Files – theory & examples

by pwkolas / August 10, 2017 / Keywords

VHDL provides mechanism to work with files. This feature is useful, when there is a need to store some data like test vectors, parameters or results of the simulation. Way of working with files in VHDL is very similar to other languages. File is treated as an object. It can be created in an architecture body, process or subprograms. To properly work with files, following steps should be done:

1. Define type of a file
2. Declare object of the defined file type
3. Open the file
4. Perform write/read operations
5. Close the file

1. Define type of a file

type *file_type_name* **is file of** type;

This command specifies what kind of data will be stored in a file. *Type* can be any scalar type (vector, integer etc.) or composite type (record and array, but only one-dimensional). Others types are forbidden.

During defining the new file type, implicitly are created procedures and functions, which give access to the files: file_open (two versions), read, write, endfile, file_close.

```
procedure file_open (
   file file_ptr     : file_type_name;
   file_path         : in string;
   open_kind         : in file_open_kind := read_mode
);
procedure file_open (
   status            : out file_open_status;
```

```
   file file_ptr     : file_type_name;
   file_path         : in string;
   open_kind         : in file_open_kind := read_mode
);
procedure read (
   file file_ptr     : file_type_name;
   value             : out data_type
);
procedure write (
   file file_ptr     : file_type_name;
   value             : in data_type
);
procedure file_close (
   file file_ptr     : file_type_name;
);

function endfile (file file_ptr : file_type_name) return
boolean;
```

Explanation of some types used in procedures:

- parameter *file_open_status*

  It is worth to notice that two different functions *file_open* are created. The difference is in one output – *file_open_status*. This output informs if there were any problems during opening the file. Possible values:

  - open_ok – everything is OK
  - status_error – file is already open
  - name_error
    - in read mode – files does not exist
    - in write mode – files can not be created
  - mode_error – file can not be open in a specified mode

  Object of this type must be declared before it is used i.e.:

  variable *variable_name* : file_open_status;

  I encourage to use this version, because in case of any problems with opening the file, it is easier to find the reason of it.

- parameter *file_open_kind*

file_open_kind describes access to a file:

- read_mode
- write_mode
- append_mode

- function *endfile*

Function *endfile* is very useful, cause it checks End Of File character. It returns true if checked value is not a declared file type.

2. Declare object of the defined file type

At this point, we know what kind of data the file stores. Now it is time to declare the object (of a given file type):

```
file file_ptr : file_type_name;
```

That declaration says that file_ptr object points to the file with data_type data.

3. Open the file

Opening the file means connecting object declared in point 2. with the file existing in the system. It can be done in two ways:

- by extending declaration part:

```
file file_ptr : file_type_name open file_open_kind is file_path;
```

- by using *file_open* procedure described in point 1. *File_open* procedure can be used only in statement part of the architecture.

What is the difference? First version opens the file during declaration. It still uses *file_open* procedure but this is done implicitly.

Second, explicit method, uses procedure *file_open*. File is open when that procedure is executed. What is an advantage? Before you open a file, you can do some other tasks.

4. Perform write/read operations

To do any operations on the file *read/write* procedures are used.

5. Close the file

Although *file_close* procedure is used implicitly, when architecture or subprogram is finished, always try to close the file when all operations are done.

After theory, time to examples

Example 1: read/write integers to the files

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  use ieee.numeric_std.all;
4.
5.  entity ReadIntFromFile is
6.  end ReadIntFromFile;
7.
8.  architecture ReadIntFromFile_rtl of ReadIntFromFile is
9.
10.    constant C_FILE_NAME_RD :string  := "./dat/ReadIntFromFileIn.dat";
11.    constant C_FILE_NAME_WR :string  :=
"./dat/ReadIntFromFileOut.dat";
12.    constant C_CLK          :time := 10 ns;
13.
14.    signal clk              :std_logic := '0';
15.    signal rst              :std_logic := '0';
16.    signal data             :integer := 1;
17.    signal eof              :std_logic := '0';
18.
19.    type INTEGER_FILE is file of integer;
20.    file fptrrd             :INTEGER_FILE;
21.    file fptrwr             :INTEGER_FILE;
22.
23. begin
24.
25. ClockGenerator: process
26. begin
27.    clk <= '0' after C_CLK, '1' after 2*C_CLK;
28.    wait for 2*C_CLK;
29. end process;
30.
31. rst <= '1', '0' after 100 ns;
32.
33. GetData_proc: process
34.
```

```
35.     variable statrd : FILE_OPEN_STATUS;
36.     variable statwr : FILE_OPEN_STATUS;
37.
38.     variable varint_data    :integer := 1;
39.
40.  begin
41.
42.     data      <= 1;
43.     eof       <= '0';
44.
45.     wait until rst = '0';
46.
47.     file_open(statrd, fptrrd, C_FILE_NAME_RD, read_mode);
48.     file_open(statwr, fptrwr, C_FILE_NAME_WR, write_mode);
49.
50.     while (not endfile(fptrrd)) loop
51.        wait until clk = '1';
52.        read(fptrrd, varint_data);
53.        write(fptrwr, varint_data);
54.        data  <= varint_data;
55.     end loop;
56.     wait until rising_edge(clk);
57.     eof       <= '1';
58.     file_close(fptrrd);
59.     file_close(fptrwr);
60.     wait;
61.  end process;
62.
63.  end ReadIntFromFile_rtl;
```

Example 2: read/write vectors to the files

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  use ieee.numeric_std.all;
4.
5.  entity ReadVecFromFile is
6.  end ReadVecFromFile;
7.
8.  architecture ReadVecFromFile_rtl of ReadVecFromFile is
9.
10.    constant C_FILE_NAME_RD :string  := "./dat/ReadVecFromFileIn.dat";
11.    constant C_FILE_NAME_WR :string  :=
    "./dat/ReadVecFromFileOut.dat";
```

```
12.     constant C_CLK          :time := 10 ns;
13.
14.     signal clk              :std_logic := '0';
15.     signal rst              :std_logic := '0';
16.     signal data             :std_logic_vector(4-1 downto 0);
17.     signal eof              :std_logic := '0';
18.
19.     type STD_FILE is file of std_logic_vector(4-1 downto 0);
20.     file fptrrd             :STD_FILE;
21.     file fptrwr             :STD_FILE;
22.
23.  begin
24.
25.  ClockGenerator: process
26.  begin
27.     clk <= '0' after C_CLK, '1' after 2*C_CLK;
28.     wait for 2*C_CLK;
29.  end process;
30.
31.  rst <= '1', '0' after 100 ns;
32.
33.  GetData_proc: process
34.
35.     variable statrd         :FILE_OPEN_STATUS;
36.     variable statwr         :FILE_OPEN_STATUS;
37.
38.     variable varstd_data    :std_logic_vector(4-1 downto 0);
39.
40.  begin
41.
42.     data      <= (others => '0');
43.     eof       <= '0';
44.
45.     wait until rst = '0';
46.
47.     file_open(statrd, fptrrd, C_FILE_NAME_RD, read_mode);
48.     file_open(statwr, fptrwr, C_FILE_NAME_WR, write_mode);
49.
50.     while (not endfile(fptrrd)) loop
51.        wait until clk = '1';
52.        read(fptrrd, varstd_data);
53.        write(fptrwr, varstd_data);
54.        data <= varstd_data;
55.     end loop;
```

```
56.     wait until rising_edge(clk);
57.     eof       <= '1';
58.     file_close(fptrrd);
59.     file_close(fptrwr);
60.     wait;
61.  end process;
62.
63.  end ReadVecFromFile_rtl;
```

Example 3: read/write integers to the files, but with use extended declaration part instead of *file_open* procedure

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  use ieee.numeric_std.all;
4.
5.  entity ReadIntFromFileImpOpen is
6.  end ReadIntFromFileImpOpen;
7.
8.  architecture ReadIntFromFileImpOpen_rtl of ReadIntFromFileImpOpen is
9.
10.     constant C_FILE_NAME_RD :string  := "./dat/ReadIntFromFileIn.dat";
11.     constant C_FILE_NAME_WR :string  :=
    "./dat/ReadIntFromFileOut.dat";
12.     constant C_CLK          :time := 10 ns;
13.
14.     signal clk              :std_logic := '0';
15.     signal rst              :std_logic := '0';
16.     signal data             :integer := 1;
17.     signal eof              :std_logic := '0';
18.
19.     type INTEGER_FILE is file of integer;
20.     file fptrrd             :INTEGER_FILE open READ_MODE is
    C_FILE_NAME_RD;
21.     file fptrwr             :INTEGER_FILE open WRITE_MODE is
    C_FILE_NAME_WR;
22.
23.  begin
24.
25.  ClockGenerator: process
26.  begin
27.     clk <= '0' after C_CLK, '1' after 2*C_CLK;
28.     wait for 2*C_CLK;
29.  end process;
```

```
30.
31.  rst <= '1', '0' after 100 ns;
32.
33.  GetData_proc: process
34.
35.     variable varint_data    :integer := 1;
36.
37.  begin
38.
39.     data      <= 1;
40.     eof       <= '0';
41.
42.     wait until rst = '0';
43.
44.     while (not endfile(fptrrd)) loop
45.        wait until clk = '1';
46.        read(fptrrd, varint_data);
47.        write(fptrwr, varint_data);
48.        data  <= varint_data;
49.     end loop;
50.     wait until rising_edge(clk);
51.     eof       <= '1';
52.     file_close(fptrrd);
53.     file_close(fptrwr);
54.     wait;
55.  end process;
56.
57.  end ReadIntFromFileImpOpen_rtl;
```

***** When you simulate given examples, you will see that codes do not behave as you would expect. In next post I am going to explain why.

***** I described theory of working with files with VHDL. Those are basics, but it does not mean that it is the simplest method to work with files. It just describes how it is done. In next posts I am going to describe more convenient and simpler ways to work with files.

*** *** ***

All source codes used in that post you can find on gitlab.com.

*** *** ***

Tags: Files Read Write

PREVIOUS
Developing design: moving average filter. Part 5 – model of the filter.

NEXT
Files – data representation mismatch

2 thoughts on “Files – theory & examples”

SINDHUJA
September 26, 2019 at 12:07 pm

how to read from a file and write the output to the same file? suppose input is 111 and output is 0 so finally the input file must have 1110 in it?

Reply

PWKOLAS
October 12, 2019 at 1:04 am

VHDL LRM in “File operations” chapter, defines three access modes to file objects: READ_MODE, WRITE_MODE, APPEND_MODE (for details please refer to LRM).
None of them do what you want.
Consider, if you need to write output data to the same file… I always prefer keeping results in a separated file.
However if you really need to do that, I would suggest:
1. write the output data to another file,
2. remove the input file,
3. rename output file to original input file name (from a system level, e.g. by script).
I guess, this is the simplest, fastest and optimised way.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Tags

Altera Architecture Array ASCII Char Component Configuration Counter D Flip-Flop Entity Files For Loop Function Generic HEX HighAttribute If Statement Length Log2 Matlab Modelsim Moving Average Filter Package Ports Procedure Process Read Registers Resize Simulation Square root Synchronous logic Testbench TextIO Unconstrained Wait Write Xilinx

Recent Posts

Square Root. Part 3 – summary.

Square Root. Part 2 – pipelined version.

Square Root. Part 1 – iterative version.

Array – basics

Attribute LENGTH

Archives

April 2020

February 2020

October 2019

May 2018

November 2017

October 2017

September 2017

August 2017

July 2017

May 2017

April 2017

March 2017

February 2017

September 2016

August 2016

July 2016

June 2016