SystemVerilog Basic Concepts

Ahmed Hemani

System Architecture and Methodology Group
Department of Electronic and Computer Systems
School of ICT, KTH

SystemVerilog

Rapidly getting accepted as the next generation HDL for
- *System Design*
- *Verification and*
- *Synthesis*

Improvements over Verilog

Many features of VHDL and C++

Strong in Verification
- *Assertions*
- *Functional Coverage*

Goals

Verification remains the central goal

SystemVerilog is a vehicle

Emphasis on verification aspects of SystemVerilog

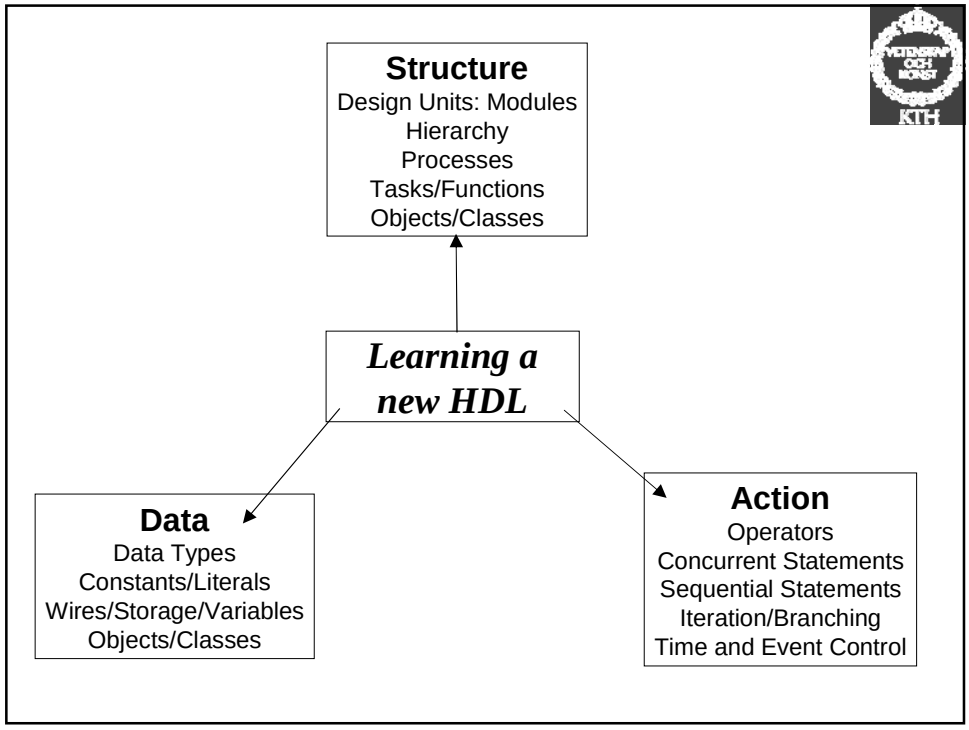
Some features of SystemVerilog will be not be covered

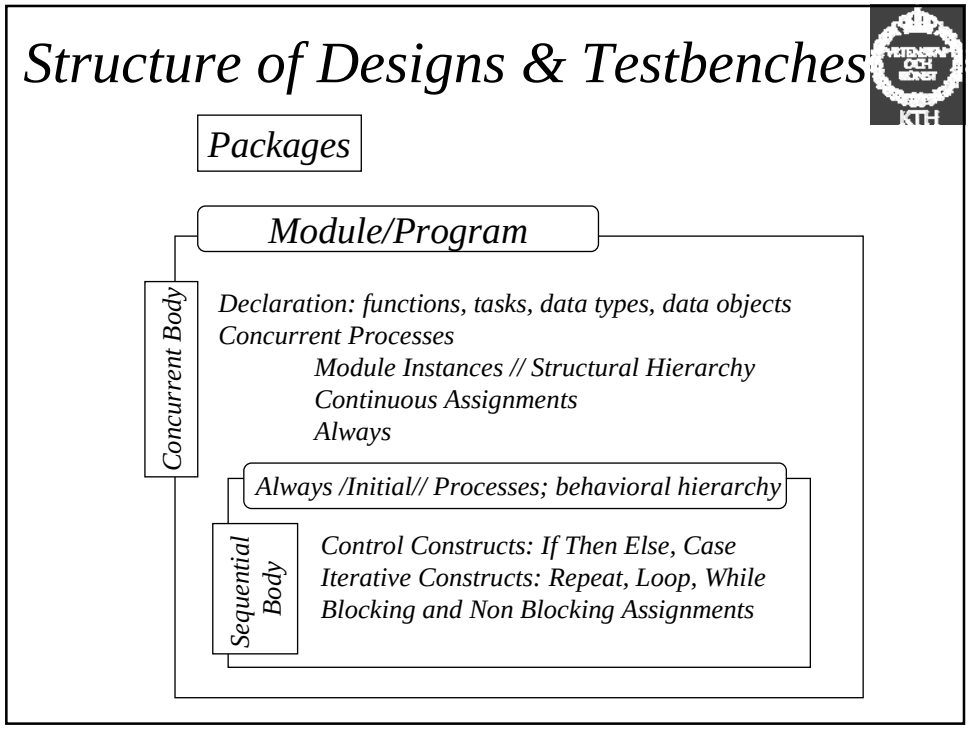
Emphasis is not on the synthesizable subset

Emphasis is to use the high level/abstract constructs to make the testbenches more readable, maintainable and modular

Why not VHDL ?

- VHDL Lacks
  - Constrained Random Generation
  - Functional Coverage
  - Assertions
- Specman E/Vera
  - Used with VHDL and Verilog for Constrained Random Generation and Functional Coverage
- PSL (Property Specification Language)
  - Used for Assertions
- Learning 1 language (SystemVerilog) is better than learning 2 languages (Specman E/Vera and PSL)
- Fluency in Verilog in addition to VHDL will give a well rounded competence profile


Structure
Design Units: Modules
Hierarchy
Processes
Tasks/Functions
Objects/Classes
Learning a new HDL
Data
Data Types
Constants/Literals
Wires/Storage/Variables
Objects/Classes
Action
Operators
Concurrent Statements
Sequential Statements
Iteration/Branching
Time and Event Control


Structure of Designs & Testbenches
Packages
Module/Program
Concurrent Body
Declaration: functions, tasks, data types, data objects
Concurrent Processes
Module Instances // Structural Hierarchy
Continuous Assignments
Always
Always /Initial// Processes; behavioral hierarchy
Sequential Body
Control Constructs: If Then Else, Case
Iterative Constructs: Repeat, Loop, While
Blocking and Non Blocking Assignments

Data Types

2 valued data types

4 valued data types

Resolved 4 valued data types

Scalars

Composites → *Static*

Composites → *Dynamic*

Class is the big new thing in SystemVerilog

Data Objects & Assignments

Wires

Non Blocking Assignments (<=)

Registers

Blocking Assignment (=)

Resolved Wires

Continuous Assignment

Use Blocking assignment to get the VHDL variable behavior

Use non blocking assignment to get the VHDL signal behavior

Basic Data Types

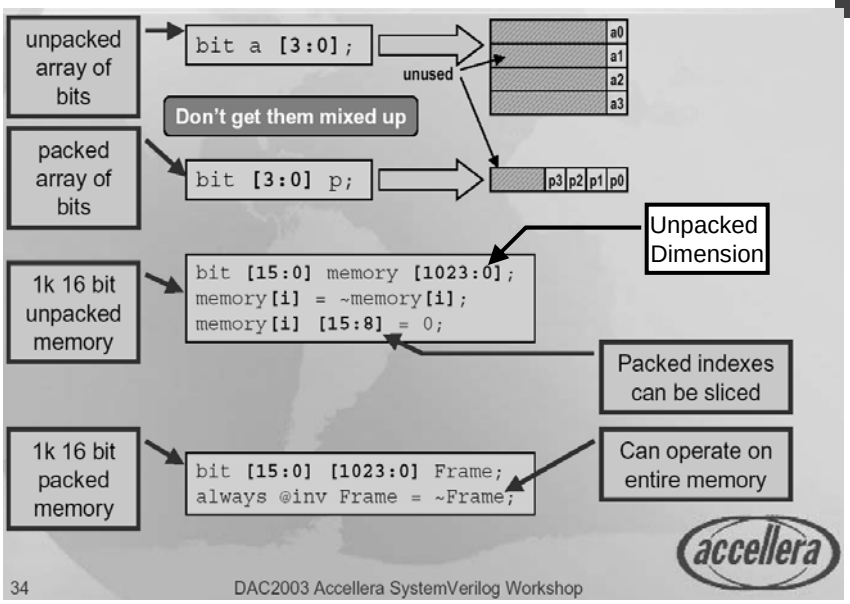
2 valued data type	*4 valued data type*
bit: 1 bit (packed array) *byte: 8 bit (character)* *shortint: 16 bit* *int: 32 bit*	*logic: 1 bit (packed array)* *integer: 32 bit* *time: 64 bit unsigned*

```
bit [14:0] bus; // unsigned
bit signed [11:0] i,q;

shortint unsigned addrs;
logic [15:0] addrs;
```

```
string myName = "John Smith";
byte c = "A"; // assign to c "A"
bit [1:4][7:0] s = "hello" ; // assigns to s "ello"
```

Packed & Unpacked arrays



Enumerated Type

Strongly typed set of named values

Very useful for declaring
FSM states
Operational and Test Modes
Instructions
Test Cases

```
enum {idle, init, reset, state0, state1, state2} fsm_states

typedef enum {idle, init, reset, state0, state1, state2} fsm_state_ty
fsm_state_ty fsm_states
```

Enumerated Type – Cont.

By default
enumerated type is of type `int`
The symbolic names get sequential nrs starting from 0

```
enum {idle, init, reset, state0, state1, state2} fsm_states

idle is 0, init is 1, reset is 2 and so on
```

In some cases there is a need to assign values different from the default sequential numbering

The values can be set for some of the names and not set for other names.
A name without a value is automatically assigned an increment of the value of the previous name.

```
// c is automatically assigned the increment-value of 8
      enum {a=3, b=7, c} alphabet;
```

```
// Syntax error: c and d are both assigned 8

       enum {a=0, b=7, c, d=8} alphabet;

If the first name is not assigned a value, it is given
the initial value of 0.
       // a=0, b=7, c=8
       enum {a, b=7, c} alphabet;
```

```
A 4 valued enum declaration allows x or z assignments
       // Correct: IDLE=0, XX='x, S1=1, S2=2
       enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;


       // Syntax error: IDLE=2'b00, XX=2'bx, S1=??, S2=??
       enum integer {IDLE, XX='x, S1, S2} state, next;
```

Name patterns can be easily expressed.

```
       typedef enum { add=10, sub[5], jmp[6:8] } E1;
```

add → 10,
sub0 → 11, sub1 → 12, sub2 → 13, sub3 → 14, sub4 → 15
jmp6 → 16, jmp7 → 17, and jmp8 → 18

```
typedef enum { red, green, blue, yellow, white, black } Colors;
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;
Colors col;
Week W
integer a, b, l;

col = green;

col = 1; // Invalid assignment

if ( 1 == col ) // OK. col is auto-cast to integer

a = blue * 3;

col = yellow;

b = col + green

col = Colors'(col+1);  // converted to an integer, then added to
                       // one, then converted back to a Colors type
col = col + 1; col++; col+=2; col = l; // Illegal because they would all be
                                       // assignments of expressions without a cast
col = Colors'(Su); // Legal; puts an out of range value into col

l = col + W; // Legal; col and W are automatically cast to int
```

```
first ()
last ()
next ()
prev ()
num ()
name ()

typedef enum {red, green, blue, yellow} Colors;
Colors c = c.first();
forever begin
       $display("%s : %d\n", c.name, c );
       if( c == c.last() ) break;
       c = c.next();
end
```

Operators

Bitwise Operators		
~	~m	Invert each bit of m
&	m&n	Bitwise AND of m and n
|	m|n	Bitwise OR of m and n
^	m^n	Bitwise EXOR of m and n
~^	m~^n	Bitwise EX NOR of m and n

Unary Reduction Operators		
&	&m	AND all bits of m together (1-bit result)
~&	~&m	NAND all bits of m together (1-bit result)
|	|m	OR all bits of m together (1-bit result)
~|	~|m	NOR all bits of m together (1-bit result)
^	^m	EXOR all bits of m together (1-bit result)
~^	~^m	EXNOR all bits of m together (1-bit result)

Source/Acknowledgement: On-line Verilog HDL Quick Reference Guide

Operators

Arithmetic Operators		
+	m+n	Add n to m
-	m-n	Subtract n from m
-	-m	Negate m (2's complement)
*	m*n	Multiply m by n
/	m/n	Divide m by n
%	m%n	Modulus of m/n

Logical Operators		
!	!n	Is m not true?
&&	m&&n	Are both m AND n true?
||	m||n	Are both m OR n true?

Equality Operators (Compares logic values of 0 and 1)		
==	m==n	Is m equal to n? (1-bit True/False result)
!=	m!=n	Is m not equal to n? (1-bit True/False result)
Identity Operators (Compares logic values of 0,1,X and Z)		
===	m^n	Is m identical to n? (1-bit True/False results)
!==	m~^n	Is m not equal to n? (1-bit True/False result)
Wild Eqality Operators		
=?=	m=?=n	A equals b, X and Z values are wild cards
!?=	m!?=n	A not equals b, X and Z values are wild cards

- The three types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z).
- The == and != operators result in X if any of their operands contains an X or Z.
- The === and !== check the 4-state explicitly, therefore, X and Z values shall either match or mismatch, never resulting in X.
- The =?= and !?= operators treat X or Z as wild cards that match any value, thus, they too never result in X.

Packages

*The **package** declaration creates a scope that contains declarations intended to be shared among one or more compilation units, modules, macromodules, interfaces, or programs.*

Items within packages are generally constants, type definitions, tasks, and functions. Items within packages cannot have hierarchical references.

```
package ComplexPkg;
      typedef struct {
              float 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
```

Referencing Data in packages

- One way to use declarations made in a package is to reference them using the scope resolution operator "**::**".

```
ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

- An alternate method for utilizing package declarations is via the **import** statement.
- The **import** statement provides direct visibility of identifiers within packages. It allows identifiers declared within packages to be visible within the current scope without a package name qualifier. Two forms of the import statement are provided: explicit import, and wildcard import. Explicit import allows control over precisely which symbols are imported:

```
import ComplexPkg::Complex;
import ComplexPkg::add;
```

- A wildcard import allows all identifiers declared within a package to be imported:

```
import ComplexPkg::*;
```

Modules

Corresponds to VHDL Entity/Architecture

1. *Defines Configurability*
2. *Defines interface*
3. *A concurrent body*
   Declarations
   Hierarchy – instantiating other modules
   Always – sequential bodies (VHDL Processes)
   Continuous assignments

Two styles of declaring ports

The old Verilog style

```
module adder #(parameter InWidth = 8,
               parameter OutWidth = 9)
              (clk, rstn, a, b, c);
   input logic clk;
   input logic rstn;
   input logic [InWidth-1:0] a;
   input logic [InWidth-1:0] b;
   output logic [OutWidth-1:0] c;
```

The new SystemVerilog style

```
module adder #(parameter InWidth = 8,
               parameter OutWidth = 9)
              (input logic clk,
               input logic rstn,
               input logic [InWidth-1:0] a,
               input logic [InWidth-1:0] b,
               output logic [OutWidth-1:0] c);
```

Instantiation

positional port connections

```
module alu (
        output reg [7:0] alu_out,
        output reg zero,
        input [7:0] ain, bin,
        input [2:0] opcode);
// RTL code for the alu module
endmodule

module accum (
        output reg [7:0] dataout,
        input [7:0] datain,
        input clk, rst_n);
// code for the accumulator
endmodule

module xtend (
        output reg [7:0] dout,
        input din,
        input clk, rst_n);
// code for the sign-extension
endmodule
```

Instance name

Module name

```
module alu_accum1 (
        output [15:0] dataout,
        input [7:0] ain, bin,
        input [2:0] opcode,
        input clk, rst_n);

   wire [7:0] alu_out;

   alu U1 (alu_out,,
           ain, bin, opcode);

   accum U2 (dataout[7:0],
        alu_out, clk, rst_n);

   xtend U3 (dataout[15:8],
        alu_out[7], clk, rst_n);

endmodule
```

Zero not connected

Instantiation

named port connections

```
module alu_accum2 (
        output [15:0] dataout,
        input [7:0] ain, bin,
        input [2:0] opcode,
        input clk, rst_n);
  wire [7:0] alu_out;

     alu U1 (.alu_out(alu_out),
                .zero(),
                .ain(ain),
                .bin(bin),
                .opcode(opcode));
     accum U2 (.dataout(dataout[7:0]),
                                .datain(alu_out),
                .clk(clk),
                .rst_n(rst_n));
     xtend U3 (.dout(dataout[15:8]),
                .din(alu_out[7]),
                .clk(clk),
                .rst_n(rst_n));
endmodule
```

Formal

Actual

Instantiation

implicity .name port connections

If the actual name and size is the same as formal

.name ≡ .name(name)

Exceptions need to be explicitly named

```
module alu_accum3 (
        output [15:0] dataout,
        input [7:0] ain, bin,
        input [2:0] opcode,
        input clk, rst_n);

  wire [7:0] alu_out;

  alu U1 (.alu_out, .zero(), .ain, .bin, .opcode);

  accum U2 (.dataout(dataout[7:0]), .datain(alu_out),
                .clk, .rst_n);

  xtend U3 (.dout(dataout[15:8]), .din(alu_out[7]),
                .clk, .rst_n);
endmodule
```

Implicit connections

. notation*

- .* → all ports that are not explicitly connected have a compatible matching data declaration
- Exceptions need to be explicitly named as is the case for dataout, datain and din in the example below
- Useful for rapidly building up the testbenches.
  - Hint: Copy and paste port declarations as data declarations in testbench.
  - Edit exceptions

```
module alu_accum4 (
        output [15:0] dataout,
        input [7:0] ain, bin,
        input [2:0] opcode,
        input clk, rst_n);

        wire [7:0] alu_out;
        alu U1 (.*, .zero());
        accum U2 (.*, .dataout(dataout[7:0]), datain(alu_out));
        xtend U3 (.*, .dout(dataout[15:8]), .din(alu_out[7]));
endmodule
```

Parameters

```
module top;
logic clk;

clockgen #(.start_value(1'b1), .delay(50),
           .ctype(int)) c (clk);

always @clk $display("t=%t clk=%b", $time, clk);

initial
  begin
    repeat(10) @(posedge clk) ;
    $finish(0);
  end
endmodule

module clockgen (output ctype clk);
  parameter logic start_value=0;
  parameter type ctype=bit;
  parameter time delay=100;

initial clk <= start_value;

always #delay clk <= !clk;

endmodule
```

Override parameters by name

Parameter used before definition

Parameters can have explicit type

accellera

43 DAC2003 Accellera SystemVerilog Workshop

Processes

Processes are concurrent statements instantiated in modules and are very similar to the VHDL processes

Principally a sequential body; but allows some concurrency

Can optionally have sensitivity list of signals that triggers its execution – just like in VHDL
Edge statement can further qualify the sensitivity

Comes in two flavours:
always – Always active – executed as soon as there is an event on the sensitivity list
Initial – Is executed only once. Initial does not mean that it is executed initially. The order of execution between always and initial is non-deterministic (decided by the compiler).

Syntax

Process type (Always | Initial) *Sensitivity List*

Local Variable Declarations

Sequential procedural statements:
If then else, Case, Repeat, Foreach etc.
Functions/Tasks invocations

Concurrent threads
Fork
If then else, Case, Repeat, Foreach etc.
Functions/Tasks invocations
Join

Always

Comes in four flavours:

always – the old verilog legacy. Not recommended

always_FF. Clocked synchronous body
posedge clk and negedge reset_n are the standard signals in the sensitivity list

always_comb: To infer combinational logic
Sensitivity list is inferred from signals involved in the RHS of expressions.

always_latch: When the intention is to infer latch
Sensitivity list is inferred from signals involved in the RHS of expressions.

Tasks

A SystemVerilog task corresponds to procedure in VHDL
Declaraed in Packages or Modules
A sequential body. Invoked in Processes
Module like interface declaration

```
task mytask1 (output int x, input logic y);
     ...
endtask

task mytask2;
     output x;
     input y;
     int x;
     logic y;
     ...
endtask
```

Functions

Functions are similar to tasks except that they can return a value, which can be void

Corresponds to functions in VHDL

```
function logic [15:0] myfunc1(int x, int y);
     myfunc1 = x – y;
endfunction

function logic [15:0] myfunc2;
     input int x;
     input int y;
     myfunc2 = x+y-5;
endfunction
```

Pass by value and reference

Pass by value (Default)

Each argument is copied to the subroutine area

```
function int crc( byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
           crc ^= packet[j];
     end
     endfunction
```

Pass by reference

Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine

```
function int crc( ref byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
           crc ^= packet[j];
    end
endfunction
```

Invoking makes no difference

```
byte packet1[1000:1];
int k = crc( packet1 ); // pass by value/reference: call is the same
```

Timing Control

```
#delay;

#delay <data object>  =              <expression>;
#delay <data object> <=            <expression>;

        <data object> = #delay   <expression>;
        <data object> <= #delay  <expression>;
```

@(*edge signal* or *edge signal* or ...)

wait (expression)

Control Constructs

```
If  <expression>
   statement or statement_group
else
   statement or statement_group
```

Unique if

Priority if

```
    case <expression>
      case_match1: statement or statement_group
      case_match2, case_match3: statement or statement_group
      default: statement or statement_group
    endcase
```

Casez
Casex
Priority case
Unique case

Iterative Constructs

```
module foo;
    initial begin
        loop1: for (int i = 0; i <= 255; i++)
       ...
        end

    initial begin
        loop2: for (int i = 15; i >= 0; i--)
             ...
        end
endmodule
```

```
jump; break; continue; return
```

Iterative Constructs – contd.

```
string words [2] = { "hello", "world" };
int prod [1:8] [1:3];

foreach( words [ j ] )
   $display( j , words[j] ); // print each index
and value

foreach( prod[ k, m ] )
      prod[k][m] = k * m; // initialize
```

```
forever statement or statement_group

repeat (number) statement or statement_group

while (expression) statement or statement_group

do statement or statement_group while (expression)
```

Structures

Encapsulates related data objects of different data types

Corresponds to Records in VHDL, and structs in C/C++

Can be hierarchical. Elements can be scalar, records, arrays, dynamic arrays, queues etc.

Can be recursive. Elements can be records themselves. Trees

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

Packed Structures

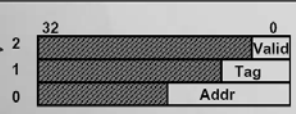
Represents bit or part selects of vectors

```
struct packed {
  bit         Valid;
  byte        Tag;
  bit [15:0] Addr;
} Entry;
iTag   = Entry.Tag;
iAddr  = Entry.Addr;
iValid = Entry.Valid
```

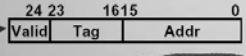
```
reg [24:0] Entry;
`define Valid 24
`define Tag 23:16
`define Addr 15:0
iTag   = Entry[`Tag];
iAddr  = Entry[`Addr];
iValid = Entry[`Valid]
```

packed struct may contain other packed structs or packed arrays

unpacked struct



packed struct

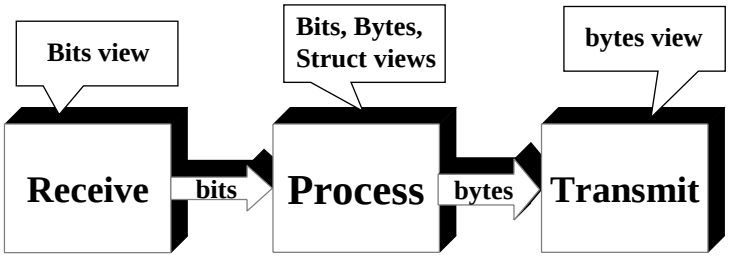


accellera

Unions

Unions enable different views of a collection of data objects.

For instance we might want to have a structured view of an ATM cell but also view it as a stream of bit and bytes



The different views, need not have same size.
Potential type loophole. Tagged Unions are the solutions

```
typedef struct packed { // default unsigned
              bit [3:0] GFC;
              bit [7:0] VPI;
              bit [11:0] VCI;
              bit CLP;
              bit [3:0] PT ;
              bit [7:0] HEC;
              bit [47:0] [7:0] Payload;
              bit [2:0] filler;
} s_atmcell;
typedef union packed { // default unsigned
                    s_atmcell acell;
                    bit [423:0] bit_slice;
                    bit [52:0][7:0] byte_slice;
              } u_atmcell;

   u_atmcell u1;

   byte b; bit [3:0] nib;

   b = u1.bit_slice[415:408]; //≡ b = u1.byte_slice[51];
   nib = u1.bit_slice [423:420]; // ≡nib = u1.acell.GFC;
```

Tagged Union

In many cases,
We do want to have different views of the bit patterns, but
We do not want to update using one view and read using other views

```
typedef union tagged {
   struct {bit [4:0] reg1, reg2, regd;} Add;
   union tagged {
        bit [9:0] JmpU;
        struct {bit [1:0] cc; bit [9:0] addr;}JmpC;
        } Jmp;
Instr;
```

```
i1 = ( e ? tagged Add '{ e1, 4, ed }; // struct members by position
         : tagged Add '{ reg2:e2, regd:3, reg1:19 }); // by name

i1 = tagged Jmp (tagged JmpU 239); /uncond. jmp instr. Sub opcode

// Create a Jump instruction, with "conditional" sub-opcode
i2 = tagged Jmp (tagged JmpC '{ 2, 83 }); // by position
i2 = tagged Jmp (tagged JmpC '{ cc:2, addr:83 }); // by name
```

Operating on Arrays

```
// 10 entries of 4 bytes (packed into 32 bits)
bit [3:0] [7:0] joe [1:10];
joe[9] = joe[8] + 1; // 4 byte add
joe[7][3:2] = joe[6][1:0]; // 2 byte copy

bit [3:0] [7:0] j; // j is a packed array
   byte k;
   k = j[2]; // select a single 8-bit element from j

   bit busA [7:0] [31:0] ; // unpacked array of 8 32-bit
   vectors
   int busB [1:0]; // unpacked array of 2 integers
   busB = busA[7:6]; // select a slice from busA

   int i = bitvec[j +: k]; // k must be constant.
   int a[x:y], b[y:z], e;
   a = {b[c -: d], e}; // d must be constant
```

```
string SA[10], qs[$];
int IA[*], qi[$];

// Find all items greater than 5
qi = IA.find( x ) with ( x > 5 );
// Find indexes of all items equal to 3
qi = IA.find_index with ( item == 3 );

// Find first item equal to Bob
qs = SA.find_first with ( item == "Bob" );

// Find last item equal to Henry
qs = SA.find_last( y ) with ( y == "Henry" );

// Find index of last item greater than Z
qi = SA.find_last_index( s ) with ( s > "Z" );

// Find smallest item
qi = IA.min;

// Find string with largest numerical value
qs = SA.max with ( item.atoi );

// Find all unique strings elements
qs = SA.unique;
```

Dynamic Arrays

A dynamic array is one dimension of an unpacked array whose size can be set or changed at runtime.

```
data_type array_name [];
bit [3:0] nibble[]; // Dynamic array of 4-bit vectors
    integer mem[]; // Dynamic array of integers
```

***new**[] sets or changes the size of the array.*
size() returns the current size of the array.
delete() clears all the elements yielding an empty array (zero size).

```
integer addr[]; // Declare the dynamic array.
addr = new[100]; // Create a 100-element array.
                ...
// Double the array size, preserving previous values.
addr = new[200](addr);
int j = addr.size;
addr = new[ addr.size() *4 ] (addr); // quadruple addr array
addr.delete; // delete the array contents
$display( "%d", addr.size ); // prints 0
```

Associative Arrays

- When the size of the collection is unknown or the data space is sparse, an associative array is a better option.
- Associative arrays do not have any storage allocated until it is used, and the index expression is not restricted to integral expressions, but can be of any type.
- An associative array implements a lookup table of the elements of its declared type. The data type to be used as an index serves as the lookup key, and imposes an ordering.
- The syntax to declare an associative array is:

```
data_type array_id [ index_type ];
```

where:

— *data_type* is the data type of the array elements. Can be any type allowed for fixed-size arrays.
— *array_id* is the name of the array being declared.
— *index_type* is the data-type to be used as an index, or *****.
If ***** is specified, then the array is indexed by any integral expression of arbitrary size.
An index type restricts the indexing expressions to a particular type.

Examples of associative array declarations are:

```
integer i_array[*]; // associative array of integer (unspecified index)
 bit [20:0] array_b[string]; // associative
```

Methods: num, delete, exists, first, last, next, prev

Queues

A queue is a variable-size, ordered collection of homogeneous elements. A queue supports constant time access to all its elements as well as constant time insertion and removal at the beginning or the end of the queue. Each element in a queue is identified by an ordinal number that represents its position within the queue, with 0 representing the first, and **$** representing the last. A queue is analogous to a one-dimensional unpacked array that grows and shrinks automatically. Thus, like arrays, queues can be manipulated using the indexing, concatenation, slicing operator syntax, and equality operators.

Queues are declared using the same syntax as unpacked arrays, but specifying **$** as the array size. The maximum size of a queue can be limited by specifying its optional right bound (last index).

```
byte q1[$]; // A queue of bytes
string names[$] = { "Bob" }; // A queue of strings with one element
integer Q[$] = { 3, 2, 7 }; // An initialized queue of integers
bit q2[$:255]; // A queue whose maximum size is 256 bits
```

The empty array literal {} is used to denote an empty queue. If an initial value is not provided in the declaration, the queue variable is initialized to the empty queue.

Que Operators and Methods

```
int q[$] = { 2, 4, 8 };
int p[$];
int e, pos;

e = q[0]; // read the first (left-most) item
e = q[$]; // read the last (right-most) item
q[0] = e; // write the first item
p = q; // read and write entire queue (copy)

q = { q, 6 }; // insert '6' at the end (append 6)
q = { e, q }; // insert 'e' at the beginning prepend e)

q = q[1:$]; // delete the first (left-most) item
q = q[0:$-1]; // delete the last (right-most) item
q = q[1:$-1]; // delete the first and last items

q = {}; // clear the queue (delete all items)

q = { q[0:pos-1], e, q[pos,$] }; // insert 'e' at position pos
q = { q[0:pos], e, q[pos+1,$] }; // insert 'e' after position pos
```

Que Methods: size, insert, delete, pop_front, pop_back, push_front, push_back

Type Casting

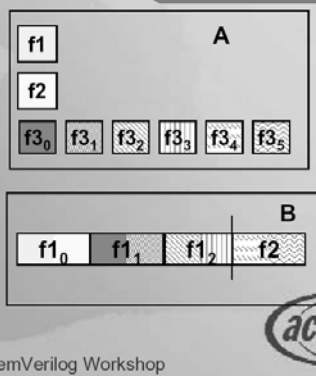
```
int'(2.0 * 3.0)
shortint'{8'hFA, 8'hCE}
17 '(x - 2)
```

A data type can be changed by using a cast (') operation

- Any aggregate bit-level object can be reshaped
  - Packed ⇔ Unpacked, Array ⇔ Structure

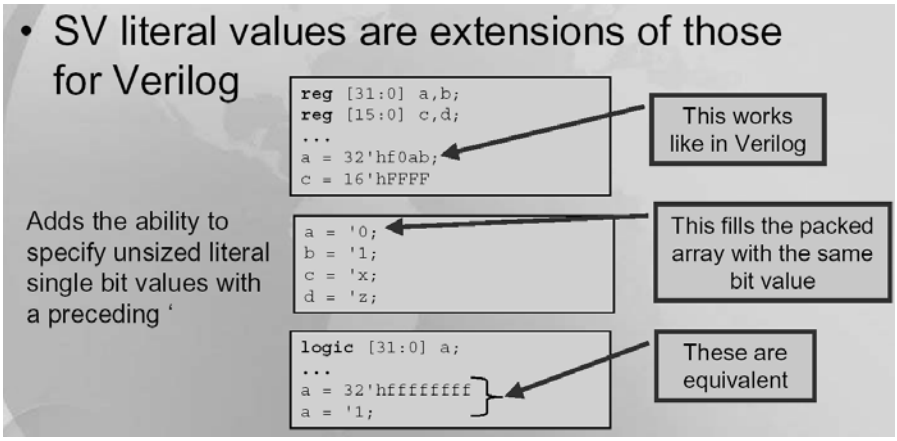
Objects must have identical bit size

```
typedef struct {
        bit [7:0] f1;
        bit [7:0] f2;
        bit [7:0] f3[0:5];
} Unpacked_s;
typedef struct packed {
        bit [15:0][0:2] f1;
        bit [15:0] f2;
} Packed_s;
Unpacked_s A;
Packed_s B;
…
        A = Unpacked_s'(B);
        B = Packed_s'(A);
```

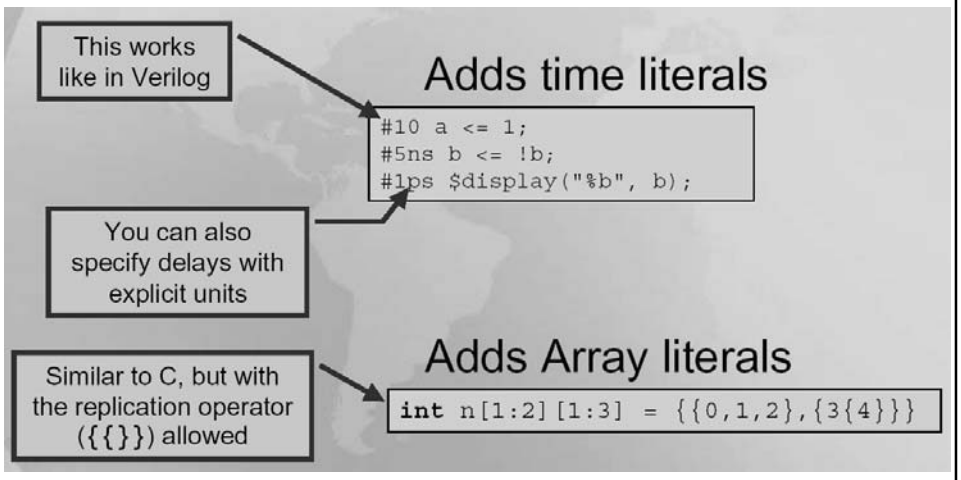


accellera

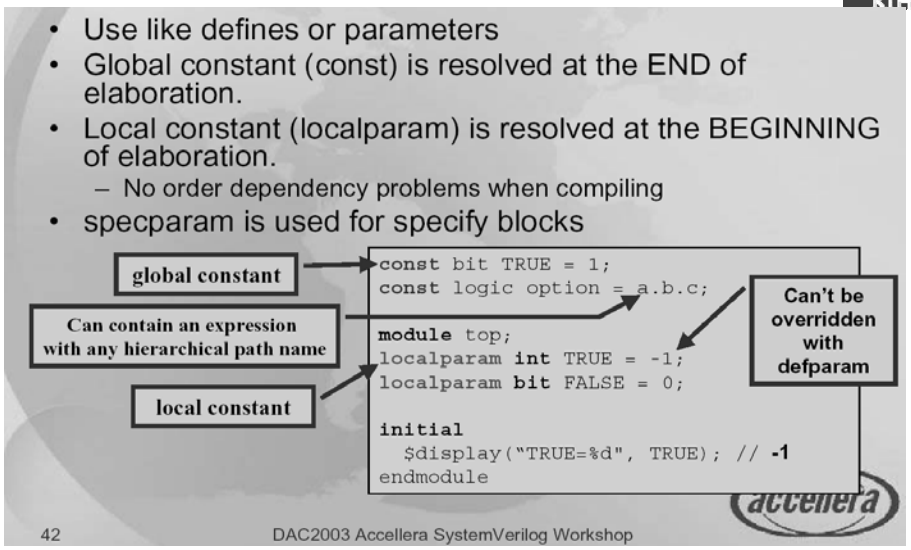
Literals



Literals



Constants



- Use like defines or parameters
- Global constant (const) is resolved at the END of elaboration.
- Local constant (localparam) is resolved at the BEGINNING of elaboration.
  - No order dependency problems when compiling
- specparam is used for specify blocks

Variables: Scope and LifeTime

- **Static** variables
  - Allocated and initialized at time 0
  - Exist for the entire simulation
- **Automatic** variables
  - Enable recursive tasks and functions
  - Reallocated and initialized each time entering a block
  - May not be used to trigger an event
- **Global** variables
  - Defined outside of any module (i.e. in $root)
  - Accessible from any scope
  - Must be static
  - Tasks and functions can be global too
- **Local** variables
  - Accessible at the scope where they are defined and below
  - Default to static, can made automatic
  - Accessible from outside the scope with a hierarchical pathname