C/UNIX Functions for VHDL Testbenches

Michael J. Knieser	Francis G. Wolff	Chris A. Papachristou
Rockwell Automation	Case Western Reserve University	Case Western Reserve University
mjknieser@ra.rockwell.com	fxw12@po.cwru.edu	cap2@po.cwru.edu

Abstract

A VHDL library is presented here which allows a designer to quickly develop test benches, monitors and virtual In Circuit Emulators (ICE) for simulation. This library mimics most of the useful C string functions, C file processing functions and UNIX-like pattern matching functions. This allows the VHDL designer to think in the C/UNIX language paradigm. These features are demonstrated for a basic virtual ICE-wrapper test bench implementation.

1.0 Introduction

When generating test benches for testing a hardware design, there are many ways this test bench can be written. There are third party test bench generation tools and languages. However to maintain a design with its tests in the most neutral format, the design and its tests should be described in the most neutral format. For hardware designs the most neutral format would be VHDL or Verilog. For software the most neutral format would be ANSI C.

For some designers, the desire is to use C as the test bench description to test the hardware design in VHDL or Verilog. This is achievable; however, this is accomplished through hardware simulation tool foreign interfaces. Since this method requires simulation dependent interfaces, this would not be the most neutral format to describe the hardware test bench.

The issue with C programmers is not so much the VHDL language syntax but the need of including standard C libraries [1] such as string and stream file I/O. For example, C programmers find it difficult to write "`printf("i=%d\n", i);`" as "`write(line,string'("i=")); write(line, i); writeline(output, line);`".

In order to please those designers desiring to use C as the test bench description and maintain the most neutral format for describing the test bench and avoid language syntax switching, the most useful C functions should be implemented in VHDL or Verilog. This paper discusses the implementation details of emulating C/UNIX functions in the VHDL language.

2.0 Issues implementing C functions within VHDL

There are many issues implementing C functions in VHDL. There are dissimilarities and similarities. Unlike VHDL, C makes no distinction between a function and a procedure (i.e. functions can be used like procedures. Unlike C, VHDL supports concurrency in addition to the

C sequential language. These issues reveal some of the inherit difficulties of converting C language algorithms into VHDL.

2.1 Why not do C++?

The main limiting factor for not implementing C++ functions is that VHDL does not have language support for C++ like objects. Since C++ builds on C traditional libraries, the use of any preprocessors which could support C++ objects was avoided in favor of using plain C.

2.2 Pointers

VHDL supports pointer access (i.e. VHDL keyword "access") in a very restrictive way. It does not support pointer arithmetic and address referencing. C statements like "**`char s[10], *p=s+2;`**" or "**`strcpy(p, &s[2]);`**" or "**`strcpy(p, s+2);`**" are not possible in VHDL. Since VHDL has strict typing it, disallows the C language type casting as shown as follows: "**`p = (char *)integer_pointer;`**" The C concept of strings cannot be directly equivalent to VHDL due to these limitations.

2.3 String Processing

There are conceptual differences between C and VHDL text strings. In C, a string is an array of character integers. Since characters behave as small integers, they can be easily used in arithmetic expressions (i.e. "**`char c; c=c–'A'+32;`**"). In VHDL, a string is an array of enumerated character types [2], described as "**`TYPE string IS ARRAY (POSITIVE RANGE <>) OF character;`**" and the definition of a character is shown as follows:

```
TYPE character IS ( NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF,
VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM,
SUB, ESC, FSP, GSP, RSP, USP, ' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/', '0', '1', '2', '3', '4', '5',
'6', '7', '8', '9', ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C',
'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q',
'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{',
'|', '}', '~', DEL );
```

The definition of string in C (i.e. "**`char s[10];`**") represents the maximum workspace and the string length varies within the allocated workspace. In VHDL, the string definition and string length are the same (i.e. "**`VARIABLE s: string(1 TO 10);`**"). These fixed length strings in VHDL make string processing unfriendly. In C, the string index begins with 0 and VHDL begins with 1. The use of a different starting index does not present any real difficulty in translating.

In C, the end of a string is terminated by a special character (i.e. '\0'). If the string terminator is not within the allocated workspace, the string copy function in C will copy beyond the workspace over writing other data in the user's workspace. In this way, VHDL string processing is actually safer to use because the allocated string length is always known.

2.4 Most common string functions

```
C prototypes
      #include <strings.h>
      char  *strcpy(char *dest,     const char *src);
      char  *strcat(char *dest,     const char *src);
      int    strcmp(const char *s1, const char *s2);
      size_t strlen(const char *s);
VHDL prototypes
      LIBRARY C;
      USE C.STRINGS_H.ALL;
      procedure strcpy(dest: OUT   string; src: IN string);
      procedure strcpy(dest: OUT   string; src: IN character);
      procedure strcat(dest: INOUT string; src: IN string);
      function  strcmp(s1:   IN    string; s2:  IN string) return integer;
      function  strlen(s:    IN    string) return integer;
```

The strcpy function copies the source string up to the allocated string length or the terminating VHDL NUL character to the destination string. The destination string will be truncated if the source string is longer than the allocated destination space. Since there is only one unique source of array type, the VHDL strcpy allows for typeless string constants. This allows writing "**`strcpy(s, "hello");`**" instead of "**`strcpy(s, string'("hello"));`**" Special cases for copying characters were also added: **`strcpy(s, c);`** or **`strcpy(s, 'c');`**

The basic VHDL for "**`strcpy`**" is shown as follows:

```
      procedure strcpy(dest: OUT string; src: IN string) is
        variable dj: integer:=dest'left;
        variable sj: integer:=src'left;
      begin
        loop
          if dj>dest'right then dest(dest'right):=NUL; exit; end if;
          if sj>src'right  then dest(dj):=NUL; exit; end if;
          if src(sj)=NUL   then dest(dj):=NUL; exit; end if;
          dest(dj):=src(sj); dj:=dj+1; sj:=sj+1;
        end loop;
      end procedure;
```

Another difference between the C and VHDL string copy is that the strcpy does not return anything. In addition, VHDL knows the length and it will not overwrite beyond the allocated length of the string. The strcpy also allows for using array slices: "**`strcpy (s, t(4 to t'length));`**".

The strcat function appends the source string to the destination string overwriting the NUL character at the end of destination, and then adds a terminating NUL character. The strcmp function compares two strings and returns the zero if equal, positive if s1 is greater and negative if s1 is less than s2. The strlen function returns the number of characters in the string, not including the terminating NUL character.

2.5 Passing variable number of arguments

In C, a function can have a variable argument list by using the "**`varargs.h`**" library and the ellipsis operator "**`...`**". By using the VHDL default argument assignment, variable argument lists can be emulated [3] in a limited way. This requires the pre-determining of the largest argument list or worst case situation as shown below:

```
      procedure fprintf
                  ( F      : OUT   text;
                    Format : IN    string;
```

```
    A1 , A2 , A3 , A4 , A5 , A6 , A7 , A8 : IN string := FIO_NIL;
    A9 , A10, A11, A12, A13, A14, A15, A16: IN string := FIO_NIL;
    A17, A18, A19, A20, A21, A22, A23, A24: IN string := FIO_NIL;
    A25, A26, A27, A28, A29, A30, A31, A32: IN string := FIO_NIL
  );
```

2.6 Passing Different Data Types

Variable argument data types are not directly recognized syntactically in the C language at compile time. The format control string within the printf function indicates which data types need to be converted for output. For example "**`printf("%s %d",a, b)`**" will interpreted through the format string the variable, **`a`**, as a character string (i.e. %s) and the variable, **`b`**, as an integer (i.e. %d).

The problem is that VHDL has strict data typing and VHDL function overloading is used to overcome this issue. The overloaded VHDL pf function can be used to convert all data types to a common string data type. For example, in C the "**`printf("%s %d",a,b)`**" would be translated to "**`printf("%s %d",a,pf(b))`**" in VHDL. The following shows a sample list of overloaded functions:.

```
function pf (Arg: std_logic_vector)  return string;
function pf (Arg: std_ulogic_vector) return string;
function pf (Arg: bit_vector)        return string;
function pf (Arg: integer)           return string;
function pf (Arg: character)         return string;
function pf (Arg: string)            return string;
```

The pf functions can be avoided by permutating the data types with additional overloaded functions. For example, the following overloaded printf functions were developed.

```
procedure printf(format: string; Arg1: string;       Arg2: string);
procedure printf(format: string; Arg1: integer;      Arg2: string);
procedure printf(format: string; Arg1: string;       Arg2: integer);
procedure printf(format: string; Arg1: integer;      Arg2: integer);
```

These overloaded functions allow the ability to mimic the C printf function as follows:

```
printf("%s %d", a, b);
```

For sscanf procedure the code skeleton would be as follows:

```
procedure sscanf(s:string; format:string; Arg1:string; Arg2:integer;
Arg3:std_logic_vector)
  variable si: integer:=1; variable fi: integer:=1;
begin
   isscanf(si, fi, s, format, Arg1);
   isscanf(si, fi, s, format, Arg2);
   isscanf(si, fi, s, format, Arg2);
end sscanf;
```

The variables **si** and **fi** point to the next character to look at in the scan string and format string, respectively. Due to VHDL overloading, the correct isscanf data type procedure is linked.

2.7 Returning values

In C, a declared function can be treated as a procedure. In VHDL, the caller must use the return value for a function. The nice feature of a function is that one can use it as part of a conditional expression. For example, "**`if (sscanf(s, "%x %x", &v1, &v2)==2) …`**" but unfortunately, unlike C, VHDL functions are not allowed to pass values back (i.e. **`OUT, INOUT`**) through the argument list. Unlike C, return values must be assigned and cannot be discarded. C makes no

distinctions between functions and procedures. For the VHDL sscanf, a function was added to just return the number of matched arguments as shown as follows: "**`if (sscanf(s, "%x %x")=2) then …`**".

In C, it is not uncommon to write "**`printf("v1=%x\n");`**" or "**`n=printf("v1=%x\n");`**" interchangeably. However, in VHDL, the former example is a function and the latter a procedure. It was decided to implement the most common usage of printf function as a VHDL procedure.

3.0 Common VHDL C Testbench Functions

With the differences between C and VHDL, this paper covers the following C functions that were converted to VHDL which were determined to be the most common. Each section will discuss the C function as it is defined and the VHDL implementation.

3.1 printf, fprintf & sprintf

```
C prototypes
      #include <stdio.h>
      int printf (              const char *format, ...);
      int fprintf(FILE *stream, const char *format, ...);
      int sprintf(char *str,    const char *format, ...);

VHDL prototypes
      LIBRARY C;
      USE C.STDIO_H.ALL;
      procedure printf (                   format:IN string; ...);
      procedure fprintf(stream:OUT text;   format:IN string; ...);
      procedure sprintf(str:   OUT string; format:IN string; ...);

      procedure printf (ret: OUT integer;                    format:IN string; ...);
      procedure fprintf(ret: OUT integer; stream:OUT text;   format:IN string; ...);
      procedure sprintf(ret: OUT integer; str:   OUT string; format:IN string; ...);
```

The functions in the printf family output according to a format control string. The printf function writes output to the standard output stream, stdout (i.e. same as "**`fprintf(output, format, …);`**"). The fprintf function writes it's output to the file variable . The sprintf function writes it's output to a character string.

3.2 scanf, fscanf & sscanf

```
C prototypes
      #include <stdio.h>
      int scanf(                const char *format, ...);
      int fscanf(FILE *stream,  const char *format, ...);
      int sscanf(const char *str, const char *format, ...);
VHDL prototypes
      LIBRARY C;
      USE C.STDIO_H.ALL;
      procedure scanf(                   format:IN string; ...);
      procedure fscanf( stream:OUT text; format:IN string; ...);
      procedure sscanf( str:   IN  string;format:IN string; ...);

      procedure scanf(  ret:OUT integer;                  format:IN string; ...);
      procedure fscanf( ret:OUT integer; stream:OUT text; format:IN string; ...);
      function  sscanf( str:IN  string;           format:IN string ) return integer;
```

The scanf family of functions scans the input according to a format control string. This format may contain conversion specifiers. The results from such conversions are read from the standard input stream stdin for scanf function, read from the file stream for fscanf and read from the character string for sscanf. Upon successful completion, these functions return the number of successfully matched and assigned input items.

3.3 Line and Character Stream I/O

```
C prototypes
       #include <stdio.h>
       char *gets(   char *s);
       char *fgets(  char *s, int size, FILE *stream);
       int   puts(   const char *s);
       int   fputs(  const char *s, FILE *stream);
       int   fputc(  int c, FILE *stream);
       int   putchar(int c);
       int   fgetc(  FILE *stream);
       int   getchar(void);
VHDL prototypes
       LIBRARY C;
       USE C.STDIO_H.ALL;
       procedure gets(   s:OUT string);
       procedure fgets(  s:OUT string; size:IN integer; stream:OUT text);
       procedure puts(   s:IN  string);
       procedure fputs(  s:IN  string;    stream:OUT text);
       procedure fputc(  c:IN  character; stream:OUT text);
       procedure putchar(c:IN  character);
       prodecure fgetc(  c:OUT character; stream:OUT text);
       procedure getchar(c:OUT character);
```

The VHDL fgets and fputs line I/O functions are shown as follows:

```
       procedure fgets (s: OUT  string; size: IN integer; stream: OUT text) is
         variable LineBuffer: line;
       begin
         readline(stream, LineBuffer); strcpy(s, LineBuffer.ALL & NUL);
       end procedure;

       procedure gets (s: OUT  string) is
         variable LineBuffer: line;
       begin
         readline(INPUT, LineBuffer); strcpy(s, LineBuffer.ALL & NUL);
       end procedure;
```

The VHDL readline procedure is equivalent to a string of text with a terminating newline. For text files this mapping is straightforward.

The fgetc and fputc character I/O functions are limited to one open input and one output file at a time. The limitation is due to using VHDL shared variables and the procedures can not distinguish what file stream it is currently looking at.

```
       procedure fgetc(c: OUT character; fp: OUT text) is
       begin
         if fgetc_i = 0 then
           readline(fp, fgetc_buffer); fgetc_i:=1;
         end if;
         if fgetc_i > fgetc_buffer'length then
           readline(fp, fgetc_buffer); fgetc_i:=1; c:=LF;
         else
           if endfile(input) then
             c:=NUL;
```

```
      else
        c:=fgetc_buffer(fgetc_i); fgetc_i:=fgetc_i+1;
      end if;
    end if;
  end fgetc;
```

3.4 Common string functions simulating pointer arithmetic

```
VHDL prototypes
LIBRARY C;
USE C.STRINGS_H.ALL;
procedure strcpy(dest: OUT   string;                  src: IN string);
procedure strcpy(dest: OUT   string;                  src: IN string; si: IN integer);
procedure strcpy(dest: INOUT string; di:  IN integer; src: IN string);
procedure strcpy(dest: INOUT string; di:  IN integer; src: IN string; si: IN integer);
procedure strcat(dest: INOUT string;                  src: IN string);
procedure strcpy(dest: OUT   string;                  src: IN string; si: IN integer);
procedure strcpy(dest: INOUT string; di: IN integer;  src: IN string);
procedure strcpy(dest: INOUT string; di: IN integer;  src: IN string; si: IN integer);
procedure strlen(s:    IN    string; si: IN integer);
```

By heavily using VHDL overloading, functions were added to allow limited pointer like arithmetic on strings. The above functions allows all possible cases of **"strcpy(dest + di, src + si);"** For example: "**strcpy(s, t(4 to t'length))**" can be written better as "**strcpy (s, t, 4);**" Also, **"strcat(dest + di, src + si);"** and **"strlen(s + si);"** are also supported.

3.5 Character processing

```
C prototypes
      #include <ctype.h>
      int isalpha(int c);
      int isupper(int c);
      ...
VHDL prototypes
      LIBRARY C;
      USE C.CTYPE_H.ALL;
      function  isalpha(c: character)  return boolean;
      function  isupper(c: character)  return boolean;
      function  islower(c: character)  return boolean;
      function  isdigit(c: character)  return boolean;
      function  isxdigit(c: character) return boolean;
      function  isalnum(c: character)  return boolean;
      function  isspace(c: character)  return boolean;
      function  ispunct(c: character)  return boolean;
      function  isprint(c: character)  return boolean;
      function  isgraph(c: character)  return boolean;
      function  iscntrl(c: character)  return boolean;
      function  isascii(c: character)  return boolean;
      function  tolower(c: character)  return character;
      function  toupper(c: character)  return character;
```

The complete "ctype.h" include library is implemented like the C version. The following example shows the use of ctype functions by converting a file to lower case.

```
      process
        variable c: character;
        FILE     fp: text OPEN READ_MODE  IS "debugio_h_testlib.sh";
        FILE     fw: text OPEN WRITE_MODE IS "debugio_h_testlib.out";

      begin
        while not endfile(fp) loop
          fgetc(c, fp);
```

```
    if isalpha(c) then
      fputc(tolower(c), fw);
    else
      fputc(c, fw);
    end if;
  end loop;
  fclose(fw);
  wait;
end process;
```

4.0 Applications of the C Functions

Two test benches were developed in order to demonstrate the features of this suite of C functions. The first VHDL test bench tests the interface of a microprocessor. The second VHDL test bench behaves as an in-circuit emulator (ICE) interfacing to a microprocessor.

4.1 Microprocessor Host Test Bench

A common suite of test benches when designing ASIC's with an external microprocessor interface is to have at least a functional host model of the desired microprocessor. This functional model would have the specific interface and would have some method for easily issuing microprocessor commands. Figure 3 shows a high-level architectural view of how this test bench would be organized.

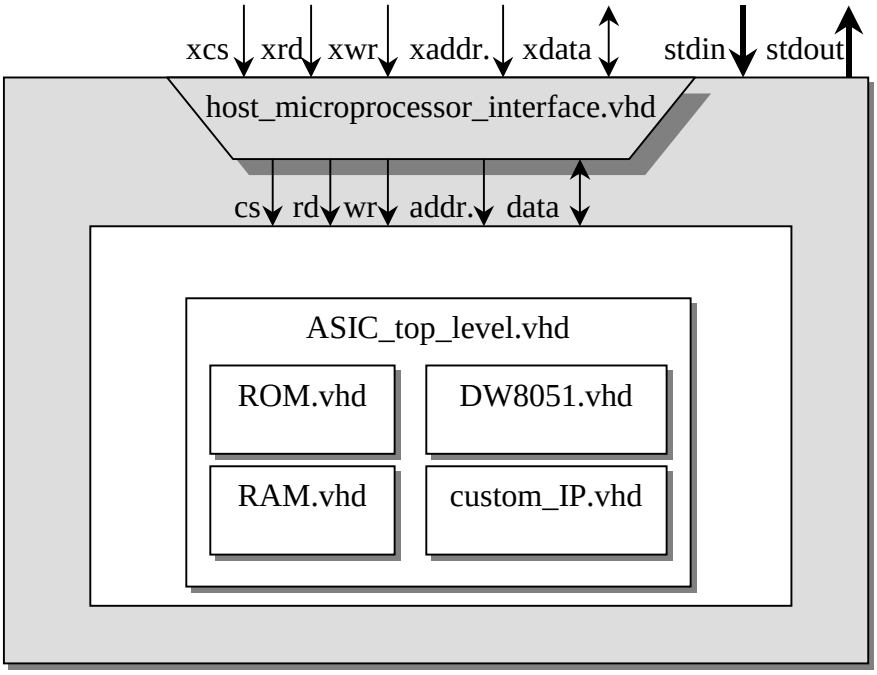

Figure 3: Microprocessor host test bench high-level architecture.

Within the "host_microprocessor_interface.vhd," this test bench would have a main process that would allow the test bench to be exercised. This main process is shown as follows.

```
process
begin
  wr <= '0'; rd <= '0'; cs <= '0';
  data_out <= "ZZZZZZZZ"; address <= "xxxxxxxxxxxxxxxx";
```

```
    printf("Host Microprocessor shell (Version 1.0)\n");
    printf("Type 'help' for command list.\n");

    loop
      printf("Host => ");
      gets(what_next);
      if ( sscanf(what_next,"help %s") = 1 ) then
        printf("help                  - Display's current supported commands\n");
        printf("write <address> <data>- Write <data> to <address>\n");
        printf("read <address> <data> - Read <address> and expect <data>\n");
        printf("load hex              - Marker to indicate hex data is next\n");
        printf("dump <start> <end>    - Dump data from <start> to <end> addr\n");
        printf("config <parm>=<time>  - Set <parm> to the <time> value\n");

      elsif ( sscanf(what_next,"write %x %x") = 2 ) then
        sscanf(what_next,"write %x %x",address,data_out);
        wait for CS_START_DELAY;
        cs <= '1'; wait for WR_START_DELAY;
        wr <= '1'; wait for WRITE_WIDTH;
        wr <= '0'; wait for CS_END_DELAY;
        cs <= '0'; wait for WR_END_DELAY;
        address <= "xxxxxxxxxxxxxxxx"; data_out <= "ZZZZZZZZ";

      elsif ( sscanf(what_next,"read %x %x") = 2 ) then …
      elsif ( sscanf(what_next,"dump %x %x") = 2 ) then …
      elsif ( sscanf(what_next,"load %s") = 1 ) then …
      elsif ( sscanf(what_next,"config %s=%d%s") = 3 ) then …
      else
        -- passthru wrapper signals
        cs <= xcs;  wr <= xwr;  rd <= xrd;  address <= xaddress;
        data_out <= xdata_out;  xdata_in <= data_in;
      end if;
    end loop;
  end process;
```

The above VHDL process would enable interactive toggling of the test bench and its interface to the target ASIC. In addition, this process would enable the load and execution of a pre-designed test for eventual regression testing.

4.2 In Circuit Emulator Wrapper Test Bench

Another common suite of test benches when designing and debugging ASIC's it to create a in-circuit emulator wrapper to effectively bypass certain logic and control the logic under test through another means. Figure 4 illustrates a high-level architecture for an in circuit emulator wrapper leveraging standard input and standard output as the secondary method for controlling logic under test.

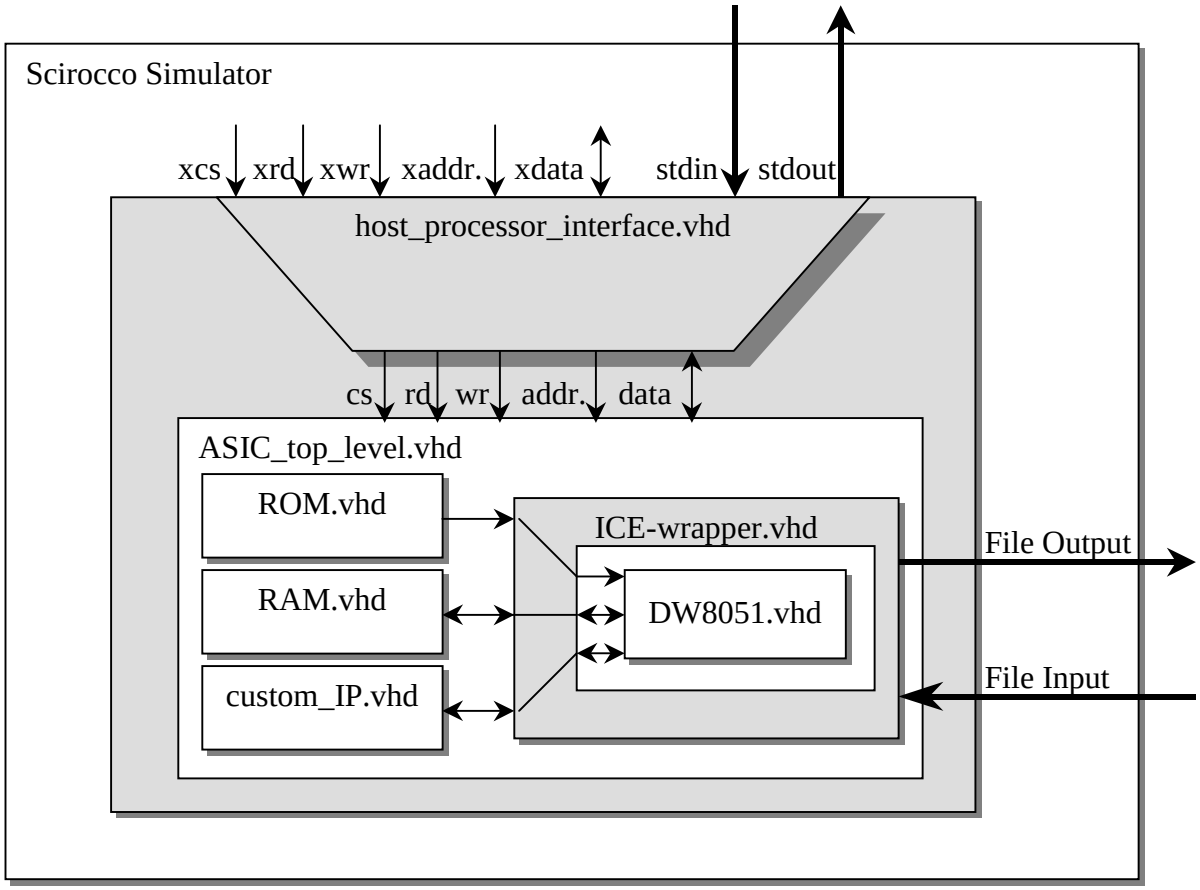

Figure 4: ICE test bench high-level architecture.

5.0 Conclusion

The most commonly standard C functions were written as C friendly as possible within VHDL. These functions and libraries simplify and ease the coding of test benches and wrappers. These libraries and their VHDL sources are freely available on the Internet [4].

6.0 References

[1] Kernighan, Brian, and Ritchie, Dennis, “The C Programming Language,” Prentice Hall, 1988.

[2] Ashenden, Peter J., “The Student’s Guide to VHDL,” Morgan Kaufmann Publishers, San Francisco, 1998.

[3] Decaluwe, Jan, “PCK_FIO.vhd,” GNU General Public License, Easics NV, Interleuvenlaan 86, B-3001 Leuven, Belgium, jand@easics.be, http://www.easics.com/method/inhouse.html

[4] Case Western Reserve University, Electrical Engineering & Computer Science Department, Computer Engineering Group, Cleveland, Ohio, http://bear.ces.cwru.edu/vhdl