System Verilog

For design and verification

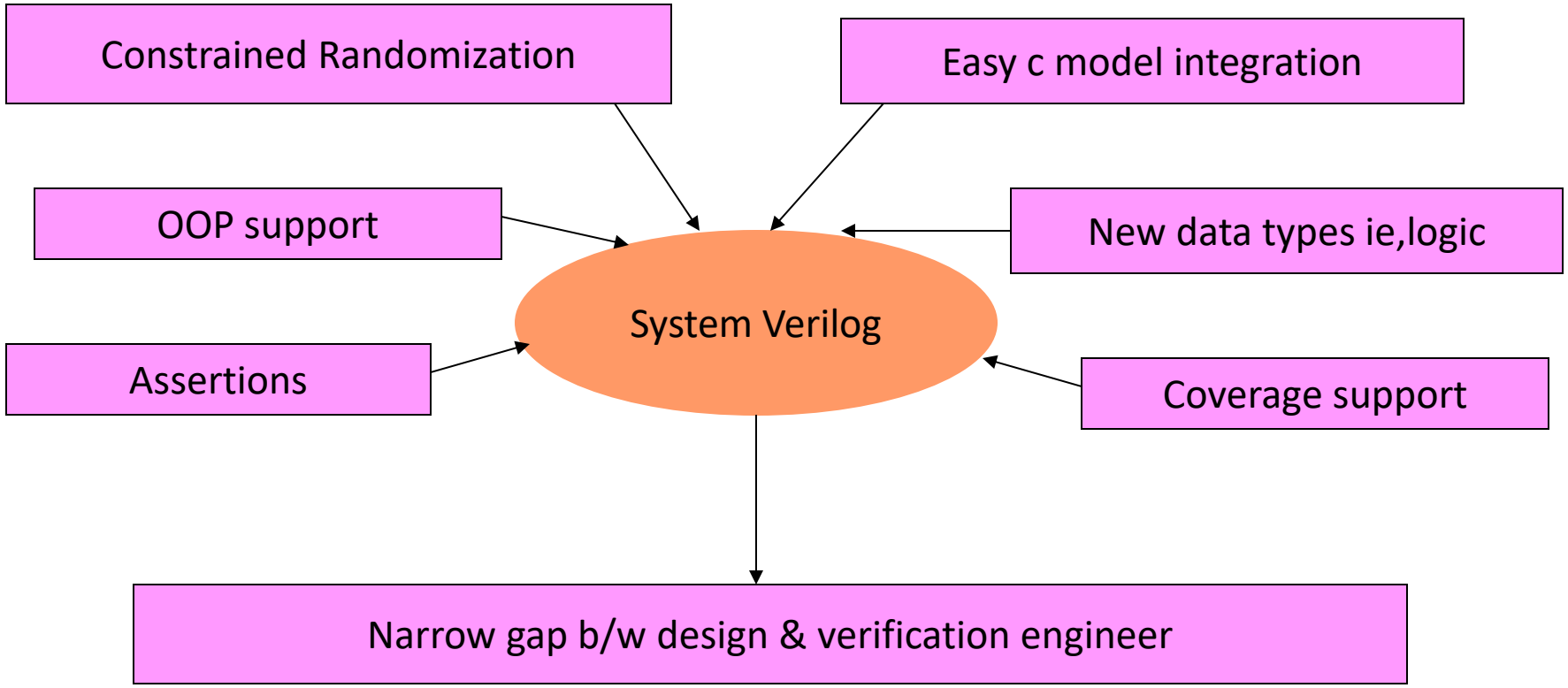
System Verilog: Origins, Overview, Need and Importance

- System Verilog unifies several proven hardware design and verification languages, in the form of extensions to the Verilog HDL. These extensions provide powerful new capabilities for modeling hardware at the RTL, system and architectural levels, along with a rich set of features for verifying model functionality.
- Hardware Description and Verification Language (HDVL) combines features of HDL's with features from C and C++
  - Borrows features from Verilog Superlog, VHDL, PSL, C, Vera,
  - Most of the verification functionality is based on the OpenVera language by Synopsys
  - Adopted as IEEE standard in 2005, most current version is IEEE standard 1800-2017
  - Can be used to describe hardware behavior but typically used for verification
  - System Verilog is an Object Oriented language that supports a single-inheritance model
  - Verification features allow for complex testbenching and random stimuli in simulation

Differences in SV and HDLs

	SystemVerilog	VHDL
RTL	C-style data types 7 control –enum, struct, typedef, ++ break return	Yes
	Packages	Yes
	Synthesis –friendly "concise" RTL notation	No
	Interfaces	No
Assertions	Systemverilog Assertions	Yes
Test Bench	Clocking Blocks	No
	Object Oriented Programming –Classes	No
	Constrained random stimulus generation	No
	Function Coverage	No
	Dynamic Processes, Dynamic arrays, queues, mailboxes, semaphores	No
DPI	Direct Programming Interface(DPI) –calling C from Systemverilog	Yes

Why SystemVerilog?



SystemVerilog Intent

Verilog

- Design entry
- Module level verification

System Verilog

- Module level design
- Gate level simulations
- System level verification
- Unified language to span almost the entire SoC design flow

Relaxed data type rules

Verilog

- ❑ Strict about usage of wire & reg data type
- ❑ Variable types are 4 state – 0,1,X,Z

System Verilog

- ❑ Logic data type can be used so no need to worry about reg & wire
- ❑ 2 state data type added – 0, 1 state
- ❑ 2 state variable can be used in test benches,where X,Z are not required
- ❑ 2 state variable in RTL model may enable simulators to be more efficient

Memory Management

Verilog

- Memories in verilog are static in nature

Example :-reg[7:0] X[0:127]; 128 bytes of memory

System Verilog

- Memories are dynamic in nature
- Allocated at runtime
- Better memory management ie,queues

Example:Logic[3:0] length[$]; an empty queue with an unbounded size of logic data type

Complexity

Verilog

- For complex designs large number of RTL code is required
- Increase in verification code to test these designs
- Extra time

System Verilog

- Less RTL & verification code
- Less code hence less no. of bugs
- Readable
- Higher level of abstraction due to algorithmic nature(inherited from C++)

Hardware specific procedures

Verilog

It uses the “always” procedure to represent

- ❑ Sequential logic
- ❑ Combinational logic
- ❑ Latched logic

System Verilog

It uses three new procedures

- ❑ always_ff - sequential logic
- ❑ always_comb - combinational logic
- ❑ always_latch - latched logic

Port connections

Verilog

- Ports are connected using either named instance or positional instance

System Verilog

- Ports are connected using

Design DUT(.*);which means connect all port to variables or nets with the same name as the ports

An Introduction to SystemVerilog Design

System Verilog Declaration Spaces

- WHAT WE ARE GOING TO SEE
- Packages definitions and importing definitions from packages
- $unit compilation declaration space
- Declarations in unnamed blocks
- Enhanced time unit definitions

- logic — a 1-bit 4-state variable, like the Verilog reg type; can be declared as any vector size
- enum — an enumerated net or variable with a labeled set of values
- typedef — a user-defined data type, constructed from built-in types or other user-defined types
- struct — a collection of variables that can be referred to individually or collectively

Declaration space

- In Verilog, declarations of variables, nets, tasks and functions must be declared within a module, between the **module...endmodule** keywords. The objects declared within a module are local to the module.
- Verilog also allows local variables to be defined in named blocks (formed with begin...end or fork...join), tasks and functions.
- Verilog does not have a place to make global declarations, such as global functions. A declaration that is used in multiple design blocks must be declared in each block. This not only requires redundant declarations, but it can also lead to errors if a declaration, such as a function, is changed in one design block, but not in another design block that is supposed to have the same function.
- SystemVerilog adds user-defined types, using typedef. It is often desirable to use the definition of user-defined types in multiple modules.

Introduction to System Verilog

SystemVerilog Declaration Spaces
Data types
- 4-state and 2-state data type
- String
- Event
- Type casting
- User defined date types (typedef)
- Enumerated data types (enum)

Data Declaration
Operators and Expression

Arrays
- Packed and unpacked arrays
- Array initialization at declaration
- Multiple dimensions
- Array Handling
- Dynamic Array
- Associative Array

Synthesis guidelines

Packages

- In Verilog, declarations of variables, nets, tasks and functions must be declared within a module, between the **module**...**endmodule** keywords.
- The objects declared within a module are local to the module. For modeling purposes, these objects should be referenced within the module in which they are declared. Verilog also allows hierarchical references to these objects from other modules for verification purposes, but these cross-module references do not represent hardware behavior, and are not synthesizable.
- Verilog does not have a place to make global declarations, such as global functions. A declaration that is used in multiple design blocks must be declared in each block.
- user-defined types, using typedef is often desirable to use the definition of user-defined types in multiple modules.

Packages

- The synthesizable constructs that a packages can contain are:
- **parameter** and **localparam** constant definitions
- **const** variable definitions
- **typedef** user-defined types
- Fully automatic **task** and **function** definitions
- **import** statements from other packages
- Operator overload definitions

- Packages can also contain
  - global variable declarations
  - static task definitions
  - static function definitions

Packages example

- Modules and interfaces can reference the definitions and declarations in a package four ways:
  - Direct reference using a scope resolution operator
  - Import specific package items into the module or interface
  - Wildcard import package items into the module or interface
  - Import package items into the $unit declaration space

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

- SystemVerilog adds a **::** "*scope resolution operator*"
- This operator allows directly referencing a package by the package name, and then selecting a specific definition or declaration within the package.
- The package name and package item name are separated by double colons (**::**).
- For example, a SystemVerilog module port can be defined as an instruction_t type, where instruction_t is defined in the package definitions

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  always_ff @(posedge clock) begin


  case (IW.opcode)
    definitions::ADD : result = IW.a + IW.b;
    definitions::SUB : result = IW.a - IW.b;
    definitions::MUL : result = definitions::
                                multiplier(IW.a, IW.b);
  endcase
 end
endmodule
```

Importing specific package items

- SV allows specific package items to be imported into a module, using an **import** statement. When a package definition or declaration is imported into a module or interface, that item becomes visible within the module or interface, as if it were a locally defined name within that module or interface. It is no longer necessary to explicitly reference the package name each time that package item is referenced.

- Importing a package definition or declaration can simplify the code within a module.

- Example is modified using import statements to make the enumerated type labels local names within the module. The case statement can then reference these names without having to explicitly name the package each time.

```
module ALU
(input definitions::instruction_t IW,
input logic clock,
output logic [31:0] result
);
import definitions::ADD;
import definitions::SUB;
import definitions::MUL;
import definitions::multiplier;
always_comb begin
        case (IW.opcode)
            ADD : result = IW.a + IW.b;
            SUB : result = IW.a - IW.b;
            MUL : result = multiplier(IW.a, IW.b);
        endcase
end
endmodule
```

Wildcard import of package items

- SystemVerilog allows package items to be imported using a wildcard, instead of naming specific package items. The wildcard token is an asterisk (*).
- **import** definitions::*; // wildcard import
- Definitions and declarations in the package that are not referenced are not imported.
- Ideal way is that the line
  ‘include "definitions.pkg“
  should be placed at the beginning of every design or testbench file that needs the definitions in the package.

```
module ALU
(input  definitions::instruction_t  IW,
 input  logic                       clock,
 output logic [31:0]                result
);
  import definitions::*;  // wildcard import

  always_comb begin
    case (IW.opcode)
      ADD : result = IW.a + IW.b;
      SUB : result = IW.a - IW.b;
      MUL : result = multiplier(IW.a, IW.b);
    endcase
  end
endmodule
```

- Time unit and precision declarations
- Variable declarations
- Net declarations
- Constant declarations
- User-defined data types, using **typedef**, **enum** or **class**
- Automatic Task and function definitions

```
/******************* External declarations *******************/
parameter VERSION = "1.2a";   // external constant

reg resetN = 1;               // external variable (active low)

typedef struct packed {       // external user-defined type
  reg [31:0] address;
  reg [31:0] data;
  reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);  // external function
  if (n <=1) return(1);
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return(log2);
endfunction
```

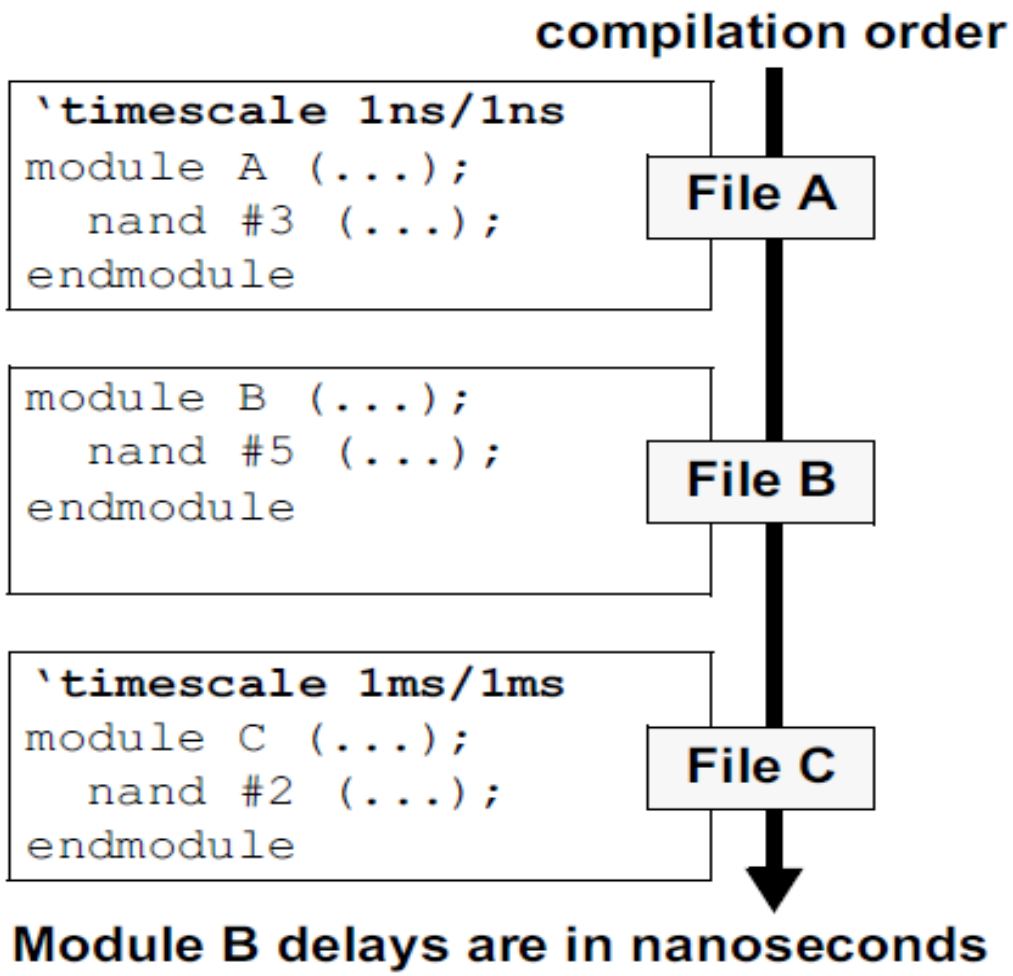
timescale

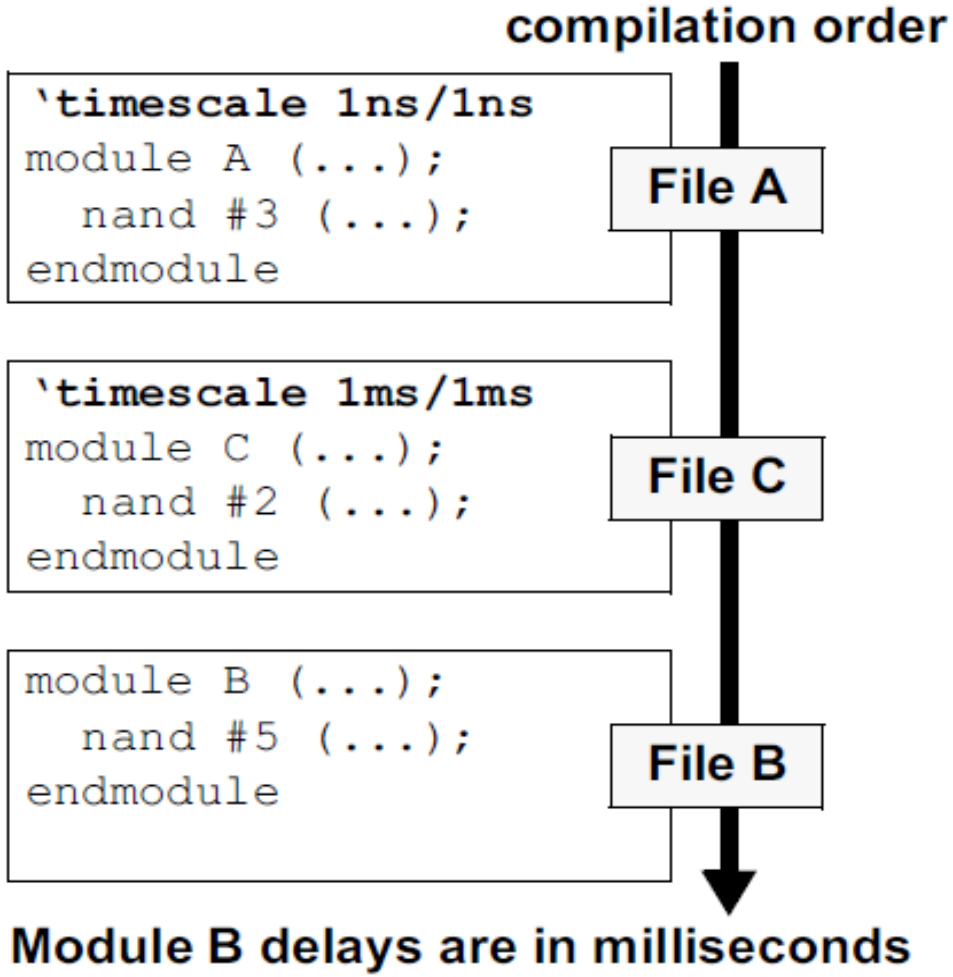
- **forever** #5 clock = ~clock;

- ‘**timescale** 1ns / 10ps the software tool is instructed to use time units of 1 nanosecond, and a precision of 10 picoseconds, which is 2 decimal places, relative to 1 nanosecond.

- **forever** #5**ns** clock = ~clock;

Time order


compilation order
`timescale 1ns/1ns
module A (...);
nand #3 (...);
endmodule
File A
module B (...);
nand #5 (...);
endmodule
File B
`timescale 1ms/1ms
module C (...);
nand #2 (...);
endmodule
File C
Module B delays are in nanoseconds


compilation order
`timescale 1ns/1ns
module A (...);
nand #3 (...);
endmodule
File A
`timescale 1ms/1ms
module C (...);
nand #2 (...);
endmodule
File C
module B (...);
nand #5 (...);
endmodule
File B
Module B delays are in milliseconds

Unit	Description
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

System Verilog Concepts

Data types :

Bit subs allowed

```
reg r;       // 4-state Verilog-2001
logic w;     // 4-valued logic, see below
bit b;       // 2-state bit 0 or 1
integer i;   // 4-state, 32-bits, signed Verilog-2001
byte b8;     // 8 bit signed integer
int i;       // 2-state, 32-bit signed integer
shortint s;  // 2-state, 16-bit signed integer
longint l;   // 2-state, 64-bit signed integer
```

Explicit 2-state variables allow compiler optimizations to improve performance

logic is has single driver (procedural assignments or a continuous assignment), can replace **reg** and single driver **wire**. (Equivalent to “std_ulogic” in VHDL)

Data types

- 4-state and 2-state data type
- The 4-state **logic** type and the 2-state **bit**, **byte**, **shortint**, **int**, and **longint** types are synthesizable. Synthesis compilers treat 2- state and 4-state types the same way.
- Relaxed usage of variables

```
module comparator (output logic  eq,
                   input  [63:0] a, b);
  always @(a, b)
    eq = (a==b);
endmodule
```

- Variables cannot be driven by multiple sources.
- Use variables for single-driver logic, and use nets for multidriver logic.

- Logic

```
logic resetN; // a 1-bit wide 4-state variable

logic [63:0] data; // a 64-bit wide variable

logic [0:7] array [0:255]; // an array of 8-bit
                              variables
```

a variable of the **logic** data type is identical to the Verilog **reg** type.

- **2-state variables**
  - **bit** — a 1-bit 2-state integer
  - **byte** — an 8-bit 2-state integer, similar to a C **char**
  - **shortint** — a 16-bit 2-state integer, similar to a C **short**
  - **int** — a 32-bit 2-state integer, similar to a C **int**
  - **longint** — a 64-bit 2-state integer, similar to a C **longlong**

Data types - Type casting

- SystemVerilog adds the ability to cast a value to a different type.
- With type casting, a value can be converted to a new type within an expression, without any assignment being made.
- **<type>'(<expression>)** — casts a value to any type, including user-defined types. **int'(**2.0 * 3.0**)**;
- **<size>'(<expression>)** — casts a value to any vector size. 16'(2);
- The expression to be cast will always be converted during run time, without any checking that the expression to be cast falls within the legal range of the type to which the value is cast.

- **Dynamic casting** Logic
- **$cast(** dest_var**,** source_exp **);**

```
int radius, area;
$cast(area, 3.154 * radius ** 2);
```

a variable of the **logic** data type is identical to the Verilog **reg** type.

- **2-state variables**

Data types – enum and typedef

- Using **typedef** to create user-defined types

```
module alu (...);

  typedef logic [3:0] nibble;

  nibble opA, opB;   // variables of the
                     // nibble type

  nibble [7:0] data; // a 32-bit vector made
                     // from 8 nibble types
  ...
endmodule
```

- A **typedef** definition can also be declared externally.

```
`ifdef TWO_STATE
  typedef bit dtype_t;
`else
  typedef logic dtype_t;
`endif
```

- Using **enum** to create enumerated types provide a means to declare an abstract variable that can have a specific list of valid values.

enum {WAIT, LOAD, STORE} State,NextState;

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_comb begin
  if (state != S3)
    next_state = states_t'(state + 1);
  else
    next_state = S1;
end
```

- The first label in the enumerated list is represented with a value of 0, the second label with a value of 1, the third with a value of 2, and so on.
- **enum** {A=1, B, C, X=24, Y, Z} list1;

Operators

+ - * /	arithmetic	~	bitwise negation
%	modulus division	&	bitwise and
++ --	increment, decrement	&~	bitwise nand
> >= < <=	relational	|~	bitwise nor
!	logical negation	|	bitwise inclusive or
&&	logical and	^	bitwise exclusive or
||	logical or	^~	bitwise exclusive nor
==	logical equality	{ }	concatenation
!=	logical inequality	&	unary and
===	case equality	~&	unary nand
!==	case inequality	|	unary or
==?	wildcard case equality	~|	unary nor
!=?	wildcard case inequality	^	unary exclusive
<<	logical shift left	~^	unary exclusive nor
>>	logical shift right	? :	conditional (ternary)
<<<	arithmetic shift left	inside	set membership
>>>	arithmetic shift right		

Assignment:

= += -= *= /= %= <<= >>= <<<= >>>= &= |= ^= ~&= ~|= ~^=

Some practical's ☺

- First program of displaying Hello World
- Add a few data types : Logic , reg
- Add a function to display a new computed value and check for 2 state, 4 state values
- Add a cast function
- Add a delay of different time unit

- COMMAND LINE

```
vlog  my_dut.sv my_test.sv
vsim –c  my_test
run –all
quit
```

```
module dff(clk,reset,din,dout);
    input clk,reset,din;
    output dout;
    logic dout;

    always@(posedge clk,negedge reset)
        if(!reset)
            dout <= 0;
        else
            dout <= din;
endmodule
```

- COMMAND LINE BATCH MODE

```
vlog  my_dut.sv my_test.sv
vsim –batch my_test –do “run –all; quit”
```

```
module my_test_dff (/*AUTOARG*/ ) ;
   reg clk;
   reg reset_n;
   reg din;
   wire dout;

   dff my_dut1 (clk,reset_n,din,dout);

   initial
     begin
        fork
           begin
              forever #5ns clk = ~clk;
           end
           begin
              reset_n = 0;
              #4 reset_n = 1;
           end
           begin
              din = 1;
              #5 din = 0;
              #5 din = 1;
              #5 din = 0;
           end
        join
     end // initial begin

endmodule // my test dff
```

Wildcard comparisons

a	b	a == b	a === b	a ==? b	a != b	a !== b	a !=? b
0000	0000	true	true	true	false	false	false
0000	0101	false	false	false	true	true	true
010Z	0101	unknown	false	unknown	unknown	false	unknown
010Z	010Z	unknown	true	true	unknown	false	false
010X	010Z	unknown	false	true	unknown	true	false
010X	010X	unknown	true	true	unknown	false	false

Session 3 and 4

Pankaj Badhe

Agenda

- Structures
- Unions
- Unpacked arrays
- Packed arrays
- Operations on arrays
- Array foreach loop
- Special system functions for working with arrays
- The $bits “sizeof” system function

Structures

- A structure is a convenient way of grouping several pieces of related information together.
- A structure is declared using the struct keyword.
- Structure members can be any variable type, including user-defined types, and any constant type.
- to assign a value to the opcode member of the preceding structure, the reference is:<structure_name>.<variable_name>

```
struct {
  int          a, b;      // 32-bit variables
  opcode_t opcode;        // user-defined type
  logic [23:0] address;   // 24-bit variable
  bit          error;     // 1-bit 2-state var.
} Instruction_Word;
```

```
Instruction_Word.address = 32'hF000001E;
```

```
int a;
logic [0:23] a;// vector
struct {  } Instruction_word;
Instruction_word_t IW = '{100, 3, 8'hFF,0};
```

```
typedef struct {  // structure definition
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instruction_word_t;

instruction_word_t IW;  // structure allocation
```

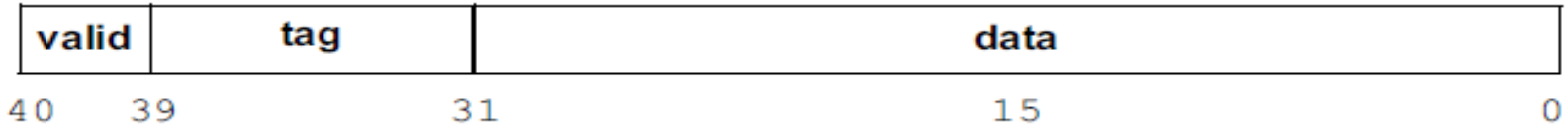
instruction_word_t IW = ’{100, 3, 8’hFF,0};
// structure initialization

Assigning values to structure members

```
typedef struct {
  logic [31:0] a, b;
  logic [ 7:0] opcode;
  logic [23:0] address;
} instr_t;

instr_t IW;

always @(posedge clock, negedge resetN)
  if (!resetN) begin
    IW.a = 100;  // reference structure member
    IW.b = 5;
    IW.opcode = 8'hFF;
    IW.address = 0;
  end
  else begin
    ...
  end
```

```
struct packed {
  logic        valid;
  logic [ 7:0] tag;
  logic [31:0] data;
} data_word;
```

Packed structures are stored as a vector




```
data_word.tag = 8'hf0;
data_word[39:32] = 8'hf0;   // same bits as tag
```

Passing structures as arguments to tasks and functions

```
module processor (...);
  ...
  typedef enum {ADD, SUB, MULT, DIV} opcode_t;

  typedef struct {  // typedef is local
    logic [31:0]  a, b;
    opcode_t      opcode;
    logic [23:0]  address;
    logic         error;
  } instruction_word_t;

  function alu (input instruction_word_t IW);
    ...
  endfunction
endmodule
```

Unions

```
union {
   int  i;
   int unsigned u;
} data;
...
data.i = -5;
$display("data is %d", data.i);

data.u = -5;
$display("now data is %d", data.u);
```

Although the declaration syntax is similar, a union is very different than a structure. A structure can store several values. It is a collection of variables under a single name. A union can only store one value. A typical application of unions is when a value might be represented as several different types, but only as one type at any specific moment in time.

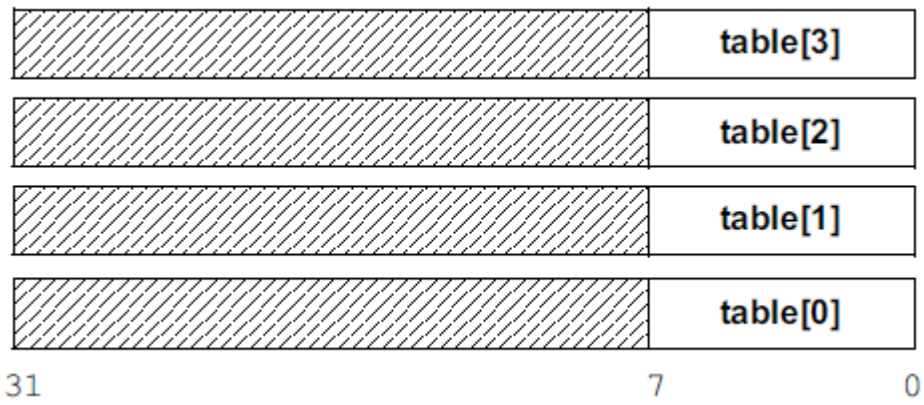
Using structures and unions

```
package definitions;

  typedef enum {ADD, SUB, MULT, DIV, SL, SR} opcode_t;

  typedef enum {UNSIGNED, SIGNED} operand_type_t;

  typedef union packed {
    logic        [31:0] u_data;
    logic signed [31:0] s_data;
  } data_t;

  typedef struct packed {
    opcode_t        opc;
    operand_type_t  op_type;
    data_t          op_a;
    data_t          op_b;
  } instr_t;

endpackage

import definitions::*;   // import package into $unit space
```

```
module alu
(input  instr_t IW,
 output data_t  alu_out);

  always @(IW) begin
    if (IW.op_type == SIGNED) begin
      case (IW.opc)
        ADD : alu_out.s_data = IW.op_a.s_data + IW.op_b.s_data;
        SUB : alu_out.s_data = IW.op_a.s_data - IW.op_b.s_data;
        MULT: alu_out.s_data = IW.op_a.s_data * IW.op_b.s_data;
        DIV : alu_out.s_data = IW.op_a.s_data / IW.op_b.s_data;
        SL  : alu_out.s_data = IW.op_a.s_data <<< 2;
        SR  : alu_out.s_data = IW.op_a.s_data >>> 2;
      endcase
    end
    else begin
      case (IW.opc)
        ADD : alu_out.u_data = IW.op_a.u_data + IW.op_b.u_data;
        SUB : alu_out.u_data = IW.op_a.u_data - IW.op_b.u_data;
        MULT: alu_out.u_data = IW.op_a.u_data * IW.op_b.u_data;
        DIV : alu_out.u_data = IW.op_a.u_data / IW.op_b.u_data;
        SL  : alu_out.u_data = IW.op_a.u_data << 2;
        SR  : alu_out.u_data = IW.op_a.u_data >> 2;

      endcase
    end
  end
endmodule
```

UnPacked arrays

- **wire** [7:0] table [3:0];



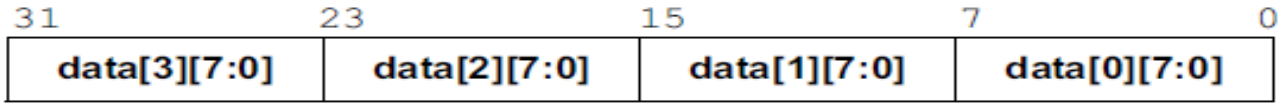

```
bit [63:0] d_array [1:128]; // array of vectors
shortreal cosines [0:89];  // array of floats
typedef enum {Mo, Tu, We, Th, Fr, Sa, Su} Week;
Week Year [1:52];  // array of Week types
```

```
int  a1 [7:0][1023:0];  // unpacked array
int  a2 [1:8][1:1024];  // unpacked array

a2 = a1;        // copy an entire array

a2[3] = a1[0];  // copy a slice of an array
```

- Even though the array dimensions are not numbered the same, the size and layout of each array is the same.
- **int** array [20]; // a C array with addresses from 0 to 19
- **logic** [31:0] data [1024];
- **logic** [31:0] data [0:1023];

Packed arrays

- **wire** [3:0] select; // 4-bit "packed array"
- **reg** [63:0] data; // 64-bit "packed array“
- **logic** [3:0][7:0] data; // 2-D packed array



Referencing packed arrays

A packed array can be referenced as a whole, as bit-selects, or as part-selects. Multidimensional packed arrays can also be referenced in slices. A slice is one or more contiguous dimensions of an array.

```
logic [3:0][7:0] data;  // 2-D packed array


wire [31:0] out = data;            // whole array
wire sign = data[3][7];            // bit-select
wire [3:0] nib = data [0][3:0];    // part-select
byte high_byte;
assign high_byte = data[3];        // 8-bit slice
logic [15:0] word;
assign word = data[1:0];           // 2 slices
```

Operations on packed arrays

Because packed arrays are stored as vectors, any legal operation that can be performed on a Verilog vector can also be performed on packed arrays. This includes being able to do bit-selects and part-selects from the packed array, concatenation operations, math operations, relational operations, bit-wise operations, and logical operations.

```
logic [3:0][15:0] a, b, result; // packed arrays
...
result = (a << 1) + b;
```

Use unpacked arrays to model:

- Arrays of **byte**, **int**, **integer**, **real**, unpacked structures, unpacked unions, and other types that are not bit-wise types
- Arrays where typically one element at a time is accessed, such as with RAMs and ROMs

```
module ROM (...);
  byte mem [0:4095]; // unpacked array of bytes
  assign data = select? mem[address]: 'z;
  ...
```

Use packed arrays to model:

- Vectors made up of 1-bit types (the same as in Verilog)
- Vectors where it is useful to access sub-fields of the vector

```
logic [39:0][15:0] packet; // 40 16-bit words

packet = input_stream;  // assign to all words
data = packet[24];      // select 1 16-bit word
tag = packet[3][7:0];   // select part of 1 word
```

Packed array initialization

Packed arrays can be initialized at declaration using a simple assignment, like vectors in Verilog. The assignment can be a constant
value, a concatenation of constant values or a replication of constant values.

```
logic [3:0][7:0] a = 32'h0;          // vector assignment
logic [3:0][7:0] b = {16'hz,16'h0}; // concatenate operator
logic [3:0][7:0] c = {16{2'b01}};   // replicate operator
```

Unpacked array initialization

- Unpacked arrays can be initialized at declaration, using a list of values enclosed between **'{** and **}** braces for each array dimension.
- The **'{ }** list and **'{**n**{ }}** replicated list operators are not the same as the Verilog **{ }** concatenate and **{**n**{ }}** replicate operators.

```
int e [0:1][0:3] = '{ 2{7,3,0,5} };
// e[0][0] = 7
// e[0][1] = 3
// e[0][2] = 0
// e[0][3] = 5

// e[1][0] = 7
// e[1][1] = 3
// e[1][2] = 0
// e[1][3] = 5
```

Multiple dimensions

logic [31:0] a [2:0][9:0];

logic [0:31] b [1:3][1:10];

a = b; // assign unpacked array to unpacked array

- **Passing arrays through ports and to tasks and functions**

```
module CPU (...);
  ...
  logic [7:0] lookup_table [0:255];

  lookup i1 (.LUT(lookup_table));
  ...
endmodule

module lookup (output logic [7:0] LUT [0:255]);
  ...
  initial load(LUT);  //task call

  task load (inout logic [7:0] t [0:255]);
    ...
```

- **Arrays of structures and unions**
- **Arrays in structures and unions**

- Arrays and assignments involving arrays are synthesizable

- FOREACH array looping construct

```
int sum [1:8] [1:3];

foreach ( sum[i,j] )
  sum[i][j] = i + j;    // initialize array
```

Array querying system functions

- **$dimensions(**array_name**)**

• Returns the number of dimensions in the array (returns 0 if the object is not an array)

- **$left(**array_name, dimension**)**

• Returns the most-significant bit (msb) number of the specified dimension. Dimensions begin with the number 1, starting from the left-most unpacked dimension.

- **$right(**array_name, dimension**)**
- **$size(**array_name, dimension**)**

• Returns the total number of elements in the specified dimension (same as $high - $low + 1).

- **$bits(**expression**)** returns how many bits are represented by any expression.

Dynamic arrays, associative arrays and strings

- SystemVerilog also adds dynamic array types to Verilog:
  - Dynamic arrays
  - Associative arrays
  - Strings (character arrays)
- These special array types are not synthesizable.

Session 5 and 6

Pankaj Badhe

Procedural Statements

- Initial Block
- Final Block
- Always Block

- Loops
  - do_while
  - while_do
  - for
  - foreach

Final Block

- Final block
  - Executesattheendofsimulation
  - Itcannothavedelays
  - Usedinverificationtoprintsimulationresults, suchaserrorreport,codecoveragereports

Relaxation of type rules

- a variable can receive a value in any one of the following ways, but no more than one of the following ways:
  - Be assigned a value from any number of **initial** or **always** procedural blocks (the same rule as in Verilog).
  - Be assigned a value from a single **always_comb**, **always_ff** or **always_latch** procedural block.
  - Be assigned a value from a single continuous assignment statement.
  - Receive a value from a single module or primitive output or inout port.

- Three new types of procedural blocks:

  always_comb, **always_latch** and **always_ff**.
- These new procedural blocks have the restriction that a variable can only be assigned from one procedural block. This further enforces the checking that a signal declared as a variable only has a single source.
- Only nets can have multiple sources, such as multiple continuous assignments and/or connections to multiple output ports of module or primitive instances.
- Therefore, a signal in a design such as a data bus or address bus that can be driven from several devices should be declared as a Verilog net type, such as **wire**.
- Bi-directional module ports, which can be used as both an input and an output, must also be declared as a net type.

- Combinational logic procedural blocks
- Latched logic procedural blocks
- Sequential logic procedural blocks
- Task and function enhancements

Combinational logic

- The **always** keyword must be followed by an edge-sensitive event control (the @ token).
- The sensitivity list of the event control cannot contain **posedge** or **negedge** qualifiers.
- The sensitivity list should include all inputs to the procedural block. Inputs are any signal read by the procedural block, where that signal receives its value from outside the procedural block.
- The procedural block cannot contain any other event controls.
- All variables written to by the procedural block must be updated for all possible input conditions.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

Latched logic

- The **always** keyword must be followed by an edge-sensitive event control (the @ token).
- The sensitivity list of the event control cannot contain **posedge**
- or **negedge** qualifiers.
- The sensitivity list should include all inputs to the procedural block. Inputs are any signal read by the procedural block, where that signal receives its value from outside the procedural block.
- The procedural block cannot contain any other event controls.
- At least one variable written to by the procedural block must *not* be updated for some input conditions.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

sequential logic

- The **always** keyword must be followed by an edge-sensitive event control (the @ token).
- All signals in the event control sensitivity list must be qualified with **posedge** or **negedge** qualifiers.
- The procedural block cannot contain any other event controls.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

Combinational logic procedural blocks

- The **always_comb** procedural block is used to indicate the intent to model combinational logic.

```
always_comb
    if (!mode)
      y = a + b;
    else
      y = a - b;
```

- It is not necessary to specify a sensitivity list with **always_comb**.
- always_comb infers its sensitivity list
- shared variables are prohibited
- **always_comb** procedural block will automatically trigger once at simulation time zero, after all **initial** and **always** procedural blocks have been activated.

```
always @* begin
  a1 = data << 1;
  b1 = decode();
  ...
end
```

Infers @(data)

```
always_comb begin
  a2 = data << 1;
  b2 = decode();
  ...
end

function decode;  // function with no inputs
  begin
    case (sel)
      2'b01:   decode = d | e;
      2'b10:   decode = d & e;
      default: decode = c;
    endcase
  end
endfunction
```

Infers @(data, sel, c, d, e)

Latched logic procedural blocks

always_latch infers its sensitivity list, just like **always_comb**.

```
always_latch
  if (enable) q <= d;
```

```
always_latch begin  // latch the ready input
  if (!resetN)
    enable <= 0;
  else if (ready)
    enable <= 1;
  else if (overflow)
    enable <= 0;
end
```

Sequential logic procedural blocks

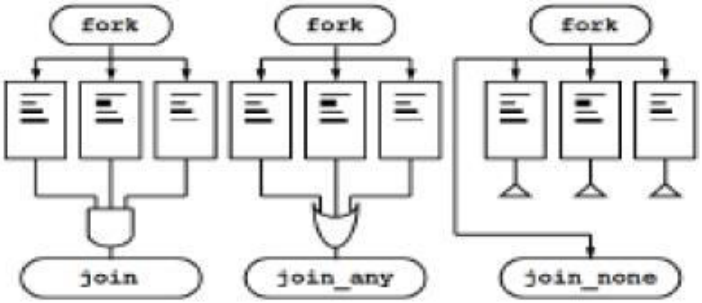
- The **always_ff** procedural block requires that every signal in the sensitivity list must be qualified with either **posedge** or **negedge**.
- This is a synthesis requirement for sequential logic sensitivity list.

The **always_ff** specialized procedural block indicates that the designer's intent is to model synthesizable sequential logic behavior.

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

Fork-join and its variant

```
fork
   statement1;
   statement2;
   statement3;
join | join_any | join_none
statement4;
```



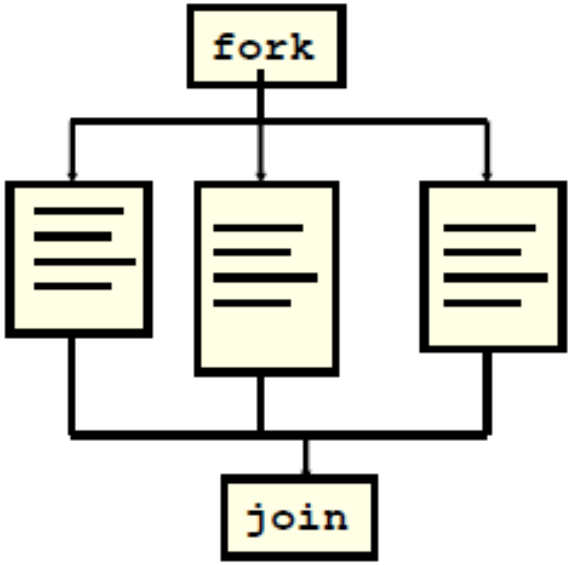
join - **child threads execute and all child threads must complete before statement4 is executed**

join_any - **child threads execute and one child thread must complete before statement4 is executed**

join_none - **child threads are queued,statement4 executes. Child threads not executed until parent thread encounters a blocking statement or completes**

```
Initial /always begin
        call to task 1;
        fork
                call to task 2; *
                call to task 3; // very long task WAITING *****
                begin
                        call to task 4; //followed by **
                        call to task 5;                **
                end
        join_none
        call to task 6; // display
end
```

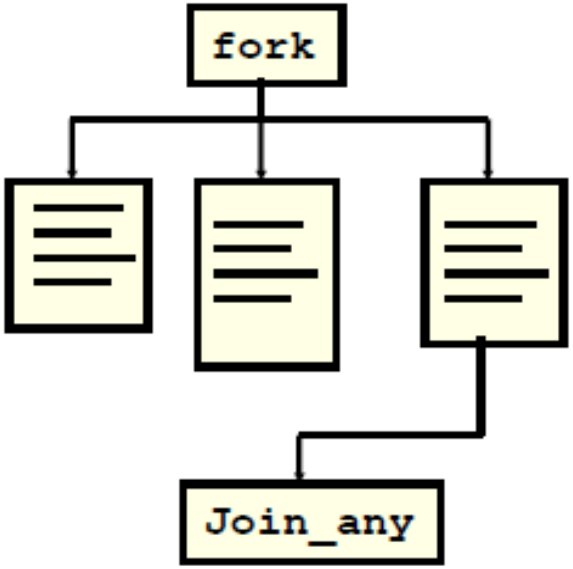
Fork/join




```
Initial
Begin
  Clk =0;
  #5
  Fork
    #5 a  = 0;
    #10 b = 0;
  Join
  Clk= 1;
end
```

Clk becomes 1 at t=15

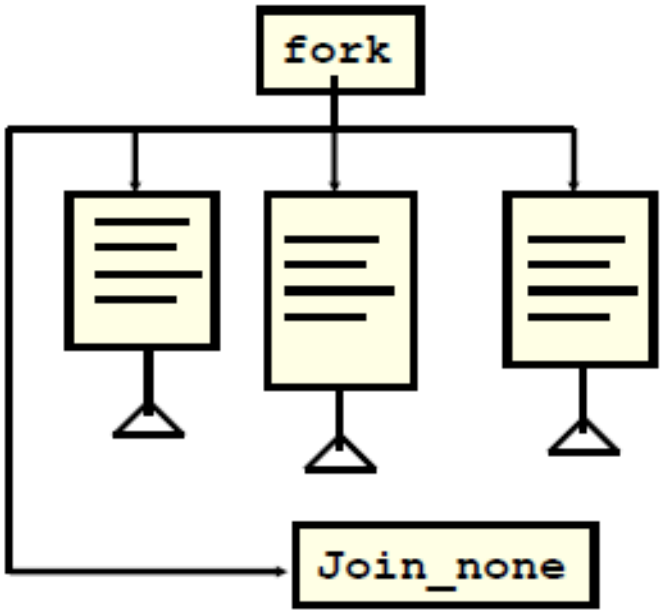
Fork/join_any




```
Initial
Begin
  Clk =0;
  #5
  Fork
    #5 a  = 0;
    #10 b = 0;
  Join_any
  Clk= 1;
end
```

Clk becomes 1 at t=10

Fork/join_none




```
Initial
Begin
  Clk =0;
  #5
  Fork
    #5 a  = 0;
    #10 b = 0;
  Join_none
  Clk= 1;
end
```

Clk becomes 1 at t=5

Sequential flow control

- **Conditional:**
  - `if (x==7) a=7; else a=8;`
  - `a = (x == y) ? 7 : 8;`
  - `assert (true condition);`
  - `case(expr) 0: …; 1: …; default: …; endcase`
- **Loops:**
  - `repeat(expr) begin … end`
  - `for(expr; expr; expr) begin … end`
  - `foreach(array[index]) begin … end`
  - `while(expr) begin … end`
  - `do begin … end while (expr);`
  - `break` to terminate loop
  - `continue` to terminate current loop iteration

Tasks and functions

- **Tasks can block**
- **Functions can not block**

Pass by reference

```
task print_sum(ref int a[], input int start=0);
  automatic int sum = 0;
  for (int j=start; j<a.size(); j++)
    sum += a[j];
  $display("Sum of array is %0d", sum);
endtask
…
print_sum(my_array);
```

Default value

task does not return value

Subroutines in program block default to static can be made automatic
Subroutines in class default to automatic

```
function automatic int fctorial(int n);
  static int shared_value = 0;
  if (n < 2) return(1);
  else return(n * fctorial(n-1));
endfunction
…
result = factorial(my_value);
```

Pass by value

function returns value

Subroutine variables default to subroutine scope and lifetime. Can be made automatic or static

Arguments to tasks

- **Type and direction are both sticky**
  - Any following arguments default to that type and direction

direction	effect
input	copy value in at beginning - default
output	copy value out at end
inout	copy in at beginning and copy out at return
ref	pass by reference, makes argument variable the same as the calling variable. Changes to argument variable will change calling variable immediately
const ref	pass by reference but read only. Saves time and memory for passing arrays to tasks & functions

Default dir is input, default type is logic

a, b: input logic
u, v: output bit [15:0]

Read-only pass via reference

```
task T3(a, b, output bit [15:0] u, v, const ref byte c[]);
```

Tasks and functions

- **`begin`...`end`** blocks are optional
- If you have a SystemVerilog task that does not consume time, you should make it a **`void function`** that is a function that does not return a value.
- C Style arguments: **`task mytask1 (output logic [31:0] x, input logic y);`**
- Default Argument Values :

```
function void print_sum (ref   int a[],
                         input int start = 0,
                         input int end = -1);
  int sum = 0, last;
  if (last == -1 || last > a.size)
    last = a.size;
  for (int i=start; i<last; i++) begin
    sum += a[i];
  $display("The sum of the arrays is ", sum);
endtask
```

```
print_sum(a);       // Sum a[0:size-1] - default
print_sum(a, 2, 5); // Sum a[2:5]
print_sum(a, 1);    // Start at 1
print_sum(a,, 3);   // Sum a[0:3]
print_sum();        // error: a has no default
```

- Returning from a Routine

```
task load_array(int len, ref int array[]);
  if (len <= 0) begin
    $display("Bad len");
    return;
  end

  // Code for the rest of the task
  ...
endtask
```

A **return** statement can simplify your functions.

- Named task and function ends

```
function int add_and_inc (int a, b);
  return a + b + 1;
endfunction : add_and_inc
```

- Empty tasks and functions

```
task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );
  static int error_count;
  ...
endtask: check_results
```

System Verilog Concepts

DPI(Direct Programming interface)

- DPI's are used to call C, C++, System C functions
- System verilog has a built in C interface
- Simple to used as compared to PLI's
- Values can be passed directly

System Verilog Concepts

DPI(Direct Programming interface)

- Imported functions
  - System verilog calls the C functions
- Exported functions
  - C calls the system verilog function
- Both sides of DPI are fully independent
  - System verilog does not analyze the C-code
  - C complier does not have to analyze the system verilog code

Session 7 and 8

Pankaj Badhe

Modeling state machines with enumerated types

- Enumerated types provide a means for defining a variable that has a restricted set of legal values. The values are represented with labels instead of digital logic values.
- Enumerated types allow modeling at a higher level of abstraction, and yet still represent accurate, synthesizable, hardware behavior.
- A simple finite state machine (FSM), using a typical three-procedural block modeling style:
- One procedural block for incrementing the state machine,
- one procedural block to determine the next state, and
- one procedural block to set the state machine output values.
- The example illustrates a simple traffic light controller.
- The three possible states are represented as enumerated type variables for the current state and the next state of the state machine.

A finite state machine modeled with enumerated types (poor style)

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {RED, GREEN, YELLOW} State, Next;  // using enum defaults

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;
```

```
  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)                Next = GREEN;
      GREEN:  if (green_downcnt == 0)    Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0)   Next = RED;
    endcase
  end: set_next_state

  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Representing state encoding with enumerated types

- SystemVerilog also allows the base type of an enumerated variable to be defined. This allows a 4-state type, such as **logic**, to be used as a base type, which can more accurately represent hardware behavior in RTL simulations.
- The logic value of each label in an enumerated type list can be specified. This allows explicitly representing one-hot, one-cold, Gray code, or any other type of state sequence encoding desired.
- The enumerated label values that represent the state sequencing are explicitly specified in the RTL model. Synthesis compilers will retain these values in the gate-level implementation.

Specifying one-hot encoding with enumerated types

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum logic [2:0] {RED    = 3'b001, // explicit enum definition
                    GREEN  = 3'b010,
                    YELLOW = 3'b100} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt  == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state
```

```
  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Pattern detection FSM

- Pattern to be detected “11010”
- Come up with State machine diagram
- Come up with the code of FSM

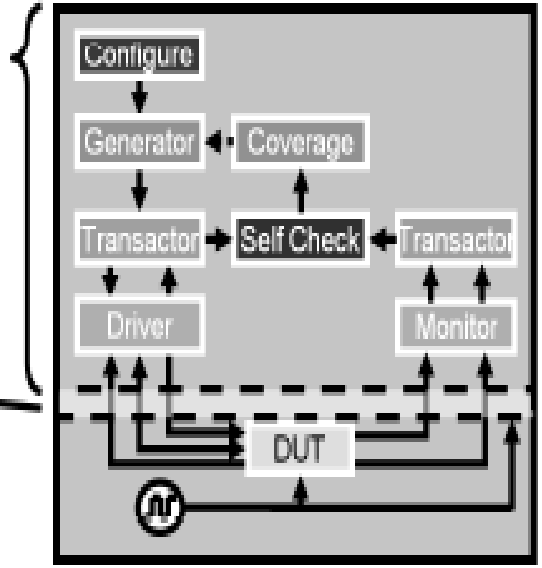
- Modify the above code for pattern “11011”
- Make the FSM overlapping

- Inputs : clk, rst, single bit detect incoming stream of bits
- Output: Pattern_detected / display

SystemVerilog Interfaces

- Interface declarations
- Connecting interfaces to module ports
- Differences between interfaces and modules
- Interface ports and directions
- Tasks and functions in interfaces
- Using interface methods
- Procedural blocks in interfaces
- Parameterized interfaces

- **SystemVerilog introduces two new design units**
  - The Program block
    - Is where you develop testbench code
    - Is entry point for testbench execution
  - The Interface
    - Is mechanism to connect testbench to DUT
    - Is a named bundle of wires
    - Can be passed just like a port in a port list



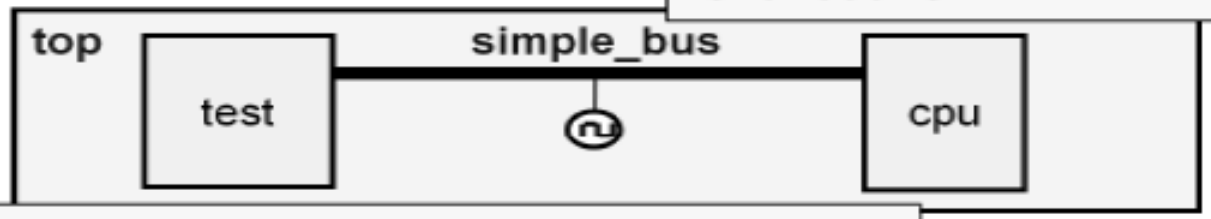
- **SystemVerilog programs use**
  - Object Oriented Programming (OOP)

- **An interface encapsulates the communication between DUT and Testbench including**
  - Connectivity (signals) – named bundle of wires
    - One or more bundles to connect
    - Can be reused for different tests and devices
  - Directional information (modports)
  - Timing (clocking blocks)
  - Functionality (routines, assertions, initial/always blocks)
- **Solves many problems with traditional connections**
  - Port lists for the connections are compact
  - No missed connections
  - Easy to add new connections

- The RTL code is connected with bundled signals

```
program automatic test(simple_bus sb);
...
endprogram
```

```
module cpu(simple_bus sb);
...
endmodule
```




```
interface simple_bus(input bit clk);
  logic req, gnt;
  logic [7:0] addr;
  wire  [7:0] data;
  logic [1:0] mode;
  logic start, rdy;
endinterface
```

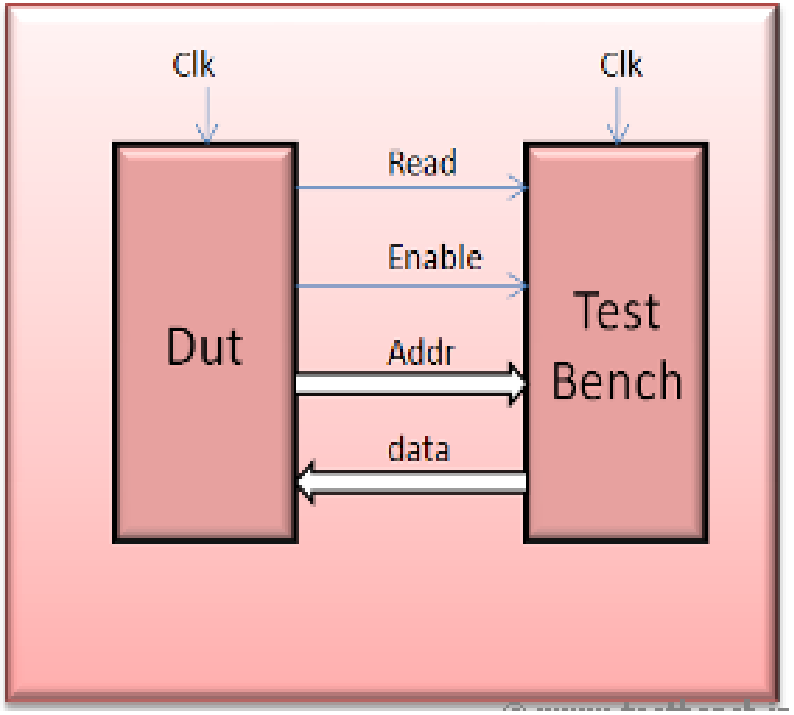
```
module top;
  logic clk = 0;
  always #10 clk = !clk;
  simple_bus sb(clk);
  test t1(sb);
  cpu c1(sb);
endmodule
```

Let us see a verilog example:

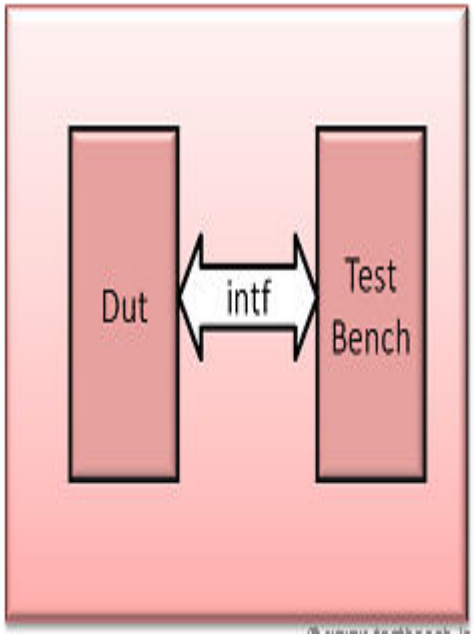
```
module Dut (input clk, read, enable,
                    Input [7:0] addr,
                    output [7:0] data);
        ....
        assign data = temp1 ? temp2 : temp3 ;
        always @(posedge clk)
        ....
endmodule

module Testbench(input clk,
                    Output read, enable,
                    output [7:0] addr,
                    input [7:0] data );
endmodule
```

Integrating the above two modules in top module.

```
module top();
        reg clk;
        wire read, enable;
        wire [7:0] addr;
        wire [7:0] data;

  Dut D (clk,read,enable,Addr,data);


Testbench TB(clk,read,enable,Addr,data);

endmodule
```



```
interface intf #(parameter BW = 8)(input clk);
logic read, enable;
logic [BW -1 :0] addr,data;
endinterface :intf
```




```
module Dut (intf dut_if); // declaring the interface

always @(posedge dut_if.clk)
if(dut_if.read) // sampling the signal
$display(" Read is asserted");

endmodule

module Testbench(intf tb_if);

initial
begin
tb_if.read = 0;
repeat(3) #20 tb_if.read = ~tb_if.read;// driving a signal
$finish;
end

endmodule
```

```
module top();
bit clk;

initial
forever #5 clk = ~clk;

intf bus_if(clk); // interface instantiation
Dut d(bus_if); // use interface for connecting D and TB
Testbench TB (bus_if);

endmodule
```

Modports

- In the above example, we did not mention the direction of signals.
- The direction of the clk signal in input for both the Dut and Testbench modules.
- But for the rest of the signals, the direction is not same.
- To specify the direction of the signal w.r.t module which uses interface instead of port list, modports are used.
- Modport restrict interface access within a module based on the direction declared.
- Directions of signals are specified as seen from the module.
- In the modeport list, only signal names are used.
- Let us see the modport usage with the previous example. 2 mod port definitions are needed, one for DUT and other for TestBench.

```
interface intf (input clk);
logic read, enable,
logic [7:0] addr,data;

modport dut (input read,enable,addr,output data);
modport tb (output read,enable,addr,input data);
endinterface :intf
```

```
module Dut (intf.dut dut_if); // declaring the interface with modport
....
assign dut_if.data = temp1 ? temp2 : temp3 ; // using the signal in interface
always @(posedge intf.clk)
....
endmodule

module Testbench(intf.tb tb_if);
.....
.....
endmodule

1 module top();
2 logic clk;
3 intf bus_if(clk); // interface instantiation
4 Dut d(bus_if); // Pass the interface
5 Testbench TB (Bus_if); // Pass the interface
6 endmodule
```

Methods In Interface

Interfaces can include task and function definitions. This allows a more abstract level of modeling.

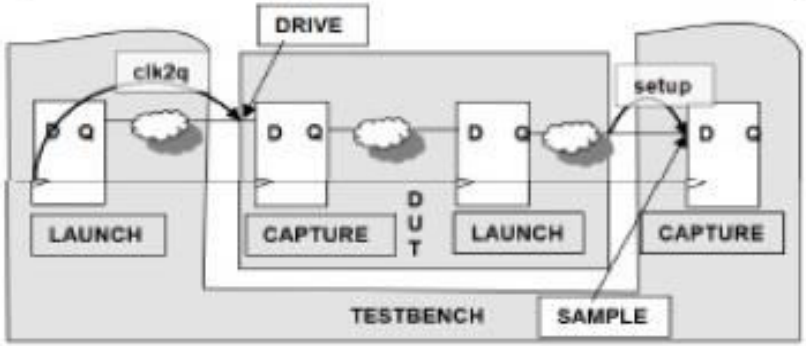
```
interface intf (input clk);
logic read, enable,
logic [7:0] addr,data;

task masterRead(input logic [7:0] raddr); // masterRead method
...
endtask: masterRead

task slaveRead; // slaveRead method
...
endtask: slaveRead

endinterface :intf
```

Clocking blocks

- **Just for testbench**
  - Emulates the Launch and Capture Flops at IO of DUT.
- **Creates explicit synchronous timing domains**
  - All signals driven or sampled at clocking event - by default all interface signals are asynchronous
  - Interaction between testbench and DUT ideally happens only at clock edges
- **Provides direction and signal *bundle that can be passed* through a modport**
  - Outputs cannot be sampled
  - Input signals cannot be driven

setup Clock to q

```
clocking cb @(posedge clk);
 default input #1ns output #1ns;
 output reset_n;
 output din;
 output frame_n;
 output valid_n;
 input  dout;
 input  busy_n;
 input  valido_n;
 input  frameo_n;
endclocking: cb
```



Session 9

Pankaj Badhe

Data types

Type	2-state / 4-state	Signed / Unsigned	Number of bits	SystemVerilog/ Verilog
shortint	2	Signed	16	SystemVerilog
int	2	Signed	32	SystemVerilog
longint	2	Signed	64	SystemVerilog
byte	2	Signed	8	SystemVerilog
bit	2	Unsigned		SystemVerilog
logic	4	Unsigned		SystemVerilog
reg	4	Unsigned		Verilog
integer	4	Signed	>= 32	Verilog
real				Verilog
shortreal				SystemVerilog
realtime				Verilog
time	4	Unsigned	64	Verilog

Signed And Unsigned :

- Integer types use integer arithmetic and can be signed or unsigned.
- The data types byte, shortint, int, integer, and longint default to signed.
- The data types bit, reg, and logic default to unsigned, as do arrays of these types.

- The void data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value.

void = function_call();

STRINGS

- SystemVerilog adds new keyword "string" which is used to declare string data types

  string myName = “SYSTEM VERILOG VERIFICATION";

- SystemVerilog also includes a number of special methods to work with strings.

01. str.len() returns the length of the string, i.e., the number of characters in the string.
02. str.putc(i, c) replaces the **i^{th}** character in str with the given integral value.
03. str.getc(i) returns the ASCII code of the **i^{th}** character in str.
04. str.toupper() returns a string with characters in str converted to uppercase.
05. str.tolower() returns a string with characters in str converted to lowercase.
06. str.compare(s) compares str and s, and return value. This comparison is case sensitive.
07. str.icompare(s) compares str and s, and return value .This comparison is case insensitive.
08. str.substr(i, j) returns a new string that is a substring formed by index i through j of str.
09. str.atoi() returns the integer corresponding to the ASCII decimal representation in str.
10. str.atoreal() returns the real number corresponding to the ASCII decimal representation in str.
11. str.itoa(i) stores the ASCII decimal representation of i into str (inverse of atoi).
12. str.hextoa(i) stores the ASCII hexadecimal representation of i into str (inverse of atohex).
13. str.bintoa(i) stores the ASCII binary representation of i into str (inverse of atobin).
14. str.realtoa(r) stores the ASCII real representation of r into str (inverse of atoreal)

String Operators

- Equality Syntax : Str1 == Str2
- Inequality Syntax : Str1 != Str2
- **Comparision**
  - Str1 < Str2
  - Str1 <= Str2
  - Str1 > Str2
  - Str1 >= Str2

- Concatenation. Syntax: {Str1,Str2,...,Strn}
- Replication. Syntax : {multiplier{Str}}

USERDEFINED DATATYPES

1. Class.
2. Enumarations.
3. Struct.
4. Union.
5. Typedef.

Enumarated Methods:

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated.

- The first() method returns the value of the first member of the enumeration.
- The last() method returns the value of the last member of the enumeration.
- The next() method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable.
- The prev() method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable.
- The name() method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the name() method returns the empty string.

DYNAMIC ARRAYS

- Declaration Of Dynmic Array:

integer dyna_arr_1[],dyn_arr_2[],multi_dime_dyn_arr[][];

- Allocating Elements:

new[]: The built-in function new allocates the storage and initializes the newly allocated array elements either to their default initial value.

dyna_arr_1 = new[10] ;// Allocating 10 elements

multi_dime_dyn_arr = new[4];// subarrays remain unsized and uninitialized

- Initializing Dynamic Arrays:

The size argument need not match the size of the initialization array. When the initialization array's size is greater, it is truncated to match the size argument; when it is smaller, the initialized array is padded with default values to attain the specified size.

dyna_arr_2 = new[4]('{4,5,6}); // elements are {4,5,6,0}

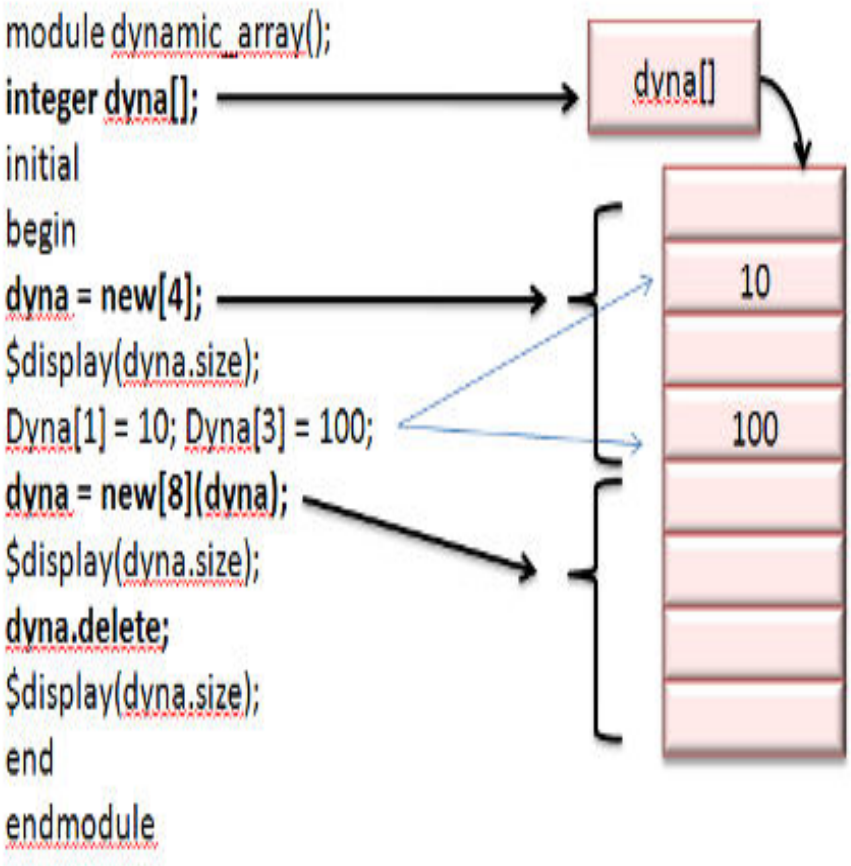
- Resizing Dynamic Arrays:

Using new[] constructor and its argument, we can increase the array size without losing the data content.

Dyna_arr_1 = new[100] (dyna_arr_1);

- Copy constructor of dynamic arrays is an easy and faster way of creating duplicate copies of data.

Dyna_arr_2 = new[100](dyna_arr_1);
//allocating & copying 100 elements.
Dyna_arr_1 = new[1000]; // Previous data lost. 1000 elements are allocated.



ASSOCIATIVE ARRAYS

Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically.

Associative arrays give you another way to store information. When the size of the collection is unknown or the data space is sparse, an associative array is a better option. In Associative arrays Elements Not Allocated until Used. Index Can Be of Any Packed Type, String or Class. Associative elements are stored in an order that ensures fastest access.

In an associative array a key is associated with a value. If you wanted to store the information of various transactions in an array, a numerically indexed array would not be the best choice. Instead, we could use the transaction names as the keys in associative array, and the value would be their respective information. Using associative arrays, you can call the array element you need using a string rather than a number, which is often easier to remember.

```
int array_name[*];//Wildcard index. can be indexed by any integral datatype.
int array_name [ string ];// String index
int array_name [ some_Class ];// Class index
int array_name [ integer ];// Integer index
typedef bit signed [4:1] Nibble;
int array_name [ Nibble ]; // Signed packed array
```

```
//associative array of 4-state integers indexed by strings, default is '1.
integer tab [string] = '{"Peter":20, "Paul":22, "Mary":23, default:-1 };
```

Associative Array Methods

- The num() or size() method returns the number of entries in the associative array.
- The delete() method removes the entry at the specified index.
- The exists() function checks whether an element exists at the specified index within the given array.
- The first() method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.
- The last() method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.
- The next() method finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.
- The prev() function finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

EXAMPLE

```
module assoc_arr;
int temp,imem[*];
initial
begin
imem[ 2'd3 ] = 1;
imem[ 16'hffff ] = 2;
imem[ 4'b1000 ] = 3;
$display( "%0d entries", imem.num );
if(imem.exists( 4'b1000) )
$display("imem.exists( 4b'1000) ");
imem.delete(4'b1000);
if(imem.exists( 4'b1000) )
$display(" imem.exists( 4b'1000) ");
else
$display(" ( 4b'1000) not existing");
if(imem.first(temp))
$display(" First entry is at index %0db ",temp);
if(imem.next(temp))
$display(" Next entry is at index %0h after the index 3",temp);
// To print all the elements alone with its indexs
if (imem.first(temp) )
do
$display( "%d : %d", temp, imem[temp] );
while ( imem.next(temp) );
end
endmodule
```

RESULT

```
3 entries
imem.exists( 4b'1000)
( 4b'1000) not existing
First entry is at index 3b
Next entry is at index ffff after the index 3
3 : 1
65535 : 2
```

QUEUES

- A queue is a variable-size, ordered collection of homogeneous elements.
- A Queue is analogous to one dimensional unpacked array that grows and shrinks automatically.
- Queues can be used to model a last in, first out buffer or first in, first out buffer.
- Queues support insertion and deletion of elements from random locations using an index.
- Queues Can be passed to tasks / functions as ref or non-ref arguments.
- Type checking is also done.

- Queues and dynamic arrays have the same assignment and argument passing semantics.

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

Queue Methods:

- The size() method returns the number of items in the queue. If the queue is empty, it returns 0.
- The insert() method inserts the given item at the specified index position.
- The delete() method deletes the item at the specified index.
- The pop_front() method removes and returns the first element of the queue.
- The pop_back() method removes and returns the last element of the queue.
- The push_front() method inserts the given element at the front of the queue.
- The push_back() method inserts the given element at the end of the queue.

EXAMPLE

```
module queues;
byte qu [$] ;

initial
begin
qu.push_front(2);
qu.push_front(12);
qu.push_front(22);
qu.push_back(11);
qu.push_back(99);
$display(" %d ",qu.size() );
$display(" %d ",qu.pop_front() );
$display(" %d ",qu.pop_back() );
qu.delete(3);
$display(" %d ",qu.size() );
end
endmodule
```

RESULTS :

```
5
22
99
```

```
module top;
typedef int qint_t[$];
// dynamic array of queues
qint_t DAq[]; // same as int DAq[][$];
// queue of queues

qint_t Qq[$]; // same as int Qq[$][$];
// associative array of queues
qint_t AAq[string]; // same as int
AAq[string][$];
initial begin
// Dynamic array of 4 queues
DAq = new[4];
// Push something onto one of the queues
DAq[2].push_back(1);
// initialize another queue with three entries
DAq[0] = {1,2,3};
$display("%p",DAq);
// Queue of queues -two
Qq= {{1,2},{3,4,5}};
Qq.push_back(qint_t'{6,7});
Qq[2].push_back(1);
$display("%p",Qq);
// Associative array of queues
AAq["one"] = {};
AAq["two"] = {1,2,3,4};
AAq["one"].push_back(5);
$display("%p",AAq);
end
endmodule : top
```

RESULTS:

```
'{'{1, 2, 3}, '{}, '{1}, '{}}
'{'{1, 2}, '{3, 4, 5}, '{6, 7, 1}}
'{one:'{5}, two:'{1, 2, 3, 4} }
```

COMPARISON OF ARRAYS

Static Array

Size should be known at compilation time.
Time require to access any element is less.
if not all elements used by the application, then memory is wasted.
Not good for sparse memory or when the size changes.
Good for contagious data.

Associative Array

No need of size information at compile time.
Time require to access an element increases with size of the array.
Compact memory usage for sparse arrays.
User don't need to keep track of size. It is automatically resized.
Good inbuilt methods for Manipulating and analyzing the content.

Dynamic Array

No need of size information at compile time.
To set the size or resize, the size should be provided at runtime.
Performance to access elements is same as Static arrays.
Good for contagious data.
Memory usage is very good, as the size can be changed dynamically.

Queues

No need of size information at compile time.
Performance to access elements is same as Static arrays.
User doesn't need to provide size information to change the size. It is automatically resized.
Rich set of inbuilt methods for Manipulating and analyzing the content.
Useful in self-checking modules. Very easy to work with out of order transactions.
Inbuilt methods for sum of elements, sorting all the elements.
Searching for elements is very easy even with complex expressions.
Useful to model FIFO or LIFO.

LINKED LIST

- The List package supports lists of any arbitrary predefined type, such as integer, string, or class object.
- First declare the Linked list type and then take instances of it.
- A double linked list is a chain of data structures called nodes. Each node has 3 members, one points to the next item or points to a null value if it is last node, one points to the previous item or points to a null value if it is first node and other has the data.
- The disadvantage of the linked list is that data can only be accessed sequentially and not in random order.

EVENTS

- An identifier declared as an event data type is called a named event which has no storage. An event can be triggered explicitly using "->" .
- Verilog Named Event triggering is recognized by using the event control "@".
- In SystemVerilog , Named Event triggering occurrence can also be recognized by using the event control wait().
- Wait() statement gets blocked until it evaluates to TRUE.

```
always
begin
#10;
if(e.triggered)
$display(" e is TRUE at %t",$time);
else
$display(" e is FALSE at %t",$time);
end
```

```
always
begin
wait(a.triggered);
$display(" TWO :: EVENT A is triggered ");
#1;
end
```

Randcase

EXAMPLE : randcase

```
program rand_case;
integer i;

initial begin
repeat(10)begin
randcase
begin
10: i=1;
20: i=2;
50: i=3;
end
$display(" i is %d \n",i);end
end
endprogram
```

RESULTS

```
i is 3
i is 2
i is 3
i is 3
i is 3
i is 3
i is 1
i is 1
i is 1
i is 2
```

When a **case**, **casex**, or **casez** statement is specified as **unique**, software tools must perform additional semantic checks to verify that each of the case selection items is mutually exclusive.

If a case expression value occurs during run time that matches more than one case selection item, the tool must generate a run-time warning message.

EXAMPLE:

```
module casee;

initial
begin

for(int a = 0;a<4;a++)
unique case(a) // values 3,5,6,7 cause a warning
0,1: $display("0 or 1");
2: $display("2");
4: $display("4");
endcase

for(int a = 0;a<4;a++)
priority casez(a) // values 4,5,6,7 cause a warning
3'b00?: $display("0 or 1");
3'b0??: $display("2 or 3");
endcase

end
endmodule
```

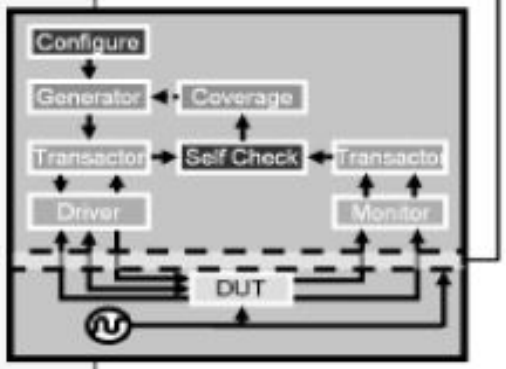
Program Block

- **The Program block provides**
  - Entry point to test execution
  - Scope for program-wide data and routines
  - Race-free interaction between testbench and design
- **Develop test code in program block**

```
program automatic test(router_io.TB rtr_io);
  //testbench code in initial block:
  initial begin
    run();
  end
  task run();
    ...
  endtask: run
endprogram: test
```

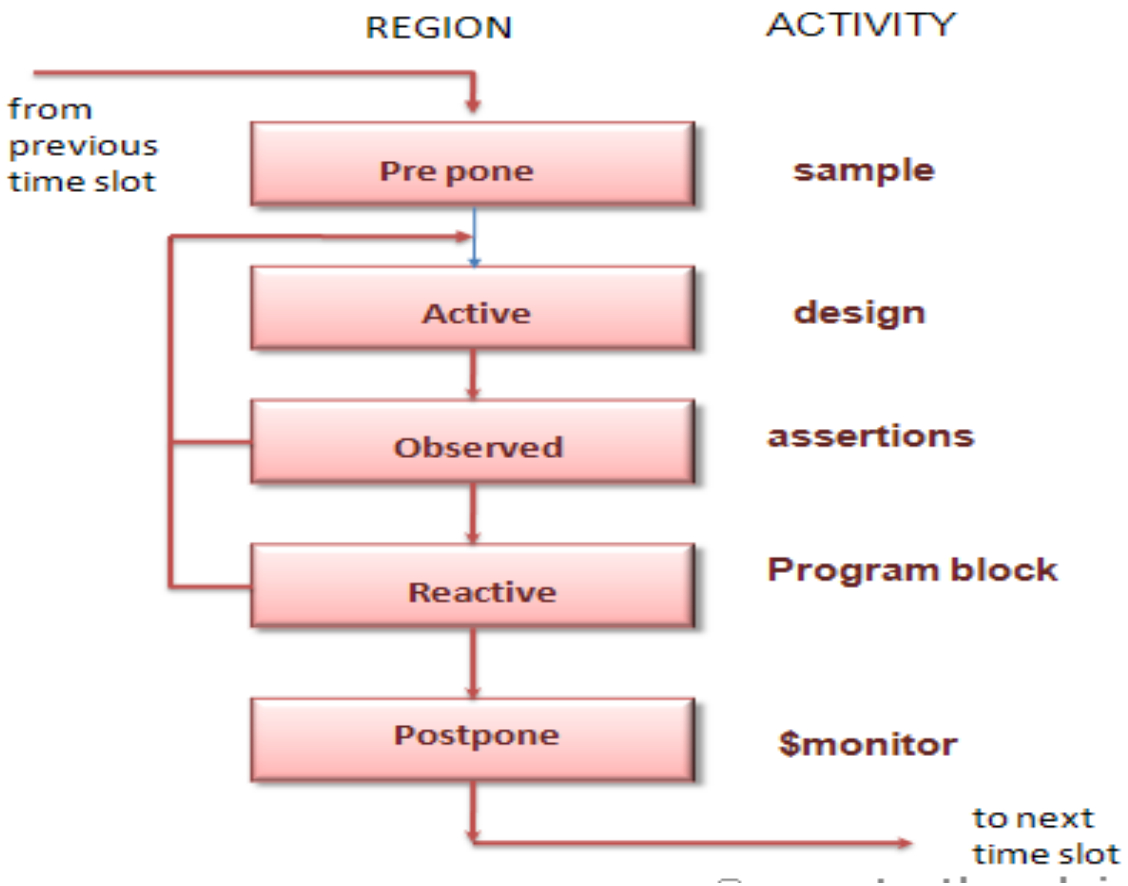
Pass interface in port list

Only initial blocks allowed in programs



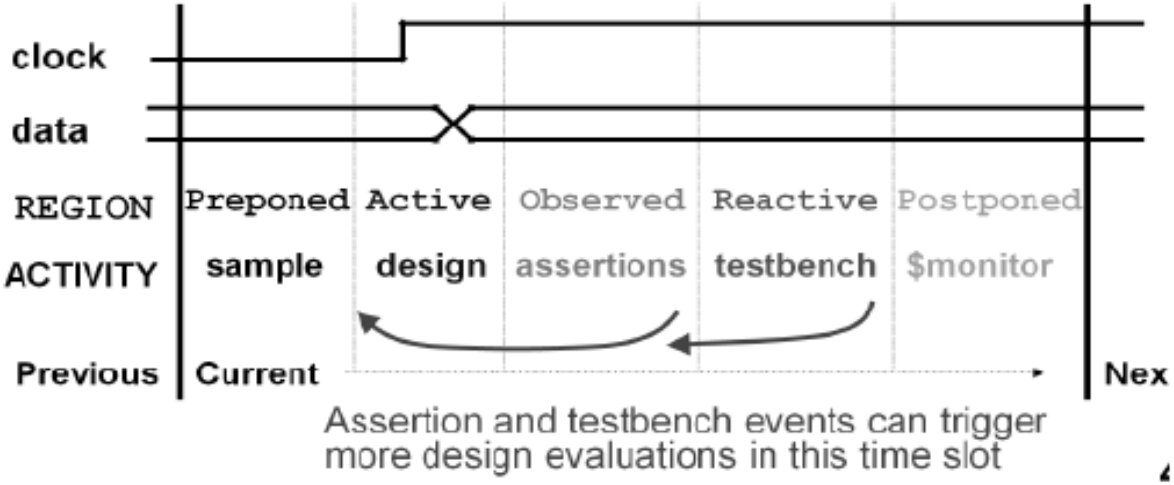
PROGRAM BLOCK

- Separates the testbench from the DUT.
- The program block helps ensure that test bench transitions do not have race conditions with the design
- It provides an entry point to the execution of testbenches.
- It creates a scope that encapsulates program-wide data.
- It provides a syntactic context that specifies scheduling in the Reactive region which avoids races.
- It doesnot allow always block. Only initial and methods are allowed, which are more controllable.
- Each program can be explicitly exited by calling the $exit system task. Unlike $finish, which exits simulation immediately, even if there are pending events.
- Just like a module, program block has ports. One or more program blocks can be instantiated in a top-level netlist, and connected to the DUT.


REGION
ACTIVITY
from previous time slot
Pre pone
sample
Active
design
Observed
assertions
Reactive
Program block
Postpone
$monitor
to next time slot

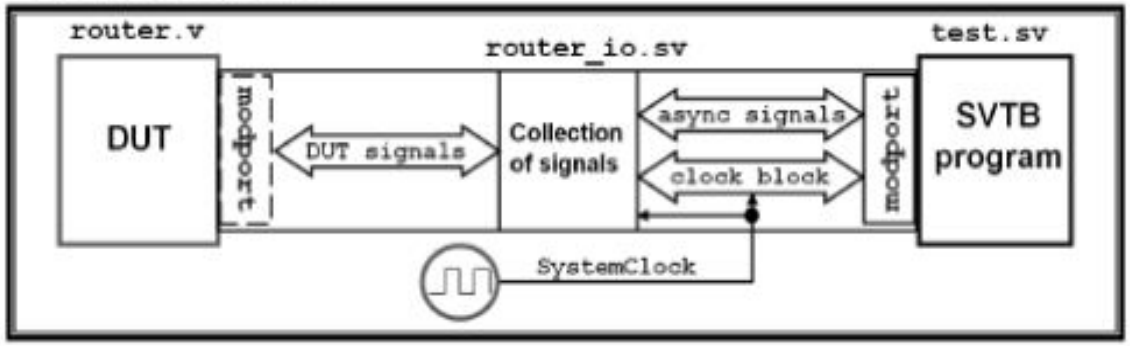
SV scheduler

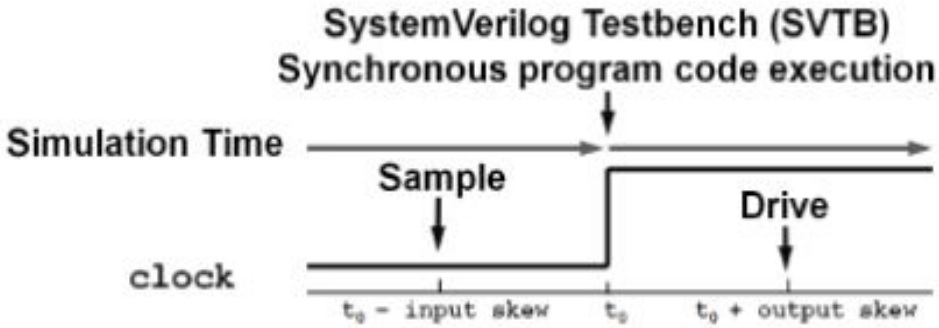
- **Each time slot is divided into 5 major regions**
  - `Preponed` Sample signals before any changes (`#1step`)
  - `Active` Design simulation (`module`), including NBA
  - `Observed` Assertions evaluated after design executes
  - `Reactive` Testbench activity (`program`)
  - `Postponed` Read only phase



- **Program code defaults to Cycle-based Simulation for synchronous drives and samples:**
  - Events occur at interface clock-block clock edges

router_test_top.sv





Clocking blocks

- **Just for testbench**
  - Emulates the Launch and Capture Flops at IO of DUT.
- **Creates explicit synchronous timing domains**
  - All signals driven or sampled at clocking event - by default all interface signals are asynchronous
  - Interaction between testbench and DUT ideally happens only at clock edges
- **Provides direction and signal *bundle that can be passed* through a modport**
  - Outputs cannot be sampled
  - Input signals cannot be driven

setup Clock to q

```
clocking cb @(posedge clk);
 default input #1ns output #1ns;
 output reset_n;
 output din;
 output frame_n;
 output valid_n;
 input  dout;
 input  busy_n;
 input  valido_n;
 input  frameo_n;
endclocking: cb
```

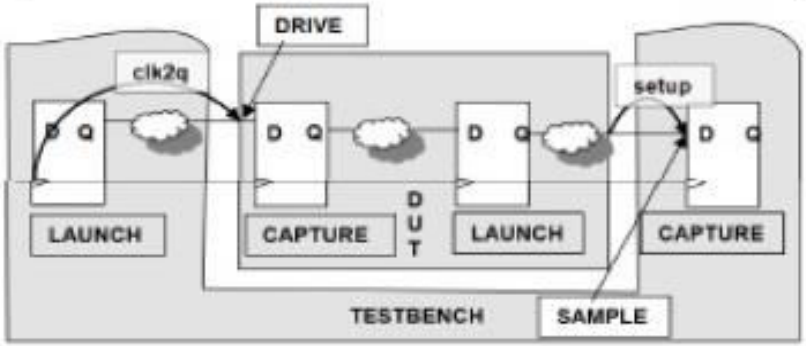
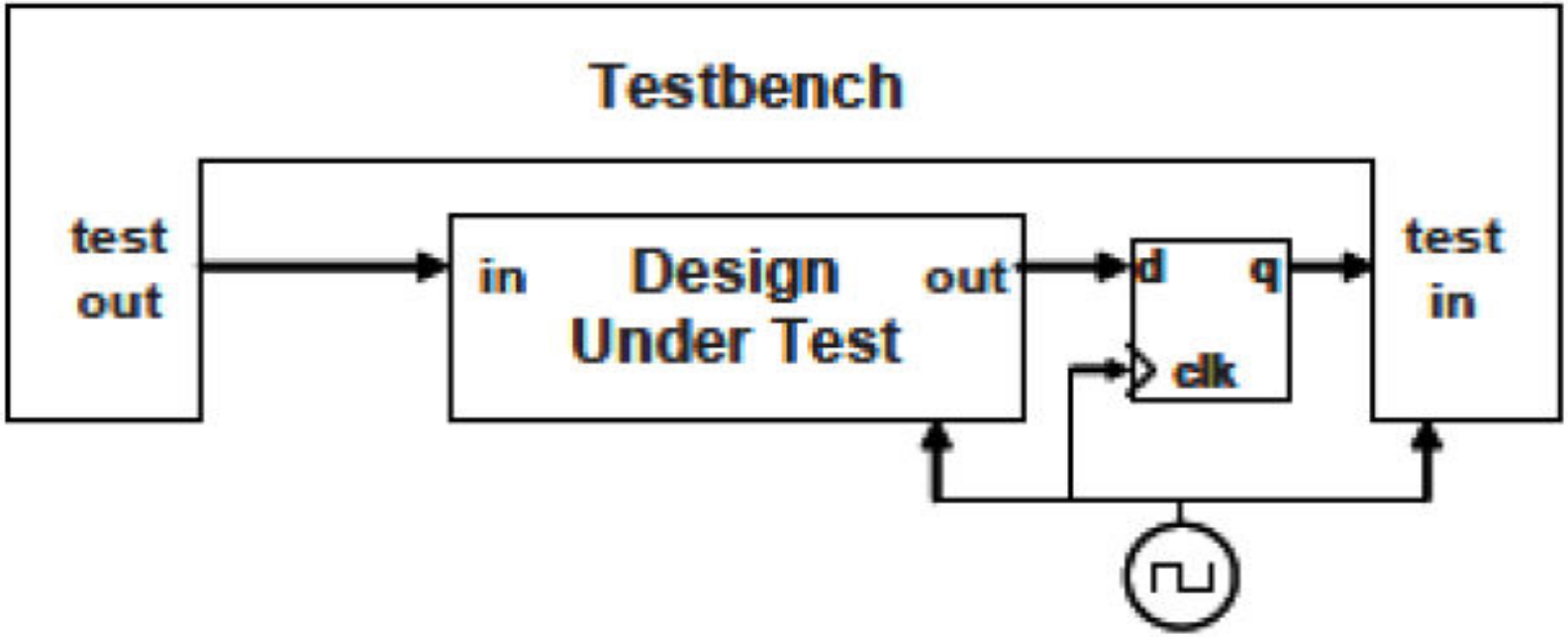


Figure 4-5 A clocking block synchronizes the DUT and testbench



- **Enforce signal access & direction with modport**

```
interface router_io(input bit clock);
  logic  reset_n ;
  …
  clocking cb @(posedge clock);
    default input #1ns output #1ns;
    output reset_n;
    output valid_n;
    …
  endclocking
   modport DUT(input reset_n, input din, output dout,...);
   modport TB(clocking cb, output reset_n);
endinterface: router_io
```

```
module router(
 router_io.DUT db,
 input bit clk);
…
endmodule: router
```

```
program automatic test(router_io.TB
rtr_io);
   initial
     rtr_io.reset_n <= 1'b0;
     rtr_io.cb.valid_n <= '1;
endprogram: test
```

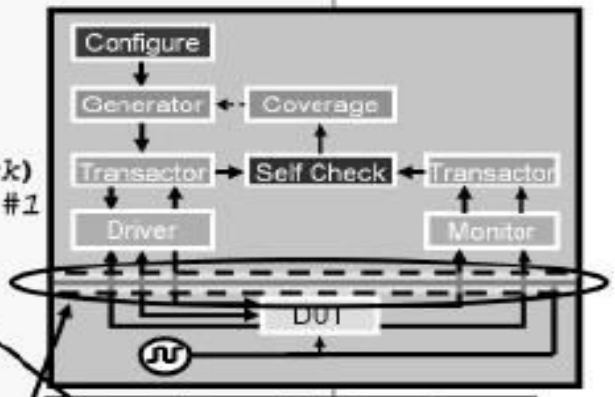
```
interface router_io(input bit clock);
  logic  reset_n ;
  logic  [15:0]  din ;
  logic  [15:0]  frame_n ;
  logic  [15:0]  valid_n ;
  logic  [15:0]  dout ;
  logic  [15:0]  busy_n ;
  logic  [15:0]  valido_n ;
  logic  [15:0]  frameo_n ;
  clocking cb @(posedge clock)
    default input #1 output #1
    output reset_n;
    output din;
    output frame_n;
    output valid_n;
    input  dout;
    input  busy_n;
    input  valido_n;
    input  frameo_n;
  endclocking
  modport TB(clocking cb, output reset_n);
endinterface
```

Named bundle of asynchronous signals

Create synchronous behavior by placing into clocking block

Define access and direction with modport

router_io.sv



Sample and drive skews

Direction w/respect to test

Synchronous

Asynchronous

Pass modport as port list

Asynchronous signals are driven without reference to clocking block

```
interface router_io(input bit clock);
          logic  reset_n ;
          logic  [15:0] din ;
                 frame_n ;
                 valid_n ;
```

```
program automatic test(router_io.TB rtr_io);
  //testbench code in initial block
  initial begin
    reset() ;
  end
  task reset() ;
    rtr_io.reset_n <= 1'b0;
    rtr_io.cb.frame_n <= 16'hffff;
    rtr_io.cb.valid_n <= ~('b0) ;
    // reset_n can be both synchronous and asyn
    ##2 rtr_io.cb.reset_n <= 1'b1;
    repeat(15) @(rtr_io.cb) ;
  endtask: reset
endprogram: test
```

Synchronous signals are referenced via clocking block

```
...
clocking cb @(posedge clock);
  default input #1 output #1;
  output reset_n;
  output din;
  output frame_n;
  output valid_n;
  ...
endclocking: cb
modport TB (
clocking cb, output reset_n);
endinterface: router_io
```

Advance clock cycles via clocking block

Instantiate test program and interface in harness file

Legacy DUT (wires)

Instantiate interface

Connect SystemClock to interface block

Instantiate test program

DUT instantiation using interface connection

```
module router(
     reset_n
     clock,
     frame_n,
     valid_n,
     din,
     dout,
     busy_n,
     valido_n,
     frameo_n);
…
endmodule: router
```

```
module router_test_top;
parameter simulation_cycle =
bit  SystemClock;
router_io top_io(SystemClock)
test router_test(top_io);
router dut(.reset_n (top_io.reset_n),
           .clock   (top_io.clock),
           .frame_n (top_io.frame_n),
           .valid_n (top_io.valid_n),
           .din     (top_io.din),
           .dout    (top_io.dout),
           .busy_n  (top_io.busy_n),
           .valido_n(top_io.valido_n),
           .frameo_n(top_io.frameo_n)
            );
…
endmodule:  router_test_top
```

Running the simulation

- Set SystemVerilog run-time option with +argument
- Retrieve +argument value with `$value$plusargs()`

```
initial begin: proc_user_args
  int user_seed;
  if ($value$plusargs("ntb_random_seed=%d", user_seed))
    $display("User seed is %d", user_seed);
  else
    $display("Using default seed");
end: proc_user_args
```

```
> ./simv +ntb_random_seed=100
User seed is 100
```

- Create your own +argument user options for simulation control and debug

- 2 adders
- 1st adder a+b (4 bit wide)
- 2nd adder c+d (4 bit wide)
- Mux logic
- Based on sel o/p a+b or c+d

- Compare a with b, also c with d
- Test cases for the above ckt.
- Clk , reset are also inputs
- Interface, DUT, program test, top

Session 10 and 11

Pankaj Badhe

VIRTUAL INTERFACE

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same subprogram to operate on different portions of a design, and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals.

```
1 module testbench(intf.tb tb_if);
2 virtual interface intf.tb local_if; // virtual interface.
3 ....
4 task read(virtual interface intf.tb l_if) // As argument to task
5 ....
6 initial
7 begin
8 Local_if = tb_if; // initializing virtual interface.
9 Local_if.cb.read <= 1; //writing to synchronous signal read
10 read(Local_if); // passing interface to task.
11 end
12 endmodule
```

1) Virtual interface can be used to make the TestBench independent of the physical interface. It allows developing the test component independent of the DUT port while working with multi port protocol.
2) With virtual interface, we can change references to physical interface dynamically. Without virtual interfaces, all the connectivity is determined during compilation time, and therefore can't be randomized or reconfigured.
3) In multi port environment, it allows to access the physical interfaces using array index.
4) Physical interfaces are not allowed in object oriented programming, as physical interface is allocated at compilation time itself. Virtual interface which are set at run time allows to do object oriented programming with signals rather than just with variables.
5) Virtual interface variables can be passed as arguments to tasks, functions, or methods.

Class

- **Similar to a module, an OOP *class* encapsulates:**
  - Variables (properties) used to model a system
  - Subroutines (methods) to manipulate the data
  - Properties & methods are called members of class

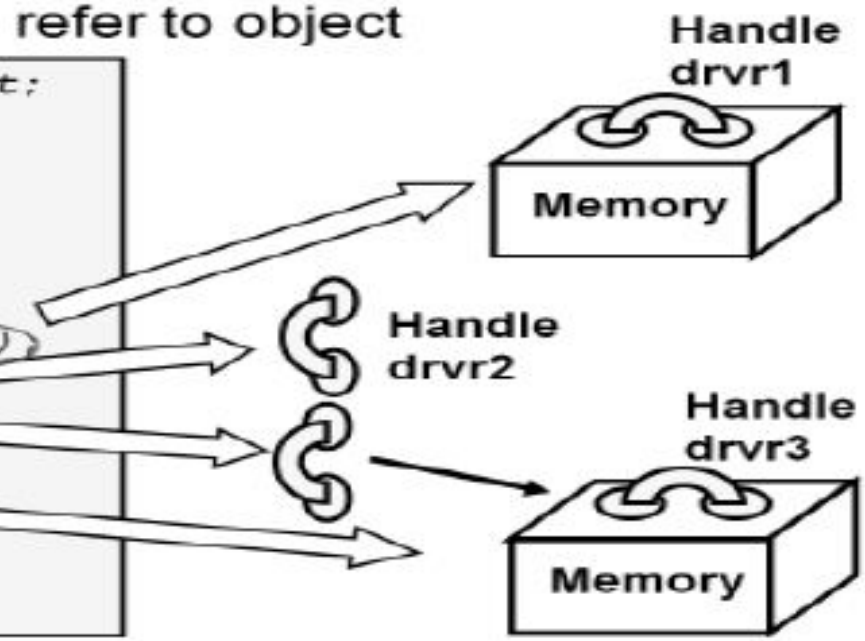
Class properties and methods are visible inside the class

```
class Driver;
  string     name;
  bit[3:0]   sa, da;      //copy of Packet properties
  bit[7:0]   payload[];   //copy of Packet properties

  task send();
    send_addrs();
    send_pad();
    send_payload();
  endtask: send

  task send_addrs();     ...  endtask
  task send_pad();       ...  endtask
  task send_payload();   ...  endtask
endclass: Driver
```

- **Why use class?**
  - Objects are dynamic, modules are static
    - Objects are created and destroyed as needed
  - Instances of classes are objects
    - A handle points to an object (class instance)
    - Object handles can be passed as arguments
    - Object memory can be copied or compared
    - Instances of modules can not be passed, copied or compared
  - Classes can be inherited, modules can not
    - Classes can be modified via inheritance without impacting existing users
    - Modifications to modules will impact all existing users

- **OOP objects are created from class definitions:**
  - Similar to instance creation from module definition
- **Object memory is constructed by calling `new()`**
  - Handle used to refer to object

```
program automatic test;
  class Driver;
    bit[3:0]  sa, da;
    byte payload[];
    task send();
    ...
  endclass: Driver
  Driver drvr1 = new();
  Driver drvr2;
  Driver drvr3;
  initial begin
    drvr3 = new();
    ...
  end
endprogram: test
```



- **Define constructor `new()` to initialize properties:**
  - No return type in declaration
  - Executes immediately after object memory is allocated
  - Not accessible via dot (.) notation

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0] init_sa, init_da, int init_payload_size);
      sa = init_sa;
      da = init_da;
      payload = new[init_payload_size];
    endfunction: new
  endclass: Packet
  initial begin
    Packet pkt1 = new(3, 7, 2);
    pkt1.new(5, 8, 3);          // syntax error!
  end
endprogram
```

- **`this` keyword**
  - An object's handle to itself
  - Unambiguously refers to class properties and methods of the current instance (object)
    - More readable – allows method arguments to have same name as class variables

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0] sa, da, int payload_size);
      this.sa = sa;
      this.da = da;
      this.payload = new[payload_size];
    endfunction: new
  endclass: Packet
  ...
endprogram: test
```

- **Often need to use a class before declaration**
  - e.g. two classes need handle to each other
    - Use `typedef`

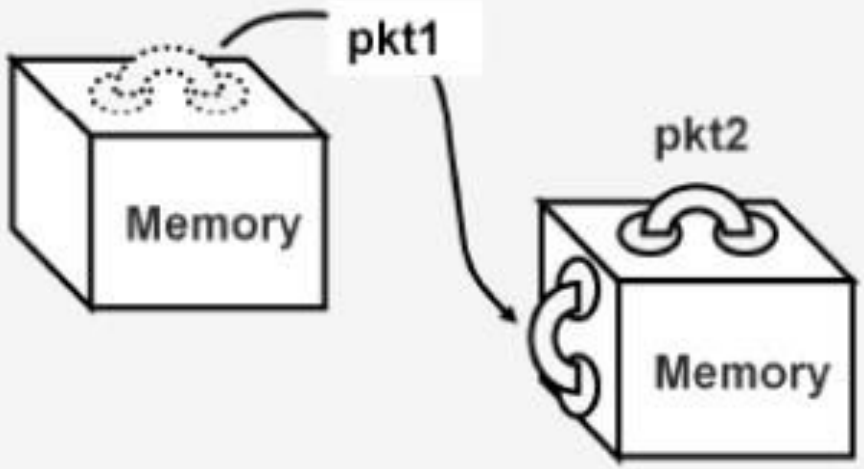
```
typedef class child;
class parent;
child c1;
...
endclass: parent

class child;
parent p1;
...
endclass: child
```

This is a compile error if typdef is missing

What happens when one object handle is assigned to another?

Like any variable, the target takes on the value of the source.

```
class Packet;
    int payload_size;
    ...
endclass: Packet

...
Packet pkt1 = new();
Packet pkt2 = new();
...
pkt1 = pkt2;
pkt1.payload_size = 5; // whose payload_size is set ?
```



- **Object can be manually de-referenced**

```
pkt1 = null;
```

- **Making an exact duplication of object memory:**

```
class Packet;
  int count;
  Payload p; //encapsulated object
endclass: Packet
…
Packet packet1 = new();
Packet packet1_copy;          // handle only
packet1.p = new();
packet1_copy = new packet1;   // shallow copy
                              // (encapsulated objects not copied)
                              // object packet1 must exist
                              // deep copy not supported in SVTB
```

- **Static properties:**
  - Associated with the class, not the object
  - Shared by all objects of that class

```
program automatic test;
  class Packet;
    static int count = 0;
    int id;
    string str_var;
    function new();
      this.id = count++;
    endfunction
  endclass: Packet

  initial begin
    Packet packet1 = new();
    Packet packet2 = new();
    $display("My id is: %0d", packet1.id);
    $display("My id is: %0d", packet2.id);
  end
endprogram: test
```

What values get printed?

- Class Analog;
- Class Digital which has instance of class Analog;
- 2 instances of Class Digital D1 and D2.
- Allocate memory to Class D1.A1 (A1 is instance of class analog inside D1)
- Try to copy D1 to D2 and check the contents of D1.A1, D1, D2, and D2.A1
- Try to code a deep copy function yourself and use if .
- Try to deep copy D1 to D2 and check the contents of D1.A1, D1, D2, and D2.A1

Parameterised class

- **Written for a generic type**
  - Type parameter passed at instantiation, just like parameterized modules
  - Allows reuse of common code

```
class stack #(type T = int);
 protected T items[$];
 function void push( T a );
 ...
 function T pop( );
 function int size(); ...
endclass: stack
```

```
program automatic test;
  stack addr_stack; //default type
  stack #(Packet) data_stack;
initial begin
  ...
  repeat(addr_stack.size()) begin
    Packet pkt = new();
    if(!pkt.randomize())
        $finish;
    pkt.addr = addr_stack.pop();
    data_stack.push(pkt);
   end
end
endprogram: test
```

- **Methods can be placed outside of the class block:**
  - Inside class block, declare a extern prototype
  - Outside class block, use a pair of colons :: to associate the method with its class
  - Double-colon is a scope/name resolution operator

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  task ping();
    ...
  endtask: ping
endclass: node
```

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  extern task ping();   // prototype
endclass: node
task node::ping();
  ...
endtask: ping
```

Place class name and double-colons before method name

- **Useful methods for Data classes (user defined):**
  - `display()`
    - Print object variables to console
    - Helpful for debugging
  - `compare()`
    - Returns match, mismatch and other status base on results of comparing object variables to variables of another object
    - Simplifies self-check
  - `copy()`
    - Copy selected variables or nested objects
    - Useful when exact duplication is not desirable
    - Allows you to do deep copy if required

- **Pass virtual interface in via constructor**
- **Drive/Sample signals with virtual interface**
  - Class can be re-used for any `router_io` instance

```
class Driver;
   virtual router_io.TB rtr_if;
   ...
   function new(virtual router_io.TB router);
     this.rtr_if = router;
   endfunction: new

   task send_addrs();
     this.rtr_if.cb.frame_n[sa] <= 1'b0;
     for(int i=0; i<4; i++) begin
       this.rtr_if.cb.din[sa] <= da[i];
       @(this.rtr_if.cb);
     end
   endtask: send_addrs
endclass: Driver
```

Create virtual interface instance

Pass virtual connections via constructor argument

Drive/Sample signals using virtual interface

Inheritance

```
class base;
local int i;
endclass

class ext extends base;
function new();
```

ENCAPSULATION

EXAMPLE: local variblg error

```
class base;
local int i;
endclass

program main;
initial
begin
base b = new();
b.i = 123;
end
endprogram
```

RESULT:

Local member 'i' of class 'base' is not accessible from scope 'main'

The above examples gives compilation error.

EXAMPLE: local varible access using method

```
class base;
local int i;

task set(int j);
i = j;
$display(i);
endtask
endclass

program main;
initial
begin
base b = new();
b.set(123);
end
endprogram
```

RESULT

```
123
```

POLYMORPHISM

EXAMPLE: with virtual

```
class A ;
virtual task disp ();
$display(" This is class A ");
endtask
endclass

class EA extends A ;
task disp ();
$display(" This is Extended class A ");
endtask
endclass

program main ;
EA my_ea;
A my_a;

initial
begin
my_a = new();
my_a.disp();

my_ea = new();
my_a = my_ea;
my_a.disp();
end
endprogram
```

RESULTS

```
This is class A
This is Extended class A
```

```
class BusTran;
  bit [31:0] addr, crc, data[8];

  function void display;
    $display("BusTran: %h", addr);
  endfunction : display

  function void calc_crc;
    crc = addr ^ data.xor;
  endfunction : calc_crc

endclass : BusTran
```

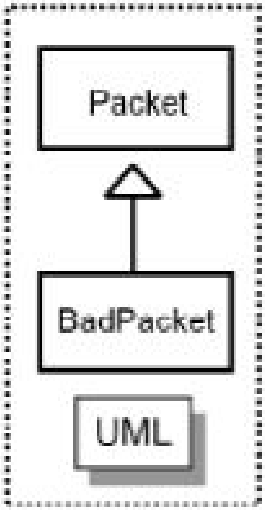
Bad generator creates only one object

```
task generator_bad(int n);
  BusTran b;
  b = new();                        // Create one new object
  repeat (n) begin
    b.addr = $random();             // Initialize variables
    $display("Sending addr=%h", b.addr);
    transmit(p);                    // Send it into the DUT
  end
endtask
```

Good generator creates many objects

```
task generator_good(int n);
  BusTran b;
  repeat (n) begin
    b = new();                    // Create one new object
    b.addr = $random();           // Initialize variables
    $display("Sending addr=%h", b.addr);
    transmit(p);                  // Send it into the DUT
  end
endtask
```

- **Object-oriented programming**
  - New classes derived from original (base) class
  - Inherits all contents of base class




```
class BadPacket extends Packet;
```

Packet

BadPacket

da
sa
data
crc

```
function void display();
   ...
endfunction: display
```

```
function bit[31:0] compute_crc();
   ...
endfunction: compute_crc
```

is_bad

```
function bit[31:0] compute_crc()
   crc = super.compute_crc();
   if (is_bad) crc = ~crc;
endfunction: compute_crc
```

Can override methods in base class

Can call overridden behavior

- **Derived classes compatible with base class**
  - Can reuse code

```
Packet pkt = new();
transmit(pkt);
```

```
BadPacket bad_pkt = new();
transmit(bad_pkt);
```

Compatible

```
task transmit(Packet pkt);
    ...
    pkt.crc = pkt.compute_crc;
    ...
endtask: transmit
```

Base class

Packet

Compatible

Use $cast()

BadPacket

Derived class

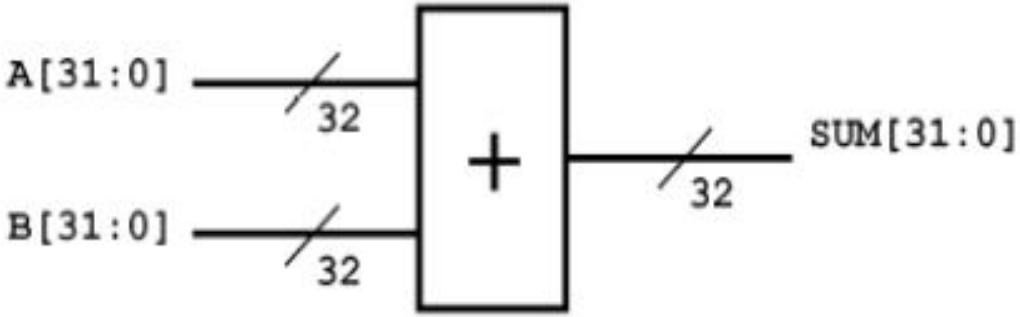
OK	`pkt = bad_pkt;`
Error	`bad_pkt = pkt;`
OK	`$cast(bad_pkt, pkt);`

Introduction to verification

It will take over 500 years to verify this simple adder.

How long will simulation run if exhaustive testing of a 32-bit adder is required?

Assume that one set of input and output can be verified every 1ns.

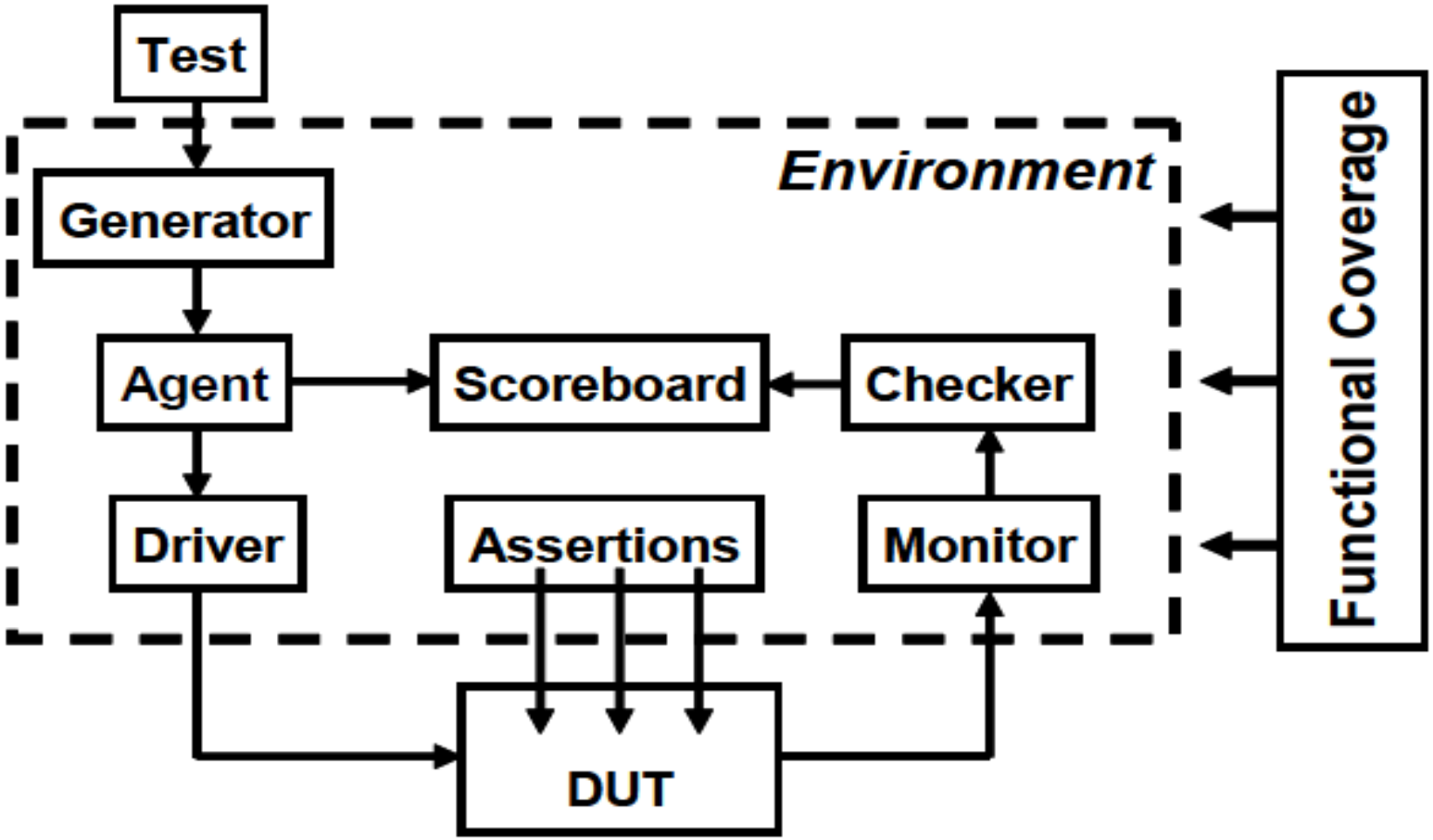


A day?

A week?

A year?

Building a Testbench



The Generator, Agent, Driver, Monitor, Checker, and Scoreboard are all classes, modeled as transactors.

A transactor is made of a simple loop that receives a transaction object from a previous block, makes some transformations, and sends it to the following one.

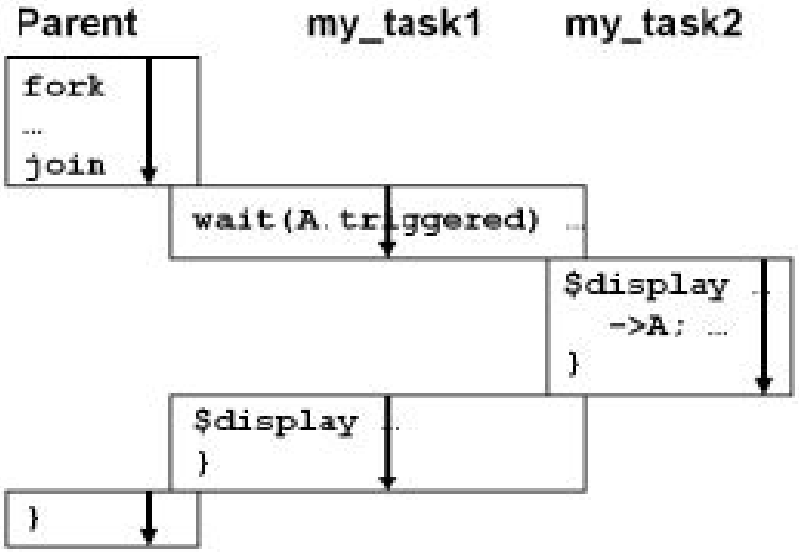
Some, such as the **Generator**, have no upstream block, so this transactor constructs and randomizes every transaction, while others, such as the **Driver**, receive a transaction and send it into the DUT as signal transitions.

```
class Transactor;  // Generic class
  Transaction tr;

  task run;
    forever begin
      // Get transaction from upstream block
      ...
      // Do some processing
      ...
      // Send it to downstream block
      ...
      end
  endtask

endclass
```

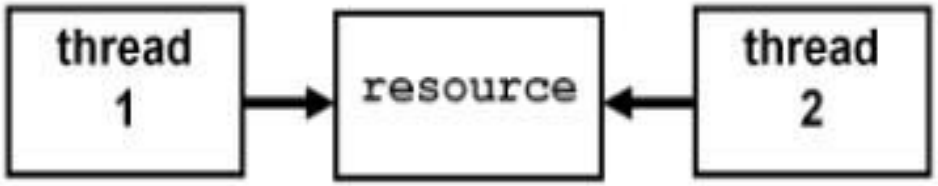
Communication with adjacent blocks

Controlling order of execution.

```
initial begin
  event A;
  fork
      my_task1();
      my_task2();
  join
  task my_task1();
    wait(A.triggered);
    $display("Print 2nd");
  endtask: my_task1
  task my_task2();
    $display("Print 1st");
    ->A;
  endtask: my_task2
end
```



Resource sharing



- **Synchronize operation of concurrent threads via access to shared resources:**
  - Thread requests for a shared resource before executing a critical section of code
  - Thread waits if the requested resource is unavailable
  - Thread resumes execution when the requested resource becomes available

Semaphores

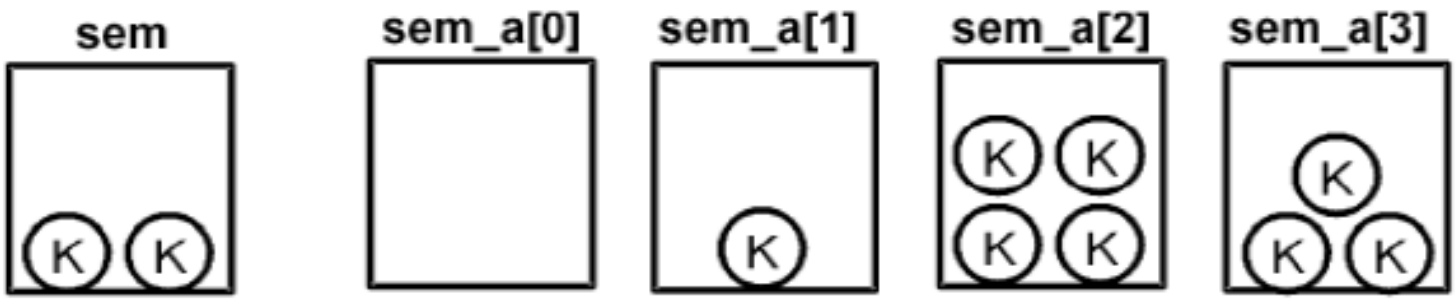
- **A Semaphore is a bucket in which keys can be deposited and removed:**
  - A thread tries to acquire keys from a semaphore bucket
  - Thread execution waits until keys requested are available in the semaphore bucket
  - When the requested keys becomes available, thread execution resumes and the requested keys removed from the semaphore bucket
- **Mainly used to prevent multiple threads from accessing the same hardware signal or using same software resource**

- **Semaphores are supported via built-in semaphore class:**

```
class semaphore;
  function new(int keyCount = 0);
  task put(int keyCount = 1);

  // the following is blocking:
  task get(int keyCount = 1);

  // the following is non-blocking: 1 for success 0 for failure
  function int try_get(int keyCount = 1);
endclass
```

- **Semaphore buckets are created when the constructor `new()` is called**

```
semaphore sem, sem_a[];
sem = new(2);            // semaphore bucket with 2 keys is created
sem_a = new[4];          // an array of size 4 is created
sem_a[2].put(2);         // ERROR: semaphore bucket does not exist!
foreach (sem_a[i])
   sem_a[i] = new(i);    // create semaphore buckets with i keys each
sem_a[2].put(2);         // okay. sem_a[2] now has four keys
```



```
program automatic test(router_io.TB router);
   semaphore sem[];
   ...;
   sem = new[16];
   foreach(sem[i])
      sem[i] = new(1);
   ...;
   task send();
      sem[da].get(1);
      send_addrs();
      send_pad();
      send_payload();
      sem[da].put(1);
   endtask: send
   ...;
endprogram: test
```

Create semaphore array to represent each output port

Construct each individual semaphore bucket

Block if others are driving the chosen output port

Re-deposit keys when done with driving port

Mailbox

- **Messages are passed between threads via mailbox:**
  - A thread sends data by putting messages a mailbox
  - A thread retrieves data by getting messages from the mailbox
  - If a message is not available, the thread can wait until there is a message to retrieve
  - Thread resumes execution once the message becomes available
- **Mainly used for passing data between concurrent threads**

- **Mailboxes are supported via built-in mailbox class:**

```
class mailbox #(type T = dynamic_type);
  function new(int bound = 0);
  function int num();      // return # of messages
  task put(T message);     // wait if mailbox full
  task get(ref T message); // wait if no message
  task peek(ref T message); // wait if no message
  function int try_put(T message);
  function int try_get(ref T message);
  function int try_peek(ref T message);
endclass
```

```
task put(message); // block if exceeds bound
function int try_put(message); // non-blocking
```

Example:

mbox

16'hcafe

```
int status;
bit[15:0] test_data = 16'hcafe
mailbox #(bit[15:0]) mbox = new(1);
Packet pkt = new();
mbox.put(test_data);
status = mbox.try_put(test_data);//status = 0 - mailbox full
mbox.put(pkt);                   //incorrect type - compiler error
```

```
task get(ref message); // block if mailbox is empty
function int try_get(ref message); // non-blocking
// if successful, return number of messages in mailbox before try_get,
// 0 if no message
```

Example:

mbox

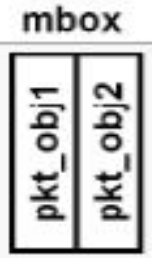
16'hcafe

```
int status;
bit[15:0] test_data = 16'hcafe
mailbox #(bit[15:0]) mbox = new(1);
mbox.put(test_data);
mbox.get(test_data);
$display("test_data = %0h", test_data);// test_data = cafe
status = mbox.try_get(test_data); // status = 0
status = mbox.try_put(test_data); // status = 1
mbox.get(test_data);              // test_data = cafe
mbox.get(test_data);              // blocking – mailbox empty
mbox.get(mbox);                   // incorrect type - compiler error
```

```
task peek(ref message);  // block if mailbox is empty
function int try_peek(ref message);  // non-blocking
// if successful, return number of message in mailbox before try_peek,
// 0 if no message
```

Example:

```
Packet pkt_obj1 = new();
Packet pkt_obj2 = new();
mailbox #(Packet) mbox = new();
mbox.put(pkt_obj1);
status = mbox.try_put(pkt_obj2);          // status = 1
$display("%0d messages", mbox.num());  // 2 messages
mbox.peek(pkt_obj1);                      // still 2 messages left
$display(mbox.try_peek(pkt_obj1));     // displays 2
$display(mbox.try_get(pkt_obj1));      // displays 2
```



BIG PICTURE

```
program mailbox_example(bus_if.TB bus, ...);

  class Generator;
    Transaction tr;
    mailbox mbx;

    function new(mailbox mbx);
      this.mbx = mbx;
    endfunction

    task run;
      repeat (10) begin
        tr = new;
        assert(tr.randomize);
        mbx.put(tr);     // Send out transaction
      end
    endtask
  endclass
```

```
class Driver;
  Transaction tr;
  mailbox mbx;

  function new(mailbox mbx);
    this.mbx = mbx;
  endfunction

  task run;
    repeat (10) begin
      mbx.get(tr);        // Fetch next transaction
      @(posedge busif.cb.ack);
      bus.cb.kind <= tr.kind;
      ...
    end
  endtask
endclass
```

```
 mailbox mbx;              // Mailbox connecting gen & drv
 Generator gen;
 Driver drv;
 initial begin
   mbx = new;
   gen = new(mbx);

    drv = new(mbx);
    fork
      gen.run();            // Spawn the generator
      drv.run();            // Spawn the driver
    join
  end
endprogram
```

Bounded mailbox

```
program automatic bounded;
  mailbox mbx;

  initial begin
    mbx = new(1);  // Size = 1
    fork

      // Producer
      for (int i=1; i<4; i++) begin
       $display("@%0d: Producer: putting %0d", $time, i);
        mbx.put(i);
        $display("@%0d: Producer: put(%0d) done %0d",
                  $time, i);
      end

      // Consumer
      repeat(3) begin
        int j;
        #1ns mbx.get(j);
        $display("@%0d: Consumer: got %0d", $time, j);
      end
    join_any
  end
endprogram
```

```
@0: Producer: before put(1)
@0: Producer: put(1) done
@0: Producer: before put(2)
@1: Consumer: got 1
@1: Producer: put(2) done
@1: Producer: before put(3)
@2: Consumer: got 2
@2: Producer: put(3) done
```

Basic Transactor

```
class Agent;

  mailbox gen2agt, agt2drv;
  Transaction tr;

  function new(mailbox gen2agt, agt2drv);
    this.gen2agt = gen2agt;
    this.agt2drv = agt2drv;
  endfunction

  function build;
  // Empty for now
  endfunction
```

```
  task run;
    forever begin
      // Get transaction from upstream block
      gen2agt.get(tr);

      // Do some processing

      // Send it to downstream block
      agt2drv.put(tr);
    end
  endfunction

  task wrapup;
  // Empty for now
  endtask

endclass
```

```
class Environment;

  Generator  gen;
  Agent      agt;
  Driver     drv;
  Monitor    mon;
  Checker    chk;
  Scoreboard scb;
  Config     cfg;
  mailbox gen2agt, agt2drv, mon2chk;

  extern function new;
  extern function void gen_cfg;
  extern function void build;
  extern task run;
  extern task wrapup;
endclass
```

```
function Environment::new;
  // Initialize mailboxes
  gen2agt = new;
  agt2drv = new;
  mon2chk = new;

  // Initialize transactors
  gen = new(gen2agt);
  agt = new(gen2agt, agt2drv);
  drv = new(agt2drv);
  mon = new(mon2chk);
  chk = new(mon2chk);
  scb = new;
  cfg = new;
endfunction

function void Environment::gen_cfg;
  assert(cfg.randomize);
endfunction
```

```
function void Environment::build;
  gen.build;
  agt.build;
  drv.build;
  mon.build;
  chk.build;
  scb.build;
endfunction

task Environment::run;
  fork
    gen.run(run_for_n_trans);
    agt.run;
    drv.run;
    mon.run;
    chk.run;
    scb.run(run_for_n_trans);
  join
endtask
```

```
task Environment::wrapup;
  fork
    gen.wrapup;
    agt.wrapup;
    drv.wrapup;
    mon.wrapup;
    chk.wrapup;
    scb.wrapup;
  join
endtask
```

```
program automatic test;

  Environment env;

  initial begin
    env = new;
    env.gen_cfg;
    env.build;
    env.run;
    env.wrapup;
  end
endprogram
```

Randomize and Callbacks

Pankaj Badhe

Example 5-22 Bidirectional signals in a program and interface

```
interface master_if (input bit clk);
  wire [7:0] data;  // Bidirectional signal

  clocking cb @(posedge clk);
  inout data;
  endclocking

  modport TEST (clocking cb);
endinterface


program test(master_if mif);

  initial begin
    mif.cb.data <= 'z;          // Tri-state the bus
    @mif.cb;
    $displayh(mif.cb.data);     // Read from the bus
    @mif.cb;
    mif.cb.data <= 7'h5a;       // Drive the bus
    @mif.cb;
    mif.cb.data <= 'z;          // Release the bus
  end

endprogram
```

Randomisation

You need to think broadly about all design input such as the following.

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

Example 6-1 Simple random class

```
class Packet;
  // The random variables
  rand bit [31:0] src, dst, data[8];
  randc bit [7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new;  // Create a packet
  assert (p.randomize());
  transmit(p);
end
```

Example 6-2 Constraint without random variables

```
class bad;
  bit [31:0] son;  // Error - should be rand or randc
  constraint c_teenager {son > 12;
                         son < 20;}
endclass
```

Example 6-3 Constrained-random class

```
class Stim;
  const bit [31:0] SRC_CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_t;
  randc stim_t type;   // Enumerated var
  rand bit [31:0] len, src, dst;
  bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    src inside {0, [2:10], [100:107]};
    if (congestion_test) {
      dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
    }
  }
endclass
```

Example 6-4 Constrain variables to be in a fixed order

```
class bad;
  rand bit [15:0] a, b, c;
  constraint good {0 < a;          // Correct way
                   a < b;
                   b < c;}
  constraint bad  {0 < a < b < c;} // Error, won't work!
endclass
```

Example 6-5 Random sets of values

```
rand int c;          // Random variable
int lo, hi;          // Nonrandom variables used as limits
constraint c_range {
  c inside {[lo:hi]};  // lo <= c and c <= hi
}
```

Example 6-6 Inverted random set constraint

```
constraint c_range {
  !(c inside {[lo:hi]});  // c < lo or c > hi
}
```

Example 6-10 Weighted random distribution with `dist`

```
rand int src, dst;
constraint c_dist {
    src dist {0:=40, [1:3]:=60};
    // src = 0, weight = 40/220
    // src = 1, weight = 60/220
    // src = 2, weight = 60/220
    // src = 3, weight = 60/220

    dst dist {0:/40, [1:3]:/60};
    // dst = 0, weight = 40/100
    // dst = 1, weight = 20/100
    // dst = 2, weight = 20/100
    // dst = 3, weight = 20/100
}
```

Example 6-12 Bidirectional constraint

```
rand logic [15:0] b, c, d;
constraint c_bidir {
  b < d;
  c == b;
  d < 30;
  c > 25;
}
```

Table 6-1. Solutions for bidirectional constraints

Solution	*b*	*c*	*d*
1	26	26	27
2	27	27	28
3	28	28	29

The SystemVerilog solver looks at all four constraints simultaneously. `b` has to be less than `d`, which has to be less than 30. But `b` is constrained to be equal to `c`, which is greater than 25. Even though there is no direct constraint on the lower value of `d`, the constraint on `c` restricts the choices.

Example 6-14 Constraint block with `if-else` operator

```
class BusOp;
  ...
  constraint c_len_rw {
    if (op == READ)
      len inside {[BYTE:LWRD]};
    else
      len == LWRD;
  }
```

Example 6-20 Class with implication and `solve...before`

```
class SolveBefore;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
    (x==0) -> y==0;
    solve x before y;
  }
endclass
```

Example 6-21 Using `constraint_mode`

```
class Packet;
  rand int length;
  constraint c_short {length inside {[1:32]}; }
  constraint c_long  {length inside {[1000:1023]}; }
endclass

Packet p;
initial begin
  p = new;

  // Create a long packet by disabling short constraint
  p.c_short.constraint_mode(0);
  assert (p.randomize());

  transmit(p);

  // Create a short packet by disabling all constraints
  // then enabling only the short constraint
  p.constraint_mode(0);
  p.c_short.constraint_mode(1);
  assert (p.randomize());
  transmit(p);
end
```

Example 6-23 The `randomize() with` statement

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t = new();

initial begin
  int s;
  t = new();

  // addr is 50-100, 1000-1500, data < 10
  assert(t.randomize() with {addr >= 50; addr <= 1500;
                             data < 10;});

  driveBus(t);

  // force addr to a specific value, data > 10
  assert(t.randomize() with {addr == 2000; data > 10;});

  driveBus(t);
  end
```

Example 6-30 Class with an external constraint

```
// packet.sv
class Packet;
  rand bit [7:0] length;
  rand bit [7:0] payload[];
  constraint c_valid {length > 0;
                      payload.size == length;}
  constraint c_external;
endclass
```

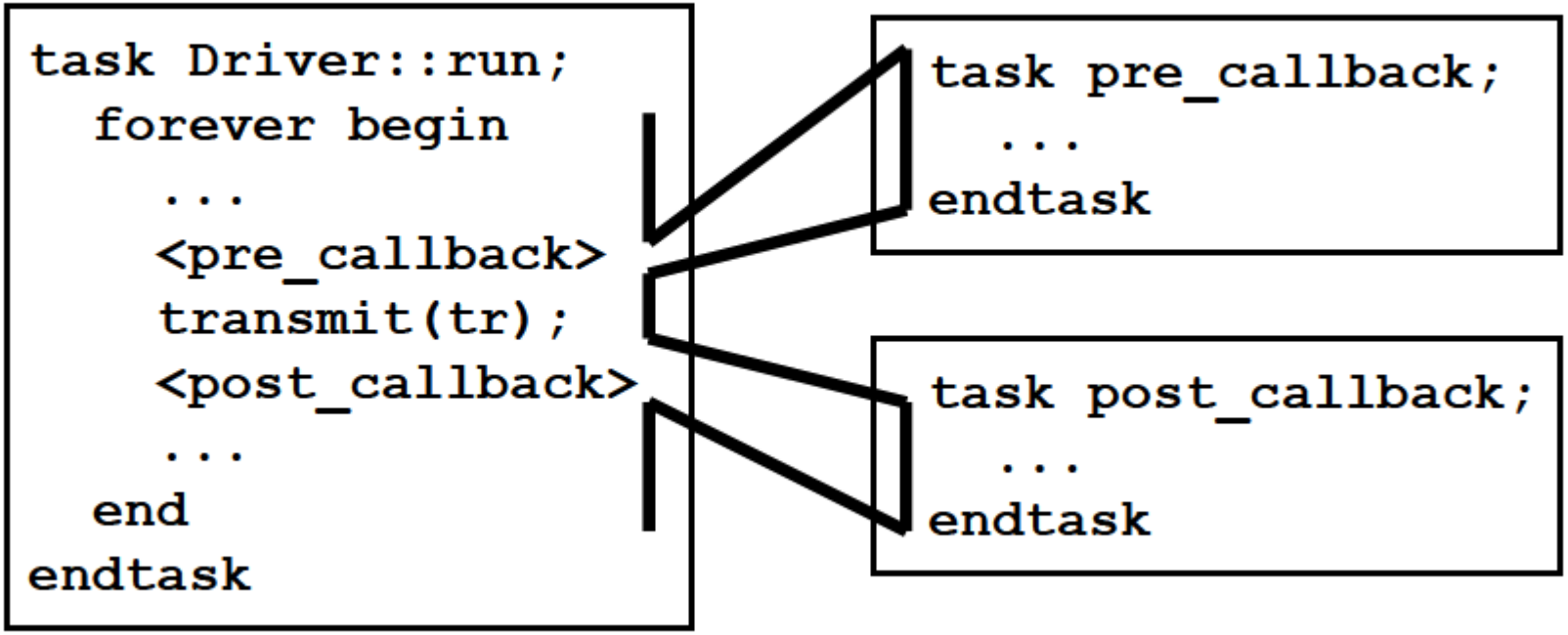
Example 6-31 Program defining external constraint

```
// test.sv
program test;
constraint Packet::c_external {length == 1;}
...
endprogram
```

Call Backs

- Inject errors
- Drop the transaction
- Delay the transaction
- Synchronize this transaction with others
- Put the transaction in the scoreboard
- Gather functional coverage data

Figure 8-8 Callback flow



A callback task is created in the top-level test and called from the driver, the lowest level of the environment. However, the driver does not have to have any knowledge of the test – it just has to use a generic class that the test can extend. The driver uses a queue to hold the callback objects, which simplifies adding new objects.

Example 8-24 Base callback class

```
class Driver_cbs;  // Driver callbacks
  virtual task pre_tx(Transaction tr, ref bit drop);
    // By default, callback does nothing
  endtask

  virtual task post_tx(Transaction tr);
    // By default, callback does nothing
  endtask
endclass
```

Example 8-25 Driver class with callbacks

```
class Driver;
  Driver_cbs cbs[$];

  task run;
    bit drop;
    Transaction tr;

    forever begin
      agt2drv.get(tr);
      foreach (cbs[i]) cbs.pre_tx(tr, drop);

      if (!drop) transmit(tr);

      foreach (cbs[i]) cbs.post_tx(tr);
    end
  endtask
endclass
```

A common use for a callback is to inject some disturbance such as causing an error or delay. The following testbench randomly drops packets using a callback object. Callbacks can also be used to send data to the scoreboard or to gather functional coverage values. Note that you can use `push_back()` or `push_front()` depending on the order in which you want these to be called.

```
class Driver_cbs_drop extends Driver_cbs;

  virtual task pre_tx(Transaction tr, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask

endclass


program automatic test;

  Environment env;

  initial begin
    env = new;
    env.gen_cfg;
    env.build;

    // Callback injection
    begin
    Driver_cbs_drop dcd;
      dcd = new;                      // Create scb callback
      env.drv.cbs.push_back(dcd); // Put into driver's Q
    end

    env.run;
    env.wrapup;
  end
```

Verification Process

Pankaj Badhe

- Verification is a process, not a set of testbenches.
- Verification can be only accomplished through an independent path between a specification and an implementation. It is important to understand where that independence starts and to know what is being verified.
- Verification can be performed at various levels of the design hierarchy, with varying degrees of visibility within those hierarchies. A black-box approach because it yields portable testbenches.
- Augment with grey and white-box testbenches to meet your goals.
- Consider verification at the beginning of the design. If a function would be difficult to verify, modify the design to give the necessary observability and controllability over the function.
- Make your verification components reusable across different testbenches, across block and system-level testbenches and across different projects.

- Verification time can be reduced through abstraction
- Using abstraction reduces control over low-level details.
- Lower levels of abstraction must remain visible.
- The testbench can switch to a lower level of abstraction to inject a physical-level parity error to verify that protocol-level operations remain unaffected.
- Equivalence checking (Formal Verification) compares two models.
- Equivalence checking compares two netlists to ensure that some netlist post-processing, such as scanchain insertion, clock-tree synthesis or manual modification, did not change the functionality of the circuit.

Black-Box Verification

- Black-box verification cannot look at or know about the inside of a design.
- All verification is accomplished through the available interfaces, without direct access to the internal state of the design, without knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability.
- The advantage of black-box verification is that it does not depend on any specific implementation.
- It is difficult to set up an interesting state combination or to isolate some functionality. CONTROLLABITY MAY BE CHALLENGING
- It is equally difficult to observe the response from the input and locate the source of the problem.

White-Box Verification

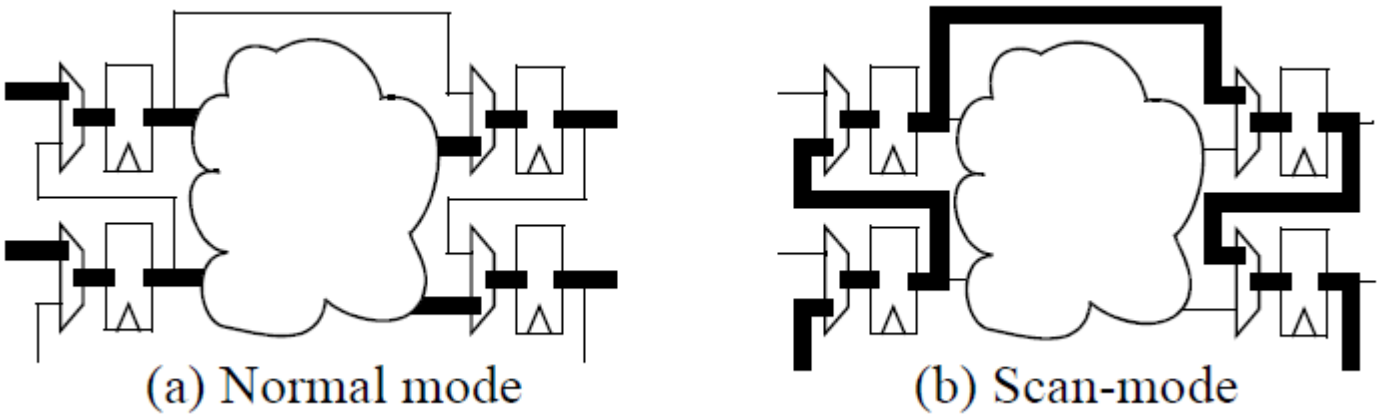
- White box verification has intimate knowledge and control of the internals of a design.
- White-box verification is tied to a specific implementation.
- Changes in the design may require changes in the testbench.
- Assertions are ideal for implementing white-box checks in RTL code.

Grey-Box Verification

- As in black-box verification, a grey-box approach controls and observes a design entirely through its top-level interfaces. However, the particular verification being accomplished is intended to exercise significant features specific to the implementation.
- A typical grey-box test case is one written to increase coverage metrics.
- Add functions to the design to increase controllability and observability

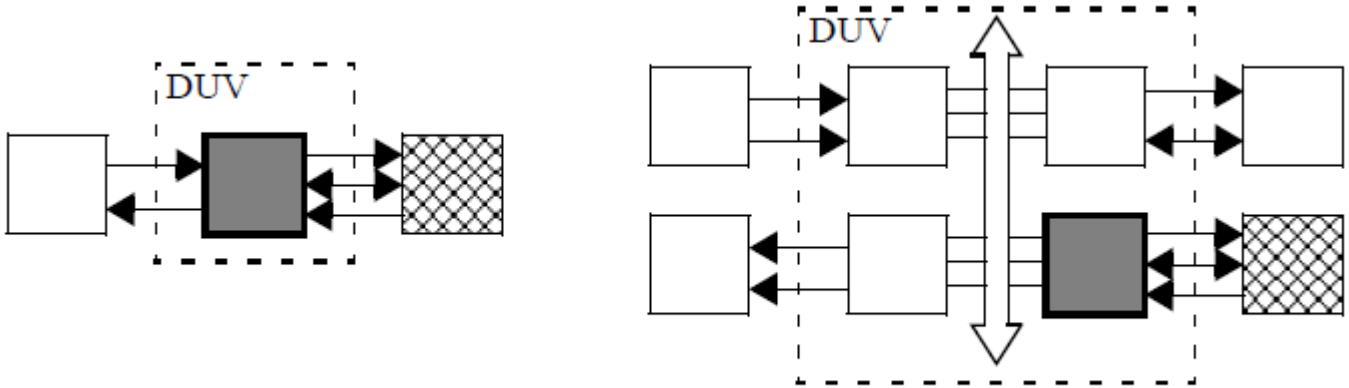
Scan-Based Testing

- Linking all registers into a long shift register increases controllability and observability.



Verification Reuse

- Use of Verification Components?



(a) Block testbench (b) System testbench

LINTING

- Linting technology finds common programmer mistakes.
- Problems are found faster than at runtime.

1. The *my_func* function is called with only one argument instead of two.

2. The *my_func* function is called with an integer value as a first argument instead of a pointer to an integer.

- Linting does not require stimulus.
- Many false negatives are reported. Naming conventions can help output filtering

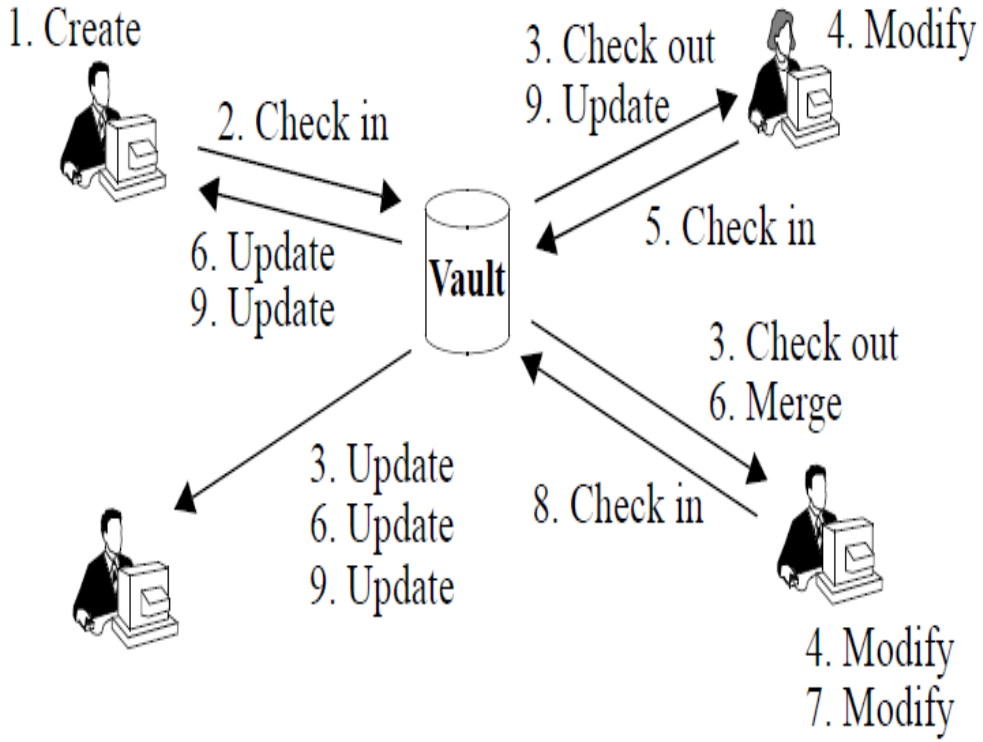
```
Warning: file decoder.sv, line 23: Latch
inferred on reg "address_lat".
Warning: file decoder.sv, line 36: Latch
inferred on reg "next_state".
```

Coverage

- Most important parameter - Code
- Functional Coverage
- Assertion Coverage
- Metrics for Coverage

REVISION CONTROL

- Why?
- How?
- GIT / CVS
- Continuous Integration
- Regression



Configuration Management

Tag Name	Description
Submit	Ready to submit to functional verification. Author has verified syntax correctness and basic level of functionality.
Bronze	Passes a basic set of functional testcases. Release is sufficiently functional for integration.
Silver	Passes all functional testcases.
Gold	Passes all functional testcases and meets coding coverage guidelines (requires additional corner-case testcases).
To_Synthesis	Ready to submit to synthesis. Usually matches "Silver" or "Gold".
To_Layout	Ready to submit to layout. Usually matches "Gold".
Version_M.N	Version that was manufactured. Matches corresponding "To_Layout" release. Future versions of the same chip will move tags beyond this point.
ON_YYMMDD	Some meaningful release on the specified date.

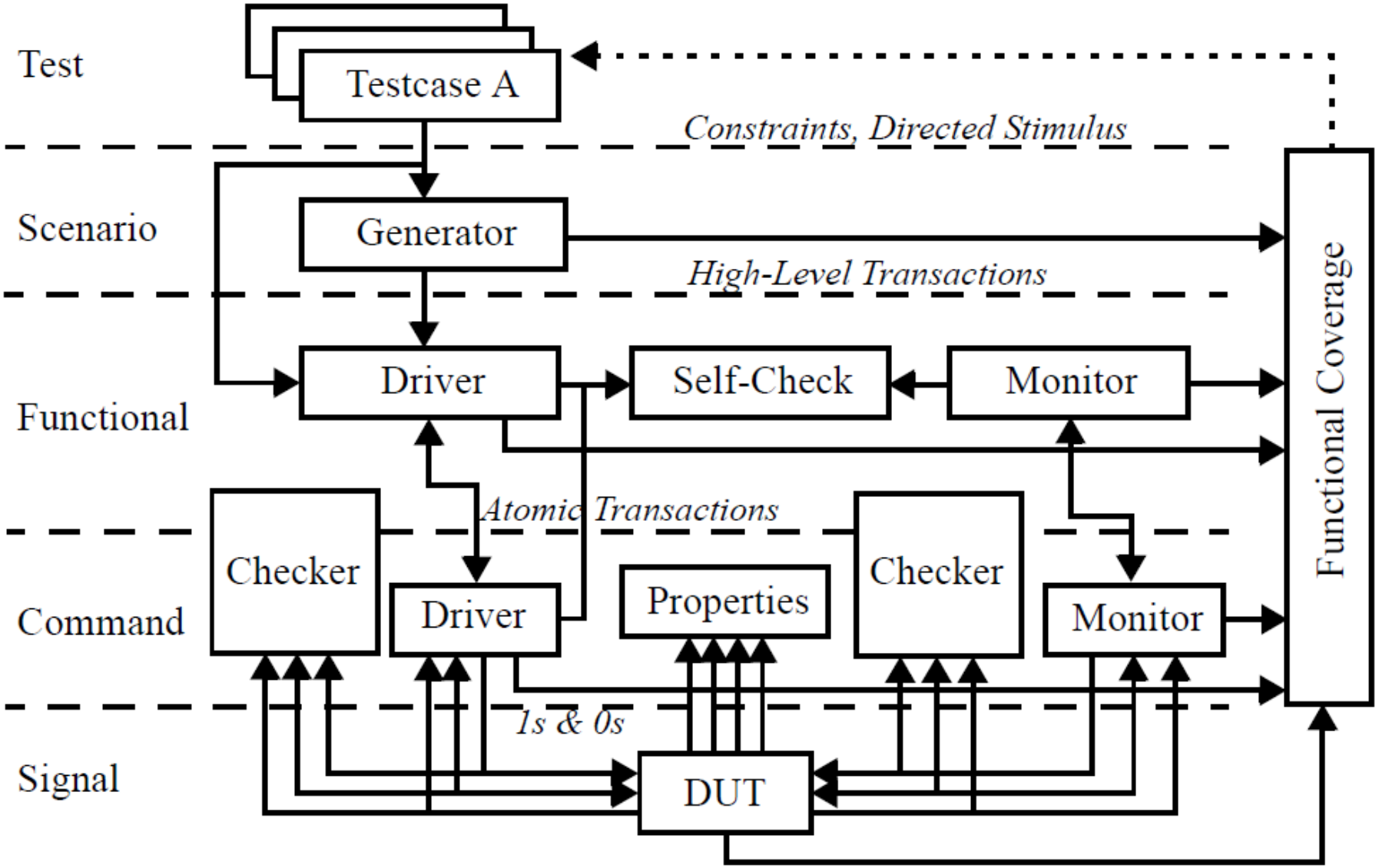


Figure 4-2. Layered Verification Environment Architecture.

Coverage and Assertions

Pankaj Badhe

	Implementation	Specification
Implicit	Code Coverage	Area of research (Currently no industrial solutions)
Explicit	Assertions	Functional Coverage Assertions

CODE COVERAGE

- Code coverage is a technology that can identify what code has been (and more importantly *not* been) executed in the design under verification.
- Only the code for the design under verification needs to be covered and thus instrumented
- Trace information is collected at runtime.

Statement Coverage

- Statement coverage can also be called block coverage, where a block is a sequence of statements that are executed if a single statement is executed.
- But block boundaries may not be that obvious.

Sample 2-8. Block vs. statement execution

```
if (dtack == 1'b1) begin: acked
   as     <= 1'b0;
   data   <= 16'hZZZZ;
   bus_rq <= 1'b0;
   state  <= IDLE;
end: acked
```

Sample 2-9. Blocks separated by a *wait* statement

```
address <= 16'hFFED;
ale     <= 1'b1;
rw      <= 1'b1;
wait (dtack == 1'b1);
read_data = data;
ale     <= 1'b0;
```

Figure 2-13. Example of statement coverage

```
☑ if (parity == ODD || parity == EVEN) begin
☐     tx <= compute_parity(data, parity);
☐     #(tx_time);
  end
☑ tx <= 1'b0;
☑ #(tx_time);
☑ if (stop_bits == 2) begin
☑     tx <= 1'b0;
☑     #(tx_time);
  end
```

Path Coverage / Expressions Coverage ???

- There is more than one way to execute a sequence of statements.

Sample 2-10. Example of statement and path coverage

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

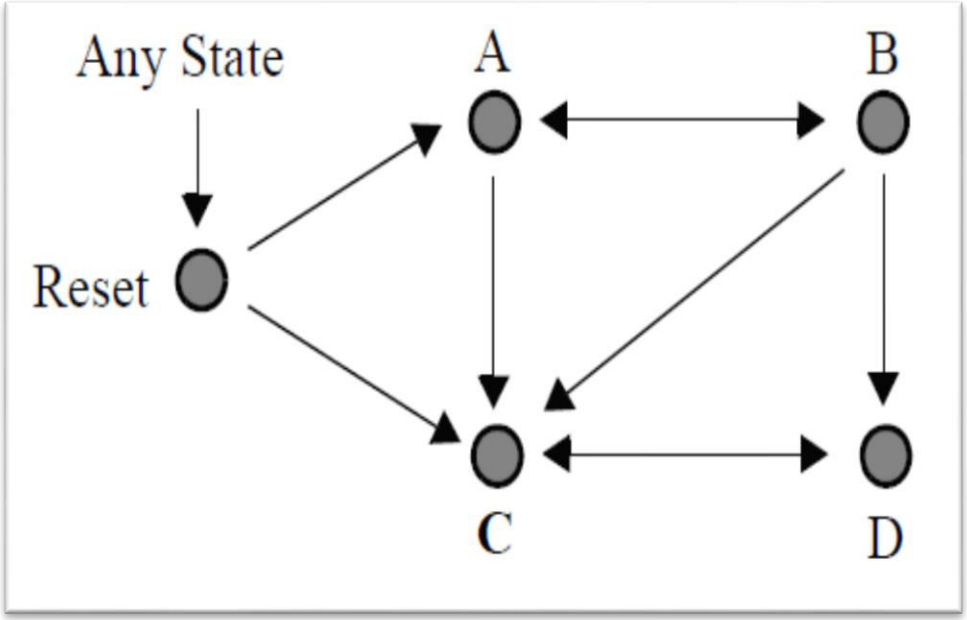
Expression Coverage

- If you look closely at the code in Sample 2-11, you notice that there are two mutually independent conditions that can cause the first *if* statement to branch the execution into its *then* clause: parity being set to either ODD or EVEN.
- Expression coverage, as shown in Sample 2-11, measures the various ways decisions through the code are made.
- Even if the statement coverage is at 100 percent, the expression coverage is only at 50 percent.

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

FSM Coverage

- FSMs are identified by tools based on coding guidelines
- States - 5
- Arcs ??
- A property checker tool may be able to determine if one-hot, is never violated.



FUNCTIONAL COVERAGE

- Functional coverage records relevant metrics (e.g., packet length, instruction opcode, buffer occupancy level) to ensure that the verification process has exercised the design through all of the interesting values.
- Whereas code coverage measures how much of the implementation has been exercised, functional coverage measures how much of the original design specification has been exercised.

Sample 2-12. Example of coding error undetectable by code coverage

```
enum {ADD, SUB, JMP, RTS, NOP} opcode;
...
case (opcode)
   ADD: ...
   SUB: ...
   JMP: ...
   default: ...
endcase
```

Coverage Points

- A coverage point is the sampling of an individual scalar value or expression. The objective of a coverage point is to ensure that all interesting and relevant values have been observed in the sampled value or expression. Examples of coverage points include, but are not limited to, packet length, instruction opcode, interrupt level, bus transaction termination status, buffer occupancy level, bus request patterns and so on.
- Define *what* to sample. For example, measuring the value of the read and write pointers in a FIFO is fine if you are concerned about the full utilization of the buffer space and wrapping around of the pointer values. But if you are interested in the FIFO occupancy level (Was it ever empty? Was it ever full? Did it overflow?), you should measure and record the *difference* between the pointer values.

- Define *where* to sample it.
- You must decide where in your testbench or design is the measured value accurate and relevant. For example, you can sample the opcode of an instruction at several places: at the output of the code generator, at the interface of the program memory, in the decoder register or in the execution pipeline. You have to ensure that a value, once recorded, is indeed processed or committed as implied by the coverage metric.
- For example, if you are measuring opcodes that were executed, they should be sampled in the execution unit. Sampling them in the decode unit could result in false samples when the decode pipeline is flushed on branches or exceptions. Similarly, sampling the length of packets at the output of the generator may yield false samples: If a packet is corrupted by injecting an error during its transmission to the design in lower-level functions of the testbench, it may be dropped.

- Define *when* to sample it.
- You must carefully choose your sampling time. Over sampling will decrease simulation performance and consume database resources without contributing additional information. The sampled data must also be stable so race conditions must be avoided.

- Define *why* it is covered.
- measuring the functional coverage of a 32-bit address will yield over 4 billion “interesting and relevant” values.
- For example, based on the decoder architecture, addresses 0x00000001 through 0x7FFFFFFF and addresses 0x80000000 through 0x8FFFFFFE are functionally equivalent, reducing the number of relevant and interesting values to 4 bins (min, 1 to mid, mid to max-1, max).

- It can detect invalid values.
- It can report holes.
- **Cross Coverage**
  - Did I generate all interesting *combinations* of values?
- **Transition Coverage**
  - Did I generate all interesting *sequences* of values?

```
program automatic test(busifc.TB ifc);

  class Transaction;
    rand bit [31:0] data;
    rand bit [ 2:0] port;      // Eight port numbers
  endclass

  covergroup CovPort;
    coverpoint tr.port;        // Measure coverage
  endgroup

  Transaction tr = new;

  initial begin
    CovPort ck = new;          // Instantiate group

    repeat (32) begin          // Run a few cycles
      assert(tr.randomize);    // Create a transaction
      ifc.cb.port <= tr.port;  //   and transmit
      ifc.cb.data <= tr.data;  //   onto interface
      ck.sample();             // Gather coverage
      @ifc.cb;                 // Wait a cycle
    end
  end
endprogram
```

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 87.50
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 7
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin              # hits     at least
    ==================================
    auto[1]          7          1
    auto[2]          7          1
    auto[3]          1          1
    auto[4]          5          1
    auto[5]          4          1
    auto[6]          2          1
    auto[7]          6          1
    ==================================
```

- As you can see, the testbench generated the values 1, 2, 3, 4, 5, 6, and 7, but never generated a port of 0. The at least column specifies how many hits are needed before a bin is considered covered.

Example 9-5 Functional coverage inside a class

```
class Transactor;
  Transaction tr;
  mailbox mbx_in;
  covergroup CovPort;
    coverpoint tr.port;
  endgroup

  function new(mailbox mbx_in);
    CovPort = new;              // Instantiate covergroup
    this.mbx_in = mbx_in;
  endfunction

  task main;
    forever begin
      tr = mbx_in.get;          // Get next transaction
      ifc.cb.port <= tr.port;   // Send into DUT
      ifc.cb.data <= tr.data;
      CovPort.sample();         // Gather coverage
    end
  endtask

endclass
```

Example 9-6 Test using functional coverage callback

```
program automatic test;
  Environment env;

  initial begin
    Driver_cbs_coverage dcc;

    env = new;
    env.gen_cfg;
    env.build;

    // Create and register the coverage callback
    dcc = new;
    env.drv.cbs.push_back(dcc); // Put into driver's Q

    env.run;
    env.wrapup.
  end

endprogram
```

Example 9-7 Callback for functional coverage

```
class Driver_cbs_coverage extends Driver_cbs;
  covergroup CovPort;
    ...
  endgroup

  virtual task post_tx(Transaction tr);
    CovPort.sample();          // Sample coverage values
  endtask
endclass
```

Example 9-8 Cover group with a trigger

```
event trans_ready;
covergroup CovPort @(trans_ready);
    coverpoint ifc.cb.port;   // Measure coverage
endgroup
```

- The cover group option auto_bin_max specifies the maximum number of bins to automatically create, with a default of 64 bins. If the domain of values in the cover point variable or expression is greater than this option, SystemVerilog divides the range into auto_bin_max bins.
- For example, a 16-bit variable has 65,536 possible values, so each of the 64 bins covers 1024 values.

Example 9-11 Using auto_bin_max set to 2

```
covergroup CovPort;
  coverpoint tr.port
    { options.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

Example 9-12 Report with auto_bin_max set to 2

```
Bin                 # hits     at least
=====================================
auto[0:3]           15         1
auto[4:7]           17         1
=====================================
```

Example 9-14 Using an expression in a cover point

```
class Transaction;
  rand bit [2:0] hdr_len;       // range: 0:7
  rand bit [3:0] payload_len;   // range: 0:15
  ...
endclass

Transaction tr;

covergroup CovLen;
  len16: coverpoint (tr.hdr_len + tr.payload_len);
  len32: coverpoint (tr.hdr_len + tr.payload_len + 5'b0);
endgroup
```

Example 9-15 Defining bins for transaction length

```
covergroup CovLen;
  len: coverpoint (tr.hdr_len + tr.payload_len + 5'b0)
    {bins len[] = {[0:23]}; }
endgroup
```

Example 9-17 Specifying bin names

```
covergroup CovKind;
  coverpoint tr.kind {
    bins zero = {0};        // A bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for all the rest
  }
endgroup // CoverKind
```

Example 9-18 Report showing bin names

```
Bin          # hits    at least
==============================
hi_8          0          1
hi_9          5          1
hi_a          3          1
hi_b          4          1
hi_c          2          1
hi_d          2          1
hi_e          9          1
hi_f          4          1
lo           16          1
misc         15          1
zero          1          1
==============================
```

Example 9-19 Conditional coverage — disable during reset

```
covergroup CoverPort;
  // Don't gather coverage when reset==1
  coverpoint port iff (!bus_if.reset);
endgroup
```

Example 9-20 Using `stop` and `start` functions

```
initial begin
  CovPort ck = new;          // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns bus_if.reset = 1;
  ck.stop();
  #100ns bus_if.reset = 0; // End of reset
  ck.start();
  ...
end
```

Example 9-21 Functional coverage for an enumerated type

```
typedef enum {INIT, DECODE, IDLE} fsmstate_t;
fsmstate_t pstate, nstate;   // declare typed variables
covergroup cg_fsm;
  coverpoint pstate;
endgroup
```

Example 9-22 Report with `auto_bin_max` set to 2

```
Bin                 # hits     at least
====================================
auto_DECODE         11         1
auto_IDLE           11         1
auto_INIT           10         1
====================================
```

Example 9-23 Specifying transitions for a cover point

```
covergroup CoverPort;
  coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

Example 9-25 Cover point with `ignore_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    ignore_bins hi = {[6,7]}; // Ignore upper 2 bins
  }
endgroup
```

Example 9-26 Cover point with `auto_bin_max` and `ignore_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    options.auto_bin_max = 4; // 0:1, 2:3, 4:5, 6:7
    ignore_bins hi = {[6,7]}; // Ignore upper 2 values
  }
endgroup
```

Example 9-27 Cover point with `illegal_bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    illegal_bins hi = {[6,7]}; // Give error if seen
  }
endgroup
```

Example 9-28 Basic cross coverage

```
class Transaction;
  rand bit [3:0] kind;
  rand bit [2:0] port;
endclass

Transaction tr;

covergroup CovPort;
  kind: coverpoint tr.kind;  // Create cover point kind
  port: coverpoint tr.port   // Create cover point port
  cross kind, port;          // Cross kind and port
endgroup
```

```
Cross Coverage report
CoverageGroup: Transaction::CovPort
    Cross: Transaction::CovPort
Summary
    Coverage: 87.50
    Goal: 100
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 128
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 112

    Automatically Generated Cross Bins

    kind        port            # hits        at least
 ==================================================
    auto[0]     auto[0]         1             1
    auto[0]     auto[1]         4             1
    auto[0]     auto[2]         3             1
    auto[0]     auto[5]         1             1
...
```

Example 9-30 Specifying cross coverage bin names

```
covergroup CovPortKind;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {bins zero = {0};          // A bin for kind==0
    bins lo   = {[1:3]};      // 1 bin for values 1:3
    bins hi[] = {[8:$]};      // 8 separate bins
    bins misc = default;      // 1 bin for all the rest
  }
  cross kind, port;
endgroup
```

```
    port            kind            # hits        at least
==================================================================
    port_0          hi_8            3             1
    port_0          hi_a            1             1
    port_0          hi_b            4             1
    port_0          hi_c            4             1
    port_0          hi_d            4             1
    port_0          hi_e            1             1
    port_0          lo              7             1
    port_0          misc            6             1
    port_0          zero            1             1
    port_1          hi_8            3             1
...
```

Example 9-32 Excluding bins from cross coverage

```
covergroup CovCovport;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {
    bins zero = {0};          // A bin for kind==0
    bins lo   = {[1:3]};      // 1 bin for values 1:3
    bins hi[] = {[8:$]};      // 8 separate bins
    bins misc = default;      // 1 bin for all the rest
  }
  cross kind, port {
    ignore_bins hi = binsof(port) intersect {7};
    ignore_bins md = binsof(port) intersect {0} &&
                     binsof(kind) intersect {[9:10]};
    ignore_bins lo = binsof(kind.lo);
  }
endgroup
```

The first **ignore_bins** just excludes bins where **port** is 7 and any value of **kind**. Since **kind** is a 4-bit value, this statement excludes 16 bins. The second **ignore_bins** is more selective, ignoring bins where **port** is 0 and **kind** is 9, 10, or 11, for a total of 3 bins.

Assertions

- Assertions detect conditions that should always be true.
- Hardware assertions require a form of temporal language.
- There are two classes of assertions.
  - Implementation assertions are specified by the designers.
  - Specification assertions are specified by the verification engineers.

APB3 Protocol Test Plan

Section	Description	Coverage Type	Priority
Unknown Signal Value Checks			
PRESETn	PRESETn is always in a known state	Assertion	1
PSEL	PSEL is always in a known state	Assertion	1
PADDR	PADDR is valid when PSEL is active	Assertion	1
PWRITE	PWRITE is valid when PSEL is active	Assertion	1
PENABLE	PENABLE is valid when PSEL is active	Assertion	1
PWDATA	PWDATA is valid when PWRITE is active	Assertion	1
PREADY	PREADY is valid when PENABLE is active	Assertion	1
PSLVERR	PSLVERR is valid when PENABLE is active	Assertion	1
PRDATA	PRDATA is valid when PREADY is asserted	Assertion	1
Timing Relationship checks			
PENABLE de-assertion	PENABLE is de-asserted once PREADY becomes active	Assertion, Cover directive	1
PSEL to PENABLE	There is only one clock delay between PSEL and PENABLE	Assertion, Cover directive	1
Signal Stability	When PSEL becomes active, the PWRITE, PADDR, and PWDATA signals should be stable to the end of the cycle	Assertion, Cover directive	1
Other Checks			
PSEL Unique	Only one PSEL line is active at a time	Assertion	1
Functional Coverage			
APB3 Protocol	All types of APB3 protocol transfers have taken place with all types of response for all active PSEL lines	Covergroup	2

- *Implementation assertions* are used to formally encode the designer's assumptions about the interface or implementation of the design or conditions that are indications of misuse or design faults.
- For example, the designer of a FIFO would add assertions to detect if it ever overflows or underflows or that, because of a design limitation, the *write* and *read* pulses are ever asserted at the same time.
- Because implementation assertions are specified by the designer, they will not detect discrepancies between the functional intent and the design. But implementation assertions will detect discrepancies between the design assumptions and the implementation.

- *Specification assertions* formally encode expectations of the design based on the functional intent. These assertions are used as a functional error detection mechanism and supplement the error detections performed in the self-checking section of testbenches.
- Specification assertions are typically *white-box* strategies because the relationships between the primary inputs and outputs of a modern design are too complex to be described efficiently in System-Verilog's temporal languages. For example, rather than relying on the scoreboard to detect that an arbiter is not fair, it is much simpler to perform this check using a white-box assertion

Procedural assertions

Example 5-28 Checking a signal with an `if`-statement

```
bus.cb.request <= 1;
@bus.cb;
if (bus.cb.grant != 2'b01)
  $display("Error, grant != 1");
// rest of the test
```

Example 5-29 Simple procedural assertion

```
bus.cb.request <= 1;
@bus.cb;
a1: assert (bus.cb.grant == 2'b01);
// rest of the test
```

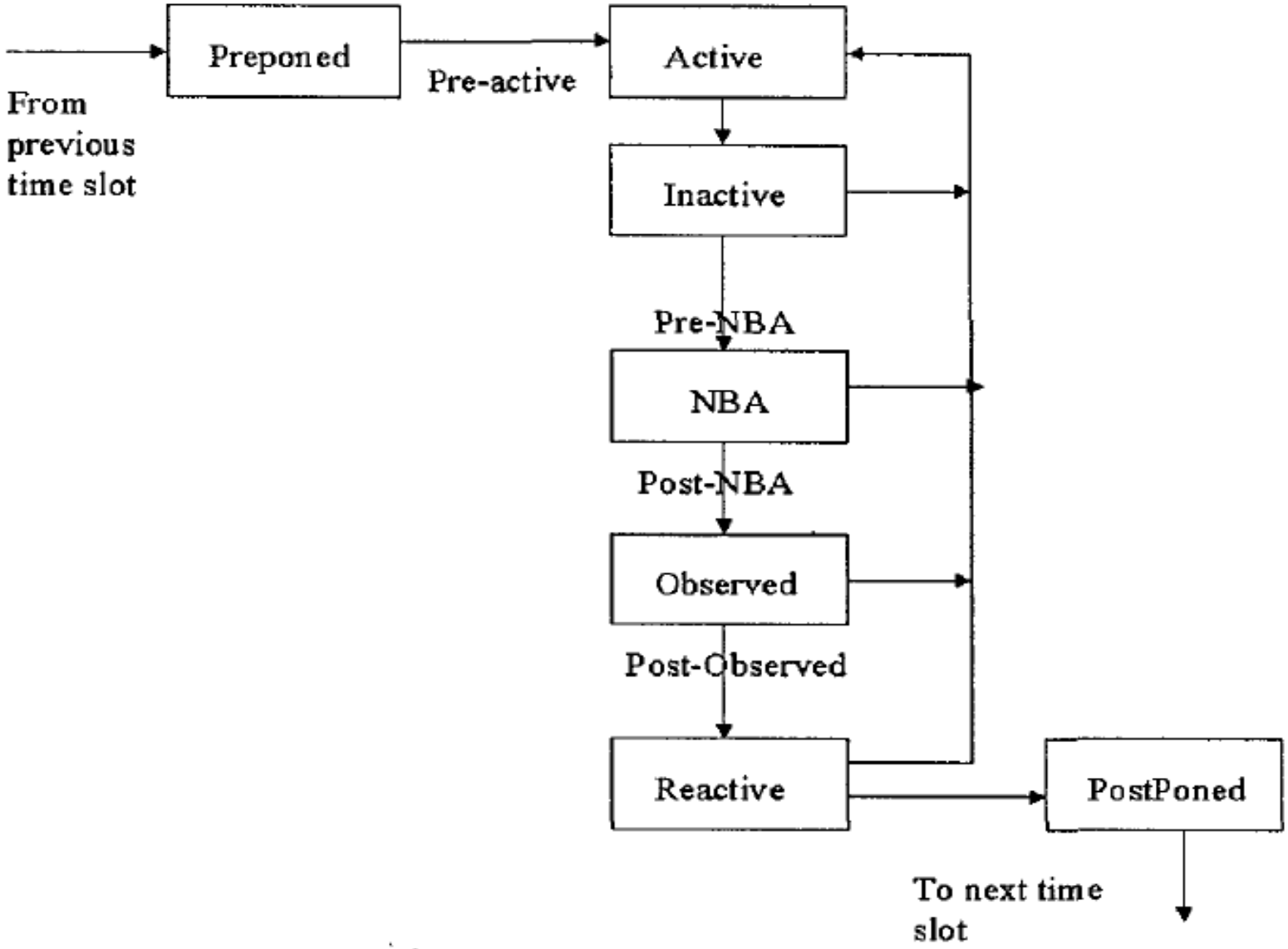
Example 5-30 Error from failed procedural assertion

```
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
```

Example 5-31 Creating a custom error message in a procedural assertion

```
a1: assert (bus.cb.grant == 2'b01)
else $error("Grant not asserted");
```

SystemVerilog has four functions to print messages: **$info**, **$warning**, **$error**, and **$fatal**. These are allowed only inside an assertion, not in procedural code, though future versions of SystemVerilog may allow this.

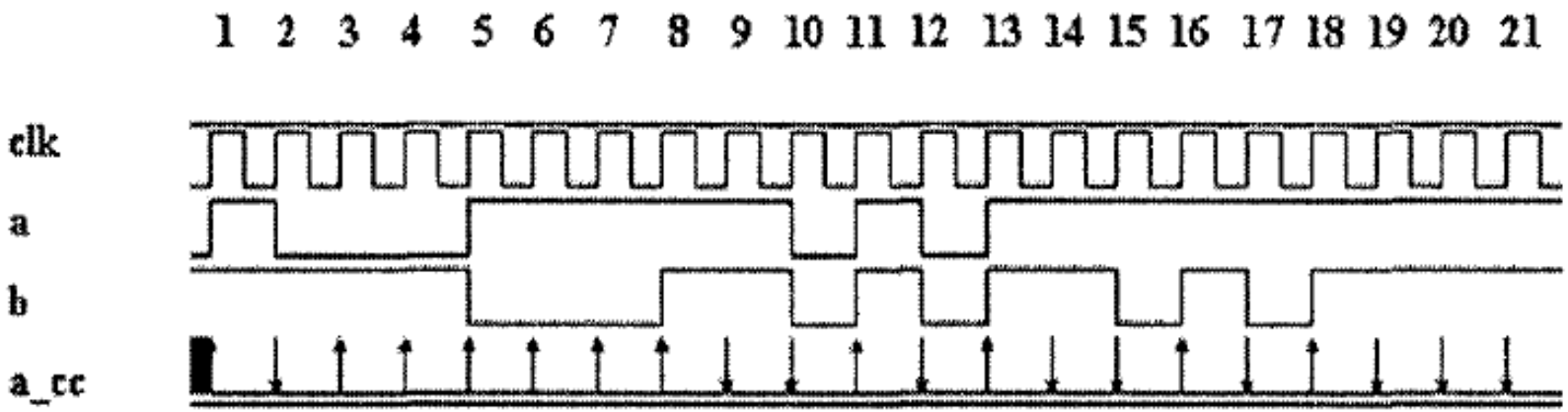

Preponed
Pre-active
Active
From previous time slot
Inactive
Pre-NBA
NBA
Post-NBA
Observed
Post-Observed
Reactive
PostPoned
To next time slot

- There are two types of assertions defined in the SystemVerilog language:
- Concurrent assertions
- Immediate assertions.

Concurrent assertions

- Based on clock cycles.
- Test expression is evaluated at clock edges based on the sampled values of the variables involved.
- Sampling of variables is done in the "preponed" region and the evaluation of the expression is done in the "observed" region of the scheduler.
- Can be placed in a procedural block, a module, an interface or a program definition.

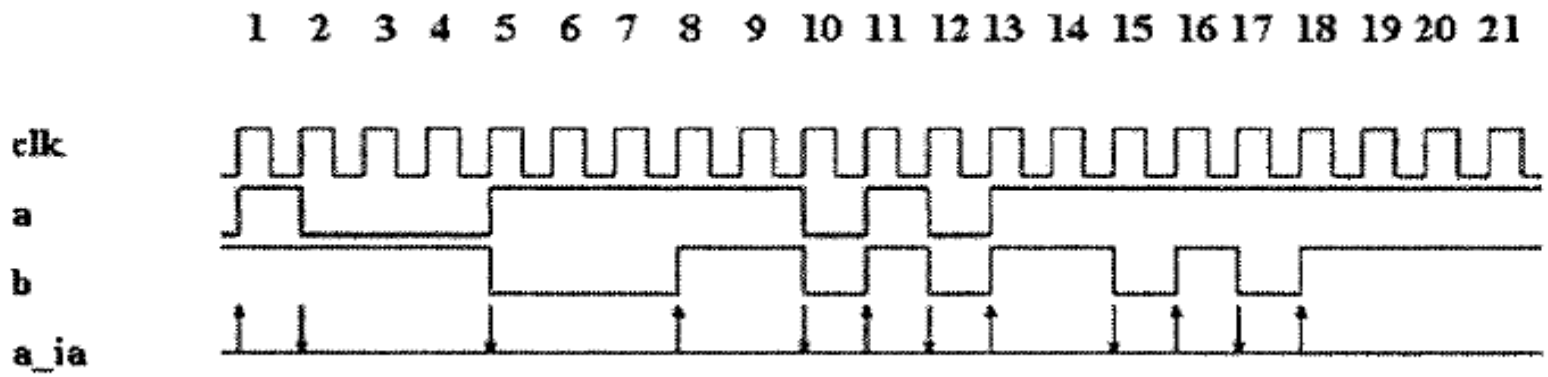
```
a_cc: assert property(@(posedge clk)
                                  not (a && b));
```



Immediate assertions

- Based on simulation event semantics.
- Test expression is evaluated just like any other Verilog expression within a procedural block. These are not temporal in nature and are evaluated immediately.
- Have to be placed in a procedural block definition.

```
always_comb
begin
   a_ia: assert (a && b);
end
```



Building blocks of SVA

- SVA provides a key word to represent these events called "sequence." The basic syntax of a sequence is as follows.

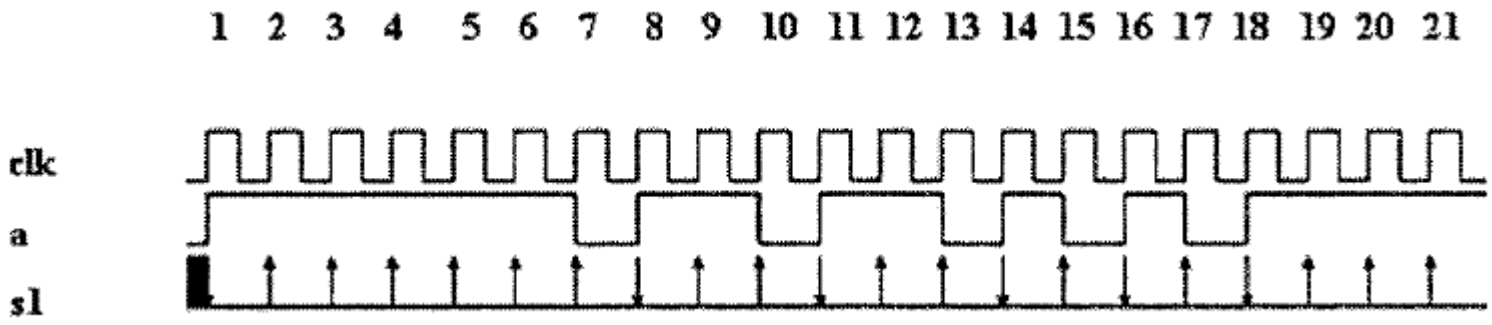
```
sequence name_of_sequence;
    < test expression>;
endsequence
```

- A number of sequences can be combined logically or sequentially to create more complex sequences. SVA provides a key word to represent these complex sequential behaviors called "property."
- The basic syntax of a property is as follows.

```
property name_of_property;
    < test expression >; or
    < complex sequence expressions >;
endproperty
```

A simple sequence

```
sequence s1;
  @(posedge clk) a;
endsequence
```



Sequence with edge definitions

$rose (*boolean expression or signal_name*)

- This returns true if LSB of signal/expression changed to 1

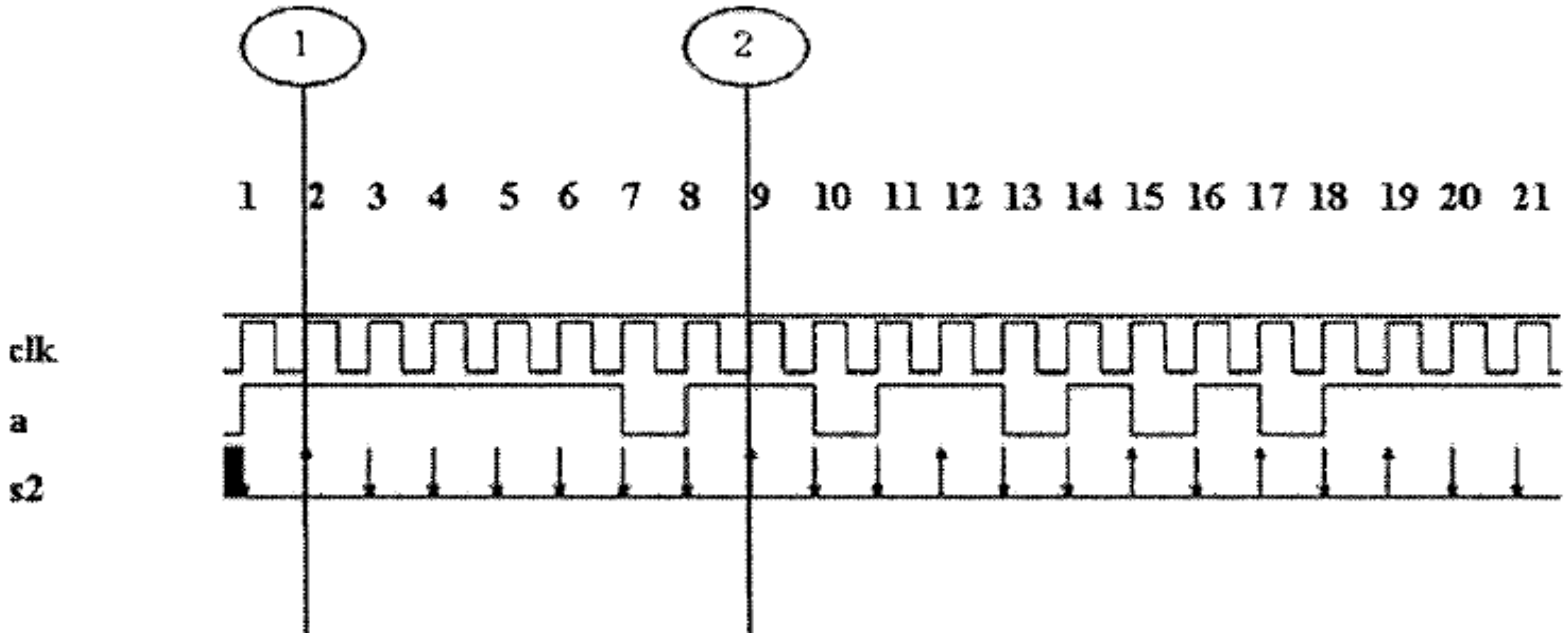
$fell (*boolean expression or signal_name*)

- This returns true if LSB of signal/expression changed to 0

$stable (*boolean expression or signal_name*)

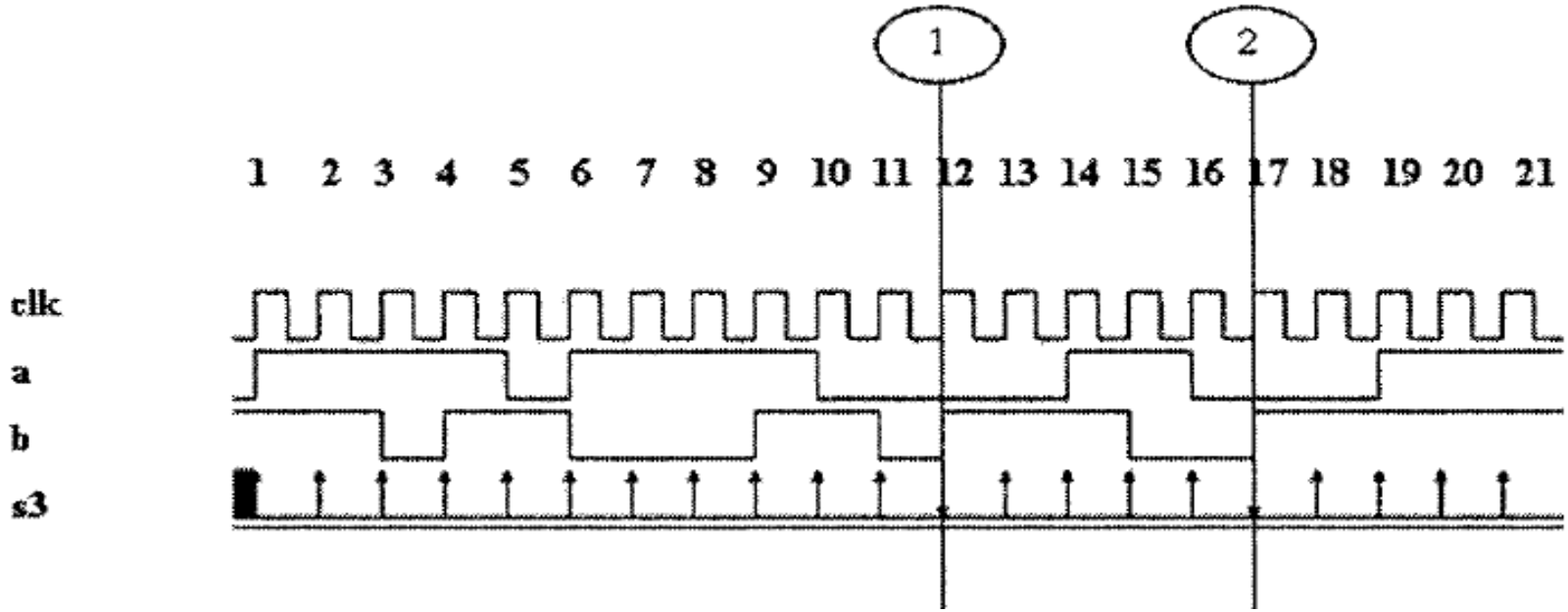
- This returns true if the value of the expression did not change

```
sequence s2;
 @(posedge clk) $rose(a);
endsequence
```



Sequence with logical relationship

```
sequence s3;
  @(posedge clk)  a || b;
endsequence
```



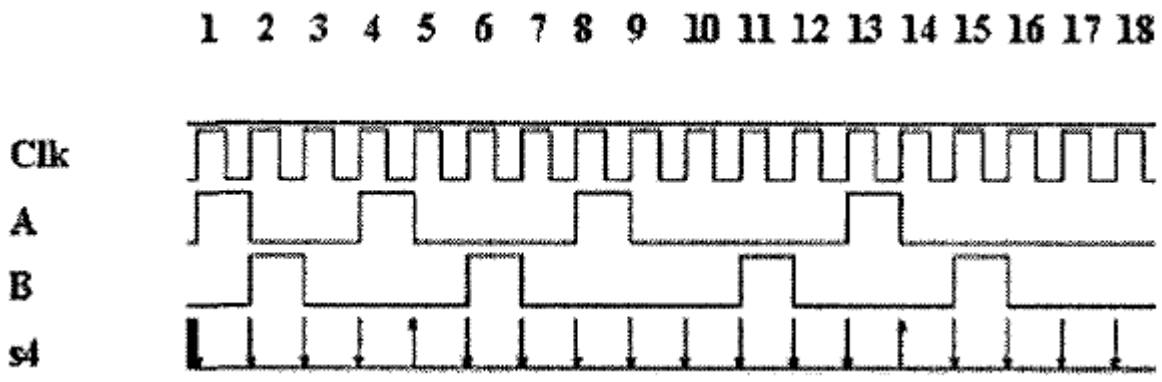
Sequence Expressions

```
sequence s3_lib (a, b);
   a || b;
endsequence
```

Sequences with timing relationship

- Simple boolean expressions are checked on every clock edge. In other words, they are simple combinational checks. A lot of times, we are interested in checking events that take several clock cycles to complete.
- These are called **"sequential checks."** ***In SVA, clock cycle delays are represented by a "##" sign. For example, ##3 means 3 clock cycles.***

```
sequence s4;
  @(posedge clk)  a ##2 b;
endsequence
```



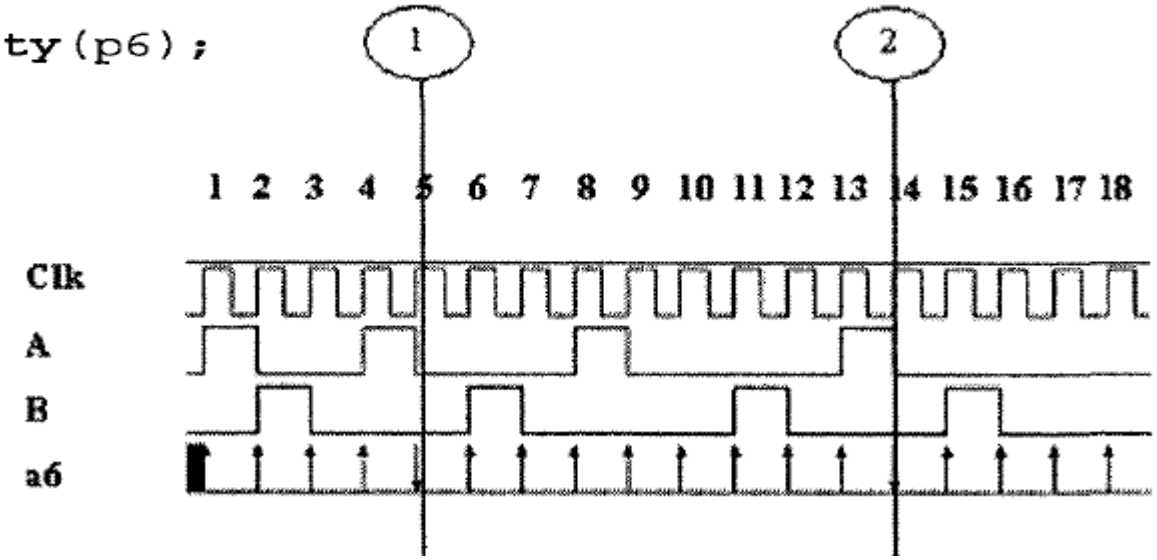
```
sequence s5;
 @(posedge clk)  a ##2 b;
endsequence

property p5;
 s5;
endproperty

a5 : assert property(p5);
```

```
sequence s5a;
 a ##2 b;
endsequence

property p5a;
 @(posedge clk)  s5a;
endproperty

a5a : assert property(p5a);
```

In general, it is a good idea to define the clocks in property definitions and keep the sequences independent of the clocks. This will help increase the re-use of the basic sequence definitions.

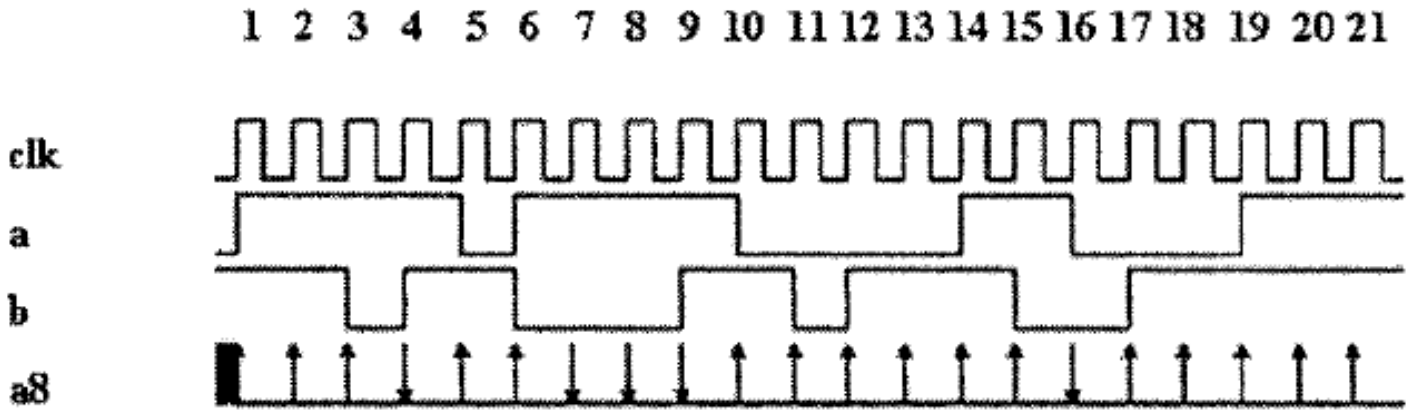
Forbidding a property

```
sequence s6;
  @(posedge clk)  a ##2 b;
endsequence

property p6;
  not s6;
endproperty

a6 : assert property(p6);
```



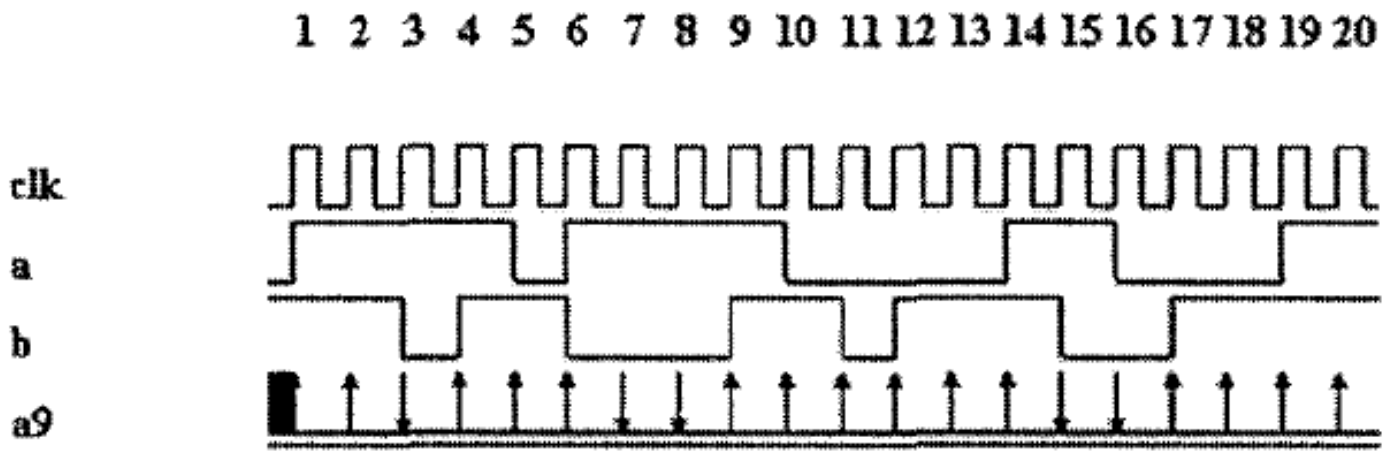
Overlapped implication

```
property p8;
  @(posedge clk)  a |-> b;
endproperty

a8 : assert property(p8);
```



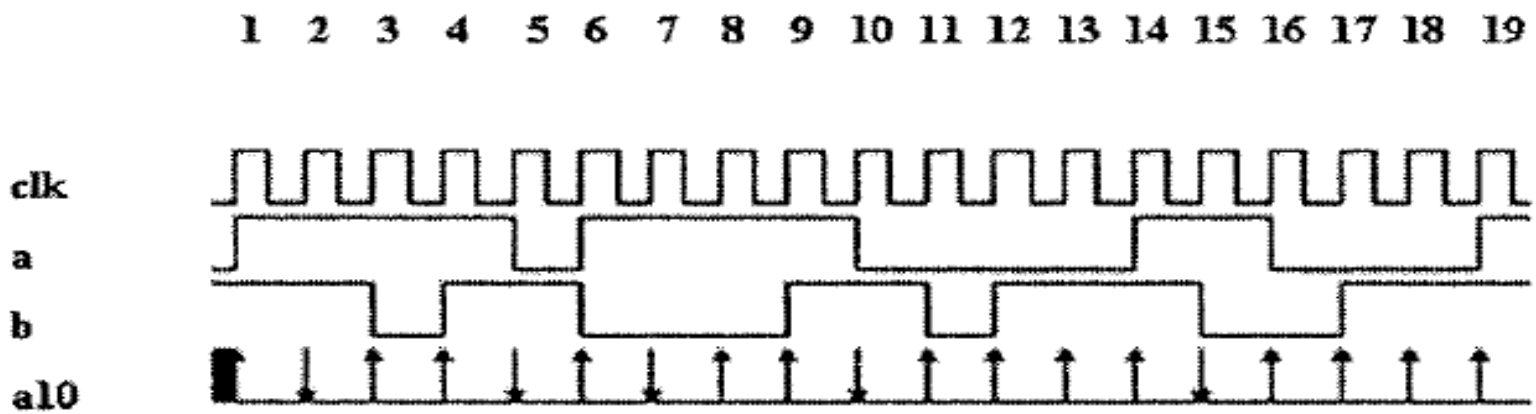
Non-overlapped implication

```
property p9;
 @(posedge clk)  a |=> b;
endproperty

a9 : assert property(p9);
```



Implication with a fixed delay on the consequent

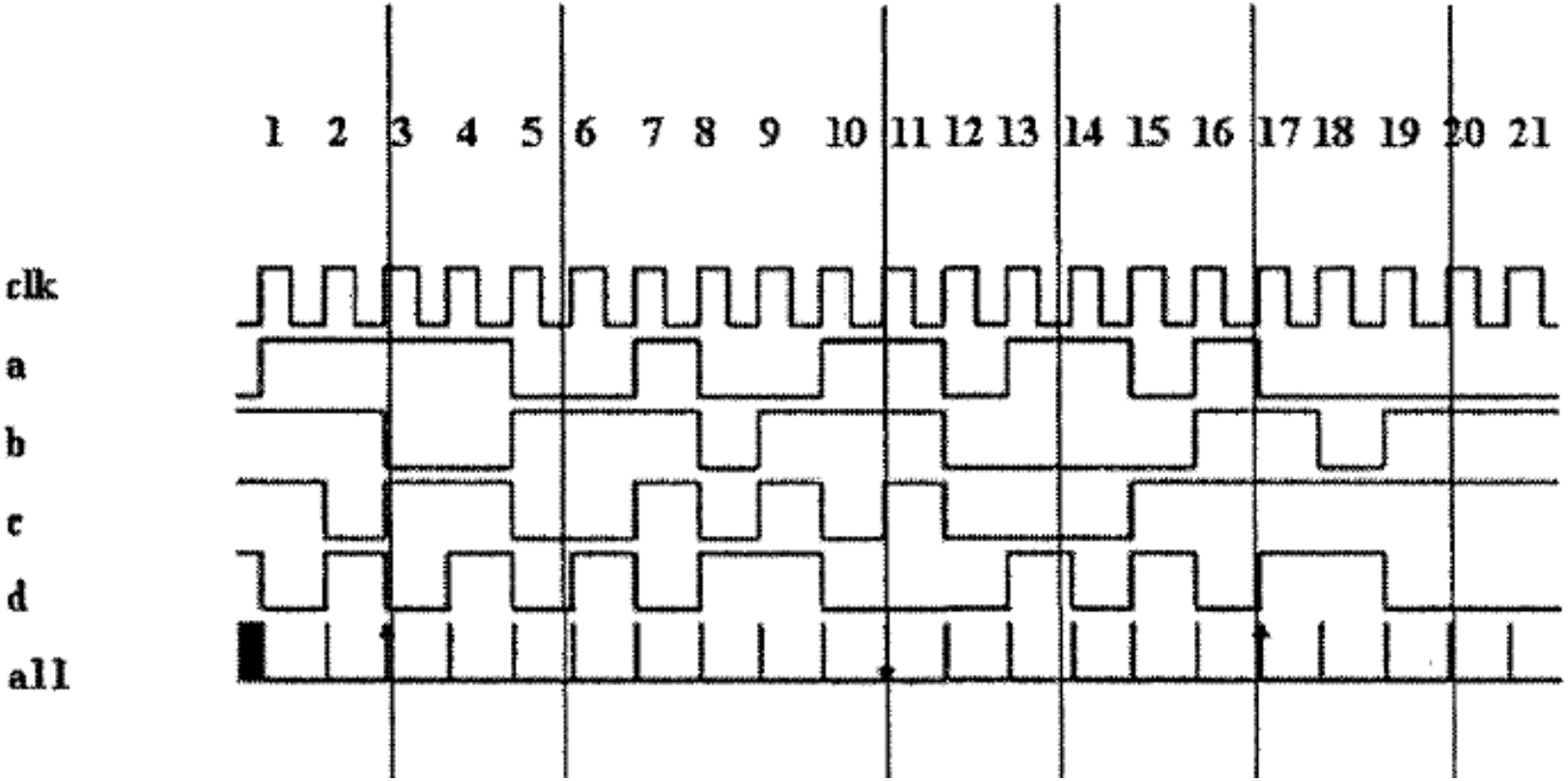
```
property p10;
  @(posedge clk)  a |-> ##2 b;
endproperty

a10 : assert property(p10);
```



Implication with a sequence as an antecedent

```
sequence s11a;
  @(posedge clk)   (a && b) ##1 c;
endsequence

sequence s11b;
  @(posedge clk)   ##2   !d;
endsequence

property p11;
  s11a |-> s11b;
endpeoperty

a11 : assert property(p11);
```

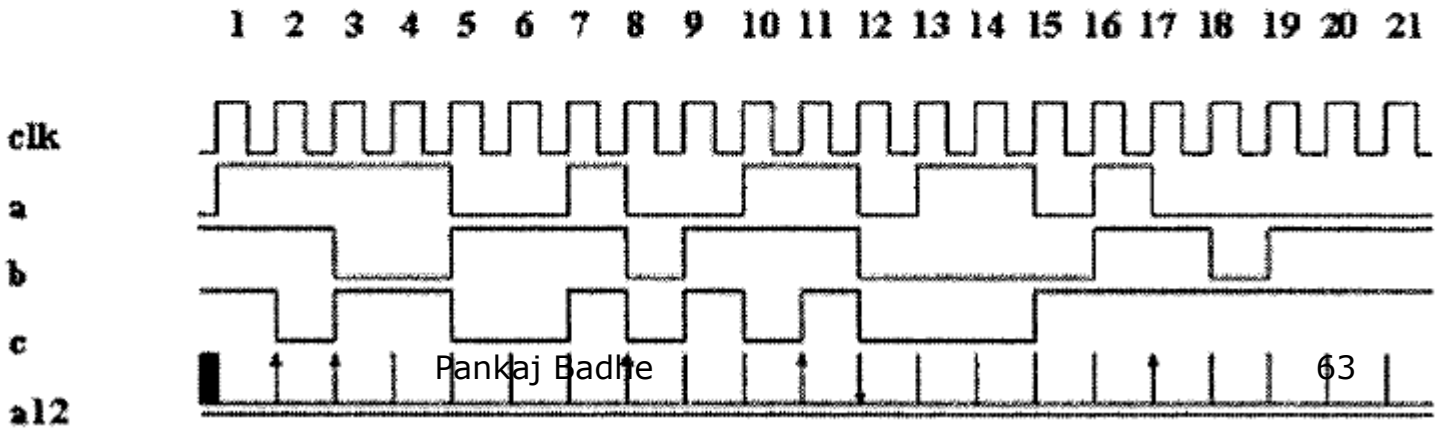
the vacuous successes are shown with a simple straight line



Timing windows in SVA Checkers

```
property p12;
  @(posedge clk)   (a && b) |->  ##[1:3] c;
endproperty

a12 : assert property(p12);
```

```
(a && b) |->  ##[1] c or
(a && b) |->  ##[2] c or
(a && b) |->  ##[3] c
```



The "ended" construct

```
sequence s15a;
 @(posedge clk) a ##1 b;
endsequence

sequence s15b;
 @(posedge clk) c ##1 d;
endsequence

property p15a;
 s15a |=> s15b;
endproperty

property p15b;
 s15a.ended |-> ##2 s15b.ended;
endproperty

a15a: assert property(p15a);
a15b: assert property(p15b);
```

The "$past" construct

$past (signal_name, number of clock cycles)

Spast (signal_name, number of clock cycles, gating signal)

```
Property p19;
 @(posedge clk) (c && d) |->
       ($past((a&&b), 2) == 1'b1);
endproperty

a19: assert property(p19);
```

```
Property p20;
 @(posedge clk) (c && d) |->
            ($past((a&&b), 2, e) == 1'b1);
endproperty

a20: assert property(p20);
```

The "and" construct

- The binary operator **"and"** can be used to combine two sequences logically. The final property succeeds when both the sequences succeed.
- ***Both sequences must have the same starting point but they can have different ending points.*** The starting point of the check is when the first sequence succeeds and the end point is when the other sequence succeeds, ultimately making the property succeed.

```
sequence s27a;
 @(posedge clk) a##[1:2] b;
endsequence

sequence s27b;
 @(posedge clk) c##[2:3] d;
endsequence

property p27;
 @(posedge clk)  s27a and s27b;
endproperty

a27: assert property(p27);
```

Built-in system functions

$onehot(expression) – checks that the expression is one-hot, in other words, only one bit of the expression can be high on any given clock edge.

$onehot0(expression) – checks that the expression is zero one-hot, in other words, only one bit of the expression can be high or none of the bits can be high on any given clock edge.

$isunknown(expression) – checks if any bit of the expression is X or Z.

$countones(expression) – counts the number of bits that are high in a vector.

Multiple clock definitions in SVA

```
sequence s_multiple_clocks;
     @(posedge clk1) a ##1 @(posedge clk2) b;
```

The sequence s_multiple_clocks checks that, on a given positive edge of clock "clkl," signal "a" is high and then on a give positive edge of clock "clk2," signal "b" is high. The sequence matches when signal "a" is high on any given positive edge of clock "clkl." The ##1 delay construct will move the evaluation time to the nearest positive clock edge of clock "clk2" and then will check for signal "b" being high. ***When multiple clocked signals are used in a sequence, only ##1 delay construct is allowed.***

```
always@(posedge clk)
begin
if(arb_sva)
begin

a_arb_onehot0:
      assert property(p_arb_onehot0)
      else if(arb_sva_severity) $fatal;

a_req_gnt:
      assert property(p_req_gnt)
      else if(arb_sva_severity) $fatal;

a_gnt_frame :
      assert property(p_gnt_frame)
      else if(arb_sva_severity) $fatal;

a_frame_gnt:
      assert property(p_frame_gnt)
      else if(arb_sva_severity) $fatal;

c_arb_onehot0: cover property(p_arb_onehot0);
c_req_gnt: cover property(p_req_gnt);
c_gnt_frame: cover property(p_gnt_frame);
c_frame_gnt: cover property(p_frame_gnt);

end
end
```

APB3 Protocol Monitor

```
//
// Interface declaration for the APB Protocol Monitor
//
interface apb_monitor #(int no_slaves = 16, int addr_width = 32, int data_width = 32)
  (
  input PCLK,
  input PRESETn,
  input[addr_width-1:0] PADDR,
  input[data_width-1:0] PWDATA,
  input[data_width-1:0] PRDATA,
  input[no_slaves-1:0] PSEL,
  input PWRITE,
  input PENABLE,
  input PREADY,
  input PSLVERR);
```

```
// Checks for unknown signal values:

// Reuseable property to check that a signal is in a known state
property SIGNAL_VALID(signal);
  @(posedge PCLK)
  !$isunknown(signal);
endproperty: SIGNAL_VALID

RESET_VALID: assert property(SIGNAL_VALID(PRESETn));
PSEL_VALID: assert property(SIGNAL_VALID(PSEL));

// Reuseable property to check that if a PSEL is active, then
// the signal is in a known state
```

```
property CONTROL_SIGNAL_VALID(signal);
  @(posedge PCLK)
  $onehot(PSEL) |-> !$isunknown(signal);
endproperty: CONTROL_SIGNAL_VALID

PADDR_VALID: assert property(CONTROL_SIGNAL_VALID(PADDR));
PWRITE_VALID: assert property(CONTROL_SIGNAL_VALID(PWRITE));
PENABLE_VALID: assert property(CONTROL_SIGNAL_VALID(PENABLE));

// Check that write data is in a known state if a write
property PWDATA_SIGNAL_VALID;
  @(posedge PCLK)
  ($onehot(PSEL) && PWRITE) |-> !$isunknown(PWDATA);
endproperty: PWDATA_SIGNAL_VALID

PWDATA_VALID: assert property(PWDATA_SIGNAL_VALID);

// Check that if PENABLE is active, then the signal is in a known state
property PENABLE_SIGNAL_VALID(signal);
  @(posedge PCLK)
  $rose(PENABLE) |-> !$isunknown(signal)[*1:$] ##1 $fell(PENABLE);
endproperty: PENABLE_SIGNAL_VALID

PREADY_VALID: assert property(PENABLE_SIGNAL_VALID(PREADY));
PSLVERR_VALID: assert property(PENABLE_SIGNAL_VALID(PSLVERR));

// Check that read data is in a known state if a read
property PRDATA_SIGNAL_VALID;
  @(posedge PCLK)
  ($rose(PENABLE && !PWRITE && PREADY)) |-> !$isunknown(PRDATA)[*1:$] ##1 $fell(PENABLE);
endproperty: PRDATA_SIGNAL_VALID

PRDATA_VALID: assert property(PRDATA_SIGNAL_VALID);
```

Timing Relationships

```
// PENABLE goes low once PREADY is sampled
property PENABLE_DEASSERTED;
  @(posedge PCLK)
  $rose(PENABLE && PREADY) |=> !PENABLE;
endproperty: PENABLE_DEASSERTED
```

```
PENABLE_DEASSERT: assert property(PENABLE_DEASSERTED);
COV_PENABLE_DEASSERT: cover property(PENABLE_DEASSERTED);

// From PSEL active to PENABLE active is 1 cycle
property PSEL_TO_PENABLE_ACTIVE;
  @(posedge PCLK)
  (!$stable(PSEL) && $onehot(PSEL)) |=> PENABLE;
endproperty: PSEL_TO_PENABLE_ACTIVE

PSEL_2_PENABLE: assert property(PSEL_TO_PENABLE_ACTIVE);
COV_PSEL_2_PENABLE: cover property(PSEL_TO_PENABLE_ACTIVE);
```

Other Properties

```
// Check that only one PSEL line is valid at a time:
property PSEL_ONEHOT0;
  @(posedge PCLK)
  $onehot0(PSEL);
endproperty: PSEL_ONEHOT0


PSEL_ONLY_ONE: assert property(PSEL_ONEHOT0);
```

Thank you!!

OOPs

Pankaj Badhe

Encapsulation

- Class
  - Data Members
  - Methods
  - Constructor
  - Destructor

Inheritance

- Parent
- Child
- Grandchild
- Friend : See next slide

Data Hiding

- Public
- Private
- Protected

What is Object

- One object
- Multiple Objects
- Copy
- Clone
- Modify

Abstraction

- Higher functionality
- Packet (request/grant, ack,NAK)
- Transaction (Control/Setup)
- Transfer (Move bulk data/application)

Polymorphism

- Overloading
  - Operator
  - Function
- Over-riding

Object Oriented Programming

- WHY OOPs
  - Difference between signal oriented vs data oriented

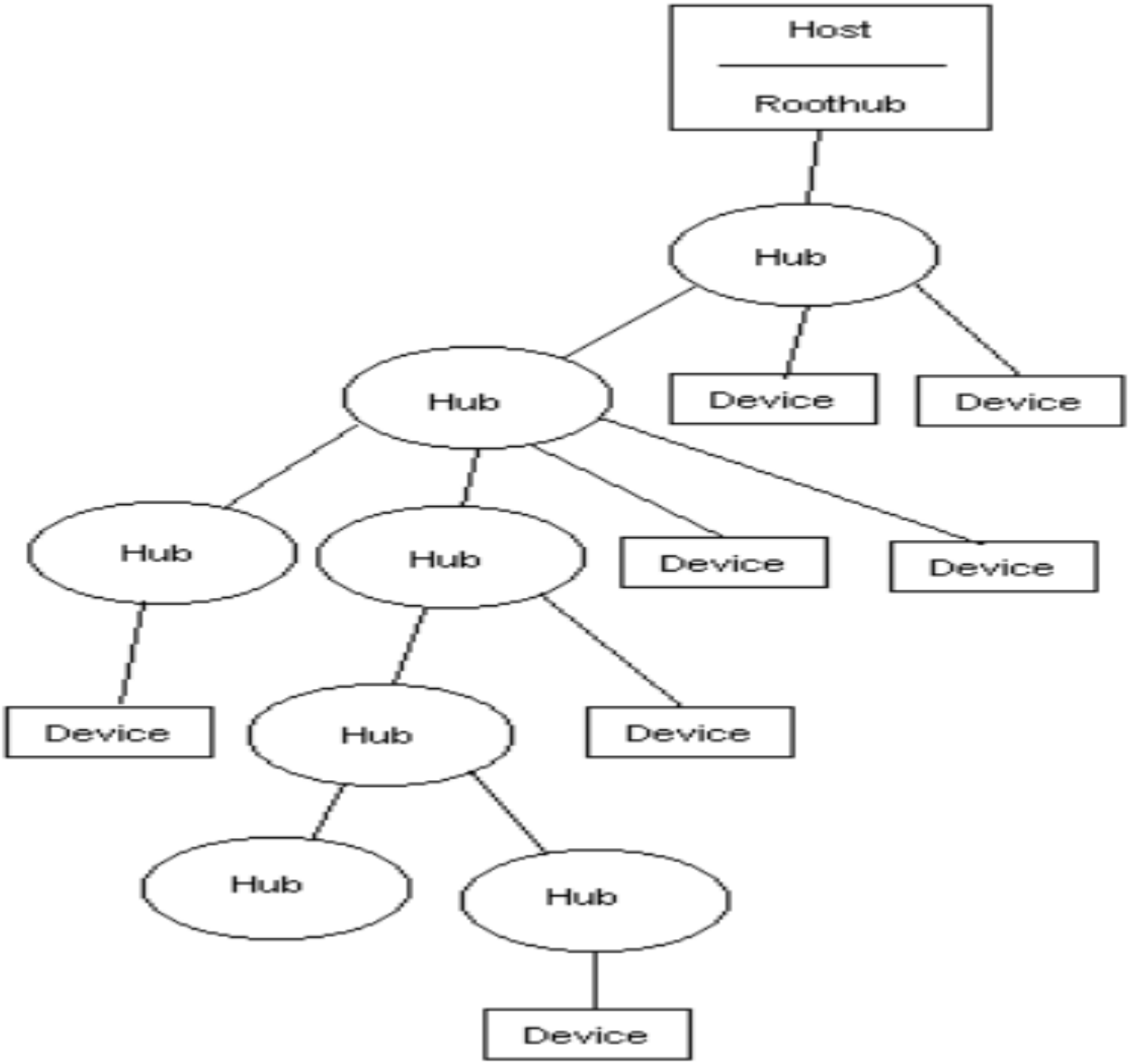
- Languages: SV, PERL, PYTHON, C++ and many more

USB

Pankaj Badhe

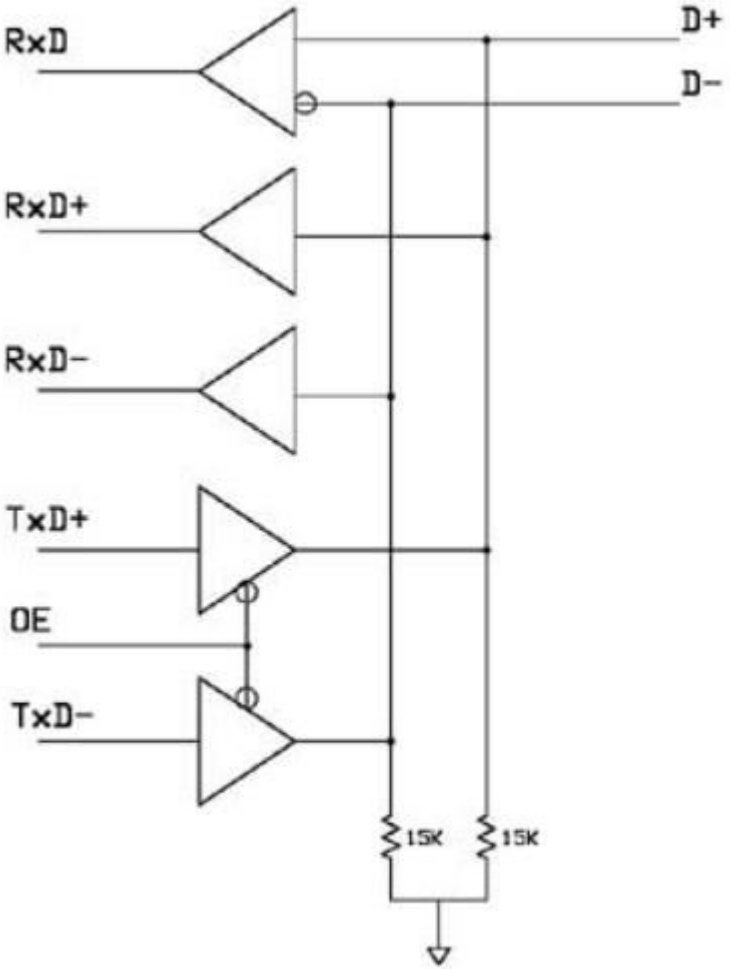
USB is a Bus

- Picture a setup of plugged-in hubs and devices such as that on the right. What we need to remember is that, at any point in time, only the host OR one device can be transmitting at a time.
- When the host is transmitting a packet of data, it is sent to every device connected to an enabled port. It travels downwards via each hub in the chain which resynchronises the data transitions as it relays it. Only one device, the addressed one, actually accepts the data. (The others all receive it but the address is wrong for them.)
- One device at a time is able to transmit to the host, in response to a direct request from the host. Each hub repeats any data it receives from a lower device in an upward only direction.
- Downstream direction ports are only enabled once the device connected to them is addressed, except that one other port at a time can reset a device to address 0 and then set its address to a unique value.


Host
Roothub
Hub
Hub
Device
Device
Hub
Hub
Device
Device
Device
Hub
Device
Hub
Hub
Device

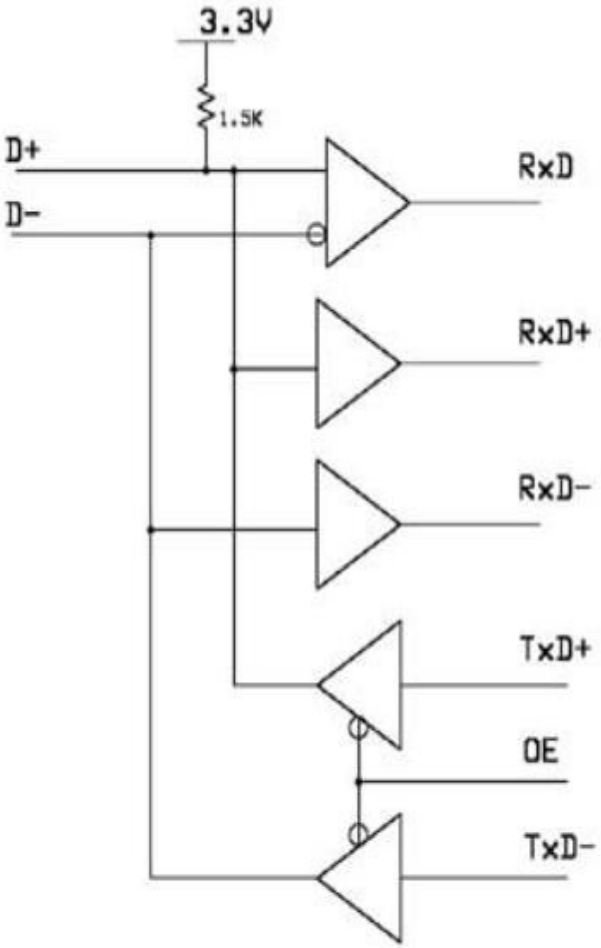
Transceivers

- At each end of the data link between host and device is a transceiver circuit. The transceivers are similar, differing mainly in the associated resistors.
- A typical upstream end transceiver is shown to the right with high speed components omitted for clarity. By upstream, we mean the end nearer to the host. The upstream end has two 15K pull-down resistors.
- Each line can be driven low individually, or a differential data signal can be applied. The maximum 'high' level is 3.3V.



Upstream End Transceiver

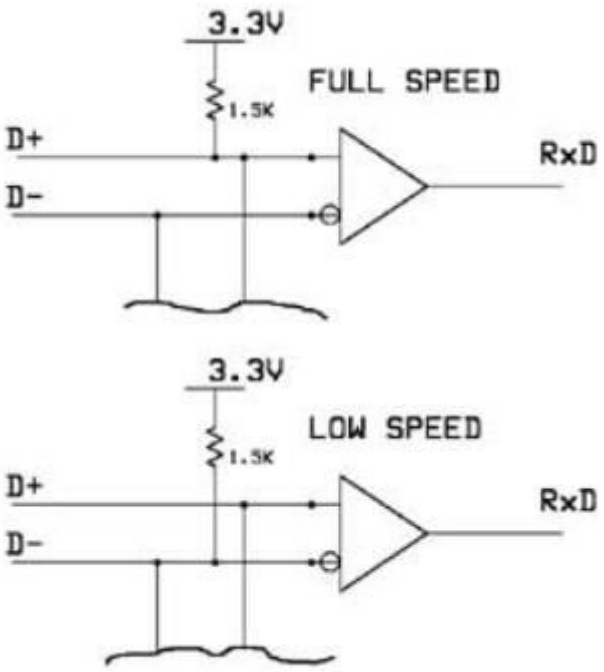
- The equivalent downstream end transceiver, as found in a device, is shown to the right.
- When receiving, individual receivers on each line are able to detect single ended signals, so that the so-called Single Ended Zero (SE0) condition, where both lines are low, can be detected. There is also a differential receiver for reliable reception of data.



Downstream End Transceiver (Full Speed)

Speed Identification

- At the device end of the link a 1.5 kohm resistor pulls one of the lines up to a 3.3V supply derived from VBUS.
- This is on D- for a low speed device, and on D+ for a full speed device.
- (A high speed device will initially present itself as a full speed device with the pull-up resistor on D+.)
- The host can determine the required speed by observing which line is pulled high.



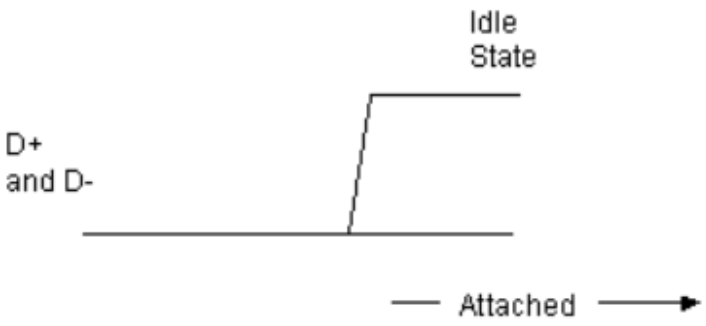
Line States

Given that there are just 2 data lines to use, it is surprising just how many different conditions are signaled using them:

- **Detached**

When no device is plugged in, the host will see both data lines low, as its 15 kohm resistors are pulling each data line low.

- **Attached**
- When the device is plugged in to the host, the host will see either D+ or D- go to a '1' level, and will know that a device has been plugged in.
- The '1' level will be on D- for a low speed device, and D+ for a full (or high) speed device.
- **Idle**
- The state of the data lines when the pulled up line is high, and the other line is low, is called the idle state. This is the state of the lines before and after a packet is sent.



J, K and SEO States

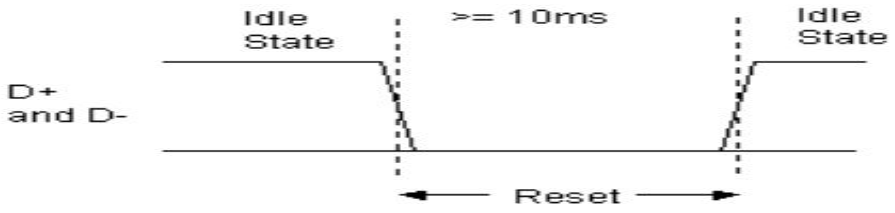
- To make it easier to talk about the states of the data lines, some special terminology is used. The 'J State' is the same polarity as the idle state (the line with the pull-up resistor is high, and the other line is low), but is being driven to that state by either host or device.
- The K state is just the opposite polarity to the J state.
- The Single Ended Zero (SE0) is when both lines are being pulled low.
- *The J and K terms are used because for Full Speed and Low Speed links they are actually of opposite polarity*

Bus State	Levels
Differential '1'	D+ high, D- low
Differential '0'	D- high, D+ low
Single Ended Zero (SE0)	D+ and D- low
Single Ended One (SE1)	D+ and D- high
Data J State: Low-speed Full-speed	 Differential '0' Differential '1'
Data K State: Low-speed Full-speed	 Differential '1' Differential '0'
Idle State: Low-speed Full-speed	 D- high, D+- low D+ high, D- low
Resume State	Data K state
Start of Packet (SOP)	Data lines switch from idle to K state
End of Packet (EOP)	SE0 for 2 bit times followed by J state for 1 bit time
Disconnect	SE0 for >= 2us
Connect	Idle for 2.5us
Reset	SE0 for >= 2.5 us

- **Single Ended One (SE1)**
- This is the **illegal** condition where both lines are high. It should never occur on a properly functioning link.

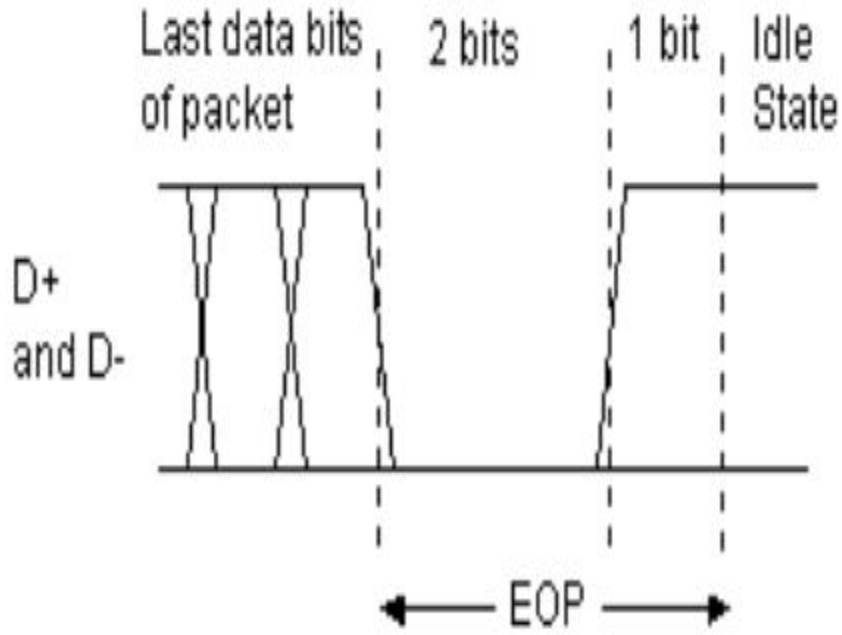
Reset

- When the host wants to start communicating with a device it will start by applying a 'Reset' condition which sets the device to its default unconfigured state.
- The Reset condition involves the host pulling down both data lines to low levels (SE0) for at least 10 ms. The device may recognise the reset condition after 2.5 us.
- This 'Reset' should not be confused with a micro-controller power-on type reset. It is a USB protocol reset to ensure that the device USB signaling starts from a known state.



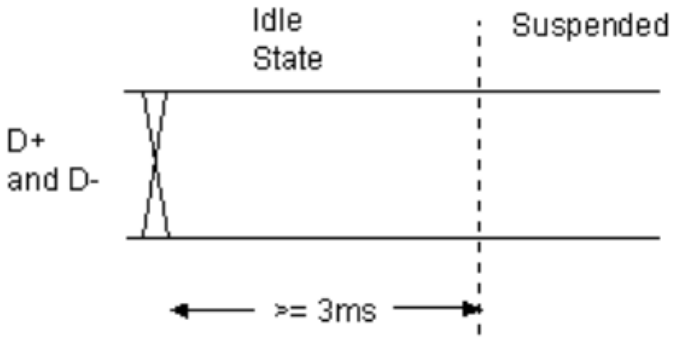
EOP signal

- The End of Packet (EOP) is an SE0 state for 2 bit times, followed by a J state for 1 bit time.



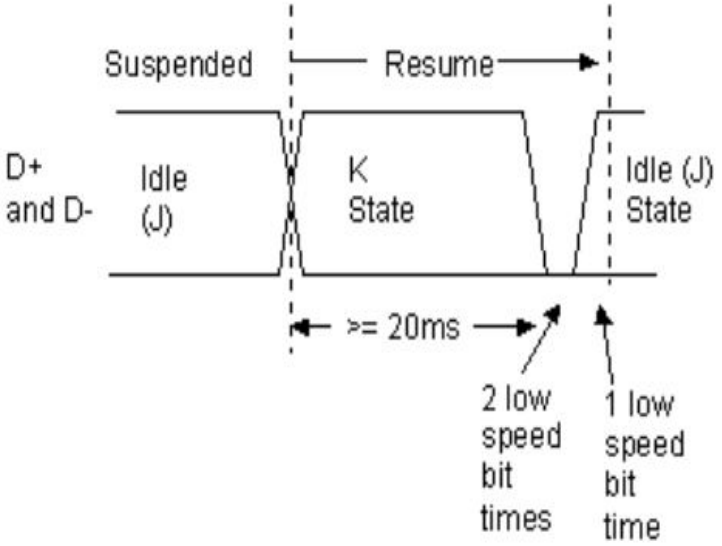
Suspend

- One of the features of USB which is an essential part of today's emphasis of 'green' products is its ability to power down an unused device. It does this by suspending the device, which is achieved by not sending anything to the device for 3 ms.
- Normally a SOF packet (at full speed) or a Keep Alive signal (at low speed) is sent by the host every 1 ms, and this is what keeps the device awake.
- A suspended device may draw no more than 0.5 mA from Vbus.
- A suspended device must recognise the resume signal, and also the reset signal.



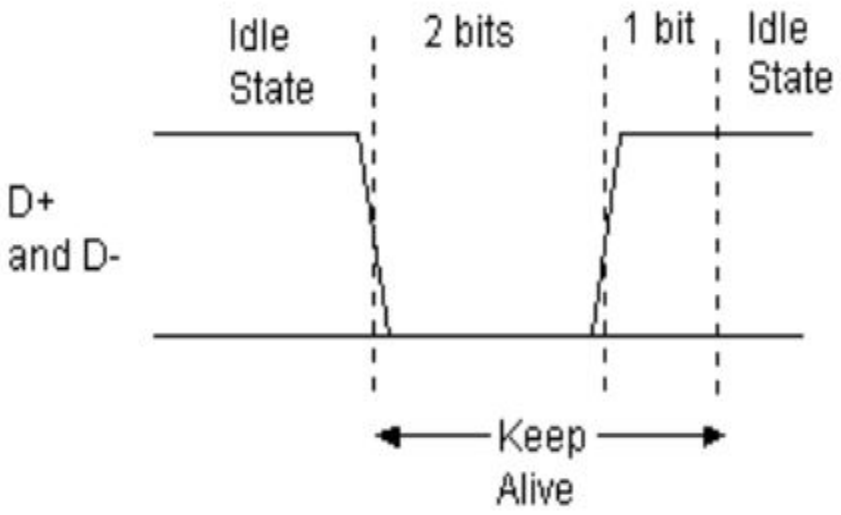
Resume

- When the host wants to wake the device up after a suspend, it does so by reversing the polarity of the signal on the data lines for at least 20ms. The signal is completed with a low speed end of packet signal.
- It is also possible for a device with its remote wakeup feature set, to initiate a resume itself. It must have been in the idle state for at least 5ms, and must apply the wakeup K condition for between 1 and 15 ms. The host takes over the driving of the resume signal within 1 ms.



Keep Alive Signal

- This is represented by a Low speed EOP. It is sent at least once every millisecond on a low speed link, in order to keep the device from suspending.



Packets

- The packet could be thought of as the smallest element of data transmission. Each packet conveys an integral number of bytes at the current transmission rate. Before and after the packet, the bus is in the idle state.
- A packet starts with a sync pattern to allow the receiver bit clock to synchronise with the data. It is followed, by the data bytes of the packet, and concluded with an End of Packet (EOP) signal. The data is actually NRZI encoded, and in order to ensure sufficiently frequent transitions, a zero is inserted after 6 successive 1's (this is known as bit stuffing).

A Single Packet

Endpoints

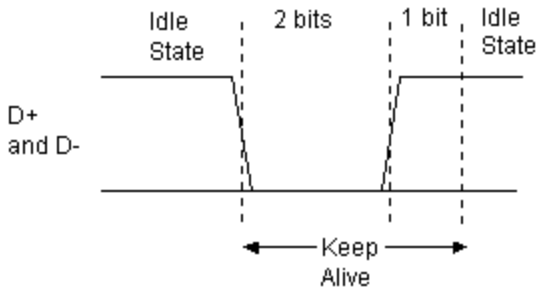
Each USB device has a number of endpoints. Each endpoint is a source or sink of data. A device can have up to 16 OUT and 16 IN endpoints.
OUT always means from host to device.
IN always means from device to host.
Endpoint 0 is a special case which is a combination of endpoint 0 OUT and endpoint 0 IN, and is used for controlling the device.

Pipe

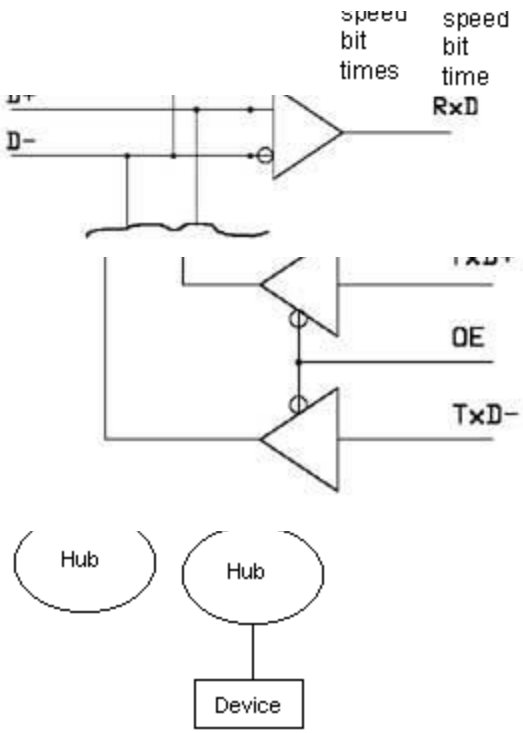
A logical data connection between the host and a particular endpoint, in which we ignore the lower level mechanisms for actually achieving the data transfers.

Transactions

Simple transfers of data called 'Transactions' are built up using packets.



rmats



ery packet is a Packet Identifier (PID) byte. This byte
ised quickly by the SIE and so is not included in any
refore has its own validity check. The PID itself is 4
bits are repeated in an complemented form.

msb

PID2	PID3	\PID0	\PID1	\PID2	\PID3

n here in the order of transmission; lsb first.

Code (CRC)

ulated from a number of data bytes to form a unique value which is transmitted along with the data bytes, and then used to validate the correct reception of the data.

USB uses two different CRCs, one 5 bits long (CRC5) and one 16 bits long (CRC16).

PID Type	PID Name	PID<3:0>*
Token	OUT	0001b
	IN	1001b
	SOF	0101b
	SETUP	1101b
Data	DATA0	0011b
	DATA1	1011b
	DATA2	0111b
	MDATA	1111b
Handshake	ACK	0010b
	NAK	1010b
	STALL	1110b
	NYET	0110b
Special	PRE	1100b
	ERR	1100b
	SPLIT	1000b
	PING	0100b
	Reserved	0000b

* Bits are transmitted lsb first

Token Packet

Sync	PID	ADDR	ENDP	CRC5	EOP
	8 bits	7 bits	4 bits	5 bits	

Used for SETUP, OUT and IN packets. They are always the first packet in a transaction, identifying the targeted endpoint, and the purpose of the transaction.

The SOF packet is also defined as a Token packet, but has a slightly different format and purpose, which is described below.

The token packet contains two addressing elements:

Address (7 bits)

This device address can address up to 127 devices. Address 0 is reserved for a device which has not yet had its address set.

Endpoint number (4 bits)

There can be up to 16 possible endpoints in a device in each direction. The direction is implicit in the PID. OUT and SETUP PIDs will refer to the OUT endpoint, and an IN PID will refer to the IN endpoint.

Data Packet

Sync	PID	DATA	CRC16	EOP
	8 bits	(0-1024) x 8 bits	16 bits	

Used for DATA0, DATA1, DATA2 and MDATA packets. If a transaction has a data stage this is the packet format used.

DATA0 and DATA1 PIDs are used in Low and Full speed links as part of an error-checking system. When used, all data packets on a particular endpoint use an alternating DATA0 / DATA1 so that the endpoint knows if a received packet is the one it is expecting. If it is not it will still acknowledge (ACK) the packet as it is correctly received, but will then discard the data, assuming that it has been re-sent because the host missed seeing the ACK the first time it sent the data packet.

DATA2 and MDATA are only used for high speed links.

Handshake Packet

Sync	PID	EOP
	8 bits	

Used for ACK, NAK, STALL and NYET packets. This is the packet format used in the status stage of a transaction, when required.

- ACK : Receiver acknowledges receiving error free packet.
- NAK : Receiving device cannot accept data or transmitting device cannot send data.
- STALL : Endpoint is halted, or control pipe request is not supported.
- NYET : No response yet from receiver (high speed only)

SOF Packet

Sync	PID	Frame No.	CRC5	EOP
	8 bits	11 bits	5 bits	

The Start of Frame packet is sent every 1 ms on full speed links. The frame is used as a time frame in which to schedule the data transfers which are required. For example, an isochronous endpoint will be assigned one transfer per frame.

Frames

On a low speed link, to preserve bandwidth, a Keep Alive signal is sent every millisecond, instead of a Start of Frame packet. In fact Keep Alives may be sent by a hub on a low speed link whenever the hub sees a full speed token packet.

At high speed the 1 ms frame is divided into 8 microframes of 125 us. A SOF is sent at the start of each of these 8 microframes, each having the same frame number, which then increments every 1 ms frame.

Transactions

A successful transaction is a sequence of three packets which performs a simple but secure transfer of data.

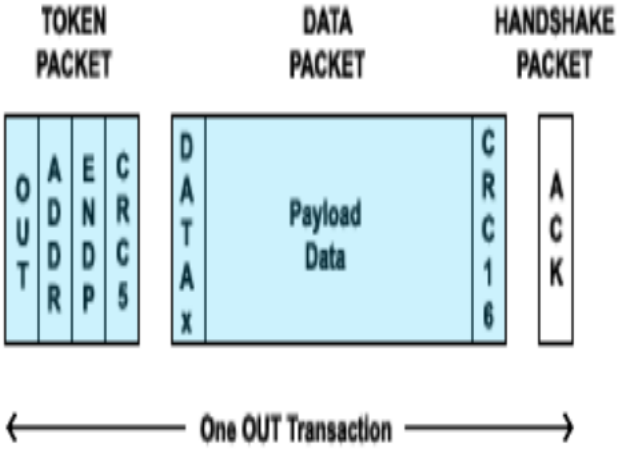
For IN and OUT transactions used for isochronous transfers, there are only 2 packets; the handshake packet on the end is omitted. This is because error-checking is not required.

There are three types of transaction. In each of the illustrations below, the packets from the host are shaded, and the packets from the device are not.

OUT Transaction

A successful OUT transaction comprises two or three sequential packets. If it were being used in an *Isochronous Transfer* there would not be a handshake packet from the device.

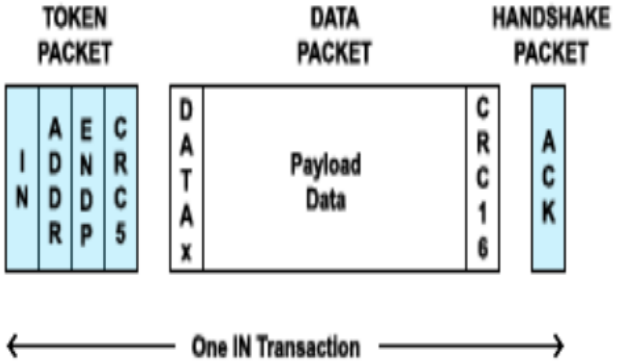
On a low or full speed link, the PID shown as DATAx will be either a DATA0 or a DATA1. An alternating DATA0/DATA1 is used as a part of the error control protocol to (or from) a particular endpoint.



IN Transaction

A successful IN transaction comprises two or three sequential packets. If it were being used in an *Isochronous Transfer* there would not be a handshake packet from the host.

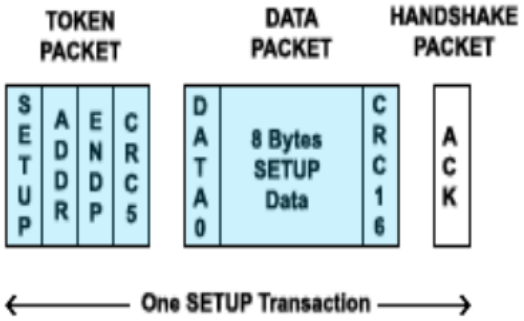
Here again, the DATAx is either a DATA0 or a DATA1.



SETUP Transaction

A successful SETUP transaction comprises three sequential packets. This is similar to an OUT transaction, but the data payload is exactly 8 bytes long, and the SETUP PID in the token packet informs the device that this is the first transaction in a *Control Transfer* (see below).

As will be seen below, the SETUP transaction always uses a DATA0 to start the data packet.



Data Flow Types

There are four different ways to transfer data on a USB bus. Each has its own purposes and characteristics. Each one is built up using one or more transaction type.

Data Flow Type	Description
Control Transfer	Mandatory using Endpoint 0 OUT and Endpoint 0 IN.
Bulk Transfer	Error-free high volume throughput when bandwidth available.
Interrupt Transfer	Regular Opportunity for status updates, etc. Error-free Low throughput
Isochronous Transfer	Guaranteed fixed bandwidth. Not error-checked.

Bulk Transfers

Bulk transfers are designed to transfer large amounts of data with error-free delivery, but with no guarantee of bandwidth. The host will schedule bulk transfers after the other transfer types have been allocated.

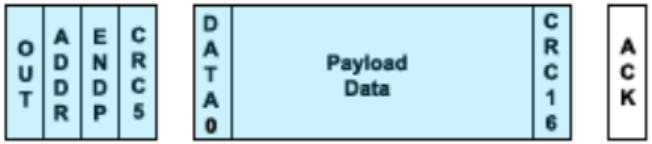
If an OUT endpoint is defined as using Bulk transfers, then the host will transfer data to it using OUT transactions.

If an IN endpoint is defined as using Bulk transfers, then the host will transfer data from it using IN transactions.

The max packet size is 8, 16, 32 or 64 at full Speed and 512 for high speed. Bulk transfers are not allowed at low speed.

Use Bulk transfers when you have a lot of data to shift, as fast as possible, but where you would not have a large problem if there is a delay caused by insufficient bandwidth.

Example Bulk Transfer



The diagrams to the right illustrate the possible flow of events in the face of errors.

Error Control - IN

If the IN token packet is not recognised, the device will not respond at all. Otherwise, if it has data to send it will send it in a DATA0 or DATA1 packet, If it is not ready to send data it will send a NAK packet. If the endpoint is currently 'halted' then it will respond with a STALL packet.

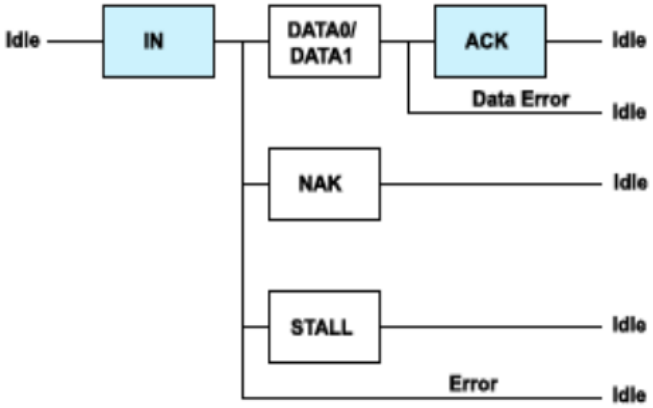
In the case of DATA0/1 being sent, the host will acknowledge with an ACK, unless the data is not validly received, in which case it does not send an ACK. (Note: the host never sends NAK!)

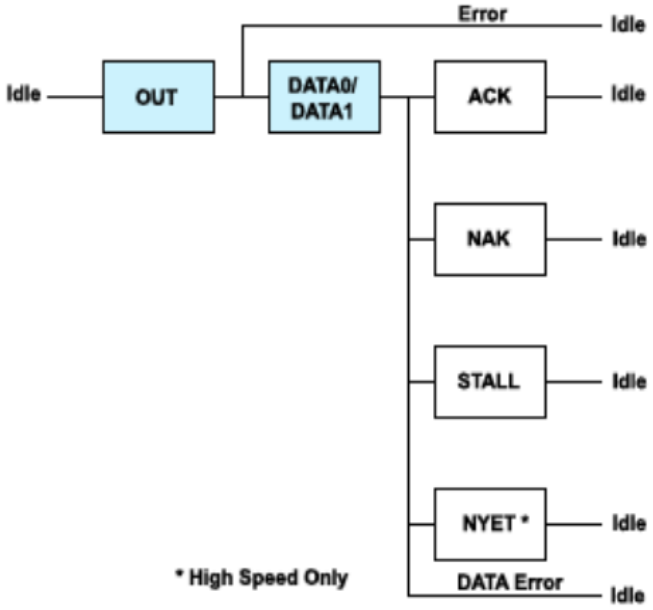
Error Control - OUT

If the OUT token packet is not recognised, the device will not respond at all. It will then ignore the DATAx packet because it does not know that it has been addressed.

If the OUT token is recognised but the DATAx packet is not recognised, then the device will not respond.

If the data is received but the device can't accept it at this time, it will send a NAK, and if the endpoint is currently halted, it will send a STALL.





Interrupt Transfers

Interrupt transfers have nothing to do with interrupts. The name is chosen because they are used for the sort of purpose where an interrupt would have been used in earlier connection types.

Interrupt transfers are regularly scheduled IN or OUT transactions, although the IN direction is the more common usage.

Typically the host will only fetch one packet, at an interval specified in the *endpoint descriptor* (see below). The host guarantees to perform the IN transaction at least that often, but it may actually do it more frequently.

Interrupt packets can have any size from 1 to 8 bytes at low speed, from 1 to 64 at full speed or up to 1024 bytes at high speed.

Use an interrupt transfer when you need to be regularly kept up to date of any changes of status in a device. Examples of their use are for a mouse or a keyboard.

Error control is very similar to that for bulk transfers.

Isochronous Transfers

Isochronous transfers have a guaranteed bandwidth, but error-free delivery is not guaranteed.

The main purpose of isochronous transfers is applications such as audio data transfer, where it is important to maintain the data flow, but not so important if some data gets missed or corrupted.

An isochronous transfer uses either an IN transaction or an OUT transaction depending on the type of endpoint. The special feature of these transactions is that there is no handshake packet at the end.

An isochronous packet may contain up to 1023 bytes at full speed, or up to 1024 at high speed. Isochronous transfers are not allowed at low speed.

Control Transfer

This is a bi-directional transfer which uses both an IN and an OUT endpoint. Each control transfer is made up of from 2 to several transactions.

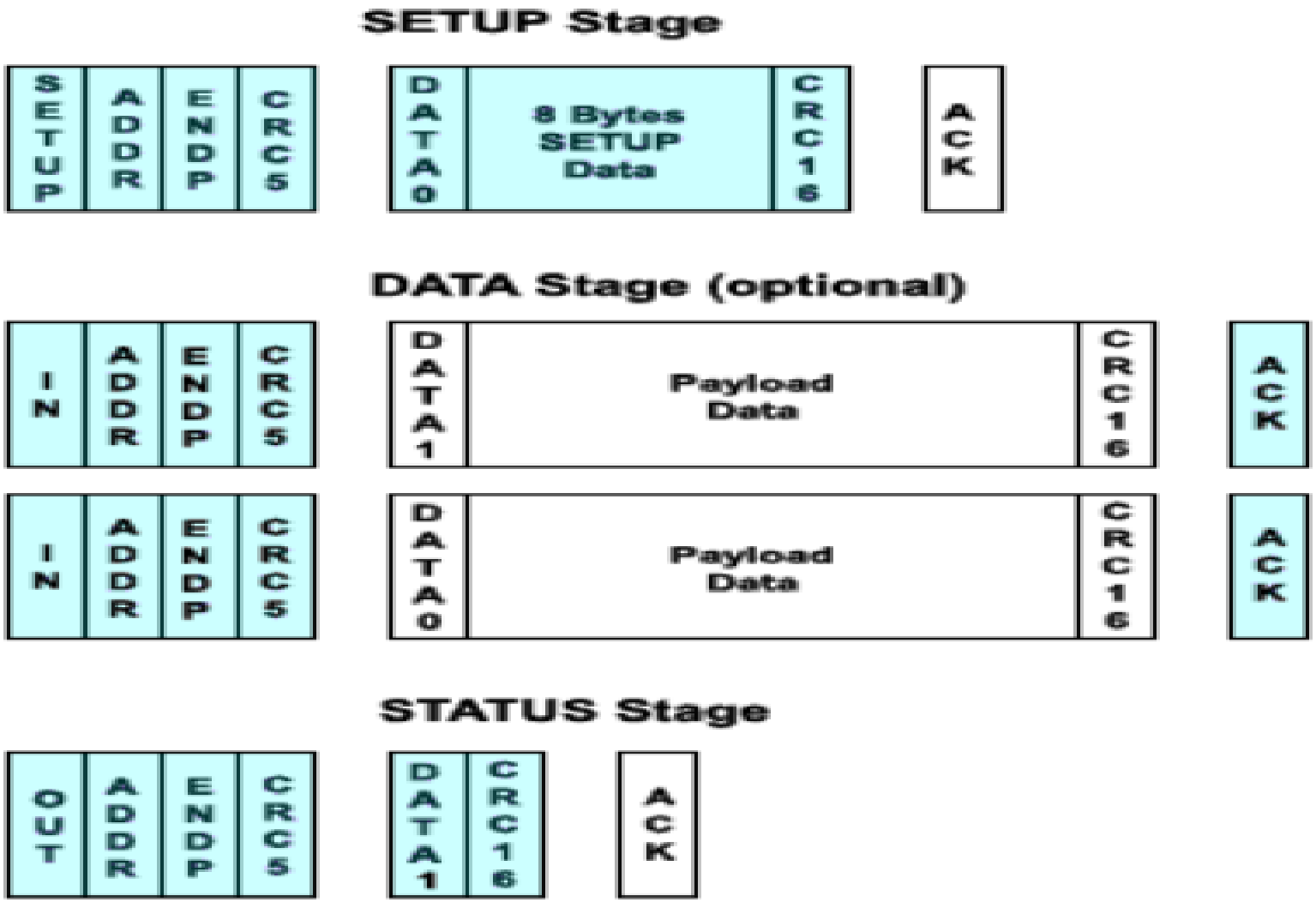
It is divided into three stages.

The SETUP stage carries 8 bytes called the Setup packet. This defines the request, and specifies whether and how much data should be transferred in the DATA stage.

The DATA stage is optional. If present, it always starts with a transaction containing a DATA1. The type of transaction then alternates between DATA0 and DATA1 until all the required data has been transferred.

The STATUS stage is a transaction containing a zero-length DATA1 packet. If the DATA stage was IN then the STATUS stage is OUT, and vice versa.

Control transfers are used for initial configuration of the device by the host, using Endpoint 0 OUT and Endpoint 0 IN, which are reserved for this purpose. They may be used (on the same endpoints) after configuration as part of the device-specific control protocol, if required.

The max packet size for the data stage is 8 bytes at low speed, 8, 16, 32 or 64 at full Speed and 64 for high speed.

Example Control Read