Chapter 5: Tasks, Functions, and UDPs

Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- Describe the features of tasks and functions
- Describe how to use tasks and functions
- Describe the features of dynamic tasks and functions
- Describe the features of UDPs (user-defined primitives)
- Describe how to use combinational and sequential UDPs

Tasks and Functions (p. 145 in LRM)

- Tasks and functions provide the ability to reuse the common piece of code from several different places in a design.
- Comparison of tasks and functions

Item	Tasks	Functions
Arguments	May have zero or more *input*, *output*, and *inout*.	At least one *input*, and cannot have *output* and *inout*.
Return value	May have multiple values via *output* and *inout*.	Only a single value via function name.
Timing control statements	Yes	No
Execution	In non-zero simulation time.	In 0 simulation time.
Invoke functions/tasks	Functions and tasks.	Functions only.

When to Use Tasks

- Tasks are declared with the keywords task and endtask.
- When to use tasks if the procedure
  - has delay or timing control constructs.
  - has at least one output arguments.
  - has no input argument.

Task Definition and Calls

```
task [automatic] task_identifier(task_port_list); …  endtask
```

```
// port list style
task [automatic] task_identifier;
[declarations] // include arguments
procedural_statement
endtask

// port list declaration style
task [automatic] task_identifier ([argument_declarations]);
[other_declarations]  // exclude arguments
procedural_statement
endtask
```

Types of Tasks

- Types of tasks
  - (static) task: declared with task … endtask.
  - automatic (reentrant, dynamic) task: declared with task automatic … endtask.
- Features of static tasks
  - All declared items are statically allocated.
  - Static tasks items can be shared across all uses of the task executing concurrently within a module.
- Features of automatic (reentrant, dynamic) tasks
  - All declared items are dynamically allocated for each invocation.
  - They are deallocated at the end of the task invocation

A Task Example

```
// an example illustrating how to count the zeros in a byte.
module zero_count_task (data, out);
input   [7:0] data;
output reg [3:0] out;  // output declared as register
always @(data)
    count_0s_in_byte(data, out);

// task declaration from here.
task count_0s_in_byte(input [7:0] data, output reg [3:0] count);
integer i;
begin   // task body
   count = 0;
for (i = 0; i <= 7; i = i + 1)
     if (data[i] == 0) count= count + 1;
end endtask
endmodule
```

A Dynamic Task Example

```
// task definition starts from here
task automatic check_counter;
reg [3:0] count;
// the body of the task
begin
    $display ($realtime,,"At the beginning of task, count = %d", count);
    if (reset) begin
        count = 0;
        $display ($realtime,,"After reset, count = %d", count);
end
endmodule
```

When to Use Functions

- Functions are declared with the keywords function and endfunction.
- When to use functions if the procedure
  - has no delay or timing control constructs (any statement introduced with #, @, or wait).
  - returns a single value.
  - has at least one input argument.
  - has no output or inout arguments.
  - has no nonblocking assignments.

Function Definition and Calls

function [automatic] [signed] [range_of_type] … endfunction

```
// port list style
function [automatic] [signed] [range_or_type] function_identifier;
input_declaration
other_declarations
procedural_statement
endfunction

// port list declaration style
function [automatic] [signed] [range_or_type] function_identifier (input_declarations);
other_declarations
procedural_statement
endfunction
```

Types of Functions

- Features of (static) functions
  - All declared items are statically allocated.

- Features of automatic (recursive, dynamic) functions
  - All function items are allocated dynamically for each recursive call.
  - Automatic function items cannot be accessed by hierarchical references.
  - Automatic functions can be invoked through use of their hierarchical name.

A Function Example

```
// an example illustrating how to count the zeros in a byte.
module zero_count_function (data, out);
input  [7:0] data;
output reg [3:0] out;                          // output declared as register
always @(data)
    out = count_0s_in_byte(data);
                                               // function declaration from here.
function [3:0] count_0s_in_byte(input [7:0] data);
integer i;
begin
   count_0s_in_byte = 0;
   for (i = 0; i <= 7; i = i + 1)
        // the following statement can be replaced by:
        // count_0s_in_byte = count_0s_in_byte + ~data[i]. Why?
         if  (data[i] == 0) count_0s_in_byte = count_0s_in_byte + 1;
end
endfunction
endmodule
```

Automatic (Recursive) Functions

```
// to illustrate the use of reentrant function
module factorial(input  [7:0] n, output [15:0] result);
// instantiate the fact function
    assign result = fact(7);
                                                        // define fact function
   function automatic [15:0]  fact;
   input [7:0] N;

   // the body of function
       if (N == 1) fact = 1;
       else fact = N * fact(N - 1);
   endfunction
endmodule
```

Constant Functions

Constant function calls are evaluated at elaboration time.

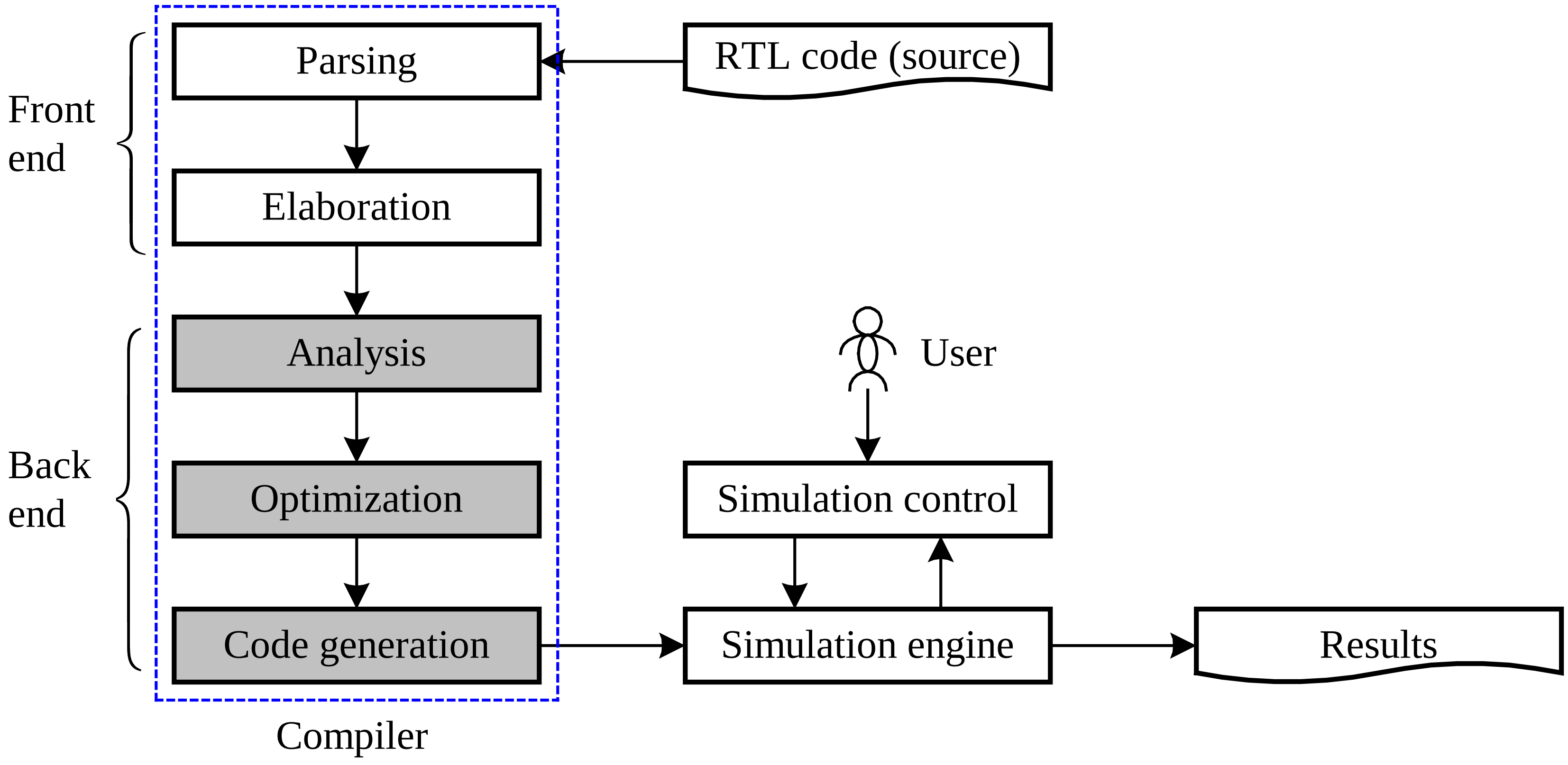
```
module RAM (addr_bus, data_bus);
parameter RAM_depth = 1024;
localparam M=count_log_b2(RAM_depth);
input [M-1:0] addr_bus;
output reg [7:0] data_bus;

…
// function declaration from here
function integer count_log_b2(input integer depth);
     begin   // function body
         count_log_b2 = 0;
         while (depth) begin
             count_log_b2 = count_log_b2 + 1;
             depth = depth >> 1;
         end
     end
   endfunction
endmodule
```

Elaboration time (p. 197 in LRM)

- Elaboration is the process that occurs between parsing and simulation.

- It binds the modules to module instances, builds the model hierarchy, computes parameter values, resolves hierarchical names, establishes net connectivity, and prepares all of this for simulation.

An Architecture of HDL Simulators



System tasks and functions (p. 277 in LRM)

- Display system tasks: $display, $write, $monitor, $strobe
- Timescale system tasks: $printtimescale, $timeformat
- Simulation time system tasks: $time, $realtime
- Simulation control system tasks: $stop, $finish
- File I/O system tasks: $fopen, $fclose
- String formatting system tasks: $swrite, $sformat
- Conversion system tasks: $singed, $unsigned, $rtoi, $itor
- Probablistic distribution system functions
- Stochastic analysis system tasks
- Command line input: $test$plusargs, $value$plusargs
- PLA modeling system tasks (chapter 10)

User Defined Primitives

Two types

- Combinational UDPs
  - are defined where the output is solely determined by the combination of the inputs.
- Sequential UDPs
  - are defined where the output is determined by the combination of the current output and the inputs.

UDP Basics

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
reg output_port;                        // only for sequential UDP
initial output-port = expression;       //only for sequential UDP
table                                   // define the behavior of UDP
<table rows>
endtable
endprimitive
```

UDP Basics

```
// port list declaration style
primitive udp_name(output output_port, input input_ports);
reg output_port;                      // only for sequential UDP
initial output-port = expression;     //only for sequential UDP
table                                 // define the behavior of UDP
<table rows>
endtable
endprimitive
```

Basic UDP Rules

- Inputs
    - can have scalar inputs.
    - can have multiple inputs.
    - are declared with input.
- Output
    - can have only one scalar output.
    - must appear in the first terminal list.
    - is declared with the keyword output.
    - must also be declared as a reg in sequential UDPs.
- UDPs
    - do not support inout ports.
    - are at the same level as modules.
    - are instantiated exactly like gate primitives.

Definition of Combinational UDPs

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
                          // UDP state table
table                     // the state table starts from here
<table rows>
endtable
endprimitive
```

Definition of Combinational UDPs

- State table entries

  <input1><input2>……<input*n*> : <output>;

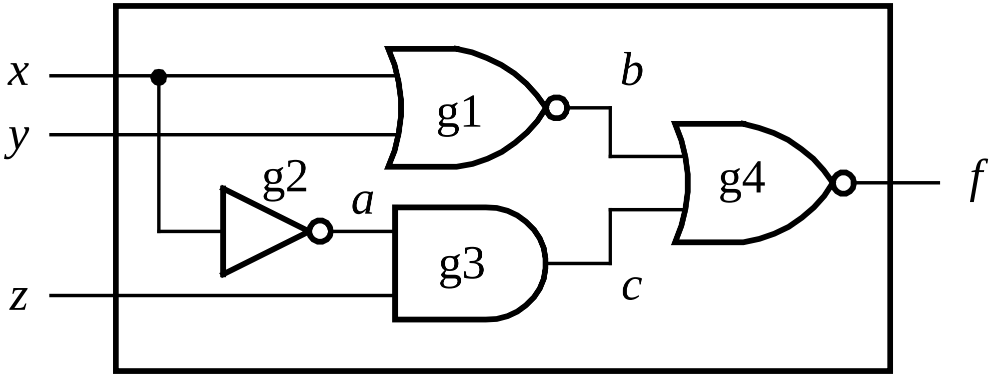
  - The <input#> values appear in a state table entry must be in the same order as in the input list.
  - Inputs and output are separated by a “:”.
  - A state entry ends with a “;”.
  - All possible combinations of inputs must be specified to avoid unknown output value.

A Primitive UDP --- An AND Gate

```
// primitive name and terminal list
primitive udp_and(out, a, b);
// declarations
output out; // must not be declared as reg for combinational UDP
input a, b;  // declarations for inputs.
table        // state table definition; starts with keyword table
//    a   b   :   out;
      0   0   :   0;
      0   1   :   0;
      1   0   :   0;
      1   1   :   1;
endtable          // end state table definition
endprimitive    // end of udp_and definition
```

Another UDP Example

```
// User defined primitive (UDP)
primitive prog253 (output f, input x, y, z);
table   // truth table for f(x, y, z,) = ~(~(x | y) | ~x & z);
//   x y z : f
     0 0 0 : 0;
     0 0 1 : 0;
     0 1 0 : 1;
     0 1 1 : 0;
     1 0 0 : 1;
     1 0 1 : 1;
     1 1 0 : 1;
     1 1 1 : 1;
   endtable
endprimitive
```



Shorthand Notation for Don’t Cares

```
// an example of combinational UDPs.
primitive udp_and(f, a, b);
output f;
input a, b;
table
// a b : f;
   1 1 : 1;
   0 ? : 0;
   ? 0 : 0;               // ? expanded to 0, 1, x
endtable
endprimitive
```

Instantiation of Combinational UDPs

```
// an example illustrates the instantiations of UDPs.
module UDP_full_adder(sum, cout, x, y, cin);
output sum, cout;
input x, y, cin;
wire s1, c1, c2;
// instantiate udp primitives
udp_xor (s1, x, y);
udp_and (c1, x, y);
udp_xor (sum, s1, cin);
udp_and (c2, s1, cin);
udp_or (cout, c1, c2);
endmodule
```

Some synthesizers might not synthesize UDPs.

Shorthand Symbols for Using in UDPs

Symbols	Meaning	Explanation
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state value	Can use only in a sequential UDP output field
r	(01)	Rising edge of a signal
f	(10)	Falling edge of a signal
p	(01), (0x), or (x1)	Potential rising edge of a signal
n	(10), (1x), or (x0)	Potential falling edge of a signal
*	(??)	Any value change in signal

Definition of Sequential UDPs

```
// port list style
primitive udp_name(output_port, input_ports);
output output_port;
input input_ports;
reg output_port;                      // unique for sequential UDP
initial output-port = expression;     // optional for sequential UDP
// UDP state table
table // keyword to start the state table
   <table rows>
endtable
endprimitive
```

Definition of Sequential UDPs

- State table entries

```
<input1><input2>……<inputn> : <current_state> : <next_state>;
```

  - The output is always declared as a reg.
  - An initial statement can be used to initialize output.
  - Inputs, current state, and next state are separated by a colon “:”.
  - The input specification can be input levels or edge transitions.
  - All possible combinations of inputs must be specified to avoid unknown output value.

Level-Sensitive Sequential UDPs

```
// define a level-sensitive latch using UDP.
primitive d_latch(q, d, gate, clear);
output q;
input d, gate, clear;
reg q;
initial q = 0;          // initialize output q
                        //state table
table
// d gate clear : q : q+;
   ?   ?    1   : ? : 0 ;  // clear
   1   1    0   : ? : 1 ;  // latch q = 1
   0   1    0   : ? : 0 ;  // latch q = 0
   ?   0    0   : ? : - ;  // no state change
endtable
endprimitive
```

Edge-Sensitive Sequential UDPs

```
// define a positive-edge triggered T-type flip-flop using UDP.
primitive T_FF(q, clk, clear);
output q;
input clk, clear;
reg q;
// define the behavior of edge-triggered T_FF
table
// clk clear : q : q+;
    ?     1   : ? : 0 ;  // asynchronous clear
    ?   (10) : ? : - ;   // ignore negative edge of clear
   (01)    0 : 1 : 0 ;  // toggle at positive edge
   (01)    0 : 0 : 1 ;  // of clk
   (1?)    0 : ? : - ;   // ignore negative edge of clk
endtable
endprimitive
```

Instantiation of UDPs

```
// an example of sequential UDP instantiations
module ripple_counter(clock, clear, q);
input   clock, clear;
output [3:0] q;

// instantiate the T_FFs.
T_FF tff0(q[0], clock, clear);
T_FF tff1(q[1], q[0],  clear);
T_FF tff2(q[2], q[1],  clear);
T_FF tff3(q[3], q[2],  clear);
endmodule
```

Guidelines for UDP Design

- UDPs model functionality only; they do not model timing or process technology.
- A UDP has exactly one output terminal and is implemented as a lookup table in memory.
- UDPs are not the appropriate method to design a block because they are usually not accepted by synthesis tools.
- The UDP state table should be specified as completely as possible.
- One should use shorthand symbols to combine table entries wherever possible.