# SystemVerilog for VHDL Users

**Tom Fitzpatrick**
**Principal Technical Specialist**
**Synopsys, Inc.**

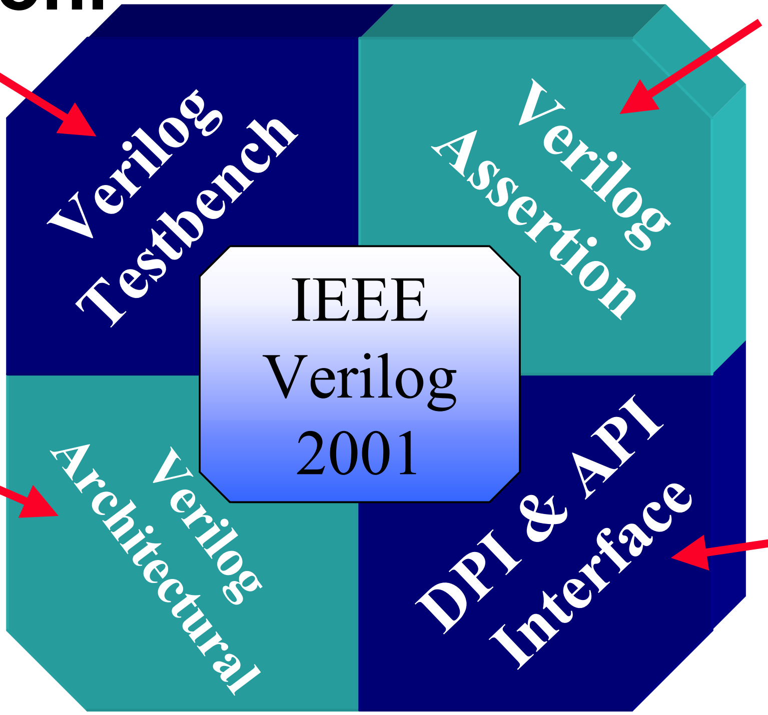
## Agenda

- Introduction
- SystemVerilog Design Features
- SystemVerilog Assertions
- SystemVerilog Verification Features
- Using SystemVerilog and VHDL Together

# SystemVerilog Charter

- Charter: Extend Verilog IEEE 2001 to higher abstraction levels for Architectural and Algorithmic Design , and Advanced Verification.

Transaction-Level Full Testbench Language with Coverage



Advanced verification capability for semiformal and formal methods. The Assertion Language Standard For Verilog

**Design Abstraction:** Interface semantics, abstract data types, abstract operators and expressions

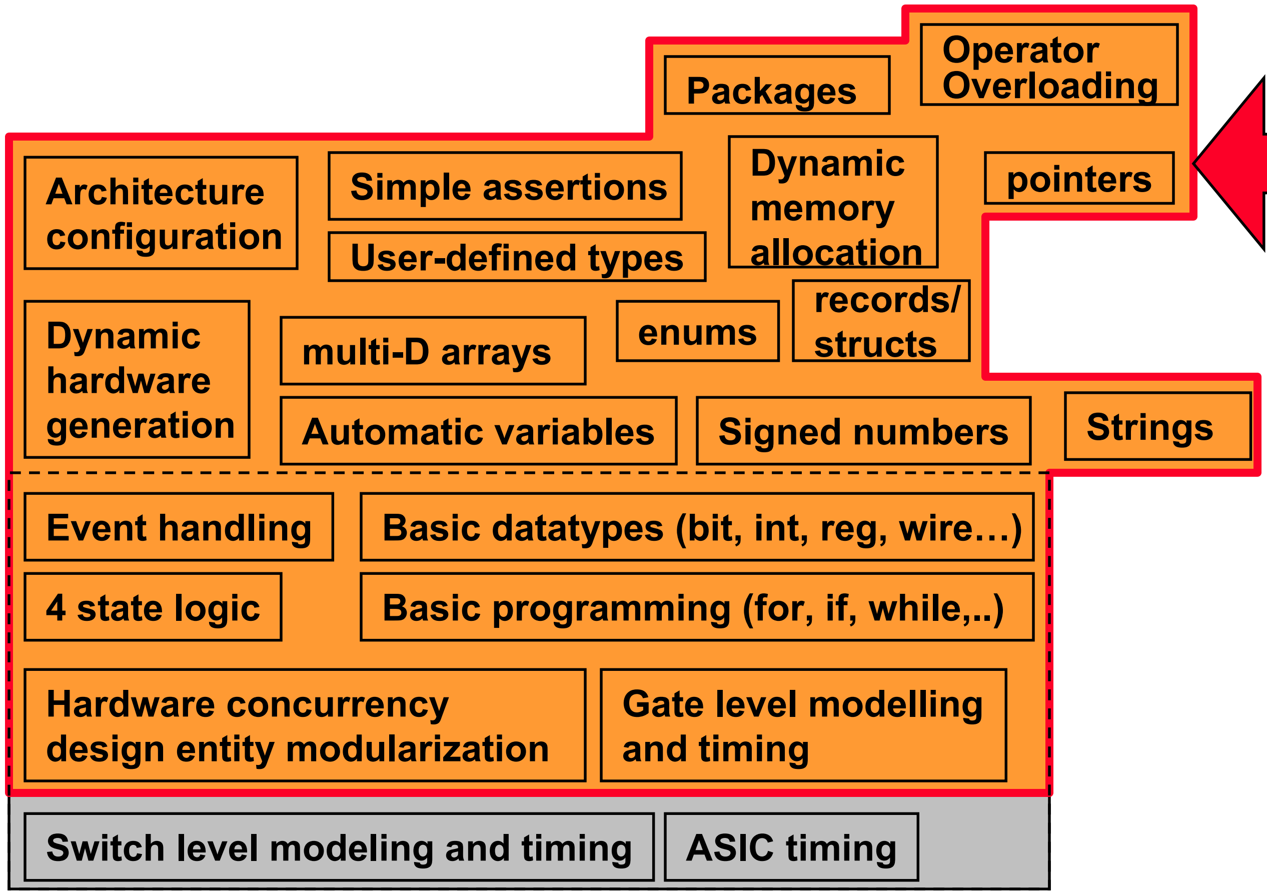
Direct C interface, Assertion API and Coverage API

# SystemVerilog: Verilog 1995

- Event handling
- Basic datatypes (bit, int, reg, wire…)
- 4 state logic
- Basic programming (for, if, while,..)
- Hardware concurrency design entity modularization
- Gate level modelling and timing
- Switch level modeling and timing
- ASIC timing

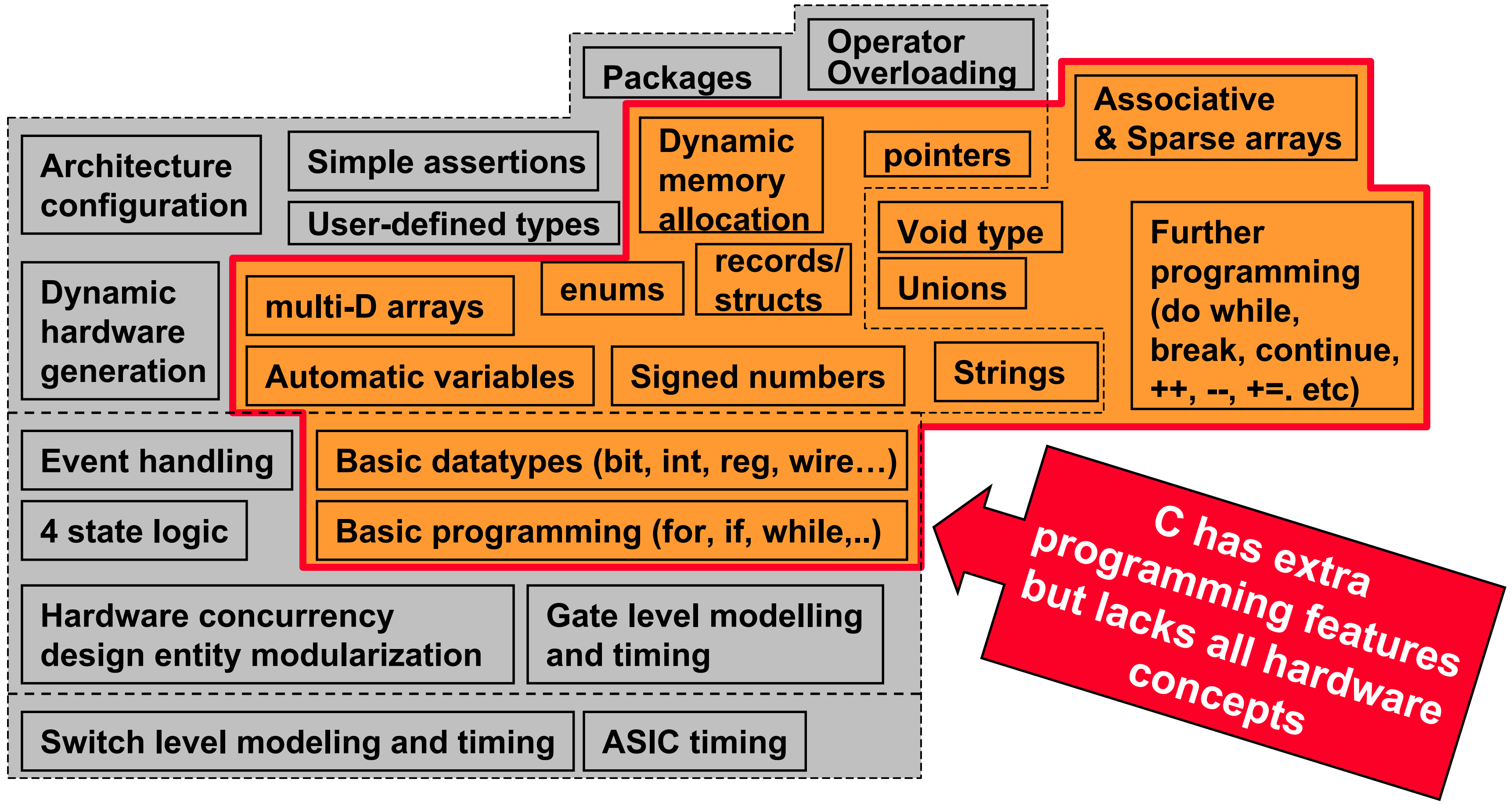
**Verilog-95: Single language for design & testbench**

# SystemVerilog: VHDL


Operator Overloading
Packages
Architecture configuration
Simple assertions
User-defined types
Dynamic memory allocation
pointers
Dynamic hardware generation
multi-D arrays
enums
records/ structs
Automatic variables
Signed numbers
Strings
Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modelling and timing
Switch level modeling and timing
ASIC timing

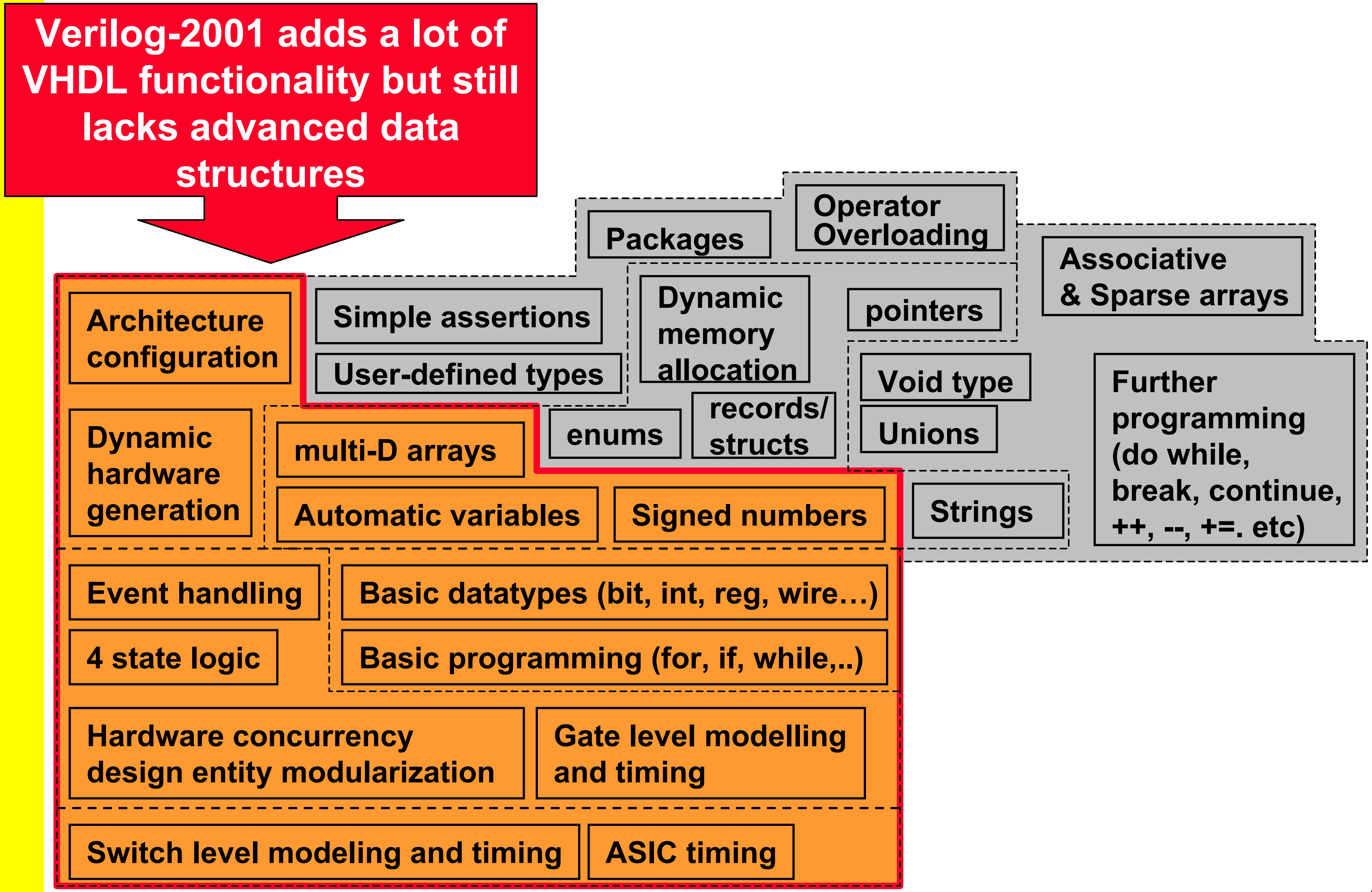
VHDL adds higher level data types and management functionality

# Semantic Concepts: C

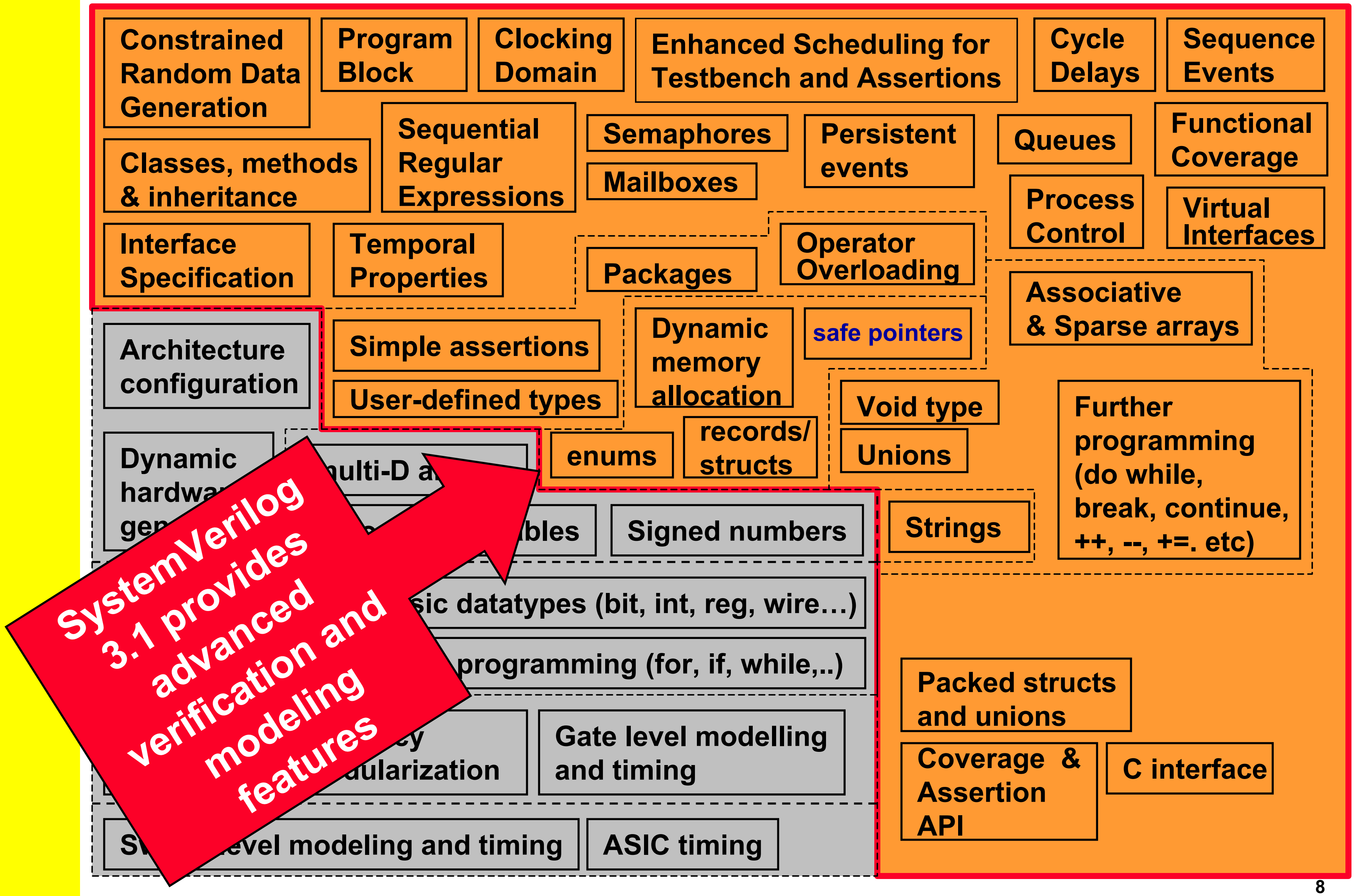


Packages
Operator Overloading
Associative & Sparse arrays
Architecture configuration
Simple assertions
User-defined types
Dynamic memory allocation
pointers
Void type
Unions
Further programming (do while, break, continue, ++, --, +=. etc)
Dynamic hardware generation
multi-D arrays
enums
records/ structs
Automatic variables
Signed numbers
Strings
Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modelling and timing
Switch level modeling and timing
ASIC timing
C has extra programming features but lacks all hardware concepts

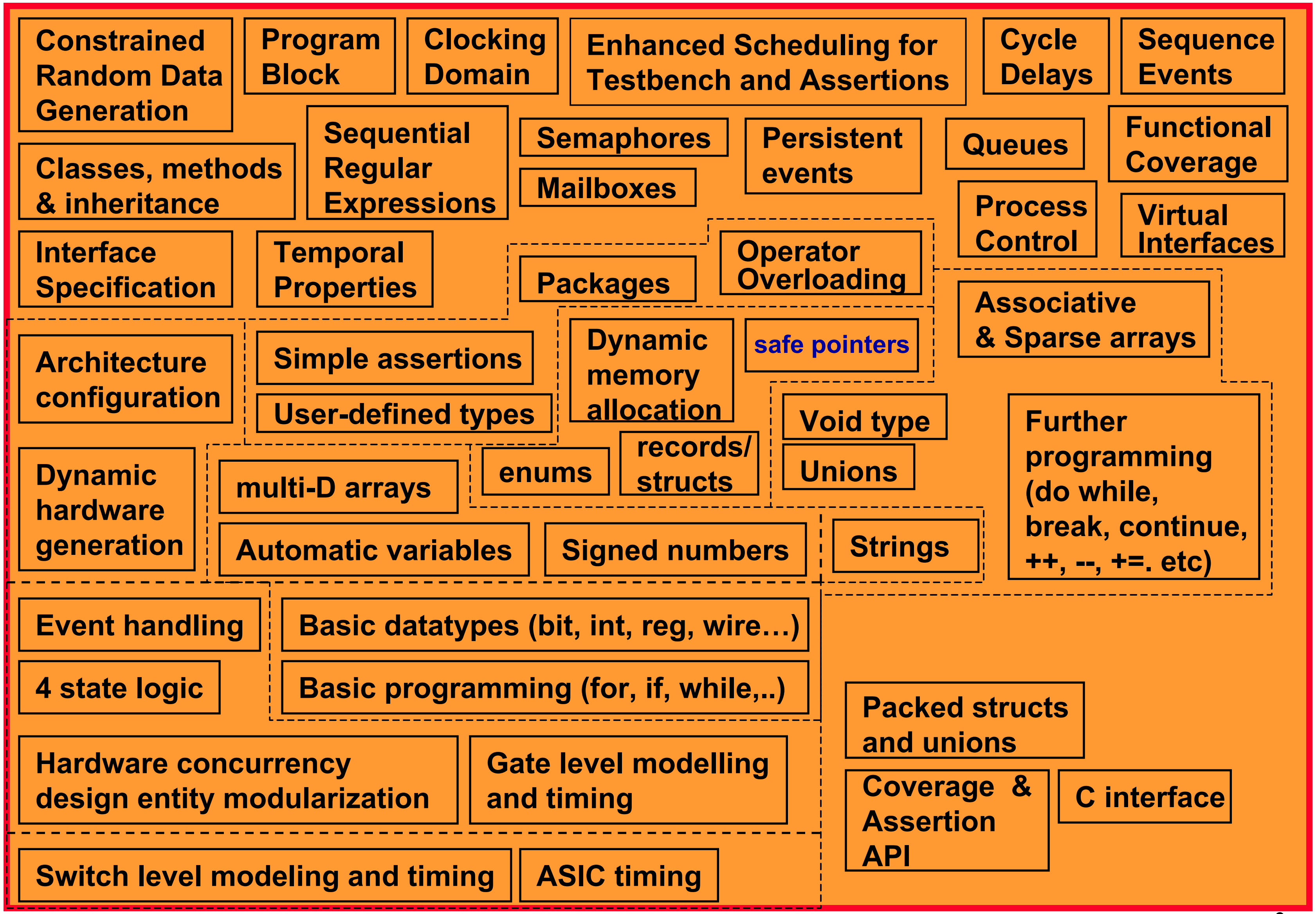
# SystemVerilog: Verilog-2001


Verilog-2001 adds a lot of VHDL functionality but still lacks advanced data structures
Packages
Operator Overloading
Associative & Sparse arrays
Architecture configuration
Simple assertions
User-defined types
Dynamic memory allocation
pointers
Void type
Unions
Further programming (do while, break, continue, ++, --, +=. etc)
Dynamic hardware generation
multi-D arrays
enums
records/ structs
Automatic variables
Signed numbers
Strings
Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modelling and timing
Switch level modeling and timing
ASIC timing

# SystemVerilog: Enhancements

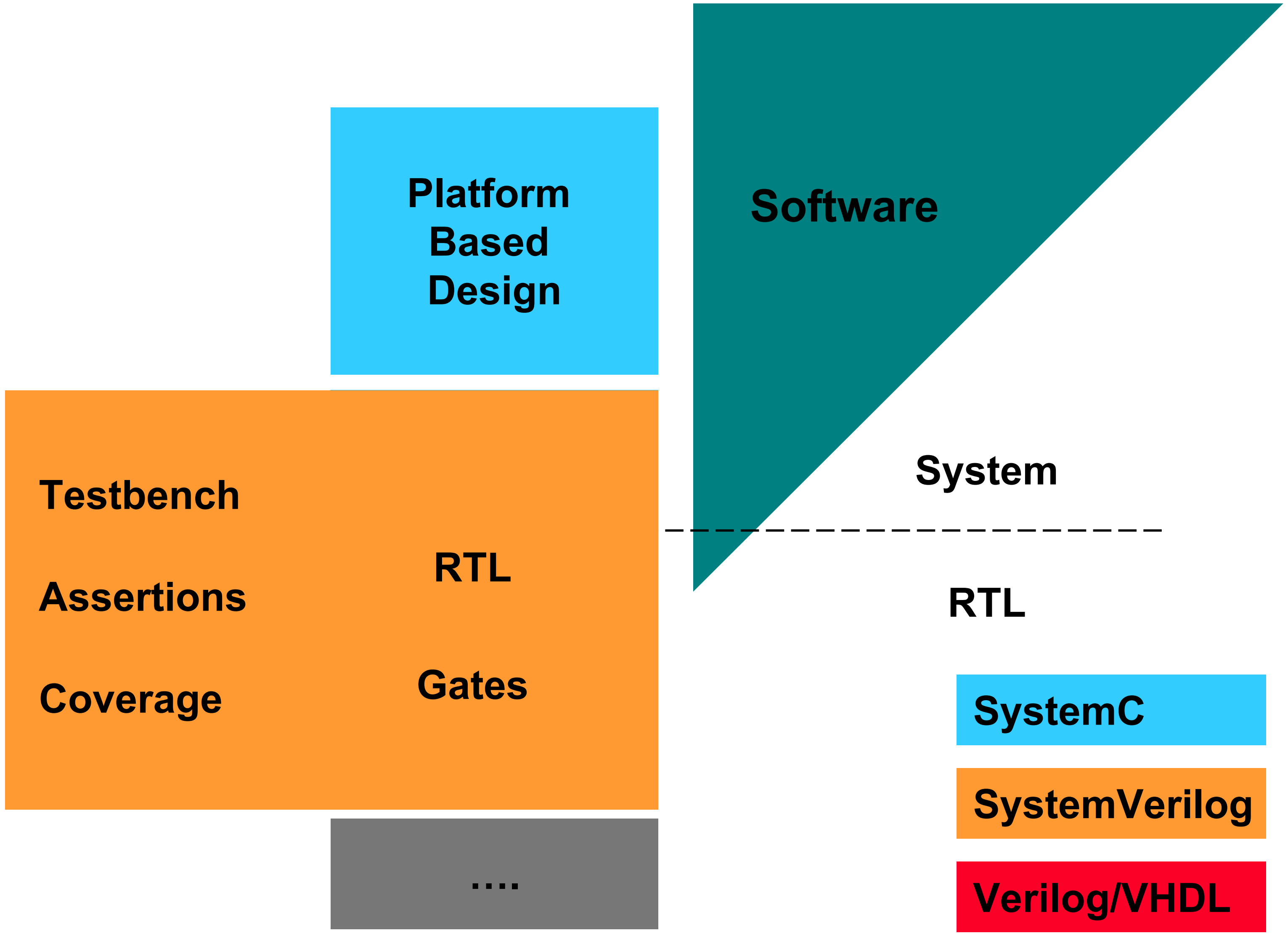

Constrained Random Data Generation
Program Block
Clocking Domain
Enhanced Scheduling for Testbench and Assertions
Cycle Delays
Sequence Events
Sequential Regular Expressions
Semaphores
Persistent events
Queues
Functional Coverage
Classes, methods & inheritance
Mailboxes
Process Control
Virtual Interfaces
Interface Specification
Temporal Properties
Packages
Operator Overloading
Associative & Sparse arrays
Architecture configuration
Simple assertions
Dynamic memory allocation
safe pointers
User-defined types
Void type
Further programming (do while, break, continue, ++, --, +=. etc)
Dynamic hardwa
ge
ulti-D a
enums
records/ structs
Unions
bles
Signed numbers
Strings
sic datatypes (bit, int, reg, wire…)
programming (for, if, while,..)
Packed structs and unions
cy
dularization
Gate level modelling and timing
Coverage & Assertion API
C interface
S
evel modeling and timing
ASIC timing
SystemVerilog 3.1 provides advanced verification and modeling features

# SystemVerilog: Unified Language



Constrained Random Data Generation
Program Block
Clocking Domain
Enhanced Scheduling for Testbench and Assertions
Cycle Delays
Sequence Events
Classes, methods & inheritance
Sequential Regular Expressions
Semaphores
Mailboxes
Persistent events
Queues
Functional Coverage
Process Control
Virtual Interfaces
Interface Specification
Temporal Properties
Packages
Operator Overloading
Associative & Sparse arrays
Architecture configuration
Simple assertions
User-defined types
Dynamic memory allocation
safe pointers
Void type
Unions
Further programming (do while, break, continue, ++, --, +=. etc)
Dynamic hardware generation
multi-D arrays
enums
records/ structs
Automatic variables
Signed numbers
Strings
Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Packed structs and unions
Hardware concurrency design entity modularization
Gate level modelling and timing
Coverage & Assertion API
C interface
Switch level modeling and timing
ASIC timing

# SystemC & SystemVerilog



## Agenda

- Introduction
- SystemVerilog Design Features
- SystemVerilog Assertions
- SystemVerilog Verification Features
- Using SystemVerilog and VHDL Together

# Basic SystemVerilog Data Types

```
reg r;      // 4-state Verilog-2001 single-bit datatype
integer i; // 4-state Verilog-2001 >= 32-bit datatype
bit b;      // single bit 0 or 1
logic w;    // 4-valued logic, x 0 1 or z as in Verilog
byte b8;    // 8 bit signed integer
int i;      // 2-state, 32-bit signed integer
```

**Explicit 2-state Variables Allow Compiler Optimizations to Improve Performance**

**The unresolved type “logic” in SystemVerilog is equivalent to “std_ulogic” in VHDL**

# Familiar C Features In SystemVerilog

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

**continue** starts next loop iteration

**break** exits the loop (= “exit” in VHDL

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;
```

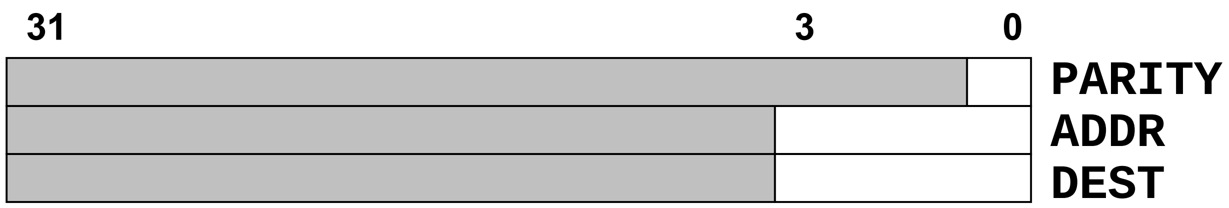
# SystemVerilog Struct = Record

```
type PKT_T is record
  PARITY: std_ulogic;
  ADDR: std_ulogic_vector(3 downto 0);
  DEST: std_ulogic_vector(3 downto 0);
end record;
```

```
typedef struct {
  logic      PARITY;
  logic[3:0] ADDR;
  logic[3:0] DEST;
} pkt_t;
```

```
signal MYPKT : PKT_T;
...
MYPKT.ADDR <= “1100”;
```

```
pkt_t mypkt;
...
mypkt.ADDR = 12;
```



The mypkt structure/ record is “unpacked”

In SystemVerilog, struct variables can also be declared directly, without the typedef

```
struct {
  bit [7:0]  opcode;
  bit [23:0] addr;
  } IR;
```

no typedef

Structure definitions are just like in C but without the optional structure tag before the ‘{‘

# Packed Structures and Unions

```
typedef struct packed {
   logic [15:0] source_port;
   logic [15:0] dest_port;
   logic [31:0] sequence;
} tcp_t;
typedef struct packed {
   logic [15:0] source_port;
   logic [15:0] dest_port;
   logic [15:0] length;
   logic [15:0] checksum
} udp_t;
```

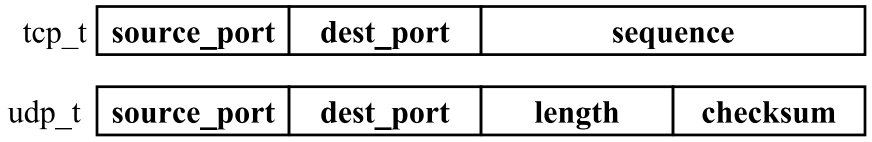
```
typedef union packed {
   tcp_t tcp_h;
   udp_t udp_h;
   bit [63:0] bits;
   bit [7:0][7:0] bytes;
} ip_t;
```

All members of a union must be the same size

```
ip_t ip_h;
logic parity;

ip_h.udp_h.length = 5;
ip_h.bits[31:16] = 5;
ip_h.bytes[3:2] = 5;

parity = ^ip_h;
```

Equivalent

Operate on entire structure as a whole



Create multiple layouts for accessing data

VHDL records not explicitly packed

# Type Conversion

```
typedef struct {
   logic       PARITY;
   logic[3:0] ADDR;
   logic[3:0] DEST;
} pkt_t;

typedef bit[8:0] vec_t;

pkt_t mypkt;
vec_t myvec;

myvec = vec_t’(mypkt);
mypkt = pkt_t’(myvec);
```

Unpacked Structure

User-defined type:
packed bit vector

Cast mypkt as type vec_t

Cast myvec as type pkt_t

**User-defined types and explicit casting improve readability and modularity**

**Similar to Qualified Expressions or conversion functions in VHDL**

## Data Organization - Enum

### VHDL:

```
type FSM_ST is
    {IDLE,
     INIT,
     DECODE,
     …};

signal pstate, nstate : FSM_ST;

case (pstate) is
  when idle:
    if (sync = ‘1’) then
          nstate <= init;
    end if;
  when init:
    if (rdy = ‘1’) then
          nstate = decode;
    end if;
…
end case;
```

### SystemVerilog:

```
typedef enum logic [2:0]
    {idle = 0,
     init = 3,
     decode, // = 4
     …} fsmstate;

fsmstate pstate, nstate;

case (pstate)
  idle: if (sync)
          nstate = init;
  init: if (rdy)
          nstate = decode;
…
endcase
```

**VHDL enums not explicitly typed**

# Packed and Unpacked Arrays

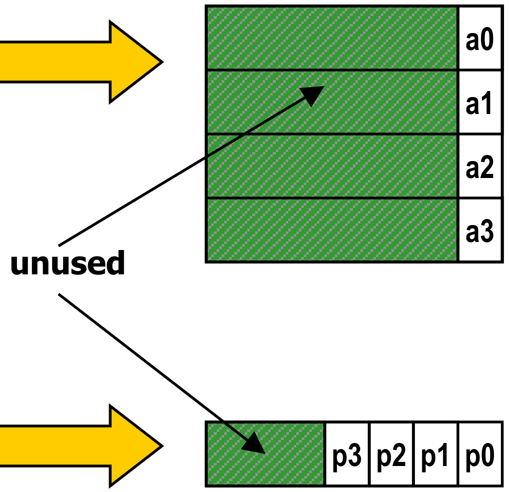
**unpacked array of bits**

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

**Don't get them mixed up**

**packed array of bits**

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



**1k 16 bit unpacked memory**

**1k 16 bit packed memory**

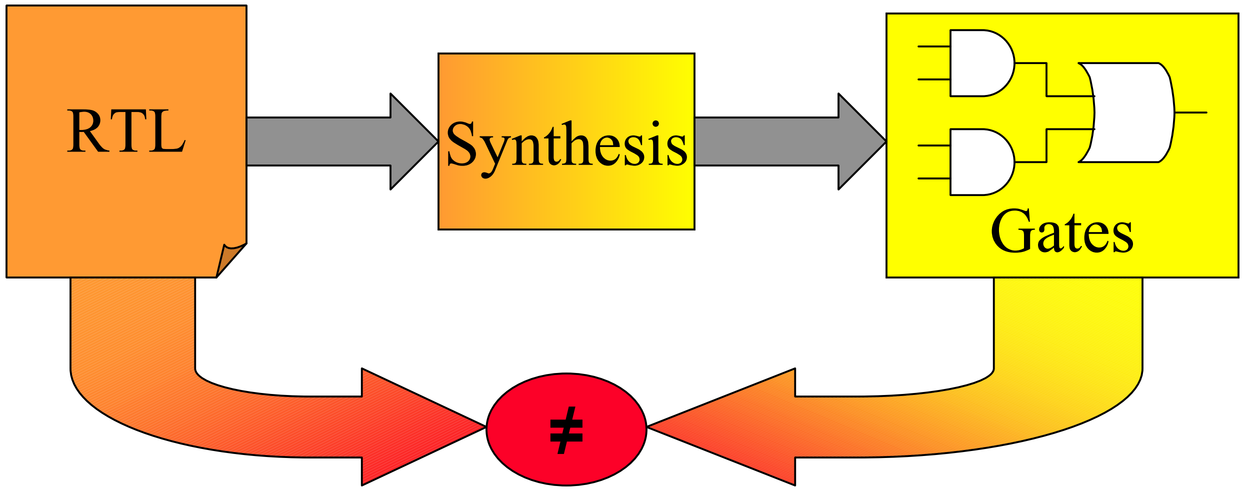
```
bit [15:0] memory [1023:0];
memory[i] = ~memory[i];
memory[i][15:8] = 0;

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

**Packed indexes can be sliced**

**Can operate on entire memory**

**SystemVerilog also includes the VHDL-like array attribute functions: $left, $right, $low, $high, $increment, $length and $dimensions**

## Pre-Post Synthesis Mismatches



- **Causes**
  - **Sensitivity list mismatches**
  - **Pragmas affect synthesis but not simulation**
- **SystemVerilog Solves these problems**
  - **Specialized `always` blocks automate sensitivity**
  - **Synthesis directives built into the language**

## Design Intent – always_{comb,latch,ff}

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

- always_comb, always_latch and always_ff are explicit
- Compiler Now Knows User Intent and can flag errors accordingly

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



//Compiler now asks
//“Where’s the else?”
always_comb
  if (b) c = a;
//Intent: Conditional
//         Assignment
always_latch
  if (clk)
   if (en) Q <= d;
//Conversely unconditionally
//assigned -is it a latch?
always_latch
        q <=d
```

## always_comb Sensitivity

- always_comb eliminates sensitivity list issues
  - Ensures synthesis-compatible sensitivity
  - Helps reduce “spaghetti code”
- Consider that always_comb derives sensitivity from
  - RHS/expr in process
  - RHS/expr of statements in Function Calls

```
logic avar,a,b,c,d,e;
     logic [1:0] sel;
always_comb begin
    a = b;
    StepA();
end
function StepA
  case (sel)
    2’b01: avar = a | c;
    2’b10: avar = d & e;
    default: avar = c;
  endcase
endfunction
```

```
always @(sel,b,c,d,e) begin
  a = b;
  case (sel)
    …
  endcase
end
```

Encapsulate blocks of combinational logic into functions – Easier to read and debug

# Design Intent – Unique/Priority

- `Parallel_case/full_case` pragmas affect *synthesis* behavior but not *simulation* behavior
- `Unique` keyword means that one and only one branch will be taken (same as `full_case parallel_case`)
- `Priority` keyword means that the first branch will be taken (same as `full_case`)
- Will cause simulation run-time error if illegal value is seen

**`sel` must be "one-hot"**

**No `default` clause needed**

```
unique case (sel)
  sel[0] : muxo = a;
  sel[1] : muxo = b;
  sel[2] : muxo = c;
endcase
```

```
unique if (sel==3'b001)
  muxo = a;
else if (sel == 3'b010)
  muxo = b;
else if (sel == 3'b100)
  muxo = c;
```

**No ending `else` needed**

**`irq0` has highest priority**

```
priority case (1'b1)
 irq0: irq = 4'b1 << 0;
 irq1: irq = 4'b1 << 1;
 irq2: irq = 4'b1 << 2;
 irq3: irq = 4'b1 << 3;
endcase
```

```
priority if (irq0)
  irq = 4'b1;
else if (irq1)
  irq = 4'b2;
else if (irq2)
  irq = 4'b4;
```

# Syntax – Implicit Named Port

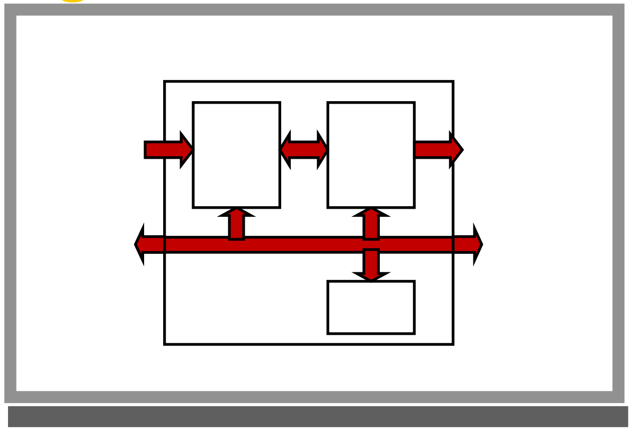
- Creating netlists by hand is tedious
- Generated netlists are unreadable
  - Many signals in instantiations
  - Instantiations cumbersome to manage
- Implicit port connections dramatically improve readability
- Use same signal names up and down hierarchy where possible
- Port Renaming Accentuated
- .name allows explicit connections with less typing (and less chance for error)

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

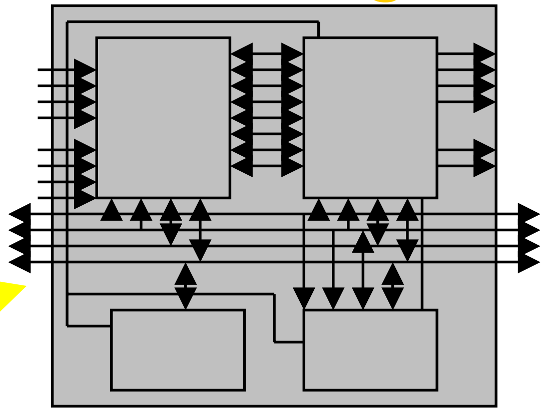
```
module top();
  logic rd,wr;
  tri [31:0] dbus,abus;
  tb tb(.*, .ireset(start),
            .oreset(tbreset));
  dut d1(.*,.reset(tbreset[0]));
  dut d2(.rd, .wr, .dbus, .abus,
         .reset(tbreset[1]));
endmodule
```

# SystemVerilog Interfaces

## Design On A White Board



## HDL Design

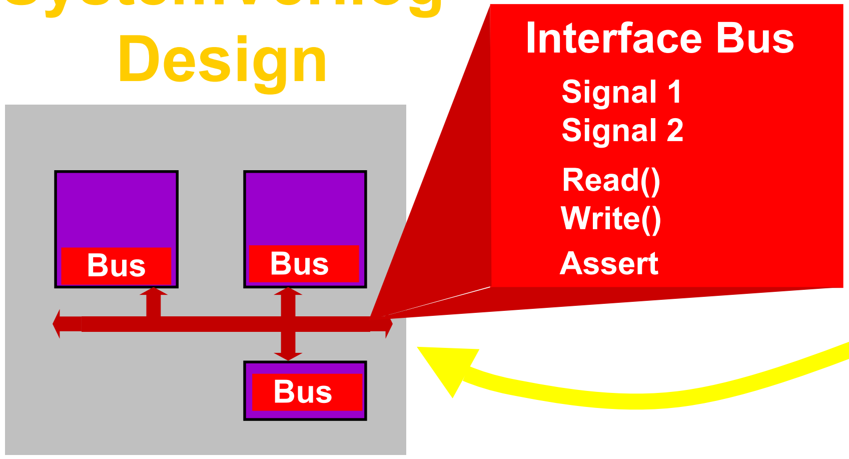


**Complex signals**

Bus protocol repeated in blocks

Hard to add signal through hierarchy

## SystemVerilog Design



**Interface Bus**

Signal 1
Signal 2

Read()
Write()

Assert

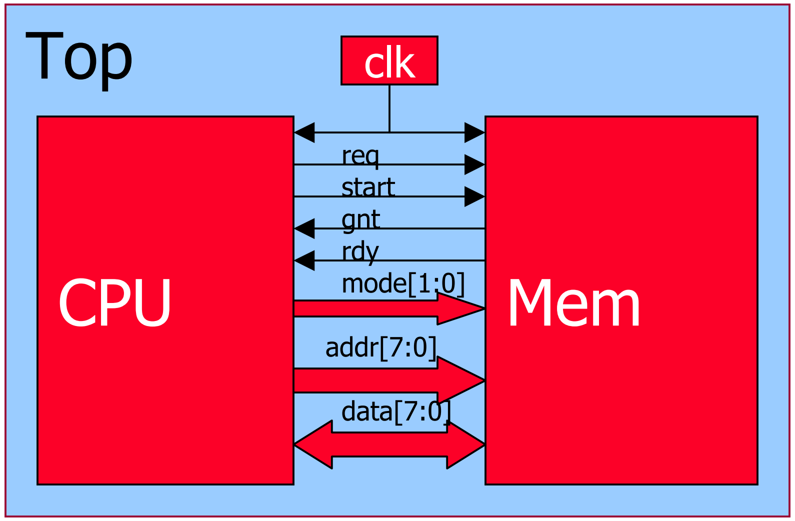
**Communication encapsulated in interface**

- Reduces errors, easier to modify
- Significant code reduction saves time
- Enables efficient transaction modeling
- Allows automated block verification

# Example without Interface

```
entity memMod is
 port(reg,clk,start : in bit;
  mode : in std_logic_vector(1 downto 0);
  addr : in std_logic_vector(7 downto 0);
  data : inout std_logic_vector(7 downto 0);
  gnt, rdy : out bit);
end memMod;

architecture RTL of memMod is
  process (clk) begin
    wait until clk'event and clk='1';
      gnt <= req AND avail;
end architecture RTL;

entity cpuMod is
 port(clk, gnt, rdy : in bit;
  data : inout std_logic_vector(7 downto 0);
  req, start : out bit;
  mode : out std_logic_vector(1 downto 0);
  addr : out std_logic_vector(7 downto 0));
end cpuMod;

architecture RTL of cpuMod is
...
end architecture RTL;
```

```
architecture netlist of top is
  signal req,gnt,start,rdy : bit;
  signal clk : bit := '0';
  signal mode :
    std_logic_vector(1 downto 0);
  signal addr, data :
    std_logic_vector(7 downto 0);

mem: memMod port map
          (req,clk,start,mode,
           addr,data,gnt,rdy);
cpu: cpuMod port map
          (clk,gnt,rdy,data,
           req,start,mode,addr);
end architecture netlist;
```



# 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
```

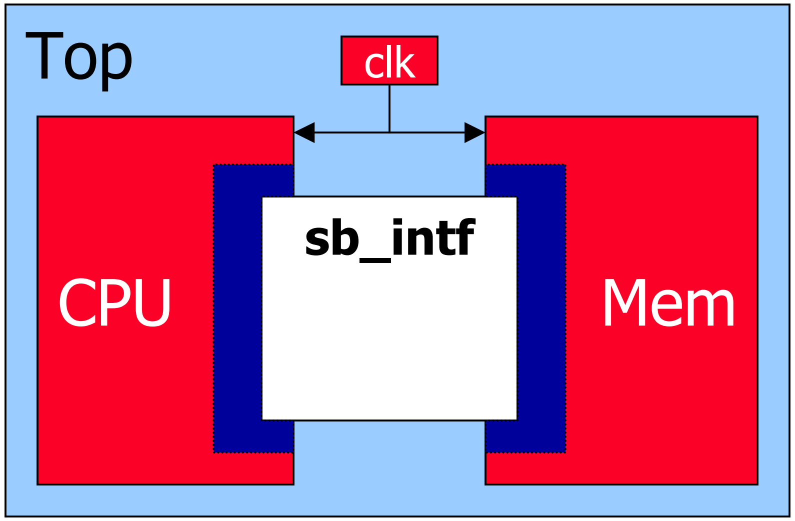
**Refer to intf signals**

```
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**

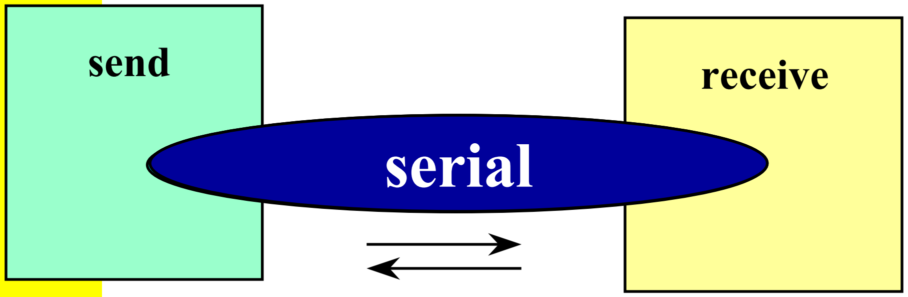
**Connect interface**



# Using Different Interfaces

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

parallel channel(clk);
send     s(clk, channel);
receive r(clk, channel);
```

```
typedef logic [31:0]
data_type;

bit clk;
always #100 clk = !clk;

serial channel(clk);
send     s(clk, channel);
receive r(clk, channel);
```



```
module send(input bit clk,
            interface ch);
   data_type d;
...
   ch.write(d);
endmodule
```

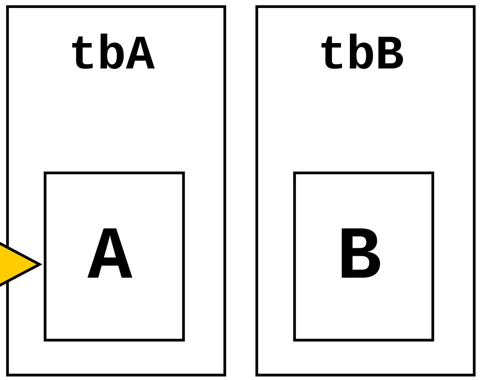
**Module inherits communication method from interface**

**Simplifies design exploration**
**Extends block-based design to the communication between blocks**

# Conventional Verification

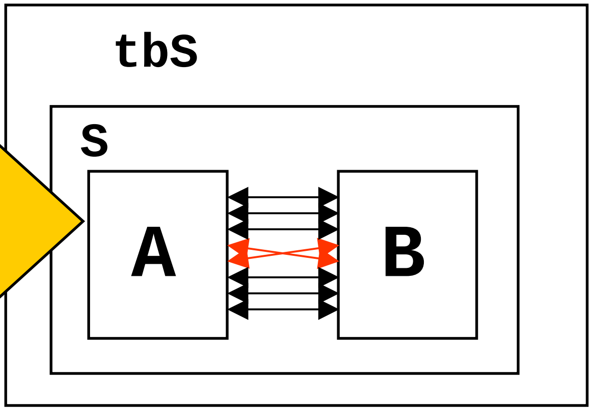
- Pre-Integration

Test Subblocks in isolation



- Testbench reuse problems
- tbA and tbB separate

Only need to test interconnect structure. (missing wires, twisted busses)

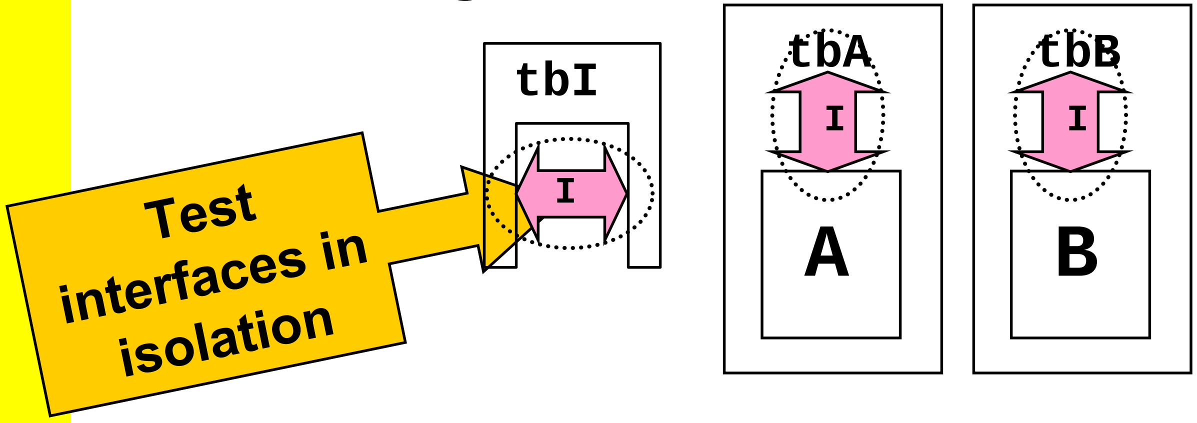


- Complex interconnect
- Hard to create tests to check all signals
- Slow, runs whole design even if only structure is tested

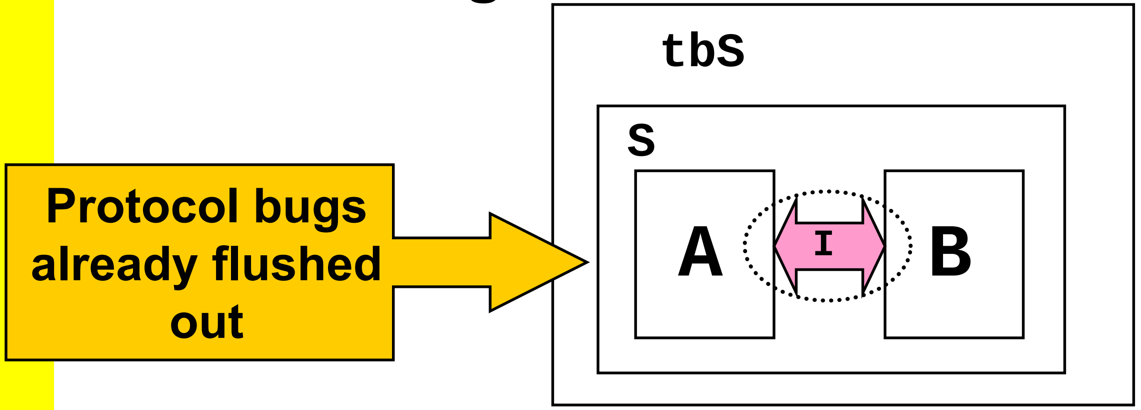
- Post-Integration

# SystemVerilog Verification

- Pre-Integration



- Post-Integration



- Interfaces provide reusable components
- tbA and tbB are 'linked'
- Interface is executable spec
- Wiring up is simple and not error prone
- Interfaces can contain protocol checkers and coverage counters
- Start Chip-Level Verification at the Block Level

## Operator Overloading

- **Enable use of simple operators with Complex SV Types**

  **struct3 = struct1 + struct2**

- **Operator Overloading is allowed for type combinations not already defined by SV Syntax**

```
bind overload_operator function data_type
  function_identifier (overload_proto_formals)
```

```
typedef struct {
  bit sign;
  bit [3:0] exponent;
  bit [10:0] mantissa;
} float;

bind + function float faddfr(float, real);
bind + function float faddff(float, float);

float A, B, C, D;

assign A = B + C; //equivalent to A = faddff(B, C);
assign D = A + 1.0; //equivalent to A = faddfr(A, 1.0);
```

## Packages and Separate Compilation

- **Allows sharing of:**
  - **nets, variables, types,**
  - **tasks, functions**
  - **classes, extern constraints, extern methods**
  - **parameters, localparams, spec params**
  - **properties, sequences**
- **Allows unambiguous references to shared declarations**
- **Built-in functions and types included in `std` package**
- **Groups of files can now be compiled separately**

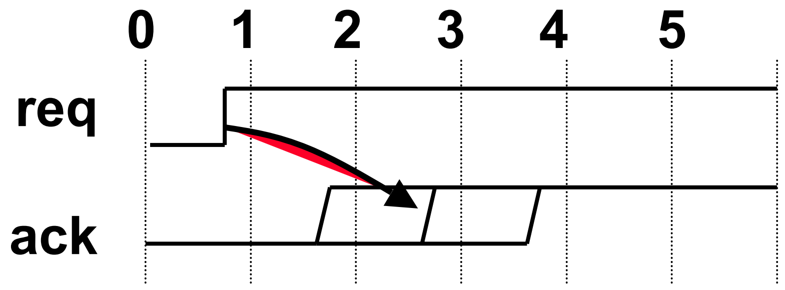
```
package ComplexPkg;
  typedef struct {
    float i, r;
    } Complex;
  function Complex add(Complex a, b)
    add.r = a.r + b.r;
    add.i = a.i + b.i;
  endfunction
  function Complex mul(Complex a, b)
    mul.r = (a.r * b.r) + (a.i * b.i);
    mul.i = (a.r * b.i) + (a.i * b.r);
  endfunction
endpackage : ComplexPkg


module foo (input bit clk);
  import ComplexPkg::*
  Complex a,b;
  always @(posedge clk)
    c = add(a,b);
endmodule
```

## Agenda

- Introduction
- SystemVerilog Design Features
- SystemVerilog Assertions
- SystemVerilog Verification Features
- Using SystemVerilog and VHDL Together

# What is an Assertion?

A concise description of [un]desired behavior



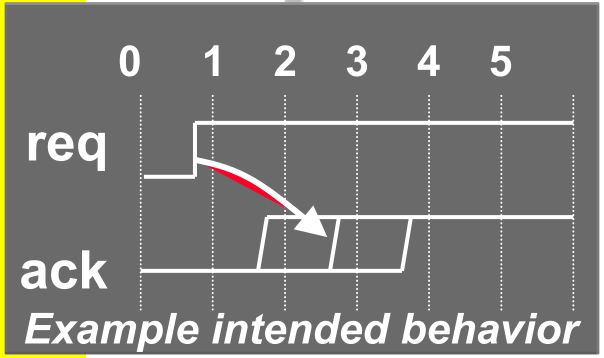
*Example intended behavior*

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

# Concise and Expressive SVA

SVA Assertion

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



*Example intended behavior*

VHDL

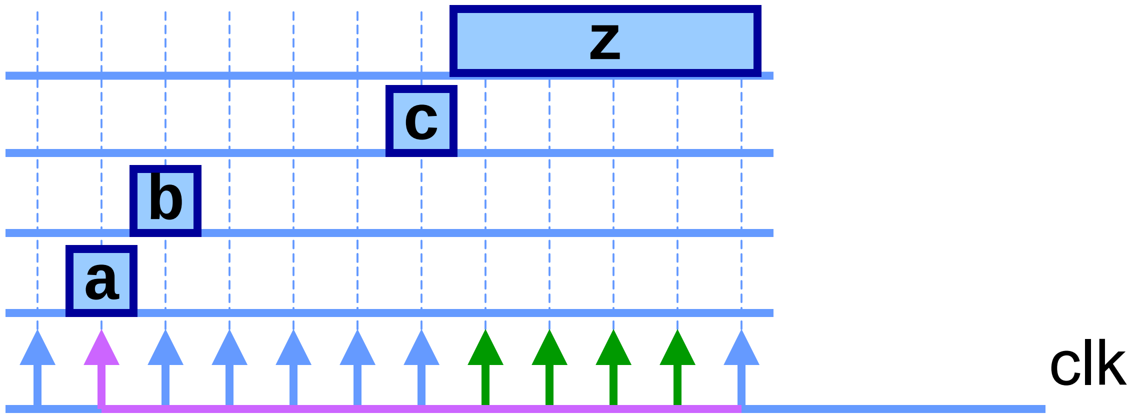
```
sample_inputs : process (clk)
begin
  if rising_edge(clk) then
    STROBE_REQ <= REQ;
    STROBE_ACK <= ACK;
  end if;
end process;
protocol: process
  variable CYCLE_CNT : Natural;
begin
  loop
    wait until rising_edge(CLK);
    exit when (STROBE_REQ = '0') and (REQ = '1');
  end loop;
  CYCLE_CNT := 0;
  loop
    wait until rising_edge(CLK);
    CYCLE_CNT := CYCLE_CNT + 1;
    exit when ((STROBE_ACK = '0') and (ACK = '1')) or (CYCLE_CNT = 3);
  end loop;
  if ((STROBE_ACK = '0') and (ACK = '1')) then
    report "Assertion success" severity Note;
  else
    report "Assertion failure" severity Error;
  end if;
end process protocol;
```

HDL Assertion

# Sequential Regular Expressions

- Describing a sequence of events
- Sequences of Boolean expressions can be described with a specified time step in-between

```
@(posedge clk) a ##1 b ##4 c ##[1:5] z;
```



## Property Definition

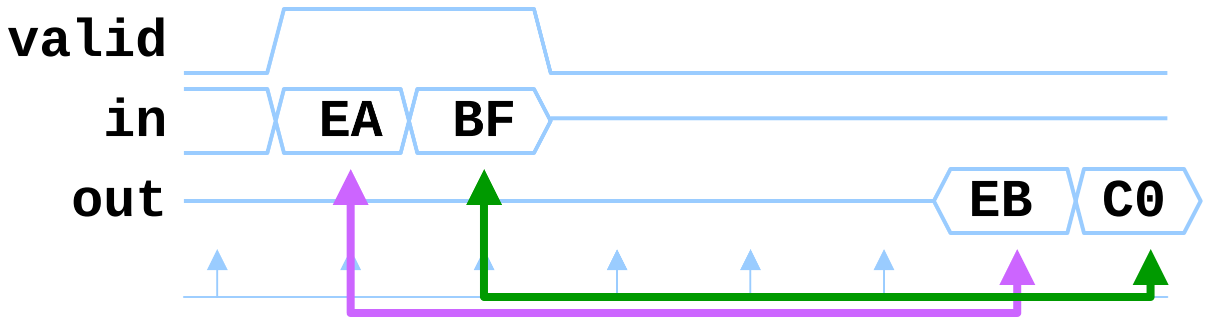
- **Property Declaration:** `property`
  - **Declares property by name**
  - **Formal parameters to enable property reuse**
  - **Top Level Operators**
    - `not` ***desired/undesired***
    - `disable iff` ***reset***
    - `|->, |=>` ***precondition***
- **Assertion Directives**
  - `assert` **– checks that the property is never violated**
  - `cover` **– tracks all occurrences of property**

```
property prop1(a,b,c,d);
  disable iff (reset)
            (a) |-> [not](b ##[2:3]c ##1 d);
endproperty
assert1: assert prop1 (g1, h2, hxl, in3);
```

# Manipulating Data: Local Dynamic Variables

- Declared Locally within Sequence/Property
  - New copy of variable for each sequence invocation
- Assigned anywhere in the sequence
- Value of assigned variable remains stable until reassigned in a sequence

Local Dynamic Variable Example



```
property e;
int x;
(valid,(x=in))|=> ##5(out==(x+1));
endproperty
```

# Embedding Concurrent Assertions

```
property s1;
    (req && !gnt)[*0:5] ##1 gnt && req ##1 !req ;
endproperty

always @(posedge clk or negedge reset)
  if(reset == 0) do_reset;
  else if (mode == 1)
    case(st)
    REQ: if (!arb)
              if (foo)
                 st <= REQ2;
                 PA: assert property (s1);
```

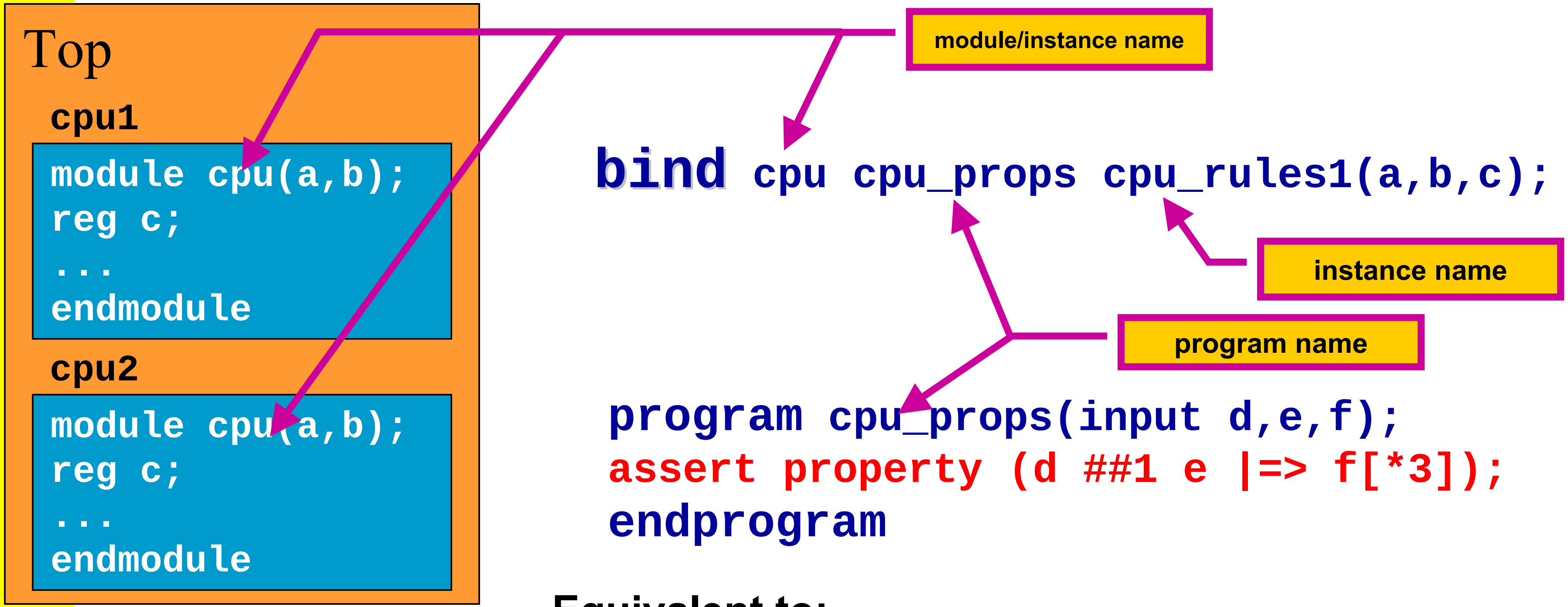
- Automatically Updates Enabling Condition as Design Changes
- Infers clock from instantiation

```
property p1;
  @(posedge clk) ((reset == 1) && (mode == 1)
                          && (st == REQ) && (!arb) && (foo)) => s1;
endproperty

DA: assert property (p1);
```

- Requires User to Update Manually as Design Changes

# Bind statement

```
bind module_or_instance_name instantiation;
```



```
cpu1
module cpu(a,b);
reg c;
...
endmodule
cpu2
module cpu(a,b);
reg c;
...
endmodule
```

```
bind cpu cpu_props cpu_rules1(a,b,c);
```

```
program cpu_props(input d,e,f);
assert property (d ##1 e |=> f[*3]);
endprogram
```

**Equivalent to:**

```
assert property (top.cpu1.a ##1 top.cpu1.b |=> top.cpu1.c[*3]);
assert property (top.cpu2.a ##1 top.cpu2.b |=> top.cpu2.c[*3]);
```

or

```
cpu_props cpu_rules1(a,b,c); // in module cpu
```

**Bind assertions to VHDL code**

## Flexible Assertions Use-Model

- Design Engineers
  - Able to define assertions in-line with design code
  - Assertions typically cover implementation-level detail
  - Capture assumptions while they're fresh in the designer's mind
- Verification Engineers
  - Able to define assertions external to RTL code and "bind" them to the design
  - Assertions typically cover interface/system behavior
  - Do not need to modify the golden RTL to add assertions

## Agenda

- Introduction
- SystemVerilog Design Features
- SystemVerilog Assertions
- **SystemVerilog Verification Features**
- Using SystemVerilog and VHDL Together

# Dynamic Arrays

## Declaration syntax

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

## Initialization syntax

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

**Equivalent to:**

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

## Size method

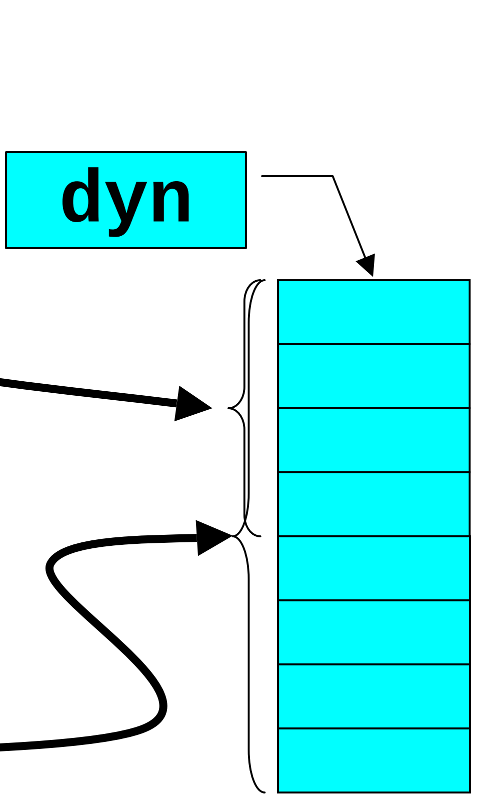
```
function int size();
int j = dyn.size;//j=4
```

## Resize syntax

```
dyn = new[j * 2];
```

**Equivalent to:**

```
bit[3:0] dyn[0:7];
```



# Associative Arrays

- 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
```

## Example

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

## Built-in Methods

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

**Ideal for Dealing with Sparse Data**

## Queues

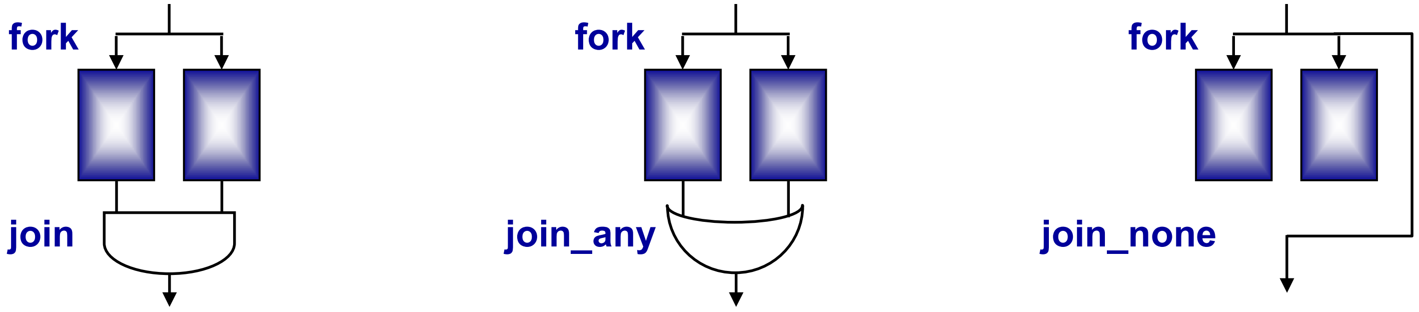
- **Variable-sized Array: data_type name [$]**
  - **Uses array syntax and operators**

```
int q[$] = { 2, 4, 8 };  int e, pos, p[$];
e = q[0];    // read the first (leftmost) item
e = q[$];    // read the last (rightmost) item
q = { q, 6 };   // append: insert ‘6’ at the end
q = { e, q };   // insert ‘e’ at the beginning
q = q[1:$];  // delete the first (leftmost) item
q = q[1:$-1];   // delete the first and last items
```

- **Synthesizable if maximum size is known**
  - **`q[$:25]`**  **// maximum size is 25**

# Dynamic Processes and Threads

- **SystemVerilog adds dynamic parallel processes using `fork/join_any` and `fork/join_none`**



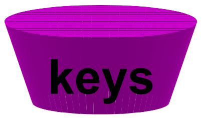
- **Threads created via fork…join**
- **Threads execute until a blocking statement**
  - **wait for: (event, mailbox, semaphore, variable, etc.)**
  - **disable fork to terminate child processes**
  - **wait_child to wait until child processes complete**
- **Built-in process object for fine-grain control**

**Multiple Independent Threads Maximize Stimulus Interactions**

## Inter-Process Synchronization

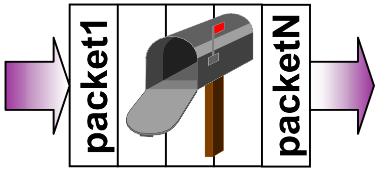
- **Events**
  - Events are variables – can be copied, passed to tasks, etc.
  - event.triggered; // persists throughout timeslice, avoids races
  - wait_order(), wait_any(), wait_all(<events>);
- **Semaphore – Built-in Class**

```
semaphore semID = new(1);
semID.get(1);
semID.put(1);
```



- **Mailbox – Built-in Class**
  - Arbitrary type

```
mailbox #(type) mbID = new(5);
mbID.get(msg);
mbID.put(msg);
```



**Ensures meaningful, race-free communication between processes**

# Class Definition

## Definition syntax

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

```
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
```

**extern** keyword allows for out-of-body method declaration

"`::`" operator links method declaration to Class definition

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

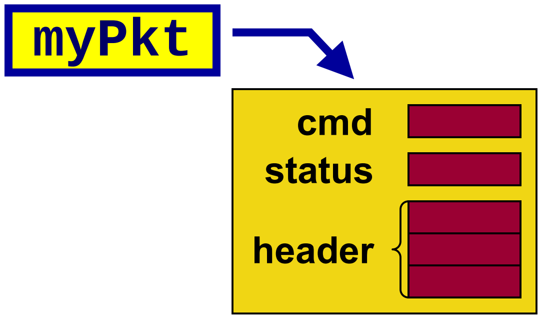
**Note: Class declaration does not allocate any storage**

# Class Instantiation

```
Packet myPkt = new;
```

Declare myPkt handle;
No storage allocated yet

Call to "new" method
allocates storage for object

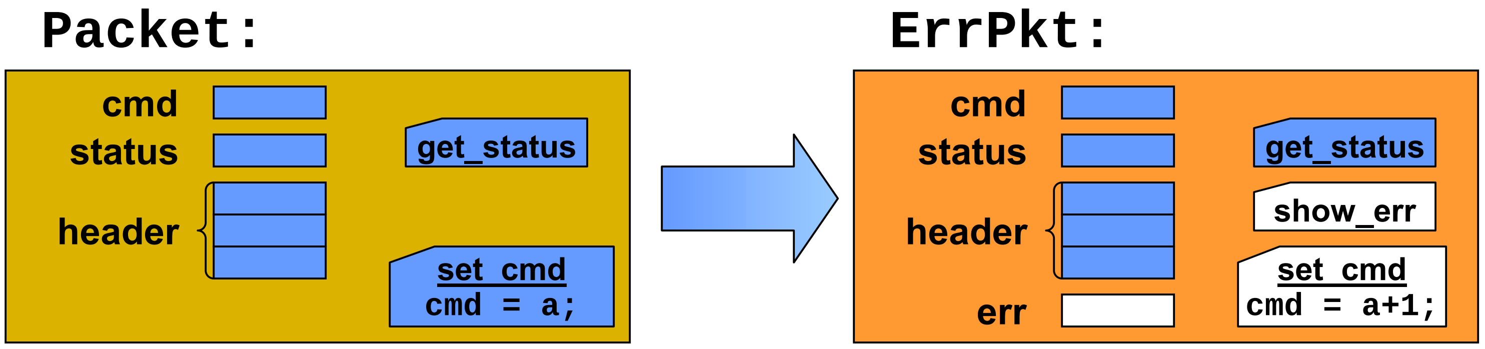


- User may override default "new" method
  - Assign values, call functions, etc.
  - User-defined new method may take arguments
- Garbage Collection happens automatically

# Class Inheritance

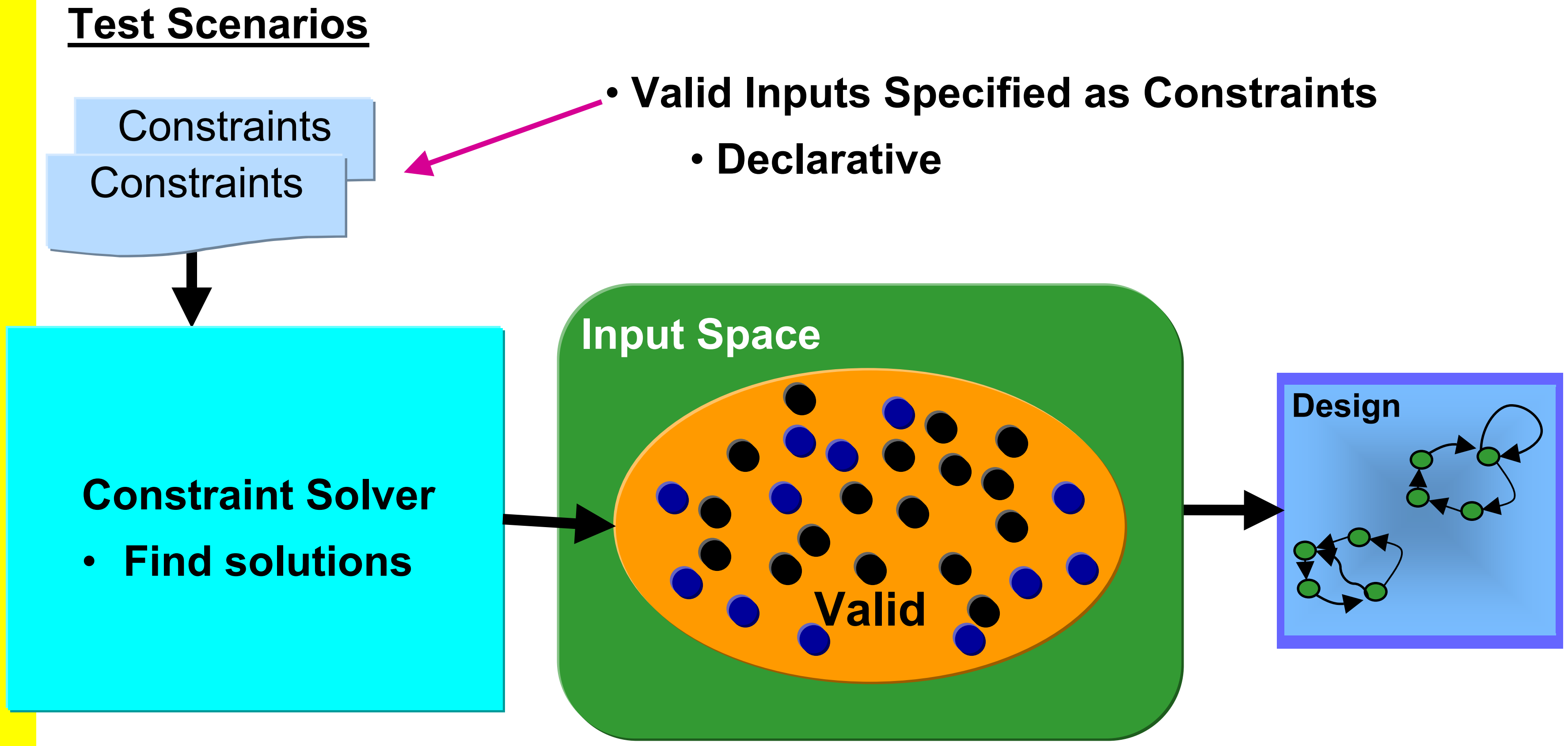
- **Keyword *extends* Denotes Hierarchy of Class Definitions**
  - **Subclass inherits properties, constraints and methods from parent**
  - **Subclass can redefine methods explicitly**

```
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 Base Class**

# Constrained Random Simulation



**Exercise Hard-to-Find Corner Cases While Guaranteeing Valid Stimulus**

## Basic Constraints

- **Constraints are Declarative**

```
    class Bus;
    rand bit[15:0] addr;
    rand bit[31:0] data;
    randc bit[3:0] mode;
    constraint word_align {addr[1:0] == 2'b0;}
    endclass
```

- **Calling `randomize` selects values for all random variables in an object such that all constraints are satisfied**
  - Generate 50 random data and word_aligned addr values

```
Bus bus = new;
repeat (50)
  if ( bus.randomize() == 1 ) // 1=success,0=failure
    $display ("addr = %16h data = %h\n", bus.addr,
    bus.data);
  else
    $display ("Randomization failed.\n");
```

## In-Line Constraints

- **Additional Constraints In-line Via**
  **obj**.randomize()with **<constraint_blk>**

```
task exerBus(MyBus m);
  int r;
  r = m.randomize() with {type==small};
endtask
```

Force **type** to be **small**

- **In-Line Constraints Pick Up Variables From the Object**

## Layered Constraints

- Constraints Inherited via Class Extension
  - Just like data and methods, constraints can be inherited or overridden

```
typedef enum { low, high, other } AddrType ;

class MyBus extends Bus;
  rand AddrType type;
  constraint addr_rang {
    ( type == low  ) => addr in { [  0 :  15] };
    ( type == high ) => addr in { [128 : 255] }; }
endclass
```

type variable selects address range

- `Bus::word_align` Constraint is also active
  - Inheritance allows layered constraints
  - Constraints can be enabled/disabled via `constraint_mode()` method

**Allows Reusable Objects to be Extended and/or Constrained to Perform Specific Functions**

# Weighted Random Case

- Randomly select one statement
  - Label expressions specify distribution weight

```
randcase
    3 : x = 1;     // branch 1
    1 : x = 2;     // branch 2
    a : x = 3;     // branch 3
endcase
```

  - If a == 4:
    - branch 1 taken with 3/8 probability (37.5%)
    - branch 2 taken with 1/8 probability (12.5%)
    - branch 3 taken with 4/8 probability (50.0%)

# Scope Randomization & Constraint Checking

- randomize method can be applied to any variable

```
[std::] randomize ( [ variable_list ] ) [ with { constraint_block } ]
module stim;
  bit[15:0] a;
  bit[31:0] b;

  function bit gen_stim();
    bit success, rd_wr;
    success = randomize( a, b, rd_wr ) with { a > b };
    return rd_wr ;
  endfunction
...
endmodule
```

Optional "::" namespace operator to disambiguate method name

- Constraints can be checked in-line

```
status = class_obj.randomize(null);
```

Passing "null" argument to randomize checks the constraints
0 = valid, 1 = invalid

## Functional Coverage

- New `covergroup` container allows declaration of
  - coverage points
    - variables
    - expressions
    - transitions
  - cross coverage
  - sampling expression : clocking event

```
enum { red, green, blue } color;
bit [3:0] pixel_adr;

covergroup g1 @(posedge clk);
   c: coverpoint color;
   a: coverpoint pixel_adr;
   AxC: cross color, pixel_adr;
endgroup;
```

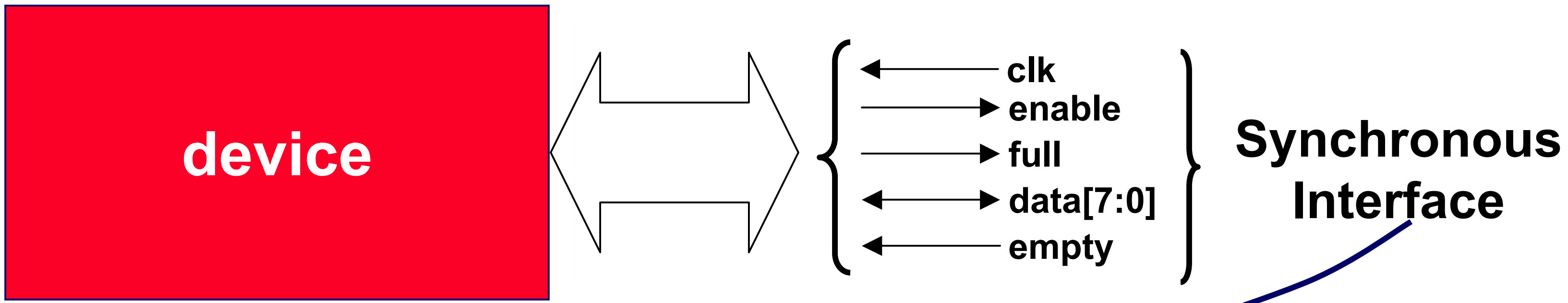
Sample event

3 bins for color

16 bins for pixel

48 (=16 * 3) cross products

# Synchronous Interfaces: Clocking



```
clocking bus @(posedge clk);
  default input #1ns output #2ns;
  input          enable, full;
  inout          data;
  output         empty;
endclocking
```

*Clocking Event "clock"*

*Default I/O skew*

```
initial begin
  tb_en = bus.enable;  // read sampled value of enable
  bus.empty <= 1;  // write "empty" after 2 ns
end
```

## Program Block

- Purpose: Identifies verification code
- A `program` differs from a `module`
  - Only initial blocks allowed
  - Special semantics
    - Executes in *Reactive* region

      design → clocking/assertions → program
  - Program block variables cannot be modified by the design

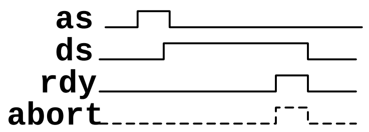
```
   program name (<port_list>);
     <declarations>;// type, func, class, clocking…
     <continuous_assign>
     initial <statement_block>
   endprogram
```

**The Program block functions pretty much like a C program**
**Testbenches are more like software than hardware**

# TB + Assertions Example

A new bus cycle may not start for 2 clock cycles after an abort cycle has completed

```
sequence abort_cycle;
 !rdy throughout (as ##1 ds[*1:$] ##1 abort);
endsequence
```



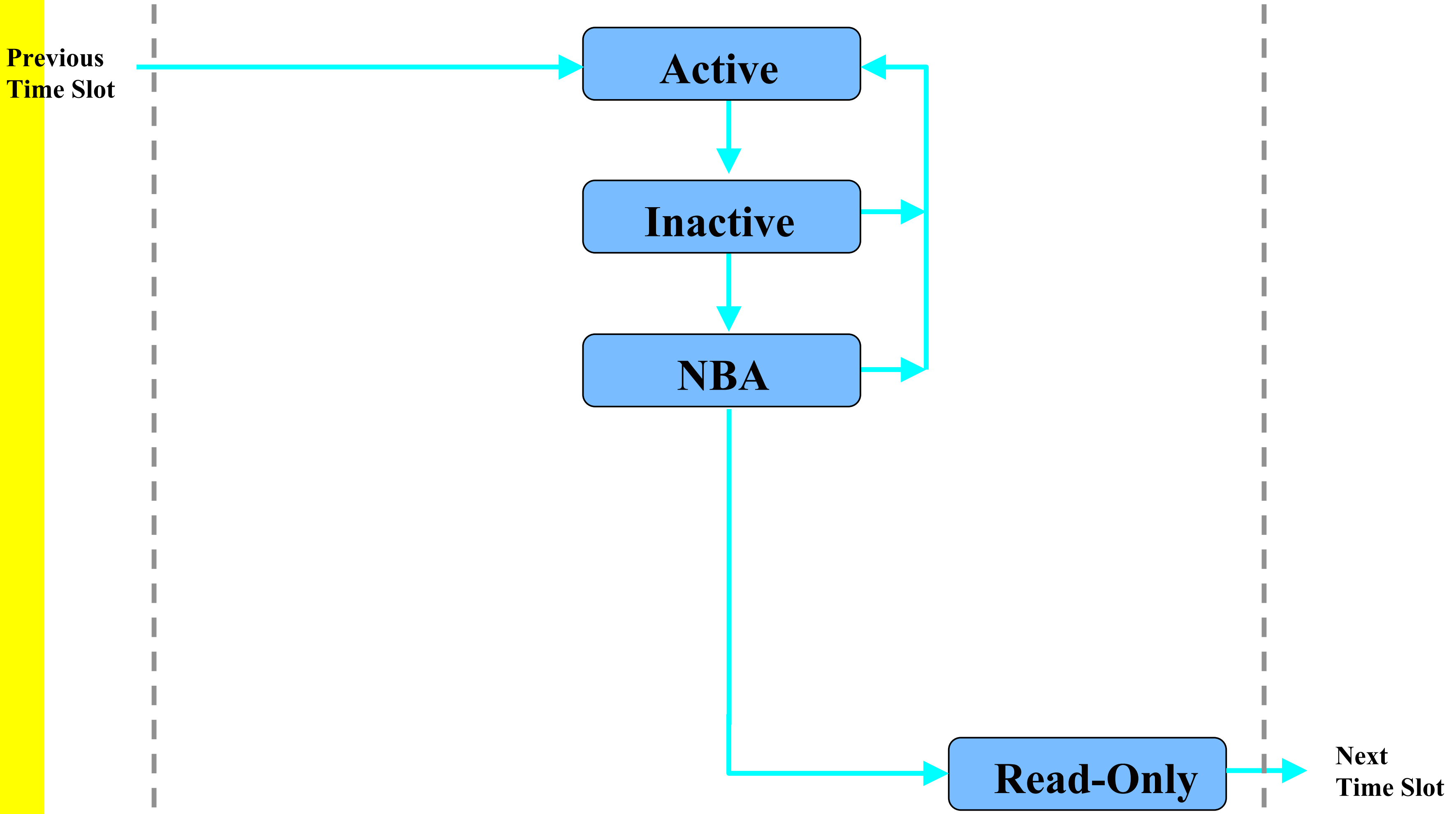
```
cover property (@(posedge clk) abort_cycle)
   wait_cnt = 2;
```

```
property wait_after_abort;
  @(posedge clk) abort_cycle |=> !as[*2];
endproperty
assert property (wait_after_abort);
```

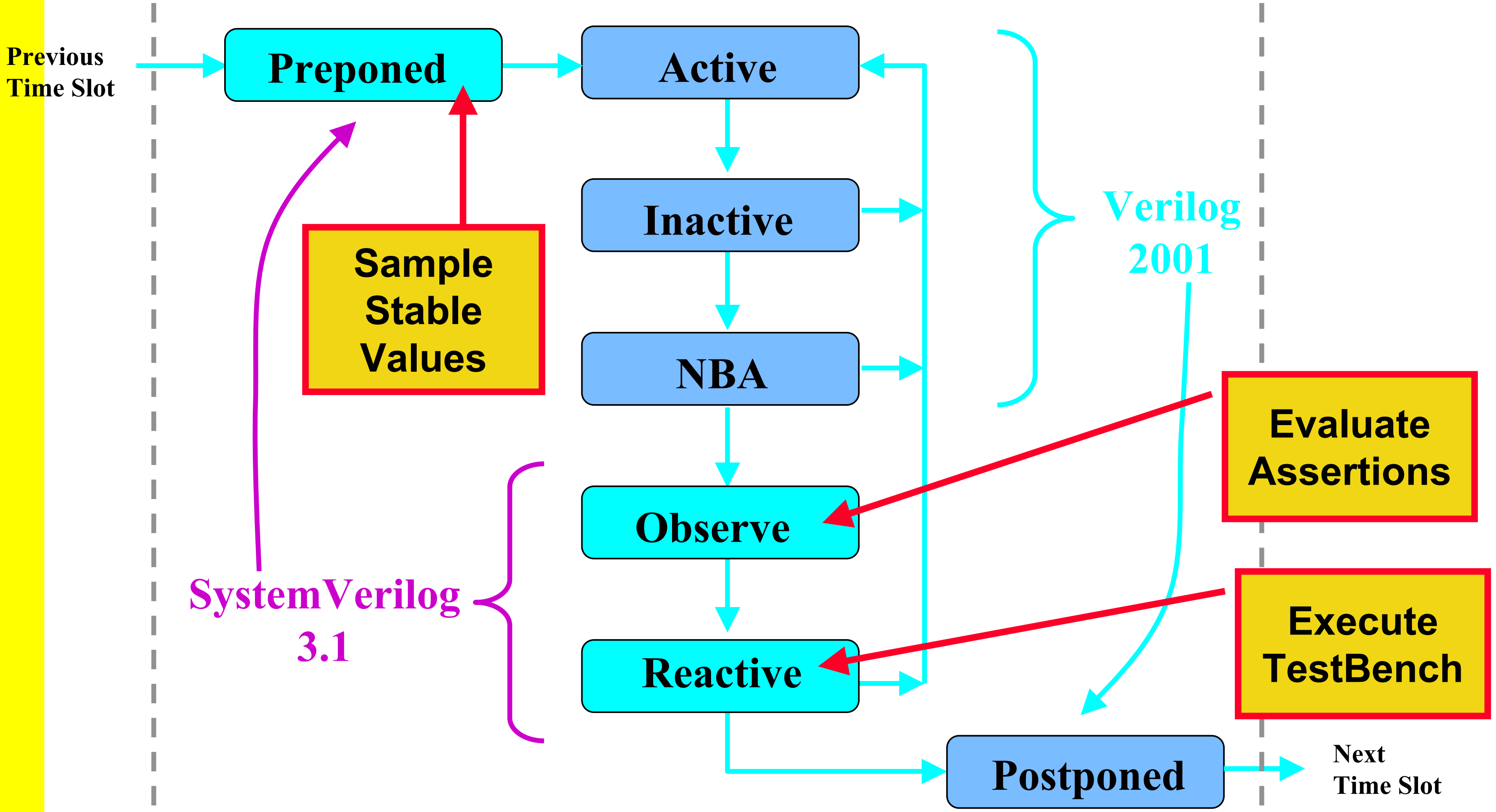
Simulation Monitors and Constraints for Formal Analysis

```
program manual_stimulus_generator;
repeat(1000) begin
  generate_transaction(addr,data);
  while(wait_cnt > 0)
    @(posedge clk) wait_cnt--;
  end
endprogram
```

# SystemVerilog Enhanced Scheduling



# SystemVerilog Enhanced Scheduling



## Agenda

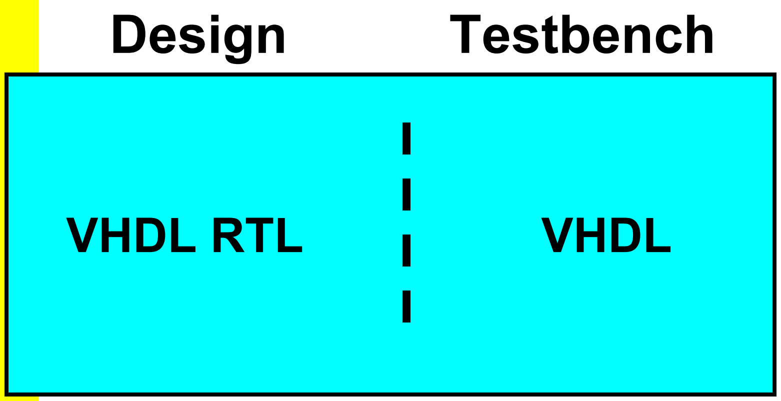
- Introduction
- SystemVerilog Design Features
- SystemVerilog Assertions
- SystemVerilog Verification Features
- **Using SystemVerilog and VHDL Together**

## SystemVerilog With VHDL

- **Verilog-VHDL Interface limited to net/vector types**
  - **VHDL records and arrays packed into bit vectors**
- **SystemVerilog supports higher-level data types**
  - **Synthesizable types are synthesizable across the interface**

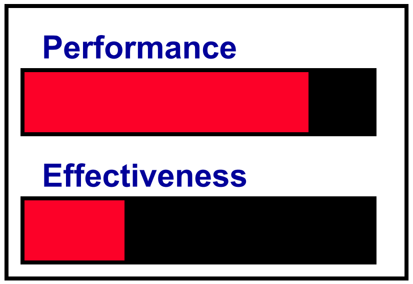
| VHDL | SystemVerilog |
|---|---|
| Record | Struct |
| Array | Array |
| Multi-D Array | Multi-D Array |
| Enum | Enum |

# Pure VHDL Simulation Flow



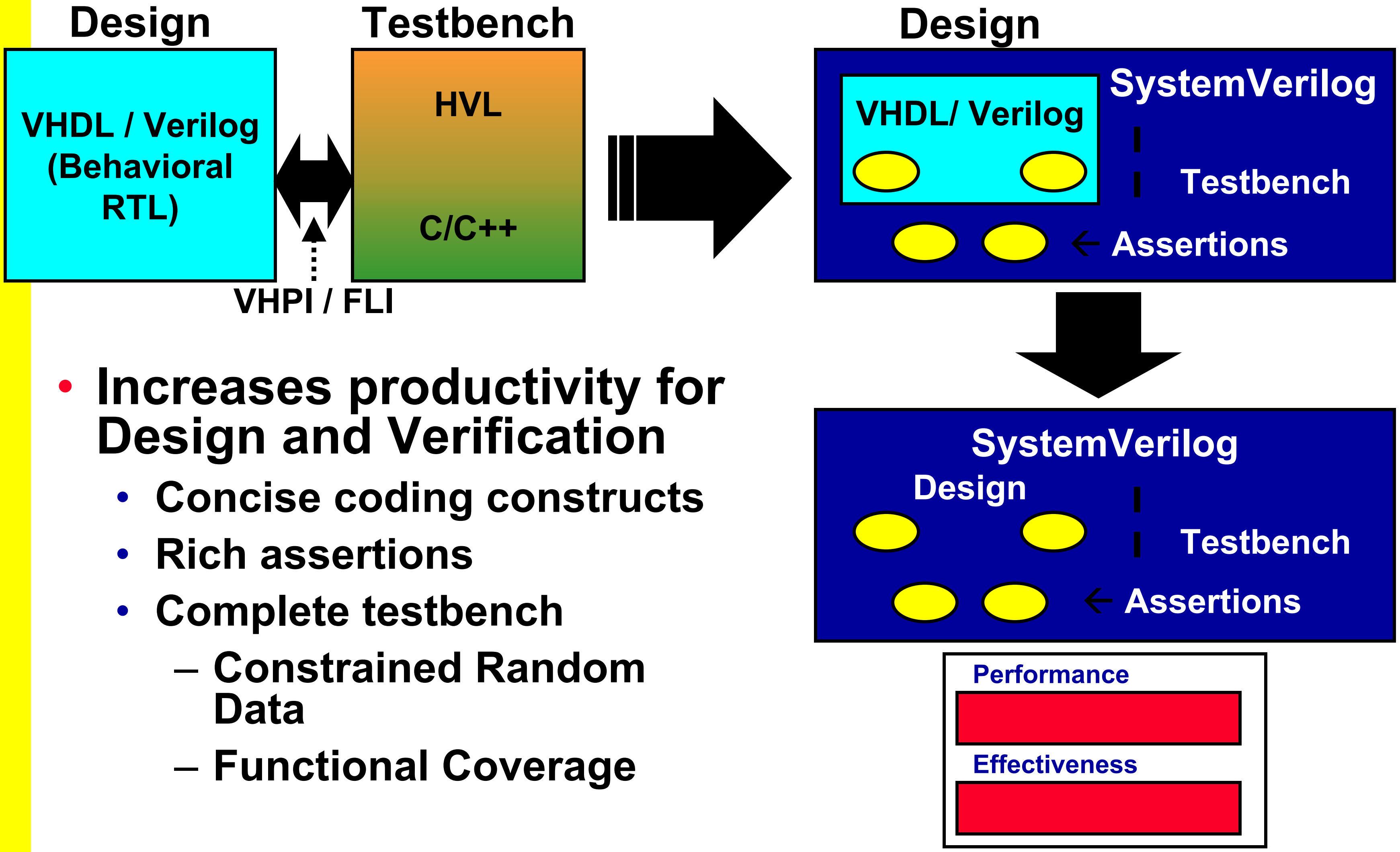
+ Single language (VHDL) for design and testbench

- No constrained random TB
- No temporal assertions
- No functional coverage



- Coherent environment for design and verification
- Limited testbench capabilities in VHDL “promote” a directed test based verification methodology
- Lack of constrained random / assertion / coverage → low “bug-finding effectiveness”

# SystemVerilog is Evolutionary for VHDL and Verilog Users

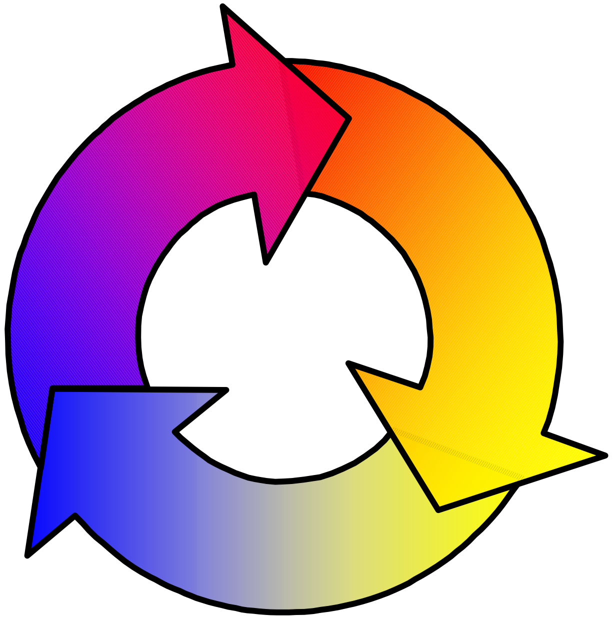


- **Increases productivity for Design and Verification**
  - **Concise coding constructs**
  - **Rich assertions**
  - **Complete testbench**
    - **Constrained Random Data**
    - **Functional Coverage**

# The Importance of a Single Language

<u>Unified Scheduling</u>

- Basic Verilog won’t work
- Ensures Pre/Post-Synth Consistency
- Enables Performance Optimizations



<u>Knowledge of Other Language Features</u>

- Testbench and Assertions
- Interfaces and Classes
- Sequences and Events

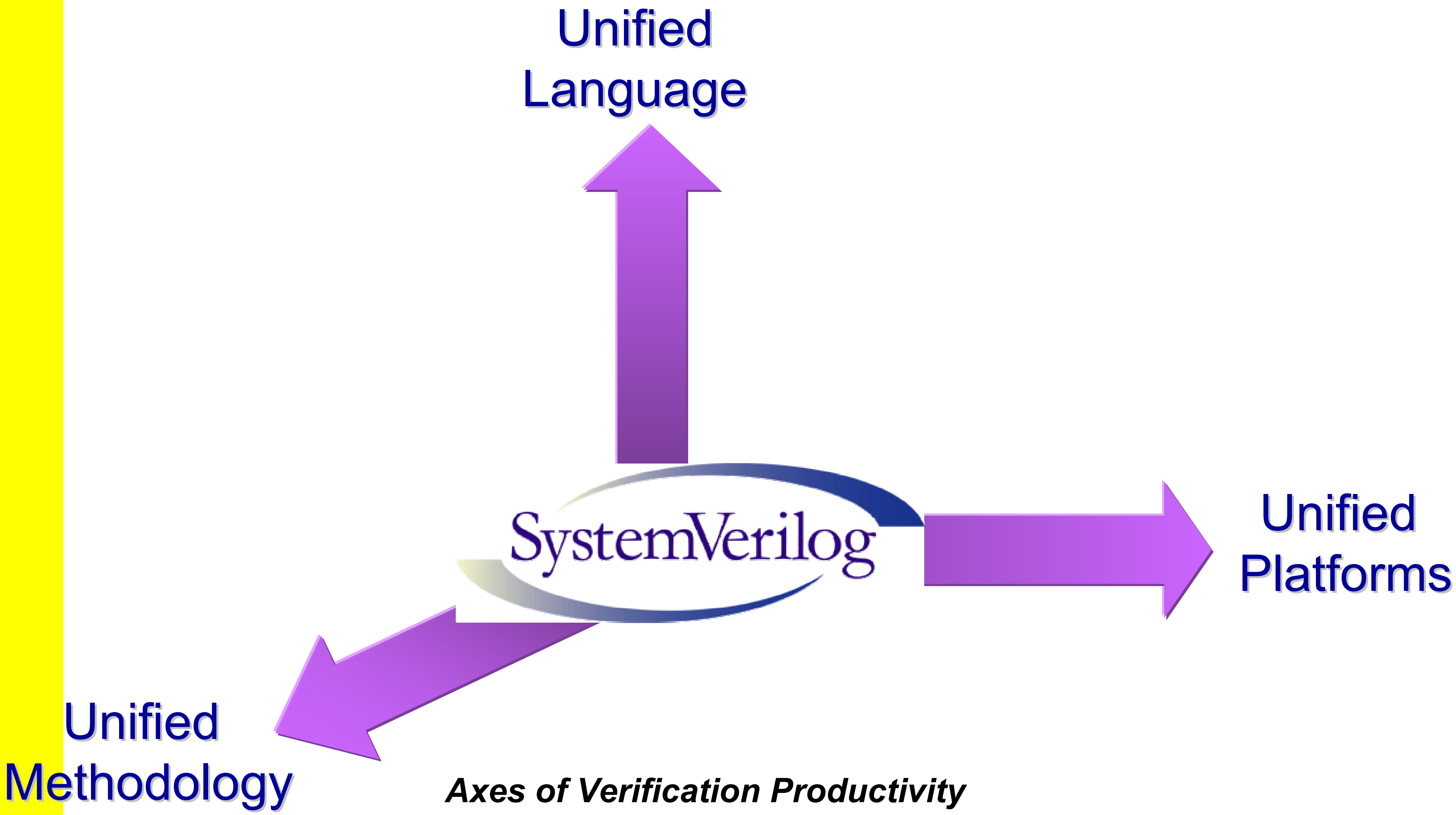
<u>Reuse of Syntax/Concepts</u>

- Sampling for assertions and clocking domains
- Method syntax
- Queues use common concat/array operations
- Constraints in classes and procedural code

## SystemVerilog Benefits for VHDL Users

- Many VHDL modeling features are in SystemVerilog
  - Don’t have to give up high-level data types
  - Some features (enums) extended beyond VHDL capabilities
- Mixed-HDL environments are a reality
  - Higher-level data types supported across boundary
  - Continue to use VHDL legacy blocks
  - Easier to adopt SystemVerilog incrementally
- Industry-Standard Verification Language works with VHDL designs
  - Constrained random data generation
  - Object-oriented
  - Assertions
- SystemVerilog supports *Design for Verification*
  - Interfaces and assertions capture design intent
  - Efficient and intuitive interactions between testbench and assertions

# Evolution of Verification Productivity



***Axes of Verification Productivity***