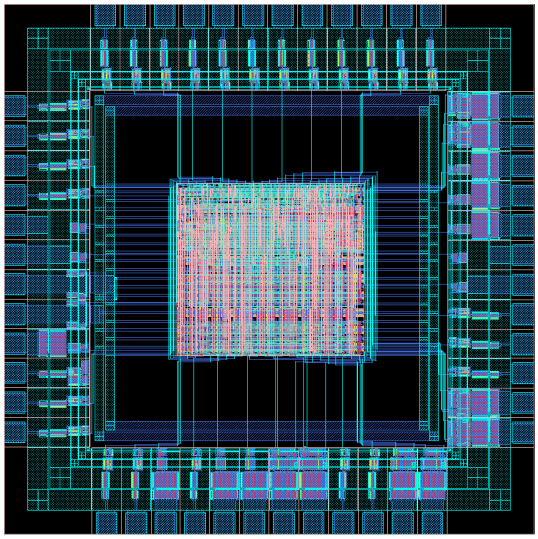
0pt 0.6pt

ALLIANCE TUTORIAL

Pierre & Marie Curie University
2001 - 2004

PART 2
Logic synthesis

Ak Frederic Lam Kai-shing
Modified by LJ



The purpose of this tutorial is to provide a quick turn of some **ALLIANCE** tools, developed at the LIP6 laboratory of Pierre and Marie Curie University.

The tutorial is composed of 3 main parts independent from each other:

- VHDL modeling and simulation
- Logic synthesis
- Place and route

Before going further you must ensure that all the environment variables are properly set (source alcenv.sh or alcenv.csh file) and that the Alliance tools are available when invoking them at the shell prompt.

All tools used in this tutorial are documented at least with a manual page.

Contents

1 **Introduction**
2 **Finite states machine Synthesis**
2.1 Introduction
2.2 MOORE and MEALY automatons
2.3 SYF and VHDL
2.4 Example
2.5 Step to follow
3 **Automat for digicode**
3.1 Step to follow
4 **Logic synthesis and structural optimization**
4.1 Introduction
4.1.1 Logic synthesis
4.1.2 Solve fan-out problems
4.1.3 Long path visualization
4.1.4 Netlist Checking
4.1.5 Scan-path insertion
4.2 Step to follow
4.2.1 *Mapping* on predefined cells
4.2.2 Netlist visualization
4.2.3 Boolean network optimization
4.2.4 Netlist optimization
4.2.5 Netlist checking
4.2.6 Scan-path insertion in the netlist
5 **AMD 2901**
5.1 exercise
5.2 step to follow
5.3 error found
6 **AMD2901 structure**
7 **Part controls design**
7.1 **genlib** description example
7.2 provided files checking
7.3 Part controls description
8 **Data-path design**
8.1 Example of description with genlib macro-functions
8.2 Data-path description
9 **The** *Makefile* **or how to manage tasks dependency**
9.1.1 Rules
9.1.2 models Rules
9.1.3 Variables definitions
9.1.4 Predefined variables
10 **Appendix: Diagrams as an indication but not-in conformity with the behavioral**

PART 2 : Logic Synthesis

All the files used in this part are located under `/usr/share/doc/alliance-doc-5.0/tutorials/synthesis/src` directory. This directory contents four subdirectories and one Makefile :

- Makefile
- amdbug
  - Makefile
  - amdfindbug.pat : tests file
  - several files amd.vbe : behavioral description
- counter
  - Makefile
  - cpt5.fsm : description in fsm
  - cpt5.pat : tests file
- digicode
  - Makefile
  - digicode.fsm : description in fsm
  - paramfile.lax : use to modify the fan-out
  - digicode.pat : tests file
  - scan.path : make it possible to observe registers contents
- amd2901
  - Makefile
  - amd2901_ctl.vbe : behavioral description of control part
  - amd2901_dpt.vbe : behavioral description of data-path
  - amd2901_ctl.c : file .c of control part
  - amd2901_dpt.c : file .c of data-path
  - amd2901_core.c : file .c of heart
  - amd2901_chip.c : file .c of the circuit with their pads
  - pattern.pat : tests file

1 Introduction

The goal of this section is to present some ALLIANCE tools which are:

- Logic synthesis tools **SYF, BOOM, BOOG, LOON, SCAPIN** ;
- Data-path generation tool**GENLIB** ;
- *netlist* graphical viewer **XSCH** ;
- formal proof Tools **FLATBEH, PROOF**;
- The simulator **ASIMUT** ;

The first two sections will relate to the *netlist* **generation and validation** methods of predefined cells. Indeed, even if it is acquired that the tools for ALLIANCE generation function correctly, the validation of each generated view is **essential** . It makes it possible to limit the cost and the time of the design.
The two other sections will be reserved for the **data-path generation and the control part** of AMD2901.

2 Finite states machine Synthesis

2.1 Introduction

A pure combinatorial circuit has no internal registers. So its outputs depend only on its primary inputs. On the contrary a synchronous sequential circuit having internal registers sees its outputs changing according to its inputs but also memorized values in its registers. As consequence, the circuit state at the moment t+1 also depends on its state at the moment t. This type of circuit can be formally modelized as a **finite states machine**.

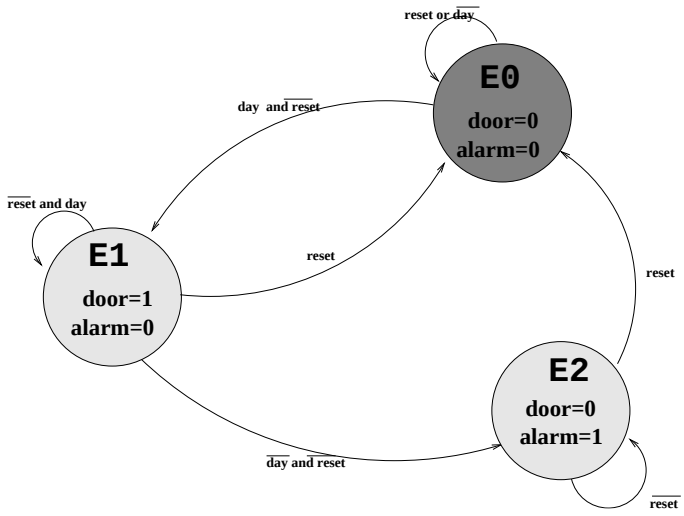
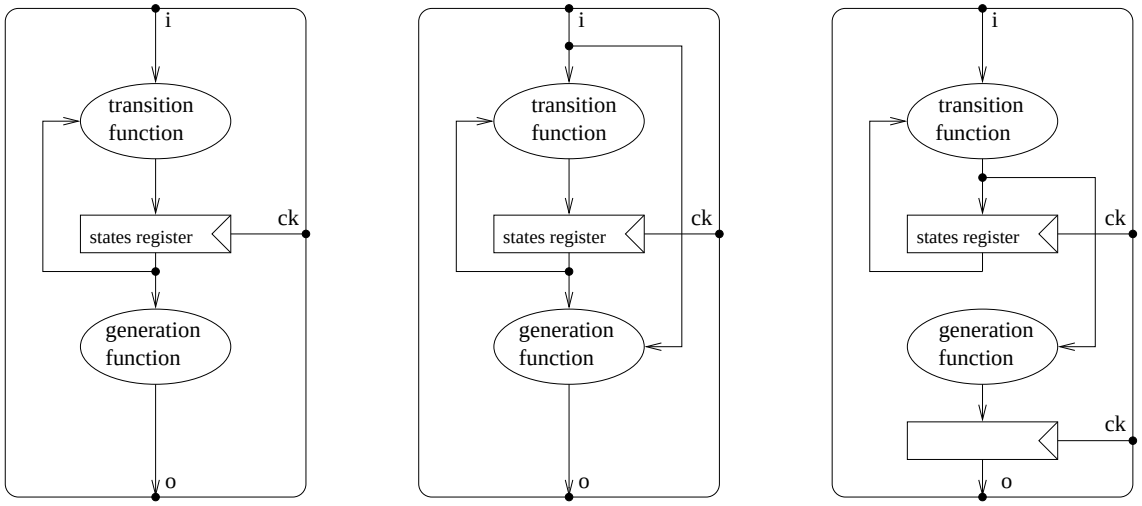


Figure 1: Automat

2.2 MOORE and MEALY automaton

The MOORE automaton sees the state of its outputs changing only on clock-edges. The inputs can thus move between two clock-edges without modifying the outputs. But in

the case of MEALY automaton, the variation of the inputs can modify at any time the value of the outputs. It will be essential to separate the generation function from the transition function (Moore automaton). Two distinct processes will then modelize the next state computation and the current state register update.



Moore Automat *Mealy Automat* *Moore Automat optimized on propagation time of o*

Figure 2: Automats

2.3 SYF and VHDL

In order to describe the automatons, we use a particular **VHDL** style description that defines architecture "FSM" (**F**inite-**S**tate **M**achine).

The corresponding file also has the extension **fsm** . From this file, the tool **SYF** makes the automaton synthesis and after state encoding, it transforms this abstracted automaton into a Boolean network and a state register. **SYF** then generates a **VHDL** file using the **vbe** subset. Like most of all tools used in alliance, it is necessary to set some variables before using **SYF** . You can refer to the **man** page of **syf** for more details.

2.4 Example

In order to take in hand the particular syntax of a **fsm** file, an example of **three** successive "1" counter is presented. Its vocation is to detect for example on a connection series, a sequence of **three** successive "1" counter. The state graph is represented on the figure 3.
The **fsm** format is detailed in the man page **fsm(5)** .

```
    entity circuit is
          port (
               ck, i, reset, vdd, vss : in bit;
               o : out bit
          );
    end circuit;
    architecture MOORE of circuit is
          type ETAT  _TYPE is (E0, E1, E2, E3);
          signal EF, EP : ETAT  _TYPE;
- - pragma CURRENT  _STATE EP
- - pragma NEXT  _STATE EF
- - pragma CLOCK CK
    begin
          process (EP, i, reset)
          begin
               if (reset='1') then
                    EF<=E0;
               else
                    case EP is
                         when E0 =
                              if (i='1') then
                                   EF <= E1;
                              else
                                   EF <= E0;
                              end if;
                         when E1 =
                              if (i='1') then
                                   EF <= E2;
                              else
                                   EF <= E0;
                              end if;
                         when E2 =
                              if (i='1') then
                                   EF <= E3;
                              else
                                   EF <= E0;
                              end if;
                         when E3 =
                              if (i='1') then
                                   EF <= E3;
                              else
                                   EF <= E0;
                              end if;
                         when others =  assert ('1')
                              report "etat illegal";
                   end case;
               end if;
               case EP is
                    when E0 =
                         o <= '0' ;
                    when E1 =
                         o <= '0' ;
                    when E2 =
                         o <= '0' ;
                    when E3 =
                         o <= '1' ;
                    when others =  assert ('1')
                         report "etat illegal";
              end case;
          end process;
          process(ck)
          begin
               if (ck='1' and not ck'stable) then
                    EP <= EF;
               end if;
          end process;
    end MOORE;
```

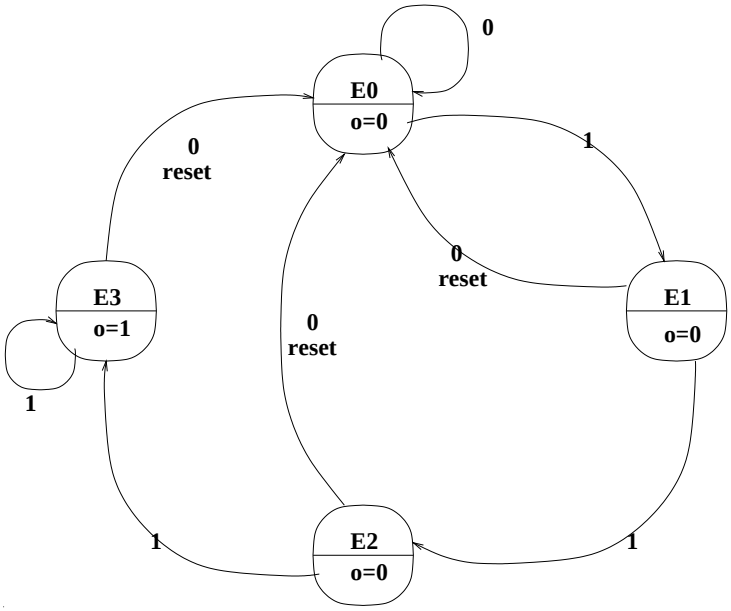


Figure 3: states graph of three successive "1" counter

2.5 Step to follow

Now you can use this example to write the description of a **five** successive "1" counter in a**Moore** automaton.

- position the environment variables .
- launch **SYF** with the coding options **-a, -J, -m, -O, -R** and by using the options **-CEV** .

  -a Uses "Asp" as encoding algorithm.

  -j Uses "Jedi" as encoding algorithm.

  -m Uses "Mustang" as encoding algorithm.

  -o Uses the one hot encoding algorithm.

  -r Uses distinct random numbers for state encoding.

```
> syf -CEV -a <fsm_source>
```

- visualize the files **enc** . Those files contains one state name followed by its hexadecimal code value.
- write stimuli (test vectors) and simulate with **ASIMUT** .

3 Automaton for digicode

We want to design a digicode circuit whose keyboard is represented on the figure 4. The specifications are as follows:

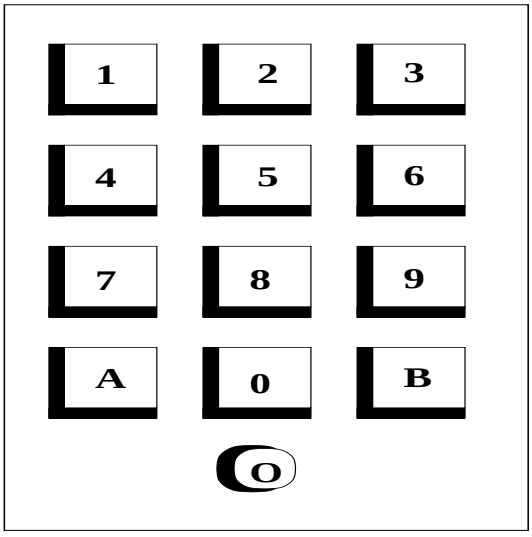


Figure 4: Clavier

- The numbers from 0 to 9 are coded in natural binary on 4 bits. A and B are coded in the following way:
  - A: 1010
  - B: 1011
- The digicode works in two modes:
  - Day Mode: The door opens while pressing on "O" or if entering the good code
  - Night Mode: The door opens only if the code is correct.

  To distinguish the two cases an external "timer" calculates the signal **day** which is equal to ' 1 ' between 8h00 and 20h00 and ' 0 ' otherwise.
- The digicode order an alarm as soon as one of the entered numbers is not the good.
- The digicode automaton returns in idle state if nothing returned to the keyboard at the end of 5 seconds or if alarm sounded during 2mn - signal **reset** -. For that it receives a signal from **reset** external timer.
- The chip works at 10MHz.
- Any pressure of a key of the keyboard is then followed by the signal **press_kbd** . This one announces to the chip that the output data of the keyboard is valid. This signal is set to 1 during a clock-edge.

The code is **53A17** (but you can take the code who agrees to you). The interface of this automaton is as follows:

- in **ck**
- in **reset**
- in **day**

- in **i[3:0]**
- in **O**
- in **press_kbd**
- out **door**
- out **alarm**

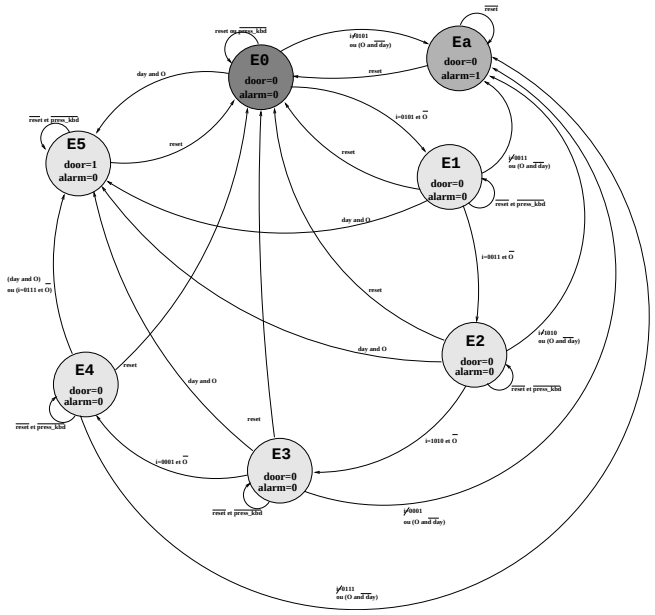


Figure 5: Digicode states graph

3.1 Step to follow

- draw the states graph.
- describe it in the **fsm** format .
- synthesize your description with **SYF** using different state encoding algorithms **-a, -j, -m, -o, -r** and by using the options **-CEV**.

```
> syf -CEV -a <fsm_source>
```

- write stimuli (test vectors).
- simulate with **ASIMUT** all the resulting **vbe** descriptions.

4 Logic synthesis and structural optimization

4.1 Introduction

4.1.1 Logic synthesis

The logic synthesis permits to obtain a *netlist* of gates given a Boolean network (format **vbe**). Several tools are available:

- The tool **BOOM** allows the Boolean network optimization before mapping with **BOOG** .
- The tool **BOOG** synthesizes a *netlist* by using a library with predefined cells such as **SXLIB** . The *netlist* can be either with the format **vst** or with the format **al** . Check the environment variable **MBK_OUT_LO**=vst.

4.1.2 Solve fan-out problems

Generated *netlists* may contain internal signals that drive a significant number of gates (large FAN-OUT). In order to solve this problem, the tool **LOON** replaces the cells having a too large fan-out by more powerful cells and/or insert buffers.

4.1.3 Long path visualization

At any moment, the *netlists* can be graphically displayed using **XSCH**. This tool permits also to highlight the longest path on the schematic thanks to the files **xsc** and **vst** generated by **BOOG** and **LOON** .

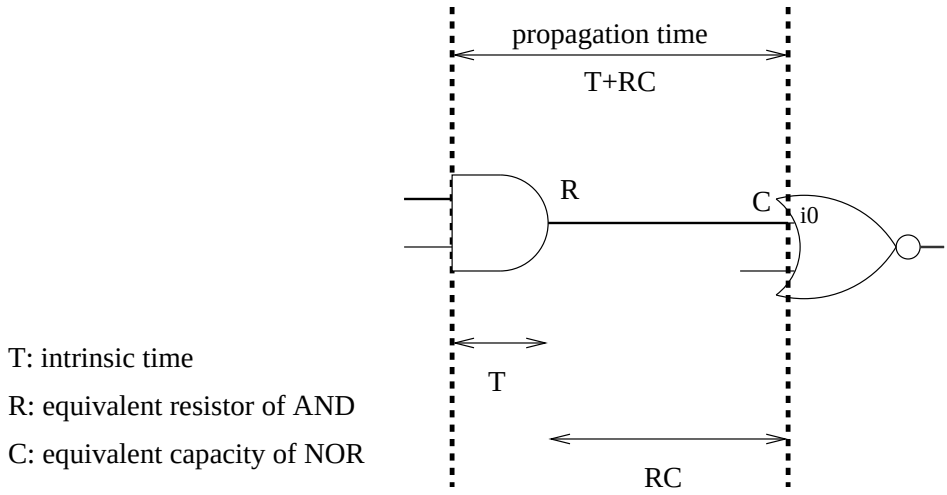


Figure 6: Simplified timing diagram

Equivalent resistor *R* of the *figure 6* is calculated on the totality of the transistors of the *AND* belonging to the active way. In the same way, the capacity *C* is calculated on the busy transistors of the *NOR* corresponding to the way between *i0* and the output of the cell.

4.1.4 Netlist Checking

The netlist must be validated. For that, you have **ASIMUT** , but also the tool **PROOF** which proceeds to a formal comparison of two behavioral descriptions (**vbe**). The tool **FLATBEH** is useful to obtain a new behavioral file starting from a *netlist* (given a **vbe** file for each leave cells of the hierarchy).

4.1.5 Scan-path insertion

With **SCAPIN** we can insert a scan-path into the netlist. The scan-path allow the designer to observe in test mode the value of all registers of your circuit. The path is created by changing each registers into a mux_register (or by inserting a multiplexer in front of all registers).

4.2 Step to follow

4.2.1 *Mapping* on predefined cells

For each Boolean network obtained previously:

- set properly environment variables;
- synthesize the structural view:

```
> boog <vbe_source>
```

- launch **BOOG** on different *netlist*s to observe **SYF** options influence (different state encoding technics).
- validate the work of **BOOG** with **ASIMUT** , the *netlist*s obtained with stimuli which were used to validate the initial Boolean network.

4.2.2 Netlist visualization

- The longest path (critical path) is described in the **xsc** file produced by **boog** . The **XSCH** tool will use it to highlight this path on the schematic. To launch the graphical schematic viewer:

```
>xsch -I vst -l <vst_source>
```

- The red color indicates the critical path.
- you can use the option ' *-slide* ' followed by netlist names to display one by one a set of schematics. The keys ' + ' and ' - ' can then be used to display respectively next and previous netlist.

4.2.3 Boolean network optimization

To analyze Boolean optimization effect :

- launch Boolean optimization with the tool **BOOM** by asking an optimization in **surface** then in **delay** ;

```
>boom -V <vbe_source> <vbe_destination>
```

- test **BOOM** with the various algorithms - S, - J, - B, - G, - p..., the options specifie which algorithm has to be used for the boolean optimization.
- compare the literal number after factorization.
- remake the Boolean networks synthesis with the tool **BOOG** and compare the results.

4.2.4 Netlist optimization

For all the structural view obtained previously:

- launch **LOON** with the command:

```
>loon <vst_source> <vst_destination> <lax_param>
```

- carry out an fanout optimization by modifying the fanout factor in the option file **.lax** .The optimization mode and level are able to be change in this file.
- impose capacities values on the outputs.

4.2.5 Netlist checking

to carry out on the best of your *netlist*s:

- validate the work of **LOON** by running **ASIMUT** on the different *netlist*s obtained, using the stimuli that were defined to validate the initial behavioral view.
- Make a formal comparison of your netlist with the original behavioral file resulting from **SYF** :

```
>flatbeh <vst_source> <vbe_dest>
```

```
>proof -d <vbe_origin> <vbe_dest>
```

Checks if the files are formally identicals.

4.2.6 Scan-path insertion in the netlist

to carry out on the best of your *netlist*s:

- insert a scan-path connecting all the digicode registers.

```
>scapin -VRB <vst_source> <path_file> <vst_dest>
```

Example of .path file

```
BEGIN_PATH_REG

cs_0
cs_1
cs_2
END_PATH_REG

BEGIN_CONNECTOR

SCAN_IN      scin
SCAN_OUT     scout
SCAN_TEST    test
END_CONNECTOR
```

- build ten patterns to test the scan-path and simulate with **ASIMUT** .

5 AMD 2901

5.1 exercise

First of all, here is an exercise to understand the AMD2901 chip functionality. The goal is to design it using Alliance, as described in the following parts of this tutorial.

To explore all functionalities, you will have to validate the behavioral view that will be provided. All needed informations will be find in appendix.

The validation will have to be done using stimuli generated by **genpat**. The vectors must be carefully written to enable you to detect **BUG** in your behavioral file **.vbe** . Approximately 500 patterns will be enough for debugging your AMD 2901.

5.2 step to follow

It is necessary to generate stimuli that tests all the parts and all functions of the AMD following the specifications described in the documentation.

- filling and reading the 16 boxes memories of the RAM .
- test the RAM shifter
- filling and reading of the accumulator.
- test the accumulator shifter .
- test the arithmetic and logic operations (addition, subtraction, overflow, carry, propagation, etc...) .
- exhaustive test of the inputs conditioned by I[2:0].
- data-path test vectors

5.3 error found

You can notice that for the RAM shifter values "101" and "111" of i[8:6], the AMD causes a shift of the accumulator that should not take place.

for the values "000" and "001" of i[8:6], the circuit writes the ALU output in RAM .

The AMD carries out the operation R xor S for I[5:3]=111 instead of carrying out the operation for I[5:3]=110.

It carries out the operation /(R Xor S) for I[5:3]=110 instead of I[5:3]=111.

6 AMD2901 structure

We break up Amd2901 into 2 blocks:

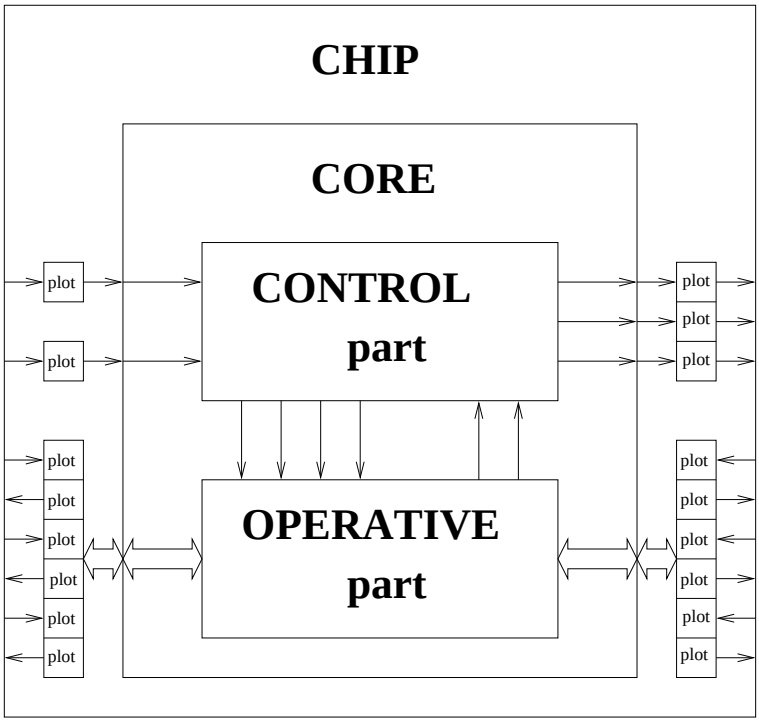


Figure 7: Amd2901 Organization

- The data-path contains the Amd2901 regular parts , the registers and the arithmetic logic unit.
- The control part contains irregular logic, the instructions decoding and the flags computation.

We will use the following hierarchical description:

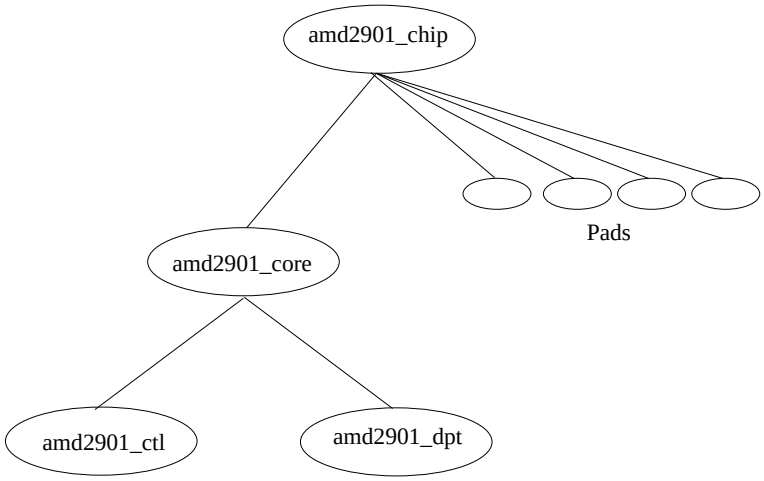


Figure 8: Hierarchy

The provided files are as follows:

- amd2901_ctl.vbe, behavioral description of the part controls
- amd2901_dpt.vbe, behavioral description of the part data-path
- amd2901_ctl.c, file **C** of the part controls
- amd2901_dpt.c, file **C** of the part of data path
- amd2901_core.c, file **C** of the heart
- amd2901_chip.c, file **C** of the circuit containing the pads
- pattern.pat, tests file
- CATAL, file listing the behavioral files, to be modify
- Makefile, to automate the generation

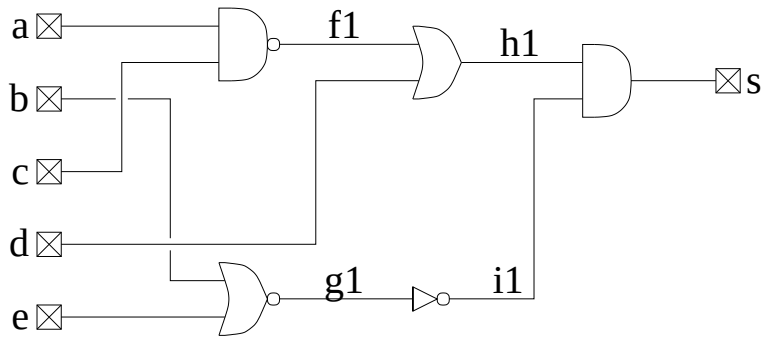
7 Part controls design

This part of irregular logic will be carried out with the cells of the library **SXLIB**.

Description in VHDL netlist (i.e *.vst*) of the various gates hazardous when the circuit contain several thousands of them. there exists a tool for procedural signals lists generation , **genlib** . It is then enough to describe in C using macro-functions the *signals list* in gates of the block. The library of macro-functions C is called **genlib** . The **genlib** execution produces a description VHDL with the format *.VST* . For more details, consult the manual (man) on **genlib** .

7.1 genlib description example

here a simple circuit:



The equivalent **genlib** file is as follows:

```
#include <genlib.h>
    main()
      {
       GENLIB_DEF_LOFIG("circuit");

       /* Connectors declaration */
       GENLIB_LOCON("a",IN,"a1");
       GENLIB_LOCON("b",IN,"b1");
       GENLIB_LOCON("c",IN,"c1");
       GENLIB_LOCON("d",IN,"d1");
       GENLIB_LOCON("e",IN,"e1");
       GENLIB_LOCON("s",OUT,"s1");

       GENLIB_LOCON("vdd",IN,"vdd");
       GENLIB_LOCON("vss",IN,"vss");

       /* Combinatorial gates instanciation */
       GENLIB_LOINS("na2_x1","nand2","a1","c1","f1","vdd","vss",0);
       GENLIB_LOINS("no2_x1","nor2","b1","e1","g1","vdd","vss",0);
       GENLIB_LOINS("o2_x2","or2","d1","f1","h1","vdd","vss",0);
       GENLIB_LOINS("inv_x1","inv","g1","i1","vdd","vss",0);
       GENLIB_LOINS("a2_x2","and2","h1","i1","s1","vdd","vss",0);

       /* Save of the figure */
       GENLIB_SAVE_LOFIG();
       exit(0);
      }
```

Save it under the name “ circuit.c ” then compile the file with the command :

```
> genlib circuit
```

You obtain the file “ circuit.vst ”. (if is not it, it may be due to environment variables that are not properly set for **genlib**).

7.2 provided files checking

Create the file **CATAL** in your simulation directory . It must contain the following lines:

```
amd2901_ctl C
amd2901_dpt C
```

It makes the simulator use the behavioral files (.vbe) of “ amd2901_ctl ” and of “ amd2901_dpt ’ ’.

```
> asimut amd2901_chip pattern result
```

You can verify the resulting patterns by using **xpat** on the file “ result ”.

7.3 Part controls description

The diagrams corresponding to the signals list to design are provided to you. compile it by using the steps below.

Generate the signals list **vst** starting from the file **c** by the command:

```
> genlib amd2901_ctl
```

Then validate the structural view obtained by simulating the complete circuit with the tests vectors which are provided to you. Replace the behavioral view of the part controls by his structural view by removing the name *amd2901_ctl* of **CATAL** file.

```
> asimut -zerodelay amd2901_chip vecteurs result
```

Note that one carries out a simulation “ without delay ” of the netlist. In the event of problem, do not hesitate to use **xpat** .

```
> asimut amd2901_chip pattern result
```

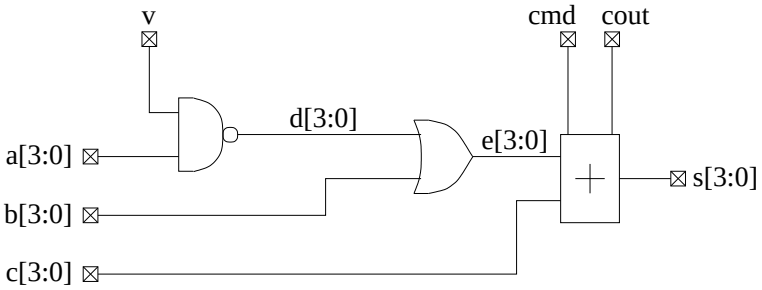
After having validated the functional behavior of the netlist, simulate it using propagation delays. Modify time values between the patterns. Indeed, **asimut** is able to evaluate the propagation times for each cell of the netlist (taken into account the "after" clauses specify in vbe files).

8 Data-path design

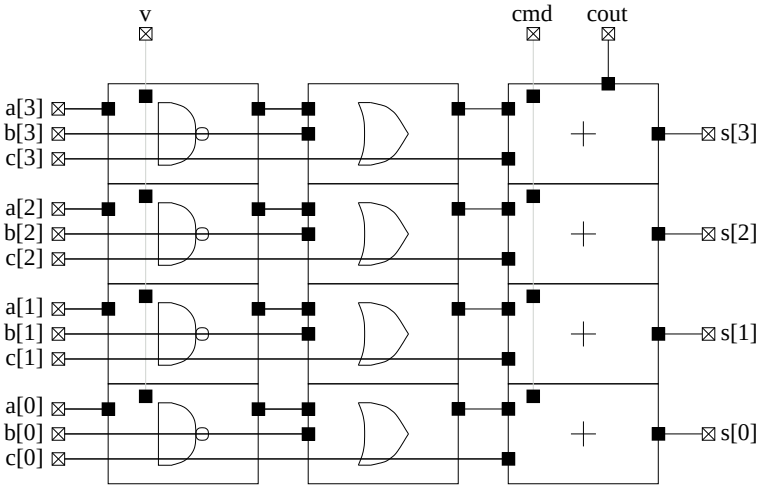
The data path is formed by the regular logic of the circuit. In order to benefit from this regularity, we generates the signals list in the vectorial operators form (or columns) *via* the macro-functions of the tool **genlib** . That makes it possible to save place by using several times the same material . For example, the *NOT* of a mux of N bits is instanciate only once for these N bits...

8.1 Example of description with genlib macro-functions

Let us consider the following circuit:



Here the corresponding data-path structure :



Each gate occupies a column, a column making it possible to treat a whole of bits for the same operator. The first line represents bit 3, the last bit 0 .

The file **genlib** correspondent is as follows:

```
#include <genlib.h>
    main()
      {
       GENLIB_DEF_LOFIG("data_path");

       /* connectors declaration */
       GENLIB_LOCON("a[3:0]",IN,"a[3:0]");
       GENLIB_LOCON("b[3:0]",IN,"b[3:0]");
       GENLIB_LOCON("c[3:0]",IN,"c[3:0]");
       GENLIB_LOCON("v",IN,"w");
       GENLIB_LOCON("cout",OUT,"ct");
       GENLIB_LOCON("s[3:0]",OUT,"s[3:0]");
       GENLIB_LOCON("cmd",IN,"cmd");
       GENLIB_LOCON("vdd",IN,"vdd");
       GENLIB_LOCON("vss",IN,"vss");

       /* operators creation */
       GENLIB_MACRO(GEN_NAND2, "model_nand2_4bits", F_PLACE, 4, 1);
       GENLIB_MACRO(GEN_OR2, "model_or2_4bits", F_PLACE, 4);
       GENLIB_MACRO(GEN_ADSB2F, "model_add2_4bits", F_PLACE, 4);

       /* operators Instanciation */
       GENLIB_LOINS("model_nand2_4bits", "model_nand2_4bits",
                        "v", "v", "v", "v",
                        "a[3:0]",
                        "d_aux[3:0]",
                        vdd, vss, NULL);
       GENLIB_LOINS("model_or2_4bits", "model_or2_4bits",
                        "d_aux[3:0]",
                        "b[3:0]",
                        "e_aux[3:0]",
                        vdd, vss, NULL);
       GENLIB_LOINS("model_add2_4bits", "model_add2_4bits",
                        "cmd",
                        "cout",
                        "ovr",
                        "e_aux[3:0]",
                        "c[3:0]",
                        "s[3:0]",
                        vdd, vss, NULL);

       /* Save of figure */
       GENLIB_SAVE_LOFIG();
       exit(0);
      }
```

Save it under the name “ data_path.c ”, then compile the file with the command:

```
> genlib data_path
```

You obtain the file “ data_path.vst ” (in the contrary case, it may be that your environment is badly configured for **genlib**).In this case, pass to the section “ Data path description”.

Note: genlib can also create the physical placement (the drawing) of a structural description .

8.2 Data-path description

The diagrams corresponding to the signals list to design are given. Compile it following the steps below .

Generate the signals list **vst** starting from the **c** file, using the command:

```
> genlib amd2901_dpt
```

Validate the netlist in the same way as it has been done for the control part. Remove CATAL file and simulate the circuit with **asimut** .

```
> asimut -zerodelay amd2901_chip pattern result
```

9 The *Makefile* or how to manage tasks dependencies

The synthesis under **Alliance** breaks up into several tools being carried out chronologically on a data flow. Each tool has its own options giving the results more or less adapted according to the use of the circuit.

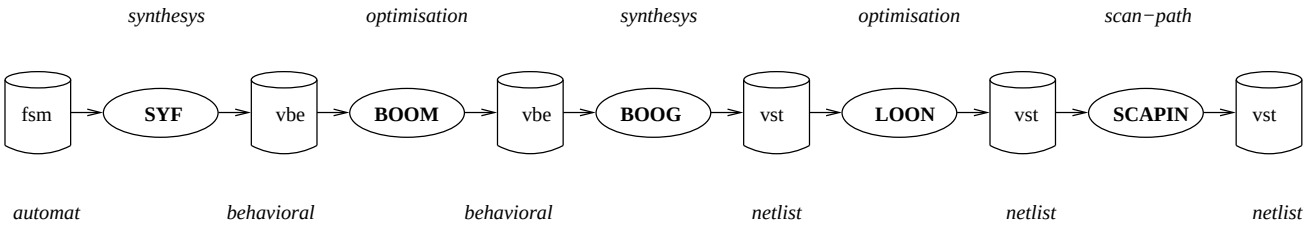


Figure 9: the synthesis

The data dependency in the flow are materialized in reality by file dependency. The file **Makefile** carried out using the command **make** makes it possible to manage these dependencies.

9.0.1 Rules

A **Makefile** is a file containing one or more rules translating the dependency between the actions and the files.

example :

```
target1 : dependence1 dependence2 ....
       #Rq: each command must be preceded by a tabulation
       command_X
       command_Y
          .
          .
          .
```

The dependencies and targets represent files in general.
Only the first rule (except the models cf 9.0.2) of the **Makefile** is examined. The following rules are ignored if they are not implied by the first.
So some dependencies of a rule **X** are themselves of the rules in the **Makefile** then these last will be examined before the appealing rule **X** .
For each rule **X** examined, so at least one of its dependencies is more recent than its target then the commands of the rule **X** will be carried out. *Note::* the commands are generally used to produce the target (i.e a new file).
A target should not represent a file. In this case, the commands of this rule will be always carried out.

9.0.2 models Rules

These rules are more general-purpose because you can specify more complex dependency rules. A model rule be similar to a normal rule, except a symbol (%) appears in the target name. The dependencies also employ (%) to indicate the relation between

the dependency name and the target name. The following model rule specifies how all the files **vst** are formed starting from the **vbe** .

```
#example of rule for the synthesis
%.vst : %.vbe
      boog $*
```

9.0.3 Variables definitions

You can define variables in any place of the file **Makefile** , but for legibility we will define them at the beginning of file.

```
#variables definitions
MY_COPY   = cp -r
MY_NUM    = 42
MY_STRING ="hello"
```

They are usable in any place of the **Makefile** . They must be preceded by the character **$**

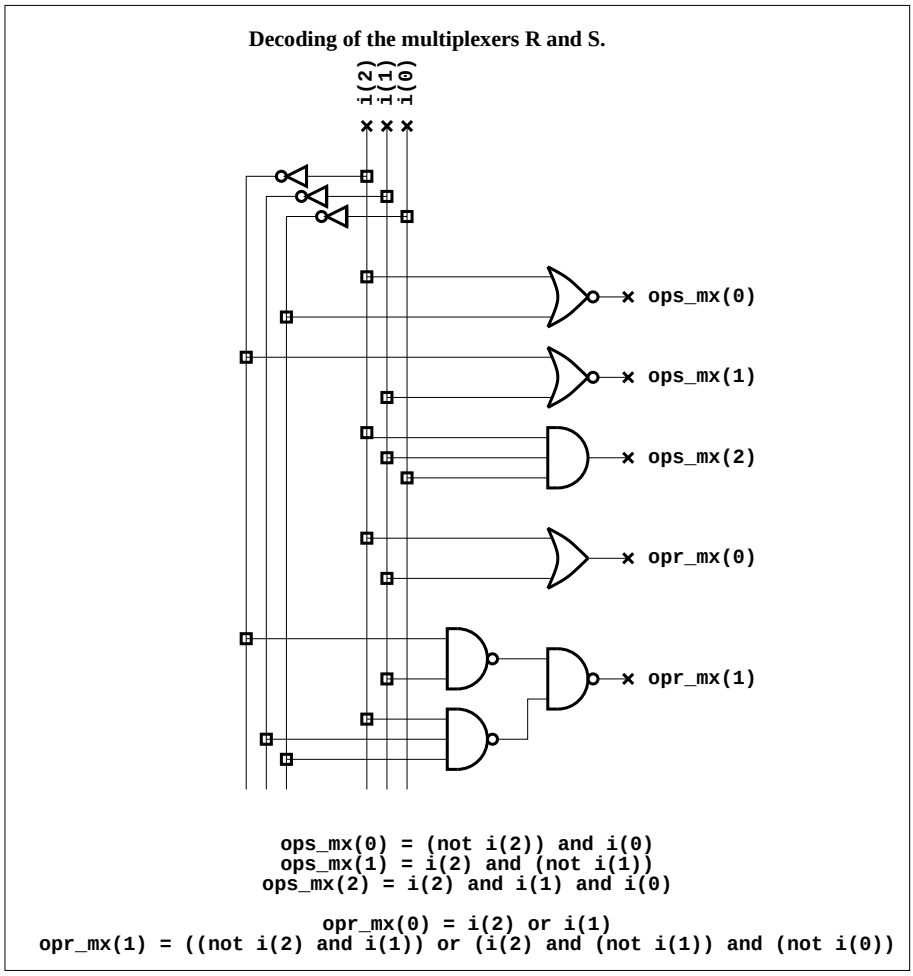
```
#use a variable in a rule

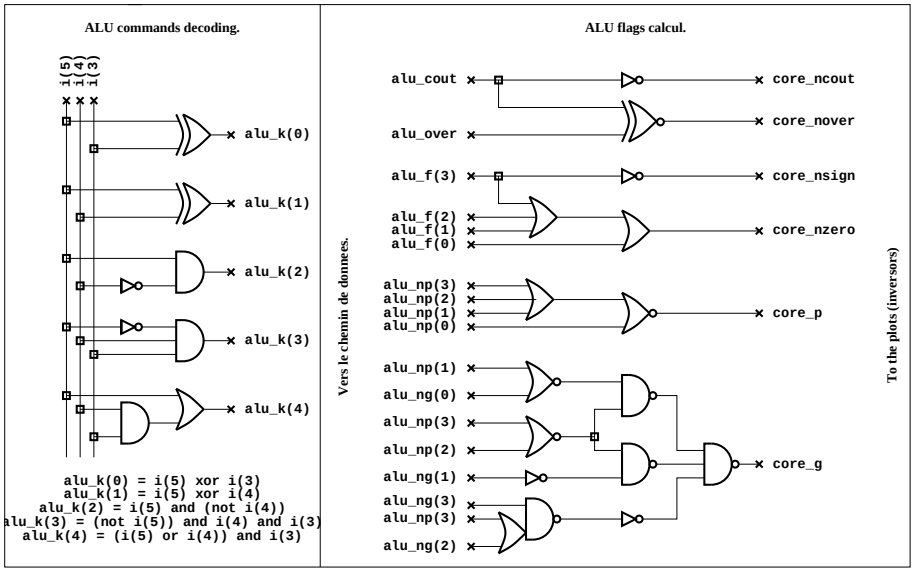
copy:
      ${MY_COPY} digicode.vbe tmp/
```

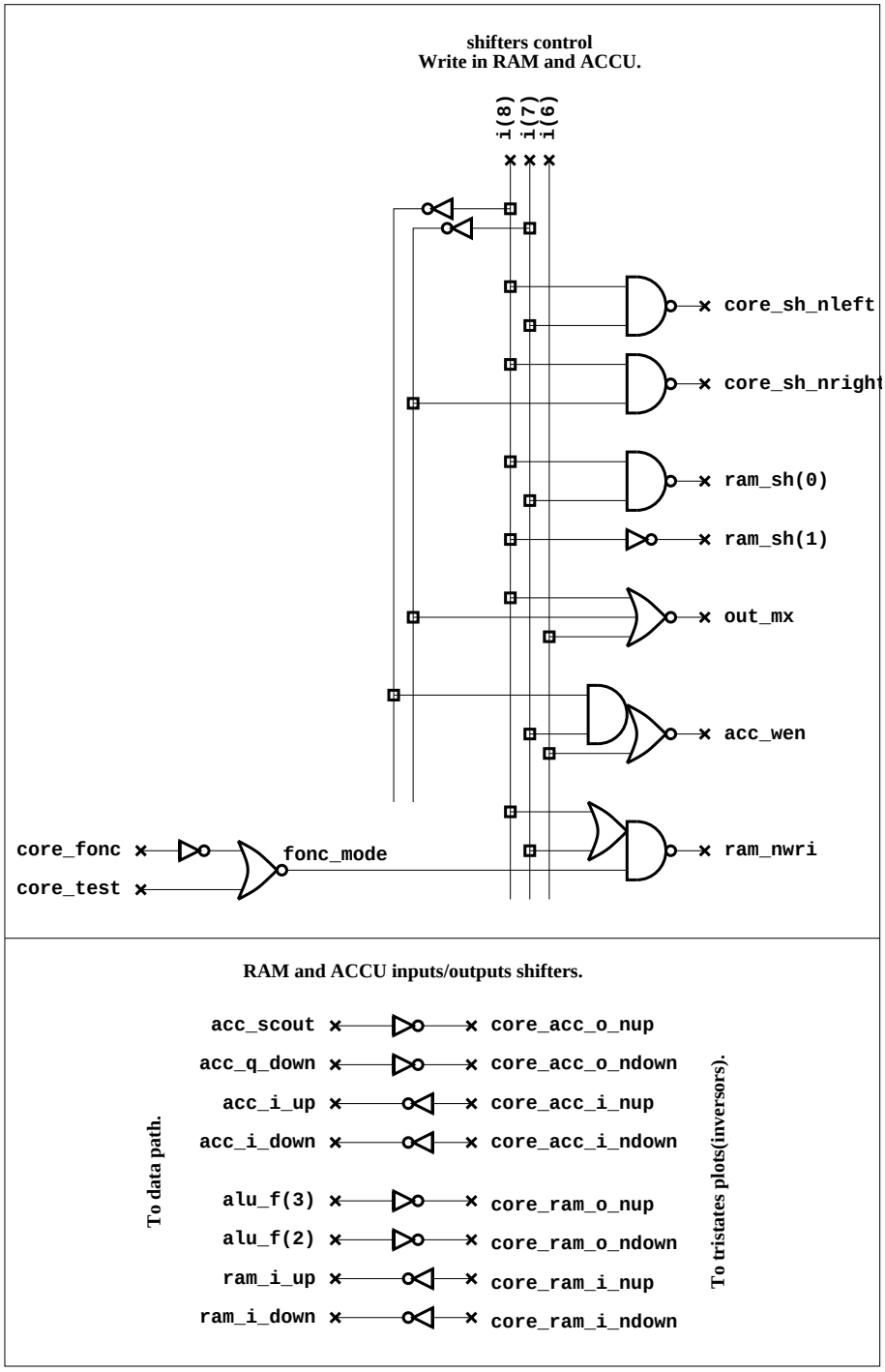
9.0.4 Predefined variables

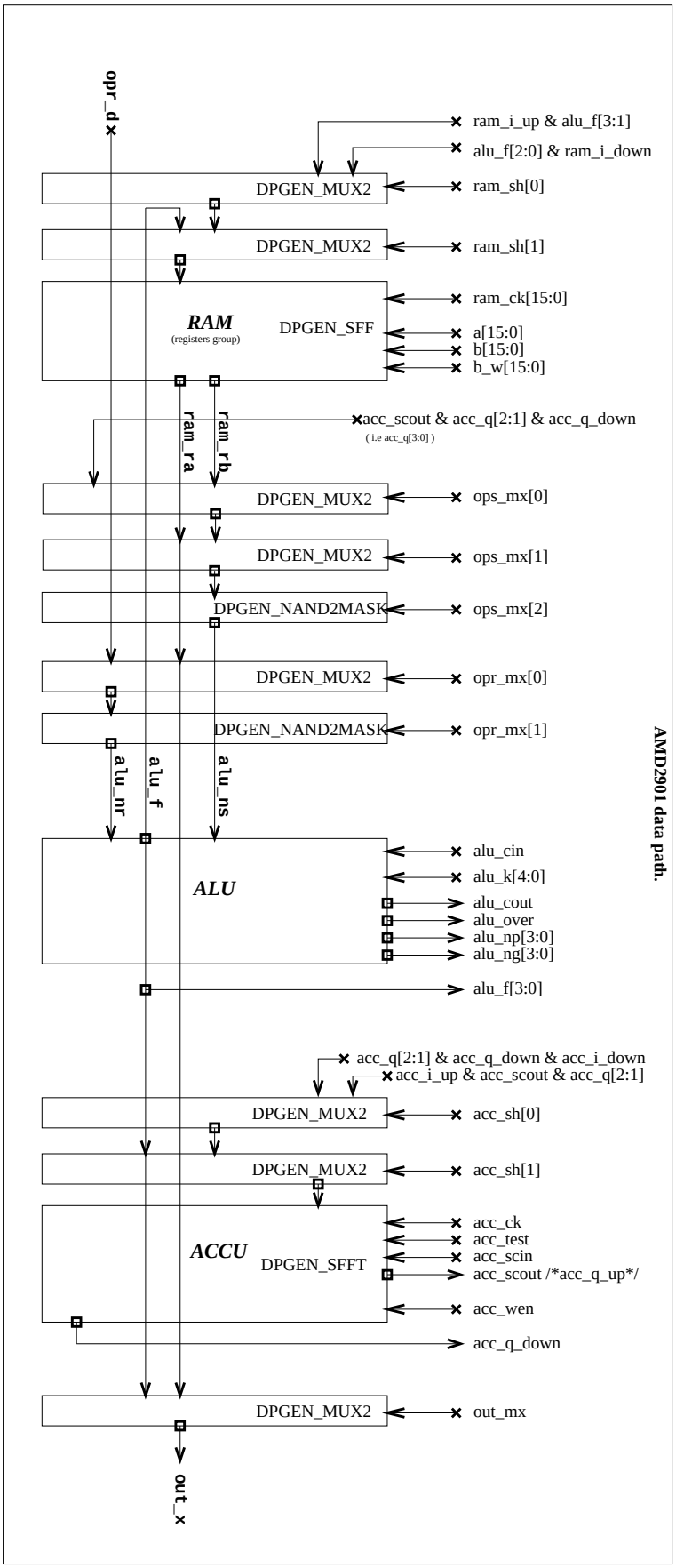
- **$@** Complete target name.
- **$*** Name of the targets file without the extension.
- **$<** Name of the first dependent file.
- **$+** Names of all the dependent files with double dependencies indexed in their order of appearance.
- **$^** Names of all the dependent files. The doubles are remote.
- **$?** Names of all the dependent files more recent than the target.
- **$%** Name of member for targets which are archives (language C). If, for example, the target is *libDisp.a(image.o)* , **$%** is *image.o* and **$@** is *libDisp.a* .

10 Appendix: Diagrams as an indication but not-in conformity with the behavioral

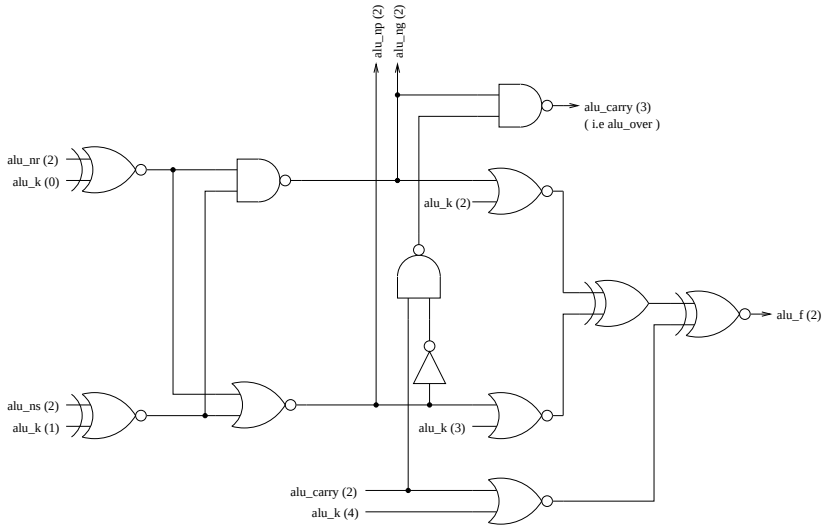




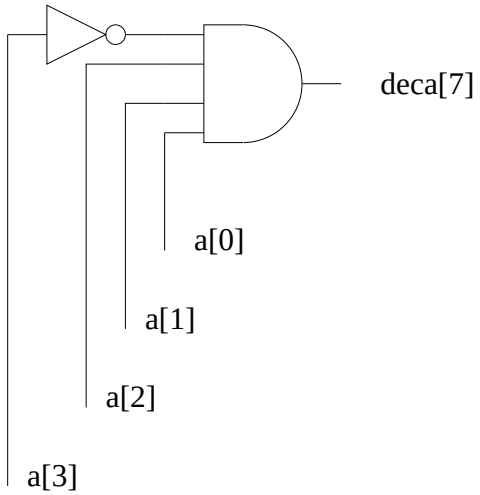

shifters control
Write in RAM and ACCU.
i(8)
i(7)
i(6)
core_sh_nleft
core_sh_nright
ram_sh(0)
ram_sh(1)
out_mx
acc_wen
ram_nwri
core_fonc
core_test
fonc_mode
RAM and ACCU inputs/outputs shifters.
To data path.
acc_scout
acc_q_down
acc_i_up
acc_i_down
alu_f(3)
alu_f(2)
ram_i_up
ram_i_down
core_acc_o_nup
core_acc_o_ndown
core_acc_i_nup
core_acc_i_ndown
core_ram_o_nup
core_ram_o_ndown
core_ram_i_nup
core_ram_i_ndown
To tristates plots(inversors).

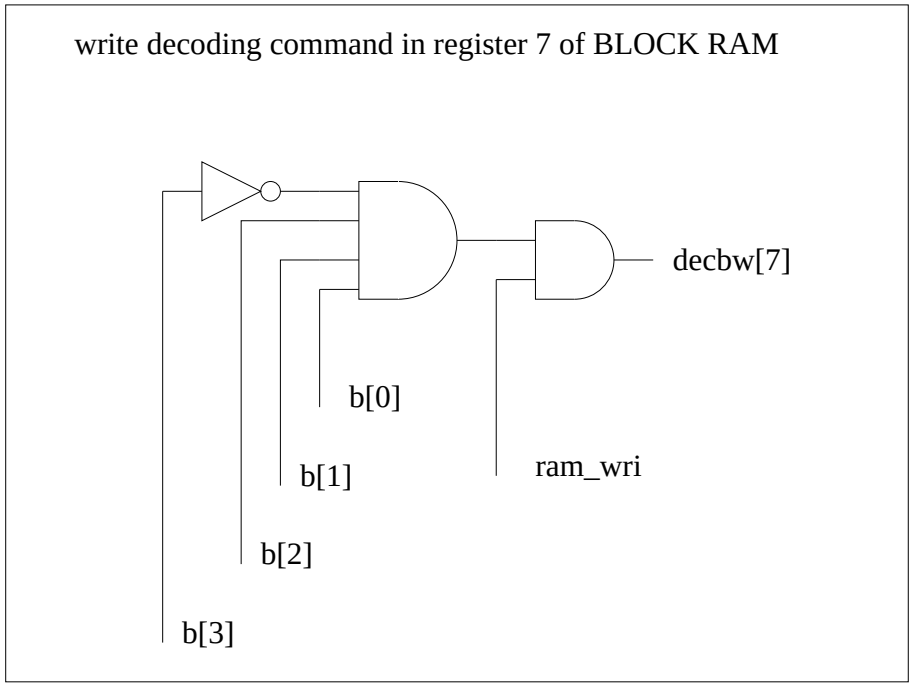

opr_d
ram_i_up & alu_f[3:1]
alu_f[2:0] & ram_i_down
DPGEN_MUX2
ram_sh[0]
DPGEN_MUX2
ram_sh[1]
RAM
(registers group)
DPGEN_SFF
ram_ck[15:0]
a[15:0]
b[15:0]
b_w[15:0]
acc_scout & acc_q[2:1] & acc_q_down
(i.e acc_q[3:0])
ram_ra
ram_rb
DPGEN_MUX2
ops_mx[0]
DPGEN_MUX2
ops_mx[1]
DPGEN_NAND2MASK
ops_mx[2]
DPGEN_MUX2
opr_mx[0]
DPGEN_NAND2MASK
opr_mx[1]
alu_nr
alu_f
alu_ns
ALU
alu_cin
alu_k[4:0]
alu_cout
alu_over
alu_np[3:0]
alu_ng[3:0]
alu_f[3:0]
acc_q[2:1] & acc_q_down & acc_i_down
acc_i_up & acc_scout & acc_q[2:1]
DPGEN_MUX2
acc_sh[0]
DPGEN_MUX2
acc_sh[1]
ACCU
DPGEN_SFFT
acc_ck
acc_test
acc_scin
acc_scout /*acc_q_up*/
acc_wen
acc_q_down
DPGEN_MUX2
out_mx
out_x
AMD2901 data path.

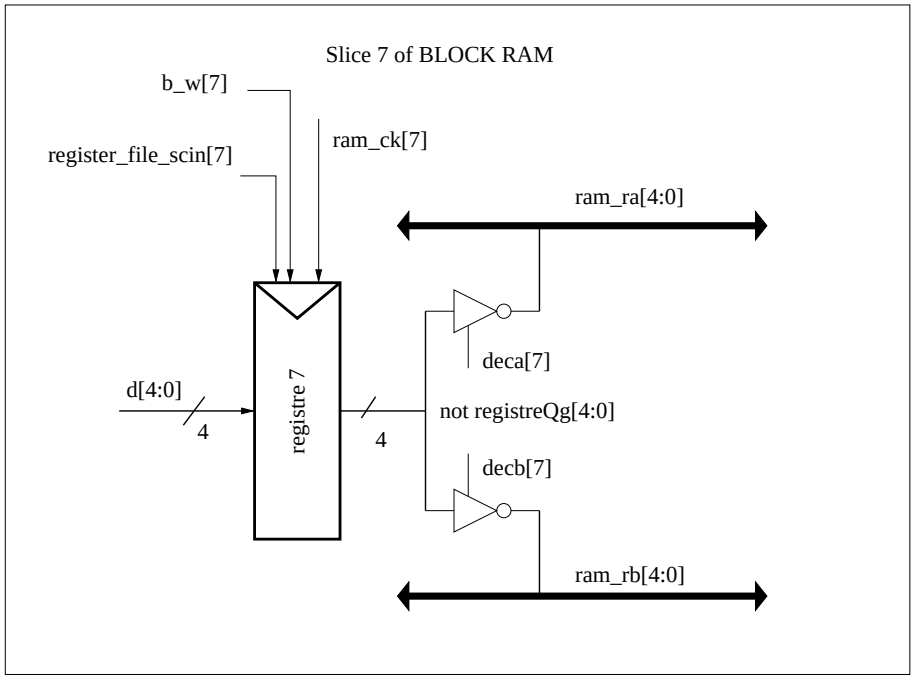
ALU slice 2 representation



Decoding of the reading command of register 7
BLOCK RAM on port A




write decoding command in register 7 of BLOCK RAM
decbw[7]
b[0]
b[1]
ram_wri
b[2]
b[3]


Slice 7 of BLOCK RAM
b_w[7]
ram_ck[7]
register_file_scin[7]
ram_ra[4:0]
deca[7]
d[4:0]
4
registre 7
4
not registreQg[4:0]
decb[7]
ram_rb[4:0]