ENCS5337: Chip Design Verification

Spring 2023/2024

SystemVerilog Part I

Dr. Ayman Hroub

Outline

- **Introduction**
- Data Types
- Operators
- Procedural Statements and Procedural Blocks
- Design and Verification Building Blocks

What is SystemVerilog?

- **SystemVerilog** is a unified hardware **specification**, description (**design**) and **verification** language.
- SystemVerilog is an extension of Verilog.
- SystemVerilog can be divided based on its roles,
  - SystemVerilog for design is an extension of Verilog-2005
  - SystemVerilog for verification.

History

- It is Verilog-2001 extension
- Contains hundreds of enhancements and extensions to Verilog
- In 2005, became an IEEE standard.
- Officially superseded Verilog in 2009
- Updated with more features in 2012, 2017, and 2023

SV Verilog-2001 Extensions

- Added data types and relaxation of rules on existing data types
- Higher abstraction-level modeling features
- Language enhancements for synthesis
- Interfaces to model communication between design blocks
- Language features to enable verification methodologies
- Assertions, constrained randomization, functional coverage
- Lightweight interface to C/C++ programs

Outline

- Introduction
- **Data Types**
- Operators
- Procedural Statements and Procedural Blocks
- Design and Verification Building Blocks

What is a Datatype

- A datatype is a set of values (2-state or 4-state) that can be used to declare data objects or to define user-defined data types.

- All Verilog datatypes except **`real`** have 4-state values: (0, 1, Z, X).
- 4-state logic is essential for gate-level modeling and for initialization at RTL

- Howeover, simulating everything in 4-state logic can add significant performance overhead.

SystemVerilog 2-State Datatype

- SystemVerilog adds 2-state value types based on bit:
  - Has values 0 and 1 only.
  - Direct replacements for reg, logic or integer.
  - Greater efficiency at higher-abstraction level modeling (RTL).
  - SystemVerilog defines the following predefined **`bit`** types of various widths

Type	Description	Sign
`bit`	Single bit, Scalable to vector	Default unsigned
`byte`	8-bit vector or ASCII character	Default signed
`shortint`	16-bit vector	Default signed
`int`	32-bit vector	Default signed
`longint`	64-bit vector	Default signed

SystemVerilog Real Data Type

- SystemVerilog also defines a 32-bit floating-point type, **`shortreal`**, to complement the 64-bit Verilog **`real`** type.

System Verilog `Logic` DataType

- **`logic`** defines that the variable or net is a 4-state data type.
- Initial value is x
- It can be driven in both procedural blocks and assign statements
- Cannot have multiple drivers

System Verilog `Bit` DataType

- 2-state data type (0,1)
- Initial value is 0
- bit is not signed by default
- It is useful in the cases where not all 4 values are needed. This helps to reduce the simulation time and the required memory
- When a 4-state value is converted to a 2-state value, any unknown or high impedance bits shall be converted to zeros

Integer Data Types

- There are two data types of integers:
  - 2-state types can take only 0 , 1 values.
  - 4-state types can take 0 , 1 , X , Z values.
- Integer types can be **signed** or **unsigned**, which can change the result of a arithmetic operation.
- **`$bits(var)`** is a system task that returns the number of bits in **`var`**
- signed and unsigned can be defined explicitly, e.g.,

```
reg unsigned var1;
shortint signed var2;
```

(4-state) Data Type	description
logic	User defined vector types
reg	
wire	
integer	32-bit signed integer

(2-state) Data Type	description
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer
byte	8-bit signed integer

Outline

- Introduction
- Data Types
- **Operators**
- Procedural Statements and Procedural Blocks
- Design and Verification Building Blocks

Assignment Operators (1)

- Operators that join an operation along with a blocking assignment to the first operand of the operator.
- Assignment operators are blocking assignments.
- Therefore, they are suitable for use only in:
  - RTL combinational logic
  - Temporary variables in RTL sequential code
  - Testbench and stimulus.

Assignment Operators (2)

Symbol	Usage	Meaning	Description
+=	a += b	a = a + b	add
-=	a -= b	a = a - b	subtract
*=	a *= b	a = a * b	multiply
/=	a /= b	a = a / b	divide
%=	a %= b	a = a % b	modulus
&=	a &= b	a = a & b	logical and
|=	a |= b	a = a | b	logical or
^=	a ^= b	a = a ^ b	logical xor
<<=	a <<= b	a = a << b	left shift logical
>>=	a >>= b	a = a >> b	right shift logical
>>>=	a >>>= b	a = a >>> b	right shift arithmetic
<<<=	a <<<= b	a = a <<< b	left shift arithmetic

Pre- and Post-Increment/Decrement Operators

- Pre-form (++a, --a) adds or subtracts and then uses new value.
- Post-form (a++, a--) uses a value and then adds or subtracts.

As Statement

```
for (int i=0; i < 7; i++)
  ...
```

As Expression

```
initial begin
  b = 1;
  a = b++; // post a=1, b=2
  a = ++b; // pre  a=3, b=3
  a = b--; // post a=3, b=2
  a = --b; // pre  a=1, b=1
end
```

SV Operators Associativity and Precedence

Operator	Associativity	Precedence
() [] :: .	Left	Highest
+ - ! ~ & ~& | ~| ^ ~^ ^~ ++ -- (unary)	Right	
**	Left	
* / %	Left	
+ - (binary)	Left	
<< >> <<< >>>	Left	
< <= > >= inside **dist**	Left	
== != === !== ==? !=?	Left	
& (binary)	Left	
^ ~^ ^~ (binary)	Left	
| (binary)	Left	
&&	Left	
||	Left	
?: (conditional operator)	Right	
–>	Right	
= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>= := :/ <=	None	
{} {{}}	Concatenation	Lowest

Strings

- The string data type in SystemVerilog is an ordered collection of characters (ASCII characters).

```
string myString = “Welcome to HW DV Course”;
$display (“%s”, myString);
$display (“%s”, myString[0]);
```

```
string  variable_name [= initial_value];
```

SystemVerilog String Operators

	Operator	Semantics
Equality	Str1 == Str2	Returns 1 if the two strings are equal and 0 if they are not
Inequality	Str1 != Str2	Returns 1 if the two strings are not equal and 0 if they are
Comparison	Str1 < Str2 Str1 <= Str2 Str1 > Str2 Str1 >= Str2	Returns 1 if the correspondig condition is true and 0 if false
Concatenation	{Str1, Str2, ..., StrN}	All strings will be concatenated into one resultant string
Replication	{multiplier{Str}}	Replicates the string N number of times, where N is specified by the multiplier
Indexing	Str[index]	Returns a byte, the ASCII code at the given index. If given index is out of range, it returns 0
Methods	Str.method([args])	The dot(.) operator is used to call string functions

SystemVerilog String Methods

Usage	Definition	Comments
str.len()	function int len()	Returns the number of characters in the string
str.putc()	function void putc (int i, byte c);	Replaces the i^{th} character in the string with the given character
str.getc()	function byte getc (int i);	Returns the ASCII code of the i^{th} character in str
str.tolower()	function string tolower();	Returns a string with characters in str converted to lowercase
str.compare(s)	function int compare (string s);	Compares str and s, as in the ANSI C strcmp function
str.icompare(s)	function int icompare (string s);	Compares str and s, like the ANSI C strcmp function
str.substr (i, j)	function string substr (int i, int j);	Returns a new string that is a substring formed by characters in position i through j of str

Enumeration

- An enumerated type declares a set of integral named constants, i.e., it defines a set of named values
- In the absence of a data type declaration, the default data type shall be `int`
- E.g., `enum {red, yellow, green} light1, light2;`

```
// silver=4, gold=5
enum {bronze=3, silver, gold} medal;
enum {a=0, b=7, c, d=8} alphabet;//error
```

Defining New data types as Enumerated Types

- A type name can be given so that the same type can be used in many places.
- **e.g., `typedef enum {NO, YES} boolean;`**
  `boolean myvar; // named type`

Some Enumerated Types Methods

- `first()` returns the value of the first member of the enumeration.
- `last()` returns the value of the last member of the enumeration.
- `next()` returns the Nth next enumeration value
- `prev()` returns the Nth previous enumeration value
- `num()` returns the number of elements in the given enumeration

Structures

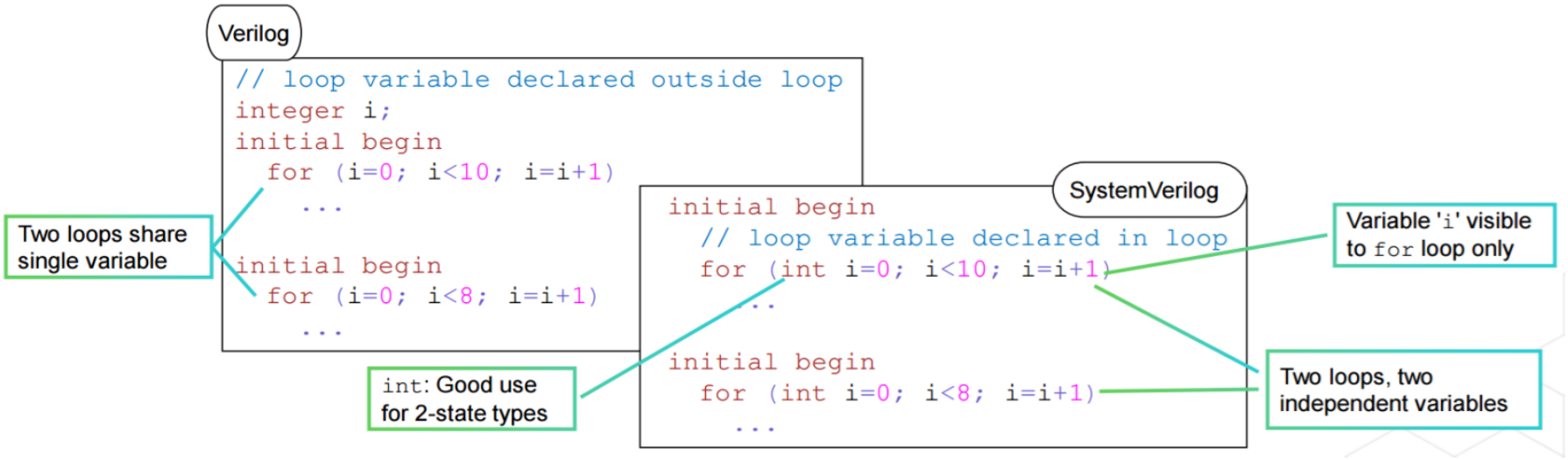
- A *structure* represents a collection of data types that can be referenced as a whole, or the individual data types that make up the structure can be referenced by name.

```
// named structure type
typedef struct
{ bit [7:0] opcode; bit [23:0] addr;}
instruction;
instruction IR; // define variable
IR.opcode = 8;
```

Outline

- Introduction
- Data Types
- Operators
- **Procedural Statements and Procedural Blocks**
- Design and Verification Building Blocks

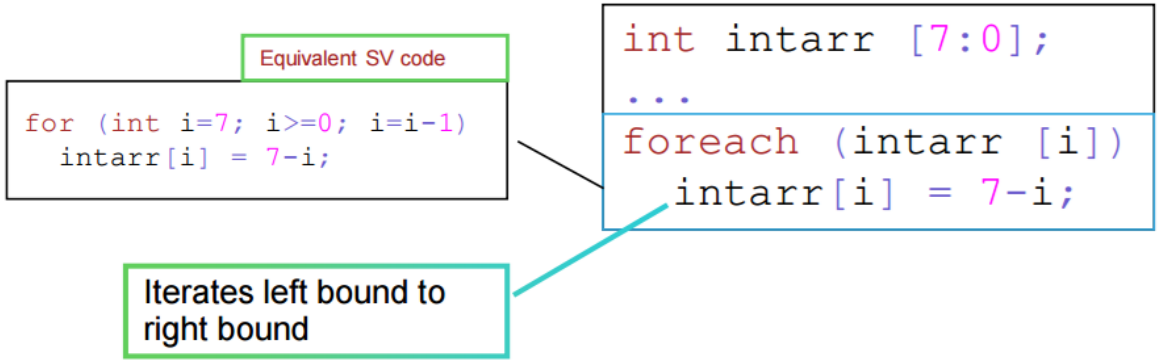
`for` Loop

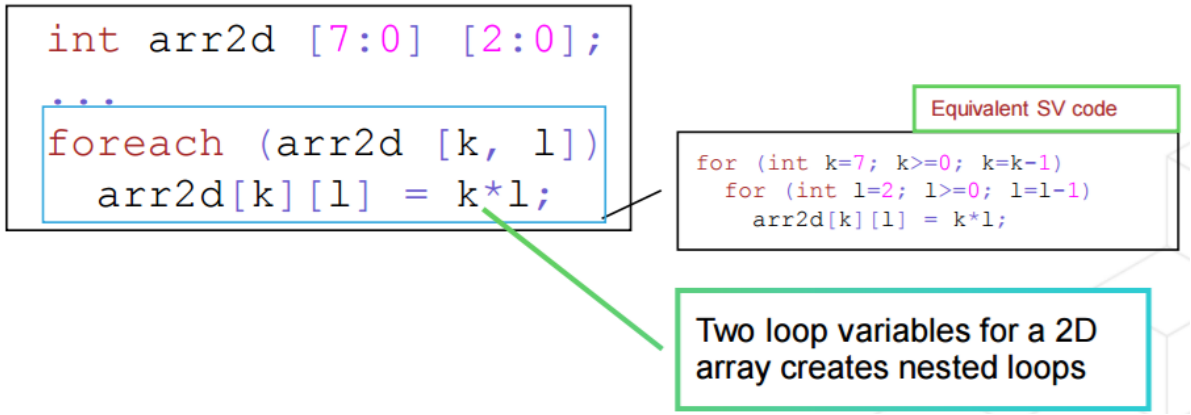

Verilog
// loop variable declared outside loop
integer i;
initial begin
for (i=0; i<10; i=i+1)
...
initial begin
for (i=0; i<8; i=i+1)
...
Two loops share single variable
SystemVerilog
initial begin
// loop variable declared in loop
for (int i=0; i<10; i=i+1)
...
initial begin
for (int i=0; i<8; i=i+1)
...
int: Good use for 2-state types
Variable 'i' visible to for loop only
Two loops, two independent variables

`foreach` Loop

- This loop iterates over all the elements of an array
- Loop variable characteristics:
  - Does not have to be declared.
  - is read only.
  - Only visible inside loop.
- Use multiple loop variables for multidimensional arrays.
  - Equivalent to nested loops.
- Useful for initializing and array processing.

`foreach` Loop (cont.)

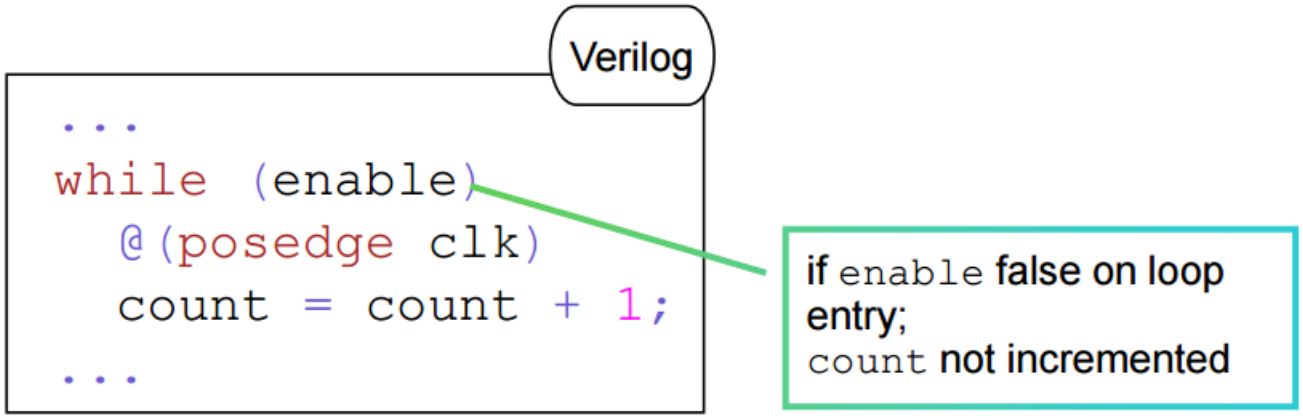




`while` Loop

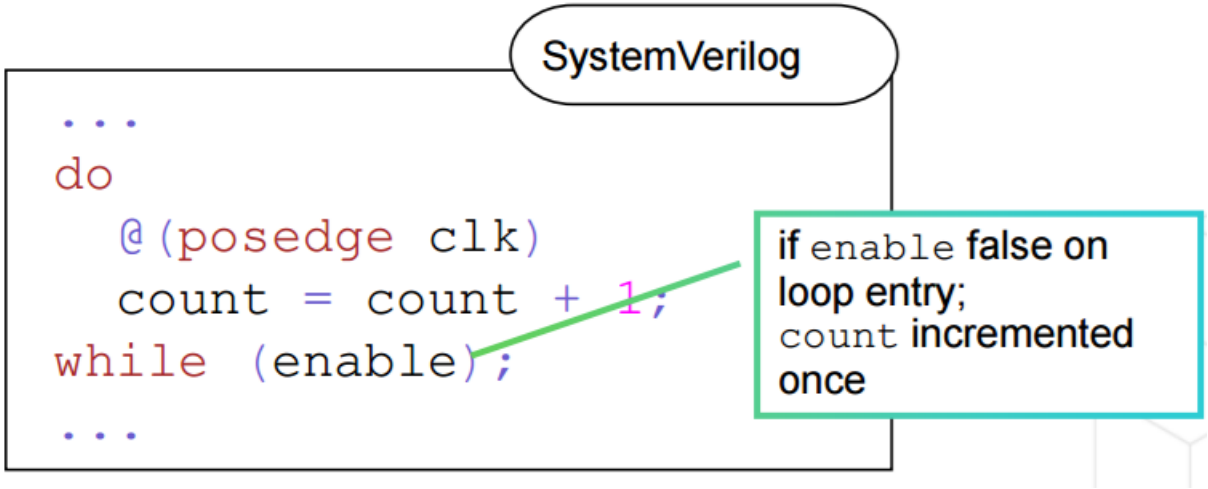
- The **`while`** loop executes a group of statements until **`expression`** becomes false.

- **`expression`** is checked at the beginning.



`do...while` Loop

- The **`expression`** is checked after statements execute.
- The statement block executes at least once.
- This makes certain loop functions easier to create.



break and **continue**

- SystemVerilog adds the **break** and **continue** keywords to control execution of loop statements.
- **break**
  - Terminates the execution of loop immediately.
  - Usually under conditional control.
- **continue**
  - Jumps to the next iteration of a loop.
  - Usually under conditional control.
- Also used in **for**, **while, repeat** and **do-while** loops

`break` and `continue` (Cont.)

```
repeat (8) begin
  data = {data[6:0], data[7]};
  if (data[7])
    break;
end
...
```

```
foreach (data [i]) begin
  if (data[i])
    continue;
  count = count + 1;
end
```

always_comb

- It is specialized procedural block for modeling combinational logic
- Implied, complete sensitivity list.
- Any variable assigned in an **always_comb** cannot be assigned by another procedure.
- Cannot contain further blocking timing or event control.

`always_comb` (cont.)

- Automatically executed once at time 0 without waiting for an event.
  - After all initial and always blocks have executed.
  - Ensures outputs are consistent with inputs.
- Tools may issue warnings if the block does not infer combinational logic.

always_comb (cont.)

```
always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

```
logic op;

always_comb
  if (sel)
    op = a;
  else
    op = b;

always_comb
  if (sel2)
    op = c;
  else
    op = d;
```

always_comb vs. always@*

always@*

- Can include timing and additional event controls.
- Can assign to variables which are assigned elsewhere.
- Triggered at time 0 with other blocks only if event on sensitivity list.

```
always @*
  if (sel == 1)
    op = a;
  else
    op = b;
```

always_comb

- Cannot contain any timing or event controls.
- Cannot assign to variables which are assigned elsewhere.
- Automatically triggered at time 0 after **always** and **initial** blocks.

```
always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

always_latch

- It is a specialized procedural block for modeling latched logic.
- Implied, complete sensitivity list.
- Variables assigned in an **always_latch** cannot be assigned by another procedure.
- Cannot contain further block timing or event control.

`always_latch` (Cont.)

- Automatically executed once at time 0 without waiting for an event:
  - After all **`initial`** and **`always`** blocks have executed.
  - Ensuring outputs are consistent with inputs.
- Tools can issue warnings if the block does not infer latched logic.

always_latch (Cont.)

```
always_latch
  if (gate == 1)
    op <= a;
```

```
always_latch
  if (en1)
    op <= c;

always_latch
  if (en2)
    op <= d;
```

`always` flip-flop (`always_ff`)

- It is a specialized procedural block for modeling registered logic.
- Variables assigned in **`always_ff`** cannot be assigned by another procedure.
- Contains one and only one event control.
- Cannot contain any block timing.
- Tools may issue warnings if the block does not infer registered logic.

always flip-flop (always_ff)

```
always_ff @(posedge clk or posedge rst)
  if (rst)
    op <= 1'b1;
  else
    op <= ip;
```

Outline

- Introduction
- Data Types
- Operators
- Procedural Statements and Procedural Blocks
- **Design and Verification Building Blocks**

Design Elements

- **`module`**, **`program`**, **`interface`**, **`checker`**, **`package`**, **`primitive`** and **`config`** (configuration) are called design elements in a SystemVerilog.

- These elements are primary building blocks used to model a design and verification environment.

Module

- The basic building block in SystemVerilog is the **`module`**
- Modules are used to represent design blocks.

- The module is enclosed between the keywords **`module`** and **`endmodule`**.

- Modules are primarily used to represent **design blocks**, but can also serve as containers for **verification code** and interconnections between verification blocks and design blocks.

Module Example

```
1  module <name> ([port_list]);
2      // Contents of the module
3  endmodule
4
5  // A module can have an empty portlist
6  module name;
7      // Contents of the module
8  endmodule
```

```
module flipflop(d, q, clk);
    input d, clk;
    output logic q;

    always_ff @(posedge clk) begin
        q <= d;
    end
endmodule
```

Packages

- New design element similar to a module:
  - Must be compiled separately.
  - Must be compiled before elements that reference the package.
- They contain declarations to be shared between elements:
  - Types, variables, subroutines...

Package

```
package mytypes;
  typedef enum {start,done} mode_t;
endpackage : mytypes
```

Package Example

```
package ComplexPkg;
    typedef struct {
        shortreal i, r;
    } Complex;

    function Complex add(Complex a, b);
        add.r = a.r + b.r;
        add.i = a.i + b.i;
    endfunction

    function Complex mul(Complex a, b);
        mul.r = (a.r * b.r) - (a.i * b.i);
        mul.i = (a.r * b.i) + (a.i * b.r);
    endfunction
endpackage : ComplexPkg
```

Importing a Package

Package

```
package mytypes;
  typedef enum {start,done} mode_t;
endpackage : mytypes
```

- Explicit – specifically named. Allows to reference selected package declarations

Explicit import into CUS

```
import mytypes::mode_t;
module mone(input  mode_t mode...);
    ...
```

- Implicit – all using wildcard (*)

Wildcard import

```
module mone(...);
  import mytypes::*;

  mode_t mode,
...
```

- Or directly access a declaration using the resolution operator (::):
  - Does not require import.

Resolved name

```
module mone(...);

  mytypes::mode_t mode,
...
```

Explicit Import

- An explicit import only imports the symbols specifically referenced by the **`import`**.
- It directly loads declaration into the design element as if it was declared in the design element.
- Declaration must be unique in the current scope:
  - Compilation error if local and other explicitly imported declarations have same name.

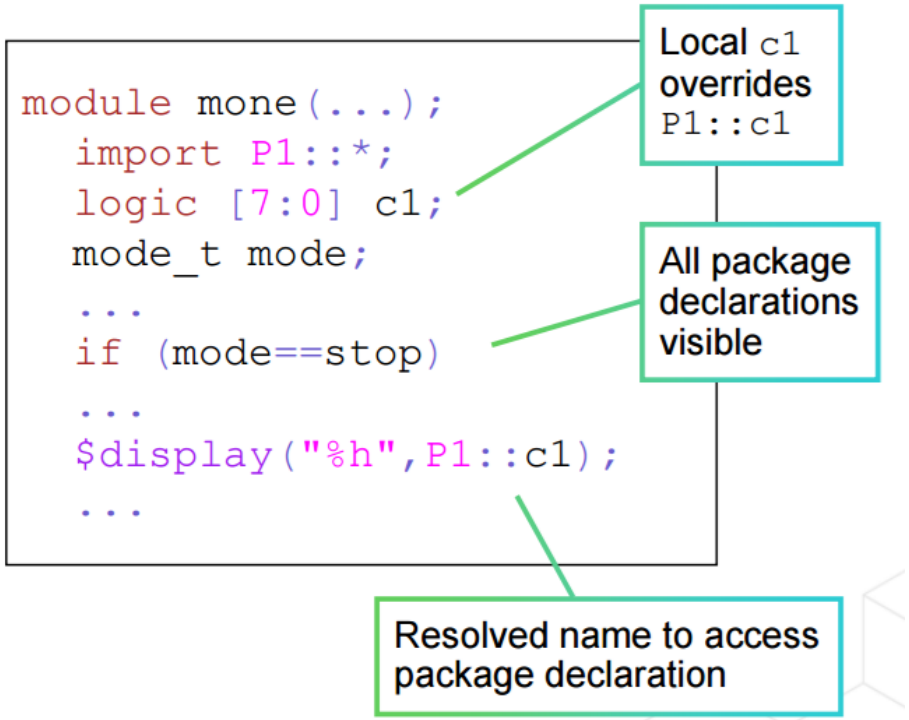
Wildcard Import

- A wildcard import allows all identifiers declared within a package to be imported provided the identifier is not otherwise defined in the importing scope.
- Local or explicitly imported declarations can override wildcard imported declarations.
- Package declarations still visible through resolved name.

Local or explicitly imported declarations take precedence over wildcard imports.

Wildcard Import Example

```
package P1;
  localparam int c1 = 10;
  typedef enum {start,stop} mode_t;
endpackage : P1
```



Program Block

- A program is very similar to a module, but intended for testbench code.
- Program blocks have special features and restrictions for testbench use.
- In particular, a program cannot instantiate hierarchy.
  - Programs are leaf elements.
  - Must be instantiated in a module.

Program Block (Cont.)

- The program is usually declared in a separate file, compiled separately and then instantiated in a module or interface.
- In a verification methodology using programs, all testbench code would be contained in programs, and all design (synthesizable) code in modules.
- By using programs only for verification code and modules or interfaces only for design code, race conditions between design and testbench can be reduced

Program Example

```
program memtest (
  output wire [7:0] data;
  output bit [4:0] addr;
  output bit read, write);
  ...
  initial begin
    ...
  end
endprogram : memtest
```

```
module top;
  wire [7:0] data;
  wire [4:0] address;
  wire       read, write;

  memtest test (.*);
  memory mem8x32 (.*);

endmodule  : top
```

Allowed Constructs in Program

- Local data declarations
  - Variables
  - User-defined types
  - Classes
- `initial` and `final` blocks
- `generate` blocks
- Task and function declarations
  - Only visible in program block
- Continuous assignments
- Clocking blocks
- Concurrent assertions
- Functional coverage groups (covergroups)

`final` Procedural Block

- It is a procedural block that executes **once** at the end of simulation:
- After explicit or implicit call to **`$finish`**.
- Cannot invoke scheduler (no scheduled assignments or delays).
- Can be used to calculate and display simulation statistics.

```
final begin
  if (timeout_error)
    $display ("ERROR: %0t: Test Timed Out", $time);
  else
    $display ("INFO: %0t: Test Complete", $time);
  $display("Error Count: %d", error_count);
  $display("Fifo Overflow Count: %d", fifo_overflow);
end
```

Not Allowed Constructs in Program

- As a general rule, constructs that clearly represent design rather than verification are not allowed.
  - `always` blocks
  - Declaration or instantiation of:
    - Interface
    - Module
    - Primitive
    - Program
  - Anything specific to modules
    - Parameter overrides (`defparams`)
    - `specify` blocks
    - `specparam` declarations

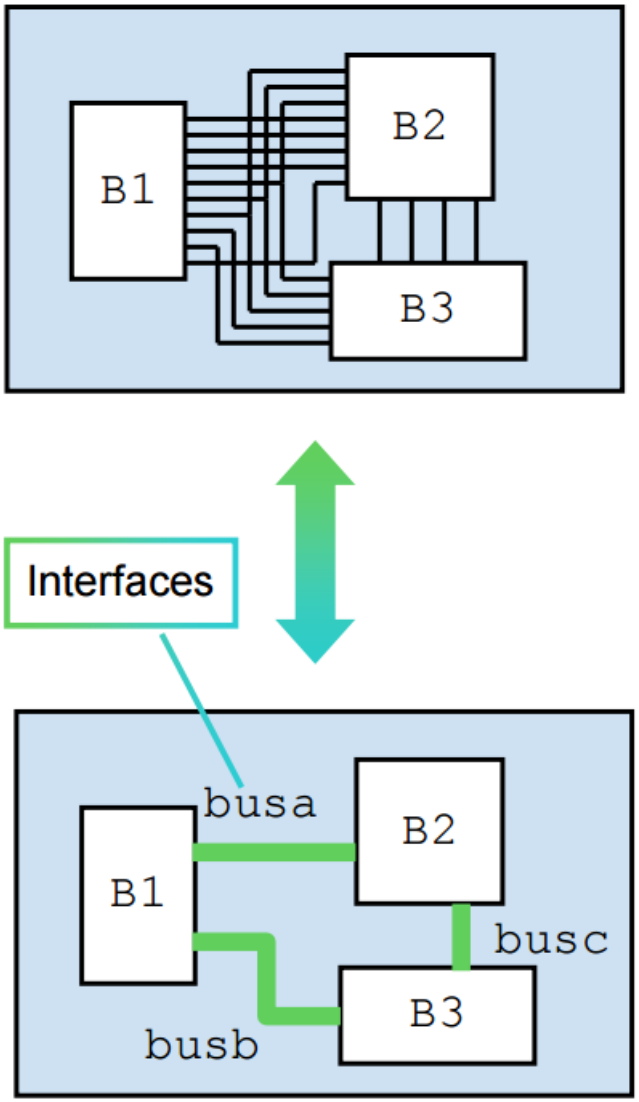
Interfaces

- The interface encapsulates the communication between design blocks, and between design and verification blocks
- It is a construct representing a bundle of defined wires. In other words, it is a separately declared and named group of signals.
- All connections associated with a specific interface are declared and maintained in one place.
- Normally used to represent the signals which comprise a single instance of a protocol interface between a DUT and a testbench.

Interfaces (Cont.)

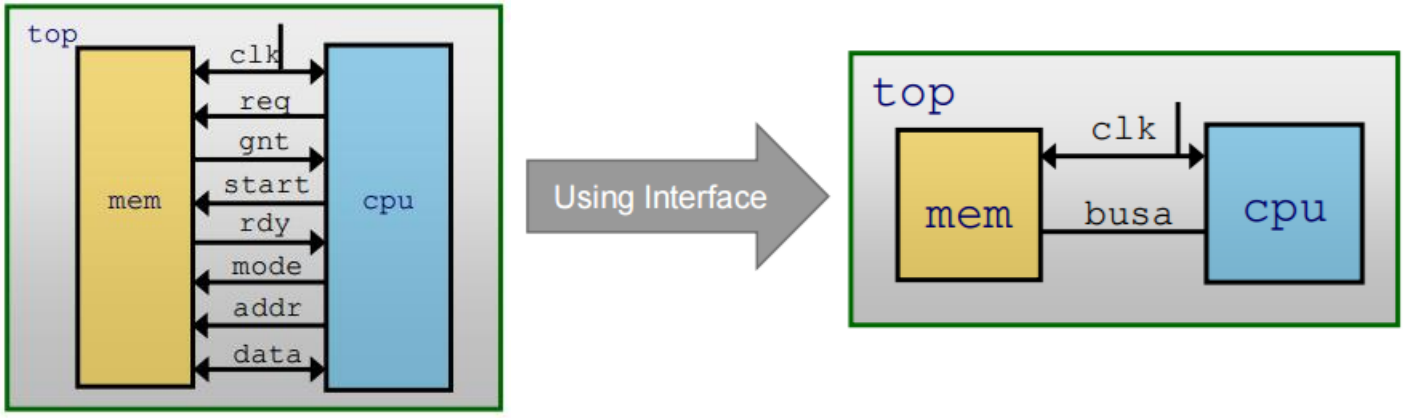
- The interface is created in a separate file and must be compiled separately by the simulator.
- A useful abstraction used during testbench definition, avoiding the need to replicate declarations for each member signal along the way.
- The interface is instantiated in a design and can be connected to interface ports of other instantiated modules, interfaces and programs

Interfaces (Cont.)



Interfaces: Motivation

- One Verilog hierarchical connection between and a CPU and memory modules requires 5 declarations:
  - Two port declarations in modules mem and cpu
  - Signal declaration in top
  - Signal added to each instantiation of mem and cpu
- **Problem:** Creating and maintaining multiple connections is tedious.



Example: Without Interface

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

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

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

memory  mem(.clk, .req, .start,
   .mode, .addr, .data, .gnt, .rdy);

cpucore cpu(.clk, .gnt, .rdy, .data,
   .req, .start, .addr, .mode);
...
```

Example: With Interface

```
interface ifa;
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire  [7:0] data;
  ...
endinterface : ifa
```

① Interface declaration

```
module memory(
  input bit clk,
  ifa bus);
...
endmodule
```

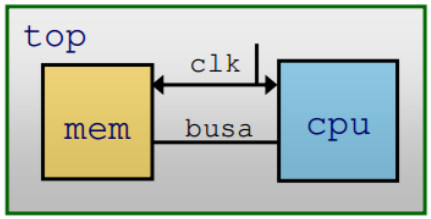
② Interface used as a directionless port type in module declarations

```
module cpucore(
  input bit clk,
  ifa bus);
...
endmodule
```

```
module top;
  logic clk = 0;

  ifa busa();

  memory  mem (clk, busa);

  cpucore cpu (clk, busa);
...
```

③ Interface instantiation

④ Instantiation mapped to module ports



More Facts on Interfaces

- A SystemVerilog interface is declared as a design element like a module.
- It can be instantiated in a module, like a module instantiation, but the interface name is also used as port type in module declarations to create interface ports
- Interfaces can also contain module-like features for defining signal relationships:
  - Continuous assignments, tasks, functions, initial/always blocks, etc.
  - Can further instantiate interfaces.
- Cannot declare or instantiate module-specific items: Modules, primitives, specify blocks, and configurations.
- Interfaces can instantiate other interfaces to create nested interface structures

Access Interface Items

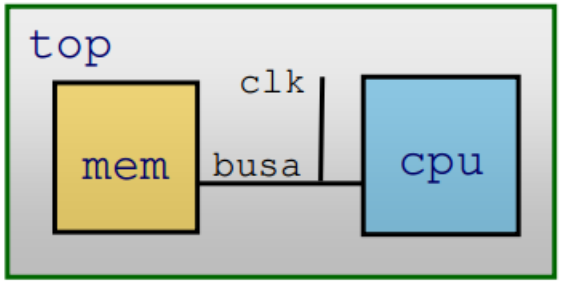
```
interface ifa;
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire  [7:0] data;
  ...
endinterface : ifa
```

```
module memory ( input bit clk, ifa bus );
  reg [31:0] mem [0:31];
  logic read, write;
  assign read  = (bus.gnt && (bus.mode == 0) );
  assign write = (bus.gnt && (bus.mode == 1) );
  always @(posedge clk)
    if (write)
      mem[bus.addr] = bus.data;
   assign bus.data = read ? mem[bus.addr] : 'z;
endmodule
```

1. Pass the Interface as port `bus`.
2. Access the Interface item using module port name.

Interface Ports

- An interface can have its own ports:
  - Connected like any module port.
  - Used to share an external signal.




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

```
module memory( ifa bus );
...
endmodule
```

```
module cpucore( ifa bus );
...
endmodule
```

```
module top;
  logic clk = 0;

  ifa busa(clk);


  memory  mem (busa);

  cpucore cpu (busa);
...
```

1. Define 'clk' port in Interface declaration.

2. Connect 'clk' port of 'top' module to Interface port during instantiation.

modport

- To restrict interface access within a module, there are **modport** lists with directions declared within the interface.

- The keyword **modport** indicates that the directions are declared as if inside the module.

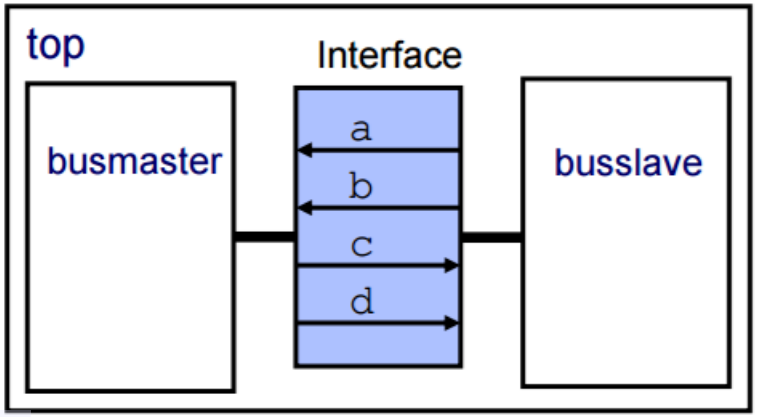
- Modports create different views of an interface.
  - Specify a subset of interface signals accessible to a module.
  - Specify direction information for those signals.
- We can specify a **modport** view for a specific module in two ways:
  - In the module declaration.
  - In the module instantiation.

Modport (Cont.)

- An interface can have any number of modports, and each defines a different view of the interface contents

- A module can specify which modport to use in its port list declaration

```
interface mod_if;
  logic a, b, c, d;
  modport master (input  a,b, output c,d);
  modport slave  (output a,b, input  c,d);
  modport subset (output a, input b);
endinterface
```



Modport (Cont.)

- The interface **mod_if** declares the following modports:
- ▪ Modport **master** which defines signals **a** and **b** as input and signals **c** and **d** as output.
- Modport **slave** which defines signals **a** and **b** as output and signals **c** and **d** as input.
- Modport **subset** which defines signal **a** as output and **b** as input. Any connections to the interface via modport subset would not be able to access signals **c** or **d**.

Selecting the Interface Modport by Qualifying the Module Port Interface Type

- An interface modport can be selected using the module declaration port of interface type.

```
interface mod_if;
  logic a, b, c, d;
  modport master (input  a,b, output c,d);
  modport slave  (output a,b, input  c,d);
endinterface
```

```
module busmaster (mod_if.master mbus);
  ...
endmodule
```

```
module busslave (mod_if.slave sbus);
  ...
endmodule
```

```
module testbench;
  mod_if busa();
  busmaster M1 (.mbus(busa));
  busslave  S1 (.sbus(busa));
  ...
endmodule
```

Selecting the Interface Modport (Cont.)

- **`busmaster`** declares interface port **`mbus`**:
  - Type is **`mod_if`**
  - Modport is **`master`**
- **`busslave`** declares interface port **`sbus`**:
  - Type is **`mod_if`**
  - Modport is **`slave`**
- **`testbench`** instantiates interface and modules as before.

Selecting the Interface Modport by Qualifying the Module Port Interface Binding

- An interface modport can be selected during the port mapping of module instantiation.

```
interface mod_if;
  logic a, b, c, d;
  modport master (input  a,b, output c,d);
  modport slave  (output a,b, input  c,d);
endinterface
```

```
module busmaster (mod_if mbus);
en
```

```
module busslave (mod_if sbus);
  ...
endmodule
```

```
module testbench;
  mod_if busb();
  busmaster M1 (.mbus(busb.master));
  busslave  S1 (.sbus(busb.slave));
  ...
endmodule
```

Interface Methods

- A sub-routine defined within an interface is called an interface method.
- We can declare tasks as part of the interface
- These tasks are accessible to any module connected to the interface

Interface Methods Example

```
module cpucore(ifa bus);
  ...
  bus.read(addr,data);
  ...
endmodule
```

```
interface ifa (input clk);
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire  [7:0] data;

  task read (input  byte raddr,
             output byte rdata);
    @(posedge clk);
    addr = raddr;
    ...
    rdata = data;
  endtask
endinterface : ifa
```