

Verification

SystemVerilog *Testbench Constructs*

Alex Wakefield
Synopsys, Inc.

October 21, 2004

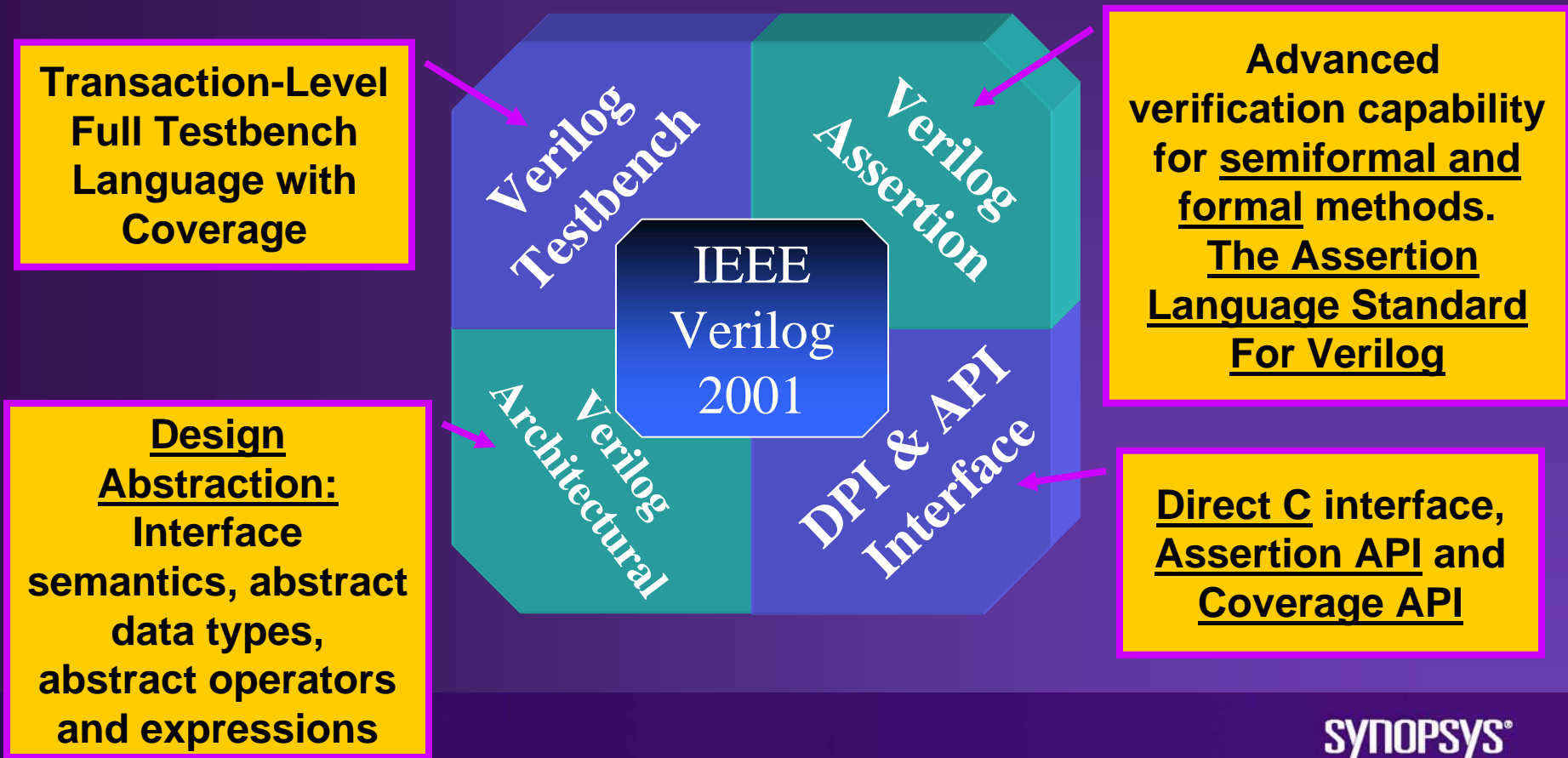


Agenda

- **Introduction**
- **SystemVerilog Language Features**
 - **Data Types**
 - **Classes**
 - **Constraints**
 - **Threads**
 - **Functional Coverage**
- **Conclusion**

SystemVerilog Charter

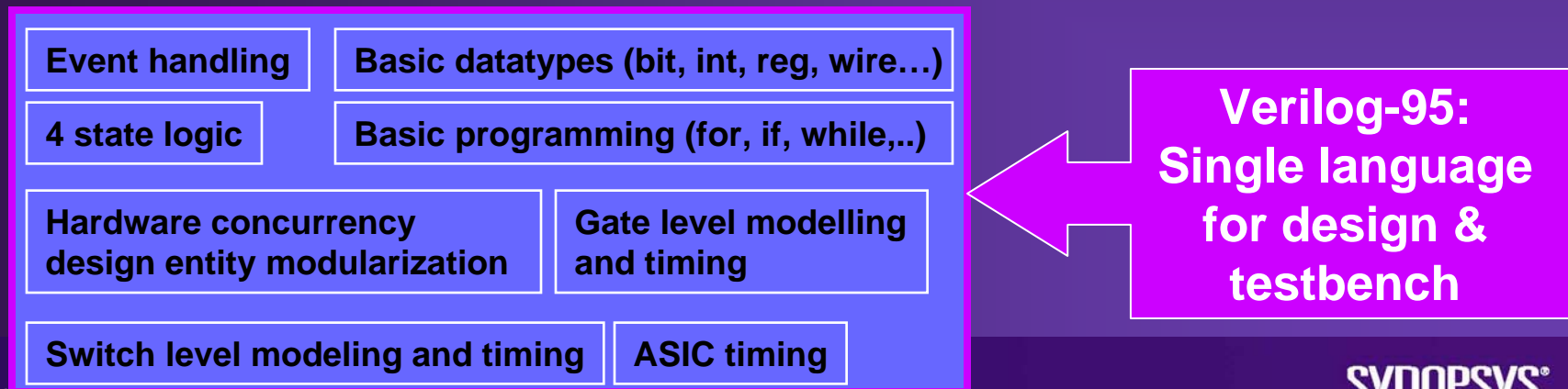
- Charter: Extend Verilog IEEE 2001 to higher abstraction levels for Architectural and Algorithmic Design , and Advanced Verification.



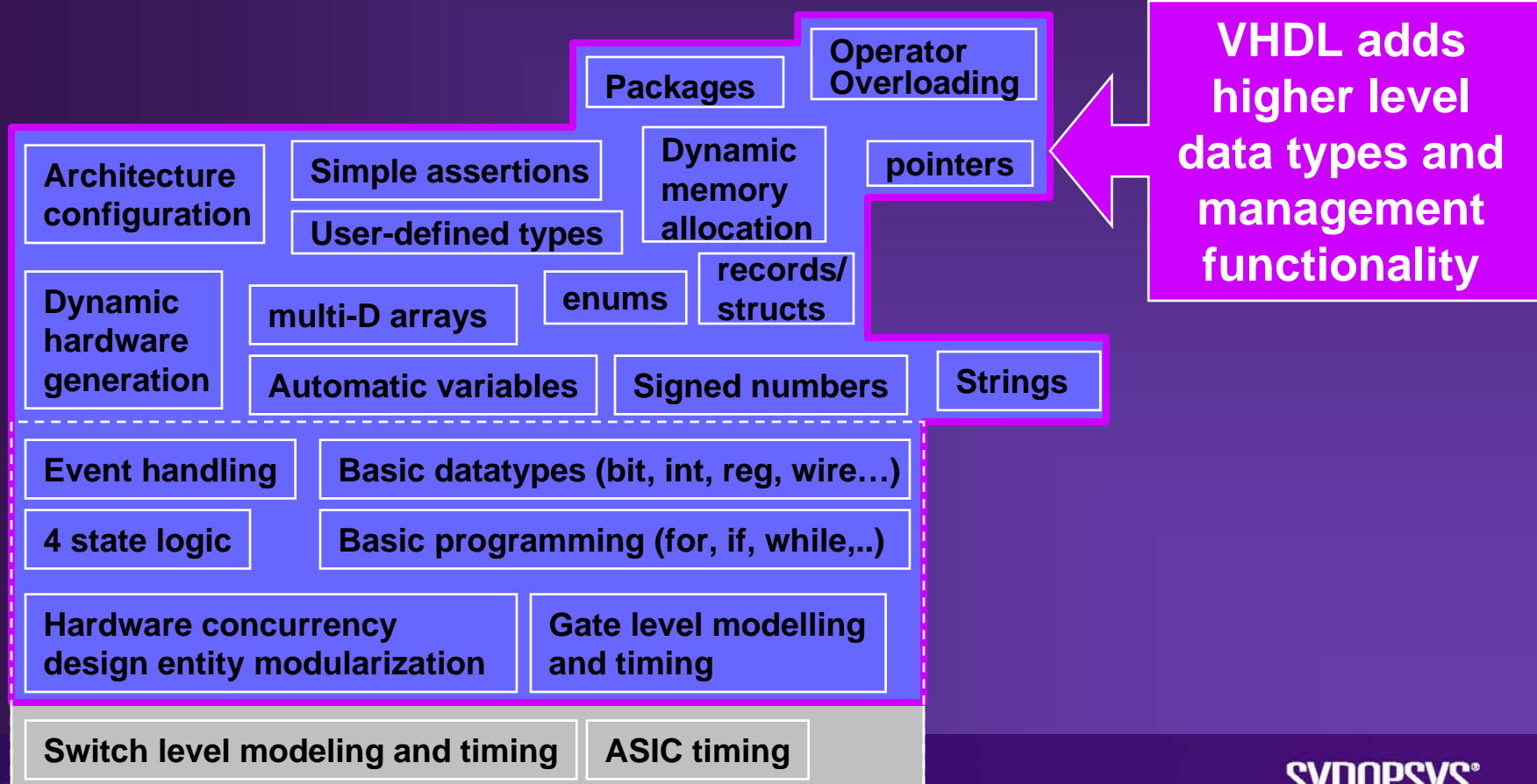
SystemVerilog Benefits (Testbench)

- **Powerful Testbench features**
 - **Classes / Object Oriented Programming**
 - **Constrained Random Testing**
 - **Functional Coverage**
 - **Assertions**
 - **Enables Advanced Verification Methodology**
- **Result is an increase in verification productivity**

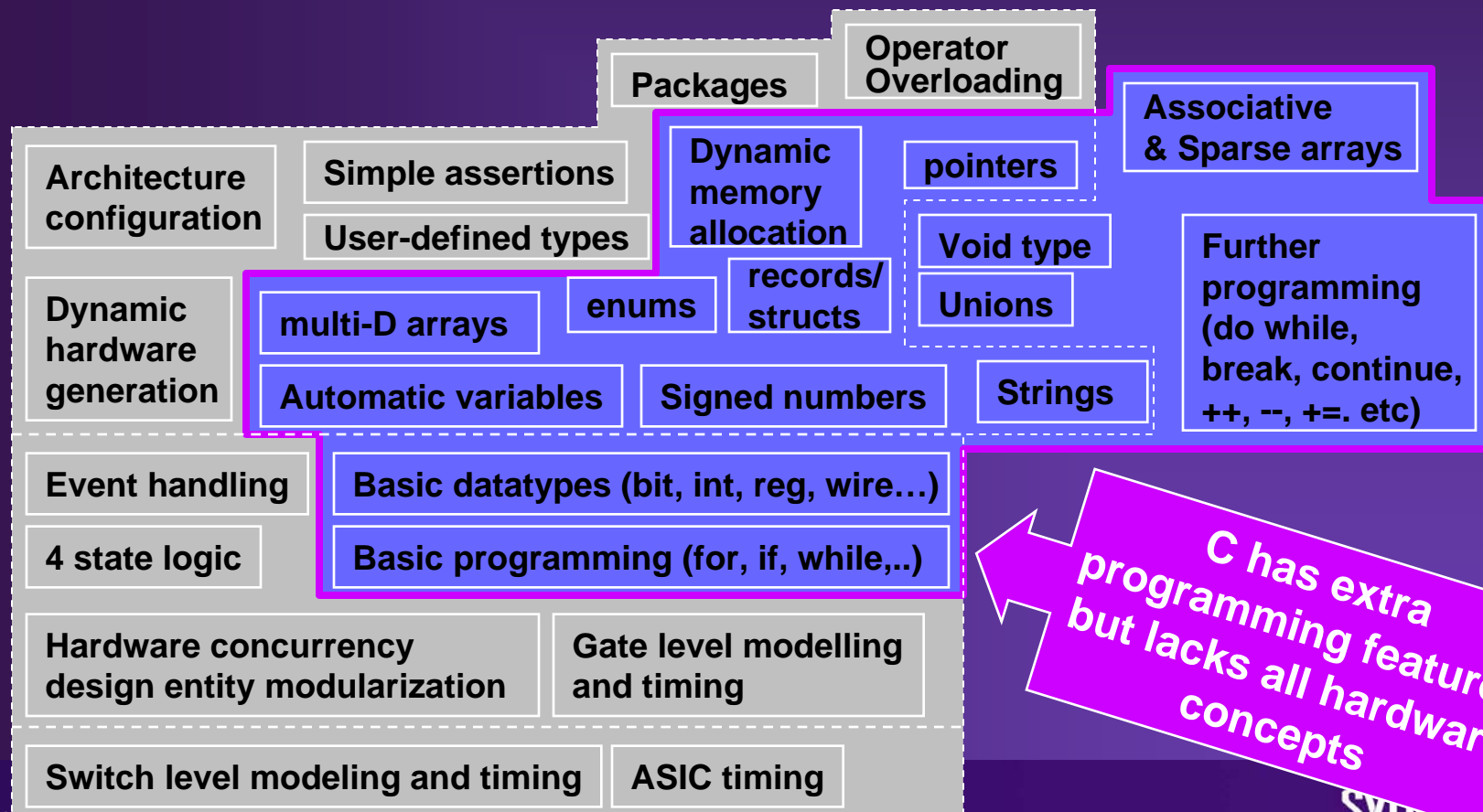
SystemVerilog: Verilog 1995



SystemVerilog: VHDL



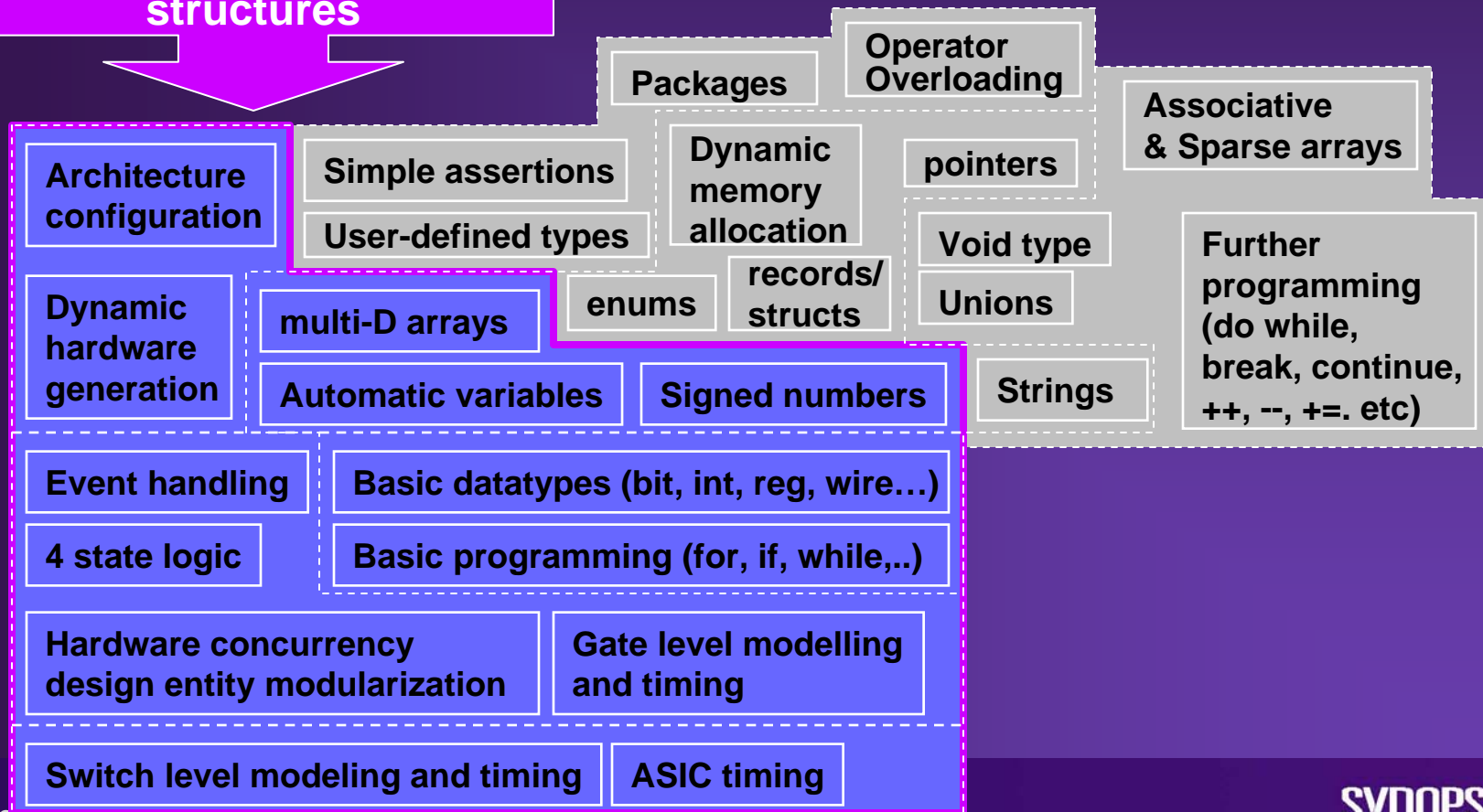
SystemVerilog: C



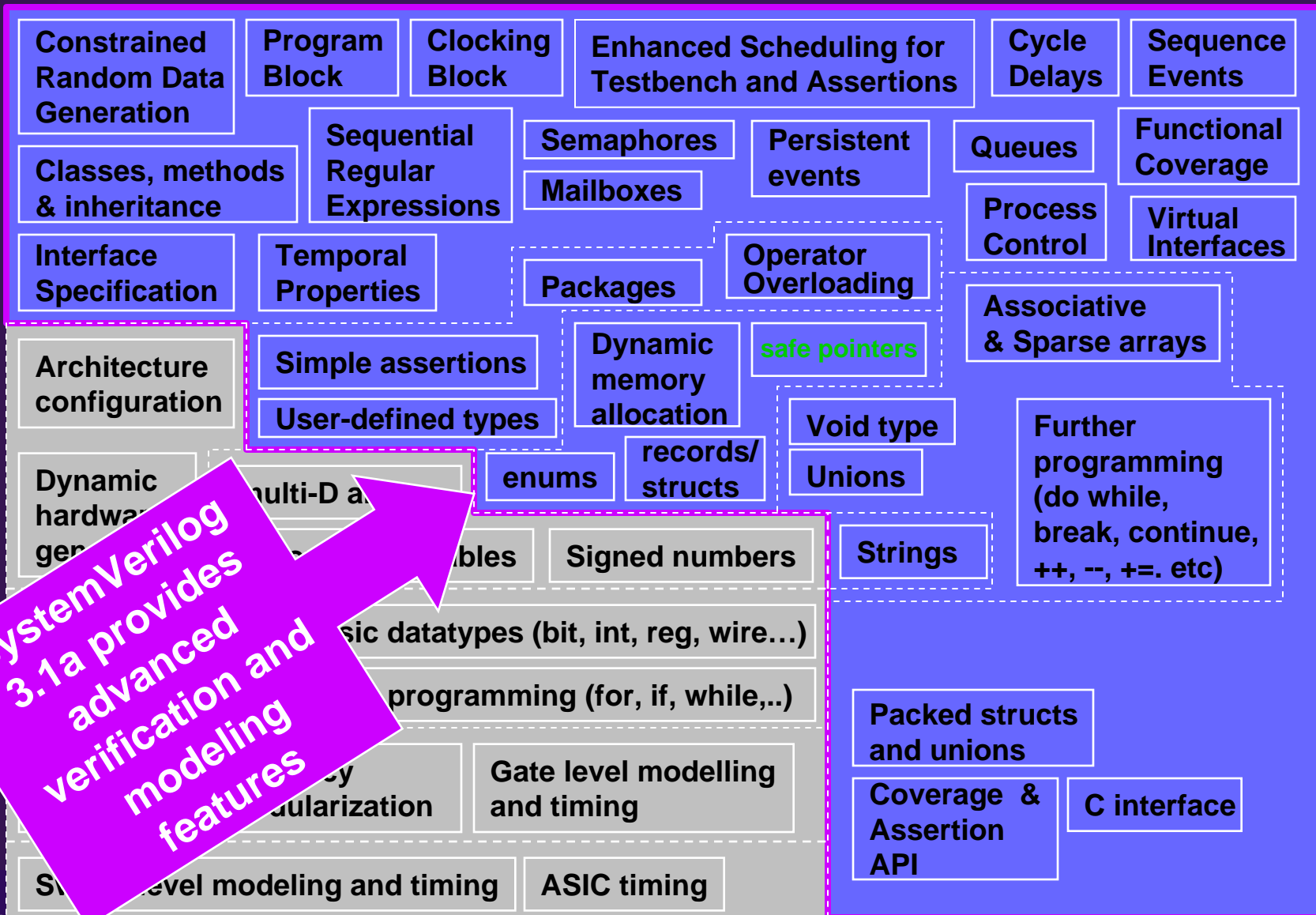
C has extra programming features but lacks all hardware concepts

SystemVerilog: Verilog-2001

Verilog-2001 adds a lot of VHDL functionality but still lacks advanced data structures



SystemVerilog: Enhancements

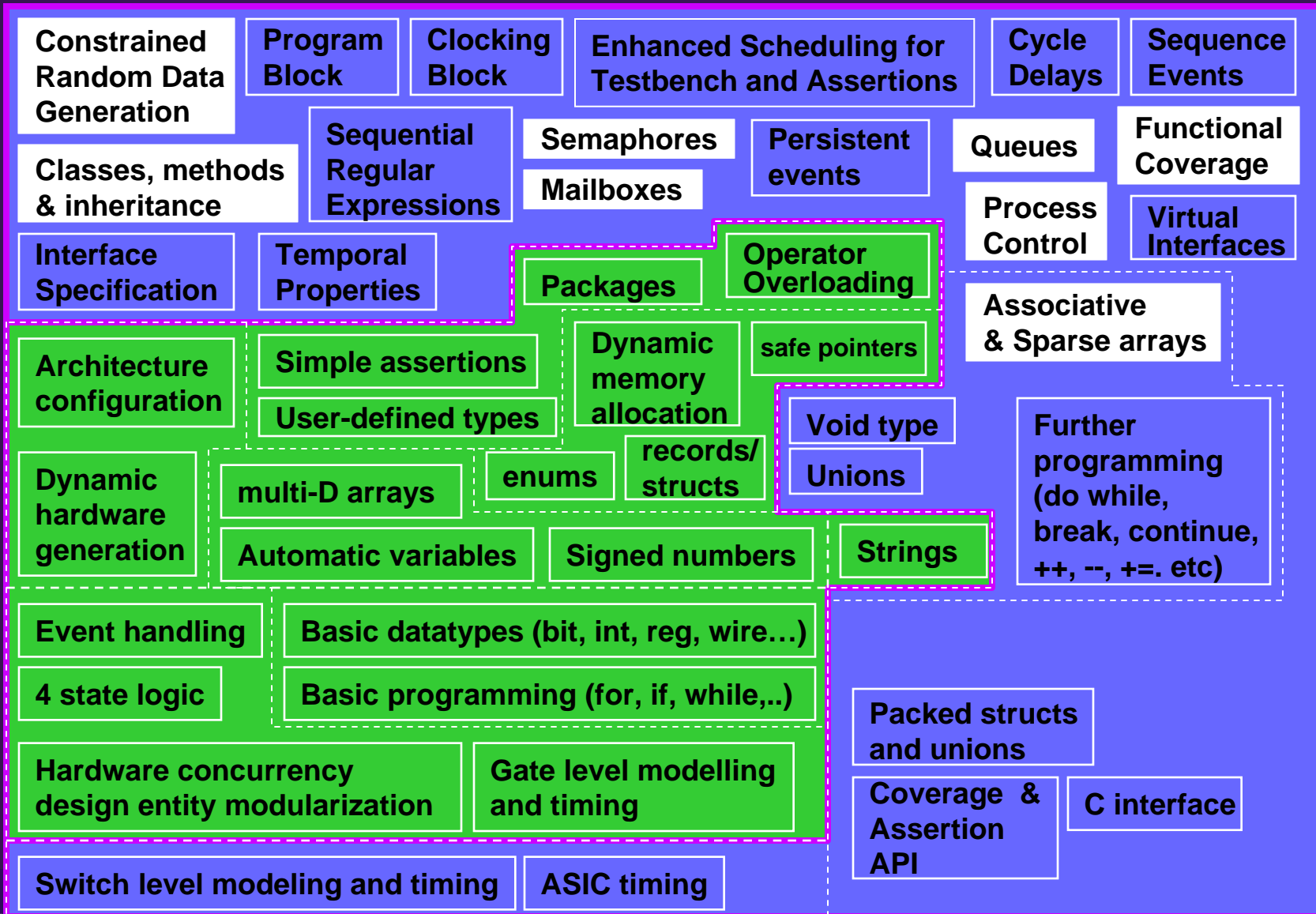


SystemVerilog 3.1a provides advanced verification and modeling features

SystemVerilog: Unified Language

Today's Focus

VHDL



Agenda

- Introduction
- **SystemVerilog Language Features**
 - **Data Types**
 - **Classes**
 - **Constraints**
 - **Threads**
 - **Functional Coverage**
- **Conclusion**

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

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

Structures

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

IR is a struct variable

```
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
```

instruction is a user-defined struct type

Structure definitions are just like in C but without the optional structure tag before the '{'
Equivalent to VHDL records

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 must be the same size



```

ip_t ip_h;

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

Equivalent

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

- **Explicitly Typed**
 - **Allows compile time error checking**

```
typedef enum    { init, decode, ...} fsmstate;  
fsmstate pstate, nstate;  
  
case (pstate)  
  idle: if (sync)  
        nstate = init;  
  init: if (rdy)  
        nstate = decode;  
  ...  
endcase
```

Queues

- **Variable-sized Array: data_type name [\$]**
 - Uses array syntax and operators
- **Synthesizable if maximum size is known**
 - `q[$:25]` // maximum size is 25

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



Associative Arrays

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

Declaration

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

Methods

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

Example

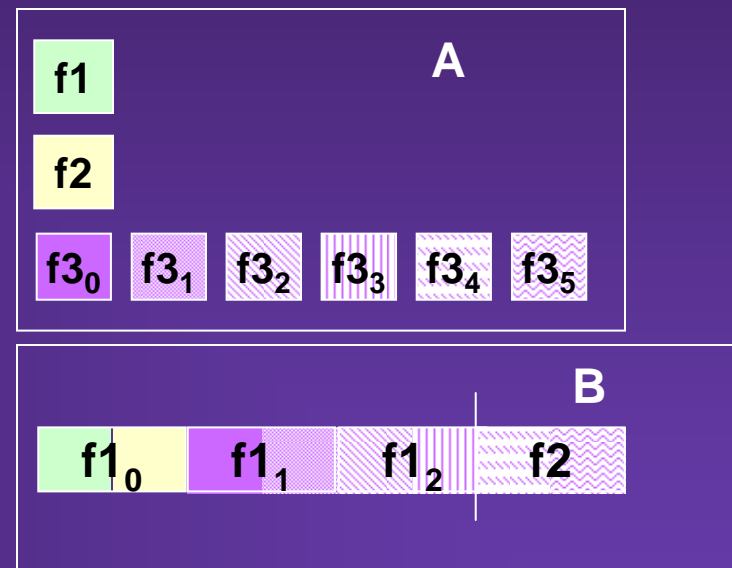
```
struct packed {int a; logic[7:0] b} mystruct;  
int myArr [mystruct]; //Assoc array indexed by mystruct
```

Ideal for Dealing with Sparse Data

Packing and Unpacking

- Reshape any aggregate bit-level object
- Packed \Leftrightarrow Unpacked, Array \Leftrightarrow Structure

```
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);
```



Objects must have identical bit size

Agenda

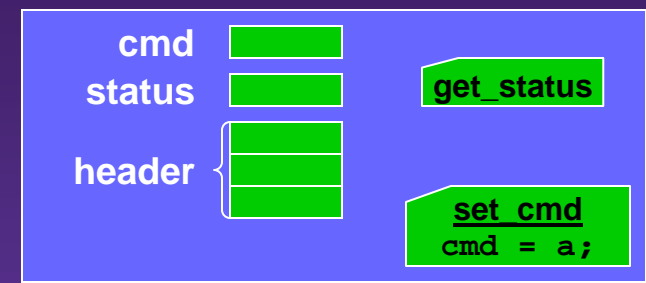
- Introduction
- SystemVerilog Language Features
 - Data Types
 - **Classes**
 - Constraints
 - Threads
 - Functional Coverage
- Conclusion

Object-Oriented Programming

•Classes

- Encapsulate data and methods
- Storage allocated via “new” method (User may define custom “new” method)
- Automatic garbage collection

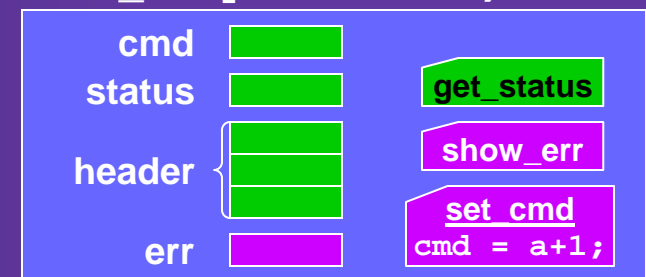
```
pkt_t MyPkt = new;
```



•Inheritance

- Allows hierarchical definition of objects. Subclass inherits from base class
- Can redefine properties, methods and constraints

```
err_t extends pkt_t;  
err_t myPkt = new;
```



Class Definition

Definition syntax

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

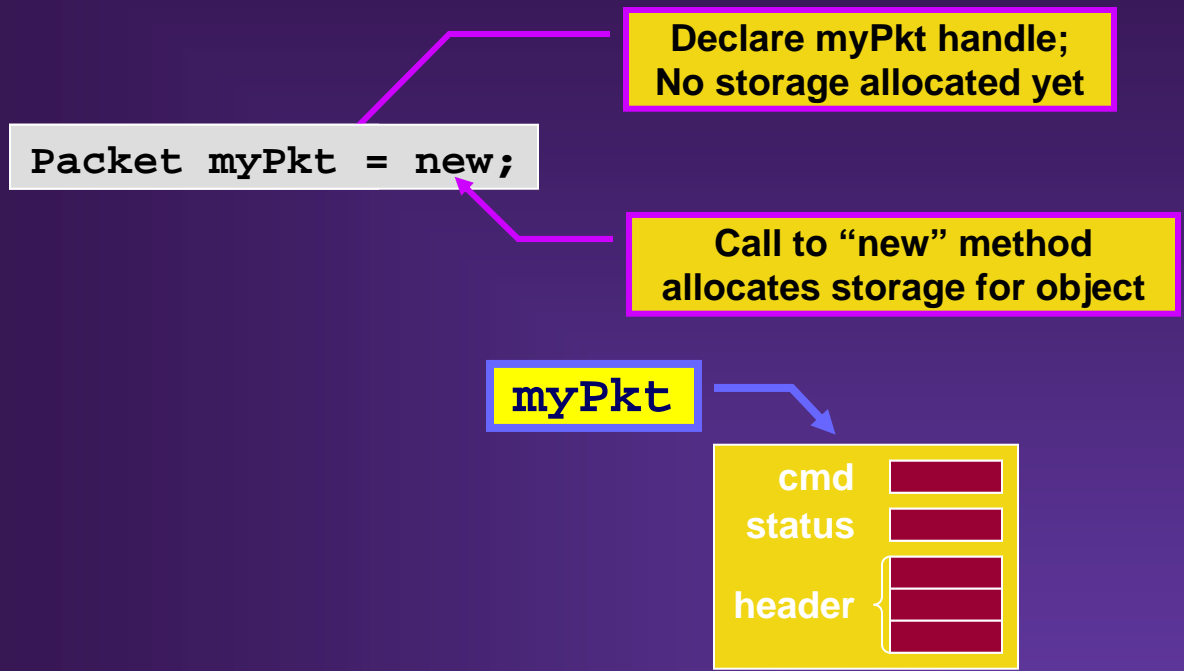
extern keyword
allows for out-of-body
method declaration

“::” operator links
method declaration to
Class definition

```
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
```

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

Class Instantiation



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

Class Inheritance & Extension

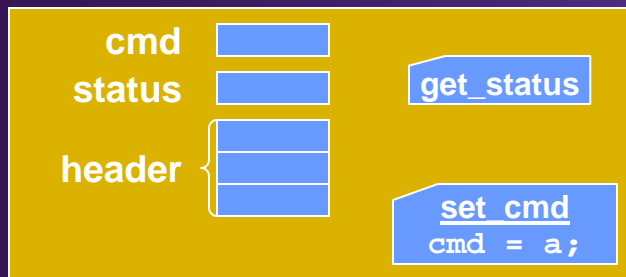
• Keyword *extends*

Denotes Hierarchy of Definitions

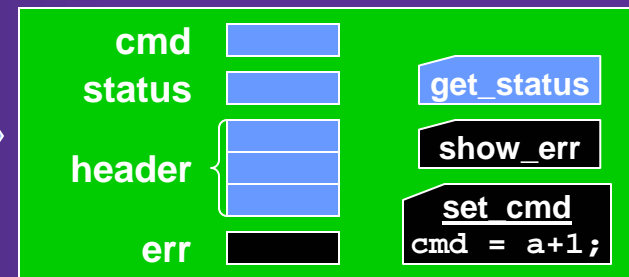
- Subclass inherits properties, constraints and methods from parent
- Subclass can redefine methods explicitly

```
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
```

Packet:



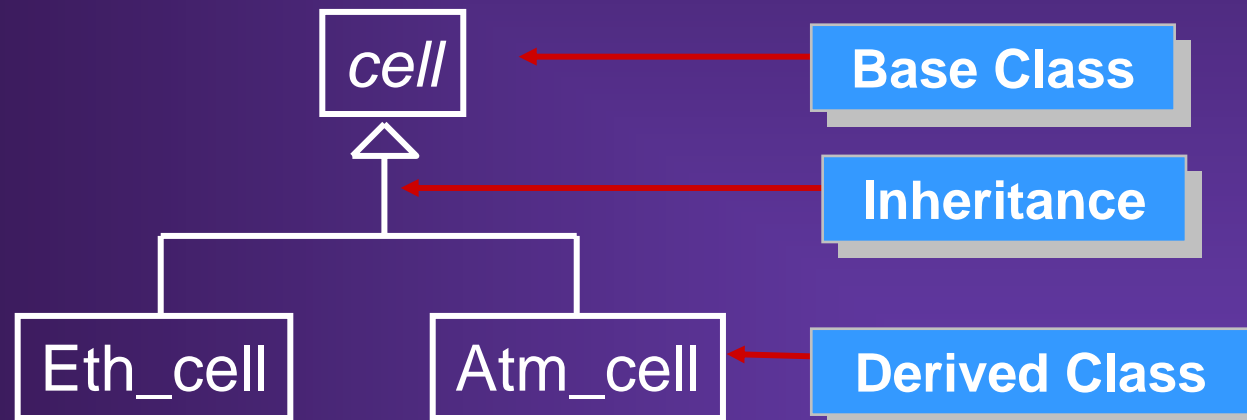
ErrPkt:



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

Class Hierarchy : UML

- Class design can be documented using Universal Modeling Language diagrams

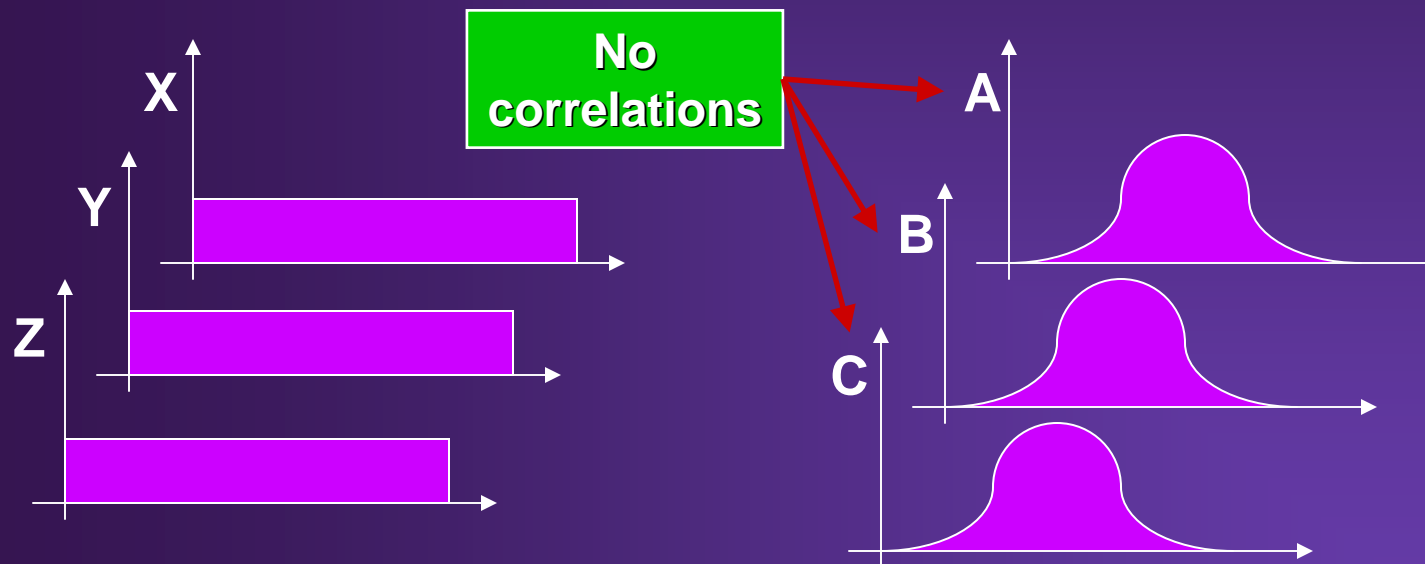


Agenda

- Introduction
- SystemVerilog Language Features
 - Data Types
 - Classes
 - **Constraints**
 - Threads
 - Functional Coverage
- Conclusion

Constraints vs Distribution Functions

- Constraints allow you to define the distribution of random variables



Constrained Random Data

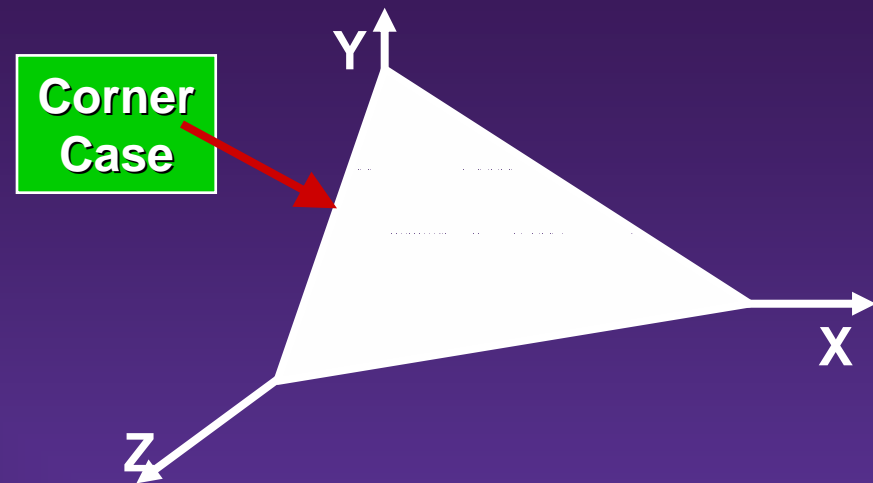
- Constraints
 - Declarative constraints built into class system
 - Built-in *randomize* method calls constraint solver
 - *randomize with* construct adds constraints in-line

```
class packet {  
    rand bit [7:0] src;  
    rand bit [7:0] dst;  
    constraint valid { src[0] == 0; }  
    constraint legal { dst < 10; }  
}
```

```
packet pkt = new();  
void = pkt.randomize() with { dst > 2 };
```

Constraints vs Distribution Functions

- Constraints describe solution space and can constrain multiple random variables simultaneously



```
class packet {
  rand bit [7:0] src;
  rand bit [7:0] dst;
  constraint valid { src[0] == 0; }
  constraint legal { dst < 10; }
  constraint corner { src == dst + 1; }
}
```

Layered Constraints

- **Constraints Inherited via Class Extension**
 - Just like data and methods, constraints can be inherited or overridden
 - All constraints are solved at one time (child and parent)

```
class packet_src0 extends packet {  
    constraint src0 { src == 0 };  
}
```

- **Constraints can be checked in line**

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

Passing “null” argument to randomize checks that the current state variable values satisfy the constraints
0 = valid, 1 = invalid



Iterative Constraints


- The foreach construct
 - Specifies iteration over elements of an array
 - Values of all elements of array can be solved for simultaneously

```
class Instruction;  
  rand Opcode op[];  
  rand byte Address[];  
  constraint c1 {foreach(op[i])  
                (i<op.size-1)->(op[i+1] != op[i]);}  
  constraint c2 {foreach(Address[j]) Address[j] > 2*j;}  
  extern task init_arrays(n_op, n_addr);  
endclass
```

Associative
Array



Consecutive
Opcodes
Differ

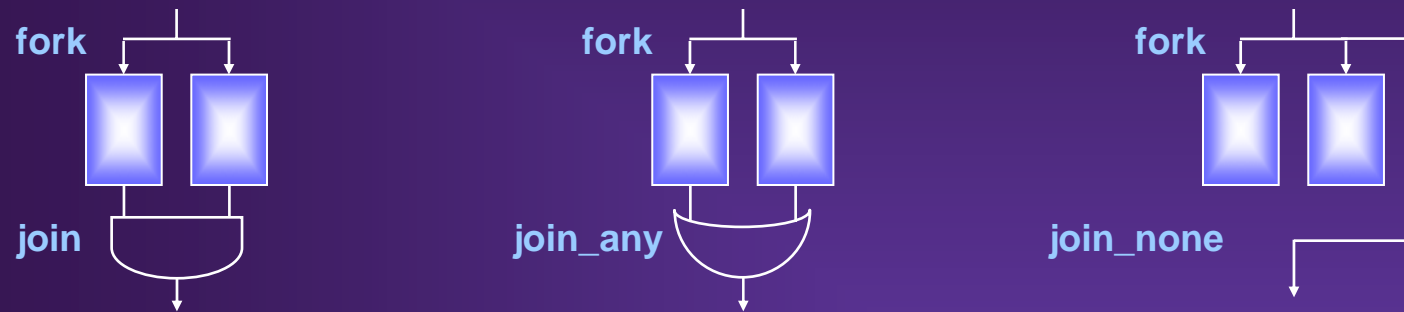


Agenda

- Introduction
- SystemVerilog Language Features
 - Data Types
 - Classes
 - Constraints
 - **Threads**
 - Functional Coverage
- Conclusion

Dynamic Processes and Threads

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



- 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
 - `$exit` terminates the main program thread

Events – Enhanced from V2K

- Events are variables
 - Can be copied, passed to tasks
- `wait_order()`, `wait_any()`, `wait_all(<events>)`;
- `event.triggered`;
 - Persists throughout time-slice, avoids races

```
sequence abc;  
    @(posedge clk) a ##1 b ##1 c;  
endsequence
```

```
program test;  
    initial begin  
        @abc $display("Detected a-b-c seq");  
    end  
endprogram
```



Semaphore

- Semaphores are used to control unique access to a resource
- Built-in Class and methods
 - `get`, `put`, `try_get`

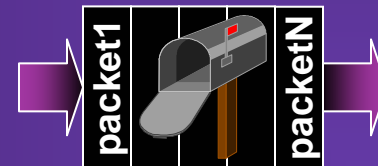
```
semaphore semID = new(1);  
semID.get(1);  
semID.put(1);
```



Mailbox

- Mailbox – Built-in Class
 - Built-in methods: num(), put(), try_put(), get(), try_get(), peek(), try_peek()
 - Arbitrary type

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



Ensures meaningful, race-free communication between processes

Agenda

- Introduction
- SystemVerilog Language Features
 - Data Types
 - Classes
 - Constraints
 - Threads
 - **Functional Coverage**
- Conclusion

Coverage

- **Several types of coverage:**
 - **Code Coverage**
 - **Functional Coverage**
 - **Assertion Coverage**

- **Focus on Functional Coverage in this presentation...**

Functional Coverage

Coverage Objects

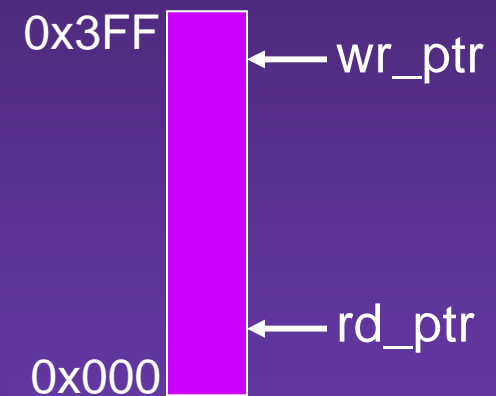
- **Random testing:**
 - Did it work?
 - What did it do?
- **Coverage is a list of items to test**
 - Replaces test list
- **When you get to 100%, design is done**
- **Shows testing progress**



Functional Coverage

Coverage Objects

- **Example: 1k FIFO buffer**
 - **Sampled: read ptr, write ptr**
 - 1M coverage points
 - **Of interest: FIFO occupancy level**
 - Cover: $(rd_ptr - wr_ptr) \% 1k$
 - Still 1k coverage points
 - **Of real interest: empty, full**
 - Cover: occupancy in $\{0; 1k\}$
 - 2 coverage points
 - **Other potential points of interests**
 - Empty->Full->Empty transitions
 - Read ptr ahead of write ptr
 - Write ptr ahead of read ptr
 - Empty/Full with ptr in $\{0; 1k\}$



Functional Coverage

- New covergroup allows declaration of
 - Coverage points
 - Variables, Expressions, Transitions
 - Cross coverage
 - Sampling expression : clocking event

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

3 bins for color

```
covergroup g1 @(posedge clk);
```

```
  c: coverpoint color;
```

16 bins for pixel

```
  a: coverpoint pixel_adr;
```

```
  AxC: cross color, pixel_adr;
```

48 cross products

```
endgroup;
```

Agenda

- Introduction
- SystemVerilog Language Features
 - Data Types
 - Classes
 - Constraints
 - Threads
 - Functional Coverage
- Conclusion

SystemVerilog Feature Overview

Testbench Features

Dynamic Arrays Associative Arrays
Classes Inheritance Parameterized Classes
Program Block Clocking Domain Cycle Delays
Randomization & Constraints Dynamic Processes
Enhanced Events Mailboxes & Semaphores
Enhanced Constraints General Randomization
foreach Process Control Pack/Unpack
Functional Coverage Virtual Interfaces Queues

Design Features

Structs Unions Packed Structs & Unions
Multi-d arrays Multi-d packed arrays
User-defined types Enums
Interfaces Modports
Implicit Port Connections
Always_comb Always_latch Always_ff
Packages Separate Compilation Operator Overloading

Sequence Events
Expect

Procedural Instantiation

Assertion Features

Sequences Properties
Sequence Operations
Bind Implication
Declarative Instantiation

Property Composition
Property Operations



The Importance of a Single Language

Unified Scheduling

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



Knowledge of Other Language Features

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

Reuse of Syntax/Concepts

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

Thank You

Q&A