שפת תכנון חומרה
ורילוג - Verilog

Verilog – Tasks & Functions

Dr. Avihai Aharon

Objectives

- Describe the differences between tasks and functions
- Identify the conditions required for tasks to be defined.
- Understand task declaration and invocation
- Explain the conditions necessary for functions to be defined
- Understand function declaration and invocation

Tasks & Functions

- A designer is frequently required to implement the same functionality at many places in a behavioral design
  - ✓ The commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code
- Verilog provide tasks and functions to break up large behavioral design into smaller pieces
  - ✓ Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design
  - ✓ Tasks have input, output and inout arguments
  - ✓ Functions have input arguments
  - ✓ Values can be passed into and out from tasks and functions
- Tasks and functions are included in the design hierarchy
  - ✓ Like named blocks, tasks and functions can be addressed by means of hierarchical names

Tasks & Functions

- Both tasks and functions must be defined in a module and are local to the module
- Tasks or functions cannot have wires
- Tasks and functions contain behavioral statements only
- Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions
- Be careful:
  - ✓ About making references in a task of function to variables declared in the parent module. If you want to be able to call a task or function from other modules, all variables used inside the task or function should be in its port list

Task: Key Features

- **Key features:**
  - ✓ **Task is typically used to perform debugging operations, or to describe a separate piece of hardware.**
  - ✓ **Task is enabled when the task name is encountered in the Verilog description**
  - ✓ **Task definition is contained in the module definition**
  - ✓ **The arguments passed to the task are in the same order as the task I/O declaration. You can use timing control in a task**
  - ✓ **Tasks define a new scope in Verilog**
  - ✓ **Task can enable function**
  - ✓ **Tasks can be disabled (as a named blocks: disable <task_name>)**
- **Be careful:**
  - ✓ **About calling the task from more than one section of code. Because a task maintains only one copy of its local variables, calling it twice concurrently may lead to incorrect results. This happens most often if you have used timing controls in a task**

Task: Declaration & Invocation

- **Tasks are declared with the keywords task and endtask**

```
// Task Declaration/Disable Syntax
<task>
        ::= task <name_of_task_identifier>;
            <parameters/inputs/outputs/inouts_declaration>
            <reg/time/integer/real/event_declaration>
            <statements_or_null>
            disable <name_of_task_identifier>;          // optional
            endtask

// Task Invocation Syntax
<task_enable>
        ::= <name_of_task_identifier>;
        ||= <name_of_task_identifier>(<expression><,<expression>>*);
```

Function: Key Features

- **Key features:**
  - ✓ **Function is typically used to create a new operation, or to represent combinational logic**
  - ✓ **A function definition cannot contain any timing-control statements**
  - ✓ **It must contain at least one input and cannot contain any output or inout port**
  - ✓ **A function returns only one value, which is the value of the function itself**
  - ✓ **The arguments passed to the function are in the same order as the function input parameter declarations**
  - ✓ **A function cannot enable a task**
  - ✓ **Functions define a new scope in Verilog**

Function: Declaration & Invocation

- **Functions are declared with the keywords function and endfunction**

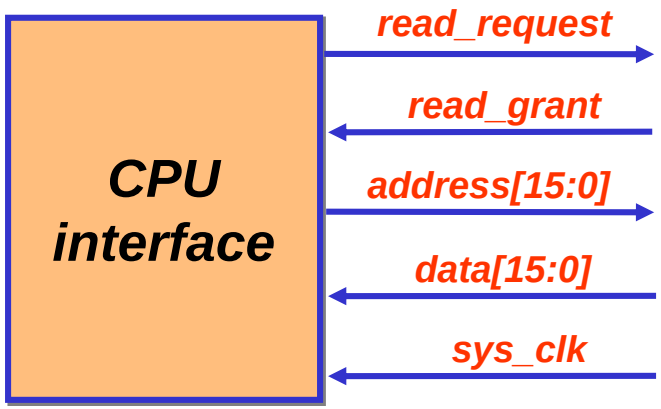
```
// Function Declaration Syntax
<function>
        ::= function <range/integer/real_type>? <name_of_function_identifier>;
            <parameters/inputs_declaration>
            <reg/time/integer/real_declaration>
            <statement>
            endfunction

// Function Invocation Syntax
<function_call>
        ::= <name_of_function_identifier>(<expression><,<expression>>*);
```

Differences Between Tasks & Functions

Functions	Tasks
A function can enable another function but not another task	*A task can enable other task and function*
Functions always execute in 0 simulation time	*Tasks may execute in non-zero simulation time*
Functions must not contain any delay, event, or timing control statement	*Tasks may contain any delay, event, or timing control statement*
Functions must have at least one input argument. They can have more than one input.	*Tasks may have zero or more arguments of type input, output, or inout*
Functions always return a single value. They cannot have output or inout arguments	*Tasks do not return with a value but can pass multiple values through output and inout arguments*

Tasks & Functions Example

- The CPU assert read_request and waits for read_grant
- When is asserted, the CPU places the address on the address bus and reads the data
- After reading the data, CPU cleans up by de-asserting read_request and driving the address bus to high impedance
- The CPU swaps the bits of the data
- If read_grant is de-asserted before read_request is, abort the read




```
module cpu_ iface(/* ports*/);
// IO declarations
reg [15: 0] IR, PC, address;
always @( posedge sys_ clk)
begin
        if (read_ request == 1)
        // Call the read task
        // Call function to swap bits
        // signal read complete
end
// Clean up after read
// Abort read
endmodule
```

Task Example

Task invocation

Task definition

```
module cpu_iface (/* ports* /);
// IO declaration goes here
reg [15: 0] IR, PC, address;
wire[15:0] data_out;
. . .
always @( sys_ clk) begin
   if (read_ request == 1)
        begin
        read_ mem( IR, PC);
        // Event and Function calls will go here
        end
end . . .
task read_ mem;
output [15: 0] data_ out;
input [15: 0] addr_ in;
@( posedge read_ grant)
        begin
        address = addr_ in;
        #15 data_ out = data;
        end
endtask
// Function definition will go here
endmodule
```

Function Example

Function invocation

Function definition

```
module cpu_iface (/* ports */);
// IO declaration goes here
reg [15: 0] IR, PC, address;
. . .
always @( sys_ clk) begin
        if (read_ request == 1)
        begin
        read_ mem( IR, PC);
        IR = swap_ bits( IR);
        // Signal read complete
        end
end . . .
// Task definition goes here
function [15: 1] swap_ bits;
        input [15: 1] in_ vec;
        integer i;
        for (i = 15; i >= 0; i = i - 1)
        swap_ bits[15- i] = in_ vec[i];
endfunction

endmodule
```

Tasks & Functions Summary

- **Tasks and functions are used to define common Verilog functionality that is used at many places in the design. Tasks and functions help to make a module definition more readable by breaking it up into manageable subunits**
- **Tasks can take any number of input, inout, or output arguments. Delay, event, or timing control constructs are permitted in tasks. Tasks can enable other tasks or functions**
- **Functions are used when exactly one return value is required and a least one input argument is specified. Delay, event, or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks**
- **A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register**
- **Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing**