DLX TUTORIAL

January 20, 2000

This tutorial requires about 6 hours in batch mode on a Sparc Station

1 Introduction

Welcome to the **ALLIANCE CAD** system. This file contains a complete tutorial based on the design of the 32 bit microprocessor DLX. This tutorial does not focus on the processor architecture. The goal is to present the available CAD tools in the ALLIANCE 3.0 release (especially logic synthesis and the data path compiler). Beginners who are not familiar with ALLIANCE should start using the ADDACCU tutorial or the AMD tutorial. Each **ALLIANCE** tool can operate as a standalone program but in this tutorial the tools are used according to a precise design flow. The starting point is a behavioural VHDL model. The output is a CIF file.

The tools used in the design are:

- **asimut** : **VHDL** compiler and simulator.
- **dlx_asm** : DLX assembler.
- **syf** : Finite State Machine compiler.
- **logic** : Logic synthesizer
- **netoptim** : Net list optimizer
- **scr** : Standard Cell placer and router.
- **fpgen** : Data path net list capture.
- **dpr** : Data path placer and router.
- **genlib** : Net list capture.
- **bbr** : Block to block router.
- **ring** : Core to pads router.
- **lynx** : Layout to net list extractor.
- **lvx** : Net list comparator.
- **s2r** : Symbolic to real layout converter.

You can get on-line information on any **ALLIANCE** tool using the command :

man <tool name>

DLXm is a 32 bit microprocessor with a micro-programmed internal architecture. The description of the DLX is given by J.L.Hennessy and D.A.Patterson in "Computer Architecture, A quantitative Approach", Morgan Kaufman Publishers, Inc. 1990.

In order to simplify the task of validating the chip, an entire CPU board has been described in **VHDL**.

Thus the simulations are done using small assembler programs which are stored in the CPU board's external memory. This tutorial shows you how to create the DLXm into four steps, which must be followed in order to produce a valid chip.

- **Step 1** : Behavioural Specification and Validation

- **Step 2** : Gate Level net list generation and validation
- **Step 3** : Physical place and route with extraction and validation
- **Step 4** : Translation from symbolic layout to the target process

In order to build the chip, all source files are included with this tutorial (see Appendix).

You will use the **ALLIANCE** tools to validate these sources and to generate first the gate level net list, then the physical layout of the DLXm. The final output is a **CIF** format physical layout file. This can be sent directly to a manufacturer for fabrication.

All you have to do to build the chip is type the commands given in this tutorial. If you are feeling too lazy to type these commands by hand, you can build the entire chip automatically using the command :

```
> make
```

If you want to start again from scratch, you can type the command :

```
> make clean
```

which will remove all the generated files.

The fully automatic chip generation requires about 6 hours on a SPARC station. If you have less time, you can also run separately the four main steps listed above, thanks to special entries in the makefile. These entries are referenced all along this tutorial.

The **ALLIANCE** tools use **UNIX** environment variables. They are accessed by the **UNIX** command *setenv*. For example:

```
> setenv MBK_IN_LO vst
> setenv MBK_OUT_LO vst
> setenv MBK_IN_PH ap
> setenv MBK_OUT_PH ap
> setenv MBK_WORK_LIB .
> setenv MBK_CATAL_NAME CATAL
> setenv MBK_CATA_LIB
    $(ALLIANCE_TOP)/cells/scr:
    $(ALLIANCE_TOP)/cells/bsg: $(ALLIANCE_TOP)/cells/rfg: $(ALLIANCE_TOP)/cells/rsa:
    $(ALLIANCE_TOP)/cells/fitpath/fplib:
    $(ALLIANCE_TOP)/cells/ring
```

- **MBK_IN_LO** : Logical input file format (and filename extension).
- **MBK_OUT_LO** : Logical output file format (and filename extension).
- **MBK_IN_PH** : Physical input file format (and filename extension).
- **MBK_OUT_PH** : Physical output file format (and filename extension).
- **MBK_CATAL_NAME** : Name of the catalogue file in **MBK_WORK_LIB** directory.
- **MBK_CATA_LIB** : Paths to the directories that are to be searched for read-only cell libraries.
- **MBK_WORK_LIB** : Directory where are saved the output files.

You can get on-line information on **ALLIANCE** environment variables using the command :

```
man < envir >
```

Some of the path names may have to be modified, in order to correspond to your particular installation of **ALLIANCE**. In this tutorial we will assume that the directory structure of the source files has not been altered.

In this tutorial, the commands which are inside [ ] are preset. If you make the DLXm from the beginning to the end, you do not have to set again these environment variables. All operations should be executed in the root directory.

2 Simulation method for design validation

2.1 Behavioural model

The specification of the chip which you are going to build is given in the file **dlxm_chip.vbe** in the form of a **VHDL** behavioural description. This allows simulations to be performed immediately.

In order to simplify the task of validating the chip, an entire CPU board has been described in **VHDL**. The CPU board interconnections are described in the file **dlxm_cpu.vst** using VHDL structural syntax. Each component is described in a separate file using VHDL behavioural syntax :

- **dlxm_dec.vbe** : Address decoder.
- **sr64_1a.vbe** : RAM.
- **timer.vbe** : timer for external interrupts.
- **roms.vbe** : Supervisor ROM.
- **romu.vbe** : User ROM.

2.2 Validation Technique

In order to test the chip, a large number of short assembly language programs have been written. The programs were each designed to test one particular aspect of the chip (e.g. an instruction, or a register). Each works on the same principle: a test is performed, if test is OK, the program branches to a fixed address (defined as **good**) and stops with the following message :
ERROR : " assert violation on cell dlxm_dec : " ==== ok : simulation has ended with functional test good ===="
, if not, the program branches to another fixed address and stops with the following message :
ERROR : " assert violation on cell dlxm_dec : " ==== ko : simulation has ended with functional test bad ===="

The programs are assembled using the command **dlx_asm**, which assembles the DLX mnemonics but, instead of generating object code as output, it generates the **VHDL** behavioural description of a 256 byte ROM. It is therefore used to produce the **romu.vbe** and **roms.vbe** files for the board.

2.3 Simulation

The VHDL simulator **asimut** can mix structural and behavioural descriptions : a special file, defined by the environment variable **MBK_CATAL_NAME** (see *man catal*), tells the simulator which behavioural models are to be taken as leaf cells.

Simulation is used to check the initial behavioural description of the DLXm processor and the output results for each phase . However as simulation is time greedy, simulation is done in the tutorial with a single assembly program, to show the design flow and the tools as quickly as possible. Yet the reader must feel free to use all the provided programs for simulation.

3 Interactive Design

3.1 Behavioural Specification

Before starting the chip design, remember that all operations should be executed in the root directory.

The circuit behaviour is described in the *.vbe files using the **ALLIANCE** VHDL subset (see *man vhdl* and *man vbe*).

The assembly language program add000.u is used. To assemble it, do :

```
> setenv MBK_WORK_LIB .
> dlx_asm add000.u romu
> dlx_asm add000.s roms
```

- **add000.u** and **add000.s** are the chosen example of assembly source files.
- **romu** and **roms** are the target files (**romu.vbe** and **roms.vbe**).

You can now perform the simulation :

```
> setenv VH_BEHSFX vbe
> setenv MBK_MAXERR 10
> setenv VH_PATSFX pat
> setenv MBK_IN_LO vst
> setenv MBK_CATAL_NAME CATAL_CPU_CHIP
[> setenv MBK_WORK_LIB .]
> setenv MBK_CATA_LIB
    $(ALLIANCE_TOP)/cells/scr:
    $(ALLIANCE_TOP)/cells/bsg: $(ALLIANCE_TOP)/cells/rfg: $(ALLIANCE_TOP)/cells/rsa:
    $(ALLIANCE_TOP)/cells/fitpath/fplib:./mclib:
    $(ALLIANCE_TOP)/cells/ring
> asimut -l 1 -p 50 -bdd dlxm_cpu dlxm_cpu add000_chip
```

- **-l 1** : size of the label in the **dlxm_cpu.pat** and the **add000_chip.pat** files.
- **-p 50** : simulation will use sets of 50 patterns (see *man asimut*)
- **-bdd** : simulation uses bdd representation
- **dlxm_cpu** : structural description of the board (**dlxm_cpu.vst**)
- **dlxm_cpu** : pattern input filename (**dlxm_cpu.pat**)
- **add000_chip** : result filename (**add000_chip.pat**)

You should get the message :
ERROR : " assert violation on cell dlxm_dec : " ==== ok : simulation has ended with functional test good ====" which means that the test has been performed correctly.

The Status Register and the Program Counter are initialized thanks to the RESET input on the board and convenient assembly instructions in the superuser rom (see **add000.s**).

The result of the simulation is placed in the **pat** file called **add000_chip.pat** . You can take a look at this file using your favorite viewer/editor.

The reader who is willing to do more simulation test should follow the same procedure :

- select a new assembly program (user and supervisor) in the **stock_asm** directory and copy it into the root directory
- create a new romu.vbe and a new roms.vbe by assembling a source file.
- run asimut
- check the simulation result :
  ERROR : " assert violation on cell dlxm_dec : " ==== ok : simulation has ended with functional test good ===="

All the output files resulting from operations of the paragraph 3.1 can be written automatically using the target **functional** of the Makefile by typing :

```
> Make functional
```

3.2 Structural Design

3.2.1 Design Hierarchy

In this step, the structural descriptions of the chip (**dlxm.vst**), and the core (**core.vst**) are used. The chip is described as a core surrounded by pads. The core is divided into two structural blocks: control and data path, the control block being also divided into two structural blocks : sequencer and status, each of which must be represented by its own behavioural description. The following source files are provided :

- **dlxm_chip.vst** : VHDL structural model of the dlxm chip instantiating core and pads.
- **dlxm_core.vst** : VHDL structural model of the core instantiating the data path, and the control.
- **dlxm_ctl.vst.h** : VHDL structural model of the control instantiating the sequencer and the status (the control model (**dlxm_ctl.vst** being saved in the file **dlxm_ctl.vst.h** to prevent future erasing) .
- **dlxm_seq.fsm** : VHDL finite state machine model of the sequencer.
- **dlxm_sts.vbe** : VHDL behavioural model of the status.
- **dlxm_dpt.vbe** : VHDL behavioural model of the data path.

3.2.2 Sequencer state assignment

The sequencer is written using a subset of **VHDL** specifically designed for the description of finite state machines. You must therefore compile this into a VHDL data-flow behavioural model (**vbe**) using the **ALLIANCE** tool **syf** :

```
[> setenv MBK_WORK_LIB . ]
> syf -s dlxm_seq -of dlxm_seq -scan -save
```

- **-s** : uses a vertical encoding algorithm
- **dlxm_seq** : fsm source file (**dlxm_seq.fsm)**
- **-of dlxm_seq** : output behavioural description (**dlxm_seq.vbe**)
- **-scan** : adds a scan-path to the state register.
- **-save** : saves encoding result in **dlxm_seq.cod** file

3.2.3 Validation of the DLXm block view

You can then simulate the resulting model after having copied the structural description of the control in two blocks **dlxm_core.vst.h** :

```
> cp dlxm_ctl.vst.h dlxm_ctl.vst
> chmod 644 dlxm_ctl.vst
```

```
[> setenv VH_BEHSFX vbe]
[> setenv MBK_MAXERR 10]
[> setenv VH_PATSFX pat]
[> setenv MBK_IN_LO vst]
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> setenv MBK_CATAL_NAME CATAL_CPU_BLOCKS
> asimut -l 1 -p 50 -bdd dlxm_cpu dlxm_cpu add000_blocks
```

- **-l 1** : size of the label in the **dlxm_cpu.pat** and the **add000_blocks.pat** files.
- **-p 50** : simulation will use sets of 50 patterns (see *man asimut*)
- **-bdd** : simulation uses bdd representation
- **dlxm_cpu** : structural description of the board (**dlxm_cpu.vst**)
- **dlxm_cpu** : pattern input filename (**dlxm_cpu.pat**)
- **add000_blocks** : result filename (**add000_blocks.pat**)

You should get the message :
ERROR : " assert violation on cell dlxm_dec : " ==== ok : simulation has ended with functional test good ====" which means that the test has been performed correctly.

The **CATAL_CPU_BLOCKS** file tells **asimut** to use the behavioural models for the three blocks data path, sequencer and status.

3.2.4 Data path compilation

The first stage in the synthesis of the structural description of the chip is the generation of the data path. The structural description of the data path is given in the source file **dlxm_dpt.c**. This textual description is equivalent to a schematic capture of the data path. This description uses a set of predefined macros (see *man fpgen*). You must compile this using the **ALLIANCE** data path generator **fpgen**, you will thus use the provided subdirectory **mclib** to store the generated operators:

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_LO vst]
> setenv MBK_OUT_LO vst
> setenv MBK_IN_PH ap
> setenv MBK_OUT_PH ap
> setenv FPGEN_LIB ./mclib
> fpgen dlxm_dpt
```

- **dlxm_dpt** : input filename (**dlxm_dpt.c**)

This generates a hierarchical **VHDL** net list of the data path **dlxm_dpt.vst** . The generated operators (**vbe**, **ap** and **vst** formats) instantiated in **dlxm_dpt.vst** are stored with their associated CATAL into the subdirectory **./mclib** defined by the environment variable **FPGEN_LIB**.

3.2.5 Sequencer synthesis

A standard cell net list of the sequencer is synthesized by the logic synthesis tool **logic** from the behavioural description contained in **dlxm_seq.vbe** :

```
[> setenv MBK_IN_LO vst]
[> setenv MBK_OUT_LO vst]
[> setenv MBK_WORK_LIB . ]
> setenv MBK_TARGET_LIB $(ALLIANCE_TOP)/cells/scr
> logic -o dlxm_seq dlxm_seqo
> logic -s dlxm_seqo dlxm_seq
```

- **-o** : activates the behavioural optimizer and creates **dlxm_seqo.vbe**
- **-s** : activates the standard cells mapper and creates **dlxm_seq.vst**
- **dlxm_seq** : input behavioural description for logic behavioural optimiser (**dlxm_seq.vbe**)
- **dlxm_seqo** : behavioural optimized description (**dlxm_seqo.vbe**)
- **dlxm_seq** : output gate net list description by logic (**dlxm_seq.vst**)

This generates a VHDL gate net list **dlxm_seq.vst** using the standard cell library defined by the environment variable **MBK_TARGET_LIB**.

3.2.6 Status synthesis

A standard cell net list of the status block is synthesized from the behavioural description contained in **dlxm_sts.vbe**. This is done using **logic** as for the sequencer.

```
[> setenv MBK_IN_LO vst]
[> setenv MBK_OUT_LO vst]
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_TARGET_LIB $(ALLIANCE_TOP)/cells/scr]
> logic -o dlxm_sts dlxm_stso
> logic -s dlxm_stso dlxm_sts
```

- **-o** : activates the behavioural optimizer and creates **dlxm_stso.vbe**
- **-s** : activates the standard cells mapper and creates **dlxm_sts.vst**
- **dlxm_sts** : input behavioural description for logic behavioural optimiser (**dlxm_sts.vbe**)
- **dlxm_stso** : behavioural optimized description (**dlxm_stso.vbe**)
- **dlxm_sts** : output gate net list description by logic (**dlxm_sts.vst**)

This generates a VHDL gate net list **dlxm_sts.vst** using the standard cell library defined by the environment variable **MBK_TARGET_LIB**.

3.2.7 Control block generation

The logic synthesizer does not take care of fanout constraints. In order to minimize delays you now use the **ALLIANCE** tool **netoptim**. This performs fan-out optimisation and buffering of a structural description in order to minimise propagation delays of critical paths. **netoptim** operates on the hierarchical view **dlxm_ctl.vst** and flattens it to the gate level, thus creating the gate-level **dlxm_ctl.vst** model (and erasing the hierarchical view **dlxm_ctl.vst**):

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_LO vst]
[> setenv MBK_OUT_LO vst]
> setenv MBK_VDD vdd
> setenv MBK_VSS vss
> netoptim -g dlxm_ctl dlxm_ctl
```

- **-g** : netoptim performs a global optimization
- **dlxm_ctl** : input structural description (**dlxm_ctl.vst**)
- **dlxm_ctl** : output optimised structural description (**dlxm_ctl.vst**) flattened to gate level.

3.2.8 DLXm structural view validation

We have now a complete gate-level net list of the processor. This stage can be validated using **asimut** :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[ > setenv VH_BEHSFX vbe]
[> setenv MBK_MAXERR 10]
[> setenv VH_PATSFX pat]
[> setenv MBK_IN_LO vst]
> setenv MBK_CATAL_NAME CATAL_CPU_GATES
> asimut -l 1 -p 50 -bdd dlxm_cpu dlxm_cpu add000_gates
```

- **-l 1** : size of the label in the **dlxm_cpu.pat** and the **add000_gates.pat** files.
- **-p 50** : simulation will use sets of 50 patterns (see *man asimut*)
- **-bdd** : simulation uses bdd representation
- **dlxm_cpu** : structural description of the board (**dlxm_cpu.vst**)
- **dlxm_cpu** : pattern input filename (**dlxm_cpu.pat**)
- **add000_gates** : result filename (**add000_gates.pat**)

You should get the message :
ERROR : " assert violation on cell dlxm_dec : " ==== ok : simulation has ended with functional test good ====" which means that the test has been performed correctly.

The **CATAL_CPU_GATES** file tells **asimut** to use the behavioural models for the gates and for the generated blocks.

3.2.9 Design for testability

All registers, except the 32 word register file, are in the scan-path :

- **data path** : The structural description uses scanable registers.
- **sequencer** : SYF has been used with the option **-scan** that automatically uses scanable registers.
- **status** : The behavioural description explicitly describes scanable registers.

Here we check the scan path with the simulator **asimut** with a dedicated file that fills in the scan path and check the scan output.

```
[> setenv VH_PATSFX pat]
[ > setenv VH_BEHSFX vbe]
[> setenv MBK_MAXERR 10]
[> setenv VH_PATSFX pat]
[> setenv MBK_IN_LO vst]
[> setenv MBK_CATAL_NAME CATAL_CPU_GATES]
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> asimut -l 10 -p 50 -bdd dlxm_cpu dlxm_scan dlxm_scan_res
```

- **-l 10** : size of the label in the **dlxm_scan.pat** and the **dlxm_scan_res.pat** files.
- **-p 50** : simulation will use sets of 50 patterns (see *man asimut*)
- **-bdd** : simulation uses bdd representation
- **dlxm_cpu** : structural description of the board (**dlxm_cpu.vst**)
- **dlxm_scan** : pattern input filename (**dlxm_scan.pat**)
- **dlxm_scan_res** : result filename (**dlxm_scan_res.pat**)

The output files resulting from commands of the paragraph 3.2, can be created automatically using the target **structural** of the Makefile by typing:

```
> Make structural
```

3.3 Physical Layout

To get the symbolic layout description of the chip, you will :

- route the control block with the standard cells router **scr**
- route the data path with the specialized router **dpr**
- route the data path and the control together with **bbr** (block to block channel router)
- route the core to the pads with the **ring** router

Each place and route step will be validated using the following method :

- extracting a net list from the symbolic layout file using **lynx**
- comparing input net list and extracted net list with **lvx**

3.3.1 Control block routing

You must now use **scr** (Standard Cell Router) to generate the physical layout of the control. We will use the ALLIANCE format for input and output symbolic layout by setting the appropriate environment variables :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_VDD vdd]
[> setenv MBK_VSS vss]
[> setenv MBK_IN_LO vst]
[> setenv MBK_OUT_LO vst]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_PH ap]
> scr -p -r -l 5 -i 3000 -a 5 dlxm_ctl
```

- **-p** : Automatic placement.
- **-r** : Perform routing.
- **-l 5** : Number of rows.
- **-i 3000** : Number of iterations.
- **-a 5** : Number of vertical supplies (power and ground wires).
- **dlxm_ctl** : Input net list (**dlxm_ctl.vst**) and connector placement parameter file (**dlxm_ctl.scr**)

The router takes the net list specified in the file **dlxm_ctl.vst** and generates a physical layout in the file **dlxm_ctl.ap** following the connector parameter file requirements.

In order to verify that no errors occured in the generation of the physical layout, we extract a net list from the layout using the **ALLIANCE** tool **lynx**, and then compare the result with the original net list using **lvx**.

To avoid confusion between original and extracted net lists we use the **al** format to represent extracted net lists. For **lynx** to generate its output in this format, you must set the appropriate environment variable :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_PH ap]
> setenv MBK_OUT_LO al
> lynx dlxm_ctl dlxm_ctl
```

- **dlxm_ctl** : input file (symbolic layout) **dlxm_ctl.ap**
- **dlxm_ctl** : output file (extracted net list) **dlxm_ctl.al**

Then compare :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> lvx vst al dlxm_ctl dlxm_ctl
```

- **vst** and **dlxm_ctl** : input net list **dlxm_ctl.vst**
- **al** and **dlxm_ctl** : extracted net list **dlxm_ctl.al**

And you should get the reply “Net Lists are Identical”.

3.3.2 Routing the Data Path

The data path uses a special type of cell library which is designed to allow routing over the cells, thus saving a considerable amount of space compared with standard-cell implementation. For this routing you must use the **ALLIANCE** tool **dpr** (Data Path Router).

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_VDD vdd]
[> setenv MBK_VSS vss]
[> setenv MBK_IN_LO vst]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_PH ap]
> dpr -o -p -r dlxm_dpt dlxm_dpt
```

- **-o** : Placement optimizatiom.
- **-p** : Automatic placement .
- **-r** : Automatic routing.
- **dlxm_dpt** : Output net list (**dlxm_dpt.ap**).
- **dlxm_dpt** : Input net list (**dlxm_dpt.vst**) and Connector placement parameter file **dlxm_dpt.dpr**.

The router takes the net list specified in the file **dlxm_dpt.vst** and generates a physical layout in the file **dlxm_dpt.ap** following the connector parameter file requirements.

Now repeat the verification procedure for the data path. First the extraction :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_LO al]
> lynx dlxm_dpt dlxm_dpt
```

- **dlxm_dpt** : input file (symbolic layout) **dlxm_dpt.ap**
- **dlxm_dpt** : output file (extracted net list) **dlxm_dpt.al**

Then compare :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> lvx vst al dlxm_dpt dlxm_dpt
```

- **vst** and **dlxm_dpt** : input net list **dlxm_dpt.vst**
- **al** and **dlxm_dpt** : extracted net list **dlxm_dpt.al**

3.3.3 Routing the Core

You must now interconnect the control block and the data path in order to generate the core. You will use **ALLIANCE** tool **bbr** to route them together.

First of all, you generate a "placement" file, which tells **bbr** how the blocks are physically orientated with respect to each other. The simplest way for you to do this is to use **genlib** with the source file **dlxm_core.c**, which is provided:

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> genlib dlxm_core
```

- **dlxm_core** : input file name **dlxm_core.c**

This generates the placement file **dlxm_core.ap**.

The file format environment variables remain the same as before, so to produce a routed core all you have to do is :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_LO vst]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_PH ap]
> bbr dlxm_core -o dlxm_core vdd 12 vss 12
```

- **dlxm_core** : Input net list **dlxm_core.vst** and placement file **dlxm_core.ap**.
- **-o** : Output file **dlxm_core.ap** (erasing placement file).
- **vdd 12 vss 12** : Width of power and ground wires.

You now have the file **dlxm_core.ap** containing the physical layout of the core.

Verification of the Core

Now you perform the same verification process at the core level :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_LO al]
> lynx dlxm_core dlxm_core
```

- **dlxm_core** : input file (symbolic layout) **dlxm_core.ap**
- **dlxm_core** : output file (extracted net list) **dlxm_core.al**

Then compare :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> lvx vst al dlxm_core dlxm_core
```

- **vst** and **dlxm_core** : input net list **dlxm_core.vst**
- **al** and **dlxm_core** : extracted net list **dlxm_core.al**

And again you should get the reply "Net Lists are Identical".

3.3.4 Routing the Chip

The final stage in generating the physical layout of the chip is to route the core to the pads using **ring**. The pad placement depends on external constraints (see *man ring*) and is therefore defined in the file **dlxm_chip.rin**.

Again the file format variables remain unchanged so you can now perform the routing :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATAL_NAME CATAL
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_LO vst]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_PH ap]
> ring dlxm_chip dlxm_chip
```

- **dlxm_chip** : input netlist **dlxm_chip.vst** and placement file **dlxm_chip.rin**
- **dlxm_chip** : output symbolic layout file name **dlxm_chip.ap**

Thus a physical layout of the chip is generated in the file **dlxm_chip.ap**.

Verification of the Chip

Finally you must verify at the chip level :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
[> setenv MBK_IN_PH ap]
[> setenv MBK_OUT_LO al]
> lynx dlxm_chip dlxm_chip
```

- **dlxm_chip** : input file (symbolic layout) **dlxm_chip.ap**
- **dlxm_chip** : output file (extracted net list) **dlxm_chip.al**

Then compare :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATA_LIB ...]
> lvx vst al dlxm_chip dlxm_chip
```

- **vst** and **dlxm_chip** : input net list **dlxm_chip.vst**
- **al** and **dlxm_chip** : extracted net list **dlxm_chip.al**

And if you get the reply "Net Lists are Identical" your entire chip has been correctly routed.

You can visualize the chip using **graal**. Use for example:

```
[> setenv MBK_CATA_LIB ...]
> graal dlxm_chip
```

The output files resulting from commands of the paragraph 3.3 can be created automatically using the target **physical** of the Makefile by typing:

```
> Make physical
```

3.4 The Final Touch

The last step is to translate the symbolic layout (coordinates in lambda units) into a physical layout for the target CMOS process (two output formats are supported : CIF and GDSII). This task is done by **s2r**, which performs symbolic to real expansion, gap filling, denotching, and instantiates preexisting layout cells (necessary for the pads).

You must first define an environment variable with the name of a file containing the parameters of the target process (in our case a 1 micron process) and then specify the cif format :

```
[> setenv MBK_WORK_LIB . ]
[> setenv MBK_CATAL_NAME CATAL_CPU_GATES]
[> setenv MBK_CATA_LIB .:...]
[> setenv MBK_IN_PH ap]
> setenv RDS_TECHNO_NAME $(ALLIANCE_TOP)/etc/prol10.rds
> setenv RDS_OUT cif
> setenv RDS_IN cif
```

The variable **RDS_IN** is required to specify the format of preexisting layout cells.

Then to perform the conversion :

```
> s2r dlxm_chip dlxm_chip
```

- **dlxm_chip** : input symbolic layout file (**dlxm_chip.ap**).
- **dlxm_chip** : output physical layout file (**dlxm_chip.cif**).

The output files resulting from commands of the paragraph 3.4 can be written automatically using the target **real** of the Makefile by typing:

```
> Make real
```

Congratulations! The chip is ready for the foundry.

DLX TUTORIAL - APPENDIX

January 20, 2000

You can find below all the source files for the DLX microprocessor.

functional specification

- **dlxm_cpu.vst** : VHDL structural model of the board instantiating RAM, ROM, timer, decoder, dlxm.
- **sr64_1a.vbe**, **sr64_8a.vst** and **sr64_32a.vst** : Behavioural and structural description for the RAM .
- **dlxm_dec.vbe** : VHDL behavioural description of the address decoder.
- **timer.vbe** : VHDL behavioural description of the timer.
- **dlxm_chip.vbe** : VHDL behavioural description of the dlxm.
- **add000.u and add000.s** : Short assembly language programs.
- **CATAL_CPU_CHIP** : List of behavioural models required for a simulation using the behavioural model of DLXm.
- **dlxm_cpu.pat** : Pattern input file for testing various views of dlxm chip.

structural design

- **dlxm_chip.vst** : VHDL structural model of the dlxm instantiating core and pads.
- **dlxm_core.vst** : VHDL structural model of the core instantiating the data path and the control.
- **dlxm_ctl.vst.h** : VHDL structural model of the control instantiating the sequencer and the status.
- **dlxm_seq.fsm** : Finite State Machine description for sequencer.
- **dlxm_dpt.vbe** : Behavioural description for data path.
- **CATAL_CPU_BLOCKS** : List of behavioural models required for a simulation using behavioural descriptions for data path, sequencer and status.
- **dlxm_dpt.c** : Source code for data path generator.
- **./mclib** : Subdirectiry used by **fpgen** to store the generated data path operators.
- **dlxm_sts.vbe** : Behavioural description for status and interrupts.
- **CATAL_CPU_GATES** : List of behavioural models required for a simulation using structural descriptions for data path, sequencer and status.
- **dlxm_scan.pat** : Pattern input file for scan path simulation.

physical layout

- **dlxm_dpt.dpr** : Placement file for data path block connectors (**dpr** tool)
- **dlxm_ctl.scr** : Placement file for control block connectors (**scr** tool)
- **dlxm_core.c** : Placement file for control and data path (**bbr** tool)
- **dlxm_chip.rin** : Placement file for dlxm pads (**ring** tool)