 no210	85
13.3.3 no310	86
13.3.4 na210	87
13.3.5 na310	88
13.3.6 ex210	89
13.3.7 buf40	90
13.3.8 mu111	91
13.3.9 mu210	92
13.3.10 de211	94
13.3.11 dfn10	96
13.3.12 dfr11	98
13.4 analib8_93: the analog library	100
13.4.1 NMOS Compound transistor ln3x3	100
13.4.2 PMOS Compound transistor lp3x3	101
13.4.3 NMOS mirrors mir_nin en mir_nout	102
13.4.4 PMOS mirrors mir_pin and mir_pout	104
13.4.5 osc10	106
13.5 bonding8_93: the bonding patterns	108
13.5.1 bond_leer	108
13.5.2 bond_bar	109
13.6 exlib8_93: the extended library	110
13.6.1 buf20	110
13.6.2 dfn102	111
13.6.3 dfr112	113
13.7 The file 'image.seadif'	115
14 Acknowledgments	121
Index	123

Chapter 1

Introduction

1.1 About this manual

This document describes the OCEAN tool suite for creating and verifying Sea-of-Gates layout. First of all: do not get intimidated by the size of this document. It contains a sizable amount of reference material, so you certainly don't need to read all of it to get started. We recommend to read the first chapters, and then do the tutorial in chapter 3.

OCEAN, is a comprehensive chip design package which was developed at Delft University of Technology, the Netherlands. It includes a full set of powerful tools for the synthesis and verification of semi-custom sea-of-gates and gate-array chips. OCEAN covers the back-end of the design trajectory: from circuit level, down to layout and a working chip. In a nutshell, OCEAN has the following features:

- Hierarchical (full-custom-like) layout style on sea-of-gates.
- Powerful tools for placement, routing, simulation and extraction. Any combination of automatic and interactive manual layout.
- Short learning curve makes it suitable for student design courses. Robust and 'combat-proven' on hundreds of people.
- Available for free, including all source code. Running on popular HP and Sun workstations and on PC, easy installation.
- Includes three sea-of-gates images with libraries and template chip. Interface programs for other tools and systems (SIS, Cadence, etc.)

Section 1.2 gives an overview of the structure of the entire system. In semi-custom layout design we use a *master image* (called just *image* here), which is the regular pattern containing the transistors. We explain the way in which OCEAN deals with the images and the cell libraries on those images in chapter 2. In that chapter we will also go deeper into the sea-of-gates layout design style.

To get started with the system, chapter 3 contains a simple step-by-step tutorial. It takes you for a guided tour along the OCEAN tools. Using two examples you will get accustomed with operations like circuit input, circuit simulation, layout synthesis, and layout verification.

In the following chapter the tools in the OCEAN system are described in more detail. Chapter 4 starts with listing all tools that are included in the OCEAN distribution.

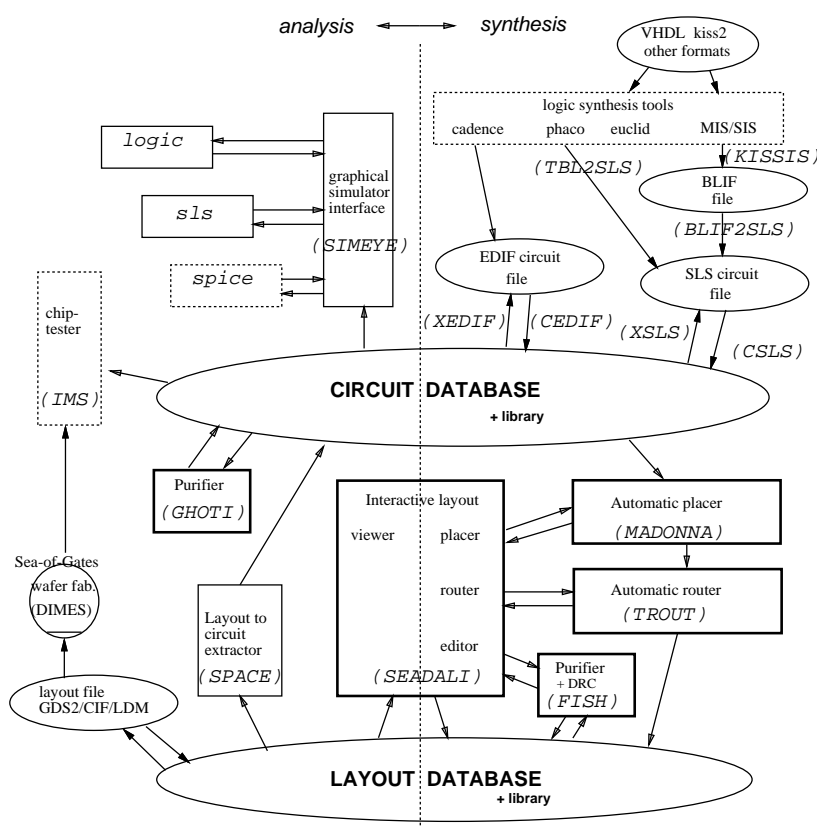


Figure 1.1: The main components in the OCEAN Sea-of-Gates design system. The arrows indicate the design flow. The boxes are programs (the programs in hashed boxes are not included in the distribution). The ovals are files or other means of storage such as database.

Chapter 5 deals with the tool that is at the very heart of OCEAN, namely *seadali*. It is worthwhile to read this carefully, since it contains many useful hints on how to click the buttons efficiently.

In chapter 6 we have a closer look at the program *fish*, a tool to make your layout-cell design rule correct. Then in the sections 7 and 8 we describe how to use the automatic placer and router. These automatic tools allow you to assemble your cell very quickly.

Then, in chapter 12, a manual of the interactive simulator program is included. Finally, chapter 13 describes the 'fishbone' sea-of-gates library that we include with the distribution.

1.2 Overview of the system

To get the full perspective of OCEAN, consider the system architecture in figure 1.1. To the left the analysis tools are drawn, while the synthesis programs are displayed on the right. The bottom half deals with layout, while to top part deals with circuit descriptions. In this way 4 'quadrants' can be distinguished. The two big ovals denote the circuit and layout level descriptions, which are stored in the NELSIS database. In general the design flow in OCEAN looks as follows:

1. Functional design and (if possible) functional simulation.
2. Circuit generation (the first quadrant, right top corner in figure 1.1). The result is a circuit description which is stored in the NEL SIS database. The circuit can be generated by a logic synthesis tool or entered manually using a circuit editor. We support two textual input formats which can be converted into or extracted from the database: *sls* and *edif*.
3. Circuit simulation (the fourth quadrant, left top corner in figure 1.1). The circuit is checked for correctness using simulators like *sls* and *spice*. If necessary, you can modify the circuit and re-simulate it. We just repeat this loop until the simulations indicate the proper operation of the circuit.
4. Layout generation (the second quadrant, right bottom corner in figure 1.1). That is, generating a sea-of-gates layout description which matches the circuit description.
5. Layout extraction (the third quadrant, left bottom corner in figure 1.1), to re-extract the transistor net list from the layout. The extractor also finds the parasitic capacitances and resistances.
6. Circuit simulation again, to verify the layout and to check the effect of the parasitics in the layout. If necessary, the circuit description is modified and a new layout is generated.
7. If the layout is satisfactory, the cell can be used as a son cell at the next level of hierarchy. The top level cell in the hierarchy contains the entire chip. The layout description can be converted into a file for mask generation and chip fabrication.

You will get accustomed with the program names which are related to the design flow of figure 1.1. A list of them can be found in chapter 4. The easiest way to do this, however, is just 'learning by doing'. Therefore we suggest to follow the tutorial of chapter 3.

The two big ovals denote the circuit and layout level descriptions, which are stored in the NEL SIS database. All tools read and write from this database. Nelsis is an open database, which allows you to plug in new tools or formats. For more information on the details of the database, we refer to the *Nelsis IC Design system User's manual*, which can be obtained through the authors.

The hierarchical layout design style is the key feature of OCEAN. It allows you to structure the layout in the same way as the circuit. Larger structured blocks (such as registers, RAM, ALU) can be designed efficiently, much in the same way as on a full-custom chip, but at the cost and design speed of a gate-array. Many levels of hierarchy may be used to smash the complexity and to speed up design. Unlike other standard-cell, gate-array or sea-of-gates design packages, there is no need to break up the entire circuit into a two-level hierarchy with small equal-sized modules. From our experience, OCEAN's unique clear and visible relation between layout and circuit has many advantages. Not only does it give the novice user (e.g. student) a better comprehension of the design, it also allows expert designers and tools to make better use of the inherent structure and regularity of the circuit.

Figure 1.2 depicts the layout of a microprocessor that was developed with the OCEAN sea-of-gates design system. This picture clearly shows that the hierarchy and the regularity of the microprocessor circuit is preserved in its layout.

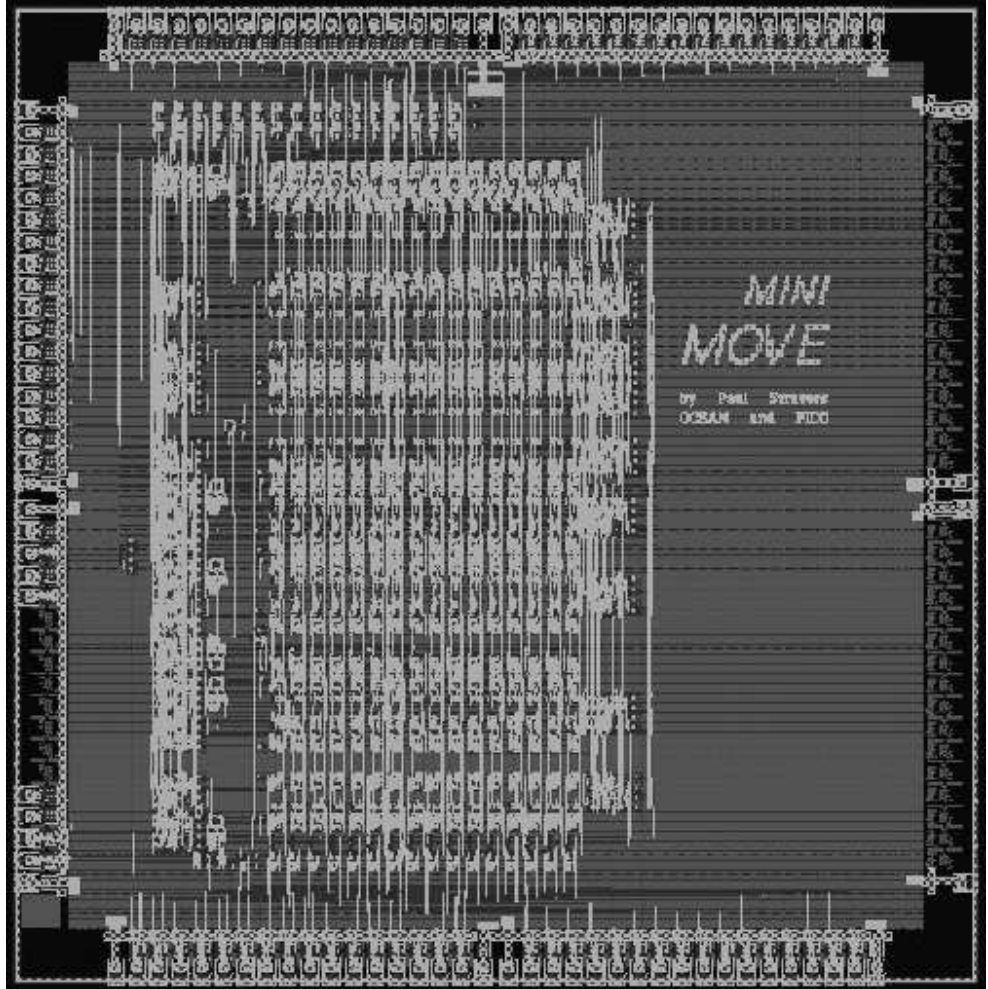


Figure 1.2: The layout of the MINIMOVE microprocessor, developed with the OCEAN sea-of-gates design system. This microprocessor is capable of executing three native instructions in parallel. At a clockspeed of 30 MHz, this allows a maximum throughput of 90 native MIPS. The first stage in the realization of this processor –circuit design and simulation–took 4 man-days of work. The main tools involved where *csls*, *simeye* and *sls*. The second stage of the design –layout design, extraction and simulation– took another 4 man-days of work. This involved the tools *seadali*, *madonna* and *trout* for the layout synthesis and *space* for the layout extraction. Simulation of the extracted circuit was again done with *simeye* and *sls*. Almost the entire layout of the MINIMOVE is based on the library cells from "oplib1.93" that is distributed with the OCEAN sea-of-gates design system. The cells in this library implement static complementary MOS gates and transmission gates on the *fishbone* image.

1.3 How to retrieve the OCEAN system?

The entire OCEAN system is available for free as part of the free SPACE distribution and can be downloaded via the SPACE web-site (<http://www.space.tudelft.nl>). First, a special address to those nasty lawyers, in particular the American ones:

- OCEAN is a sea-of-gates chip design package designed at Delft University of Technology in the Netherlands. Copyright ©1992,1993 Patrick Groeneveld and Paul Stravers.
- The OCEAN package is free software; you can redistribute it and/or modify it under the terms of the ISC License.
- The software and libraries are being provided on an 'as is' basis.
- The software and libraries have no warranties and no provisions for support or future enhancements. So the system is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE, or that the use of it will not infringe any patent or copyright. See the ISC License for more details.
- Delft University and its employees have no liability in connection with the use of the software.

If you have any questions or problems, just contact the SPACE support team:

Nick van der Meijs or Simon de Graaf
Circuit and Systems Group, Electrical Engineering Faculty
Delft University of Technology
The Netherlands
Email: space-support-ewi@tudelft.nl

1.4 How to run the OCEAN tools?

Before you can run the OCEAN tools, you or your system administrator must install the system on your computer. (You do not need root access to install OCEAN, but you do need at least 30 Megabytes disk space). A detailed description of the install procedure is in the file `INSTALLATION` that comes with the OCEAN distribution.

The users of OCEAN tools must set up an environment before they can start. This includes adding a few directories to their `$PATH` environment, and setting a bunch of other environment variables. The same `INSTALLATION` file describes the required user setup in detail.

1.5 Additional information and related documents

Although we tried, this manual is by no means complete. After or even during the reading this document, you might wish to consult the following items:

- *SLS: Switch-Level Simulator User's Manual* by Arjan van Genderen and Sander de Graaf

Describes the logic and switch-level simulator *sls* and the *sls* circuit description language. This manual is included as pdf file in the distribution ('sls.pdf', 43 pages). A related document is *Functional Simulation User's Manual* (Oscal Hol, 46 pages), which describes the use of functional modules in the *sls* simulator. Also available in the distribution as file 'funman.pdf'.

- *Space User's Manual* by Nick van der Meijs, Arjan van Genderen e.a.

Describes the powerful layout to circuit extractor *space* in full detail. Also included as pdf file in the distribution ('spaceman.pdf', 68 pages). Related to this manual is the *Space Tutorial* (30 pages, file 'spacetutor.pdf').

- *SIS: A System for Sequential Circuit Synthesis* by Ellen M. Santovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton and Alberto Sangiovanni-Vincentelli. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720. (1992).

Describes how to use the *SIS* logic synthesis tool from the UC Berkeley. For convenience, we included it as a pdf file in the distribution ('SIS-paper.pdf', 45 pages).

- On-line documentation: *icdman*, *icddoc* or the tool '-h' option
Just type

```
/user/hillary/myproject % icdman simeye
```

to get the tool description of *simeye*. This works similarly for any other nelsis tool. For the OCEAN-tools the '-h' option is useful. They'll print a short summary of their usage. For instance, type 'madonna -h' to get a brief description and a list of options.

- *The NELSIS IC design system manuals*

A three-volume set of manuals of the NELSIS design system. The previous manuals and items are contained in this set. Available through the authors.

- *Studentenhandleiding Ontwerppraciticum*

This is the elaborate student's manual (250 pages) for the second year chip design course at Delft University of Technology. Unfortunately it is in Dutch. It contains all information the students need. Available from the Diktatenverkoop electrical engineering, or through the authors.

Chapter 2

The Sea-of-Gates image

2.1 Why Sea-of-Gates?

Before we explain how to use the OCEAN layout tools, it is important to understand the kind of layout that they produce. OCEAN tools produce *sea-of-gates* layout. This means that they deal with a prefabricated *image* consisting of MOS transistors. To implement a circuit, the tools (or the designer, for that matter) interconnect these transistors with metal wires. The sea-of-gates layout strategy aims at four goals:

1. *Minimization of the fabrication time.* Because the chips are prefabricated (the transistors are already on the master image), the silicon foundry only processes the masks related to metal wires. As compared to full custom layout, the number of masks processed by the silicon foundry is often reduced by more than 60%.
2. *Minimization of the design time.* The time involved in designing a cell layout is reduced dramatically (as compared to full-custom) because the transistors are pre-placed on the image. Typically, it takes only a few minutes to layout a flipflop or a combinatorial gate, and the designer does not need to know much about the process design rules.
3. *Minimization of the chip cost.* The layout design starts with a prefabricated master image. This is a semi-manufactured article that can be produced in large quantities. Consequently, it can be cheap.
4. *Full-custom properties on Semi-Custom chips: Efficient implementation of structured logic (RAM, PLA etc.).* In contrast to the conventional Gate-Array, a sea-of-gates image does not have pre-defined routing channels. This enables a much more compact and clean implementation of structured circuits such as processors.

The OCEAN suite of tools handles a wide variety of sea-of-gates master images and process technologies. Three of them are included in the distribution:

- fishbone* A gate-isolation image in a 1.6μ CMOS process with two layers of metal (figures 2.1, 2.4 and 2.6).
- octagon* A remarkable octagonal image in a 0.8μ CMOS process with three layers of metal interconnect (figure 2.2 and 2.5).
- gatearray* An old fashioned gate-array in a 4μ single metal layer CMOS process (figures 2.3 and 2.5).

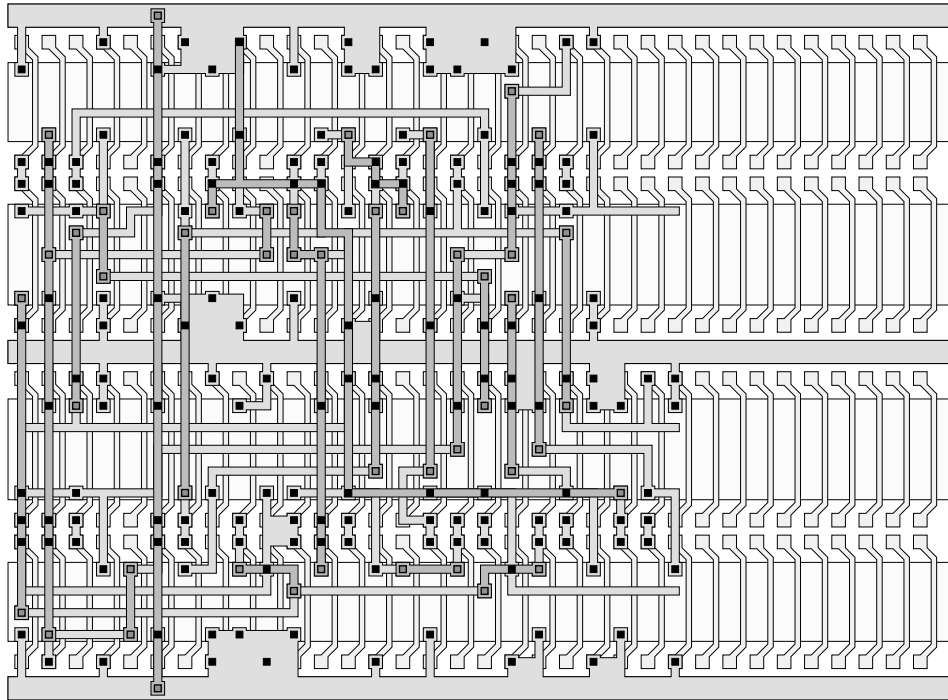


Figure 2.1: A circuit on the *fishbone* sea-of-gates image. This image was developed at Delft University of Technology. Our chip with this image contains 191,312 transistors on an area of approx. $10 \times 10 \text{ mm}^2$. 144 pins are available.

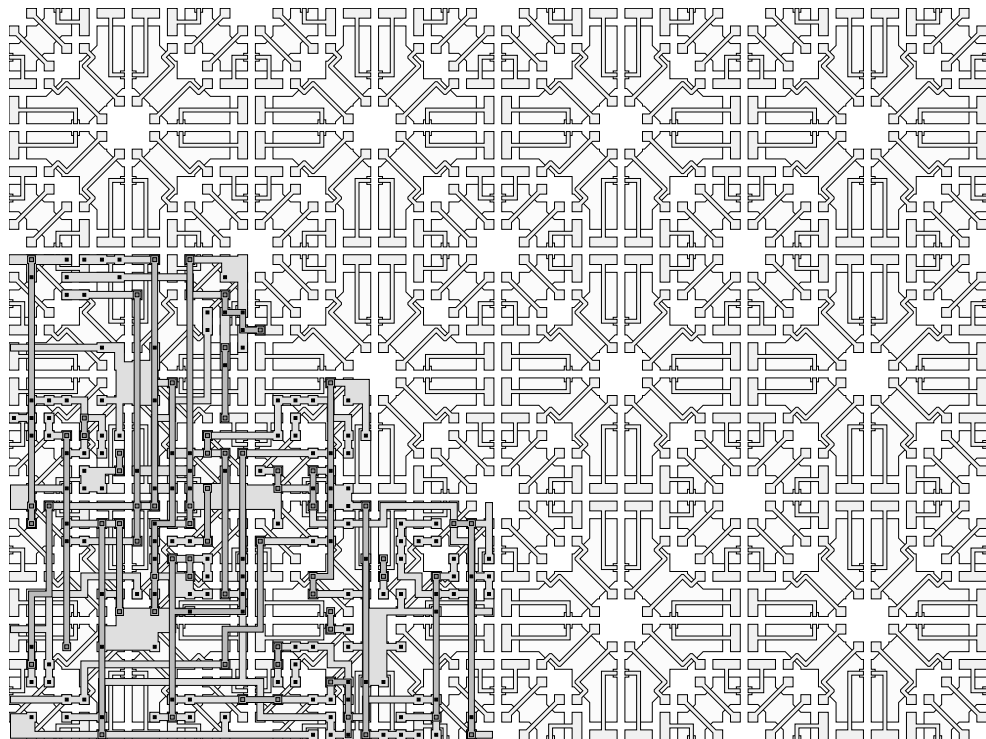


Figure 2.2: The *octagon* image which was developed at Twente University in the Netherlands. In the lower left corner the same circuit as in figure 2.1 is implemented. This highly symmetrical image allows the cells to be rotated.

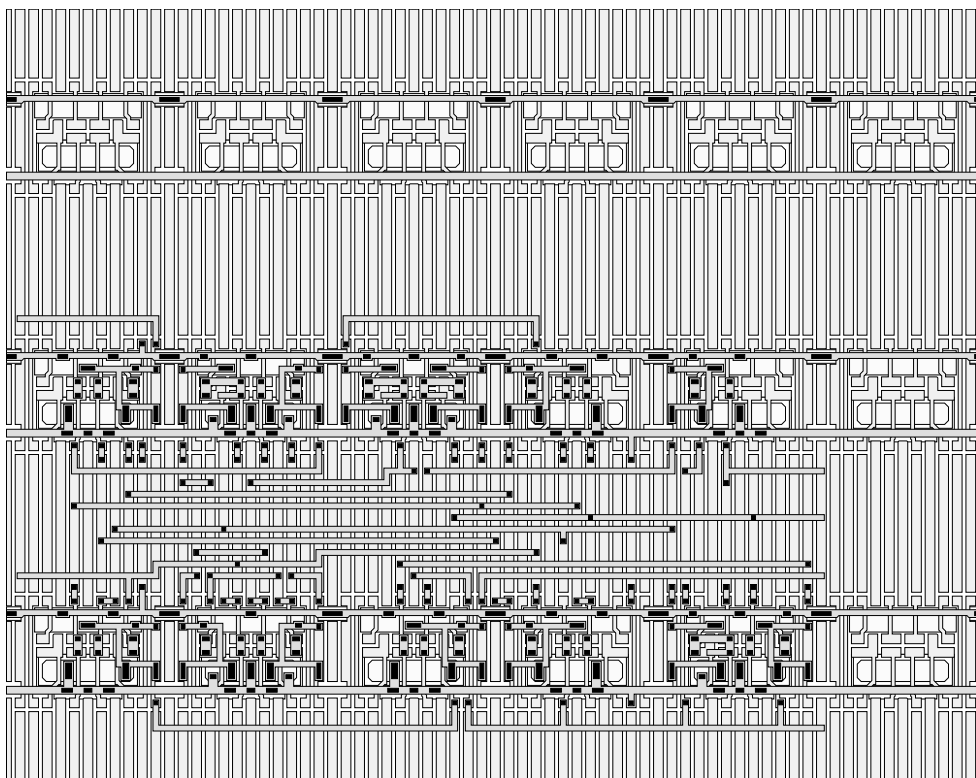


Figure 2.3: The single-layer *gatearray* image is a traditional row-based gate-array. The figure shows the same circuit as in the previous figures.

Although the three images differ from each other in many respects, the OCEAN tools can take advantage of their specific properties. For instance, the placer *madonna* uses the properties of the octagon image to mirror cells with respect to the 45° mirror axis. And the router *trout* takes advantage of the vertical poly strips in the gate array image to find the best route for a wire. Especially in the single-layer *gatearray* image the polysilicon strips are an essential feature for the routing.

In figures 2.4 and 2.5 the implementation of a 2-input nand is shown for each of the three images. Notice that the layout style is very different, even for such a simple circuit.

Currently, our local foundry DIMES runs a production line for metallization of the *fishbone* master image, designed at Delft University of Technology. It therefore makes sense to concentrate on this image and go into some of its details. The other images are supplied mainly to demonstrate the features of the placer and the router in the OCEAN system.

2.2 The TU-Delft fishbone chip

At Delft university of technology we developed a sea-of-gates chip. Figure 1.2 (on page 4) shows one such chip. Its core contains 191,312 transistors in the *fishbone* image. Each fishbone row is a series connection of 1087 transistors. There are 88 n-transistor rows and 88 rows with p-transistors. The chip has 144 pins, of which 128 are user programmable as input, output, bi-directional or direct pins. The remaining 16 pins are for power connections. Generally we combine a number of separate designs on one chip. For this purpose we created a multi-project chip concept, in which the chip is partitioned into 4, 3 or 2 separate 'pseudo-chips', depending on the sizes of the circuits. The pseudo chip are

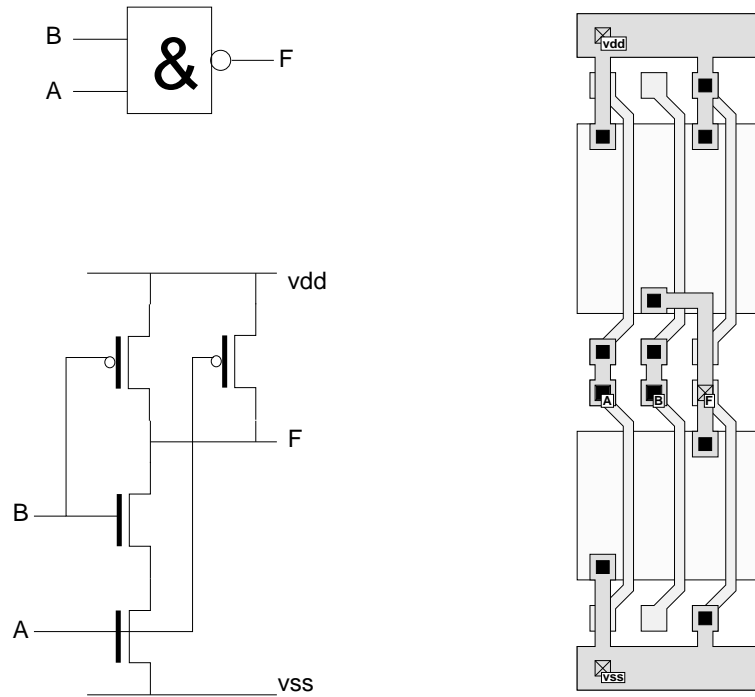


Figure 2.4: A two-input NAND gate and its layout on the fishbone sea-of-gates image. This is the library cell 'na210'.

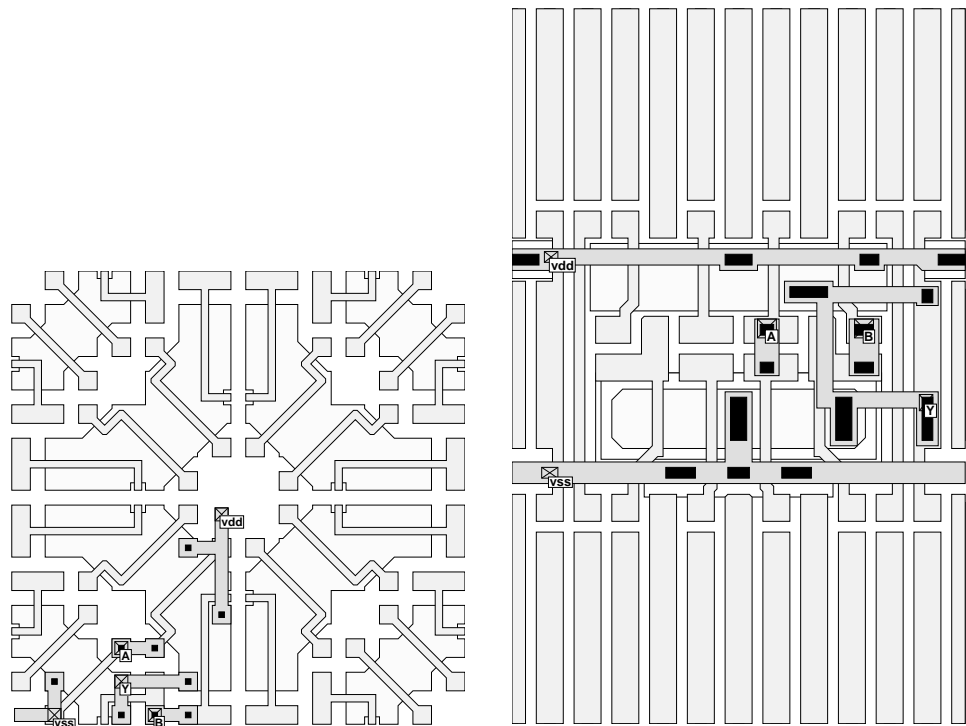


Figure 2.5: The same two-input NAND gate on the octagon (left) and the gatearray (right) image.

process:	C3DM (Philips) 1.6μ double layer CMOS, $\lambda = 0.2\mu m$
dimensions:	bruto: $10 \times 10mm$, <i>fishbone</i> core area: $8.7 \times 8.7mm$
transistors:	191,312 transistors in 176 rows of 1087, <i>fishbone</i> image.
pins:	144 bonding pads, 128 user programmable, 16 power
transistor sizes:	$1.6\mu m \times 23.2\mu m$ (nmos) and $1.6\mu m \times 29.6\mu m$ (pmos)
transistor pitch:	$8.0\mu m$ (horizontal)
metal wire width:	$2.4\mu m$ (both metal 1 and metal 2)
contact size:	$2.0 \times 2.0\mu m$ (hole), $4.0 \times 4.0\mu m$ (incl. metal overlap)
poly gate resistance:	$700\Omega m$ (nmos) $950\Omega m$ (pmos)
contact resistance:	$22\Omega m$ (ps-in), $0.13\Omega m$ (in-ins)
metal resistance:	$0.056\Omega m/$ (in), $0.026\Omega m/$ (ins)
threshold:	$V_t = 0.7V$ (nmos), $V_t = -1.2V$ (pmos)

Table 2.1: Summary of the parameters of the TU Delft sea-of-gates chip with the *fishbone* image.

isolated from each other and have separate power supply connections.

125 Wafers containing the raw transistor pattern have been pre-produced by a commercial foundry. At our own chip production facility (DIMES) the chips are metalized with a very short turn-around time. Each 4-inch wafer contains 46 usable chips. In the past year (1992-1993) 12 wafers with different designs have been produced. At this moment we (that is, the guys from the OCEAN team) are trying to make this cheap production facility available to other than university users. Please contact us if you are interested.

2.3 The general structure of the fishbone image

Figure 2.6 on page 12 shows the structure of the fishbone image. This image is fabricated in the C3TU process, the Delft derivative of the Philips C3DM process. The length of a transistor gate is 1.6μ . The designer can program two layers of metal and a number of contact layers. Each layer is referred to as a 2- or 3 character acronym:

1. **in** (metal1, this is the metal layer that is closest to the image)
2. **ins** (metal2, this is the second level metal layer)
3. **con**, **cop** and **cps** (contacts from **in** to n-type diffusion, p-type diffusion or poly)¹
4. **cos** (contact between **in** and **ins**)

The fishbone image consists of alternating rows of n-type and p-type transistors. Each row is a seemingly endless series connection of transistors.

Wire segments and contacts may only be placed at certain “grid points”. *Seadali* optionally marks these grid points with little white dots. Once you have grown accustomed to the image, you will know these positions by heart. In figure 2.6 the gridpoints are indicated with \times marks. In the vertical direction they have been numbered 0 ... 28.

¹For you as a designer, these three masks are equivalent. The OCEAN tools (*fish*) take care of selecting the proper one of these three.

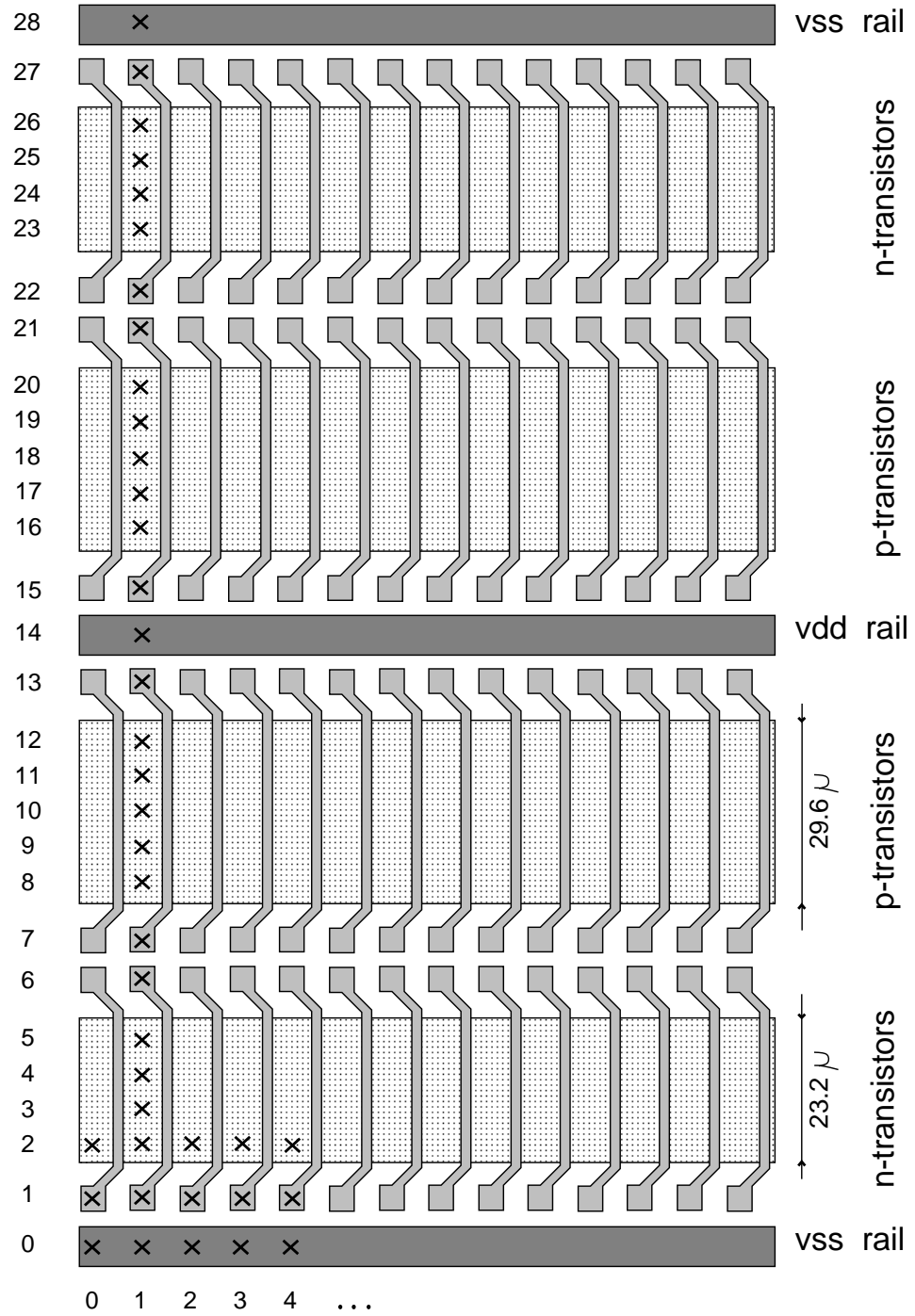


Figure 2.6: The basic structure of the fishbone sea-of-gates image.

2.4 Restrictions on placing contacts

Unfortunately it is not allowed to place a contact at every grid position. You have to take the following design rules for contacts into account:

1. Contacts may not be stacked. It is therefore not allowed to place a **con**, **cop** or **cps** contact on top of a **cos** contact (metal1-metal2). As a result, all connections between diffusion (or poly) to metal2 have to “dogleg” in metal1.
2. On top of (poly) gate contacts (grid rows 1, 6, 7, 13 in fig. 2.6) **cos** contacts (metal1-metal2) are not allowed. All **cos** contacts must be placed on the diffusion tracks (2-5 and 8-12) or on the power supply (0 and 14).
3. Substrate contacts can be placed under the power rails (rows 0 and 14). As a rule of thumb, place one substrate contact for every 3 transistors. The placement of the substrate contact can be performed automatically.

Needless to say that the OCEAN tools take these rules into account.

2.5 Transistor layout on fishbone: gate isolation technique

The gate of each transistor consist of a polysilicon pattern. Each transistor gate can be reached at two grid points using a **cps** contact. In figure 2.6 the n-transistor can be connected in grid rows 1 and 6, the p-transistor in rows 7 and 13. Notice that the p-transistor is taller than the n-transistor.

The source and drain of the transistors (diffusion) can be connected to grid rows 2 through 5 (n-transistor) and 8 through 12 (p-transistor) using a **con** or **cop** contact², respectively.

Adjacent transistors (of the same type) share their source/drain connections. Therefore, in each row, the transistors are arranged in a seemingly endless series connection. This does not imply, however, that only series connections can be programmed on this image.

To isolate circuits from each other, so called “isolation-transistors” are used. An n-type isolation transistor is a transistor of which the gate is connected to vss. Therefore it is always switched off. In this way the source and drain of this transistor are “isolated”. Similarly a p-type isolation transistor can be created by connecting its gate to vdd. A 2-input NAND gate requires two isolation transistors. This is shown in figure 2.4 (page 10), where the two rightmost transistors serve the purpose of isolation. As a consequence we are able to place several NAND gates next to each other, without causing any short circuits between the gates.

It is clear that a layout cell only needs isolation transistors on either its left or its right side, but not both, as long as this strategy is applied to all the cells on the chip. We will always take the right side of a cell to accommodate the isolation transistors, like it’s done in figure 2.4. Note, however, that we can only do this because the fishbone image does not allow to mirror a cell in the y-axis!

²If you are using *seadali*, just select an arbitrary mask from **cps**, **con** or **cop**. The OCEAN-tools will take care of selecting the proper mask

2.6 Power net strategy

The fishbone image consists of alternating rows of n-type and p-type transistors. Each row is a series connection of many, many transistors. Between the rows there are two power “rails” in metal1 which run horizontally across the entire width of the chip. Vertically we use a scheme of alternating VSS and VDD power rails. Next to a VSS-power rail there are two rows of n-type transistors, next to a VDD-power rail there are two rows of p-type transistors. This results in a nnppnnppnnpp.... order of transistor rows (figure 2.6).

Under the power line well contacts (substrate contacts) are placed to keep the wells at the appropriate voltage. In our system the well contacts will be added automatically at the last stage.

2.7 Some practical hints

You will soon notice that the wiring space is limited. In order to prevent unconnected points it can be useful to follow these guidelines:

- Avoid using long horizontal metal2 wires, because they may block future vertical metal2 wires. Use metal2 mainly for vertical wires.
- Use metal1 mainly for horizontal wires. The layout of the power wires almost force you to do this.
- Design each cell such that its “permeability” is maximized. That is, leave as much useful free space as possible. Always bear in mind that the routing capacity of a cell is desperately needed (by you or an automatic router) at a higher level in the hierarchy.
- Often the routing capacity in the horizontal direction is the bottleneck. Leave an unused transistor row if necessary.
- Ensure that all terminals of a cell can be reached from some point outside that cell. Remember that stacked contacts are not allowed, and that a connection to metal2 therefore requires at least a contact-less spot on a wire.
- Take advantage of the fact that the fishbone image contains series connections of transistors of the same type (n-mos or p-mos). Try to find chains of n-mos transistors in the circuit diagram of your cell and lay these chains out on the fishbone image. Isolate the chains from each other with isolation transistors. Follow the same procedure for p-transistor chains. If you are very lucky, the entire circuit can be implemented with only one n-chain and one p-chain. An example of such a circuit is the D-flipflop depicted in figure 2.7.
- Sometimes, when routing area is in short supply, it may be necessary to use a poly gate to make a vertical connection. Poly, however, has a high electrical resistance. In combination with the parasitic capacitance of a long metal wire, this can significantly slow down signal transmission times on such a wire. As a rule of thumb, you should only use poly strips on a short distance of signal inputs and never in the neighborhood of signal outputs.
- Do not place substrate contacts. They will be added automatically by the router in the last stage of the design.

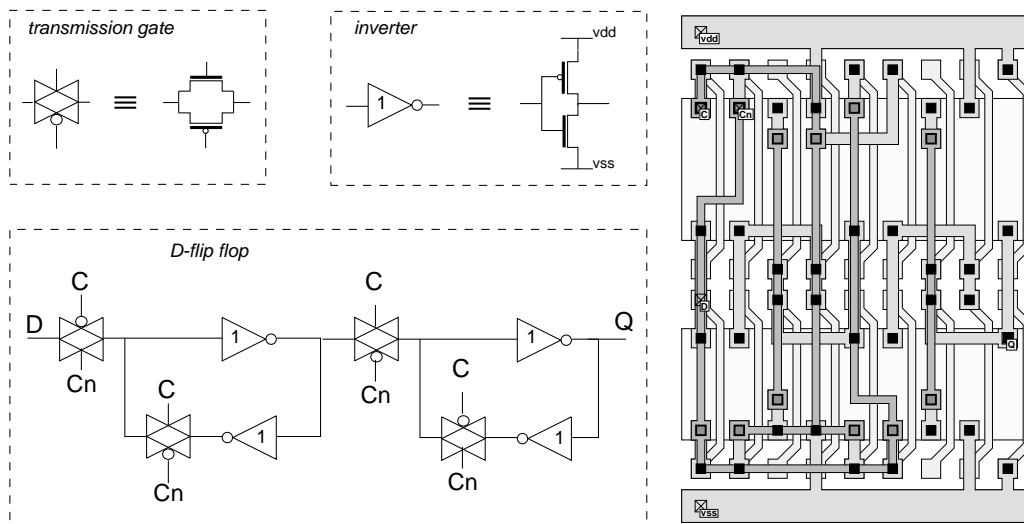


Figure 2.7: Circuit diagram of a D-flipflop and its layout on the fishbone image.

Chapter 3

A quick tutorial

3.1 Let's get behind the terminal

This chapter takes you for a guided tour along the OCEAN tools. Using two examples you will get accustomed to operations like circuit input, circuit simulation, layout synthesis, and layout verification. This tutorial contains three exercises to introduce you to the OCEAN sea-of-gates layout tools and to the NEL SIS tools for circuit extraction and simulation.

The first exercise (section 3.2) deals with the design of a simple library cell, a 3-input NOR gate. It shows a powerful aspect of the OCEAN environment, namely that you can get very close to the sea-of-gates image to optimize your design for area or speed. You do not depend on a cell library because you can very easily design a cell library on your own.

The second example (section 3.3) shows the hierarchical approach of the OCEAN tools. In contrast to most sea-of-gates systems, OCEAN does not require a flat circuit description of the design. The hierarchy present in the designer's circuit is retained in the sea-of-gates layout.

The third example deals with using the logic synthesis tool SIS with the OCEAN system. It shows how easy you can translate the description of a Finite State Machine (FSM) into a Sea-of-Gates layout.

The examples are intended to be self-contained and can be dealt with independent from each other. For instance, you can immediately skip to section 3.3 since it does not assume that you have read section 3.2. We advise, however, to do them in the indicated order.

The examples use the *fishbone* image. You can also run them in the other two images. This will be explained in section 3.5.

We distinguish four major steps in the design cycle:

1. Creating the circuit level
2. Simulating the circuit on circuit-level.
3. Creating layout from the circuit.
4. Extraction and simulation of the layout.

Keeping this design flow is crucial for the hierarchical design strategy. In figure 1.1 on

page 2 an overview of the system and its design flow is given. To the right the synthesis tools are displayed (steps 1 and 3), while the analysis tools can be found on the left (steps 2 and 4).

3.2 Designing a simple library cell

The purpose of this section is to get you accustomed to circuit design on the 'fishbone'-image using the OCEAN tool suite. We will start by designing the simple 3-input nor which is shown in figure 3.1. Normally you'll import such a simple cell from the library and you don't have to bother so much making wires by hand. It is very illustrative, however, to take you through all design steps using a simple example.

3.2.1 Before we start: making a project

In OCEAN we perform the design process in a 'project'. This is nothing other than a UNIX directory tree which contains a database. For the tutorial we have a simple command to create the project. Type¹:

```
/user/hillary % tutorial hotel
/user/hillary % cd hotel
```

This puts you in a project directory that contains all the data related to both examples which we use. You must start all programs of OCEAN and NELIS in a project directory.

How to make an empty project? The command *mkopr* <name> creates an empty project for you. This project automatically imports all library cells. See section 5.1 for more details.

3.2.2 Creating a circuit description

- In figure 3.1 the schematic is already given. Normally you'd have to draw the schematic yourself, or create it using other tools. In this case the schematic of the 3-input nor is very simple: It consists of 3 parallel n-enhancement MOS-transistors (bottom of figure 3.1) and 3 p-enhancement MOS-transistors in series (on top).
- We describe the circuit in the **sls**-circuit language². First we have to identify the terminals:

input terminals: Our nor has 3 inputs: A, B and C.

output terminal: The output is tagged F.

power terminals: Power connections must be called **vss** and **vdd**.

internal terminal: This circuit has two internal nodes which we called v1 and v2. You can give them any name you want.

It would take a lot of words to describe the **sls**-syntax, so let's just look at the example file in figure 3.2. The first line declares the network 'nor3' and its (external)

¹The string `/user/hillary %` is the prompt, so don't type it!

²At the moment we do not yet have a stable schematic editor to perform the task of schematic entry. You'll soon notice, however, that the **sls**-language is powerful enough to describe anything you want.

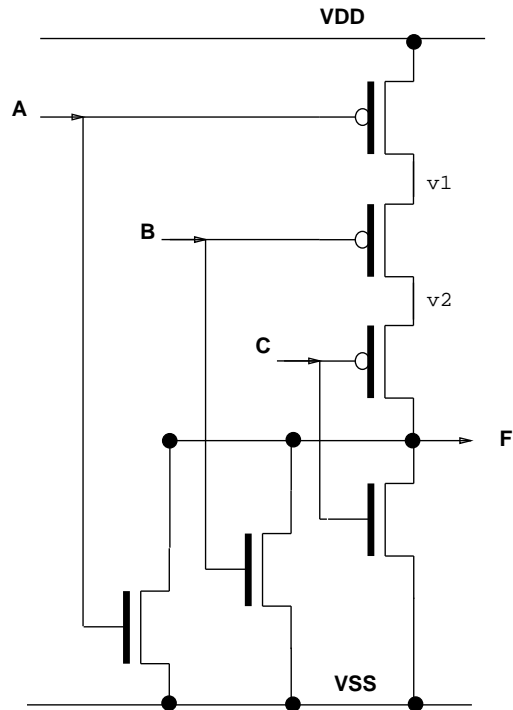


Figure 3.1: The schematic of the 3-input nor circuit.

```

network nor3 (terminal A, B, C, F, vss, vdd)
{
    penh w=29.6u l=1.6u (A, v1, vdd);
    penh w=29.6u l=1.6u (B, v1, v2);
    penh w=29.6u l=1.6u (C, v2, F);
    nenh w=23.2u l=1.6u (A, F, vss);
    nenh w=23.2u l=1.6u (B, F, vss);
    nenh w=23.2u l=1.6u (C, F, vss);
}

```

Figure 3.2: The `s1s` circuit description of the 3-input nor.

terminals³. Between the parenthesis follows the description: 6 transistors. Each line declares one instance (in this case a transistor) and the way it is connected. The type, width and length of the transistors must be specified. Between the brackets are its terminal connections. For the transistor the order of the connections is *gate*, *source* and *drain*. The first transistor, for example, is a p-type enhancement which has its gate connected to A, its source to node v1 and its drain to vdd.

You are lucky. The file containing this sls-description already exists in the tutorial directory. It is called `nor3.sls`. You can check its contents using your favorite editor.

- The circuit description must be entered into the Nelsis database. This is done using the program `csls`:

```
/user/hillary/hotel % csls nor3.sls
```

In this way the contents of the file is translated into the binary circuit format of the NELSIS database. You can check whether the circuit is there:

```
/user/hillary/hotel % dblist
```

This command lists the contents of the current database.

3.2.3 Simulating your circuit

- Start the simulation interface::

```
/user/hillary/hotel % simeye &
```

A window will appear with many fancy buttons. Move the mouse to the widget `cell:` (on the left top). Type there the name of the circuit ('nor3').

- Just start simulating! Click `sls-logic` and then `run` to perform logic simulation. After a short while the simulation results will appear in the window. Check that the circuit really implements a 3-input nor.

Just play around a bit with the buttons. You can zoom in and out (`in` , `out` and `full`) and using the arrow keys you can pan the window.

- The simulation requires a 'stimuli' file in which the input voltages and other simulation commands are specified. The tutorial directory already contains a proper stimuli-file for this example (called `nor3.cmd`). You can modify the input stimuli in this file, or graphically in *simeye*. In a later section we will show how you can do this yourself. Let's first concentrate on the basics.
- There are 3 simulators available from *simeye*: a logical simulator, a switch-level simulator and the *spice* simulator. In this way you can simulate at three levels of accuracy.

The button `sls-logic` (which you just pressed) performs a logical simulation of the circuit. It assumes that the transistors are ideal switches and that the network has no parasitics. Therefore there is no delay in the circuit, and the rising and falling edges are sharp.

- Now try the button `sls-waves`, followed by `run`. This performs a switch-level simulation of the circuit. The switch-level simulator uses a simple model for the parasitics and gives a reasonable accurate timing result. Check that the output signal is now (slightly) delayed.

³also called the 'ports' of the circuit

- Finally, try `spice` for the most accurate simulation of the circuit. Notice that the output waveforms are now continuous, and look much more like the real-life signal. You will notice that it will take much more time before the results show up on the screen. *Spice* is notorious for its load. From our experience you can only simulate networks up to several hundred transistors using *spice*.
- The small buttons labeled `A` and `D` are toggles that either cause an “analogue” or a “digital” interpretation of the signals to be displayed. *Spice* only allows an analogue interpretation of course.
- For now, we’ve completed the circuit simulation. Iconify the window and get ready for the next step.

3.2.4 Creating a layout

At the heart of the layout generation is *seadali*. It allows you to create layout manually, automatically or both. Chapter 5 describes *Seadali* in greater detail. Here we only outline the minimal commands to get started.

- Start *seadali*:

```
/user/hillary/hotel % seadali &
```

Click `boxes` to enter the box manipulation menu.

- Click `fish` to create an empty piece of image. In fact, you are looking at a small (20×4 transistors) portion of the sea-of-gates chip. Have a good look at the structure. Play around with `visible` (in the main menu) to activate masks. The bigger transistors (along the middle power rail) are p-type. The smaller ones are n-type. The white dots denote the grid points.
- Now let’s think about the layout of our nor3. Try to get some inspiration from the way in which the nand-2 circuit was implemented (figure 2.4 on page 10). The nor-3 is the reverse of the nand-2. We need 3 n-type transistors (from the bottom row) in parallel. We also need 3 p-type transistors in series.
- Enough philosophy, let’s click!. First enable the layer in which you want to add wires by switching the bulb `in` (= metal1 mask code) on. It is the dark blue one in the bottom row of the window. Click `APPEND` (in the boxes menu) and try to make some wires. Start, for instance, in the left-bottom by making a vertical wire of 2 grid points long. The bottom of that wire should touch the lower power rail.
- Add the individual wires. If you want, you can also use the second metal layer (**ins**, light blue) to cross a wire in **in** the first layer.
- Contacts to the transistors are made in the **cps**-mask. Click `cps` and disable the other masks to add them. The contacts are a bit harder to distinguish on the screen because they are black.
You can make contacts between metal1 (**in**) and metal2 (**ins**) by adding boxes in the **cos**-mask. These contacts are white. Do not stack **cos** contact on top of **cps** contacts!
- Now press `fish` to check and purify the layout. *Fish* will make the layout design-rule correct by rounding all the wires to the nearest grid points and substituting the proper contact patterns. It will also warn you for errors, such as stacked contacts

or **cos** contacts on top of a poly gate. You can press any time and as often as you want. Just try it.

- To isolate the cell from its surroundings we add an isolation transistor on the right side. Just connect the first unused n-type and p-type transistors on the right to their nearest power line. The gate of the n-type transistor must be connected to **vss**, the gate of the p-type to **vdd**. Do not forget the contacts, and press as often as you want.
- Once the layout looks like it is really going to work it is time to add the terminals. Click , and and switch only the **in**-mask on. Click at the terminal position⁴ and enter the terminal name. Give the terminal the same name as in the circuit description. Only add the external terminals (**A**, **B**, **C**, **F**, **vss** and **vdd**). Put the power terminals on a spot on the power rails.
- Write the layout in the database under the name **nor3**. You'll find the necessary buttons in the menu.

3.2.5 Extracting the layout and simulating it

We can verify the correctness of the layout which we just created by extracting the transistors from the layout using *space* and then simulating it again:

- Start the circuit extractor:

```
/user/hillary/hotel % space -c nor3
```

The result is a circuit cell of which the character of the name has been capitalized: "Nor3". The capitalization is necessary to distinguish between the original circuit and the extracted circuit. Just for fun, have a look at it by extracting the sls-description from the database:

```
/user/hillary/hotel % xsls Nor3
```

Do you recognize the circuit? Why are there so many transistors? Notice that the circuit also contains the (parasitic) capacitances.

- Let's get rid of the unused transistors:

```
/user/hillary/hotel % ghoti Nor3
```

Ghoti strips all unused transistors from a circuit description. It is essential to run it before running *spice*. If you skip *ghoti*, *spice* aborts with complains about "singular matrices". After running *ghoti* have a look at the circuit again:

```
/user/hillary/hotel % xsls Nor3
```

Only 6 transistors and the capacitors remain. With some fantasy you should now be able to recognize your nor-3.

- Start a new *simeye* and simulate the circuit. This works exactly the same way as was described in 3.2.3. The only difference is the name of the circuit ('Nor3' instead of 'nor3'). Compare the results with the original circuit.

You may want to go to the menu of *simeye*. There you could click the button, type "0.5" return, click , type "y" return, and then simulate again. How fast is the nor gate?

⁴This must be an **in**-wire or an **ins** wire, depending on which mask was activated. Other masks cannot be used for terminals

3.3 The second example: a hotel switch

This section describes the implementation of a simple circuit *hotel switch* on the fishbone image. The circuit size is about 24 gates, and its implementation uses a three level hierarchy. You can play with the hotel switch yourself by typing the following commands.

```
/user/hillary % tutorial hotel
/user/hillary % cd hotel
```

This puts you in a project directory that contains all the data related to the hotel switch example. (Do not execute the `tutorial` command if you already did it for the example of section 3.2.)

3.3.1 Functionality

The function of the hotel switch is to control a light bulb in a hotel room by means of three switches `s1`, `s2`, `s3` distributed throughout the room. Whenever one of the switches is pushed the bulb switches on if it is off and vice versa. There is also a “freeze” switch to prevent the bulb from changing state, see Figure 3.3.

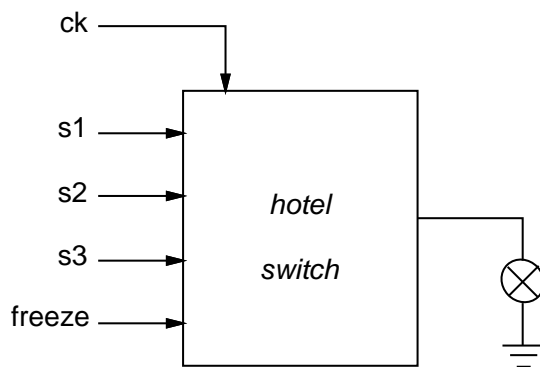


Figure 3.3: Functional diagram of the hotel switch

3.3.2 A Moore machine

The functionality of this hotel switch is implemented by a Moore-type state machine. First we observe that we can combine the three buttons `s1`, `s2`, `s3` by making a logic OR function $s = s1 + s2 + s3$. The state diagram of the hotel switch then looks as depicted in Figure 3.4.

3.3.3 A three level hierarchy

With the aid of circuit synthesis tools like Berkeley SIS, MIS or Philips PHACO the state diagram of Figure 3.4 is translated into a combinational circuit that we call *hotelLogic*. Together with a 2-flipflop state register and a NOR gate this implements the hotel switch as depicted in Figure 3.5.

Section 13 lists the gates that are available from the fishbone cell library *oplib*. Figure 3.6 shows the three level hierarchy of the hotel switch as implemented with the cells from

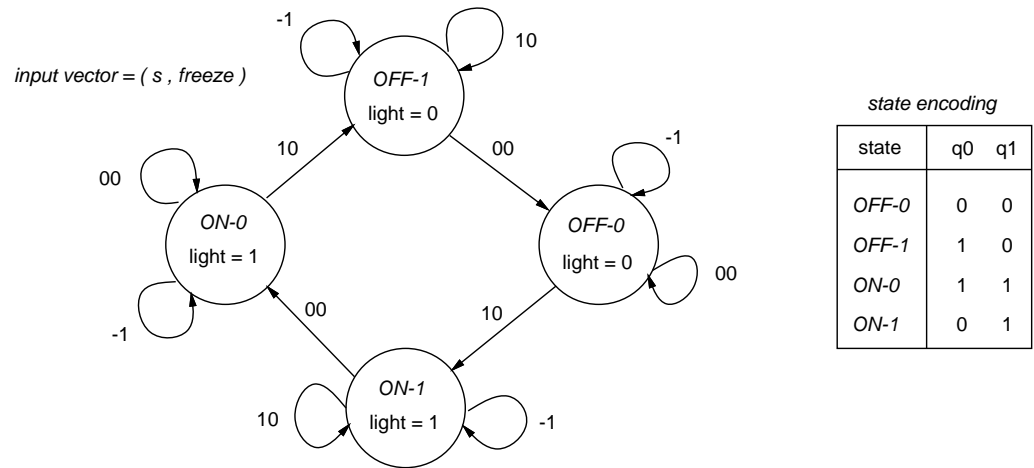


Figure 3.4: State diagram of the Moore machine

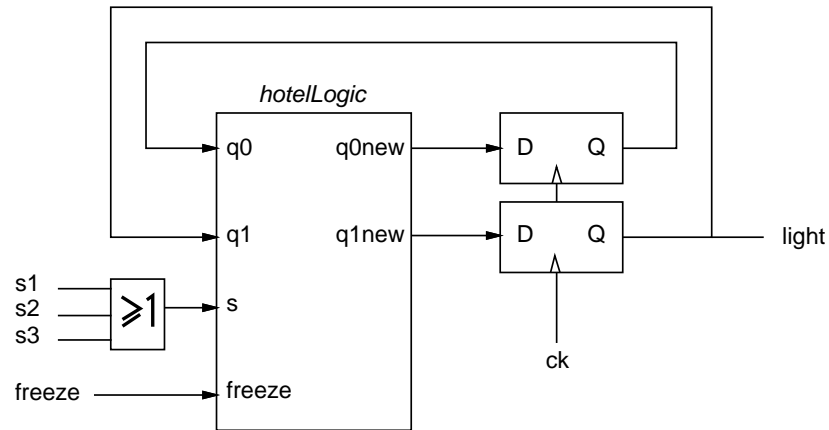


Figure 3.5: Schematic of the highest hierarchical level

this library. Since the library *oplib* does not provide a 3-input OR gate, it is implemented with two gates *no310* and *iv110*.

The circuit appears in the project directory as 2 ascii files named **hotel.sls** and **hotelLogic.sls**. You can use a text editor to view their contents. Now compile these circuits into the database by typing

```
/user/hillary/hotel % csls hotelLogic.sls
/user/hillary/hotel % csls hotel.sls
```

You may want to check that it really worked by extracting a copy of the circuits from the database:

```
/user/hillary/hotel % xsls hotel
/user/hillary/hotel % xsls hotelLogic
```

It should look like in figure 3.7.

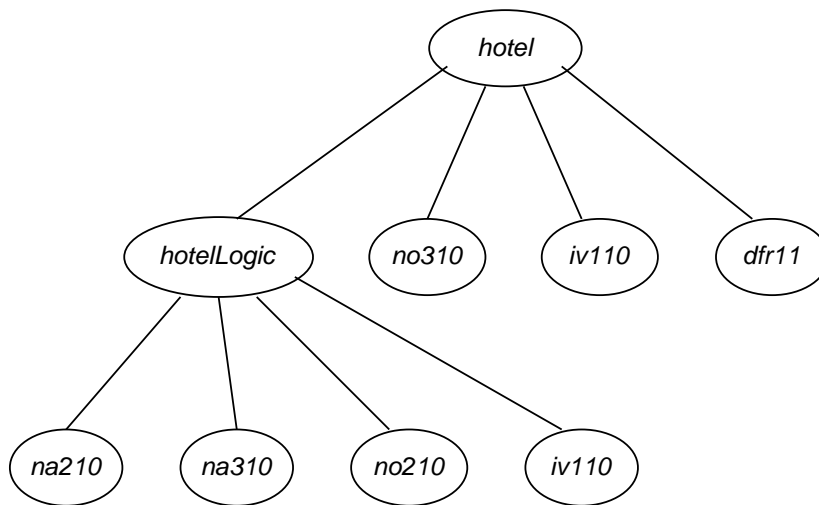


Figure 3.6: The three level hierarchy of the hotel switch

3.3.4 Layout implementation with the OCEAN tools

The hierarchy of the layout is similar to the circuit hierarchy of Figure 3.6. First we create the layout of *hotelLogic* by automatically placing and routing it. Then we use this piece of layout as a sub cell (instance) of the top level cell *hotel*. To make each cell, we traverse along some tools. This design flow in the OCEAN environment is depicted in figure 4.1 (page 36).

- Start *seadali* as described in section 5.2. Click **inst_menu**, then **Madonna**. Enter the circuit name *hotelLogic*, then click **DO IT**. After a few seconds a placement appears. If you don't like the result, you may want to read section 7.2 and see what buttons you can click to influence the placement.

Madonna has created a placement containing all the instances (= child cells) in the netlist description. She tried to place them in such a way that to total wire length and the total area are as small as possible.

- Now route the placement. First go to the **boxes** menu, then click **trout** and **DO IT**.

You will notice that *trout* created the necessary wires to connect the terminals according to the netlist specification. The pattern which *trout* created is indicated in bright colors, the existing layout of the child-cells (instances) is drawn in a hashed pattern.

- Go to *seadali*'s database menu and write the resulting layout to the database using the name *hotelLogic*.
- Just for fun, let's play around a bit with the router. Go to the **instances** menu and try to move one (or more) instances (use **move**). Go back to the boxes menu and click **trout** again. Click **erase wires** before **DO IT**, to cause *trout* to remove the existing wires in the layout. If you would like to try other features of the router, you can proceed to section 3.5.

```

network hotelLogic(terminal s, freeze, q0, q1, q0new, q1new, vss, vdd)
{
    {inst1} na210(s, q0, q0new_1, vss, vdd);
    {inst2} na210(freeze, q0, q0new_2, vss, vdd);
    {inst3} na310(s_n, freeze_n, q1, q0new_3, vss, vdd);
    {inst4} na310(q0new_1, q0new_2, q0new_3, q0new, vss, vdd);
    {inst5} no210(freeze_n, q1_n, h_1, vss, vdd);
    {inst6} no210(q0, q1_n, h_2, vss, vdd);
    {inst7} no210(h_1, h_2, q1new_1, vss, vdd);
    {inst8} na310(s_n, q0, q1, q1new_2, vss, vdd);
    {inst9} na310(s, freeze_n, q0_n, q1new_3, vss, vdd);
    {inst10} na310(q1new_1, q1new_2, q1new_3, q1new, vss, vdd);
    {inst11} iv110(s, s_n, vss, vdd);
    {inst12} iv110(freeze, freeze_n, vss, vdd);
    {inst13} iv110(q0, q0_n, vss, vdd);
    {inst14} iv110(q1, q1_n, vss, vdd);
}

network hotel(terminal s1, s2, s3, freeze, light, ck, reset, vss, vdd)
{
    {inst1} no310(s1, s2, s3, s_n, vss, vdd);
    {inst2} iv110(s_n, s, vss, vdd);
    {inst3} hotelLogic(s, freeze, q0, q1, q0new, q1new, vss, vdd);
    {inst4} dfr11(q0new, q0, reset, , ck, vss, vdd);
    {inst5} dfr11(q1new, q1, reset, , ck, vss, vdd);
    net {q1, light};
}

```

Figure 3.7: The sls circuit descriptions of 'hotel' and 'hotelLogic'.

At this point, you may want to verify that the layout cell *hotelLogic* really implements the corresponding circuit. This is not strictly necessary, because the OCEAN tools generate layout that is “correct by construction”. See subsection 3.3.5 for details on how to verify a layout cell.

For the moment, we’ll just go on and use the layout cell *hotelLogic* to implement the top level circuit *hotel*. Making the layout of *hotel* goes in the same way as we just made its child cell *hotelLogic*:

- Go to the **instances** menu and click **madonna**. Enter as cell name *hotel* and click **DO IT** to put *madonna* to work for you. Verify that *madonna* indeed placed the 5 instances of *hotel*, including *hotelLogic*.
- Now route this placement of *hotel*. First go to the **boxes** menu, then click **trout**, followed by **DO IT**.
- Verify the result of the routing by clicking **check nets**. This button compares the current layout with the circuit description. It reports any unconnects or short-circuits.
- If no errors are reported, write the layout under the name *hotel* to the database.

3.3.5 Verification

If everything went well then now the time has come to verify that the layout has the desired functionality. Although we say “verification” we actually mean “simulation”. In

order to simulate the behavior of the layout we need 2 items.

1. *The extracted circuit.* This is the circuit that the tool *space* creates by scanning the actual layout. Depending on the options passed to *space*, the extracted circuit may or may not contain parasitic resistors and capacitances.
2. *A stimuli file.* This file defines how the input signals for the extracted circuit change as a function of time. It also defines which output signals of the extracted circuit must be observed.

The tool *simeye* takes these 2 items, calls the simulator and displays the simulation results. Figure 3.8 shows this verification flow.

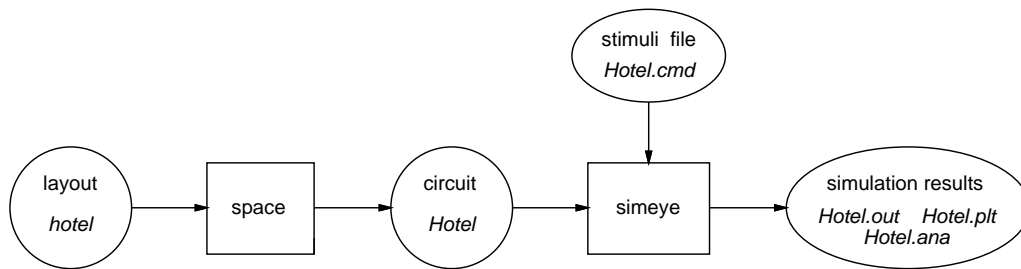


Figure 3.8: The verification flow

Note that after extraction we have **two** circuit representations of *hotel* in the database! The first *hotel* was created by *csls* as described in Subsection 3.3.3 and the second *Hotel* (with the first letter a capital) is created by *space*. We can tell them apart by looking at the first character of the circuit name: extracted cells start with a capital.

3.3.6 Extracting the circuit

Now type the following command to extract the circuit *Hotel* from the layout *hotel*:

```
/user/hillary/hotel % space -c hotel
```

The option `-c` specifies that *space* must also extract capacitors. You can have a look at the extracted circuit by typing

```
/user/hillary/hotel % xsls Hotel | more
```

Watch that capital H. Now purify this circuit description by removing all unconnected transistors:

```
/user/hillary/hotel % ghoti Hotel
```

Finally, start the simulator by typing:

```
/user/hillary/hotel % simeye Hotel &
```

Click the run button to start the simulation. Within a few seconds the simulation results are displayed in the *simeye* window.

3.3.7 Simulating at various levels of abstraction

Simeye allows you to simulate a circuit at several levels of abstraction. The most abstract level is obtained by clicking before clicking the button. This calls a logic simulator that models a transistor as an ideal switch. A slightly less abstract level of simulation is obtained by clicking . This models a transistor as a switch with a linear resistance and linear capacitances and it recognizes the parasitic capacitances of the wires. Finally, the most detailed simulation level is obtained by clicking . This calls the Berkeley circuit simulator *spice*.

The small buttons labeled and are toggles that either cause an “analogue” or a “digital” interpretation of the signals to be displayed. Spice only allows an analogue interpretation of course.

3.3.8 Graphically editing the stimuli file

You may want to experiment with different stimuli for *Hotel*. You can easily edit the stimuli with the graphical editor that is part of *simeye*. To do this, click the button in *simeye*. This brings you in a graphical editor. The five most important buttons of this editor are

- define a new input signal for the circuit.
- edit an existing input signal.
- change the frequency of the input signals
- change the moment that the simulation stops
- exit the editor, back to the simulator

3.4 The third example: using logic synthesis to make ‘hotel’

To demonstrate the use of the *sis*⁵ logic synthesis program we included another tutorial directory:

```
/user/hillary % tutorial sis
```

Essentially it contains the same circuit as the previous ‘hotel’ tutorial. In this case, however, we will also generate the circuit description automatically. The main difference is that *sis* generates the circuit *hotel* in one step without hierarchy.

The program *kissis* is performing this entire FSM synthesis trajectory for you. The format of the input table for *sis* is simple. It basically declares the number inputs and outputs, and state transition conditions. For example, the *kiss2* file in figure 3.9 describes a simple hotel switch with 4 states. Each line in this table corresponds to a state transition (“an

⁵You can run this tutorial with or without having the *sis* logic synthesis package on your machine. In the latter case you can just use the pre-synthesized file ‘hotel.sls’. We cannot distribute the *sis* package with the *OCEAN* tools, unfortunately. You can, however, retrieve *sis* yourself using anonymous ftp (host ic.berkeley.edu, directory pub/sis). *sis* runs on many hardware platforms.

```

.i 2      /* 2 inputs: switch (1st column) and freeze (2nd) */
.o 1      /* 1 output: light (last column) */
.r off1   /* reset to state off1 */
-1 off1 off1 0
10 off1 off1 0
00 off1 off0 0
-1 off0 off0 0
00 off0 off0 0
10 off0 on1 0
-1 on1 on1 1
10 on1 on1 1
00 on1 on0 1
-1 on0 on0 1
00 on0 on0 1
10 on0 off1 1

```

Figure 3.9: A file in `kiss2`-format, describing the hotel circuit.

arrow”) in the state diagram of Figure 3.4, page 24. Each line in the table contains four fields: the input signals, the current state, the next state and the output signal.

A file which contains this state transition table is present in the ‘sis’ tutorial under the name ‘hotel.kiss2’. With this tutorial you can run the logic synthesis tools yourself⁶:

```

/user/hillary/sis % kissis hotel.kiss2

```

The output of *kissis* is a circuit description, which was mapped on the ‘fishbone’ Sea-of-Gates library. The file `hotel.sls` which was created by *kissis* looks as in figure 3.10 if you look at it using *xsls hotel*. SIS just numbers the inputs and outputs:

IN_0 is the switch button input *s*.

IN_1 is the *freeze* signal which inhibits the input of *s*.

OUT_0 the output of the switch: *light*.

R is the reset input for the flip-flops.

CK is the clock input.

To avoid problems with undefined flipflop states in *sls*, the program *kissis* instructs SIS to use the resettable flip-flop ‘dfr11’. The reset-signals of the flip-flops are automatically connected to the terminal ‘R’ of hotel. In this way you can force the FSM to a known state at the beginning of the simulation.

Now simulate the hotel switch using *simeye*. An appropriate command file is present in the tutorial directory. Click run to start the simulation. Notice that there is also a line called ‘State’ in the output, which indicates the symbolic state of the Finite State Machine. Click on value and move to the ‘State’-bar to see the state as a function of the time.

When you finished simulating, generate the layout using *seadali* with *madonna* and *trout*:

- Go to the instances menu and click madonna. Enter as cell name *hotel* and click DO IT to put *madonna* to work for you. Verify that *madonna* indeed placed the 5 instances of *hotel*, including *hotelLogic*.

⁶If you do not have the sis package installed, just type “`csls hotel.sls`” instead to use a pre-synthesized version of hotel.

```

/* This file was automatically created by blif2sls.
   Thu Jun 24 09:57:52 1993 */
#include<oplib.ext>

/*      Code Assignment: off1  00          */
/*      Code Assignment: off0  10          */
/*      Code Assignment: on1   11          */
/*      Code Assignment: on0   01          */

network hotel( terminal IN_0, IN_1, OUT_0, R, CK, vss, vdd)
{
  {inst0}    dfr11( n_2355_, R, CK, LatchOut_v2, vss, vdd);
  {inst1}    dfr11( OUT_0, R, CK, LatchOut_v3, vss, vdd);
  {inst2}    iv110( LatchOut_v3, n_2314_, vss, vdd);
  {inst3}    iv110( IN_0, n_2315_, vss, vdd);
  {inst4}    no210( IN_0, IN_1, n_18_, vss, vdd);
  {inst5}    mu111( LatchOut_v2, n_2314_, n_18_, n_2355_, vss, vdd);
  {inst6}    no210( IN_1, n_2315_, n_17_, vss, vdd);
  {inst7}    mu111( LatchOut_v3, LatchOut_v2, n_17_, OUT_0, vss, vdd);
}

```

Figure 3.10: The output of the logic synthesis program SIS for the hotel circuit.

- Now route this placement of *hotel*. First go to the **boxes** menu, then click **trout**, followed by **DO IT**.
- Verify the result of the routing by clicking **check nets**⁷. This button compares the current layout with the circuit description. It reports any unconnects or short-circuits.
- If no errors are reported, write the layout under the name *hotel* to the database.
- A quick way of doing all of the above in one step is by going to the **database** menu and clicking **Place and Route**.

Now you can extract the circuit and simulate it again. This works in exactly the same way as in the previous section. Please proceed to section 3.3.6.

As second example the circuit 'three' is included, which performs essentially the same as 'hotel', but has 3 button inputs instead of one.

3.5 Some advanced features

In this section we demonstrate some advanced features of the OCEAN system using the hotel circuit which you've just made.

3.5.1 Playing around with *Madonna*

Madonna doesn't like to be fooled around with too much. But let's try to see how far we can go using the circuit *hotelLogic*. For full details on the options you can have a look in chapter 7 on page 53.

⁷It is not absolutely necessary to do this. *trout* automatically performs this connectivity check after routing.

First read the layout of the cell *hotelLogic* into *seadali* (the one which you've made a while ago). Then go to the instances menu and click **Madonna**. Tell *madonna* that you are sure to do that by clicking **yes** and just press [return] at the cell name to select *hotelLogic*.

Now let's ask *madonna* to place the cell in a larger area: use **zoom Out** to zoom out a bit. Then press **set box** and drag a box over a larger area and click **DO IT**. *Madonna* will spread out the modules over the specified area. If you make the area very large, you will notice that it takes *trout* longer to route it.

Obviously, we can also do the opposite. If you specify a very small box *Madonna* will squeeze the cells into the minimum area. If the area of the box is too small, she will expand into the horizontal direction. If you press **y-expand**, however, the layout will grow taller because she expands in the vertical direction.

3.5.2 Using *trout* in various ways

Partial manual and automatic routing

The router *trout* is a very powerful tool. Let's try to use *trout* to complete a partially routed circuit. Read a routed layout of *hotelLogic* in and go to the **boxes**-menu. Enable the masks **in**, **ins**, **cps** and **cos** in the bottom of the screen. Now use **DELETE** to erase some part of the existing routing. Notice that the layout of the son-cells cannot be erased. Now click **trout** and just **DO IT** to cause *trout* to 'repair the damage'. *Trout* automatically discovers which nets are already connected and which existing wires can be used to make connections which are as short as possible. In this way you can manually pre-route certain wires and let *trout* do the rest. Just play around a bit with this feature. Do not press **erase wires** in this case because that will remove all existing wires prior to routing.

Verifying a layout

If you have created a (partially) manual layout, it is interesting to check whether it is correct or not. This can be done simply by pressing **check nets** in the **boxes**-menu. If no window appears, everything is OK. Just for fun, make some deliberate errors in the (routed) layout of *hotelLogic*. Cut one wire, and make a short circuit between two others (use **DELETE** and **APPEND**). If you press **check nets** again, a window will warn you for the problems. The unconnects and shorts are also indicated in the layout. The indicator arrow shows the terminals of the nets which are involved in the short-circuit or the unconnect. The shorts are also indicated by the white pattern in the **bb**-mask. Unfortunately it is impossible to show the exact location of the short.

Automatic capacitors

Now let's go for the really wild stuff!! *trout* has the capability to convert all unused transistors into capacitors which are switched between power and ground. In this way, the noise on the power lines can be reduced effectively. The best way to try this is on a rather empty layout, for instance the one which you made using *madonna* in a big box. Now press **trout** and **Option menu** and select **capacitors**. Then click **- return -** and **DO IT** and watch what happens! It should look like figure 3.11. Notice that the power wires are made as fat as possible, and that the layout contains many additional

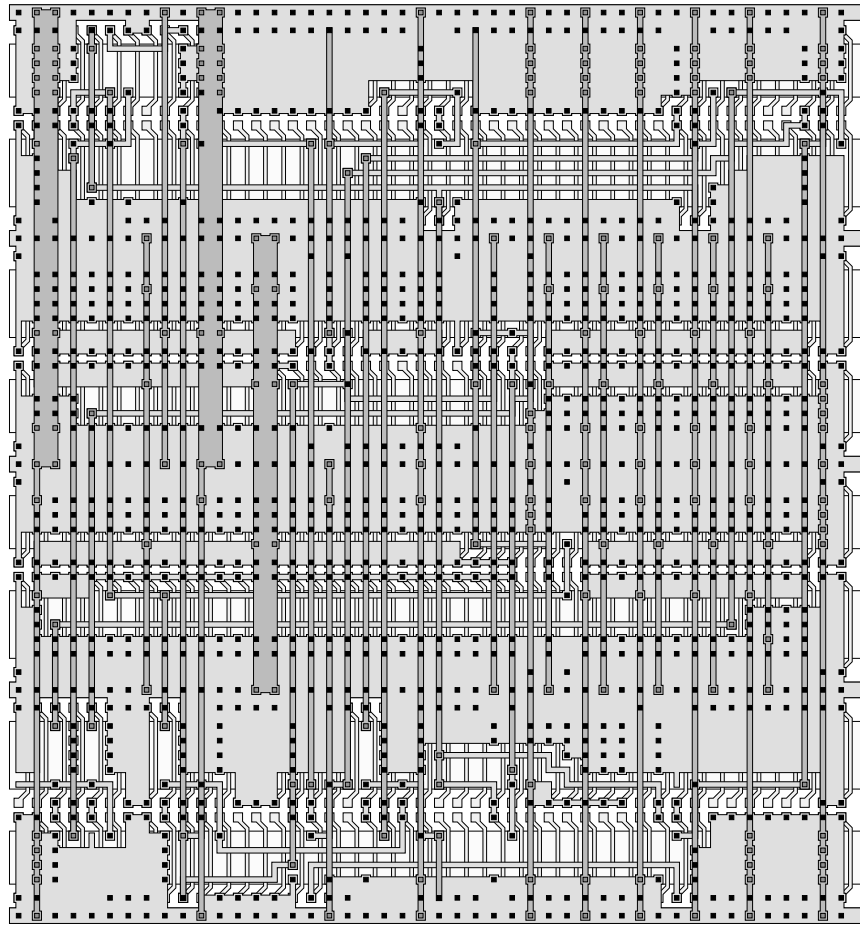


Figure 3.11: The same circuit as figure 2.1 (page 8), but routed with the option `capacitors`. This causes *trout* to connect all unused transistors and, if possible, to use them as capacitors. In this way the noise on the power lines is reduced considerably.

vias and wires to 'nail' the power lines properly together. Notice also that there is now hardly any way left to route additional wires through this module.

3.5.3 Inserting your own nor in the circuit

If you finished the example of section 3.2 successfully, you may want to edit the files `hotel.sls` and `hotelLogic.sls`, replacing the calls to the library cell `no310` by calls to your own `nor3`. You'll have to use the `extern` declaration of the SLS language to declare the order of the terminals. For instance, insert the line “extern network nor3 (terminal A, B, C, F, vss, vdd)” on top of the files `hotel.sls` and `hotelLogic.sls`. See the SLS manual for more details.

3.5.4 Making the hotel switch on a different image

In this tutorial we have been using the *fishbone* image. You can run the same tutorial with the other two images. To make a project in the *octagon* image, type:


```
/user/hillary % setenv OCEANPROCESS octagon
/user/hillary % tutorial hotel
```

similarly, you can use the *gatearray* image: type:

```
/user/hillary % setenv OCEANPROCESS gatearray
/user/hillary % tutorial hotel
```

The environment variable *OCEANPROCESS* must be set to the image you're working in. Also, make sure that no directory *hotel* already exists in the directory. In the new *hotel* directory you are able to run the same *hotel* tutorial. Only the *spice* simulations will most likely not work properly. Figures 2.1, 2.2 and 2.3 (page 8) show the layout of *hotelLogic* in the *fishbone*, *octagon* and *gatearray* image.

For now, let's just use the *octagon* image and try to make the layout of cell *hotelLogic*:

```
/user/hillary/hotel % csls hotelLogic.sls
/user/hillary/hotel % seadali &
```

Just click instances followed by madonna to place the circuit. Have a good look at the image, and try to move some instances. Notice that *seadali* is automatically mirroring the instances in the proper way. Notice also that *madonna* is clever enough to overlap some gates in the same 'quarter'.

Next, route it using trout. This is a tree-layer process, which makes it sometimes hard to interpret the layout. Use visible to switch off some masks.

3.5.5 Plotting your layout

It is easy to make a plot of your layout. The program *getepslay* converts the layout into a postscript file, which you can directly send to the laser printer. For example:

```
/user/hillary/myproject % getepslay hotelLogic
```

will create the postscript file *hotelLogic.eps*. See the manual page of *getepslay* for more details.

3.5.6 Removing cells and projects

To remove an entire project you can just use the normal UNIX command. For example:

```
/user/hillary % rm -rf hotel
```

removes the entire project in subdirectory *hotel*. A less rigorous way is to remove individual cells using *rmdb*:

```
/user/hillary/hotel % rmdb -c hotel layout
```

removes the layout of cell *hotel* from the database. You must be in a project directory to call this command.

Chapter 4

The programs in the OCEAN system: a glossary

All tools that are available from the OCEAN distribution are listed below, with a one-liner briefly describing each tool. The most important programs are indicated in figure 4.1, and are also described in more detail in the following sections. Some programs document themselves briefly if you pass them the '-h' option. Other tools have manual pages (use *icdman*).

4.1 Programs to generate netlists

<i>kissis</i>	Run SIS logic synthesis for automatic FSM generation.
<i>ghoti</i>	purifies sea-of-gates extracted circuits.
<i>makebus</i>	analyzes a netlist to extract buses from it.
<i>space</i>	extracts a circuit from a layout.
<i>vspace</i>	special space extract script.

4.2 Programs to generate layout

<i>fish</i>	purifies a sea-of-gates layout.
<i>getldm</i>	script to get a flat ldm description.
<i>layflat</i>	removes all hierarchy from a layout cell.
<i>madonna</i>	partitioning based sea-of-gates placer.
<i>seadali</i>	interactive tool for layout editing, placing and routing.
<i>trout</i>	Through-the-cell sea-of-gates router.
<i>sea</i>	front-end for calling <i>madonna</i> and/or <i>trout</i> from a nelsis environment.

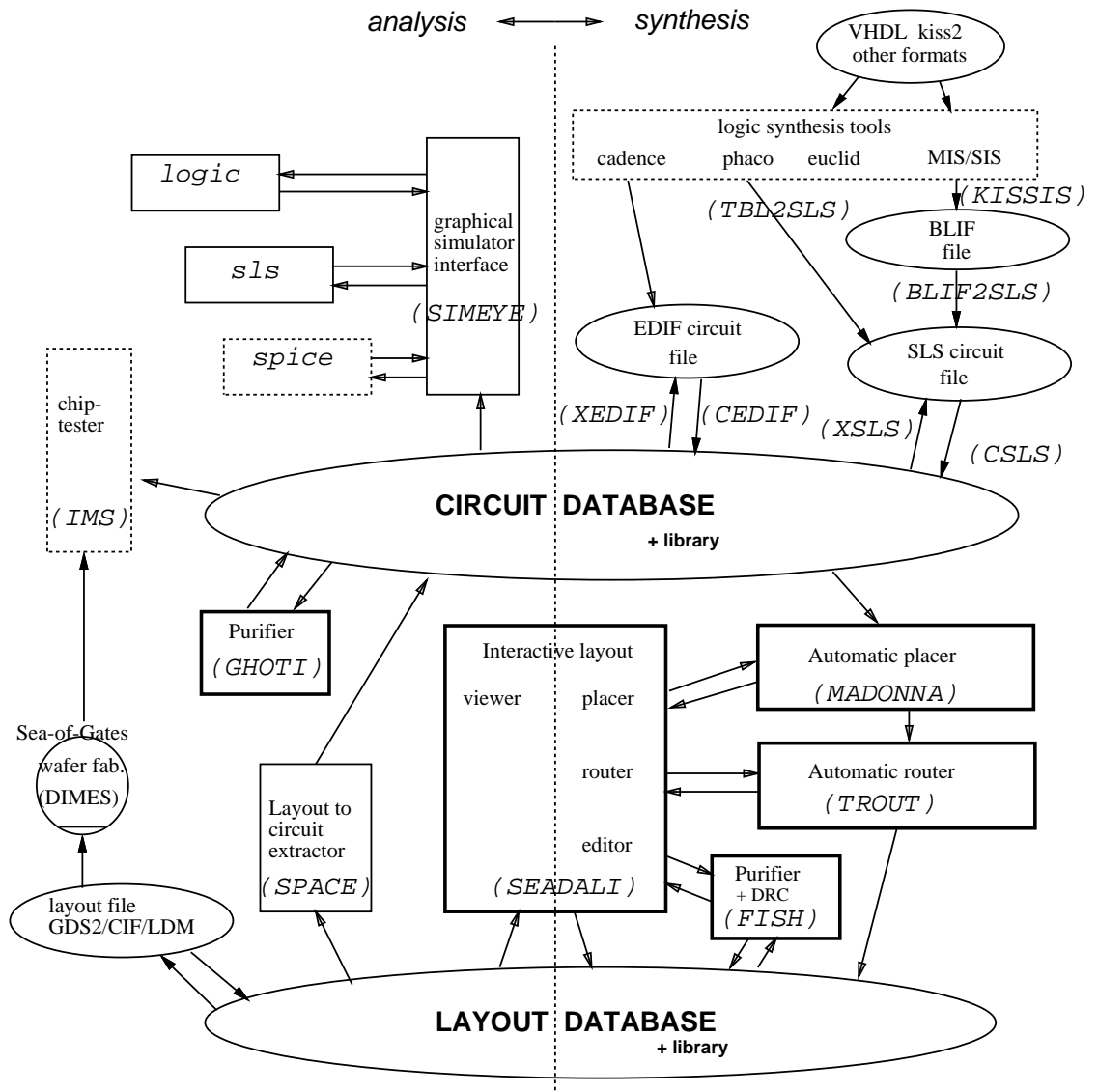


Figure 4.1: The main programs in the OCEAN Sea-of-Gates design system. The arrows indicate the design flow.

4.3 Programs for simulation

<i>arrexp</i>	expands the arrays in an SLS commandfile, used by <i>simeye</i> .
<i>dofunc</i>	front-end for the functional simulator tool <i>cfun</i> .
<i>cfun</i>	creates a functional (“behavioral”) simulator.
<i>nspice_bs</i>	pre-processor for SPICE-2 and SPICE-3, used by <i>simeye</i> .
<i>nspice_pp</i>	post-processor for SPICE-2 and SPICE-3, used by <i>simeye</i> .
<i>nspice</i>	front-end for SPICE-2 and SPICE-3, used by <i>simeye</i> .
<i>simeye</i>	interactive switch-level and SPICE simulator for X-windows.
<i>sls_exp</i>	pre-processor for <i>sls</i> .
<i>sls</i>	switch-level simulator, used by <i>simeye</i> .

4.4 Programs to list and manipulate design projects

<i>dblist</i>	lists the contents of the nelsis database.
<i>freedif</i>	removes cells from the seadif database.
<i>mkopr</i>	creates a new sea-of-gates design project.
<i>mkpr</i>	creates a nelsis project (see <i>mkopr</i>).
<i>mksls</i>	constructs a Makefile for updating a nelsis database.
<i>rmdb</i>	removes cells from the nelsis database.
<i>rmpr</i>	removes a nelsis design project.
<i>rmsdflock</i>	removes (blasts) the locks from a seadif database.
<i>seedif</i>	lists the contents of the seadif database.
<i>tutorial</i>	creates a sea-of-gates design project containing a tutorial.

4.5 Programs to list and manipulate cells

<i>clambda</i>	changes the lambda value of a design project.
<i>colaps</i>	flatten the hierarchy of a seadif circuit cell.
<i>dbcat</i>	lists the contents of cells that are in the nelsis database.
<i>dbclean</i>	removes data from the database that can be regenerated.
<i>exp</i>	expands a layout cell (generate derived data from it).
<i>gnarp</i>	performs various kind of operations on a seadif cell.
<i>impcell</i>	imports a cell from another project (see also <i>addproj</i>).
<i>macro</i>	sets or unsets a macro status for a layout cell.

<i>makeboxh</i>	expands a layout cell hierarchically to boxes (see <i>exp</i>).
<i>makeboxl</i>	expands a layout cell linearly to boxes (see <i>exp</i>).
<i>makegln</i>	creates a non-vertical line segment representation (see <i>exp</i>).
<i>makevln</i>	creates a set of “_vln” data files (see <i>exp</i>).
<i>mplot</i>	makes a plotfile for a masklayout.
<i>putdevmod</i>	puts a device model description into the circuit database.
<i>showbus</i>	shows the buses in a netlist, as found by <i>makebus</i> .

4.6 Programs to convert to and from the database

<i>bliff2sls</i>	Converts BLIFF circuit format into SLS format.
<i>ccif</i>	compiles CIF data format into the nelsis database.
<i>cedif</i>	compiles EDIF sources into the nelsis database.
<i>cga</i>	converts GDS-II data format to ASCII.
<i>cgi</i>	compiles GDS-II sources into the nelsis database.
<i>cig</i>	converts info from the nelsis database to GDS-II format.
<i>cldm</i>	compiles LDM (= layout description language) into the database.
<i>csls</i>	compiles SLS (= circuit description language) into the database.
<i>esea</i>	compiles EDIF sources into the seadif database.
<i>getepslay</i>	converts a layout cell to PostScript suitable for printing.
<i>layout</i>	print a hardcopy of a given layout cell (uses <i>getepslay</i>).
<i>nelsea</i>	nelsis to seadif and seadif to nelsis database converter.
<i>cspice</i>	compiles a SPICE network description into the database.
<i>xcif</i>	extracts CIF format from the layout database.
<i>xcmk</i>	extracts CIRCUITMASK format from the layout database.
<i>xedif</i>	extracts EDIF format from the circuit database, see <i>cedif</i> .
<i>xldm</i>	extracts LDM format from the layout database, see <i>cldm</i> .
<i>xsls</i>	extract SLS format from the circuit database, see <i>csls</i> .
<i>xspice</i>	extracts SPICE format from the circuit database, see <i>cspice</i> .

4.7 Miscellaneous programs

<i>ICELLS</i>	compiles and installs sea-of-gates cell libraries.
<i>seatail</i>	temporarily shows output from a sea-of-gates tool.
<i>sea</i>	front-end for calling seadif tools in a nelsis environment.
<i>setcmap</i>	manipulates the X-window colormap for nelsis and OCEAN tools.
<i>tecc</i>	technology compiler for <i>space</i> .

Chapter 5

'Seadali': your gateway to sea-of-gates layout

This chapter deals with *seadali*, the interactive tool at the very heart of the OCEAN layout system. It is worthwhile to read this chapter carefully, since it contains many useful hints on how to click the buttons efficiently. *Seadali* is an interactive program which allows you to:

- View or modify any existing layout cell.
- Create an empty layout cell from scratch with the help of *fish*.
- Place instances in your layout manually or automatically (*madonna*).
- Automatically route your layout using *trout*.
- Verify the correctness of your layout against the design rules and the circuit description.

There are many buttons and options to control this program. But don't worry: you'll get accustomed to them quite soon. We try to describe only the most vital buttons and functions, so you can get started quickly. All trivial or less useful buttons are not described.

In all examples of this, and the following chapters we assume the *fishbone* image. Some mask names, design constraints and commands work slightly different in the other images.

5.1 Setting up the environment: creating a project using 'mkopr', 'mkepr' or 'mkpr'

Before we can start, we must create the necessary environment to run the design system. In our system the design information is concentrated in *projects*, which is just a UNIX directory with a database structure. You can make an empty project directory by typing:

```
/user/hillary % mkopr myproject
/user/hillary % cd myproject
```

Obviously, 'myproject' can be any name you like.

As alternative of *mkopr*, there is *mkepr* (notice the 'e') which imports the same default libraries as *mkopr*, but also additional cells in the extended (or expert) library 'exlib'.

The more simple *mkpr* creates a project without any imported libraries. Many OCEAN tools cannot deal with such a bare project since they need at least a library called **primitives** describing the primitive electronic components on the master image.

The contents of other projects can be 'imported' in your project using the commands *addproj* followed by *impcell*. For instance:

```
/user/hillary/myproject % addproj /user/chelsea/counter
/user/hillary/myproject % impcell -l -a /user/chelsea/counter
```

imports all layout cells from the project */user/chelsea/counter* into *myproject*. In the same way library cells are imported into the current project. If you use *mkopr* or *tutorial*, the default library is automatically imported into your new project.

To switch between the different processes and design rules, you can set the environment variable **OCEANPROCESS** before you make the project:

```
/user/hillary % setenv OCEANPROCESS octagon
/user/hillary % mkopr myoctagonproject
```

creates a project in the *octagon* process which includes the standard library for that image. Similarly, you can set **OCEANPROCESS** to *gatearray*. If **OCEANPROCESS** is not set, the *fishbone* image is assumed. Always make sure that **OCEANPROCESS** is set to the image you are currently working with, because the OCEAN tools need it to find their technology files.

All design tools can only run in a project directory! Any tool in OCEAN or NEL-SIS (except *mkpr* and *mkopr*) will fail if you try to start it in an arbitrary directory. Unfortunately some of them do not print clear error messages (something like 'cannot find .dmrc').

5.2 Starting 'seadali'

The best way to start *seadali* is by typing:

```
/user/hillary/myproject % seadali &
```

The '&' instructs UNIX to start the program in the background. Therefore the prompt returns immediately. After starting a window pops up¹, which shows the main menu on the left. From this menu you can select various sub-menus by clicking the appropriate box. The bottom-most command of any menu always brings you back to the main menu. *Seadali* is also sensitive to keyboard input. Table 5.1 (page 49) shows all commands.

¹If starting *seadali* doesn't work and no window appears, remember that you must be in a project directory. You can make a project directory using *mkopr*. Also, check whether your **DISPLAY** environment variable is set to the proper screen (the name of your terminal, followed by ':0').

5.3 Getting on-line help in 'seadali'

In the main menu you'll find the button help. This will start a hypertext on-line help page on ocean. You need the program *xmosaic* to for this help facility. Initially it will display the hypertextpage `ocean/lib/html/help.html`.

5.4 Reading and writing layout: database

Pressing database brings you in a special menu for reading and writing cells from/into the database. Let's have a look at the buttons:

Read Load a cell from the database into the editor. The list of all cells in your database will be displayed. If this list is empty, your database has no cells. Notice that it is not possible to read or edit imported cells from libraries. You can only have a look at them by loading them as instances in the menu instances (see 5.6 on page 42). It is possible to start *seadali* with a cell name as argument. For instance:

```
/user/hillary/myproject % seadali adder &
```

causes the cell adder to be read automatically at startup.

Expand all Set the hierarchical level (that is, the amount of detail) you wish to see of the cell. Initially all son-cells are expanded 'to the bottom'. Alternatively, you can just use the keys 1-9 on your keyboard (see table 5.1 on page 49). Initially all son-cells are expanded 'to the bottom'. You can set this default expansion level for all cells in the `.dalirc`-file (see 5.10.1 on page 47). Initially all son-cells are expanded 'to the bottom'.

Expand instance Set expansion level of a specific instance. This can be useful to prevent long drawing times on large designs.

show Sub terminals Show the terminals of the sub-cells. The key 's' on your keyboard is doing the same (see table 5.1 on page 49).

New Erase the current cell in *seadali*'s memory to start with a new cell. This only clears the memory of *seadali*. It doesn't remove the cell from the database.

Fish If the design is empty (initially, or after pressing NEW) this button creates a 20×1 array of the fishbone image. This is useful to start a new cell. If the design is not empty, this button will purify your layout (see chapter 6 on page 51).

Fish -i This button performs the same as Fish, except that the image is removed from the cell. Also, the cell is automatically moved as close as possible to the origin. This button can be useful if you want to use the cell as a child cell on a higher level of hierarchy.

Write Write the cell into the database. If necessary, you can write it under a new name.

The title bar of *seadali*'s window shows the name of the cell which you are editing now, together with the expansion level.

You can interrupt seadali while it is drawing! Just hit any key to stop the drawing and regain control. This can be especially useful for extremely large layouts and/or

nervous designers. Hit the key `r` or `next` (on the keyboard) to redraw the screen, in case you want to complete an interrupted drawing. During input activities (reading, expansion) *seadali* cannot be interrupted.

5.5 Running other programs: the `automatic tools` menu

This menu contains all automatic tools which can be run from *seadali*. Most of them are dealt with in other sections. In addition, this menu contains the `print hardcopy`-button, which prints the layout which is currently being edited. See subsection 5.10.5 (page 48) for more details.

5.6 Importing cells from a library: the `instances` menu

Hierarchy is the most important way to handle the complexity of any circuit. In the `instances` menu you can import other cells into you design as 'son cells' (also called 'instances'). The son cells can be imported from a remote library, which is nothing else than just another project. Obviously you can also add son-cells which are in the current project. *Seadali* displays the names of the instances in your design. It also displays the cell name of each instance between '@' characters. The following buttons are important to play around with instances:

`Madonna` Automatically place the instances of a circuit. See chapter 7 on page 53 for all the naked details about Madonna.

`ADD instance` Add a local instance to your design. After you have selected the instance you must click its left bottom position. It is not necessary to align the instances exactly because *seadali* will automatically snap them to the grid (if the 'image mode' is on). *Seadali* even performs automatic mirroring of small instances, so that they will always match the grid. Also *fish* can take care of the snapping.

`ADD imported instance` Just like the previous command, but this button adds an imported instance to your design. Use this button to add library cells. *Seadali* shows a list of all imported (library) cells. You can add libraries to your project using *mkopr* (see section 5.1).

`set instance name` Set the name of a specific instance. Notice that an instance which you add by hand using `ADD` has no name yet. This button can be useful to help the automatic router select the correct instance.

`DELETE instance` Does just what you expect it to do: delete a specific instance.

`move` , `mirror` and `rotate` Use these buttons to put the cell on its right place. In 'image mode', *seadali* will shift and/or mirror the cell automatically. On certain places in the fishbone the cells must be mirrored in the x-axis.

`Fish` Run the layout purifier, which will shift the instances to a proper grid position. See chapter 6 on page 51 for more details.

5.7 Editing mask layout: the boxes menu

5.7.1 Selecting active masks

To add a wire or any other mask pattern in *seadali* you first have to activate a mask. The bottom of the window shows the list of available masks. In the sea-of-gates process you must press in to activate the first metal layer and ins to activate the second metal layer. For your convenience the **ins**-mask is automatically activated when *seadali* is started.

Warning: Only activate the **in**, **ins**, **con**, **cop**, **cps** or **cos** masks. On our sea-of-gates chip it is not possible to add or remove patterns in any of the other masks. If you do so, *fish* will complain and remove them automatically.

5.7.2 Making wires using APPEND

To add an arbitrary box in the active layer(s), just press APPEND and click at the appropriate positions in the layout. You can change the active masks also while this command is activated. If image mode is activated, the boxes you append are automatically snapped on the grid. Using the keyboard keys you can zoom or pan the window using the keyboard keys (see table 5.1 on page 49).

5.7.3 Making wires with wire

Long, continuous wires can be added more easily with the wire button. Just enter the corners of the wire. Click next to start a new wire or -ready- to return to the boxes menu.

Huh?! I am adding a wire but nothing is happening! The answer is simple: you forgot to activate the layer, or you also activated another more dominant layer (such as **cps** or other contacts). Also it is possible that you are adding layout over an existing contact. Since the contacts are displayed dominant, they hide other layout which covers them. Finally it is possible that you accidentally made a specific layer invisible in the visible menu (see section 5.9, page 46).

Huh?! Some wires are drawn in a lighter color You cannot edit the layout of con-cells. To indicate the difference between the different hierarchical levels, the layout of son-cells is drawn in a 'hashed' style. See section 5.10.3 on page 47.

5.7.4 Ensuring design-rule correctness

For Sea-of-Gates layout it is useful to select the image mode in the settings menu or the the *.dalirc*-file. If this mode is enabled, the boxes you append are automatically snapped to the grid of the image. Also, the wires have the proper width, and the vias are aligned precisely. This behavior is disabled by switching off the image mode. See section 5.10.2 for more details.

Despite the features of image mode, the layout purifier *fish* (see chapter 6 on page 51) should still be used before writing anything to back to the database. *Fish* performs more elaborate design-rule checks on your layout.

5.7.5 How to make contacts between layers?

In the *fishbone* sea-of-gates image there are basically only two types of contacts:

1. Between the image (**ps** or **od**) and metal 1 (**in**): **con**, **cop** or **cps**
2. Between metal 1 and metal 2 (**ins**): **cos**.

To add a contact, use the **APPEND** command to make a small box at the desired position. For a contact between the image and metal 1, you must activate the mask **con**, **cop** or **cps**. Which of these three you use doesn't matter because *fish* takes care of that. For a contact between metal 1 and metal 2 you must use the **cos**-mask.

Press **fish** to convert the contact mask into a design rule correct pattern.

Warning: There are some restrictions in the placement of the contacts. They may not be stacked and they are not allowed in certain rows. See section 2.4 on page 13 for more details.

In the *octagon* image there are three layers and therefore masks for contacts:

1. Between the image (**ps**, **nn** or **pp**) and metal 1 (**in**): **co**
2. Between metal 1 and metal 2 (**ins**): **cos**.
3. Between metal 2 and metal 3 (**int**): **cot**.

In the single-layer *gatearray* image there is only one contact.

5.7.6 Deleting layout: **DELETE**

Deleting is simple. Just activate the appropriate layers and click the corners of the box to be deleted. To delete all the wires of you current design, activate all masks and drag the box over your entire design.

Huh?! I try to delete but it won't go away! As usual, you are doing something wrong. First of all, check if you have selected the right active layers. Secondly, remember that it is NOT possible to edit the mask patterns of the son cells (instances). If you want to change that you must edit that particular son cell. It is convenient to use the hashed drawing style for instances to see which wires belong to the son-cells (see section 5.10.3 on page 47).

Can I undo a deletion or any other command? No, unfortunately you cannot undo any action. Therefore you must be careful. You should also be careful with the masks you activate. Adding a big box in the wrong layer can destroy hours of work. Save your layout regularly to avoid frustration!

5.7.7 Copying layout: **yank** and **put**

Sometimes it is convenient to copy or to move a part of the mask. Just like **DELETE**, **yank** puts the indicated box in a paste buffer. The difference is that **yank** doesn't remove the box. You can copy the contents of the paste buffer to a certain position by pressing **put**.

Huh?! It doesn't seem to work! Remember that it is not possible to yank the layout of son cells. You can only yank and put the layout of the (father) cell on which you are currently working.

5.7.8 Purifying and checking your layout: fish

Pressing fish makes your manual design design rule correct. All wire segments will be shifted on the grid, and the instances will be properly aligned. *Fish* also shifts sufficient repetitions of the fishbone image under your design. A more elaborate description of *fish* will be given in chapter 6 on page 51.

5.7.9 Automatically routing your design: trout

This button will call the automatic router called *trout*. This is a rather complicated command. Pressing it without the proper preparations will most likely result in an error message. See chapter 8 for more details on how to use the router.

5.7.10 Verifying layout connectivity: Check nets

Pressing this button will verify your layout by comparing it with the net list in the circuit description. Your layout will be 'fished' automatically if you press this button. The check which is hidden behind check nets is quite strong. The following errors will be detected:

- All errors which *fish* detects. In this case the errors of *fish* will be logged in the file `seadif/sea.out`. In contrast to the button fish, no window with errors will pop up.
- Missing or superfluous instances in your layout.
- Unconnected terminals of nets. The unconnects will be indicated in the layout by a small box with the name of the net and an arrow. The indicator will be placed on top of each terminal that has no connection to another terminal of the same net.
- Short-circuits between different nets. The program reports the names of the nets which form a short-circuit. These nets will also be indicated in the layout in the familiar way: `SHORT_no` (`_no` is the number of the short circuit). Notice that the indicators do not indicate the exact spot of the short circuit. The actual short is somewhere on the path between two indicators of the same short.

Huh?! How do I find the indicators in this huge layout? In some cases the scale of the design is so large that the individual indicators are hard to distinguish. One of the ways to find an indicator is by zooming in and moving the window around. For small layouts this is the easiest way. For really large designs, however, this is a hell of a job, a bit like finding a needle in a haystack. The following trick will help you to find that single stacked contact (or unconnect or whatever is indicated). Go to the main menu and press visible (see also section 5.9). Make everything invisible except instances. The indicators will now be visible as little white boxes (or dots). In this way you'll know where to zoom.

5.8 Adding terminals: the `terminals` menu

For simulation it is required to indicate the positions of the terminals in the layout. If you routed the layout automatically using *trout* the terminals will also be placed automatically. In the menu `terminals` you can modify the terminal placement or make one manually.

5.8.1 How to add a terminal?

Again, you must set the active layer of a terminal by clicking the appropriate mask in the bottom row of the window. A terminal can only be placed in the **in** and **ins** layer. Next press `ADD terminal` and drag a small rectangular box on the grid position of the terminal. Next, *seadali* will ask you to give a name to this terminal. Enter it using the keyboard. The name must be unique. Just give it the same name as in the circuit description you made.

5.8.2 How can I get rid of a terminal?

That is very simple: use `DELETE terminal` and point at the terminal you wish to delete. You can also delete all terminals within a specific area using `DELETE terms in box`. Don't forget to set the proper active layer.

5.9 Setting the visible layers: `visible`

In many cases it is not so useful to display all the masks of a cell. Especially on our sea-of-gates many masks are not so relevant for the electrical design, and will only make the picture unclear. In the `visible` menu you can switch on or of any mask you want. The default settings of *seadali* are such that the **ps**, **in** and **ins** plus the contact masks are visible. All the other masks are switched of. You can change the default settings by editing the `.dalirc` file in your project directory (see 5.10.1).

How can I make the grid of the Sea-of-Gates image visible?? You can do this just by clicking `show grid` or by pressing `[g]` on the keyboard. The dots indicate the grid positions (provided that 'image mode' is on). The grid will not be drawn if there are too many grid points in sight.

When does *seadali* display the image? *Seadali* will not draw the image if the scale of the layout is too big. In this way long drawing times are prevented. If you zoom in enough the image will become visible again. You can set the maximum amount of image elements which should be drawn in the file `.dalirc` (see section 5.10.1). You want to see the image at all times, switch of the "image mode" in the settings menu (see subsection 5.10.2).

5.10 Some other nice features: the `settings` menu

There are a few ways to customize the behavior of *seadali* according to your own wishes. In this section we'll list a few of them.

5.10.1 Customizing *seadali*: the `.dalirc`-file

Your project directory automatically contains a file called `.dalirc`. This file is read by *seadali* during startup. You can edit this file to change various settings. Just have a look at it and try change some parameters. Obviously, you must re-start *seadali* for the changes to have any effect.

Huh?! I don't see any file called `.dalirc` in my project! That is normal. In UNIX all files which start with a `'.'` are not visible. You can make them visible by typing:

```
/user/hillary/myproject % ls -a
```

If it is still not there, copy the default file into your project:

```
/user/hillary/myproject % cp $ICDPATH/lib/process/fishbone/.dalirc .
```

5.10.2 Switching on the Sea-of-Gates mode:

Image Mode

Seadali has a special 'image mode' for working with semi-custom layout. This mode should always be turned on if you are working with Sea-of-Gates. The image mode has the following features:

- Layout boxes and instances are snapped to the grid. *Seadali* will warn you if you try to place something at the wrong place. Library cells are automatically mirrored in the x-axis if necessary or possible.
- The position of the tracker is displayed in grid points, rather than in lambda's.
- The grid of the image is displayed if you press

show grid

. The grid is only displayed if it is useful to do so.
- The image is not drawn under your layout if the scale of the picture becomes too big. If the image mode is switched off, the image is drawn always².

This mode should generally be turned on if you are working with Sea-of-Gates. Only in some exceptional cases (e.g. to force the mirroring of a certain instance) you may want to turn it off.

5.10.3 Setting drawing style for instances:

draw hashed

In many cases it is not so clear to which level of hierarchy a certain wire belongs. Therefore *seadali* displays the layout of all son-cells in a less bright color than the wires at the current level. The less bright color is achieved by hashing (45 degree lines). You can switch this feature on or off by pressing

draw hashed

 (also in the `.dalirc`-file).

5.10.4 Setting dominant or transparent drawing style:

draw dominant

There is another drawing mode which can be set by you. In the transparent drawing mode the colors mix. In this way any crossing of the masks results in a mix-color. With many masks this makes the picture rather confusing.

²Provided its present as a instance in your layout.

With the dominant drawing style, the layers are drawn as if they are stacked on top of each other. In this case metal 2 (the top layer) obscures any layout below it. The default drawing mode is dominant. You can switch by pressing the button `draw dominant` or by setting the parameter in the file `.dalirc`. In the same file you can set the order in which the layers should be drawn.

5.10.5 Setting the print command for the `hardcopy` -button

The button `print hardcopy` in the `automatic tools` menu causes the current layout to be printed in the laserprinter. It might be necessary to configure the proper print command to perform this action. The proper command sequence can be set using the keyword `Print_command` in the file `.dalirc`. To set the proper command, it is relevant to understand what *seadali* is doing after you pressed this command:

1. The current layout is written away in the nelsis database using a temporary cell name.
2. Any occurrences of %s in the print command are replaced by the name of the temporary layout cell.
3. The print command sequence is executed by a shell.
4. The temporary cell is removed.

If no print command is set, *seadali* will run the command *playout*. *Playout*, on its turn, will create a postscript file using *geteps*, print it, and show the printer queue.

5.10.6 X-window redrawing strategy: `backing store`

Seadali asks the X-window server (your screen), to maintain a copy of the window in its memory. This feature (called *backing store*) makes it possible to move, obscure, iconify and then re-expose *seadali*'s window without redrawing the entire picture. Especially for large layouts this saves a considerable amount of drawing time and CPU load. The price we have to pay for this nice feature is some extra memory consumption by the server. In general this is not a problem, but for certain Xterminals without memory management the extra (500K to 1M byte) memory load might cause some problems. Therefore this feature can be switched off in the settings menu. You can also switch it off in the file `.dalirc` by adding the line: `"backingstore off"`.

5.10.7 In case you were wondering.....

how and when the unexpected digital pictures appear on the screen, please have a look at the file `ocean/lib/seadali/README`. During the time that *madonna*, *trout* or the verifier are running, you can pop up a picture by clicking one of the menu items. You need the program *xv* for this purpose.

5.11 Interrupting Seadali: keyboard input

Seadali is sensitive for keyboard input. Table 5.1 shows all the keycombinations which can currently be used. They serve as a convenient alternative for clicking with the mouse

key	function
space /any key	interrupt (stop) drawing
'i' or '+'	zoom in by a factor of 2
'o' or '-'	zoom out by a factor of 2
leftarrow or 'h'	pan left
rightarrow or 'l'	pan right
uparrow or 'k'	pan up
downarrow or 'j'	pan down
Select or 'c'	center window around current cursor position
Home or 'b'	set window to bounding box
Prev or 'p'	previous window
Next or 'r'	redraw screen
'1','2','3',...,'9'	set expansion level
'0'	expand maximum
's'	show sub terminals
'd'	toggle hashed drawing style
'D'	toggle dominant drawing style
'g'	toggle display of grid
't'	toggle tracker (cursor position display)


Table 5.1: The key commands for *seadali*.


in the menu. If *seadali* is drawing the picture, pressing *any* key interrupts the drawing³ within one second.

³There is one exception: during the reading or expansion process *seadali* cannot be interrupted.

Chapter 6

'Fish': the layout purifier and design rule checker

In some of the menus of *seadali* you've encountered the button . In this chapter we'll describe a bit more about what is behind this button. *Fish* is a layout purifying program for sea-of-gates layout. It reads a layout, processes it, and produces a new enhanced cell. The basic idea is to allow some design-rule errors during the manual design of a gate-array cell. *Fish* corrects these errors and produces a design rule correct result. In this way the process of manual design using *seadali* can be speeded up considerably. *Fish* also prints warning messages for a variety of design rule errors. You can call fish in two ways:

1. By pressing the button  in *seadali*. This is the most common way of using it. What is actually happening is that *seadali* writes your design as a temporary cell into the database, then it calls *fish*. If everything goes well, *seadali* re-reads the purified cell. This will take a few seconds, depending on the load of the computer.
2. By typing `fish` on the command line:

```
/user/hillary/myproject % fish <cell_name>
```

This purifies the cell named `<cell_name>`. If you want to know more: try `fish -h` for a brief list of all options.

Fish needs an image description file. This file is called `image.seadif` and should be present in the directory `seadif` in your project. It contains data about the positions of grid points, the way in which the image should be repeated, the kind of contact which should be used, etc. If you like you can have a look at it.

6.1 What errors does *fish* detect?

Fish performs limited design rule checking of your layout. It will report errors for:

- Stacked contacts. The process design rules of the fishbone image do not allow to stack an **in** to **ins** contact (= **cos**) on top of another via. The wire would crack at that point. *fish* will indicate the error in the layout by a small box with an arrow pointing to it. *Fish* does NOT detect a pair of stacked contacts if one of the contacts is in a son-cell.

- Contacts at an illegal position. A **cos** contact is not allowed in the rows of the polysilicon gate contacts. The error is again indicated in the layout by a small box with an arrow.
- Patterns (boxes) in illegal masks. *Fish* removes any patterns in **ps**, **od**, etc. because they may not be used on a semi-custom chip.
- Patterns which are not (entirely) in the first quadrant. In our system all layout boxes must have positive coordinates, that is, it must be to the right-top of the origin. The illegal boxes will be removed.

Anyway, you must keep in mind that *fish* is not too clever. It will not warn you of short-circuits and mis-aligned instances (such as forgotten mirroring in the x-axis). For that you must use the button (see section 5.7.10 on page 45). A list of all the errors which were detected by *fish* will be displayed in a pop-up window.

6.2 How does *fish* handle instances?

Fish snaps instances to the nearest grid point. To perform this snapping, *fish* opens the instance and looks for contacts. The first contact it encounters is used for the snapping. Notice that in this way *fish* assumes that the instances are on grid (that is, fished). If it can't find any contacts *fish* gets angry and it doesn't align the instance.

6.3 How to create an empty array of image?

Before editing it is convenient to have an empty array to indicate the transistor and grid positions. This can be done by pressing the button in *seadali* when the design is empty. A 20×1 array will be generated. You can also do the same from the command line by creating a new (empty) cell into the database:

```
/user/hillary/myproject % fish -x 20 -y 1 -o nand2
```

The option **-o nand2** instructs *fish* to write the empty array into a cell called **nand2**. You can now start editing your cell by adding wires.

But what should I do if the 20×1 image doesn't fit anymore? Should the underlying image become too small, simply hit the button to enlarge it automatically so that it fits the wire pattern. Alternatively you can enlarge the basic image by pressing press in the menu. Set or to the new value. Do not touch the dx and dy buttons!

Chapter 7

Automatic placement with 'Madonna'

The job of placing the instances can be performed by an automatic tool which was called after one of the most famous singers of the 1980'ies¹. She places the instances such that the total wire length is (hopefully) minimized. As we all know, Madonna is rather unpretentious; If you like, you can improve her results manually using the other commands in the instance menu. A circuit which was placed using Madonna can be routed automatically, since Madonna makes sure that all instances are present in the layout.

7.1 What is required before I can call 'Madonna'?

Madonna is very sensitive to errors in her input. To prevent offending her the following data must be present:

- A proper circuit description in the database. *Seadali* complains if a circuit description with the specified name doesn't exist. Do not forget to convert your *sls* network description into the database using *csls*.
- For each of the son-cells (instances) in the circuit description a layout must exist which has the same name. We made sure that this is always the case for all library cells.

If there is an error in the net list description Madonna opens a window to explain what you did wrong.

7.2 Running 'Madonna'

Just like *fish*, *madonna* can be called both from *seadali* and the command shell.

¹And 90'ies.

7.2.1 From 'seadali'

Calling her by pressing the button **Madonna** in the instances menu is the most convenient for you. Pressing this button will overwrite the design which you are currently editing in *seadali*. Therefore you will be asked whether you are sure to continue in case the workspace is not empty.

As next step you must enter the name of the circuit which has to be placed. A small menu will appear, in which you just click **DO IT !** to start *Madonna*. In the menu some optional features of can be set for Madonna:

set box Set the box within which Madonna has to place the instances. Click the right-top corner. In this way you can control the shape and the size of the placement. Specifying a big box will leave quite some unused transistors. Specifying a very small box will squeeze the instances on minimum area. If the box is too small, Madonna will expand it in either the x- or the y-direction, depending on what you specified. It is better not to make the box much too small, because Madonna will have a hard time recovering from that.

If the workspace was empty, *seadali* will make an empty image array before you can specify the box. In this way you have some reference for the size of the box.

X-expand Force Madonna to expand horizontally if the box was too small. This is default.

Y-expand Expand vertically if the box was too small.

Channels Create a placement with routing channels. If this feature is active, *madonna* computes the amount of space that *trout* needs to complete the routing process successfully. This computation actually involves a call to *madonna*'s internal global router. A report of the global routing process is written to the file *seadif/groutes*.

Note that the routing channels increase the layout area in both the X and the Y direction, regardless of the preferences indicated by means of the **X/Y-expand** buttons or the **set box** button.

options Set any other (unofficial) option which you want to propagate to *Madonna*. Don't touch this button unless you know what you are doing.

DO IT! Start *Madonna*. This may take anywhere from 30 seconds up to 10 minutes, depending on the number of instances and the system load. You can kill *madonna* using the **KILL** -button.

You can try to place the circuit with various shapes and sizes.

Can I pre-place some instances? No, Madonna is quite possessive. She wants to do everything herself. She will always place all instances in your circuit description. Pressing **Madonna** a second time will overwrite the existing placement. You can only request her to place the instances in a certain box. Obviously, you can modify Madonna's placement manually using the buttons in the instance menu.

7.2.2 Calling 'madonna' from the command line using 'sea'

The non-interactive way to call her is by using the tool *sea* from the command line:

```
/user/hillary/myproject % sea -p name
```

in which *name* is the name of the circuit cell. As a result, the output placement will be written into the layout database. The program *sea* is a kind of bodyguard to *madonna*: it checks the correctness of your input. Only if everything is correct, it calls *madonna* to look at your circuit.

Sea also allows you to place and route a circuit in one step:

```
/user/hillary/myproject % sea hotelLogic
```

for instance, will place and route the cell *hotelLogic* and write the result into the layout database under the name *hotelLogic* (use the '-o name' option to write it under a different name). Just type

```
/user/hillary/myproject % sea -h
```

to see all the options.

Chapter 8

Automatic routing: 'trout'

Making all connections by hand using the editor is a nasty and error-sensitive job. *Trout*, your automatic router, can do this job for you much quicker and without errors. One of the most interesting features of *trout* is that it allows any amount of pre-routed nets and obstacles. For instance, you can route some critical part of your circuit by hand, and then let the router complete the remaining parts. You can also ask the router to make only a part of the design. Many creative combinations of automatic and manual design are possible.

Another special property of *trout* is that it routes *through* the cells, rather than around them. This approach is especially suitable for Sea-of-Gates layout.

8.1 What is required before I can click Trout ?

The router is a very complicated tool, which requires all kinds of data before it can start its job. If any of this data is missing, it will fail miserably. Basically it needs the same *madonna*: a proper net list which specifies the desired connectivity of the modules. But apart from this, the router expects a placement layout, in which all instances are placed. This placement can be made by hand or using *madonna* (or both). It should contain all instances which were mentioned in the net list description. It is not necessary to place the terminals, because the router will do it itself if they are missing. It is also not necessary to give the instances the proper instance names.

You can expect an error- or warning window in the following cases:

- One or more instances are missing in the placement. In this case the circuit will be routed without the terminals on the missing instances. This will most likely lead to an incorrect layout.
- There are unknown instances in the placement, that is, the layout contains instances which do not appear in the circuit description. These instances will be treated as dummies: apart from the power rails they will not be connected. They will only be treated as obstacles. To prevent unknown instances, use the set instance name in the instances menu.
- The placement contains a terminal which is not in the net list.
- If the router fails to route certain nets.

8.2 Running *trout*

8.2.1 Calling *trout* from *seadali*

After pressing **trout**, *seadali* will ask for the name of the circuit. Then you'll see a small menu. Just press **DO IT !** to start the router. After a short while the routed circuit should appear on your screen. Optionally you can set the following parameters in the menu:

no power If selected, the horizontal power rails of the circuit will not be connected. By default all vdd (and vss) rails will be connected to each other by a vertical wires in **ins**. *Trout* will do this even if the power nets are not present in the net list.

erase wires Erase all wires (boxes) and terminals which are present in the layout before routing. By default the router will only add wires to the existing layout.

Border terminals Place all unspecified terminals on the boundary the cell. In this way it is ensured that all terminals can be reached on a higher hierarchical level. This button is enabled by default.

route box only After pressing this button you can drag a box over the layout. *Trout* will only route the nets of which the terminal patterns are in this box. All wires which are generated by the router (except power) will only be inside the box. In this way you can locally re-route a large design. On large empty designs routing in a number of smaller boxes can be considerably faster than routing in one piece.

Notice that the connectivity verifier (which is run automatically after *trout*) is very likely to report errors for unconnected nets because some nets cannot be routed inside the box.

Single-pass routing Prevent the trout from retrying in case the routing was not complete.

Option menu Opens a second menu with additional options for the router. See section 8.7 for more details.

DO IT! Start the router. Depending on the size of the circuit and the load of the system, this will take 5 seconds up to 20 minutes. The routing time increases quadratically with the size of the layout and linearly with the number of nets.

Stopping the router

There are two ways to stop *trout* while it is executing from within *seadali*:

1. By clicking **KILL** in the menu. This is the brute-force way. It will immediately kill *trout* and leave the layout unchanged.
2. By clicking **STOP** in the menu. This will request *trout* to interrupt the routing as soon as possible and return the current layout. This will most likely be an incompletely routed circuit. This button can be useful if routing takes too long, but you are still interested in the (partial) solution.

8.2.2 Running *trout* from the command line

The non-interactive way to call *trout* is similar to the way *madonna* is run:

```
/user/hillary/myproject % sea -r name
```

in which *name* is the name of the circuit cell. It is assumed that a layout cell *name* with all instances already exists in the layout view. See subsection 7.2.2 for more details on *sea*.

8.3 What does *trout* do for me?

Apart from just routing the nets according to the net list description, the router performs a number of other tasks. Most of these features were added just to make the 'default behavior' of *trout* as intuitive and useful as possible. *Trout* performs the following automatically:

Orphan instances The router tries to assign unknown instances ('orphans') to missing instances if they belong to the same cell type. For instance: if an instance 'x' of cell 'nand2' was missing in the placement, but an unknown instance 'y' of cell 'nand2' was found (that is, 'y' is not in the circuit description) then instance 'y' will be renamed to 'x'. Notice that this does not necessarily result in a good placement for the instances. This feature was added for your convenience during manual placement.

Father terminals The terminals of the father cell can be placed automatically by *trout*. The router automatically places any missing terminal of the father circuit on the border of the cell or on an optimal position. No message will be printed in this case. It is better not to place father terminals unless you really want to force them on a specific position. See section 8.4 for more details.

Power terminals The router automatically adds the 'vdd' and 'vss' terminals to your layout, in case they are not yet present yet. *Trout* will do this even if the circuit does not contain power nets.

Power rails The horizontal vdd- and vss power rails (in metal 1) are connected by vertical wires in metal 2. In this way all power terminals are connected to the vdd or vss terminals, even if the design contains many rows of transistors. Click to disable power rails connection.

Connectivity check *Trout* runs the connectivity verifier automatically after routing. If there is any short or unconnect a warning window will pop up and markers will be placed in the layout. This is the same verifier as under the button (see 5.7.10 on page 45).

8.4 Guidelines for successful routing

Trout is doing its best to route all nets for you. It can happen, however, that *trout* is unable to route them all. Some of the problems may be caused by a design error at an earlier phase. To ensure the routability of the nets, you must design the layout of your instances in a certain way. Since the cell which you are constructing now may also be used as a leaf cell on a higher level, it is wise to comply with the following guidelines:

- Make sure that all terminals can be reached by a wire from the 'outside' of cell. A terminal which cannot be connected is not useful. The best way to ensure the 'accessibility' is by placing the terminals on the border of the cell. The router helps you with that.
- Let *trout* take care of the placement of the terminals (the button Border terminals). If you do not specify the terminal positions (e.g. if you pressed erase wires), *trout* will place them on the border of the cell. The router will automatically find the shortest (and most clever) route to the border. In special cases you can always enforce a terminal placement by placing terminals manually.
- Try to make the cell as transparent as possible. Avoid using horizontal wires in metal 2. You will notice that your router itself does not like to generate this kind of wires. If *trout* does generate them, that is a sure indication that area is very congested (close to its maximum routing capacity).
- Leave some space in critical areas by shifting the modules apart. Manually update your *madonna* placement: she is not too clever.
- *Trout* does not make short-circuits by itself, it can only make existing short-circuit bigger. If the automatic checker detects a short-circuit after routing, this generally indicates the presence of an error (short-circuit) in one of the son-cells. In many of the cases a terminal in a sub-cell makes a short circuit with the terminal of another net in that sub-cell. Since *trout* expands the entire pattern of the terminal, including the entire short-circuit, connecting more terminals of the net will just make the short bigger. Unfortunately finding the cause of these short is not too easy, since the exact position of the cause of the problem cannot be indicated in the layout.

8.5 What should I do in the case of incomplete routing?

Despite all precautions, incomplete routing may still occur. Basically we can distinguish two situations for this case:

1. Many nets are unconnected. With 'many' we mean more than 5 or more than a few percent.
2. A few shattered nets are unconnected.

The strategies for these situations will be discussed in the following subsections.

8.5.1 Many nets are unconnected

In this case the conclusion is simple: the cell must be enlarged. We have to live with the fact that the routing capacity on our Sea-of-Gates chip is limited. In almost all cases we will have to sacrifice some transistors to create space for the wires. You can do this by:

1. Running *madonna* again in a larger box.
2. Running *madonna* again in a box with a different width/height ratio. In 'flat' horizontal cells the capacity for horizontal wires is easily too small. In narrow vertical cells there might not be enough space for vertical wires. Have a look at the

density of the wires in each of the layers so see which is the case. Use X-expand or Y-expand to guide *madonna*.

3. Manually replacing some blocks. Try to distribute the wiring density over the entire cell area.

Prevent spending too much time squeezing the size of your placement. Always try the easiest and quickest way first: use *madonna*.

8.5.2 A small number of nets is unconnected

At this point it is useful to get a little feeling for way in which the router works. *Trout* routes the nets one at a time in a 'greedy' way. For each net he tries to find the best wires which connect the terminals. The router is very clever in finding this path. He uses all tricks in the image (such as using the polysilicon gates for wires). You can be sure that if the router reports that it cannot connect a net, you will also not be able to do it manually. Unfortunately the router does not have much overview over the entire set of nets and wires. Therefore it is possible that the wires of a net block a terminal of a still unrouted net. This was the cause of the incomplete routing in the case of a few shattered unconnects. Notice that this is very dependent on the order in which the nets were routed.

The problem can be solved in the following ways:

1. Trout itself tries to solve the problem by ripping up wires and re-routing. It will only do this with a small number of unconnected nets and if the CPU-consumption is not too high.
2. Make a new placement by running *madonna* again in the same or a slightly larger box. This will create a new one which - with a little bit of luck - will be routable. Any time you press madonna, you'll get a different placement.
3. Delete a part of the congested area and run *trout* again. You can specifically delete some pieces of wire which you suspect to be the cause of the problem. The router will try to glue the nets back together again, in a different order. You can use the route box only-button to focus the router. Have a look at which spot the unconnects are located.
4. Route the cell in a few overlapping boxes using the route box only-button.
5. Modify the layout manually. At any case you can call the router again to finish the result automatically.

8.6 The run-time of *trout*

The router will automatically keep you informed about its progress. If it takes a longer time, *trout* will also print a rough estimate of the time it needs to finish. The cpu-time consumption of *trout* depends on the following factors:

- The dominating factor is the **size** of your placement (in square grid points). The run time increases quadratically with the size: if a layout is twice as large, it takes four times as much time to route it.

- The run time increases approximately linear with the number of nets.
- The router is at its best for large complicated layouts with many wires. It is quite slow, however, if there is a lot of empty space in your placement. This is because in the latter case there are many different ways of making a wire. *Trout*'s algorithm evaluates many routes for each wire.
- For small placements, the overhead time for database IO is determining the time you have to wait.
- If *trout* fails to route a wire, it may spend some time trying to complete the routing in a different way. It might also spend some time trying to solve problems by rip-up and re-routing.

The worst-case CPU-consumption should be in the order of an hour for 200 nets spanning the entire 200.000 transistor sea-of-gates chip. The typical cases are much better: generally it is much less than a minute.

8.7 Additional options of *trout*

Clicking **Option menu** opens a menu with a few more optional features of the router. They are only useful for the final assembly of your layout, and they should (in general) not be used to make leaf cells.

make capacitors This is the most powerful option of all. It causes trout to convert all unused transistors into capacitors between the power lines. In this way the noise on the power lines decreases. This option creates many contacts and therefore makes your layout very big. Generally you should only use it at the highest level of hierarchy. See figure 3.11 on page 32 for an example of the result this option.

dangling transistors Connect all unused (dangling) transistors to the appropriate power wire. In this way the state of all transistors is known and the capacity of the power nets increases.

substrate contacts Create substrate contacts under the horizontal power rails. The substrate contacts are required by the design rules of the process.

make fat power Make the horizontal power rails as big as possible. In this way the resistance of the power wires decreases and the capacitance increases. *Trout* will also attempt to make as many as possible vertical connections between the horizontal power rails. The latter can be turned off by clicking **no power**. In a large hierarchical design we advise you to click **flood mesh** as well.

use borders Normally the router doesn't use the upper- and lower-most row, which is on top of a power rail. In this way it prevents short-circuits with adjacent cells. In some cases you might wish to switch off this feature by pressing this button.

no routing No routing of the signal nets. Use this option to add the special features to the layout (fat power, substrate contacts, etc.). This option turns off the automatic verification of the connectivity.

flood mesh In a hierarchical design some fat wires may contain some holes. Especially after using **fat power** this could be the case. Pressing **flood mesh** causes the router to fill all holes in your design. Please note that pressing **fish** again will

undo the result of this option, because *fish* does not have a hierarchical view of your design.

overlap wires On the Sea-of-Gates chip a metal 2 wire (**ins**) has a higher probability to crack if it is running in parallel with a metal 1 (**in**) wire. These 'co-incident edges', however, are hard to avoid. To reduce the strain the button **overlap wires** will makes the **in** wires wider at the spots where they overlap with **ins** wires. In this way the edges are less co-incident. Please note that pressing **fish** again will undo the result of this option.

options Set any other (unofficial) option which should be propagated to *trout*.

Warning: The options **make capacitors**, **make fat power**, **substrate contacts** and **dangling transistors** will reduce the transparency of the cell considerably. This makes it almost impossible to route new wires through the cell. These options should therefore only be used on the highest level of hierarchy, or for critical cells. A secondary effect of these options is that the router is likely to generate a huge amount of boxes, which makes *seadali* slower and your database very big.

Warning: Pressing **fish** or **check nets** will undo the results of the options **flood mesh** and **overlap wires**. Therefore you should not use *fish* anymore after you've used these options.

8.8 The final assembly of your layout design on a chip

Suppose that after a few days of hard work you made a really wonderful layout design of your circuit (e.g the dialmemo). The simulations indicate that it should work. But the real proof of the pudding is the eating. In its current state, however, your circuit cannot be processed 'as is' at DIMES.

In this section we will describe an overview of the final preparations which are required before your circuit can be processed on a real Sea-of-Gates wafer. In the following subsection we will describe the steps to prepare your design for processing

8.8.1 The pad ring

On the actual 'fishbone' Sea-of-Gates chip we will merge more than one design into one chip. Each design must fit in one of the slots (spaces) which are on the chip. Each slot is a portion of the chip which is surrounded by terminals (which are called *pads*). Each terminal of the pad ring is connected to one of the 144 pins of the actual Sea-of-Gates chip. Your design must be placed inside a pad ring. The pad ring is available as a library cell. For this description we will use the pad ring for the small test cell which is called 'bond_leer'.

8.8.2 Connecting terminals to the pad ring

The terminals of your top-level design must be properly connected to the pad ring. The easiest way to do this is by using the router.

We first have to make a new cell which calls your design and the terminal ring as instances. You can call it 'topcell', because it will be the highest hierarchical level of layout.

Use *seadali* to place the pad ring and your design (as instances). The pad ring must be placed at (0,0), and your design must be placed in such a way that it aligns with the proper power wires. Your design may never exceed the boundaries of the pad ring!

Create a circuit description for 'topcell' in which the appropriate terminals of the two instances are connected. Each terminal (pad) on the pad ring should be connected to one terminal of your design. Connect the terminals in such a way that the wires are as short as possible. For instance, it is better not to connect a terminal on the left bottom of your design with a right top pad terminal.

Now we can press to connect the terminals of 'topcell'¹. If necessary, improve your design a little by modifying the placement or by changing the pad assignment of the terminals or the placement.

8.8.3 Final touches

We are not ready yet! We have to make sure that the following features are OK:

- Substrate contacts must be placed under the power rails. In this way latch-up is prevented. There should be approx. one substrate contact every three grid points.
- All vss and vdd power rails must be connected.
- The power nets must be properly connected to the appropriate terminals of the pad ring.
- The power wires must be wide enough for the current which your circuit consumes.
- It is better to connect the unused transistors to the power nets. In this way the state of each transistor is known (it is closed).
- To improve the reliability of the circuit the presence of co-incident edges of metal 1 and metal 2 must be reduced as much as possible.

You are lucky, because the router can perform all of the above tasks for you at once. It can be done by routing 'topcell' in the following way:

1. Click and select the -menu. This menu contains the more exotic features of the router. We refer to section 8.7 for a more elaborate description of the buttons. In this section we will only use them.
2. Click ,
3. Return to the previous menu by clicking and start the router by clicking .
4. Obviously, if you are unsatisfied by the result you can modify it or re-route it again. The final routing, however, should ALWAYS be with the options as under item 2.
5. Do not click or anymore, because this may undo part of the finishing touches.
6. Have a good look at the connection of the power nets to the power terminals on the pad ring. Make them wider if necessary.

¹It is also possible to combine this with the 'final touches' which are described in the next section. In that way the router is called only once, but small modifications become more complicated.

You will be amazed by the looks of the circuit after this treatment. Have a good look to see that the router filled all unused space with power wires. Many vertical wires in **ins** were added to make a solid connection between the power wires. Don't be afraid for short circuits. The connectivity verifier was run automatically to check for short circuit between the nets or unconnects.

Chapter 9

Using OCEAN: tips and tricks

9.1 Simple troubleshooting: the seadif database

'Seadif' is a special representation for grid-based sea-of-gates layout. It is stored in the subdirectory 'seadif' of every project. The seadif representation of a cell is generated automatically, so you don't need to worry about it. In some cases, however, the seadif representation could become corrupt or incomplete. This might happen after a program crash or two users writing at the same time. Fortunately, the seadif representation is redundant, so it can be re-created automatically as soon as you run *madonna* or *trout* the next time. Therefore, the following command will solve most of your troubles:

```
/user/hillary/myproject % rm -rf seadif
```

This command removes the entire seadif directory and its contents. Only for the case that you are importing cells from this directory, you should re-create the seadif database manually using the command *nelsea*:

```
/user/hillary/myproject % nelsea
```

9.2 Assembling cells from various projects

Combining cells from different projects is quite easy. Using the commands *addproj* and *impcell* you can import remote cells into your project. To make sure that everything will work smoothly, please comply with the following three simple rules. Suppose that the want to make a cell in the project called 'top-project', which needs imported son-cells from remote projects 'project1', 'project2' and 'project3'. In order to make things more realistic (and complicated), suppose also that in some of the imported projects also other projects are imported: the son-cell in 'project1' imports grandchildren from projects 4 and 5, and the cell in 'project2' imports some kids from 'project5'. Confused? The situation is shown in figure 9.1.

Rule 1: Add ALL projects in the tree with *addproj*, also the projects which are imported indirectly.

For the example, we should not only add projects 1, 2 and 3, but also the other ones with grandchildren or even the most remote relatives in the tree:

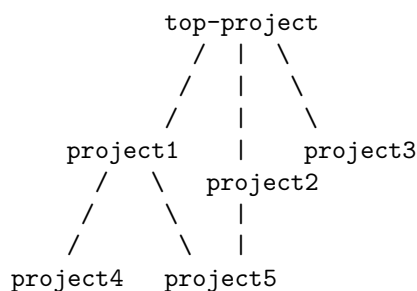


Figure 9.1: Example of a project hierarchy

- `/user/hillary/top-project % addproj /users/neuzeltje/project1`
- `/user/hillary/top-project % addproj /users/calimero/project2`
- `/user/hillary/top-project % addproj /users/prince/project3`
- `/user/hillary/top-project % addproj /users/neuzeltje/project4`
- `/user/hillary/top-project % addproj /users/hillary/project5`

With the command *impcell*, on the other hand, you only need to import the son-cells which you really need in 'top-project'.

Rule 2: All project names in the imported tree must be unique.

For example, you will run into troubles if you try to import both the projects `'/users/neuzeltje/memory'` and `'/users/calimero/memory'`, because the project name "memory" is used twice. You can prevent name clashes simply by renaming a project. See the section 9.3 (on page 69) to find out how to do that.

Rule 3: All cells in the tree must have a proper seadif representation.

For every son-cell there must be a proper seadif-representation. Normally this is performed automatically, but problems with the 'seadif' representation are more likely with imported cells. *Madonna* or *trout* will tell you exactly which seadif representation is missing. The cure is easy. First you go to the project about which *madonna* (or *trout*) complains. Then you use the program *nelsea* to force an update of the seadif representation:

```

/user/hillary/project5 % cd /users/neuzeltje/project4
/user/hillary/project5 % nelsea

```

You must have write permissions in the directory where you run *nelsea*. Without arguments, the command *nelsea* will create or update the seadif representation of all cells in the project. If there are many unimportant cells, you can speed things up by specifically updating one cell. For instance, the command:

```

/user/hillary/project5 % nelsea myadder

```

will update cell `myadder` and all the children (and grandchildren, if necessary) of `myadder`.

9.3 Renaming, copying or moving projects: how to proceed

In some cases it might be necessary to rename project, to duplicate one, or to move a project to a different place in the file system. In order to prevent unpleasant surprises, follow the guidelines below.

Step 1: Make sure that no other project is importing from the project.

Since the project will be renamed, any old references to it will point "ins Blaue hinein", like the Germans say. Generally you should only move top-level projects.

Step 2: Move or copy the project

Within the same file system (that is, the same disk), you can use the UNIX command *mv*:

```
/user/hillary % mv oldproject newproject
```

This is the fastest way. To move to a different disk, or if you want to make a copy, use *cp -r*:

```
/user/hillary % cp -r oldproject /otherdisk/prince/newproject
```

Step 3: Clean up the seadif representation

After moving or copying, the subdirectory 'seadif' of the project still contains old name and path of the project. There is a simple way to solve this. First go to the project, remove the existing seadif representation, and then create a new one:

```
/user/hillary % cd newproject  
  
/user/hillary/newproject % rm -rf seadif  
/user/hillary/newproject % nelsea
```

Also on other situations where you get strange database errors, you can do this to clean things up.

Chapter 10

Logic Synthesis with 'kissis'

SIS (Berkeley Logic Synthesis System) is an interactive tool for synthesis and optimization of sequential circuits. Given a state transition table, a logic level description of a sequential circuit or PLA (programmable logic array) description, it produces an optimized net-list in the target technology while preserving the sequential input-output behavior.

Kissis is a front-end to *SIS* which allows automatic generation of a circuit for given STG (state transition graph) or PLA. For this purpose it runs *SIS* in batch mode. The resulting circuit is automatically placed in *Nelsis* database. The following steps are performed:

- state minimization using program called *stamina* (STG input only).
- state assignment using program called *nova* (STG input only).
- combinatorial optimization.
- technology mapping.
- retiming (STG input only and when *-r* specified).
- conversion from *blif* output format to *sls* (Switch Level Simulator) using program *blif2sls*.
- conversion from *sls* format to the *Nelsis* database using *csls* program.

We cannot distribute the *SIS* package with the OCEAN tools, unfortunately. You can retrieve *SIS* yourself using anonymous ftp (host *ic.berkeley.edu*, directory *pub/sis*) or contact us.

10.1 How to use *kissis*?

You start the tool from command line in the following way:

```
/user/hillary/newproject % kissis [options] <filename>
```

where *<filename>* is the input file name.

10.2 Legal options.

The kissis can be started with the following options:

- h display help info.
- c run combinatorial synthesis (PLA input format).
- k run sequential synthesis (kiss2 input format).
- b run sequential synthesis (blif input format).
- r run retiming (default not)

If none of these options is specified then kind of input is determined based on filename extension (default `.kiss2`). If options are specified than extension does not matter. Default extension is `.kiss2`.

10.3 What files are required before I can call *kissis*?

To run properly the command requires that the following files are present in your current directory. If the OCEAN system was properly installed, all the default files should be in the right position.

- `proto_file` - a file with sls prototypes (see *blif2sls*).
- `<cellname>.<kiss2|pla|blif>` - input file with STG (PLA) description.

Other files:

- `$OCEAN/celllibs/$OCEANPROCESS/<lib_name>.genlib` - a file with a library description (to be used during technology mapping).
- `<cell_name>_out.blif` - intermediate file (output in blif format).
- `<cell_name>.sls` - intermediate file (output in sls format).
- `<cell_name>.sta` - output from *blif2sls* used by *simeye*.
- `sis_logfile` - output from *SIS* (look here if something goes wrong).

For an example of running *kissis*, please have a look in section 3.4 on page 28.

10.4 More info about *SIS*.

For more information about *SIS* we refer to the following document:

"SIS: A System for Sequential Circuit Synthesis" Ellen M. Santovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, Alberto Sangiovanni-Vincentelli. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720. (1992)

It is present in the ocean documentation distribution.

Chapter 11

Layout to circuit extraction: 'space' and 'ghoti'

OCEAN provides powerful tools for verifying a layouts. They are indicated in the design flow diagram of figure 1.1 on page 2. The most commonly used way is layout-to-circuit extraction using *space*, where a netlist description (including parasitics) is extracted for a layout.

11.1 The extractor *space*

Space is an advanced layout-to-circuit extractor for analog as well as digital circuits. From mask-level layout, *space* produces a circuit netlist consisting of transistors. Optionally, this netlist contains also the wiring capacitances to substrate (`-c` option) and inter-wire coupling capacitances (`-C` option). *Space* can also extract wire resistances, but on the version which is supplied with the OCEAN system this feature is disabled¹. These parasitics are extracted very accurately, which enables you to tune your circuit for maximum performance with a high reliability. *Space* can handle very large circuits.

For a more detailed description of *space* we refer to the *SPACE User's manual* by Nick van der Meijs and Arjan van Genderen, which is included in the OCEAN documentation distribution. Examples of how to use *space* can be found in chapter 3 (especially section 3.3.6).

11.2 *space*: Capitalization of the cell name

Calling *space* on a cell `mycell` results in a new circuit cell called `Mycell` (watch the capital 'M'!). The capitalization of the first character is done to prevent that *space* overwrites the original circuit description `mycell`. This convention is also recognized by *simeye*. If you run *simeye* on the extracted circuit `Mycell`, *simeye* will use the `sls-command` file `mycell.cmd` if the file `Mycell.cmd` does not exist. In this way you can easily simulate the original and the extracted circuit with the same command file.

¹Contact the author Nick van der Meijs if you are interested in this feature (email him at: n.p.vandermeijs@tudelft.nl)

11.3 Purifying netlists: *ghoti*

Ghoti is a purifier for extracted circuit descriptions, much in the same way like *fish* is for layouts. The extracted network which results from *space* may contain some irrelevant pieces. Especially on a sea-of-gates there will be many isolation transistors or totally unused transistors, which have no (or hardly any) influence on the function of the circuit. These useless elements, however, may cause the simulator some difficulties. The *spice* simulator crashes the unconnected transistors (complains about “singular matrices”). *Spice* will also consume considerably more CPU-time if useless transistors are present.

Ghoti was made to solve these problems by purifying the netlist. It reads a circuit description, processes it, and write a purified circuit with the following features:

- all unconnected transistors are removed,
- all isolation transistors are removed,
- power nets which are not yet connected are joined.
- (with `-r` option) reduce series and parallel connections of transistors. This option, where *ghoti* collapses any parallel or series connection of transistors into one equivalent transistor, is especially useful for analog circuits which exceed the capacity of *space*.

Therefore: **Always *ghoti* before running *spice*!** You can call *ghoti* as often as you like on a cell.

If the circuit is large (e.g. thousands of transistors, or the entire chip) using *ghoti* also reduces the run time of the switch level simulator *sls* because the number of elements to keep track of decreases drastically. See chapter 3 for examples on how to run *ghoti*.

Chapter 12

Simulating circuits with 'simeye'

Simeye provides a graphical interface to the circuit simulators. It supports three levels of circuit simulation: logic, switch-level and spice. This section discusses the *simeye* command line options and its user interface.

12.1 Description

Simeye is an X-window based simulation user interface for the *sls(1ICD)* simulator and the *spice(1ICD)* simulator. The program can be used to inspect the graphical representation of the output signals of both simulators and it can be used to edit input signals that are described in the sls command file format (see "sls: switch-level simulator user's manual"). Moreover, from *simeye* simulations can be run by starting either the *sls* simulator or the *spice* simulator (The latter is done via the script *nspice(1ICD)*).

The following program argument may be specified:

cell : This name specifies the name of the cell for which simulation signals are displayed and/or edited.

The corresponding file from which the signals are read or to which the signals are written has a name that is equal to the cell name, with an extension ".res" for sls logical and timing (output) signals, an extension ".plt" for sls waveform (output) signals, an extension ".ana" for spice (output) signals, and an extension ".cmd" for command file (input) signals. The spice output file should contain the spice signals in tabular format. When running the program *spice* directly this is achieved by using the card `.print tran ...` in the spice input file. When running *nspice*, this is achieved by using sls plot commands in the command file.

12.2 Commands of simeye

Commands that can interactively be given are:

CELL : This field specifies the name of the cell to which the signals belong. The cell name is given by the program argument (see above) or it is specified by the user

while running the program. To edit it, use the left arrow, right arrow and backspace and/or delete keys of the keyboard. To load the signals of the specified cell, click the read button.

SLS-LOGIC SLS-TIMING SPICE : These buttons define the type of simulation. When clicking the run button, an sls simulation at level 1 or 2 is performed when SLS-LOGIC is active, an sls simulation at level 3 is performed when SLS-TIMING is active, and a spice simulation is performed when SPICE is active. When clicking the read button, These buttons define the type of simulation. When clicking the run button, an sls simulation at level 1 or 2 is performed when SLS-LOGIC is active, an sls simulation at level 3 is performed when SLS-TIMING is active, and a spice simulation is performed when SPICE is active. When clicking the read button, sls signals are read when SLS-LOGIC or SLS-TIMING is active, and spice signals are read when SPICE is active.

READ : Load a new signal file. Normally, current signals are cleared from the program. However, if the shift key is held down while pressing the read button the new signals will be appended to the current signals.

RUN : Perform a simulation by using either the *sls* simulator or the *spice* simulator. For both simulators *simeye* uses the command file *cell.cmd*. The spice simulator is run by calling the program *nspice*. If any error message occurs it will be displayed by *simeye*. If the simulation succeeds, the simulation output will automatically be displayed by *simeye*.

FULL : Draw all present signals from the beginning of the simulation time till the end of the simulation time.

REDRAW : Redraw the current window.

IN : Zoom in on the current window. You'll have to indicate the rectangle that you want to zoom in. First click one corner, then the other corner of the zoom-in area.

OUT : Zoom out on the current window. After clicking OUT you'll have to indicate a rectangle on the screen (with two mouse clicks). This is the rectangle that will contain the *current* window *after* the zoom-out has been performed. Consequently, for a large zoom-out you indicate a small window and for a marginal zoom-out you indicate a big window.

LEFT, RIGHT, UP AND DOWN ARROW : Move the current window leftwards, rightwards, upwards or downwards respectively.

S : Save the current window settings (i.e. start-time, stop-time and the names of the signals that are displayed) in a file (The name of this file is specified in the configuration file).

L : Load the current window settings (i.e. start-time, stop-time and the names of the signals that are displayed) from a file (The name of this file is specified in the configuration file).

MOVE : With this command the order of the signals can be changed or one signal can be placed over another signal in order to compare them. First the signal that is moved or that is placed over another signal is selected, and then the new position of the signal is selected. To place the selected signal over another signal one should hold the shift key down while selecting the new position. To remove a signal that is placed over another signal, initially select the signal with the shift key held down.

VALUE : Move a vertical scanline over the display window, according to the position of the pointer, and show the value of the corresponding x position (and y position). When clicking the middle (or right) button of the mouse, one may toggle between displaying in the right margin (if the number of signals is not too large) the y values of the signals at the selected x position. To store the value of a particular position, click the left button of the mouse. This position will then be subtracted from the next positions of the scanline, so as to measure delay times and/or voltage differences.

PRINT : Generate a hardcopy of the current screen.

SPACE : Run the layout-to-circuit extraction program. A layout-to-circuit extraction may be aborted by typing Ctrl-C.

QUIT : Quit the program

INPUT : Read the command file *cell.cmd* and enable the edit menu. If the command file *cell.cmd* does not exist, *simeye* will try to read a default command called "simeye.def.d". First, it tries to read this file from the current directory and second it tries to read this file from the process directory.

The following commands are part of the edit menu:

GRID : Specify the smallest unit for the x-axis (= time axis).

NEW : Create a new signal. The user has to enter the name of the new signal, and the signal will be placed at the bottom of the window.

DELETE : Delete one or more signals by selecting them with the cursor.

CLEAR : Delete all signals.

COPY : Copy the signal description from one signal to another signal.

EDIT : Edit a particular signal by inserting a new logical level for a particular time interval (t1, t2). The signal description may eventually already be defined for this interval. The insertion of the new logical level is done in two steps: During the first click of the mouse the signal, the new logical level and t1 are selected. During the second click t2 is selected and the new signal description is drawn. To insert an interval that has a free state one should hold the shift key pressed down while making the first selection.

YANK : Store a (part of a) signal description in the buffer.

PUT : Insert a copy of the signal description that is in the buffer onto a particular position. The user has to select the signal to which the contents of the buffer is added and the time from which on the new signal description is valid. The new signal description may (partly) override the existing signal description for the selected signal. Furthermore, the user is asked to type the number of repetitions for the signal description that is added.

SPEED : Speed up the signals by some factor. The value of 'sigunit' in the command file will be divided by the value that is specified with this command. If a value < 1 is specified, the signals will be slowed down.

T_END : Update the end time of the input signals (= end time of simulation).

READY : Disable the edit menu and update the command file *cell*

12.3 Command file for sls

In order to simulate a circuit, *simeye* uses a command file called *cell.cmd*. This file should contain a description of the input signals, values for the simulation control variables, and a listing of the terminals for which output should be generated. This is explained in more detail in the user manuals of the sls simulator and the spice simulator.

The "set" commands in the command file may be edited by enabling the edit menu of *simeye*. The new signal descriptions will then be written back to the command file when updating the command file. However, when a set command is followed by the keyword "no_edit", between comment signs, this set command will not be changed. This is for example useful to define supply signals and periodical signals like clock signals.

When the command file contains on separate line, between comment signs, the keyword "auto_print" ("auto_plot"), *simeye* will automatically add a print (plot) statement to the command file for each terminal of the cell that is simulated, prior to simulation. The new print (plot) commands will have a keyword "auto" between comment signs following the keyword print (plot) to indicate that the print (plot) command will be replaced each time when a new simulation is started.

An example of a default command file that can be used for both sls and spice simulations is:

```
/* auto_set */
set /* no_edit */ vdd = h*~
set /* no_edit */ vss = l*~
set /* no_edit */ phi1 = (l*110 h*80 l*10)*~
set /* no_edit */ phi2 = (l*10 h*80 l*110)*~
option sigunit = 1n
option outacc = 10p
option simperiod = 4000
option level = 3
/*
*%
tstep 0.1n
trise 0.5n
tfall 0.5n
*%
*%
*/
/* auto_print */
/* auto_plot */
```

12.4 Startup file for simeye

At start-up of the program, *simeye* will read some information from a file called ".simey-erc". First, it tries to find this file in the current directory. Second, it tries to open this file in the process directory. The start-up file may contain the following keywords, followed by a specification on the same line if the keyword ends with ':':

SLS: Specifies the command for running the *sls* simulator.

SLS_LOGIC_LEVEL: Specifies the level of simulation when "sls-logic" is selected (use 1 or 2).

SLS_LOGIC_SIGNAL: Specifies the default signal representation for sls-logic simulations (use A or D).

SLS_TIMING_SIGNAL: Specifies the default signal representation for sls-timing simulations (use A or D).

SPICE: Specifies the command for running the *spice* simulator (use *nspice* or a derivative of it).

XDUMP_FILE: Specifies the name of the X Window dump file that is generated when the print button is clicked.

PRINT: Specifies the command that is executed when the print button is clicked in order to process the X Window dump file (e.g. to convert the window dump to PostScript and to send the output to a laser-printer).

PRINT_LABEL: Specifies an optional label that is placed in upper left corner of the display window when an X Window dump is generated.

SETTINGS_FILE: Specifies the name of the file in (from) which the window settings are stored (loaded) when using the "S" button ("L" button or -L option).

DETAIL_ZOOM_IN if this keyword is specified in the configuration file, the zoom-in function is also defined for the y-axis of a (single analog) signal. In this case it is not necessary to hold the shift key down (see command IN).

DETAIL_ZOOM_ON If this keyword is specified in the configuration file, the zoom-in function is also defined for the y-axis of a (single analog) signal. In this case it is not necessary to hold the shift key down

TRY_NON_CAPITAL_ON If this keyword is specified in the configuration file and simeye fails to open a command file that starts with a capital letter, the program will try to open the same command file but now starting with a non-capital letter. If this succeeds and when performing a simulation, simeye will first run the program on a copy of the command file to expand one dimensional array node names into single node names (e.g. a[1..3] is converted into a_1_ a_2_ a_3_).

In the above specifications, the string '\$cell' may be used to refer to the current cell name, '\$date' may be used to refer to the current date, and '\$time' may be used to refer to the current time.

Example of a start-up file (default values are shown):

SLS: sls \$cell \$cell.cmd

SLS_LOGIC_LEVEL: 2

SPICE: nspice nspice \$cell \$cell.cmd XDUMP_FILE: simeye.wd

PRINT: xtops -white 1 -in simeye.wd -out \$cell.ps; pspr \$cell.ps; rm simeye.wd

PRINT_LABEL: \$cell \$date \$time SETTINGS_FILE: simeye.set

Chapter 13

The sea-of-gates libraries

13.1 Introduction

This chapter describes the images and libraries supplied with the Ocean system. The 'image' is the basic pattern on a semi-custom chip. We distribute three types of images:

fishbone A gate-isolation image in a 1.6μ process with two layers of metal.

octagon A remarkable octagonal image in an imaginary 0.8μ process with three layers of metal interconnect.

gatearray An old fashioned gate-array in a single metal layer process.

See section 2 for examples of designs in these images. At Delft university we can only process the fishbone image. Therefore most of this chapter deals with this image. The other images are supplied mainly to demonstrate the features of the placer and the router in the OCEAN system. They have only a small cell library which is a subset of the fishbone library. Also SPICE and SLS simulations will not give realistic results for *octagon* and *gatearray*.

In the remainder of this chapter we describe the 'fishbone' Sea-of-Gates library which is used at Delft university of technology, in the release of august 1993. The library we supply is quite small, since it doesn't contain the intricate combinatorial logic cells. This was done deliberately to keep it simple for the users. Moreover, in our sea-of-gates style complex combinatorial cells can be constructed quite efficiently, so there is not really a need for such cells in the library. We split the libraries according to their function:

primitives: Library with the basic description of the Sea-of-Gates image.
It contains fishbonec3tu2, Via_on_in, Via_op_in, Via_in_ins and Error_Marker

digilib8_93: Library with elementary digital cells.
It contains: buf40, de211, de310, dfn10, dfr11, ex210, iv110, mu111 mu210, na210, na310, no210 and no310

analib8_93: Library with analog cells. It contains: ln3x3, lp3x3, mir_nin, mir_nout, mir_pin, mir_pout and osc10.

bonding8_93: Library with elementary bonding patterns, containing bond_leer, bond_bar and bondflap_8.

exlib8_93: The 'expert' or 'extended' library, containing a few more cells which are not (yet) intended for practicum users. This library also contains the old cells which were kicked out of the oplib. The cells dfr112, dfn102 are the old (and smaller) flipflops. Contains currently: buf20, buf100, dfr112, dfn102, add10, itb20

If you use *mkopr*, the following libraries are imported into the new project: 'primitives', 'digilib8_93', 'analib8_93' and 'bonding8_93'.

For each library there is a file with the 'external' sls descriptions. they are useful to help you with inserting the proper terminal order in your sls circuit files. The fiels with the external definitions are in the directory 'sls_prototypes' of your project directory.

If you use the command *mkepr* (notice the 'e'), also the extended library will be imported into the new project. *mkepr* Imports the following libraries: 'primitives', 'digilib8_93', 'analib8_93', 'bonding8_93' and 'exlib8_93'.

At the end of this chapter we include some information about the image description file 'image.seadif' which is required by all OCEAN programs.

13.2 Information of each library cell

The names of the digital cells were chosen using the convention which was used by Philips for its gate-array packages. The first three characters describe the function of the cell. Sometimes the number of inputs is included in the names: 'na210' is a 2-input nand, while 'na310' is a 3-input nand. The last 2 digits describe the fan-out of the output. But sometimes I don't understand this convention myself.

The description of each cell consists of:

- Function
- Terminal connections
- IEC symbol
- Truth table
- Parameters to determine the delay
- Equivalent chip area
- Fanout

The parameters for the circuit delay consist of four parts:

T_{PLH} en T_{PHL} the fixed delay times of the cell under unit load (0.12pF).

ΔT_{PLH} en ΔT_{PHL} The delay coefficients with a capacitative load.

C_{in} The input capacitance.

T_{su} en T_{hold} Setup- and hold-times of the flipflops.

The delay times and delay coefficients are specified for the rising (LH) as well as the falling (HL) edge of the output signal. The total delay can be calculated using the formula:

$$T_{P_{tot}} = T_P + \Delta T_P(C_{load} - C_{unit})$$

in which C_{load} is the load capacitance and C_{unit} the unit load. This load capacitance is the sum of the input capacitances of the cells which are driven plus the capacity of the interconnection wires. The unit of chip area is defined as the smallest piece of the fishbone image, which consists of one nmos and one pmos transistor.

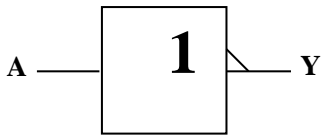
13.3 digilib8_93: the basic digital library

13.3.1 iv110

Function: Inverter

Terminals: (A, Y, vss, vdd)

IEC-symbol:



Function table:

A	Y
L	H
H	L

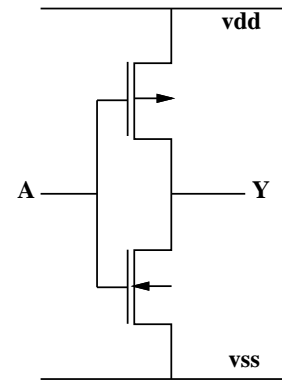
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.13	0.19	0.35	ns
T_{PHL}	A	Y	0.15	0.23	0.40	ns
ΔT_{PLH}	A	Y	-	-	1.1	ns/pF
ΔT_{PHL}	A	Y	-	-	0.8	ns/pF
C_{in}	A	vss	-	0.12	0.18	pF

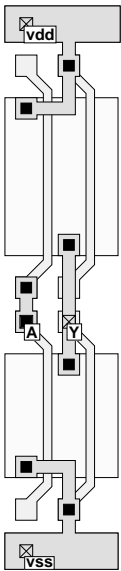
Equivalent chip area: 2

Fanout: 3.0pF

circuit:



layout:

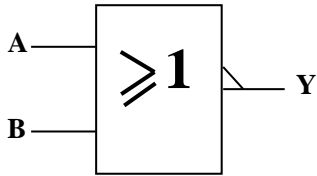


13.3.2 no210

Function: 2-input nor

Terminals: (A, B, Y, vss, vdd)

IEC-symbol:



Function table:

A	B	Y
L	L	H
-	H	L
H	-	L

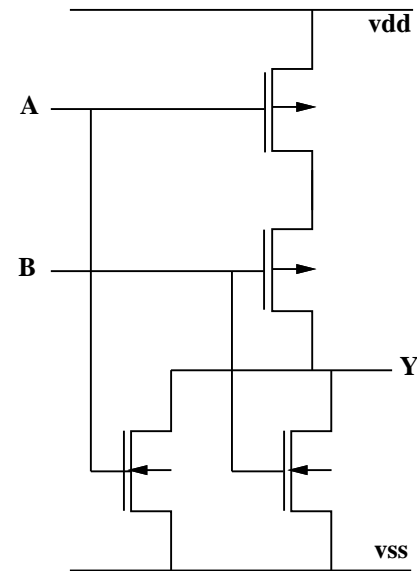
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.23	0.34	0.59	ns
T_{PHL}	A	Y	0.10	0.25	0.38	ns
T_{PLH}	B	Y	0.21	0.32	0.59	ns
T_{PHL}	B	Y	0.10	0.22	0.40	ns
ΔT_{PLH}	any	Y	-	-	1.8	ns/pF
ΔT_{PHL}	any	Y	-	-	0.8	ns/pF
C_{in}	any	vss	-	0.12	0.18	pF

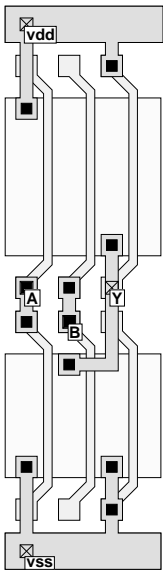
Equivalent chip area: 3

Fanout: 2.4pF

circuit:

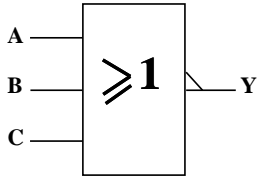


layout:



13.3.3 no310

Function: 3-input nor
Terminals: (A, B, C, Y, vss, vdd)
IEC-symbol:



Function table:

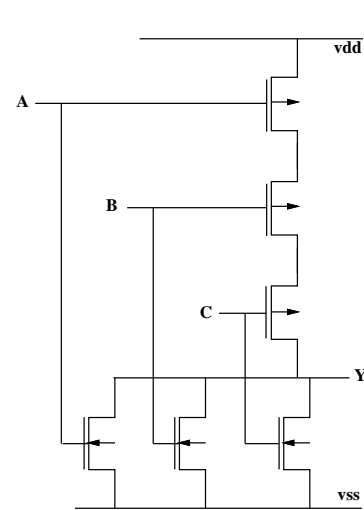
A	B	C	Y
L	L	L	H
-	-	H	L
-	H	-	L
H	-	-	L

Timing parameters:

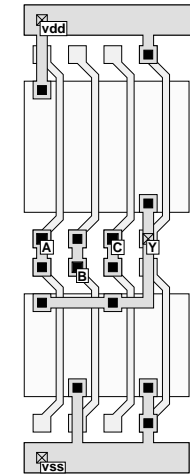
Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.36	0.54	1.1	ns
T_{PHL}	A	Y	0.08	0.27	0.47	ns
T_{PLH}	B	Y	0.36	0.53	1.1	ns
T_{PHL}	B	Y	0.08	0.27	0.47	ns
T_{PLH}	C	Y	0.30	0.46	1.1	ns
T_{PHL}	C	Y	0.08	0.24	0.41	ns
ΔT_{PLH}	any	Y	-	-	2.4	ns/pF
ΔT_{PHL}	any	Y	-	-	0.9	ns/pF
C_{in}	any	vss	-	0.12	0.18	pF

Equivalent chip area: 4
Fanout: 1.9pF

circuit:



layout:

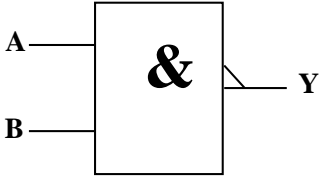


13.3.4 na210

Function: 2-input nand

Terminals: (A, B, Y, vss, vdd)

IEC-symbol:



Function table:

A	B	Y
-	L	H
L	-	H
H	H	L

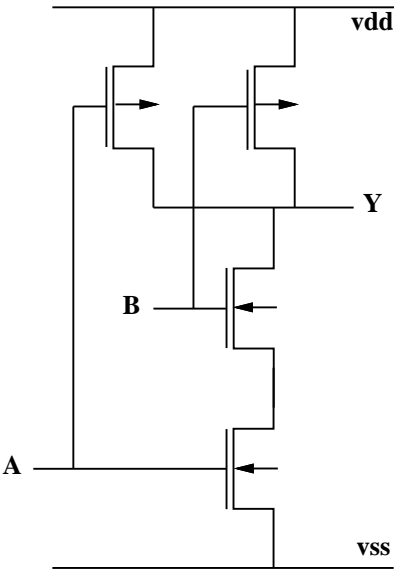
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.08	0.33	0.50	ns
T_{PHL}	A	Y	0.19	0.29	0.52	ns
T_{PLH}	B	Y	0.08	0.20	0.36	ns
T_{PHL}	B	Y	0.20	0.30	0.52	ns
ΔT_{PLH}	any	Y	-	-	1.0	ns/pF
ΔT_{PHL}	any	Y	-	-	1.1	ns/pF
C_{in}	any	vss	-	0.12	0.18	pF

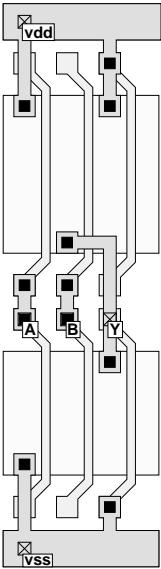
Equivalent chip area: 3

Fanout: 2.9pF

circuit:



layout:

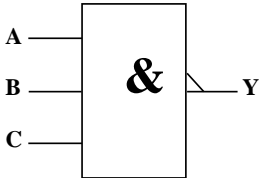


13.3.5 na310

Function: 3-input nand

Terminals: (A, B, C, Y, vss, vdd)

IEC-symbol:



Function table:

A	B	C	Y
-	-	L	H
-	L	-	H
L	-	-	H
H	H	H	L

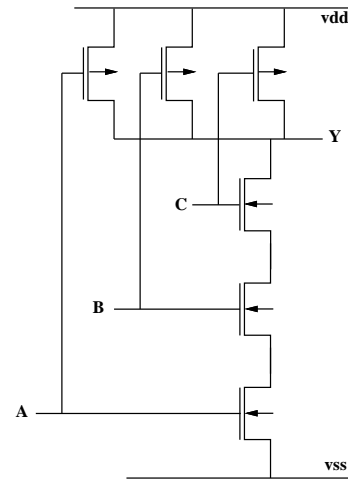
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.06	0.37	0.61	ns
T_{PHL}	A	Y	0.26	0.39	0.77	ns
T_{PLH}	B	Y	0.06	0.35	0.58	ns
T_{PHL}	B	Y	0.27	0.40	0.77	ns
T_{PLH}	C	Y	0.06	0.21	0.37	ns
T_{PHL}	C	Y	0.26	0.39	0.77	ns
ΔT_{PLH}	any	Y	-	-	0.9	ns/pF
ΔT_{PHL}	any	Y	-	-	1.4	ns/pF
C_{in}	any	vss	-	0.12	0.18	pF

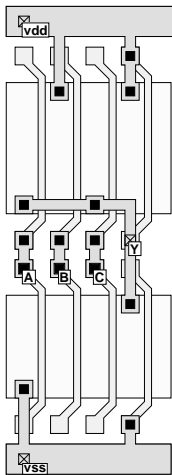
Equivalent chip area: 4

Fanout: 2.5pF

circuit:

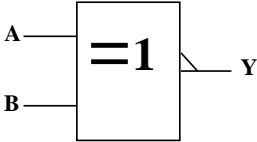


layout:



13.3.6 ex210

Function: Exclusive or
Terminals: (A, B, Y, vss, vdd)
IEC-symbol:



Function table:

A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

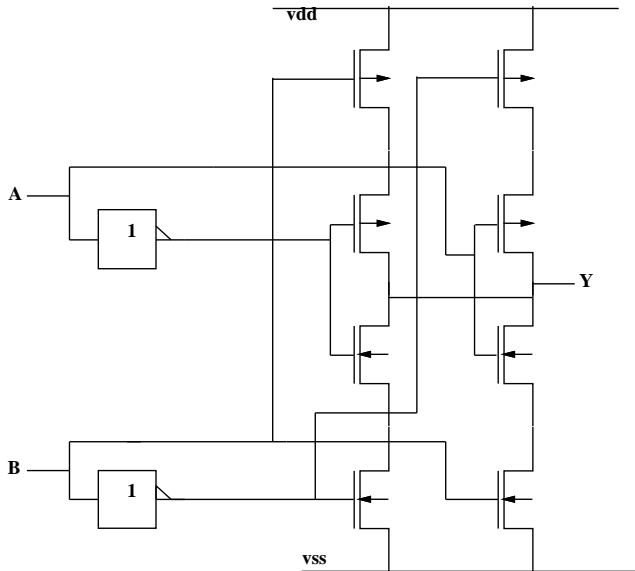
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	any	Y	0.22	0.69	1.6	ns
T_{PHL}	any	Y	0.24	0.41	0.68	ns
ΔT_{PLH}	any	Y	-	-	1.4	ns/pF
ΔT_{PHL}	any	Y	-	-	1.1	ns/pF
C_{in}	any	vss	-	0.24	0.36	pF

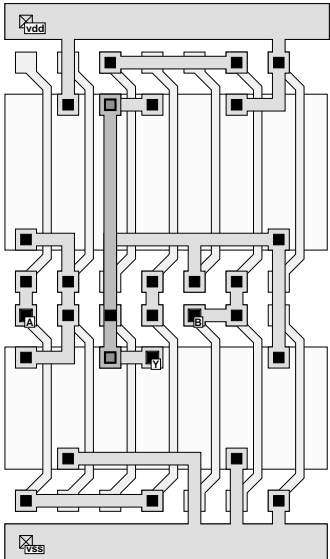
Equivalent chip area: 7

Fanout: 2.4pF

circuit:



layout:

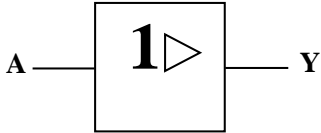


13.3.7 buf40

Function: Buffer (4x-drive)

Terminals: (A, Y, vss, vdd)

IEC-symbol:



Function table:

A	Y
L	L
H	H

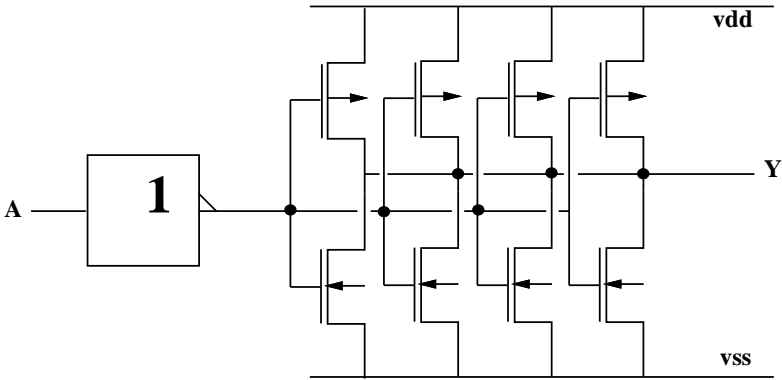
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y	0.46	0.69	1.0	ns
T_{PHL}	A	Y	0.56	0.84	1.3	ns
ΔT_{PLH}	A	Y	-	-	0.3	ns/pF
ΔT_{PHL}	A	Y	-	-	0.4	ns/pF
C_{in}	A	vss	-	0.12	0.18	pF

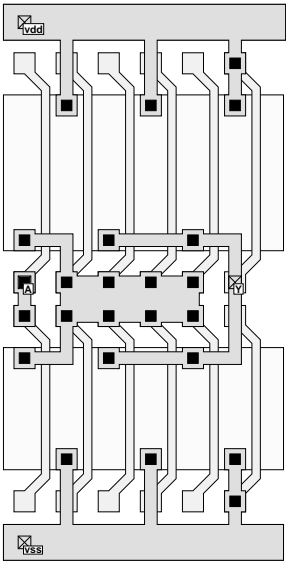
Equivalent chip area: 6

Fanout: 12pF

circuit:



layout:

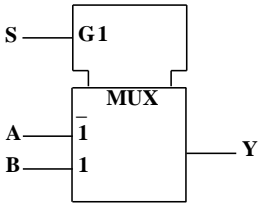


13.3.8 mu111

Function: 2-line-to-1-line data selector/multiplexer

Terminals: (A, B, S, Y, vss, vdd)

IEC-symbol:



Function table:

S	A	B	Y
L	L	-	L
L	H	-	H
H	-	L	L
H	-	H	H

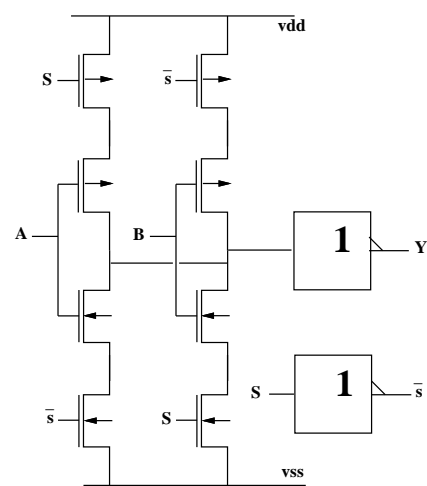
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A, B	Y	0.31	0.50	0.81	ns
T_{PHL}	A, B	Y	0.37	0.60	0.98	ns
T_{PLH}	S	Y	0.37	0.60	0.95	ns
T_{PHL}	S	Y	0.37	0.65	1.1	ns
ΔT_{PLH}	any	Y	-	-	1.2	ns/pF
ΔT_{PHL}	any	Y	-	-	1.0	ns/pF
C_{in}	A, B	vss	-	0.12	0.18	pF
C_{in}	S	vss	-	0.24	0.36	pF

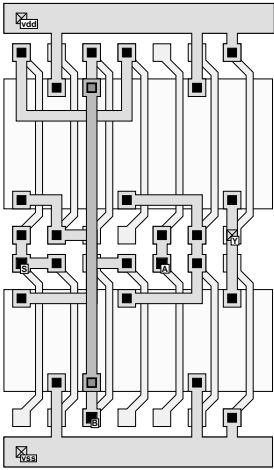
Equivalent chip area: 7

Fanout: 3.0pF

circuit:



layout:

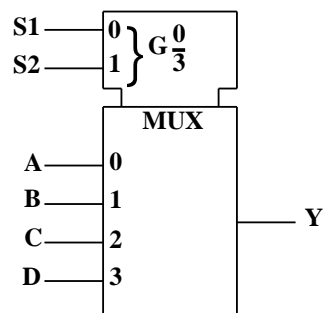


13.3.9 mu210

Function: 4-line-to-1-line data selector/multiplexer

Terminals: (S1, S2, A, B, C, D, Y, vss,vdd)

IEC-symbol:



Function table:

S2	S1	A	B	C	D	Y
L	L	L	-	-	-	L
L	L	H	-	-	-	H
L	H	-	L	-	-	L
L	H	-	H	-	-	H
H	L	-	-	L	-	L
H	L	-	-	H	-	H
H	H	-	-	-	L	L
H	H	-	-	-	H	H

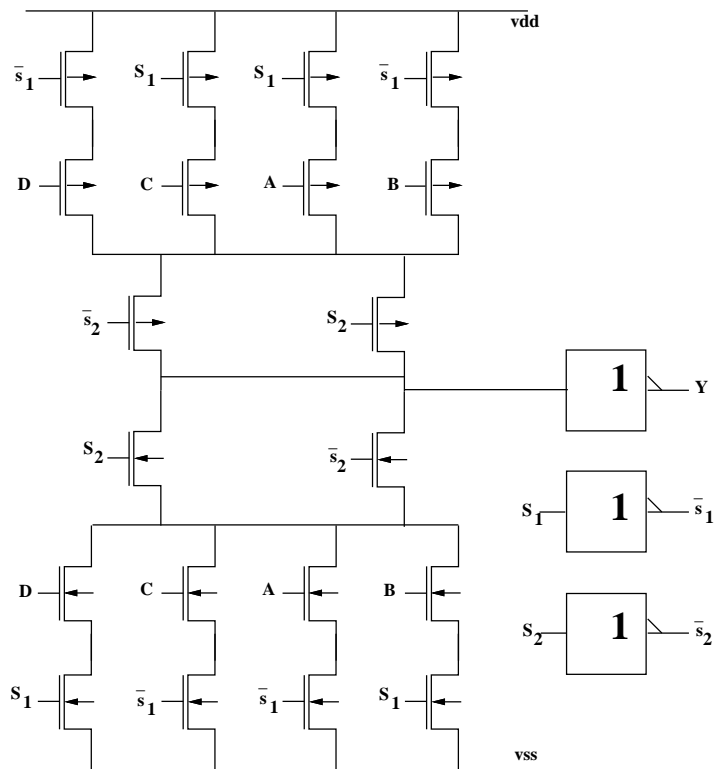
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A, B, C, D, S1	Y	0.88	1.3	2.0	ns
T_{PHL}	A, B, C, D, S1	Y	1.4	2.1	3.2	ns
T_{PLH}	S2	Y	0.77	1.2	1.7	ns
T_{PHL}	S2	Y	0.57	0.86	1.3	ns
ΔT_{PLH}	any	Y	-	-	1.3	ns/pF
ΔT_{PHL}	any	Y	-	-	1.4	ns/pF
C_{in}	A, B, C, D	vss	-	0.12	0.18	pF
C_{in}	S1	vss	-	0.36	0.54	pF
C_{in}	S2	vss	-	0.24	0.36	pF

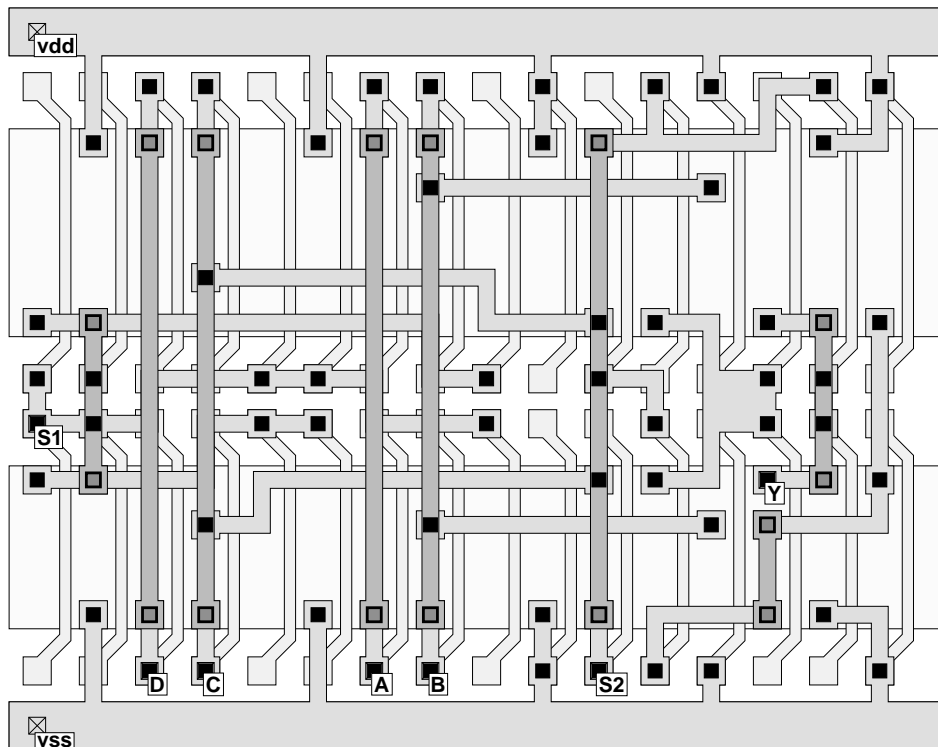
Equivalent chip area: 16

Fanout: 3.0pF

circuit:



layout:

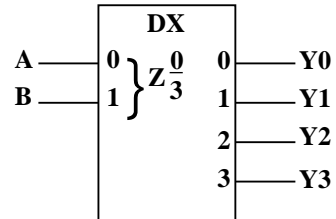


13.3.10 de211

Function: 2-to-4 decoder/demultiplexer

Terminals: (A, B, Y0, Y1, Y2, Y3, vss, vdd)

IEC-symbol:



Function table:

B	A	Y0	Y1	Y2	Y3
L	L	H	L	L	L
L	H	L	H	L	L
H	L	L	L	H	L
H	H	L	L	L	H

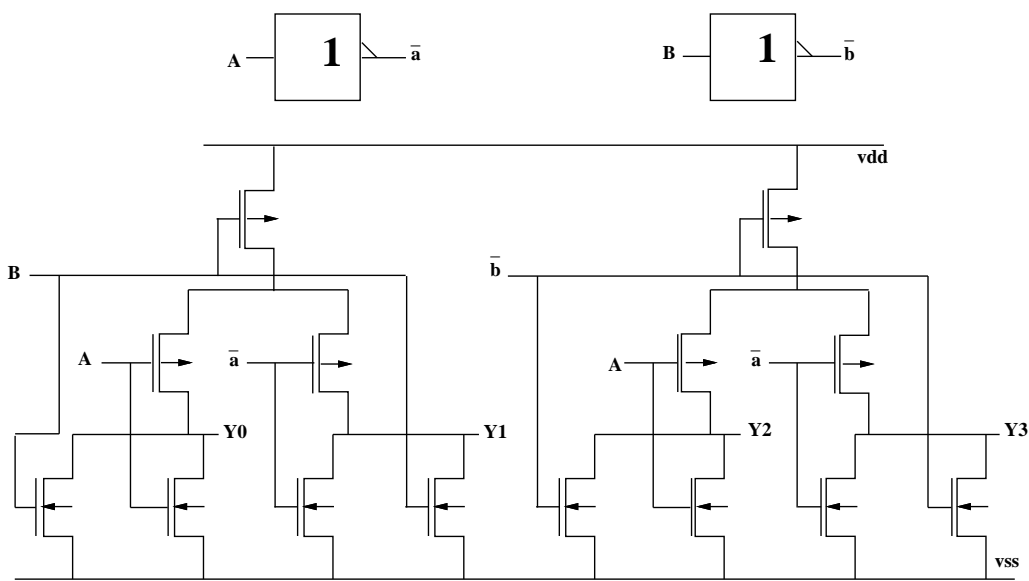
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	A	Y0,Y2	0.23	0.34	0.51	ns
T_{PHL}	A	Y0,Y2	0.17	0.25	0.38	ns
T_{PLH}	A	Y1,Y3	0.41	0.62	0.93	ns
T_{PHL}	A	Y1,Y3	0.36	0.54	0.81	ns
T_{PLH}	B	Y0,Y1	0.20	0.30	0.45	ns
T_{PHL}	B	Y0,Y1	0.15	0.22	0.33	ns
T_{PLH}	B	Y2,Y3	0.38	0.57	0.87	ns
T_{PHL}	B	Y2,Y3	0.34	0.51	0.76	ns
ΔT_{PLH}	any	any	-	-	1.8	ns/pF
ΔT_{PHL}	any	any	-	-	0.8	ns/pF
C_{in}	A	vss	-	0.36	0.54	pF
C_{in}	B	vss	-	0.30	0.45	pF

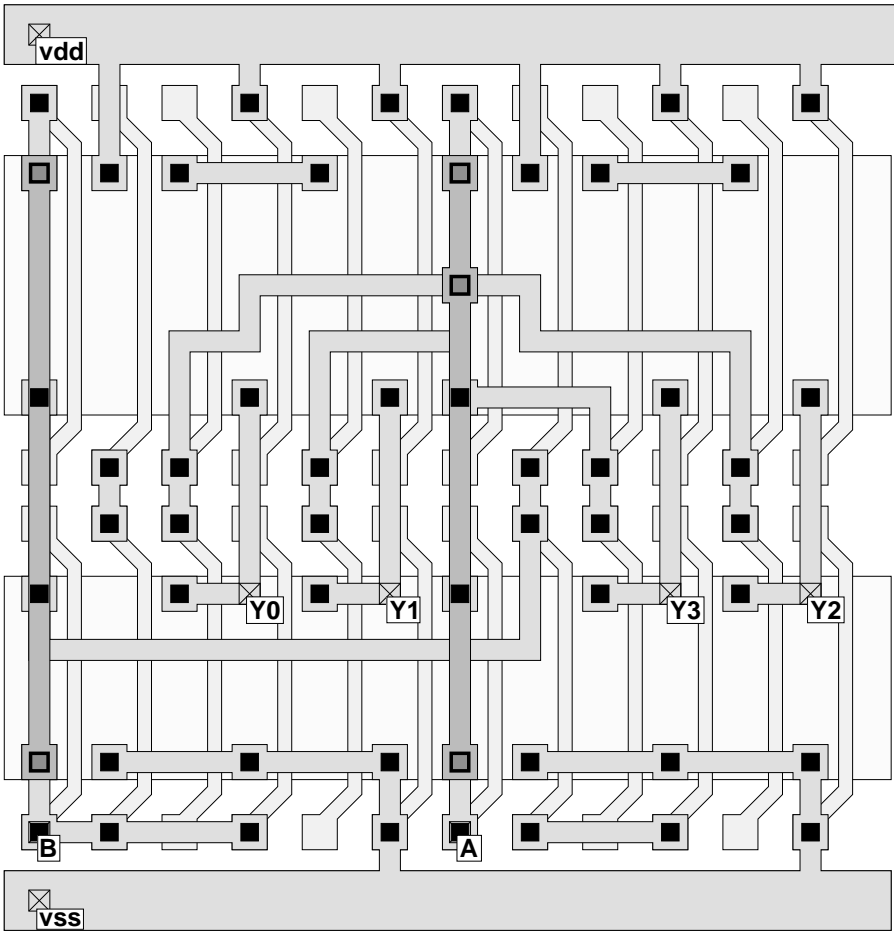
Equivalent chip area: 12

Fanout: 2.4pF

circuit:



layout:

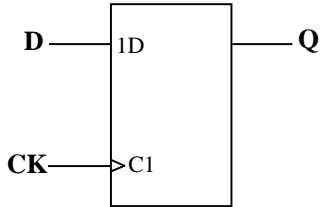


13.3.11 dfn10

Function: D-type positive edge triggered flipflop

Terminals: (D, CK, Q, vss, vdd)

IEC-symbol:



Function table:

D	CK	Q
L	↑	L
H	↑	H

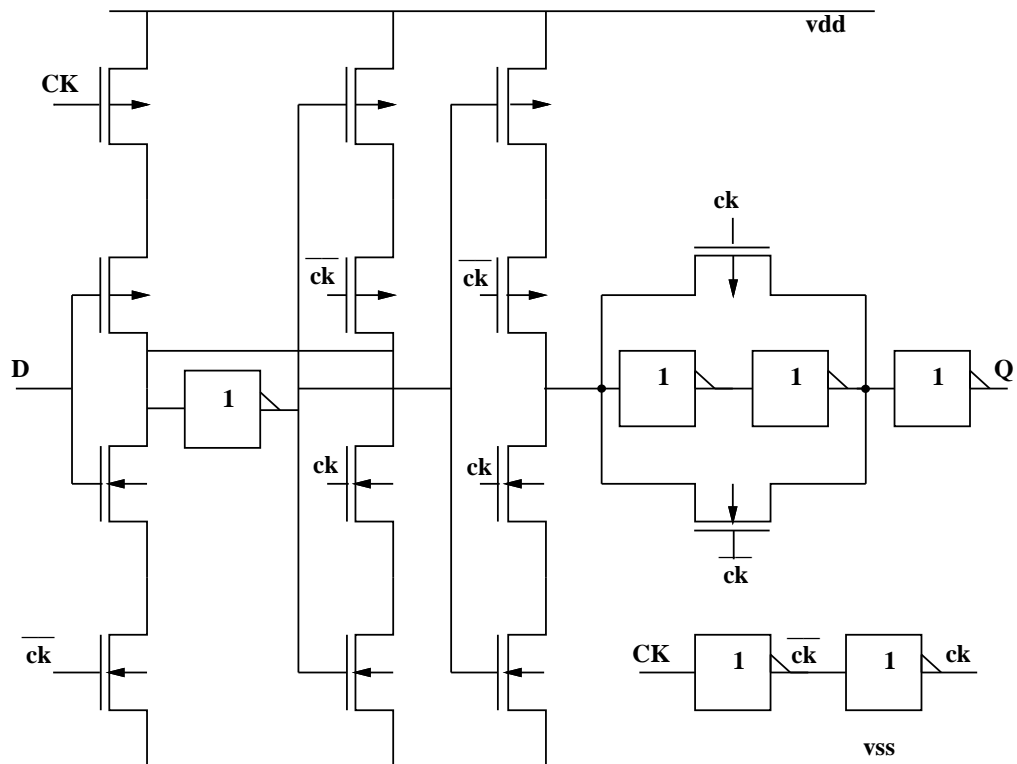
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	CK	Q	1.4	2.1	3.2	ns
T_{PHL}	CK	Q	1.3	1.9	2.9	ns
ΔT_{PLH}	CK	Q	-	-	1.1	ns/pF
ΔT_{PHL}	CK	Q	-	-	0.8	ns/pF
T_{su}	D	CK	-	-	1.4	ns
T_{hold}	D	CK	-	-	0.9	ns
C_{in}	D	vss	-	0.12	0.18	pF
C_{in}	CK	vss	-	0.18	0.27	pF

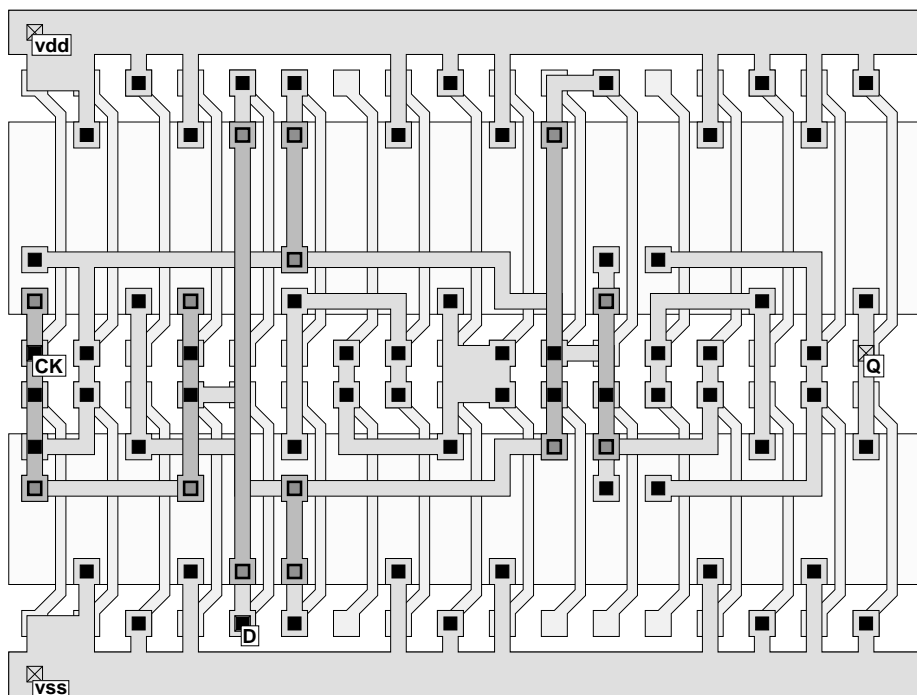
Equivalent chip area: 17

Fanout: 3.0pF

circuit:



layout:

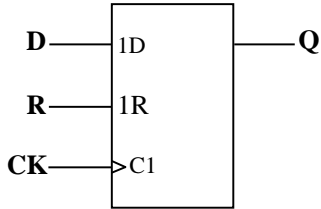


13.3.12 dfr11

Function: D-type positive edge triggered flipflop with synchronous reset

Terminals: (D, R, CK, Q, vss, vdd)

IEC-symbol:



Function table:

R	D	CK	Q
L	L	↑	L
L	H	↑	H
H	-	↑	L

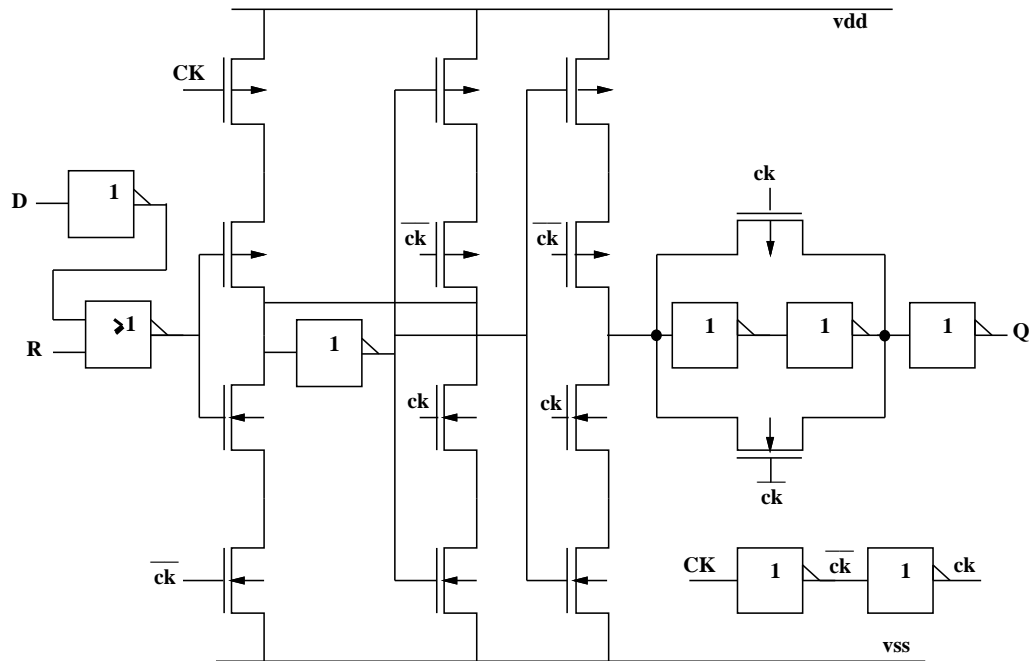
Timing parameters:

Parameter	From	To	Min	Typ	Max	Unit
T_{PLH}	CK	Q	1.4	2.1	3.2	ns
T_{PHL}	CK	Q	1.3	1.9	2.9	ns
ΔT_{PLH}	CK	Q	-	-	1.1	ns/pF
ΔT_{PHL}	CK	Q	-	-	0.8	ns/pF
T_{su}	D,R	CK	-	-	1.4	ns
T_{hold}	D,R	CK	-	-	0.9	ns
C_{in}	D,R	vss	-	0.12	0.18	pF
C_{in}	CK	vss	-	0.18	0.27	pF

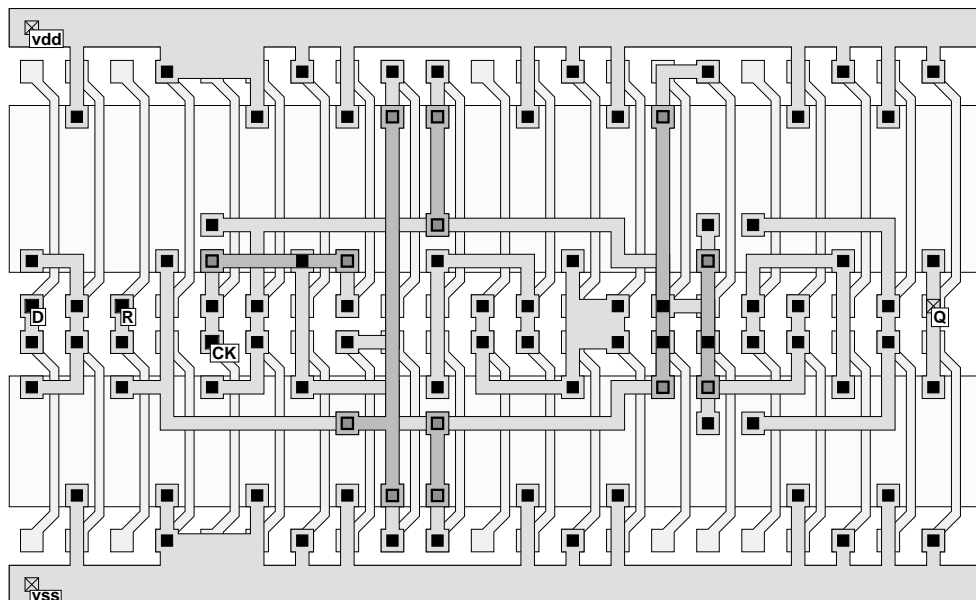
Equivalent chip area: 21

Fanout: 3.0pF

circuit:



layout:

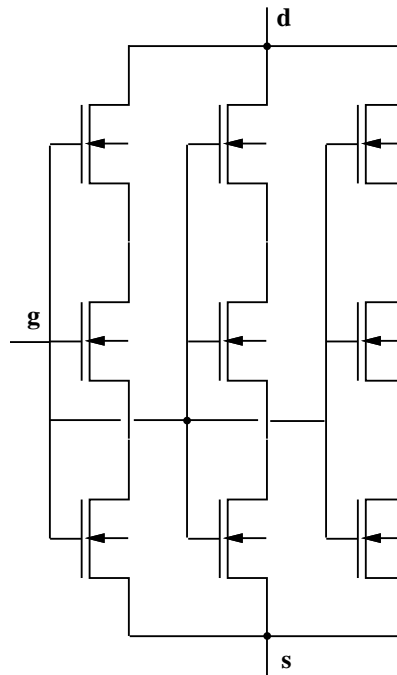


13.4 analib8_93: the analog library

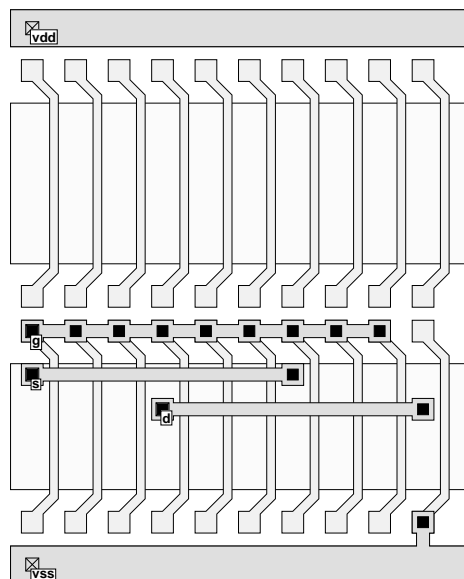
13.4.1 NMOS Compound transistor ln3x3

The NMOS Compoundtransistor ln3x3 consists of 9 NMOS transistors which are connected in a 3x3 matrix.

Circuit:



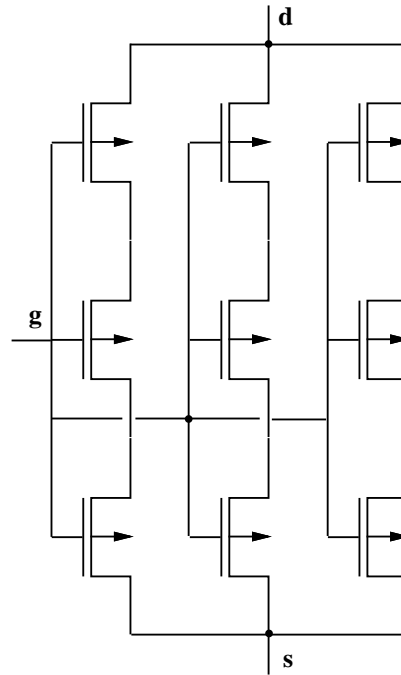
Layout:



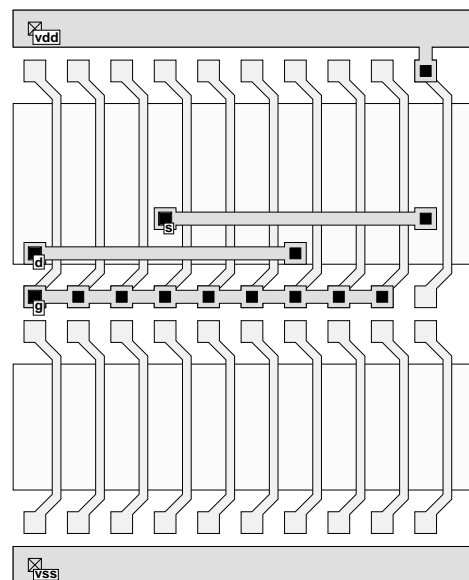
13.4.2 PMOS Compoundtransistor lp3x3

The PMOS Compoundtransistor lp3x3 consists of 9 PMOS transistors which are connected in a 3x3 matrix.

Circuit:



Layout:

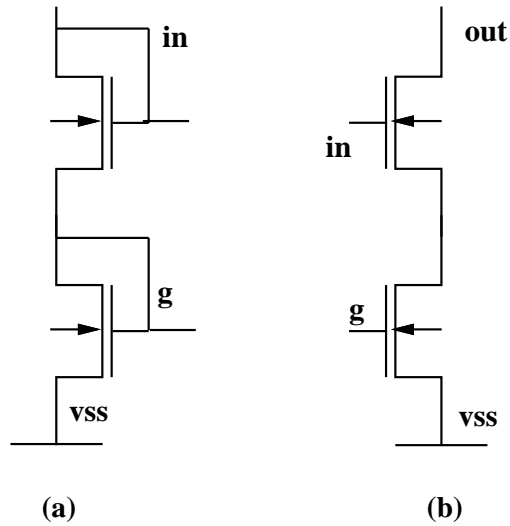


13.4.3 NMOS mirrors mir_nin en mir_nout

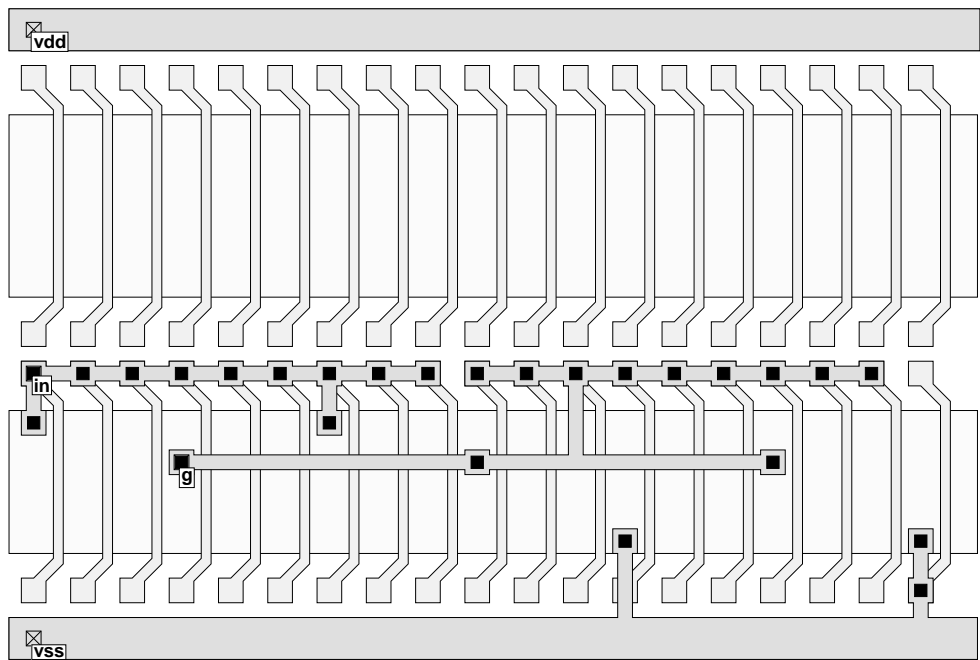
The MNOS mirrors are building blocks for a cascoded current mirror. It consists of the input mir_nin and the output mir_nout. In a mirror, both can be repeated a number of times to achieve the entire circuit.

Circuit:

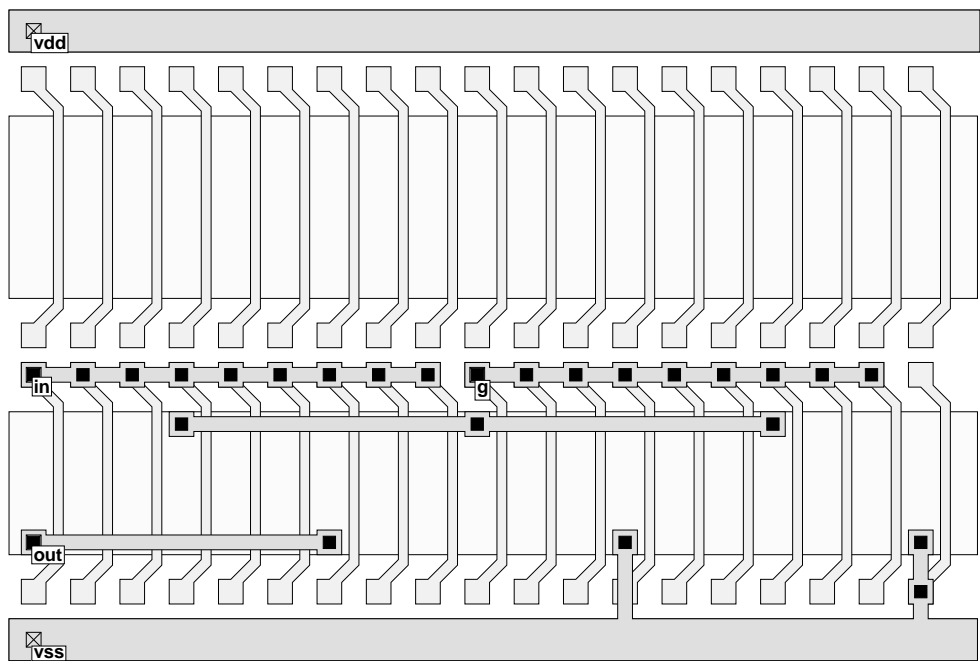
(a) mir_nin (b) mir_nout



Layout mir_nin:



Layout mir_nout:

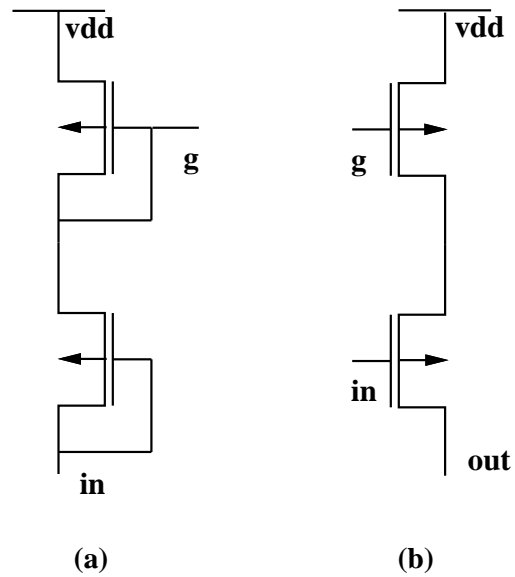


13.4.4 PMOS mirrors mir_pin and mir_pout

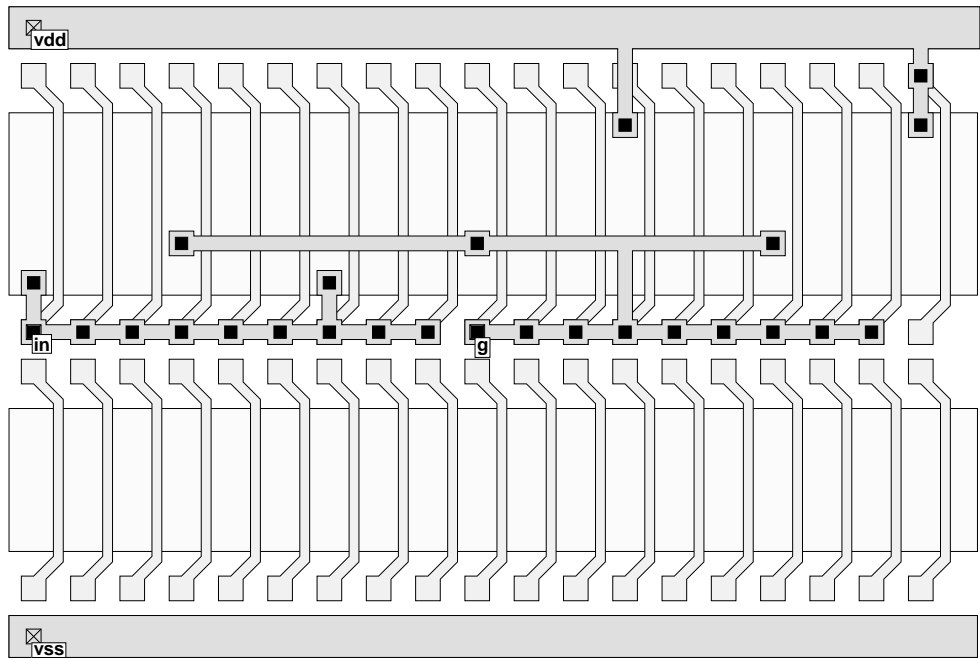
The PMOS mirrors are building blocks for a cascoded current mirror. It consists of the input mir_pin and the output mir_pout. In a mirror, both can be repeated a number of times to achieve the entire circuit.

Circuit:

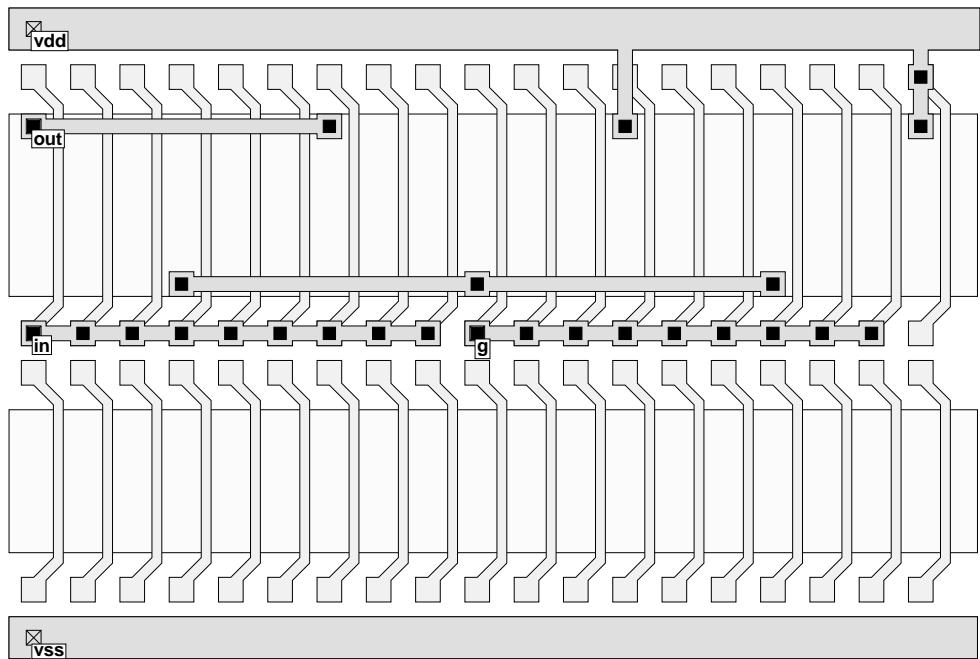
(a) mir_pin (b) mir_pout



Layout mir_pin:



Layout mir_pout:

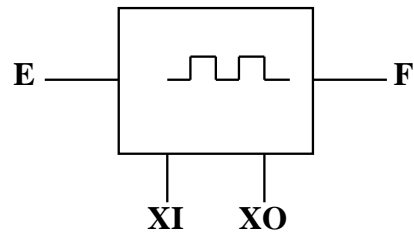


13.4.5 osc10

Function: Crystal oscillator with enable

Terminals: (E, F, XI, XO, vss, vdd)

Symbol:



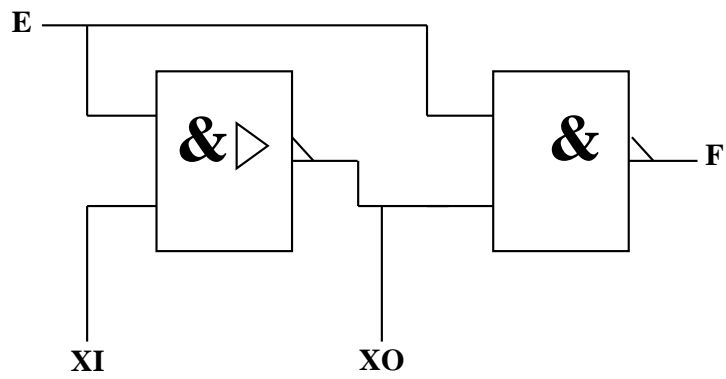
Frequency range: 1MHz to 20MHz

Equivalent chip area: 16

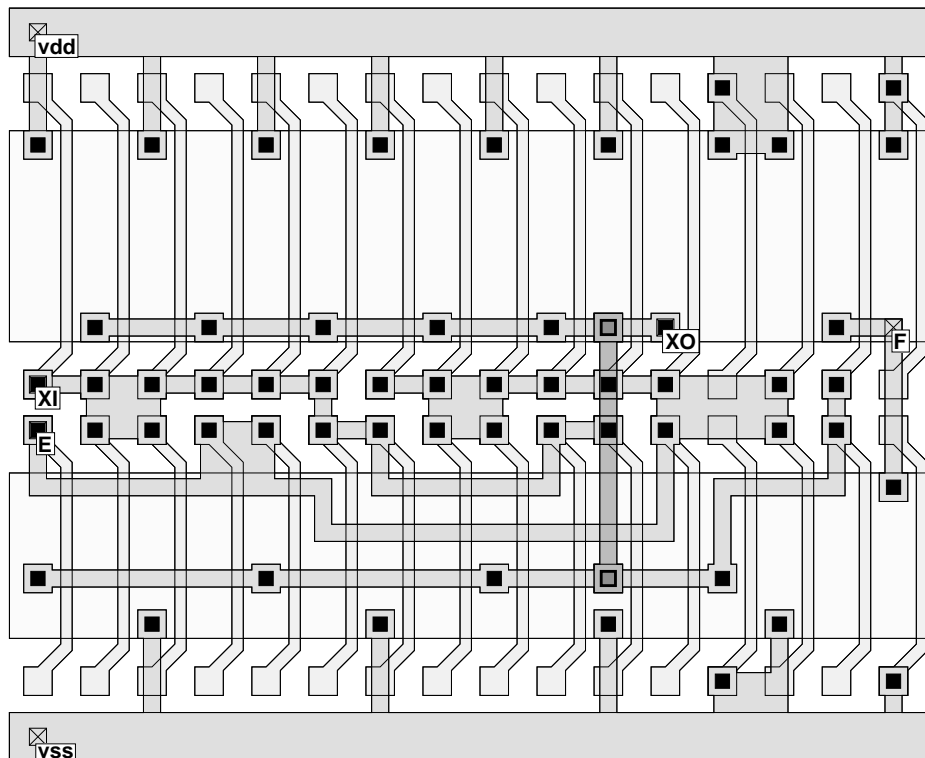
Description:

XI and XO are the connections for the crystal. XI must be buffered using an input buffer, X) should not be buffered. If E (enable) is high, the oscillator is active, If E is low, the output F is high.

circuit:



layout:



13.5 bonding8_93: the bonding patterns

13.5.1 bond_leer

Function: Bonding cell for small SoG circuits

Terminals: (vss, bf1, bf2, bf3, vdd, bf4, bf5, bf6)

Equivalent chip area: 800

Maximum size of embedded circuit: 350

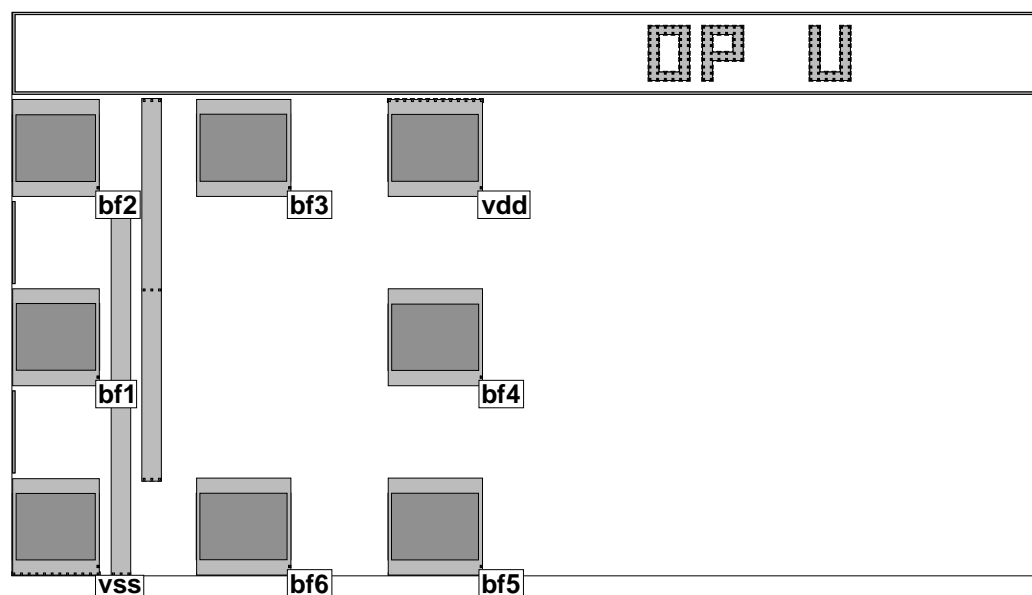
Pins:

- 2 power pins (vss, vdd), which are already pre-connected to the internal power rails.
- 6 user-configurable unbuffered pins.

Protection:

This bonding pattern is intended for use with a simple 8-pin fixed-probe card. The pins are *not* buffered and *not* protected. Therefore the needles of the fixed-probe card should be protected by diodes, otherwise the circuit blows up. With proper diode protection, static charge is no problem. The output drive of any arbitrary gate in the library turned out to be sufficient to drive the input of the tester via a 1 meter cable.

Layout:



13.5.2 bond_bar

Function: virtual bonding pattern for one quarter of the chip.

Terminals: (t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23, t24, t25, t26, t27, t28, t29, t30, t31, t32)

Equivalent chip area: 25000

Maximum size of the circuit: 20000

Pins:

- 2 power pins (vss, vdd)
- 2 power buffers
- 32 user-configurable pins (buffered input, buffered output, buffered bi-directional, unbuffered direct).

Four cells of this type can be placed on one chip. The pin positions connect to an outer ring of interconnect wires, which connects the pins to the bonding pads on the border of the chip. The pins of one cell bond_bar are spread out over the 144 pins in such a way that they connect to every 4th bonding pad. Each of the 4 virtual chips has its own vdd power supply, which is also used to drive the buffers in the bonding pads. In this way the 4 circuits on the chip are completely independent. All circuits share one common vss ground supply connection. For larger circuits, multiple instances of bond_bar can be used.

Layout:



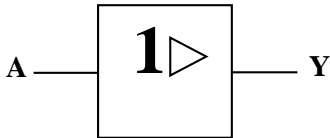
13.6 exlib8_93: the extended library

13.6.1 buf20

Function: Buffer (2x-drive)

Terminals: (A, Y, vss, vdd)

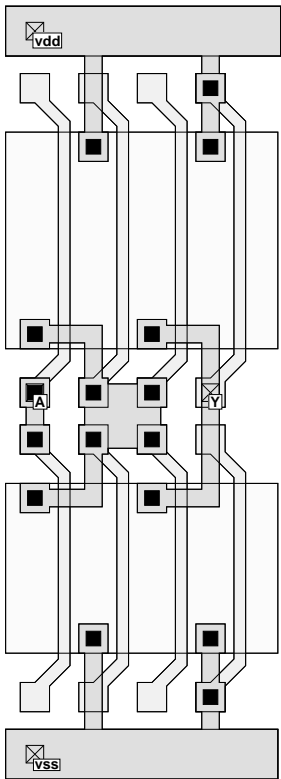
IEC-symbol:



Function table:

A	Y
L	L
H	H

layout:

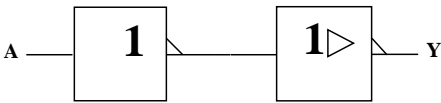


Propagation and load dependent delays:

Parameter	Van	Naar	Typ	Eenheid
T_{PLH}	A	Y	0.6	ns
T_{PHL}	A	Y	0.7	ns
ΔT_{PLH}	A	Y	1.1	ns/pF
ΔT_{PHL}	A	Y	1.0	ns/pF
C_{in}	A	vss	0.12	pF

Equivalent chip area: 4

circuit:

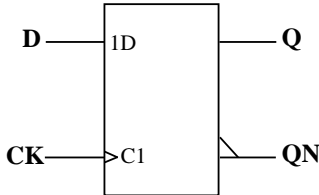


13.6.2 dfn102

Function: D-type positive edge triggered flipflop

Terminals: (D, CK, Q, QN, vss, vdd)

IEC-symbol:



Function table:

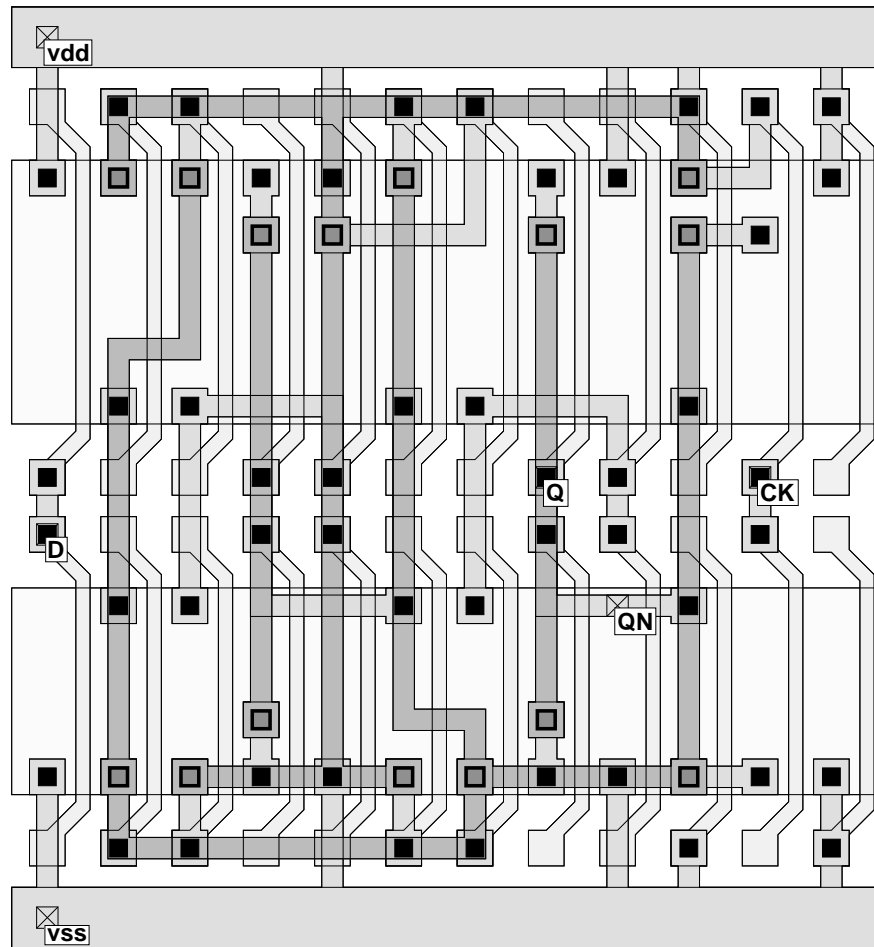
D	CK	Q	QN
L	↑	L	H
H	↑	H	L

Propagation and load dependent delays:

Parameter	From	To	Typ	Unit
T_{PLH}	CK	Q	1.8	ns
T_{PHL}	CK	Q	2.3	ns
T_{PLH}	CK	QN	1.9	ns
T_{PHL}	CK	QN	2.3	ns
ΔT_{PLH}	CK	any	8.0	ns/pF
ΔT_{PHL}	CK	any	7.0	ns/pF
C_{in}	D	vss	0.12	pF
C_{in}	CK	vss	0.43	pF

Equivalent chip area: 12

layout:

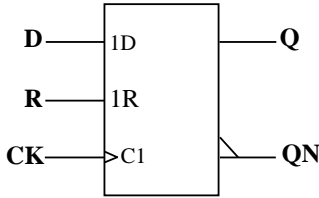


13.6.3 dfr112

Function: D-type positive edge triggered flipflop with synchronous reset

Terminals: (D, R, CK, Q, QN, vss, vdd)

IEC-symbol:



Function table:

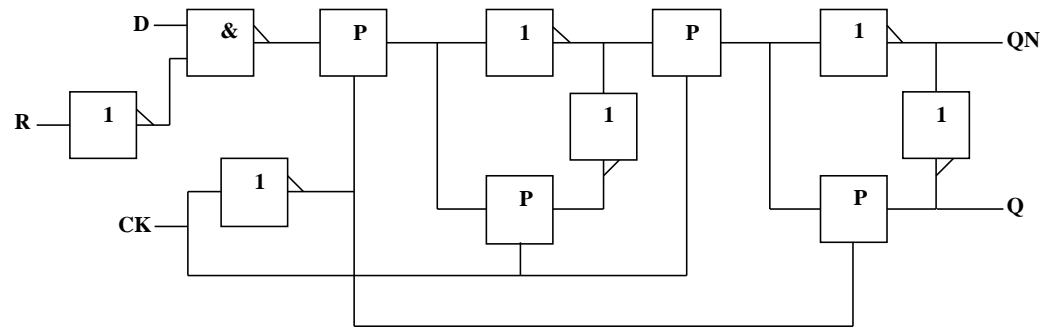
R	D	CK	Q	QN
L	L	↑	L	H
L	H	↑	H	L
H	-	↑	L	H

Propagation and load dependent delays:

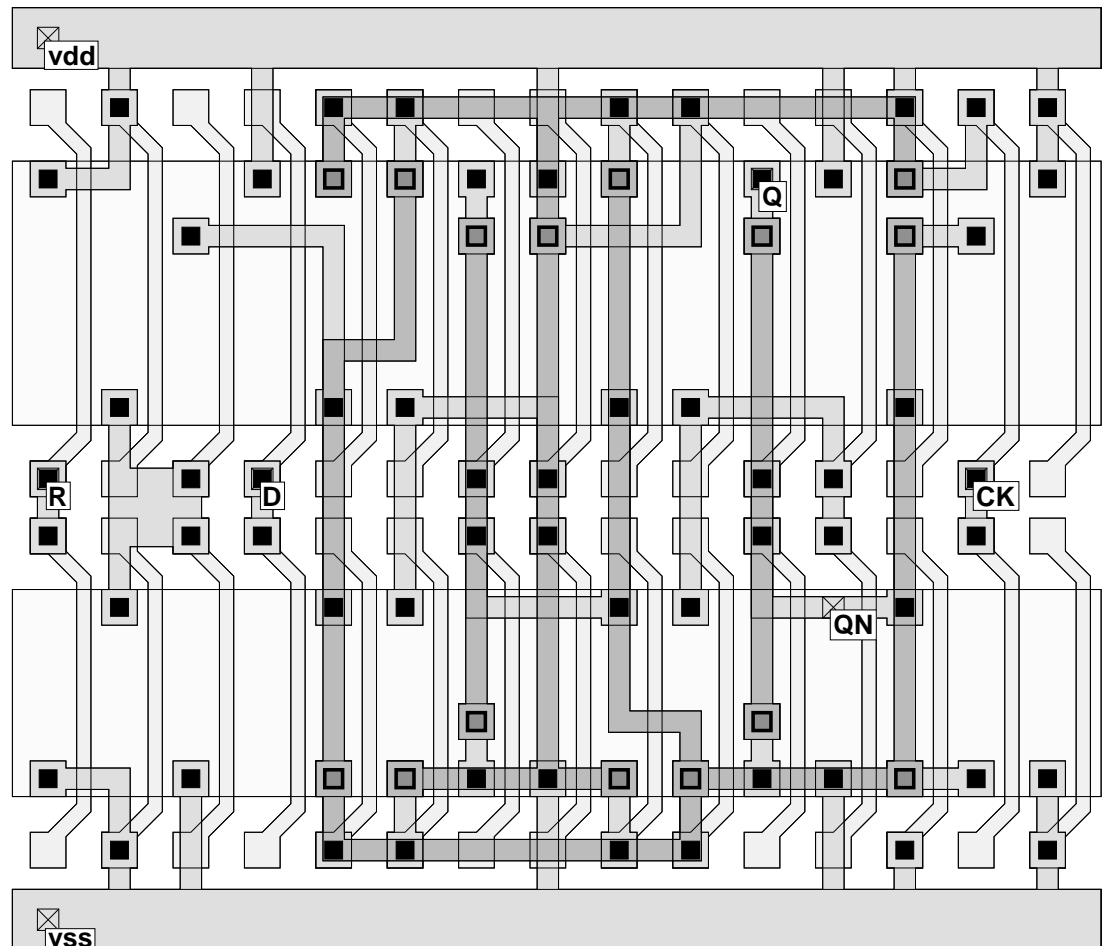
Parameter	From	To	Typ	Unit
T_{PLH}	CK	Q	1.8	ns
T_{PHL}	CK	Q	2.3	ns
T_{PLH}	CK	QN	1.9	ns
T_{PHL}	CK	QN	2.3	ns
ΔT_{PLH}	CK	any	8.0	ns/pF
ΔT_{PHL}	CK	any	7.0	ns/pF
C_{in}	D,R	vss	0.12	pF
C_{in}	CK	vss	0.43	pF

Equivalent chip area: 15

circuit:



layout:



13.7 The file 'image.seadif'

Process information is stored in the NELSYS system in the directory '\$ICD-PATH/lib/process'. The OCEAN tools read the information about the image from a file called 'image.seadif', which is present in '\$OCEANPATH/celllibs/\$OCEANPROCESS'. In some cases you might want to modify this file to change the behaviour of the tools. If you copy an 'image.seadif' in your project directory, it will automatically be read by all tools.

The syntax of the image description file 'image.seadif' is lisp-like and easy to interpret. Below we include the 'image.seadif' file of the *fishbone* image.

```
/*
 *
 * This is the file 'image.seadif'
 * Contents: Image description for the the 'fishbone' image. The fishbone
 *           image was developed at Delft University of Technology by
 *           Patrick Groeneveld and Paul Stravers. It requires the
 *           c3tu process, which is a derivative of the 1.6 micron C3DM
 *           process from Philips.
 * Purpose:  All OCEAN tools read this file to get process and image
 *           specific information. This file is read by fish, sea nelsea,
 *           ghoti, madonna, trout and seadali. You may edit this file to
 *           change the behaviour of (for instance) the router. In that
 *           case you can copy a version of this file into your process
 *           directory. All OCEAN tools will read your local image
 *           description file in that case.
 * Created:  by Patrick Groeneveld and Paul Stravers
 *           march 1993
 *
 * (c) The OCEAN/NELSYS Sea-of-Gates design system
 * Delft University of Technology
 *
 */
```

(Seadif "Image description file for fishbone image"

```
/*
 * Set the size of an empty image if you press 'fish'
 * for fishbone 25 times 1 is nice
 */
(ChipDescription chip
  (ChipSize 25 1)
)

(ImageDescription "fishbone"      /* name is important */
  (Technology "fishbone"        /* nelsis technology name */
    (DesignRules
      /*
       * Specify data about routing layers. First metal layer
       * (which is closest to the solicon) has index 0:
       *   -1 <- polysilicon/diffusion
       *   0  <- metal 1
       *   1  <- metal 2
       *   2  <- metal 3
       *   etc.
       * Take as many layers as you want or have.
       * Layer -1 contains all feautues of the iamge which cannot
```

```

    * be edited by the user
    */
    (NumberOfLayers 2)
    /* set wire width of the layer in lambda */
    (WireWidth 0 12)
    (WireWidth 1 12)
    /* specify which mask corresponds to each layer */
    /* every layer MUST have one unique mask assigned */
    (WireMaskName 0 "in")      /* layer0 name = in */
    (WireMaskName 1 "ins")     /* layer1 name = ins */
    /* dummy layer for markers */
    (DummyMaskName "bb")       /* bb mask as no-op */
    /* set style of the layers for trout, must be alternating */
    (WireOrient 0 horizontal)
    (WireOrient 1 vertical)

    /*
    * Declare the vias. For each via there must be an
    * instance containing the mask pattern. Between the
    * metal layers there is one via per adjacent pair of
    * layers. Between metal 1 and the silicon (poly,
    * diffusion, etc) there may be more than one type of via.
    */
    (ViaCellName 0 1 "Via_in_ins") /* between metal1 and metal 2 */
    (ViaCellName -1 0 "Via_ps_in") /* polysilicon and metal 1 */
    (ViaCellName -1 0 "Via_on_in") /* n-diffusion and metal 1 */
    (ViaCellName -1 0 "Via_op_in") /* p-diffusion and metal 1 */

    /*
    * specify the mask name associated with each via. In this way
    * fish can map a mask pattern to the proper via. Any
    * of the contacts from metal1 down to the image is
    * translated to the proper mask pattern, using the feed
    * statements in the GridConnectList
    */
    (ViaMaskName 0 1 "cos")      /* between metal1 and metal 2 */
    (ViaMaskName -1 0 "cps")     /* polysilicon and metal 1 */
    (ViaMaskName -1 0 "con")     /* n-diffusion and metal 1 */
    (ViaMaskName -1 0 "cop")     /* p-diffusion and metal 1 */
)

/*
* set the spice parameters, which allow ghoti to collapse
* parallel transistors into one transistor. (not so relevant)
*/
(SpiceParameters
  (Model "nenh"
    (Parameter "ld" "0.325e-6") /* ghoti -r needs this parameter */
    (Parameter "vto" "0.7"))
  (Model "penh"
    (Parameter "ld" "0.300e-6")
    (Parameter "vto" "-1.1"))
)

/*
* The next block describes how the grid of a single core cell
* looks like
*/
(GridImage

```

```

/*
 * declaration of the image grid.
 * this image is a rectangular grid which is repeated
 *
 * Set the size of a single grid cell in grid points
 * in fishbone this is 1 (x) by 28 (y), which means that
 * the gridpositions range from 0,0 to 0,2
 */
(GridSize 1 28
/*
 * Specify how the gridpoints map to the layout coordinates
 * In this case, for instance, the grid point (0, 1) will
 * map to point (26, 64) lambda in the actual image. In this
 * way it is possible to specify a non-regular grid.
 */
(GridMapping horizontal /* 1 point */
 44)
(GridMapping vertical /* contains 28 points */
 17 56 96 128 160 192 232 264 304 336 368 400 432 472
 511 550 590 622 654 686 718 758 790 830 862 894 926 966)

/*
 * tearlines of the image (currently disabled, not so relevant)
 *
(TearLine vertical 2
 0 13)
(TearLine horizontal 1
 0)
*/

/*
 * Specify the implicit power lines which are present in
 * the image. This is used by the router.
 * format:
 * (PowerLine <orient> <ident> <layer_no> <row/column number>) */
(PowerLine horizontal vss 0 0 )
(PowerLine horizontal vdd 0 14)

/* overlap of the repeated image in x and y direction: */
(ImageOverlap 0 1)
)

(Axis
/*
 * Mirror axis information for the detailed placer. Format is
 * (MirrorAxis x1 x2 y1 y2) where (x1,y1) and (x2,y2) are two
 * coordinates thru which the mirror axis runs. NOTE: because
 * a mirror axis can run just IN BETWEEN two gridpoints, the
 * arguments of MirrorAxis specify HALF GRIDPOINT UNITS. E.g.
 * a mirror axis running thru the gridpoints (1,1) and (5,5)
 * would be specified as (MirrorAxis 2 10 2 10).
 * used by madonna.
 */
(MirrorAxis 0 1 28 28) /* mirror around vdd */
(MirrorAxis 0 1 0 0 ) /* mirror around vss */
)

```

```

/*
 * The next block specifies the connections/feeds
 * between grid points.
 */
(GridConnectList
/*
 * Default vias are allowed at EACH grid point between metal
 * layers (layer no >= 0) and at NO gridpoint to the core (from
 * layer 0 to -1). Using the 'block' statement it is possible to
 * remove edges from the graph and therefore also possible vias.
 * In the image of the example no 'cos' vias are allowed at
 * some positions.
 */
/* means: no connection between pos (0,1,0) and (0,1,1)
 * = no via allowed there */
(Block 0 1 0 0 1 1)
(Block 0 6 0 0 6 1) (Block 0 7 0 0 7 1)
(Block 0 13 0 0 13 1)
(Block 0 15 0 0 15 1)
(Block 0 21 0 0 21 1) (Block 0 22 0 0 22 1)
(Block 0 27 0 0 27 1)

/*
 * Declare the internal feeds of the image (e.g. feeds
 * through a poly wire). This declaration also implies
 * that it is possible to make a via at a certain grid
 * position. Therefore the cell name of the required via
 * must be given at each feed position. The via cell name
 * must be declared previously in the technology block
 * using the 'ViaMaskName' statement. We recognize two
 * types of feeds: UniversalFeed and RestrictedFeed. For
 * fish there is no difference, but for the router there
 * is. The router can use universal feeds for arbitrary
 * wire segments. Restricted feeds have a limited use:
 * they can only indicate that a set of gridpoints (in the
 * image = layer -1) are electrically equivalent. In the
 * example image the the poly gates can serve as universal
 * feeds, while the diffusion source and drain cannot. A
 * feed position can also point to a position outside the
 * size of the image cell. Fish does not use the feed
 * information, but it needs a via cell name for each
 * position where a connection to layer -1 can be made.
 * It is allowed to add dummy feeds: e.g.
 * (UniversalFeed (Feed "Via_ps_in" 0 1))
 * indicates that at 0,1 a via to core can be placed, with
 * name Via_ps_in. ExternalFeed - if this exists then
 * this feed has an extension to the neighbor cluster in
 * this direction ("hor" or "ver")
 * BusFeed - power or ground net
 *
 */
/* e.g. points (0,1) and (0,6) are equivalent */
(UniversalFeed (Feed "Via_ps_in" 0 1) (Feed "Via_ps_in" 0 6))
(UniversalFeed (Feed "Via_ps_in" 0 7) (Feed "Via_ps_in" 0 13))
(UniversalFeed (Feed "Via_ps_in" 0 15) (Feed "Via_ps_in" 0 21))
(UniversalFeed (Feed "Via_ps_in" 0 22) (Feed "Via_ps_in" 0 27))

(RestrictedFeed (Feed "Via_on_in" 0 2) (Feed "Via_on_in" 0 3)
 (Feed "Via_on_in" 0 4) (Feed "Via_on_in" 0 5))

```

```

(RestrictedFeed (Feed "Via_op_in" 0 8) (Feed "Via_op_in" 0 9)
  (Feed "Via_op_in" 0 10) (Feed "Via_op_in" 0 11)
  (Feed "Via_op_in" 0 12))
(RestrictedFeed (Feed "Via_op_in" 0 16) (Feed "Via_op_in" 0 17)
  (Feed "Via_op_in" 0 18) (Feed "Via_op_in" 0 19)
  (Feed "Via_op_in" 0 20))
(RestrictedFeed (Feed "Via_on_in" 0 23) (Feed "Via_on_in" 0 24)
  (Feed "Via_on_in" 0 25) (Feed "Via_on_in" 0 26))

(RestrictedFeed (Feed "Via_op_in" 0 0) (ExternalFeed "hor"))
(RestrictedFeed (Feed "Via_on_in" 0 14) (ExternalFeed "hor"))

(BusFeed (Feed "Via_op_in" 0 0) (ExternalFeed "hor"))
(BusFeed (Feed "Via_on_in" 0 14) (ExternalFeed "hor"))
)

/*
 * The cost parameters to steer the router.
 */
(GridCostList
  /*
   * The format is:
   *   <cost> <vector> <startpoint_range> <endpoint_range>
   *
   * GridCost specifies the cost of a metal wire or a via
   * The offset <vector> must be in the unity coordinate system:
   *   1 0 0 = horizontal metal wire to the right
   *  -1 0 0 = horizontal metal wire to the left
   *   0 0 1 = a via to the layer above this layer
   * example:
   *   (GridCost 1 1 0 0 0 0 0 0 27 0)
   * gives offset vector (1 0 0) = L cost 1 over range
   * (0 0 0) to (0 27 0) = entire layer 0
   *
   * FeedCost has the same syntax format, but it specified the
   * cost of a feed through the image (a non-metal layer) PLUS
   * the cost of the two vias associated with that feed.
   * The range is implicitly over the image layer (-1).
   *
   * Tuning the costs may take some experimenting, some
   * thinking and just some common sense. If you need any help,
   * contact us at space-support-ewi@tudelft.nl
   */
  /* horizontal cost in layer 0 */
  (GridCost 1 1 0 0 0 0 0 0 27 0)
  (GridCost 1 -1 0 0 0 0 0 0 27 0)

  /* horizontal cost over vias in layer 0
   * (higher cost over poly vias) */
  (GridCost 3 1 0 0 0 0 0 0 1 0)
  (GridCost 3 -1 0 0 0 0 0 0 1 0)
  (GridCost 3 1 0 0 0 6 0 0 7 0)
  (GridCost 3 -1 0 0 0 6 0 0 7 0)
  (GridCost 3 1 0 0 0 13 0 0 15 0)
  (GridCost 3 -1 0 0 0 13 0 0 15 0)
  (GridCost 3 1 0 0 0 21 0 0 22 0)
  (GridCost 3 -1 0 0 0 21 0 0 22 0)
  (GridCost 3 1 0 0 0 27 0 0 27 0)
  (GridCost 3 -1 0 0 0 27 0 0 27 0)

```

```

/* vertical cost in layer 0 */
(GridCost 5 0 1 0 0 0 0 0 27 0)
(GridCost 5 0 -1 0 0 0 0 0 27 0)
/* vertical cost over vias in layer 0 */
(GridCost 3 0 1 0 0 0 0 0 1 0)
(GridCost 3 0 -1 0 0 0 0 0 2 0)
(GridCost 3 0 1 0 0 5 0 0 7 0)
(GridCost 3 0 -1 0 0 6 0 0 8 0)
(GridCost 3 0 1 0 0 12 0 0 15 0)
(GridCost 3 0 -1 0 0 13 0 0 16 0)
(GridCost 3 0 1 0 0 20 0 0 22 0)
(GridCost 3 0 -1 0 0 21 0 0 23 0)
(GridCost 3 0 1 0 0 26 0 0 27 0)
(GridCost 3 0 -1 0 0 27 0 0 27 0)
/* horizontal cost in layer 1 */
(GridCost 10 1 0 0 0 0 1 0 27 1)
(GridCost 10 -1 0 0 0 0 1 0 27 1)
/* vertical cost in layer 1 */
(GridCost 1 0 1 0 0 0 1 0 27 1)
(GridCost 1 0 -1 0 0 0 1 0 27 1)
/* cost of a via between layer 0 and layer 1 */
(GridCost 5 0 0 1 0 0 0 0 27 0)
(GridCost 5 0 0 -1 0 0 1 0 27 1)

/* cost of a tunnel in poly */
(FeedCost 1000 0 5 0 0 0 0 0 27 0)
(FeedCost 1000 0 -5 0 0 0 0 0 27 0)
(FeedCost 1000 0 6 0 0 0 0 0 27 0)
(FeedCost 1000 0 -6 0 0 0 0 0 27 0)
/* diff tunnels */
(FeedCost 80 0 1 0 0 0 0 0 27 0)
(FeedCost 80 0 -1 0 0 0 0 0 27 0)
(FeedCost 3 0 2 0 0 0 0 0 27 0)
(FeedCost 3 0 -2 0 0 0 0 0 27 0)
(FeedCost 3 0 3 0 0 0 0 0 27 0)
(FeedCost 3 0 -3 0 0 0 0 0 27 0)
(FeedCost 3 0 4 0 0 0 0 0 27 0)
(FeedCost 3 0 -4 0 0 0 0 0 27 0)
)
)

/*
 * describe the actual layout of the image cell
 */
(LayoutImage
/*
 * The cell is just imported as a model-call.
 */
(LayoutModelCall "fishbonec3tu2")
/*
 * the repetition distance in x is 40 lambda,
 * the repetition distance in y is 988 lambda.
 */
(LayoutImageRepetition 40 988)
)
)
)

```


Chapter 14

Acknowledgments

All software and design effort for the entire 'fishbone' Sea-of-Gates design trajectory (figure 1.1) was performed 'in house' at Delft University of Technology, department of Electrical Engineering. People from various groups within Electrical Engineering participated, making it a faculty-wide project. The people who contributed noticeably to the creation of the ocean system are listed below:

- SOFTWARE

Circuit tools 'sls', 'simeye': Arjan van Genderen

Extractor 'space': Nick van der Meijs and Simon de Graaf

Ocean placer 'Madonna': Paul Stravers and Irek Karkowski

Ocean router 'trout': Patrick Groeneveld

Layout editor 'seadali': Patrick Groeneveld (based on 'dali' by Pieter van der Wolf)

Purifiers 'fish' and 'ghoti': Patrick Groeneveld and Paul Stravers

Port to LINUX and sun4: Paul Stravers and Irek Karkowski

Database support (seadif): Paul Stravers and Patrick Groeneveld

Database support (Nelsis3): Nick van der Meijs, Simon de Graaf and Patrick Groeneveld

Database support (Nelsis4): Alfred van der Hoeven and Pieter van der Wolf

Nelsis backbone system: Peter Bingley, Hong Cai, Arjan van Genderen, Patrick Groeneveld, Alfred van der Hoeven, Fons de Lange, René van Leuken, Nick van der Meijs, Paul Stravers and Pieter van der Wolf.

Overall software coordination: Patrick Groeneveld

- THE CHIP

The 'fishbone' image: Paul Stravers and Patrick Groeneveld

Multi-project chip: Paul Stravers, Patrick Groeneveld, Anton Bakker, Antoon Frehe

Library design: Anton Bakker, Emile Hendriks, Peter van de Reest, Paul Stravers and Patrick Groeneveld

Design chip: Reinder Nouta, Paul Stravers, and Antoon Frehe

Processing empty wafers: Philips Research Laboratories (via Reinder Nouta)

Metalization (DIMES): Hugo Schellevis, prof. Cees Beenakker

Chip testing: Rob Bruinink and Hubert Schuit

- DOCUMENTATION

Manuals/tutorials: Paul Stravers, Patrick Groeneveld, Nick van der Meijs, Arjan van Genderen, Simon de Graaf, Anton Bakker, Emile Hendriks, and Reinder Nouta.

• AND FINALLY

The OP practicum commission: Emile Hendriks, Anton Bakker, Reinder Nouta, Chris Verhoeven,
Cees Beenakker and Patrick Groeneveld

Financial support by: Faculty of Electrical Engineering, TU Delft
PICO (wafer processing, test equipment)

Related projects: IOP Sea-of-Gates project
Royal Dutch Academy of Arts and Sciences

Special thanks to: Prof. Ralph Otten, Prof. Patrick Dewilde and many others.

Index

- addproj*, 40
- automatic
 - logic synthesis, → *kissis*
 - placement, → *madonna*
 - placement and routing in one step, 30, **55**
 - routing, → *trout*
- backing store, 48
- blif2sls*, 71
- buffer
 - buf20, 110
 - buf40, 90
- cannot find .dmrc error message, 40
- capitalization of cell name, 73
- channels, 54
- check nets, 26, 30, 31, **45**, 59
- circuit
 - purification, → *ghoti*
 - to layout extraction, → *space*
- creating
 - circuit descriptions, 3, 18
 - contacts, 21, **44**
 - layout, → *seadali*
 - projects, → *mkopr*
 - terminals in layout, 22, **46**
 - wires and boxes, 43
- csls*, 20
- .dalirc file, 47
- design rules, 43
 - checking, → *fish*
 - checking connectivity, 31, → *check nets*
 - file, → *image.seadif*
 - for contacts, **13**, 51
 - illegal vias, 52
 - substrate contacts, 62
- editing layout, → *seadali*
- extraction
 - layout, → *space*
 - sls-descriptions, → *xsls*
- fish*, 45, **51–52**
- Fish -i button, 41
- fishbone image, 7, **9–14**, 81
- fishbone library, 81–115
- flipflop
 - dfn10, 96
 - dfn102, 111
 - dfr11, 98
 - dfr112, 113
- FSM, 17
- gate array image, **7**, 81
- gate isolation technique, 13
- getepslay*, 33
- ghoti*, 22, **74**
- hardcopy, 42, 48
- icdman*, 6
- image, 7
 - displaying, 46
 - fishbone, 7, **9–14**, 81
 - gate array, **7**, 81
 - making an empty array, 52
 - octagon, **7**, 81
 - switch to another image, 32, **40**
- image.seadif* file, 51, **115**
- INSTALLATION file, 5
- instance, 42
- instances, 25, 42
- isolation transistors, **13**, 74
- kissis*, 28, **71–72**
- layout to circuit extraction, → *space*
- madonna
 - channels, 54
 - example, 25, 30
 - set placement box, 31, **54**
- madonna*, 42, **53–55**
- mkepr*, **40**, 82
- mkopr*, 18, **39**, 82
- mkpr*, 40
- Moore machine, 23
- nelsea*, 68
- ocean
 - copyrights, 5

- example of a chip, 4
 - installing the system, 5
 - system overview, 2–3
- OCEANPROCESS environment variable, 33, **40**
- octagon image, 7, 81
- permeability of a cell, 14
- playout*, 48
- plotting layout, → *getepslay*
- ports, → *terminals*
- power terminals, 59
- project, 18, **39**
 - copying, 69
 - creating, → *mkopr*
 - importing library cells, → *addproj, im-*
pcell
 - removing, 33
 - renaming, 69
- removing
 - cells, → *rmdb*
 - projects, 33
 - terminals, 46
 - wires, 44
- rmdb*, 33
- sea, 54, 59
- seadali
 - active masks, 43
 - adding instances, 42
 - adding terminals, 22, **46**
 - check nets button, 26, 30, 31, 45
 - customizing, 47
 - displaying the image, 46
 - dominant drawing style, 47
 - expanding, 41
 - hashed drawing style, 43, 44, **47**
 - image mode, 42, 43, 46, **47**
 - instance name, 42, **42**, 57
 - interrupting, 48
 - keyboard input, 48
 - madonna, 25
 - on-line help, 41
 - pictures, 48
 - setting visible masks, 46
 - starting, 21, **40**
 - sub terminals, 41
 - trout, 25, 45
 - undo, 44
 - wire button, 43
- seadali*, 21, 39–49
- contacts between layers, 44
- seadif* database, 67
- short circuits, 60
- simeye*, 20, **75–79**
 - .simeyerc* file, 78
- simulation
 - interactive, → *simeye*
 - logic level, 20
 - on three levels, 20, **28**, 75
 - spice-level, 21, 75
 - switch-level, 20, 75
- SIS*, 71, 72
- sls*
 - command file, 20, 73, **78**
 - language syntax, 18
 - stimuli file, → *sls command file*
- sls*, 75
- sls_prototypes*, 82
- space*, 22, **73**
- spice*, 21, 75
- stacked contacts, 13, 51
- terminals, 18
 - adding to layout, 22, **46**
 - removing, 46
- trout
 - decoupling capacitors, 31, **62**
 - example, 25, 31
 - handling of short circuits, 60
 - incomplete routing, 60
 - orphan instances, 59
- trout*, 45, **57–65**
- tutorial* tool, 18
- tutorials, 17–33
- xsls*, 22, 24