SYNOPSYS®

CUSTOMER EDUCATION SERVICES

SystemVerilog for RTL Design Workshop

Student Guide

50-I-054-SSG-005 2019.03

Synopsys Customer Education Services
690 E. Middlefield Road
Mountain View, California 94043

Workshop Registration: https://training.synopsys.com

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Document Order Number: 50-I-054-SSG-005
SystemVerilog for RTL Design Workshop Student Guide

Table of Contents

Table of Contents

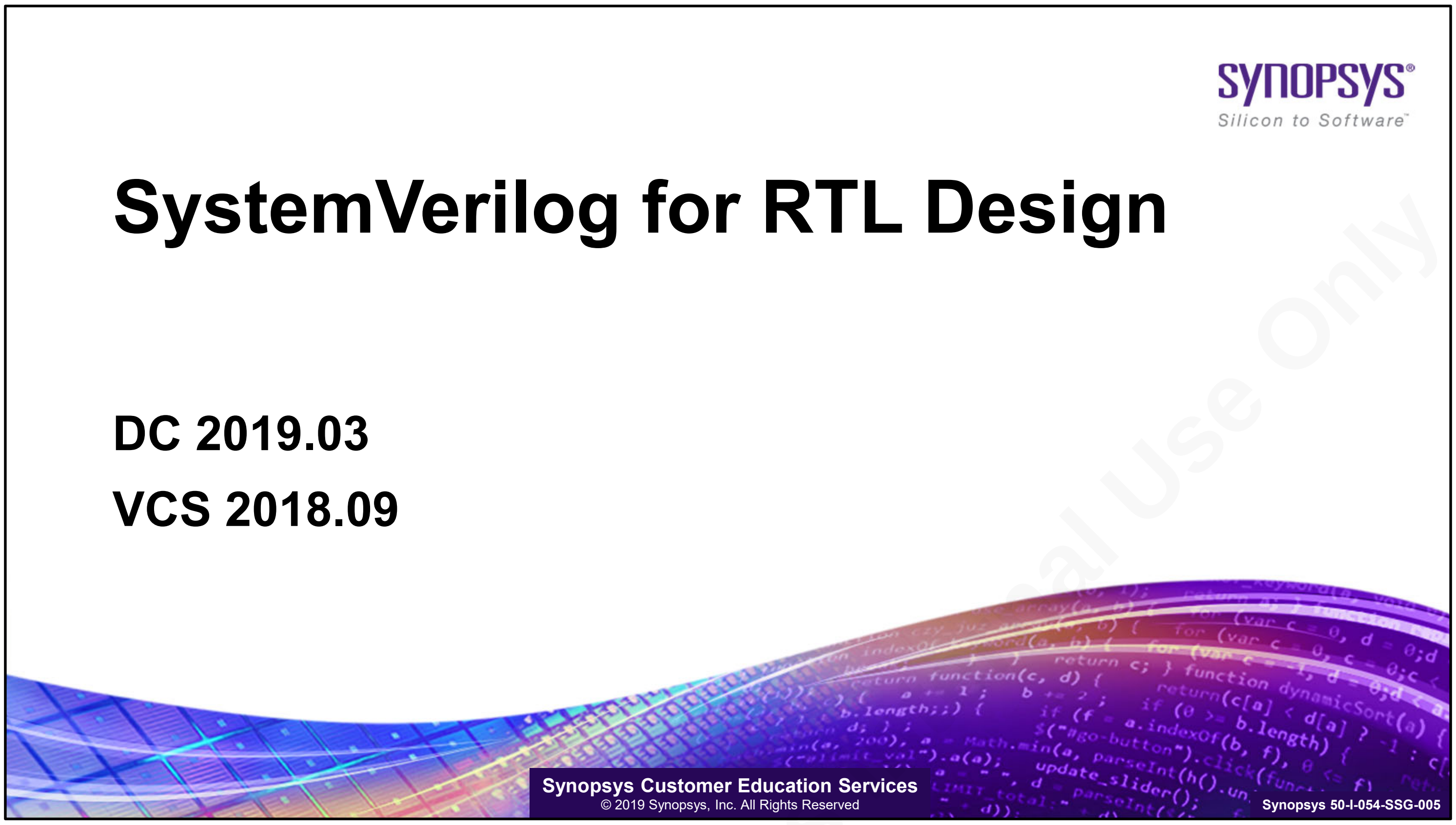
Table of Contents

Table of Contents

Table of Contents

Table of Contents

This page is intentionally left blank



SystemVerilog for RTL Design

DC 2019.03

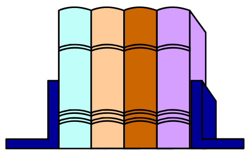
VCS 2018.09

Synopsys Customer Education Services

Synopsys 50-I-054-SSG-005

Workshop Overview

- **Focused on achieving desired SystemVerilog RTL code QoR for Synthesis and Simulation**
  - Concentrating on the most important SystemVerilog synthesizable syntax
  - HDL Compiler for SystemVerilog User Guide
- **Does not cover**
  - How to write constraints for synthesis
  - Topics related to physical layout
    - i.e. DC Topological and MilkyWay Database
- **For the labs:**
  - DC 2019.03 and VCS 2018.09 will be used



Online Documentation available from SolvNet

i- 2

Target Audience

Design engineers moving to SystemVerilog as the primary RTL design language
And verification engineers wanting a better understanding of RTL design

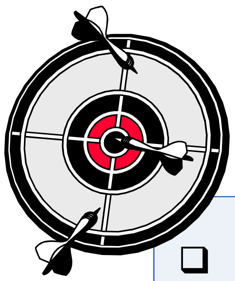
i- 3

Workshop Prerequisite Knowledge

- **You should have experience in the following areas:**
  - Familiarity with a UNIX text editor
  - Synthesis with Verilog, Verilog 2001 or VHDL
  - Familiarity with verification can also help

i- 4

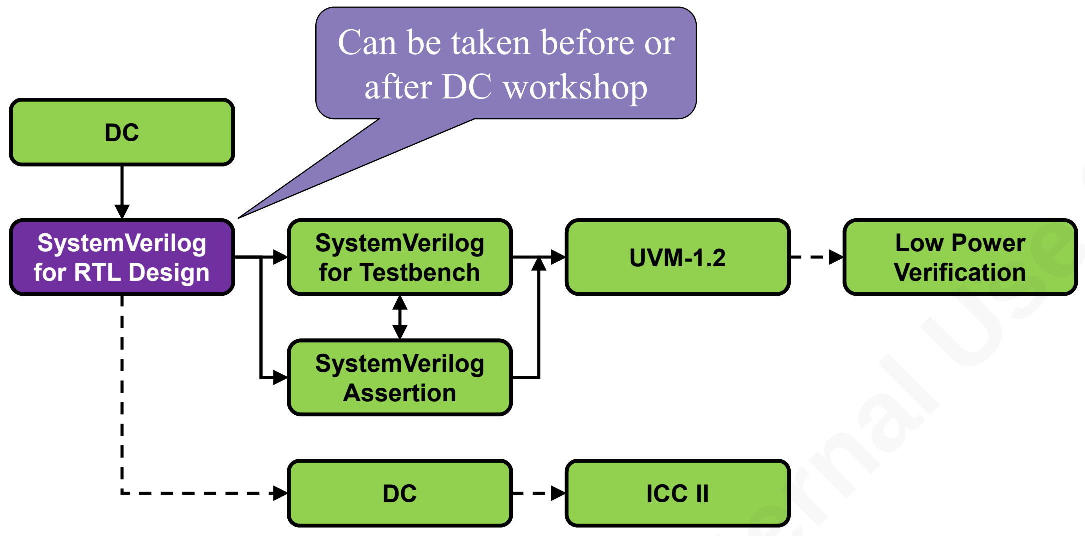
Workshop Goal



- ❑ Correctly implement combinatorial, latch and register logic
- ❑ Use SystemVerilog Interface to simplify module connectivity
- ❑ Manage parameterized netlist for integration and simulation
- ❑ Understand impact of RTL coding style on QoR

i- 5

Workshop Flow


Can be taken before or after DC workshop
DC
SystemVerilog for RTL Design
SystemVerilog for Testbench
UVM-1.2
Low Power Verification
SystemVerilog Assertion
DC
ICC II

The entire Synopsys Customer Education Services course offering can be found at:
http://training.synopsys.com

Synopsys Customer Education Services offers workshops in two formats: The "classic" workshops, delivered at one of our centers, and the online eLearning classes, that one can subscribe to.

Both flavors are delivered by expert Synopsys instructors.

Agenda

i- 7

Syntactic Convention Used in This Workshop

- `Non-italicized` font represent either SystemVerilog or tool keyword
  - Not allowed to be used as user variable method names
- *`Italicized`* font represent user variable or method names
- `[`*`optional`*`]` text enclosed within `[]` are optional

Example:

`typedef enum [`*`val_type`*`] {`*`named_representations`*`}` *`type_e`*`;`

- *`val_type`* defaults to `int`

```
typedef enum {IDLE, INIT, START} state_enum; // val_type is int
```

- Highlighted texts are meant to emphasize key elements of example

i-8

Agenda

Unit Objectives

After completing this unit, you should be able to:

- **Describe SystemVerilog improvement of Verilog features:**
  - Number format
  - Data type
  - Module port list
  - Operators
  - Loop statements
  - Subroutines
  - Parameters

SystemVerilog Lexical Convention

Same as Verilog

- **Case sensitive identifiers (names)**
  - Any sequence of letters, digits, `$`, and `_`
    - First character cannot be a digit or `$`
  - Escaped identifiers start with `\` and end with white space
    - Allow any printable ASCII character in identifier
- **Comments:**
  - `//` …
  - `/*` … `*/` (Does not nest! As in `/*     /*     */     */`)

1-3

SystemVerilog Number Format

- Number format now support `sign`/`unsigned` declaration
  - <size>'[s|S]b (binary) :[01xXzZ]
  - <size>'[s|S]d (decimal) :[0123456789]
  - <size>'[s|S]o (octal) :[01234567xXzZ]
  - <size>'[s|S]h (hex) :[0123456789abcdefABCDEFxXzZ]
  - Sign conversion: `signed'(myvar); unsigned'(myvar);`
- Generic 4-state data type: `logic` // Recommended data type for RTL code

What if you need to set all bits to 1?

```
logic[7:0] my_value; // defaults to unsigned
// logic[7:0] signed my_value; // can be signed
my_value = 1'b1;         // 8'b0000_0001
my_value = 8'b1;         // 8'b0000_0001
my_value = 1'sb1;        // 8'b1111_1111
my_value = 8'sb1;        // 8'b0000_0001
my_value = 8'b1111_1111;
my_value = '1;           // all bits are ones
```

- Un-sized padding of bits

1-4

SystemVerilog Data Types

- **Net data type**
  - Represents types of physical connections
    - Typically used in `module` port list
  - Created with `wire` keyword
    - `wire logic[15:0] dout // dout is a net data object`
  - Can only be driven with a continuous assignment statement
    - Not allowed to be driven by an `always` block
- **Variable data type**
  - Represents a data storage element
    - Typically used inside `module` for local storage of values
  - Created with `var` keyword
    - `var logic[15:0] bus // bus is a variable data object`
  - Can be driven by an `always` block or continuous assignment statement

1-5

Net Data Type Example in `module` Port List (Verilog)

- **Net data type is typically used for connectivity in `module` port list**
- **Verilog `module` non-ANSI style port list**
  - Port list does not have: direction and data type
  - Each port must be re-declared again inside module

```
module my_module (clk, en, chksum, bus);
  input  wire      clk;
  input  wire      en;
  inout  reg[7:0]  bus;
  output reg[7:0]  chksum;
  ...
endmodule
```

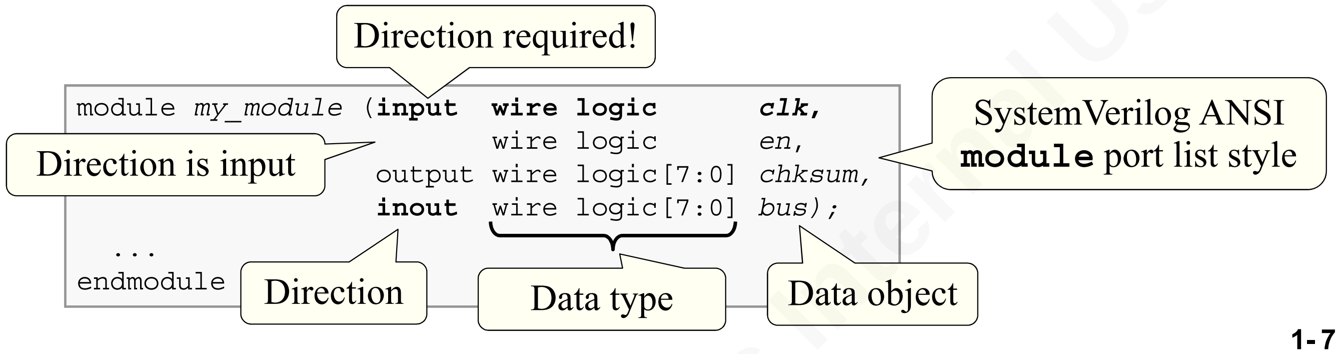
non-ANSI Verilog **`module`** port list style

Port list data objects must be re-declared inside the **`module`** with direction and data type

1-6

Net Data Type Example in `module` Port List (SystemVerilog)

- **Net data type is typically used for connectivity in `module` port list**
- **SystemVerilog `module` ANSI style port list**
  - Each port in list can have: direction, data type and data object
  - First port in the list must have a direction
    - Otherwise defaults to non-ANSI Verilog style
  - For remaining ports, if direction is unspecified, inherit the direction of previous port



```
module my_module (input  wire logic      clk,
                         wire logic      en,
                  output wire logic[7:0] chksum,
                  inout  wire logic[7:0] bus);
  ...
endmodule
```

1-7

SystemVerilog `input` and `inout` Ports Shorthands (1/3)

- **SystemVerilog `module` ANSI style port list allows for shorthands**
  - When the port direction is declared to be `input` or `inout`
    - ♦ If `wire` is specified but the data type `logic` is left off, the default data type is `logic`

```
module my_module (input wire        clk,
                  input wire        en,
                  input wire[7:0]   din,
                  inout wire[7:0]   bus,
                  ...);
  ...
endmodule
```

```
module my_module(input wire logic      clk,
                 input wire logic      en,
                 input wire logic[7:0] din,
                 inout wire logic[7:0] bus,
                 ...);
  ...
endmodule
```

1-8

SystemVerilog `input` and `inout` Ports Shorthands (2/3)

- **SystemVerilog `module` ANSI style port list allows for shorthands**
  - When the port direction is declared to be `input` or `inout`
    - ♦ If `wire` is specified but the data type `logic` is left off, the default data type is `logic`
    - ♦ If data type is declared as `logic` without the keyword `wire`, the data type also defaults to net as though the keyword `wire` was typed

```
module my_module (input logic       clk,
                  input logic       en,
                  input logic[7:0]  din,
                  inout logic[7:0]  bus,
                  ...);
  ...
endmodule
```

Recommended coding style

```
module my_module(input wire logic      clk,
                 input wire logic      en,
                 input wire logic[7:0] din,
                 inout wire logic[7:0] bus,
                 ...);
  ...
endmodule
```

1-9

SystemVerilog `input` and `inout` Ports Shorthands (3/3)

- **SystemVerilog `module` ANSI style port list allows for shorthands**
  - When the port direction is declared to be **`input`** or **`inout`**
    - If **`wire`** is specified but the data type **`logic`** is left off, the default data type is **`logic`**
    - If data type is declared as **`logic`** without the keyword **`wire`**, the data type also defaults to net as though the keyword **`wire`** was typed
    - If the entire data type is undefined, the default data type is once again **`wire logic`**
  - **`input`** signals cannot be driven inside the module
  - **`inout`** signals can only be driven by a continuous assignment statement

```
module my_module (input logic       clk,
                  input logic       en,
                  input logic[7:0]  din,
                  inout logic[7:0]  bus,
                  ...);
  ...
endmodule
```

```
module my_module(input wire logic      clk,
                 input wire logic      en,
                 input wire logic[7:0] din,
                 inout wire logic[7:0] bus,
                 ...);
  ...
endmodule
```

1-10

SystemVerilog `output` Ports Shorthand - Be Careful! (1/4)

- **SystemVerilog `module` port list shorthand for `output`**
  - When the port direction is declared to be **`output`**
    - If **`wire`** is specified but the data type **`logic`** is left off, the default data type is **`logic`**

```
module my_module(...,
                 output wire[7:0] dout,
                 output wire[7:0] bus);
```

```
module my_module(...,
                 output wire logic[7:0] dout,
                 output wire logic[7:0] bus);
```

1- 11

SystemVerilog `output` Ports Shorthand - Be Careful! (2/4)

- **SystemVerilog `module` port list shorthand for `output`**
  - When the port direction is declared to be **`output`**
    - If **`wire`** is specified but the data type **`logic`** is left off, the default data type is **`logic`**
    - If the entire data type is undefined, the default data type is **`wire logic`**

```
module my_module(...,
                 output [7:0] dout,
                 output [7:0] bus);
```

```
module my_module(...,
                 output wire logic[7:0] dout,
                 output wire logic[7:0] bus);
```

1-12

SystemVerilog `output` Ports Shorthand - Be Careful! (3/4)

- **SystemVerilog `module` port list shorthand for `output`**
  - When the port direction is declared to be **`output`**
    - ♦ If **`wire`** is specified but the data type **`logic`** is left off, the default data type is **`logic`**
    - ♦ If the entire data type is undefined, the default data type is **`wire logic`**
    - ♦ If data type is declared as **`logic`** without the keyword **`wire`**, the data type now defaults to variable as though the keyword **`var`** was typed

```
module my_module(...,
                 output logic[7:0] dout,
                 output logic[7:0] bus);
```

```
module my_module(...,
                 output var logic[7:0] dout,
                 output var logic[7:0] bus);
```

1-13

SystemVerilog `output` Ports Shorthand - Be Careful! (4/4)

- **SystemVerilog `module` port list shorthand for `output`**
  - When the port direction is declared to be **`output`**
    - If **`wire`** is specified but the data type **`logic`** is left off, the default data type is **`logic`**
    - If the entire data type is undefined, the default data type is **`wire logic`**
    - If data type is declared as **`logic`** without the keyword **`wire`**, the data type now defaults to variable as though the keyword **`var`** was typed
  - Net data type signals can only be driven by a continuous assignment statement
    - Net data type signals <u>cannot</u> be driven by an always block
  - Only variable data type signals can be driven by an always block
    - Variable data type signals can <u>also</u> be driven by a continuous assignment statement

```
module my_module(...,
                 output logic[7:0] dout,
                 output logic[7:0] bus);
```

```
module my_module(...,
                 output var logic[7:0] dout,
                 output var logic[7:0] bus);
```

Recommended coding style

Declare net data type for output only if tri-state logic is required!

1-14

Recommended SystemVerilog Port List Coding Style

Recommended coding style

```
module my_module(input  logic      clk,
                 input  logic      en,
                 input  logic[7:0] din,
                 inout  logic[7:0] bus,
                 output logic[7:0] dout,
                 output logic[7:0] bus);
```

```
module my_module(input wire logic      clk,
                 input wire logic      en,
                 input wire logic[7:0] din,
                 inout wire logic[7:0] bus,
                 output var logic[7:0] dout,
                 output var logic[7:0] bus);
```

1- 15

SystemVerilog Data Types Local within Module (1/2)

- **4-state data type for creating digital circuitry in `module`**

  `logic [msb:lsb] variable_name;`

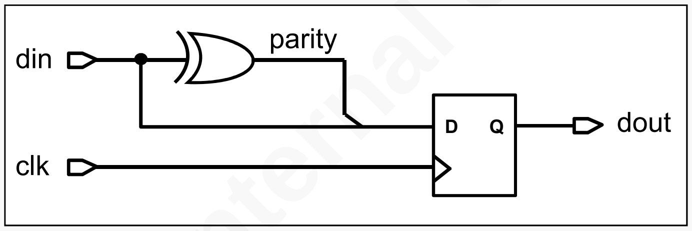
  - Defaults to variable type (`var`), can be changed to net type (`wire`)

    Example: `wire logic [31:0] data;`

  - Defaults to `unsigned`, can be changed to `signed`

    Example: `logic [31:0] signed data;`

```
module Always_Block (input logic clk, logic[3:0] din, output logic[4:0] dout);
  logic      parity;
  always_comb begin
    parity = ^din;
  end
  always_ff @(posedge clk) begin
    dout <= {parity, din};
  end
endmodule
```



1-16

SystemVerilog Data Types Local within Module (2/2)

2-state variables:

- **Used to specify constants in `module`**

  ```
  int variable_name;
  ```

  - Initializes to `'0`

- **Can be signed or unsigned**
  - `unsigned` – `bit`
    - ♦ Should only be used for simulation
  - `signed` – `byte, int, shortint, longint`

```
for (int i=0; i<8; i++) begin
  ...;
end
```

Typically used as loop variables

- **Do not use for synthesis/simulation where `x` and `z` values are important!**

1- 17

SystemVerilog Operators

- RTL code needs to perform logic operation
- From IEEE 1800 spec

Table 11-1—Operators and data types

`=`	Binary assignment
`+= -= /= *=`	Binary arithmetic assignment
`%=`	Binary arithmetic modulus assignment
`&= |= ^=`	Binary bitwise assignment
`>>= <<=`	Binary logical shift assignment
`>>>= <<<=`	Binary arithmetic shift assignment
`?:`	Conditional
`+ -`	Unary arithmetic
`!`	Unary logical negation
`~ & ~& | ~| ^ ~^ ^~`	Unary logical reduction
`+ - * / **`	Binary arithmetic
`%`	Binary arithmetic modulus
`& | ^ ^~ ~^`	Binary bitwise
`>> <<`	Binary logical shift
`>>> <<<`	Binary arithmetic shift
`&& || -> <->`	Binary logical
`< <= > >=`	Binary relational
`=== !==`	Binary case equality
`== !=`	Binary logical equality
`==? !=?`	Binary wildcard equality
`++ --`	Unary increment, decrement
`inside`	Binary set membership
`dist`	Binary distribution
`{} {{}}`	Concatenation, replication
`{<<{}} {>>{}}`	Stream

1-18

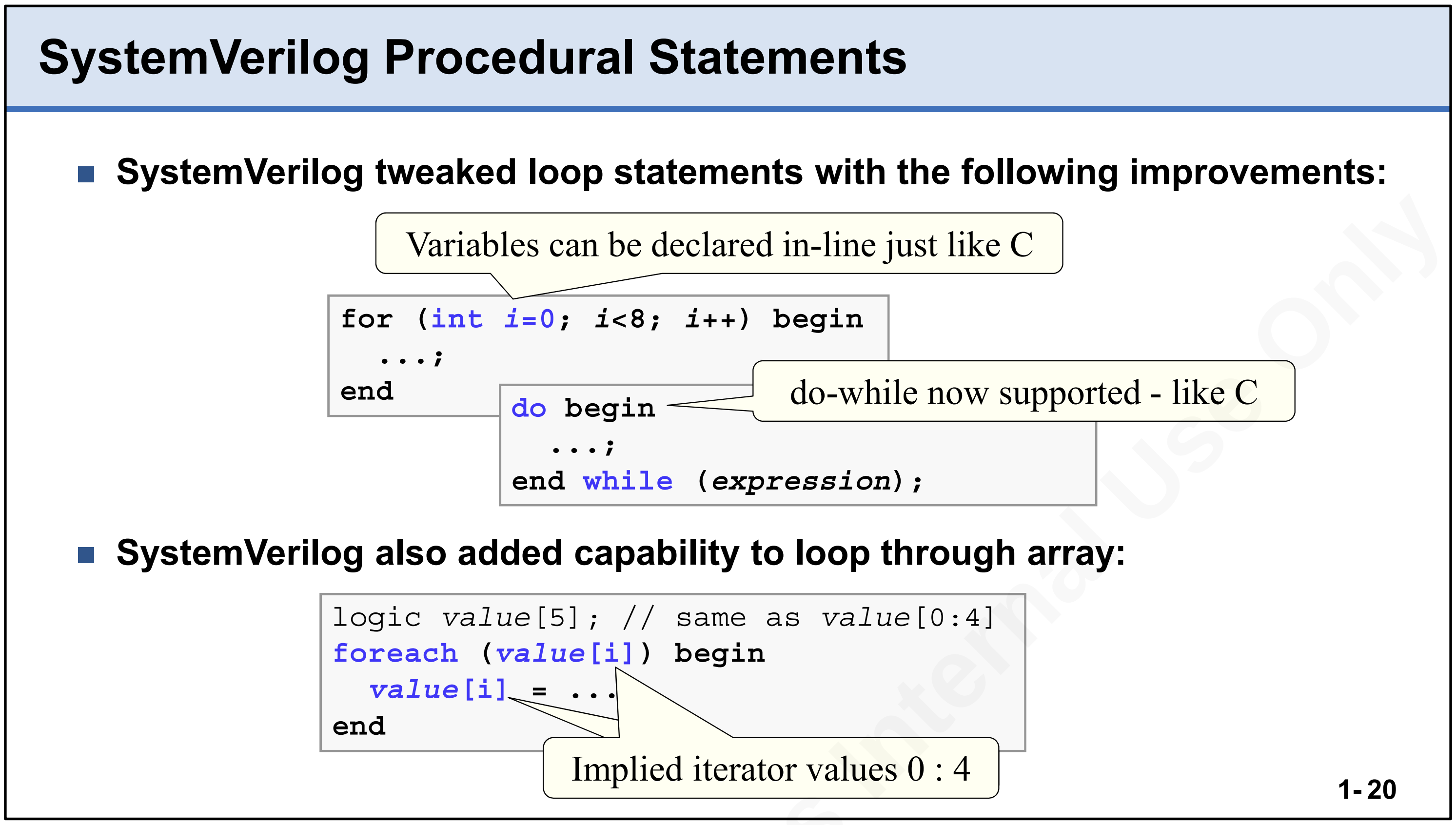
Operator Precedence

- Table from IEEE 1800
- Refer to IEEE for details

Table 11-2—Operator precedence

`() [] :: .`	Highest
`+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ --` (unary)	
`**`	
`* / %`	
`+ -` (binary)	
`<< >> <<< >>>`	
`< <= > >= inside dist`	
`== != === !== ==? !=?`	
`&` (binary)	
`^ ~^ ^~` (binary)	
`|` (binary)	
`&&`	
`||`	
`?:` (conditional operator)	
`-> <->`	
`= += -= *= /= %= &= ^= |=`	
`<<= >>= <<<= >>>= := :/ <=`	
`{} {{}}`	Lowest

1- 19



SystemVerilog Procedural Statements

- SystemVerilog tweaked loop statements with the following improvements:

Variables can be declared in-line just like C

```
for (int i=0; i<8; i++) begin
  ...;
end
```

do-while now supported - like C

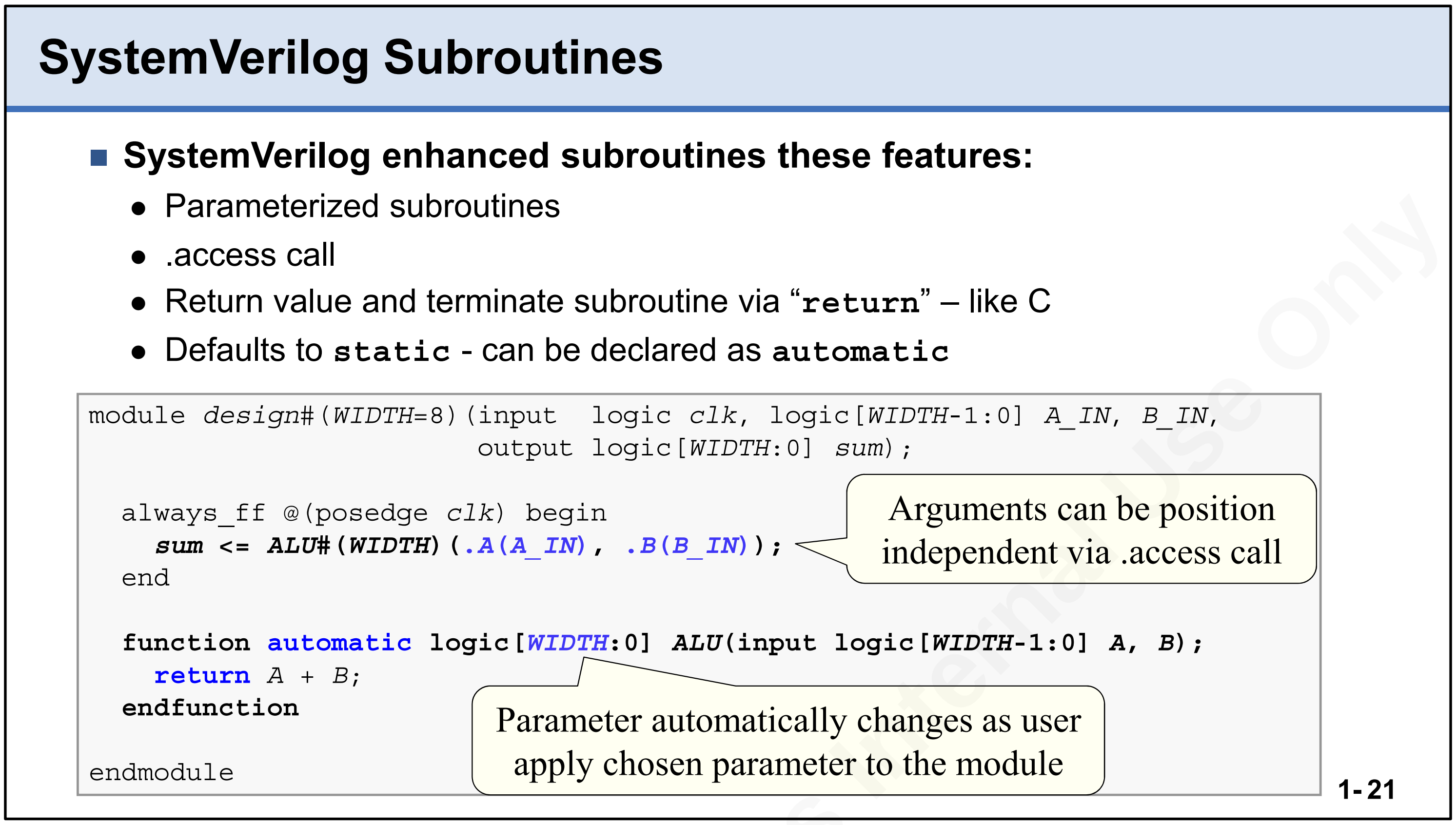
```
do begin
  ...;
end while (expression);
```

- SystemVerilog also added capability to loop through array:

```
logic value[5]; // same as value[0:4]
foreach (value[i]) begin
  value[i] = ...
end
```

Implied iterator values 0 : 4

1-20



SystemVerilog Subroutines

- **SystemVerilog enhanced subroutines these features:**
  - Parameterized subroutines
  - .access call
  - Return value and terminate subroutine via "**`return`**" – like C
  - Defaults to **`static`** - can be declared as **`automatic`**

```
module design#(WIDTH=8)(input  logic clk, logic[WIDTH-1:0] A_IN, B_IN,
                        output logic[WIDTH:0]  sum);

  always_ff @(posedge clk) begin
    sum <= ALU#(WIDTH)(.A(A_IN), .B(B_IN));
  end

  function automatic logic[WIDTH:0] ALU(input logic[WIDTH-1:0] A, B);
    return A + B;
  endfunction

endmodule
```

Arguments can be position independent via .access call

Parameter automatically changes as user apply chosen parameter to the module

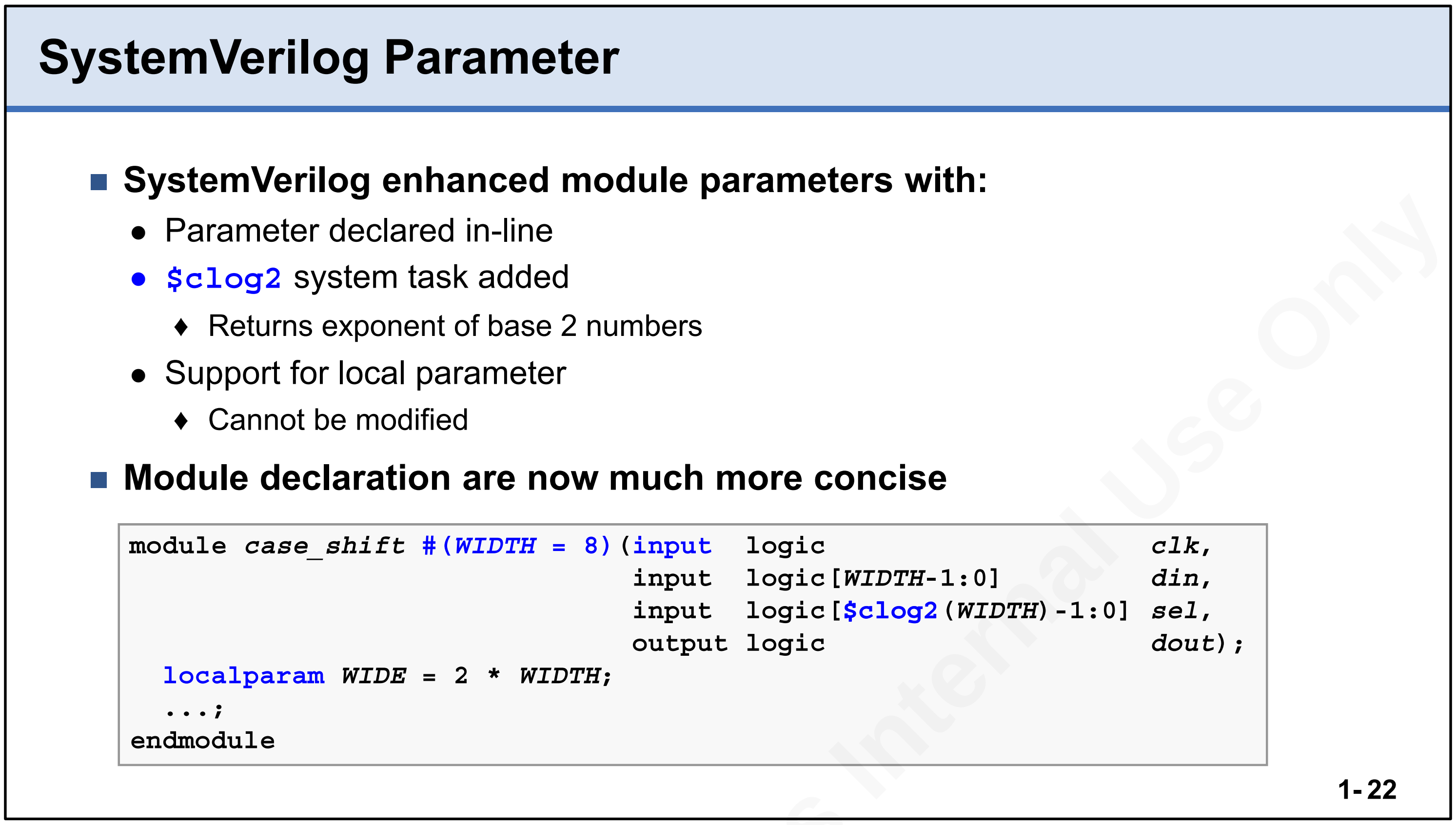
1-21

Tasks and functions can be declared to be "static" or "automatic":

```
task do_it(input A, B, en, output D); // defaults to static
   D = ((A & B) | en);
endtask
task automatic do_it(input A, B, en, output D);
   D = ((A & B) | en);
endtask
function logic do_it(input A, B, en, output D); // defaults to static
   return ((A & B) | en);
endfunction
function automatic logic do_it(input A, B, en, output D);
   return ((A & B) | en);
endtask
```

For simulation, one must pay a great deal of attention. In simulation, automatic subroutines have their own individual memory for each call. In simulation, static subroutines share the same memory for each call.

The coding recommendation for synthesis is to avoid task and only implement functions. And, to avoid synthesis/simulation mismatch, make all tasks and functions automatic.



SystemVerilog Parameter

- **SystemVerilog enhanced module parameters with:**
  - Parameter declared in-line
  - `$clog2` system task added
    - Returns exponent of base 2 numbers
  - Support for local parameter
    - Cannot be modified
- **Module declaration are now much more concise**

```
module case_shift #(WIDTH = 8)(input  logic                     clk,
                               input  logic[WIDTH-1:0]          din,
                               input  logic[$clog2(WIDTH)-1:0]  sel,
                               output logic                     dout);
  localparam WIDE = 2 * WIDTH;
  ...;
endmodule
```

1-22

For Information - Documentation

- **Public IEEE document:**
  - Accellera Public Website – http://www.accellera.org/downloads/ieee
- **SNUG & SolvNet:**
  - https://www.synopsys.com/community/snug.html
  - https://solvnet.synopsys.com

1- 23

Unit Objectives Review

Having completed this unit, you should be able to:

- **Describe SystemVerilog improvement of Verilog features:**
  - Number format
  - Data type
  - Module port list
  - Operators
  - Loop statements
  - Subroutines
  - Parameters

1- 24

Appendix

Basic Mapping of VHDL to SystemVerilog

Case Sensitivity

- **VHDL is not case sensitive**
  - `a` is the same as `A`
  - `begin` is the same as `Begin`
  - `begin` is the same as `BEGIN`
- **SystemVerilog is case sensitive**
  - `a` is <u>NOT</u> the same as `A`
  - `begin` is <u>NOT</u> the same as `Begin`
  - `begin` is <u>NOT</u> the same as `BEGIN`

1-26

Comment

VHDL:

```
-- This is VHDL line comment

-- No VHDL block comment
```

SystemVerilog:

```
// This is SystemVerilog line comment

/*
   This is SystemVerilog
   block comment
*/  // Cannot be nested
```

1-27

Data Type

VHDL:

Predefined
```
  bit (0,1)
  boolean (true,false)
  bit_vector
  integer
```

From `ieee.std_logic_1164.all`
```
  std_logic (u,x,0,1,z,w,l,h,-)
  std_logic_vector
```

SystemVerilog:

2-state (`0,1`)

unsigned (can be a vector)
```
  bit
```

signed
```
  byte      (8-bit)
  shortint  (16-bit)
  int       (32-bit)
  longint   (64-bit)
```

4-state

unsigned (can be a vector)
```
  logic (0, 1, x, z)
```

1-28

2-state data types are typically used in verification code, not RTL code. For RTL code, stick with logic.

Data Object

VHDL:

signal
```
  signal   count : std_logic_vector(3 downto 0);
```

variable
```
  variable i     : std_logic_vector(3 downto 0);
  variable index : integer range 0 to 255;
```

SystemVerilog:

net
```
  wire logic[3:0] count;
```

variable
```
  var  logic[3:0]    i;
  var  byte unsigned index;
```

Assignment rules:

(depends on data object type)		**(independent of data object type)**	
signal:	`<=`	**delayed update:**	`<=`
variable:	`:=`	**immediate update:**	`=`

1-29

The way data object contents are updated is very different for SystemVerilog from VHDL.

In SystemVerilog, nets are meant to represent connectivity and should only be used in RTL to generate tri-state logic.

In SystemVerilog, there are two ways to update the content of the data object. One is called non-blocking (`<=`), the other is called blocking (`=`).

The non-blocking assignment introduces a delay in updating the content of the data object being assigned. The update only happens after all right-hand side of assignments are evaluated (almost exactly the same as the signal assignment in VHDL).

The blocking assignment updates the content of the data object begin assigned immediately without delays – just like the variable assignment in VHDL.

The non-blocking assignment in SystemVerilog is to be used only for generating synchronous logic like flip-flops and latches. Do not use them for combinatorial logic!

Operators

VHDL

logical compare	`a = b`
	`a /= b`
	`a > b`
	`a >= b`
	`a < b`
	`a <= b`
	`a and b`
	`a or b`
	`not a`

concatenation	`a & b`

bit-wise operators (overloaded)	`a and b`
	`a or b`

reduction operators (overloaded)	`and a`
	`or a`

SystemVerilog

logical compare	`a == b`
	`a != b`
	`a > b`
	`a >= b`
	`a < b`
	`a <= b`
	`a && b`
	`a || b`
	`!a`

concatenation	`{ a, b }`

bit-wise operators	`a & b`
	`a | b`

reduction operators (overloaded)	`& a`
	`| b`

1-30

Be very careful of the differences! Especially the concatenation!

Caution on the Operator Differences! (1/2)

VHDL

```
signal a, b : std_logic_vector (3 downto 0);
a <= B"0101";
b <= B"0001";
if (a and b) then -- will not compile
  report("same");
else
  report("different");
end if ;
-- Fails compilation because vectors do not
-- resolve to logical true or false
```

VHDL is strongly type enforced

SystemVerilog is weakly type enforced

SystemVerilog

```
logic[3:0] a, b;
a = 4'b0101;
b = 4'b0001;
if (a && b) begin // Will compile
  $display("same");
end else begin
  $display("different");
end

// a values are OR'ed
//   resulting in 1'b1
// b values are OR'ed
//   resulting in 1'b1
// 1 means true in SystemVerilog
// logical and of two 1's results
// in "same" being printed
```

Caution on the Operator Differences! (2/2)

VHDL

```
signal a, b : std_logic_vector (3 downto 0);
signal c    : std_logic_vector (7 downto 0);
variable  i : integer;
a <= B"0101";
b <= B"0111";
c <= a & b; -- concatenation
report "Value of c is " & to_string(c);

-- Will result in:
-- Value of c is 01010111
```

SystemVerilog

```
logic[3:0] a, b;
logic[7:0] c;

a = 4'b0101;
b = 4'b0111;
c = a & b; // bit-wise and

$display("Value of c is %d", c);

// Will result in:
// Value of c is 5
```

- **Before using an operator, check the IEEE P1800 spec for meaning of the operator!**

1-32

User Defined Data Type

VHDL:

```
type state_e is (IDLE, START, FINISH);
type nibble_t is std_log_vector (3 downto 0);

signal   curr_state, next_state : state_e;
variable nib                    : nibble_t;
```

SystemVerilog:

```
typedef enum logic[2:0] {IDLE, START, FINISH} state_e;
typedef logic[3:0] nibble_t;

state_e  curr_state, next_state;
nibble_t nib;
```

1-33

Data Object Arrays

VHDL:

```
type byte_array is array (0 to 15) of std_logic_vector(7 downto 0);

signal payload : byte_array;

payload(10) <= 20;
```

SystemVerilog:

```
typedef logic[7:0] byte_array_t [16];

byte_array_t payload;

// Or, much easier
logic[7:0] payload[16];
logic[7:0] payload[0:15];

payload[10] = 20;
```

1-34

Data Object Record/Struct

VHDL:

```
type fifo_record_t is record
  full  : std_logic;
  empty : std_logic;
  read  : std_logic;
  write : std_logic;
end record ;

signal fifo_cntrl : fifo_record_t;

fifo_cntrl.read <= '0';
```

SystemVerilog:

```
typedef struct {
  logic full;
  logic empty;
  logic read;
  logic write;
} fifo_struct_t

fifo_struct_t fifo_cntrl;

fifo_cntrl.read = 1'b0;
```

1-35

Port Definition

VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity vhdl_adder is
  port( a    : in  std_logic_vector(3 downto 0);
        b    : in  std_logic_vector(3 downto 0);
        dout : out std_logic_vector(4 downto 0));
end entity ;
```

Data type in port are signals

directions
`in,out,inout,buffer`

SystemVerilog:

```
module sv_adder (
                  input  wire logic[3:0] a,
                  input  wire logic[3:0] b,
                  output var  logic[4:0] dout
                );
```

Data type in port can be net or variable

directions
`input,output,inout`

1-36

Instantiation

VHDL:

```
entity A is
  port ( x, y: in  std_logic;
         z   : out std_logic);
entity B is
  port ( x, y: out std_logic;
         z   : in  std_logic);
entity top is
end entity ;
architecture top_block of top is
  signal x, y, z : std_logic;
begin
 c0: entity work.A
     port map ( x => x, y => y, z => z);
 c1: entity work.B
     port map ( x => x, y => y, z => z);
end architecture ;
```

SystemVerilog:

```
module A (input  logic x, y,
          output logic z);


module B (output logic x, y,
          input  logic z);


module top;
  logic x, y, z;
  A c0(.x(x), .y(y), .z(z));
  B c1(.x(x), .y(y), .z(z));
endmodule
```

1-37

Generic/Parameter

VHDL:

```
// library reference left off
entity vhdl_adder is
  generic (width : integer :=8)
  port(a : in  std_logic_vector(0 to width-1);
       b : in  std_logic_vector(0 to width-1);
       c : out std_logic_vector(0 to width));
end entity ;
```

SystemVerilog:

```

module sv_adder #(width = 8)
     ( input  logic[0:width-1] a,
       input  logic[0:width-1] b,
       output logic[0:width]);
…
endmodule
```

1-38

Behavior – Direct assignment

VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity vhdl_adder is
  port( a    : in  std_logic_vector(3 downto 0);
        b    : in  std_logic_vector(3 downto 0);
        dout : out std_logic_vector(4 downto 0));
end entity ;

architecture adder of vhdl_adder is
begin
  dout <= ( '0' & a ) + ( '0' & b );
end architecture ;
```

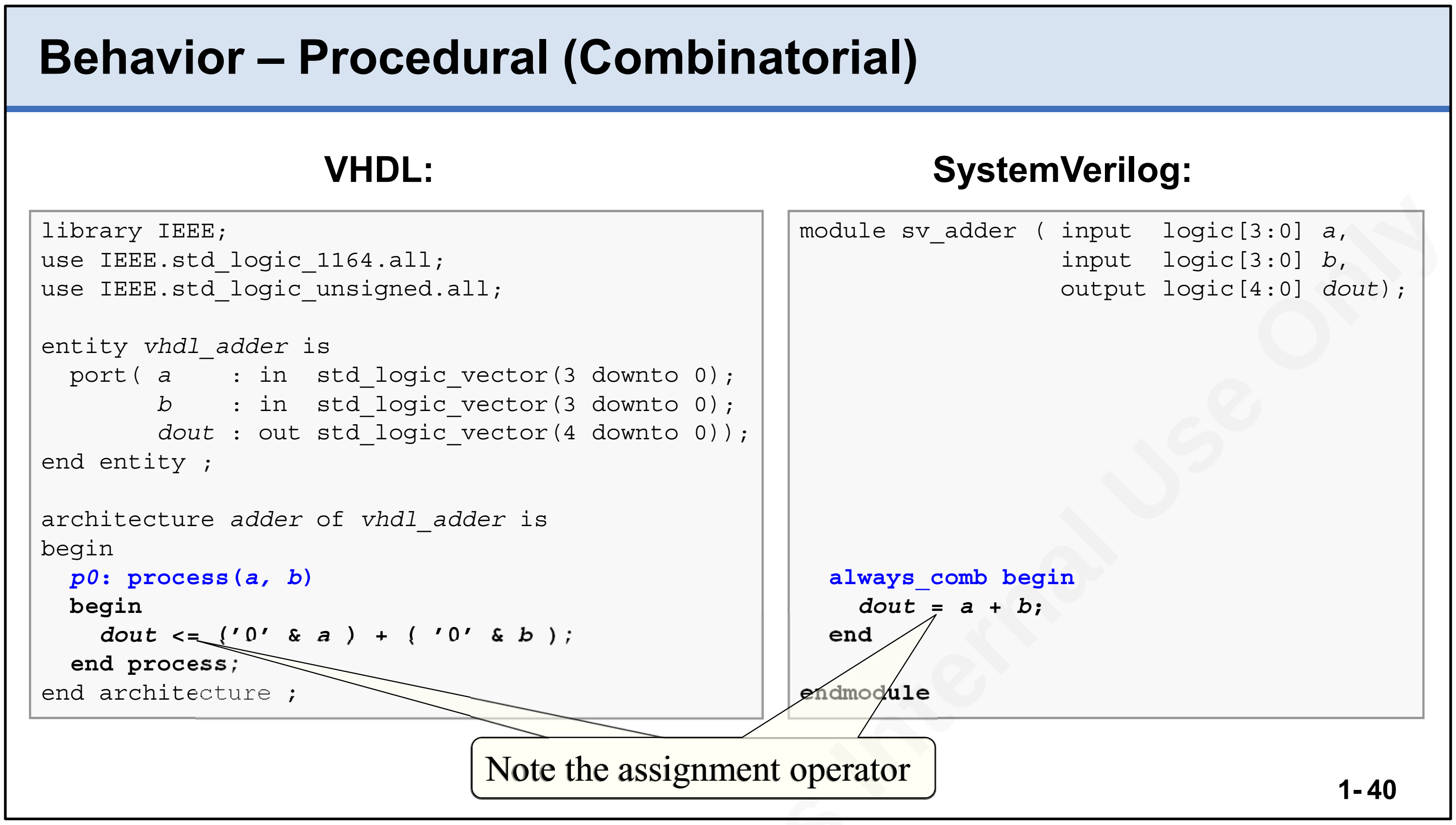
SystemVerilog:

```
module sv_adder ( input  logic[3:0] a,
                  input  logic[3:0] b,
                  output logic[4:0] dout);

  assign dout = a + b;
endmodule
```

Note the assignment operator

1-39

In SystemVerilog, the blocking assignment operator is used in combinatorial logic.



Behavior – Procedural (Combinatorial)

VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity vhdl_adder is
  port( a    : in  std_logic_vector(3 downto 0);
        b    : in  std_logic_vector(3 downto 0);
        dout : out std_logic_vector(4 downto 0));
end entity ;

architecture adder of vhdl_adder is
begin
  p0: process(a, b)
  begin
    dout <= ('0' & a ) + ( '0' & b );
  end process;
end architecture ;
```

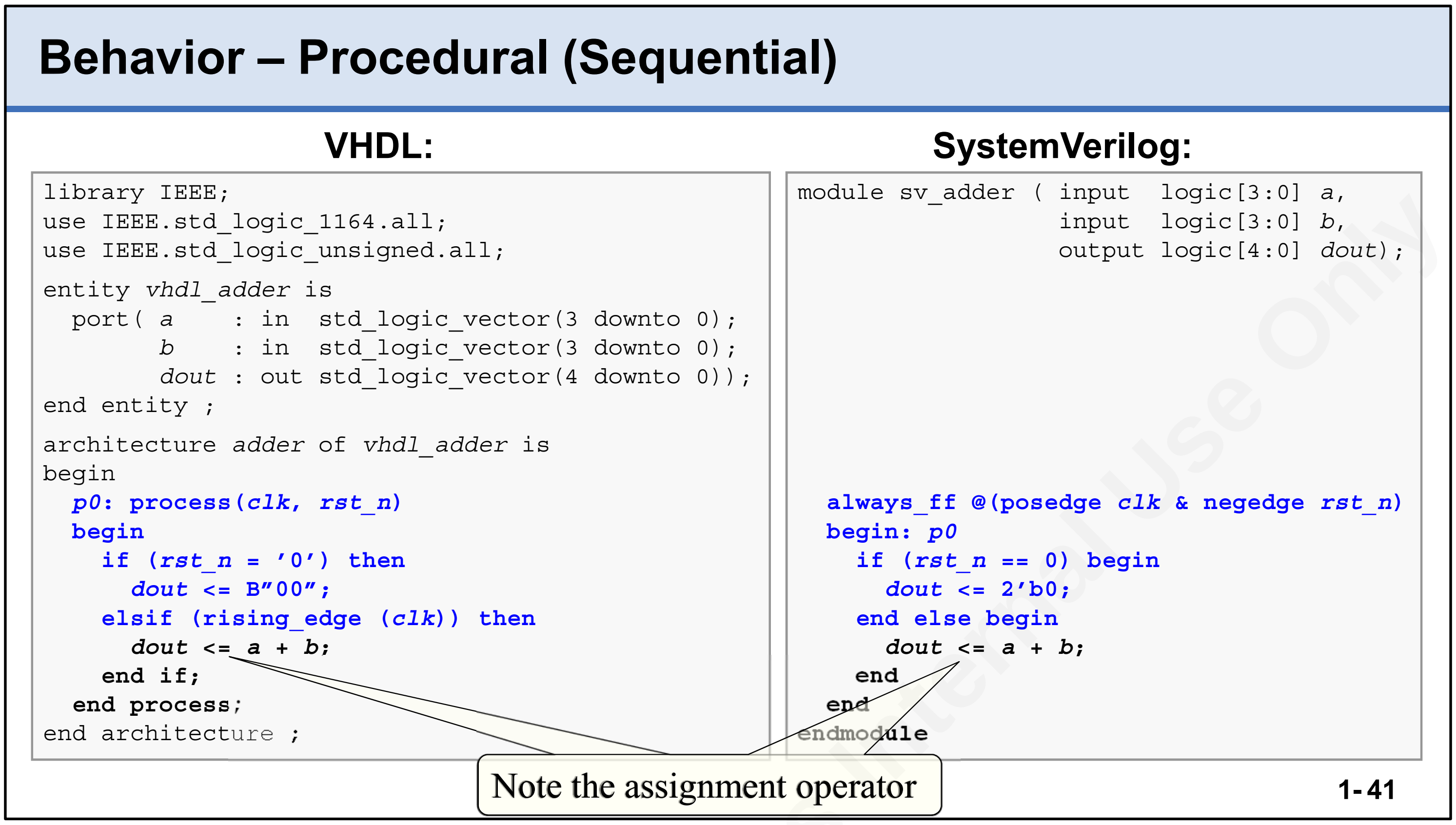
SystemVerilog:

```
module sv_adder ( input  logic[3:0] a,
                  input  logic[3:0] b,
                  output logic[4:0] dout);









  always_comb begin
    dout = a + b;
  end

endmodule
```

Note the assignment operator

1-40

In SystemVerilog, the blocking assignment operator is used in combinatorial logic.



Behavior – Procedural (Sequential)

VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity vhdl_adder is
  port( a    : in  std_logic_vector(3 downto 0);
        b    : in  std_logic_vector(3 downto 0);
        dout : out std_logic_vector(4 downto 0));
end entity ;
architecture adder of vhdl_adder is
begin
  p0: process(clk, rst_n)
  begin
    if (rst_n = '0') then
      dout <= B"00";
    elsif (rising_edge (clk)) then
      dout <= a + b;
    end if;
  end process;
end architecture ;
```

SystemVerilog:

```
module sv_adder ( input  logic[3:0] a,
                  input  logic[3:0] b,
                  output logic[4:0] dout);










  always_ff @(posedge clk & negedge rst_n)
  begin: p0
    if (rst_n == 0) begin
      dout <= 2'b0;
    end else begin
      dout <= a + b;
    end
  end
endmodule
```

Note the assignment operator

1-41

In SystemVerilog, the non-blocking assignment operator is used in creating sequential logic.

Behavior – Procedural (Output Signal used Internally)

VHDL:

```
// library reference left off
entity vhdl_adder is
  port( a    : in     std_logic_vector(3 downto 0);
        b    : in     std_logic_vector(3 downto 0);
        sum  : buffer std_logic_vector(4 downto 0);
        dout : out    std_logic_vector(4 downto 0));
end entity ;
architecture adder of vhdl_adder is
begin
  sum <= ('0' & a ) + ( '0' & b );
  p0: process(clk, rst_n)
  begin
    if (rst_n = '0') then
      dout <= B"00";
    elsif (rising_edge (clk)) then
      dout <= sum;
    end if;
  end process;
end architecture ;
```

SystemVerilog:

```
module sv_adder ( input  logic[3:0] a,
                  input  logic[3:0] b,
                  output logic[4:0] sum,
                  output logic[4:0] dout);







assign sum = a + b;
  always_ff @(posedge clk & negedge rst_n)
  begin: p0
    if (rst_n == 0) begin
      dout <= 2'b0;
    end else begin
      dout <= sum;
    end
  end
endmodule
```

1-42

In SystemVerilog, output signals driven inside the module and used internal to the module direction-wise is called an output.

Structure – If Statements

VHDL

```
type state_e is (IDLE, START, FINISH);
variable state_e state;
p0: process(state) is
begin
  if (state = IDLE) then
    -- do something
  elsif (state = START or state = FINISH) then
    -- do other things
  else
    -- do default things
  end if;
end process ;
```

SystemVerilog

```
typedef enum logic[2:0] {IDLE, START, FINISH} state_e;
state_e state;
always_comb begin
  if (state == IDLE) begin
    -- do something
  end else begin
    if (state == START || state == FINISH) begin
      -- do other things
    end else begin
      -- do default things
    end
  end
end
```

1-43

Structure – Case Statements

SystemVerilog

```
typedef enum logic[2:0] {IDLE, START, FINISH} state_e;
state_e state;
always_comb begin
  case (state)
    IDLE    : // do something
    START   : // do something
    FINISH  : // do same thing as START
    default : // do something
  endcase
end
```

```
// an alternative
case (state) inside
    IDLE          : // do something
    START, FINISH : // do something
    default       : // do something
  endcase
end
```

VHDL

```
type state_e is (IDLE, START, FINISH);
variable state_e state;
p0: process(state) is
begin
  case (state) is
    when IDLE             => -- do something
    when START or FINISH  => -- do something
    when others           => -- do something
  end case ;
end process ;
```

1-44

Structure – Loop Statements

VHDL

```
p0: process is -- forever loop
begin
  loop
    …
    next when next_expression;
    exit when exit_expression;
  end loop ;
end process ;
```

```
-- while loop
  while condition loop
    …
    next when next_expression;
    exit when exit_expression;
  end loop ;
```

```
-- for loop
  for identifier in some_range loop
    …
    next when next_expression;
    exit when exit_expression;
  end loop ;
```

SystemVerilog

```
// forever loop
always_comb begin
  while(1) begin
    …
    if (continue_expression) continue;
    if (exit_expression) break;
  end
end
```

```
// while loop
  while (condition) begin
    …
    if (continue_expression) continue;
    if (exit_expression) break;
  end
```

```
// for loop
  for (int i=0; i<8; i++) begin
    …
    if (continue_expression) continue;
    if (exit_expression) break;
  end
```

1-45

Implementation – Function

VHDL:

```
// library reference left off
entity vhdl_add is
  port(a, b : in  std_logic;
       dout : out std_logic_vector(1 down to 0);
end entity ;
architecture adder of vhdl_add is
  function vhd_add (x, y : in std_logic)
    return std_logic_vector is
    variable sum : std_logic_vector(1 downto 0);
  begin
    sum := ('0' & x) + ('0' & y);
    return sum;
  end function vhd_add ;
begin
  dout <= vhd_add(a, b);
end architecture ;
```

SystemVerilog:

```
module sv_add (input  logic      a, b,
               output logic[1:0] dout);
  function logic[1:0] sv_add(logic x, y);
    return x + y;
  endfunction: sv_add
  assign dout = sv_add(a, b);
endmodule
```

1-46

Implementation – Procedure/Task

VHDL:

```
// entity definition left off
architecture read_memory of vhdl_cpu is

-- Only used for simulation
  procedure read_data ( … ) is
  begin
    -- read_data code
  end procedure ;
begin
  behavior_code: process is
    -- some signal/variable declaration
    begin
      -- do something
      read_data(…); -- execute procedure
      -- do more things
    end
  end process ;
end architecture ;
```

SystemVerilog:

```

module sv_cpu ( /* list left off */ );

// Only create for simulation
  task read_data( … );
    // read_data code
  endtask: task


initial begin
  // some variable declaration
  // do something
  read_data(…); // execute task
  // do more things
end


endmodule
```

1-47

Package

VHDL:

```
package PKG_EXAMPLE is
  generic (width : integer);
  type STATE is (RESET, IDLE, DONE);
  function some_function (…) return return_type;
end package ;
package body PKG_EXAMPLE is
  function some_function (…) return return_type is
  begin … end function ;
end package body ;
package my_pkg is new work.PKG_EXAMPLE generic map (width => 32);
use work.my_pkg.all;
```

User change-able

User must instantiate the package

SystemVerilog:

```
package pkg_example;
  parameter width = 32;
  typedef enum logic[1:0] {RESET, IDLE, DONE} state_e;
  function automatic return_type some_function(...)
    ...
  endfunction
endpackage
import pkg_example::*;
```

Cannot be changed!

Can only have one definition

User must specify functions to be automatic

1-48

Generate Mechanism: VHDL

```
entity fa is port (cin, a, b: in std_logic; sum, cout: out std_logic); end entity ;
entity adder is
  port ( … ); -- port list unimportant to subject matter
end entity ;
architecture implementation of adder is
  signal carry_i, carry_o : std_logic_vector(7 downto 0);
  component fa
    port (cin, a, b : in  std_logic;
          sum, cout : out std_logic);
  end componet ;
begin
  carry_i(0) <= cin; carry_i(7 downto 1) <= carry_o(6 downto 0); cout <= carry_o(7);
  gen_adder: for i in 0 to 7 generate
  begin
    fa_inst : fa
      port map(cin=>carry_i(i), a=>a(i), b=>b(i), cout=>carry_o(i), sum=>sum(i));
  end generate ;
end architecture ;
```

1-49

Generate Mechanism: SystemVerilog

```
module fa(input logic cin, a, b, output logic cout, sum);
  assign { cout, sum } = cin + a + b ;
endmodule
module adder(input logic cin, logic[7:0] a, b, output logic cout, logic[7:0] sum);
  logic[7:0] carry_o, carry_i;
  assign carry_i[0]    = cin;
  assign carry_i[7:1] = carry_o[6:0];
  assign cout          = carry_o[7];
  genvar i;
  generate
    for (i = 0; i <= 7; i = i+1) begin
    fa fa_i(.cin(carry_i[i]), .a(a[i]), .b(b[i]), .cout(carry_o[i]), .sum(sum[i]));
  end
  endgenerate
endmodule
```

1- 50

Agenda

2- 1

Unit Objectives

After completing this unit, you should be able to:

- **Write RTL code for combinatorial logic**
- **Avoid unintended latch**
- **Avoid synthesis/simulation mismatch**
- **Create registers with synchronous/asynchronous reset**
- **Understand the meaning of full and parallel**
- **Use `enum` data type for state machines**

2- 2

Typical Hardware Logic Needs

- **Combinatorial logic**
  - Mathematical operation, multiplexer
- **Latch**
  - Level sensitive data capture
- **Register**
  - Pipeline segment, state machine, shift register, register file
- **Tri-state logic**
  - Single direction (output), bi-direction (inout)
- **Implement in SystemVerilog with: (same as Verilog)**
  - `always`
  - `assign`

2- 3

Achieving User Logic Intent

- **Combinatorial Logic/Latches**
- **Meaning of full/parallel**
- **Registers**
- **State Machines**
- **Wildcard & Tri-state Logic**

2- 4

Issues with Verilog `always` Block: Simulation Mismatch

- **Incomplete sensitivity list**
  - Results in synthesis & simulation **mismatch**

Original Verilog code

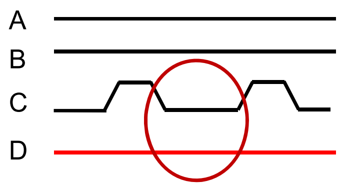
```
always @(A, B) begin
  D = (A & B) | C;
end
```

Missing C

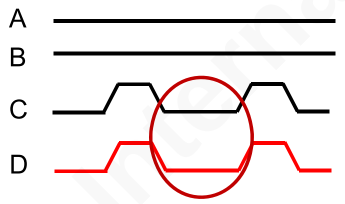
Synthesis view of original Verilog code

```
always @(A, B, C) begin
  D = (A & B) | C;
end
```

Pre-synthesis simulation



Post-synthesis simulation



2-5

Issues with Verilog `always` Block: Unintended Latch

- **Incomplete branch**
  - Results in unintended latch – both synthesis & simulation

Original Verilog code

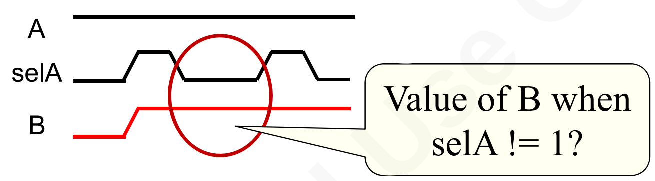
Missing `else`

```
always @(selA, A) begin
  if (selA) B = A;
end
```

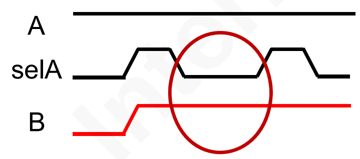
Synthesis view of original Verilog code

```
always @(selA, A) begin
  if (selA) B = A;
end
```

Pre-synthesis simulation



Post-synthesis simulation



2-6

SystemVerilog `always` Block

User can now specify design intent

- **`always_ff` models sequential logic**

```
always_ff @(posedge clk or negedge reset)
  if (!reset) q <= 0;
  else        q <= d;
```

- **`always_comb` models combinational logic**

```
always_comb
  if (!mode) y = a + b;
  else       y = a - b;
```

No sensitivity list **Executes at time zero**
Auto triggers at change of variables

- **`always_latch` models latch-based logic**

```
always_latch
  if (enable)
    q <= d;
```

No sensitivity list **Executes at time zero**
Auto triggers at change of variables

2-7

Unintentional Latch – Example #1 (Verilog & SystemVerilog)

- **Incomplete `if` branch specification**

Missing else

```
always @(a) begin
  if (a) begin
     b = 1'b1;
  end
end
```

```
============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|        b_reg         | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
============================================================================
Presto compilation completed successfully.
```

Missing else

```
always_comb begin
  if (a) begin
     b = 1'b1;
  end
end
```

```
============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|        b_reg         | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
============================================================================
Warning:  ./rtl/test.sv:3: Netlist for always_comb block contains a latch.
Presto compilation completed successfully.
```

2-8

Unintentional Latch – Solution for Example #1

- **Drive output under all cases**

```
always_comb begin
  b = 1'b0;
  if(a) begin
     b = 1'b1;
  end
end
```

```
always_comb begin
  if(a) begin
     b = 1'b1;
  end else begin
     b = 1'b0;
  end
end
```

```
Running PRESTO HDLC
Presto compilation completed successfully.
```

2-9

Unintentional Latch – Example #2 (Verilog)

- **Incomplete `if` branch specification**
  - With else statements

```
module case_latch (input logic [2:0] sel, din, output logic dout);
   always @(sel, din) begin // What if sel == 0 ?
      if (sel[0]) begin
         dout = din[0];
      end else begin
         if (sel[1]) begin
            dout = din[1];
         end else begin
            if (sel[2]) begin
               dout = din[2];
            end
         end
      end
   end
endmodule
```

Missing `else`

```
===============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       dout_reg       | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
===============================================================================
Presto compilation completed successfully.
Elaborated 1 design.
```

2-10

Unintentional Latch – Example #2 (SystemVerilog)

- **Incomplete `if` branch specification**
  - With else statements

```
module case_latch (input logic [2:0] sel, din, output logic dout);
   always_comb begin // What if sel == 0 ?
      if (sel[0]) begin
         dout = din[0];
      end else begin
         if (sel[1]) begin
            dout = din[1];
         end else begin
            if (sel[2]) begin
               dout = din[2];
            end



   end
endmod
```

Missing **`else`**

```
=======================================================================================
|     Register Name      | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=======================================================================================
|       dout_reg         | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
=======================================================================================
Warning:  ./rtl/test.sv:2: Netlist for always_comb block contains a latch (dout_reg).
Presto compilation completed successfully.
```

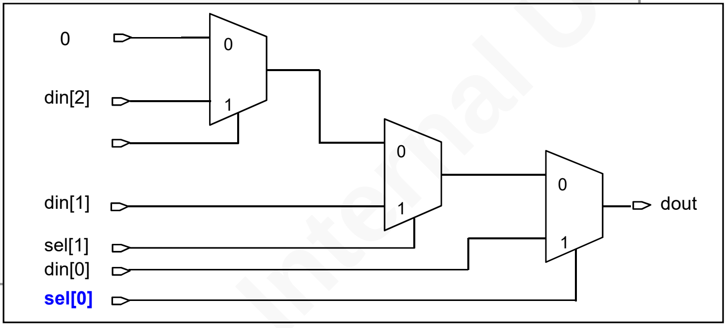
2- 11

Unintentional Latch – Solution for Example #2

- Drive output under all cases

```
module case_latch (input logic [2:0] sel, din, output logic dout);
   always_comb begin // What if sel == 0 ?
      dout = 1'b0;    // set output to known default value if sel == 0
      if (sel[0]) begin
         dout = din[0];
      end else begin
         if (sel[1]) begin
            dout = din[1];
         end else begin
            if (sel[2]) begin
               dout = din[2];
            end
         end
      end
   end
endmodule
```

Coding priority is from top to bottom



2- 12

Unintentional Latch – Avoid x as Solution for Example #2

- Drive output under all cases

```
module case_latch (input logic [2:0] sel, din, output logic dout);
   always_comb begin // What if sel == 0 ?
      dout = 1'b0;    // set output to known default value if sel == 0
//    dout = 1'bx;    // 'x is treated by DC as a don't care, BUT!
                      // 'x is treated by simulation as unknown and may
                      // cause a pre/post synthesis simulation mismatch.
      if (sel[0]) begin
         dout = din[0];
      end else begin
         if (sel[1]) begin
            dout = din[1];
         end else begin
            if (sel[2]) begin
               dout = din[2];
            end
         end
      end
   end
endmodule
```

See the next module on the meaning of `full` and `parallel` for an equivalent solution to `dout = 1'bx`

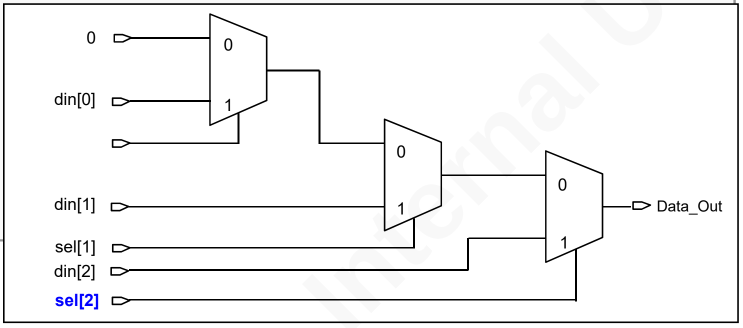
2- 13

Alternative if Statement Coding Style

- **Serialized if statements**

```
module case_latch (input logic [2:0] sel, din, output logic dout);
   always_comb begin // What if sel == 0 ?
      dout = 1'b0;    // set output to known default value if sel == 0
      if (sel[0]) begin
         dout = din[0];
      end
      if (sel[1]) begin
         dout = din[1];
      end
      if (sel[2]) begin
         dout = din[2];
      end
   end
endmodule
```

Coding priority is from bottom to top



Unintentional Latch – Example #4 (Verilog)

- **Incomplete `case` branch specification**
  - DC can tell you

```
module case_latch (input logic [2:0] din, output logic [2:0] dout);
  always @(din) begin
    case (din)  // Missing case for din == 6 and 7
      0,2,4     : dout[0] = 1'b1;
      1,3,5     : dout[1] = 1'b1;
    endcase
  end
endmodule
```

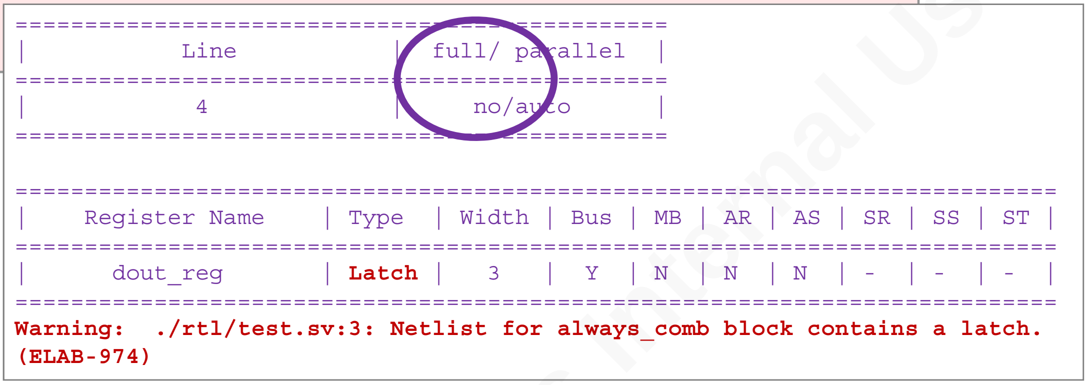
```
========================================================
|           Line            |  full/ parallel  |
========================================================
|            4              |     no/auto      |
========================================================

=============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|       dout_reg       | Latch |   3   |  Y  | N  | N  | N  | -  | -  | -  |
=============================================================================
Presto compilation completed successfully.
Elaborated 1 design.
```

2- 15

Unintentional Latch – Example #4 (SystemVerilog)

- **Incomplete `case` branch specification**
  - DC can tell you

```
module case_latch (input logic [2:0] din, output logic [2:0] dout);
  always_comb begin
    case (din)  // Missing case for din == 6 and 7
      0,2,4     : dout[0] = 1'b1;
      1,3,5     : dout[1] = 1'b1;
    endcase
  end
endmodule
```



```
========================================================
|           Line            |   full/ parallel   |
========================================================
|             4             |      no/auto        |
========================================================

=============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|       dout_reg       | Latch |   3   |  Y  | N  | N  | N  | -  | -  | -  |
=============================================================================
Warning:  ./rtl/test.sv:3: Netlist for always_comb block contains a latch.
(ELAB-974)
```

2- 16

Unintentional Latch – Example #4 (SystemVerilog)

- **Complete `case` branch specification**
  - Incomplete output specification

```
module case_latch (input logic [2:0] din, output logic [2:0] dout);
  always_comb begin
    case (din)
      0,2,4     : dout[0] = 1'b1;  // What about dout[1] & [2] ?
      1,3,5     : dout[1] = 1'b1;  // What about dout[0] & [2] ?
      default   : dout    = '0;    // takes care of all else
    endcase
  end
endmodule
```

```
========================================================
|           Line            |  full/ parallel  |
========================================================
|            4              |    auto/auto     |
========================================================

==============================================================================
|    Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|       dout_reg       | Latch |   3   |  Y  | N  | N  | N  | -  | -  | -  |
==============================================================================
Warning:  ./rtl/test.sv:3: Netlist for always_comb block contains a latch.
(ELAB-974)
```

?

2- 17

Unintentional Latch – Solution for Example #4

- **Drive all output under all cases**

```
module case_latch (input logic [2:0] din, output logic [2:0] dout);
  always_comb begin
    dout = '0;  // set output to default value
    case (din)
      0,2,4     : dout[0] = 1'b1;
      1,3,5     : dout[1] = 1'b1;
    endcase
  end
endmodule
```

```
================================================
|           Line           |  full/ parallel  |
================================================
|            5             |     no/auto      |
================================================
```

Coding priority is from top to bottom

Even without completing all possible “else” conditions, latches can be eliminated by driving all outputs by default at the beginning of the **`always_comb`** block

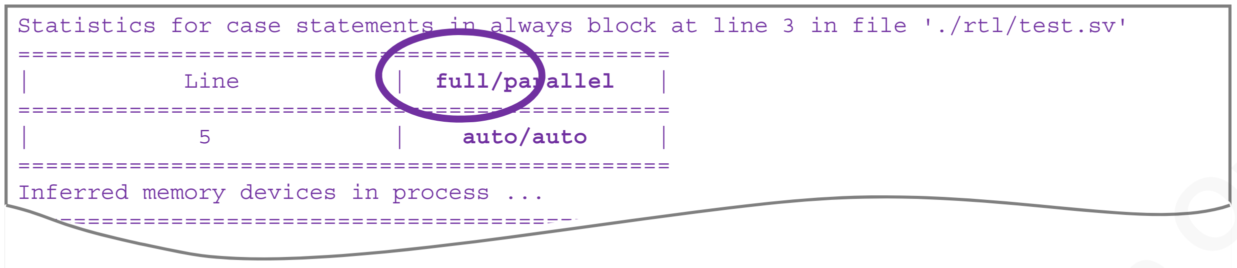
2- 18

Achieving User Logic Intent

- Combinatorial Logic/Latches
- Meaning of full/parallel
- Registers
- State Machines
- Wildcard & Tri-state Logic

2- 19

Meaning of `full`



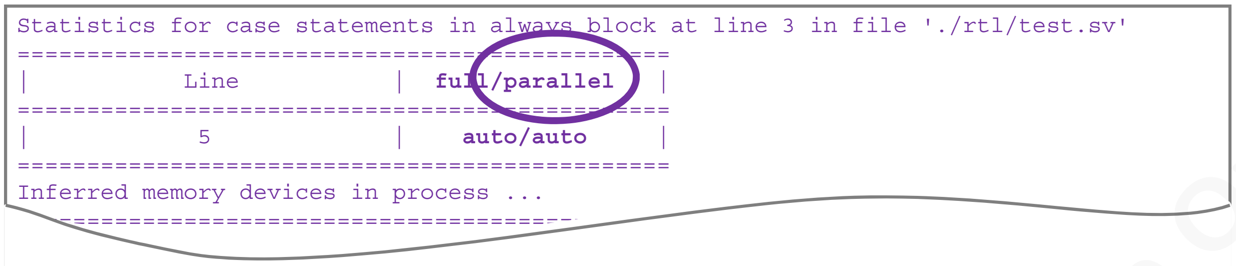
```
Statistics for case statements in always block at line 3 in file './rtl/test.sv'
===============================================
|            Line            |   full/parallel  |
===============================================
|             5              |     auto/auto    |
===============================================
Inferred memory devices in process ...
  ==========================================
```

- `full`: All <u>possible</u> case branches are coded

	full	parallel
auto	DC detected all case branches are coded (may prevent latch)	
no	DC detected <u>not</u> all case branches are coded (possible latch)	
user	User specified all <u>possible</u> case branches are coded (may prevent latch)	

2- 20

Meaning of `parallel`



```
Statistics for case statements in always block at line 3 in file './rtl/test.sv'
===============================================
|           Line           |  full/parallel  |
===============================================
|            5             |    auto/auto    |
===============================================
Inferred memory devices in process ...
    ========================================
```

- `full`: **All <u>possible</u> case branches are coded**
- `Parallel`: **Branches are <u>mutually exclusive</u> - priority logic not needed**

	full	parallel
auto	DC detected all case branches are coded (may prevent latch)	DC detects all branches are mutually exclusive (enables logic sharing optimization)
no	DC detected <u>not</u> all case branches are coded (possible latch)	DC does <u>not</u> detect all branches are mutually exclusive (priority logic may be required)
user	User specified all <u>possible</u> case branches are coded (may prevent latch)	User specifies all branches are mutually exclusive (enables logic sharing optimization)

2- 21

User Specified `full` Condition

```
Statistics for case statements in always block at line 3 in file './rtl/test.sv'
===============================================
|           Line           |  full/parallel   |
===============================================
|            5             |     user/        |
===============================================
Inferred memory devices in process ...
   ====================================
```

```
  always_comb begin
    dout = '0;
    priority case (din)
      0,2,4       : dout[0] = 1'b1;
      1,3,5       : dout[1] = 1'b1;
    endcase
  end
```

- fu… e coded
- Pa… xclusive - priority logic not needed

		parallel
auto	DC detected all case branches are coded (may prevent latch)	DC detects all branches are mutually exclusive (enables logic sharing optimization)
no	DC detected not all case branches are coded (possible latch)	DC does not detect all branches are mutually exclusive (priority logic may be required)
user	User specified all possible case branches are coded (may prevent latch)	User specifies all branches are mutually exclusive (enables logic sharing optimization)

2- 22

User Specified `parallel` Condition

```
Statistics for case statements in always block at line 3 in file './rtl/test.sv'
===============================================
|           Line           |  full/parallel   |
===============================================
|            5             |        /user     |
===============================================
Inferred memory devices in process ...
   ========================================
```

CAUTION!
requires **VCS 2017.12**
or later version

```
always_comb begin
  dout = '0;
  unique0 case (din)
    0,2,4     : dout[0] = 1'b1;
    1,3,5     : dout[1] = 1'b1;
  endcase
end
```

c not needed

	full	
auto	DC detected all case branche (may not prevent latch)	utually exclusive (enables logic sharing optimization)
no	DC detected not all case branches are coded (possible latch)	DC does not detect all branches are mutually exclusive (priority logic may be required)
user	User specified all possible case branches are coded (may not prevent latch)	User specifies all branches are mutually exclusive (enables logic sharing optimization)

2- 23

User Specified `full` and `parallel` Condition

```
Statistics for case statements in always block at line 3 in file './rtl/test.sv'
===============================================
|           Line           |  full/parallel   |
===============================================
|            5             |    user/user     |
===============================================
Inferred memory devices in process ...
    ====================================-
```

- **`full`: All possible case**
- **`Parallel`: Branches ar** **c not needed**

```
always_comb begin
  dout = '0;
  unique case (din)
    0,2,4     : dout[0] = 1'b1;
    1,3,5     : dout[1] = 1'b1;
  endcase
end
```

	full	
auto	DC detected all case branche (may not prevent latch)	utually exclusive (enables logic sharing optimization)
no	DC detected not all case branches are coded (possible latch)	DC does not detect all branches are mutually exclusive (priority logic may be required)
user	User specified all possible case branches are coded (may not prevent latch)	User specifies all branches are mutually exclusive (enables logic sharing optimization)

2- 24

Effects of Full and Parallel Examples

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated

```
always_comb begin

   case (1'b1)
      sel[0]   : dout = A && B;
      sel[1]   : dout = A || B;
      sel[2]   : dout = ^(A + B);
   endcase
end
```

What if `sel == '0`?

```
================================================================
|              Line              |   full / parallel  |
================================================================
|               3                |      no / no       |
================================================================
================================================================
|     Register Name     |  Type  | Width |  ...  |
================================================================
|       dout_reg        | Latch  |   1   |  ...  |
================================================================
```

2- 25

Effects of Full and Parallel Examples – One Solution

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **A possible solution:**
  - Drive all outputs before case statements

```
always_comb begin
   dout = '1;
   case (1'b1)
      sel[0]   : dout = A && B;
      sel[1]   : dout = A || B;
      sel[2]   : dout = ^(A + B);
   endcase
end
```

What if `sel == '0` is <u>not allowed</u> by design?

```
================================================
|              Line          |   full/ parallel  |
================================================
|              3             |       no/no       |
================================================
Presto compilation completed successfully.
```

2- 26

Effects of Full and Parallel Examples – Downside to Solution

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **A possible solution:**
  - Drive all outputs before case statements

```
always_comb begin
   dout = '1;
   case (1'b1)
      sel[0]
      sel[1]
      sel[2]
   endcase
end
```

Assigning **dout** to an arbitrary value may produce inefficient logic

What if **sel == '0** is not allowed by design?

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27

2- 27

Effects of Full and Parallel Examples – A Temptation

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **A possible solution:**
  - Drive all outputs before case statements

DO NOT USE X as DON'T CARE!!!

```
always_comb begin
   dout = 'x;
   case (1'b1)
      sel[0]
      sel[1]
      sel[2]
   endcase
end
```

One might be tempted to use **'x** as don't cares to get logic optimization

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27
Use 'x as don't care	40	-3.75	94.29	49

Can cause a lot of problems for simulation testbenches

2- 28

Effects of Full and Parallel Examples – Better Solution

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **A better solution:**
  - Set SystemVerilog `priority` on the `case` statement

```
always_comb begin
// dout = 'x;
   priority case (1'b1)
      sel[0]   : dout = A && B;
      sel[1]   : dout = A || B;
      sel[2]   : dout = ^(A + B);
   endcase
end
```

If `sel == '0`
is not allowed by design

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27
Use `'x` as don't care	40	-3.75	94.29	49

```
===========================================================
|              Line             |  full / parallel |
===========================================================
|               32              |   user     no    |
===========================================================
```

2-29

Effects of Full and Parallel Examples – Good for Simulation

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **A better solution:**
  - Set SystemVerilog `priority` on the `case` statement

```
always_comb begin
// dout = 'x;
   priority case (1'b1)
      sel[0]   : dout = A && B;
      sel[1]   : dout = A || B;
      sel[2]   : dout = ^(A + B);
   endcase
end
```

`sel == 0` will be reported as warning in simulation

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27
Use `'x` as don't care	40	-3.75	94.29	49
Use `priority`	40	-3.75	94.29	49

```
=============================================
|        Line         |  full / parallel  |
=============================================
|         32          |  user     no      |
=============================================
```

2-30

Effects of Full and Parallel Examples – Applying `unique`

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **Another possible solution:**
  - If each case statement is mutually exclusive with each other

```
always_comb begin
// dout = 'x;
   unique case (1'b1)
      sel[0]   : dout = A && B;
      sel[1]   : dout = A || B;
      sel[2]   : dout = ^(A + B);
   endcase
end
```

Specify `case` as `unique`

`$countones(sel) != 1` will be reported as warning in simulation

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27
Use `priority`	40	-3.75	94.29	49
Use `unique`	29	-3.68	61.50	38

```
===============================================
|        Line        |  full / parallel  |
===============================================
|         32         |    user / user    |
===============================================
```

2-31

Effects of Full and Parallel Examples

For synthesis

- **If all outputs are driven in all cases**
  - But, not all possible cases are specified, latches are generated
- **Another possible solution:**
  - If each case statement is mutually exclusive with each other

```
always_comb begin
   unique if (sel[0]==1'b1)
      dout = A && B;
   else if (se [1]==1'b1)
      dout = A      B;
   else if (sel    ==1'b1)
      dout = ^(A      );
end
```

Also works for **if** statements

Code Style	Cell Count	Timing (WNS)	Area	Net
Set default output	18	-5.15	49.30	27
Use **priority**	40	-3.75	94.29	49
Use **unique**	29	-3.68	61.50	38

```
=======================================================
|                                          arallel |
=====                                       ========
|                                           er     |
=======================================================
```

full/parallel condition is not reported for if statements

2- 32

Effects of Full and Parallel Examples

For synthesis

- If all c
  - But,
- Anoth
  - If ea

```
always_com
// dout =
   unique
      sel[
      sel[1]
      sel[2]     : dout = ^(A + B);
   endcase
end
```

			Area	Net
			49.30	27
			94.29	49
			61.50	38

```
================================================
|              Line            |  full / parallel |
================================================
|               32             |   user / user    |
================================================
```

CAUTION:

The effectiveness of `priority` and `unique` specifications are heavily dependent on logic operations being synthesized. There are no guarantees on the type of QoR that one might achieve with these switches. Try them yourself to see if they are appropriate for your design.

2- 33

Achieving User Logic Intent

- Combinatorial Logic/Latches
- Meaning of full/parallel
- Registers
- State Machines
- Wildcard & Tri-state Logic

2- 34

Register Synthesis

Registers are synthesized when:

- **ALL signals in the `always` event list have an associated edge:**
  - `posedge` *`clk`*, `negedge` *`reset_n`*
- **Coding recommendations:**
  - Use `always_ff` construct to synthesize register
  - Use Verilog non-blocking assignment (`<=`) for variable assignments inside the `always_ff` block
    - Otherwise, a simulation race condition may occur
    - And, simulation mismatch between pre-synthesis and post-synthesis can happen

2- 35

Register – Example #1 (Verilog and SystemVerilog)

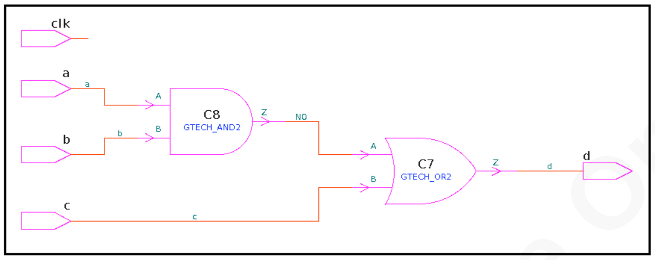
Where is the flip flop?

- **Verilog:**

Missing edge

```
always @(clk) begin
  d <= (a & b) | c;
end
```

```
Running PRESTO HDLC
Presto compilation completed successfully.
```



- **SystemVerilog:**

Missing edge

```
always_ff @(clk) begin
  d <= (a & b) | c;
end
```

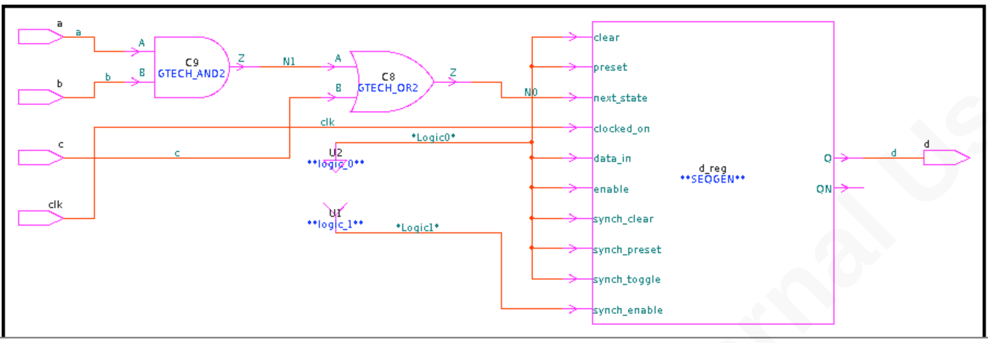
```
Running PRESTO HDLC
Warning:  ./rtl/ff.sv:2: Netlist for always_ff block does not contain a flip-flop.
(ELAB-976)
```

2- 36

Register – Solution for Example #1

- **Specify an edge for the clock**

```
always_ff @(posedge clk) begin
  d <= (a & b) | c;
end
```



```
===============================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        d_reg        | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
===============================================================================
```

Register – Example #2 (Verilog and SystemVerilog)

- **Intent: asynchronous reset register**

```
always @(posedge clk, rstN) begin
  if (!rstN)
    d <= 0;
  else
    d <= (a & b) | c;
end
```

Missing edge

```
always_ff @(posedge clk, rstN) begin
  if (!rstN)
    d <= 0;
  else
    d <= (a & b) | c;
end
```

Missing edge

- **Reported error:**

```
Running PRESTO HDLC
Error:  ./rtl/ff.sv:2: The event depends on both edge and nonedge expressions,
which synthesis does not support. (ELAB-91)
```

2- 38

Register – Solution for Example #2

- Specify an edge for the asynchronous control

```
always @(posedge clk, negedge rstN) begin
  if (!rstN)
    d <= 0;
  else
    d <= (a & b) | c;
end
```

```
always_ff @(posedge clk, negedge rstN) begin
  if (!rstN)
    d <= 0;
  else
    d <= (a & b) | c;
end
```

```
Running PRESTO HDLC
Presto compilation completed successfully.
```

```
=========================================================================
|    Register Name     |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|        d_reg         | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
=========================================================================
```

2- 39

Register – Example #3 (SystemVerilog)

- **Intent: synchronous set register**

```
always_ff @(posedge clk, negedge setN) begin
  if (!setN)
    d <= '1;   // 1 implies set
  else
    d <= (a & b) | c;
end
```

Unintentional edge

- **Not recognized as synchronous event**
  - A signal listed as an edge event, when used in the first `if` statement, is interpreted as an asynchronous event

```
=================================================================================
|    Register Name     |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|        d_reg         | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
=================================================================================
```

2-40

Register – Partial Solution for Example #3

- Intent: synchronous set register

```
always_ff @(posedge clk) begin
  if (!setN)
    d <= '1;   // 1 implies set
  else
    d <= (a & b) | c;
end
```

Remove synchronous signal from event list

- Not recognized as a synchronous implementation pin in a cell

```
===============================================================================
|    Register Name     |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        d_reg         | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
===============================================================================
```

Register – Solution for Example #3

- **Intent: <u>synchronous</u> set register**

```
// synopsys sync_set_reset "setN"
always_ff @(posedge clk) begin
  if (!setN)
    d <= '1;   // 1 implies set
  else
    d <= (a & b) | c;
end
```

Synopsys directive to use sync set/reset cells

Signal name must match

- **Will be mapped to a synchronous pin if technology library supports synchronous register cells**

```
===============================================================================
|    Register Name     |    Type     | Width | Bus | MB | AR | AS | SR | SS  | ST |
===============================================================================
|        d_reg         | Flip-flop |   1   |  N  | N  | N  | N  | N  | Y   | N  |
===============================================================================
```

2-42

Achieving User Logic Intent

Combinatorial Logic/Latches

Meaning of full/parallel

Registers

State Machines

Wildcard & Tri-state Logic

SystemVerilog Enumerated Variables (1/2)

- **Create enumerated data types:**

Enumerated data type

```
typedef enum [val_type] {named representation} type_e;
```

- *val_type* defaults to `int`

ASCII representation of value

INIT = 1

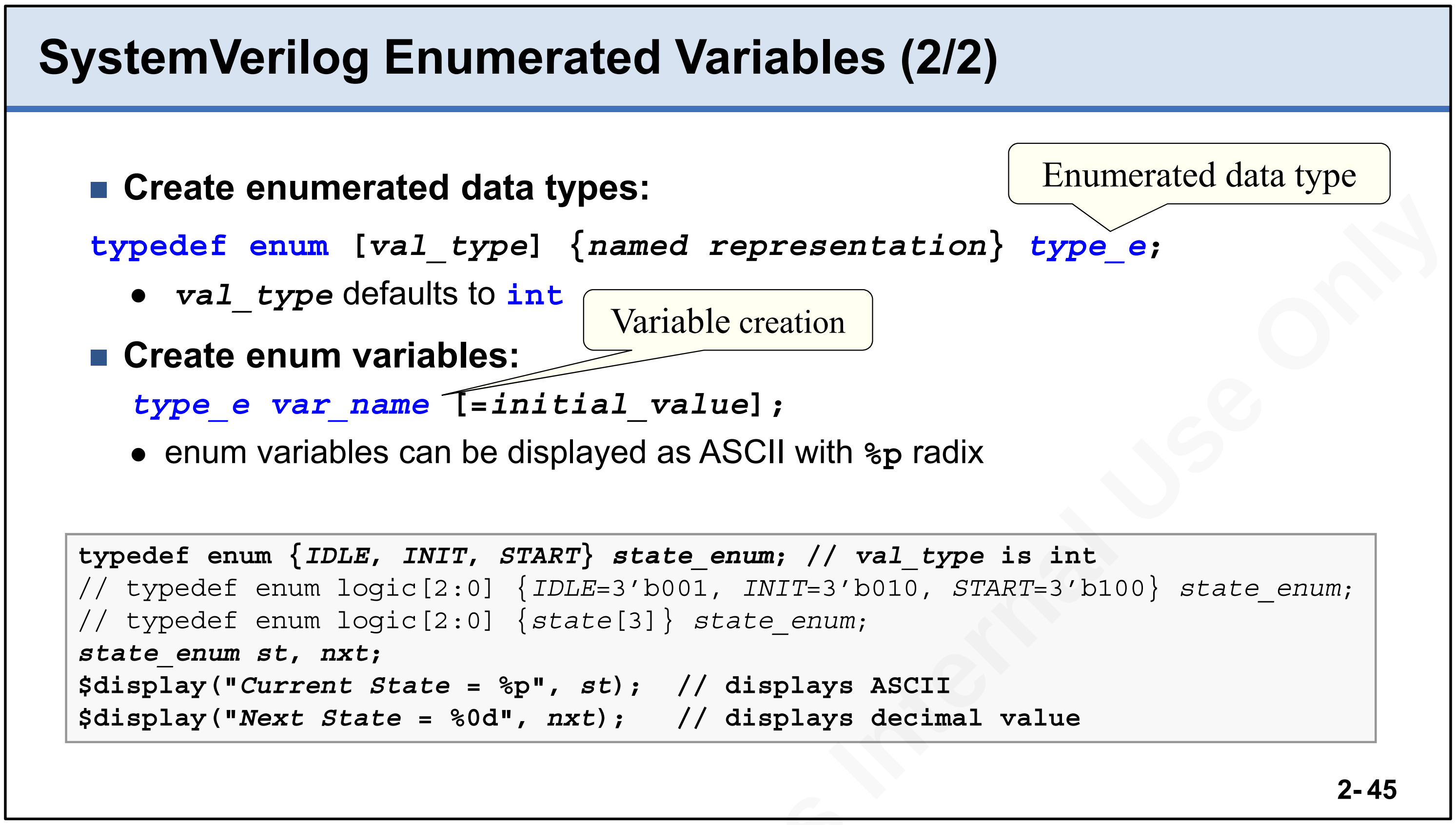
IDLE = 0

START = 2

```
typedef enum {IDLE, INIT, START} state_enum; // val_type is int
// typedef enum logic[2:0] {IDLE=3'b001, INIT=3'b010, START=3'b100} state_enum;
// typedef enum logic[2:0] {state[3]} state_enum;
```

User can specify value for the enumerated representation

2-44



SystemVerilog Enumerated Variables (2/2)

- **Create enumerated data types:**

Enumerated data type

```
typedef enum [val_type] {named representation} type_e;
```

  - *val_type* defaults to `int`

Variable creation

- **Create enum variables:**

```
type_e var_name [=initial_value];
```

  - enum variables can be displayed as ASCII with `%p` radix

```
typedef enum {IDLE, INIT, START} state_enum; // val_type is int
// typedef enum logic[2:0] {IDLE=3'b001, INIT=3'b010, START=3'b100} state_enum;
// typedef enum logic[2:0] {state[3]} state_enum;
state_enum st, nxt;
$display("Current State = %p", st);  // displays ASCII
$display("Next State = %0d", nxt);   // displays decimal value
```

2-45

Binary Encoded FSM Example (1/2)

Enum variable creation

Binary state coding style

State register

```
// Binary encode three machine states
typedef enum {IDLE, INIT, START} state_enum;
state_enum st, nxt;
always_comb begin // state transition logic
  case (st)
    IDLE:  if (select) nxt = INIT;
    INIT:  begin dout = 1'b1; nxt = START; end
    START: dout = 1'b0;
  endcase
end
always_ff @(posedge clk, negedge rstN) begin
  if (!rstN) st <= IDLE;
  else       st <= nxt;
end
```

Why 32 bits?

Why latches?

```
Inferred memory devices in process ...
===============================================================================
|     Register Name     | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        nxt_reg        | Latch |  32   |  Y  | N  | N  | N  | -  | -  | -  |
|       dout_reg        | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
===============================================================================
```

2-46

Binary Encoded FSM Example (2/2)

```
// Binary encode three machine states
typedef enum {IDLE, INIT, START} state_enum;
state_enum          nxt;
always                                  ansition logic
  case
    ID                                 INIT;
    INIT:  begin dout = 1'b1; nxt = START; end
    START: dout = 1'b0;
  endcase
nd
lways_ff @(posedge clk, negedge rstN) begin
  if (!rstN) st <= IDLE;
  else       st <= nxt;
end
```

Unspecified value data type defaults to **int**

Output not driven in all cases!

Why 32 bits?

Why latches?

```
Inferred memory devices in process ...
===========================================================================
           Name     | Type  | Width |  us | MB | AR | AS | SR | SS | ST |
===========================================================================
|      nxt_reg      | Latch |  32   |  Y  | N  | N  | N  | -  | -  | -  |
|     dout_reg      | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
===========================================================================
```

2-47

Binary Encoded FSM Solution

```
// Binary encode three machine states
typedef enum logic [1:0] {IDLE, INIT, START} state_enum;
state_enum st, nxt;
always_comb begin
  dout = 1'b0;
  nxt  = IDLE;
  case (st)
    IDLE:  if (select) nxt = INIT;
    INIT:  begin dout = 1'b1; nxt = START; end
    START: dout = 1'b0;
  endcase
end
always_ff @(posedge clk, negedge rstN) begin
  if (!rstN) st <= IDLE;
  else       st <= nxt;
end
```

Drive all outputs to avoid unintentional latch

Specify the desired number of bits

HDL Compiler creates a finite state machine inference report when you set **`hdlin_reporting_level`** to **`basic+fsm`**.

```
statistics for FSM inference:
  state register: st
  states
  ======
  IDLE:        00
  INIT:        01
  START:       10
  total number of states: 3
```

2-48

State Machine Coding Styles and Effects of Switches

- **Three common forms of coding styles for state machine:**
  - Binary
  - One-hot (binary case)
  - One-hot
- **`unique case` and `unique0 case` effects on these coding styles**

2-49

Binary Encoded FSM Example – QoR

```
typedef enum logic [2:0] {IDLE, INIT, START, WORK, END} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  case (st)
    IDLE:  if (en) nxt = INIT;
    INIT:  begin dout = ^a; nxt = START; end
    START: begin dout = ^(a+b); if (en) nxt = WORK; else nxt = END;
    WORK:  if (en) begin dout = ^(a+b); nxt = WORK; end else nxt = END;
    END:   dout = ^b;
  endcase
end
```

Binary **case**

```
=================================================
|          Line          |  full / parallel  |
=================================================
|          123           |     no / auto     |
=================================================
```

Set **hdlin_reporting_level** to **basic+fsm**

```
statistics for FSM inference:
  state register: st
  states
  ======
  IDLE:          000
  INIT:          001
  START:         010
  WORK:          011
  END:           100
  total number of states: 5
```

Binary

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary case	118	-4.33	329.88	140

```
end
```

2- 50

Binary Encoded FSM Example with `unique0 case` – QoR

```
typedef enum logic [2:0] {IDLE, INIT, START, WORK, END} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  unique0 case (st)
    IDLE:  if (en) nxt = INIT;
    INIT:  begin dout = ^a; nxt = START; end
    START: begin dout = ^(a+b); if (en) nxt = WORK; else
    WORK:  if (en) begin dout = ^(a+b); nxt = WORK; end e
    END:   dout = ^b;
  endcase
end
```

Apply **`unique0`**

```
==============================================
|          Line          |  full / parallel |
==============================================
|          123           |    no / user     |
==============================================
```

Parallel recognition changes from **`auto`** to **`user`**

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary **`case`**	118	-4.33	329.88	140
with **`unique0 case`**	118	-4.33	329.88	140

Results are identical!

`unique0` <u>does</u> help simulation to detect violation of mutually exclusive case!

2- 51

Binary Encoded FSM Example with `unique case` – QoR

```
typedef enum logic [2:0] {IDLE, INIT, START, WORK, END} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  unique case (st)
    IDLE:  if (en) nxt = INIT;
    INIT:  begin dout = ^a; nxt = START; end
    START: begin dout = ^(a+b); if (en) nxt = WORK; else nxt = E
    WORK:  if (en) begin dout = ^(a+b); nxt = WORK; end else nxt
    END:   dout = ^b;
  endcase
end
```

Apply **`unique`**

```
================================================
|          Line          |  full / parallel  |
================================================
|           123          |    user / user    |
================================================
```

Full recognition also changes to **`user`**

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary **`case`**	118	-4.33	329.88	140
with **`unique case`**	111	-4.38	315.91	133

Improvement in one QoR may comes at cost of another

`unique/unique0` can help simulation catch full/parallel violations

2- 52

An Alternative FSM Solution: One Hot (Binary `case`)

```
typedef enum logic[4:0]{IDLE='b00001,INIT='b00010,START='b00100,WORK='b01000,END='b10000} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  case (st)
    IDLE:  if (en) nxt = INIT;
    INIT:  begin dout = ^a; nxt = START; end
    START: begin dout = ^(a+b); if (en) nxt = WORK; else nxt = END;
    WORK:  if (en) begin dout = ^(a+b); nxt = WORK; end else nxt = END;
    END:   dout = ^b;
  endcase
end
always_ff @(posedge clk, negedge rstN) begin
```

Only one bit is high for all states

```
===============================================
|          Line          |  full / parallel  |
===============================================
|           123           |    no / auto     |
===============================================
```

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary `case`	118	-4.33	329.88	140
with `unique case`	111	-4.38	315.91	133
One Hot Binary `case`	107	-4.84	313.62	131

2- 53

An Alternative FSM Solution: One Hot (Binary `case`)

```
typedef enum logic[4:0]{IDLE='b00001,INIT='b00010,START='b00100,WORK='b01000,END='b10000} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  unique case (st)
    IDLE:  if (en) nxt = INIT;
    INIT:  begin dout = ^a; nxt = START; end
    START: begin dout = ^(a+b); if (en) nxt = WORK; else nxt = END;
    WORK:  if (en) begin dout = ^(a+b); nxt = WORK; end else nxt = END;
    END:   dout = ^b;
  endcase
end
always_ff @(posedge clk, negedge rstN) begin
```

Apply `unique`

```
===============================================
|         Line          |  full / parallel |
===============================================
|          123          |   user / user    |
===============================================
```

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary `case`	118	-4.33	329.88	140
with `unique case`	111	-4.38	315.91	133
One Hot Binary `case`	107	-4.84	313.62	131
One Hot Binary `unique case`	103	-4.56	303.96	124

2- 54

Another FSM Solution: One Hot (Binary One Hot case)

```
typedef enum logic[4:0]{IDLE='b00001,INIT='b00010,START='b00100,WORK='b01000,END='b10000} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  case (1'b1)
    st[0]:  if (en) nxt = INIT;
    st[1]:  begin dout = ^a; nxt = START; end
    st[2]:  begin dout = ^(a+b); if (en) nxt = WORK; else nxt = END;
    st[3]:  if (en) begin dout = ^(a+b); nxt = WORK; end else nxt = END;
    st[4]:  dout = ^b;
  endcase
```

case on "true" state

```
===============================================
|           Line           |  full / parallel |
===============================================
|            123           |     no / no      |
===============================================
```

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary `case`	118	-4.33	329.88	140
with `unique case`	111	-4.38	315.91	133
One Hot Binary `case`	107	-4.84	313.62	131
One Hot Binary `unique case`	103	-4.56	303.96	124
One Hot `case(1'b1)`	124	-4.33	354.79	146

2- 55

Another FSM Solution: One Hot (Binary One Hot `case`)

```
typedef enum logic[4:0]{IDLE='b00001,INIT='b00010,START='b00100,WORK='b01000,END='b10000} state_enum;
state_enum st, nxt;
always_comb begin
  dout = c;
  nxt  = IDLE;
  unique case (1'b1)
    st[0]:  if (en) nxt = INIT;
    st[1]:  begin dout = ^a; nxt = START; end
    st[2]:  begin dout = ^(a+b); if (en) nxt = WORK; else
    st[3]:  if (en) begin dout = ^(a+b); nxt = WORK; end e
    st[4]:  dout = ^b;
```

Apply **`unique`**

```
================================================
|            Line            |  full / parallel |
================================================
|            123             |   user / user    |
================================================
```

Full and parallel both changes to **`user`**

Code Style	Cell Count	Timing (WNS)	Area	Net
Binary `case`	118	-4.33	329.88	140
with `unique case`	111	-4.38	315.91	133
One Hot Binary `case`	107	-4.84	313.62	131
One Hot Binary `unique case`	103	-4.56	303.96	124
One Hot `case(1'b1)`	124	-4.33	354.79	146
One Hot `unique case(1'b1)`	103	-4.56	303.96	124

2- 56

Another FSM Solution: One Hot (Binary One Hot `case`)

```
typedef enum logic[4:0]{IDLE='b00001,INIT='b00010,START='b00100,WORK='b01000,END='b10000} state_enum;
state_enum st
always_comb
  dout  = c;
  nxt   = ID
  unique ca
    st[0]:
    st[1]:
    st[2]:
    st[3]:
    st[4]:
```

```
========
rallel |
========
ser    |
========
```

oth
r

CAUTION:

The effectiveness of `priority` and `unique` specifications are heavily dependent on logic operations being synthesized. There are no guarantees on the type of QoR that one might achieve with these switches. Try them yourself to see if they are appropriate for your design.

Code				
Binary				
with `unique case`	111	-4.38	315.91	133
One Hot Binary `case`	107	-4.84	313.62	131
One Hot Binary `unique case`	103	-4.56	303.96	124
One Hot `case(1'b1)`	124	-4.33	354.79	146
One Hot `unique case(1'b1)`	103	-4.56	303.96	124

2- 57

Achieving User Logic Intent

Combinatorial Logic/Latches

Meaning of full/parallel

Registers

State Machines

Wildcard & Tri-state Logic

2- 58

Unknown (x) Caveat

For simulation

- **x is defined to be unknown NOT don't care!**

Logical equality (==) comparison views x as an unknown and the result is unknown. unknown is not true.

```
if (In_A[7:0] == 8'b00xx11xx)
   begin
     Data_Out = 1'b1;
   end
else
   begin
     Data_Out = 1'b0;
   end
```

Simulation result

```
Data_Out = 1'b0;
```

Unknown (x) Caveat

For synthesis

- **x is also treated as unknown NOT don't care!**

Synthesis also views any comparison with **x** as false.
Consequently, synthesized hardware agrees with simulation result.

```
if (In_A[7:0] == 8'b00xx11xx)
   begin
     Data_Out = 1'b1;
   end
else
   begin
     Data_Out = 1'b0;
   end
```

Synthesis result

```
Data_Out = 1'b0;
```

SystemVerilog Equality (`==?`) & Inequality (`!=?`) Operators

SystemVerilog comparison operators supports wildcard

- **`==?` and `!=?` (new in SystemVerilog)**
  - Allows don't–care bits to be masked from the comparison
- **Treats `x`, `z` and `?` in the right operand, as wildcards and are not compared to the left operand**
- **Does not treat `X` and `Z` on the left hand side as wildcards!** ⚠

Right operand must be a constant

```
logic [3:0] din;
assign dout = (din ==? 4'b???0);
// assign dout = ~din[0] // only bit 0 is considered
```

2- 61

SystemVerilog `case inside` Wildcard Statement

- **`case` statement supports set membership through `inside` keyword**
  - Wildcard (`?`) and range (`[low:high]`) are supported

```
logic [2:0] status;
always_comb begin
  case (status) inside
    1, 3          : task1(); // matches 'b001 and 'b011
    3'b0?0, [4:7] : task2(); // matches 'b000 'b010 'b0x0 'b0z0
                             //         'b100 'b101 'b110 'b111
    // case fails all other values including 'b00x 'b01x 'bxxx
  endcase
end
```

2- 62

`casex` and `casez` Statements (Verilog)

Legacy Verilog `case` wildcard

- **`casex` allows "`x`", "`z`", "`?`" to be treated as "don't care" in case items**
- **`casez` allows "`z`", "`?`" to be treated as "don't care" in case items**
- **Synthesis does not support wildcard in `casex` and `casez` expressions**
  - `casex(3'b?0?)` // Not supported by Synthesis
  - `casez(3'b?0?)` // Not supported by Synthesis

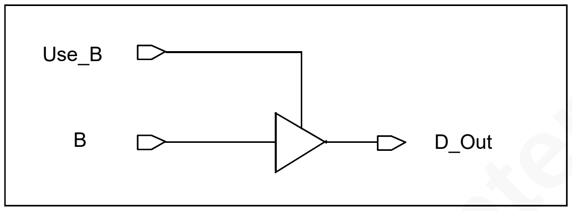
```
casex (status)
  3'b0?0 : task2(); // matches 'b000 'b010 'b0x0 'b0z0
endcase
casez (status)
  3'b0?0 : task2(); // matches 'b000 'b010 'b0z0
endcase
casex (3'b?0?)      // Not supported by synthesis
casez (3'b?0?)      // Not supported by synthesis
```

2- 63

Synthesis of a Tri-state Gate

- **Tri-state signal must be a net (`wire`) type**
- **Tri-state signal must be driven in a continuous assignment statement**
  - Conditional assignment of `'z` implies the synthesis of a tri-state gate

```
wire logic D_Out;
assign D_Out = (Use_B) ? B : 'z;
```



2- 64

Unit Objectives

After completing this unit, you should be able to:

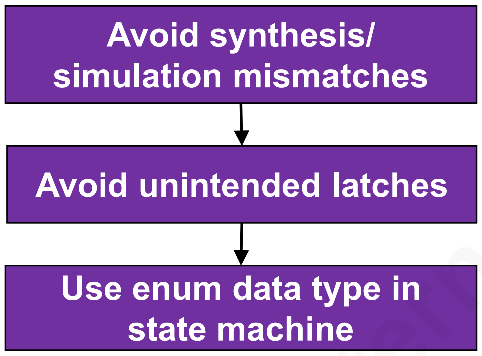
- Write RTL code for combinatorial logic
- Avoid unintended latch
- Avoid synthesis/simulation mismatch
- Create registers with synchronous/asynchronous reset
- Understand the meaning of full and parallel
- Use `enum` data type for state machines

2- 65

Lab 1: RTL Logic Intent

30 min

Getting intended logic from Synthesis


Avoid synthesis/ simulation mismatches
Avoid unintended latches
Use enum data type in state machine

2- 66

Appendix

Disabling unique/unique0/priority warning at time 0

Disabling `unique/unique0/priority` Warnings at Time 0

- **SystemVerilog `unique/unique0/priority` key words serves different purposes for synthesis and simulation**
- **For synthesis, these key words allow synthesis tools to perform synthesis optimization**
- **For simulation, these key words serve as assertions to catch problems during simulation**
  - But, there is a downside for simulation at time 0
  - At time 0, variables are typically at the default value of **`'x`**, which causes all statements with these key words to issue a warning
  - A typical solution is to disable the assertion at time 0, and reenable them after reset

2- 68

`unique/unique0/priority` Warnings at Time 0

- SystemVerilog `unique/unique0/priority` key words causes simulation time 0 warning

```
module unique_case(input clk, rst_n, a, b, c, output logic[1:0] d);
  logic[1:0] temp;
  always_comb begin
    unique case (1'b1) // will issue warning at time 0
      a: temp = 1;
      b: temp = 2;
      c: temp = 3;
    endcase
  end
  always_ff @(posedge clk, negedge rst_n) begin
    if (!rst_n) begin d <= 0;    end
    else        begin d <= temp; end
  end
endmodule
```

2-69

An example of the warning:

```
Warning-[RT-NCMUCS] No condition matches in statement
./test.sv, 55
  No condition matches in 'unique case' statement. 'default' specification is
  missing, inside top.dut, at time 0ps.
```

unique/unique0/priority Warnings Solution

- **Solution is to turn assertion off at time 0, and reenable after reset:**

```
// $assertcontrol(control[,[assert_type][,directive_type][,[levels][,list_scopes]] ]])
let ON = 3; let OFF = 4;
let UNIQUE = 32; let UNIQUE0 = 64; let PRIORITY = 128; // assert_type
let ASSERT = 1; // directive_types
let ALL_LVLS = 0; let THIS_LVL = 1; // 0 (all levels).  1 (scope of assertion)

module top;
  unique_case dut(.*);
  initial begin
    $assertcontrol(OFF, UNIQUE | UNIQUE0 | PRIORITY , ASSERT, ALL_LVLS, dut);
    // choose when you want them to be turned on
    @(negedge rst_n);
    $assertcontrol(ON, UNIQUE | UNIQUE0 | PRIORITY, ASSERT, ALL_LVLS, dut);
  end
endmodule
```

2-70

The assertion control would show in the log file like the following:

```
Stopping Unique/Priority checks at time 0ps : Level = 0 arg = top.dut (Source - ./test.sv,100)
Starting Unique/Priority checks at time 110000ps : Level = 0 arg = top.dut (Source - ./test.sv,124)
```

Agenda

3- 1

Unit Objectives

After completing this unit, you should be able to:

- Describe the difference between `packed` and `unpacked` array
- Use `struct` to create encapsulated data structure
- Use `interface` to simplify module connectivity
- Perform bottom-up synthesis on modules with `interface`

3- 2

Advanced SystemVerilog Features

Pack/Unpack Array

Struct/Union

Interface

Package

3- 3

SystemVerilog Enhancement - Array

Verilog:

RTL & simulation

```
reg [7:0] d [0:127];
```

Declaration of array

SystemVerilog:

RTL & simulation

```
logic [7:0] d [128];
logic [7:0] d [0:127]; // same as logic [7:0] d [128]
```

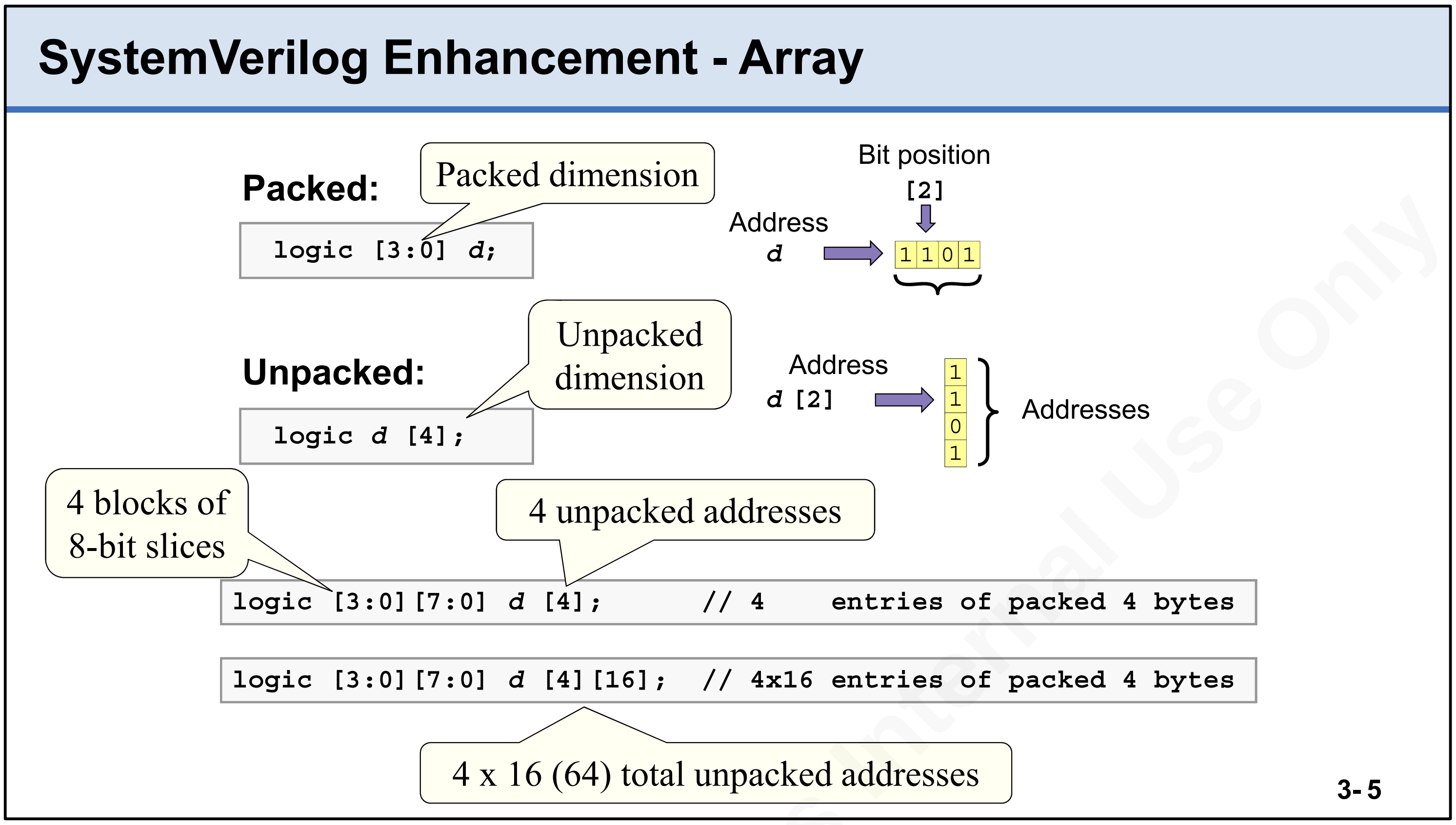
simulation only

```
bit [7:0] d [];           // dynamic
bit [7:0] d [$];          // queue
bit [7:0] d [data_type];  // associative
```

Packed array range

Unpacked array size

3- 4



SystemVerilog Enhancement - Array

Packed:

```
logic [3:0] d;
```

Packed dimension
Bit position
[2]
Address
d
1 1 0 1

Unpacked:

```
logic d [4];
```

Unpacked dimension
Address
d [2]
1
1
0
1
Addresses

4 blocks of 8-bit slices
4 unpacked addresses

```
logic [3:0][7:0] d [4];       // 4    entries of packed 4 bytes
```

```
logic [3:0][7:0] d [4][16];   // 4x16 entries of packed 4 bytes
```

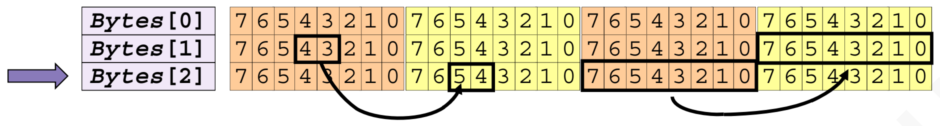
4 x 16 (64) total unpacked addresses

3- 5

Array Example

```
logic [3:0][7:0] Bytes [3];   // 3 entries of packed 4 bytes
```

- **Examples of array assignment:**



```
Bytes[2][2][5:4] = Bytes[1][3][4:3];
```

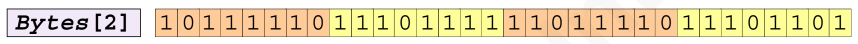
Packed bit/bit slice assignment

Packed assignment

```
Bytes[1][0] = Bytes[2][1];
```

```
Bytes[2] = 32'hbeef_deed;
```

Unpacked assignment



3- 6

Advanced SystemVerilog Features

Pack/Unpack Array

Struct/Union

Interface

Package

SystemVerilog `struct` Data type

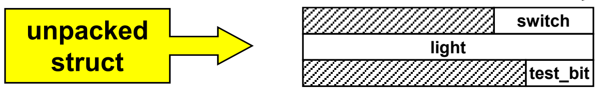
```
typedef enum logic[1:0] {OFF = 2'd0, ON = 2'd3} switch_val_enum;
typedef enum            {RED, GREEN, BLUE}      colors_enum;

typedef struct {
  switch_val_enum   switch;   // 2  bits
  colors_enum       light;    // 32 bits
  logic             test_bit; // 1  bit
} sw_lgt_pair_unpacked_struct;

typedef struct packed {
  switch_val_enum   switch;   // 2  bits
  colors_enum       light;    // 32 bits
  logic             test_bit; // 1  bit
} sw_lgt_pair_packed_struct;

sw_lgt_pair_packed_struct slp_up;
```

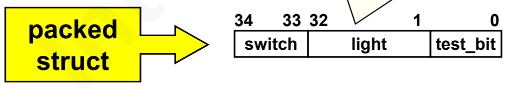
Creates an unpacked structure

Creates a packed structure

Treated as three distinct words

Can be accessed separately or as a single 35-bit word



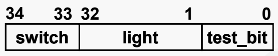


3- 8

Packed `struct` Example

```
typedef enum logic[1:0] {OFF = 2'd0, ON = 2'd3} switch_val_enum;
typedef enum            {RED, GREEN, BLUE}      color_enum;

typedef struct packed {
  switch_val_enum   switch;   // 2  bits
  colors_enum       light;    // 32 bits
  logic             test_bit; // 1  bit
} sw_lgt_pair_packed_struct;

sw_lgt_pair_packed_struct slp;

slp.light = GREEN;
slp = '1;
slp = '{switch:ON, light:RED, test_bit:1'b0};    // similar to .reference
// slp = '{test_bit:1'b0, switch:ON, light:RED}; // position independent
```

packed struct



Entire set of **`struct`** members can be assigned via member name

Note the additional '

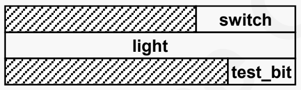
All members must be listed

3- 9

Un-Packed struct Example

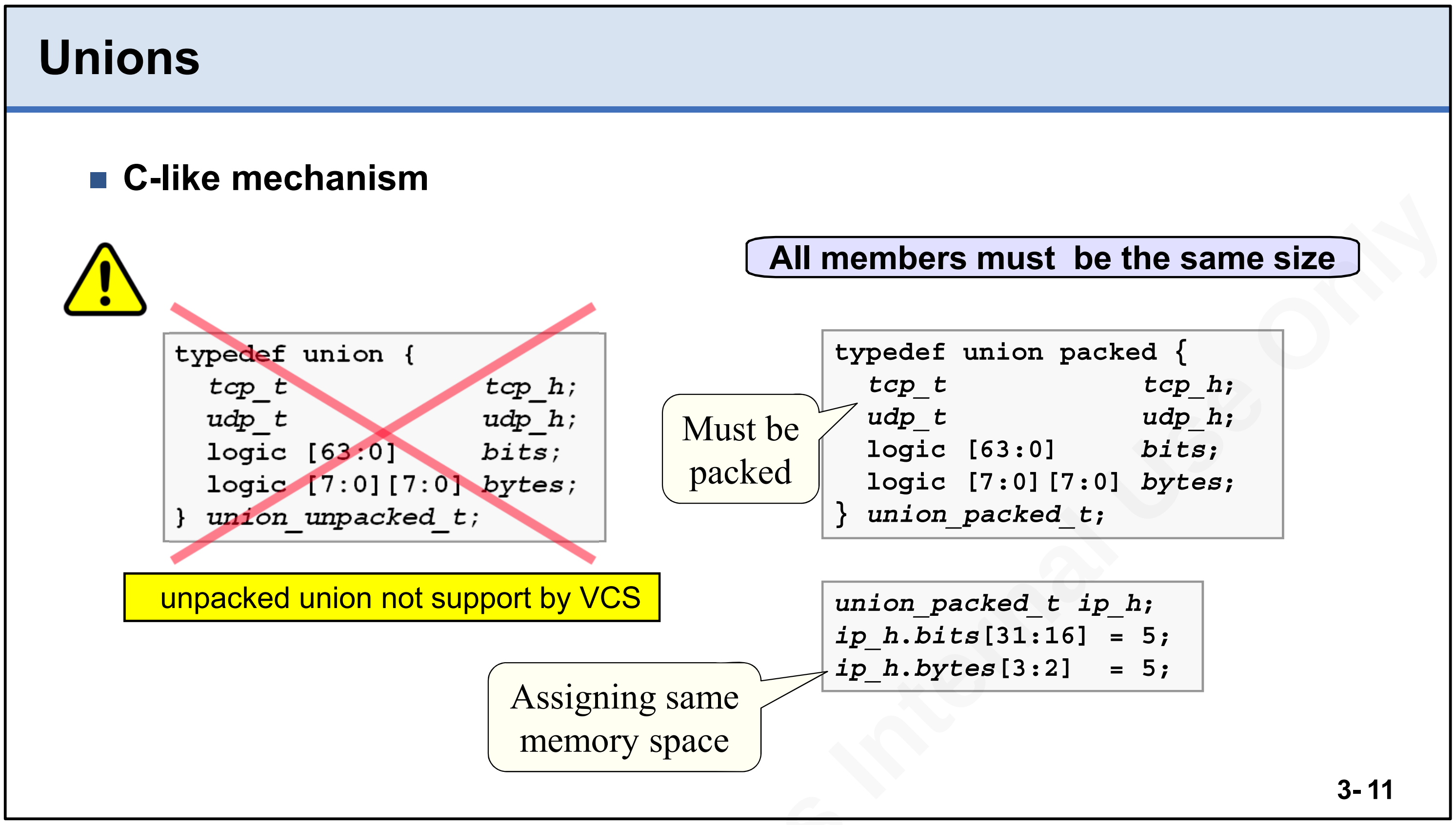
```
typedef enum logic[1:0] {OFF = 2'd0, ON = 2'd3} switch_val_enum;
typedef enum            {RED, GREEN, BLUE}      colors_enum;

typedef struct {
  switch_val_enum   switch;   // 2  bits
  colors_enum       light;    // 32 bits
  logic             test_bit; // 1  bit
} sw_lgt_pair_unpacked_struct;

sw_lgt_pair_unpacked_struct slp;

slp.light = GREEN;
~~slp = '0;~~
slp = '{switch:ON, light:RED, test_bit:1'b0};    // similar to .reference
// slp = '{test_bit:1'b0, switch:ON, light:RED}; // position independent
```

unpacked struct



Each member of packed **struct** can be assigned with a value

Set assignment works exactly the same as packed **struct**



Unions

- C-like mechanism

All members must be the same size

```
typedef union {
  tcp_t              tcp_h;
  udp_t              udp_h;
  logic  [63:0]      bits;
  logic  [7:0][7:0]  bytes;
} union_unpacked_t;
```

unpacked union not support by VCS

Must be packed

```
typedef union packed {
  tcp_t              tcp_h;
  udp_t              udp_h;
  logic  [63:0]      bits;
  logic  [7:0][7:0]  bytes;
} union_packed_t;
```

```
union_packed_t ip_h;
ip_h.bits[31:16]   = 5;
ip_h.bytes[3:2]    = 5;
```

Assigning same memory space

3- 11

Packed v/s Unpacked Structs/Union/Array

- **For simulation, unpacked structure offers better memory footprint and runtime performance**
  - Allows strong type checking
    - Operation on the entire unpacked structure is <u>illegal</u>
  - Can contain packed structure
  - But, VCS does not support unpacked union! ⚠
- **Packed structure allows for greater flexibility**
  - Multiple views possible and allowed
    - Operation on the entire packed structure is legal
  - Additional memory/runtime overhead to track both views
  - Cannot contain unpacked structure
- **Synthesis quality of Result (QoR)**
  - In general no synthesis QoR differences with packed/unpacked

3- 12

Advanced SystemVerilog Features

Pack/Unpack Array

Struct/Union

Interface

Package

3- 13

What Is An Interface?

- **Encapsulates signals like a struct encapsulates data**

At the simplest level an interface for a hardware wire is similar to what a struct is to a variable

```
typedef struct {
  int          i;
  logic [7:0]  a;
} s_type_struct;
```

```
interface intf();
  logic [2:0] sel;
  logic [7:0] bus;
endinterface
```

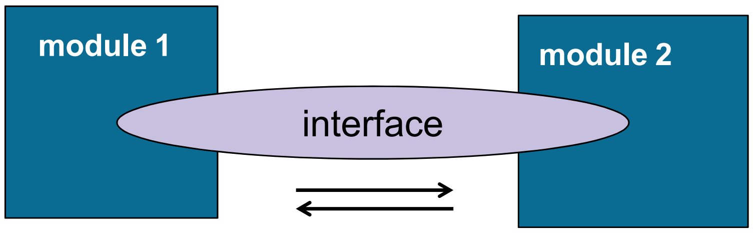
```
s_type_struct if1;

if1.a = 10;
```

```
intf if1();

modA a (w, if1);
  logic val;
  assign val = if1.bus[if1.sel];
endmodule
```

3- 14

Communication Object

- An interface provides communication between modules



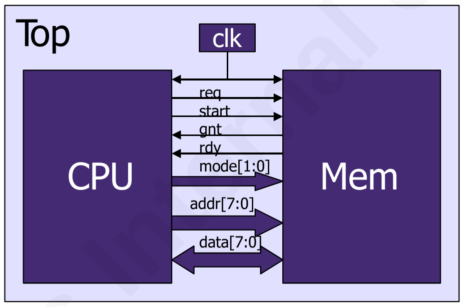
- The interface contains all the signals used in communication between module1 and module2
- The interface can also describe how the data is sent and received

Simple Example Without Interfaces

```
module memMod(input  logic      req,
                     logic      clk,
                     logic      start,
                     logic[1:0] mode,
                     logic[7:0] addr,
              inout  logic[7:0] data,
              output logic      gnt,
                     logic      rdy);
  ...
endmodule

module cpuMod(input  logic      clk,
                     logic      gnt,
                     logic      rdy,
              inout  logic[7:0] data,
              output logic      req,
                     logic      start,
                     logic[7:0] addr,
                     logic[1:0] mode);
  ...
endmodule
```

```
module top;
  logic req, gnt, start, rdy;
  logic clk;
  logic [1:0] mode;
  logic [7:0] addr, data;

memMod mem(req, clk, start, mode,
           addr, data, gnt, rdy);
cpuMod cpu(clk, gnt, rdy, data, req,
           start, addr, mode);
endmodule
```

Complex connectivity



3- 16

Simple Example Using Interfaces

```
interface simple_bus();
  logic        req, gnt;
  logic [7:0]  addr, data;
  logic [1:0]  mode;
  logic        start, rdy;
endinterface

module memMod(simple_bus  a,
              input logic clk);
  logic avail;
  always @(posedge clk)
    a.gnt <= a.req & avail;
endmodule

module cpuMod(simple_bus  b,
              input logic clk);
endmodule
```

Bundle signals in interface

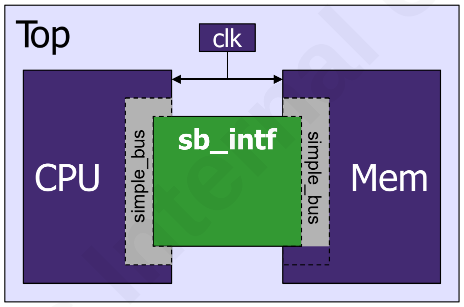
Use in port list

Refer to signals

```
module top;
  logic clk;
  simple_bus sb_intf();

  memMod mem(sb_intf, clk);
  cpuMod cpu(.b(sb_intf), .clk(clk));
endmodule
```

Instantiate interface

Connect via interface instance



3- 17

Using `modports` In Interface

```
interface simple_bus();
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
  modport slave  (input  req, addr, mode,
                         start,
                  output gnt, rdy,
                  inout  data);
  modport master (input  gnt, rdy,
                  output req, addr, mode, start,
                  inout  data);
endinterface

module memMod(input logic clk, simple_bus.slave a);
  ...
endmodule


module cpuMod(input logic clk, simple_bus.master b);
endmodule
```

Specify signal direction

Enforce modport rules for connectivity error checks

```
module top;
  bit clk;
  simple_bus sb_intf();

  memMod mem(clk, sb_intf);
  cpuMod cpu(clk, sb_intf);
endmodule
```

modport not needed at connection

3- 18

Using `modports` and Embedded Function

```
interface chip_int ();
  logic       in_d;
  logic       in_p;
  logic [7:0] out_d;
  logic       out_p;
  logic [7:0] into_fifo, from_fifo;
  logic       r_err;
  modport outo(input  from_fifo,
               output out_d, out_p,
               import function parity);
  modport into(input  in_d, in_p,
               output into_fifo, r_err,
               import function perr);
  function automatic logic parity (logic [7:0] data);
    parity = ^data;
  endfunction
  function automatic logic perr (logic [7:0] data, logic par);
    perr = ~^{par, data};
  endfunction
endinterface
```

Import functions

implement functions

```
module in_stage (input logic clk,
                 chip_int.into ci);
  always_ff @(posedge clk)
    err <= ci.perr(ci.in_p, ci.in_d);
endmodule
module out_stage (input logic clk,
                  chip_int.outo co);
  always_ff @(posedge clk)
    p <= co.parity(co.from_fifo);
endmodule
```

Calling the functions

3- 19

Synthesis Tool Support of `interface`

- **Synthesis tool changes the module port list**
- **RTL:**

```
interface fifo_io #(WIDTH=8) ();
  logic rd_n, wr_n, empty, full;
  logic [WIDTH-1:0] din, dout;
  modport fifo(input rd_n, wr_n, din, output empty, full, dout);
endinterface
module fifo#(WIDTH=8,BUF_SIZE=16)(input clk,reset_n,fifo_io.fifo fifo_if);
```

- **Gate:**

Synthesize with:

```
analyze -format sverilog { fifo_io.sv fifo.sv }
elaborate fifo # using default parameter values
```

```
module fifo (clk, reset_n, \fifo_if.rd_n,  \fifo_if.wr_n, \fifo_if.din,
                           \fifo_if.empty, \fifo_if.full , \fifo_if.dout);
```

  - Interface and other non-integrals are renamed

3- 20

Synthesis Tool Support of SystemVerilog

- **Gets a lot more complex if using non-default parameter values**

```
module fifo#(WIDTH=8, BUF_SIZE=16)(input clk, reset_n, fifo_io.fifo fifo_if);
```

Synthesize with:

```
analyze -format sverilog { fifo_io.sv fifo.sv }
elaborate fifo -parameter "WIDTH=4, BUF_SIZE=8"
```

```
module fifo_WIDTH4_BUF_SIZE8 ( clk, reset_n, \fifo_if.rd_n, \fifo_if.wr_n,
                \fifo_if.din, \fifo_if.empty, \fifo_if.full, \fifo_if.dout  );
```

- **Not only**
  - Interface and other non-integrals are renamed
- **The module name also changes!** ⚠
  - This creates problems for block integration and simulation

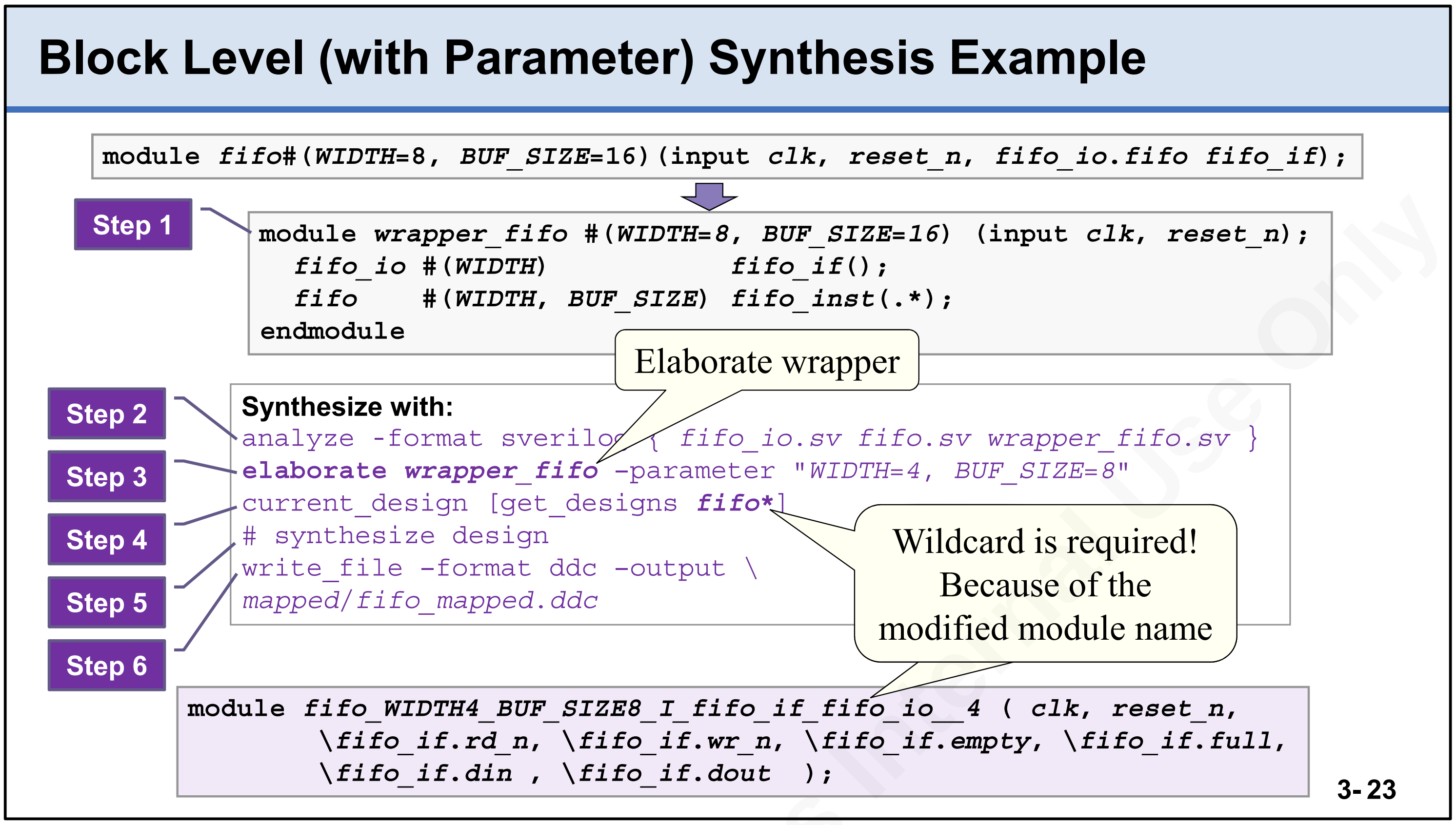
3- 21

Parameter Synthesis – At Block Level

From bottom-up synthesis and gate-level verification, the block level RTL code synthesis need to execute the following steps

- Step 1: Create a SystemVerilog wrapper module for the design
- Step 2: Analyze the SystemVerilog RTL and the wrapper modules
- Step 3: Elaborate the wrapper module
- Step 4: Set `current_design` to the target design
- Step 5: Proceed with the synthesis flow
- Step 6: Save the design by using the write command

3- 22



Block Level (with Parameter) Synthesis Example

```
module fifo#(WIDTH=8, BUF_SIZE=16)(input clk, reset_n, fifo_io.fifo fifo_if);
```

Step 1

```
module wrapper_fifo #(WIDTH=8, BUF_SIZE=16) (input clk, reset_n);
  fifo_io #(WIDTH)           fifo_if();
  fifo    #(WIDTH, BUF_SIZE) fifo_inst(.*);
endmodule
```

Elaborate wrapper

Step 2
Step 3
Step 4
Step 5
Step 6

Synthesize with:

```
analyze -format sverilog { fifo_io.sv fifo.sv wrapper_fifo.sv }
elaborate wrapper_fifo -parameter "WIDTH=4, BUF_SIZE=8"
current_design [get_designs fifo*]
# synthesize design
write_file -format ddc -output \
mapped/fifo_mapped.ddc
```

Wildcard is required! Because of the modified module name

```
module fifo_WIDTH4_BUF_SIZE8_I_fifo_if_fifo_io__4 ( clk, reset_n,
       \fifo_if.rd_n, \fifo_if.wr_n, \fifo_if.empty, \fifo_if.full,
       \fifo_if.din , \fifo_if.dout  );
```

3- 23

Integrating Synthesized Block Level Netlist at Top Level

- **Step T1:** Read the complete RTL design with analyze command
- **Step T2:** Elaborate the top-level design
- **Step T3:** Remove the low-level RTL design
- **Step T4:** Replace with with the synthesized version
- **Step T5:** Continue with normal synthesis

Synthesize with:

```
analyze -format sverilog { fifo_io.sv fifo.sv top.sv ... }
elaborate top
remove_design [get_designs fifo*]
read_ddc fifo_mapped.ddc
current_design top
link
# synthesize design
# save design
```

Step T1

Step T2

Step T3

Step T4

Step T5

3- 24

Simulating at Gate Level for the Synthesized Block

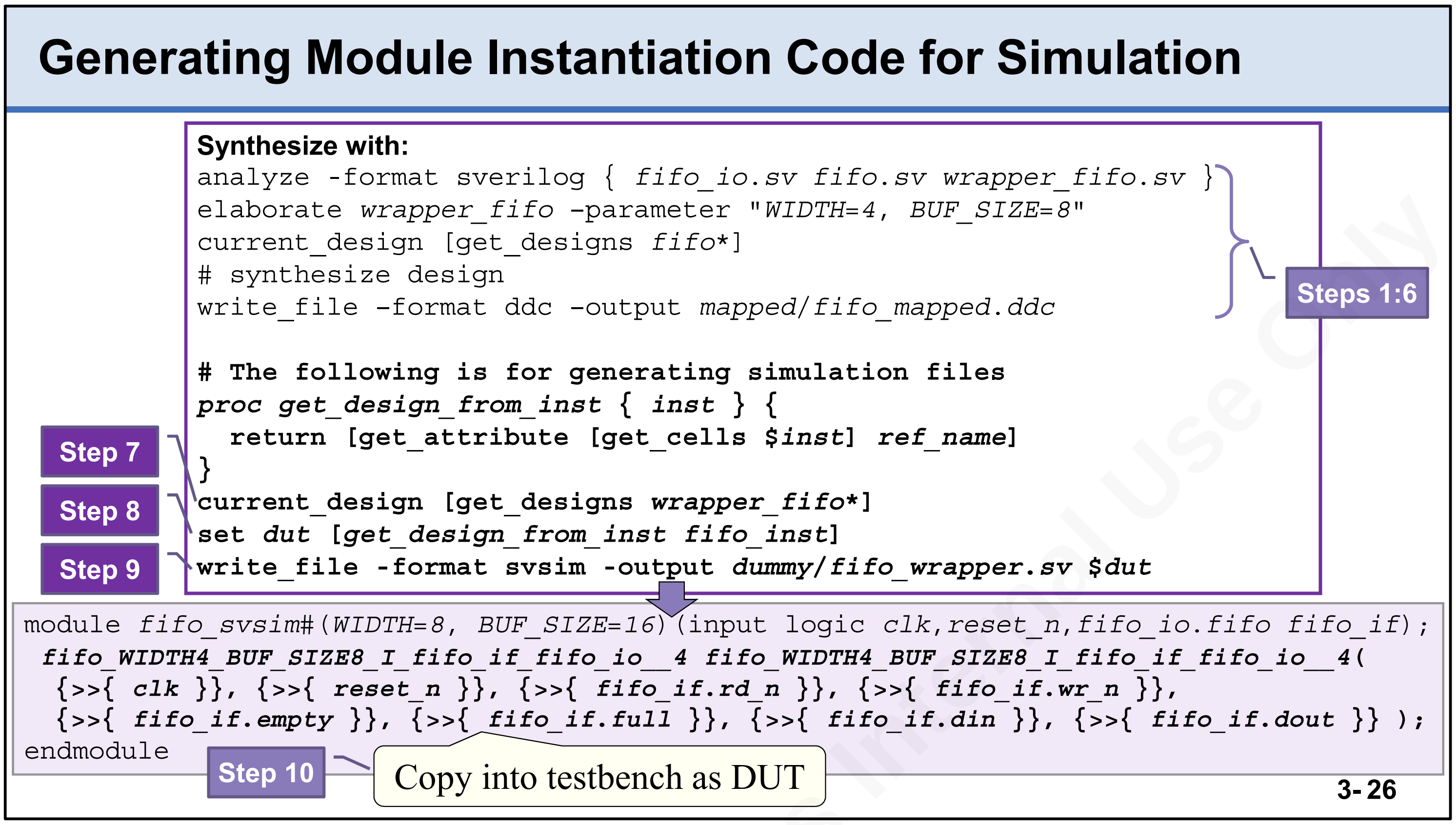
- Follow the 6 block level synthesis steps shown previously

Add the following steps:

- Step 7: Set current design to wrapper
- Step 8: Get design via instance name
- Step 9: Write out a new wrapper file for the design
- Step 10: Copy and paste the design instance from the new wrapper file into your testbench

Compile and run gate-level simulation

3- 25



Generating Module Instantiation Code for Simulation

Synthesize with:

```
analyze -format sverilog { fifo_io.sv fifo.sv wrapper_fifo.sv }
elaborate wrapper_fifo -parameter "WIDTH=4, BUF_SIZE=8"
current_design [get_designs fifo*]
# synthesize design
write_file -format ddc -output mapped/fifo_mapped.ddc

# The following is for generating simulation files
proc get_design_from_inst { inst } {
  return [get_attribute [get_cells $inst] ref_name]
}
current_design [get_designs wrapper_fifo*]
set dut [get_design_from_inst fifo_inst]
write_file -format svsim -output dummy/fifo_wrapper.sv $dut
```

```
module fifo_svsim#(WIDTH=8, BUF_SIZE=16)(input logic clk,reset_n,fifo_io.fifo fifo_if);
 fifo_WIDTH4_BUF_SIZE8_I_fifo_if_fifo_io__4 fifo_WIDTH4_BUF_SIZE8_I_fifo_if_fifo_io__4(
  {>>{ clk }}, {>>{ reset_n }}, {>>{ fifo_if.rd_n }}, {>>{ fifo_if.wr_n }},
  {>>{ fifo_if.empty }}, {>>{ fifo_if.full }}, {>>{ fifo_if.din }}, {>>{ fifo_if.dout }} );
endmodule
```

3- 26

"`>>`" operator streams data from left to right
"`<<`" operator streams data from right to left

```
bit[7:0] s;
s = {>> {4'b1101}};
$display("s = %b", s);  // s = 11010000

s = {<< {4'b1101}};
$display("s = %b", s);  // s = 10110000
```

One common use is for bit reversal:

```
bit[7:0] a, b;
a = 8'b1001_0110;
b = {<< {a}};
$display("a = %8b", a);  // a =
    1001_0010
$display("b = %8b", b);  // b =
    0100_1001
```

Advanced SystemVerilog Features

Pack/Unpack Array

Struct/Union

Interface

Package

3- 27

SystemVerilog Packages

- **Packages are a mechanism for sharing among `module`, `program` and `interface` the following:**
  - Parameters (can only be `localparam`)
  - Type definitions
  - Tasks & functions
  - Sequence and property declarations // simulation
  - Classes // simulation
- **Import package into appropriate scope**
  - Explicit use of package content
    `pkg16::WIDTH;`
  - Implicit import of all content of package
    `import pkg16::*;`

```
package pkg16;
  localparam WIDTH=16;
  typedef int unsigned uint;
  task automatic my_task(…);
    …
  endtask
  function automatic bit f(…);
    …
  endfunction
endpackage
```

3-28

About Packages
In any SystemVerilog design projects, it is common for a design team to reuse types, functions, and tasks. When you put these common constructs in packages, they can be shared among the team. This allows developers to use existing code based on their requirements without any ambiguity. After specifying all types, functions, and tasks in a package, you analyze the package. Modules that use the package declarations can be analyzed separately without the need to reanalyze the package. This can save runtime when large packages are used.
The following restrictions apply when you use packages:

- Wire and variable declarations in packages are not allowed.

The tool issues an error message.

- Functions and tasks that are declared inside packages need to be automatic.
- Sequence, property, and program blocks are ignored.

Using Packages
To use a package in SystemVerilog,
1. Analyze the package by using the analyze command.
The command creates a temporary package_name.pvk file. If you modify this analyzed package by adding or removing functions, tasks, types, and so on, the tool overwrites this temporary file and issues a VER-26 warning message similar to the following:
Warning: ./test.sv:1: The package p has already been analyzed. It is being replaced. (VER-26)
2. Analyze and elaborate the modules that use the package created in step 1 by using the analyze and elaborate commands respectively. If your modules were analyzed using a previous version of the package, repeat step 2 so that the tool uses the latest declarations from the package.

Rules Governing Packages for Synthesis

- **Restrictions for packages used in RTL code:**
  - Net (`wire`) and variable (`var`) declarations not allowed at package scope
  - Functions and tasks inside packages must be automatic
  - Sequence, property, and classes are ignored
  - Packages must not contain any processes
    - `always`, `initial` or `assign`
  - Packages must be self contained
- **Analyze package with –library switch to store result for reuse**
- **Analyze and elaborate the modules that use the package by adding library directory path to search path**

⚠ package tasks and functions are static unless explicitly automatic

3- 29

SystemVerilog Virtual Class

- **Limitation in package - Parameters cannot be changed**
  - – Parameterized functions in packages may be problematic

```
package pkg_16;
  localparam WIDTH=16;
  function automatic logic[WIDTH-1:0] Barrel_shift(...) ...
endpackage
```

- **Two potential solutions**
  - Create a library for each variation of the parameter
  - Embed parameterized static function in virtual class
    - – DC requires declaration must be in same $unit (file) as module

```
virtual class functions #(parameter WIDTH=16);
  static function automatic logic[WIDTH-1:0] Barrel_shift(...) ...
endclass
module encryptor #(WIDTH=16)(...);
  assign dout = functions#(WIDTH)::Barrel_shift(...);
endmodule
```

3-30

Using Package & Virtual Class: Script

```
encryptor
├─ script/
│  └ run.tcl
├─ rtl/
│  └ encryptor.sv
├─ package/
│  └ pkg.sv
└─ pkg_lib/
   └ pkg.pvk
```

```
# run.tcl
source ./script/common_setup.tcl
source ./script/dc_setup.tcl
define_design_lib pkg_lib -path ./pkg_lib                }
lappend search_path { ./package ./pkg_lib }              }
analyze -format sverilog -library pkg_lib { pkg.sv }
analyze -format sverilog { encryptor.sv }                }
elaborate encryptor; # using package parameter           }
link
source constraint.tcl
compile
# report qor violations and save output
```

Setup package library and path

Analyze package (save in library)

Analyze and elaborate RTL

```
unix% dc_shell -f script/run.tcl | tee -i run.log
```

3-31

Unit Objectives Review

Having completed this unit, you should be able to:

- **Describe the difference between `packed` and `unpacked` array**
- **Use `struct` to create encapsulated data structure**
- **Use `interface` to simplify module connectivity**
- **Perform bottom-up synthesis on modules with `interface`**

3- 32

Lab 2: SystemVerilog Interface

45 min

Managing SystemVerilog interface and parameters

$$\begin{tikzcd}
\boxed{\text{Implement SystemVerilog interface}} \arrow[d] \\
\boxed{\text{Synthesize RTL with parameterized interface}} \arrow[d] \\
\boxed{\text{Develop synthesis script for integration and simulation}}
\end{tikzcd}$$

3- 33

This page was intentionally left blank

Agenda

Unit Objectives

After completing this unit, you should be able to:

- Become aware of different RTL coding styles
- Instantiate DesignWare component for synthesis and simulation
- Describe the issue associated with datapath leakage

Advanced SystemVerilog Features

Contrasting Coding Styles

For Loop Coding Efficiency

Datapath Coding QoR

4- 3

Example Illustrating Different Coding Styles

- **Specification:**
  - **Rotate an input vector, din, k positions to the left**

For example:

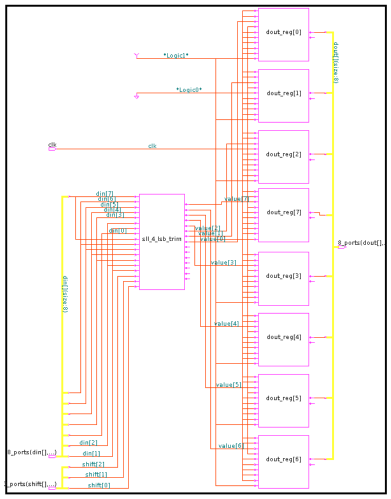
If the variable k currently is the value 5, then din must be rotated left by 5 bit positions

– Original din bit sequence:	`11001011`
– Rotated left by 5 bit positions:	`01111001`

4-4

Illustrating Different Coding Styles (cont'd)

- `assign`



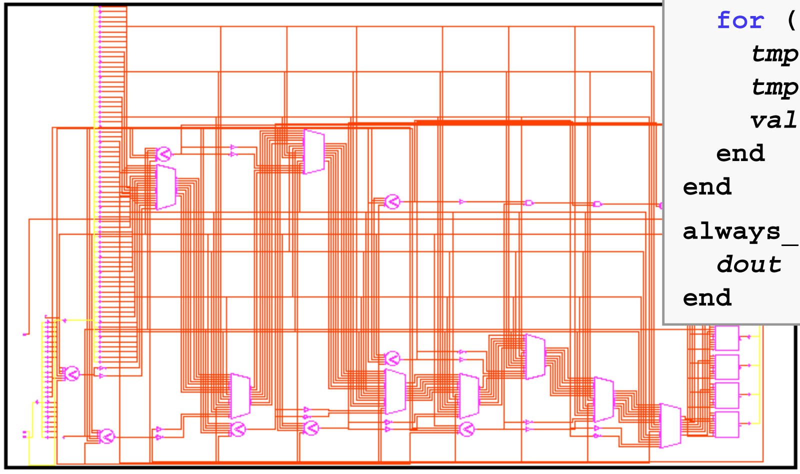
```
logic [WIDTH-1:0] value, tmp;

assign {value, tmp} = {din, din} << k;

always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
10.89	35	-3.46	138.76	173 MB

4- 5

Illustrating Different Coding Styles (cont'd)

- **For loop**



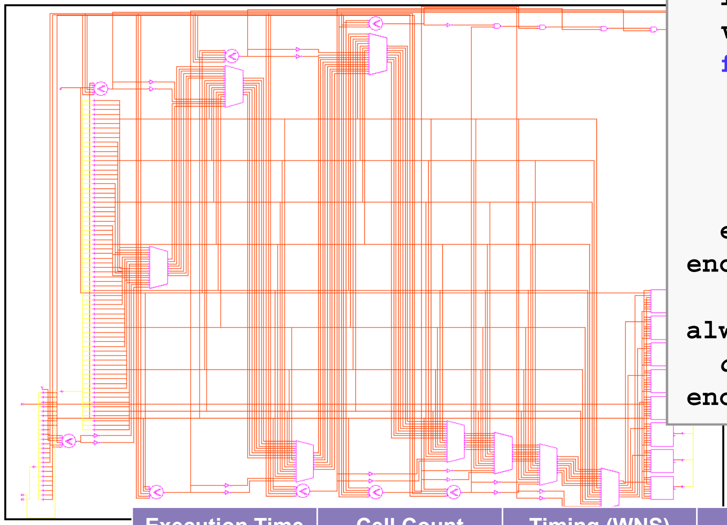
```
always_comb begin
  logic [WIDTH-1:0] tmp = '0;
  value = din;
  for (int i=0; i<k; i++) begin
    tmp[0] = value[WIDTH-1];
    tmp[WIDTH-1:1] = value[WIDTH-2:0];
    value = tmp;
  end
end
always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
15.43	84	-1.58	221.87	174 MB

4-6

Illustrating Different Coding Styles (cont'd)

- For loop with break



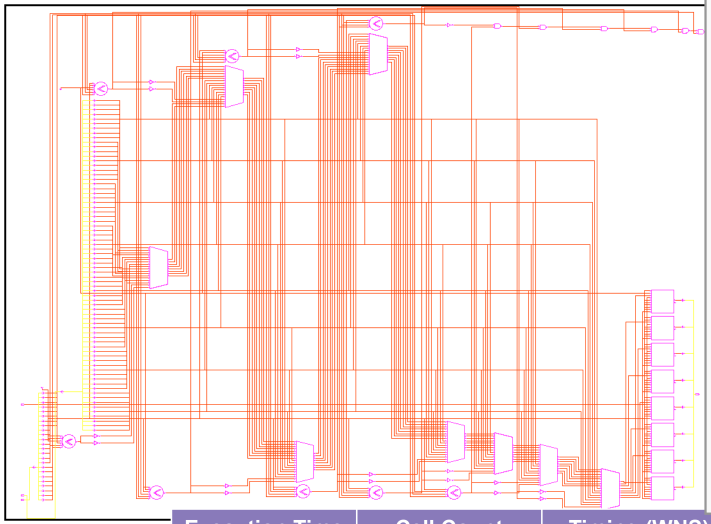
```
always_comb begin
  logic [WIDTH-1:0] tmp = '0;
  value = din;
  for (int i=0; i<WIDTH; i++) begin
    if (i == k) break;
    tmp[0] = value[WIDTH-1];
    tmp[WIDTH-1:1] = value[WIDTH-2:0];
    value = tmp;
  end
end

always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
16.05	92	-1.45	228.48	174 MB

4-7

Illustrating Different Coding Styles (cont'd)

- **While loop with break**



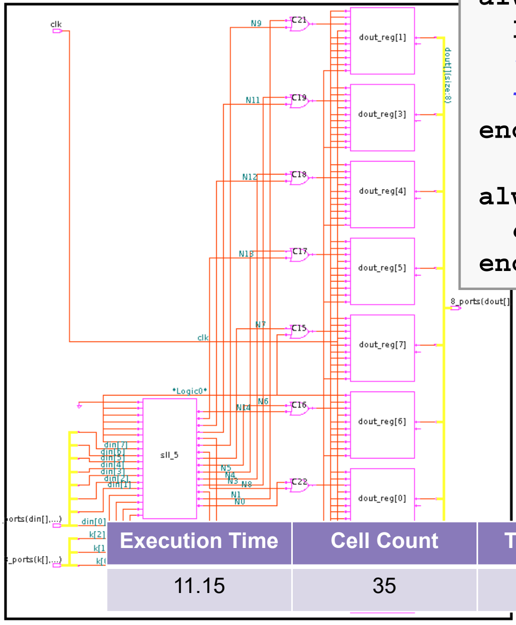
```
int unsigned i;
always_comb begin
  logic [WIDTH-1:0] tmp = '0;
  i = 0;
  value = din;
  while (i != WIDTH) begin
    if (i == k) break;
    i = i + 1;
    tmp[0] = value[WIDTH-1];
    tmp[WIDTH-1:1] = value[WIDTH-2:0];
    value = tmp;
  end
end

always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
16.04	92	-1.45	228.48	174 MB

4-8

Illustrating Different Coding Styles (cont'd)

- Left shift



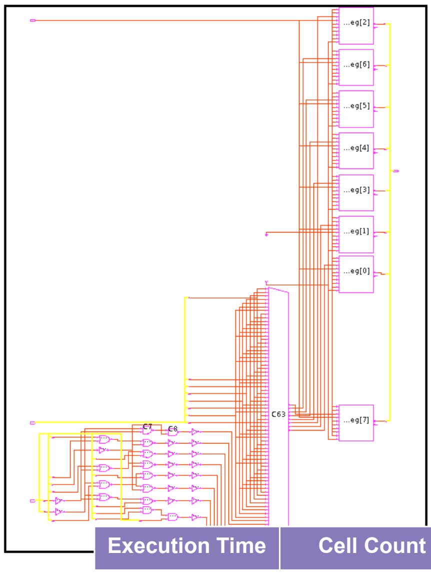
```
always_comb begin
  logic [WIDTH*2-1:0] tmp = '0;
  tmp = din << k;
  value = tmp[WIDTH-1:0] | tmp[WIDTH*2-1:WIDTH];
end

always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
11.15	35	-3.46	138.76	173 MB

4-9

Illustrating Different Coding Styles (cont'd)

- **Case**



```
always_comb begin
  unique case (k)
    3'b000: value = din;
    3'b001: {value[0],   value[7:1]} = din;
    3'b010: {value[1:0], value[7:2]} = din;
    3'b011: {value[2:0], value[7:3]} = din;
    3'b100: {value[3:0], value[7:4]} = din;
    3'b101: {value[4:0], value[7:5]} = din;
    3'b110: {value[5:0], value[7:6]} = din;
    3'b111: {value[6:0], value[7]}   = din;
  endcase
end

always_ff @(posedge clk) begin
  dout <= value;
end
```

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
11.32	35	-3.46	138.76	173 MB

4-10

Summary of Coding Styles

Coding Style	Execution Time	Cell Count	Timing (WNS)	Area	Memory Usage
Assign	10.89	35	-3.46	138.76	173 MB
For Loop	15.43	84	-1.58	221.87	174 MB
For with Break	16.05	92	-1.45	228.48	174 MB
While Loop	16.04	92	-1.45	228.48	174 MB
Shift	11.15	35	-3.46	138.76	173 MB
Case	11.32	35	-3.46	138.76	173 MB

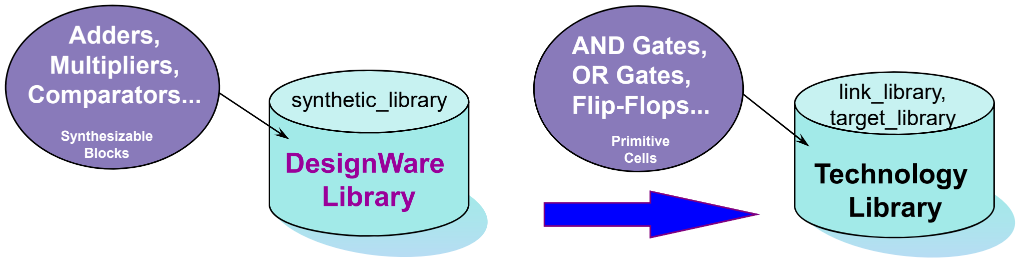
4- 11

Efficiency of Case vs. For Loop

- **The for loop is more coding efficient and allows for parameterization**
- **However, the for loop will be unrolled and each conditional statement treated as a separate statement**
  - This can have good and bad consequences
    - The bad – potential extra logic (larger area)
    - The good – the extra logic may lead to better timing.
- **The key to achieving area QoR is to reduce number of operators (share resource)**
  - Side effect may be reduced timing QoR
- **The key to achieving timing QoR is to dedicate operation for each signal (duplicate resource)**
  - Side effect may be reduced area QoR

4- 12

An Alternative: DesignWare



- Technology-independent “soft macros” such as adders, comparators, etc.
- Enables user to imply large and complex arithmetic operations to be synthesized:

```
if (A1 >= A2) Y = M * X + B;
```

- Multiple architectures for each "soft macro" allow synthesis to evaluate speed/area tradeoffs and choose the best implementation

  https://www.synopsys.com/dw/buildingblock.php

4- 13

RTL Using DesignWare Component

- **DesignWare "shifter"**

```
// Instance of DW01_bsh
DW01_bsh #(.A_width(WIDTH), .SH_width($clog2(WIDTH)))
bsh ( .A(din), .SH(k), .B(value) );

always_ff @(posedge clk) begin
  dout <= value;
end
```

For simulation: `-y ${SYNOPSYS}/dw/sim_ver +libext+.v+.sv \`
`+incdir+${SYNOPSYS}/dw/sim_ver+`

For Synthesis: `set_app_var link_library "* $target_library`
`dw_foundation.sldb"`

4- 14

DesignWare Function Call

- **Useful for inline coding of DesignWare operation**

```
localparam A_width  = WIDTH;
localparem SH_width = $clog2(WIDTH);
`include "DW01_bsh_function.inc"

always_ff @(posedge clk) begin
  dout <= DWF_bsh(din, k);
end
```

Same setup requirement as shown in previous page

4- 15

Advanced SystemVerilog Features

4- 16

Can Timing & Area QoR Exist at Same Time?

- **Operations inside a for loop**

```
for (int K = 0; K < 8; K++) begin
   if (K > (A - 1))
      begin
         S[K] = 1'b1;
      end
   else
      begin
         S[K] = 1'b0;
      end
end
```

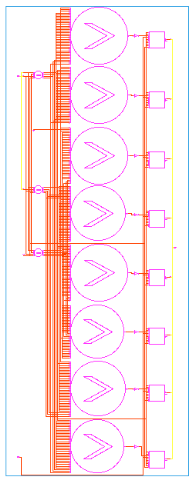
A - 1
has the same fixed value during all iterations of the `for` loop.

Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
6.37	32	-0.4	104.45	174 MB

4- 17

Synthesis Perspective of Given Example

- **If the synthesis tool does not move the re-computed fixed value (A-1) outside of the for loop, then 8 decrementer resource blocks <u>may</u> be synthesized**

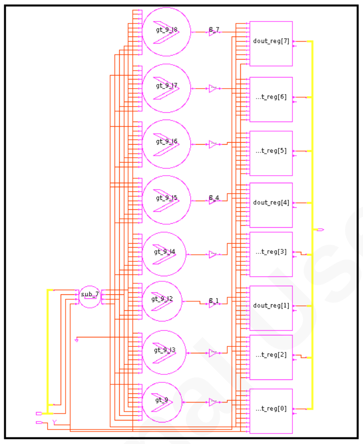


4- 18

Operation Outside the Loop

```
Temp = A - 1;
for (int K = 0; K < 8; K++) begin
      if (K > Temp)
         begin
            S[K] = 1'b1;
         end
      else
         begin
            S[K] = 1'b0;
         end
end
```

A - 1 was manually removed from the `for` loop.



Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
6.58	32	-0.74	96.57	174 MB

4- 19

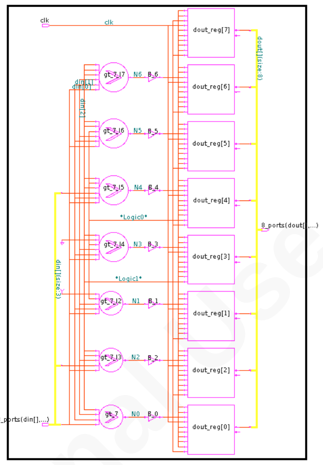
This coding trick used to work. But, the new generation of synthesizers are now able to automatically share the operator. So, this coding trick no longer has value.

Synthesis Specific Coding Optimization

```
for (int K = 0; K < 8; K++) begin
   if ((K + 1) > A)
      begin
         S[K] = 1'b1;
      end
   else
      begin
         S[K] = 1'b0;
      end
   end
```

Remember that when this `for` loop is unrolled, K is a fixed value on each iteration of the loop.

`K + 1` can be synthesized as the fixed values 2, 3, 4, 5, 6, 7, 8.



Execution Time	Cell Count	Timing (WNS)	Area	Memory usage
5.24	16	0	63.79	173 MB

Optimum Synthesis Results

- **Synthesized hardware included:**
  - 0 Decrementer resource block
  - 7 Comparator resource blocks
    - One resource block for each of the comparisons

```
(K + 1) > A
```

    - Synthesizer can optimize the resource blocks

Coding Style	Execution Time	Cell Count	Timing (WNS)	Area	Memory Usage
Operation inside loop	6.37	32	-0.4	104.45	174 MB
Operation outside loop	6.58	32	-0.74	96.57	174 MB
No operation	5.24	16	0	63.79	173 MB

4- 21

Advanced SystemVerilog Features

Contrasting Coding Styles

For Loop Coding Efficiency

Datapath Coding QoR

4- 22

Datapath QoR - Signed Arithmetic

- **For signed arithmetic, take advantage of the new signed feature of SystemVerilog**
  - Manual sign extension may lead to excess logic
  - Let the language help you

Bad QoR	Good QoR

`input  [7:0]  a, b;`
`output [15:0] z;`
`// a, b sign-extended to width of z`
`assign z = {{8{a[7]}}, a[7:0]} *`
`{{8{b[7]}}, b[7:0]};`
`// unsigned 16x16=16 bit multiply`
	`input         [7:0]  a, b;`
	`output        [15:0] z;`
	`logic signed  [15:0] z_sgn;`
	`assign z_sgn = signed'(a) * signed'(b);`
	`assign z = unsigned'(z_sgn);`
	`// signed 8x8=16 bit multiply`

	`input  signed [7:0]  a, b;`
	`output signed [15:0] z;`
	`assign z = a * b;`
	`// signed 8x8=16 bit multiply`

4-23

Datapath QoR – Mixed Signed Arithmetic

- **For mixed signed arithmetic, be careful of rules of operand**
  - If either operand is unsigned, the result is unsigned

Functionally Incorrect	Functionally Correct

`input        [7:0]  a; // unsigned`
`input  signed [7:0]  b;`
`output signed [15:0] z;`
`// expression becomes unsigned`
`assign z = a * b;`
`// a * b is an unsigned multiply`
	`input        [7:0]  a; // unsigned`
	`input  signed [7:0]  b;`
	`output signed [15:0] z;`
	`// 0-extended (positive value), cast to signed`
	`assign z = signed'({1'b0, a}) * b;`
	`// signed multiply`

`input signed [7:0]  a;`
`output signed [11:0] z;`
`// constant is unsigned`
`assign z = a * 4'b1011;`
`// unsigned multiply`
	`input  signed [7:0]  a;`
	`output signed [15:0] z1, z2;`
	`// cast constant into signed`
	`assign z1 = a * signed'(4'b1011);`
	`// mark constant as signed`
	`assign z2 = a * 4'sb1011;`
	`// -> signed multiply`

4- 24

Datapath QoR - Signed Arithmetic Part Select

- **Part select of a signed variable results in a unsigned value**

Functionally Incorrect	Functionally Correct

`input  signed [7:0]  a, b;`
`output signed [15:0] z1, z2;`
`// a[7:0] is unsigned – zero extend`
`assign z1 = a[7:0];`
`// a[6:0] is unsigned`
`assign z2 = a[6:0] * b;`
`// unsigned multiply`
	`input  signed [7:0]  a, b;`
	`output signed [15:0] z1, z2;`
	`// a is signed - sign-extended`
	`assign z1 = a;`
	`// cast a[6:0] to signed - signed multiply`
	`assign z2 = signed’(a[6:0]) * b;`

- **Follow guideline described here for better datapath QoR:**

https://solvnet.synopsys.com/retrieve/015771.html

4- 25

Datapath QoR - Leakage

- **Bit vector not wide enough to capture full resolution of operation**
  - Use linting tools to catch the problem
  - `analyze_datapath_extraction` may also detect issue

Bad QoR	Good QoR

`module bad (input logic[7:0] a, b, c, d,`
`                   output z);`
`  logic [15:0] t;`
`  assign t = a * b + 3 * c;`
`  assign z = t + d;`
`endmodule`
	`module good (input logic[7:0] a, b, c, d,`
	`                    output logic[16:0] z);`
	`  logic [16:0] t;`
	`  assign t = a * b + 3 * c;`
	`  assign z = t + d;`
	`endmodule`

Output from **`analyze_datapath_extraction:`** Information: Operator associated with resources 'add_5 (bad.v:5)'in design ‘bad' breaks the datapath extraction because there is leakage due to truncation on its fanout.	

- **Follow guideline described here to solve issue:**
  https://www.synopsys.com/Company/Publications/DWTB/Pages/dwtb-analyze-datapath-extraction-2014Q2.aspx

4- 26

Unit Objectives Review

Having completed this unit, you should be able to:

- **Become aware of different RTL coding styles**
- **Instantiate DesignWare component for synthesis and simulation**
- **Describe the issue associated with datapath leakage**

4- 27

This page was intentionally left blank

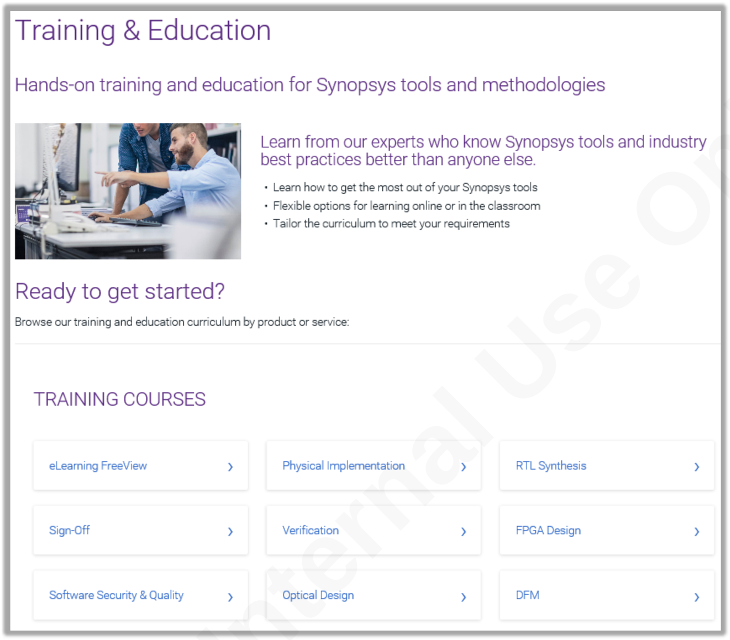
SYNOPSYS®
Silicon to Software™

Customer Support

20181001

Synopsys Support Resources

- **Build a solid foundation:**
  Hands-on training for Synopsys tools and methodologies
  https://synopsys.com/support/training.html
  - Workshop Schedule and Registration
  - Download Labs (SolvNet ID required)
- **Drill down to areas of interest:**
  SolvNet online support
  https://solvnet.synopsys.com
  - Online technical information and access to support resources
  - Documentation & Media
- **Ask an Expert:**
  Synopsys Support Center
  https://onlinecase.synopsys.com



Training & Education

Hands-on training and education for Synopsys tools and methodologies

Learn from our experts who know Synopsys tools and industry best practices better than anyone else.

- Learn how to get the most out of your Synopsys tools
- Flexible options for learning online or in the classroom
- Tailor the curriculum to meet your requirements

Ready to get started?

Browse our training and education curriculum by product or service:

TRAINING COURSES

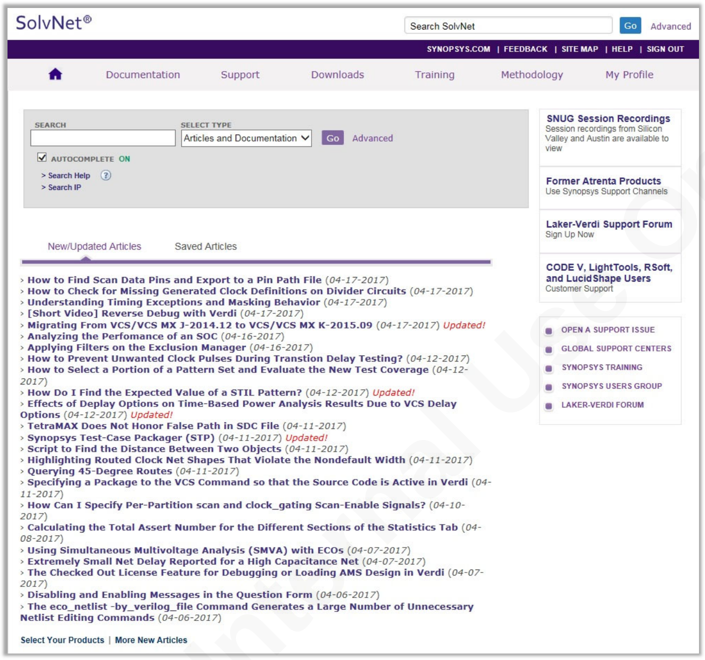
eLearning FreeView ›	Physical Implementation ›	RTL Synthesis ›
Sign-Off ›	Verification ›	FPGA Design ›
Software Security & Quality ›	Optical Design ›	DFM ›

https://training.synopsys.com

CS - 2

SolvNet Online Support

- Immediate access to the latest technical information
- Product Update Training
- Methodology Training
- Thousands of expert-authored articles, Q&As, scripts and tool tips
- Open a Support Center Case
- Release information
- Online documentation
- License keys
- Electronic software downloads
- Synopsys announcements (latest tool, event and product information)



SNUG Session Recordings
Session recordings from Silicon Valley and Austin are available to view

Former Atrenta Products
Use Synopsys Support Channels

Laker-Verdi Support Forum
Sign Up Now

CODE V, LightTools, RSoft, and LucidShape Users
Customer Support

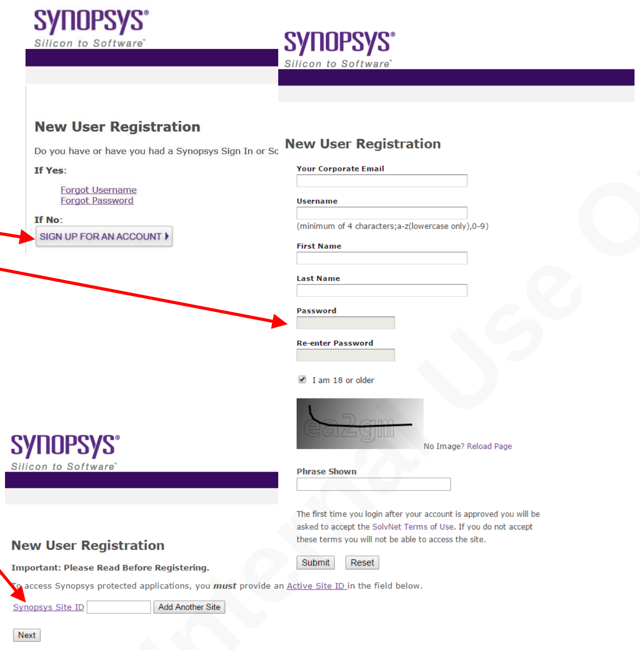
› How to Find Scan Data Pins and Export to a Pin Path File (04-17-2017)
› How to Check for Missing Generated Clock Definitions on Divider Circuits (04-17-2017)
› Understanding Timing Exceptions and Masking Behavior (04-17-2017)
› [Short Video] Reverse Debug with Verdi (04-17-2017)
› Migrating From VCS/VCS MX J-2014.12 to VCS/VCS MX K-2015.09 (04-17-2017) Updated!
› Analyzing the Perfomance of an SOC (04-16-2017)
› Applying Filters on the Exclusion Manager (04-16-2017)
› How to Prevent Unwanted Clock Pulses During Transtion Delay Testing? (04-12-2017)
› How to Select a Portion of a Pattern Set and Evaluate the New Test Coverage (04-12-2017)
› How Do I Find the Expected Value of a STIL Pattern? (04-12-2017) Updated!
› Effects of Deplay Options on Time-Based Power Analysis Results Due to VCS Delay Options (04-12-2017) Updated!
› TetraMAX Does Not Honor False Path in SDC File (04-11-2017)
› Synopsys Test-Case Packager (STP) (04-11-2017) Updated!
› Script to Find the Distance Between Two Objects (04-11-2017)
› Highlighting Routed Clock Net Shapes That Violate the Nondefault Width (04-11-2017)
› Querying 45-Degree Routes (04-11-2017)
› Specifying a Package to the VCS Command so that the Source Code is Active in Verdi (04-11-2017)
› How Can I Specify Per-Partition scan and clock_gating Scan-Enable Signals? (04-10-2017)
› Calculating the Total Assert Number for the Different Sections of the Statistics Tab (04-08-2017)
› Using Simultaneous Multivoltage Analysis (SMVA) with ECOs (04-07-2017)
› Extremely Small Net Delay Reported for a High Capacitance Net (04-07-2017)
› The Checked Out License Feature for Debugging or Loading AMS Design in Verdi (04-07-2017)
› Disabling and Enabling Messages in the Question Form (04-06-2017)
› The eco_netlist -by_verilog_file Command Generates a Large Number of Unnecessary Netlist Editing Commands (04-06-2017)

https://solvnet.synopsys.com

CS - 3

SolvNet Registration

1. **Go to SolvNet page:**
   - https://solvnet.synopsys.com/
2. **Click on:**
   - "Sign Up for an Account"
3. **Pick a username and password.**
4. **You will need your "Site ID"**
   - For Information on how to find your Site ID, select the "Synopsys Site ID" link
5. **Authorization typically takes just a few minutes.**


Silicon to Software

New User Registration

Do you have or have you had a Synopsys Sign In or Sc

If Yes:

Forgot Username
Forgot Password

If No:

SIGN UP FOR AN ACCOUNT

Silicon to Software

New User Registration

Your Corporate Email

Username

(minimum of 4 characters;a-z(lowercase only),0-9)

First Name

Last Name

Password

Re-enter Password

[✓] I am 18 or older

No Image? Reload Page

Phrase Shown

The first time you login after your account is approved you will be asked to accept the SolvNet Terms of Use. If you do not accept these terms you will not be able to access the site.

Submit Reset

Silicon to Software

New User Registration

Important: Please Read Before Registering.

To access Synopsys protected applications, you ***must*** provide an Active Site ID in the field below.

Synopsys Site ID [ ] Add Another Site

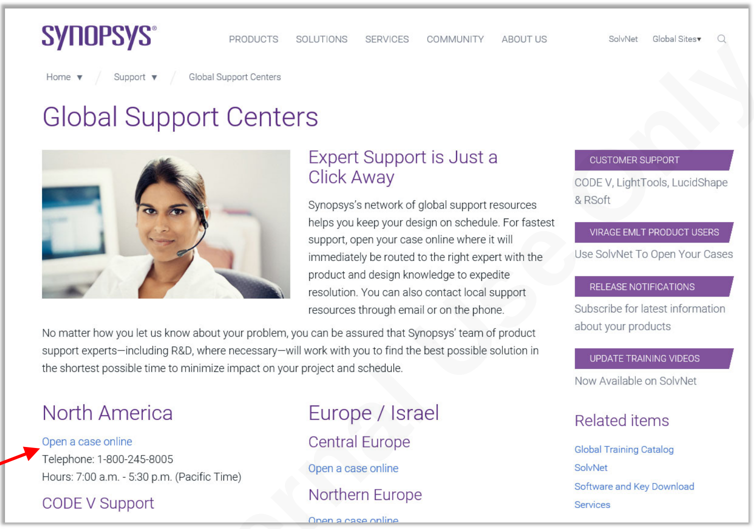
Next

https://solvnet.synopsys.com/ProcessRegistration **CS - 4**

Support Center

- **Industry seasoned Application Engineers:**
  - 50% of the support staff has >5 years applied experience
  - Many tool specialist AEs with >12 years industry experience
  - Engineers located worldwide
- **Great wealth of applied knowledge:**
  - Service >2000 issues per month
- **Remote access, and interactive debug, available via WebEx**

Contact us:
Open a support case



SYNOPSYS® PRODUCTS SOLUTIONS SERVICES COMMUNITY ABOUT US SolvNet Global Sites▾

Home ▾ / Support ▾ / Global Support Centers

Global Support Centers

Expert Support is Just a Click Away

Synopsys's network of global support resources helps you keep your design on schedule. For fastest support, open your case online where it will immediately be routed to the right expert with the product and design knowledge to expedite resolution. You can also contact local support resources through email or on the phone.

No matter how you let us know about your problem, you can be assured that Synopsys' team of product support experts—including R&D, where necessary—will work with you to find the best possible solution in the shortest possible time to minimize impact on your project and schedule.

CUSTOMER SUPPORT
CODE V, LightTools, LucidShape & RSoft

VIRAGE EMLT PRODUCT USERS
Use SolvNet To Open Your Cases

RELEASE NOTIFICATIONS
Subscribe for latest information about your products

UPDATE TRAINING VIDEOS
Now Available on SolvNet

North America
Open a case online
Telephone: 1-800-245-8005
Hours: 7:00 a.m. - 5:30 p.m. (Pacific Time)

CODE V Support

Europe / Israel
Central Europe
Open a case online
Northern Europe
Open a case online

Related items
Global Training Catalog
SolvNet
Software and Key Download
Services

https://www.synopsys.com/support/global-support-centers.html

CS - 5

Other Technical Sources

- **Application Consultants (ACs):**
  - Tool and methodology pre-sales support
  - Contact your Sales Account Manager for more information

- **Synopsys Professional Services (SPS) Consultants:**
  - Available for in-depth, on-site, dedicated, custom consulting
  - Contact your Sales Account Manager for more details

- **SNUG (Synopsys Users Group):**
  https://www.synopsys.com/community/snug.html

Summary: Getting Support

- **Customer Training**
  https://www.synopsys.com/support/training.html
  - Register for a Class
  - Download Labs
- **SolvNet**
  https://solvnet.synopsys.com
  - Tool Documentation and Support Articles
  - Product Update and Methodology Information / Training
  - Open a Support Case (Support Center)
- **Other Technical Resources**
  - Synopsys Users Group (SNUG)
  - Application Consultants
  - Synopsys Professional Services

CS - 7

This page was intentionally left blank.