PLS

Macro Block Handling: Macro+
# Table of Contents

**Macro Block Handling: Macro+**

1. PLS Arithmetic Macro Generators .................................................. 1
   1.1. Adders ................................................................................. 2
       1.1.1. Area oriented adder .................................................. 2
       1.1.2. Speed oriented adder .................................................. 2
       1.1.3. Particular case of LUT based technologies (XC4000 and FLEX8000) ............ 2
   1.2. Subtractors ........................................................................ 4
   1.3. Incrementer/Decrementer ...................................................... 5
   1.4. Comparator ....................................................................... 5
       1.4.1. Equality and inequality comparators ............................... 5
       1.4.2. Less than, less than or equal, greater than and greater than or equal comparators .... 5
   1.5. Multiplier ......................................................................... 6
   1.6. How to generate these operators ......................................... 7
       1.6.1. Inference call ............................................................... 7
       1.6.2. Structural call ............................................................. 8

2. Vendor macro blocks call .............................................................. 10
   2.1. XBLOX macro blocks call .................................................... 10
       2.1.1. Structural call ............................................................. 10
       2.1.2. Inference call ............................................................. 13
   2.2. XC3000 and XC4000 macro blocks call ............................... 14
   2.3. ACTgen Macro Builder Call .................................................. 15
       2.3.1. Software Requirements ................................................. 15
       2.3.2. Inferring ACTgen Counters .......................................... 15
       2.3.3. Inferring ACTgen Adders and Subtractors ..................... 17
       2.3.4. Inferring ACTgen Multiplexors ..................................... 18
       2.3.5. Inferring ACTgen Decoders ......................................... 23
       2.3.6. How to use ACTgen Macros ......................................... 24
       2.3.7. Hints for synthesis ....................................................... 24
   2.4. ACT1, ACT2 and ACT3 macro blocks call ............................. 25

3. Predesigned block call ................................................................. 27

**Appendix 1: Macro Block Directive File Syntax** ........................................ 29
Macro Block Handling: Macro+

For common HDL arithmetic or logic operators, PLS provides macro generators or parameterized generators. These generators are optimized for each target technology and exist with several options (speed or area). These blocks are called PLS macro generators.

Of course, the different vendors prepare their own arithmetic generators and have a large set of macros available. Knowing perfectly well their technology, they are the best experts in preparing the best blocks. Therefore in PLS, it is also possible to call any external macro or parameterized generators. In the second section, explanation will be given on external or vendor macro calls.

1. PLS Arithmetic Macro Generators

PLS provides adder/subtractor, incrementer/decrementer, comparator, multiplier of any bit width in all target technologies. These blocks may be created as isolated blocks and used in a structural mode or called through the VHDL operators (by inference) in a general VHDL description. For these macro blocks 2 options exist:

• area where a minimal area is targeted,
• speed where the best critical path is obtained.

Notice that if in the target technology library, dense and fast 1 (or 2) bit operation cell exist, PLS may use it as a leaf or an elementary hard macro (hard cell) to build the parameterized generators. For area option, this basic blocks are automatically used when available. For speed option, dedicated synthesis is used and these hard cells are often not interesting.

Notice also that for LUT (Look-Up Table) based technologies like XC4000 for Xilinx and FLEX8000 for Altera, the adders are constructed by using the fast carry logic. Especially for these technologies, this technique allows the obtaining of very efficient adders versus the generation of adders from equations.
1.1. Adders

1.1.1. Area oriented adder

A traditional ripple carry adder structure is used. The synthesis is done either using standard cells or elementary basic cells, or a hard cell, namely a 2 bit adder, at the leaf, when available. Notice that hard cells are automatically used when available. Figure 1 shows a 9 bit adder created on the VSC370 standard cells library of VLSI technology. This adder contains ad01d1 and ad02d1 cells which are respectively the macro full adders 1 and 2 bit of the VSC370 library, plus glue logic for the first and last bit cell.

![Figure 1: A 9 bit area adder](image)

1.1.2. Speed oriented adder

For the speed option, the carry is generated by a binary tree structure (conditional sum) to provide speed optimization. Figure 2 shows an 8 bit high speed adder. G and P are respectively the generation and propagation group terms used for the carry computation.

![Figure 2: An 8 bit speed adder](image)

In fact, according to the target technology, 2 kinds of conditional sum adders are used. For multiplexers based technology like Actel or Quicklogic, the generated adder is based on the Brent-Kung adder and for libraries based technology, the generated adder is based on the Von Neuman adder.

1.1.3. Particular case of LUT based technologies (XC4000 and FLEX8000)

Especially for LUT based technologies, for area as well as for speed criterion, PLS implements an N bit full adder using the fast carry logic.
For the FLEX8000 technology of Altera, the carry chain provides a very fast carry-forward function between logic elements. A 4 bit adder on the FLEX8000 technology is shown in figure 3.

For the XC4000 technology of Xilinx, the carry logic module CY4 allows the computation of a 2 bit addition in each CLB, by setting the "carry mode" to the "add" value. The carry logic module CY4 allows also the implementation of the carry input and the carry output value of the adder, by setting the "carry mode" respectively to "force" and "examine". A 4 bit adder constructed with CY4 modules of XC4000 technology is shown in figure 4.
**1.2. Subtractors**

For the subtractor, the same equations are used according to the synthesis criteria:
- area: ripple carry,
- speed: conditional sum.

Like for adders, in case of a speed option, for multiplexers based technology, the generated adder is based on the Brent-Kung adder and for libraries based technology, the generated adder is based on the Von Neuman adder.

To obtain a subtractor from an adder, inverting one operand plus possibly the input carry is performed. In fact the equation \(A-B\) is derived from \(A+(-B)\) where \((-B)\) is the two’s complement of \(B\) i.e. not \(B\) plus 1.
1.3. Incrementer/Decrementer

PLS synthesizes also incrementer/decrementer. The equations are the same than the ones for the adder/subtractor explained in the previous parts. According to the synthesis criteria, ripple carry equations are selected for area criterion and conditional sum equations for speed. Like for adder/subtractor, for speed option, Brent-Kung or Von Neuman based equations are used for respectively multiplexers or libraries based technologies.

More generally, when the operator is used with a constant operand (as for example A+4 or B>8), the constant is propagated in the equations for minimization purpose.

1.4. Comparator

1.4.1. Equality and inequality comparators

Independently of the synthesis criteria, only one type of equations is used for equality or inequality comparators.

For a N bit equality comparator, the equation is:
\[ S = \neg ( (A[0] \oplus B[0]) \lor (A[1] \oplus B[1]) \lor \ldots \lor (A[N-1] \oplus B[N-1]) ) \]

For a N bit inequality comparator, the equation is:
\[ S = ( (A[0] \oplus B[0]) \lor (A[1] \oplus B[1]) \lor \ldots \lor (A[N-1] \oplus B[N-1]) ) \]

These equations can be minimized according to 2 synthesis criteria which are area or speed.

1.4.2. Less than, less than or equal, greater than and greater than or equal comparators

For the less than, less than or equal, greater than and greater than or equal comparators, the equations used are the same than the ones for the subtractors explained in the first part. According to the synthesis criteria, ripple carry equations are selected for area criterion and conditional sum (Brent-Kung or Von Neuman) equations for speed.
1.5. Multiplier

In this version, only one multiplier is available in PLS: it is based on the Braun multiplier. It can be synthesized according to 2 options: area or speed. A 4 bit multiplier is given in figure 5.

A full adder elementary cell which computes an addition on 3 bit operands is used. This elementary cell is given in figure 6.
1.6. How to generate these operators

All the operators enumerated above can be generated by PLS for any bit width and for any target technology. They can either be inferred in VHDL using arithmetic operators: +, -, *, or relational operators: =, /=, <, <=, >, >=, or be called structurally using structural VHDL descriptions.

1.6.1. Inference call

Figure 7 shows an example of a VHDL file where the operators are inferred.

```vhdl
library ASYL;
use ASYL.ARITH.all;
entity MACRO1 is
port ( DATA1 : in BIT_VECTOR (7 downto 0);
       DATA2 : in BIT_VECTOR (15 downto 0);
       CLK : in BIT;
       OUT1 : out BIT_VECTOR (7 downto 0);
       OUT2 : out BIT_VECTOR (15 downto 0) );
end MACRO1;

architecture ARCH of MACRO1 is
signal S1: BIT_VECTOR (7 downto 0);
signal S2, S3: BIT_VECTOR (15 downto 0);
begin
  OUT1 <= S1;
  S3 <= S1 & DATA1;
  process (CLK)
  begin
    if CLK'EVENT and CLK='1' then
      S1 <= DATA1 + S1;
      OUT2 <= S2 - S3;
      S2 <= DATA2;
    end if;
  end process;
end ARCH;
```

Figure 7: Operators inference in VHDL

The resulting netlist is given in figure 8.
1.6.2. Structural call

Figure 9 shows an example of a VHDL file where the operators are called in a structural way. This example is equivalent to the one given in figure 7.

```vhdl
library ASYL;
use ASYL.ARITH.all;

entity ADD8 is
port ( IN1, IN2 : in BIT_VECTOR (7 downto 0);
       SUM : out BIT_VECTOR (7 downto 0) );
end ADD8;

architecture ARCH of ADD8 is
begin
  SUM <= IN1 + IN2;
end ARCH;

library ASYL;
use ASYL.ARITH.all;

entity SUB16 is
port ( IN1, IN2 : in BIT_VECTOR (15 downto 0);
       SUM : out BIT_VECTOR (15 downto 0) );
end SUB16;

architecture ARCH of SUB16 is
begin
  SUM <= IN1 - IN2;
end ARCH;
```
library ASYL;
use ASYL.ASYL_1164.all;

entity MACRO2 is
port ( DATA1 : in BIT_VECTOR (7 downto 0);
DATA2 : in BIT_VECTOR (15 downto 0);
CLK : in BIT;
OUT1 : out BIT_VECTOR (7 downto 0);
OUT2 : out BIT_VECTOR (15 downto 0) );
end MACRO2;

architecture ARCH of MACRO2 is
signal S1,S5 : BIT_VECTOR (7 downto 0);
signal S2,S3,S4 : BIT_VECTOR (15 downto 0);
component ADD8
port ( IN1,IN2 : in BIT_VECTOR (7 downto 0);
SUM : out BIT_VECTOR (7 downto 0) );
end component;
component SUB16
port ( IN1,IN2 : in BIT_VECTOR (15 downto 0);
SUM : out BIT_VECTOR (15 downto 0) );
end component;
for all : ADD8 use entity work.ADD8(ARCH);
for all : SUB16 use entity work.SUB16(ARCH);
begin
ADD : ADD8 port map (DATA1,S1,S5);
SUB : SUB16 port map (S2,S3,S4);
DFF_V (S5,CLK,S1);
DFF_V (S4,CLK,OUT2);
DFF_V (DATA2,CLK,S2);
OUT1 <= S1;
S3 <= S1 & DATA1;
end ARCH;

Figure 9: Operators inference in VHDL

The synthesized netlist is the same than the one given in figure 8.
2. Vendor macro blocks call

For any technology targets, it is very important to make a consistent use of library macro blocks. Especially for FPGA targets, the resynthesis of arithmetic operators would lead to poor results compared with designs using the library macro blocks. It is obvious that the FPGA vendors designed carefully the basic blocks and that for some targets placement and routing is pretty well handled or prepared for these blocks. Similarly in the ASIC world, parameterized generators are widely used and moreover data path compilers realize efficient bit slice abutment. Therefore it seems very important to realize a good link from a VHDL description to fully use library capabilities.

2.1. XBLOX macro blocks call

2.1.1. Structural call

All the XBLOX parameterized generators can be called in a structural style using VHDL component instantiations. All the corresponding VHDL components have been described in the “XB_BV”, “XB_LV” and the “XB_UV” packages corresponding respectively to the BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types. They have been compiled in the ASYL library. These packages are located in $ASYLDIR/vhdl/packages and are respectively named "xb_bv.vhd", "xb_lv.vhd" and "xb_uv.vhd". The available components are:

- ANDBUS
- ANDBUS1
- ANDBUS2
- INVBUS
- ORBUS
- ORBUS1
- ORBUS2
- XORBUS
- XORBUS1
- XORBUS2
- DECODE
- MUXBUS
- MUXBUS2
- MUXBUS4
- MUXBUS8
- ACCUM
- ADD_SUB
- COMPARE
- COUNTER
- TRISTATE
- DATA_REG
- SHIFT
- PROM
- SRAM
- BIDIR_IO
- INPUTS
- OUTPUTS
- CLK_DIV
- INC_DEC

To use them, the following VHDL lines must be placed in the VHDL description file before the entity declaration:

- for the BIT_VECTOR type:

  ```vhdl
  library ASYL;
  use ASYL.XB_BV.all;
  ```

- for the STD_LOGIC_VECTOR type:

  ```vhdl
  library ASYL;
  use ASYL.XB_LV.all;
  ```

- for the STD_ULOGIC_VECTOR type:

  ```vhdl
  library ASYL;
  use ASYL.XB_UV.all;
  ```

An example using 2 ADD_SUB and 3 DATA_REG components is shown in figure 10.
library ASYL;
use ASYL.XB_BV.all;

entity MACRO3 is
port (
  DATA1 : in BIT_VECTOR (7 downto 0);
  DATA2 : in BIT_VECTOR (15 downto 0);
  CMD : in BIT;
  CLK : in BIT;
  OUT1 : out BIT_VECTOR (7 downto 0);
  OUT2 : out BIT_VECTOR (15 downto 0)
);
end MACRO3;

architecture ARCH of MACRO3 is
begin
  ZERO <= '0';
  ONE <= '1';
  ADD : ADD_SUB
    generic map (N => 8, STYLE => ALIGNED)
    port map (DATA1,S1,ZERO,ONE,S5,open,open);
  ADDSUB : ADD_SUB
    generic map (N => 16, STYLE => ALIGNED)
    port map (S2,S3,ZERO,CMD,S4,open,open);
  R1 : DATA_REG
    generic map (N => 8, STYLE => IOB)
    port map (S5,ONE,CLK,ZERO,ZERO,S1);
  R2 : DATA_REG
    generic map (N => 16, STYLE => IOB)
    port map (S4,ONE,CLK,ZERO,ZERO,OUT2);
  R3 : DATA_REG
    generic map (N => 16, STYLE => CLB)
    port map (DATA2,ONE,CLK,ZERO,ZERO,S2);
  OUT1 <= S1;
  S3 <= S1 & DATA1;
end ARCH;

Figure 10: Structural calls of XBLOX parameterized generators

To use the XBLOX library the “xblox_m.lib” library description file must be specified in the synthesis command. This file is located in $ASYLDIR/lib/macrolib.

The resulting netlist is given in figure 11.
As shown in figure 10, the XBLOX attributes are given through the "generic map" clause. For each component, all the available attributes are defined as generic parameters in the corresponding VHDL component declaration. These attributes are set to default value. So when instantiating a XBLOX component, the attributes can be omitted: no "generic map" clause is needed in this case. But also, only some attributes can be specified as shown in figure 10.

For more details on the meaning and the possible values of each parameter, please refer to the Xilinx "X-Blox User Guide" and the corresponding PLS VHDL packages: xb_bv.vhd, xb_lv.vhd and xb_uv.vhd.

Note that the attribute names and the generic parameter names are the same except for the LOC[i] attribute which is referred by LOC_TAB in the generic parameter list of the components DATA_REG, SHIFT, BIDIR_IO, INPUTS, OUTPUTS, TRISTATE and ANDBUS. For more details, please refer to the VHDL packages: xb_bv.vhd, xb_lv.vhd and xb_uv.vhd.
2.1.2. Inference call

Some of the XBLOX components can be called by inference. These blocks are:

- ADD_SUB which is inferred by the arithmetic operators + or -.
- COMPARE which is inferred by the relational operators =, /=, >, >=, <, <=.
- INC_DEC which is inferred by the arithmetic operators + or - and which one operand is equal to 1.

The example of figure 12 shows how to infer ADD_SUB XBLOX component with arithmetic operators. The registers are called using component instantiations as showed in the previous section. Note that all the VHDL styles can be mixed in the same description.

```vhdl
library ASYL;
use ASYL.XB_BV.all;
use ASYL.ARITH.all;

entity MACRO4 is
  port (DATA1 : in BIT_VECTOR (7 downto 0);
        DATA2 : in BIT_VECTOR (15 downto 0);
        CMD : in BIT;
        CLK : in BIT;
        OUT1 : out BIT_VECTOR (7 downto 0);
        OUT2 : out BIT_VECTOR (15 downto 0));
end MACRO4;

architecture ARCH of MACRO4 is
  signal S1,S5 : BIT_VECTOR (7 downto 0);
  signal S2,S3,S4 : BIT_VECTOR (15 downto 0);
  signal ZERO, ONE : BIT;
begin
  ZERO <= '0';
  ONE <= '1';
  S5 <= DATA1 + S1;
  S4 <= S2 + S3 when CMD = '1' else S2 - S3;
  R1 : DATA_REG
    generic map (N => 8)
    port map (S5,ONE,CLK,ZERO,ZERO,S1);
  R2 : DATA_REG
    generic map (N => 16)
    port map (S4,ONE,CLK,ZERO,ZERO,OUT2);
  R3 : DATA_REG
    generic map (N => 16)
    port map (DATA2,ONE,CLK,ZERO,ZERO,S2);
  OUT1 <= S1;
  S3 <= S1 & DATA1;
end ARCH;
```

Figure 12: Inference and structural calls of XBLOX parameterized generators

To use the XBLOX library the “xblox_m.lib” library description file must be specified in the synthesis command. This file is located in $ASYLDIR/lib/macrolib.

The resulting netlist is the same then the one given in figure 11.
Note that when inferring an XBLOX generator, the attributes take into account are the number of bits and the style of the generator. The other attributes take a default value.

2.2. XC3000 and XC4000 macro blocks call

XC3000 and XC4000 macro blocks can also be called similarly to the XBLOX macro blocks. For these libraries, a macro block has been defined as a block whose bit width is greater than or equal to 4.

In a structural VHDL description all the XC3000 and XC4000 macro blocks can be called using component instantiations.

For XC3000 macro blocks, all the corresponding VHDL components have been described in the “xc3_bv.vhd”, “xc3_lv.vhd” and “xc3_uv.vhd” packages, compiled in the ASYL library, for respectively BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types.

For XC4000 macro blocks, all the corresponding VHDL components have been described in the “xc4_bv.vhd”, “xc4_lv.vhd” and “xc4_uv.vhd” packages, compiled in the ASYL library, for respectively BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types.

These packages are located in $ASYLDIR/vhdl/packages. To use these components, the following VHDL lines must be placed in the VHDL description file before the entity declaration:

- for XC3000 macro blocks call, using BIT_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC3_BV.all;
  ```

- for XC3000 macro blocks call, using STD_LOGIC_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC3_LV.all;
  ```

- for XC3000 macro blocks call, using STD_ULOGIC_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC3_UV.all;
  ```

- for XC4000 macro blocks call, using BIT_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC4_BV.all;
  ```

- for XC4000 macro blocks call, using STD_LOGIC_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC4_LV.all;
  ```

- for XC4000 macro blocks call, using STD_ULOGIC_VECTOR types:
  
  ```vhdl
  library ASYL;
  use ASYL.XC4_UV.all;
  ```

The adders/subtractors and comparators can be called by inference using the VHDL operators +, -, =, /=, <, <=, >, >=.

To use the XC3000 or XC4000 macro blocks library the “xc3000_m.lib” or respectively “xc4000_m.lib” library description file must be specified in the synthesis command. These files are located in $ASYLDIR/lib/macrolib.
2.3. ACTgen Macro Builder Call

The ACTgen Macro Blocks generated by ACTgen Macro Builder can also be called from PLS when ACT2 or ACT3 technology is selected. Available macros are COUNTERs, MULTIPLEXORs, DECODERs and ADDERs. These macros can be only called by inference from VHDL specification.

2.3.1. Software Requirements

In order to call the ACTgen Macro Builder, the user must define an environment variable “ALSDIR”, which corresponds to the directory were ALS software has been installed. Thus, PLS will look for the ACTgen Macro Builder shell program, called “actgen”, located in “$ALSDIR/bin” directory. Please make sure that this program is available.

2.3.2. Inferring ACTgen Counters

PLS is able to recognize counters from VHDL specification and call ACTgen Macro Builder to generate an optimized counter which will be included in the final design.

Several counters can be inferred with various functions: asynchronous reset, synchronous load, clock enable and up-down count facilities. The general counter template is shown in figure 13.

```
library ASYL;
use ASYL.ARITH.all;

entity COUNTER is
  port ( DATA_IN : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         RST : in BIT;
         EN : in BIT;
         SLOAD : in BIT;
         UPDOWN : in BIT;
         DATA_OUT : inout BIT_VECTOR (7 downto 0) );
end COUNTER;

architecture ARCH of COUNTER is
begin
  p : process (CLK, RST)
  begin
    if (RST = <'0','1'>) then
      DATA_OUT <= "00000000";
    elsif (CLK'EVENT and CLK = <'0','1'>) then
      if (EN = <'0','1'>) then
        if (SLOAD = <'0','1'>) then
          DATA_OUT <= DATA_IN;
        elsif (UPDOWN = <'0','1'>) then
          DATA_OUT <= DATA_OUT + '1';
        else
          DATA_OUT <= DATA_OUT - '1';
        end if;
      end if;
    end if;
  end process p;
end ARCH;
```

Figure 13: General counter template
The user may use these four functions to design his own counter; for instance, a customized counter with synchronous load and up-down facility is shown in figure 14.

```vhdl
library ASYL;
use ASYL.ARITH.all;

entity COUNTER is
  port ( DATA_IN : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         UPDOWN : in BIT;
         SLOAD : in BIT;
         DATA_OUT : inout BIT_VECTOR (7 downto 0));
end COUNTER;

architecture ARCH of COUNTER is
begin
  p : process (CLK)
  begin
    if (CLK'EVENT and CLK = '1') then
      if (SLOAD = '0') then
        DATA_OUT <= DATA_IN;
      elsif (UPDOWN = '0') then
        DATA_OUT <= DATA_OUT + '1';
      else
        DATA_OUT <= DATA_OUT - '1';
      end if;
    end if;
  end process p;
end ARCH;
```

Figure 14: Up-down counter with synchronous load

When recognition of the counter is performed, PLS prepares data file for ACTgen Macro Builder and calls this program which will generate the requested element; then the EDIF netlist generated by ACTgen Macro Builder is automatically read and merged in the final design.

PLS is able to call the three following ACTgen Counters: “Balanced” for Area Optimization, “FastEnable” when Enable signal is used and “Prescaled” for Speed Optimization.
Macro+

2.3.3. Inferring ACTgen Adders and Subtractors

PLS is also able to infer Adders and Subtractors and call ACTgen Macro builder to generate these elements. Three variations of adders (respectively subtractors) are available: “Fast Carry” for Speed Optimization, “Medium Fast Carry” for trade-off and “Ripple Carry” for Area Optimization.

The VHDL example shown in figure 15 illustrates the inference for Subtractors. The resulting netlist is shown in figure 16.

```vhdl
library ASYL;
use ASYL.ARITH.all;

entity SUBTRACTOR is
  port ( DATA1 : in BIT_VECTOR (18 downto 0);
         DATA2 : in BIT_VECTOR (18 downto 0);
         DATA_OUT : out BIT_VECTOR (18 downto 0) );
end SUBTRACTOR;

architecture ARCH of SUBTRACTOR is
begin
  DATA_OUT <= DATA1 - DATA2;
end ARCH;
```

Figure 15: Subtractor’s inference

![Subtractor's inference](image)

![Resulting netlist](image)
2.3.4. Inferring ACTgen Multiplexors

PLS is able to infer multiplexors and call ACTgen Macro Builder to generate such elements. Any width from 2 to 32 is allowed with busses of up to 16 bits for every input of the multiplexors.

There are three main templates to infer multiplexors in VHDL with PLS and generate them with ACTgen Macro Builder.

The first template uses “if...elsif...else” VHDL statements. An example is shown in figure 17.

```
entity MULTIPLEXOR is
port ( DATA0 : in BIT_VECTOR (11 downto 0);
      DATA1 : in BIT_VECTOR (11 downto 0);
      DATA2 : in BIT_VECTOR (11 downto 0);
      DATA3 : in BIT_VECTOR (11 downto 0);
      DATA4 : in BIT_VECTOR (11 downto 0);
      DATA5 : in BIT_VECTOR (11 downto 0);
      DATA_OUT : out BIT_VECTOR (11 downto 0);
      A : in BIT_VECTOR (2 downto 0);
      B : in BIT_VECTOR (1 downto 0);
      C : in BIT;
      D : in BIT);
end MULTIPLEXOR;

architecture ARCH of MULTIPLEXOR is
begin
  P : process(A, B, C, D, DATA0, DATA1, DATA2, DATA3,
             DATA4, DATA5)
  begin
    if A(0)='1' then
      DATA_OUT <= DATA0;
    elsif B="00" then
      DATA_OUT <= DATA1;
    elsif C='1' then
      DATA_OUT <= DATA2;
    elsif A(2 downto 1)="11" then
      DATA_OUT <= DATA3;
    elsif D='0' then
      DATA_OUT <= DATA4;
    else
      DATA_OUT <= DATA5;
    end if;
  end process;
end ARCH;
```

Figure 17: Multiplexor’s inference

A 6 to 1 multiplexors will be generated by ACTgen. Some logic will be generated and connected to the selection port of the multiplexor in order to select the right inputs according the value of signals A, B, C and D. The resulting netlist is given in figure 18.
The second template uses "with...select" VHDL statements to infer multiplexers. An example is shown in figure 19.

```vhdl
entity MULTIPLEXOR is
    port(
        DATA0 : in BIT_VECTOR (11 downto 0);
        DATA1 : in BIT_VECTOR (11 downto 0);
        DATA2 : in BIT_VECTOR (11 downto 0);
        DATA3 : in BIT_VECTOR (11 downto 0);
        DATA4 : in BIT_VECTOR (11 downto 0);
        DATA5 : in BIT_VECTOR (11 downto 0);
        DATA6 : in BIT_VECTOR (11 downto 0);
        DATA_OUT : out BIT_VECTOR (11 downto 0);
        SEL : in BIT_VECTOR (2 downto 0);
    );
end MULTIPLEXOR;

architecture ARCH of MULTIPLEXOR is
begin
    with SEL select
    begin
        DATA_OUT <= DATA6 when "110" | "111",
                   DATA2 when "010",
                   DATA1 when "001",
                   DATA5 when "101",
                   DATA0 when "000",
                   DATA3 when "011",
                   DATA4 when others;
    end with;
end ARCH;

Figure 18: Resulting netlist

Figure 19: Multiplexor's inference
Macro+

With this template PLS infers a 8 to 1 multiplexor. The signal SEL is connected to the selection port directly and the inputs are connected according to the corresponding value. The figure 20 shows the resulting netlist.

![Figure 20: Resulting netlist](image)

To avoid waste of area, PLS detects if it is worth inferring a multiplexer this way: if there are not enough inputs compared to the number of selection allowed by the selection signal, PLS will generate a multiplexor with the number of inputs used but with some logic to build the selection port.

The figure 21 gives an example of a VHDL description where such a situation occurs.

```vhdl
entity MULTIPLEXOR is
  port ( DATA0 : in BIT_VECTOR (11 downto 0);
         DATA1 : in BIT_VECTOR (11 downto 0);
         DATA2 : in BIT_VECTOR (11 downto 0);
         DATA_OUT : out BIT_VECTOR (11 downto 0);
         SEL : in BIT_VECTOR (2 downto 0) );
end MULTIPLEXOR;

architecture ARCH of MULTIPLEXOR is
begin
  with SEL select
  DATA_OUT <= DATA0 when "000",
             DATA1 when "111",
             DATA2 when others;
end ARCH;
```

Figure 21: Multiplexer’s inference
A 3 to 1 multiplexor will be inferred instead of a 8 to 1 as the surface of an over sized multiplexor will be larger than the surface of logic generated for the selection port.

The resulting netlist is shown in figure 22.

![Multplexor Diagram](image)

**Figure 22: Resulting netlist**

The third template uses the VHDL syntactical statement "case...when". This template is rather equivalent to the previous one as it requires a single signal to select input ports.

The VHDL example of the Figure 23 generates the same netlist as the example of the Figure 19.
entity MULTIPLEXOR is
port ( DATA0 : in BIT_VECTOR (11 downto 0);
      DATA1 : in BIT_VECTOR (11 downto 0);
      DATA2 : in BIT_VECTOR (11 downto 0);
      DATA3 : in BIT_VECTOR (11 downto 0);
      DATA4 : in BIT_VECTOR (11 downto 0);
      DATA5 : in BIT_VECTOR (11 downto 0);
      DATA6 : in BIT_VECTOR (11 downto 0);
      DATA_OUT : out BIT_VECTOR (11 downto 0);
      SEL : in BIT_VECTOR (2 downto 0) );
end MULTIPLEXOR;

architecture ARCH of MULTIPLEXOR is
begin
P : process (DATA0, DATA1, DATA2, DATA3, DATA4,
             DATA5, DATA6, SEL)
begin
  case SEL is
    when "110" | "111" =>
      DATA_OUT <= DATA6;
    when "010" =>
      DATA_OUT <= DATA2;
    when "001" =>
      DATA_OUT <= DATA1;
    when "101" =>
      DATA_OUT <= DATA5;
    when "000" =>
      DATA_OUT <= DATA0;
    when "011" =>
      DATA_OUT <= DATA3;
    when others =>
      DATA_OUT <= DATA4;
  end case;
end process;
end ARCH;

Figure 23: Multiplexor’s inference

Remark:
If a constant value is given for a data input, the macro block is not inferred as a lot of
simplifications are possible in the netlist due to this constant value. It is not worth
inferring a macro block with constant vector connected to its inputs.
2.3.5. Inferring ACTgen Decoders

The decoder’s inference uses the same templates as the multiplexor’s one: “if..elsif..else” , “with..select” and “case..when” templates are allowed. Nevertheless, to infer a decoder a one-hot encoding is required for the output. The one-hot bit can be active high or low (values as 00010 or 11101 are allowed). A width of up to 32 bits is allowed.

Figure 24 gives a VHDL example of a decoder inference.

```vhdl
entity DECODER is
  port ( DATA_OUT : out BIT_VECTOR (8 downto 0);
         SEL : in BIT_VECTOR (2 downto 0) );
end DECODER;

architecture ARCH of DECODER is
begin
  with SEL select
    DATA_OUT <= "000000001" when "111",
              "100000000" when "000" | "001",
              "001000000" when "100",
              "000100000" when "101",
              "000010000" when "010",
              "000001000" when "011",
              "000000100" when others;
end ARCH;
```

Figure 24: Multiplexor’s inference

The resulting netlist is shown below, in figure 25.

Figure 25: Resulting netlist
An enable can be added to the decoder. The enable can be active high or low. A value can be affected to the output when the decoder is disabled (enable not active). This value must be the value imposed by ACTgen: all the bits of the output port are inactive.

An example is shown in figure 26.

```vhdl
entity DECODER is
port ( DATA_OUT : out BIT_VECTOR (8 downto 0);
      SEL : in BIT_VECTOR (2 downto 0);
      EN : in BIT);
end DECODER;

architecture ARCH of DECODER is
begin
P : process (EN, SEL)
begin
  if EN='1' then
    if SEL="111" then
      DATA_OUT <= "000000010";
    elsif SEL="000" or SEL="001" then
      DATA_OUT <= "100000000";
    elsif SEL="100" then
      DATA_OUT <= "001000000";
    elsif SEL="110" then
      DATA_OUT <= "000100000";
    elsif SEL="010" then
      DATA_OUT <= "000001000";
    elsif SEL="101" then
      DATA_OUT <= "000000001";
    else
      DATA_OUT <= "010000000";
    end if;
  else
    DATA_OUT <= "000000000";
  end if;
end process;
end ARCH;
```

Figure 26: Decoder's inference

2.3.6. How to use ACTgen Macros

In order to used ACTgen Macro Builder, the “actgen_m.lib” library description file must be specified in the synthesis command. This file is located in the “$ASYLDIR/lib/macrolib” directory.

2.3.7. Hints for synthesis

One might consider that for combinatorial elements as multiplexers or decoders the synthesis with ACTgen provides good results only for broad elements. Special logic is generated in addition to the inferred element and in few cases results with ACTgen could be worth than results without macros blocks selection. Thus a netlist optimization should be performed after VHDL synthesis, in order to reduce the area or the critical path of the design.
2.4. ACT1, ACT2 and ACT3 macro blocks call

ACT1, ACT2 and ACT3 macro blocks can also be called similarly to the ACTgen macro blocks. For these libraries, a macro block has been defined as a block whose bit width is greater than or equal to 4.

In a structural VHDL description all the ACT1, ACT2 and ACT3 macro blocks can be called using component instantiations.

For ACT1 macro blocks, all the corresponding VHDL components have been described in the “act1_bv.vhd”, “act1_lv.vhd” and “act1_uv.vhd” packages, compiled in the ASYL library, for respectively BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types.

For ACT2 macro blocks, all the corresponding VHDL components have been described in the “act2_bv.vhd”, “act2_lv.vhd” and “act2_uv.vhd” packages, compiled in the ASYL library, for respectively BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types.

For ACT3 macro blocks, all the corresponding VHDL components have been described in the “act3_bv.vhd”, “act3_lv.vhd” and “act3_uv.vhd” packages, compiled in the ASYL library, for respectively BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR types.

These packages are located in $ASYLDIR/vhdl/packages. To use these components, the following VHDL lines must be placed in the VHDL description file before the entity declaration:

- for ACT1 macro blocks call, using BIT_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT1_BV.all;
  ```

- for ACT1 macro blocks call, using STD_LOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT1_LV.all;
  ```

- for ACT1 macro blocks call, using STD_ULOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT1_UV.all;
  ```

- for ACT2 macro blocks call, using BIT_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT2_BV.all;
  ```

- for ACT2 macro blocks call, using STD_LOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT2_LV.all;
  ```

- for ACT2 macro blocks call, using STD_ULOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT2_UV.all;
  ```

- for ACT3 macro blocks call, using BIT_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT3_BV.all;
  ```

- for ACT3 macro blocks call, using STD_LOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT3_LV.all;
  ```

- for ACT3 macro blocks call, using STD_ULOGIC_VECTOR types:
  ```vhdl
  library ASYL;
  use ASYL.ACT3_UV.all;
  ```
The adders/subtractors and comparators can be called by inference using the VHDL operators +, -, =, /=, <, <=, >, >=.

To use respectively the ACT1, ACT2 or ACT3 macro blocks library, the “act1_m.lib”, “act2_m.lib” or “act3_m.lib” library description file must be specified in the synthesis command. These files are located in $ASYLDIR/lib/macrolib.
3. Predesigned block call

Most of the designers use a meet-in-the-middle design methodology. This means that they design some dedicated, efficient basic blocks and they assemble or use them in a more complex design. This section explains how to call predesigned blocks.

The user can call predesigned blocks using component instantiation in a structural VHDL description. In this case the component must not be configured (or configured only with "use entity <library_name>.<entity_name>;"). In this case a black box is created. Its name is the name of the component (or entity) and its ports are the ones of the component (or entity) with the same names. Figure 27 gives an example of such a call.

library ASYL;
use ASYL.ASYL_1164.all;
use ASYL.ARITH.all;

entity MACRO6 is
  port ( DATA1, DATA2 : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         OUT1 : out BIT_VECTOR (7 downto 0) );
end MACRO6;

architecture ARCH of MACRO6 is
  signal S1,S2,S3,S4 : BIT_VECTOR (7 downto 0);
  component MY_BLOCK
    port ( A,B,C : in BIT_VECTOR (7 downto 0);
           D,E : out BIT_VECTOR (7 downto 0) );
  end component;
begin
  C1 : MY_BLOCK
    port map (S1,DATA2,S2,S3,S4);
  DFF_V (DATA1,CLK,S1);
  S2 <= S3 + S4;
  OUT1 <= S2;
end ARCH;

Figure 27: Predesigned block call by component instantiation

The example MACRO6 described in figure 27 is made up a register, a predesigned block named MY_BLOCK and an adder. The created black box is named MY_BLOCK and it has three input ports named A, B and C and two output ports named D and E. The resulting netlist is given in figure 28.
Figure 28: Synthesized netlist
Appendix 1:

Macro Block Directive File Syntax

\[
\text{<DirectiveOperator> } \rightarrow \text{ OPERATOR <Idf> <Line> <Index>}
\]
\[
\text{<Synthesis> END}
\]
\[
\text{<Line> } \rightarrow \text{ [ LINE <Number> [ '; ' <Number>]* [ LINE All ]}
\]
\[
\text{<Synthesis> } \rightarrow \text{ <SynthesisLibrary> | <SynthesisBlackBox>}
\]
\[
\text{<SynthesisLibrary> } \rightarrow \text{ LIBRARY <Idf> [ PATH <Idf> ]}
\]
\[
\text{ [ COMPONENT <Idf> ]}
\]
\[
\text{<SynthesisBlackBox> } \rightarrow \text{ BLACKBOX <Idf>}
\]
\[
\text{<Idf> } \rightarrow \text{ <Letter> [ <Letter> | <Digit> ]*}
\]
\[
\text{<Number> } \rightarrow \text{ <Digit> +}
\]
\[
\text{<Letter> } \rightarrow \text{ A | ... | Z | a | ... | z | _}
\]
\[
\text{<Digit> } \rightarrow \text{ 0 | ... | 9}
\]

where:

\[
<...> \text{ : Terminal name}
\]
\[
--> \text{ : Is derived in}
\]
\[
| \text{ : Alternatives}
\]
\[
[[...]] \text{ : Optional element}
\]
\[
... * \text{ : 0, 1 or several occurrence(s)}
\]