System Verilog Introduction & Usage

IBM Verification Seminar

Author: Johny Srouji, Intel Corporation

IEEE P1800 Chair

Presentation Objectives

- Provide a brief history and status update of SystemVerilog as a HW Design & Verification Language
- Provide a high level overview of the language capabilities and usage aspects

→ This is not meant to be a tutorial of the language

→ Parts of this presentation are based on material which was presented in DAC SystemVerilog workshop by technical committees chairs

Agenda

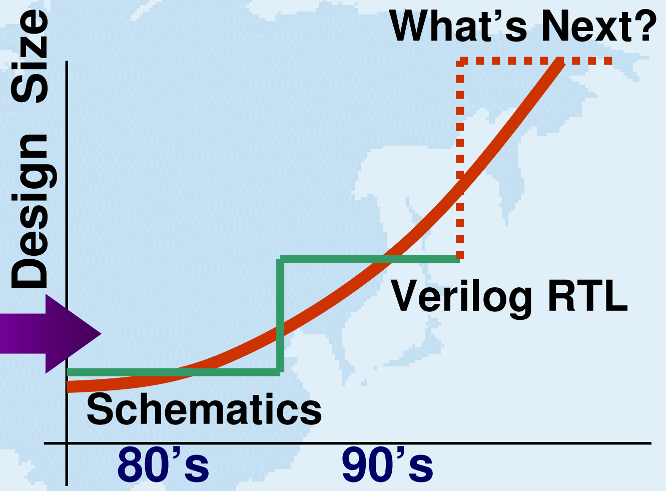
- Background
  - HW Design Languages
  - Problem Statement
  - IEEE P1800 Activity
- SystemVerilog Design Modeling
- SystemVerilog for Verification
  - Extensions for Test-Bench Modeling
  - Assertions
- SystemVerilog DPI
- A Quiz
- Conclusions

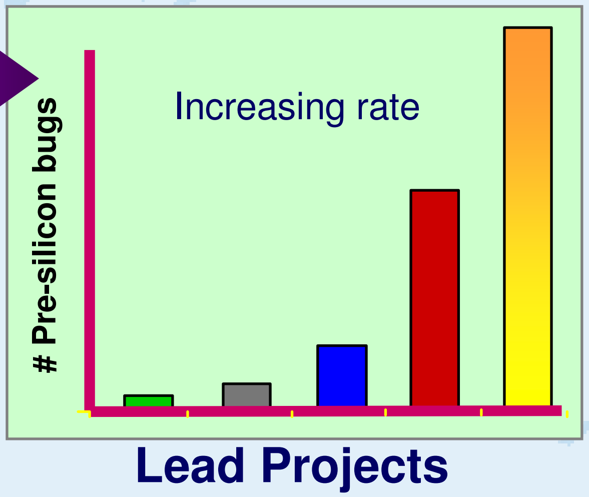
HW Design Languages

- **H**ardware **D**escription **L**anguage (HDL) is used to describe digital hardware elements
- Various levels of design abstractions are used:
  - **Behavioral**: flow control, arithmetic operators, complex delays
  - **Register Transfer Level** (RTL): Structural description of the registers and the signal changes between registers
  - **Gate level**: combinatorial logic gates (and, or, not,…)
  - **Switch level**: layout description of the wires, resistors and transistors (CMOS,PMOS, etc)
- Two main subsets:
  - Synthesizable – reflecting HW / Silicon
  - Non-Synthesizable – reflecting *instrumentation* code

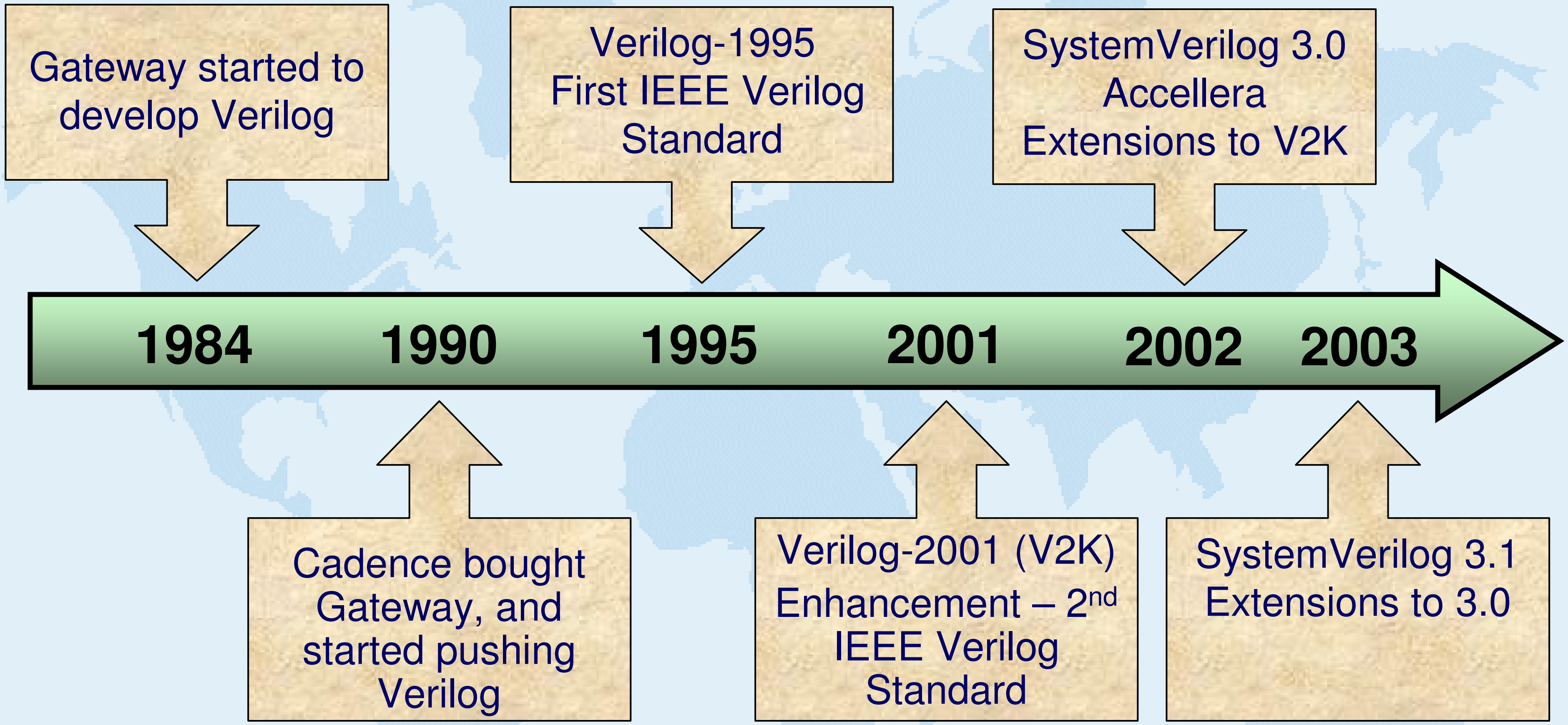
The Problem!

- Quest for Performance is driving up complexity
  - Deeper pipelines, increase in logic functionality and complexity, power issues, explosion in flops
- Explosion in lines of RTL Code making verification a lot harder
- Low Abstraction level of the RTL is driving higher verification effort and lower simulation speed
  - Trillions of cycles per lead project and more pre-silicon bug escapes
- Verification effort is reaching 60% of the total design cycle
  - Usage of different languages makes it even harder: reference models in C/C++, RTL in Verilog or VHDL, Test Benches, Assertions, Checkers, Coverage



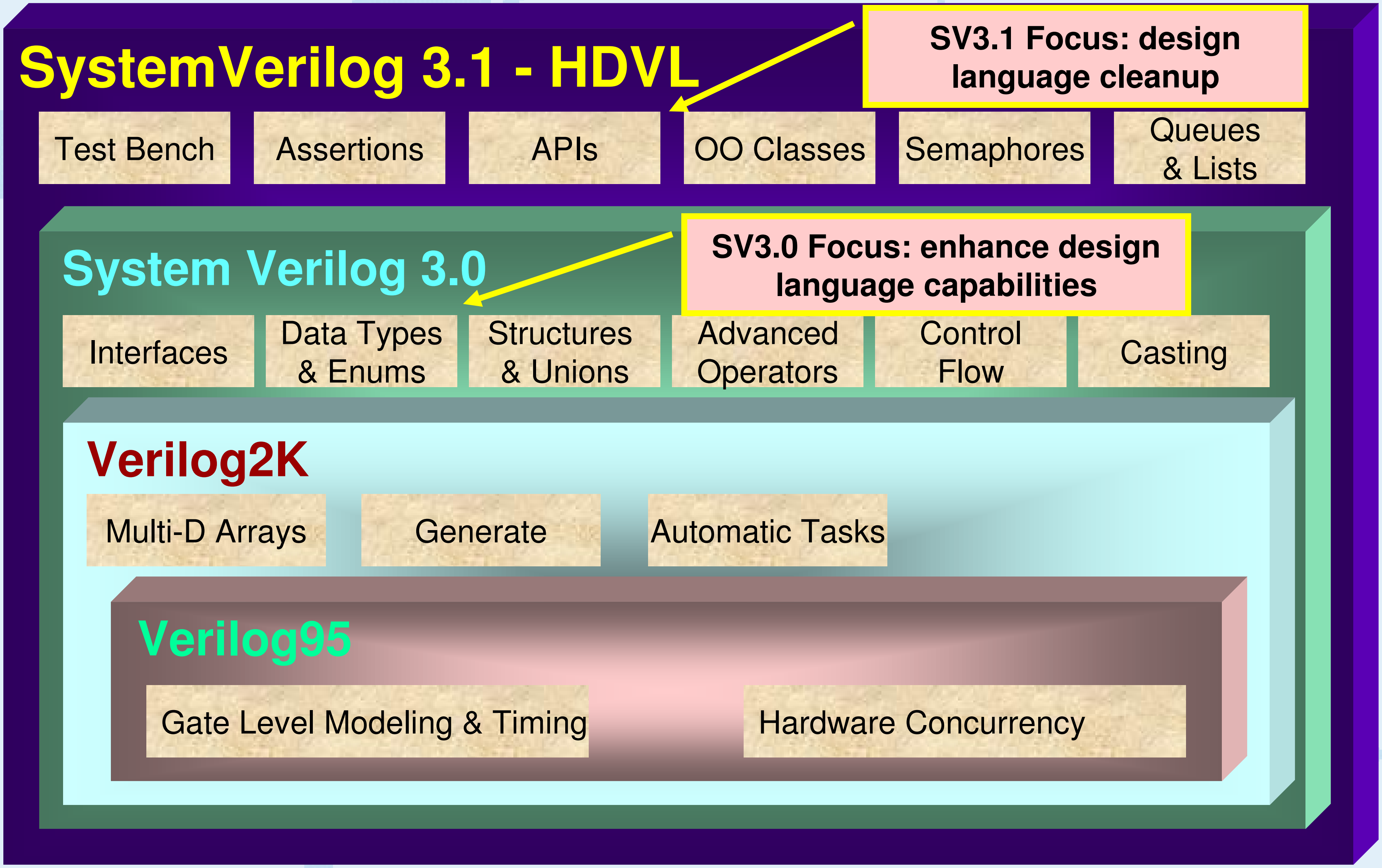


Verilog History


Gateway started to develop Verilog
Verilog-1995 First IEEE Verilog Standard
SystemVerilog 3.0 Accellera Extensions to V2K
1984
1990
1995
2001
2002
2003
Cadence bought Gateway, and started pushing Verilog
Verilog-2001 (V2K) Enhancement – 2nd IEEE Verilog Standard
SystemVerilog 3.1 Extensions to 3.0

SystemVerilog IEEE Standard is expected to become available by Oct'2005

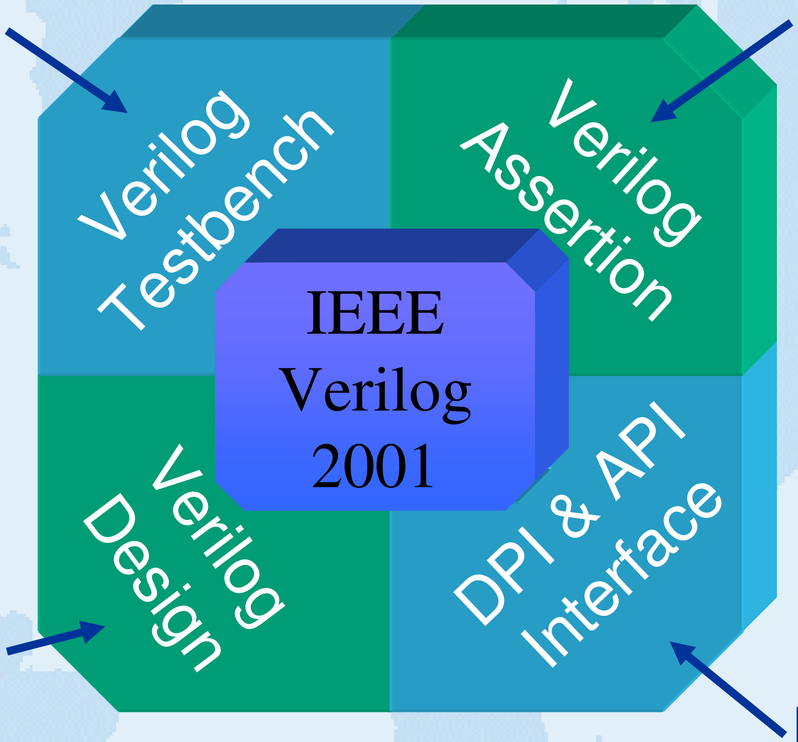
SystemVerilog



SystemVerilog Components

Transaction-Level Full Testbench Language with Coverage

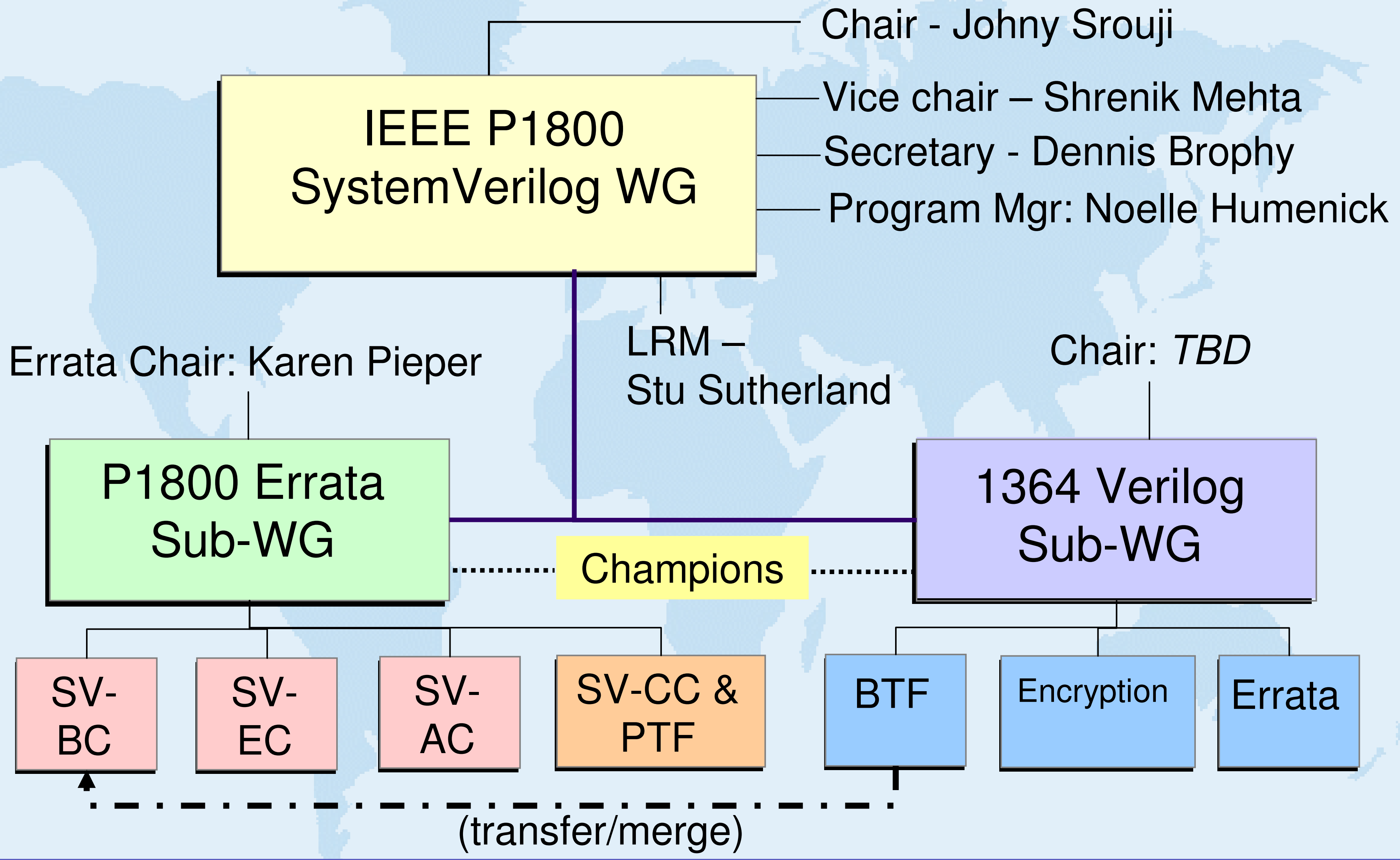
Advanced verification capability for <u>semiformal and formal</u> methods. <u>The Assertion Language Standard for Verilog</u>



<u>Design Abstraction:</u> Interface semantics, abstract data types, abstract operators and expressions

<u>Direct C</u> interface, <u>Assertion API</u> and <u>Coverage API</u>

IEEE P1800 Structure



SystemVerilog Design Modeling

SystemVerilog enhances Verilog for Design Modeling

- Data Types
  - SystemVerilog Data Types
  - Packed & Unpacked Arrays
- Data Organization
  - Structures & Unions
  - Type Casting
  - Enumerated Data Types
- C-like functionality
- Capturing Design Intent
  - always_* Procedural blocks
  - Unique and Priority Case
  - Nets and Variables
- Powerful Syntax
  - Copy / Buffering
  - Port Connections

Basic SV3.1 Data Types

```
reg r;       // 4-value Verilog-2001 single-bit datatype
integer i; // 4-value Verilog-2001 >= 32-bit datatype
bit b;       // single  bit 0 or 1
logic w;     // 4-value logic, x 0 1 or z as in Verilog
byte b;      // 2 value, 8 bit signed integer
int i;       // 2-value, 32-bit signed integer
shortint s;// 2-value, 16-bit signed integer
longint l; // 2-value, 64-bit signed integer
```

Make your own types using typedef
Use typedef to get C compatibility

```
typedef       shortint              short;
typedef       longint               longlong;
typedef       real                  double;
typedef       shortreal             float;
```

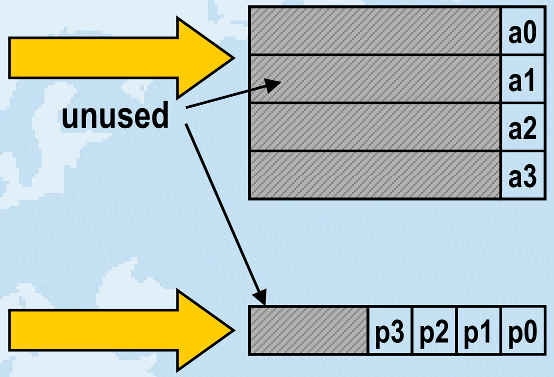
Packed And Unpacked Arrays

unpacked array of bits

```
bit a [3:0];
```

packed array of bits

```
bit [3:0] p;
```



1k 16 bit unpacked memory

```
bit [15:0] memory [1023:0];
memory[i] = ~memory[i+1];
memory[i] [15:8] = 0;
```

Packed indexes can be sliced

1k 16 bit packed memory

```
bit [15:0] [1023:0] Frame;
always @ (posedge inv)
      Frame = ~Frame;
```

Can operate on entire memory

Data Organization

- Signals are Meaningful In Groups
  - Instructions: Operation, Operands
- Verilog Provides Only Informal Grouping

```
reg [47:0] PktSrcAdr;
reg [47:0] PktDstAdr;
reg [7:0]  InstOpCde;
reg [7:0]  InstOpRF [127:0];
```

By Name

```
reg [31:0] Instruction;
`define opcode 31:16
Instruction[`opcode]
```

By Vector Location

- Better to organize data in explicit, meaningful relationships between data elements
- SystemVerilog Structs, Unions & Arrays alone or combined better capture design intent

Data Organization - Structures

```
struct {
   addr_t SrcAdr;
   addr_t DstAdr;
   data_t Data;
} Pkt;

Pkt.SrcAdr = SrcAdr;

if (Pkt.DstAdr == Node.Adr)
```

- Structs Preserve Logical Grouping
- Reference to Struct facilitates more meaningful code

Like in C but without the optional structure tags before the {

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

instruction IR;         // define variable

IR.opcode = 1;          // set field in IR
```

Data Organization - Packed Structures

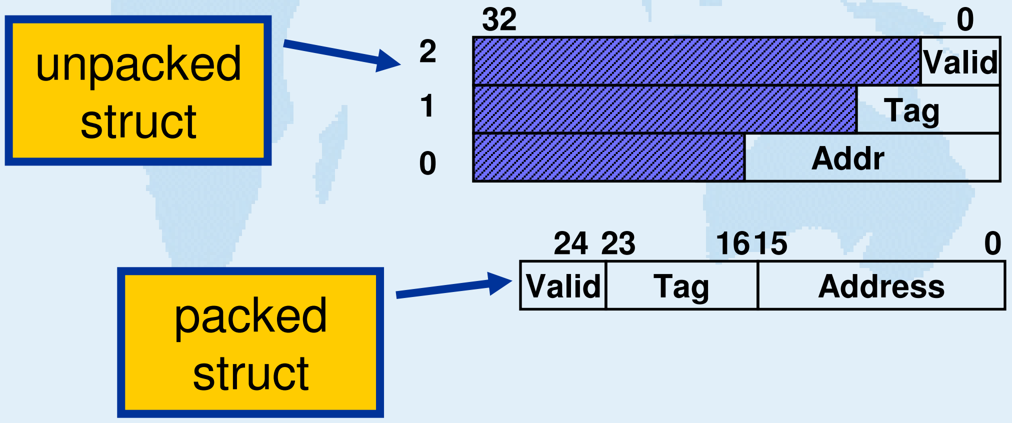
Consists of bit fields, which are packed together in memory without gaps

– They are easily converted to and from bit vectors.

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

```
`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



Data Organization - Type Casting

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

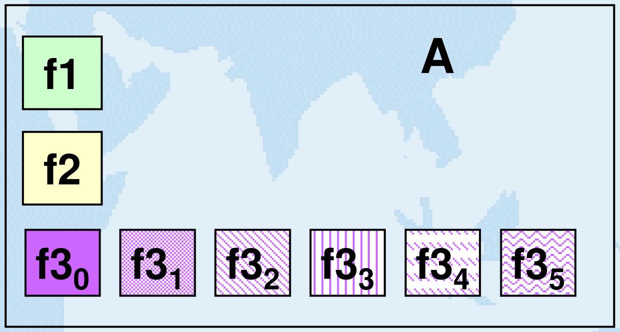
A data type can be changed by using a cast (') operation

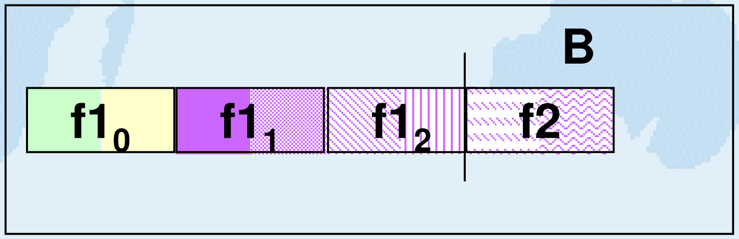
Any aggregate bit-level object can be reshaped

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);
```





Data Organization - Unions

```
typedef union {
    int n;
    real f;
    } u_type;

u_type u;
```

union

provide storage for either `int` or `real`

structs and unions can be assigned as a whole

can contain fixed size packed or unpacked arrays

Unpacked Unions Enable Single Variable to Contain Data from Multiple Types

Data Read from Unpacked Union Must Be from Last Field Written

Requires Type Awareness

```
typedef union {
   byte [5:0] bytes;
   real rlevel;
   integer ilevel;
} Data_u_t;

struct {
  Data_u_t Data;
  logic isBytes;
  logic isReal;
  logic isInteger;
} DPkt;
DPkt.Data = 3.124;
DPkt.IsReal = 1;
if (DPkt.IsReal)
 realvar = DPkt.rlevel;
```

Data Organization – Packed Unions

- Packed Unions Enable Multiple Namespaces for Same-Sized, Integer Data

```
typedef logic [7:0] byte_t;
typedef struct packed {
  logic [15:0] opcode;
  logic [1:0] Mod;
…
  logic [2:0] Base;
} Instruction_t;
typedef union packed {
   byte_t [3:0] bytes;
   Instruction_t fields;
} Instruction_u;
Instruction_u inst;
```

- Packed Unions Enable Many Convenient Name References

```
inst.fields.opcode = 16’hDEAD;
inst.bytes[1] = 8’hBE;
inst[7:0] = 8’hEF;

inst == 32’hDEADBEEF;
```

- No Need To Test Type
- Data Maps To All Members

Data Organization – Tagged Unions

- Provide type-safety and brevity
- Improve correctness
- Improve ability to reason about programs for FV

```
typedef tagged union {
    struct {
        bit [4:0] reg1, reg2, regd;
    } A
    tag


    } J
} Inst
```

```
Instr instr;
...
case (instr) matches
    tagged Add {r1,r2,rd}: rf[rd] = rf[r1] + rf[r2];
    tagged Jmp j:          case (j) matches
                             tagged JmpU a : pc = pc + a;
                             tagged JmpC {c,a}:
                                if (rf[c]) pc = a;
                           endcase
endcase
```

Data Organization – Enum

```
typedef enum {red, green, blue, yellow,
              white, black} Colors;
Colors   col;
integer  a, b;

a = blue * 3;
col = yellow;
b = col + green;
```

a=2*3=6
col=3
b=3+1=4

```
typedef enum logic [2:0] {idle,
   init, decode …} fsmstate;
fsmstate pstate, nstate;
case (pstate)
  idle: if (sync)
          nstate = init;
  init: if (rdy)
          nstate = decode;
…
endcase
typedef enum {lo,hi} byteloc;
memory[addr][hi] = data[hi];
```

- Finite State Machines
  - Currently a List of Parameters
  - Why Not A Real List of Values?
  - Enumerate formally defines symbolic set of values
- Enumerates are strongly typed to ensure assignment to value in set
- Symbolic Indexes to Make Array References More Readable

Design Intent – always_*

- In Verilog, always blocks do not guarantee capture of intent
  - If not edge-sensitive then only a warning if latch inferred

```
//forgot Else but it's
//only a synthesis warning
always @ (a or b)
    if (b) c = a;
```

- SystemVerilog introduces three new logic specific processes
  - Combinational Coding Style
  - Latch Coding Style
  - Sequential Logic

```
always_comb
    a = b & c;
```

```
always_latch
    if (en) q <= d
```

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

Allows simulation to perform some DRC

Design Intent – always_*

- Compiler Now Knows User Intent and can flag errors

```
always_comb
    if (en) q <= d;
```

ERROR: combinational logic requested but latch was inferred

```
always_ff @ (clk, rst_n)
    if (!rst_n) q <= 0;
    else        q <= d;
```

ERROR: incorrect sensitivity list
flip flop not inferred

- Both always @* and always_comb
  - Ensures synthesis-compatible sensitivity
  - Helps reduce "spaghetti code"
  - But, always_comb infers the sensitivity list from within functions called in the always block

Design Intent – Unique/Priority

- Verilog synthesis pragmas are “full_case” & “parallel_case”
  - Pass different information to synthesis and simulator
  - May cause pre- & post-synthesis simulations differences
- SystemVerilog introduces Unique and Priority
  - Passing the same information to the simulator and synthesis tool
  - Enables Simulation, Synthesis and FV to behave consistently

```
bit       [2:0] a;
unique case (a) // values 3,5,6,7 cause
                // run time warning
    0,1: $display(“0” or “1”);
    2:   $display(“2”);
    4:   $display(“4”);
endcase
```

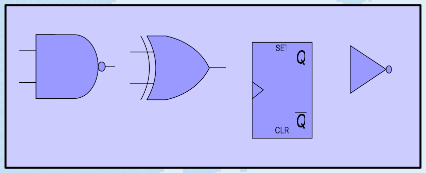
tests all case conditions and makes sure that one and only one condition matches

```
priority casez (a)// values 4,5,6,7
                  // cause run time
                 // warning
    3’b00?: $display(“0” or “1”);
    3’b0??: $display(“2” or “3”);
endcase
```

tests each case condition in order and makes sure there is at least one branch taken

Design Intent – nets vs. variables

- In Synthesized Design most nets have a single driver



- Combination of SystemVerilog variables and processes can enforce single driver/avoid multiple drivers
- Variables may be driven by only one of the following
  - continuous assignment
  - always_comb
  - always_latch
  - always_ff
  - module port
- Compilers Can Catch Your Mistakes

- Multiply Driven, Variable-Strength Nets generally take great care to design. Would like these nets to stand out in the design.
- Use Net Types to Distinguish

```
trireg [1:0] dbus;
assign dbus = wen[0] ? cff : 'z;
assign dbus = wen[1] ?
{cff[0],cff[1]} : 'z;
```

```
logic [1:0] n0,n1,n2,n3,n4;
assign n0 = {ina,inb};
always_comb
  if (sela) n1 = ina;
  else n1 = inb;
always_latch
  if (sela) n2 <= ina;
always_ff @(posedge ck)
  n3 <= ina;
sbuf buf1 (.di(n1),.do(n4));
```
~~sbuf buf2 (.di(n2),.do(n4));~~

Powerful Syntax – .* Port Connections

- Creating netlists by hand is tedious
- Generated netlists are unreadable
  - Many signals in instantiations
  - Instantiations cumbersome to manage
- Implicit port connections dramatically improve readability
- Use same signal names up and down hierarchy where possible
- Emphasize where port differences occur

```
module top();
  logic rd,wr;
  tri [31:0] dbus,abus;
  tb tb1(.*);
  dut dut1(.*);
endmodule
```

```
module top();
  logic rd,wr;
  tri [31:0] dbus,abus;
  tb tb(.*, .ireset(start),
            .oreset(tbreset));
  dut d1(.*,.reset(tbreset[0]));
  dut d2(.*,.reset(tbreset[1]));
endmodule
```

Familiar C Features

```
do
  begin
    if ( (n%3) == 0 ) continue;
    if (foo == 22) break;
  end
while (foo != 0);
…
```

continue starts next loop iteration

break exits the loop

works with:
for
while
forever
repeat
do while

Blocking Assignments as expressions

```
if ( (a=b) ) …
while ((a = b || c))
```

Extra parentheses required to distinguish from **if(a==b)**

Auto increment/ decrement operators

```
x++;
if (--c > 17) c=0;
```

```
a += 3;
s &= mask;
f <<= 3;
```

Assignment Operators Semantically equivalent to blocking assignment

Wildcard Comparisons X and Z values act as wildcards

```
a =?= b
a !?= b
```

SystemVerilog for Verification

- SystemVerilog extends the language to include Verification and Test-Bench modeling constructs

- Extended Data Types
  - Dynamic Arrays
  - Associative Arrays
- Process Synchronization
  - Semaphors and Event Variables
  - Mailbox queues
  - Processes and Threads
- Object Oriented Programming
  - Class; Object; Methods
- Communication Encapsulation
  - Interfaces
- Assertion Based Design
  - Assertions Constructs
  - Usage

Dynamic Arrays

Declaration syntax

```
<type> <identifier> [ ];
bit[3:0] dyn[ ];
```

Initialization syntax

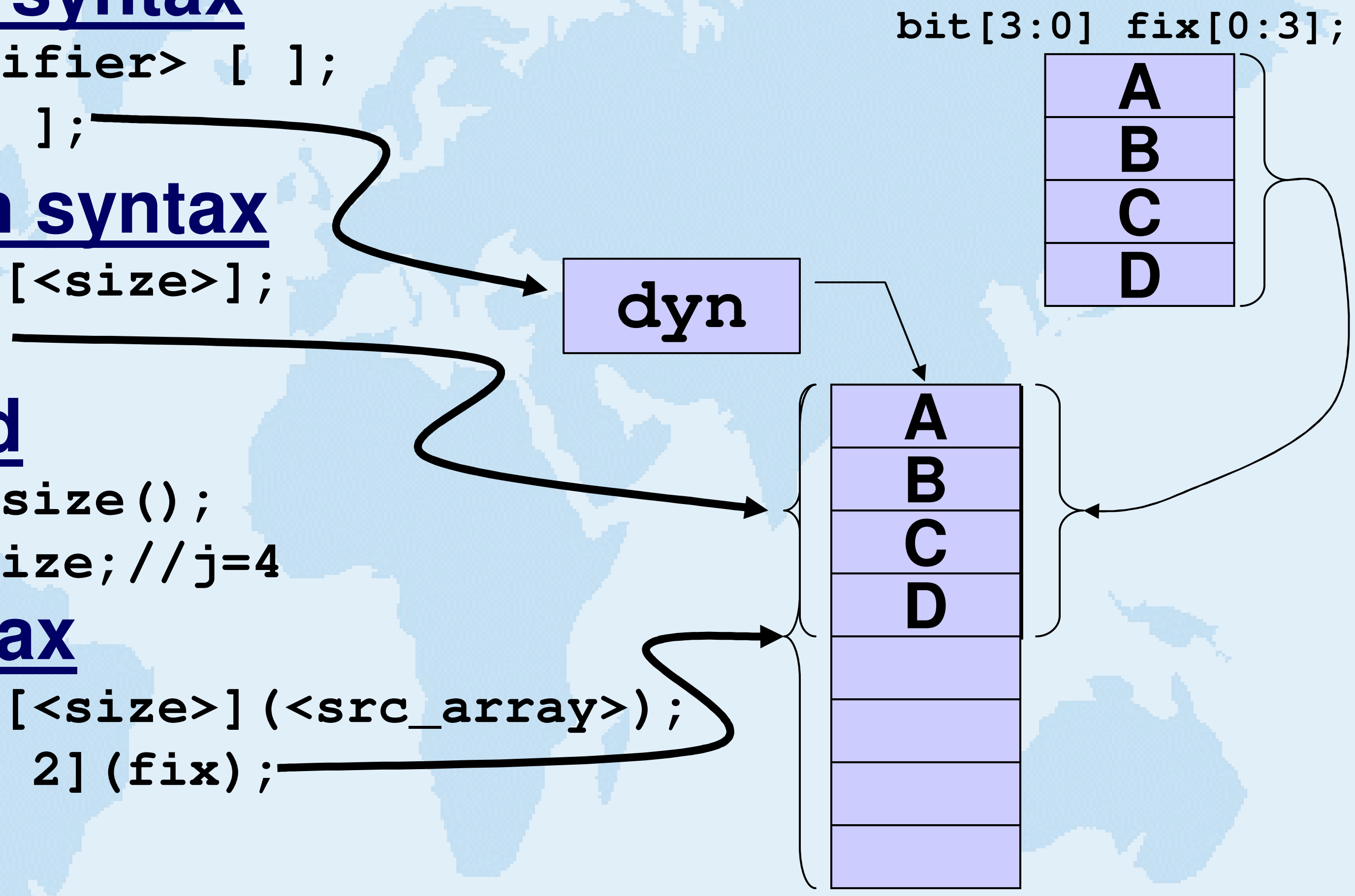
```
<array> = new[<size>];
dyn = new[4];
```

Size method

```
function int size();
int j = dyn.size;//j=4
```

Resize syntax

```
<array> = new[<size>](<src_array>);
dyn = new[j * 2](fix);
```



Dynamic Arrays

Declaration syntax

```
<type> <identifier> [ ];
bit[3:0] dyn[ ];
```

Initialization syntax

```
<array> = new[<size>];
dyn = new[4];
```

Size method

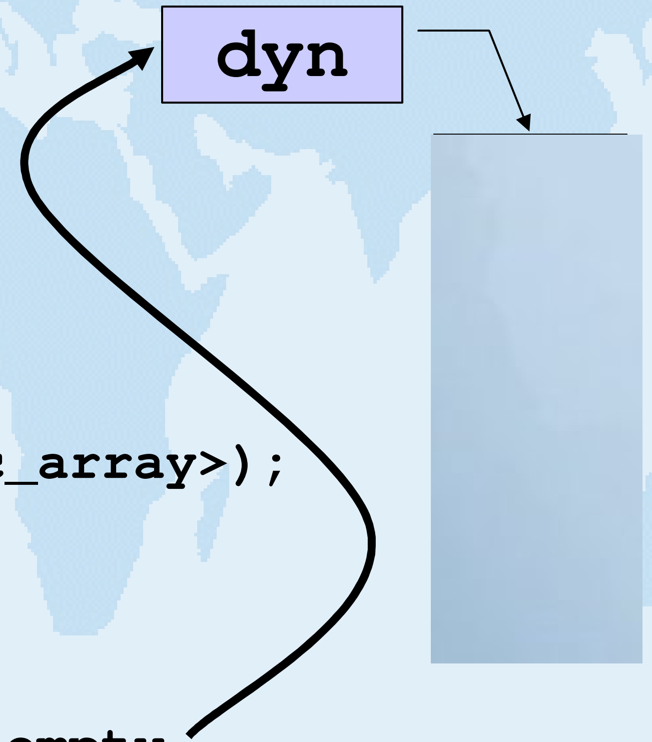
```
function int size();
int j = dyn.size;//j=4
```

Resize syntax

```
<array> = new[<size>](<src_array>);
dyn = new[j * 2](fix);
```

Delete method

```
function void delete();
dyn.delete; // dyn is now empty
```



Associative Arrays

- Excellent for Sparse Storage
- Elements Not Allocated Until Used
- Index Can Be of Any Packed Type, String or Class

Declaration syntax

```
<type> <identifier> [<index_type>];
<type> <identifier> [*]; // “arbitrary” type
```

```
int imem[*];
imem[ 2’b3 ] = 1;
imem[ 16’hffff ] = 2;
imem[ 4b’1000 ] = 3;
```

```
struct packed {int a; logic[7:0] b}
mystruct;
int myArr [mystruct];
  //associative array indexed by mystruct
```

Built-in Methods

```
num(), delete([index]), exists(index);
first/last/prev/next(ref index);
```

Ideal for Dealing with Sparse Data

Object-Oriented Programming

- Organize programs in the same way that objects are organized in the real world
- Break program into blocks that work together to accomplish a task, each block has a well defined interface
- Focuses on the data and what you are trying to do with it rather than on procedural algorithms

- Class – A blueprint for a house
  - Program element "containing" related group of features and functionality.
  - Encapsulates functionality
  - Provides a template for building objects
- Properties – It has light switches
  - Variables specific to the class
- Methods – Turn on/off the lights
  - Tasks/functions specific to the class
- Object – The actual house
  - An object is an instance of a class

OOP - Class Definition

Definition Syntax

```
class name;
<data_declarations>;
<task/func_decls>;
endclass
```

extern keyword allows for out-of-body method declaration

```
class Packet;
  bit[3:0]    cmd;
  int         status;
  myStruct    header;
  function int get_status();
    return(status);
  endfunction
  extern task set_cmd(input bit[3:0] a);
endclass
```

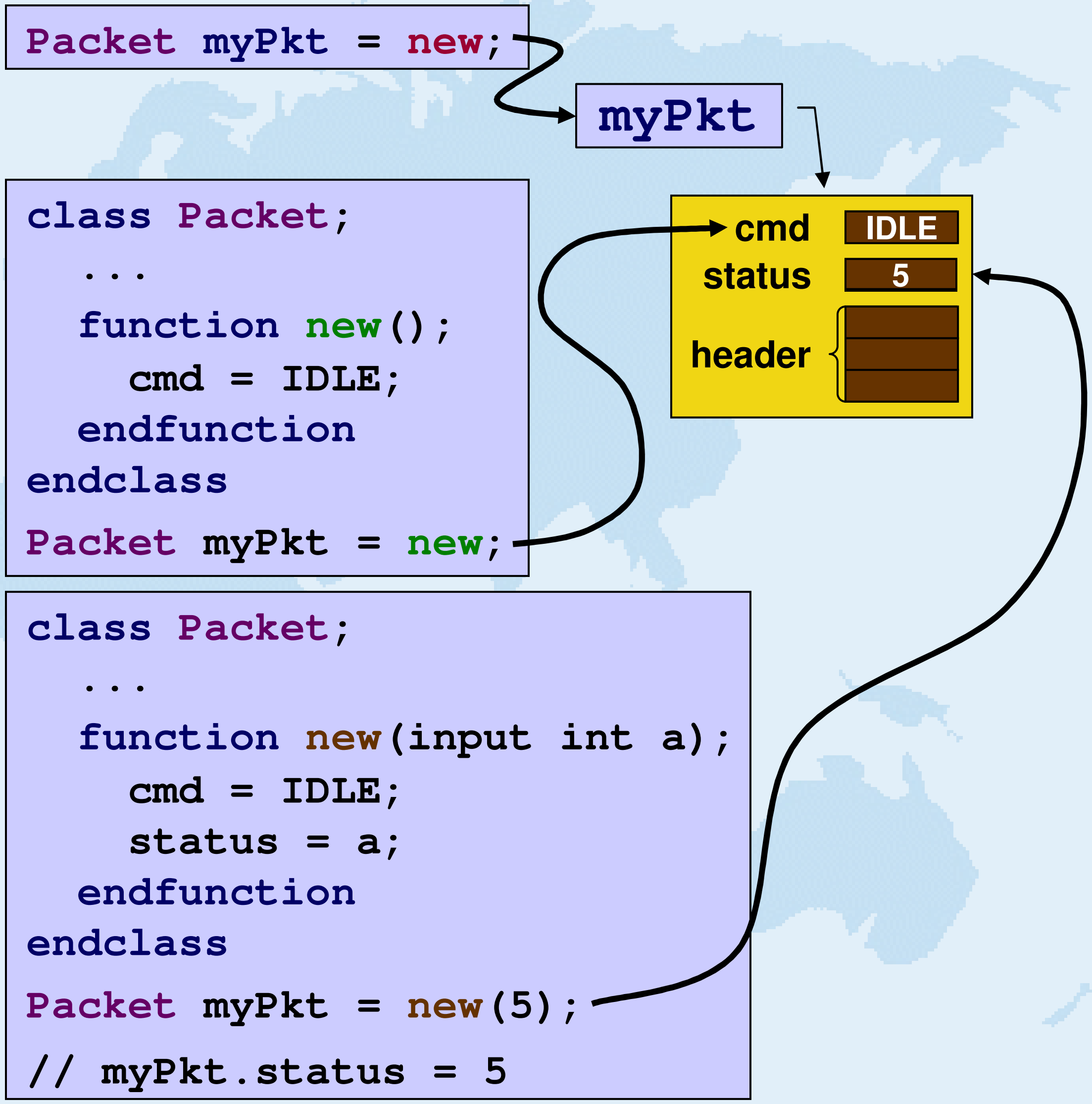
"::" operator links method declaration to Class definition

```
task Packet::set_cmd(input bit[3:0] a);
  cmd = a;
endtask
```

Class declaration does not allocate any storage

OOP - Class Instantiation

- Objects Allocated and Initialized Via Call to the **`new`** *Constructor* Method
  - All objects have built-in **`new`** method
    - No arguments
    - Allocates storage for all data properties
  - User-defined **`new`** method can initialize and/or do other things



```
Packet myPkt = new;
```

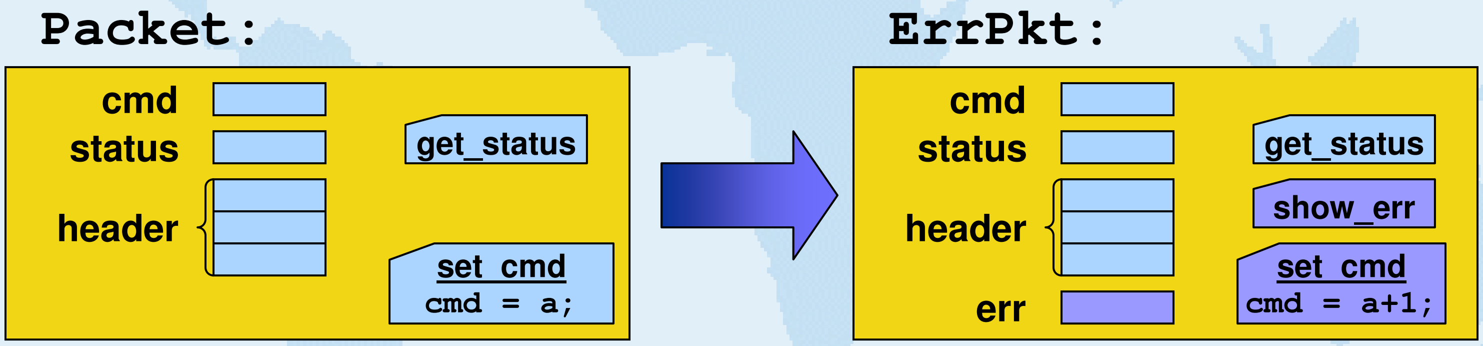
```
class Packet;
  ...
  function new();
    cmd = IDLE;
  endfunction
endclass
Packet myPkt = new;
```

```
class Packet;
  ...
  function new(input int a);
    cmd = IDLE;
    status = a;
  endfunction
endclass
Packet myPkt = new(5);
// myPkt.status = 5
```

OOP - Class Inheritance & Extension

- Keyword ***extends*** Denotes Hierarchy of Definitions
  - Subclass inherits properties and methods from parent
  - Subclass can redefine methods explicitly

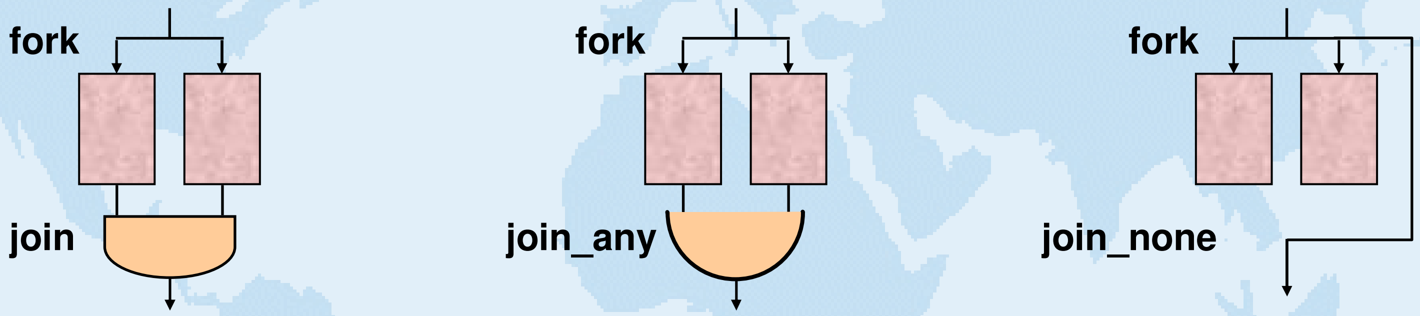
```
class ErrPkt extends Packet;
  bit[3:0] err;
  function bit[3:0] show_err();
    return(err);
  endfunction
  task set_cmd(input bit[3:0] a);
    cmd = a+1;
  endtask // overrides Packet::set_cmd
endclass
```



Allows Customization Without Breaking or Rewriting Known-Good Functionality in the Parent Class

Process Synchronization

- SystemVerilog adds a powerful and easy-to-use set of synchronization and communication mechanisms
  - Which can be created and reclaimed dynamically
  - Dynamic parallel processes using **`fork/join_any`** and **`fork/join_none`**

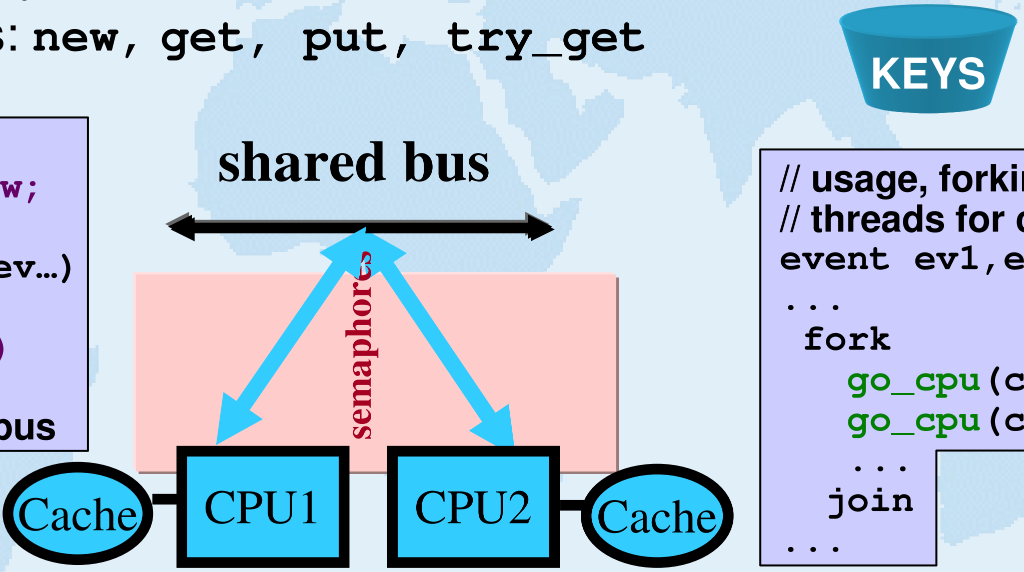


- SystemVerilog adds
  - a ***semaphore*** built-in class - for synchronization and mutual exclusion to shared resources
  - a ***mailbox*** built-in class - communication channel between processes
  - **event** data type to satisfy system-level synchronization requirements

Process Synchronization – Semaphores & Event Variables

- Events – enhanced from V2K
  - Events are variables – can be copied, passed to tasks, etc.
  - `event.triggered;` // persists throughout timeslice, avoids races
- Semaphore – Built-in Class
  - Synchronization for arbitration of shared resources, keys.
  - Mutual Exclusivity control
  - Built-in methods: `new, get, put, try_get`

```
// main program .....
 semaphore  shrdBus =new;
  ...
task go_cpu(id, event ev…)
begin ...
    @(ev) shrdBus.get ()
    ...//access granted
    ...//activity on the cpu bus
    shrdBus.put()
    ...
endtask
```



```
// usage, forking parallel
// threads for cpu access
event ev1,ev2;
...
 fork
   go_cpu(cpu1,ev1);
   go_cpu(cpu2,ev2);
   ...
  join
...
```

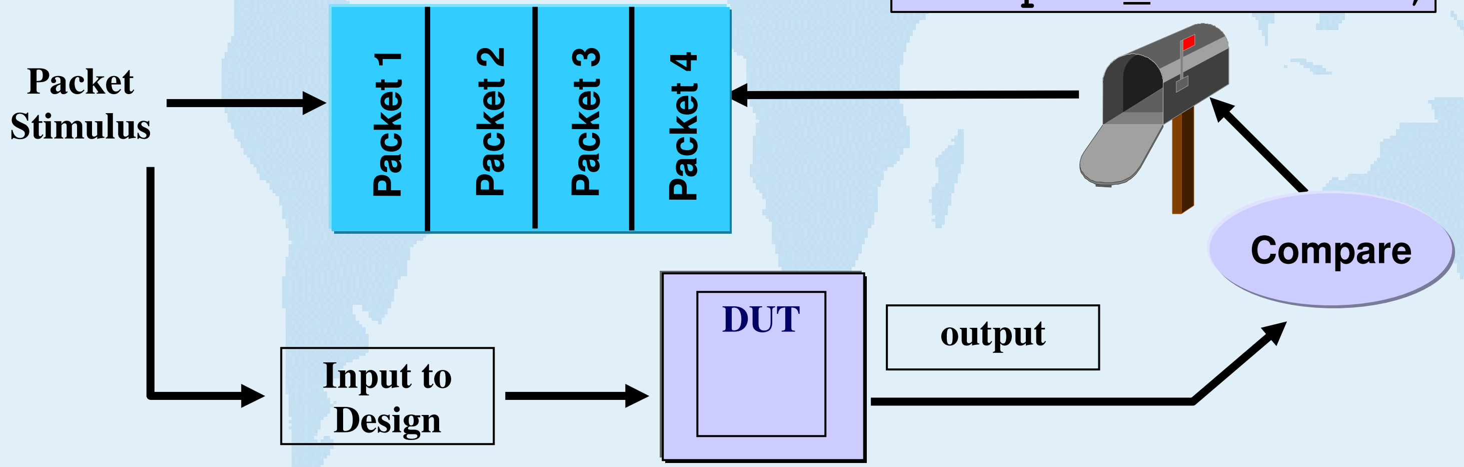
Guarantees Race-Free Synchronization Between Processes

Process Synchronization - Mailbox

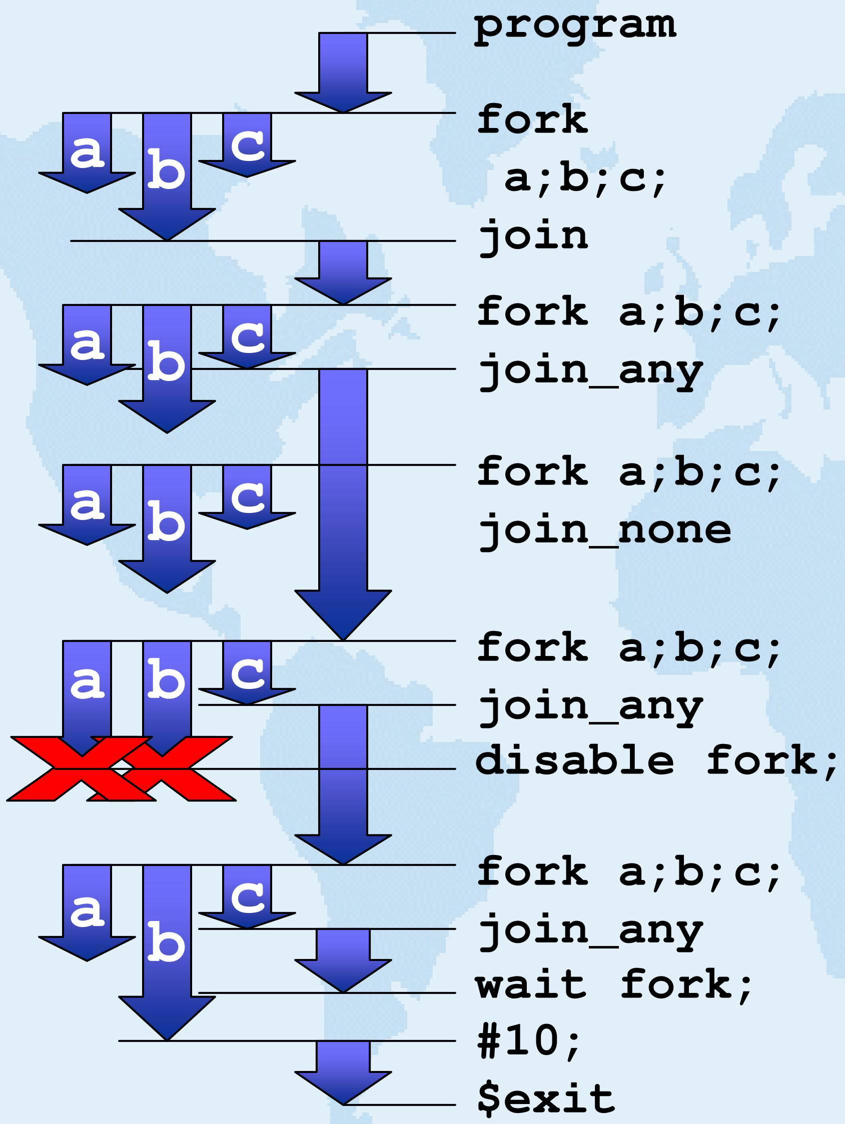
- Mailbox features
  - FIFO message queue: passes data between threads
  - Can suspend thread, used for data checking
- Mailbox built-in methods
  - new(), num(), put(), try_put(), get(), try_get(), peek(), try_peek()

Testbench compares the actual output with expected output [Packets]

```
 mailbox  pktMbx = new;
pktMbx.put(inPkt1);                    pktMbx.get(outPkt);
                                       compare_twoPackets;
```



Process Synchronization - Example



```
program
fork
 a;b;c;
join
fork a;b;c;
join_any
fork a;b;c;
join_none
fork a;b;c;
join_any
disable fork; // kill all child processes
fork a;b;c;
join_any
wait fork;     // wait for children to complete
#10;
$exit
```

- Threads are created via fork…join
- Threads execute until a blocking statement
  - mailbox, semaphore, etc
- “wait fork”; “disable fork” are introduced
- $exit terminates the main program thread

Interface

- An Interface Provides a new hierarchical structure
  - Encapsulates communication
  - Captures Interconnect and Communication
  - Separates Communication from Functionality
  - Eliminates “Wiring” Errors
  - Enables abstraction in the RTL

```
int i;
logic [7:0] a;

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

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

```
int i;
wire [7:0] a;

interface intf;
    int i;
    wire [7:0] a;
endinterface : intf
```

How Interfaces work

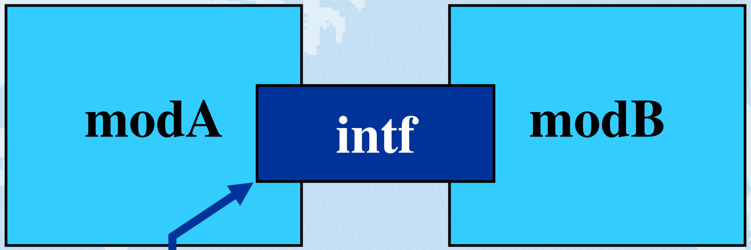
```
interface intf;
  bit   A,B;
  byte  C,D;
  logic E,F;
endinterface

module top ( );
intf w;

modA m1(w);
modB m2(w);
endmodule

module modA (intf i1);
endmodule
module modB (intf i1);
endmodule
```

Instantiate Interface



An interface is similar to a module straddling two other modules

An interface can contain anything that could be in a module except other module definitions or instances

Allows structuring the information flow between blocks

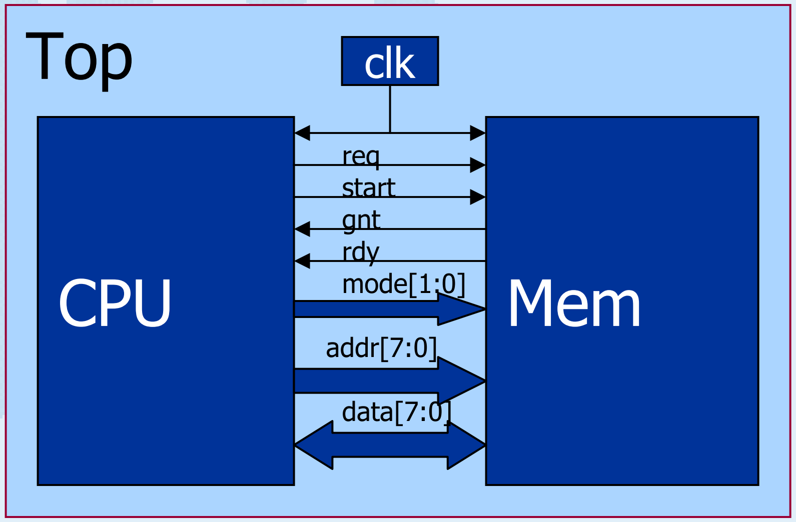
Example without Interface

```
module memMod(input  logic req,
                     bit clk,
                     logic start,
                     logic[1:0] mode,
                     logic[7:0] addr,
              inout  logic[7:0] data,
              output logic gnt,
                     logic rdy);
always @(posedge clk)
  gnt <= req & avail;
endmodule

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

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

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



Example Using Interfaces

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

Bundle signals in interface

Use `interface` keyword in port list

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

Refer to intf signals

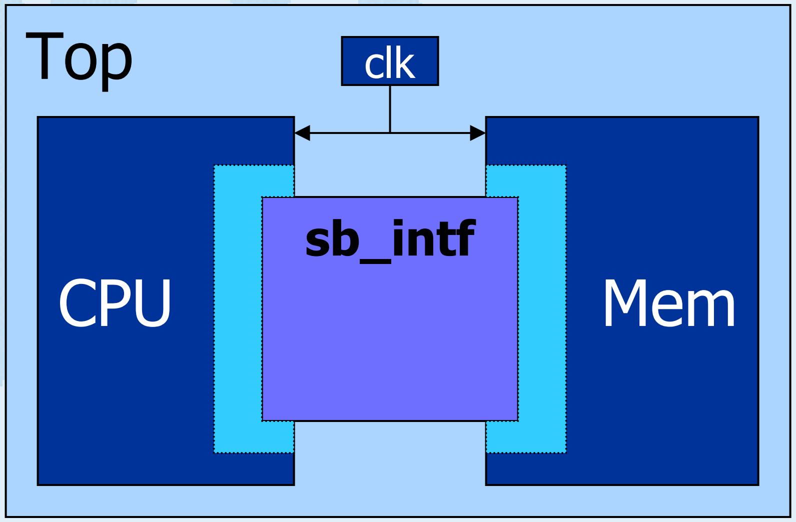
```
module cpuMod(interface b,
              input bit clk);
endmodule
```

```
module top;
  bit clk = 0;
  simple_bus sb_intf;

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

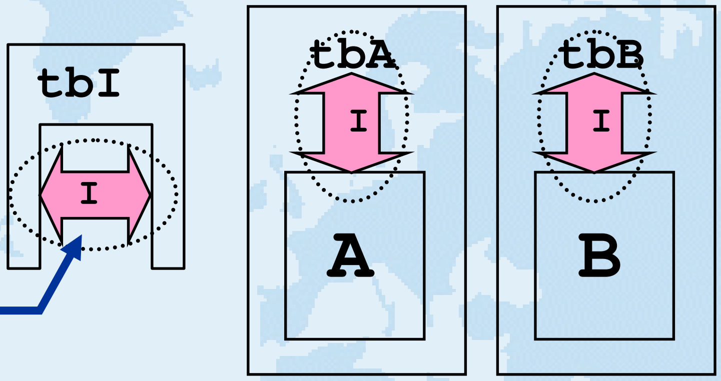
interface instance

Connect interface



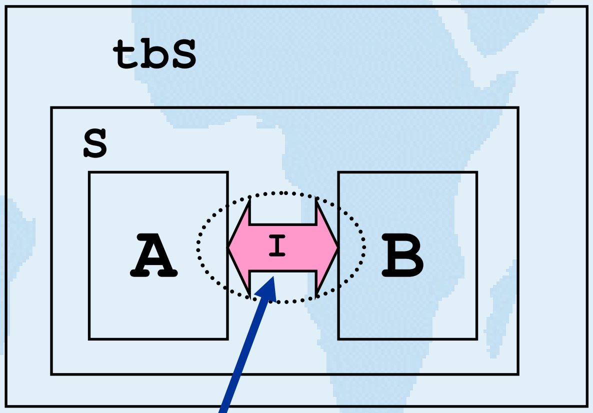
Interface Verification Benefits

- Pre-Integration



Test interface in isolation

- Post-Integration



Protocol bugs already flushed out

- Interfaces provide reusable components
- tbA and tbB are 'linked'
- Interface is an executable spec
- Wiring up is simple and not error prone
- Interfaces can contain protocol checkers and coverage counters

SystemVerilog Assertions

- A concise description of desired / undesired behavior
- Supports Assertion Based Verification methodology
  - White-box (inside block) assertions
  - Black-box (at interface boundaries) assertions
- Usability
  - Easy to code, understand and use by Design and Verification Engineers
- Monitoring the design
  - Concurrent (“standalone”) assertions
  - Procedural (“embedded”) assertions
- Formalism
  - Formal semantics to ensure correct analysis
  - Consistent semantics between simulation and formal design validation approaches

Assertion Types

- Two kinds of Assertions: Immediate and Concurrent
- Immediate Assertions:
  - Appears as a procedural statement
  - Follows simulations semantics, like an “if”

```
assert ( expression ) action_block;
```

  - Action block executes *immediately and c*an contain system tasks to control severity, for example: `$error`, `$warning`
- Concurrent Assertions:
  - Appears outside/inside procedural context
  - Follows cycle semantics using sampled values

```
assert property ( property_instance_or_spec )
           action_block;
```

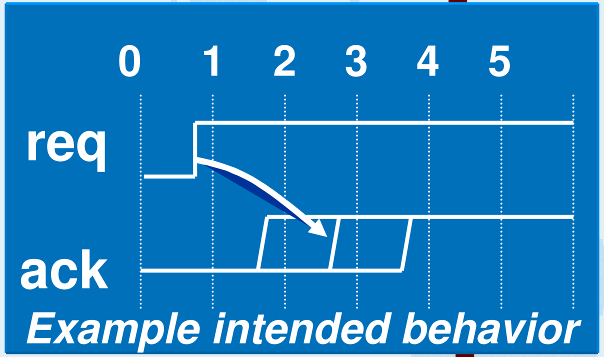
  - Action block executes in *reactive* region and can contain system tasks to control severity, for example: `$error`, `$warning`

Assertions Example

➔ “After request signal is asserted, acknowledge signal must come 1 to 3 cycles later”

SVA Assertion

```
property req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endproperty
as_req_ack: assert property (req_ack);
```



Example intended behavior

V2K Assertion

```
always @(posedge req)
  begin
     repeat (1) @(posedge clk);
     fork: pos_pos
        begin
           @(posedge ack)
           $display("Assertion Success",$time);
            disable pos_pos;
        end
       begin
          repeat (2) @(posedge clk);
          $display("Assertion Failure",$time);
          disable pos_pos;
       end
     join
  end // always
```

Assertions Usage @Intel

- RTL assertions are a powerful validation tool, and have been used in Intel for over a decade
- Basic combinational assertions
  - Most are either FORBIDDEN or MUTEX
- Sequential assertions improve the ability to capture design intent. Used to capture:
  - Assumptions on the interface
  - Expected output
  - Local relations
- Template Library: consists of dozens of temporal and combinatorial properties for:
  - Safety Properties: expresses that "something bad will not happen" during a system execution
  - Liveness Properties: expresses that "something good must happen" during an execution

Assertions Usage @Intel

- Easier for designers to write assertions using the template library
  - No need to ramp-up on a formal specification language
  - Hides the subtle implementation details
- RTL assertions caught >25% of all bugs!
  - High in bug hunting in CTE – right after designated CTE checkers
  - Very high in bug hunting at the FC level
- RTL assertions were the first to fail
  - They were more local than checkers and mostly combinatorial
- RTL assertions shortened the debug process
  - Assertions point directly at the bug

SystemVerilog DPI – WHY?
Direct **P**rogramming **I**nterface

- Users need a simple way of invoking foreign functions from Verilog and getting results back
- VPI and PLI are not easy to use
  - Even trivial usage requires detailed knowledge
  - Many users don’t need the sophisticated capabilities
  - Verilog can invoke C functions but C functions can’t invoke Verilog functions
- SystemVerilog includes assertions. These were not addressed by any prior Verilog API
- DPI and VPI extensions are based on Direct-C donation from Synopsys

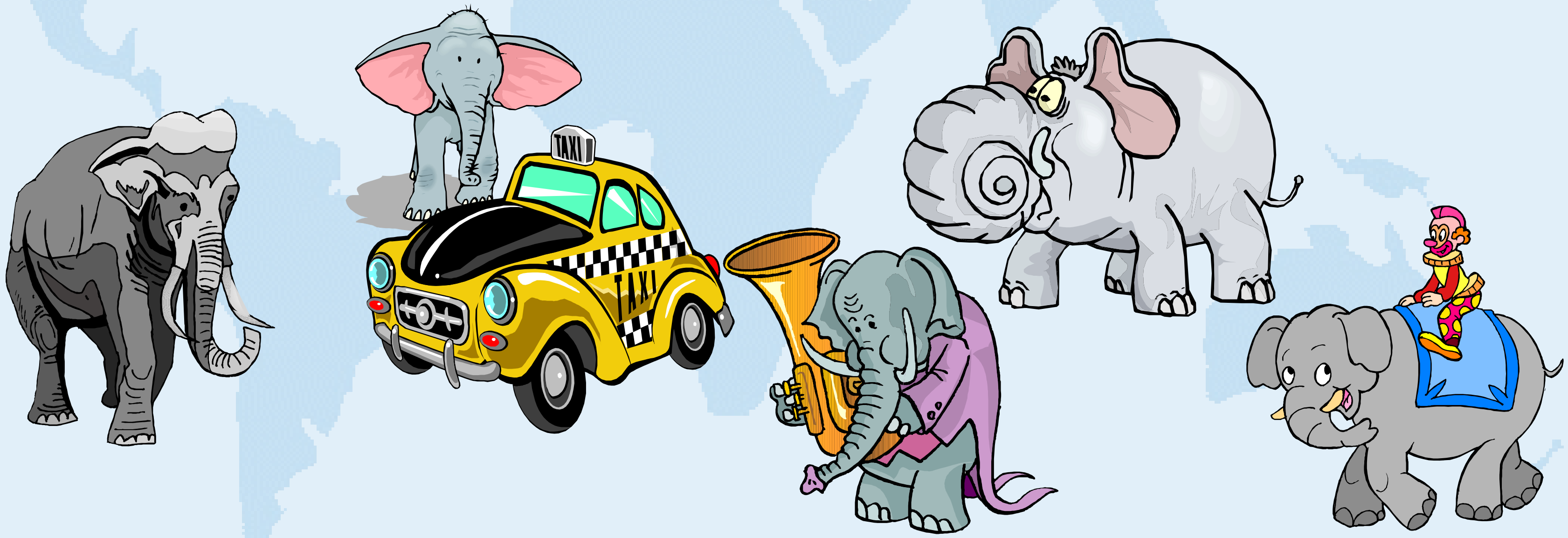
SystemVerilog DPI - Overview

- DPI is a natural inter-language function call interface between SystemVerilog and C/C++
  - The standard allows for other foreign languages in the future
- DPI relies on C function call conventions and semantics
- On each side, the calls look and behave the same as native function calls for that language
  - On SV side, DPI calls look and behave like native SV functions
  - On C side, DPI calls look and behave like native C functions
- Binary or source code compatible
  - Binary compatible in absence of packed data types (svdpi.h)
  - Source code compatible otherwise (svdpi_src.h)

The QUIZ

➔Question: How do you get 20 Elephants into a Volkswagen beetle?

I don’t know! BUT …



QUIZ

```
module m(
        pipe1_clk ,pipe1_dat,pipe1_ctrl,
        pipe2_clk ,pipe2_dat,pipe2_ctrl);
input pipe1_clk,pipe1_dat,pipe1_ctrl;
output pipe2_clk,pipe2_dat,pipe2_ctrl;
wire pipe1_clk,pipe1_dat,pipe1_ctrl;
reg pipe2_clk,pipe2_dat,pipe2_ctrl;
always @(pipe1_clk, pipe1_dat,
pipe1_ctrl) begin
 pipe2_clk<=pipe1_clk;
 pipe2_dat<=pipe1_dat;
 pipe2_ctrl<=pipe1_ctrl;
end
endmodule
```

→How do you get 20 lines of Verilog into 4 cm?

```
typdef struct ( logic clk,dat,ctrl) pipe;
module m(input pipe pipe1,
                   output pipe pipe2);
always_comb
        pipe2<= pipe1;
endmodule
```

QUIZ

```
module m(
        pipe1_clk ,pipe1_dat,pipe1_ctrl,
        pipe2_clk ,pipe2_dat,pipe2_ctrl);
input pipe1_clk,pipe1_dat,pipe1_ctrl;
output pipe2_clk,pipe2_dat,pipe2_ctrl;
wire pipe1_clk,pipe1_dat,pipe1_ctrl;
reg pipe2_clk,pipe2_dat,pipe2_ctrl;
always @(pipe1_clk, pipe1_dat,
pipe1_ctrl) begin
 pipe2_clk<=pipe1_clk;
 pipe2_dat<=pipe1_dat;
 pipe2_ctrl<=pipe1_ctrl;
end
endmodule
```

→How do you get 20 lines of Verilog into 4 cm?

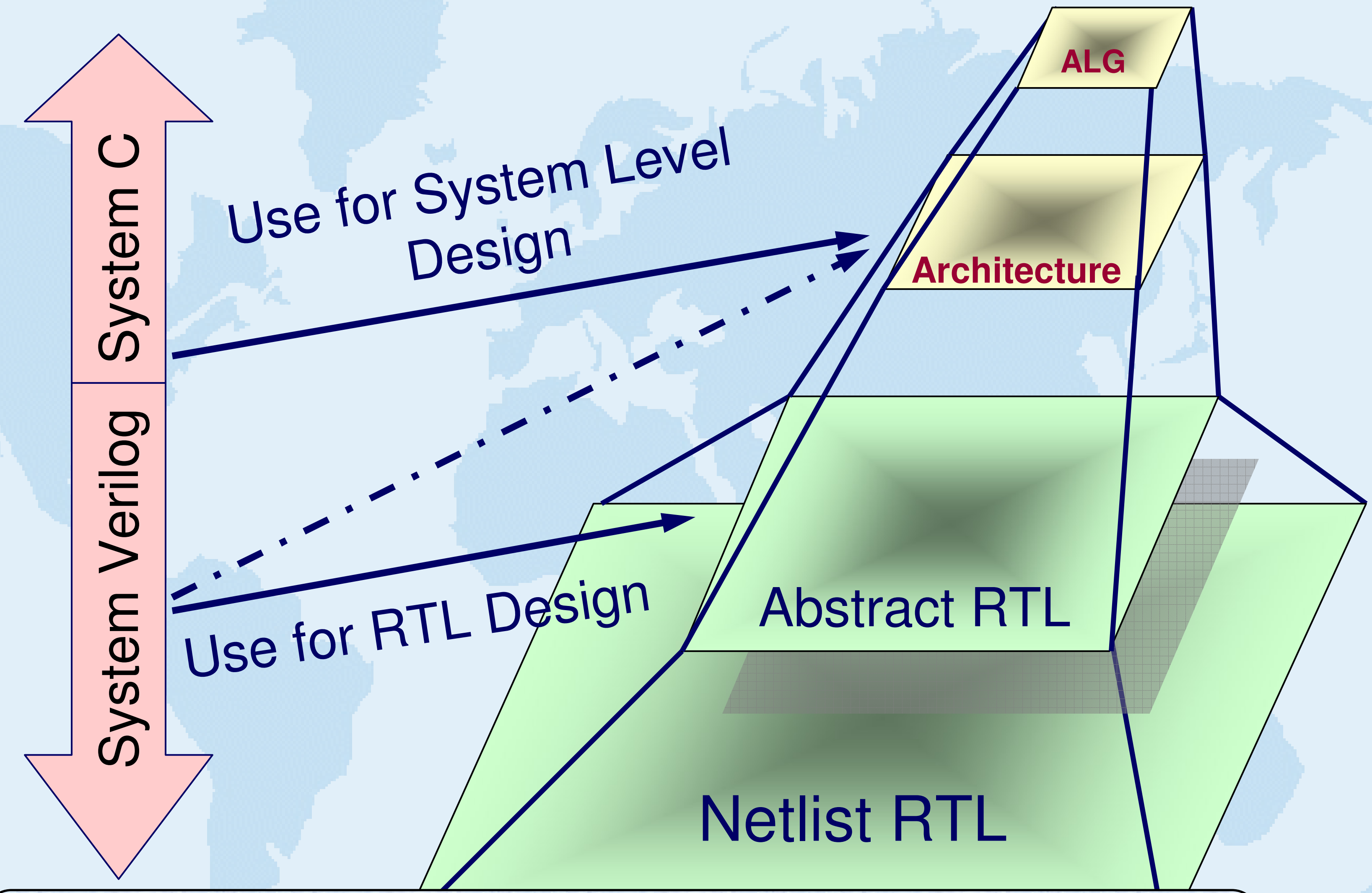
```
typdef struct ( logic clk,dat,ctrl) pipe;
module m(input pipe pipe1,
                 output pipe pipe2);
always_comb
        pipe2<= pipe1;
endmodule
```

Conclusions

- SystemVerilog was demonstrated as:
  - The next generation of HW Design Language
  - Provides a comprehensive language coverage for Modeling and Verification (HDVL)
  - Enables design abstraction; more accurate modeling and capture of designer intent
  - Integrates to external languages
  - Has wide and increasing support in the EDA
  - Being used on real life complex design projects
  - Being standardized under IEEE P1800 with the goal of convergence with Verilog

Q&A

SystemVerilog & SystemC



System Verilog & System C Complement Each Other in ARCH → Physical Design Flow