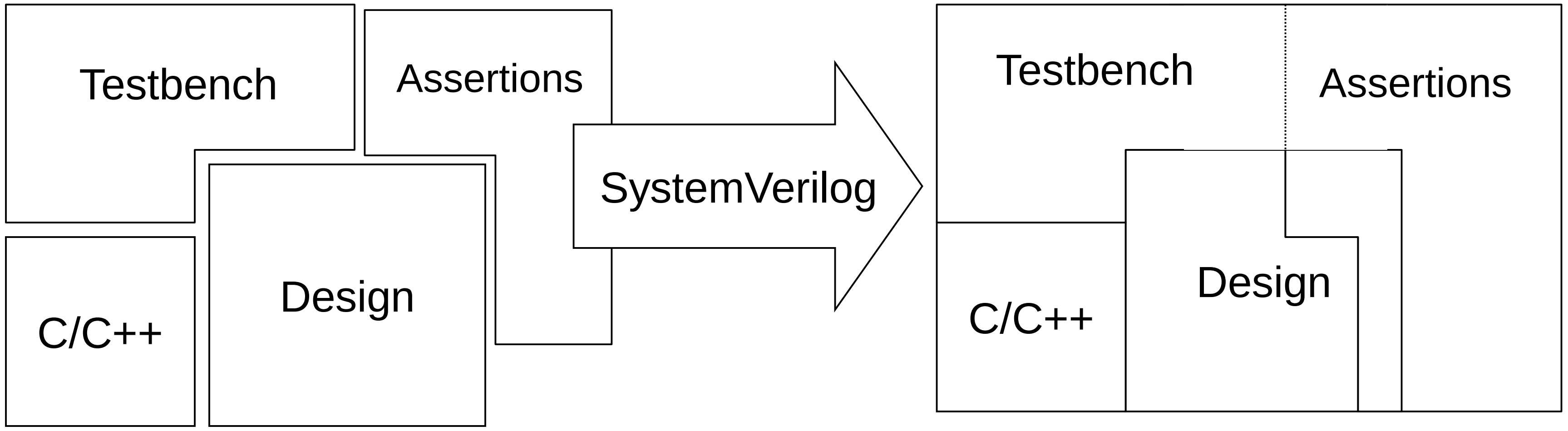
SystemVerilog Basics - Design

- SystemVerilog For Design by Stuart Sutherland
- Digital System Design with SystemVerilog by MarkZ

Contents

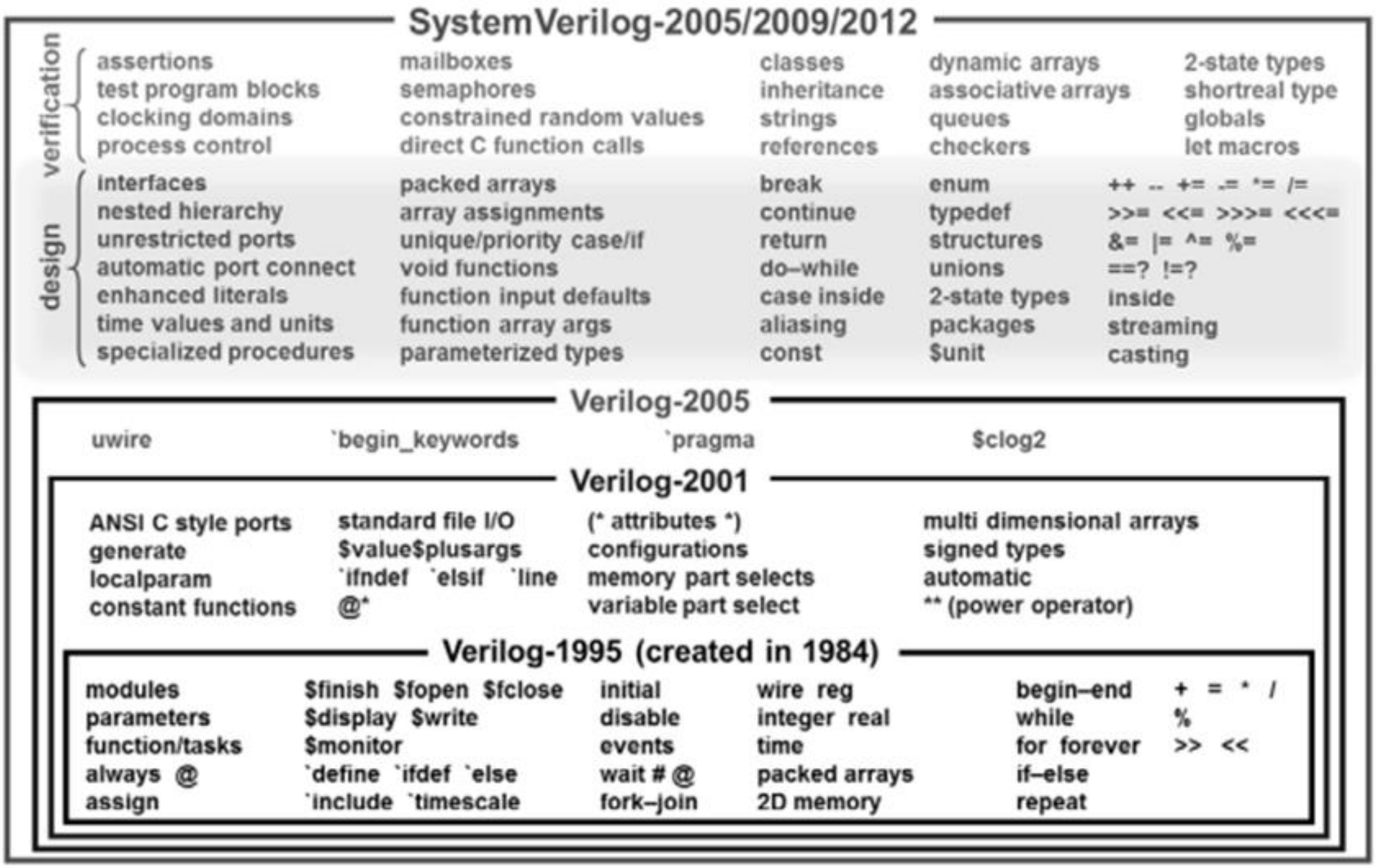
SystemVerilog Landscape



- Multiple Languages
  - HDLs for Design
  - HVLs for Testbench
  - Assertions
  - C/C++
- Slow Interfaces and Inefficient Communication

- Single Language for Design and Verification
  - SystemVerilog HDVL
- Single Verification Platform
- Standard Language Enables Complementary Tools and Methodologies

SystemVerilog Semantic Concepts



SystemVerilog-2005/2009/2012

verification

assertions
test program blocks
clocking domains
process control

mailboxes
semaphores
constrained random values
direct C function calls

classes
inheritance
strings
references

dynamic arrays
associative arrays
queues
checkers

2-state types
shortreal type
globals
let macros

design

interfaces
nested hierarchy
unrestricted ports
automatic port connect
enhanced literals
time values and units
specialized procedures

packed arrays
array assignments
unique/priority case/if
void functions
function input defaults
function array args
parameterized types

break
continue
return
do–while
case inside
aliasing
const

enum
typedef
structures
unions
2-state types
packages
$unit

++ -- += -= *= /=
>>= <<= >>>= <<<=
&= |= ^= %=
==? !=?
inside
streaming
casting

Verilog-2005

uwire
`begin_keywords
`pragma
$clog2

Verilog-2001

ANSI C style ports
generate
localparam
constant functions

standard file I/O
$value$plusargs
`ifndef `elsif `line
@*

(* attributes *)
configurations
memory part selects
variable part select

multi dimensional arrays
signed types
automatic
** (power operator)

Verilog-1995 (created in 1984)

modules
parameters
function/tasks
always @
assign

$finish $fopen $fclose
$display $write
$monitor
`define `ifdef `else
`include `timescale

initial
disable
events
wait # @
fork–join

wire reg
integer real
time
packed arrays
2D memory

begin–end
while
for forever
if–else
repeat

+ = * /
%
>> <<

Basic Data Types

2 and 4 State Datatypes

Verilog

```
reg a;
integer i;
```

Verilog reg and integer type bits can contain x and z values

SystemVerilog

```
logic a;
logic signed [31:0]]
i;
```

Equivalent to these 4-valued SystemVerilog types

SystemVerilog

```
bit a;
int i;
```

These SystemVerilog types have two-valued bits (0 and 1)

If you don't need the x and z values then use the SystemVerilog bit and int types which MAKE EXECUTION MUCH FASTER and use only half the memory

Basic SystemVerilog Datatypes

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

- Make up your own types with typedef
- Define arrays of bits and logic

Arrays of logic and bit default to unsigned

Using Literals

```
reg [31:0] a,b;
reg [15:0] c,d;
...
a = 32'hf0ab;
c = 16'hFFFF
```

This works like in Verilog

```
a = '0;
b = '1;
c = 'x;
d = 'z;
```

This fills the packed array with the same bit value

```
logic [31:0] a;
...
a = 32'hffffffff;
a = '1;
```

These are equivalent

Convenient way to fill up a vector with a bit constant

Variable Types – Global vs. Local

- Global variables
  - Defined outside of any module (i.e. in $root)
  - Shared by all module instances
  - Must be static
  - Tasks and functions can be global too
  - Can be referenced anywhere in the design hierarchy

Keep global definitions in one place (ie. a separate file)

SystemVerilog Timeunit & Precision

- With SystemVerilog, time units and time precision can be specified as keywords

```
module test (…);
   timeunit 1ns;
   timeprecision 10ps;
   …
```

No space is allowed between the number and the unit

You can specify units with your delays

```
#10 a = 1;
#5ns b = 1b;
#1fs $display("%b", b);
```

•legal time units are s, ms, us, ns, ps, fs

SVlog Timeunit & Timeprecision Rules

- Take precedence over `timescale directives
- Remove all ambiguity about a module’s time units
- Can be specified globally, for all modules in a design
- Can be specified within a module
  - Local declarations take precedence over global
  - Must be specified immediately after the module declaration
- Mixing `timescale and timeprecision uses smallest precision in Verilog or SystemVerilog

Type Casting

- SystemVerilog adds casting operations to Verilog

  <type>'(value>) - cast a value to any data type, including user-defined types

```
int'(2.0 * 3.0) //cast operation results to int
```

  <size>'(value>) - cast a value to any vector size

```
17'(n -2) //cast operation results to 17 bits wide
```

```
signed'(x) //cast value to a signed value
```

Casts And Typedefs

```
int i;
real pi = 3.14159;

...

i = int'(pi * 0.5);
```

type cast

complex type casts are possible but need a typedef

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

address_bus_type address_bus;
...
address_bus = address_bus_type'i;
```

cast i as type address_bus_type

Enumerated Data Types

```
typedef enum {bashful, doc, dopey, grumpy, happy, sleepy} dwarf_type;
dwarf_type dwarf;
enum {yes, no=3, maybe} choice;
```

Typedef

User-defined encoding

Can print enums as text or as number

```
dwarf = dopey;
choice = maybe;
$display("dwarf=%s choice=%s", dwarf.name, choice.name );
$display("dwarf=%d choice=%d", dwarf, choice );
```

```
dwarf=dopey choice=maybe
dwarf=2 choice=4
```

```
dwarf=dwarf_type'(2);
dwarf=dwarf_type'(no);
myint=int'(happy);
```

Must use cast

Enumerated Data Types

- Enumerated types are “strongly typed”
  - Can by any data type – default is int (2-state logic)
  - The default initial value is 0.
  - Can be used in expressions (treated like a constant)
  - Can be assigned to a variable
  - Can’t be assigned a value that is not in the type set
  - Can’t be directly assigned the results of an expression
    - The expression result can be cast to the enumerated type, and then assigned

Methods for Enums

```
typedef enum {red, green, blue, yellow} Colors;
```

```
Colors color = color.first;

$display(“Name is %s”, color.name);

$display(“There are %d constants in the enum”, color.num);


color = color.next
$display(“The next color is %s”, color);

color = color.last
$display(“The last color is %d”, color);

$display(“The previous color is %s”, color.prev);
```

Initializes with the color red

Name is red

There are 4 constants in the enum

The next color is green

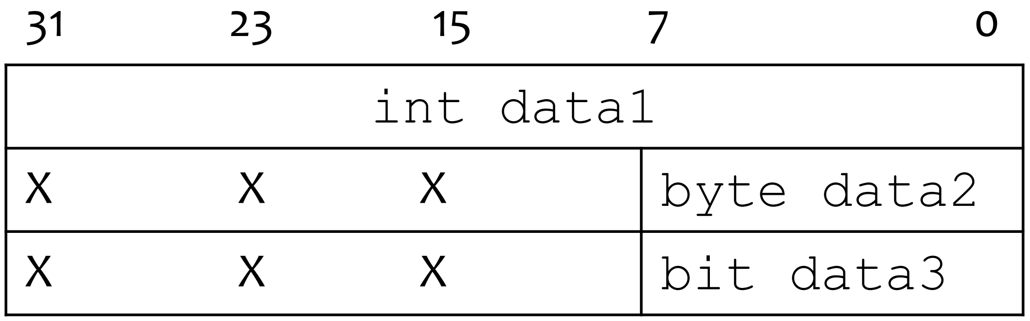
The next color is 3

The previous color is blue

Structures

- In arrays, all the elements stored are the same data type
- A structure is a collection of variables and/or constants under a single name
- Structures can be user defined types
- A method for packing data of different types

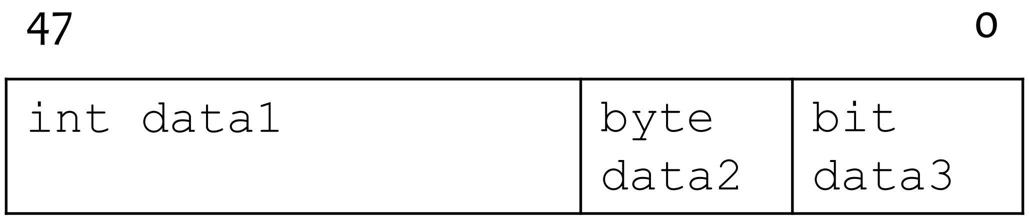
```
typedef struct {
        int data1;
        byte data2;
        bit [7:0] data3;
       } my_data;
```



Structures

- A packed structure is used for subdividing a vector into subfields

```
typedef struct packed{
        int data1;
        byte data2;
        bit [7:0] data3;
       } my_data;
```

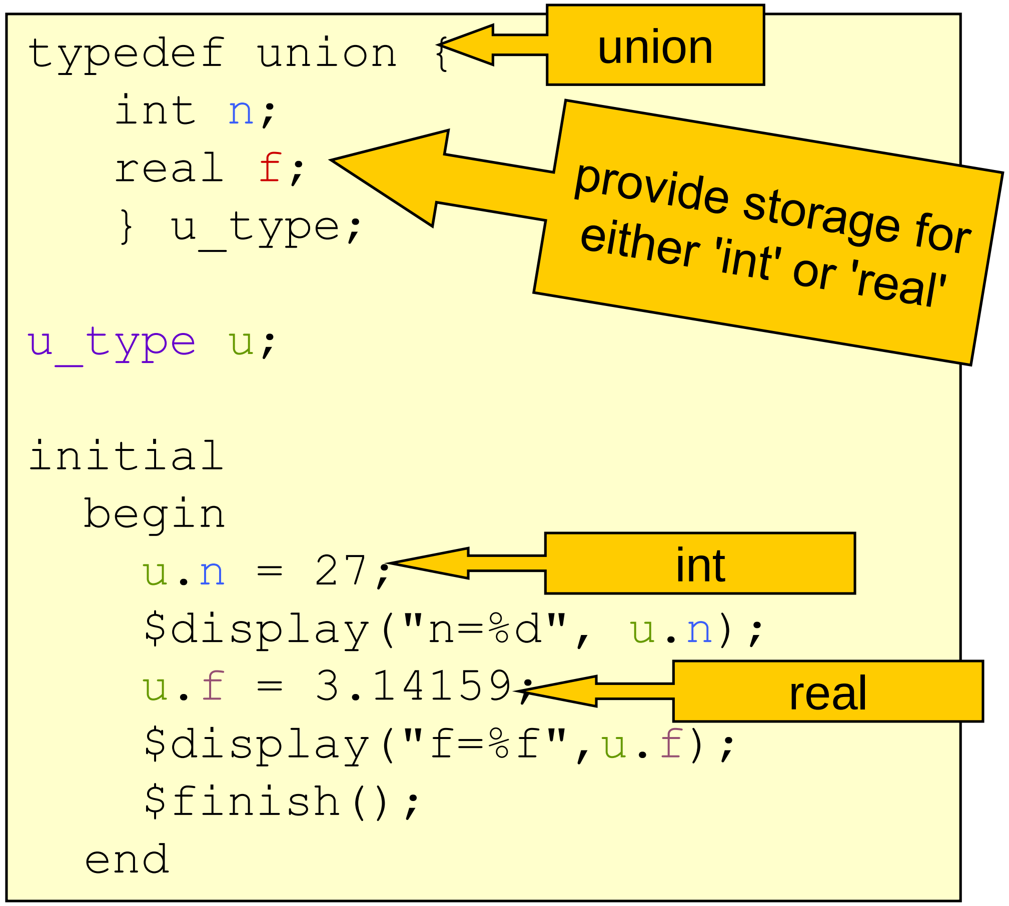


Structure Initialization

```
typedef struct {
            int a, b;
            byte c;
            bit [7:0] d;
            } data;
data data1 = {50, 100, 8’hf0, 8’h0f};
data data2;
always@(posedge clk, negedge rst)
   If(!rst) begin
      data2.a = 0; data2.b = 0;
      data2.c = ‘0; data2.d = ‘1;
  else begin
      data2.a = $random(); data2.b = (a-3);
      data2.c = 8’h37; data2.d = 8’hdd;
end
```

- structures can be initialized during definition like C
- structures can also be assigned in a procedural block

Unions



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

u_type u;

initial
  begin
    u.n = 27;
    $display("n=%d", u.n);
    u.f = 3.14159;
    $display("f=%f",u.f);
    $finish();
  end
```

- Unions allow storage of different data types in the same space
- Unions can be used to define storage before the type of the value to be stored is known

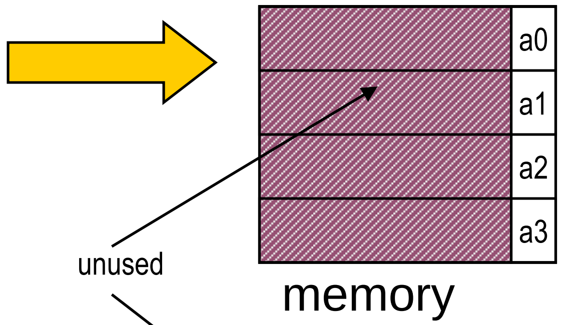
Packed and Unpacked Arrays

- Unpacked
  - Can be any datatype
  - You can access an entire array at a time
  - You can copy an entire array
  - Uses a range: int Mem [1023:0]
- Packed
  - Must be: `reg, wire, logic, bit, other nets`
  - Access whole array or slice as a vector
  - Can have multiple dimensions

Packed And Unpacked Arrays

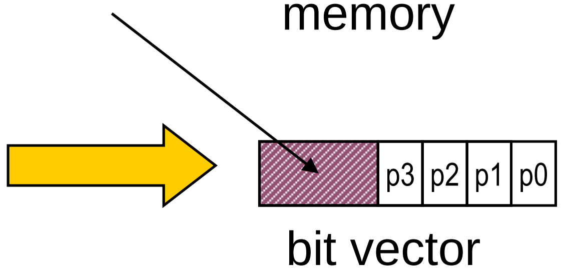
unpacked array of bits

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



packed array of bits

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



1k 16 bit unpacked memory

```
bit [15:0] memory [1023:0];
```

1k 16 bit packed memory

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

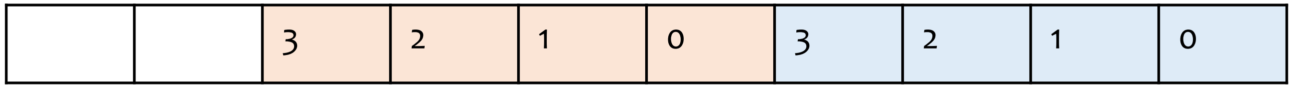
Can operate on entire memory

Packed Array Part Selects

- A part select or slice is a concatenation of bits

```
logic [1:0][3:0] var;
```

This is a 8 bit vector



This is a 3 bit vector

```
var[0][3:1] = 3'b000;
{var[0][3],var[0][2],var[0][1]} = 3'b000;
```

Indexed Array Part Selects

- Starting point of the slice is an expression
- Width of the slice is fixed after compilation

```
bit[255:0] frame;
bit[63:0] grab1,grab2;
assign grab1 = frame[255 -: 64];
assign grab2 = frame[offset +: 64];
```

same as frame[255:192]

variable offset from MSB

Constant Width

- +:indicates the slice increases from the starting point
- -:indicates the slice decreases from the starting point

Multidimensional Arrays

2D array

```
int a [7:0][7:0];
...
a[x][y] = 0;
```

3D array

```
bit ucube [maxx:0][maxy:0][maxz:0];
bit [maxy:0][maxz:0] pcube[maxx:0];
```

Unpacked dimensions referenced first, then packed
From left to right-most dimensions

```
bit [a:0][b:0] ref_order [c:0][d:0][e:0];
ref_order [c][d][e][a][b] = 1'b1;
```

Array Literals

```
int A[2:0] = {0,1,2};
int nines[1:9] = {9{9}};

A = {3,4,5};

real R[1:0][2:0]
  = {{1.5,4.5,4.3},{5.0,0.5,2.1}};

byte Frame[WIDTH:1][HEIGHT:1]
  = {WIDTH{{HEIGHT{8'hA5}}}};
```

Initialization

like concatenation

Assignment

Braces reflect array layout

Array Initialization

```
typedef bit [24:0] mydata_t;

mydata_t [0:3] data_pack = {
        25'd2,25'd4,25'd78,25'd27};
// 100-bit concatenation


mydata_t data_dense[0:3] = {
2,4,78,27};
// data_dense[0] = 2;
// data_dense[1] = 4;
// data_dense[2] = 78;
// data_dense[3] = 27;
```

User-defined type

Packed array

{}=concatenation

unpacked array

{}=array element initialization

SystemVerilog Array System Functions

- $dimensions (array_name) : returns the # of dimensions in the array (or 0 if object is scalar)
- $left (array_name, dimension) : returns MSB no. of specified dimension
- $right (array_name, dimension) : returns LSB
- $low (array_name, dimension) : returns the lowest bit number.
- $high (array_name, dimension) : returns the highest bit number.
- $increment (array_name, dimension) : returns 1 if: $left is >= $right,
  returns –1 if: $left is < $right.
- $length (array_name, dimension) : returns the total # of elements in the specified dimension (same as $high - $low +1)

Advanced Procedural Blocks

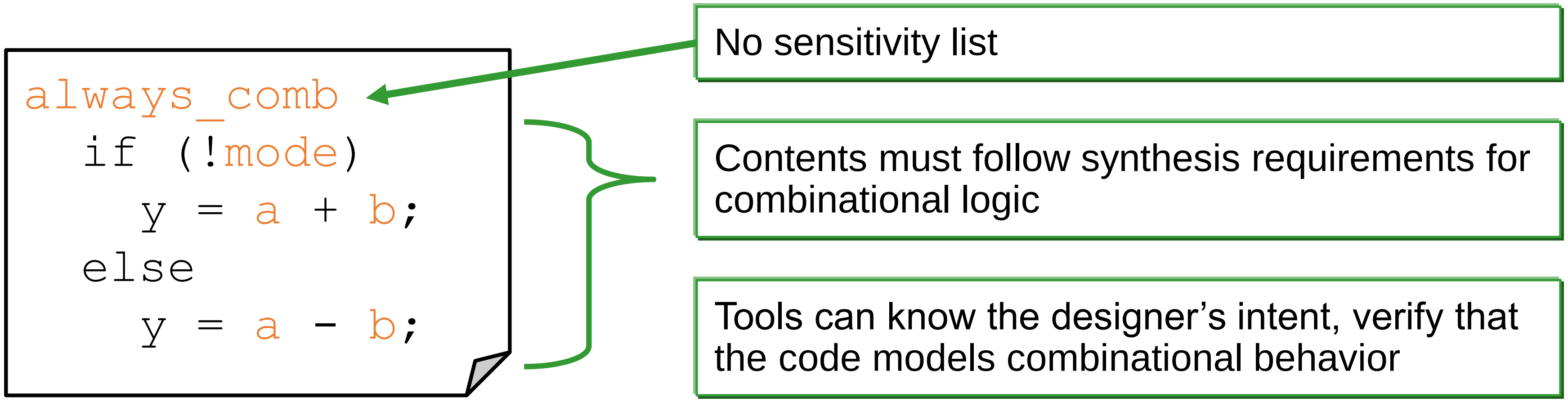
- Logic procedural blocks
- Unique/priority decisions
- Enhancements to tasks and functions

Issues with Verilog General Purpose Procedures

- always procedure used to model multiple logic types:
  - Combinational
  - Latched
  - Sequential
  - Testbench
- Tools must “infer” (guess) hardware intent

SystemVerilog Hardware-Specific Procedures

- always_ff procedure models sequential logic
- always_comb procedure models combinational logic
- always_latch procedure models latch-based logic
- Enables simulation, synthesis, formal tools to use same rules
- Enables designer’s intent to be checked
- Software may issue warnings if behavior doesn’t match style.



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

No sensitivity list

Contents must follow synthesis requirements for combinational logic

Tools can know the designer’s intent, verify that the code models combinational behavior

Combinational Logic Procedure

- always_comb models combinational logic

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

- always_comb rules
  - The sensitivity list is inferred
    - Includes every variable read by the procedure
  - Variables on the left-hand side of the assignment can’t be written to by any other procedure
  - Executes automatically at time 0
    - After all initial and always blocks have been started
    - Ensures the outputs are consistent with the inputs at time 0

Latched Logic Procedures

- always_latch models combinational logic with storage

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

- always_latch rules:
  - sensitivity list is inferred (same rules as always_comb)
  - Variables on the left-hand side of the assignments can’t be written to by any other procedure
  - automatically triggered at time zero

Sequential Logic Procedures

- The always_ff procedure models synthesizable sequential logic behavior
- always_ff is different from a normal always procedure
    - The sensitivity list must specify an edge for each signal
    - No event controls are permitted within the procedure

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

Unique and Priority Decisions

- Synthesis pragmas cause results to diverge
  - full_case –Statement in which all possible case-expression binary patterns can be matched to a case item or to a case default.
  - If a case statement does not include a case default and if it is possible to find a binary case expression that does not match any of the defined case items, the case statement is not "full."

Unique and Priority Decisions

- Synthesis pragmas cause results to diverge
  - parallel_case – Statement in which it is only possible to match a case expression to one and only one case item.
  - If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel."

Unique and Priority Decisions

- SystemVerilog adds unique and priority modifiers for if-else-if decisions and case statements :
  - Gives same information to the simulator and synthesis tool

Unique and Priority Decisions

- priority case
  - Tests each case condition in order and makes sure there is at least one branch taken
  - Equivalent to //pragma full_case
- unique case
  - Tests all case conditions and makes sure that one and only one condition matches
  - Equivalent to //pragma parallel_case

```
unique if (!t[0])
     $display("even");
else if (t[1])
     $display("big");
else $display("small odd");
```

`unique if` means that the `else if` conditions do not overlap

"priority if" gives warning if there is no final else but the else condition happens

Unique and Priority Examples

```
bit [2:0] opcode
always_comb
unique case (opcode)
  3’b000: y = a+b;
  3’b001: y = a-b;
  3’b010: y = a*b;
  3’b100: y = a/b;
endcase
```

```
always_comb begin
  y = '0;
  priority case ({en,a})
    3'b100: y[a] = 1'b1;
    3'b101: y[a] = 1'b1;
    3'b110: y[a] = 1'b1;
    3'b111: y[a] = 1'b1;
  endcase
end
```

Find the errors!

```
always @(posedge clk or negedge rst_)
  priority if (!rst_)
    cnt <= 0;
  else if (en)
    cnt <= cnt + 1 ;
```

warnings issued *during simulation* when (rst_=1 and en=0) or whenever either is "x"

```
always_comb
  if (ctrl == 1)
    op <= a;
```

errors should be issued *during compilation* – does not infer combinational logic

```
always_ff
 begin
  #(period/2) clk <= 0;
  #(period/2) clk <= 1;
  end
```

errors should be issued *during compilation* – block timing inside an always_ff

```
always @(posedge clk iff rst == 0)
  // synchronous reset
  if (rst)
    data_out <= 8'h00;
  else
    data_out <= data_in;
```

procedure never triggered when rst=1, so never reset

SystemVerilog Tasks

- In Verilog:
  - The directions of task arguments must be declared
  - Task arguments default to reg
- SystemVerilog adds:
  - Task arguments have a default direction of input
  - Task arguments default to logic
  - Arrays and Structures may be passed as arguments

SystemVerilog Functions

- In Verilog:
  - Functions can only have inputs
  - The direction of arguments must be declared
  - Arguments default to reg
- SystemVerilog adds:
  - Functions can have input, output, and inout
    - The default direction is input
  - Function arguments default to logic
  - Arrays and Structures may be passed as arguments

SystemVerilog Functions

```
function logic [15:0] func( logic m,
   bit [15:0] in1, in2 );


   return((in1 + in2) ^ m);
// Verilog: func = (in1+in2)^m;
endfunction
```

input keyword optional

return keyword

The return is assigned to the name of the function

can have void functions (or use tasks)

```
function void print(int n);
…
endfunction
```

- Verilog functions – it's an error to ignore the return value
- SystemVerilog functions – it's a warning to ignore the return value

SystemVerilog Tasks With Return

```
task add_to_max (input [5:0] max, output [63:0 result);
  result = 1;
  If(max == 0) return; // exit task
  for(int i=1; i<63; i=i+1)
  begin
            result = result + 5;
            if(i== max)
            return; // exit task
 end
endtask
```

Explicit Task/Function Arguments

- Connect by name like module instance
- Default values for optional arguments

```
function int func(int a,
                  int b = 3,
                  string c = "");
…
endfunction
```

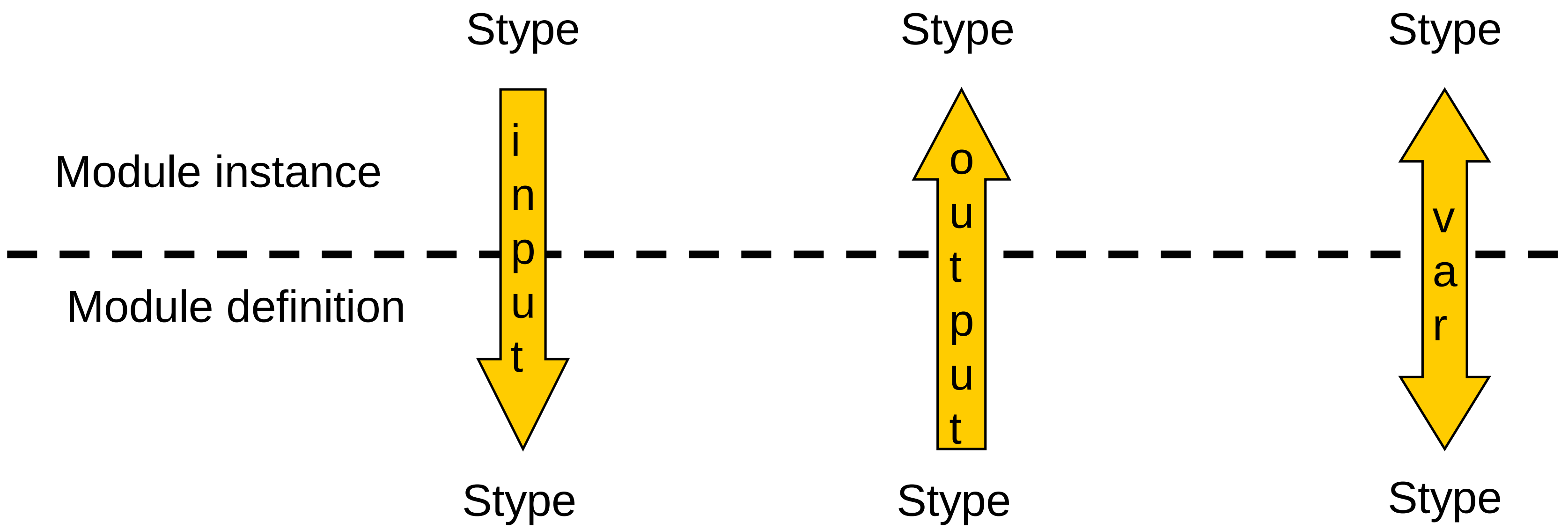
a must be specified when the function is called

```
int i = func(.b(2),.a(4),.c("hello"));
int j = func(.c("bye"),.a(3));// b = 3
int k = func(4);// a = 4, b = 3, c = ""
```

Design Hierarchy, Interfaces and Modports

SystemVerilog Module Ports Rules

- Any SystemVerilog type can pass through a port as input or output
- SystemVerilog types are shared across ports



SystemVerilog's Implicit .name Port Connection

- SystemVerilog simplifies the named port connection syntax
  - If the signal and port are exactly same (name and size), only the port needs to be renamed
- Following rules apply:
  - the signal name and port name must be the same
  - the signal size and the port size must be the same
  - the data types of both the modules must be compatible (e.g. can't implicitly connect an integer to a real)

```
module chip (output q1, q0,
         input [3:0] d,
         input  clk, rst);

      dff dff_inst1 (.clk(clk), .rst(rst), .q(q0), .d(d[0]));

      dff dff_inst2 (.clk, .rst, .q(q0), .d(d[0]));

endmodule
```

```
module dff (output q,
         input clk, d, rst);
..
endmodule
```

Look at clk and rst

Named ports and Implicit .name ports

- Named port
  - Named port connections avoid inadvertent connection errors and is a preferred style for documenting (Advantage!)
  - Named ports are very verbose (Disadvantage !)
- Implicit .name ports
  - .name is an abbreviation of named port connections
  - .name simplifies connections to module instances
  - .name infers a connection of a net and port of the same name
  - .name can be combined with named port connections

Implicit .* Port Connection

- SystemVerilog can automatically connect all ports and signals that have the same name
  - Uses the same rules as .name port connections
  - This really helps in situations of huge netlists

```
module chip (output q1, q0,
             input [3:0] d,
             input clk, rst);

   dff dff_inst (.*, .q(q0),
   .d(d[0]));
endmodule
```

Look at how clk and rst are connected

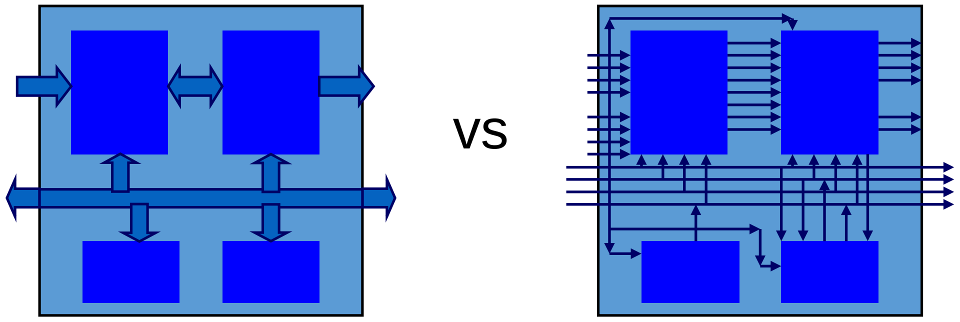
```
module dff (output q, input clk,
   d, rst);
..
endmodule
```

All ports with names that match signals in this module are automatically connected via .*

Communication-Based Design And Verification

- Most Bugs Occur Between Blocks
- Encapsulation is Key
- Capture Interconnect and Communication
- Separate Communication from Functionality
- Eliminates "Wiring" Errors
- Reuse Interface Objects
- Facilitates Divide-and-Conquer Methodology

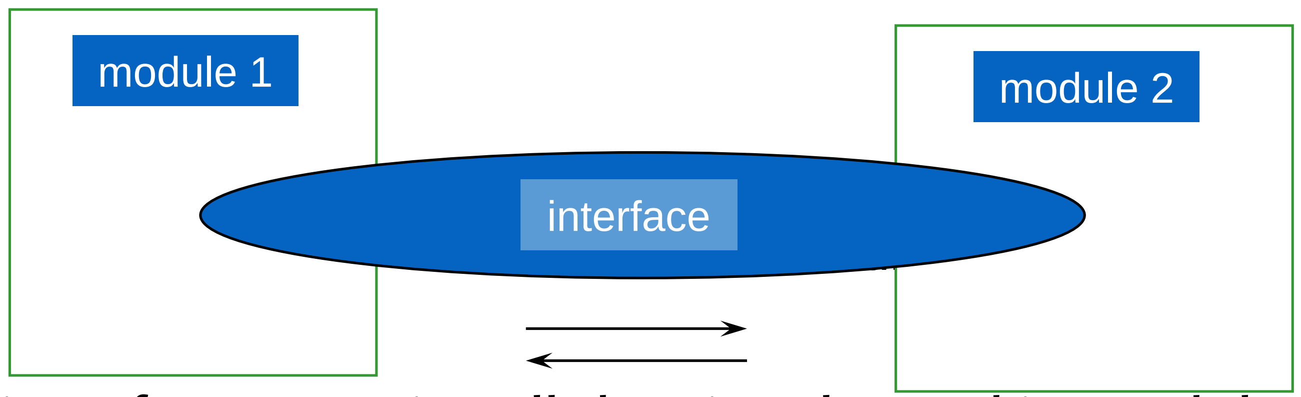
Encapsulation



- Think About the Fat Arrows on a Block Diagram as More Than Just Wire Bundles
  - Wires are an implementation choice of how to communicate information between blocks
  - Focus on the information that is being communicated

Connection Object

- An interface describes the communication between modules



- The interface contains all the signals used in module1 and module2
- The interface can also describe how the data is sent and received

Interfaces are not just for encapsulation ...

What Is An Interface?

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

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

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

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

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

```
wire w;
intf if1;

modA a (w, if1);
```

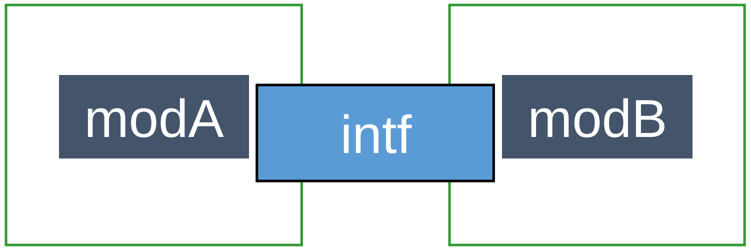
You can think of a wire as a built in interface

- Encapsulates communication like a struct encapsulates data

How Does An Interface Work?

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

intf w;

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

module modA (intf i1);
endmodule

module modB (intf i1);
endmodule
```

Instantiate Interface

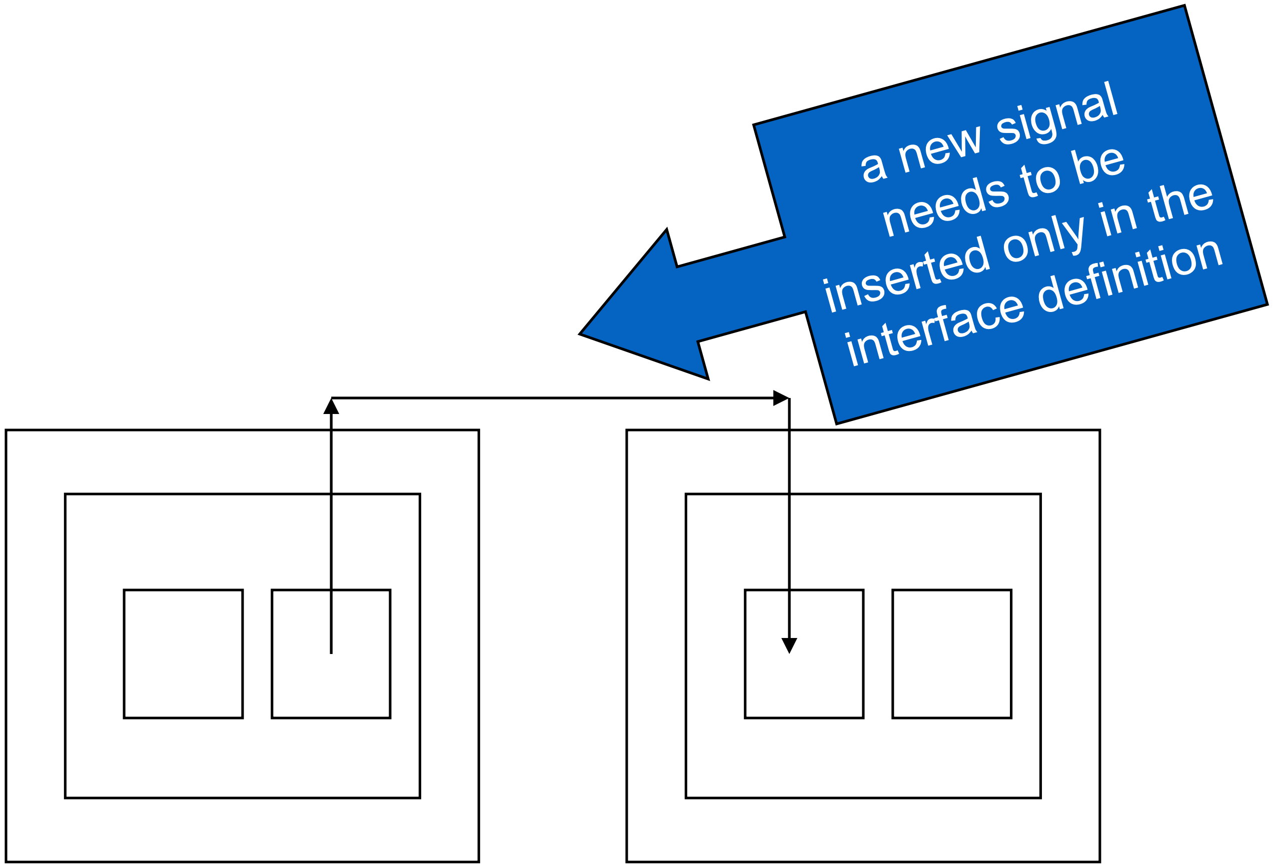


An interface is similar to a module straddling two other modules

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

- Allows structuring the information flow between blocks

Interfaces Keep The Code Maintainable


a new signal needs to be inserted only in the interface definition

- No need to edit dozens of files of intermediate levels to insert just one signal

Interface Contents

- Interfaces are more than just a bundle of wires
  - Interfaces can contain declarations
    - Variables, parameters and other data can be declared in one location and shared with multiple modules
  - Interfaces can contain task and functions
    - Operations shared by all connections to the interface can be coded in one place
  - Interfaces can contain procedures
    - Protocol checking and other verification can be built into the interface

What Is An Interface?

```
interface utopia_i;
  wire  soc;          // start of cell
  wire  en;           // enable
  wire  [7:0] data;   // data
  wire  clav;         // cell available
  wire  clk;          // ATM layer clock
endinterface

interface cpu_i(input bit rst);
   wire        BusMode;
   logic [11:0] Addr;
   logic        Sel;
   wire  [ 7:0] Data;
   logic        Rd_DS;
   logic        Wr_RW;
   wire         Rdy_Dtack;
endinterface

module netproc(utopia_i ux, cpu_i cpu,
                                  input bit clk);
endmodule
```

SystemVerilog

```
module netproc (SX_ux_soc, SX_ux_en, SX_ux_data, SX_ux_clav,
SX_ux_clk, SX_cpu_BusMode, SX_cpu_Addr, SX_cpu_Sel,
SX_cpu_Data, SX_cpu_Rd_DS, SX_cpu_Wr_RW,
SX_cpu_Rdy_Dtack, rst, clk);
  inout SX_ux_soc;
  inout SX_ux_en;
  inout [7:0] SX_ux_data;
  inout SX_ux_clav;
  inout SX_ux_clk;
  inout SX_cpu_BusMode;
  inout [11:0] SX_cpu_Addr;
  inout SX_cpu_Sel;
  inout [7:0] SX_cpu_Data;
  inout SX_cpu_Rd_DS;
  inout SX_cpu_Wr_RW;
  inout SX_cpu_Rdy_Dtack;
  input rst;
  input clk;
  wire SX_ux_soc;
  wire SX_ux_en;
  wire [7:0] SX_ux_data;
  wire SX_ux_clav;
  wire SX_ux_clk;
  wire SX_cpu_BusMode;
  wire [11:0] SX_cpu_Addr;
  wire SX_cpu_Sel;
  wire [7:0] SX_cpu_Data;
  wire SX_cpu_Rd_DS;
  wire SX_cpu_Wr_RW;
  wire SX_cpu_Rdy_Dtack;
  wire rst;
  wire clk;
endmodule
```

Verilog95

- 3X more compact
- Fewer wiring mistakes

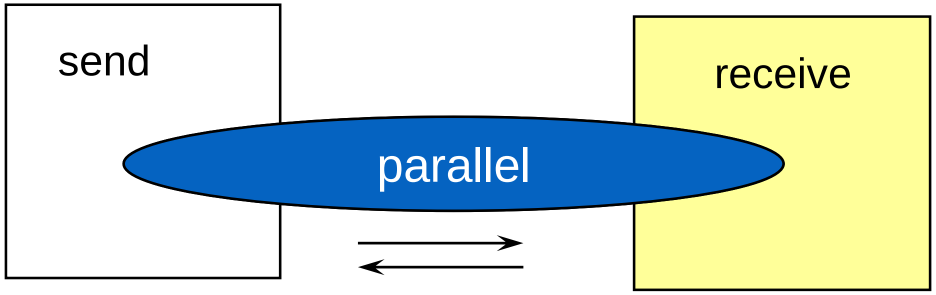
Example System

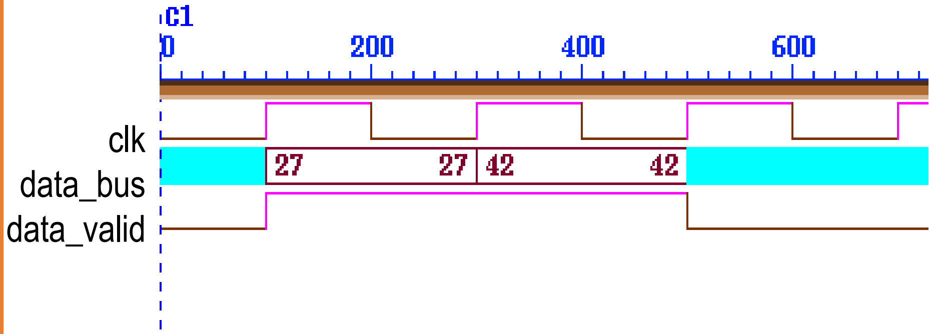
```
module send(input bit clk,parallel ch);
  initial
    begin
      @(posedge clk) ;
        ch.write(27);
        ch.write(42);
      @(posedge clk) ;
      @(posedge clk) ;
      $finish(0);
    end
endmodule
```

Inherit interface

interface task call

```
module receive(input bit clk, parallel ch);

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

  always
    begin
      ch.read(d);
      $display("%t READ a %d", $time,  d);
    end
endmodule
```





Encapsulating Communication

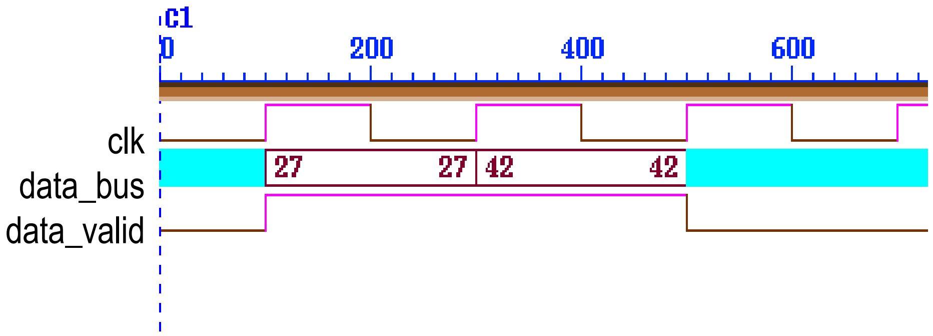
Parallel Interface

```
interface parallel(input bit clk);

  logic [31:0] data_bus;
  logic data_valid=0;

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

  task write(input data_type d);
    data_bus <= d;
    data_valid <= 1;
    @(posedge clk)
      data_bus <= 'z;
      data_valid <= 0;
  endtask


  task read(output data_type d);
    while (data_valid !== 1)
      @(posedge clk);
        d = data_bus;
      @(posedge clk) ;
  endtask

endinterface
```

Interface port

Member signals

write and read tasks/methods



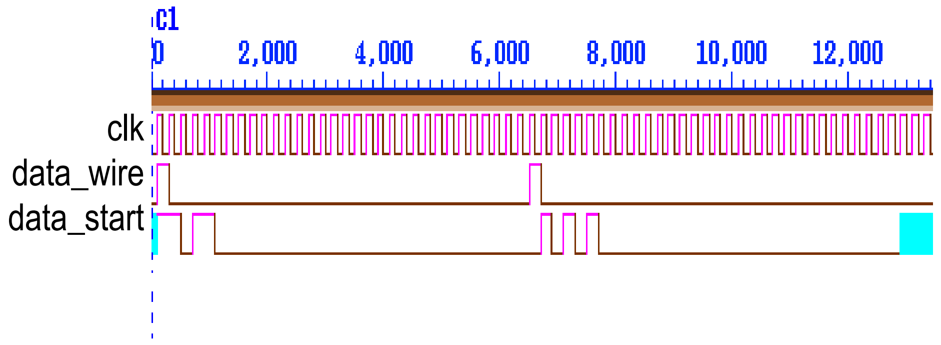
- An interface task/function is a communication method

Alternate Interface

```
interface serial(input bit clk);
  logic data_wire;
  logic data_start=0;

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

  task write(input data_type d);
    for (int i = 0; i <= 31; i++)
      begin
        if (i==0)
          data_start <= 1;
        else
          data_start <= 0;
          data_wire = d[i];
        @(posedge clk)
          data_wire = 'x;
      end
  endtask

task read(output data_type d);
    while (data_start !== 1)
      @(negedge clk);
    for (int i = 0; i <= 31; i++)
      begin
        d[i] <= data_wire;
        @(negedge clk) ;
      end
  endtask

endinterface
```

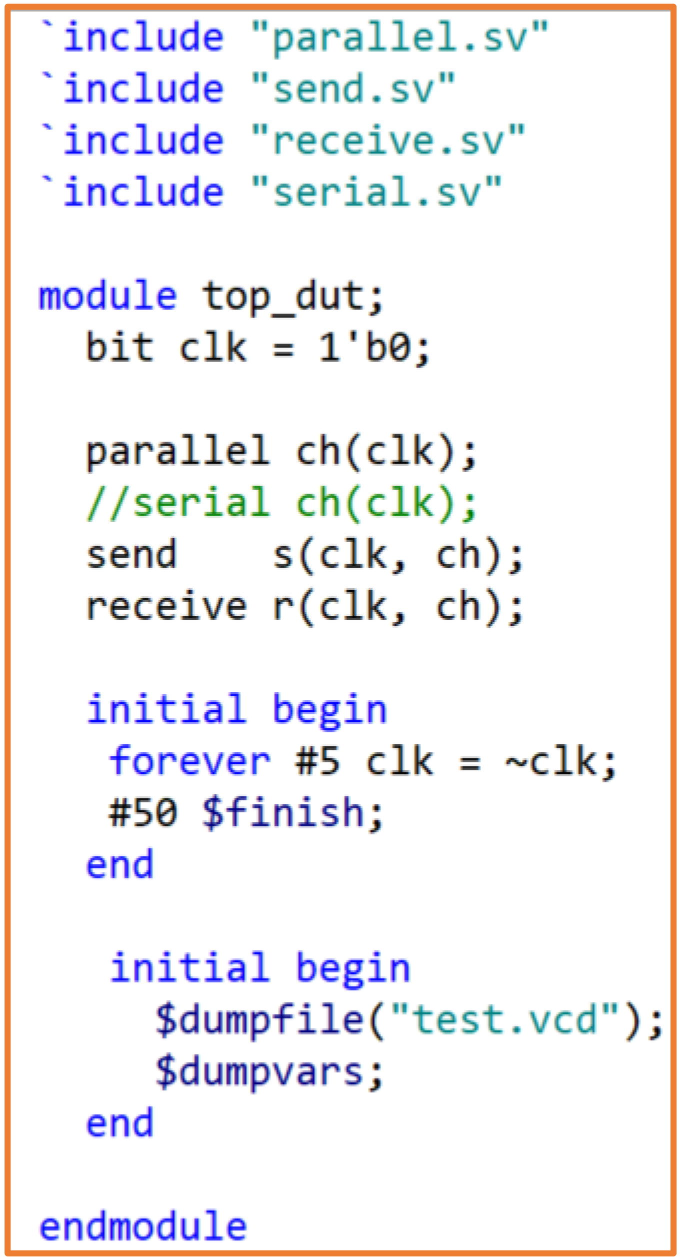
serial signals

Serial Interface



Using Different Interfaces

Top Module



```
`include "parallel.sv"
`include "send.sv"
`include "receive.sv"
`include "serial.sv"

module top_dut;
  bit clk = 1'b0;

  parallel ch(clk);
  //serial ch(clk);
  send    s(clk, ch);
  receive r(clk, ch);

  initial begin
   forever #5 clk = ~clk;
   #50 $finish;
  end

   initial begin
     $dumpfile("test.vcd");
     $dumpvars;
  end

endmodule
```

Controlling Signal/Port Direction: Modports

- If Direction is Not Specified, Signals in Interfaces Are inout
- How to Specify Direction When Ports Are Different for Different Modules?
  - CPU module uses addr as an output
  - Mem module uses addr as an input
- Group Signals/Directions in Interface
  - “Modport” implies direction from point-of-view of the module using the interface
  - Can also control visibility of signals and methods

Simple Example Using Modports In Definition

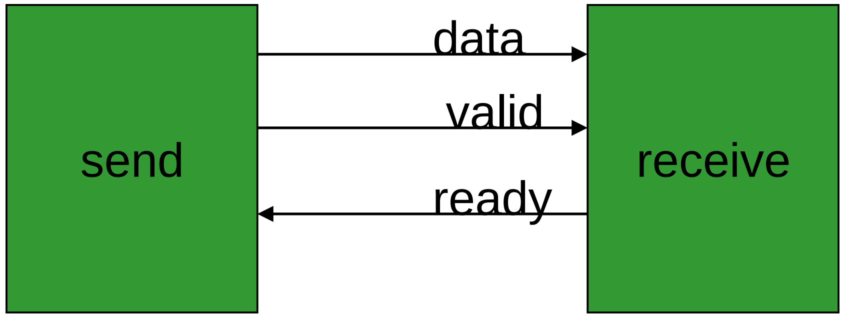
```
interface simple_bus(input bit clk);
  logic req,gnt;
  logic [7:0] addr,data;
  logic [1:0] mode;
  logic start,rdy;
  modport slave(input req,addr,mode,
                      start, clk,
                output gnt,rdy,
                inout data);
  modport master(input gnt,rdy,clk,
          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

module memMod(simple_bus.slave a);
  ...
endmodule

module cpuMod(simple_bus.master b);
endmodule
```

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

  memMod mem(sb_intf);
  cpuMod cpu(sb_intf);
endmodule
```

modport picked up from module definition

interface.modport hard-coded in module definition

Assignment



- Send data when ready is active low and make sure valid signal is active high while sending data
- Receive data when valid is active low and make sure ready is active high while receiving data

- The model sends and receives random data. The data can be either parallel (32bits) or serial 1 bit at a time.
- send the data using the write task
- receive the data using the read task
- The Testbench
  - 10, 32 bit random data are transmitted in this testbench.