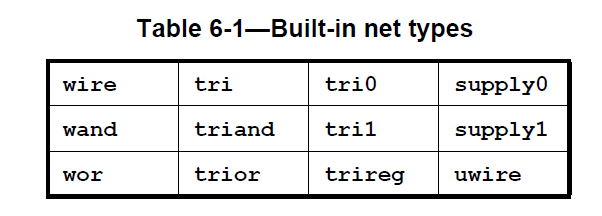
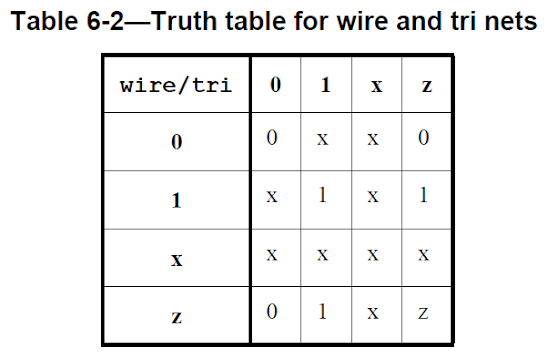
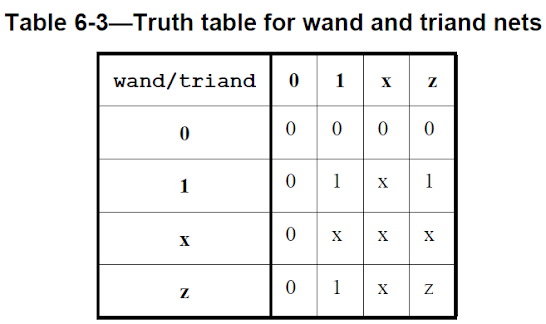
* Data Types
  + 2 main groups
    - nets
      * written by continuous assignments
      * cannot be procedurally assigned (if variable on RHS, continuous assignment implied)
      * 2 types
        + built-in
        + user defined
      * shall not store a value (except trireg)
      * Built-in net types

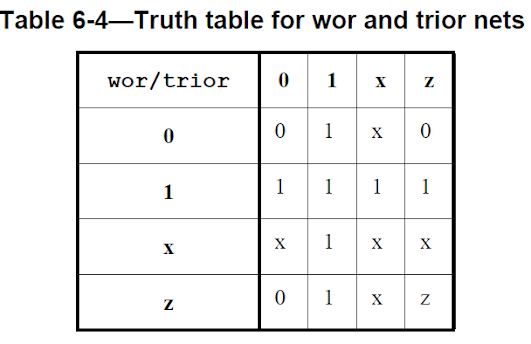


* wire and tri are identical…
  + wire net can be used for nets that are driven by a single gate or continuous assignment
  + tri net type can be used where multiple drivers drive a net.



* uwire net is an unresolved or unidriver wire and is used to model nets that allow only a single driver
  + error to connect any bit of a uwire net to more than one driver
* Wired nets are of type wor, wand, trior, and triand and are used to model wired logic configurations
  + wor/trior : wired or
  + wand/triand: wired and





* Supply nets (supply0, supply1): can be used to model the power supplies in a circuit. These nets shall have

supply strengths.

* To read: 6.6.4: trireg (net that can store value), 6.6.5: tri0/tri1
* Declaration:
  + net\_type [ drive\_strength | charge\_strength ] [ vectored | scalared ] data\_type\_or\_implicit [ delay3 ] list\_of\_net\_decl\_assignments

* Any 4-state data type can be used to declare a net, eg:

trireg (large) logic #(0,0,0) cap1; // charge strength = large

typedef logic [31:0] addressT;

wire addressT w1;

wire struct packed { logic ecc; logic [7:0] data; } memsig;

* If no data type given, defaults to logic:  
    
  wire w; // equivalent to "wire logic w;"

wire [15:0] ww; // equivalent to "wire logic [15:0] ww;"

* Valid data types:
  + A 4-state integral type, including a packed array or packed structure.
  + A fixed-size unpacked array or unpacked structure, where each element has a valid data type for a net.
* net type keyword shall not be followed directly by the reg keyword.
  + reg keyword can be used in a net or port declaration if there are lexical elements between the net type keyword and the reg keyword.
* default initialization = z (except trireg, which defaults to x)
* For interconnect: no data type, no drive strength, no charge strength, at most 1 delay value
* Initialization
* Userdefined nettype:
  + similar to typedef
  + provides a name for a particular data type and optionally an associated resolution function.
  + resolution for a user-defined nettype is specified using a SystemVerilog function declaration.
  + when a driver of the net changes value, an update event is scheduled on the net in the Active (or Reactive) region. When the update event matures, the simulator calls the resolution function to compute the value of the net from the values of the drivers
  + Two different nettypes can use the same data type, but have different resolution functions
  + If no resolution function, error to have multiple driver
  + Valid data types:
    - A 4-state integral type, including a packed array, packed structure or union
    - A 2-state integral type, including a packed array, packed structure or union with 2-state data type members.
    - A real or shortreal type
    - A fixed-size unpacked array, unpacked structure or union, where each element has a valid data type for a net of a user-defined nettype
  + Eg:

// user-defined data type T

typedef struct {

real field1;

bit field2;

} T;

// user-defined resolution function Tsum

function automatic T Tsum (input T driver[]);

Tsum.field1 = 0.0;

foreach (driver[i])

Tsum.field1 += driver[i].field1;

endfunction

nettype T wT; // an unresolved nettype wT whose data type is T

// a nettype wTsum whose data type is T and

// resolution function is Tsum

nettype T wTsum with Tsum;

// user-defined data type TR

typedef real TR[5];

// an unresolved nettype wTR whose data type

// is an array of real

nettype TR wTR;

// declare another name nettypeid2 for nettype wTsum

nettype wTsum nettypeid2;

* resolution function for any net of a user-defined nettype shall be activated at time zero at least once
* initial value of a net with a user-defined nettype shall be set before any initial or always procedures

are started and before the activation of the guaranteed time zero resolution call

* Atomic net: a net whose value is updated and resolved as a whole (eg: user defined net, logic net)
  + intended to describe a single connection point in the design

* Generic Interconnect
  + Generic form of nets
  + In order to support netlist designs, which primarily specify design element instances and the net connections between the design elements
  + A net or port declared as interconnect (an interconnect net or port) indicates a typeless or generic net
  + only able to express net port and terminal connections and shall not be used in any procedural context nor in any continuous or procedural continuous assignments
  + shall not be used in any expression other than a net\_lvalue expression in which all nets or ports in the expression are also interconnect nets.
  + interconnect array shall be considered valid even if different bits in the array are resolved to different net types
  + Eg:

package NetsPkg;

nettype real realNet;

endpackage : NetsPkg

module top();

interconnect [0:1] iBus;

lDriver l1(iBus[0]);

rDriver r1(iBus[1]);

rlMod m1(iBus);

endmodule : top

module lDriver(output wire logic out);

endmodule : lDriver

module rDriver

import NetsPkg::\*;

(output realNet out);

endmodule : rDriver

module rlMod(input interconnect [0:1] iBus);

lMod l1(iBus[0]);

rMod r1(iBus[1]);

endmodule : rlMod

* Eg: see example at end of 6.6.8

* variables
  + written by procedural statements (including procedural continuous assignments)
  + can be packed or unpacked aggregates
  + struct or array can have 1 part assigned procedurally, another continuously
    - illegal to have multiple continuous assignments to same part
    - illegal to mix continuous and procedural for same part

* continuous assignment shall be implied when a variable is connected to an input port declaration
* continuous assignment shall be implied when a variable is connected to the output port of an instance
* Variables cannot be connected to either side of an inout port
* a variable cannot have an implicit continuous assignment as part of its declaration (unlike nets)

wire w = vara & varb; // net with a continuous assignment

logic v = consta & constb; // variable with initialization

logic vw; // no initial assignment

assign vw = vara & varb; // continuous assignment to a variable

real circ;

assign circ = 2.0 \* PI \* R; // continuous assignment to a variable

* store a value from one assignment to the next
* assignment statement in a procedure acts as a trigger that changes the value in the data storage

element.

* Declaration
  + data type followed by one or more instances
    - shortint s1, s2[0:9];
  + using var keyword

var byte my\_byte; // equivalent to "byte my\_byte;"

var v; // equivalent to "var logic v;"

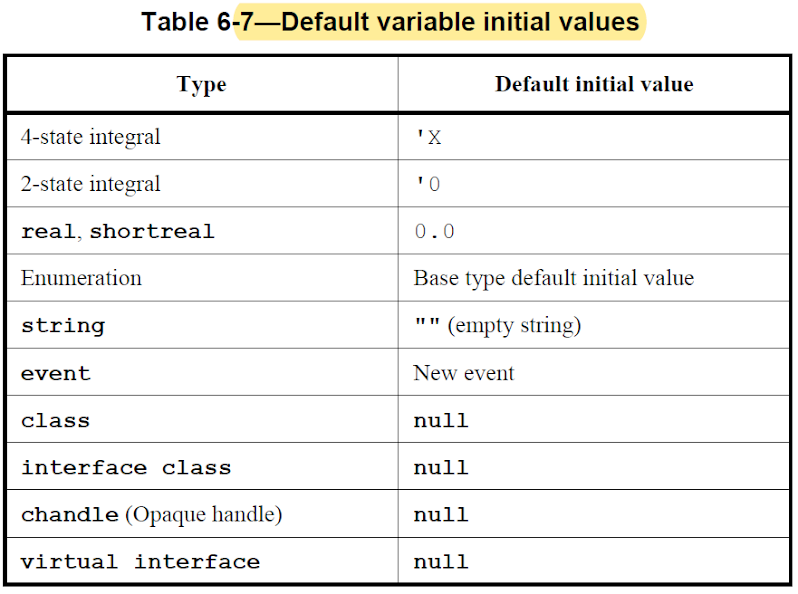
var [15:0] vw; // equivalent to "var logic [15:0] vw;"

var enum bit { clear, error } status;

input var logic data\_in;

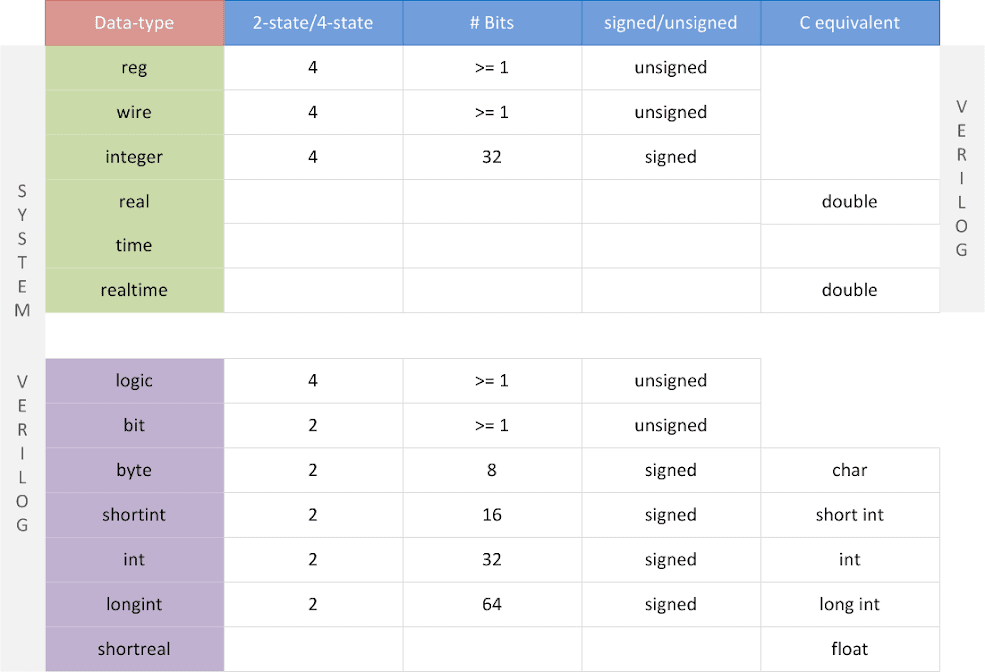
var reg r;

* can be declared with an initializer
  + int i = 0;
    - shall occur before any initial or always procedures are started
    - can include run-time expressions
* byte, shortint, int, integer, and longint types are signed types by default
* reg, logic, bit : 1 bit wide scalars

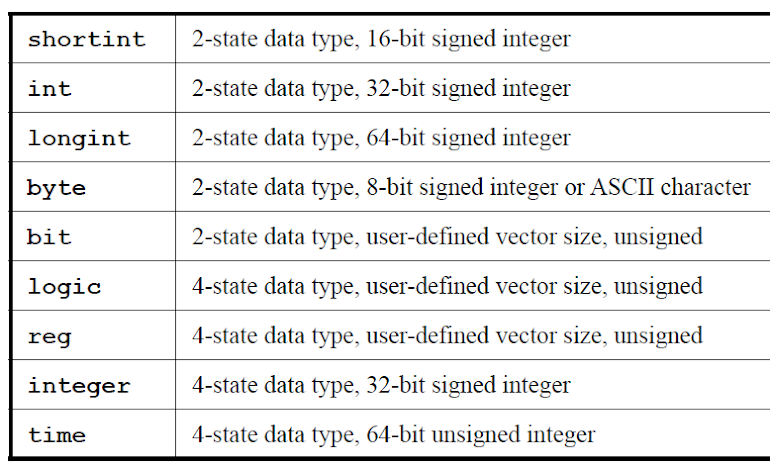


byte, shortint, int, integer, and longint types are signed types by default

* reg, logic, bit : 1 bit wide scalars



Integer Data Types (first read verilog basics page):



Notice difference b/w int and integer

* Signedness can be explicitly defined:  
  int unsigned ui;

int signed si;

* Integral types are always singular (non-aggregate)

* Logic
  + Can be driven in both procedural blocks and using continuous assign statements
    - In verilog, reg can be driven only in procedural block and wire only using assign
  + Supports 4 states

module tb;  
  logic [3:0]  my\_data; // Declare a 4-bit logic type variable  
  logic en; // Declare a 1-bit logic type variable

  initial begin  
    $display ("my\_data=0x%0h en=%0b", my\_data, en); // Default value of logic type is X  
    my\_data = 4'hB; // logic datatype can be driven in initial/always blocks  
    $display ("my\_data=0x%0h en=%0b", my\_data, en);  
    #1;  
    $display ("my\_data=0x%0h en=%0b", my\_data, en);  
  end

  assign en = my\_data[0];         // logic datatype can also be driven via assign statements  
endmodule

* Bit
  + 2 state data type (0 or 1) -> faster simulation, less memory

module tb;  
  bit       var\_a;  // Declare a 1 bit variable of type "bit"  
  bit [3:0] var\_b;  // Declare a 4 bit variable of type "bit"

  logic [3:0] x\_val;  // Declare a 4 bit variable of type "logic"

  initial begin

    // Initial value of "bit" data type is 0  
    $display ("Initial value var\_a=%0b var\_b=0x%0h", var\_a, var\_b);

    // Assign new values and display the variable to see that it gets the new values  
    var\_a = 1;  
    var\_b = 4'hF;  
    $display ("New values    var\_a=%0b var\_b=0x%0h", var\_a, var\_b);

    // If a "bit" type variable is assigned with a value greater than it can hold  
    // the left most bits are truncated. In this case, var\_b can hold only 4 bits  
    // and hence 'h481 gets truncated leaving var\_b with only 'ha;  
    var\_b = 16'h481a;  
    $display ("Truncated value: var\_b=0x%0h", var\_b);

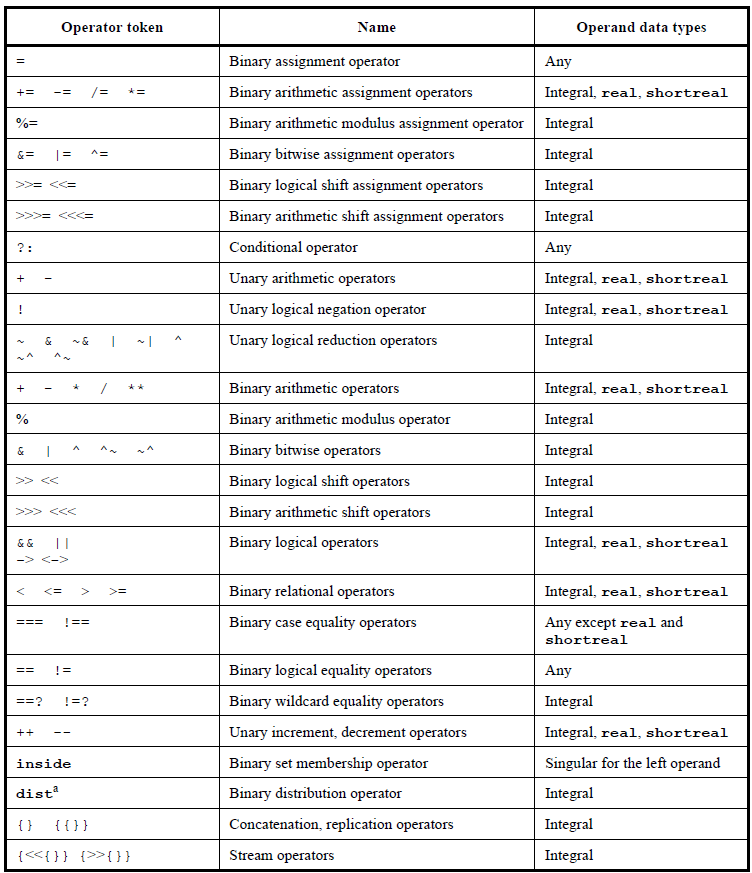
    // If a logic type or any 4-state variable assigns its value to a "bit" type  
    // variable, then X and Z get converted to zero  
    var\_b = 4'b01zx;  
    $display ("var\_b = %b", var\_b);  
  end  
Endmodule

* **Real Variables:** real, shortreal, and realtime
  + real -> same as C double
  + shortreal -> same as C float
  + realtime : synonymous with real
  + real variables and constants prohibited for following cases:
    - Edge event controls (posedge, negedge, edge) applied to real variables
    - Bit-select or part-select references of variables declared as real
    - Real number index expressions of bit-select or part-select references of vectors
  + Conversion
    - converted to integer type by rounding to nearest integer (0.5 rounded up)
    - implicit conversion when real no assigned to int
    - explicit conversion can be done using casting or system tasks

* A signal with more than one driver needs to be a wire type
* 2 state types require less memory, simulate faster

* string data type is always singular

* **Operators and Data Types**



**Vectors**

* + Examples

wand w; // a scalar "wand" net

tri [15:0] busa; // a 16-bit bus

trireg (small) storeit; // a charge storage node of strength small

logic a; // a scalar variable

logic[3:0] v; // a 4-bit vector made up of (from most to

// least significant)v[3], v[2], v[1], and v[0]

logic signed [3:0] signed\_reg; // a 4-bit vector in range -8 to 7

logic [-1:4] b; // a 6-bit vector

wire w1, w2; // declares two nets

logic [4:0] x, y, z; // declares three 5-bit variables

* **Type Compatibility**
  + **Assignment Compatibility**
    - All equivalent types are assignment compatible
    - Non-equivalent types with implicit casting defined are assignment compatible
      * implicit casting only works in one direction, eg: enum to int but not int to enum
    - Conversion may lead to loss of data by truncation or rounding
* **Assignment Statements**
  + **Assignment-like Contexts**
    - Following are assignment-like contexts:
      * Continuous/procedural assignment
      * For a param with an explicit type declaration:
        + Param value assignment in module/interface/program/class
        + Param value override in module instantiation/interface/
        + Param value override in class instantiation/LHS of class scope resolution operator
      * Port connection to input/output port of module/interface/program
      * Passing of value to subroutine input/output/inout port
      * Return statement
      * Tagged union expression
      * For an expression used as RHS in assignment-like context:
        + Expression within parentheses
        + Colon-separated expressions in a mintypemax expression
        + Second and third operand in a conditional operator expression
    - Following ARE NOT assignment-like contexts
      * Static cast
      * Default correspondence b/w an expression in an assignment pattern and a field/element in a data object/data value
      * Port expression in module/interface/program declaration
      * Passing of a value to a subroutine ref port
      * Port connection to inout/ref port of a module/interface/program
  + **Assignment Patterns**
    - Used for assignments to describe patterns of assignments to structure fields and array elements
    - Specifies correspondence b/w a collection of expressions and fields/elements in a data object/data value
    - Has no self-determined data type, but can be used as one side in an assignment-like context when other side has a self-determined type
    - Built from braces, keys, and expressions
    - Prefixed with apostrophe
    - Eg:  
      var int A[N] = '{default:1};

var integer i = '{31:1, 23:1, 15:1, 8:1, default:0};

typedef struct {real r, th;} C;

var C x = '{th:PI/2.0, r:1.0};

var real y [0:1] = '{0.0, 1.1}, z [0:9] = '{default: 3.1416};  
var int B[4] = '{a, b, c, d};

var C y = '{1.0, PI/2.0};

'{a, b, c, d} = B;

* Can be used to construct/deconstruct a structure or array by prefixing the pattern with the name of the data type to form an *assignment pattern expression* 
  + Has self-determined data type
  + Not restricted to being one side of an assignment-like context
  + When used in RHS, yields the value that a variable would if initialized using assignment pattern
  + Eg:  
    typedef logic [1:0] [3:0] T;

shortint'({T'{1,2}, T'{3,4}}) // yields 16'sh1234

* When used in LHS, positional notation required
* Eg:

typedef byte U[3];

var U A = '{1, 2, 3};

var byte a, b, c;

U'{a, b, c} = A;

U'{c, a, b} = '{a+1, b+1, c+1};

* Cannot be used in port expression in module/interface/program
* Array Assignment Patterns
  + Each element must match properly
  + Braces must match array dimensions
  + Eg:  
    bit unpackedbits [1:0] = '{1,1}; // no size warning required as

int unpackedints [1:0] = '{1'b1, 1'b1}; // no size warning required as

* Replication like syntax can be used:  
  unpackedbits = '{2 {y}} ; // same as '{y, y}

int n[1:2][1:3] = '{2{'{3{y}}}}; // same as '{'{y,y,y},'{y,y,y}}

* Can set array elements to a value without keeping track of size using **default:**initial unpackedints = '{default:2}; // sets elements to 2
* Array of structures:  
  struct {int a; time b;} abkey[1:0];

abkey = '{'{a:1, b:2ns}, '{int:5, time:$time}};

* Index:value: Cannot specify same index more than once in a single array pattern expression
* Type:value: If more than one type matches same element, last one used
* default:value:
* Structure Assignment Patterns
  + To construct/deconstruct a structure
  + Members should be in declaration order
  + Can use replication operator
  + Eg:  
    module mod1;

typedef struct {

int x;

int y;

} st;

st s1;

int k = 1;

initial begin

#1 s1 = '{1, 2+k}; // by position

#1 $display( s1.x, s1.y);

#1 s1 = '{x:2, y:3+k}; // by name

#1 $display( s1.x, s1.y);

#1 $finish;

end

endmodule

* Using default:  
  initial s1 = '{default:2}; // sets x and y to 2
* Using member:value or type:value syntax:  
  ab abkey[1:0] = '{'{a:1, b:1.0}, '{int:2, shortreal:2.0}};
* Default applies to members of nested structures:  
  struct {

int A;

struct {

int B, C;

} BC1, BC2;

} ABC, DEF;

ABC = '{A:1, BC1:'{B:2, C:3}, BC2:'{B:4,C:5}};

DEF = '{default:10};

* For members of different types, type used as key:  
  typedef struct {

logic [7:0] a;

bit b;

bit signed [31:0] c;

string s;

} sa;

sa s2;

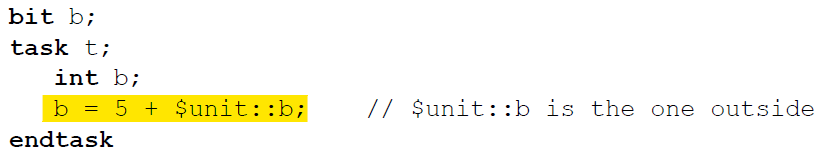
initial s2 = '{int:1, default:0, string:""}; // set all to 0 except the

                                             // array of bits to 1 and

* Setting individual member to override default:  
  initial #10 s2 = '{default:'1, s : ""}; // set all to 1 except s to ""
* Matching rules
  + member:value
    - Member must be at top level
  + Type:value
    - Specifies value for each field whose type matches (and not set by member:value)
    - If mutltiple, last one used
  + Default:value
    - For members that don't match by name or type
  + Every member must be covered by one of the rules

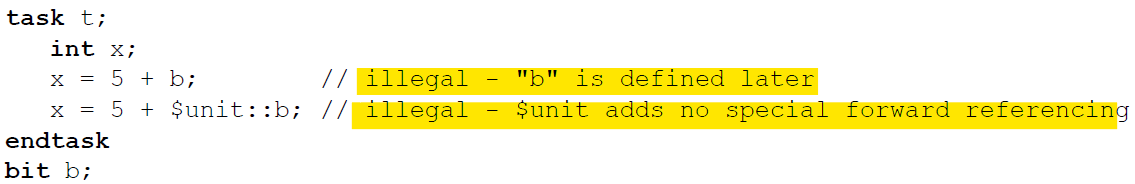
* **Constant Expressions**
  + Operands can be:
    - constant numbers
    - strings
    - parameters
    - constant bit-selects/part-selects of parameters
    - constant function calls
    - constant system function calls: calls to certain built-in functions where arguments are constants
      * evaluated at elaboration time
      * pure functions, i.e. value depends on input only, no side effects
      * eg: conversion functions, mathematical system functions, bit vector system functions
    - LRM Section 11.2.1
  + Can use any operator

* **Compilation and Elaboration**
  + *Compilation*: process of reading in source code, decrypting encrypted code, and analysing source code for syntax and semantic errors. Not all syntax/semantics can be checked at this phase.
  + *Elaboration*: Process of binding together the components that make up the design. Involves:
    - Expanding instantiations
    - Computing parameter values
    - Resolving hierarchical names
    - Establishing net connectivity
    - Preparing design for simulation
  + Compilation in general refers to any phase before simulation
  + *Compilation Unit (CU):*  A collection of 1 or more source files compiled together
  + *Compilation-unit Scope:* The scope local to the CU. Contains all declaration that lie outside any other scope.
  + *$unit:* Name used to explicitly access identifiers in the CU scope., Eg:



2 use models

* + All files in a compilation command form 1 CU (declarations accessible from all files)
  + Each file forms its own CU (only local declarations accessible)
* Contents of file included using `include become part of including file's CU
* Following items visible in all CUs:
  + Modules
  + Primitives
  + Programs
  + Interfaces
  + Packages
* Other than task and function names, references should be to items already defined in the CU



* Namespaces
  1. *Definitions:* unifies all non-nested modules, primitives, program and interface identifiers outside all other declarations. Can't have 2 of these with same name
  2. *Package:* Unifies package definitions among all CU's. (package names should be unique across CU's)
  3. *Compilation-unit Scope:* Outside the module, primitive, interface, package, checker, and program constructs  
     Unifies definitions of functions, tasks, checkers, parameters, named events, net declarations, variable declarations, and user-defined types
  4. *Text Macro:* Global within the CU. Text macro names introduced and used using a leading ` character. Subsequent definition of the same name override previous definitions.
  5. *Module:* Introduced by the module, interface, program, package, checker and primitive constructs. Unifies modules, interfaces, programs, packages, checkers, functions, tasks, named blocks, instance names, parameters, named events, net declarations, variable declarations, and user-defined types within the enclosing construct
  6. *Block:* introduced by named or unnamed blocks, specify, function, and task constructs. Unifies definitions within the enclosing construct
  7. *Port:* Introduced by the module, interface, primitive and program constructs. Provides a means of structurally defining connections between objects in 2 different namespaces. Port name space overlaps the module and block name spaces.
  8. *Attribute:* Enclosed by the (\* and\*) constructs attached to a language element

* **Module (First Read Verilog Page)**
  + Basic building block
  + Encapsulates data, functionality, timing of digital hardware objects
  + Can represent low level design (eg AND gate) or entire complex digital system
  + Can represent function and timing at detailed level, abstract level or mix of both
  + Can instantiate other design elements, creating a *hierarchy*
  + macromodule keyword interchangeable with module
  + Header includes name, port list, port size and direction, parameter constants, package import list, default lifetime (static/automatic) of subroutines
  + 2 header styles:
    - non-ANSI:

module\_name ( port\_list ) ;

parameter\_declaration\_list

port\_direction\_and\_size\_

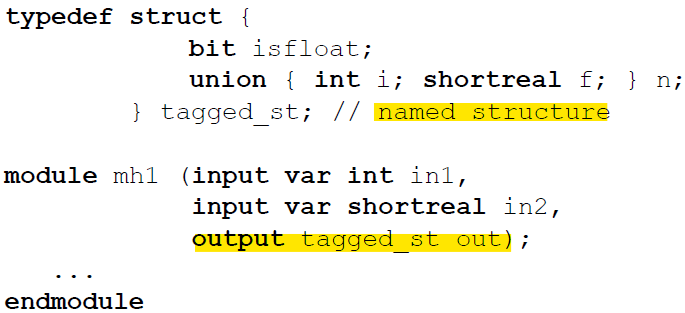
port\_type\_declarations

* ANSI

module\_name #( parameter\_port\_list )

( port\_direction\_and\_type\_list ) ;

* Port Declarations
  + Can be an interface, event, variable, net of any allowed type including array, structure or union
  + Eg:

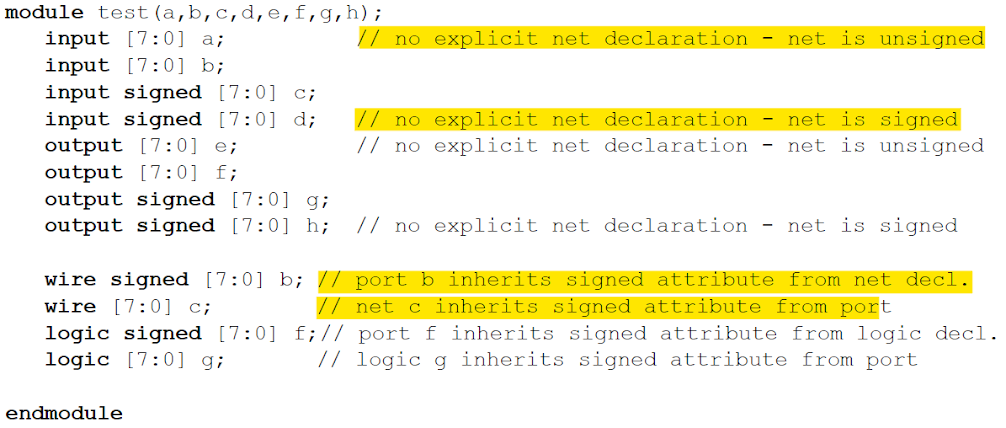


Can be Non-ANSI or ANSI style:

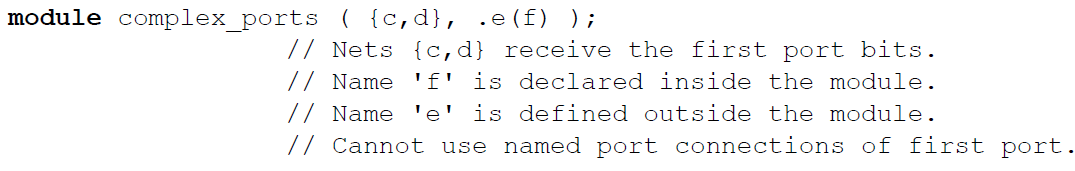
**input** Clk;  
**output** [7:0] Q;  
**input wire** Clk;                            
**output reg** [7:0] Q;                          // Verilog-2001

**module** Cnt (**output reg** [7:0] Q,  
            **input wire** Clk, Reset, Enable,  
            **input wire** [7:0] D );            // Verilog-2001 ANSI-style

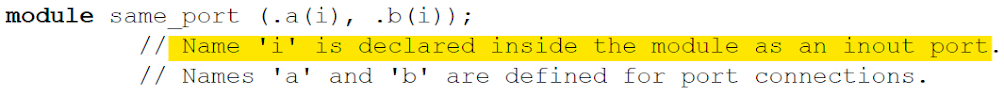
* Nets connected to ports without explicit 'net' declaration are considered unsigned
* illegal to specify an interconnect port as signed
* Output ports can only drive a net, not variable
* **Non-ANSI Examples:**
  1. implicitly named ports connected to internal nets or variables of same name

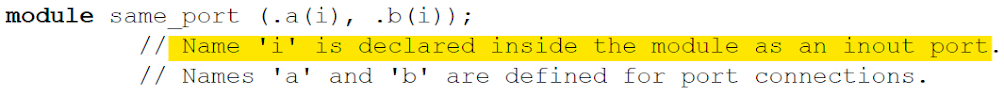


1. Ports connected to internal nets of different name

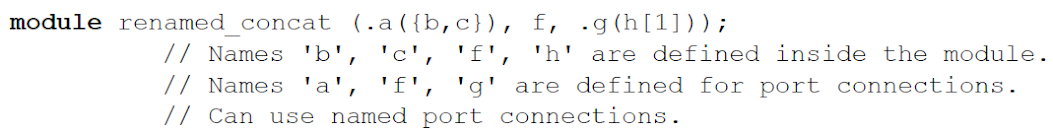


1. Ports Connected to Split of Internal Vector

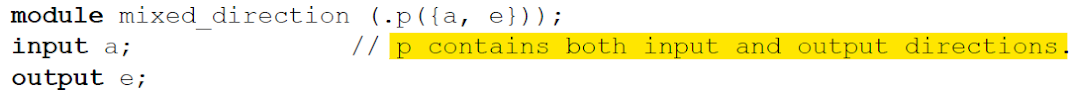


1. Different named ports connected to same internal net  
    

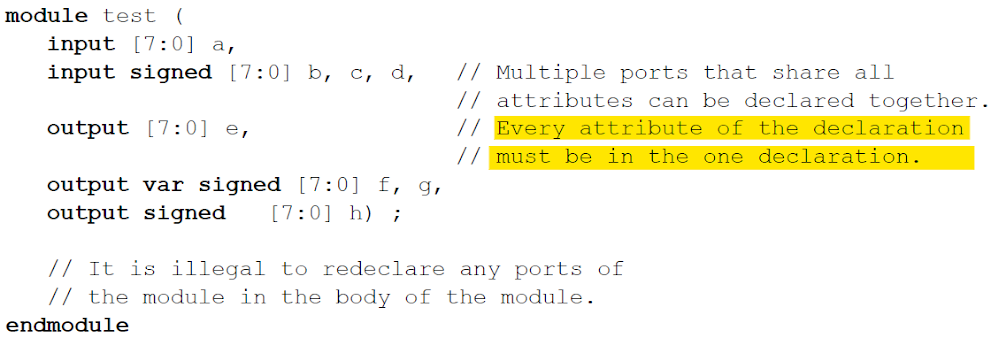
1. Explicitly named port connected to concatenation of internal nets/variables



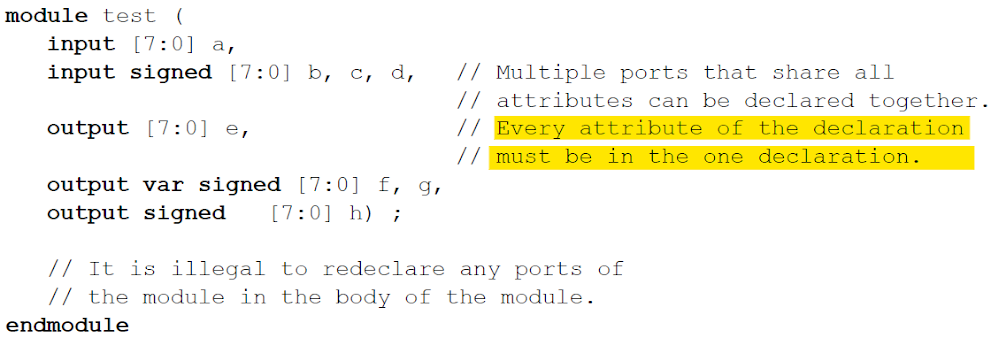
1. Implicitly named ports with same internal net



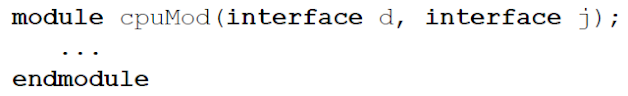
1. Explicitly named ports with mix of input and output directions



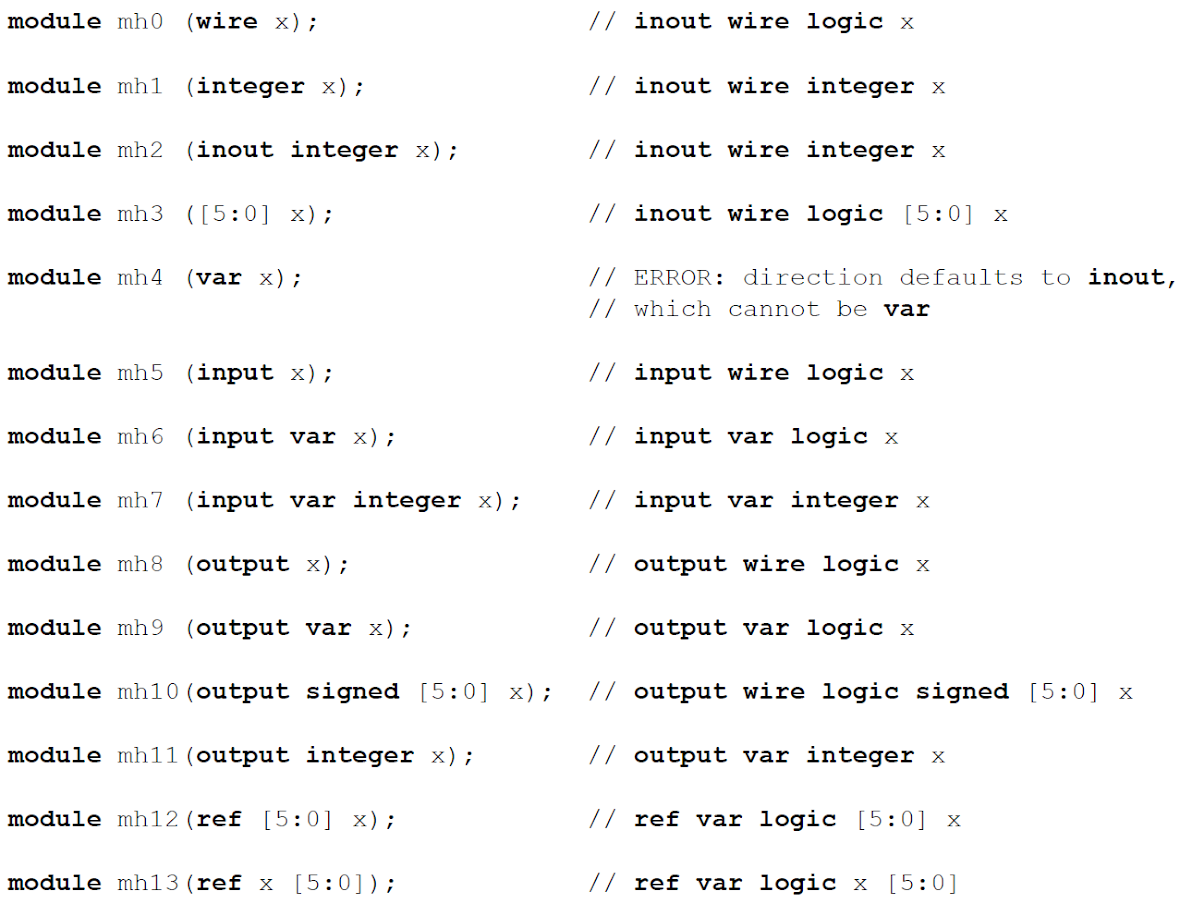
* **ANSI style Examples (no redeclarations)**

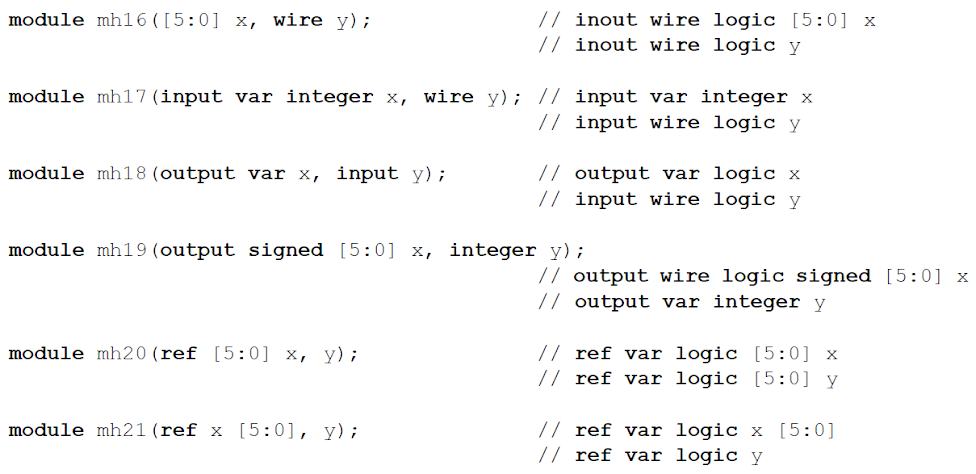


Generic Interface ports (only possible using ANSI style)



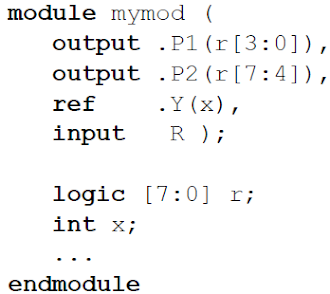
* If port direction, kind and data type all omitted, assume non-ANSI style (direction and type redeclarations)
* Else, ANSI style
  + Rules for first port
    - if direction omitted, default to inout
    - if data type omitted, default to logic except interconnect ports (no data type)
    - if kind omitted
      * for input and inout, default to net of default net type
      * for output ports
        + read LRM section 23.2.2.3
      * …
    - …
    - Examples:



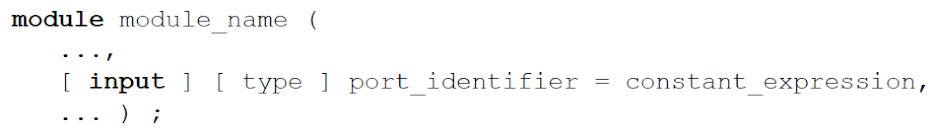
* For subsequent ports
  + if direction, port, type omitted, inherit from previous port
  + Otherwise
    - if direction or kind omitted, inherit from previous
    - if data type omitted, default to logic
  + Examples:
  + 

Above rules don't apply to explicit port declarations (.port\_identifier(expression)

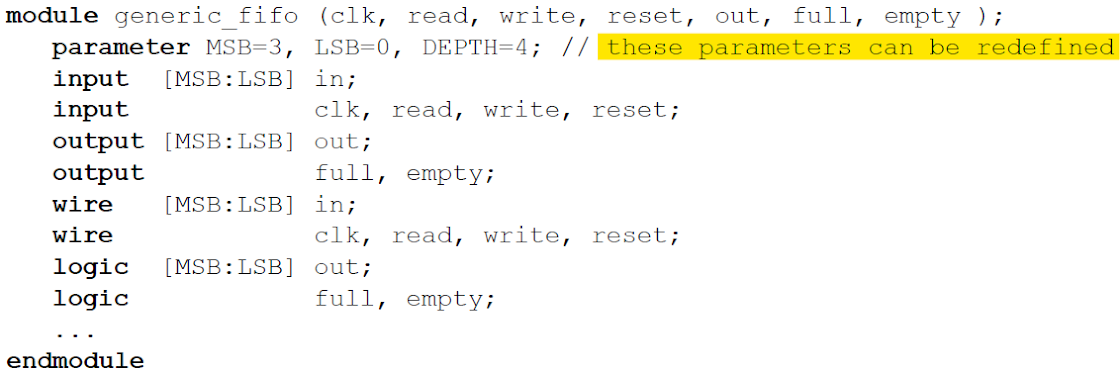
* + Example of ANSI style explicit declaration:



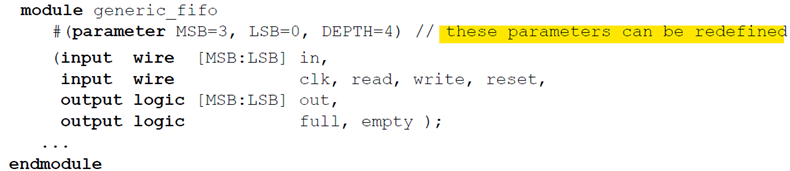
* More details: LRM Sec 23.2.2.3
* Default port values
  + Only applicable to input ports
  + Only applicable to ANSI style

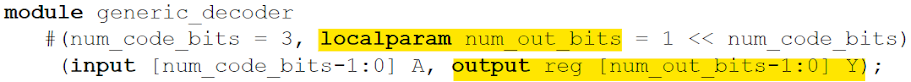


* Parameterized Modules
  + non-ANSI style:

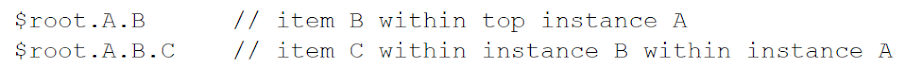


* ANSI Style:

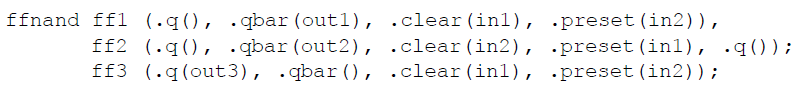




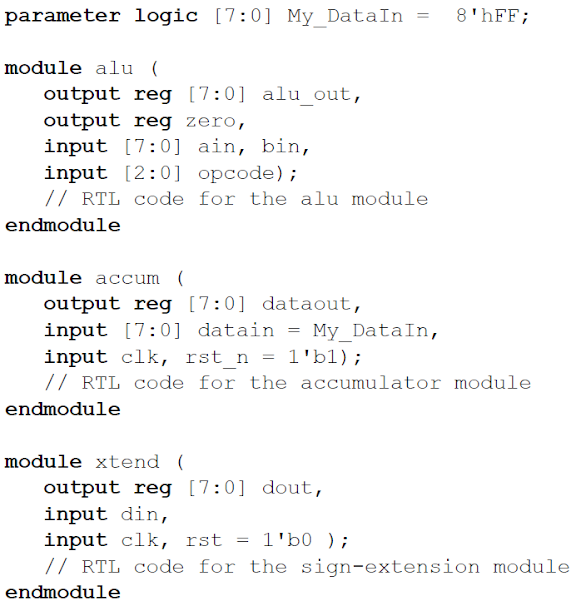
* Module Instances (hierarchy)
  + 2 ways of instantiation
    - Top level - implicitly
    - Hierarchical - implicitly (as a nested module) or explicitly
  + Top Level
    - included in source text, not appearing in instantiation statement
    - Design contains at least 1 top level module
    - Instantiated once, instance name is same as module name (aka top-level instance)
  + $root
    - Unambiguously refers to top level instance or to the root of instantiation tree
    - Eg:



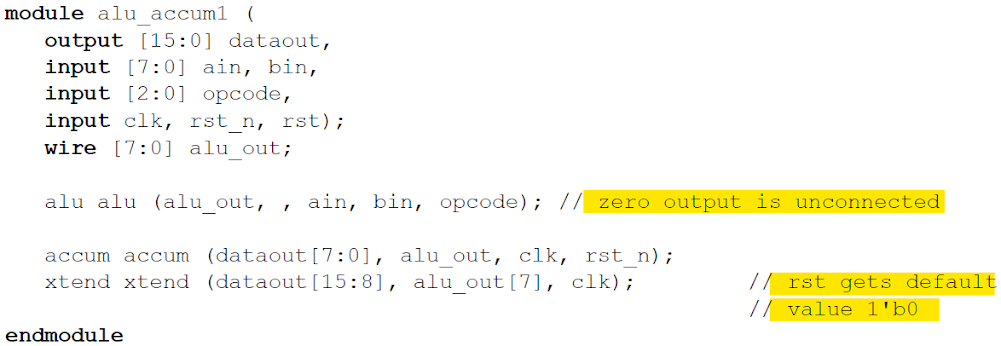
* Without $root, hierarchical path is ambiguous (A.B.C could mean local A.B.C or top-level A.B.C)
  + local scope given priority
* Explicit Instantiation
  + Multiple instances in one statement:



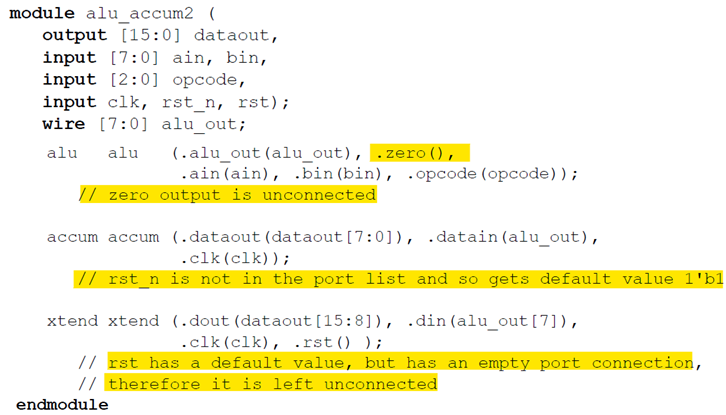
* Ways to connect module instances   
  Example Module definition:



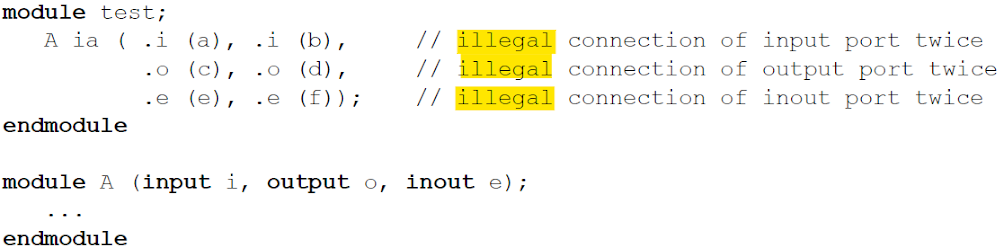
Positional connections by port order



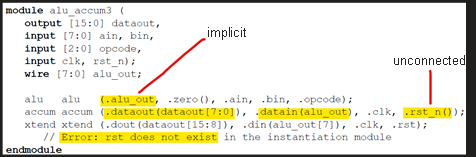
1. Named port connections using fully explicit connections



* order doesn't matter
* multiple port connections not allowed:



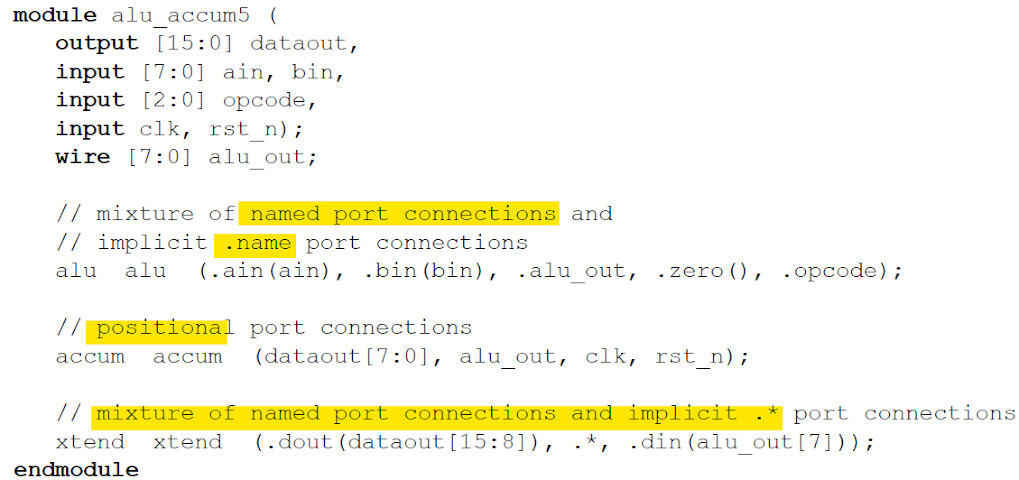
Named port connections using implicit connections



1. Named port connections using a wildcard port name

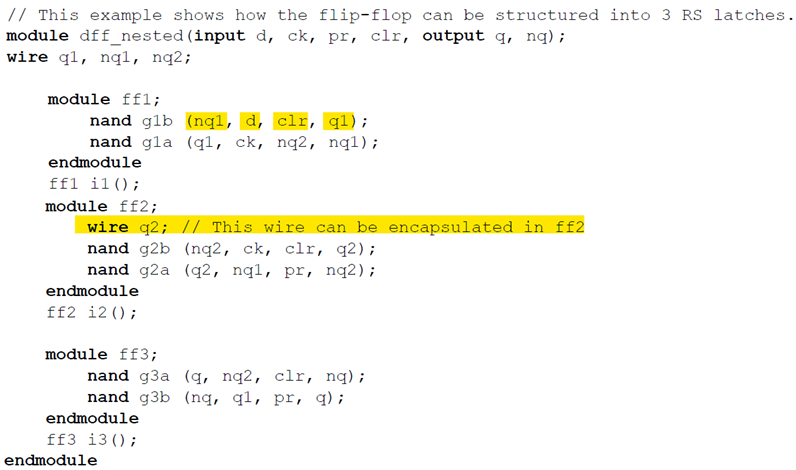


1. Mix of different types:

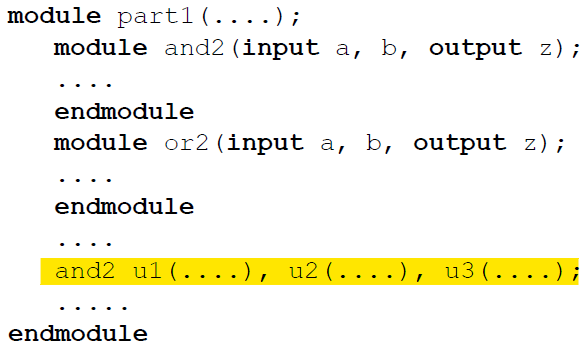


*Note: .\* can appear only once in the port list*

* Port Connection Rules
  + Port connection = continuous assignment of signal source to signal sink
  + Non-strength reducing transistor connection for inout ports
  + Same rules as assignment compatibility (see previous notes)
  + Read 23.3.3 LRM for details
* Nested Modules
  + Have access to outer level names, unless hidden.
  + Purpose: to show logical partition of module w/o using ports Eg:

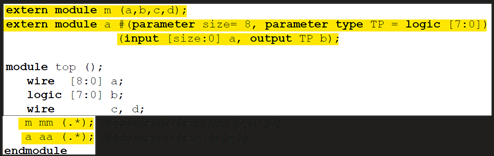


* Can be used to create a library of modules local to the outer module:

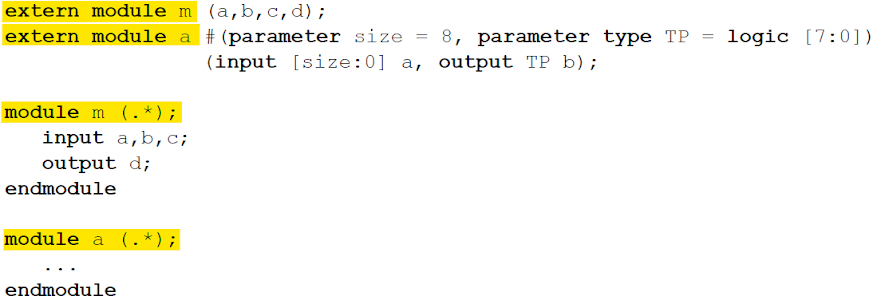


Nested modules with no ports and no explicit instantiations are implicitly instantiated once. Instance name identical to module name

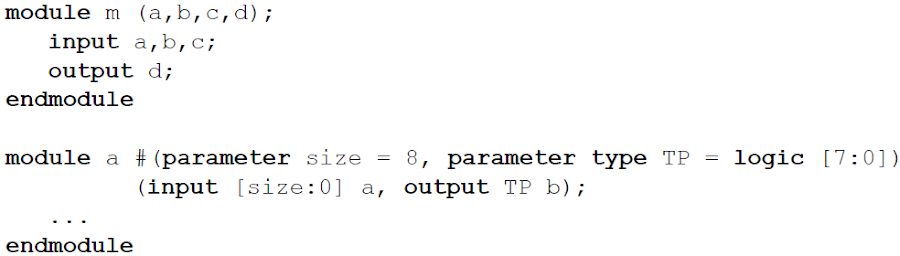
* Extern Modules
  + Can be used to declare ports without defining the module
  + Both non-ANSI and ANSI styles allowed
  + Only visible within the hierarchy level where declared
  + Eg:



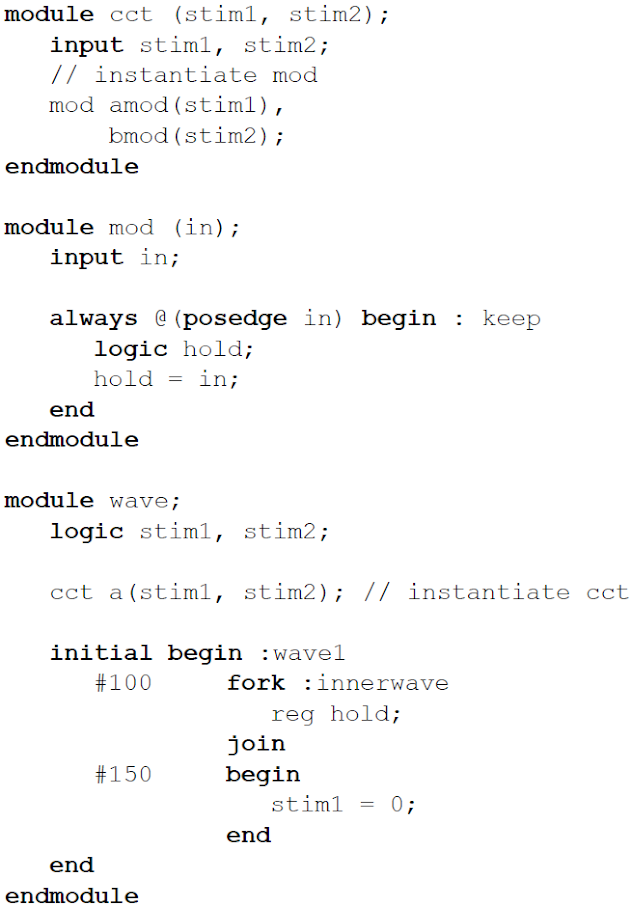
* If an extern declaration for a module exists, can use .\* as ports of the modules:



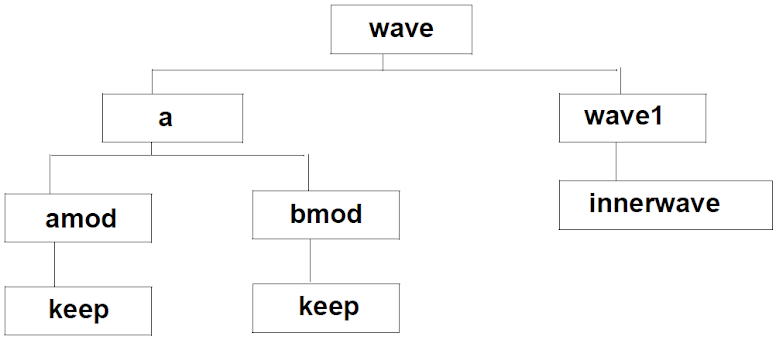
the above is equivalent to:



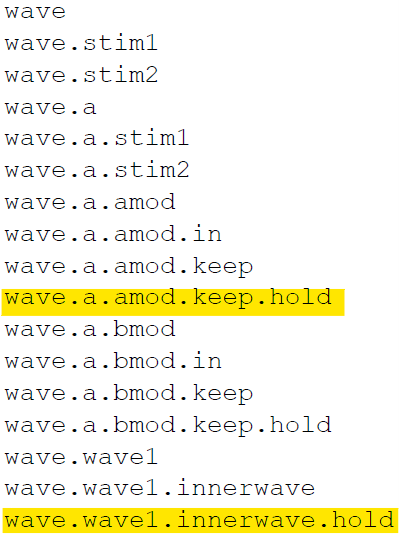
* Hierarchical Names
  + Every identifier has a unique hierarchical path name
  + Hierarchy can be viewed as a tree structure
    - Each module instance, generate block instance, task, function, named begin-end, fork-join block defines new hierarchical level, aka *scope*
  + Each top module forms top of a name hierarchy
    - Lead to one hierarchy each in a *design description* or *description*
    - Each scope element (see above) forms a new branch in the hierarchy
  + Each node in a hierarchical name tree is a separate scope wrt identifiers
    - Identifier can be declared at most in 1 scope
  + Complete path name (from top) to any object is accessible from any level in the hierarchy or from a parallel hierarchy
    - Can be sampled/changed from anywhere in the description
  + Objects in automatic subroutines and unnamed generate blocks not accessible from outside
  + Can be used to reference subroutine names
  + Example:  
    *HDL:*



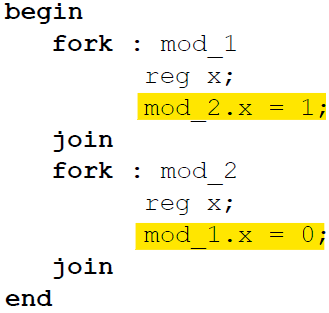
*Corresponding Hierarchy:*



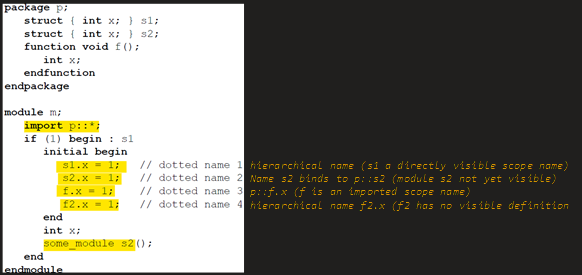
*Available Hierarchical Names (can be preceded  by $root)*



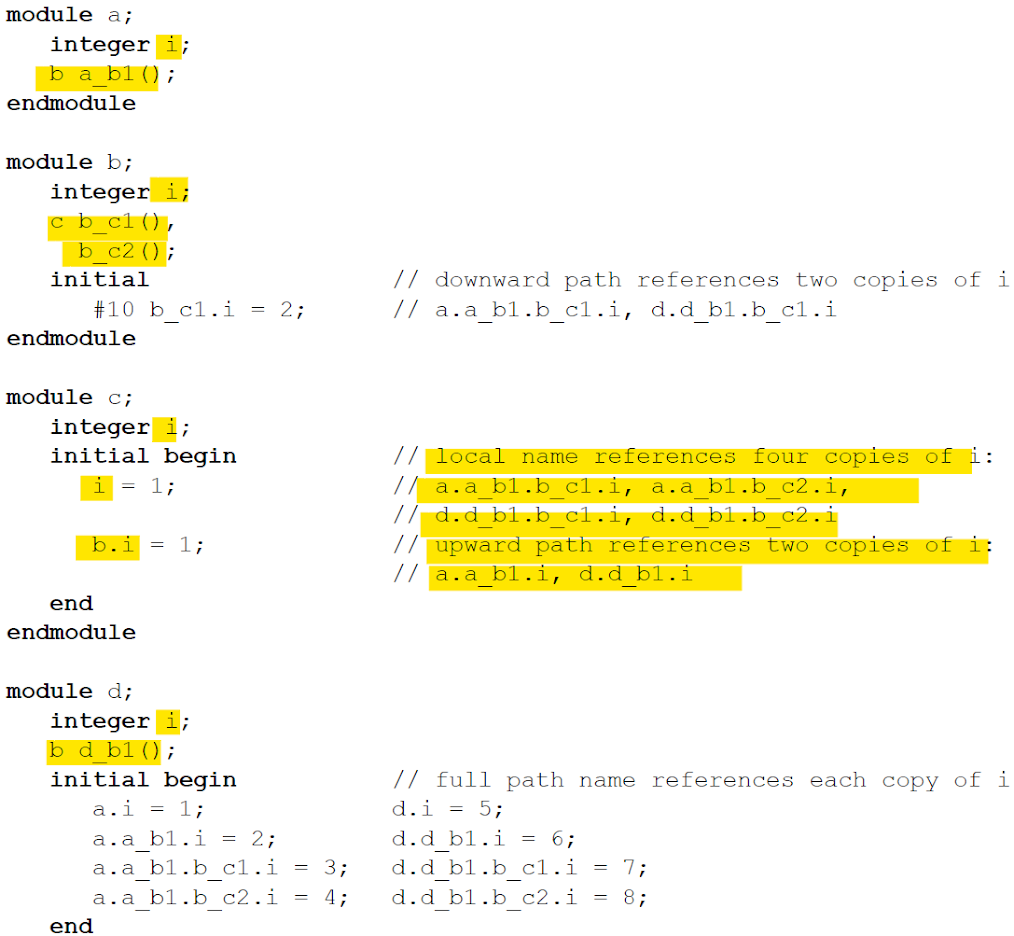
Eg: Accessing objects of a parallel hierarchy:



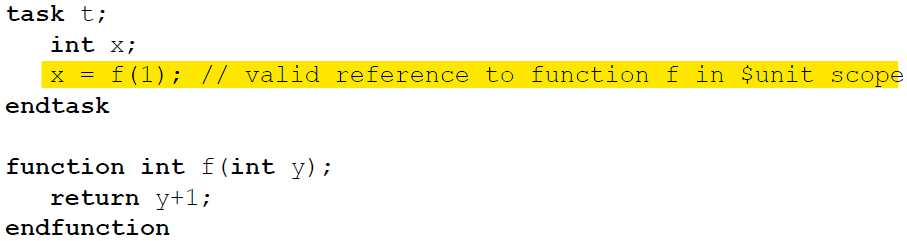
* Objects in unnamed blocks cannot be referred to
* Member Select and Hierarchical Names
  + Share same syntax form (separated by dot)



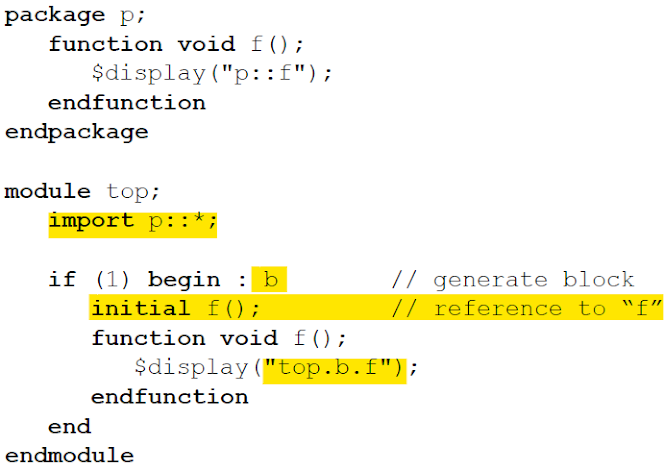
* Upwards name referencing done to find only names in enclosing modules and not module instances



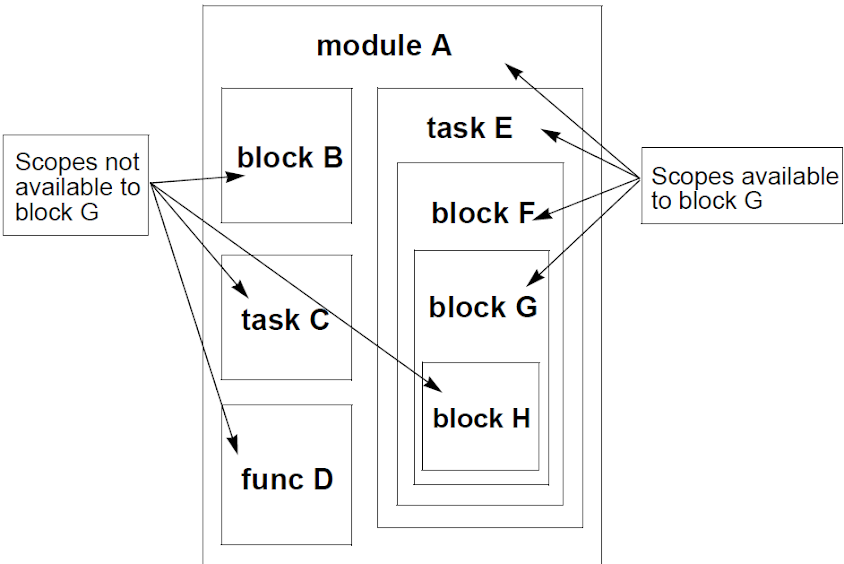
* Task/Function Name Resolution
  + Example 1:



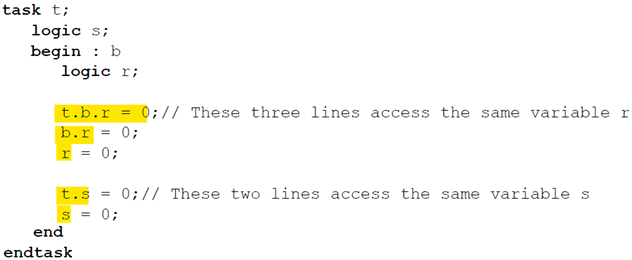
Example 2:  
The output here is top.b.f (not p::f) as f is resolved first



* Scope Rules
  + Following elements define new scope
    1. Modules
    2. Interfaces
    3. Programs
    4. Checkers
    5. Packages
    6. Classes
    7. Tasks
    8. Functions
    9. begin-end blocks
    10. fork-join blocks
    11. Generate blocks
  + If an identifier is referenced directly (w/o hierarchical path) must be declared locally in the scope or in the scope higher in the same branch of the name tree
    - First searched for locally then iteratively in the higher scope until found or a module, interface, program or checker boundary is reached
  + If referenced using hierarchical name, first searched locally then in higher scopes until found
  + Example:

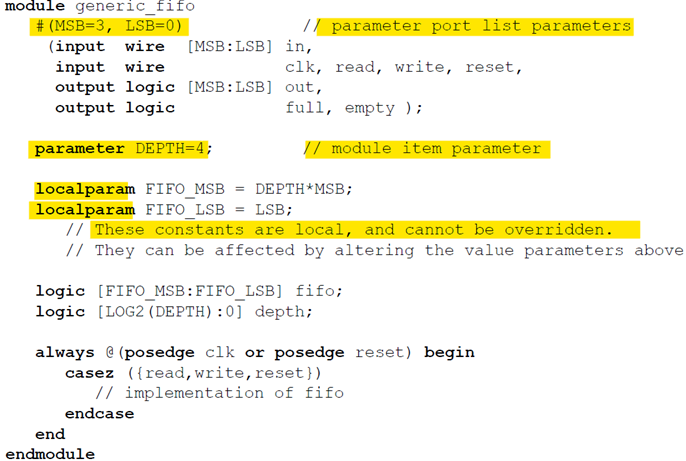


* Example:



Overriding Module Parameters

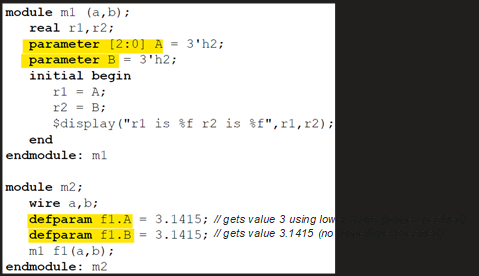
* + 2 types of params
    - value
    - type
  + Places where param can be defined
    - port list
    - module item
  + Example:



* Ways to alter parameters
  1. *defparam* (using hierarchical names)
  2. *module instance parameter value assignment* (during module instantiation)
     1. by ordered list
     2. by name

Cant mix the above two

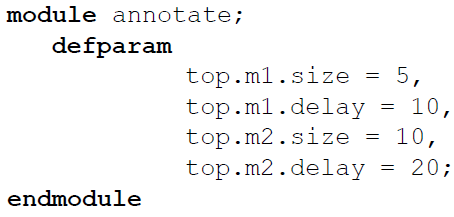
* If param defined in a subroutine or named block, can only be overridden using defparam
* Defparam takes precedence over module instance param
* Local params can't be overridden
* Value parameter
  + has type and range
  + if no type and range specified, default to final override assignment
  + If type or range specified, override value gets converted to that type and/or range
  + Example:



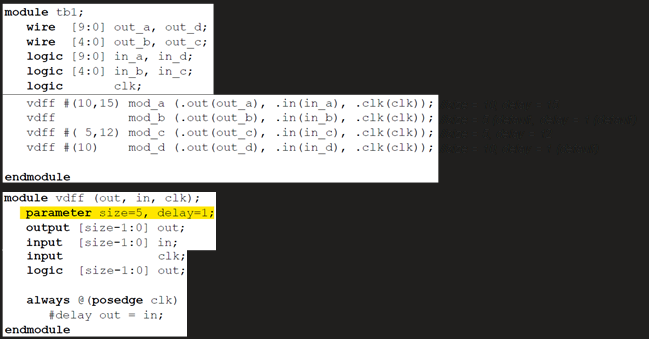
* Defparam
  + if in a hierarchy in or under a generate block, cannot override params outside that hierarchy
  + if in a generate block, can't override a param in another instance of the same generate block even if generated from same loop. So following code is not allowed:



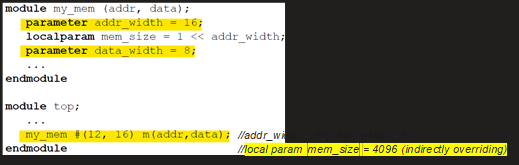
* If in an array of instances, cannot override param in another instance of the same array
* RHS of must be constant expression involving only numbers or refs to params in the same module
* Useful for grouping of param overriding statements, eg:



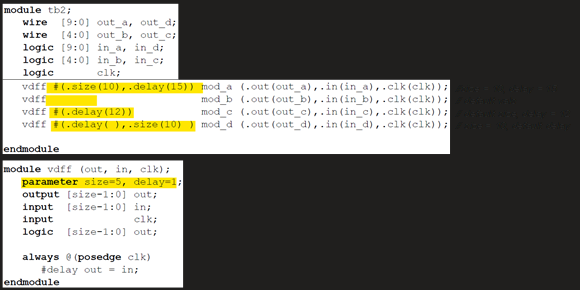
* Since defparam can override param values from almost anywhere, final module hierarchy depends on order in which defparams and generate constructs are evaluated
* Parameter Value Assignment by Ordered List
  + order should be same as module declaration
  + Eg:



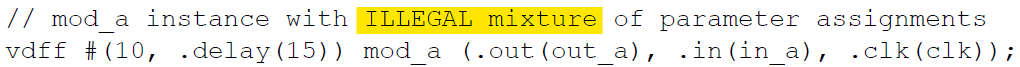
* Eg:



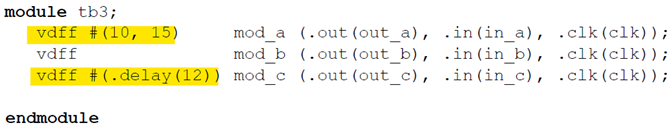
* Parameter Value Assignment by Name
  + Explicitly link param name to new value
  + Name must match the instantiated module
  + All params need not be overridden
  + Order doesn't matter
  + Eg:



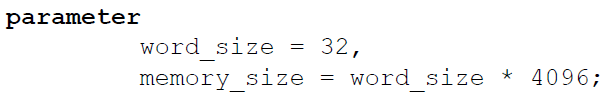
Can't mix assignment style in the same top level module, eg:



* But can have different styles in different top level modules, eg:

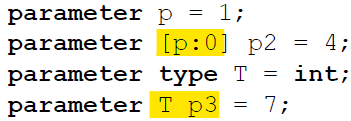


* A parameter can be assigned using other parameter, but defparam can update this, eg:



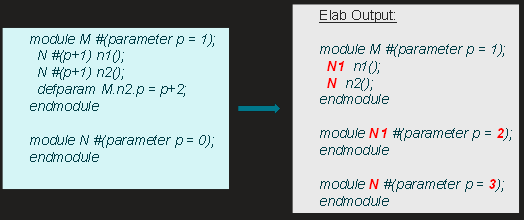
In the above example, if word\_size is overridden, memory\_size gets overridden too. But if memory\_size is then updated using defparam, it takes the defparam value irrespective of word\_size value

* Parameters can have type dependencies, eg:

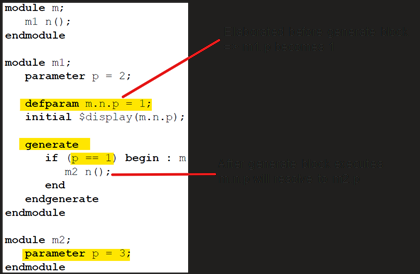


Order of Elaboration:

* + Result of elaboration can be ambiguous when generate constructs are involved as defparam can change params from almost anywhere
  + Algo:
    1. List of starting points = list of top-level modules
    2. Hierarchy below each starting point expanded as much as possible w/o expanding generate constructs. Params encountered here given values by applying initial values, overrides and defparams  
       defparam statements whose targets can't be resolved deferred to next iteration
    3. Each generate construct encountered in step 2 revisited and generate scheme evaluated. These generate instantiations become the new starting points. If new list of starting points is not empty, go to step 2
  + Modules are split (duplicated) during elaboration if required, Eg:



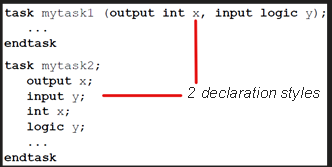
* Early Resolution of Hierarchical Names
  + To follow above algo, some hierarchical names in defparam statements need to be resolved before full elaboration
    - It's possible that name resolves differently without early resolution
    - This could lead to same hierarchical name resolving to differently, eg:



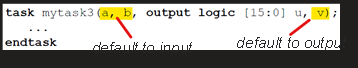
* **Bind: See LRM Sec 23.11**
* **Subroutines: Tasks and Functions**

|  |  |
| --- | --- |
| **Function** | **Task** |
| Statements in the body execute in 1 simulation time unit | May contain time controlling statements |
| Cannot enable (call) a task | Can enable a function or other tasks |
| If nonvoid, returns a single value | Doesn't return a value |
| If nonvoid, can be used as an operand (return value) |  |
| Eg: outputVal = func(inputVal) | Eg: task(inputVal,outputVal) |

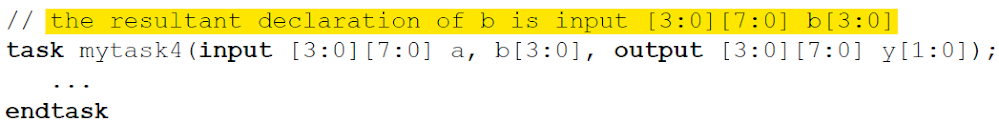
**Tasks**



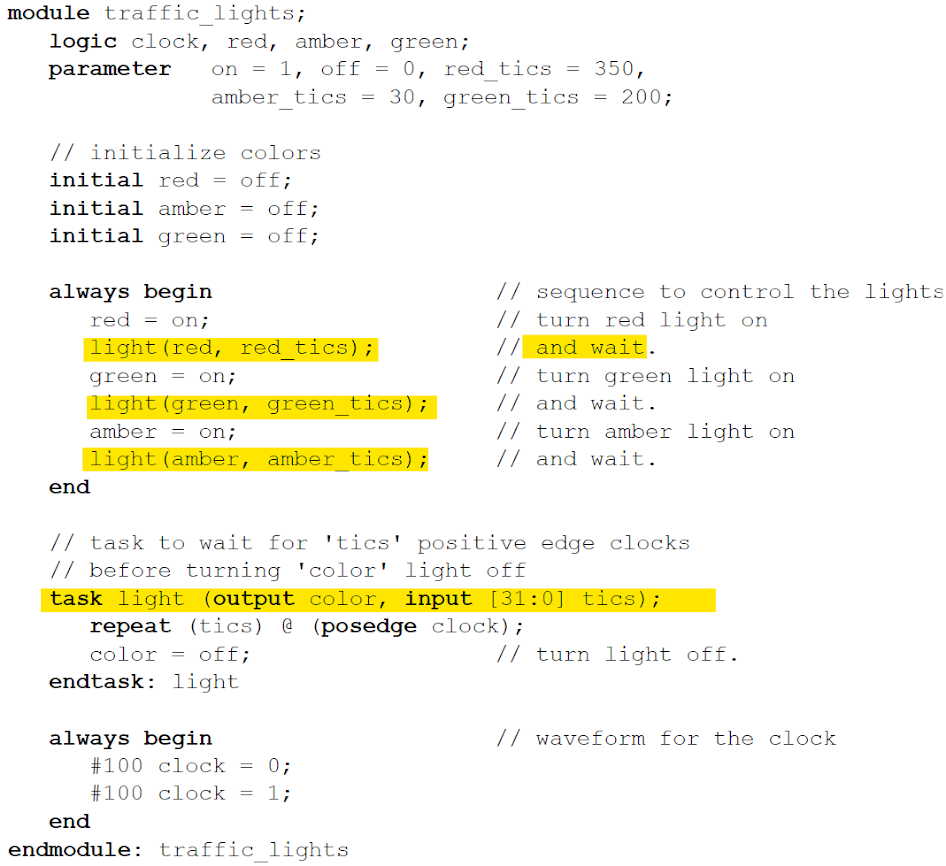
* Arguments can have following directions:
  1. input     *// copy value in at beginning*
  2. output    *// copy value out at end*
  3. inout     *// copy in at beginning and out at end*
  4. ref       *// pass reference*
* If direction not given, default to input
* If direction given, subsequent args default to same as previous:



* Argument data type defaults to *logic* for first arg, defaults to previous arg otherwise

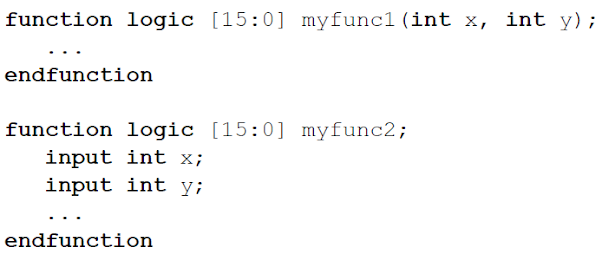


* Statements b/w *task* and *endtask* execute sequentially (like begin…end)
* *return* statement can be used to exit before *endtask*
* Example:

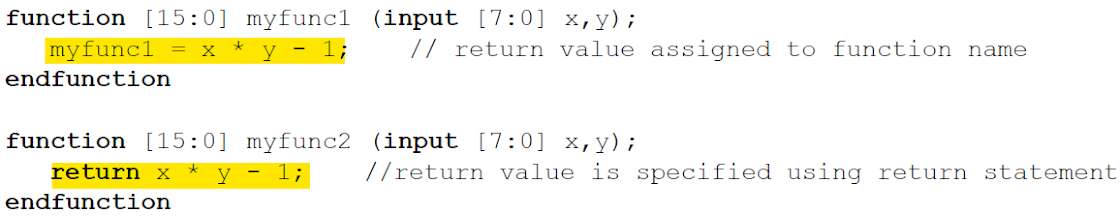


Static vs Automatic Tasks

* + By default static (all declared items statically allocated)
    - All items shared across all uses of tasks executing concurrently
  + To define automatic storage:
    - using *automatic* keyword in task declaration
    - Implicitly by defining within a module/interface/program/
  + Tasks defined within classes are always automatic
  + Items in static tasks defined in different module instances have separate storage
  + Variables in static tasks
    - retain their value between invocations (including all args)
    - must be initialized to a default value
  + Variables in atutomatic tasks
    - initialized to default value when execution enters the scope (including output args)
    - input/inout initialized to passed values
    - deallocated at end of task invocation
    - can't be assigned values using nonblocking assignments/procedural continuous assignments
    - can't be referenced by procedural continuous assignments/procedural force statements
    - cannot be traced with system tasks like $monitor and $dumpvars
* **Functions**

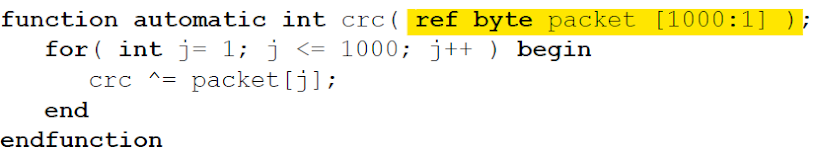


* Cannot contain any time-controlled statements ie #, ##, @, fork-join, fork-join\_any, wait, wait\_order, or expect
* Cannot enable (call) tasks
* May enable fine-grain process control methods
* Argument handling same as tasks
* Default direction/data type same as tasks
* Can't call a function with output, inout or ref args in an event expression, in an event expression within a procedural continuous assignment, or in an expression not in a procedural statement
* statements executed just like tasks
* 2 ways to return a value: using function name, or return statement:

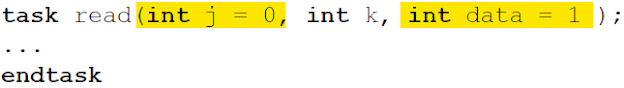


* return overrides function name statement
* Can return a struct/union
* Outside the function, function name used to define function scope (for hierarchical names)
* Execute with no delay
* Non blocking statements allowed in a function ie, nonblocking assignments, event triggers, clocking drives, and fork\_join\_none (more info in LRM Sec 13.4.4 - Background Process spawned by function calls)
* Static vs Automatic Handling same as tasks
  + specific variables can be declared static in an automatic function or automatic in a static function
* Constant functions
  + Cannot have output, inout or ref args
  + Can't be void type
  + Can't have any fork constructs
  + Can't contain hierarchical references
  + Function invoked in a const function should be const
  + All identifiers that are not parameters or function must be declared locally
  + Can't be declared inside a generate block
  + Any default argument must be a constant expression
  + Calls to constant functions evaluated at elaboration time

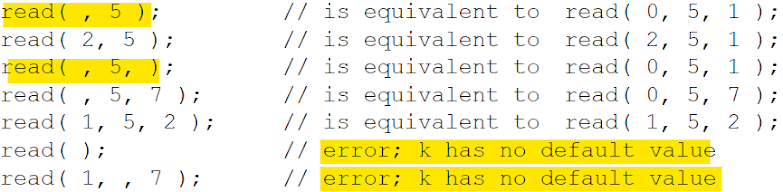
* **Passing Arguments**
  + Can pass by value or reference (passing by value is default)
  + Passing by reference
    - Done using *ref* keyword
    - Can't do it for static functions
    - Call is identical to functions with pass by value
    - Eg



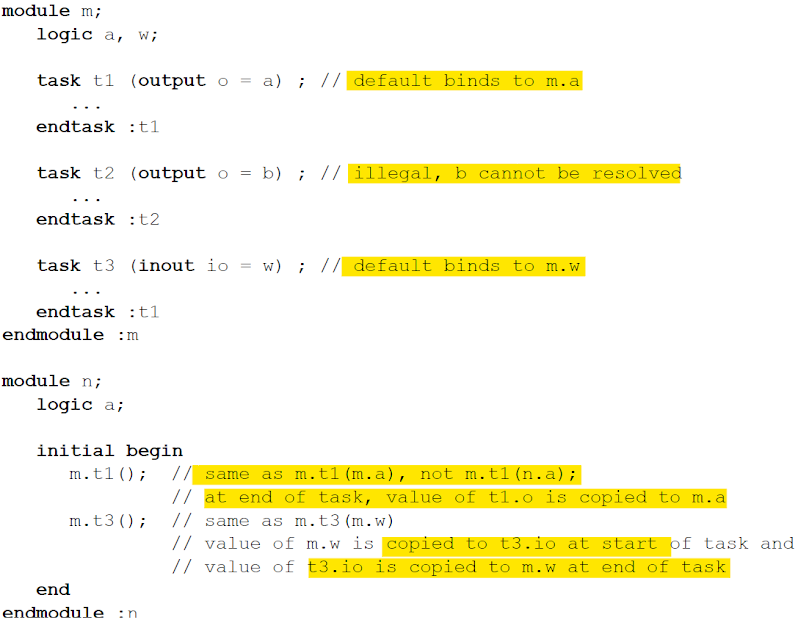
* Only following are allowed to pass by reference:
  + variable
  + class property
  + member of unpacked structure
  + element of unpacked array
* Outdated references (read LRM sec 13.5.2)
* Can't combine *ref* with directional keyword (similar to inout except inout is copied twice- entry, exit)
* const ref is legal (cant be modified by subroutine)
* **Default arguments**
  + Only allowed with ANSI style declaration
  + Eg:



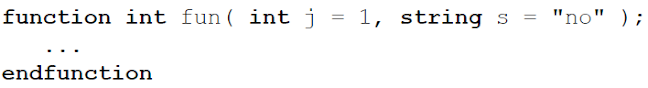
Calls to above example:

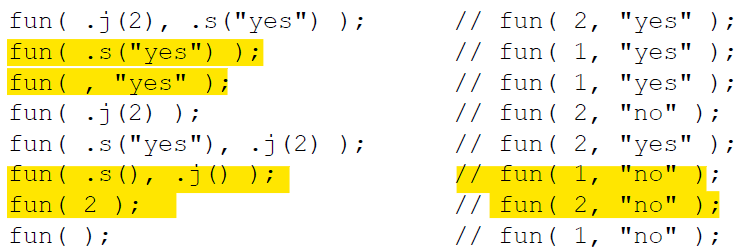


* Output arg with default value:



* Can bind arguments by name:



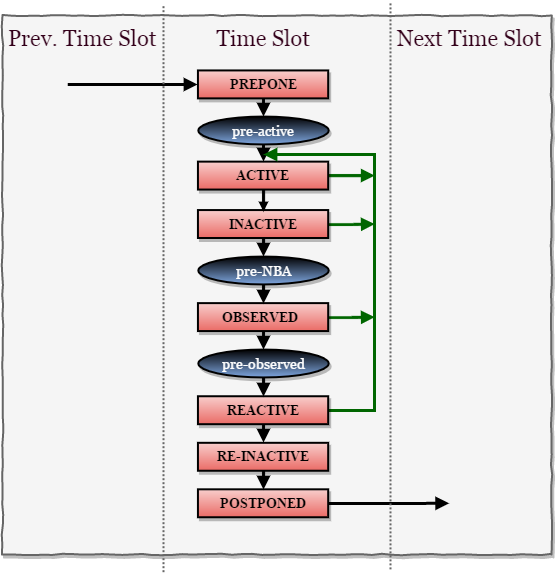


if specifying mix of positional and named arguments, all positional must come first:

fun ( 2) // 0K fun ( .s ("yes 2, .s ("yes") illegal 

* **Import/Export Functions**
  + Can import external foreign language subroutine
  + Can export SystemVerilog subroutine to foreign language
* **Parameterized Subroutines**
  + Allows for multiple implementations using one definition
  + Done using static methods in parameterized classes (see LRM section 13.8)

* **Event Scheduling**
  + SystemVerilog time slot divided into 17 ordered regions
    - 9 for execution of statements
    - 8 for PLI code
    - Provides predictable interaction b/w design and testbench
    - Prevents race conditions
  + *Update:* change in state of a net or variable
  + On an update, processes sensitive to that update considered for evaluation: *Evaluation Event*
  + Process eg: initial, always, always\_comb, always\_latch, always\_ff, continuous assignments, asynchronous tasks, procedural assignments



Preponed region

* + Executed once in each time slot, immediately after advancing simulation time
  + Values used for concurrent assertion are sampled
* Pre-active region
  + For PLI callbacks - allow users to read/write values and create events before events are evaluated in Active region
* Active Region
  + Holds current events being evaluated
  + Events can be processed in any order
    - Execute module blocking assignments
    - Evaluate RHS of non-blocking assignments, schedule updates into the NBA region
    - Execute module continuous assignments
    - Evaluate inputs and update outputs of Verilog Primitives
    - Execute $display and $finish commands
* Inactive Region
  + Holds events to be evaluated after active events are processed
  + #0 blocking assignments are scheduled
* Non-blocking Assignments Events Region (NBA)
  + Execute updates to LHS variables that were scheduled in Active region for currently executing non-blocking assignments
* Post-NBA Region
  + PLI callback control point for user code to read/write values and create events after NBA events are evaluated
* Observed Region
  + Evaluate concurrent assertions using values sampled in preponed region
* Post-Observed Region
  + PLI callback control point for user code to read values after properties are evaluated (in observed region or earlier)
* Reactive Region
  + Code specified in program block
  + Pass/fail code from property expressions
  + Evaluate and execute all program activity in any order:
    - Execute all program blocking assignments
    - Execute pass/fail code from concurrent assertions
    - Evaluate RHS of program nonblocking assignments
    - Execute program continuous assignments
    - Execute $exit and implicit $exit commands
* Re-inactive Events Region
  + #0 blocking assignments in a program process are scheduled
* Postponed Region
  + Execute $strobe and $monitor commands
  + Collect functional coverage of items that use strobe sampling
* **Modport**
  + **Todo :** [https://www.chipverify.com/](https://www.chipverify.com/systemverilog/systemverilog-modport)
* **Interface**
  + Encapsulates signals into a block
  + Can be reused for other projects
  + Easier to connect with DUT and other components
  + Eg: ABP bus protocol:  
    interface apb\_if (input pclk);  
            logic [31:0]    paddr;  
            logic [31:0]    pwdata;  
            logic [31:0]    prdata;  
            logic           penable;  
            logic           pwrite;  
            logic           psel;  
    Endinterface
  + To define port directions: ***modport*** 
    - Different modport definitions can be passed to different components
      * Allows definition of input/output directions for each component
    - Eg:  
      interface myBus (input clk);  
        logic [7:0]  data;  
        logic      enable;

  // From TestBench perspective, 'data' is input and 'write' is output  
  modport TB  (input data, clk, output enable);

  // From DUT perspective, 'data' is output and 'enable' is input  
  modport DUT (output data, input enable, clk);  
endinterface

* Connecting an Interface with a DUT:

module dut (myBus busIf);  
  always @ (posedge busIf.clk)  
    if (busIf.enable)  
      busIf.data <= busIf.data+1;  
    else  
      busIf.data <= 0;  
endmodule

// Filename : [tb\_top.sv](http://tb_top.sv/)  
module tb\_top;  
  bit clk;

  // Create a clock  
  always #10 clk = ~clk;

    // Create an interface object  
  myBus busIf (clk);

  // Instantiate the DUT; pass modport DUT of busIf  
  dut dut0 (busIf.DUT);

     // Testbench code : let's wiggle enable  
  initial begin  
    busIf.enable  <= 0;  
    #10 busIf.enable <= 1;  
    #40 busIf.enable <= 0;  
    #20 busIf.enable <= 1;  
    #100 $finish;  
  end  
endmodule

* Advantages
  + Can contain tasks, functions, parameters, variables, functional coverage, assertions
    - Can monitor and record transactions via interface
  + Easier to connect to design:

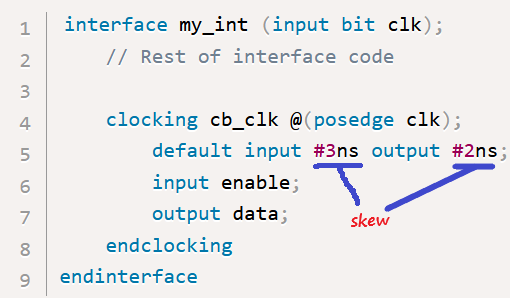
//Before interface  
dut dut0  (.data (data),  
           .enable (enable),  
           //  all other signals  
           );

// With interface - higher level of abstraction possible  
dut dut0  (busIf.DUT);

* Parameterized Interface:

interface myBus #(parameter D\_WIDTH=31) (input clk);  
        logic [D\_WIDTH-1:0] data;  
        logic               enable;  
endinterface

* Can have multiple clocking blocks in an interface
* Example of clocking block in interface:



To wait for positive edge of clock  
@busif.cb\_clk;  
  
To use clocking block signals:  
busif.cb\_clk.enable = 1;

* When an interface is referenced as a port, its variables and nets are assumed to have ref and inout access respectively
* Interface bundles:
  + using named bundle (as in above example)
  + using generic bundle:

interface ifc(…);

  …

endinterface

module mod (interface \_if);    // any interface can be passed to this

…

endmodule

module top();

  ifc \_if;

    mod m(if);

   …

endmodule

* **Program Block**
  + To have clear separation b/w testbench and design
  + Contains full environment for testbench
  + Provides entry point to testbenches
  + Creates a scope that provides program-wide data, tasks, and functions
  + Provides a syntactic context that specifies scheduling in the Reactive region
  + Cannot contain always block
  + A module cannot call a function/task inside a program block, but a program can call a function/task inside a module
  + Example:

//++++++++++++++++++++++++++++

// Simple Program with ports

//++++++++++++++++++++++++++++

program simple(input wire clk,output logic reset,

            enable, input logic [3:0] count);

  //============================

  // Initial block inside program block

  //============================

  initial begin

    $monitor("@%0dns count = %0d",$time,count);

    reset = 1;

    enable = 0;

     #20  reset = 0;

    @ (posedge clk);

    enable = 1;

    repeat (5) @ (posedge clk);

    enable = 0;

    // Call task in module

    simple\_program.do\_it();

  end

  //============================

  // Task inside a module

  //============================

  task do\_it();

    $display("%m I am inside program");

  endtask

endprogram

//============================

//  Module which instantiates program block

//============================

module simple\_program();

   l  logic [3:0] count;

  wire reset,enable;

  //============================

  // Simple up counter

  //============================

  always @ (posedge clk)

   if (reset) count <= 0;

   else if (enable) count ++;

  //============================

  // Program is connected like a module

  //============================

  simple prg\_simple(clk,reset,enable,

  //============================

  // Task inside a module

  //============================

  task do\_it();

    $display("%m I am inside module");

  endtask

  //============================

  // Below code is illegal

  //============================

  //initial begin

  //  prg\_simple.do\_it();

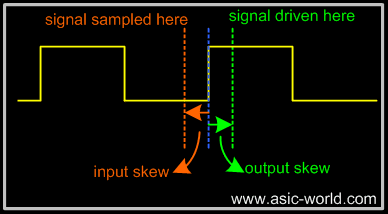
  //end

endmodule

* Calling $exit closes all threads spawned from current program
* $exit called automatically when execution reaches end of initial block in a program. Example: [http://ftp.smart-dv.com/](http://ftp.smart-dv.com/systemverilog/program2.html)
* Most of the times use interface to connect to external world, eg: [http://ftp.smart-dv.com/](http://ftp.smart-dv.com/systemverilog/program3.html)
* **Clocking Block**
  + Encapsulates signals that share a common clock
  + Signals specified inside the clocking block will be sampled/driven wrt that clock
  + Identifies clock signals
  + Captures timing and synchronization requirements of blocks being modelled
  + Used to control when the testbench drives and samples signals from DUT
  + Features supported
    - Input sampling
    - Synchronous events
    - Synchronous drives
  + Syntax:

[default] clocking [identifier\_name] @ [event\_or\_identifier]  
        default input #[delay\_or\_edge] output #[delay\_or\_edge]  
        [list of signals]  
endclocking

* Signal directions in a clocking block are wrt testbench and not DUT
* Skews (delay above)
  + Normally: input sampled at clock edge, output driven at clock edge
  + With skew: input sampled at clock edge + skew, output driven at clock edge + skew:



Skew must be a constant expression

* Can be specified as parameter
* Skew values can be parameterized
* If a default skew is not specified, all inputs sampled #1step and output driven 0ns after the event
* A testbench can have many clocking blocks, but only one per clock

* clocking ck1 @ (posedge clk);  
          default input #5ns output #2ns;  
          input data, valid, ready = top.ele.ready;  
          output negedge grant;  
          input #1step addr;  
  endclocking

* A clocking block called **ck1** is created which will be active on the positive edge of **clk**
* By default, all input signals within the clocking block will be sampled 5ns before and all output signals within the clocking block will be driven 2ns after the positive edge of the clock **clk**
* **data**, **valid** and **ready** are declared as inputs to the block and hence will be sampled 5ns before the posedge of **clk**
* **grant** is an output signal to the block with its own time requirement. Here **grant** will be driven at the negedge of **clk** instead of the default **posedge**.

* Example:  
  `timescale 1ns/1ns

// program declaration with ports.

program clocking\_skew\_prg (

  input  wire        clk,

  output logic [7:0] din,

  input  wire  [7:0] dout,

  output logic [7:0] addr,

  output logic       ce,

  output logic       we

  );

  // Clocking block

  clocking ram @(posedge clk);

     input   #1  dout;

     output  #1  din,addr,ce,we;

  endclocking

  initial begin

    // Init the outputs

    ram.addr <= 0;

    ram.din <= 0;

    ram.ce <= 0;

    ram.we <= 0;

    // Write Operation to Ram

    for (int i = 0; i < 2; i++) begin

      @ (posedge clk);

      ram.addr <= i;

      ram.din <= $random;

      ram.ce <= 1;

      ram.we <= 1;

      @ (posedge clk);

      ram.ce <= 0;

    end

    // Read Operation to Ram  
   for (int i = 0; i < 2; i++) begin

      @ (posedge clk);

      ram.addr <= i;

      ram.ce <= 1;

      ram.we <= 0;

      // Below line is same as  @ (posedge clk);

      @ (ram);

      ram.ce <= 0;

    end

     #40  $finish;

  end

endprogram

// Simple top level file

module clocking\_skew();

  logic        clk = 0;

  wire   [7:0] din;

  logic  [7:0] dout;

  wire   [7:0] addr;

  wire         ce;

  wire         we;

  reg    [7:0] memory [0:255];

  // Clock generator

  always  #10  clk++;

  // Simple ram model

  always @ (posedge clk)

   if (ce)

     if (we)

       memory[addr] <= din

        else

       dout <= memory[addr];

// Monitor all the signals

initial begin

$monitor("@%0dns addr :%0x din %0x dout %0x we %0x ce %0x",

           $time, addr, din,dout,we,ce);

end

// Connect the program

clocking\_skew\_prg U\_program(

.clk   (clk),

.din   (din),

.dout  (dout),

.addr  (addr),

.ce    (ce),

 .we    (we)

);

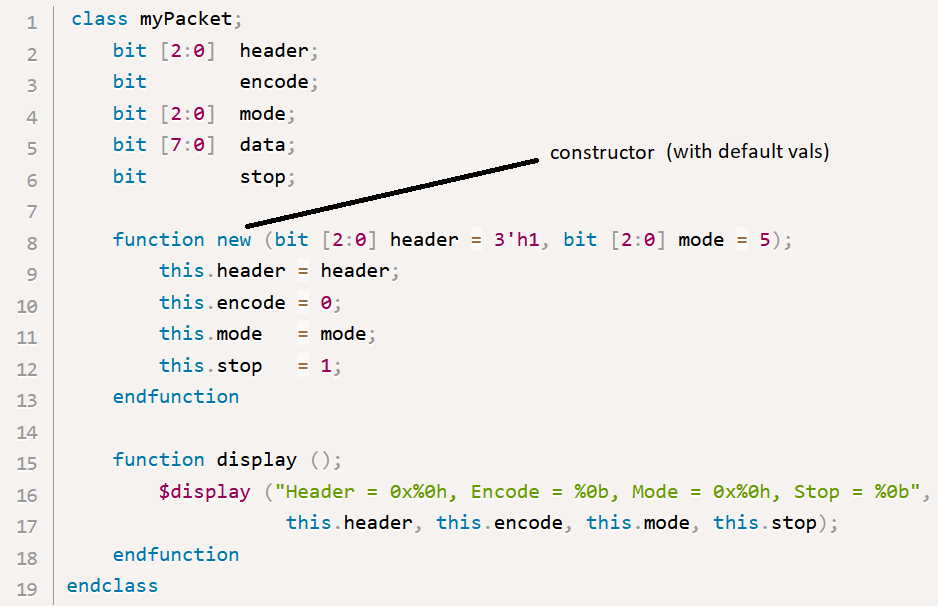
endmodule

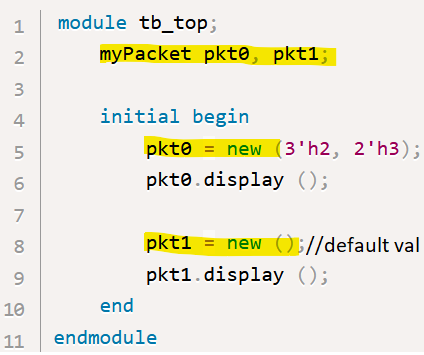
* **Package**
  + Provides mechanism for storing and sharing data, methods, property, parameters that can be reused in multiple modules, interfaces, or programs
  + Has explicitly named scope, at the same level as the top-level module
  + Avoids cluttering global namescope
  + Can be *imported* into current scope for use
  + Items in packages cannot have hierarchical references to identifiers except those created within the package or made visible by import of another package
  + Eg:



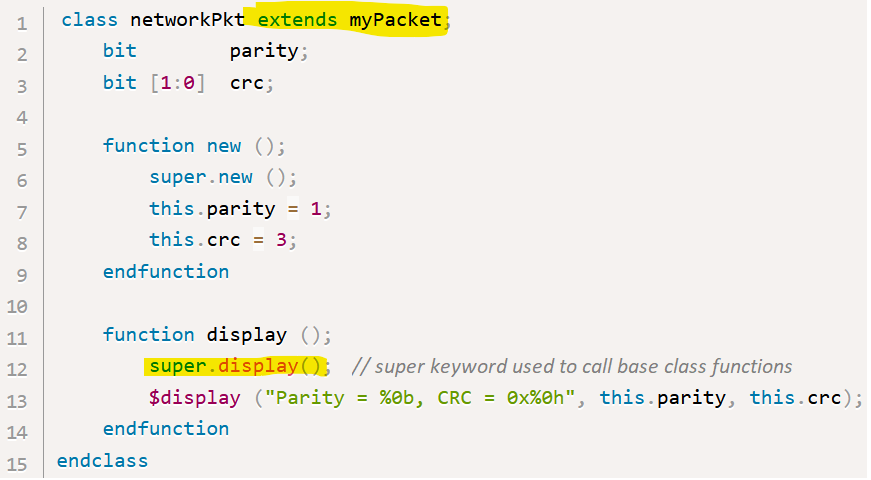
**Class**

* + Similar to C++

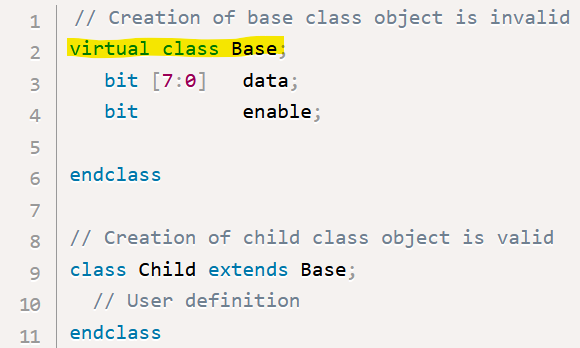




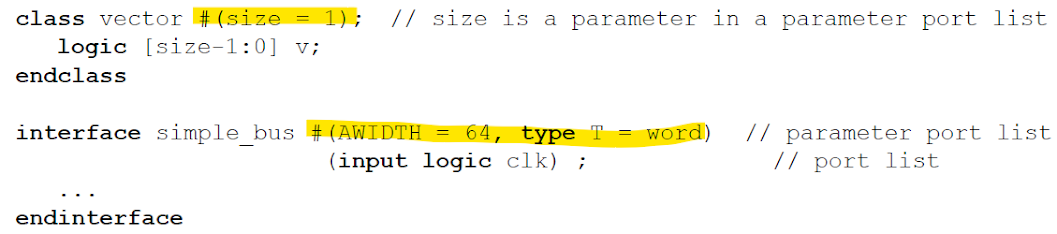
* Inheritance



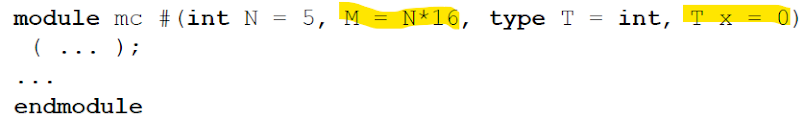
* Abstract class



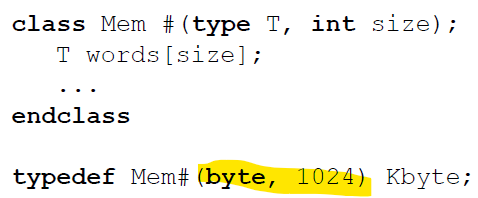
* **Parameters (see Verilog Parameter notes first)**
  + Parameter keyword can be omitted from port lists:



* A parameter can depend on earlier parameters:

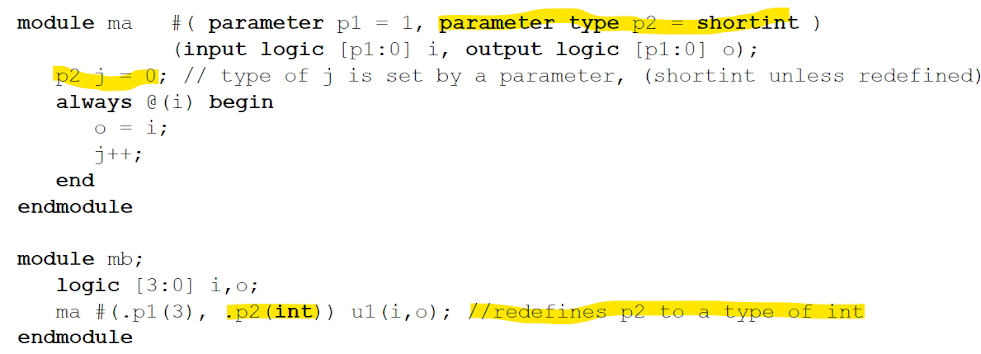


* Default value of parameter may be omitted (each instantiation/object must then specify it)



* **Value parameters** (parameter, localparam, specparam)
  + can only be set to and expression of:
    - literals
    - Value parameters
    - Local parameters
    - Genvars
    - Enumerated names
    - Constant function of above
  + Hierarchical names not allowed
  + Specparam can be assigned to expression containing one or more specparams
  + individual members of an aggregate parameter (eg unpacked array/unpacked structure) may not be assigned or overridden separately
  + *cannot be declared in a generate block, package, class body, or compilation-unit scope.*

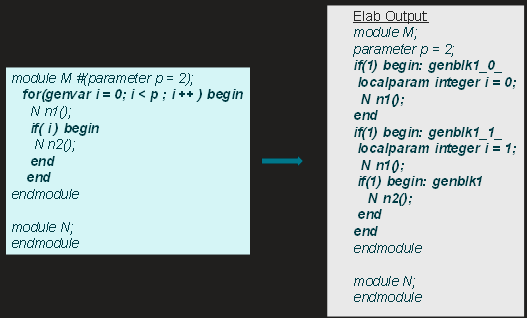
* **Type Parameters:**



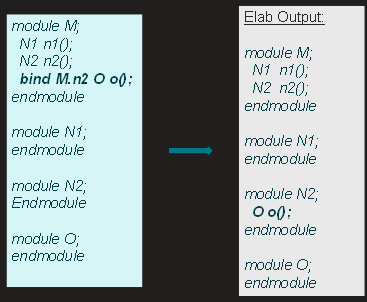
* **Local Parameters (localparam)**
  + Identical to parameters except cant be modified using defparam, or instance parameter value assignments
  + can be assigned constant expressions containing parameters, which in turn can be modified with

defparam statements or instance parameter value assignments

* **Generate Blocks**
  + **Elaboration**
    - Generate constructs are flattened, Eg:

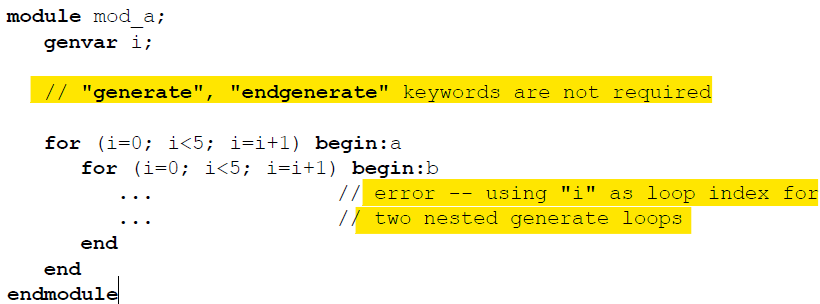


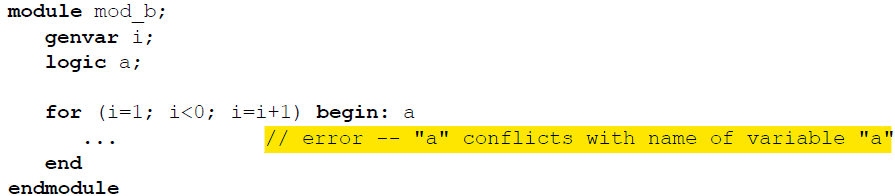
* **Bind Directives**
  + **Elaboration**
    - Needs to be stitched into the target module, eg:

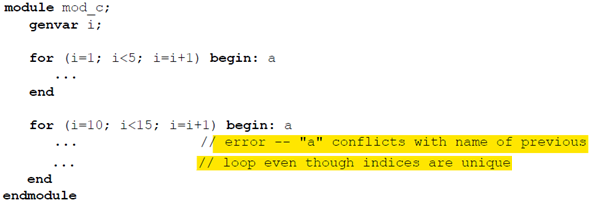


**Generate Constructs**

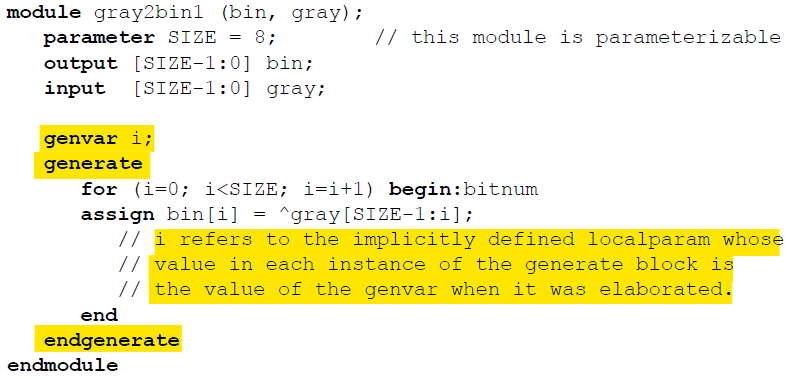
* + used to either conditionally or multiply instantiate *generate blocks* into a model
  + Generate Block: collection of 1 or more module item
    - May not contain:
      * port declarations
      * specify blocks
      * specparam declarations
    - Params inside generate blocks treated as localparams
    - provide the ability for parameter values to affect the structure of the design
    - allow for modules with repetitive structure to be described more concisely
    - make recursive module instantiation possible
    - 2 types
      * Loops
      * Conditionals
    - *Generate scheme*: the method for determining which or how many generate blocks are instantiated
      * Includes
        + Conditional expressions
        + Case alternatives
        + Loop control statements
      * Evaluated during elaboration, result determined before simulation
    - Elaboration results in 0 or more instances of the generate block (creating new levels of hierarchies)
    - Object declarations from enclosing scope can be accessed directly (unlike module instantiations)
    - Instantiated names can be accessed using hierarchical names
    - *Generate Region:* between *generate* and *endgenerate*
      * optional
      * Generate regions do not nest, and they may only occur directly within a module
  + **Loop Constructs**
    - loop index variable shall be declared in a *genvar* declaration prior to its use in a loop generate scheme
      * elaborated as integer
      * doesn't exist during simulation
      * shall not be referenced anywhere other than in a loop generate scheme
      * initialization and iteration assignments in the loop generate scheme shall assign to the same genvar
      * initialization assignment shall not reference the loop index variable on the right-hand side
    - There is an implicit declaration of a localparam
      * same name and type as the loop index variable
      * value = value of the index variable at the time the instance was elaborated
      * can be used anywhere within the generate block like a normal param
      * can be referenced with a hierarchical name
      * any reference to this name inside the loop generate block will be a reference to the localparam, not to the genvar
      * not possible to have two nested loop generate constructs that use the same genvar
    - Generate blocks in loop generate constructs can be named or unnamed
      * can consist of only one item
      * may or may not be between *begin* and *end*
      * comprises a separate scope and new level of hierarchy
      * if named, it is a declaration of an array of generate block instances
        + index values in are the values assumed by the genvar during elaboration
        + need not be a contiguous range of integers (*sparse array)*
        + array considered to be declared even if the loop generate scheme resulted in no instances of the generate block
      * if unnamed, the declarations within it cannot be referenced using hierarchical names other than from within the hierarchy instantiated by the generate block itself.
      * Error if the name of a generate block instance array conflicts with any other declaration, including any other generate block instance array
      * Error if the loop generate scheme does not terminate
      * Error if a genvar value is repeated during the evaluation of the loop generate scheme
      * Error if any bit of the genvar is set to x or z during the evaluation of the loop generate scheme
    - Error Examples:



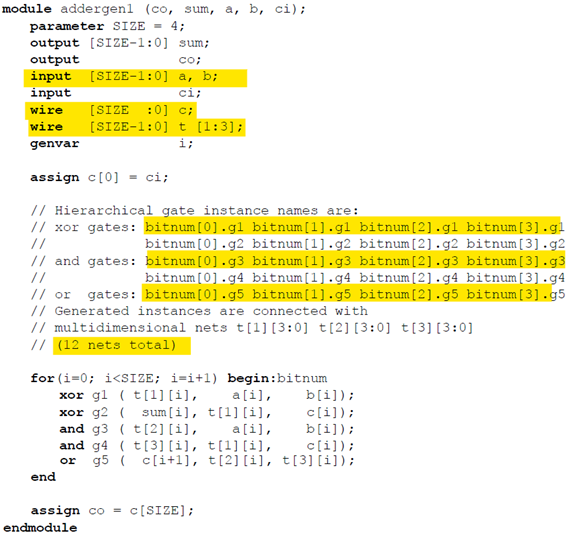




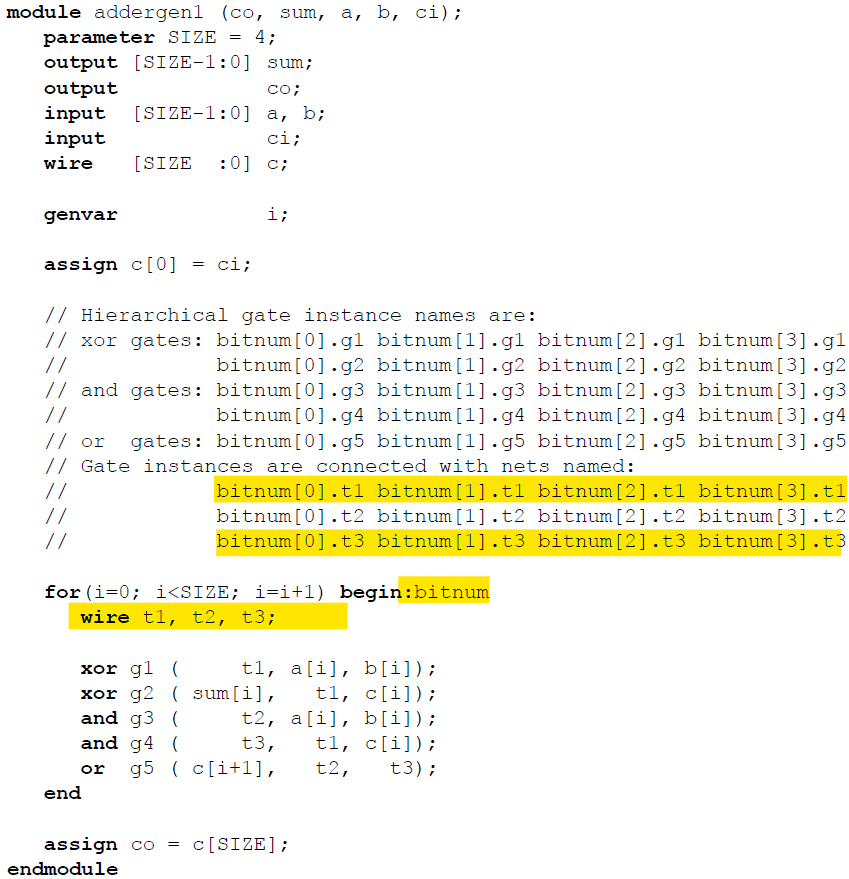
Working Example: gray to binary code converter



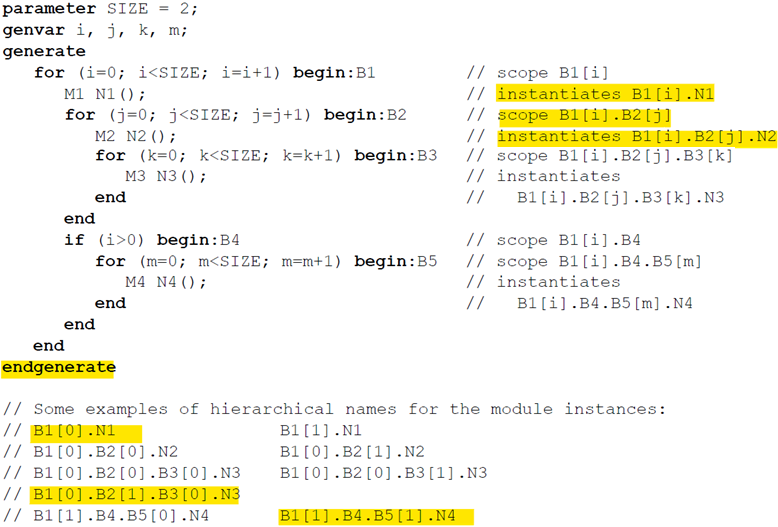
* Example: Ripple Adder using multidimensional nets declared outside loop



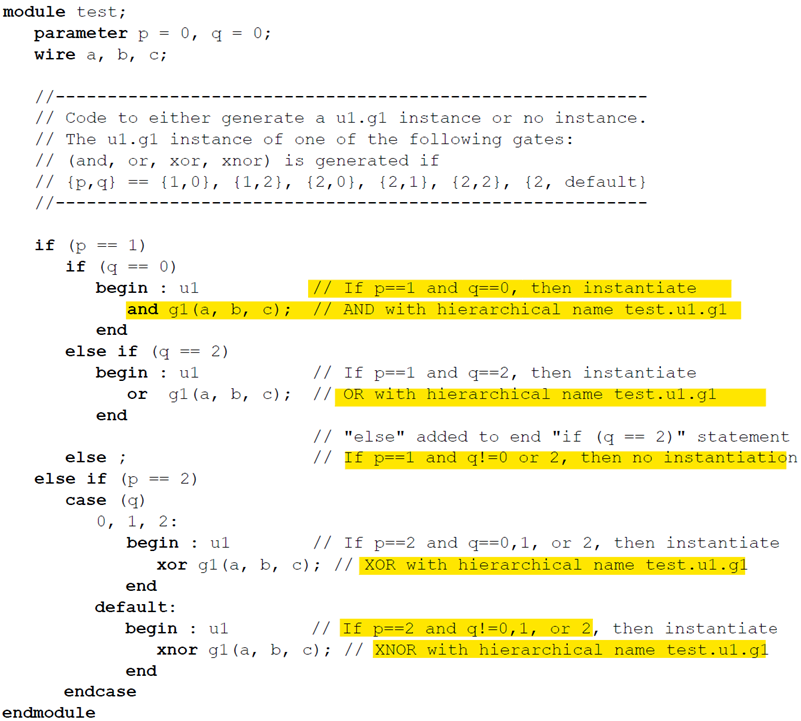
* Example: Ripple Adder with nets declared inside loop:



Example: Multi-level Loop



* **Conditional Generate Constructs**
  + Select at most one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration
  + Can be named or unnamed
  + Need not be surrounded by begin-end keywords
  + Comprise a separate scope and a new level of hierarchy when instantiated
  + Permissible for there to be more than one block with the same name within a single conditional generate construct
  + Not permissible for any of the named generate blocks to have the same name as generate blocks in any other conditional or loop generate construct in the same scope
  + Not permissible for any of the named generate blocks to have the same name as any other declaration in the same scope
  + Declarations inside an unnamed generate block cannot be referenced using hierarchical names other than from within the instantiated hierarchy itself
  + Conditional block with only one item that is a generate construct not surrounded by begin-end is not treated as a separate scope: *directly nested*
    - can have the same name as the generate blocks of the outer construct
    - cannot have the same name as any declaration in the scope enclosing the outer construct
    - If-else-if (with as many else ifs) is an example of this
  + Example: Multiple generate blocks with same names u1, g1 (test.u1.g1)



* Possible for a module to contain an instantiation of itself
  + A module containing an instantiation of itself will not be a top-level module
* Each generate construct in a given scope is assigned a number.
  + The number will be 1 for the construct that appears textually first in that scope and will increase by 1 for each subsequent generate construct in that scope.
  + All unnamed generate blocks will be given the name “genblk<n>” where <n> is the number assigned to its enclosing generate construct. If such a name would conflict with an explicitly declared name, then leading zeros are added in front of the number until the name does not conflict
  + Example

