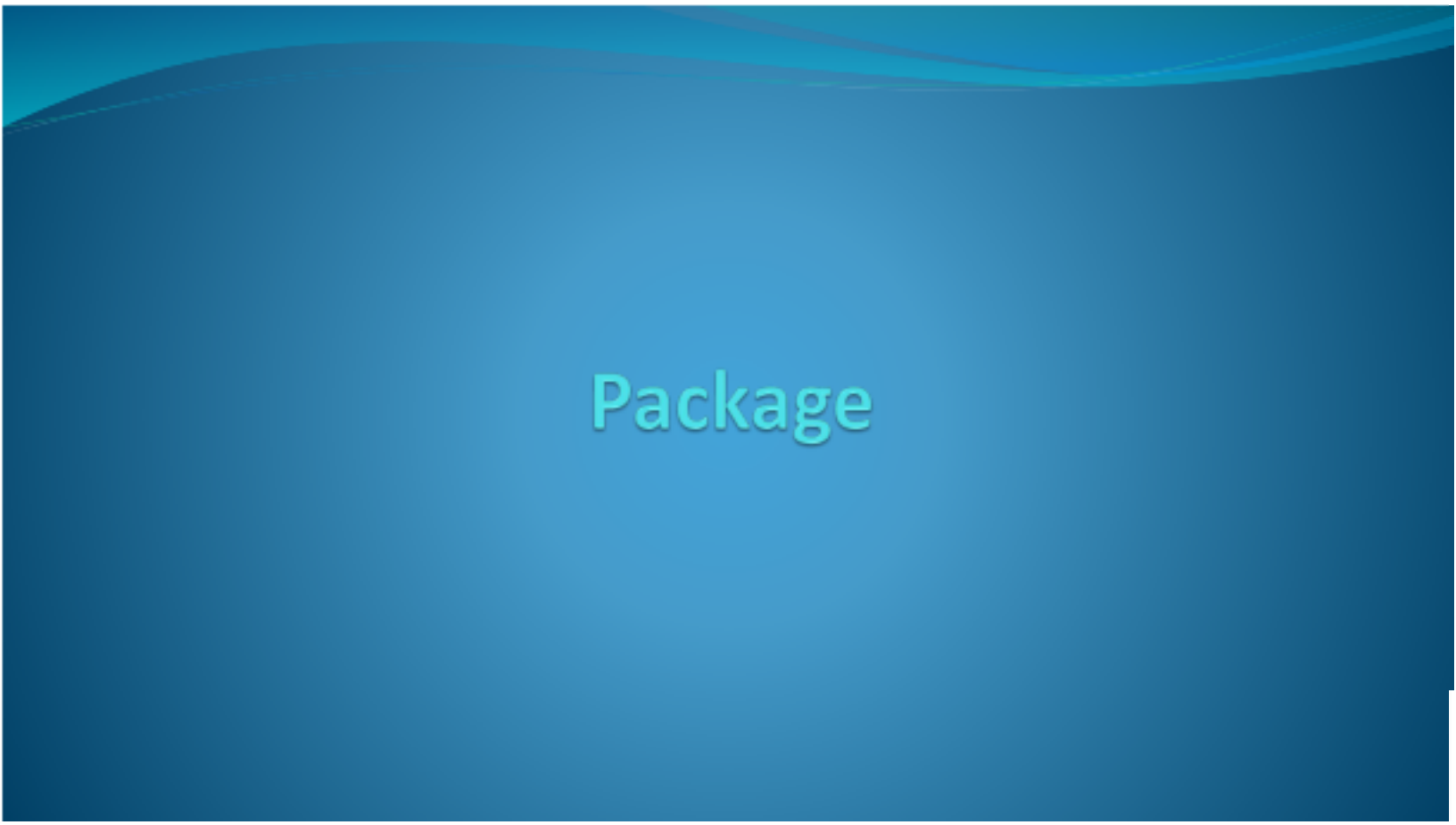

Package

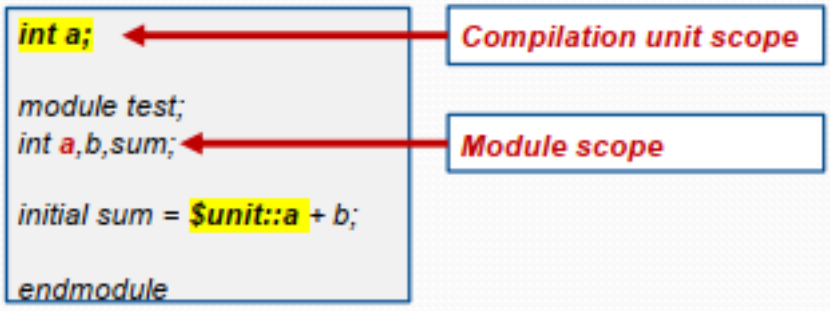
$unit: compilation unit

- Compilation is the process of reading in SystemVerilog source code and analyzing the source code for syntax and semantic errors.
- SystemVerilog supports both single file and multiple file compilation through the use of compilation units.

- **compilation unit:** A collection of one or more SystemVerilog source files compiled together.
  - All files on a given compilation command line make a single compilation unit.
- **compilation-unit scope**: A scope that is local to the compilation unit.
  - It contains **all declarations that lie outside any other scope (module ,interface , package and program)**.
- **$unit:** The name used to explicitly access the identifiers in the compilation-unit scope.
  - Within a particular compilation unit, the special name $unit can be used to explicitly access the declarations of its compilation-unit scope.

```
int a;

module test;
int b,sum;

initial sum = a + b;

endmodule
```

```
int a;

module test;
int a,b,sum;

initial sum = $unit::a + b;

endmodule
```



Packages

- *To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds* ***packages*** *to the Verilog language.*
- *SystemVerilog packages are defined between the keywords* ***package*** *and* ***endpackage.***

```
package pack_ex;

typedef enum {ADD, SUB, MUL} opcodes_t;

typedef struct {
bit [3:0] addr;
bit [31:0] data;
} struct_var;

endpackage
```

Package

```
package global_pkg;

parameter ADDR_WIDTH = 4;
typedef enum {ADD, SUB, MUL} opcodes_t;

typedef struct {
logic [31:0]  op1, op2;
opcodes_t opcode;
} instruction_t;

function automatic [31:0] multiplier (input [31:0] a, b);
return a * b;
endfunction

endpackage
```

Referencing package contents

- You can reference the definitions and declarations in a package four ways:
- Direct reference using a **scope resolution operator**
  package_name::package_var
- **Import specific package items** into the module or interface or program or package
  import package_name::struct_var;
- **Wildcard import** package items into the module or interface or package or program
  import package_name::*;
- Import package items into the **$unit space**
  import package_name::*;

Direct reference using a scope resolution operator

- *Using :: "scope resolution operator" to access package items.*
- *This operator allows directly referencing a package by the package name, and then selecting a specific definition or declaration within the package*

*Ex: **package_name**::**package_item**;*

```
module memory (input global_pkg::enum_t  wr
                                ……..  );
```

```
Function global_pkg::struct_t  foo (global_pkg::enum_t ……..);
```

Importing specific package items

```
program p1;
//import specific item
import global_pkg::struct_st;

struct_st  pst;

initial begin
pst.addr=10;
pst.data=50;
global_pkg::pack_var=30;
 end
endprogram
```

```
package global_pkg;
typedef enum {ADD, SUB, MUL} opcodes_t;

typedef struct {
logic [31:0]  op1, op2;
opcodes_t opcode;
} instruction_t;

function automatic [31:0] multiplier (input [31:0] a, b);
return a * b;
endfunction

typedef struct {
bit [3:0] addr;
bit [31:0] data;
} struct_st;

logic [31:0] pack_var;

endpackage
```

Wildcard Importing

```
program p1;
import global_pkg::*;

struct_st  pst;

initial begin
pst.addr=10;
pst.data=50;
pack_var=30;
 end
endprogram
```

```
package global_pkg;
typedef enum {ADD, SUB, MUL} opcodes_t;

typedef struct {
logic [31:0]  op1, op2;
opcodes_t opcode;
} instruction_t;

function automatic [31:0] multiplier (input [31:0] a, b);
return a * b;
endfunction

typedef struct {
bit [3:0] addr;
bit [31:0] data;
} struct_st;

logic [31:0] pack_var;

endpackage
```

Wildcard import of package items

- *SystemVerilog allows package items to be imported using a wildcard, instead of naming specific package items*

Ex: ***import global_pkg****::**;*

- *A wildcard import does not automatically import all package contents*
- *Only items actually used in the program or module or interface are actually imported.*
- *Definitions and declarations in the package that are not referenced are not imported*

importing packages into $unit space

- Directly referencing a package by the package name.

module memory (input **global_pkg::enum_t** wr ……..);

function **global_pkg::struct_t** foo (**global_pkg::enum_t** ..);

- **Importing package into $unit scope.**

import global_pkg::*;

module memory (input **enum_t** wr ……..);

function **struct_t** foo (**enum_t** ……..);

Synthesizable constructs in package

- *The synthesizable constructs that a packages can contain are:*
- ***parameter** and **localparam** constant definitions*
- ***const** variable definitions*
- ***typedef** user-defined types*
- *Automatic **task** and **function** definitions*
- ***import** statements from other packages*

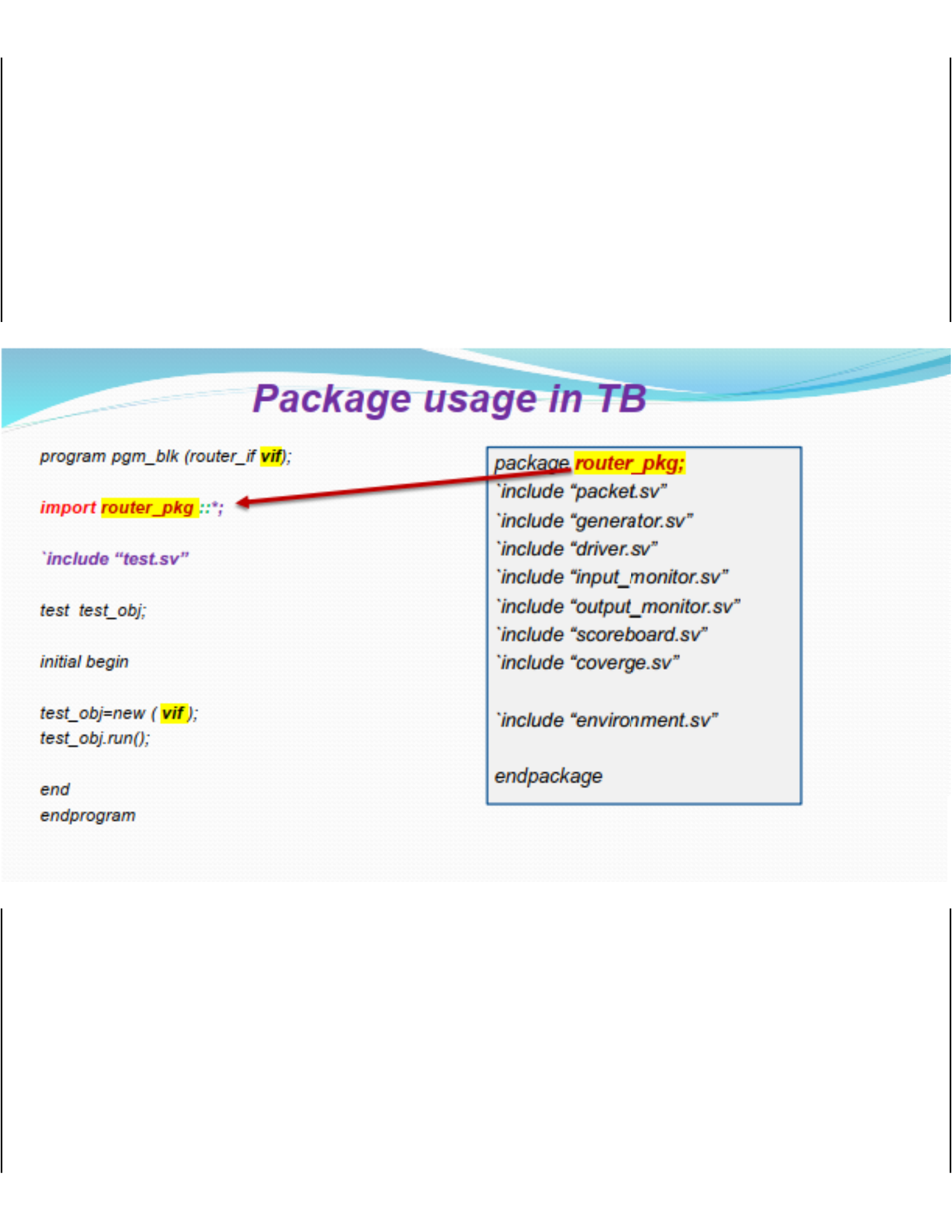
Non-Synthesizable

- *Global variable declarations*
- *static task definitions and static function definitions. These are not synthesizable*

Package usage in TB

```
program pgm_blk (router_if vif);

import router_pkg ::*;

`include "test.sv"

test  test_obj;

initial begin

test_obj=new ( vif );
test_obj.run();

end
endprogram
```

```
package router_pkg;
`include "packet.sv"
`include "generator.sv"
`include "driver.sv"
`include "input_monitor.sv"
`include "output_monitor.sv"
`include "scoreboard.sv"
`include "coverge.sv"

`include "environment.sv"

endpackage
```



Synthesis guidelines

- *When a module references a task or function that is defined in a package, synthesis will duplicate the task or function functionality and treat it as if it had been defined within the module.*
- *To be synthesizable, tasks and functions defined in a package must be declared as **automatic**, and cannot contain static variables.*
- *This is because storage for an automatic task or function is effectively allocated each time it is called.*
- *Thus, each module that references an automatic task or function in a package sees a unique copy of the task or function storage that is not shared by any other module.*
- *Synthesis does not support variables declarations in packages. In simulation, a package variable will be shared by all modules that import the variable. One module can write to the variable, and another module will see the new value. This type of inter-module communication without passing values through module ports is not synthesizable.*