Agenda

Introduction: SystemVerilog Motivation
Vassilios Gerousis, Infineon Technologies
Accellera Technical Committee Chair

Session 1: SystemVerilog for Design

- **Language Tutorial**
  Johny Srouji, Intel
- **User Experience**
  Matt Maidment, Intel

Session 2: SystemVerilog for Verification

- **Language Tutorial**
  Tom Fitzpatrick, Synopsys
- **User Experience**
  Faisal Haque, Verification Central

Lunch: 12:15 – 1:00pm

Session 3: SystemVerilog Assertions

- **Language Tutorial**
  Bassam Tabbara, Novas Software
- **Technology and User Experience**
  Alon Flaisher, Intel
- **Using SystemVerilog Assertions and Testbench Together**
  Jon Michelson, Verification Central

Session 4: SystemVerilog APIs
Doug Warmke, Model Technology

Session 5: SystemVerilog Momentum

- **Verilog2001 to SystemVerilog**
  Stuart Sutherland, Sutherland HDL
- **SystemVerilog Industry Support**
  Vassilios Gerousis, Infineon

End: 5:00pm

Testbench : Recurring Theme

- Increase Level of Abstractions
  - Timing
  - Synchronization
  - Data type
  - Data manipulation
- Eliminate Static Limitations
  - Dynamic data
  - Dynamic fan-out
  - Dynamic processes

Testbench environments are much more like software programs than like hardware descriptions

New Basic Data Types

- String
  - Arbitrary length char array. Automatically resized.
- Arrays
  - Dynamic and associative
- Mailbox and Semaphore
  - Built-in synchronization classes.
- Event Variables
  - Can be used in assignments and as arguments
- Class
  - Allows object-oriented programming

Dynamic Arrays

Declaration syntax

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

dyn

Dynamic Arrays

Declaration syntax

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

Initialization syntax

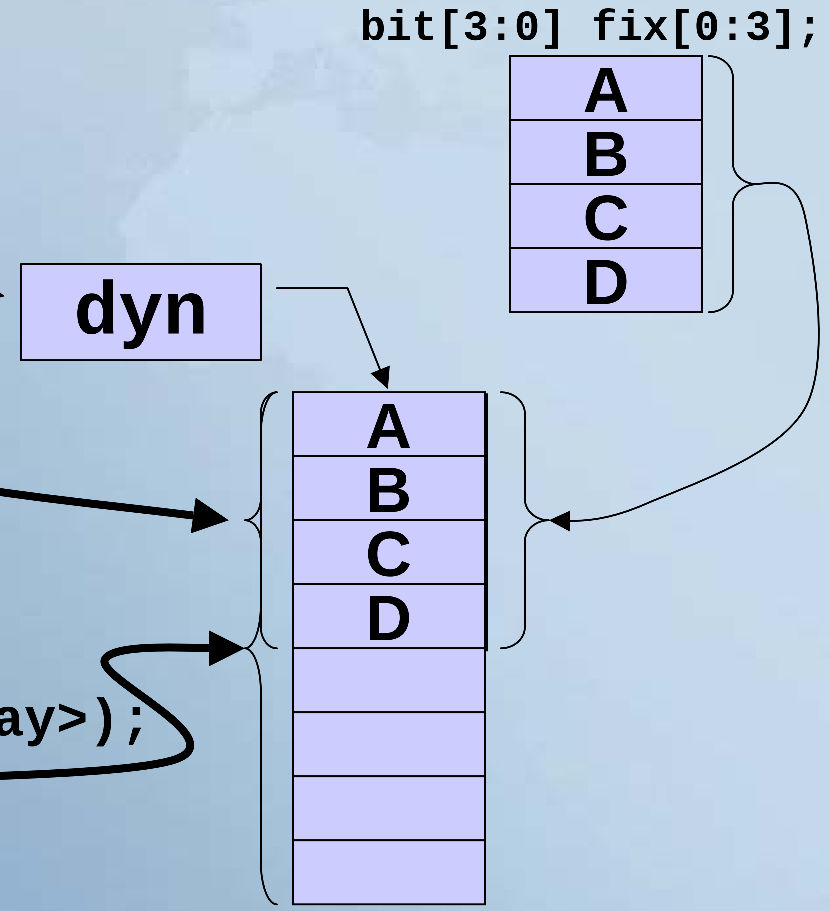
```
<array> = new[<size>];
dyn = new[4];
```

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

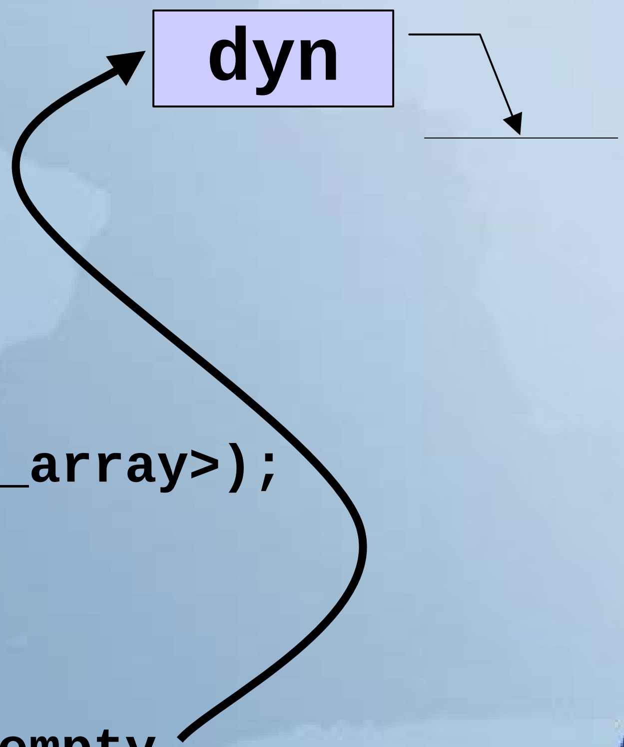
```
function int size();
int j = dyn.size;//j=4
```

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

- 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

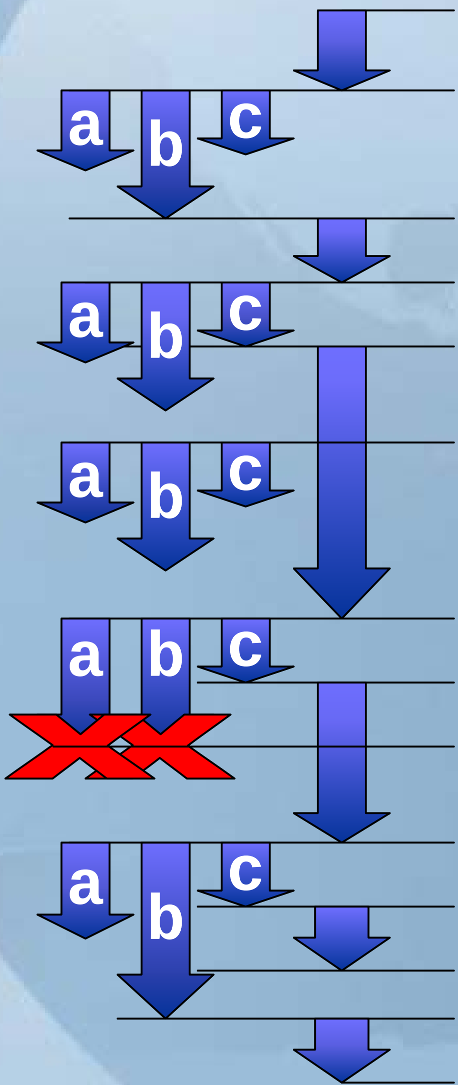
```
struct packed {int a; logic[7:0] b} mystruct;
int myArr [mystruct]; //Assoc array indexed by mystruct
```

Built-in Methods

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

Ideal for Dealing with Sparse Data

Processes and Threads



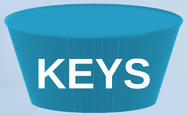

```
program
fork
 a;b;c;
join
fork a;b;c;
join_any
fork a;b;c;
join_none
fork a;b;c;
join_any
disable fork;// kill all child processes
fork a;b;c;
join_any
wait fork;// wait for children to complete
#10;
$exit
```

- Threads created via fork…join
- Threads execute until a blocking statement
  - wait for: (event, mailbox, semaphore, variable, etc.)
- $exit terminates the main program thread

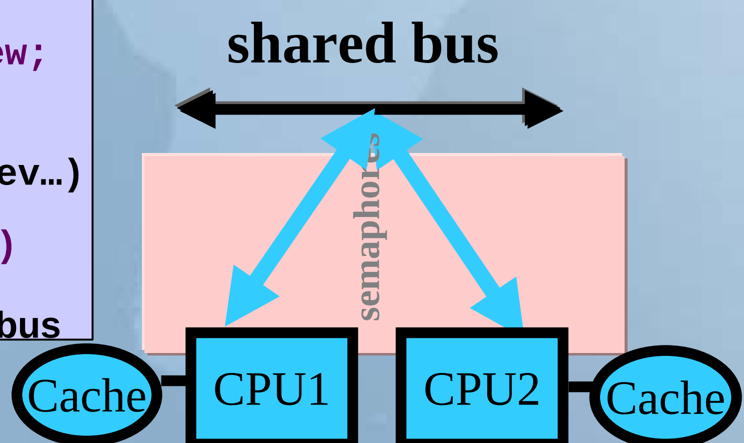
Multiple Independent Threads Maximize Stimulus Interactions

Inter-Process Synchronization

- Events – enhanced from V2K
  - 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
  - Synchronization for arbitration of shared resources, keys.
  - Mutual Exclusivity control
  - Built-in methods: **`get, put, try_get`**




```
// main program .....
 semaphore  shrdBus =new;
  ...
task go_cpu(id,
            ref 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
...
```

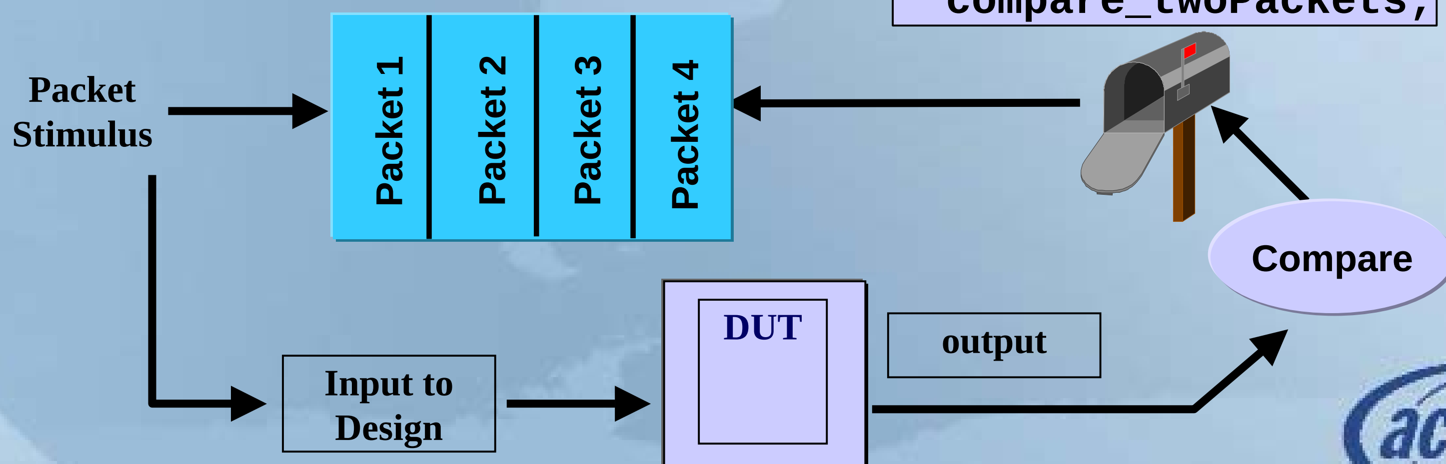
Guarantees Race-Free Synchronization Between Processes

Mailbox: communication

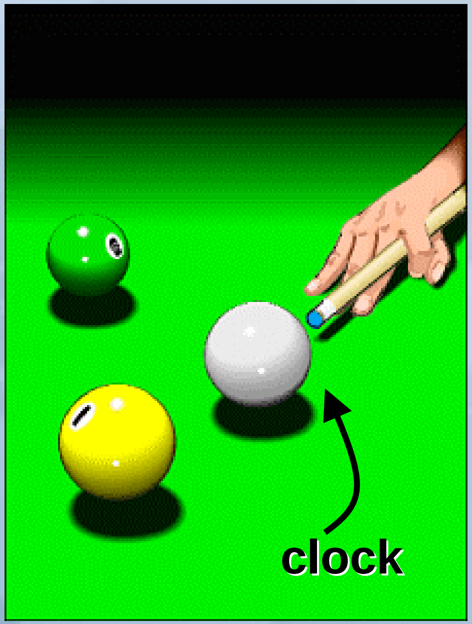
- **Mailbox features**
  - FIFO message queue: passes data between threads
  - Can suspend thread, used for data checking
- **Mailbox built-in methods**
  - new(), num(), put(), try_put(), get(), try_get(), peek(), try_peek()

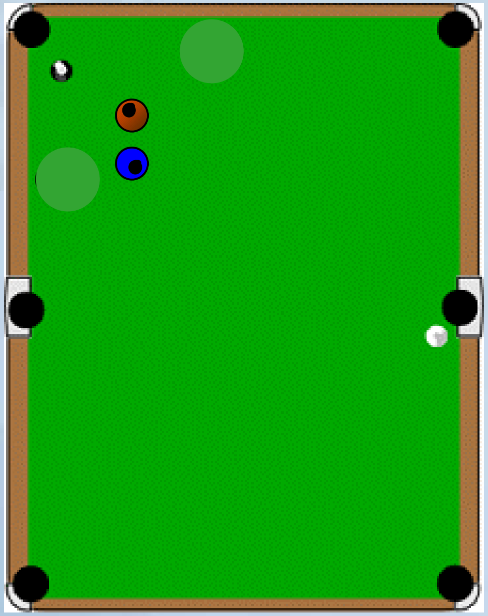
Testbench compares the actual output with expected output [Packets]

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



Skews and Sampling: Why?





Examine current state

Compute next move

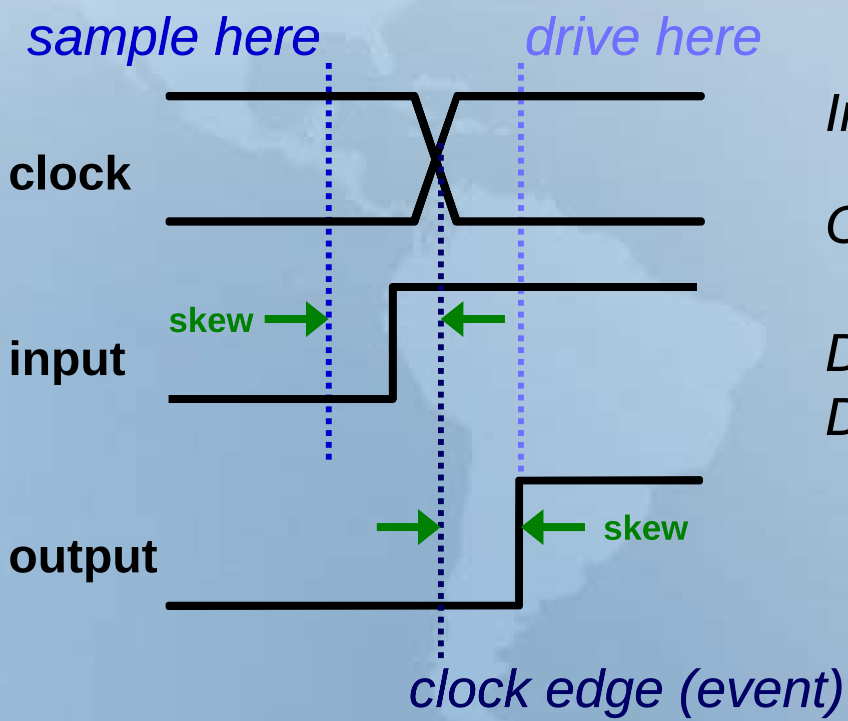
Bet more money

Shoot

Need the steady state !

Skew Declaration

- **Specify Synchronous Sample / Drive Times**

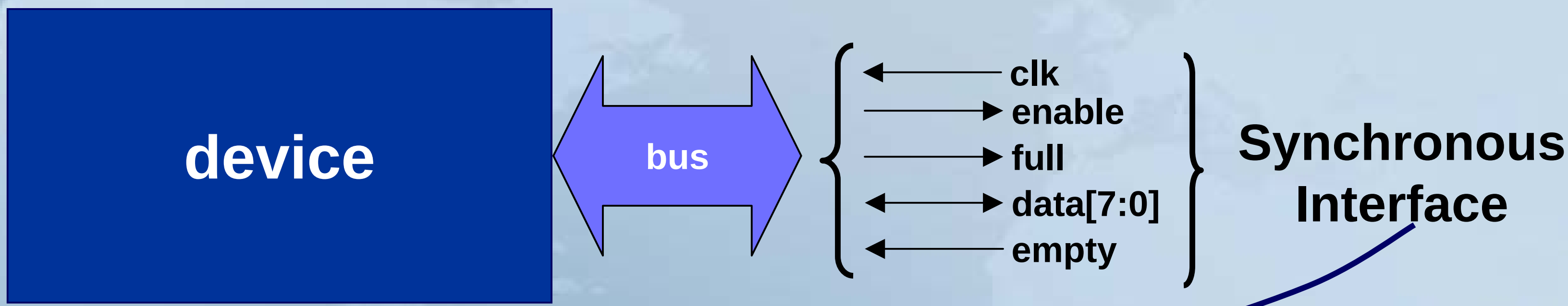


Input skew is for sampling

Output skew is for driving

Default input skew is ***1step***
Default output skew is 0.

Synchronous Interfaces: Clocking




```
clocking bus @(posedge clk);
  default input #1ns output #2ns;
  input          enable, full;
  inout          data;
  output         empty;
  output  #6ns   reset = top.u1.reset;
endclocking
```

Clocking Event "clock"

Default I/O skew

Hierarchical signal

Override Output skew

Testbench Uses:

```
bus.enable
bus.data
 ...
```

Default Clocking and Synchronous Drives

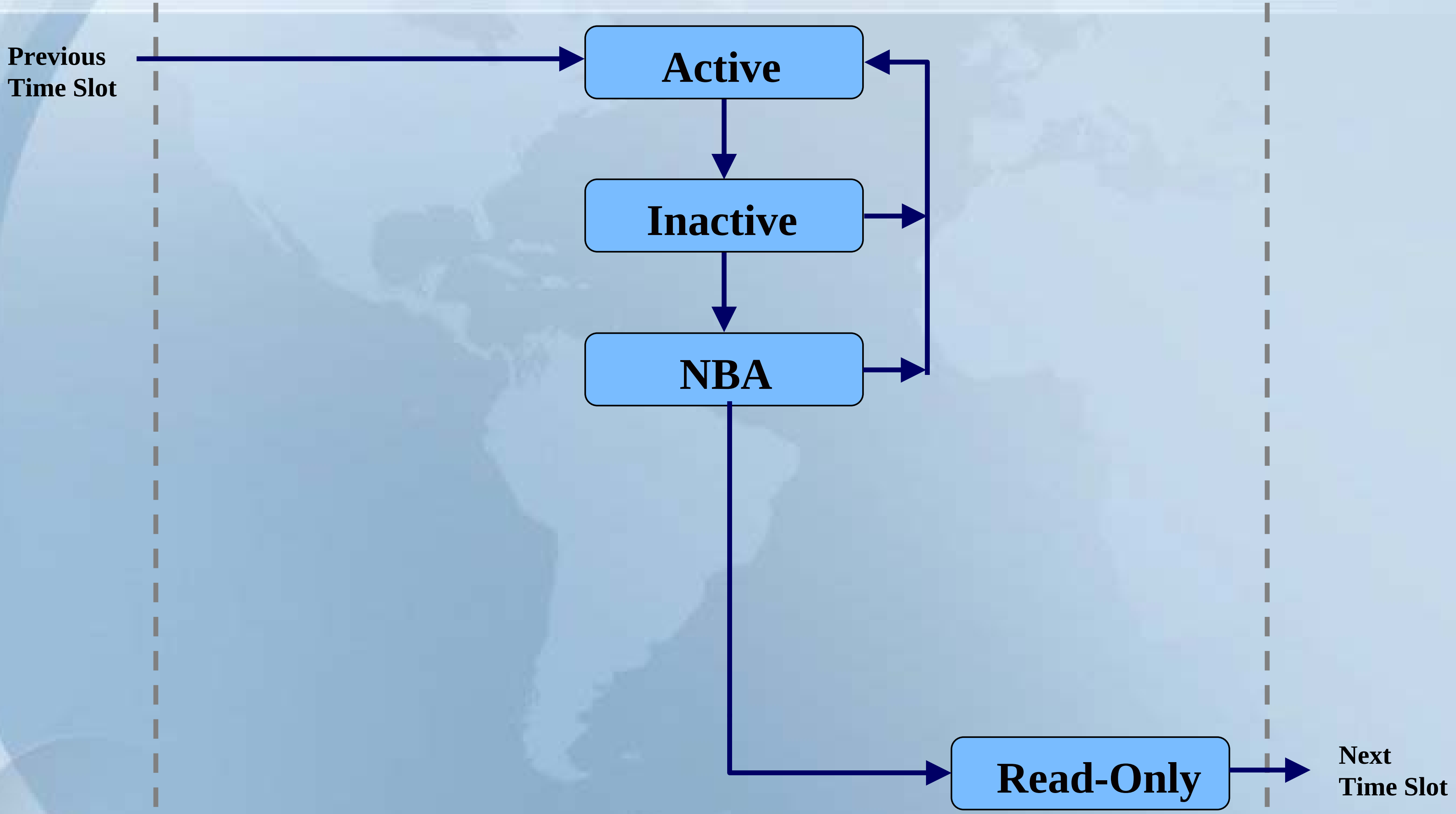
- Designate one clocking as default

```
default clocking modA.clkDomain;
```

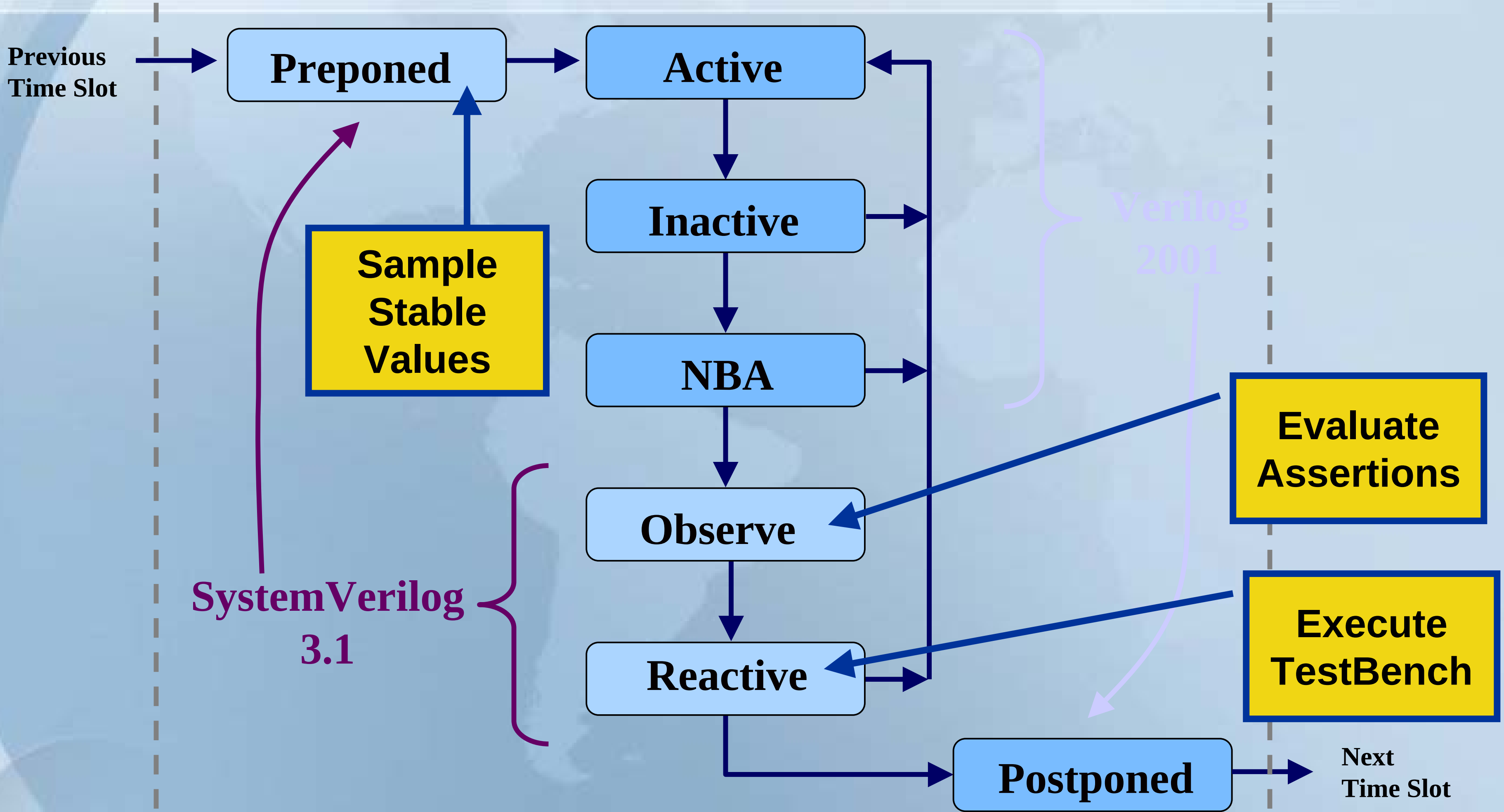
- One default per module, interface, program
- Cycle Delay Syntax:

```
## <integer_expression>
##5;  // wait 5 cycles
##1 bus.data <= 8'hz;// wait 1 (bus) cycle and
                     //   then drive data
##2; bus.data <= 2;// wait 2 default clocking
                   //   cycles, then drive data
bus.data <= ##2 r;// remember the value of r and
                  //   then drive data 2 (bus) cycles later
```

SystemVerilog Enhanced Scheduling



SystemVerilog Enhanced Scheduling



Enhanced Scheduling in SystemVerilog

- Avoids Race Conditions Between Testbench and Design
- Guarantees that Assertions and Testbench have Access to Stable Values
- Allows Testbench to React to Assertions (Pass or Fail)
- Ensures Common Semantics between Simulation, Formal Verification, Synthesis, HW Accelerators, Emulation, etc.

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
```

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
- Object – The actual house
  - An object is an instance of a class
- Properties – It has light switches
  - Variables specific to the class
- Methods – Turn on/off the lights
  - Tasks/functions specific to the class

Class Basics

- Class Definitions Contain Data and Methods
- Classes Are Instantiated Dynamically to Create *Objects*
  - *Static* members create a single element shared by all objects of a particular class type
- *Objects* Are Accessed Via *Handles*
  - Safe Pointers, Like Java
- Classes Can *Inherit* Properties and Methods From Other Classes
- Classes Can Be Parameterized

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
```

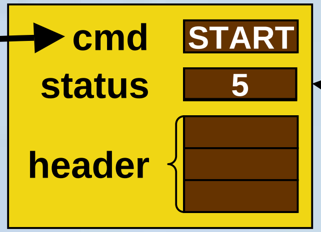
Note: Class declaration does not allocate any storage

Class Instantiation

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

```
Packet myPkt = new;
```

myPkt



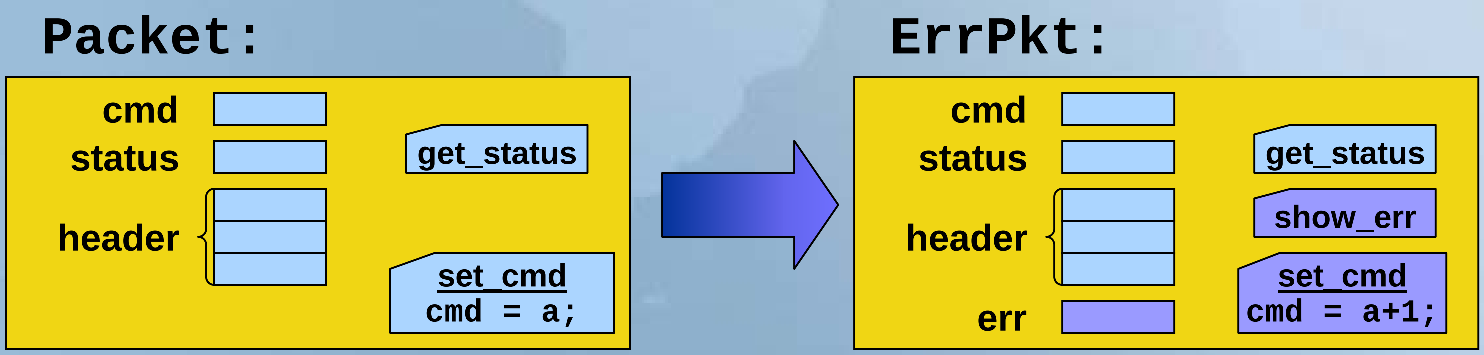

```
class Packet;
  ...
  function new();
    cmd = IDLE;
  endfunction
endclass
Packet myPkt = new;
```

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

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 Base Class

Class Hierarchy

- Class members can be hidden from external access
  - **`local`** members can only be referenced from within the class, *not from a subclass*
  - **`protected`** members can be referenced from within a subclass
- **`this`** pointer refers to current instance
- **`super`** pointer refers to parent class

```
class Base;
  local int i;
  int a,d;
  protected task set_i(input int i);
    this.i = i;
  endtask
  function new();...endfunction
endclass
```

```
class Sub extends Base;
  int a;
  function new();
    super.new();...
  endfunction
  task set_a(input int c);
    a = c; super.a = c+1;
    set_i(c); // inherited method
  endtask
endclass
```

Class Hierarchy

- Class members can be hidden from external access
  - **`local`** members can only be referenced from within the class
  - **`protected`** members can be referenced from within a subclass
- **`this`** pointer refers to current instance
- **`super`** pointer refers to parent class

```
class Base;
  local int i;
  int a,d;
  protected task set_i(input int i);
    this.i = i;
  endtask
  function new();...endfunction
endclass
```

```
class Sub extends Base;
  int a;
  function new();
    super.new();...
  endfunction
  task set_a(input int c);
    a = c; super.a = c+1;
    set_i(c);// inherited
  endtask
endclass
```

```
Sub S = new;
initial begin
  S.i = 4; // illegal – i is local to Base
  S.set_i(4);// illegal – set_i is protected
  S.a = 5; // legal – Base::a is hidden by Sub::a
  S.set_a(5);// legal – set_a is unprotected
  S.d = 3;// legal – d is inherited from Base
end
```

Parameterized Classes

- Allows Generic Class to be Instantiated as Objects of Different Types
  - Uses module-like parameter passing

```
class vector #(parameter int size = 1;);
  bit [size-1:0] a;
endclass
vector #(10) vten;  // object with vector of size 10
vector #(.size(2)) vtwo;  // object with vector of size 2
typedef vector#(4) Vfour;  // Class with vector of size 4
```

```
class stack #(parameter type T = int;);
  local T items[];
  task push( T a ); ... endtask
  task pop( ref T a ); ... endtask
endclass
stack i_s;  // default: a stack of int's
stack#(bit[1:10]) bs;  // a stack of 10-bit vector
stack#(real) rs;  // a stack of real numbers
stack#(Vfour) vs;  // stack of classes
```

Avoid Writing Similar Code More than Once

Working With Objects

- After calling new( ), values can be assigned to the object variables
- Each object created from the class has a different name and a separate memory space
- Data and subroutines are accessed through the object "handle" (like a pointer)
- Object Destruction/De-allocation
  - De-allocation of memory is taken care of automatically (like Java, unlike C++)
    - When an object is no longer being referenced, the memory used by this object is released
    - No destructors ☺
    - No memory leaks or unexpected side effects ☺

Constrained Random Simulation

<u>Test Scenarios</u>

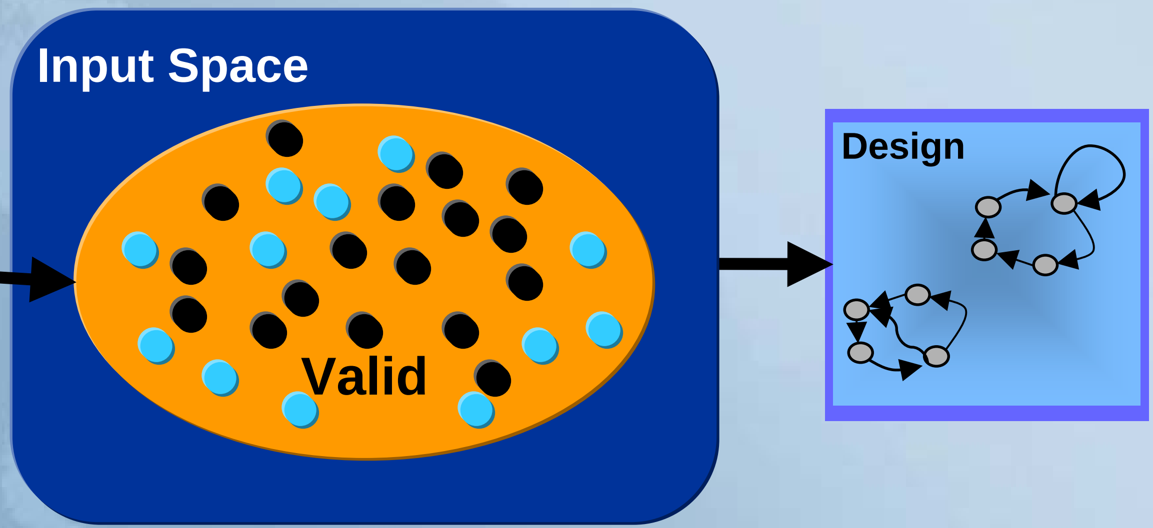
Constraints

Constraints

- **Valid Inputs Specified as Constraints**
  - **Declarative**

Constraint Solver

- **Find solutions**



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

Random Constraints

- Constraints are built onto the Class system
- Random variables use special modifier:
  - **`rand`** – random variable
  - **`randc`** – random cyclic variable
- Object is randomized by calling **`randomize( )`** method
  - Automatically available for classes with random variables.
- User-definable methods
  - **`pre_randomize()`**
  - **`post_randomize()`**

Basic Constraints

- Constraints are Declarative

```
class Bus;
rand bit[15:0] addr;
rand bit[31:0] data;
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");
```

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

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

SystemVerilog Testbench Language Summary

- Testbench Extensions
  - Useful for modeling environment
    - Classes
    - Random Constraints
    - Pass-by-Reference
    - Process Control
      - Interprocess communication
      - Synchronization
    - Extended data types
- Verification Extenstions
  - Enhanced Scheduling
  - Program Block
  - Clocking Domains
  - Assertions

Testbench Language Evolution

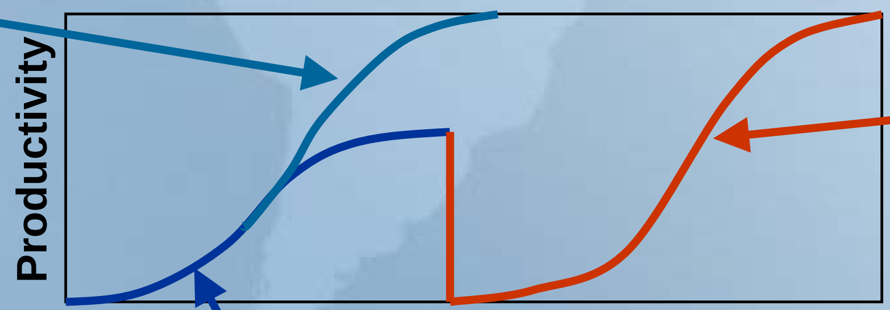
Advantages of Extending an Existing Language

SystemVerilog:

- Builds on Verilog
- Easy to Use/Debug
- All the Power in One Tool

Separate Verification Language:

- Have to Start Over From Scratch
- Separate tool = Bottleneck
- Different Language from Design



Basic Verification Using Verilog