SNUG SAN JOSE 2002 2004 Update

C/UNIX Functions for VHDL Testbenches

(with additional notes on Unix pipes & rsh)

Updated for 2004 version

Michael J. Knieser
Rockwell Automation
mjknieser@ra.rockwell.com

Francis G. Wolff
Case Western Reserve University
fxw12@po.cwru.edu

Chris A. Papachristou
Case Western Reserve University
cap@eecs.cwru.edu

Starting with “stdio_h” version 3.0 library, you can do the following:

```
VARIABLE fp, fp2: CFILE;              --use "CFILE" not "FILE"
fp:=fopen("pipe1", "w");              --file_open(...) not used.
if fp=0 then fprintf(stderr, "cannot open file\n"); end if;
fprintf(fp,"ALU_OUT = %u\n",v);
fp2:=fopen("data.txt", "r");
while not feof(fp) loop
   c:=fgetc(fp); fputc(c, fp);
end loop;
fclose(fp2); fclose(fp);
```

The goal of “stdio_h” library was to be as faithful to UNIX/C programming style as possible. The VHDL “file_open” and “WRITE_MODE” was always difficult to remember especially when switching from C-style to VHDL-style. Also, everything has been re-written for “big-endian” numbers (all users requested this feature).

SNUG SAN JOSE 2002 2004 Update

Overview

- Motivation
- Issues Implementing C Functions within VHDL
- Common VHDL C Testbench Functions
- Applications of the C Functions (including Unix pipes & rsh)
- Conclusions

SNUG SAN JOSE 2002 — 2004 Update

Motivation

- Desire to maintain the most portable format for a design
  - "Portability" means that the design is not vendor dependent
  - For hardware… …the portable format is VHDL or Verilog.
  - For software… …the portable format is C.

- Simulation tool foreign interfaces are not the most portable
  - Design builds (netlist creation, simulation and verification results)
  - Use of wrappers for ICE & Embedded Testbenches
  - Use of Unix pipes to glue together external EDA tools (including legacy tools)

- Like to keep testbench coding simple…
  - In C… …printf("i=%d\n",i);
  - In VHDL… …write(line,string'("i=")); write(line,i); writeline(output,line);

The C language contains only one "process" (i.e. the function "main( )") and supports only "sequential process" statements. Concurrency and parallelism occur indirectly through the use of compilers (i.e. instruction level parallelism), operating system calls (i.e. fork( ) and join( )), coroutines, pthreads, etc. The compiler's goal is to produce machine code for a known computer architecture (i.e. processor).

The C language was designed to make operating systems (i.e. Unix) as portable as possible by (1) being one step above assembly language (i.e. pointer arithmetic), (2) operating system features such as file access and i/o to be handled not within the language but by function calls to libraries (i.e. #include <stdio.h>) and (3) the compiler should be written in it's own language.

The VHDL language supports "sequential" and "concurrent" process statements with event scheduling. These features are necessary for hardware designs which are inherently parallel by nature. VHDL is foremost a "concurrent" simulation language for hardware architectures that do not typically exist yet or are inaccessible. This means that not every feature can be practically translated into hardware. A subset of the VHDL language allows a synthesizer to produce a netlist of gates representing new hardware architecture. It could also be said that the VHDL concurrent features is a superset of of the C language's sequential statements.

The VHDL/Verilog were developed ultimately to make hardware designs as portable as possible (i.e. ASIC, FPGA, different micron process technologies).

SAN JOSE 2002 2004 Update

Issues Implementing C Functions within VHDL

- Pointers
- String Processing
- Passing Variable Number of Arguments
- Passing Different Data Types
- Returning Values

SAN JOSE 2002 — 2004 Update

Pointer Issues

- VHDL implements pointer using “ACCESS”; however,...
  - It does not support pointer arithmetic
  - It does not support address referencing
  - due to VHDL strict typing, casting pointer is not supported

- EXAMPLES...

```
char s[10];  …   *p=s+2;    strcpy(p, &s[2]);  strcpy(p,s+2);

P = (char *)integer_pointer;
```

A classic example of ACCESS is used in TEXTIO:

```
USE  std.textio.all
TYPE bit       IS ('0', '1');
TYPE character IS (nul, soh, stx, etx, ..., 'A','B','C', ...);
TYPE string    IS ARRAY (POSITIVE range <>) of CHARACTER;
TYPE line      IS ACCESS STRING; --pointer to string of characters
...

USE     std.textio.all;
...
VARIABLE printf_buffer: line;   --char **printf_buffer;
...
write(printf_buffer, string'("ALU_OUT="));
write(printf_buffer, std_logic_vector'("100UX10"));
writeline(output, printf_buffer);
```

SNUG SAN JOSE 2002 — 2004 Update

String Processing Issues

- In C a string is an array of character integers
  - `char s[10]; s[1] = s[1] - 'A' + 32;`
- In VHDL a string is an array of enumerated character types
  - `TYPE string IS ARRAY ( POSITIVE RANGE <>) OF character;`
- In C a string is
  - Fixed in size allocation: "`char s[10];`"
  - '`\0`' is used to indicate the termination of a string.
  - But can easily read or write beyond the allocation if a '\0' is not found.
- In VHDL a string is fixed in size and cannot write beyond the limits.
  - `VARIABLE s: string(1 TO 10);`

C treats "**char**" as a "tiny integer" (i.e. 8 bit integer):

```
char c='A';    c='A'; c=65; c=0x41;  /* 8 bit integer */
int  i;        i='A'; i=65; i=0x41;  /* 32 bit integer */
```

other sizes would typically be: `short x;` (i.e. 16 bits); `long y;` (i.e. 64 bits);

C treats arrays as a type of pointer beginning from 0:

```
char s[10]="Bonjour";     c=s[0]; c=*(s+0); strcpy(s, t);
char *t="Konichiwa";      c=t[0]; c=*(t+0); t=s;
char w;                   w=t;  /* copy pointers */
```

C treats boolean TRUE as an "non-zero" integer value and FALSE as "zero".

```
int b;     b=10; if ( b ) { ... } else { ... }
```

VHDL treats the "character" as a type:

```
variable c: character:='A';     c:='A'; c:=character'val(65); c:=character'val(16#41#);
variable i: integer;                     i:=character'pos('A'); i:=65; i:=16#41#;
variable s: string(1 TO 10):="Bonjour   ";     c:=s(1); read(t, s);
variable t: line := new string'("Konichiwa");  c:=t(1); write(t, s);
variable w: line;                        w:=t;
variable b: boolean;                     b:=TRUE; if b then ... else ... end if;
```

SNUG SAN JOSE 2002 2004 Update

Passing Variable Number of Arguments

- A C function can use the “varargs.h” library and the ellipsis operator “…” to address any number of arguments

- VHDL supports a fixed number of arguments.

The VHDL workaround is to create the function with the expected largest argument list and utilize VHDL’s default argument assignments.

```
procedure fprintf
( stream          : INOUT  text;
  format          : IN     string;
  a1, a2, a3, a4  : IN     string := " ";
  a5, a6, a7, a8  : IN     string := " " );
```

Other examples:

```
procedure sprintf(s: INOUT line; format: IN string;
                  a1, a2, a3, a4, a5, a6, a7, a8 : IN string := " ";
                  a9, a10, a11, a12, a13, a14, a15, a16: IN string := " ");
procedure sprintf(s: INOUT string; format: IN string;
                  a1, a2, a3, a4, a5, a6, a7, a8 : IN string := " ";
                  a9, a10, a11, a12, a13, a14, a15, a16: IN string := " ");
```

SNUG SAN JOSE 2002 2004 Update

Passing Different Data Types

- Variable argument data types are not directly recognized syntactically in C at compile time.
  - `printf("%s %d", a, b);`
- VHDL has strict data typing.
  - The VHDL "`printf("%s %d", a, b);`" --a: string; b: integer
  - is a different procedure than "`printf("%d %s", a, b);`".

The VHDL workaround is to utilize VHDL's overloading capabilities and create all the most useful permutations of all possible data type.

C treats "functions" as "procedures" which return an optional value

```
int abs(int x) { if (x<0) { x=-x; } return x; }
int a; a=abs(-1);  abs(2);
```

VHDL functions and procedures are treated as 2 separate definitions:

```
function    abs(x: integer) return integer is begin if x<0 then x:=-x; end if; return x; end abs;
procedure abs(a: IN integer) is begin if x<0 then x:=-x; end if; return x; end abs;
variable a: integer; a:=abs(-1); abs(2);
```

VHDL functions have the following properties:

(1) Functions can only be passed "IN" (i.e. INOUT and OUT are not allowed).
(2) The keyword "IMPURE" allows functions to access data outside the function (i.e. global).
(3) Access types (i.e. LINE) are always treated as INOUT and cannot be function arguments.
(4) "FILE" data types cannot be used as function arguments or returned.
(5) Return value "must" used (i.e. it is not optional).

Other examples of VHDL overloading:

```
procedure printf(format: IN string; a1: string; a2: std_logic);
procedure printf(format: IN string; a1: integer; a2: std_logic);
procedure sscanf(s: IN string; format: IN string; a1: INOUT std_logic);
procedure sscanf(s: IN string; format: IN string; a1: INOUT std_logic_vector);
function pf(a1: IN time) return string;
function pf(a1: IN integer) return string;
```

SNUG SAN JOSE 2002 2004 Update

Returning Values

- VHDL functions are useful for ‘if’ or ‘while’ statements.
- A C function maps to a VHDL function given the following:
  - The C function’s caller never ignores the return value: n=atoi(s);
  - The C function’s arguments cannot modify original caller’s data: strcpy(d, s);
  - Otherwise the C function maps into a VHDL procedure.
- C prototypes

  int fscanf(FILE *stream,   const char *format, ...);

  int sscanf(const char *str, const char *format, ...);
- VHDL prototypes

  procedure fscanf(ret: OUT integer; stream: OUT text; format: IN string; ...);

  function sscanf(str: IN string; format: IN string) return integer; --Special case!

Designers would prefer to do:

```
if sscanf(s, “memset %x , %x”, address, data)=2 then …
```

than:

```
sscanf(n, s, “memset %x , %x”, address, data);
if n=2 then ...
```

Common VHDL C Testbench Functions

- printf, fprintf & sprintf
- scanf, fscanf & sscanf
- Character Stream I/O
- Common String Functions
- Additional Libraries

2004 Update

SNUG SAN JOSE 2002

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:IN CFILE;    format:IN string; ...);   --2004`
  - `procedure sprintf(str:   OUT string; format:IN string; ...);`
- Examples
  - `variable v:std_logic_vector(15 downto 0);`
    `printf("ALU_OUT = %20s(%#x)(%o)(%d)\n",v,v,v,v);`
  - `VARIABLE fp: CFILE:=fopen("pipe1", "w");               --2004 update`
    `fprintf(fp,"ALU_OUT = %u\n",v);`

The goal of "stdio_h" library was to be as faithful to UNIX/C style as possible.

The VHDL "**file_open**" was always difficult to remember especially when switching from C-style to VHDL-style. Also, remembering whether to use "**WRITE_MODE**" instead of "w"" or "**FILE**" instead of "**VARIABLE**" before a definition "fp" also seemed to be frustrating

Starting with "stdio_h" version 3.0 library

```
FILE fp: test OPEN WRITE_MODE IS "pipe1";
      fprintf(fp,"ALU_OUT = %u\n",v);
```

was change to

```
VARIABLE fp: CFILE:=fopen("pipe1", "w");
fprintf(fp,"ALU_OUT = %u\n",v);
```

for the following primary reason:
VHDL functions (i.e. not procedures) do not allow the passing of FILE type. So a new data type was introduced called "CFILE".

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:IN CFILE;  format:IN string; ...); --2004`
  - `procedure sscanf( str:   IN  string;format:IN string; ...);`
- Examples
  - ```
    variable v:std_logic_vector(15 downto 0);
    scanf("ALU_OUT = %x\n",v);
    ```
  - ```
    VARIABLE fp: CFILE:=fopen("pipe3", "r");            --2004 update
    fscanf(fp,"ALU_OUT = %s\n",v);
    ```

SNUG SAN JOSE 2002 — 2004 Update

Character Stream I/O

- C prototypes
  - `#include <stdio.h>`
  - `int   fputc(  int c, FILE *stream);`
  - `int   fgetc(  FILE *stream);`
- VHDL Example

```
process
  variable c: character;
  VARIABLE fin:  CFILE:=fopen("pipe2", "r");           --2004 update
  VARIABLE fout: CFILE:=fopen("pipe3", "w");           --2004 update
begin
  while not feof(fin) loop                             --2004 update
    c:=fgetc(fin);                                     --2004 update
    if isalpha(c) then fputc(tolower(c), fout);
    else fputc(c, fout); end if;
  end loop;
  fclose(fout); wait;
end process;
```

```
IMPURE FUNCTION fgetc(stream: IN CFILE) RETURN CHARACTER IS
    VARIABLE more:   BOOLEAN:=FALSE;  VARIABLE c: CHARACTER:=NUL;
  BEGIN
    ASSERT stream>0 AND stream<=streamNFILE
    REPORT "fgetc(): passed in bad CFILE stream id" SEVERITY FAILURE;
    IF stream>0 AND stream<=streamNFILE THEN
      IF    streamiob(stream).buf=NULL      THEN more:=TRUE;
      ELSIF streamiob(stream).buf'LENGTH<=0 THEN more:=TRUE; END IF;
      IF more AND streamiob(stream).fstat=OPEN_OK THEN
        more:=feof(stream);
        IF NOT more THEN
          CASE stream IS
          WHEN stdin => readline(input, streamiob(stream).buf);
          WHEN 4     => readline(streamfile4, streamiob(stream).buf);
          WHEN 5     => readline(streamfile5, streamiob(stream).buf);
          WHEN 6     => readline(streamfile6, streamiob(stream).buf);
          WHEN OTHERS =>
          END CASE;
          write(streamiob(stream).buf, LF);
        END IF;
      END IF;
      IF streamiob(stream).buf/=NULL THEN
        IF streamiob(stream).buf'LENGTH>0 THEN
          read(streamiob(stream).buf, c);
        END IF;
      END IF;
    END IF;
    RETURN c;
  END fgetc;
```

SNUG SAN JOSE 2002 — 2004 Update

Single File Stream Out: big endian

```
variable v07: std_logic_vector(0 to 7):="wwwwhhhh";        --little endian
variable v70: std_logic_vector(7 downto 0):="11110000";    --big endian
...
VARIABLE fout:  CFILE:=fopen("data_out.txt", "w");          --2004 update

    fprintf(fout,"%s\n", v07);        --hhhhwwww            --print big endian
    fprintf(fout,"%s\n",pf(v07));                           --alternative
   --warning actual output only occurs whenever a \n is encountered!
   --otherwise the output is held in the shared streamiob(fout).buf variable.

  --fprintf & printf will always output in big endian       --2004
    fprintf(fout, "v70=%s\n", v70);  --11110000             --print big endian

  --mixing between fprintf, fputc, fputs, write is allowed
    fputc('#', fout); fputs("end data;" & LF, fout); --"\n" is ignored

fclose(fout);
```

The 2004 version has been changed from little endian to big endian (see endian_h) and also avoids the following f1buf: By default, upto 8 files can be opened. Recompile stdio_h.vhd if more than 8 are needed.

The fclose(fout) can also be done explicitly fclose(fprintf_buffer, fout);

Example of multi-stream file (explicit buffer) which do not use any shared variables in the stdio_h package

```
variable f1buf, f2buf: line;
...
file_open(f1out, "xxx_out.txt", WRITE_MODE);
file_open(f2out, "xxx_out.txt", WRITE_MODE);

    fprintf(f1buf, f1out, "%s\n", v07); --little endian
    ...
    fprintf(f2buf, f2out, "%s\n", v70\n"); --big endian
fclose(f1buf, f1out);
fclose(f2buf, f2out);
```

Stream I/O

```
fin:=fopen("data_out.txt", "r");                          --2004 update

  --fscanf can read within a line or multiple text lines
    fscanf(fin, "%3s", s); --read 3 characters max

  --mixing fscanf, fgetc, ungetc, read allowed
  s(1):=fgetc(fin); s(2):=fgetc(fin); s(5):=NUL;       --2004 update

  --vectors can be read in as strings or as numbers
  --for formats: %x %o %d %u a std_logic 0 is HWXUL0
  --%u is unsigned & %d is signed on msb of std_logic_vector
  fscanf(fin, "%s", v07); fscanf(fin, "%x", v07);
  fscanf(fin, "%u", v07); fscanf(fin, "%d", v07);

fclose(fin);                                              --2004 update
```

New style is "**s(1):=fgetc(fin);**" which is much more cleaner code.
Compared to the old style was: "**fgetc(s(1), fin);**"

SNUG SAN JOSE 2002 2004 Update

Common String Functions

- VHDL prototypes

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

- Examples

```
– variable s, t : string ( 1 to 256 );
– strcpy(s, "hello world");
– strcpy(t, s(8 to 9));          -- array slice supported
– strcat(t, "12345");
– strcpy(t(30 to t'length), "xyzpdq");
– strcpy(t, 30, "xyzpdq");              -- simulating pointer arithmetic
```

Code example of string copy:

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

Additional Libraries

```
– LIBRARY   C;
– USE       C.CTYPE_H.ALL;
– function  isalpha(c: character)  return boolean;
– function  toupper(c: character)  return character;
– USE       C.STDLIB_H.ALL;
– function  atoi(s: string) return integer;
– USE       C.ENDIAN_H.ALL;
– function  to_bigendian_std_logic_vector(x: IN STD_LOGIC_VECTOR)
            return STD_LOGIC_VECTOR;
– function  to_littleendian_std_logic_vector(x: IN STD_LOGIC_VECTOR)
            return STD_LOGIC_VECTOR;
– USE       C.REGEXP_H.ALL;
– prodecure regmatch ( ai: OUT integer;  -- alternate match number
                       si: INOUT integer;-- next unmatched character
                       s : IN string;    -- input string
                       f : IN string;    -- PERL pattern matching
                       m1: OUT string;   -- () matching
                       m2: OUT string ); -- () matching
```

- Example: `regmatch(ai,fj,fmt,”%([ 0#+-]*)\.?([0-9]*).”,m1,m2);`

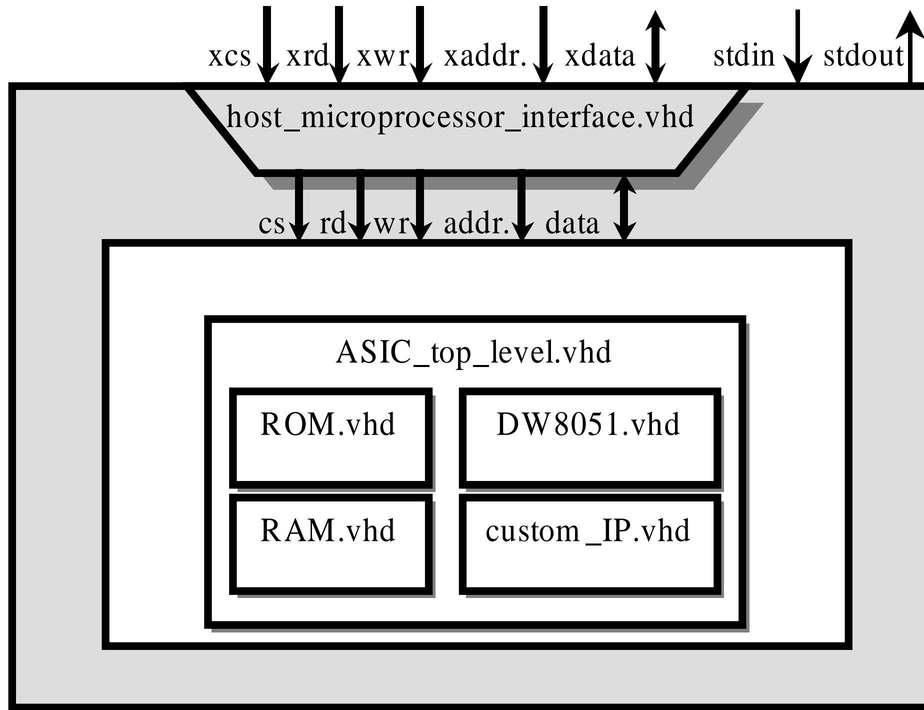
Note:In order to simplify coding, both fprintf & fscanf internally use regmatch to parse the format control string.. Additional library, “use endian_h” has been added for independent endian designs:

```
LIBRARY STD;
USE     STD.textio.all;            --write(buf, bit_vector);
LIBRARY ieee;
USE     ieee.std_logic_1164.all;   --define std_logic_vector;
USE     ieee.std_logic_textio.ALL; --write(buf, std_logic_vector);
LIBRARY C;
USE     C.stdio_h.all;
USE     C.endian_h.all;

ENTITY endian_h_test IS END;

ARCHITECTURE endian_h_test_arch OF endian_h_test IS BEGIN
  PROCESS
    VARIABLE v07: STD_LOGIC_VECTOR(0 TO 7)    :="0LWXUZH1";
    VARIABLE v70: STD_LOGIC_VECTOR(7 DOWNTO 0):="1UX-HWZ0";
  BEGIN
    printf(“--begin test;\n”); --write(buf, string'("--begin test;")); writeline(output, buf);

    printf("VARIABLE v07: STD_LOGIC_VECTOR(0 TO 7):=0LWXUZH1;\n");
    printf("v07=%s                        ==1HZUXWL0\n", v07);  --print big endian by default
    printf("to_littleendian(v07)=%s       ==0LWXUZH1\n”, to_littleendian_std_logic_vector(v07));
    printf("to_bigendian(v07)=%s          ==1HZUXWL0\n", to_bigendian_std_logic_vector(v07));
    ...
```

SNUG San Jose 2002 2004 Update

Applications of the C Functions

- Microprocessor Host Testbench



SNUG SAN JOSE 2002 — 2004 Update

Skeleton Testbench Code with passthru

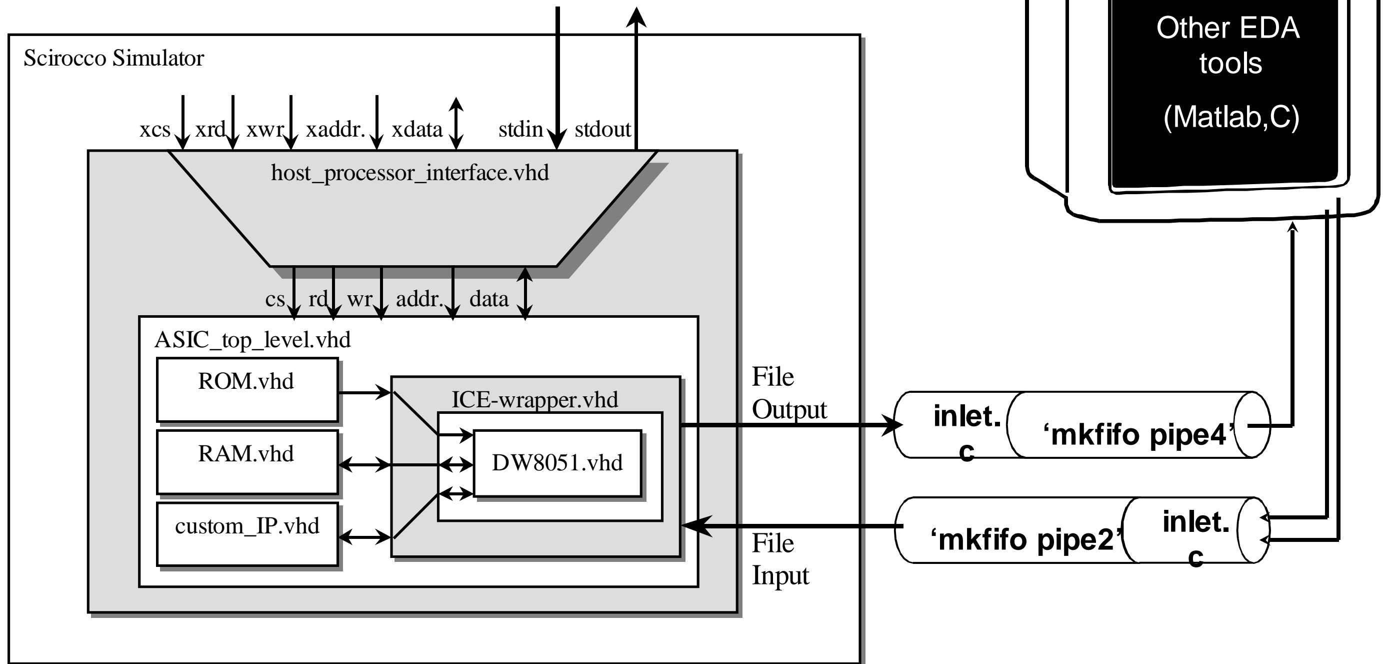
```
loop
    printf("Host => ");
    gets(what_next);
    if ( sscanf(what_next,"write %x %x") = 2 ) then
      sscanf(what_next,"write %x %x",address,data_out);
      wait for CS_START_DELAY;
      cs <= '1'; wait for WR_START_DELAY;
      wr <= '1'; wait for WRITE_WIDTH;
      wr <= '0'; wait for CS_END_DELAY;
      cs <= '0'; wait for WR_END_DELAY;
      address <= "xxxxxxxxxxxxxxxx"; data_out <= "ZZZZZZZZ";
    elsif ( sscanf(what_next,"read %x %x") = 2 ) then …
    else              -- passthru wrapper signals
      cs <= xcs;  wr <= xwr;  rd <= xrd;  address <= xaddress;
      data_out <= xdata_out;  xdata_in <= data_in;
    end if;
  end loop;
```

Also, files or pipes can be used:

```
loop
    fprintf(fout, "Host => ");
    fgets(what_next, what_next'length, fin);
    ...
```

Embedded Testbench using UNIX pipes…

- In Circuit Emulator Wrapper Testbench



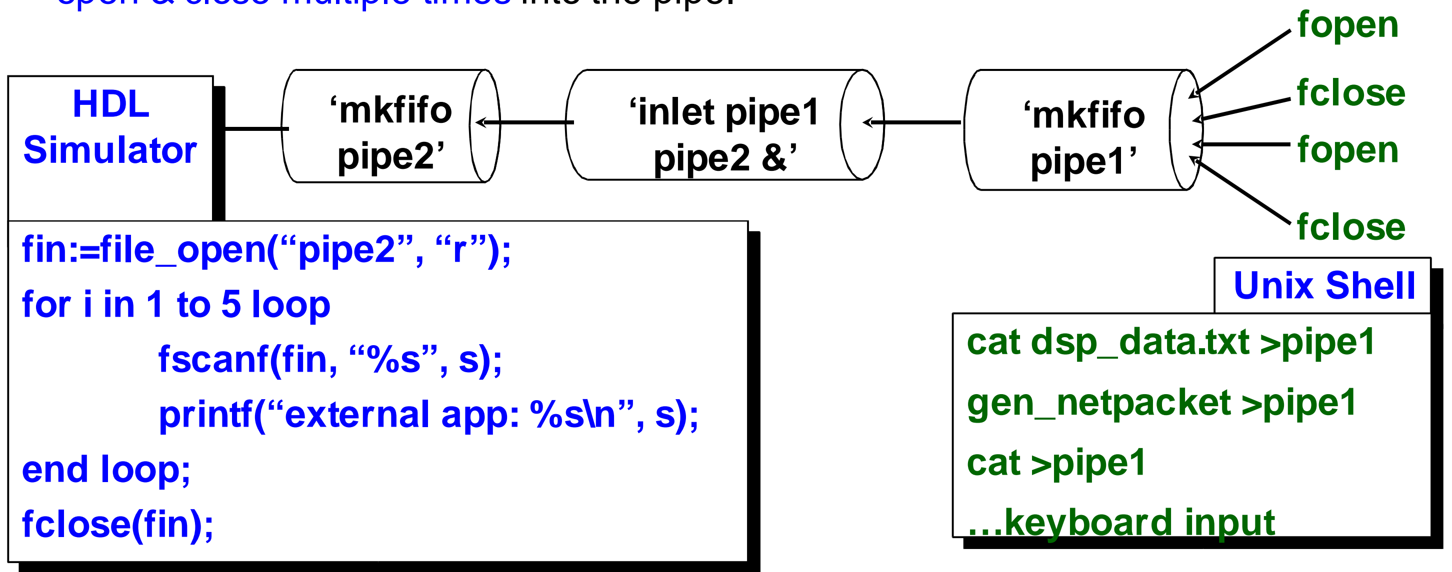
FIFO pipes: Have the nice property in that they can avoid disk space. Also, The producer generates only as much as the consumer needs. When the consumer cannot absorb the producer pipe, then the producer is I/O blocked until later.

Common example of pipes: `gzip -dc file.tar.gz | tar xvf -`

In this example, the output of gzip is stream directly into the tar program and no additional disk space (or disk access time) is use.

inlet pipe

- One difficulty using FIFO pipes is that once an external application issues a file close the pipe also closes and becomes broken.
- The program inlet always keeps the pipe open and let's one or more application(s) open & close multiple times into the pipe.



Use "ps -a" and "kill -9 <process id of inlet>" to terminate the inlet pipe.

The Unix "tee file1 file2 <pipe2 >pipe3" may useful to listen in or create additional pipe outputs.

Even though Network File System, NFS, uses remote procedure call (RPC), it does not support FIFO files (that I know of) across machines.

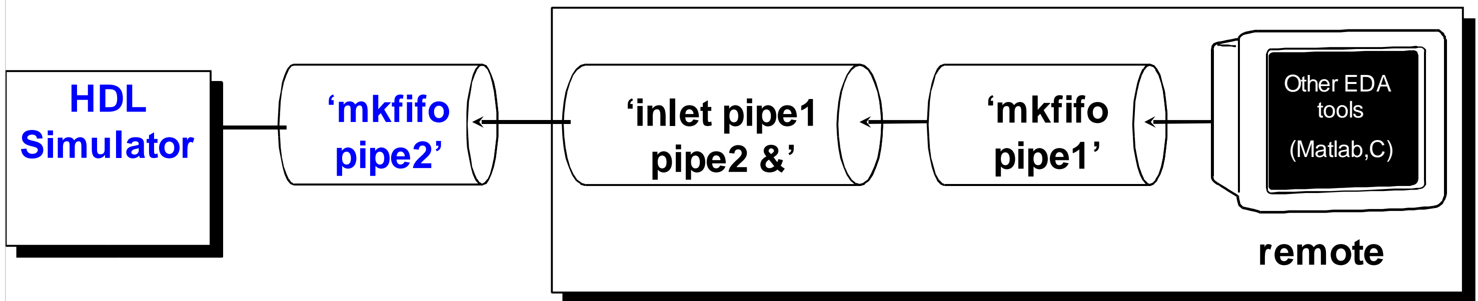
Skeleton inlet.c code

```
fout = open(argv[2], O_WRONLY, 0666);
for(;;) { /* re-open pipe */
       fin = open(argv[1], O_RDONLY, 0);

       while ((len = read(fin, buf, sizeof(buf))) > 0) {
              if (write(fout, buf, len) != len) {
                     perror("inlet: write"); return 1;
              }
       }

       /* len was <= 0; If len = 0, no more data is available.
       Otherwise, an error occurred. */
       if (len < 0) { perror("inlet: read"); return 1; }

       close(fin);
}
```

```
#include <stdio.h> /* compacted version */
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <strings.h>
int main(int argc, char *argv[ ] ) {  int fin, fout; char buf[1024]; int len;
   if (argc == 3) { if (!strcmp(argv[2], "-")) { fout=1; }
     else {  fout = open(argv[2], O_WRONLY, 0666);
       if (fout<0) {  fprintf(stderr, "inlet: open(%s, O_WRONLY, 0666) error\n", argv[2]); exit(1);  }
   }
   for(; ; ) {
     fin = open(argv[1], O_RDONLY, 0);
     if (fin<0) {  fprintf(stderr, "inlet: open(%s, O_RDONLY, 0) error\n", argv[1]); exit(1);  }
     while ((len = read(fin, buf, sizeof(buf))) > 0) {
        if (write(fout, buf, len) != len) {  perror("inlet: write");  return 1;   }
     }
     if (len < 0) { perror("inlet: read"); return 1; }  close(fin);
  } }
  else { fprintf(stderr, "inlet <pipe in> <pipe out>\n  If filename is dash then stdout\n");  }
  return 0;
}
```

rsh pipes

- Need to pipe across machines for simulation parallelism or legacy issues.
- (That legacy EDA tool that can only work on a certain OS, hostid, etc.)



- `rsh remote mkfifo pipe1`
- `mkfifo pipe2`
- `(rsh remote -n "cd /home/users/wolff; ./inlet pipe1 -" | cat >pipe2) &`
- `# run EDA tool which outputs to pipe1`
- `# run HDL simulator which inputs from pipe2`

We expand on the classic example:

```
tar cf - . | rsh remote "cd /directory; tar xf -"
```

rsh -n flag redirects input to /dev/null. This prevent stdin from interacting with the console shell terminal.

( ) : Creates a new shell
& : Ampersand starts the remote shell as a background process

Tips for rsh:
(1) Keep the remote ".cshrc" as simple as possible. It basically only needs a path. If necessary create a new user account. Some .cshrc output to stdout which create problems.

(2) Make sure "~/.rhosts", "/etc/hosts.equiv", "/etc/hosts.deny" and "/etc/hosts.allow" are set up correctly.

SAN JOSE 2002 2004 Update

Conclusions

- Most commonly standard C functions were written.
- These functions simplified the coding of test benches and wrappers with UNIX files and pipes.
- The web site for this library is located at…

http://bear.ces.cwru.edu/vhdl

Other interesting applications of intercommunications with simulators:

(1) "RT-level Fault Simulation Techniques based on Simulation Command Scripts," F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, DCIS2000: XV Conference on Design of Circuits and Integrated Systems, Le Corum, Montpellier, November 21-24, 2000, pp. 825-830.

"...we implemented two programs, the Fault List Generator and the Fault Simulator. The implementation consists of about 300 lines of C code for VHDL code analysis and Fault List creation, linked to the LEDA LPI interface and interacting with the ModelSim simulator, and of 700 lines of C code for the Fault Simulator, that is interfaced to the ModelSim simulator **through Unix pipes**."

(2) "An Application of Parallel Discrete Event Simulation Algorithms to Mixed Domain System Simulation," D. K. Reed, S. P. Levitan, J. Boles, J. A. Martinez, D. M. Chiarulli, University of Pittsburgh, DATE '04, Paris, France.

"...**By using shared memory IPC** (Inter-Process Communication) and PDES (Parallel Discrete Event Simulation) techniques, we achieve two orders of magnitude speedup over standard pipe/socket communication."