Tasks and Functions

❑ Learning 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 and Functions - Introduction

- A designer is frequently required to implement the same functionality at many places in a behavioral design
- This means that the commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code
- Most programming languages provide procedures or subroutines to accomplish this
- Verilog provides tasks and functions to break up large behavioral designs 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
- Thus,values can be passed into and out from tasks and functions
- Tasks and functions are included in the design hierarchy. Like named blocks, tasks or functions can be addressed by means of hierarchical names

Differences between 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.

Differences between Tasks and Functions –Cont’d

- Both tasks and functions must be defined in a module and are local to the module
- Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments
- Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output
- Functions are typically used for conversions and commonly used calculations
- Tasks can have input, output, and inout arguments

Differences between Tasks and Functions –Cont’d

- functions can have input arguments. In addition, they can have local variables, registers, time variables, integers, real, or events
- 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

Tasks

- Tasks are declared with the keywords task and endtask

- Tasks must be used if any one of the following conditions is true for the procedure:

    - There are delay, timing, or event control constructs in the procedure

    - The procedure has zero or more than one output arguments

    - The procedure has no input arguments

- Functions are declared with the keywords function and endfunction
- Functions are used if all of the following conditions are true for the procedure:
  - There are no delay, timing, or event control constructs in the procedure
  - The procedure returns a single value
  - There is at least one input argument
  - There are no output or inout arguments
  - There are no nonblocking assignments

- Tasks are normally static in nature
- All declared items are statically allocated and they are shared across all uses of the task executing concurrently
- Therefore, if a task is called concurrently from two places in the code, these task calls will operate on the same task variables
- It is highly likely that the results of such an operation will be incorrect
- To avoid this problem, a keyword automatic is added in front of the task keyword to make the tasks re-entrant. Such tasks are called automatic tasks

- All items declared inside automatic tasks are allocated dynamically for each invocation
- Each task call operates in an independent space. Thus, the task calls operate on independent copies of the task variables. This results in correct operation
- It is recommended that automatic tasks be used if there is a chance that a task might be called concurrently from two locations in the code

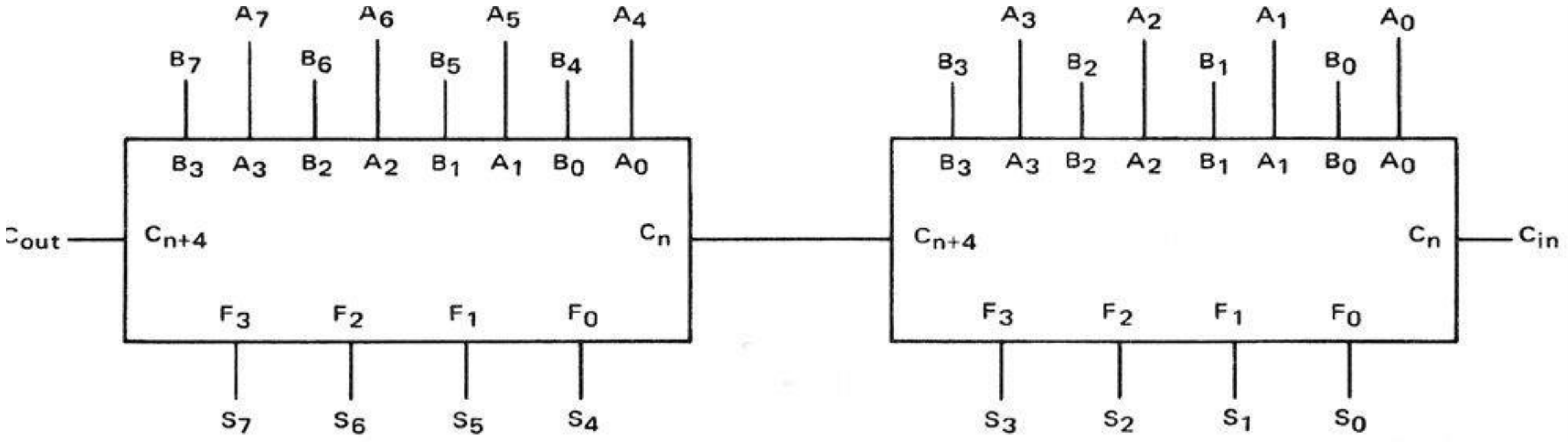
- Functions are normally used non-recursively
- If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space
- However, the keyword automatic can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive calls
- Each call to an automatic function operates in an independent variable space
- Automatic function items cannot be accessed by hierarchical references
- Automatic functions can be invoked through the use of their hierarchical name

Other types of Functions

Constant Functions
A constant function is a regular Verilog HDL function, but with certain restrictions. These functions can be used to reference complex values and can be used instead of constants.

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

8-bit parallel adder using 4-bit tasks and functions



8-bit parallel adder using 4-bit tasks

```
module adder8bit_task4bit(a,b,sum,co);
input [7:0]a,b;
output reg [7:0]sum;
output reg co;
reg cint;
task task4bit;
input [3:0]a_task, b_task;
input cin_task;
output [3:0]sum_task;
output co_task;
{co_task,sum_task}=a_task + b_task + cin_task;
endtask
always@(*)
begin
task4bit(a[3:0], b[3:0], 0, sum[3:0], cint);
task4bit(a[7:4], b[7:4], cint, sum[7:4], co);
end
endmodule
```

8-bit parallel adder using 4-bit tasks

//Test bench

```
`timescale 1ns/1ps
module test_adder8bit;
reg  [7:0]a,b;
wire [7:0]sum;
wire co;
adder8bit_task4bit  add_task(a,b,sum,co);
/*INSTANTIATE THE MODULE NAME(adder8bit_task4bit) THAT NEEDS BE  TESTED WITH INSTANTIATION NAME add_task*/
initial
begin
repeat(10)
begin
 #5 a=$random; b=$random;
end
end
initial
#150 $finish;
initial
$monitor($time, “a=%b, b=%b, sum=%b, co=%b)” , a, b, sum, co);
endmodule
```

8-bit parallel adder using 4-bit functions

```
1.  module adder8bit_function4bit(a,b,sum,co);
2.  input [7:0]a,b;
3.  output reg [7:0]sum;
4.  output reg co;
5.  reg cint;
6.  function [4:0]function4bit;
7.  input [3:0]a_function, b_function;
8.  input cin_function;
9.  function4bit=a_function + b_function + cin_function;
10. endfunction
11. always@(*)
12. begin
13. {cint,sum[3:0]} = function4bit(a[3:0],b[3:0],0);
    {co,sum[7:4]} = function4bit(a[7:4],b[7:4],cint);
14. end
15. endmodule
```

//Test bench

8-bit parallel adder using 4-bit functions

```
`timescale 1ns/1ps
module test_adder8bit;
reg  [7:0]a,b;
wire [7:0]sum;
wire co;
adder8bit_function4bit  add_function(a,b,sum,co);
/*INSTANTIATE THE MODULE NAME(adder8bit_function4bit) THAT NEEDS BE  TESTED WITH INSTANTIATION NAME add_function*/
initial
begin
repeat(10)
begin
#5 a=$random; b=$random;
end
end
initial
#150 $finish;
initial
$monitor($time, “a=%b, b=%b, sum=%b, co=%b)” , a, b, sum, co);
endmodule
```