KNU KYUNGPOOK NATIONAL UNIVERSITY

COMP311-1 Logic Circuit Design Chap. 8

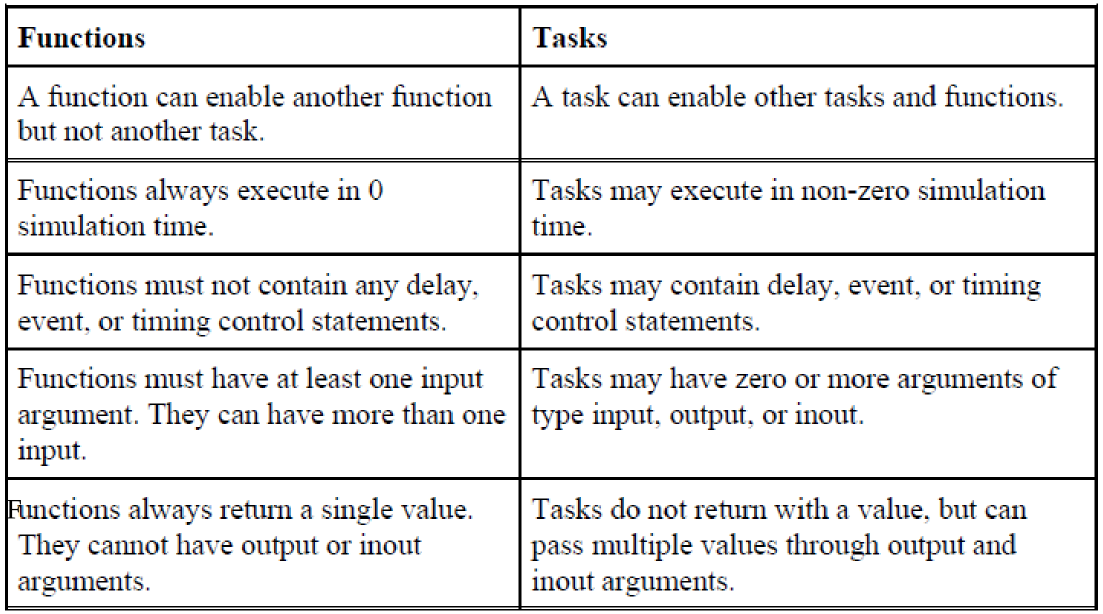
Instructor: Taigon Song (송대건)

2019 Fall

8.1 Differences between Tasks and Functions

펑션은 조합적, 딜레이 없음, 1개의 리턴값
태스크는 순차적, 딜레이 사용가능, 여러 리턴값 가능
둘다 모듈안에 정의되어야 한다. (local)

Table 8-1. Tasks and Functions



Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
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 statements.	Tasks may contain delay, event, or timing control statements.
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.

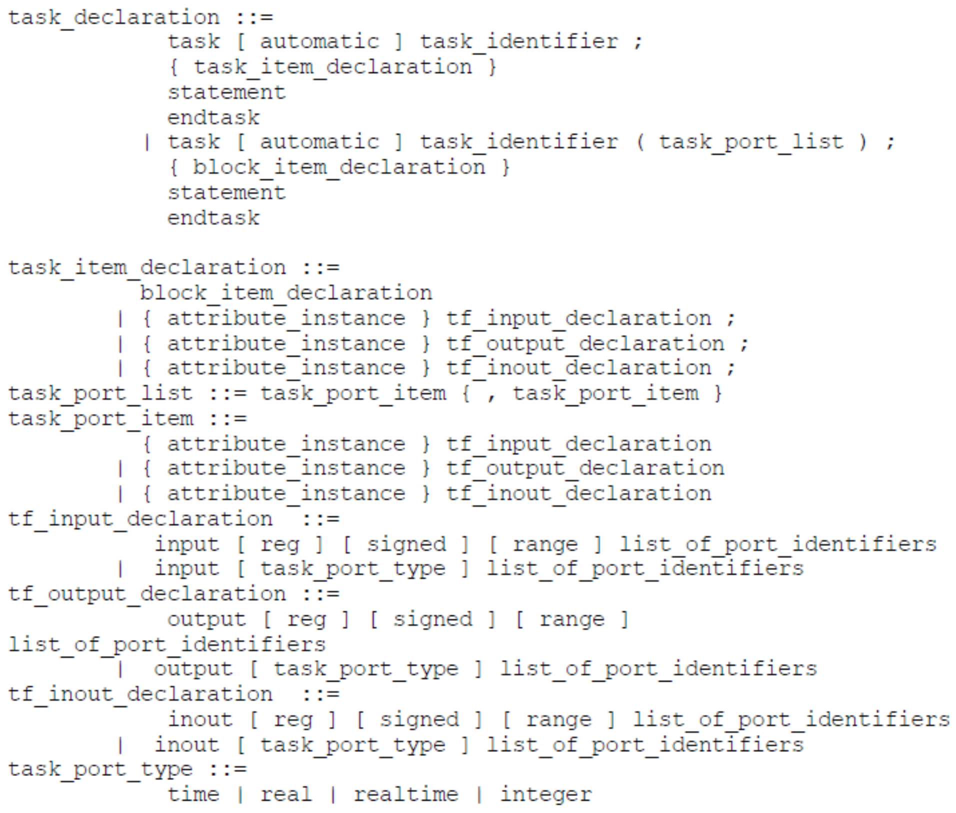
- Note
  - Both tasks and functions must be defined in a module and are local
  - Functions are used for purely combinational purposes w/ one output
  - Tasks and functions do not contain always or initial statements

8.2 Tasks

8.2.1/8.2.2 Task Declaration and Invocation/Task Examples

태스크는 또다른 태스크를 불러올수 있다.

- Some points to notice
  - I/O uses input, output, or inout, based on the type of argument declared
  - Tasks can invoke other tasks or functions
  - “Task” Ports are used to connect external signals to the module
  - I/O arguments in a task are used to pass values to and from the task
  - Can operate on “reg” variables



```
task_declaration ::=
            task [ automatic ] task_identifier ;
            { task_item_declaration }
            statement
            endtask
          | task [ automatic ] task_identifier ( task_port_list ) ;
            { block_item_declaration }
            statement
            endtask

task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } tf_input_declaration ;
        | { attribute_instance } tf_output_declaration ;
        | { attribute_instance } tf_inout_declaration ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::=
          { attribute_instance } tf_input_declaration
        | { attribute_instance } tf_output_declaration
        | { attribute_instance } tf_inout_declaration
tf_input_declaration  ::=
           input [ reg ] [ signed ] [ range ] list_of_port_identifiers
        |  input [ task_port_type ] list_of_port_identifiers
tf_output_declaration ::=
            output [ reg ] [ signed ] [ range ]
list_of_port_identifiers
        |  output [ task_port_type ] list_of_port_identifiers
tf_inout_declaration  ::=
            inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
        |  inout [ task_port_type ] list_of_port_identifiers
task_port_type ::=
            time | real | realtime | integer
```

8.2 Tasks

8.2.2 Task Examples

Example 8-2 Input and Output Arguments in Tasks

```
//Define a module called operation that contains the task bitwise_oper
module operation;
...
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

always @(A or B) //whenever A or B changes in value
begin
        //invoke the task bitwise_oper. provide 2 input arguments A, B
        //Expect 3 output arguments AB_AND, AB_OR, AB_XOR
       //The arguments must be specified in the same order as they
       //appear in the task declaration.
        bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
...
...
//define task bitwise_oper
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor; //outputs from the task
input [15:0] a, b; //inputs to the task
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
...
endmodule
```

태스크 선언 (always 안에)

아웃풋, 인풋 순서 맞춰서 선언
모듈이 길어질 경우 태스크 사용가능

Note.
With arguments

Example 8-3 Task Definition using ANSI C Style Argument Declaration

```
//define task bitwise_oper
task bitwise_oper (output [15:0] ab_and, ab_or, ab_xor,
                   input [15:0] a, b);
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
```

8.2 Tasks

8.2.2 Task Examples

Example 8-4 Direct Operation on reg Variables

```
//Define a module that contains the task asymmetric_sequence
module sequence;
...
reg clock;
...
initial
      init_sequence; //Invoke the task init_sequence
...
always
begin
      asymmetric_sequence; //Invoke the task asymmetric_sequence
end
...
...
//Initialization sequence
task init_sequence;
begin
        clock = 1'b0;
end
endtask

//define task to generate asymmetric sequence
//operate directly on the clock defined in the module.
task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5 clock = 1'b1;
        #3 clock = 1'b0;
        #10 clock = 1'b1;
end
endtask
...
...
endmodule
```

Note.
Without arguments

8.2 Tasks

8.2.3 Automatic (Re-entrant) Tasks

- Tasks are normally static in nature
  - In short, it shouldn't be called at the same time
  - Use the keyword "automatic" when needed to make tasks re-entrant

Example 8-5 Re-entrant (Automatic) Tasks

```
// Module that contains an automatic (re-entrant) task
// Only a small portion of the module that contains the task definition
// is shown in this example. There are two clocks.

// clk2 runs at twice the frequency of clk and is synchronous
// with clk.
module top;
reg [15:0] cd_xor, ef_xor; //variables in module top
reg [15:0] c, d, e, f; //variables in module top
-
task automatic bitwise_xor;
output [15:0] ab_xor; //output from the task
input [15:0] a, b; //inputs to the task
begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
end
endtask
```

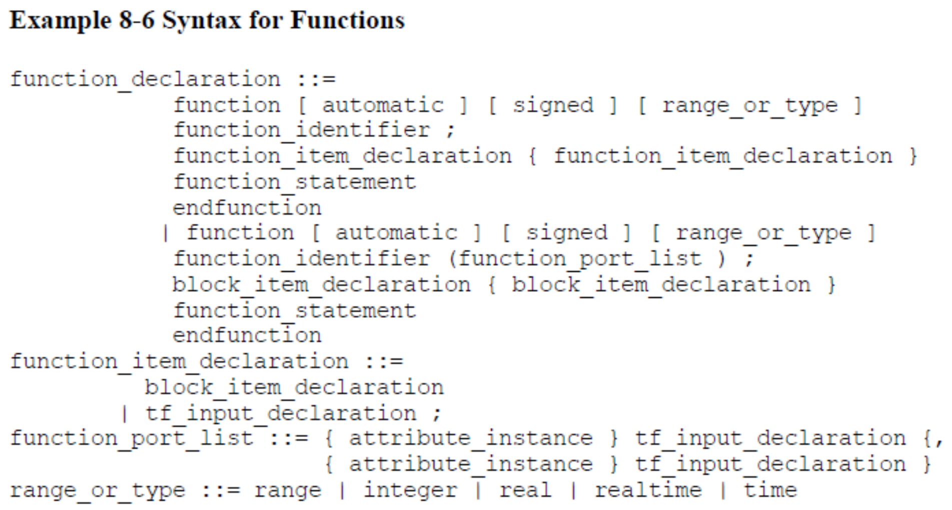
동시에 같은 태스크 여러개 수행시 automatic 키워드 쓴다.

// (continue)

```
...
-
// These two always blocks will call the bitwise_xor task
// concurrently at each positive edge of clk. However, since
// the task is re-entrant, these concurrent calls will work correctly.
always @(posedge clk)
    bitwise_xor(ef_xor, e, f);
-
always @(posedge clk2) // twice the frequency as the previous block
    bitwise_xor(cd_xor, c, d);
-
-
endmodule
```

8.3 Functions

8.3.1 Function Declaration and Invocation (Similar to FORTRAN)



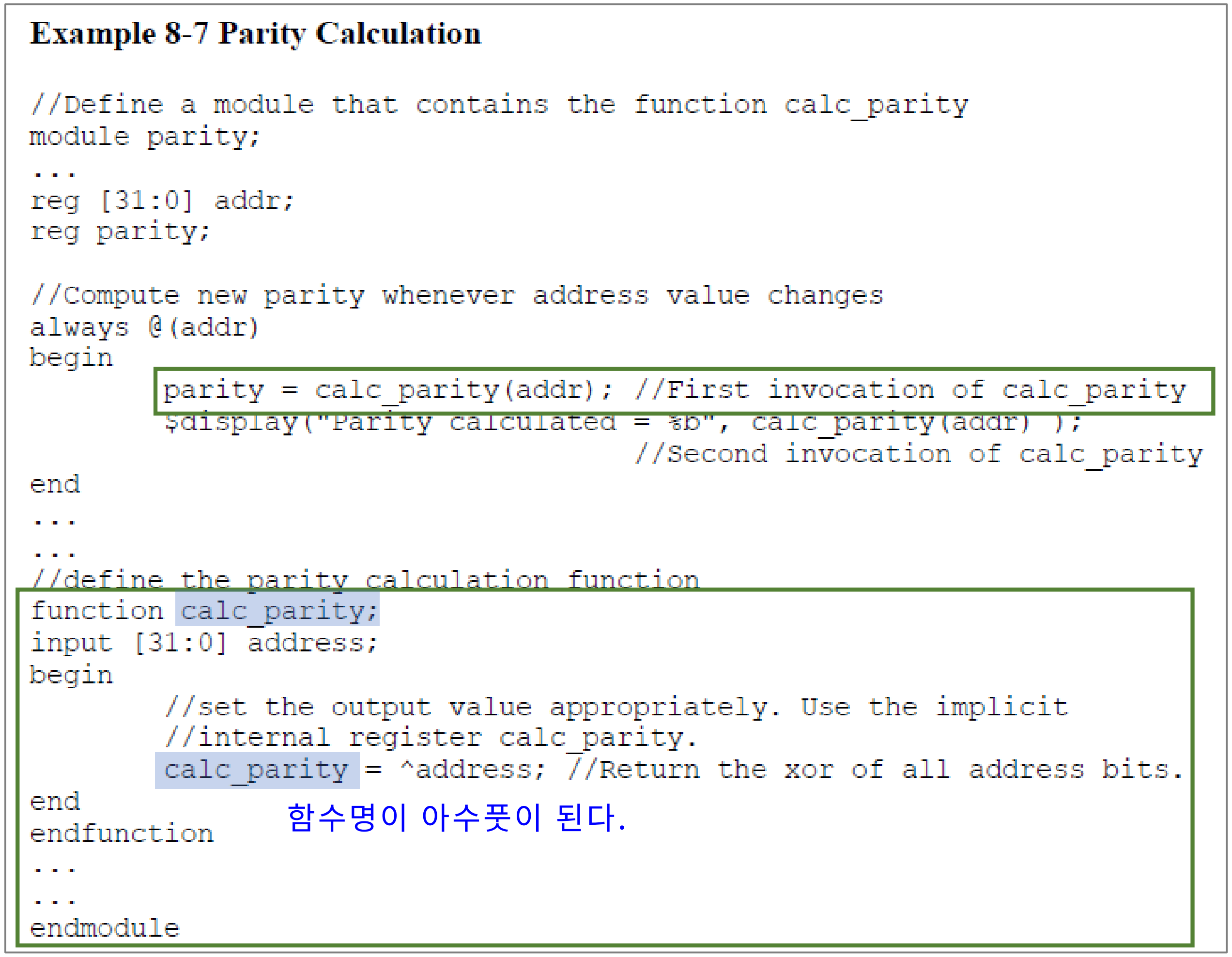
Example 8-6 Syntax for Functions

```
function_declaration ::=
            function [ automatic ] [ signed ] [ range_or_type ]
            function_identifier ;
            function_item_declaration { function_item_declaration }
            function_statement
            endfunction
           | function [ automatic ] [ signed ] [ range_or_type ]
            function_identifier (function_port_list ) ;
            block_item_declaration { block_item_declaration }
            function_statement
            endfunction
function_item_declaration ::=
          block_item_declaration
        | tf_input_declaration ;
function_port_list ::= { attribute_instance } tf_input_declaration {,
                       { attribute_instance } tf_input_declaration }
range_or_type ::= range | integer | real | realtime | time
```

- Functions can be declared if all the following conditions are true
  - No delay, timing, event control   Function 의 특징들
  - Returns a single value
  - At least one input argument
  - No output or inout arguments
  - No non-blocking assignments

8.3 Functions

8.3.2 Function Examples



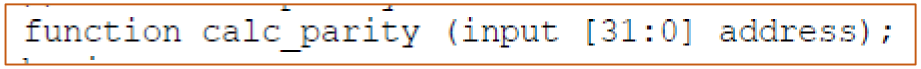
Example 8-7 Parity Calculation

```
//Define a module that contains the function calc_parity
module parity;
...
reg [31:0] addr;
reg parity;

//Compute new parity whenever address value changes
always @(addr)
begin
        parity = calc_parity(addr); //First invocation of calc_parity
        $display("Parity calculated = %b", calc_parity(addr) );
                                    //Second invocation of calc_parity
end
...
...
//define the parity calculation function
function calc_parity;
input [31:0] address;
begin
        //set the output value appropriately. Use the implicit
        //internal register calc_parity.
        calc_parity = ^address; //Return the xor of all address bits.
end
endfunction
...
...
endmodule
```

함수명이 아수풋이 된다.

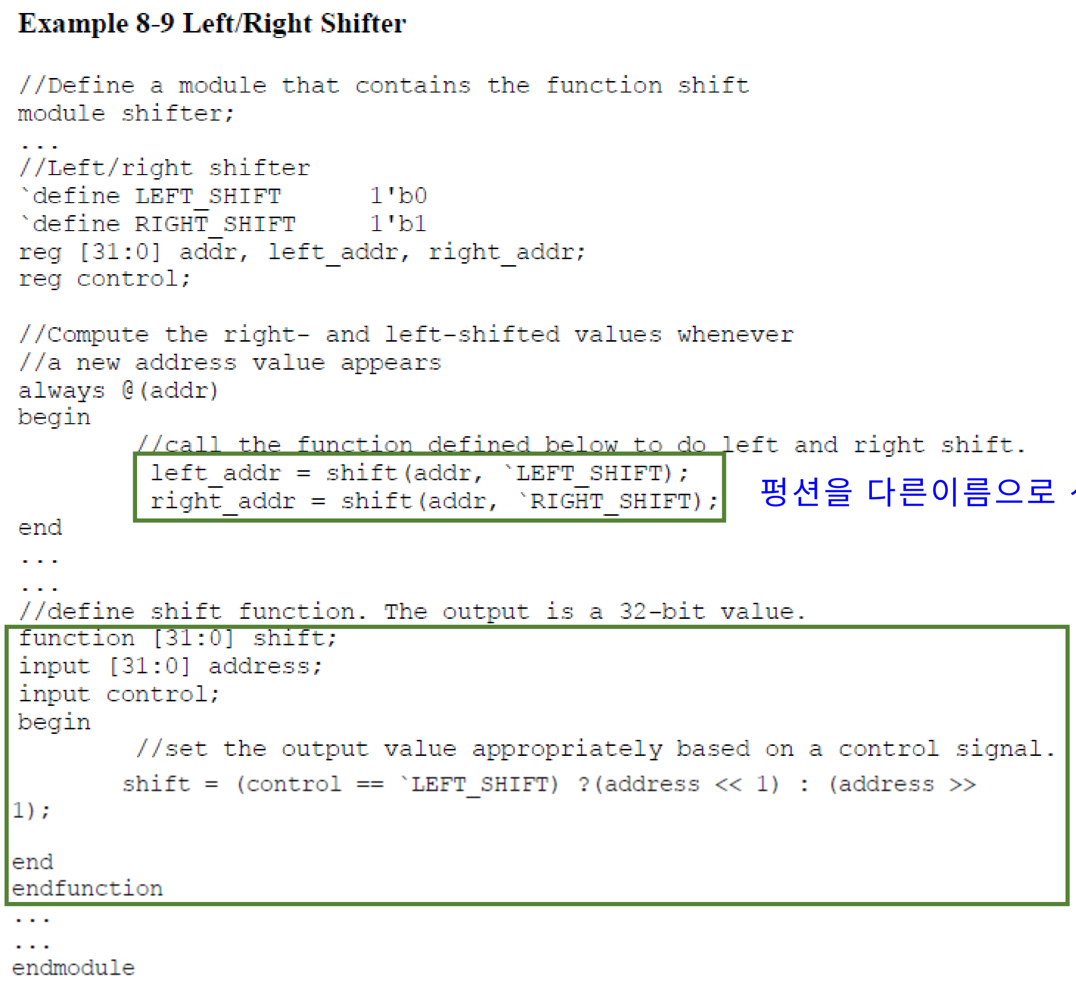
Note. C style also possible



```
function calc_parity (input [31:0] address);
```

8.3 Functions

8.3.2 Function Examples



Example 8-9 Left/Right Shifter

```
//Define a module that contains the function shift
module shifter;
...
//Left/right shifter
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

//Compute the right- and left-shifted values whenever
//a new address value appears
always @(addr)
begin
        //call the function defined below to do left and right shift.
         left_addr = shift(addr, `LEFT_SHIFT);
         right_addr = shift(addr, `RIGHT_SHIFT);
end
...
...
//define shift function. The output is a 32-bit value.
function [31:0] shift;
input [31:0] address;
input control;
begin
        //set the output value appropriately based on a control signal.
        shift = (control == `LEFT_SHIFT) ?(address << 1) : (address >>
1);

end
endfunction
...
...
endmodule
```

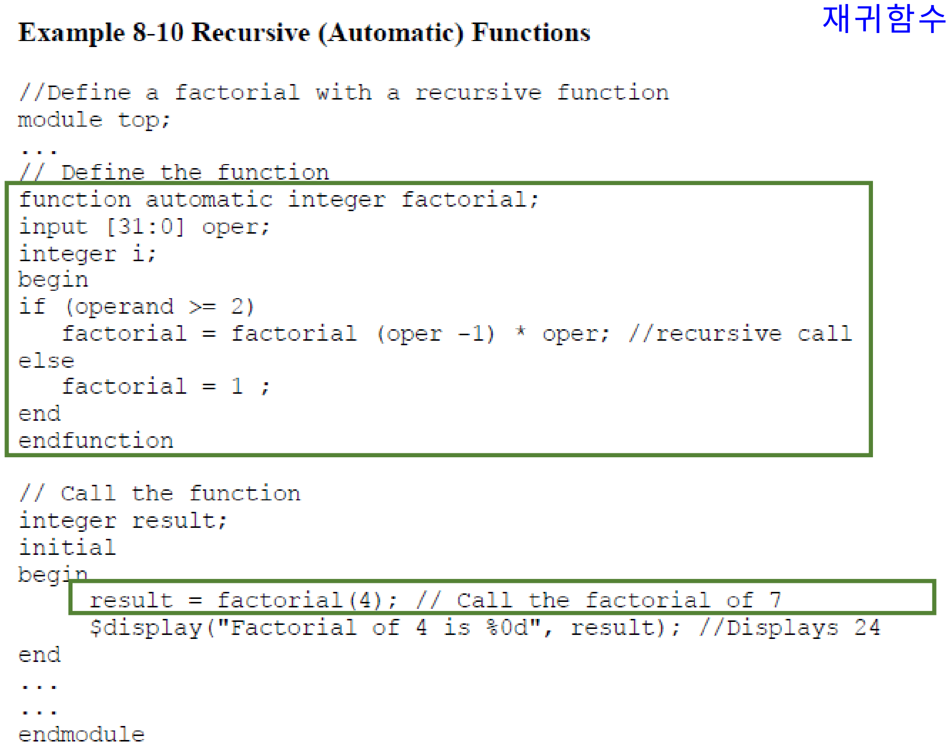
펑션을 다른이름으로 선언시 automatic없이도 동시에 가능

8.3 Functions

8.3.3 Automatic (Recursive) Functions

- Functions are normally used non-recursively
  - Like tasks, it shouldn't be called concurrently from two locations
  - Use the keyword "automatic" when needed to make recursive functions

automaic 쓰면 재귀함수 가능
재귀함수 - 시험안나옴



Example 8-10 Recursive (Automatic) Functions

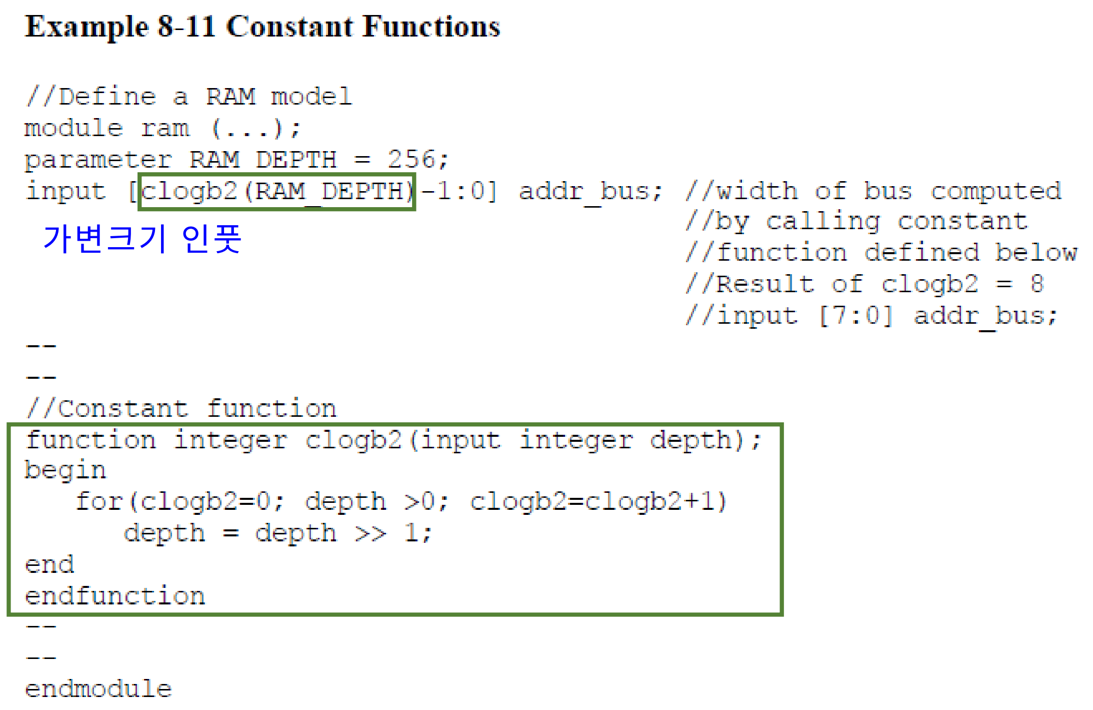
```
//Define a factorial with a recursive function
module top;
...
// Define the function
function automatic integer factorial;
input [31:0] oper;
integer i;
begin
if (operand >= 2)
   factorial = factorial (oper -1) * oper; //recursive call
else
   factorial = 1 ;
end
endfunction

// Call the function
integer result;
initial
begin
     result = factorial(4); // Call the factorial of 7
     $display("Factorial of 4 is %0d", result); //Displays 24
end
...
...
endmodule
```

8.3 Functions

8.3.4 Constant Functions

- Constant function is a regular Verilog function with certain restrictions
  - Can be used to reference complex values
  - Can be used instead of constants



Example 8-11 Constant Functions

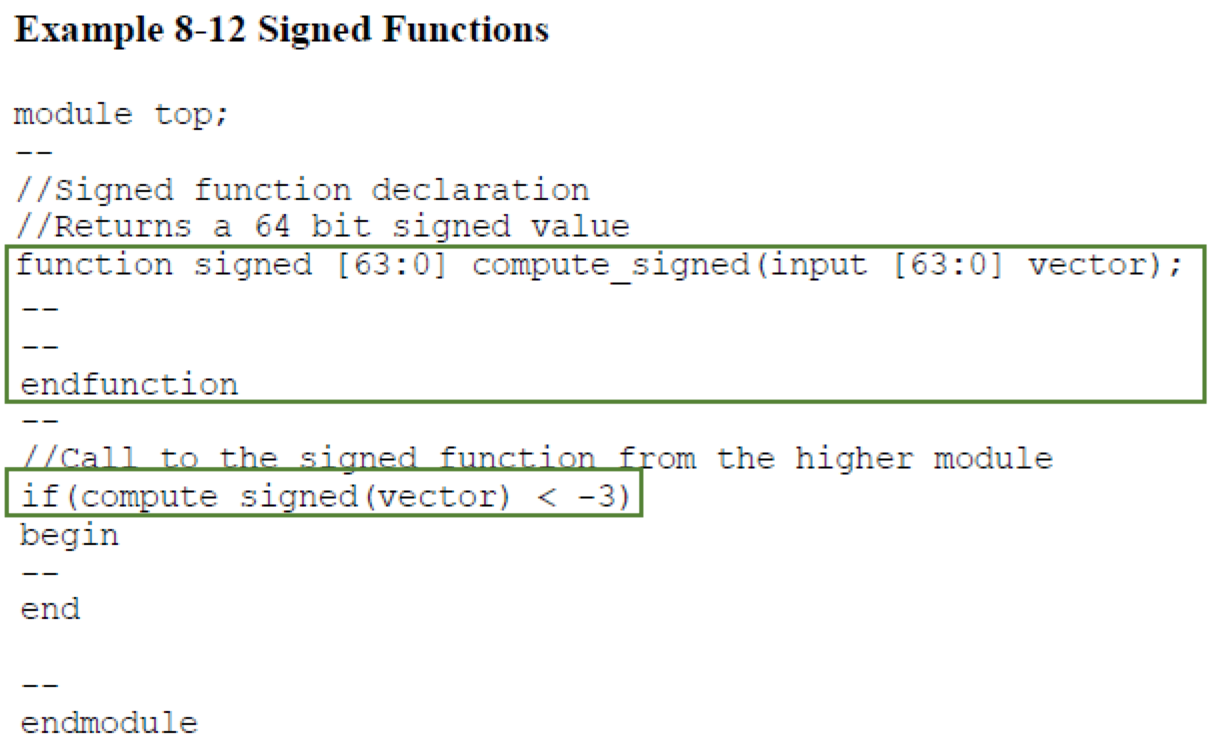
```
//Define a RAM model
module ram (...);
parameter RAM_DEPTH = 256;
input [clogb2(RAM_DEPTH)-1:0] addr_bus; //width of bus computed
                                        //by calling constant
                                        //function defined below
                                        //Result of clogb2 = 8
                                        //input [7:0] addr_bus;
--
--
//Constant function
function integer clogb2(input integer depth);
begin
   for(clogb2=0; depth >0; clogb2=clogb2+1)
      depth = depth >> 1;
end
endfunction
--
--
endmodule
```

가변크기 인풋

8.3 Functions

8.3.5 Signed Functions

- Signed functions allow signed operations to be performed on the function return values



Example 8-12 Signed Functions

```
module top;
--
//Signed function declaration
//Returns a 64 bit signed value
function signed [63:0] compute_signed(input [63:0] vector);
--
--
endfunction
--
//Call to the signed function from the higher module
if(compute signed(vector) < -3)
begin
--
end

--
endmodule
```

In-class Activity

Design a “task” and a “function”

- Task
  - Design delays in a task statement
  - Conditions
    - Design an 4bit adder
    - Two inputs of a 4 bit adder gets new input values every 5ns using task.

- Function
  - Design a function that calculates 4bit x 4bit = 8bit
  - Multiplier
  - Conditions
    - Design a module and a testbench
    - Clk to toggle in every #10
    - Function inside the module

In-class Activity

Design flip-flops (FFs)

combinational : and, or …
sequential : latches , FF, memories
FF종류 : positive edge (올라갈때 값 측정)
negative edge (내려갈때 값 측정)
초기화 : 0 (reset이 0이면 Q를 0으로 초기화,1이면 그냥둠)
1 (preset이 1이면 Q를 1로 초기화, 0이면 그냥둠)
sync : follows the clk (set,reset 값 측정안함)
Async : changes instantly
(set,reset 값의 변화 측정하고 변화즉시 초기화함)

- Design the following FFs
  - Positive-edge-triggered FF
  - Negative-edge-triggered FF

  - Positive-edge FF with async reset at 0
  - Positive-edge FF with sync reset at 0

  - Negative-edge FF with async preset at 0
  - Positive-edge FF with sync preset at 1

  - Positive-edge FF with async reset at 0, sync preset at 1, load at 1

```
module ff(q,d,clk,rst);
        input clk,d;
        output q;
        always@(posetge clk rst)begin
                if(rst==0)
                        q=0;
                else
                        q<=d;
        end
endmodule
```