Session #10

FILE I/O Operations

- These functions operate on files.
- The following are the functions available along with others described in upcoming slides:

```
$fopen (file_name) ;
$fclose (file_name) ;
$fdisplay (arguments) ;
$fwrite (arguments) ;
$fstrobe (arguments) ;
$fmonitor (arguments) ;
$readmemb ("file", memory_identifier) ;
$readmemh ("file", memory_identifier) ;
```

Opening and closing files

- The ***$fopen*** function opens a file and returns a multi-channel descriptor in the format of an unsized integer.
- This is unique for each file.
- All communications between the simulator and the file take place through the file descriptor.
- Users can specify only the name of a file as an argument, this will create a file in the default folder or a folder given in the full path description

- To close an opened file use the ***$fclose*** function.
- This function is called without any arguments, it simply closes all opened files. If an argument is specified it will close only a file in which the descriptor is given.
- By default, before the simulator terminates, all open files are closed. This means that the user does not have to close any files, and closing is done automatically by the simulator.

```
fh = $fopen (file_name) ;
$fclose (fh) ;
```

Example :

Opens file called "results.dat"

Writes some data to that file

Finally closes the file.

```
integer file ;
reg a, b, c;
initial
begin
  file = $fopen("results.dat") ;
  a = b & c ;
  $fdisplay(file, "Result is: %b", a) ;
  $fclose(file) ;
end
```

File operations

```
integer file;
file = $fopenr("filename");
file = $fopenw("filename");
file = $fopena("filename");
```

Opens an existing file for reading

Opens new file file for writing

Opens an existing file for appending

```
integer file, r;
r = $fcloser(file);
r = $fclosew(file);
```

Closes a file for input

Closes a file for output

```
integer file;
reg eof;
eof = $feof(file);
```

Checks for end of file

File operations

```
integer file;
reg error;
error = $ferror(file);
```

Return the status of file, while reading, 0 on no error

```
integer file, r;
file = $fopenw("output.log");
r = $fflush(file);
```

Causes any buffered data waiting to be written for the named stream to be written to that file

Writing data into files

- The system tasks ***$fdisplay***, ***$fmonitor*** are used to write into opened files.

```
$fdisplay (fh, arguments) ;
$fmonitor (fh, arguments) ;
```

```
integer file_handler;

initial
begin
        file_handler = $fopen("file.dat");
        $fdisplay( file_handler, "RV-VLSI" );
        $fclose( file_handler );
end
```

Display string in the file and also on the standard output.

```
initial
begin
        file_handler = $fopen("file.dat");
        $fmonitor( file_handler, "A = %d B = %d", a,b );
end
```

Continuously writes the values into the file when they change

Reading data from file

- To read data from a file and store it in memory, use the functions: ***$readmemb*** and ***$readmemh***.
- The ***$readmemb*** task reads binary data and ***$readmemh*** reads hexadecimal data. Data has to exist in a text file.
- White space is allowed to improve readability, as well as comments in both single line and block.
- The numbers have to be stored as binary or hexadecimal values. The basic form of a memory file contains numbers separated by new line characters that will be loaded into the memory.

```
$readmemb ("file", memory_identifier, start_addr, finish_addr) ;
$readmemh ("file", memory_identifier, start_addr, finish_addr) ;
```

Examples:

Reads data from file *data.bin* to memory variable.

```
reg [3:0] memory [15:0] ;
initial begin
  $readmemb("data.bin", memory) ;
end
```

Loading data in hexadecimal format from file *data.hex* into memory starting at address 4 and down to address 2.

```
reg [3:0] memory [15:0] ;
initial
$readmemh("data.hex", memory, 4, 2) ;
```

New file operations in Verilog 2001

Verilog 2001 FileIO supports following ways of reading a file.

- Reading a character at a time with $***fgetc***.
- Reading a line at a time with $***fgets***.
- Reading formatted data with $***fscanf***. The $***fscanf*** function reads characters from the file specified by the file descriptor, interprets them according to a format, and stores the results in its arguments.
- Reading binary data with $***fread***. The $***fread*** function reads binary data from the file specified by the file descriptor into a register or into a memory

Example usage:

```
while ( ! $feof(in)) begin
        @ (negedge clk);
           enable = 1;
           statusI = $fscanf(in,"%h %h\n",din[31:16],din[15:0]);
        @ (negedge clk);
           enable = 0;
end
```

File Operations

```
integer file, char;
char = $fgetc(file);
char = $getc();
```

Reads a character from the file

Reads a character from the standard input

```
integer file;
reg [7:0] char, r;
r = $ungetc(char, file);
```

Pushes the character back into the file stream. That character will be the next read by $**fgetc**

```
integer stream, r, char;
r = $fputc(stream, char);
```

Writes a single character to the specified file. It returns EOF if there was an error, 0 otherwise.

File Operations

```
integer file, n, r;
reg [n*8-1:0] string;
r = $fgets(string, n, file);
```

Reads a string from the file. Characters are read from the file into string until a newline is seen, end-of-file is reached, or n-1 characters have been read.

```
integer file, count;
count = $fscanf(file, format, args);
count = $scanf(format, args);
```

The function $**fscanf** parses formatted text from the file according to the format and writes the results to args. $**sscanf** parses formatted text from a string. $**scanf** parses formated text from stdin.

File Operations

Note on formats

The format can be either a string constant or a reg. It can contain:
Whitespace characters such as space, tab (\t), or newline (\n). One or more whitespace characters are treated as a single character, and can match zero or more whitespace characters from the input.

Conversion specifications which start with a %. Next is an optional *, which suppresses assignment. Then is an optional field width in decimal. Lastly is the operator character as follows:

b -- Binary values 0, 1, X, x, Z, z, _
d -- Decimal values 0-9, _, no X, x, Z, or z. Note that negative numbers are NOT supported because of a Verilog language limitation.
o -- Octal values 0-7, _, X, x, Z, z
h or x -- Hexadecimal values, 0-9, A-F, a-f, _, X, x, Z, z
c -- A single character
f -- A floating point number, no _, X, x, Z, or z
s -- A string
% -- The percent character

File Operations

```
integer file, position;
position = $ftell(file);
```

Returns the position in the file for use by $**fseek**. If there is an error, it returns a -1.

```
`define SEEK_SET 0
`define SEEK_CUR 1
`define SEEK_END 2
integer file, offset, position, r;
r = $fseek(file, 0, `SEEK_SET);
r = $fseek(file, 0, `SEEK_CUR);
r = $fseek(file, 0, `SEEK_END);
r = $fseek(file, position, `SEEK_SET);
```

$**fseek** allows random access in a file. You can position at the beginning or end of a file, or use the position returned from $**ftell**.

Beginning

No effect

EOF

Previous Location

File Operations

```
integer file, r, a, b;
reg [80*8:1] string;
file = $fopenw("output.log");
r = $sformat(string, "Formatted %d %x", a, b);
r = $sprintf(string, "Formatted %d %x", a, b);
r = $fprintf(file, "Formatted %d %x", a, b);
```

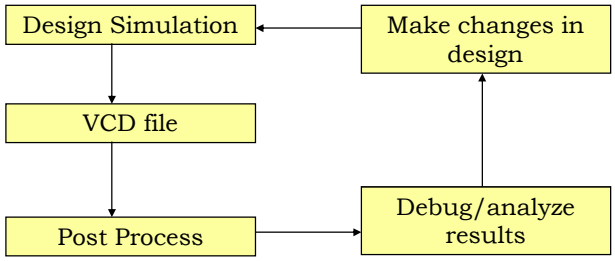
The functions $**sformat** and $**sprintf** writes a formatted string into a Verilog register. The two functions are identical. $**fprintf** writes a formatted string to a file.

Using file operations

- File I/O system functions can be used to do the following:
  - read stimulus files to apply patterns to the inputs of a model
  - read a file of expected values for comparison with your model
  - read a script of commands to drive a simulation
  - read either ASCII or binary files into Verilog registers and memories
  - have hundreds of log files open simultaneously (though they are written to one at a time)

Value change dump file

- The value change dump (VCD) file is an ASCII file that contains information about simulation time, scope, signal definitions and signal value changes in the simulation run.
- All signals or a selected set of signals in a design can be written to a VCD file during simulation.
- Post processing tools can take the VCD file as an input and visually display hierarchical information, signal values and signal waveforms.



Value change dump file

$dumpfile(filename)
This task is used to specify the VCD file name. The filename parameter is optional. If it is not given, then the file will be named "Verilog.dump".

$dumpvars(level, list_of_variables_or_modules)
This task is used to specify which variables should be dumped. Both parameters are optional and if none are used, then all variables will be dumped.

If level = 0, then all variables within the modules from the list will be dumped. If any module from the list contains module instances, then all variables from these modules will also be dumped.

If level = 1, then only listed variables and variables of listed modules will be dumped.

$dumpoff
This task stops the dumping of variables. All variables are dumped with x value and all next changes of variables will not be dumped.

Value change dump file

$dumpon
This task starts previously stopped dumping of variables.

$dumpall
When this task is used, then the current value of all dumped variables will be written to file.

$dumplimit(filesize)
This task can set the maximum size of the VCD file.

$dumpflush
This task can be used to make sure that all changes of dumped variables are written to file.

Value change dump file

Examples:

```
module top;
reg a, b;
wire y;
assign y = a & b;
always begin
  a = 1'b0;
  #10;
  a = 1'b1;
  #10;
  a = 1'bx;
  #10;
end
always begin
  b = 1'b0;
  #30;
  b = 1'b1;
  #30;
  b = 1'bx;
  #30;
end
```

```
// continued


initial begin
  $dumpfile("test.txt");
  $dumpvars(1,a,y);
  #200;
  $dumpoff;
  #200;
  $dumpon;
  #20;
  $dumpall;
  #10;
  $dumpflush;
end
endmodule
```

The dumpfile will contain only changes of 'a' and 'y' variables. After 200 time units, dumping will be suspended for 200 time units. Next, dumping will start again and after 20 time units, all variables will be dumped.

Exercise 1

- Write a file I/O module which can read numbers from a text file maximum of 32.
- Design and implement the sorting algorithm in Verilog for these numbers, it can be ascending or descending
- Write the results in to a different file.

180 Minutes