CS 207 Digital Logic - Spring 2019
Lab 5 - Task and Function

Monday, Mar. 18, 2019

1 Experiment A

1.1 `Task`

In the second lab session, we came across a task in Verilog. Tasks are used in all programming languages, generally known as procedures or subroutines. The lines of code are enclosed in `task-endtask` brackets. Data is passed to the task, the processing done, and the result returned. They have to be specifically called, with data ins and outs. Included in the main body of code, they can be called many times, reducing code repetition.

- Tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive `` `include `` to include the task in the file which instantiates the task.
- Tasks can include timing delays, like `posedge`, `negedge`, `# delay` and `wait`.
- Tasks can have any number of inputs and outputs.
- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- Tasks can take, drive and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of task execution.
- Tasks can call another task or function.
- A task must be specifically called with a statement, it cannot be used within an expression as a function call.

simple_task.v

```
module simple_task();

task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
  temp_out = (9/5) *( temp_in + 32)
end
endtask

endmodule
```

task_calling.v

```
module  task_calling (temp_a, temp_b, temp_c, temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
reg [7:0] temp_b, temp_d;
`include "mytask.v"

always @ (temp_a)
begin
  convert (temp_a, temp_b);
end

always @ (temp_c)
begin
  convert (temp_c, temp_d);
end

endmodule
```

1.2 Function

A Verilog function is the same as a task, with very little differences, like function cannot drive more than one output, can not contain delays.

- Functions are defined in the module in which they are used. It is possible to define functions in separate files and use compile directive `` `include `` to include the function in the file which instantiates the task.
- Functions can not include timing delays, which means that functions should be executed in "zero" time delay.
- Functions can have any number of inputs but only one output.
- The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.

- Functions can take, drive, and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution.
- Functions can call other functions, but can not call tasks.

simple_function.v

```
module simple_function();

function  myfunction;
input a, b, c, d;
begin
  myfunction = ((a+b) + (c-d));
end
endfunction

endmodule
```

function_calling.v

```
module  function_calling(a, b, c, d, e, f);

input a, b, c, d, e ;
output f;
wire f;
`include "myfunction.v"

assign f =  (myfunction (a,b,c,d)) ? e :0;

endmodule
```

2 Experiment B

2.1 Have a Try

Write a task to implement a half adder using gate-level design. Then write another task to implement a one-bit full adder using gate-level design and the previous task. Finally write a module implementing a 32-bit full adder based on the two tasks. The module has three inputs: `a`, `b`, and `cin`. Output `sum` and `cout`. Write testbenches to assess its correctness.

Assignment
Save the source code and testbenches in `multibit.v`. Assignment 2 requires the source file.

3 Experiment C

3.1 Have a Try

Design and verify a parity checker.

Assignment
Save the source code and testbenches in `parity.v`. Assignment 2 requires the source file.