Alliance CAD System **V.R**

e-mail: `cao-vlsi@masi.ibp.fr`

Tutorial for the Data-Path Compiler

Jean-Paul Chaput

February 24, 1995

Contents

List of Figures

1 Introduction

This tutorial aims at teaching how to use **Alliance** CAD tools dedicated to the design of *data-paths*. These tools are the followings :

FpGen : *netlist* capture using textual mode, thanks to a set of predefined **C** functions.

DPR : Placer/Router associated with the *data-paths*.

From this point on, the environment variable **ALLIANCE_TOP** represents the root directory where the **Alliance** package is located.

2 *Data-Path* basics

2.1 Structure of a *data-path*

data-path compiler are tools which are dedicated to the design of microproccessor operative units. Inside a *data-path* the basic concept is no more the scalar cell (on 1 bit), but the **operator** carrying out a vectorial treatment (on **n bits**).

From a logical point of view, an **operator** is made of a repetition of **n** cells more or less identical, each of those executing a one bit treatment. It may also contain a cell dedicated to the amplification of control signals. Physically, each cell is stacked vertically and fill one *slice*. The last two *slices* are reserved to an amplification cell, if needed. It can be talked about either **operator** or **column**, this last term receiving a physical acceptance.

A *data-path* is made of **operators** stacked horizontally. It can be represented as a bidimentionnal table in which columns are called **operators** and rows *slices*.

Figure **??** shows *data-paths* structure.

2.2 Controls and datas signals

In a *data-path*, signals associated with **operator** input/output buses are called *datas* signals. They propagate horizontally through the *data-path*.

On the contrary, *control* signals are signals which command the **operators**, for instance, the bus select of a multiplexer (*ctrl_sel* of operator `DP_MUX2CS`). They propagate vertically inside a column.

Figure **??** shows the difference between *control terminals* and *data terminals.*

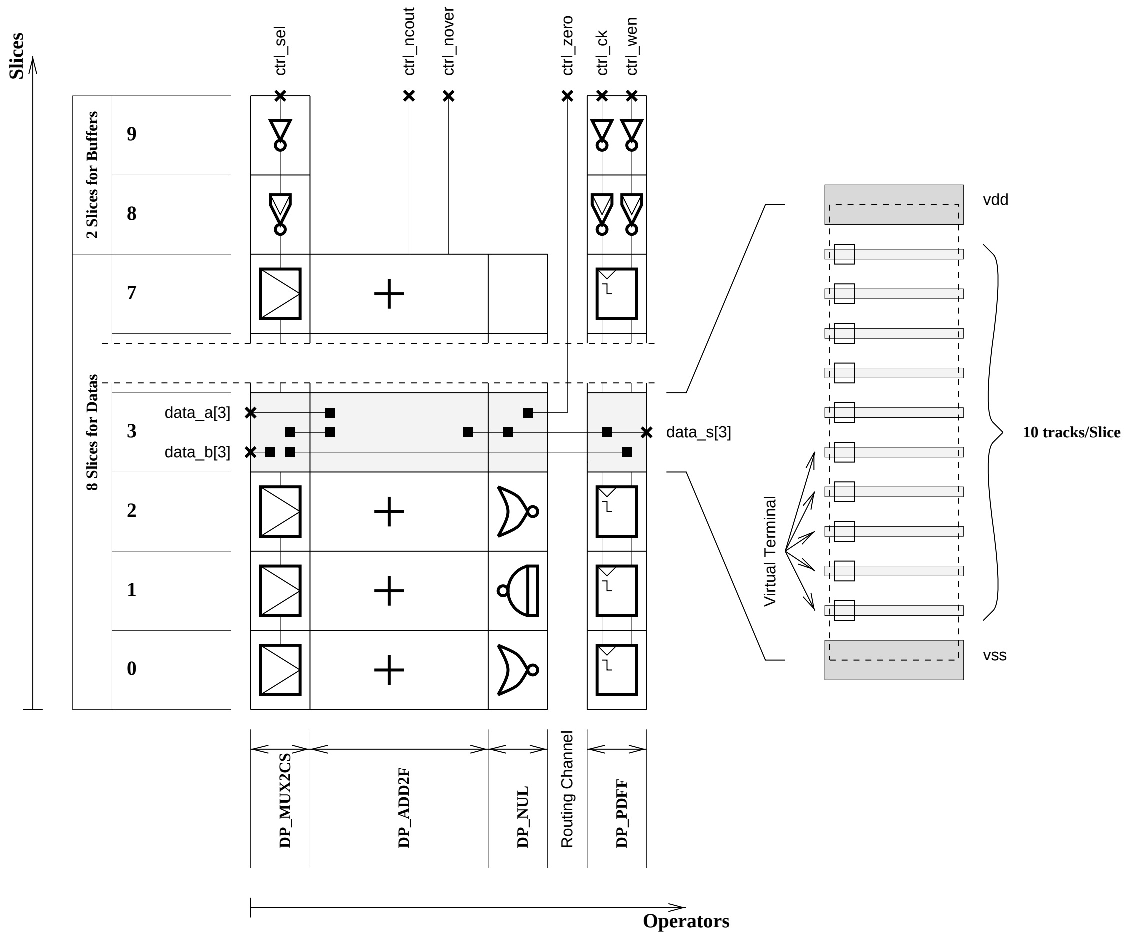


Figure 1: *data-paths structure.*

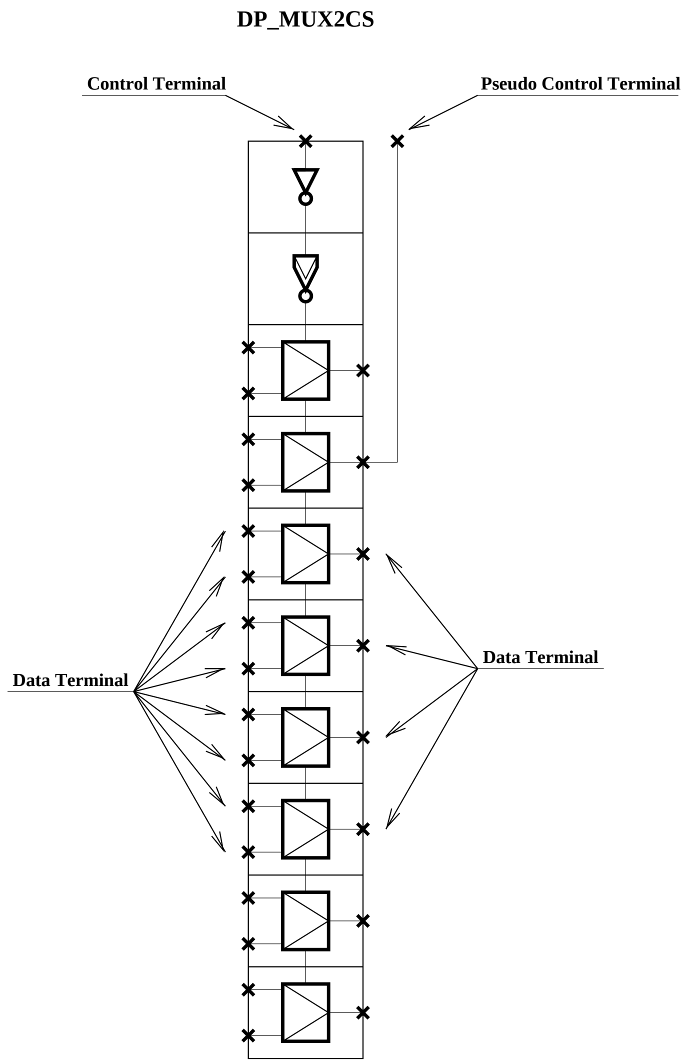


Figure 2: *Data terminals and control terminals.*

Data signals are routed horizontally, over *slices*, thanks to 10 routing *tracks* (`cf` figure **??**).

Remark :

Only data terminals can be set on any side of the *data-path*, using the `.dpr` file. Control terminals always appear on the **North** side of the *data-path*, at a predefined place.

Data terminals sets on the **North** side of the *data-path* are also called *Pseudo control terminals.*

2.3 Structural Parameters *Width* and *Slice*

In some cases we only wish to perform a treatment on a part of data bus. We have to use a smaller scaled **operator** in which empty *slices* are to be found. In order to place cells inside the **column**, we need two values :

Slice	:	The *slice* were the first cell is put in the **column**.
Width	:	Effective width of the **operator**.

Default values :

`DEFAULT_SLICE`	:	Null.
`DEFAULT_WIDTH`	:	Full width of the *data-path*. (set by the former `DP_DEFLOFIG` function call).

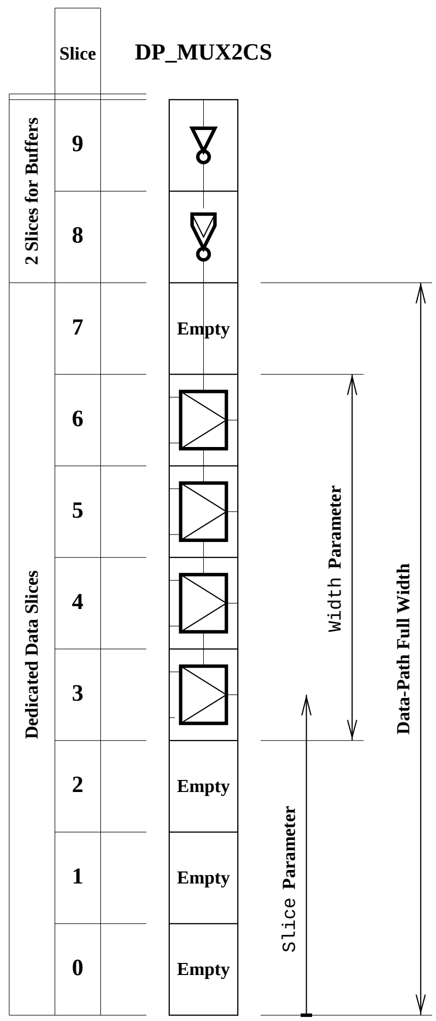


Figure 3: *Structural parameters Width and Slice*.

3 First example : `sample_dpt`

3.1 Presentation of the adder accumulator

The given example in the *data-path* tutorial is driven from the adder accumulator showed in the `addaccu` tutorial.

Differences from `addaccu` tutorial :

1. Data buses width are set from 4 to 8 bits.
2. In order to show some special features of the *data-path compiler*, a zero detect has been implemented.
3. The operator implementing sample register elements (`DP_PDFF`) obligatory provide a *Write ENable* terminal and dual outputs *q* and *nq*. So we add a *ctrl_wen* terminal to the circuit interface and a unused *data_u* signal which doesn't appear on the interface (for *nq*).
4. As the former operator, the fast adder generator (`DP_ADD2F`) obligatory provide *ncout* and *nover* outputs.

Figure **??** shows the whole architecture of the adder accumulator.

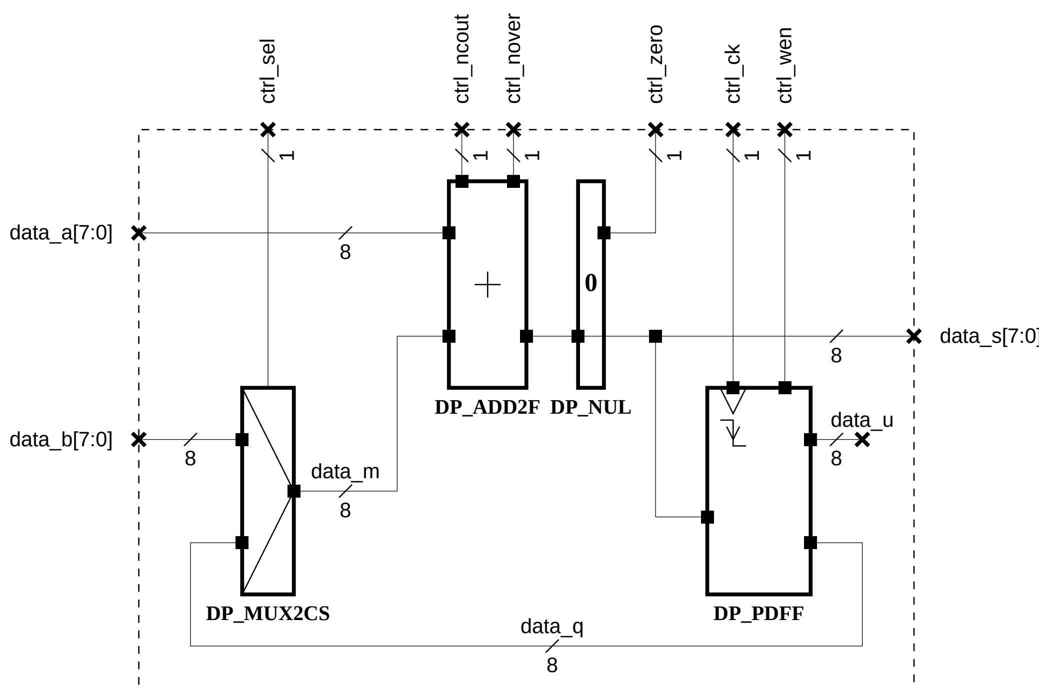


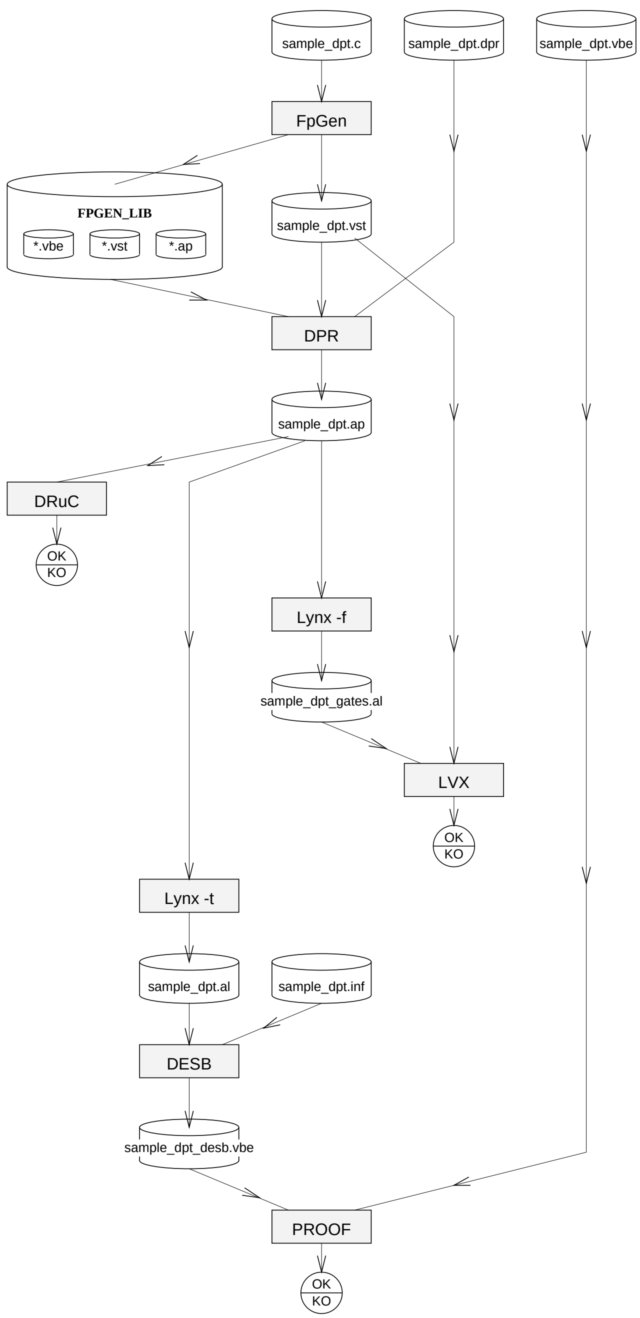
Figure 4: *Architecture of the adder accumulator.*

3.2 Methodology

As we wish not to present the whole design methodology of a circuit but only the part related to *data-path*, only a partial validation will be given.

Figure **??** describe the partial methodology.

Methology Schematic	Bourne Shell Commands




```
$ export FPGEN_LIB MBK_IN_LO MBK_OUT_LO
$ export MBK_IN_PH MBK_OUT_PH
$ export MBK_WORK_LIB MBK_CATA_LIB
$ export MBK_CATAL_NAME
$ FPGEN_LIB=./mclib
$ MBK_IN_LO=vst
$ MBK_OUT_LO=vst
$ MBK_IN_PH=ap
$ MBK_OUT_PH=ap
$ MBK_WORK_LIB=.
$ MBK_CATA_LIB=$TOP/cells/fitpath/fplib\
:$TOP/cells/rsa:$FPGEN_LIB
$ MBK_CATAL_NAME=CATAL

$ fpgen -v sample_dpt

$ dpr -V -p -r sample_dpt sample_dpt

$ druc sample_dpt

$ MBK_IN_LO=al
$ MBK_OUT_LO=al
$ lynx -v -f sample_dpt sample_dpt_gates

$ lvx vst fne sample_dpt sample_dpt_gates

$ lynx -v -t sample_dpt sample_dpt

$ desb sample_dpt sample_dpt_desb -v -i

$ proof -d sample_dpt sample_dpt_desb
```

Figure 5: *Partial methodology of validation.*

3.3 Environment set-up

Before running any tool of the **Alliance** CAD system, some environment variables must be set up[1].

1. Input/output formats :

```
$ MBK_IN_LO=vst
$ MBK_OUT_LO=vst
$ MBK_IN_PH=ap
$ MBK_OUT_PH=ap
$ export MBK_IN_LO MBK_OUT_LO MBK_IN_PH MBK_OUT_PH
```

2. Working directories :

```
$ MBK_WORK_LIB=.
$ FPGEN_LIB=./mclib
$ export MBK_WORK_LIB FPGEN_LIB
```

3. Cell library paths :

```
$ MBK_CATA_LIB=$ALLIANCE_TOP/cells/fitpath/fplib:$ALLIANCE_TOP/cells/rsa:$FPGEN_LIB
$ export MBK_CATA_LIB
```

3.4 FpGen : *netlist* capture

The *data-path netlist* is described thanks to a **C** source file which must contain the following sections :

1. Header files to include :

```
#include  <genlib.h>
#include  <fpgen.h>
```

2. Opening the *data-path* model (*netlist*) :

```
main()
{
  DP_DEFLOFIG( "sample_dpt", 8, LSB_INDEX_ZERO );
```

DP_DEFLOFIG parameters:

"sample_dpt"	:	**model name** of the *data-path*.
8	:	*data-path* bus wide.
LSB_INDEX_ZERO	:	Always set to this value.

[1] Commands are given in *Bourne Shell*.

3. Terminal declarations :

```
/* Control terminals declarations. */
DP_LOCON( "ctrl_sel"  , IN   , "ctrl_sel"   );
DP_LOCON( "ctrl_ck"   , IN   , "ctrl_ck"    );
DP_LOCON( "ctrl_wen"  , IN   , "ctrl_wen"   );
DP_LOCON( "ctrl_ncout",   OUT, "ctrl_ncout" );
DP_LOCON( "ctrl_nover",   OUT, "ctrl_nover" );
DP_LOCON( "ctrl_zero" ,   OUT, "ctrl_zero"  );

/* Data terminals declarations. */
DP_LOCON( "data_a[7:0]" , IN   , "data_a[7:0]" );
DP_LOCON( "data_b[7:0]" , IN   , "data_b[7:0]" );
DP_LOCON( "data_s[7:0]" , INOUT, "data_s[7:0]" );

/* Power supplies terminals. */
DP_LOCON( "vdd", IN   , "vdd" );
DP_LOCON( "vss", IN   , "vss" );
```

The first string is associated to a terminal model name and the second to the internal signal name to which it is connected. The behavior of this function is similar to the `LOCON` of **GenLib**.

4. Instanciation of the various *data-path* **operators** :

```
/* Multiplexer. */
DP_MUX2CS( "multiplexer", 8, 0,
           "ctrl_sel",
           "data_b[7:0]",
           "data_q[7:0]",
           "data_m[7:0]",
           EOL );

/* Fast Adder. */
DP_ADD2F( "adder",
          "data_a[7:0]",
          "data_m[7:0]",
          "ctrl_ncout",
          "ctrl_nover",
          "data_s[7:0]",
          EOL );

/* Zero Detect. */
DP_NUL( "zero", 8, 0,
        "data_s[7:0]",
        "ctrl_zero",
        EOL );

/* Register. */
DP_PDFF( "memory", 8, 0,
         "ctrl_wen",
         "ctrl_ck",
         "data_s[7:0]",
         "data_q[7:0]",
         "data_u[7:0]",  /* This bus is unused. */
         EOL );
```

5. Complete the model and save to the disk :

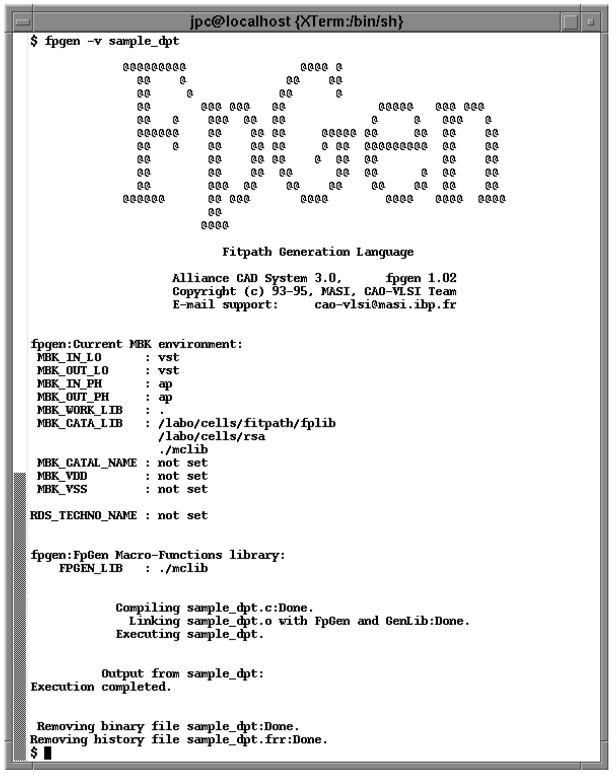
```
  DP_SAVLOFIG();

  /* A good C program must always terminate by an "exit(0)". */
  exit( 0 );
}
```

3.5 Generation of the *netlist*

After having writen the **C** source file which describe the *netlist* and correctly set up the environment, we just have to compile and execute the former file. The **FpGen** script file will do it for us, so lets type the command :

```
fpgen -v sample_dpt
```

Which normally gives you :



Back to `MBK_CATA_LIB` **and** `FPGEN_LIB`

Most of the **operators** avalaible in **FpGen** are built using the leaf cell library `$ALLIANCE_TOP/cells/fitpath/fplib`. However, some **operators** as `DP_ADD2F`, requires additionnal leaf cell libraries. These dedicated libraries are noticed in the **UNIX manual pages** associated with **operators**. In the case of `DP_ADD2F` the needed library is `$ALLIANCE_TOP/cells/rsa`.

So, `MBK_CATA_LIB` must contain :

`$ALLIANCE_TOP/cells/fitpath/fplib:$ALLIANCE_TOP/cells/rsa`

On the other hand, **FpGen** creates not only the *data-path netlist* but also the various views (*layout*, *behavioral*) of the **operators** These auxiliary views are stored in the library pointed out by the `FPGEN_LIB` environment variable. In our case, we choose `./mclib`.

3.6 Place and Route

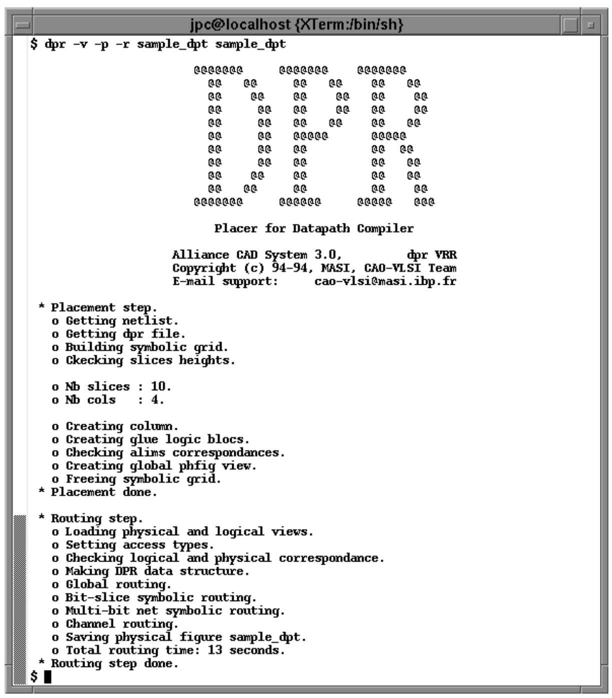
To make *data-path layout*, run the place and route tool **DPR**. The command is :

```
dpr -v -p -r sample_dpt sample_dpt
```

DPR command line arguments[2] :

-v	:	Verbose mode.
-p	:	Activate the placement step.
-r `sample_dpt`	:	Ask routing on *netlist* `sample_dpt.vst`.
`sample_dpt`	:	Name of the file that holds the place and routed *layout*.

This normally produces :



External terminal placement, `.dpr` file

In addition to the *netlist*, **DPR** attempts to load an optional file which contains some instruction about the way to place external data terminals [3]. The `.dpr` file allows the designer to choose on which side put the terminal, and for **East** and **West** sides, which *slice* and which *track*.

[2] For a more detailed description of **DPR** command line arguments, please refer to the **UNIX** on line manual.
[3] For the definitions of data terminals and control terminals, please refer to § **??**.

file `sample_dpt.dpr` used in our example :

```
#           Terminal :  Side  :  Slice  :  Track
DP_LOCON   ctrl_zero   NORTH    DEFAULT   DEFAULT
#
#
#           Terminal :  Side  :  Slice  :  Track
DP_LOCON  data_a[7:0]   WEST    DEFAULT   DEFAULT
DP_LOCON  data_b[7:0]   WEST    DEFAULT   DEFAULT
DP_LOCON  data_s[7:0]   EAST    DEFAULT   DEFAULT
#
#
# Number of vertical power refreshment.
DP_POWER  1  50
```

Back to `MBK_CATA_LIB`

Place and route tool **DPR** uses the various views of the operators formerly stored by **FpGen** inside the `FPGEN_LIB` library. This library access path must be present in `MBK_CATA_LIB`.

At the end, the complete `MBK_CATA_LIB` is:

`$ALLIANCE_TOP/cells/fitpath/fplib:$ALLIANCE_TOP/cells/rsa:$FPGEN_LIB`

3.7 Validation

3.7.1 Environnement set up and validation

All along the generation phasis we have used the *netlist* format `.vst`, which stands for **V***HDL* **ST***ructural*. For the validation phasis, we will use the `.al` format. The `.al` format is dedicated to the extractions steps because it holds, in addition to the logical informations (*netlist*), some information driven from the *layout*, such as routing wire capacitances.

Setting the new *netlist* format :

```
$ MBK_IN_LO=al
$ MBK_OUT_LO=al
```

Of course, the *layout* file format remains unchanged.

3.7.2 Routage Checking

The goal of this step is to ensure that the result of the routing phasis is correct. This will be done in three steps :

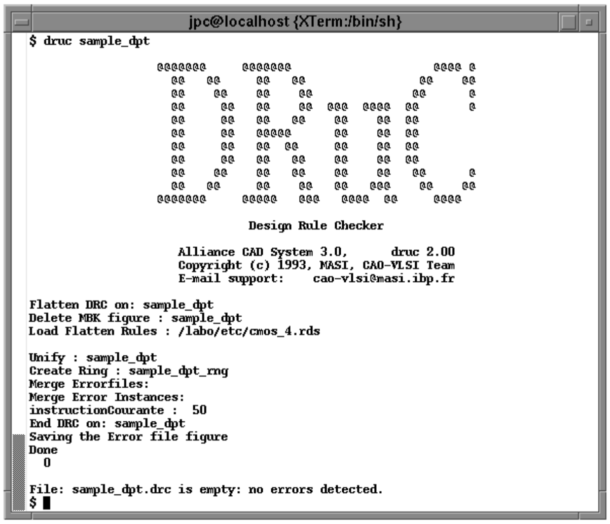
1. Design rule checking, using the *symbolic DRC* tool **DRuC**.
2. Extration of the gate *netlist* from the *layout* using **Lynx**.
3. Checking coherency between the extracted *netlist* (`.al` format) and the reference *netlist* (`.vst` format).

Hierarchy management

In some cases, the routing tool **DPR** is allowed to perform flattening in the *layout* view. As a consequence, *netlist* extracted from the *layout*may have a different hierarchy than the reference *netlist*. In order to process to the validations, we ask to the tools involved in the check to work at a gate level.

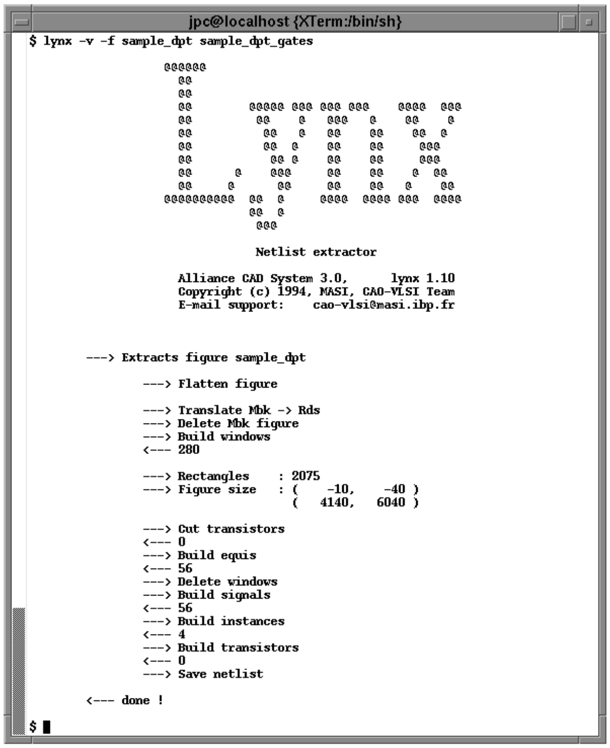
DRuC : *symbolic* **desing rules checking**

```
$ druc sample_dpt
```



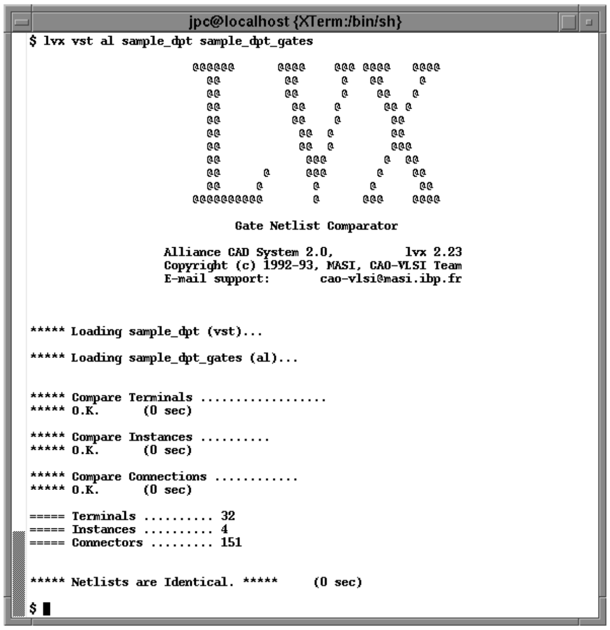
Lynx : *netlist* extraction

```
$ lynx -v -f sample_dpt sample_dpt_gates
```



LVX : *netlist* comparisons

```
$ lvx vst al sample_dpt sample_dpt_gates -f
```



3.7.3 Formal proof

The goal of this phasis is to ensure that the *data-path* we have designed here is consistent with its specification. In our case, the specification is the *VHDL behavioral* view `sample_dpt.vbe`. Three steps are needed :

1. **transistor** *netlist* extraction from the *layout* with the **Lynx** tool.
2. Restoration of a behavioral view `sample_dpt_desb.vbe` using the functional abstractor **desb**.
3. Formal proof between the behavioral model (`sample_dpt.vbe`) and the regenerated model (`sample_dpt_desb.vbe`) using the formal proofer tool **proof**.

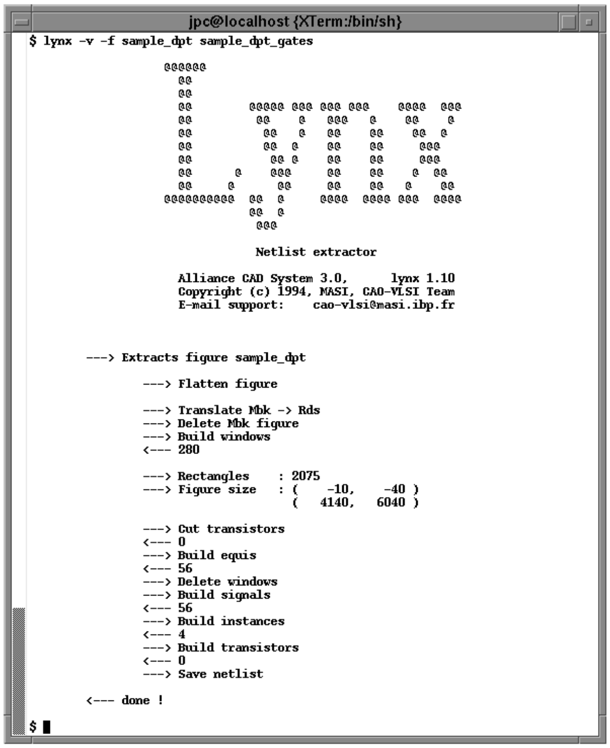
Register identification

The formal proof tool enforces that the memory elements of both behavioral views must have the same names. As these names have changed in the *layout*, we must rename them. Fortunatly, the functional abstractor **desb** allow us to do so, via an auxiliary file `.inf`.

The file `sample_dpt.inf` is supplied with the tutorial.

Lynx : transistor *netlist* extraction

```
$ lynx -v -t sample_dpt sample_dpt
```

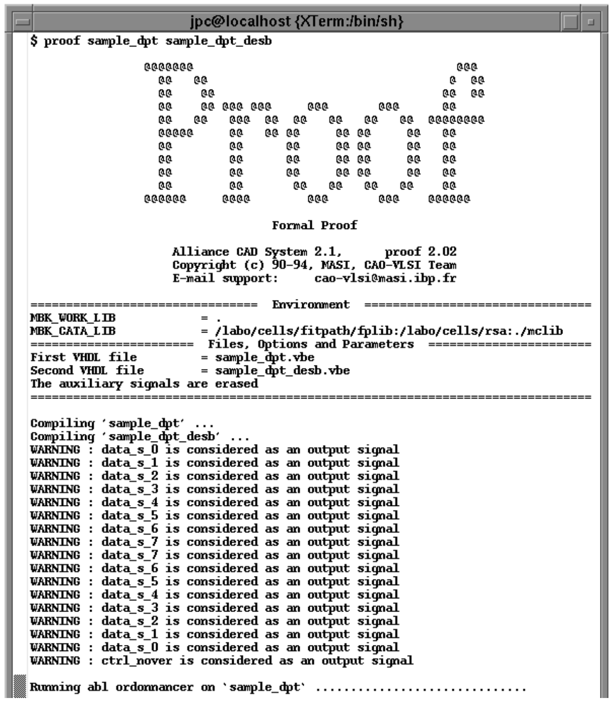


DESB : Functional abstraction

```
$ desb sample_dpt sample_dpt_desb -v -i
```

Proof : Formal proof

```
$ proof -d sample_dpt sample_dpt_desb
```



4 Advanced features

This section aims to present you some advanced features of **FpGen** and **DPR** tools. In the following examples, we do not describe the whole procedure given for `sample_dpt`. We will limits ourselves to show differences or novelties from the former methodology.

4.1 Placement

4.1.1 Initial placement

The initial placement of *data-path* operators is made from left to rigth, following the order of instanciation used in the file `sample_dpt.c`. We have, starting from the left, the followings operators :

1. Multiplexer generated by `DP_MUX2CS`.
2. Adder generated by `DP_ADD2F`.
3. Zero detect generated by `DP_NUL`.
4. D flip-flop generated by `DP_PDFF`.

Figure **??** shows initial placement.

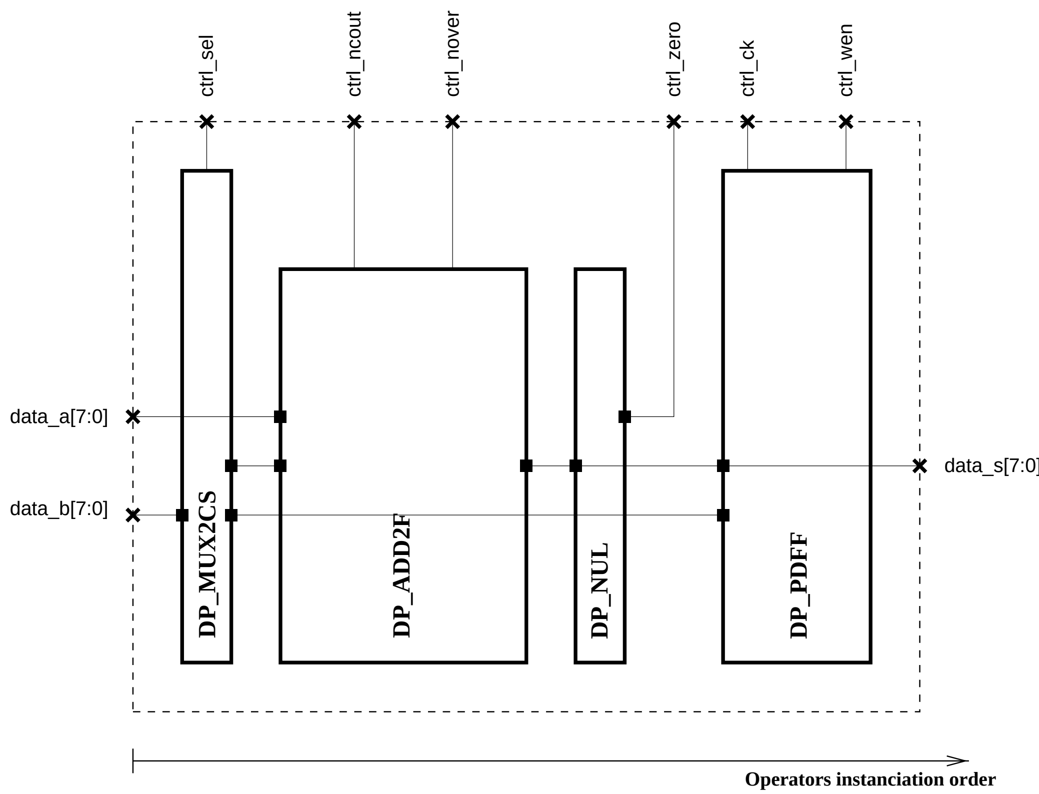


Figure 6: *Initial placement.*

The order of instanciation of the operators in C source file is meaningful. A **C** source file intended to **FpGen/DPR** that ignores it could produce non routable configurations.

To avoid this kind of problems, we just have to group together the instanciations of operators strongly connected.

4.1.2 Placement optimization

In the case of complex *data-path* where the designer does not want to check precisely the order of operators, placement optimizer inside the router tool **DPR** is avalaible. Roughly, the placement optimizer swap operators trying to reduce the total length of routing wires. This option is also interresting when the initial placement is a non routable configuration. The optimizer is able to detect such configuration in which case it attempts to reduce the track density to make possible routage.

File `place_dpt.*` gives an example of placement optimization.

Command to invoke **DPR** in optimization mode :

```
dpr -v -o -r -p place_dpt place_dpt
```

Arguments given to **DPR**[4] :

-o	:	Activate the placement optimizer.

4.2 Designing customized operators

In this part we present the features allowing the user to design his own *data-path* operators. To build his operators the user must resort to a dedicated leaf cell library : **DPLib**[5]. **DPLib** offers the same functionnalities than the *Standart Cell* library **SCLib**.

From a methodological point of view, using customized operators is unfolding itself into two distinct phases :

1. Making of the operator using one of the following three methods :
   (a) Explicit building of a column using **GenLib**. The model name must received a `"_us"` suffix.
   (b) Synthesis of a block, using **Logic**. The model name must received a `"_us"` suffix.
   (c) Buiding a sub-*data-path* using **FpGen**. The model name must not have a `"_us"`, `"_cl"` or `"_bk"` suffix.
2. Instanciation of the newly created operator inside the current *data-path* thanks to the `DP_IMPORT` function.

4.2.1 Environnement set up

We must add to `MBK_CATA_LIB` the access path to the **DPLib** library :

```
$ MBK_CATA_LIB=$ALLIANCE_TOP/cells/fitpath/dplib/ecpd10
$ MBK_CATA_LIB=$MBK_CATA_LIB:$ALLIANCE_TOP/cells/fitpath/fplib
$ MBK_CATA_LIB=$MBK_CATA_LIB:$ALLIANCE_TOP/cells/rsa
$ MBK_CATA_LIB=$MBK_CATA_LIB:$FPGEN_LIB
$ export MBK_CATA_LIB
```

4.2.2 Single Column Operator

As an example, let us replace the zero detect provided by **FpGen** (`DP_NUL` macro-function), by a designer build column.

The diagram of the zero detect we are going to build is shown in figure **??**. The files `usercol_dpt.*` gives an example of implementation.

[4] For a more detailed description of **DPR** command line arguments, please refer to the **UNIX** on line manual.
[5] On line manual of **DPLib** : `man dplib`, on line manuals of cells : `man n1_dp, man ms_dp, ...`

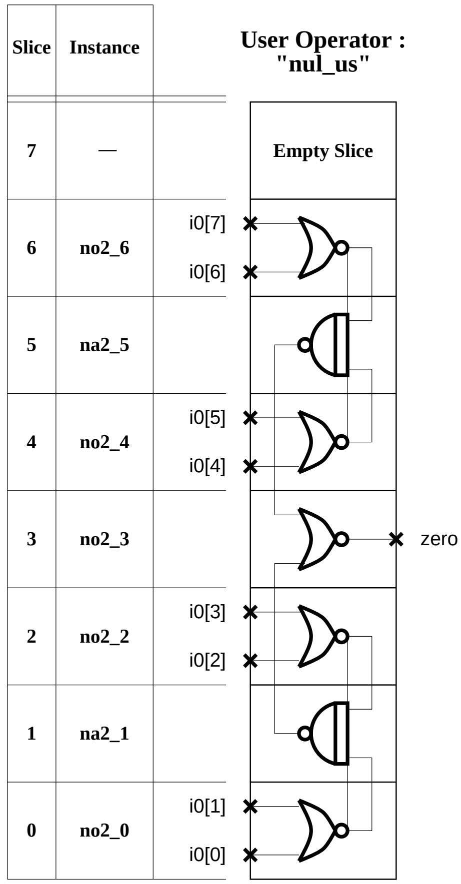


Figure 7: *Zero detect diagram.*

Description of zero detect using **GenLib** language :

```
static void  mkZeroDetect()
{
  DEF_LOFIG( "nul_us" );

  LOCON(   "i0[7:0]", IN   ,   "i0[7:0]" );
  LOCON( "zero"     ,   OUT, "zero"      );
  LOCON(  "vdd"     , IN   ,  "vdd"      );
  LOCON(  "vss"     , IN   ,  "vss"      );

  LOINS( "no2_dp", "no2_0", "i0[0]", "i0[1]", "z2_0", "vdd", "vss", 0L );
  LOINS( "no2_dp", "no2_2", "i0[2]", "i0[3]", "z2_1", "vdd", "vss", 0L );
  LOINS( "no2_dp", "no2_4", "i0[4]", "i0[5]", "z2_2", "vdd", "vss", 0L );
  LOINS( "no2_dp", "no2_6", "i0[6]", "i0[7]", "z2_3", "vdd", "vss", 0L );

  LOINS( "na2_dp", "na2_1",  "z2_0",  "z2_1", "z4_0", "vdd", "vss", 0L );
  LOINS( "na2_dp", "na2_5",  "z2_2",  "z2_3", "z4_1", "vdd", "vss", 0L );

  LOINS( "no2_dp", "no2_3",  "z4_0",  "z4_1", "zero", "vdd", "vss", 0L );

  SAVE_LOFIG();
}
```

As we can see, instance names of the zero detect must conform to a specific syntax. The semantic of this syntax is that the suffix of the instance name indicates to the **DPR** placement stage on which slice to set each instance. This agreement is recalled on the figure **??**.

Modification of the **C** function describing the *data-path netlist* :

```
main()
{
  /* Generate the Zero Detect Column. */
  mkZeroDetect();

  /* Open a new Data-Path figure. */
  DP_DEFLOFIG( "usercol_dpt", 8, LSB_INDEX_ZERO );

  /* Interface description. */
  /* ... */

  /* Data-Path netlist description. */
  /* Multiplexer ... */
  /* Fast Adder ...  */

  /* Zero Detect.    */
  DP_IMPORT( "nul_us",
             "zero",
             "data_s[7:0]",
             "ctrl_zero",
             EOL );

  /* Register ...    */

  /* Terminate the netlist description, and save on disk. */
  DP_SAVLOFIG();

  exit(0);
}
```

Remark :

The **FpGen** and **GenLib** languages are made so that only one *netlist* is taken in the course of the description proccess. On the other hand, to instanciate the model of zero detect with `DP_IMPORT`, this model must be defined before. As a conclusion, we must call the `mkZeroDetect` function before the call to `DP_DEFLOFIG`.

The sequence of commands invoked to generate and check this example is the same as the one used in `sample_dpt`.

4.2.3 Logical synthesis of an operator

In this section we are going to replace the fast adder provided by `DP_ADD2F` by a block generated by logical synthesis. The files associated with this example are named `synthese_dpt.*`.

Behavioral description of the adder :

```
ENTITY  adder_us  IS
  PORT(
         a    : in    BIT_VECTOR(7 downto 0);
         b    : in    BIT_VECTOR(7 downto 0);
      cout_n :   out BIT;
      over_n :   out BIT;
         s    :   out BIT_VECTOR(7 downto 0);
       vdd   : in    BIT;
       vss   : in    BIT
      );
END  adder_us;

ARCHITECTURE  behavior_data_flow  OF  adder_us  IS

SIGNAL cry : BIT_VECTOR(8 downto 0);

BEGIN
  cry(0) <= '0';
  cry(8 downto 1) <=    (a and b)
                   or (a and cry(7 downto 0))
                   or (b and cry(7 downto 0));

  s      <= a xor b xor cry(7 downto 0);
  over_n <= not cry(7);
  cout_n <= not cry(8);

  ASSERT((vdd = '1') and (vss = '0'))
    REPORT "Power supply is missing on adder_us"
    SEVERITY WARNING;
END  behavior_data_flow;
```

Remark :

As the terminal ordering is meaningful for instanciation with `DP_IMPORT`, we adopt for the interface of `adder_us` the same order as the one used in the operator provided by `DP_ADD2F`[6].

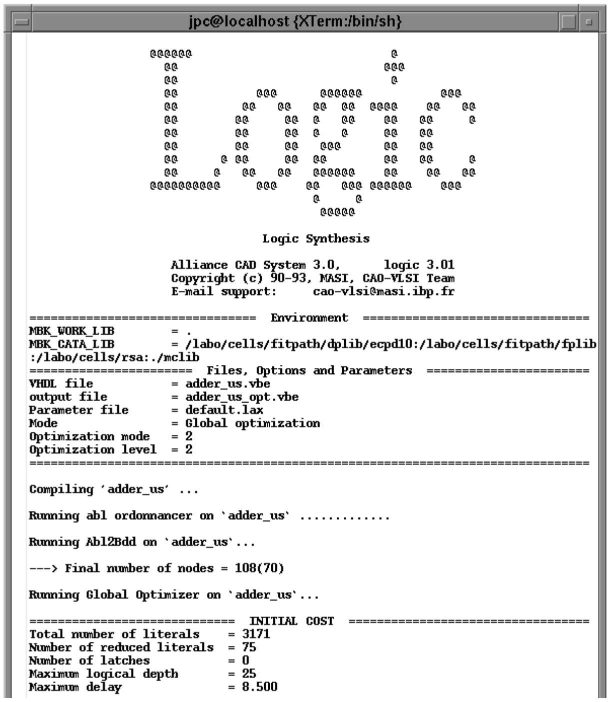
Environment set up for logical synthesis :

```
$ MBK_TARGET_LIB=$ALLIANCE_TOP/cells/fitpath/dplib/ecpd10
$ MBK_NAME_LOG=""
$ export MBK_TARGET_LIB MBK_NAME_LOG
```

[6]For curious people, this operator is named `"add2f_8x8x01_bk"`.

Logic : Optimization of behavioral equations.

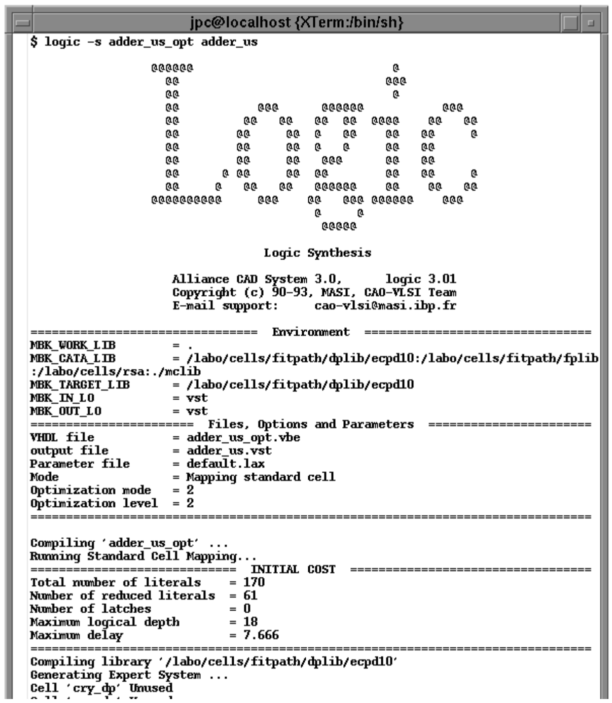
```
$ logic -o adder_us adder_us_opt
```



```
jpc@localhost {XTerm:/bin/sh}

                          Logic Synthesis

              Alliance CAD System 3.0,      logic 3.01
              Copyright (c) 90-93, MASI, CAO-VLSI Team
              E-mail support:     cao-vlsi@masi.ibp.fr

================================  Environment  ================================
MBK_WORK_LIB        = .
MBK_CATA_LIB        = /labo/cells/fitpath/dplib/ecpd10:/labo/cells/fitpath/fplib
:/labo/cells/rsa:./mclib
=====================  Files, Options and Parameters  =====================
VHDL file           = adder_us.vbe
output file         = adder_us_opt.vbe
Parameter file      = default.lax
Mode                = Global optimization
Optimization mode   = 2
Optimization level  = 2
===============================================================================

Compiling 'adder_us' ...

Running abl ordonnancer on `adder_us` ............

Running Abl2Bdd on `adder_us`...

---> Final number of nodes = 108(70)

Running Global Optimizer on `adder_us`...

============================  INITIAL COST  ============================
Total number of literals    = 3171
Number of reduced literals  = 75
Number of latches           = 0
Maximum logical depth       = 25
Maximum delay               = 8.500
```

Logic : Logical synthesis of the *netlist* starting from previously optimized behavioral description.

```
$ logic -s adder_us_opt adder_us
```



Modification of the **C** function describing the *data-path netlist* :

```
main()
{
  /* Open a new Data-Path figure. */
  DP_DEFLOFIG( "synthese_dpt", 8, LSB_INDEX_ZERO );

  /* Interface description. */
  /* ... */

  /* Data-Path netlist description. */
  /* Multiplexer ... */

  /* Synthetized Adder. */
  DP_IMPORT( "adder_us",
             "adder",
             "data_a[7:0]",
             "data_m[7:0]",
             "ctrl_ncout",
             "ctrl_nover",
             "data_s[7:0]",
             EOL );

  /* Zero Detect ... */
  /* Register ...    */

  /* Terminate the netlist description, and save on disk. */
  DP_SAVLOFIG();

  exit(0);
}
```

Further commands are the same as for `sample_dpt`.

4.2.4 Hierarchical design : sub-*data-path*

As an illustration of the hierarchical design capabilities, we have reshape the *netlist* of the adder accumulator. In this new *netlist*, the adder (`DP_ADD2F`) and the zero detect have been put together to make a sub-*data-path* which we call `alu_dpt`.

Diagram of figure **??** shows modifications done to the hierarchy. Files related to this example are named `"hierarchy_dpt.*"`.

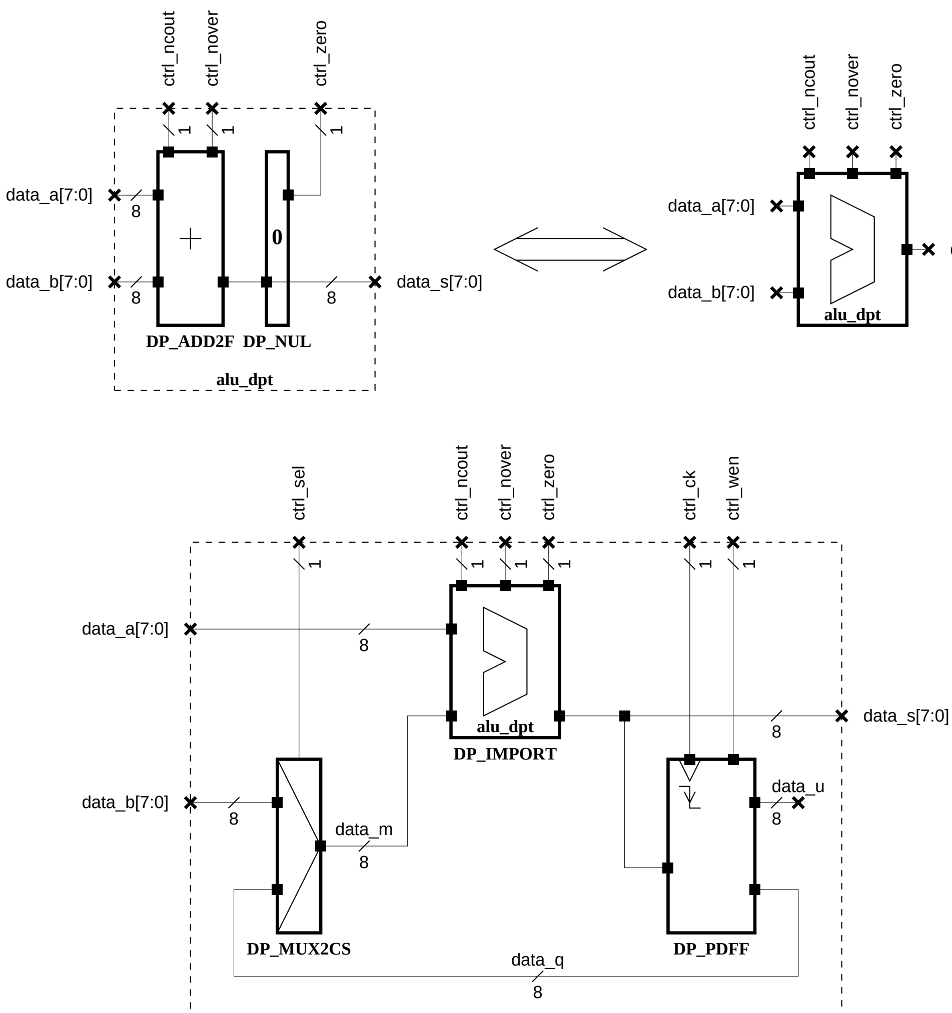


Figure 8: *netlist hierarchy.*

Description of the sub-*data-path* :

```
static void  mkSubDP()
{
  /* Open the ALU part of the Data-Path. */
  DP_DEFLOFIG( "alu_dpt", 8, LSB_INDEX_ZERO );

  /* Interface description. */

  /* Control terminals declarations. */
  DP_LOCON( "ctrl_ncout",   OUT, "ctrl_ncout" );
  DP_LOCON( "ctrl_nover",   OUT, "ctrl_nover" );
  DP_LOCON( "ctrl_zero" ,   OUT, "ctrl_zero"  );
  /* Data terminals declarations. */
  DP_LOCON( "data_a[7:0]" , IN   , "data_a[7:0]" );
  DP_LOCON( "data_b[7:0]" , IN   , "data_b[7:0]" );
  DP_LOCON( "data_s[7:0]" , INOUT, "data_s[7:0]" );
  /* Power supplies terminals. */
  DP_LOCON( "vdd", IN   , "vdd" );
  DP_LOCON( "vss", IN   , "vss" );

  /* Data-Path netlist description. */

  /* Fast Adder. */
  DP_ADD2F( "adder",
            "data_a[7:0]",
            "data_b[7:0]",
            "ctrl_ncout",
            "ctrl_nover",
            "data_s[7:0]",
            EOL );

  /* Zero Detect. */
  DP_NUL( "zero", 8, 0,
          "data_s[7:0]",
          "ctrl_zero",
          EOL );


  /* Terminate the netlist description, and save on disk. */
  DP_SAVLOFIG();
}
```

Modification of the **C** function describing the *data-path netlist* :

```
main()
{
  /* Generate the Zero Detect Column. */
  mkSubDP();

  /* Open a new Data-Path figure. */
  DP_DEFLOFIG( "hierarchy_dpt", 8, LSB_INDEX_ZERO );

  /* Interface description. */
  /* ... */

  /* Data-Path netlist description. */
  /* Multiplexer ... */

  /* Sub-Data-Path. */
  DP_IMPORT( "alu_dpt",
             "alu",
             "ctrl_ncout",
             "ctrl_nover",
             "ctrl_zero",
             "data_a[7:0]",
             "data_m[7:0]",
             "data_s[7:0]",
             EOL );

  /* Register ...    */

  /* Terminate the netlist description, and save on disk. */
  DP_SAVLOFIG();

  exit(0);
}
```

Modification of the `.dpr` file, add the following lines :

```
#         Model Name : Iterations : Height : CPC
DP_GLUE    adder_us       5000          8      2
```

The placement of *glue logic* blocks is automatically performed by the router **DPR**. The `DP_GLUE` command of the `.dpr` allows a control on the way this placement will be done.

Parameters meanings :

Iterations gives the number of iteration the placement algorithm will do.

Height height of the block (in *slices*).

CPC Number of cells per *slice* (inside a **column**). The greater the number, the smaller the surface used for the block, on the other hand the routing becomes more difficult. A good value is generally around 2–3.

Placement of *glue logic* blocks is systemetically performed at each call of the router **DPR**. As the placement of blocks coming from logical synthesis can take a long time, and in the case where we do several successive routing, we can prevent **DPR** to place the block at each routing phasis. The router will use the placement generated at a former iteration. To prevent **DPR** to place a block, just add in the `.dpr` :

```
#          Model Name
DP_KEEP     adder_us
```

Further command are the same as for `sample_dpt`.

Remark :

By the time the `DP_DEFLOFIG` function saves the *netlist* on disk, this *netlist*is automatically flattened at "operator" level. In our example, the sub-*data-path* `"alu_dpt"` included inside `"hierarchy_dpt"` will be flattened when the whole *data-path* will be terminated.

This behavior is needed by the fact that the router **DPR** is not able to manage the hierarchy. It only can route a *netlist* made of operators.

For the sake of clarity regarding the description of the *netlist* we have chosen to adopt a hierarchical concept at **FpGen** level.

A Files provided with the tutorial

Summary of the five examples :

fichiers	exemple
`sample_dpt`	first exemple
`place_dpt`	placement optimization
`usercol_dpt`	customized column
`synthesis_dpt`	syntesized block
`hierarchy_dpt`	hierarchical design

File associated to each example :

extention	type of file
`.sh`	script in *Bourne Shell*
`.c`	**C** source file for **FpGen**
`.dpr`	auxiliary file for **DPR**
`.inf`	auxiliary file for **desb**

In addition to the five shell scripts, a `Makefile` is also provided.