FUNCTIONAL SIMULATION USER'S MANUAL

O. Hol

Circuits and Systems Group
Department of Electrical Engineering
Delft University of Technology
The Netherlands

Date: September, 1993.
Last revision: May, 2006.

1. INTRODUCTION

In this manual it is explained how function blocks can be used in the Switch-Level Simulator SLS. Before a user can do this, he or she should be familiar with the SLS simulator. Therefore in this manual the SLS User's Manual [1] is assumed to be generally known.

The simulator SLS has been extended to a Mixed Level Simulator that can be used for simulations on functional level down to switch-level. This extension has been created by allowing the use of function blocks in networks. In SLS (as Switch-Level Simulator) a network can be described hierarchically: subnetworks can be called in networks. Now in the Mixed-Level Simulator SLS, a subblock can also be a function block, instead of a network. When a network consists of only function blocks, a functional simulation can be performed. When both function blocks and transistor-level parts or switch-level subnetworks exist in a network, mixed-level simulations result. And of course the possibility still exists to use SLS purely as a Switch-Level Simulator.

This new feature of SLS, function blocks, very well supports the top-down as well as bottom-up design methods. In the top-down method a design first is partitioned in a few general functional blocks. These blocks are divided into subblocks that can be described on several different levels. These subblocks can also be refined until the lowest level is reached. In the bottom-up approach one or more subblocks are replaced by a block or a higher level, obtaining a more abstract description.

A function block is a software model of a digital design. It manipulates its inputs and produces outputs. In the simulator the model of a function block is such that a good connection exists with the switch-level parts in a network. This is the case in logic as well as timing simulations. The language in which the function blocks is described is mainly the C programming language. For most of the function blocks the language only needs to be used in a simple manner. This manual can be used as a guide in doing so, even when the designer is not familiar with the C language. Section 2 in this manual gives an overview of the extended SLS package of programs. Section 3 describes the model of the function block and the syntax of the description. Section 4 explains some ways to use function blocks in simulations.

2. DESCRIPTION OF THE EXTENDED SLS-PACKAGE

The extended simulator SLS consists of the programs used in the Switch-Level Simulator plus one more program. The use of the package furthermore changes due to the fact that an executable simulator is created whenever a new function block is used in a network. In figure 1 the extended SLS package is shown.

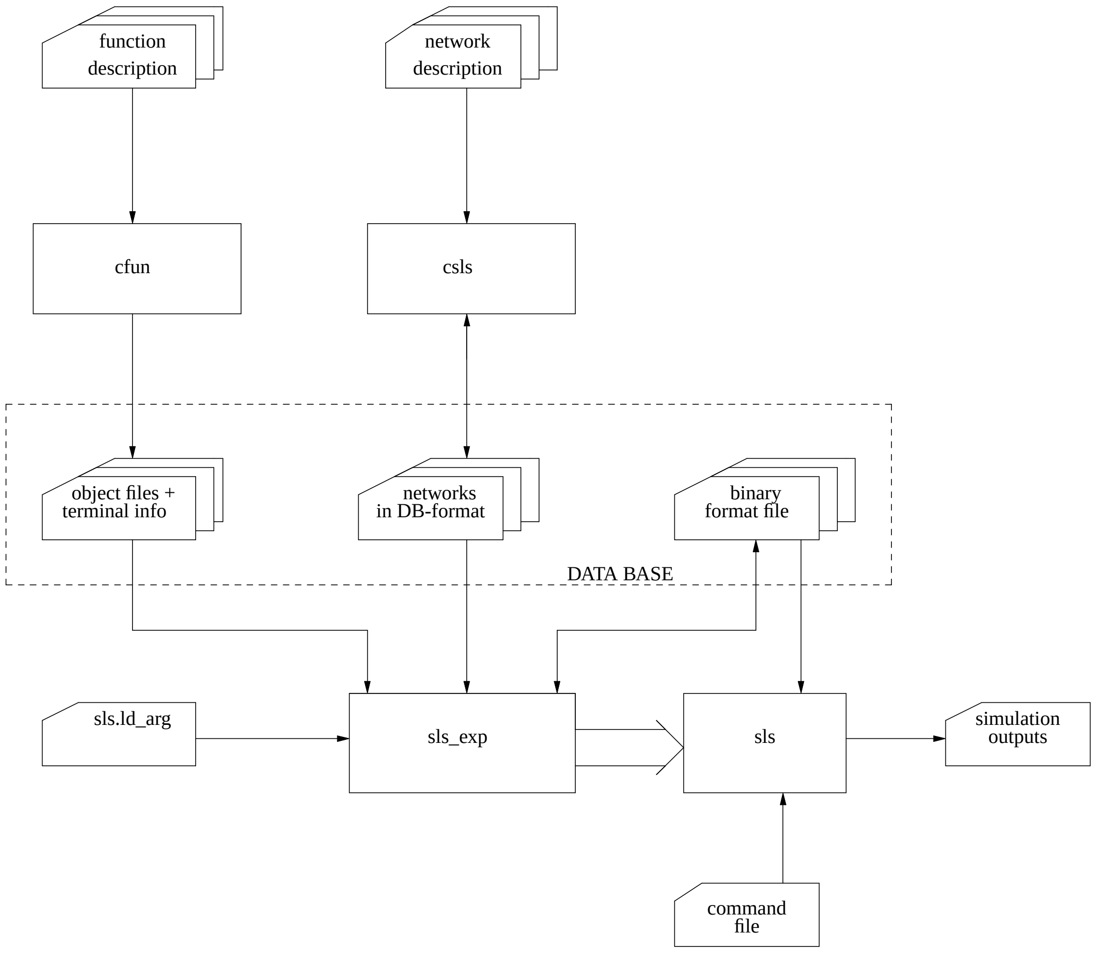


Figure 1. The extended SLS package

The extended SLS-package involves four programs

1. *cfun* maps a function block description contained in one file into a function block description in database format. The function block description is translated into C code, the C code is compiled and the object file is placed into the database.
2. *csls* maps a hierarchical *sls* network description, contained in one or several files into a network description in database format.
3. *sls_exp* maps the network description into a binary (i.e non readable) file which serves as input file for the *sls* simulator. Furthermore, when the network contains calls to function blocks, *sls_exp* creates an executable *sls* that contains the behavioral description of these function blocks. Text that is included in the file ’sls.ld_arg’ is used as argument when the executable *sls* is created.
4. *sls* is the created executable simulator program that can be used similar to the original switch-level simulator.

Once a binary format file has been obtained for a network and an executable simulator exists, different simulations can be done without the time-consuming function block compilation and linking step. A new executable will be created by *sls_exp* only when the network contains a function block of which the compiled description is not yet linked to the existing executable, or when this function block is newer than the existing executable.

3. DESCRIPTION OF THE FUNCTION BLOCKS

3.1 The function block model

In figure 2, the model of the function block is shown.

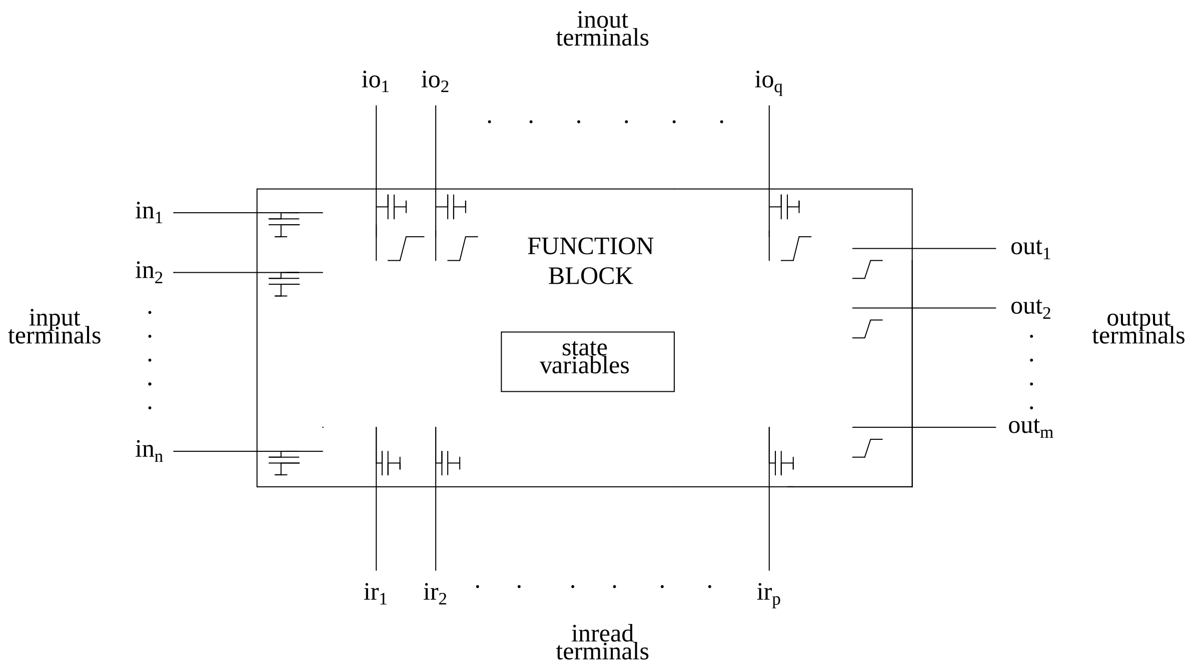


Figure 2. Model of the function block

The function block itself can be used in only one way: it gets data as input and produces data as output similar to a software routine. This routine is executed **only when one or more of its input terminals change state**. The function block must be placed as a subblock in a network. This network contains nodes that are connected with the terminals of the function block. The logic state of these nodes determine the actions of the function block. Nodes in the network that are connected with a terminal of a function block serve as input or output (or both) data for the function block. The nodes that are connected with an input terminal of a function block determine the timing of the executions of the function block in the network. In section 4, more will be explained about the function blocks in an SLS network, in this section the function block itself will be regarded.

Looking at the model, four kinds of terminals can be distinguished. Apart from the terminals, a function block can contain state variables as well. The several types of terminals and the state variables of the function block must be used in the following sense:

— input terminals: these terminals are used for two purposes. First these terminals act as ’triggers’ for the function block; whenever an input terminal changes state, the function block will be evaluated. Second the data on these terminals is used as input data during the evaluation of the function block.

— inread terminals: these terminals only serve as input data during the evaluation of the function block.

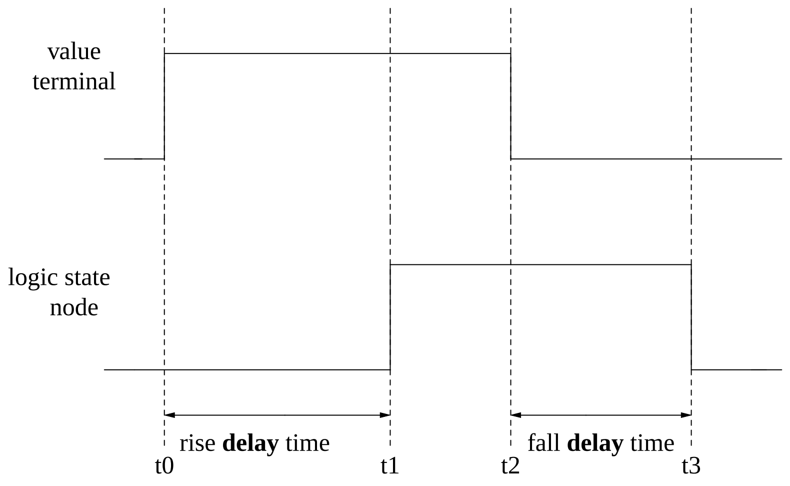
— output terminals: these terminals are calculated during an evaluation of the function block.

— inout terminals: these terminals are bidirectional terminals. At the beginning of an evaluation of the function block these terminals get the value (logic state) of the nodes in the network they are connected with. This data can be used in the evaluation of the function. Then, the values of these terminals can be changed by the function block and so they serve as output terminals.

— state variables: these variables are local to every instance of a function block and they are used to store the state of that instance. The variables can be of several types and their values can only be changed during an evaluation of the instance of the function block.

The terminals can have a zero-, one- or two-dimensional array structure. Because the terminals of a function block are connected with nodes in a network, the values these terminals can take are exactly the values the nodes can take: logic states. The logic state of a node in a Switch-Level network can be O, X or I. Therefore the value of a terminal of a function block can also be O, X, or I. In the description of a function block, which is made in the C programming language, the implementation of a terminal is a character (’char’). This character can take the value ’O’ (capital o), ’I’ (capital i) or ’X’ (capital x). With these values as input data, the software routine that forms a function block can produce output data of the same type. Moreover, an output or inout terminal can be given the value ’F’ (capital f) which means the ’free’ value. This value is the so called third state providing a high impedance mode, in which the output terminal appears as if it was disconnected from the function block. This value will have as result that the output terminal will **not have any influence on the logic state of the node it is connected with**.

In the picture of the function block model, inside the block capacitances and ’slopes’ can be found. The ’slopes’ in the picture indicate that the delay times of an inout or output terminal can be defined inside the function block. Doing so, the timing properties of the function block are defined. The delay times are the times between the moment of change of the inout or output terminal of the function block and the moment this change will have effect on the logic state of the node in the network that is connected with such a terminal. The value of an inout or output terminal changes immediately when the function block is evaluated, that is when the value of at least one of the **input** terminals changes. In figure 3 this mechanism is further illustrated.



t0	function evaluated as a result of the change of logic state of one its input terminals
t1	node changes logic state O -> I as a result of function evaluation
t2	function evaluated
t3	node changes logic state I -> O

Figure 3. State of a network node connected with one inout or output terminal

The capacitances that are shown in the function block model indicate that it is possible to define the input load of the function block. This input load has influence on the rest of the network the function block is part of. Within the function block description this input load can be defined by assigning capacitances to input, inread and inout terminals. These capacitances will be added to the capacitances of the nodes in the network that are connected with such terminals. In that way the capacitances of these nodes will account for the input load of the function block.

3.2 The syntax of the function model description

The meta language as proposed by Wirth, and the general conventions as they are defined in section 3.1. of the SLS User’s Manual, are used to describe the syntax of a function block description.

The reserved keywords are:

TABLE 1. Reserved keywords in function block descriptions

behavior	function	initial
inout	input	inread
load	output	state

3.2.1 Function Block Description

TABLE 2. Syntax of the function block part

function_decl	= *function identifier* decl_part funct_body

The function block part begins with the keyword *function* followed by the name of the function block, a declaration part describing the interface to the outside world and the local state variables, and a body specifying the initial and general behavior of the function block.

3.2.2 Declarations

TABLE 3. Syntax of the declaration part

decl_part	= term_decls [state_decls]
term_decls	= "(" term_decl {";" term_decl} ")"
term_decl	= term_type term {"," term }
term_type	= *input* | *inread* | *inout* | *output*
state_decls	= *state* "{" state_term {";" state_term } "}"
state_term	= state_type term {"," term }
state_type	= *char* | *int* | *float* | *double*
term	= *identifier* ["[" *integer* "]" ["[" *integer* "]"]]

The terminal declaration part is bracketed by a left and right parenthesis. It must contain at least one terminal declaration. A terminal declaration begins with one of the keywords *input, inread, inout* or *output* followed by a list of terminals. A list of terminals consists of one or more terminals separated by commas. A terminal can be zero-, one- or two-dimensional.

Examples:

```
function counter ( input phi, phi2;
                   inread in[8], load, up;
                   inout out[8], tc )

function registers ( output out[4][4];
                     inread reg, cntrl[2][2];
                     input phi, phi2;
                     inread in[2][4] )
state {
    char  reg_p1[2][8];
    char  reg_p2[2][8];
    int   ch_p1;
}
```

3.2.3 Function block behavior

TABLE 4. Syntax of the function block body

funct_body	= [load_body] [init_body] behav_body
load_body	= *load* "{" C_code "}"
init_body	= *initial* "{" C_code "}"
behav_body	= *behavior* "{" C_code "}"
C_code	= any C_code as may exist in the function body of a C function without declaration list

The body of the function block consists of three parts: an optional load part, an optional initial part and an obligatory behavior part. All parts contain normal C-code; the load part and the initial part are executed once at the start of the simulation (first the load part of all functions, next the initial part of all functions), the behavior part is executed every time the value of an input terminal changes. The load part is normally used to specify the terminal capacitances of the function block. The initial part is often used to specify the delay times of the function block and to initialize state variables. In both parts the setting of the output terminals has no effect on the rest of the network. In the behavior part of the function block, the values of the state variables and the values of the output terminals are specified, as well as the capacitances and delay times may be changed.

Examples:

```
load {
    int i;
    for (i=0; i<4; i++)
        cap_add (50e-15, in[i]);
}

initial {
    int i;
    for (i=0; i<4; i++)
        reg[i] = BTTRUE;
}

behavior {
    if (phi1 == BTTRUE) {
          if (load == BTTRUE)
              BSCOPY (reg, in);
          else
              BSCOPY (reg, BSROTATE (reg));
    }
    if (phi2 == BTTRUE)
          BSCOPY (out, reg);
}
```

In the following sections the C-code, auxiliary routines and other relevant subjects will be discussed.

3.3 The C programming language

In this manual only a brief introduction to the C programming language is given. For a more comprehensive explanation, see the book "The C Programming Language" by B.W. Kernighan and D.M. Ritchie. Here, by describing some basic ideas of the C language together with the discussion of several examples, a first start can be made with the use of the C language. Some familiarity is assumed with basic programming concepts like variables, assignment statements, loops and functions.

The C-code

In the previous section the syntax of the function block body has been described. It was stated that in the load, initial and behavior blocks C-code has to be used, completely similar to any C-code as may exist in the function body of a C function, without declaration-list. This has also been shown in the example. Moreover, because the function block description is to be inputed to a parser, some extra possibilities have been created in the use of the C language. These possibilities will be discussed in the next section. Here we discuss one particular aid that must be known for a better understanding of the several examples. This aid is the use of the values BTTRUE, BTFALSE, BTUNDEF and BTFREE which already appeared in the example. The definition of these values is as follows:

```
#define BTTRUE  ’I’
#define BTFALSE ’O’
#define BTUNDEF ’X’
#define BTFREE  ’F’
```

Since the terminals of the function block are of type character the value of a terminal can be defined using these ’defines’. Moreover it is much more clear for reading to use the line:

```
if (phi1 == BTTRUE)
```

then the line

```
if (phi1 == ’I’)
```

Therefore, in the rest of this manual these ’defines’ will always be used.

Variables and types

The terminals that have been declared for a function block can be used as variables in the C-code. These variables are of the type character (char), or they are a one- or two-dimensional array of type char. A one-dimensional array of characters is a string; a two-dimensional array of characters is an array of strings. So, if a terminal declaration looks like:

```
function shift ( input  phi;
                 inread select, in[2][5];
                 output out[5] )
```

then the terminals can for example be used in the following way:

```
{
    int i,j;
    if (phi == BTTRUE) {
        if (select == BTFALSE)
            j = 0;
        else
            j = 1;
        for (i=4; i>0; i--)
            out[i] = in[j][i-1];
        out[0] = in[j][4];
    }
}
```

This piece of C-code means: if the input terminal *phi* has the value value ’I’ (a character), then the following will be executed: if the inread terminal *select* has the value ’O’ (a character), then the first string of the two-dimensional character array *in* will be used (local integer variable *j* will get the value 0), else the second string will be used; then the for-loop and the assignment statement will be executed. In the for-loop variable *i* starts with value 4 and is decremented until it is 0, which is not >0. The string *out* (one-dimensional array of characters) is filled with the characters of the *j’th* string of array *in*.

Notice that a one-dimensional array with 5 elements has indices: 0,1,2,3,4: generally an array with n elements has indices: 0,1,...,n-1. Furthermore, notice that the variables *i* and *j* that are used have to be declared, like in a normal C function body. The scope of these variables is the behavior block they are used in. Other variables of all kind of types can be used in the C-code of the load as well as the initial and behavior block whenever they have been declared locally like variables *i* and *j*.

The state variables that have been declared for a function block can also be used as variables in the C-code of a load, initial or behavior block. These variables can be of types char, int, float or double. For every instance of the function block the state variables are local to that instance; for every instance apart, the variables may change value within the load, initial or behavior block and keep these values until they are changed again. State variables are used in a function block similar to the way internal static variables are used in a C function, except that their contents is unique for every instance of the function block.

Operators

Some of the operators that can be used in the C programming language are:

- Binary operators: =, -, *, / and the modulus operator %. The % operator can be used for integers only: x % y produces the remainder when x is divided by y. Integer division (operator /) truncates any fractional part.
- Unary operators: -, ! . The result of the unary - operator is the negative of its operand. The result of the logical negotiation operator ! is 1 if the value of its operand is 0. 0 if the value of its operand is non-zero.
- Relational operators: >, >=, <, <=, ==, !=. Notice that the following is **not correct**:

  ```
  if (phi1 = BTTRUE)
  ```

  while

  ```
  if (phi1 == BTTRUE)
  ```

  is correct.

  Much used are the logical connectives && (which means AND) and || (which means OR). For example:

  ```
  if (c[0] == BTTRUE && c[1] == BTFALSE)
       reg = in;
  ```

  and

```
if ( in[0][0] == BTFALSE || in[0][1] == BTFALSE &&
    ( in[1][0] != BTTRUE || in[1][1] != BTTRUE ) )
    out = BTFALSE;
else
    out = BTTRUE;
```

- C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, the decrement operator −− subtracts 1. The unusual aspect is that ++ and −− may be used either as prefix operators (before the variable, as in ++n), or postfix (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n *before* using its value, while n++ increments n *after* its value has been used.

Control flow

The control flow statements of a language specify the order in which computations are done.

- Statements and Blocks:
  An *expression* such as x=0 becomes a *statement* when it is followed by a semicolon, as in: x=0;. In C, the semicolon is a statement terminator, rather than a separator as it is in Algol-like languages.
  The braces { and } are used to group statements together into a *compound statement* or *block* so that they are syntactically equivalent to a single statement.

- If-else:
  The if-else statement is used to make decisions. Formally the syntax is:

```
if ( expression )
    statement-1
else
    statement-2
```

  where the else part is optional. As examples:

```
if (load == BTTRUE)     and         if (phi1 == BTTRUE){
    count = in_val;                     if (reset == BTFALSE)
else                                         count++;
    count++;                            else {
                                            count = 0;
                                            tc = 0;
                                        }
                                        done = 1;
                                    }
```

- Else-If:
  The following sequence of if’s is the most general way of writing a multi-way decision:

```
if ( expression_1 )
    statement-1
else if ( expression_2 )
    statement-2
else if ( expression_3 )
    statement-3
else
    statement-4
```

- Loops - While and For:
  In:

  ```
  while ( expression )
      statement
  ```

  the *expression* is evaluated. If it is true (which is non-zero) *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes false (which is zero), at which point execution resumes after *statement*.
  The for statement:

  ```
  for ( expression_1; expression_2; expression_3 )
      statement
  ```

  is equivalent to

  ```
  expression_1;
  while ( expression_2 ){
      statement
    expression_3;
  }
  ```

  Grammatically, the three components of a for-loop are expressions. Most commonly, *expression_1* and *expression_3* are assignments or similar and *expression_2* is a relational expression. Any of the three parts can be omitted, although the semicolon must retain. Examples:

  ```
  i=0;                              and      for (i=0; i<7; i++)
  c=in[i];                                       reg[i]=reg[i+1];
  while (c == BTTRUE && i<8) {               reg[7]=in;
      i++;
      c=in[i];
  }
  ```

Input and Output

In SLS the input of data into a network is normally done with the set command in the command file. The output in a network consists of a sequence of the logic states of those terminals that have been placed in a print or a plot command in the command file.

The function blocks written in the C language provide more possibilities. Input and output routines as exist in C can be used in the function blocks to define a specific input and output behavior. The two most common ways to use C input routines seem to be: get data from the user’s keyboard or from a certain file into a function block; for output the reverse: place data on the user’s terminal screen or into a certain file. Several C routines are available for these I/O matters.

Formatted output - printf; formatted input - scanf

The two routines printf for output and scanf for input permit translation to and from character representation of numerical quantities. The routine printf is used:

```
printf (control, arg1, arg2, ...)
```

The first part, control, is a string that may contain ordinary characters and conversion specifications, each of which causes conversion and printing of the next successive argument to printf. Each conversion specification is introduced by the character % and ended by a conversion character. For example:

```
printf ("phi1=%c; \tin=%8s\ncount=%d\n", phi1, in, count);
```

might print something like

```
phi1=O; in=OIIOIO
count=26
```

In the control string (between double quotes) %c converts the character value of phi1 to O or I and %s converts the string value of in to a sequence of O’s and I’s; %d converts the integer of count to decimal notation. The character \t means a tab, \n means a new line. The digit 8 between % and conversion character s specifies minimum field width. Some conversion character are:

c	The argument is taken to be a single character
s	The argument is a string
d	The argument is converted to decimal notation
f or e or g	the argument is taken to be a float or a double

The function scanf is the input analog of printf, providing many of the same conversion facilities in the opposite direction.

```
scanf (control, arg1, arg2, ...)
```

Scanf reads characters from the standard input, interprets them according to the format specifies in control and stores the results in the remaining arguments. For example:

```
scanf ("%d %f %s %c", &intgr, &rl, in, &phi);
```

with the input line

```
3    3.2e-9    OIOI    I
```

will assign the value 3 to *intgr*, the value 3.2e-9 to *rl*, the string "OIOI" to *in* and the character ’I’ to *phi*. The arguments of scanf must be pointers; this has a result that the sign & must be placed just before the arguments unless the variable is a pointer. A string variable is a pointer, like any array, therefore in the example the sign & is not placed before in. Some conversion character are:

d	a decimal integer is expected in the input; the corresponding argument must be an integer pointer (e.g. integer array or &*variable*).
c	a single character is expected; the corresponding argument should be a character pointer. The normal skip over white space characters is suppressed; to read the next non-while space character, use %1s
s	a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string.
f	a floating number is expected; the corresponding argument should be a pointer to a float.
lf	a floating number is expected and the corresponding argument should be a pointer to a double.

More detailed information about printf and scanf can be found in the book of Kernighan and Ritchie.

File access

To read from or write into a file, several rules must be followed. First the declaration:

```
#include <stdio.h>
```

must be placed at the top of the file. Second the declaration:

```
FILE *fp;
```

must be placed in the load, initial or behavior part, depending on in which part a file is accessed. Then a file has to be opened with:

```
fopen (name, mode)
```

with name being the string indicating the name of the file and mode being the string indicating the mode for which the file is opened. For example:

```
fp = fopen ("ram.cont", "r");
```

opens the file ram.cont for reading and

```
fp2 = fopen ("netw.rslt", "w");
```

opens the file netw.rslt for writing. The two variables fp and fp2 are file pointers; notice that the declaration: FILE *fp2; was needed!
The next thing needed is a way to read or write the file once it is open. There are several

possibilities of which getc and putc are the simplest. Getc returns the next character from a file: it needs the file pointer to tell it what file. Thus

```
    c = getc (fp);
```

places in c the next character from the file referred to by fp. Putc is the reverse of getc:

```
    putc (c, fp);
```

It is also possible to use the functions fscanf and fprintf for formatted input and output:

```
    fscanf (fp, control, arg1, arg2, ...)
    fprintf (fp, control, arg1, arg2, ...)
```

The language C offers far more possibilities for I/O handling but the discussion of those do not fall within the scope of this manual.

Curses

A short introduction of the curses package will be given here. Using a small subset of the curses package, it is possible to move the cursor to any point in the screen and to place data anywhere on the screen. When curses is used, in the initial part of the function block the initialization of curses must be done. When this is done, in the behavior part of the function block all the curses routines can be used. As an example a function block description using curses:

```
#include <curses.h>

function regc (inread in[4], load;
               input phi, phi2;
               output out[4])

state {
    char mem[4];
}

initial {
    initscr ();
    clear ();
    refresh ();
}

behavior {
    int i;

    if (phi == BTTRUE && load == BTTRUE)
        BSCOPY (mem, in);
    if (phi2 == BTTRUE)
        BSCOPY (out, mem);
    move (3,3);
    printw ("phi=%c phi2=%c", phi, phi2);
    move (4,3);
    for (i=0; i<4; i++)
        addch (mem[i]);
    move (5,3);
    printw ("out=%s", out);
    move (23,0);
    refresh ();
}
```

The curses routines used in this example are all that are needed for many possibilities. The routines are:

initscr()	initialization: initscr() must **always** be called before any curses routines are used.
clear()	clears the screen.
refresh()	the effect of used curses routines will be placed on the user’s screen at the moment refresh() is called; before that, nothing will be seen on the screen.
move(x,y)	move cursor to line x and column y.
printw()	a formatted output routine similar to printf but the output is placed on the point where the cursor is at the moment.
addch	outputs character ch on the place where the cursor is.

Since the use of curses requires the linking command to have the options:

```
        -lcurses -ltermlib
```

included, the file sls.ld_arg (see section 2) must contain that line.

Furthermore, notice that the statement #include <curses.h> has to be included in the function block description file in order to use the curses routines.

The C preprocessor

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. A compiler-control line of the form

```
     #define   identifier  token-string
```

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. For example

```
     #define    NAME    counter
     #define    nmbel   8
```

will replace NAME with counter and nmbel with 8, everywhere one of these two identifiers appears in the source text.
The control-line

```
     #include "filename"
```

causes that line to be replaced by the entire contents of the file *filename*.
A compiler control-line of the form

```
     #if  constant-expression
```

checks whether the constant expression evaluates to non-zero. For example:

```
     #if (CURS==1)
        move (3,0);
        printw("%s", in);
        move (23,0);
        refresh ();
     #endif
```

Only when CURS==1, the part between #if and #endif will be included in the compiled object code. The control lines

```
     #ifdef identifier
     #ifndef identifier
     #else
```

can be used as well.

The C-code in the function block description

The C-code in the load, initial and behavior part is completely similar to the C-code of function body of any C function without declaration list. However some remarks have to be made. First of all it is not allowed to use the

```
return();
```

statement in the C-code of the load, initial or behavior part. This is because the C-code of the those parts is supplemented with C-code when it is converted by the parser into genuine C functions. When the return() statement is used, the necessary supplementary C-code at the end of the function can not be executed.

Second it must be known that a file may contain only one function block description but, that any other correct C-code may be added to that description in the file. This means that a file can be any file with C-code as long as only one function block description exists within the file. The program *cfun* will convert this file into a file with normal C-code, which is compiled and placed into the database. (See section 2).

The auxiliary routines and defines

In the C-code of the load, initial or behavior part several auxiliary routines and defines can be used. The defines that can be used are:

```
#define BTTRUE  'I'
#define BTFALSE 'O'
#define BTUNDEF 'X'
#define BTFREE  'F'
#define BSCOPY strcpy
#define BSCMP  strcmp
#define BSNCMP strncmp
```

The first four defines have been discussed previously, of the others, examples are shown here:

BSCOPY(mem,in)	copies the contents of (bit-)string in into (bit-)string mem.
int BSCMP(mem1,mem2)	compares the (bit-) strings mem1 and mem2; 0 is returned when the strings are identical, non-zero else.
int BSNCMP(mem1,mem2,n)	compares the first n characters of the (bit-)strings mem1 and mem2.

It should be understood by now that a bit-string is in fact an ordinary string of characters in C, so for example strcpy and BSCOPY can be (and are) exactly the same routine.

It is very important to notice that bit-strings must **never** be used in assignment statements:

When *in* and *mem* both are bit-strings, **not correct** is

```
    in = mem;
```

while

```
    BSCOPY (in, mem);
```

can very well be used.

The auxiliary routines that can be used in the C-code are:

- routines for operations on bit variables (e.g. zero dimensional terminals):

char BTINVERT (bt)	returns inverted value of bt
char BTAND (bt1, bt2,...)	returns logical-AND value of bt1, bt2,...
char BTNAND (bt1, bt2,...)	returns logical-NAND value of bt1, bt2,...
char BTOR (bt1, bt2,...)	returns logical-OR value of bt1, bt2,...
char BTNOR (bt1, bt2,...)	returns logical-NOR value of bt1, bt2,...
char BTEXOR (bt1, bt2,...)	returns logical-EXOR value of bt1, bt2,...
char BTEXNOR (bt1, bt2,...)	returns logical-EXNOR value of bt1, bt2,...

  Warning: when a variable argument list is used and the routine call is not in a file that is parsed by *cfun*, the argument list should be terminated by a '@' character.

- routines for operations on bit string variables (e.g. one dimensional terminals):

char * BWINVERT (bs)	returns bit-string which is the bitwise complement of bs
char * BWAND (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-AND of bs1, bs2,...
char * BWNAND (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-NAND of bs1, bs2,...
char * BWOR (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-OR of bs1, bs2,...
char * BWNOR (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-NOR of bs1, bs2,...
char * BWEXOR (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-EXOR of bs1, bs2,...
char * BWEXNOR (bs1, bs2,...)	returns bit-string which is the result of a bitwise logical-EXNOR of bs1, bs2,...
BSSET(bs)	sets all bits (characters) of bit-string bs to value BTTRUE (which means character: 'I')
BSRESET(bs)	sets all bits (characters) of bit-string bs to value BTFALSE (which means character: 'O')

BSUNDEF(bs)	sets all bits (characters) of bit-string bs to value BTUNDEF (which means character: ’X’)
BSFREE(bs)	sets all bits (characters) of bit-string bs to value BTFREE (which means character: ’F’)
char * BSROTATE (bs, direction)	returns bit-string that is the round rotated bit-string bs; direction can be ’l’ (left) or ’r’ (right)
int BSTOI(bs)	returns integer with converted value of bit-string bs, using the radix-2 representation; MAXINT is returned when the bit string contains a bit with value BTUNDEF (’X’)
int TCTOI(bs)	returns integer with converted value of bit-string bs, using the two’s complement representation; MAXINT is returned when the bit xstring contains a bit with value BTUNDEF (’X’)

Warning: when a variable argument list is used and the routine call is not in a file that is parsed by *cfun*, the argument list should be terminated by the string "ENDVAR".

- other routines

char * ITOBS(intgr, nmbbt)	returns a bit-string that is the converted value of the integer intgr, using the radix-2 representation; nmbbt is the number of bits of the returned bit-string.
char * ITOTC(intgr, nmbbt)	returns a bit-string that is the converted value of the integer intgr, using the two’s complement representation; nmbbt is the number of bits of the returned bit-string.
single_step()	simulation is stopped until a arbitrary character is typed: ’q’ stops the simulation immediately; ’x’ stops the simulation after the current simulation step; ’c’ continues simulation without stopping; any other character continues simulation until a next call to single_step.(do not use this routine when using curses routines).
func_error()	error message is given, question is asked whether the simulation must continue; simulation continues if ’y’ is typed. (do not use this routine when using curses routines).
single_curs_step()	same as routine single_step(); should be used

	when curses routines are used.
curs_error()	same as routine func_error(); should be used when curses routines are used.

- In section 3.1. it was stated that in a function block the load of input, inread and inout terminals can be defined, and that the delay of output and inout terminals can be specified. The routines needed for that purpose are discussed here. First of all a routine exists for obtaining the value of the capacitance of the node a function block terminal is connected with:

  ```
  cap_val ( environment, mode, terminal )
  ```

  The arguments of the routine are:

environment:	NODE	the capacitance of the node is given
	VICIN	the capacitance of the node plus the capacitances of the nodes that are connected with the investigated node through undefined or closed transistors is given
mode:	MIN	connected through **only** closed transistors
	MAX	connected through closed or undefined transistors
terminal		any terminal of the function block.

  Notice that with ’node’ is meant: the node that is connected with the terminal that is used in the argument-list.

  The routine for defining the load of input, inread or inout terminals of a function block is:

  ```
  cap_add ( expression, in_terminal )
  ```

  In this routine *expression* can be any expression that results in a real value (float), *in-terminal* is an input, inread or inout terminal of the function block. The effect of this routine is that the value of *expression* is added to the value of the capacitances of the node the relevant terminal is connected with (both static and dynamic capacitance). In that way the load of this terminal is accounted for by the capacitance of the node it is connected with.

  When the load capacitance of a terminal is constant for the whole simulation interval, the capacitance for that terminal should preferably be specified in the load part of the function description. Since the load part of a function is executed before the initial part, all total node capacitances are known then when calls to cap_val() occur in the initial parts of the function blocks.

Finally the routine that is used for defining the delay times of an output or inout terminal of a function block is:

`delay (` *mode*`,` *expression*`,` *out-terminal*`)`

The arguments of the routine can be:

mode	’r’	rise delay time
	’f’	fall delay time
	’b’	both rise- and fall delay time
expression		any expression resulting in a real value (float)
out-terminal		any output or inout terminal of the function block

In general, string that are returned by auxiliary routines remain until the end of the evaluation of the function block. After that, their memory space will be disposed.

Furthermore, be sure to use a value like 2.0 instead of 2 when a float value is required.

3.4 Examples of function block descriptions

In this section a few examples will be shown of descriptions of function blocks. These function blocks will be used in examples of networks as will be shown and explained in section 4.

The first example is a functional description of a multiplexer. The block diagram of the multiplexer is shown in figure 4.

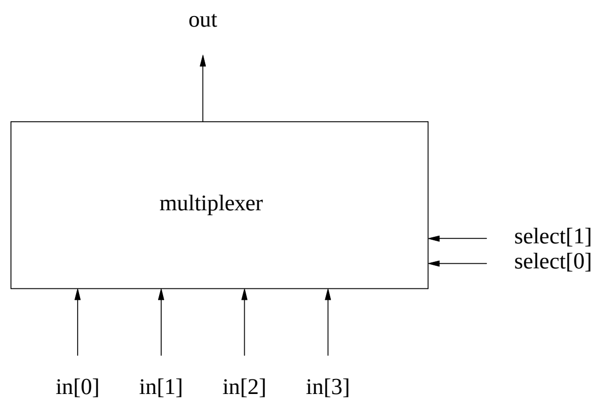


Figure 4. Block diagram of multiplexer

```
function multiplexer ( input   in[4];
                       output  out;
                       input   select[2] )

behavior {
    if ( BSTOI(select) < 4 )
        out = in[ BSTOI(select) ];
}
```

Figure 5. Functional description of multiplexer

The functional description of this block is very simple, as can be seen in figure 5. The multiplexer has no state variables and no load or initial body. Both *in* and *select* terminals are input terminals since a change of value of one of those terminals could cause a change in the *out* terminal. The one line in the behavior body defines the logic behavior of the multiplexer: the bit string *select* (2 bits) is converted to an integer and this integer determines which of the *in* terminals will give its data to the *out* terminal. This is only executed if the integer is smaller then 4. This test is necessary because when one of the *select* bits will have the value BTUNDEF (which is ’X’), the integer that will be returned by the routine BSTOI will be MAXINT (which is the maximum integer on the computer one is working on). When the MAXINT’s bit in the string *in* is asked, this will probably cause a ’segmentation violation’.

In the second example, a description of a shiftregister will be shown. The design of the shiftregister function block is shown in figure 6.

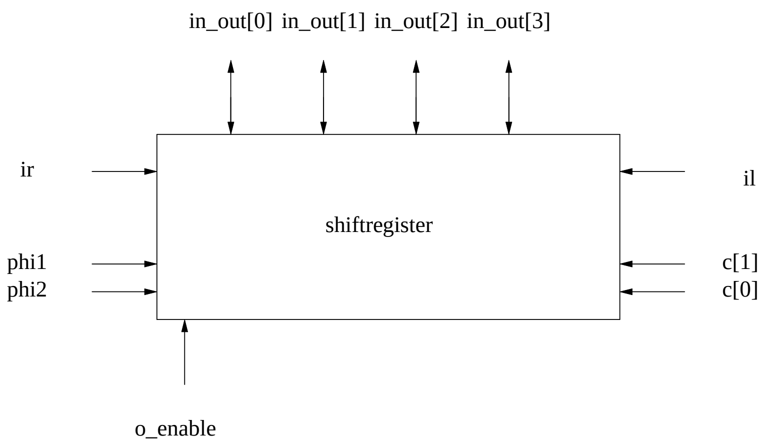


Figure 6. Block diagram of shiftregister

The block has two control variables of which the coding is:

Control	c0	c1
hold	O	O
left_shift	O	I
right_shift	I	O
load	I	I

The functional description of the shiftregister can be found in figure 7.

```
#define  NAME   shiftregister
#define  NMBEL  4

function NAME ( inread  ir, il, c0, c1, o_enable;
                inout   in_out[NMBEL];
                input   phi1, phi2 )

state {
    char  mem[NMBEL];
}

behavior {
    int i;
    if (phi1 == BTTRUE) {
        if (c0 == BTTRUE && c1 == BTTRUE)         /* load */
            BSCOPY (mem, in_out);
        else if (c0 == BTTRUE && c1 == BTFALSE) { /* shift left */
            for (i = NMBEL-1; i > 0; i--)
                mem[i] = mem[i-1];
            mem[0] = ir;
        }
        else if (c0 == BTFALSE && c1 == BTTRUE) { /* shift right */
            for (i = 0; i < NMBEL-1; i++)
                mem[i] = mem[i+1];
            mem[NMBEL-1] = il;
        }
    }
    BSFREE (in_out);
    if (phi2 == BTTRUE && o_enable == BTTRUE) /* write */
        BSCOPY (in_out, mem);
}
```

Figure 7. Functional description of shiftregister

Several remarks can be made about this description:

— Because of the use of macro definitions the number of shift-elements in the shiftregister can be changed easily: only the integer 4 in the definition needs to be changed. The same can be said about the name of the function block.

— The routine BSCOPY() is used for assigning values to the bit strings mem and out.

— It is easy to understand what the function of the block is; just by reading the simple code the actions are clear.

— The terminal bit-string in_out[4] is bidirectional. The value of the bit-string can be used as input data but values can be assigned to the bit-string as well. Notice that the bits are set to free (character 'F') when no data is to be outputted.

— No timing properties are given to the function block: the input, inread and inout terminals cause no load, and the delay of the inout terminals is zero.

In the third example, a description of a counter will be shown. In this description much attention is given to the timing properties of the function block. The designed counter is a normal counter which can load new data and count up or down. The two-phase non-overlapping clocks phi1 and phi2 are used: during phi1 new data is loaded or the counter counts, during phi2 the data is sent to the output terminals. The termcount terminal gets high during phi2 when all the bits in the counter are high. The block diagram of the counter is shown in figure 8.

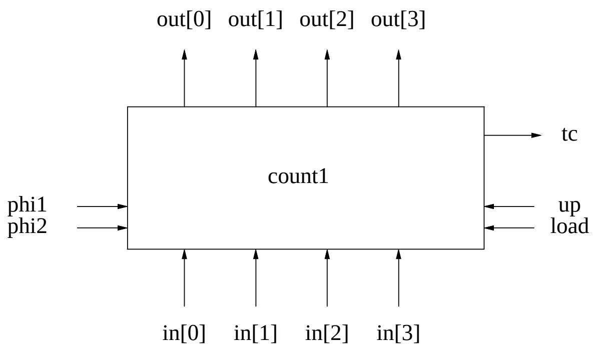


Figure 8. Block diagram of counter

The functional description of the counter can be found in figure 9.

```
#if (CURS == 1)
#include <curses.h>
#endif

#define    CNTNAME    count1
#define    NMBEL    4
#define    CURS     1
#define    CRSLI    4
#define    CRSCO    8
function CNTNAME ( input   phi1, phi2;
                   inread  in[NMBEL], up, load;
                   output  out[NMBEL], tc )
state {
        int      termcount;
        int      count;
}
load {
    cap_add (100e-15, in[0]);
    cap_add (150e-15, in[1]);
    cap_add (200e-15, in[2]);
    cap_add (250e-15, in[3]);
}
initial {
    int i;
    termcount=0;
    count=0;
#if (CURS == 1)
    initscr();
    clear();
    refresh();
#endif
}
behavior {
    delay (’b’, 1000 * cap_val(VICIN,MAX,out[0]), out[0]);
    delay (’b’, 1000 * cap_val(VICIN,MIN,out[1]), out[1]);
    delay (’b’, 1000 * cap_val(VICIN,MAX,out[2]), out[2]);
    delay (’b’, 1000 * cap_val(NODE, MAX,out[3]), out[3]);

    single_curs_step();

    if (phi1 == BTTRUE) {
        if (load == BTTRUE) {
            count = BSTOI (in);
        }
        else if (up == BTTRUE) {
            count++;
            if (count == (1<<NMBEL)-1) {
                termcount = 1;
            }
            else
                termcount = 0;
            if (count == (1<<NMBEL)) {
                count = 0;
            }
        }
        else if (up == BTFALSE) {
            count--;
            if (count == -1) {
                count = (1<<NMBEL)-1;
                termcount = 1;
            }
            else
                termcount = 0;
        }
    }
    if (phi2 == BTTRUE) {
        strcpy (out, ITOBS (count, NMBEL));
        if (termcount == 1)
            tc = BTTRUE;
        else
            tc = BTFALSE;
    }
#if (CURS == 1)
    move(CRSLI, CRSCO);
    printw("%s", out);
    printw("%6d", count);
    move(23,0);
    refresh();
#endif
}
```

Figure 9. Functional description of a counter

Several remarks can be made about this description as well:

— In the initial part the counter is set to zero.

— The behavior of the counter is easy to understand when reading the code in the behavior part. The only new aspect is

```
1<<NMBEL
```

which means 2^{NMBEL}.

— For output, curses routines have been used. Because of the auxiliary routine single_curs_step() the simulation of the function block can be done step-wise. As can be concluded from the used curses routines, on line number 4, on the 8th column, the bits of the counter followed by the decimal representation of the bits will be shown on the terminal screen during simulation.

— Notice that the state variables are integers, the counting is done in integers. The decimal value of the bit-string that is loaded into the counter is found using the conversion routine BSTOI(); the bit-values of the counter are found using the conversion routine ITOBS().

— The timing properties of the counter function block are defined with the routines:

```
delay (’b’, 1000 * cap_val(VICIN,MAX,out[0]), out[0]);
delay (’b’, 1000 * cap_val(VICIN,MIN,out[1]), out[1]);
delay (’b’, 1000 * cap_val(VICIN,MAX,out[2]), out[2]);
delay (’b’, 1000 * cap_val(NODE, MAX,out[3]), out[3]);
```

With delay() the delay times of the output terminals out[0] to out[3] are defined. For

the several rise and fall delay times, real values are calculated, using the dynamic capacitance of the nodes in the network that are connected with the output terminals of the function block. The capacitances of the nodes themselves as well as the capacitance of the node together with its vicinity are used for this purpose. The value 1000 could be seen as output resistance; in that way RC-times are found for delay times. Notice that the delay times are defined in the behavior block. This is due to the fact that the delay of the output terminals depend on the structure of the network the function block is part of. At every new evaluation of the function block the vicinities of the nodes may differ. The delay times of the three output terminals depend of the capacitance of the nodes with their vicinities and therefore these times have to be calculated again at every function block evaluation.
The input load of the function block is defined with the routine:

```
cap_add (100e-15, in[0]);
cap_add (150e-15, in[1]);
cap_add (200e-15, in[2]);
cap_add (250e-15, in[3]);
```

With cap_add() the load of the inread terminals in[0] to in[3] is defined: the capacitance of the node in the network that is connected with the inread terminal in[0] is increased with 100 femto-farad, the capacitance of the node in the network that is connected with the inread terminal in[1] is increased with 150 femto-farad, et cetera.

In the last example, a description of a function block with combinational logic will be shown. In the file that contains this functional description, other C-code has been placed as well. The contents of the file containing the functional description of this function block can be found in figure 10.

```
letsee (s1, s2, s3, s4, fun)
char s1, s2, s3, s4;
char fun[];
{
    char dum;
    dum = BTEXOR (s1, s2, s3, BTTRUE);
    fprintf (stderr,
            "e_i[0]=%c, e_i[1]=%c, e_i[2]=%c, e_i[3]=I, exorresult=%c\n",
            s1, s2, s3, dum);
    fprintf (stderr, "  i[0]=%c,   i[1]=%c,   i[2]=%c, %sresult=%c\n",
            s1, s2, s3, fun, s4);
}

char four_of_four (s1, s2, s3, s4)
char s1, s2, s3, s4;
{
    if (s1 == BTUNDEF || s2 == BTUNDEF || s3 == BTUNDEF || s4 == BTUNDEF)
        return (BTUNDEF);
    else if (s1 == BTFALSE && s2 == BTTRUE && s3 == BTFALSE && s4 == BTFALSE)
        return (BTTRUE);
    else
        return (BTFALSE);
}


function combinator ( input  phi;
                      output oinv, oand, onand, oor,
                             onor, oexor, oexnor, osp[2], us_in[4] )

behavior {
    char in[5];

    if (phi == BTTRUE) {
        printf ("typ input (4 bits): ");
        scanf ("%4s", in);
        oinv   = BTINVERT (in[0]);
        oand   = BTAND (in[0], in[1], in[2]);
        letsee (in[0], in[1], in[2], oand, "AND");
        onand  = BTNAND (in[0], in[1], in[2], in[3]);
        oor    = BTOR (in[0], in[1], in[2]);
        letsee (in[0], in[1], in[2], oor, "OR");
        onor   = BTNOR (in[0], in[1], in[2], in[3]);
        oexor  = BTEXOR (in[0], in[1], in[2]);
        oexnor = BTEXNOR (in[0], in[1], in[2], in[3]);
        osp[0] = BTEXOR (BTNOR (BTAND (BTEXNOR (in[2], in[3]), in[1]),
                                 in[0]), in[1], in[2]);
        osp[1] = BTEXNOR (four_of_four (in[0], BTINVERT (in[1]), in[2], in[3]),
                          BTAND (BTEXNOR (in[2], in[3]), in[1]), in[0],
                                  in[1], in[2]);
        BSCOPY (us_in, in);
    }
}
```

Figure 10. Contents of file containing a functional description of block: combinator

About the data in this file several remarks can be made:

— In the file two normal C-routines can be found and the functional description of the function block. The two routines are called in the behavior block of the functional description. The routine letsee() outputs the values of three of the four characters in its argument list (bits in this case!) together with the result of an EXOR operation to stderr. Stderr normally is the terminal screen. After that, again the values of all the four characters are printed together with a string. The routine four_of_four() returns the character 'I' (which is BTTRUE) when the four characters in its argument list do have the values 'O', 'I', 'O' and 'O' and the character 'O' (which is BTFALSE) else.

— In the behavior block, a question is send to stderr and four characters must be typed on the keyboard. These four characters are placed in the local variable (character string) named *in*. Notice that this array of characters must have 5 elements: four elements for characters and one element for the terminating '\0'. The typed characters must have values 'O', 'X' or 'I' or else an error will occur.

4. THE USAGE OF THE MIXED-LEVEL SIMULATOR

4.1 Examples of simulations of networks containing function blocks

In this section several examples will be shown of the usage of the Mixed Level Simulator. In the networks that are described the function blocks that have been shown in section 3.4. will be used. In the examples simulations will be done on functional level and mixed-level.

In the first example a network is simulated with the function block *multiplexer* in it. The description of this function block can be found in figure 5 in section 3.4. The SLS network description of the network is shown in figure 11. In this description can be seen that the network consists of only one subblock: the function block.

```
network mux_netw (terminal mux_in[1..4], mux_out, mux_select[1..2])
{
    {inst_mux} @ multiplexer (mux_in[1..4], mux_out,
                      mux_select[1..2]);
}
```

Figure 11. SLS description of network *mux_netw*

In the SLS description the way of calling a function block in a network is shown: the character @ is used just as is done when a built-in function is called in a network. The simulation of this network can be considered as a purely functional simulation, since the only element in the network is a function block. In the simulation the logic behavior of the function block *multiplexer* can be verified.
The function block has six **input** terminals (see figure 5): *in[0]* to *in[3]* and *select[0]* and *select[1]*. In the network *mux_netw* these terminals are connected with the nodes *mux_in[1]* to *mux_in[4]* and *mux_select[1]* and *mux_select[2]*. Whenever at least one of those nodes changes logic state, the function block will be evaluated. In the simulation these nodes will change logic state as a result of a ’set’ command in the command file. The function block has one output terminal: *out*; This terminal is connected with the node *mux_out*. This node can be used in a ’print’ command in the command file. The results of a simulation of the network *mux_netw* is shown in figure 12.

```
================================================================================
                                     S L S
                                  version: 3.0
                      S I M U L A T I O N   R E S U L T S
================================================================================
 time         | m m m m   m m     m
 in 1e-09 sec | u u u u   u u     u
              | x x x x   x x     x
              | _ _ _ _   _ _     _
              | i i i i   s s     o
              | n n n n   e e     u
              | * * * *   l l     t
              | 1 2 3 4   e e
              | * * * *   c c
              |           t t
              |           * *
              |           1 2
              |           * *
================================================================================
         0.00 | 1 1 1 1   1 1     1
         1.00 | 1 0 1 0   1 1     0
         2.00 | 0 0 0 1   1 1     1
         3.00 | 0 1 0 0   1 1     0
         4.00 | 0 0 1 1   1 1     1
         5.00 | 1 0 1 0   1 1     0
         6.00 | 1 1 0 1   1 1     1
         7.00 | 0 0 0 0   1 1     0
         8.00 | 0 0 1 1   1 0     1
         9.00 | 0 1 1 0   1 0     1
        10.00 | 1 0 0 1   1 0     0
        11.00 | 1 0 0 0   1 0     0
        12.00 | 0 1 1 1   1 0     1
        13.00 | 0 0 1 0   1 0     1
        14.00 | 0 0 0 1   1 0     0
        15.00 | 1 1 0 0   1 0     0
        16.00 | 1 0 1 1   0 1     0
        17.00 | 0 0 1 0   0 1     0
        18.00 | 0 1 0 1   0 1     1
        19.00 | 0 0 0 0   0 1     0
        20.00 | 1 0 1 1   0 1     0
        21.00 | 1 1 1 0   0 1     1
        22.00 | 0 0 0 1   0 1     0
        23.00 | 0 0 0 0   0 1     0
        24.00 | 0 1 1 1   0 0     0
        25.00 | 1 0 1 0   0 0     1
        26.00 | 1 0 0 1   0 0     1
        27.00 | 0 1 0 0   0 0     0
        28.00 | 0 0 1 1   0 0     0
        29.00 | 0 0 1 0   0 0     0
        30.00 | 1 1 0 1   0 0     1
        31.00 | 1 0 0 0   0 0     1
================================================================================
  network : mux_netw                                                  nodes : 7
================================================================================
```

Figure 12. Results of a simulation of network *mux_netw*

In the second example the function block *shiftregister* is used. The description of this block can be found in figure 7. Now the network *doub_shft* does not consist of one instance of the function block but of two instances of the same function block plus some inverters. The network is shown in figure 13.

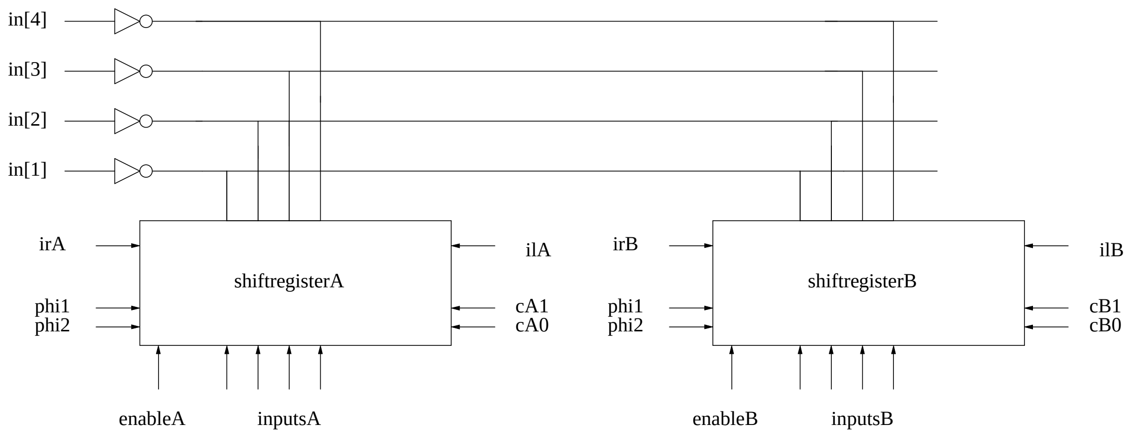


Figure 13. Network diagram of network *doub_shft*

The SLS description of the subnetwork *inverter* and the SLS description of the network *doub_shft* are shown in figure 14.

```
network inverter (terminal vdd, vss, i, o)
{
    nenh w=8u l=4u (i, o, vss);
    ndep w=4u l=8u (o, vdd, o);
}

network doub_shft (terminal vdd, vss, in[1..4], irA, ilA, cA0, cA1, irB, ilB,
                            cB0, cB1, enableA, enableB, bus[1..4], phi1, phi2)
{
    {instA} @ shiftregister (irA, ilA, cA0, cA1, enableA,
                             bus[1..4], phi1, phi2);
    {instB} @ shiftregister (irB, ilB, cB0, cB1, enableB,
                             bus[1..4], phi1, phi2);

    {inst_inv[1..4]} inverter (vdd, vss, in[1], bus[1],
                               vdd, vss, in[2], bus[2],
                               vdd, vss, in[3], bus[3],
                               vdd, vss, in[4], bus[4]);
}
```

Figure 14. SLS description of networks *inverter* and *doub_shft*

In the network *doub_shft* the function block *shiftregister* is called twice: two different instances exist of the function block. For each instance of the function block, the state variables are different. Only then it is possible for the two registers to contain different data.
The subblock *shiftregister* is described on functional level and the subblock *inverter* is described on transistor level. Therefore the simulations of this network are real mixed-level simulations.
In the network a bus exists: both registers and the inverters are connected with the bus. It

is not possible to set data on the bus with the 'set' command in the command file, because **a 'set' input node may never be connected with a function block inout or output terminal**. During the simulation a register puts data on the bus only if its *o_enable* signal is high (see functional description in figure 7). The inverters however, put data on the bus continuously. Function block output terminals force their logic value on the nodes they are connected with, unless their value is BTFREE. Therefore, the bus will only get its data from the inverters when the value of both the function block output terminals is BTFREE. The results of a simulation of the network *doub_shft* are shown in figure 15.

```
================================================================================
                                     S L S
                                  version: 3.0
                      S I M U L A T I O N   R E S U L T S
================================================================================
 time         | p p   i i i i   c c i i e   c c i i e   b b b b
 in 1e-09 sec | h h   n n n n   A A r l n   B B r l n   u u u u
              | i i   * * * *   0 1 A A a   0 1 B B a   s s s s
              | 1 2   1 2 3 4           b           b   * * * *
              |       * * * *           l           l   1 2 3 4
              |                         e           e   * * * *
              |                         A           B
================================================================================
         0.00 | 0 0   0 0 0 0   0 1 0 0 0   1 1 0 0 1   1 1 1 1
         1.00 | 1 0   0 0 0 0   0 1 0 0 0   1 1 0 0 1   1 1 1 1
         2.00 | 0 0   0 0 0 0   0 1 0 0 0   1 1 0 0 1   1 1 1 1
         3.00 | 0 1   0 0 0 0   0 1 0 0 0   1 1 0 0 1   1 1 1 1
         4.00 | 0 0   0 0 0 1   0 1 0 0 0   1 0 0 0 1   1 1 1 0
         5.00 | 1 0   0 0 0 1   0 1 0 0 0   1 0 0 0 1   1 1 1 0
         6.00 | 0 0   0 0 0 1   0 1 0 0 0   1 0 0 0 1   1 1 1 0
         7.00 | 0 1   0 0 0 1   0 1 0 0 0   1 0 0 0 1   0 1 1 1
         8.00 | 0 0   0 0 1 0   1 1 0 0 0   1 0 1 0 1   1 1 0 1
         9.00 | 1 0   0 0 1 0   1 1 0 0 0   1 0 1 0 1   1 1 0 1
        10.00 | 0 0   0 0 1 0   1 1 0 0 0   1 0 1 0 1   1 1 0 1
        11.00 | 0 1   0 0 1 0   1 1 0 0 0   1 0 1 0 1   1 0 1 1
        12.00 | 0 0   0 0 1 1   0 1 0 0 1   1 0 0 0 0   1 1 0 0
        13.00 | 1 0   0 0 1 1   0 1 0 0 1   1 0 0 0 0   1 1 0 0
        14.00 | 0 0   0 0 1 1   0 1 0 0 1   1 0 0 0 0   1 1 0 0
        15.00 | 0 1   0 0 1 1   0 1 0 0 1   1 0 0 0 0   1 0 1 0
        16.00 | 0 0   0 1 0 0   0 1 0 0 1   1 0 1 0 0   1 0 1 1
        17.00 | 1 0   0 1 0 0   0 1 0 0 1   1 0 1 0 0   1 0 1 1
        18.00 | 0 0   0 1 0 0   0 1 0 0 1   1 0 1 0 0   1 0 1 1
        19.00 | 0 1   0 1 0 0   0 1 0 0 1   1 0 1 0 0   0 1 0 0
        20.00 | 0 0   0 1 0 1   0 1 0 0 0   1 0 0 0 1   1 0 1 0
        21.00 | 1 0   0 1 0 1   0 1 0 0 0   1 0 0 0 1   1 0 1 0
        22.00 | 0 0   0 1 0 1   0 1 0 0 0   1 0 0 0 1   1 0 1 0
        23.00 | 0 1   0 1 0 1   0 1 0 0 0   1 0 0 0 1   0 1 0 1
        24.00 | 0 0   0 1 1 0   0 1 0 0 0   1 0 1 0 1   1 0 0 1
        25.00 | 1 0   0 1 1 0   0 1 0 0 0   1 0 1 0 1   1 0 0 1
        26.00 | 0 0   0 1 1 0   0 1 0 0 0   1 0 1 0 1   1 0 0 1
        27.00 | 0 1   0 1 1 0   0 1 0 0 0   1 0 1 0 1   1 0 1 0
        28.00 | 0 0   0 1 1 1   0 1 0 0 0   1 0 0 0 1   1 0 0 0
        29.00 | 1 0   0 1 1 1   0 1 0 0 0   1 0 0 0 1   1 0 0 0
        30.00 | 0 0   0 1 1 1   0 1 0 0 0   1 0 0 0 1   1 0 0 0
        31.00 | 0 1   0 1 1 1   0 1 0 0 0   1 0 0 0 1   0 1 0 1
================================================================================
  network : doub_shft                                                nodes : 22
================================================================================
```

Figure 15. Results of a simulation of network *doub_shft*

In the functional description of *shiftregister* the terminals *in_out* are set on BTFREE when no new values are assigned to these terminals during the evaluation of the function block. When this would not be done, the logic values that were inputed from the bus and assigned to the terminals *in_out*, would be outputted again. This would give rise to a 'pass-through' function in the behavior of the function block that was not intended.

The network of the third example is shown in figure 16. In this network the function block *count1* is called as a subblock. The description of this function block can be found in figure 9. The SLS description of the network *count_netw* is shown in figure 17.

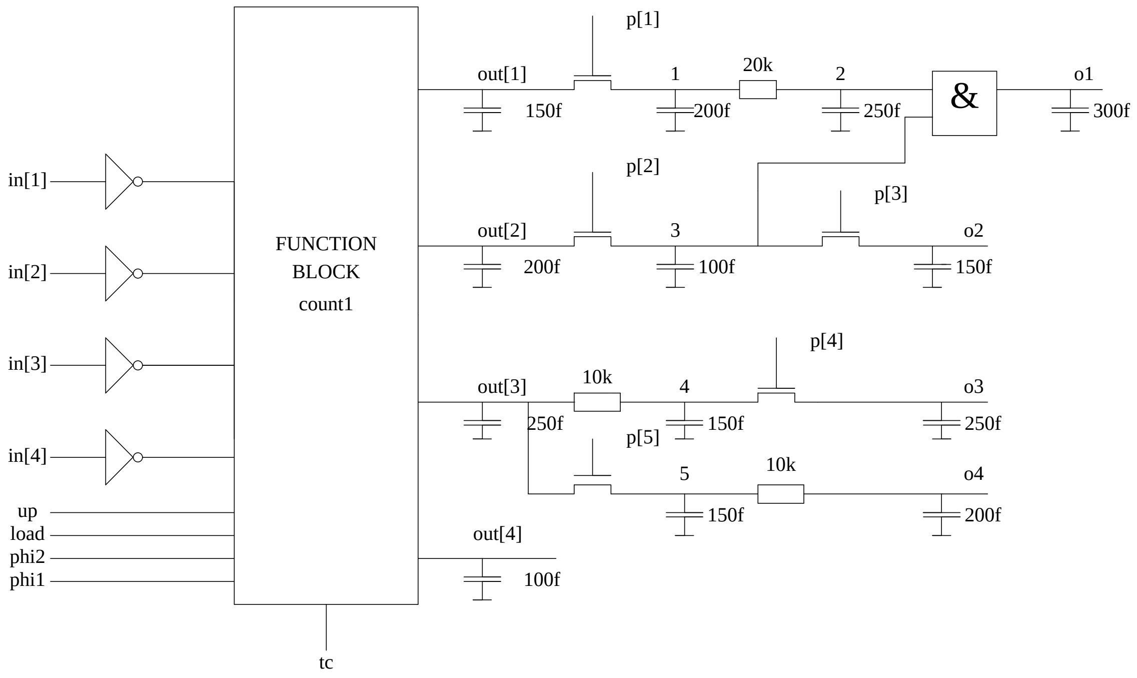


Figure 16. Network diagram of network *count_netw*

```
network invertor (terminal vdd, vss, i, o)
{
    nenh  w=16u l=4u  (i, o, vss);
    ndep  w=8u  l=8u  (o, vdd, o);
}

network count_netw (terminal vdd, vss, phi1, phi2, in[1..4], up, load, p[1..5],
                                out[1..4], tc, in_in[1..4])
{
    {inst1} @ count1 (phi1, phi2, in_in[1..4], up, load, out[1..4], tc);

    {inst_inv[1..4]} invertor (vdd, vss, in[1], in_in[1],
                               vdd, vss, in[2], in_in[2],
                               vdd, vss, in[3], in_in[3],
                               vdd, vss, in[4], in_in[4]);

    cap 150f (out[1], gnd);
    cap 200f (out[2], gnd);
    cap 250f (out[3], gnd);
    cap 100f (out[4], gnd);

    cap 300f (o1, gnd);
    cap 150f (o2, gnd);
    cap 250f (o3, gnd);
    cap 200f (o4, gnd);

    cap 200f (1, gnd);
    cap 250f (2, gnd);
    cap 100f (3, gnd);
    cap 150f (4, gnd);
    cap 150f (5, gnd);

    nenh w=8u l=4u (p[1], out[1], 1);
    nenh w=8u l=4u (p[2], out[2], 3);
    nenh w=8u l=4u (p[3], 3, o2);
    nenh w=8u l=4u (p[4], 4, o3);
    nenh w=8u l=4u (p[5], out[3], 5);

    @ nand (3, 2, o1);

    res 20k (1, 2);
    res 10k (out[3], 4);
    res 10k (5, o4);
}
```

Figure 17. Network description of network *count_netw*

A simulation of this network will again be a mixed-level simulation. The function block *count1* is used in a network full of transistors, resistors and capacitances. As can be seen in figure 9, the input load as well as the timing properties of the function block are defined in the functional description of the function block. So in this example simulations will be shown on level 3: timing simulations.

```
================================================================================
                                     S L S
                                  version: 3.0
                      S I M U L A T I O N   R E S U L T S
================================================================================
 time         | p p   l u   i i i i   i i i i     p p p p p     o o o o   t
 in 1e-09 sec | h h   o p   n n n n   n n n n     * * * * *     u u u u   c
              | i i   a     * * * *   _ _ _ _     1 2 3 4 5     t t t t
              | 1 2   d     1 2 3 4   i i i i     * * * * *     * * * *
              |             * * * *   n n n n                   1 2 3 4
              |                       * * * *                   * * * *
              |                       1 2 3 4
              |                       * * * *
================================================================================
        0.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 0 0 0 0     x x x x   x
       10.000 | 1 0   1 1   1 1 1 1   1 1 1 1     0 0 0 0 0     x x x x   x
       10.539 | 1 0   1 1   1 1 1 1   0 1 1 1     0 0 0 0 0     x x x x   x
       10.702 | 1 0   1 1   1 1 1 1   0 0 1 1     0 0 0 0 0     x x x x   x
       10.864 | 1 0   1 1   1 1 1 1   0 0 0 1     0 0 0 0 0     x x x x   x
       11.026 | 1 0   1 1   1 1 1 1   0 0 0 0     0 0 0 0 0     x x x x   x
       20.000 | 0 0   1 1   0 0 0 0   0 0 0 0     0 0 0 0 0     x x x x   x
       23.509 | 0 0   1 1   0 0 0 0   1 0 0 0     0 0 0 0 0     x x x x   x
       25.038 | 0 0   1 1   0 0 0 0   1 1 0 0     0 0 0 0 0     x x x x   x
       26.568 | 0 0   1 1   0 0 0 0   1 1 1 0     0 0 0 0 0     x x x x   x
       28.098 | 0 0   1 1   0 0 0 0   1 1 1 1     0 0 0 0 0     x x x x   x
       30.000 | 0 1   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     x x x x   0
       30.099 | 0 1   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     x x x 1   0
       30.299 | 0 1   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     x 1 x 1   0
       30.599 | 0 1   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     1 1 x 1   0
       30.649 | 0 1   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     1 1 1 1   0
       40.000 | 0 0   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     1 1 1 1   0
       50.000 | 1 0   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     1 1 1 1   0
       60.000 | 0 0   1 1   0 0 0 0   1 1 1 1     1 1 0 1 0     1 1 1 1   0
       70.000 | 0 1   1 1   0 0 0 0   1 1 1 1     0 1 1 0 1     1 1 1 1   0
       80.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 1 1 0 1     1 1 1 1   0
       90.000 | 1 0   0 1   0 0 0 0   1 1 1 1     0 1 1 0 1     1 1 1 1   0
      100.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 1 1 0 1     1 1 1 1   0
      110.000 | 0 1   0 1   0 0 0 0   1 1 1 1     x x x x x     1 1 1 1   0
      110.099 | 0 1   0 1   0 0 0 0   1 1 1 1     x x x x x     1 1 1 0   0
      110.199 | 0 1   0 1   0 0 0 0   1 1 1 1     x x x x x     1 0 1 0   0
      110.599 | 0 1   0 1   0 0 0 0   1 1 1 1     x x x x x     0 0 1 0   0
      110.999 | 0 1   0 1   0 0 0 0   1 1 1 1     x x x x x     0 0 0 0   0
      120.000 | 0 0   0 1   0 0 0 0   1 1 1 1     x x x x x     0 0 0 0   0
      130.000 | 1 0   0 1   0 0 0 0   1 1 1 1     x x x x x     0 0 0 0   0
      140.000 | 0 0   0 1   0 0 0 0   1 1 1 1     x x x x x     0 0 0 0   0
      150.000 | 0 1   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 0   0
      150.099 | 0 1   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 1   0
      160.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 1   0
      170.000 | 1 0   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 1   0
      180.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 1   0
      190.000 | 0 1   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 1   0
      190.099 | 0 1   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 0 0   0
      190.999 | 0 1   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 1 0   0
      200.000 | 0 0   0 1   0 0 0 0   1 1 1 1     0 1 1 1 1     0 0 1 0   0
================================================================================
  network : count_netw                                              nodes : 34
================================================================================
```

Figure 18. Results of a simulation of network *count_netw*

In figure 18 results are shown of a simulation of the network *count_netw*. The simulation begins with loading the counter. For that purpose data is set on the nodes *in[1]* to *in[4]* in the network. This data is inverted and placed on the nodes *in_in[1]* to *in_in[4]* which are

connected with the inread terminals *in[0]* to *in[3]* of the function block. The load of the inread terminals has been defined in the description of the function block. The effect of this load can be seen in the times it takes to charge or discharge the nodes that are connected with the terminals.
When the counter is loaded, the counter can start counting; in this case the counter will count ’up’. When *phi2* is high the output terminals can change. For the output terminals the rise and fall delay times are defined (see figure 9). The delay time is the time between the moment the value of the output terminal changes and the moment this change will influence the logic state of the node that is connected with the output terminal. As can be seen in the functional description of the block *count1*, the delay times of the output terminals depend of the load of the node the terminals are connected with. This load depends on the state of the transistors in the network. In the results of the simulation is shown that the delay times of the various output terminals differ and that they change depending on the states of the pass-transistors *p[1]* to *p[5]*.

The last example describes the simulation of the function block *combinator*, of which the functional description can be found in figure 10. The network *comb_netw* is created that contains only the function block. The description of the network is shown in figure 19.

```
network comb_netw (terminal phi, oinv, oand, onand, oor, onor,
                            oexor, oexnor, osp[1..2], us_in[1..4])
{
  {inst_comb} @ combinator (phi, oinv, oand, onand, oor, onor,
                            oexor, oexnor, osp[1..2], us_in[1..4]);
}
```

Figure 19. Network description of network *comb_netw*

When this simulation is executed, input is asked to the user by the function block (see figure 10). Therefore the results of the simulation depend on the data given by the user. The command file for a simulation is shown in figure 20.

```
set phi = (l*1 h*1)*16

option level = 1
option outunit = 1n
option outacc = 10p
option sigunit = 100n

print  us_in[1..4],,oinv,oand,onand,oor,onor,oexor,oexnor,osp[1..2]
```

Figure 20. Simulation input for simulation of network *comb_netw*

The node *phi* is set to O and I for sixteen times. Each time the node *phi* changes logic state, the function block is evaluated. When the logic state is high, the value of the input terminal *phi* of the function block is BTTRUE and the user is asked to typ in four characters. These characters must be ’O’ or ’X’ or ’I’,so the value of the output terminals can be determined. The nodes that are connected with the output terminals are printed in the simulation output file, since the ’print’ command in the simulation input file is stated that way.

```
================================================================================
                                     S L S
                                  version: 3.0
                        S I M U L A T I O N   R E S U L T S
================================================================================
 time         | u u u u    o   o   o   o   o   o   o   o o
 in 1e-09 sec | s s s s    i   a   n   o   n   e   e   s s
              | _ _ _ _    n   n   a   r   o   x   x   p p
              | i i i i    v   d   n       r   o   n   * *
              | n n n n            d           r   o   1 2
              | * * * *                            r   * *
              | 1 2 3 4
              | * * * *
================================================================================
       100.00 | 0 0 0 1    1   0   1   0   0   0   0   1 1
       300.00 | 0 0 1 1    1   0   1   1   0   1   1   0 0
       500.00 | 0 1 0 1    1   0   1   1   0   1   1   0 0
       700.00 | 0 1 1 1    1   0   1   1   0   0   0   0 0
       900.00 | 0 0 0 0    1   0   1   0   1   0   1   1 0
      1100.00 | 0 0 1 0    1   0   1   1   0   1   0   0 0
      1300.00 | 0 1 0 0    1   0   1   1   0   1   0   1 1
      1500.00 | 0 1 1 0    1   0   1   1   0   0   1   1 1
      1700.00 | 1 0 0 0    0   0   1   1   0   1   0   0 0
      1900.00 | 1 0 0 1    0   0   1   1   0   1   1   0 0
      2100.00 | 1 0 1 1    0   0   1   1   0   0   0   1 1
      2300.00 | 1 1 0 1    0   0   1   1   0   0   0   1 1
      2500.00 | 1 1 1 1    0   1   0   1   0   1   1   0 1
      2700.00 | 1 0 1 0    0   0   1   1   0   0   1   1 1
      2900.00 | 1 1 0 0    0   0   1   1   0   0   1   1 0
      3100.00 | 1 1 1 0    0   1   1   1   0   1   0   0 0
================================================================================
  network : comb_netw                                               nodes : 14
================================================================================
```

Figure 21. Simulation results of simulation of network *comb_netw*

In figure 21 the output of a simulation is shown. The logic states of the nodes *us_in[1]* to *us_in[4]* are the characters typed by the user.

4.2 Mixed Level Simulations in the NELSIS-VLSI-design system

In this section a sequence of actions and commands is shown that can be used when using the Mixed Level Simulation package. The example in the previous section in which the simulation of the network *doub_shft* was discussed, will be used here. To be able to perform simulations first the several descriptions have to be created. Suppose the functional description of the block *shiftregister* has been placed in the file called ’shift.fun’ and the network description of the network *doub_shft* has been placed in the file ’doub_shft.sls’. Then the following actions can be executed in an nMOS project directory:

cfun shift.fun	information about the function block described in the file ’shift.fun’ will be placed in the database. The functional description is converted to C functions which are compiled

	and placed in the database.
csls doub_shft.sls	the network described in the file ’doub_shft.sls’ is placed in the database.
sls_exp doub_shft	a binary description of the cell *doub_shft* is created and placed in the database. An executable program **sls** is created by linking the object files describing the function block(s) that are used in the network and the object files describing the Switch Level Simulator. This will only be done when no executable program **sls** exist, in which the function block(s) already are incorporated. The files ’deffunc.c’ and ’sls.funlist’ are created as well.
sls doub_shft doub_shft.cmd	the cell *doub_shft* is simulated using the commands contained by the command file ’doub_shft.cmd’.

References

1. A.C. de Graaf and A.J. van Genderen, ‘‘SLS: SWITCH-LEVEL SIMULATOR USER’S MANUAL,’’ The Nelsis IC Design System Documentation, Delft University of Technology (1988).

CONTENTS

LIST OF FIGURES

LIST OF TABLES