Introduction to SystemVerilog & Advanced Testbench

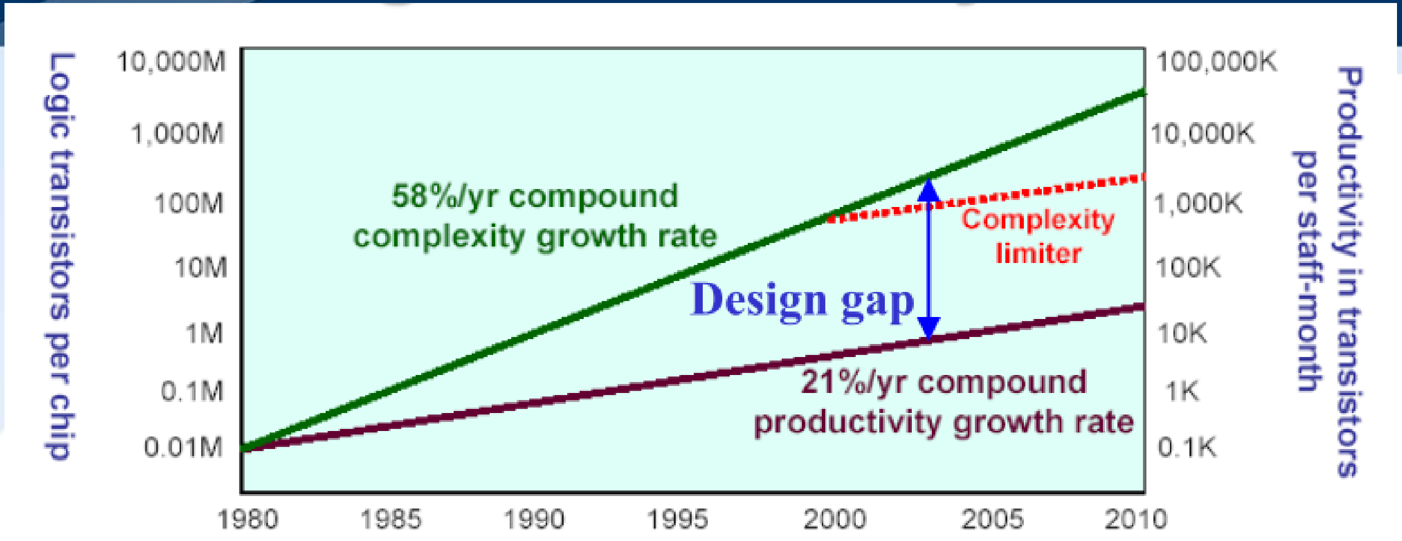
Lecturer: Jui-Huang Tsai

Outline

Outline

- ✓ **Section 1 Introduction**
- ✓ **Section 2 Design using SystemVerilog**
  - Data Type, enumerate, structure, union
  - Procedure Block
  - Interface
- ✓ **Section 3 Verification using SystemVerilog**
  - OOP (Object-Oriented Programming)
  - Randomization

Why SystemVerilog?



- ✓ **Human factors may limit design more than technology**
- ✓ **Keys to solve the productivity crisis**
  - Higher abstract language to enhance design & verification
  - Design techniques: hierarchical design, SoC design (IP reuse, platform-based design), etc
  - CAD: algorithms & methodology

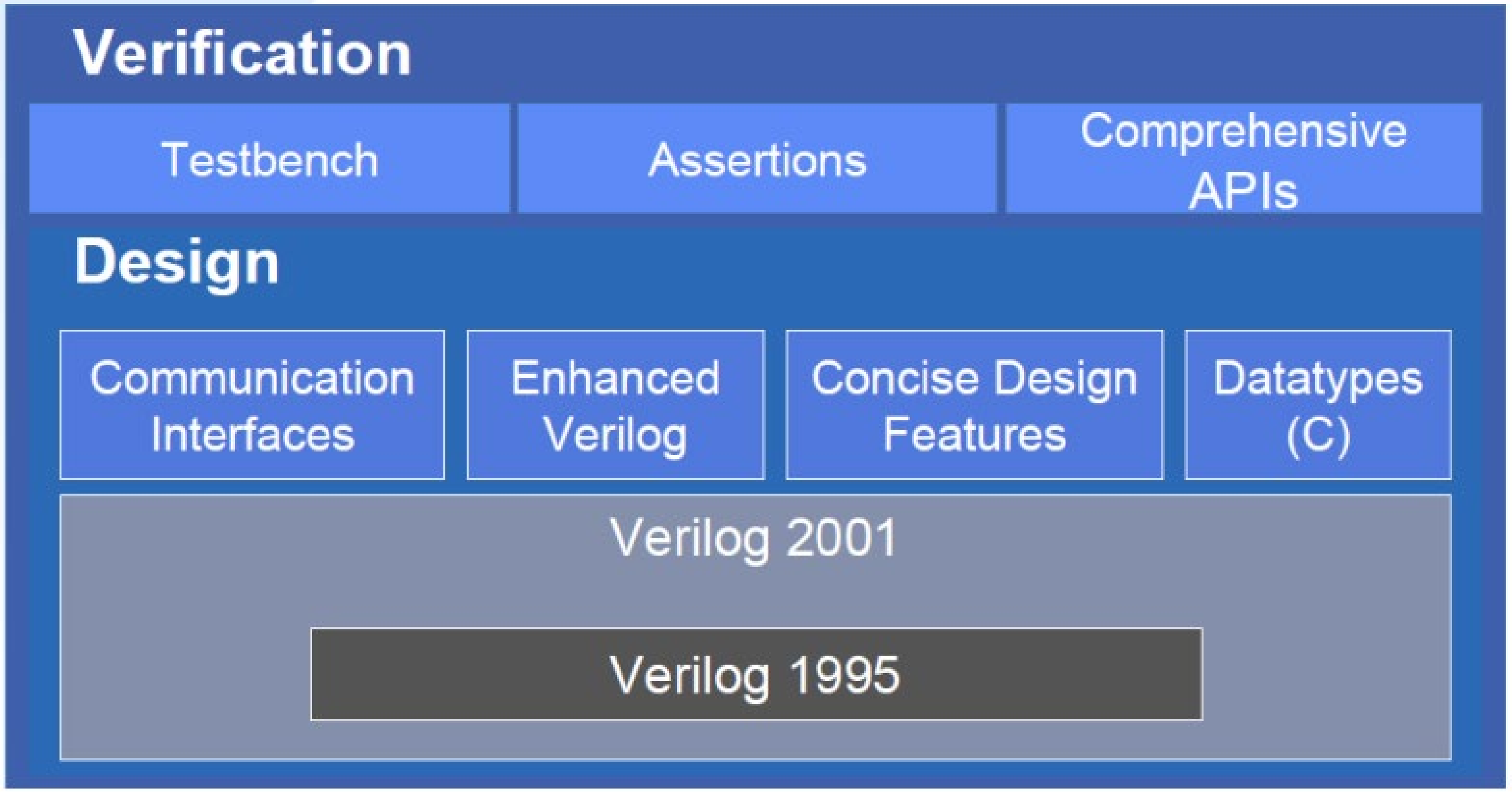
Source: http://access.ee.ntu.edu.tw/course/under_project_1011/slides/20120925_IC_Design_Flow.pdf

Why SystemVerilog?

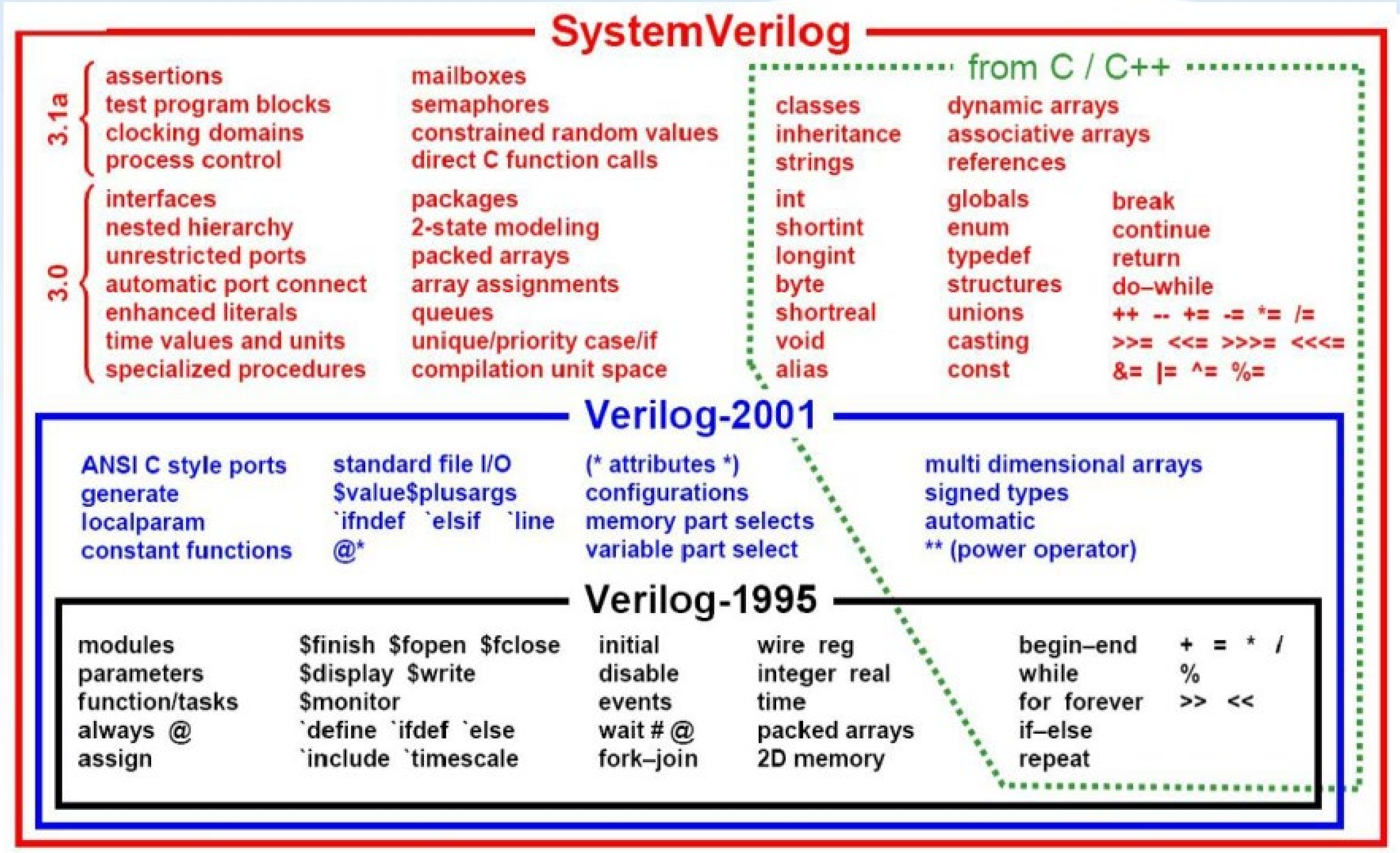
- ✓ **Designs efficiency**
  - Need to code for reuse and higher abstraction
  - Need more efficient coding constructs with native language support
- ✓ **System level hardware design/verification languages**
  - Unification of both syntax and semantics with one language improves communication between design team and verification team
- ✓ **Testbenches are growing exponentially**
  - Find a way to generate more useful test data and make testing faster
  - More advanced verification techniques (ex: OOP)
  - Verification Library (ex: UVM, Universal Verification Methodology)

What is SystemVerilog?

- ✓ **The third generation of Verilog**
  - Specified in IEEE Std 1800-2005
- ✓ **Fully backward compatible with Verilog 2001**



Overview of SystemVerilog



SystemVerilog

3.1a

assertions
test program blocks
clocking domains
process control

mailboxes
semaphores
constrained random values
direct C function calls

3.0

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

packages
2-state modeling
packed arrays
array assignments
queues
unique/priority case/if
compilation unit space

from C / C++

classes
inheritance
strings

dynamic arrays
associative arrays
references

int
shortint
longint
byte
shortreal
void
alias

globals
enum
typedef
structures
unions
casting
const

break
continue
return
do–while
++ -- += -= *= /=
>>= <<= >>>= <<<=
&= |= ^= %=

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

modules
parameters
function/tasks
always @
assign

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

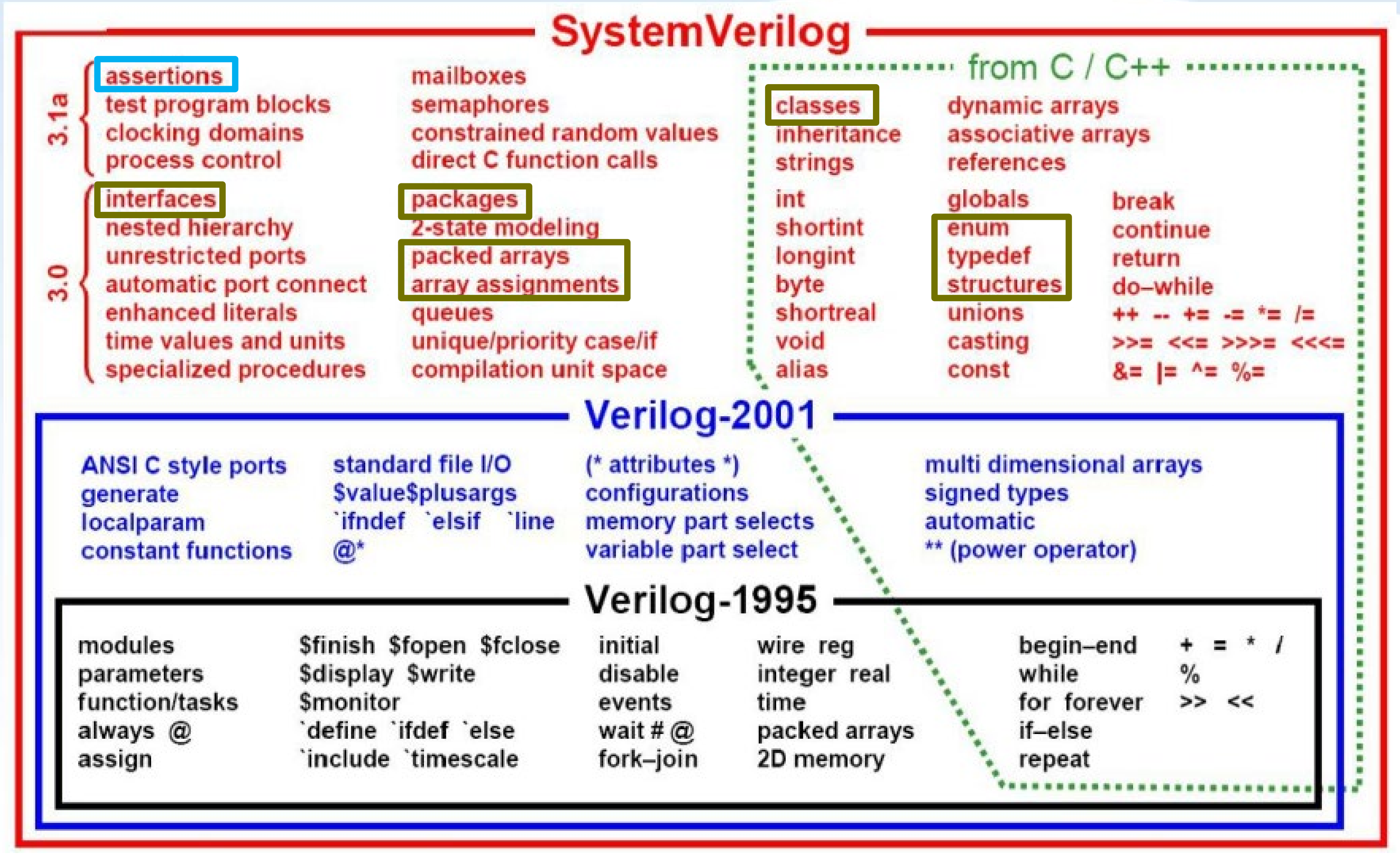
initial
disable
events
wait # @
fork–join

wire reg
integer real
time
packed arrays
2D memory

begin–end
while
for forever
if–else
repeat

+ = * /
%
>> <<

Overview of SystemVerilog



SystemVerilog

3.1a

assertions
test program blocks
clocking domains
process control

mailboxes
semaphores
constrained random values
direct C function calls

3.0

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

packages
2-state modeling
packed arrays
array assignments
queues
unique/priority case/if
compilation unit space

from C / C++

classes
inheritance
strings

dynamic arrays
associative arrays
references

int
shortint
longint
byte
shortreal
void
alias

globals
enum
typedef
structures
unions
casting
const

break
continue
return
do–while
++ -- += -= *= /=
>>= <<= >>>= <<<=
&= |= ^= %=

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

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

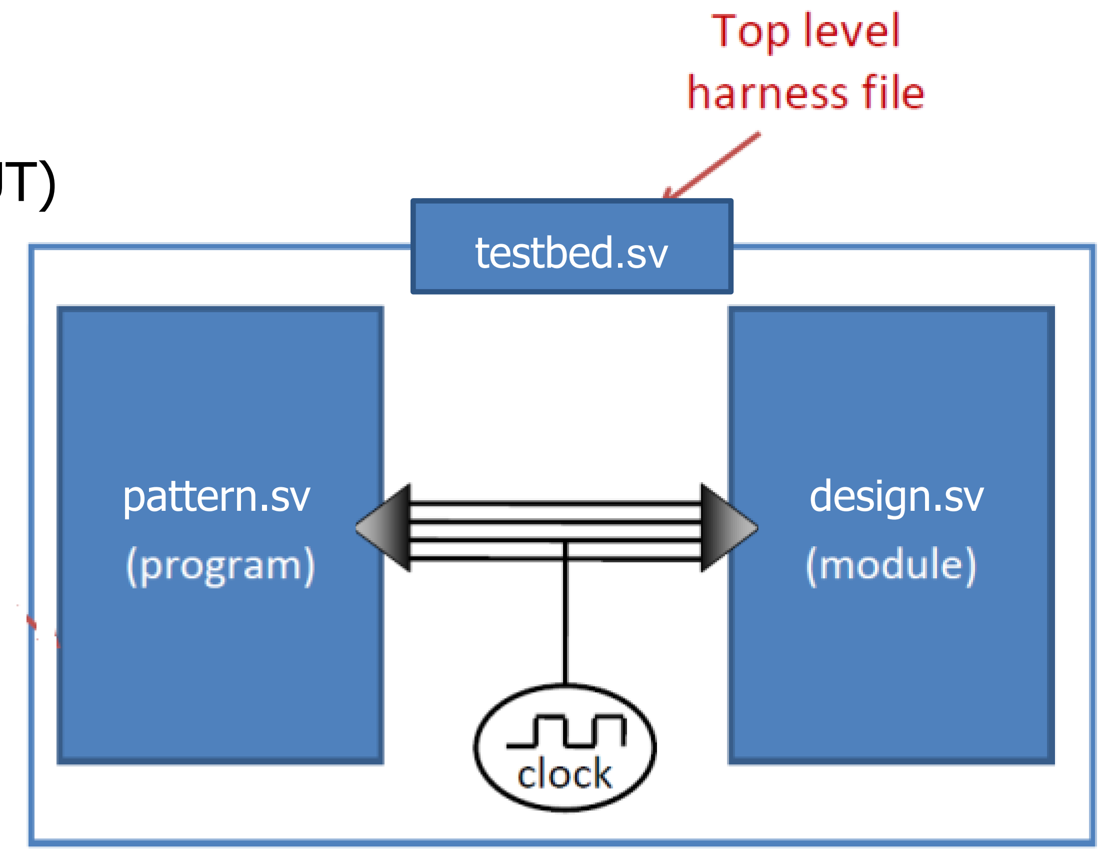
+ = * /
%
>> <<

The SystemVerilog Design Environment

✓ **testbed.sv**
- Connecting testbench and design modules
- Generating clock
- Dump waveform

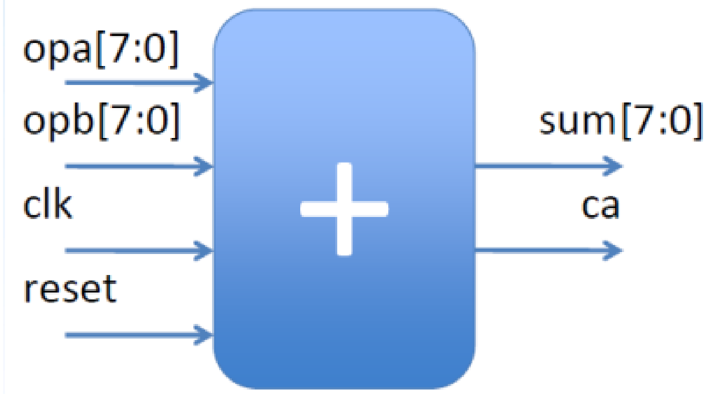
✓ **design.sv**
- Design under test (DUT)

✓ **pattern.sv**
- Pattern
- Test program
- Assertion
- Coverage



Simple Example: 8-bit Adder (1/3)

✓ RTL description



```
module adder(
        input logic [7:0] opa,opb,
        output logic [7:0] sum, output logic ca,
        input logic, clk, reset);

logic [8:0] result;

always_ff@(posedge clk , negedge reset)
begin :p_ADDER_BLOCK
        if(!reset) begin
                ca <= 0;
                sum <= 0;
        end
        else begin
                ca <= result [8] ;
                sum <= result [7:0] ;
        end
end : p_ADDER_BLOCK
always_comb
        result = opa + opb ;
endmodule
```

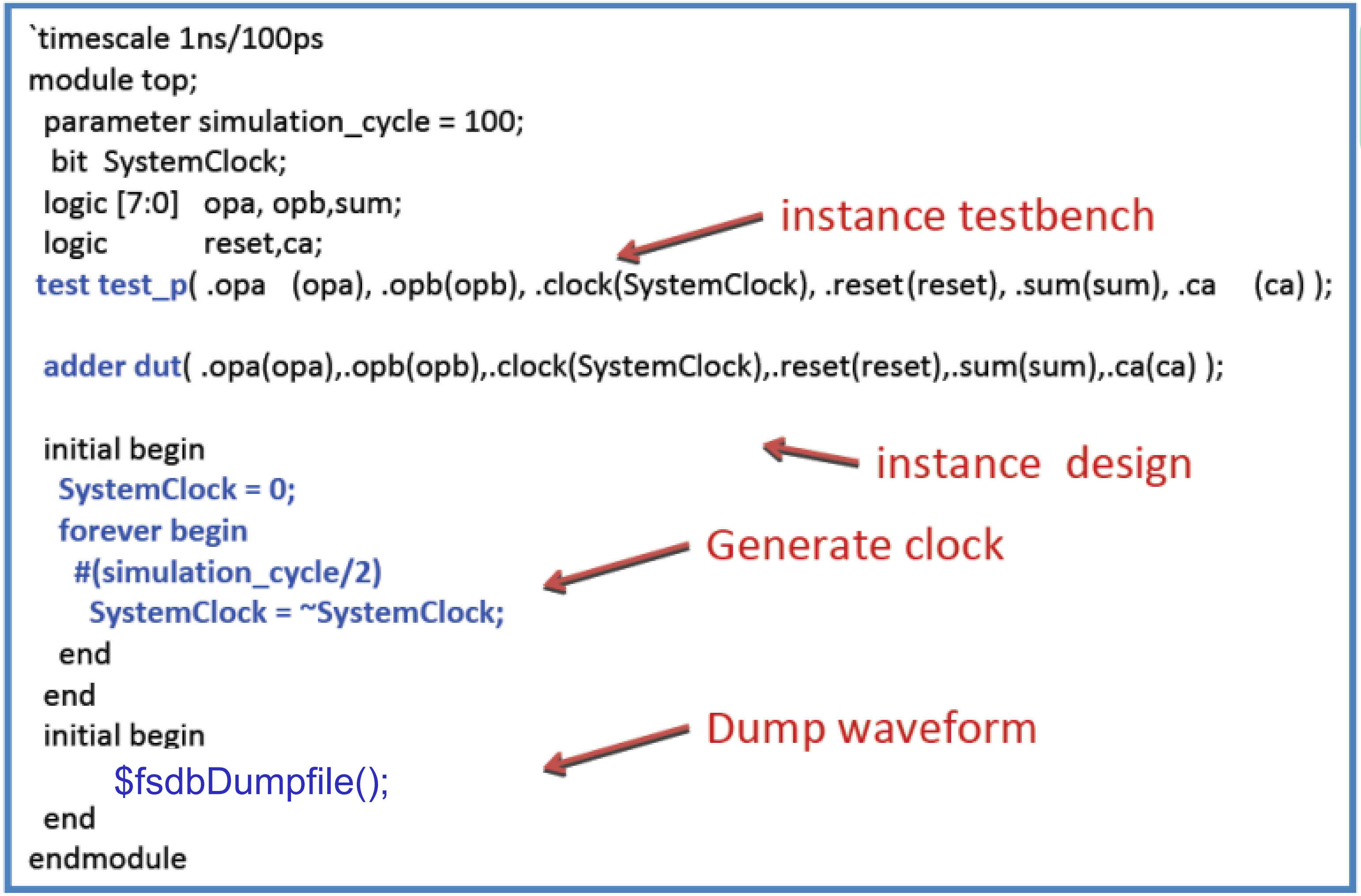
Simple Example: 8-bit Adder (2/3)

- **Test pattern**
  - Generate stimulus
  - Monitor response

```
program automatic test(output logic [7:0] opa,
                      output logic [7:0] opb,
                      output logic reset,
                      input wire clock,
                      input logic [7:0] sum,
                      input logic ca);

initial begin
        reset <= 1'b1;
        opa <=0;
        opb <=0;
        repeat(5) @ (posedge clock);
        reset <= 1'b0;
        repeat(5) @ (posedge clock);
        reset <= 1'b1;
        opa<= 10;
        opb<= 20;
        @(posedge clock);
        $display("%d + %d = %d",opa,opb,sum);
        repeat(5) @(posedge clock);
        $display("Adder Simulation End");
end
endprogram
```

Simple Example: 8-bit Adder (3/3)

✓ Top Level Harness file



```
`timescale 1ns/100ps
module top;
  parameter simulation_cycle = 100;
   bit  SystemClock;
  logic [7:0]   opa, opb,sum;
  logic         reset,ca;
 test test_p( .opa   (opa), .opb(opb), .clock(SystemClock), .reset(reset), .sum(sum), .ca    (ca) );

  adder dut( .opa(opa),.opb(opb),.clock(SystemClock),.reset(reset),.sum(sum),.ca(ca) );

  initial begin
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end
  initial begin
          $fsdbDumpfile();
  end
endmodule
```

```
Clock generated example:
forever #5ns clock = ~clock;
```

Outline

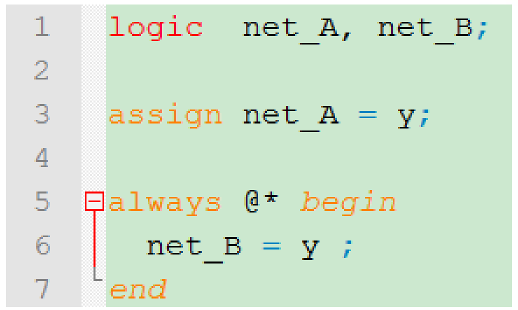
Outline

Data Type in Verilog

- **The Verilog language uses the reg type as a general purpose variable**
  - Both combinational and sequential circuits can be model using reg
  - reg type has no connection to register, which often causes misunderstanding
- **Lacks of user-defined type**

Data Type in SystemVerilog

- **4-State Variables (1|0|Z|X)**
  (default value is X if not initialized)
  - logic w;   // can be used in both assignment and procedure blocks
    - No input and output restriction anymore
    - No continuous or block procedure restriction
  - reg r;   // the same as logic
  - integer i; // Verilog-2001 >= fixed size (32-bit data type)



```
logic   net_A, net_B;

assign net_A = y;

always @* begin
  net_B = y ;
end
```

- **2-State Variables (1|0)**
  (Default value is 0 if not initialized)
  - shortint s;   // 16-bit signed integer
  - int i;   // 32-bit signed integer
  - longint l;   // 64-bit signed integer
  - byte b8;   // 8-bit signed integer
  - bit b;   // 1-bit integer

Synthesis About Data type

- ✓ **The bit, byte, shortint, int and longint types only store 2-state values. Synthesis treats these types as a 4-state reg variable with a corresponding vector size.**
- ✓ **There is a risk of a functional mismatch between simulation and the synthesized implementation, because synthesis does not maintain the 2-state behavior.**
- ✓ **Recommendation — Use logic for almost all declarations. Avoid all 2-state types in RTL models. These types can hide design problems.**
  - Can lead to simulation vs. synthesis mismatches
- ✓ **The one exception is to use an int variable for the iterator variable in for-loops.**

Array

✓ **Verilog-2001:**

- A dimension declared before the object name is referred to as the “vector width” dimension.
- The dimensions declared after the object name are referred to as the “array” dimensions.

  reg [7:0] r1 [1:256]; //[7:0] is the vector width,[1:256] is the array size

✓ **SystemVerilog :**

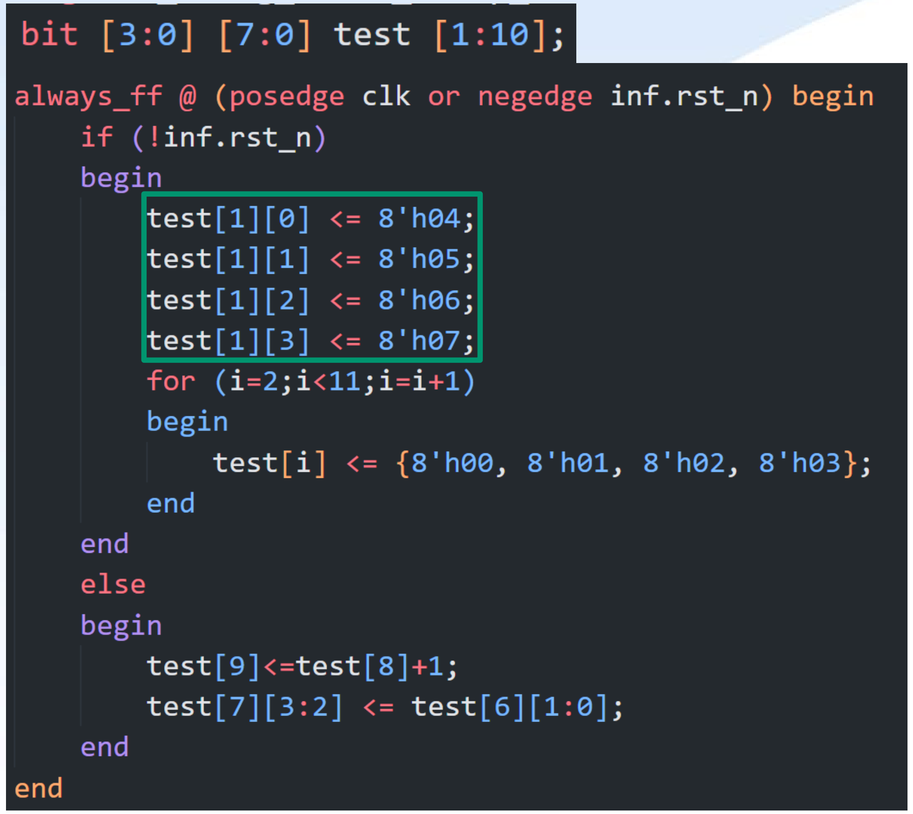
- “***packed array***” to refer to the dimensions declared before the object name (what Verilog-2001 refers to as the vector width)
- “***unpacked array***” is used to refer to the dimensions declared after the object name

```
bit [7:0] c1; // packed array
real u [7:0]; // unpacked array
```

- Example

```
bit [3:0] [7:0] test [1:10]; // 10 entries of 4 bytes (packed into 32 bits)
// can be used as follows:
test [9] = test[8] + 1; // 4 byte add
test [7][3:2] = test [6][1:0]; // 2 byte copy
```

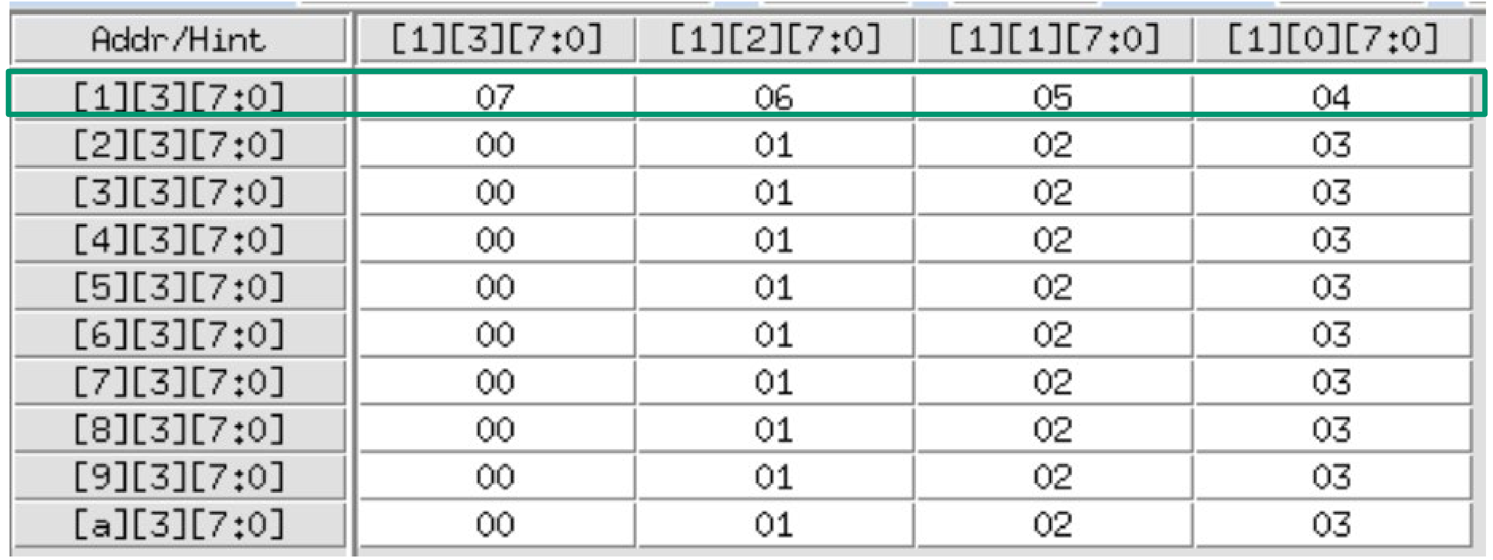
Example: bit[3:0][7:0]test[1:10];



```
bit [3:0] [7:0] test [1:10];
always_ff @ (posedge clk or negedge inf.rst_n) begin
    if (!inf.rst_n)
    begin
        test[1][0] <= 8'h04;
        test[1][1] <= 8'h05;
        test[1][2] <= 8'h06;
        test[1][3] <= 8'h07;
        for (i=2;i<11;i=i+1)
        begin
            test[i] <= {8'h00, 8'h01, 8'h02, 8'h03};
        end
    end
    else
    begin
        test[9]<=test[8]+1;
        test[7][3:2] <= test[6][1:0];
    end
end
```

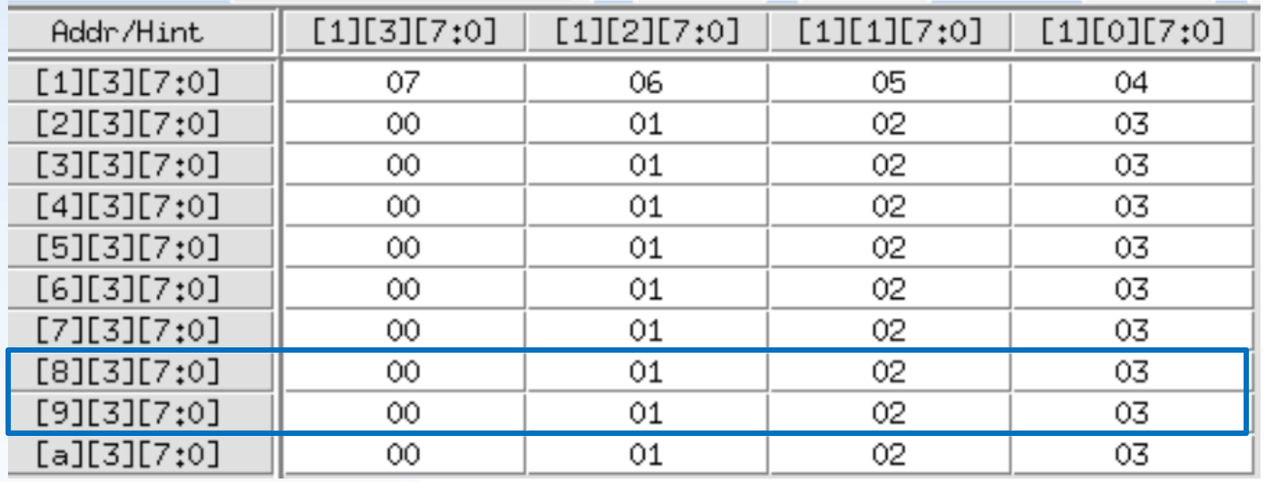
8 bits * 4

10



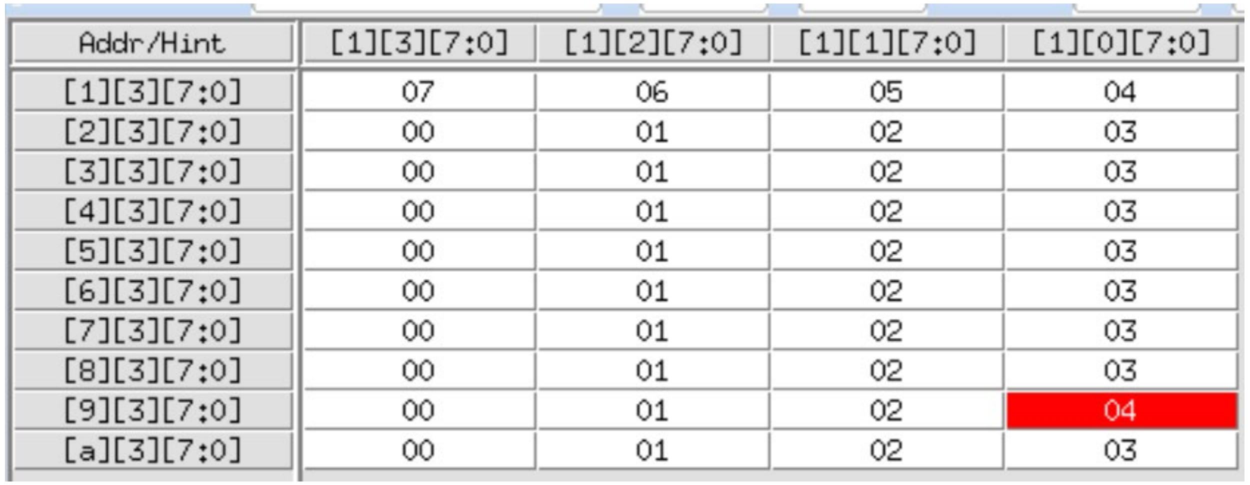
Addr/Hint	[1][3][7:0]	[1][2][7:0]	[1][1][7:0]	[1][0][7:0]
[1][3][7:0]	07	06	05	04
[2][3][7:0]	00	01	02	03
[3][3][7:0]	00	01	02	03
[4][3][7:0]	00	01	02	03
[5][3][7:0]	00	01	02	03
[6][3][7:0]	00	01	02	03
[7][3][7:0]	00	01	02	03
[8][3][7:0]	00	01	02	03
[9][3][7:0]	00	01	02	03
[a][3][7:0]	00	01	02	03

Example: test[9]<=test[8]+1;



Addr/Hint	[1][3][7:0]	[1][2][7:0]	[1][1][7:0]	[1][0][7:0]
[1][3][7:0]	07	06	05	04
[2][3][7:0]	00	01	02	03
[3][3][7:0]	00	01	02	03
[4][3][7:0]	00	01	02	03
[5][3][7:0]	00	01	02	03
[6][3][7:0]	00	01	02	03
[7][3][7:0]	00	01	02	03
[8][3][7:0]	00	01	02	03
[9][3][7:0]	00	01	02	03
[a][3][7:0]	00	01	02	03

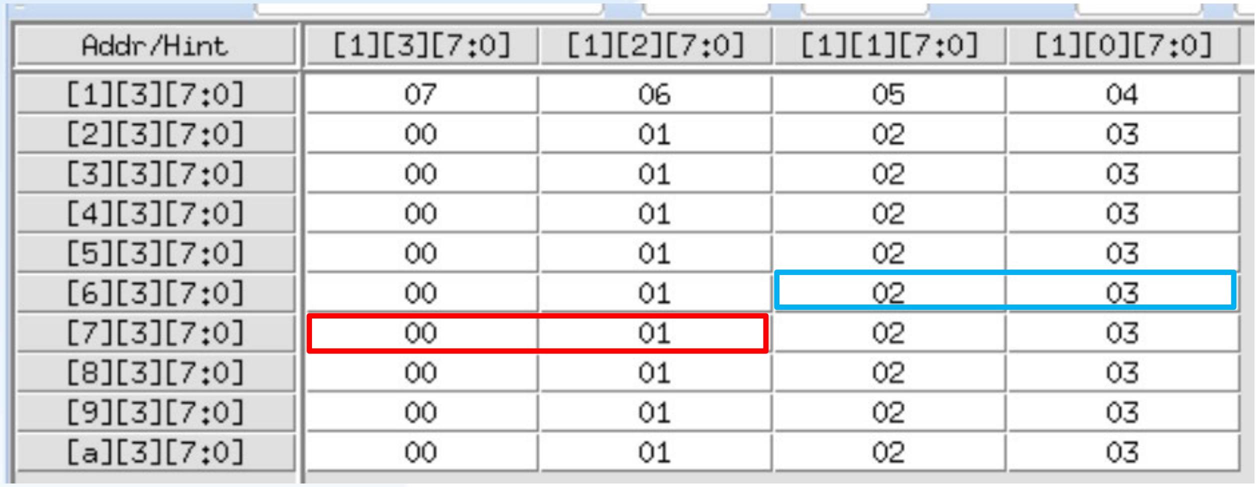
Before



Addr/Hint	[1][3][7:0]	[1][2][7:0]	[1][1][7:0]	[1][0][7:0]
[1][3][7:0]	07	06	05	04
[2][3][7:0]	00	01	02	03
[3][3][7:0]	00	01	02	03
[4][3][7:0]	00	01	02	03
[5][3][7:0]	00	01	02	03
[6][3][7:0]	00	01	02	03
[7][3][7:0]	00	01	02	03
[8][3][7:0]	00	01	02	03
[9][3][7:0]	00	01	02	04
[a][3][7:0]	00	01	02	03

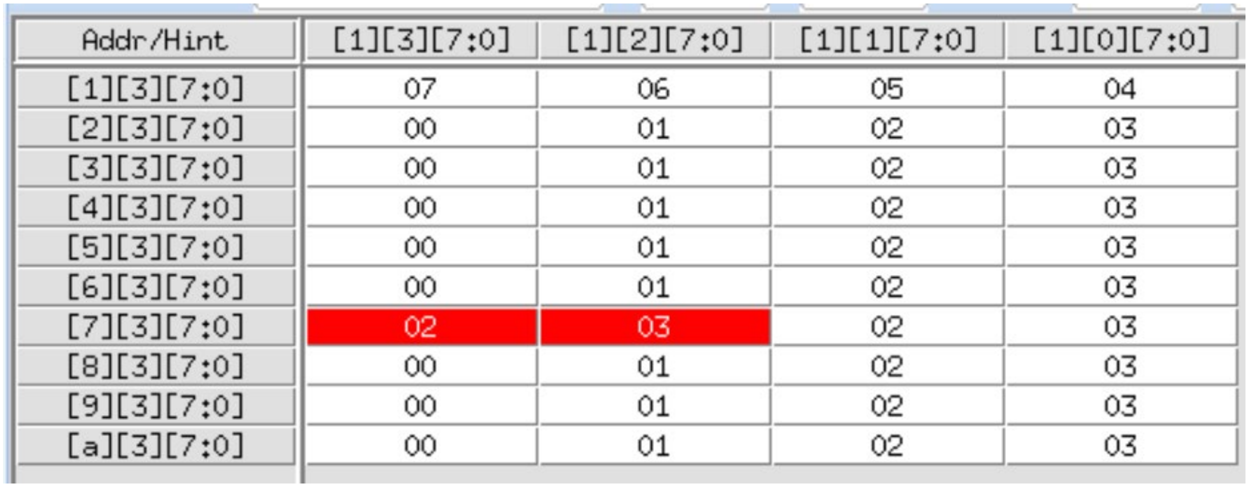
After

Example: `test[7][3:2]<=test[6][1:0];`



Addr/Hint	[1][3][7:0]	[1][2][7:0]	[1][1][7:0]	[1][0][7:0]
[1][3][7:0]	07	06	05	04
[2][3][7:0]	00	01	02	03
[3][3][7:0]	00	01	02	03
[4][3][7:0]	00	01	02	03
[5][3][7:0]	00	01	02	03
[6][3][7:0]	00	01	02	03
[7][3][7:0]	00	01	02	03
[8][3][7:0]	00	01	02	03
[9][3][7:0]	00	01	02	03
[a][3][7:0]	00	01	02	03

Before



Addr/Hint	[1][3][7:0]	[1][2][7:0]	[1][1][7:0]	[1][0][7:0]
[1][3][7:0]	07	06	05	04
[2][3][7:0]	00	01	02	03
[3][3][7:0]	00	01	02	03
[4][3][7:0]	00	01	02	03
[5][3][7:0]	00	01	02	03
[6][3][7:0]	00	01	02	03
[7][3][7:0]	02	03	02	03
[8][3][7:0]	00	01	02	03
[9][3][7:0]	00	01	02	03
[a][3][7:0]	00	01	02	03

After

Typedef

- **Make code clear**
  - SystemVerilog's data type system allows you to define quite complex types. To make this kind of code clear, the **typedef** facility was introduced
  - **typedef** allows users to create their own names for type definitions that they will use frequently in their code

- **Example1:**

```
typedef reg [7:0] octet;
octet b;
// is the same as
reg [7:0] b;
```

- **Example2:**

```
typedef octet [3:0] quadOctet;
quadOctet qBytes [1:10];
// is the same as
reg [3:0][7:0] qBytes [1:10];
```

Structure

- ✓ **Group related signals to enhance readability and clearly convey designer's intent (ex: CPU's instruction)**
- ✓ **Create structures data types:**
  - Structures are a collection of variables
    - **typedef struct**{

```
<data_type> variable0;
<data_type> variable1;
}struct_type;
```

- ✓ **Create structure variables:**
    - **struct**{

```
<data_type> variable0;
<data_type> variable1;
}struct_variable;
```

Structure

✔ **Example:**

```
typedef struct{
    int a,b;
    opcode_t opcode;
    logic[31:0] address;
    bit error;
}Instruction;

Instruction IR;
IR.address = 32'hF000001E;
```

```
struct{
    int a,b;
    opcode_t opcode;
    logic[31:0] address;
    bit error;
}IR;

IR.address = 32'hF000001E;
```

```
typedef logic [6:0] Score;

typedef struct{
 Score Chinese[0:2];
 Score Math[0:2];
 Score English[0:2];
} student;
// Three Exams

student A1;
A1.Chinese[0] = 90;
A1.Math[1] = 87;
```

A1's score	0	1	2
Chinese	90	x	x
Math	x	87	x
English	x	x	x

Enumerate

- ✓ **Create enumerated data types:**
  - Defines a set of named values, which provides <u>built-in assertion</u>!!
  - Data type defaults to ***int*** (32-bit, 2state, signed int)
  - Variable initialized to 0 if initial values aren't specified

- ✓ **Create enum variables:**

  // anonymous enumerated type
  - **enum** <type> {lists_of_enumerations} <enumvar>;

```
enum { circle, ellipse, freeform } c;
```

  // typed enumerated type
  - **typedef enum** <type> {lists_of_enumerations} <enumtype>; enumtype variable_name;

```
typedef enum { circle, ellipse, freeform } ClosedCurve;
ClosedCurve c;
```

- ✓ **Often use to represent state machine**

Enumerate

✓ Example1:

```
// default int type
enum {red, yellow, green} light1, light2;
enum {bronze=3, silver, gold} medal;          // silver=4, gold=5
enum {a=0, b=7, c, d=8} alphabet;             // Syntax error

// enumerated type with a 2-bit logic type
typedef enum logic [1:0] {WAIT, LOAD, READY} state_t;
state_t state, next_state;
```

✓ Example2:

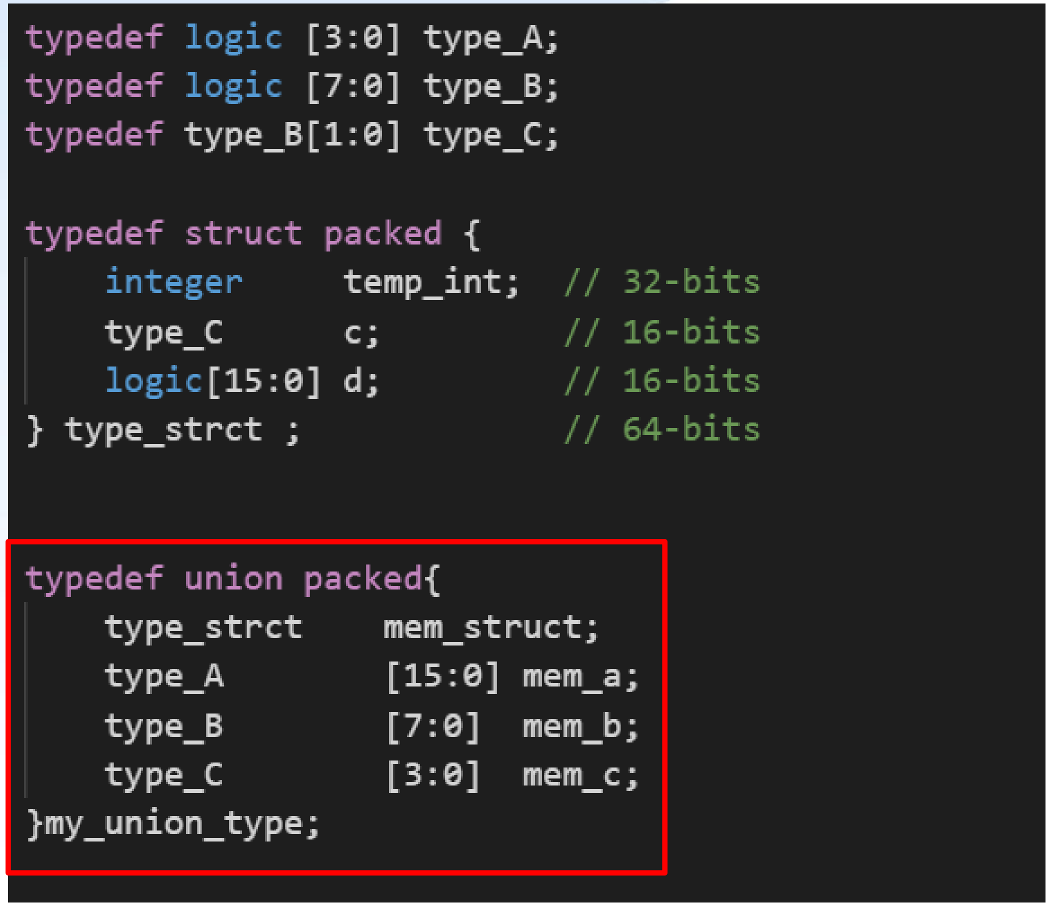
```
// Must match the size of the base type (default to int)
enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, READY=3'b100} state;

//syntax error!!! (Error enum value size)
enum {WAIT=3'b001, LOAD=3'b010, READY=3'b100} state;
```

Example3:

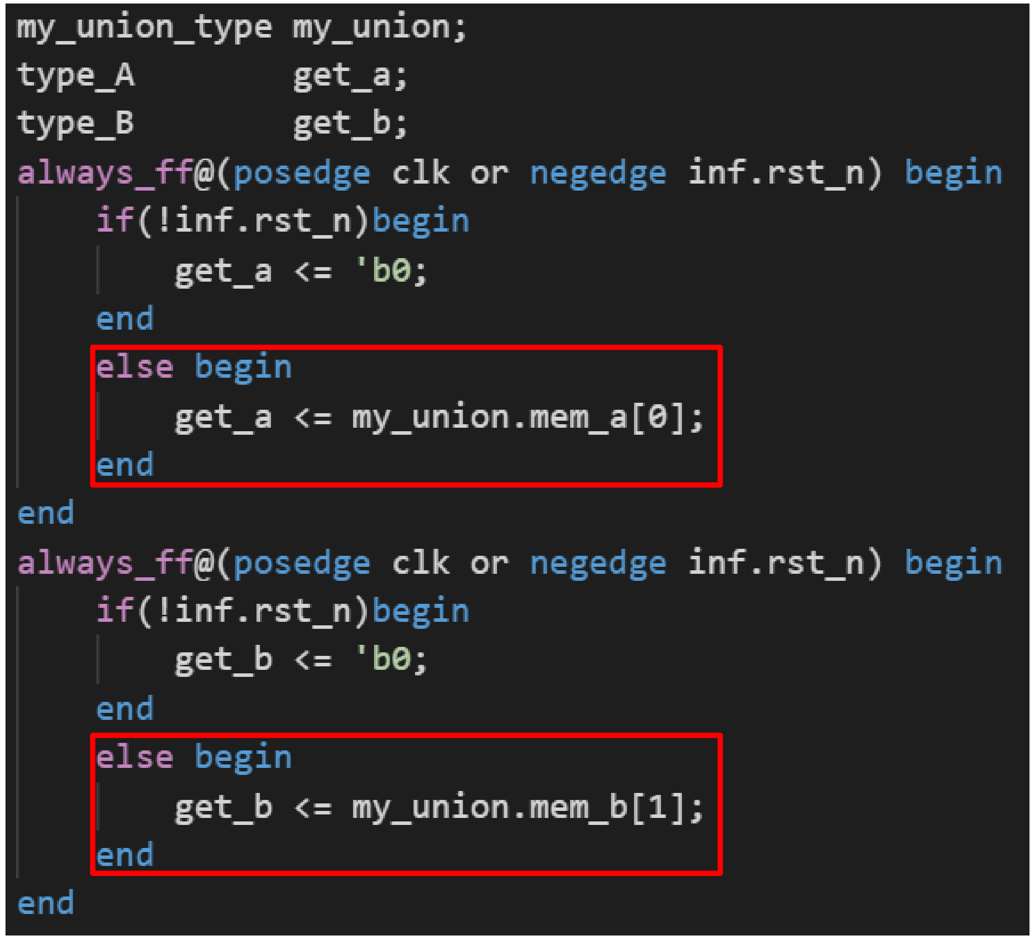
```
typedef enum logic [2:0]
{ a=3'd0, b=3'd1, c, d=3'd5 }   // ok, c=2
{ a=3'd0, b=3'd1, c, d=3'd2 }   // error!
{ a=3'd0, b=3'd1, c=3'd7, d }   // error!
```

Union (1/2)

- A SystemVerilog union allows **a single piece of storage** to be represented different ways using **different named member types**.

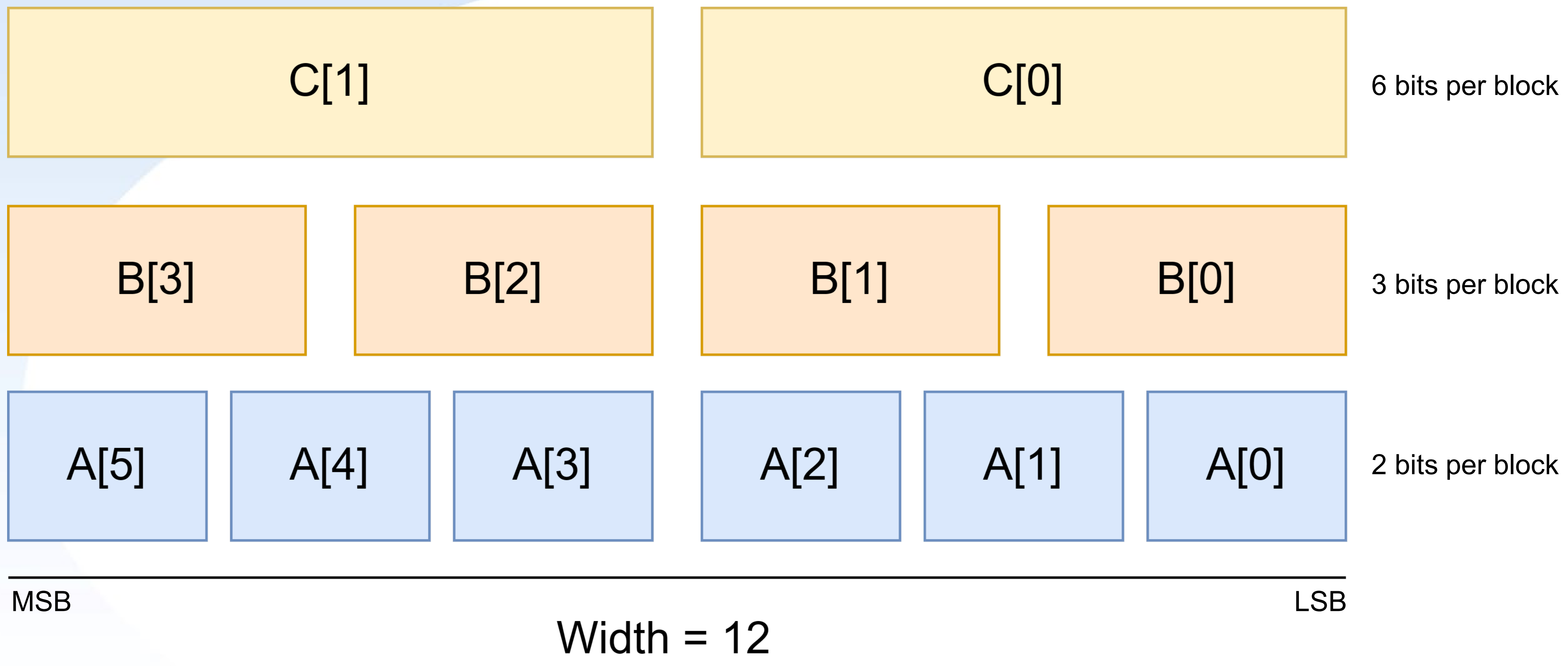


```
typedef logic [3:0] type_A;
typedef logic [7:0] type_B;
typedef type_B[1:0] type_C;

typedef struct packed {
    integer     temp_int;   // 32-bits
    type_C      c;          // 16-bits
    logic[15:0] d;          // 16-bits
} type_strct ;              // 64-bits


typedef union packed{
    type_strct    mem_struct;
    type_A        [15:0] mem_a;
    type_B        [7:0]  mem_b;
    type_C        [3:0]  mem_c;
}my_union_type;
```



```
my_union_type my_union;
type_A        get_a;
type_B        get_b;
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        get_a <= 'b0;
    end
    else begin
        get_a <= my_union.mem_a[0];
    end
end
always_ff@(posedge clk or negedge inf.rst_n) begin
    if(!inf.rst_n)begin
        get_b <= 'b0;
    end
    else begin
        get_b <= my_union.mem_b[1];
    end
end
```

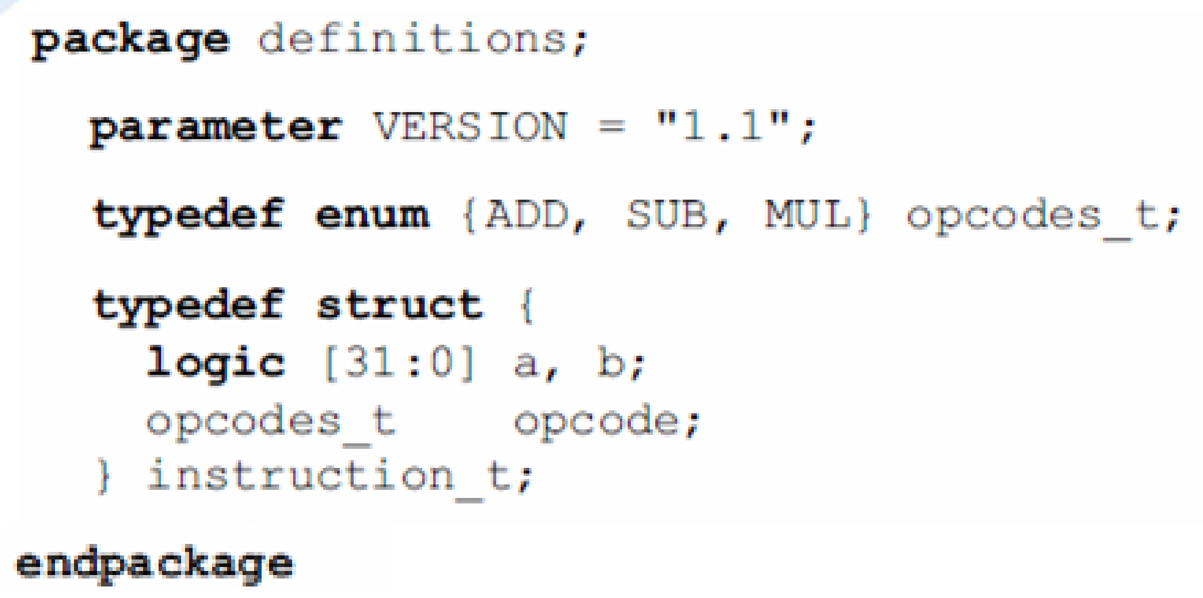
Union (2/2)



Assume this signal called “D”, we can use “D.A[0]” to access the data in D[1:0].

Package

- To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds ***packages*** to the Verilog language



```
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;

endpackage
```

Package in module

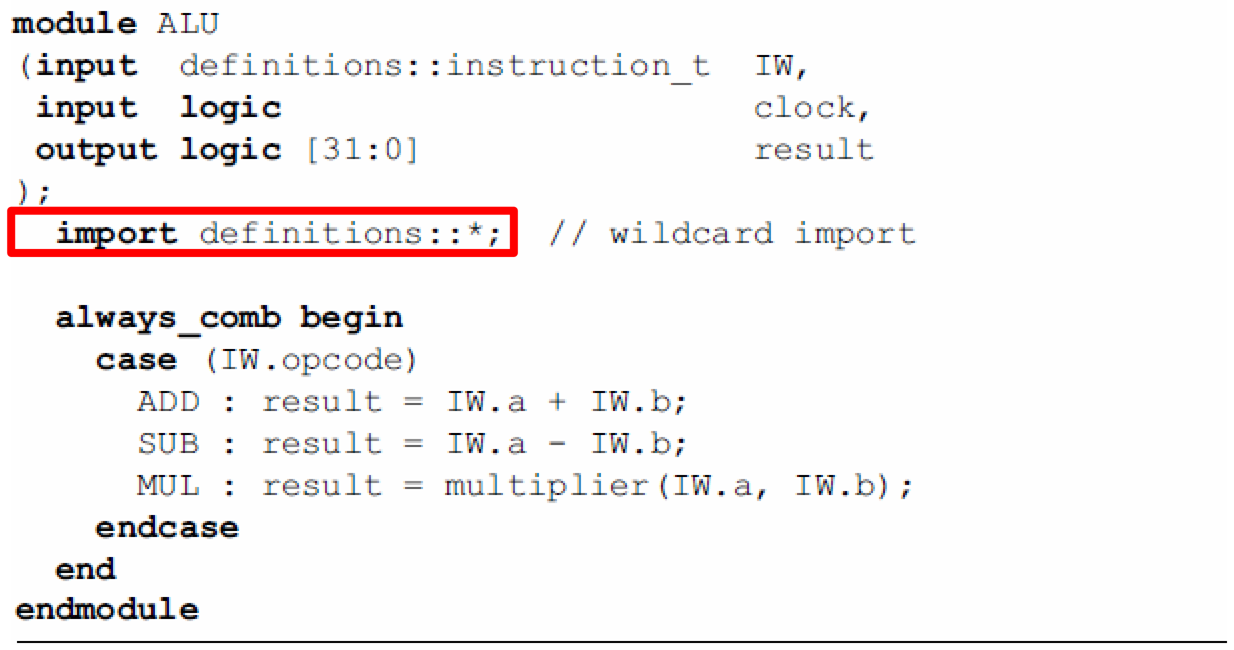
- When the case statement references the enumerated labels of ADD, SUB, and MUL, as well as the function multiplier, it will find the definitions of these names in the **definitions package**

```
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;
endpackage
```

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

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

Package in module

- When the case statement references the enumerated labels of ADD, SUB, and MUL, as well as the function multiplier, it will find the definitions of these names in the **definitions 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
```

Outline

Procedure Block in Verilog

- In Verilog, there are 2 kinds of procedure block
  - initial
  - always

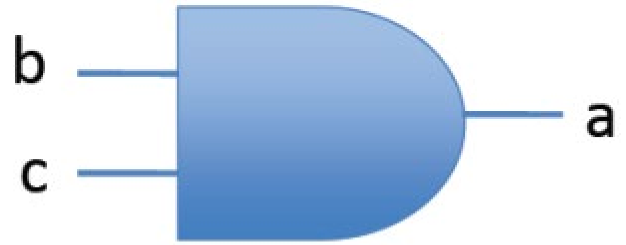
Procedure Block

- **In Verilog, always is a general purpose procedure block**
  - Can be use to model testbench, combination circuits, sequential circuits…etc
  - Depend on context which is not intuitive
  - Cannot be placed inside program or class and other procedure blocks
- **System Verilog adds three new logic specific processes to show designers intent:**
  - always_comb
  - always_ff
  - always_latch
- **In program, we may use forever@ to replace always blocks**

```
initial begin
    forever@(posedge clk)begin
        // do something
    end
end
```

always_comb

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



Verilog2001

```
always@( b or c)
    a=b&c;
```

SystemVerilog

```
always_comb
    a=b&c;
```

always_ff

- The always_ff procedural block is used to indicate that the intent to model synthesizable sequential logic behavior.
- Example:

  -
    ```
    always_ff @ (posedge clock or negedge rst_n)
     if (!rst_n) q <= 0;
     else          q <= d;
    ```

always_latch

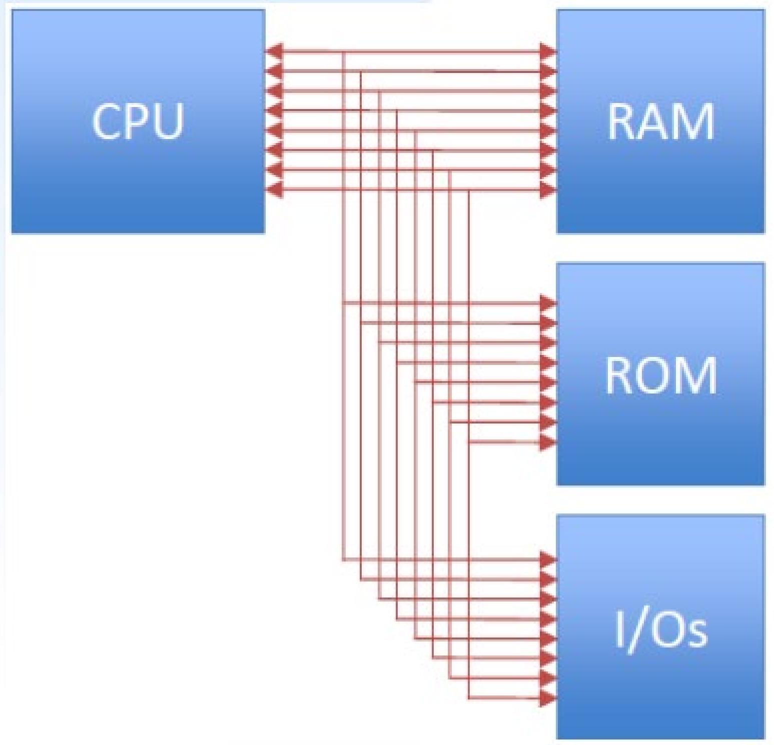
- The always_latch procedural block is used to indicate that the intent to model combinational logic with latch in it.
- Example:

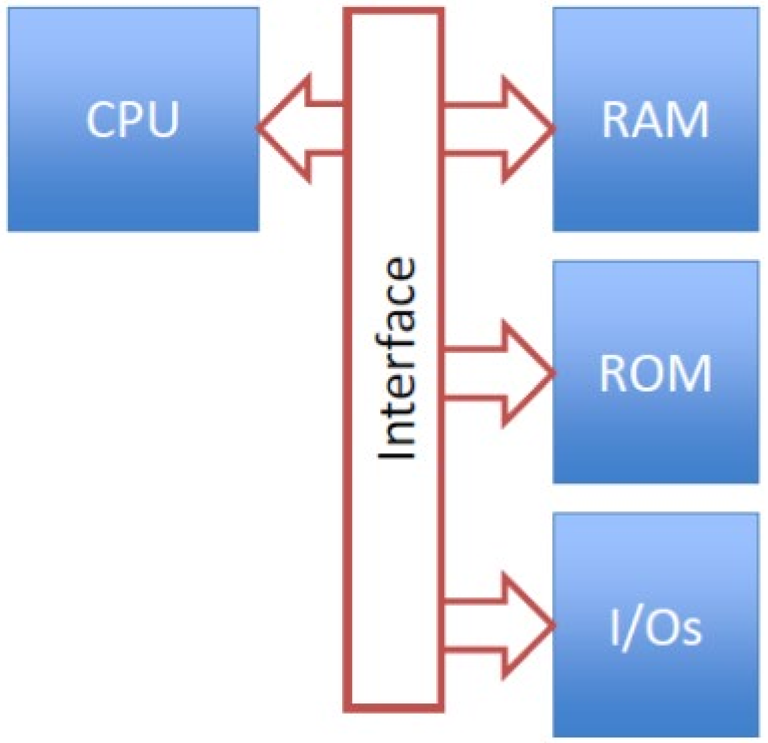
  -
    ```
    always_latch
     if (enable)
       a_latch = a;
    ```

Outline

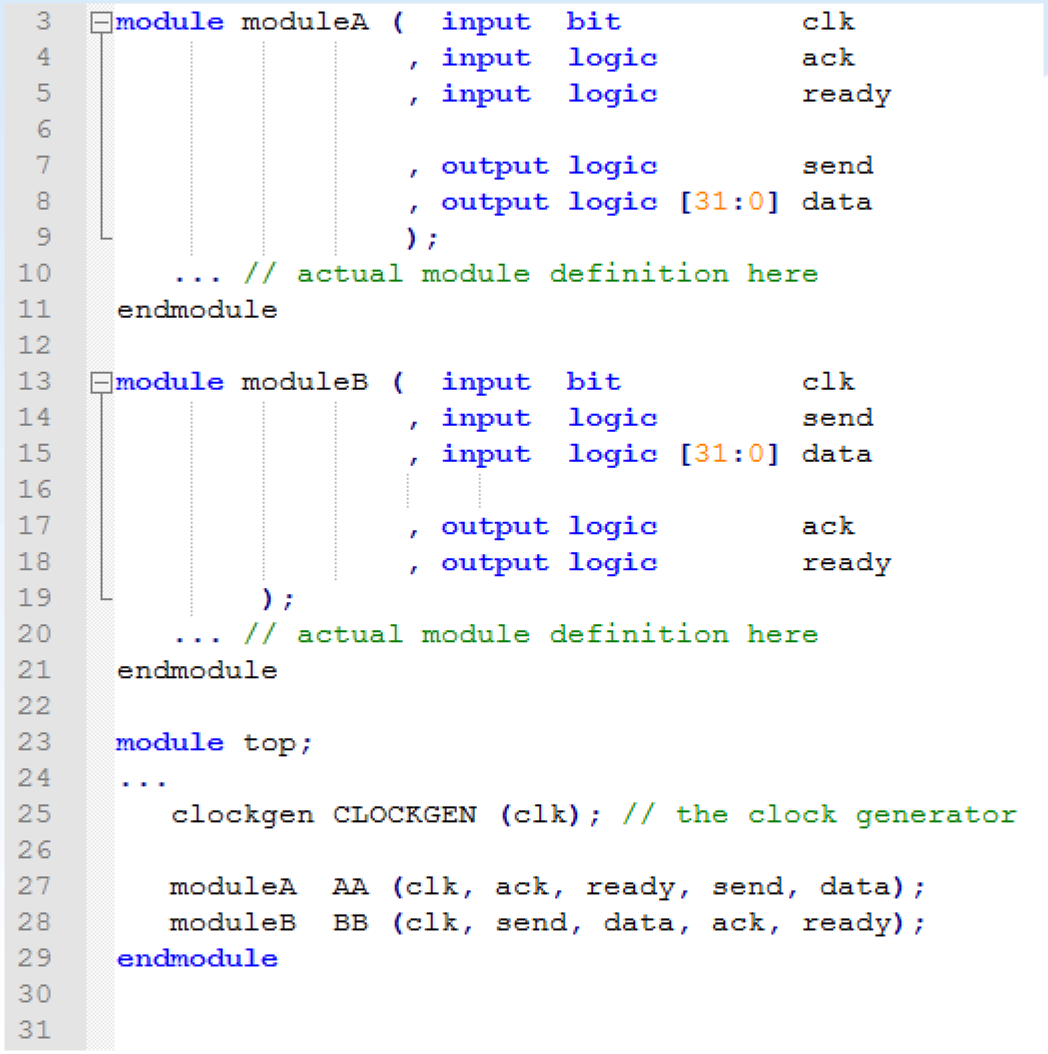
Interface

- ✓ **The interface encapsulate communication between design blocks, and between design and verification blocks.**
  - Replacing a group of names by a single one
  - An interface is a named bundle of wires, similar to a struct, except that an interface is allowed as a module port, while a struct is not.

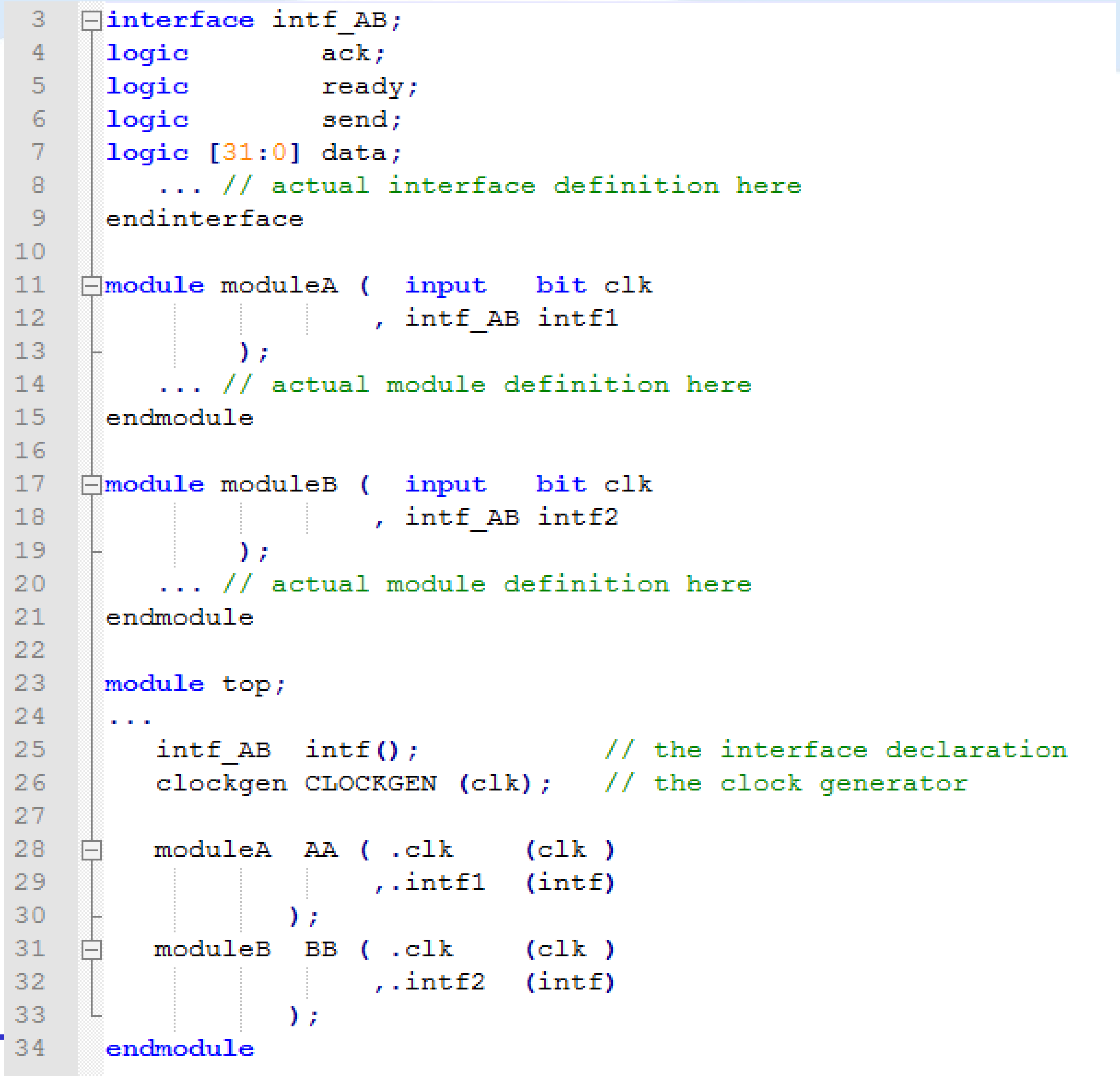




Example



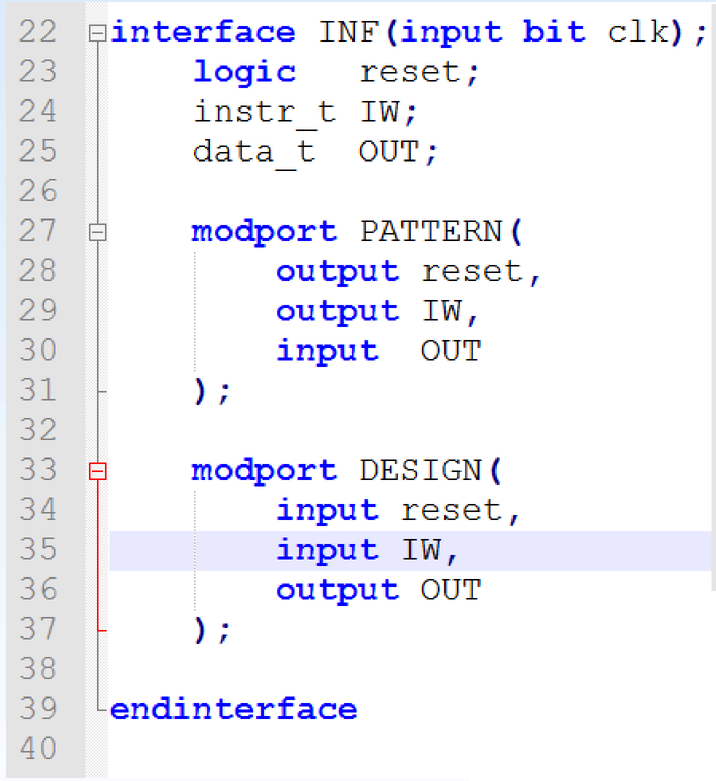
```
module moduleA (  input  bit          clk
                , input  logic        ack
                , input  logic        ready

                , output logic        send
                , output logic [31:0] data
                );
   ... // actual module definition here
endmodule

module moduleB (  input  bit          clk
                , input  logic        send
                , input  logic [31:0] data

                , output logic        ack
                , output logic        ready
        );
   ... // actual module definition here
endmodule

module top;
...
   clockgen CLOCKGEN (clk); // the clock generator

   moduleA  AA (clk, ack, ready, send, data);
   moduleB  BB (clk, send, data, ack, ready);
endmodule
```



```
interface intf_AB;
logic        ack;
logic        ready;
logic        send;
logic [31:0] data;
   ... // actual interface definition here
endinterface

module moduleA (  input   bit clk
                , intf_AB intf1
        );
   ... // actual module definition here
endmodule

module moduleB (  input   bit clk
                , intf_AB intf2
        );
   ... // actual module definition here
endmodule

module top;
...
   intf_AB  intf();              // the interface declaration
   clockgen CLOCKGEN (clk);      // the clock generator

   moduleA  AA ( .clk    (clk )
                ,.intf1  (intf)
           );
   moduleB  BB ( .clk    (clk )
                ,.intf2  (intf)
           );
endmodule
```

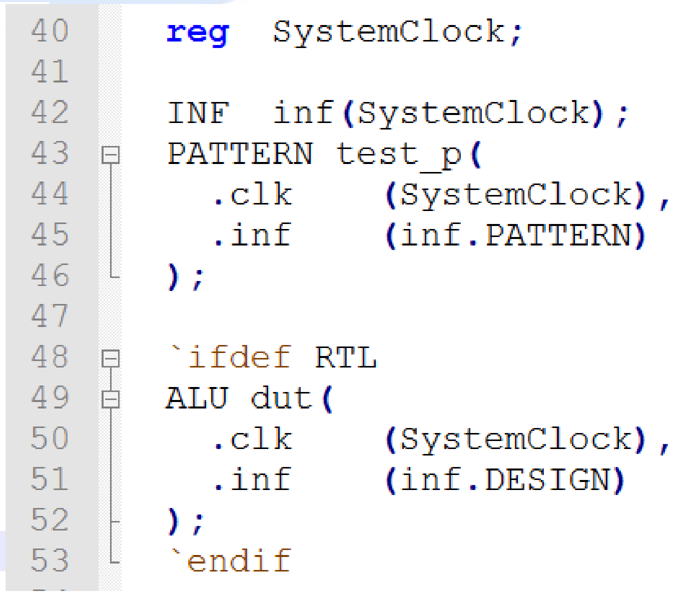
Modports of interface

✔ Modports are used to define direction of signal inside interface.



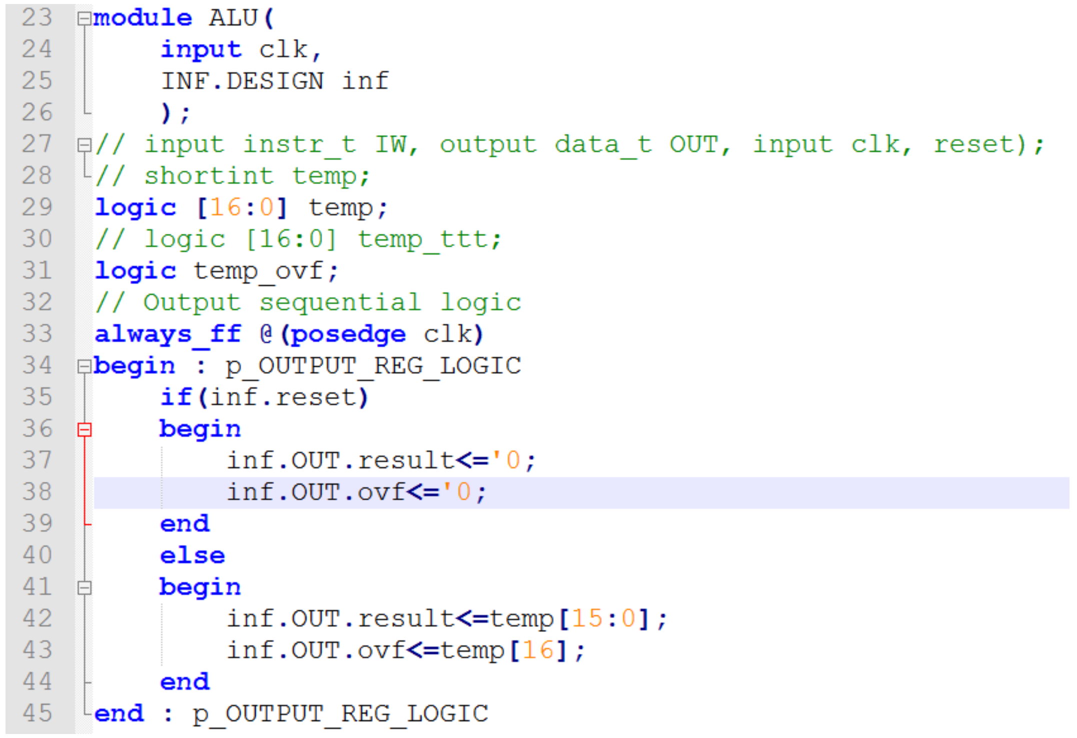
```
interface INF(input bit clk);
    logic   reset;
    instr_t IW;
    data_t  OUT;

    modport PATTERN(
        output reset,
        output IW,
        input  OUT
    );

    modport DESIGN(
        input reset,
        input IW,
        output OUT
    );

endinterface
```

INF.sv



```
reg   SystemClock;

INF   inf(SystemClock);
PATTERN test_p(
    .clk     (SystemClock),
    .inf     (inf.PATTERN)
);

`ifdef RTL
ALU dut(
    .clk     (SystemClock),
    .inf     (inf.DESIGN)
);
`endif
```

TESTBED.sv



```
module ALU(
        input clk,
        INF.DESIGN inf
        );
// input instr_t IW, output data_t OUT, input clk, reset);
// shortint temp;
logic [16:0] temp;
// logic [16:0] temp_ttt;
logic temp_ovf;
// Output sequential logic
always_ff @(posedge clk)
begin : p_OUTPUT_REG_LOGIC
        if(inf.reset)
        begin
            inf.OUT.result<='0;
            inf.OUT.ovf<='0;
        end
        else
        begin
            inf.OUT.result<=temp[15:0];
            inf.OUT.ovf<=temp[16];
        end
end : p_OUTPUT_REG_LOGIC
```

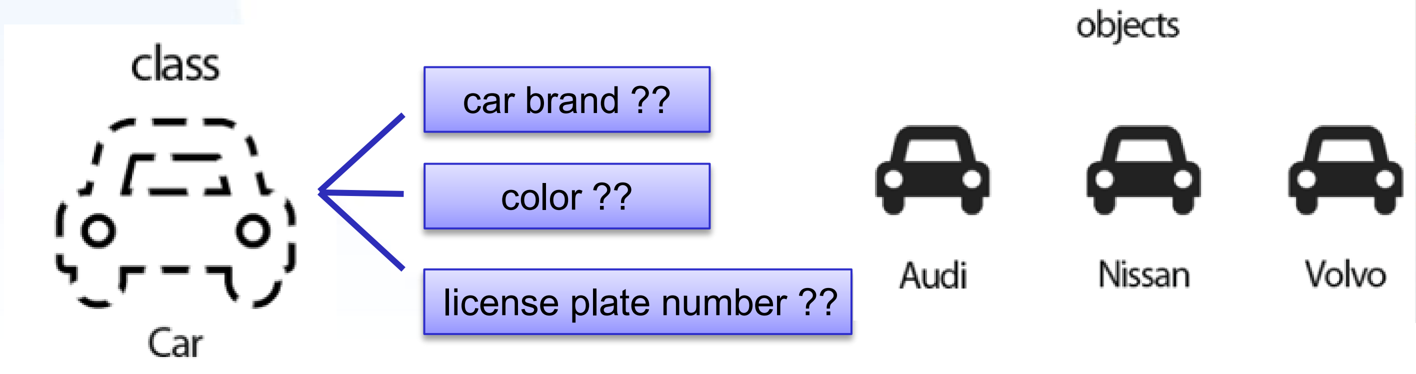
DESIGN.sv

Outline

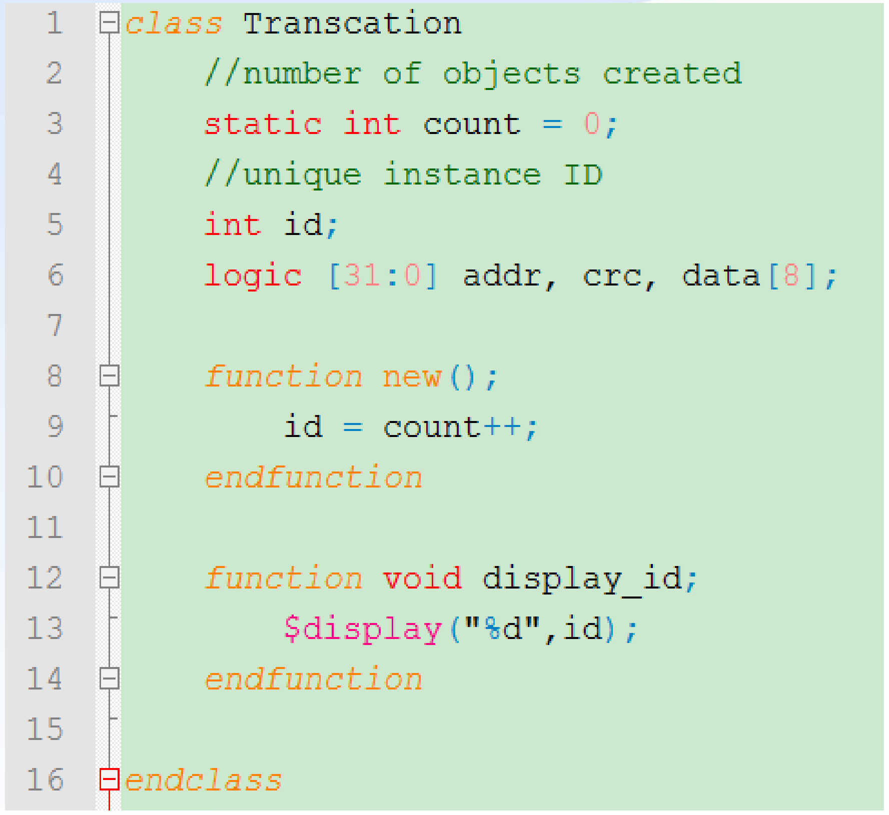
- ✓ Section 1 Introduction
- ✓ Section 2 Design using SystemVerilog
  - Data Type, enumerate, structure
  - Procedure Block, Design intent
  - Interface
- ✓ **Section 3 Verification using SystemVerilog**
  - OOP (Object-Oriented Programming)
  - Randomization

Object-oriented programming

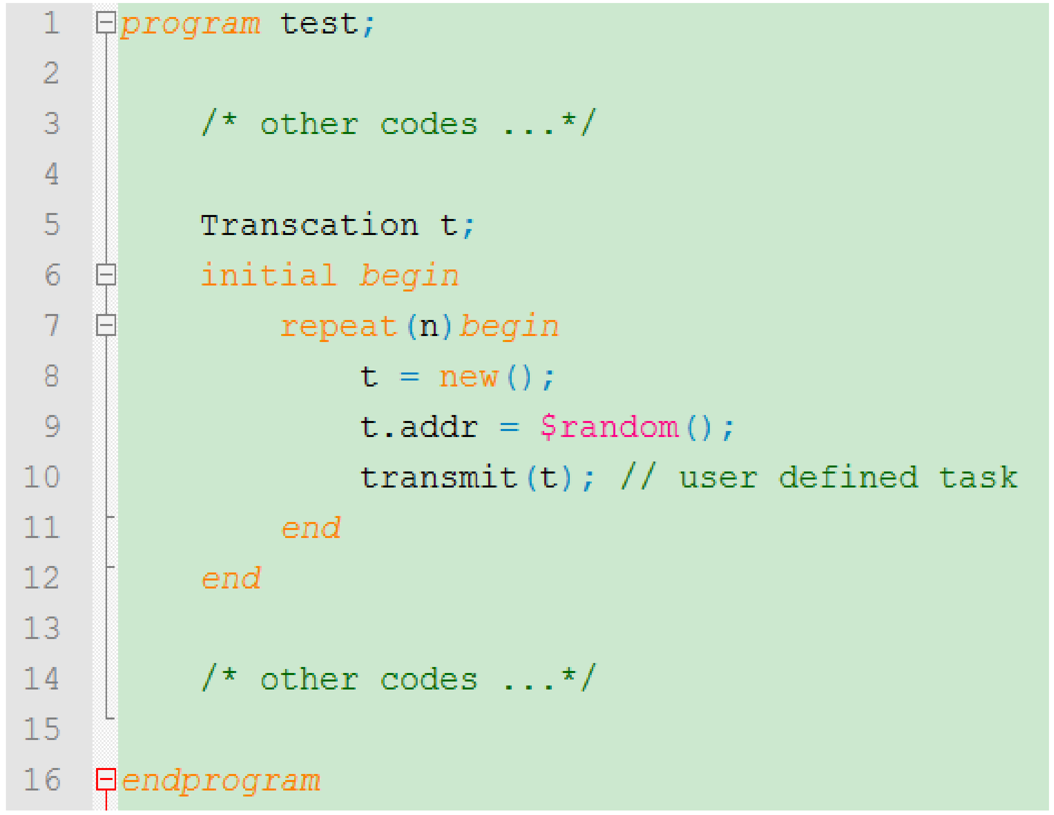
- **Objects, instances of classes**
- **Contents of “objects”**
  - Data field
  - Constructor
  - Methods
- **Each object should do what it should do, rather than several roles.**
- **Interaction between object and the object should be reduced as much as possible.**



Object-oriented programming



```
class Transcation
    //number of objects created
    static int count = 0;
    //unique instance ID
    int id;
    logic [31:0] addr, crc, data[8];

    function new();
        id = count++;
    endfunction

    function void display_id;
        $display("%d",id);
    endfunction

endclass
```

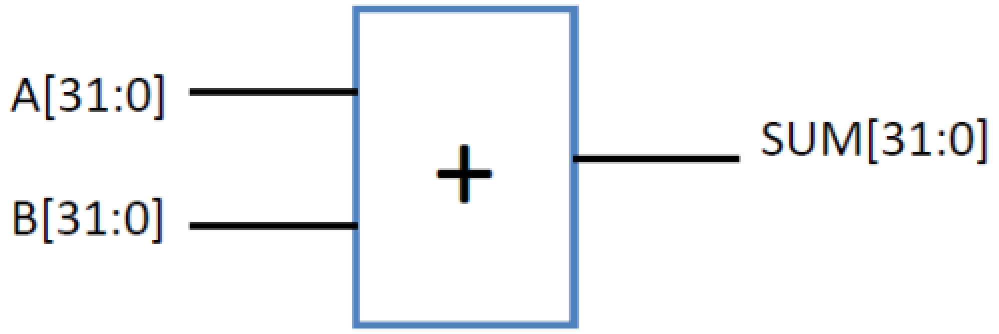


```
program test;

    /* other codes ...*/

    Transcation t;
    initial begin
        repeat(n)begin
            t = new();
            t.addr = $random();
            transmit(t); // user defined task
        end
    end

    /* other codes ...*/

endprogram
```

Outline

How To Do Verification?

- 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?

Alternatives to Exhaustive Verification?

- ✓ What is the goal of verification if exhaustive verification is unachievable?
- ✓ Verify with sufficient set of vectors to gain a level of confidence that product will ship with a tolerable field-failure rate.
- ✓ The best known mechanism is randomization:
  - Randomization of data
  - Use constraints to narrow the scope

Randomization

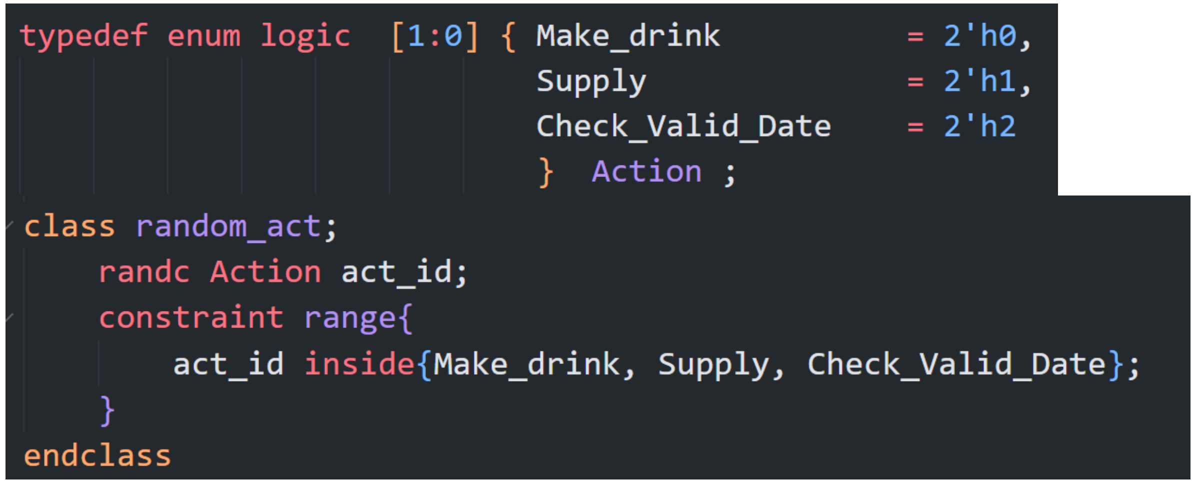
- Two types of random properties are supported:
  - rand
  - randc
- rand properties can assume any legal value:
  - Values can repeat without exhausting all possible values
- randc properties can be up to 16 bits:
  - Exhaust all values before repeating any individual value
  - For non-repeating bit values > 16 bits, use concatenation
- rand and randc properties are randomized when the class method randomize() is called:
  - randomize() is built-in with every class
  - 1 is returned if successful, 0 if randomization failed

Randomization Example

Ex1:

```
class random_interval;
        rand int interval ;
        function new ( int seed );
                this.srandom(seed) ;
        endfunction
        constraint limit
        {
                interval inside{1 , 2 , 3};
        }
endclass



random_interval interval_rand = new(1) ;
int i ;

initial
begin
        i = interval_rand.randomize() ;
        if(i == 0 ) $display("ERROR") ;
end
```

Ex2:



```
typedef enum logic  [1:0] { Make_drink          = 2'h0,
                            Supply              = 2'h1,
                            Check_Valid_Date    = 2'h2
                            }  Action ;
class random_act;
    randc Action act_id;
    constraint range{
        act_id inside{Make_drink, Supply, Check_Valid_Date};
    }
endclass
```

Constraints

- **Class properties are constrained in a** **constraint block**
  - Use operator or distribution constraints
  - Arrays can be constraint with functions like size()
- **For distribution specification:**
  - Distributed over a specified range with keyword ***inside***:

class member

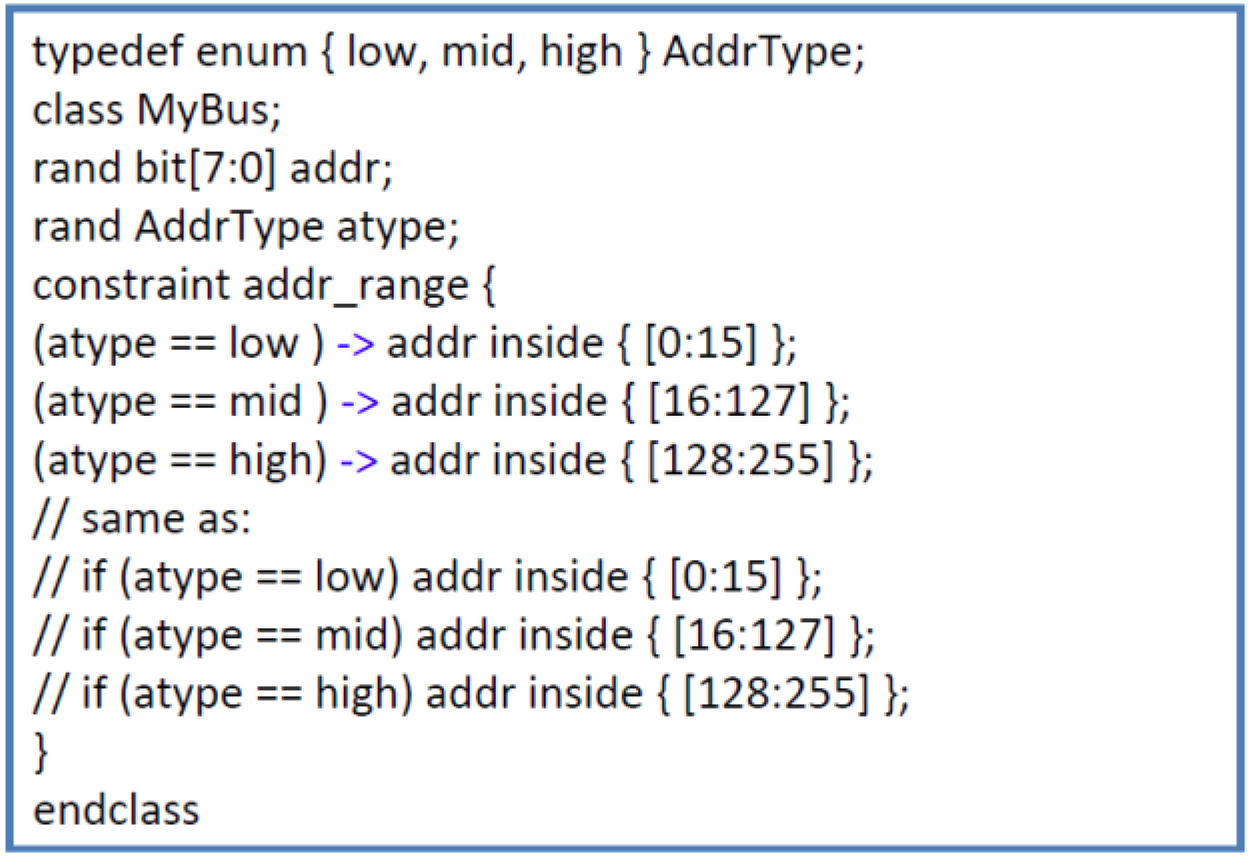
```
constraint Limit1 {
sa inside { [5:7], 10, 15 } ;
// 5,6,7,10,15 equally weighted probability

}
```

  - Excluded from a specified range with ***!*** operator:

```
constraint Limit2 {
!( sa inside { [1:10], 15 } );
// not 1 through 10 or 15

}
```

Appendix: Constraints

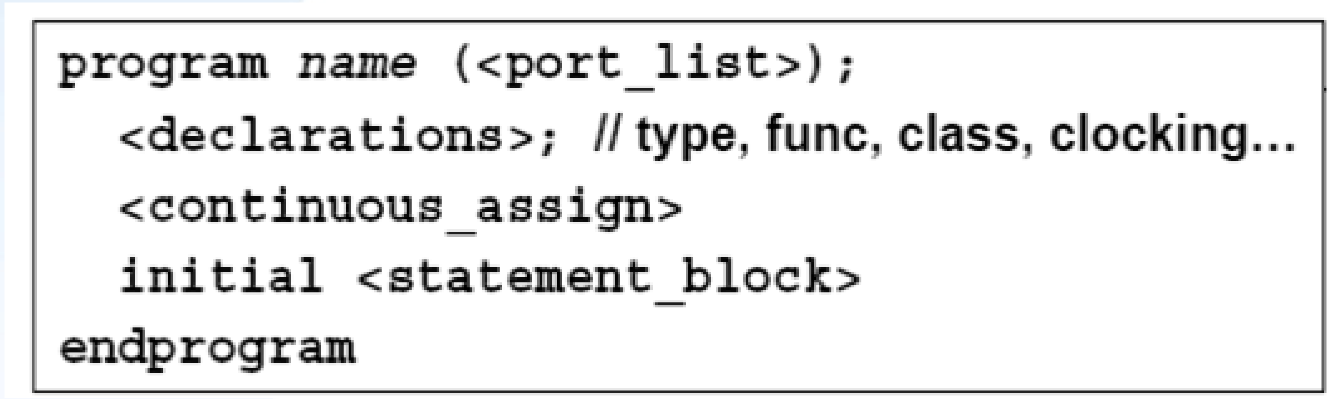
- ✓ **Operator Constraints**
- ✓ **Implication operators:**
  - ->
  - if(…)…[else]



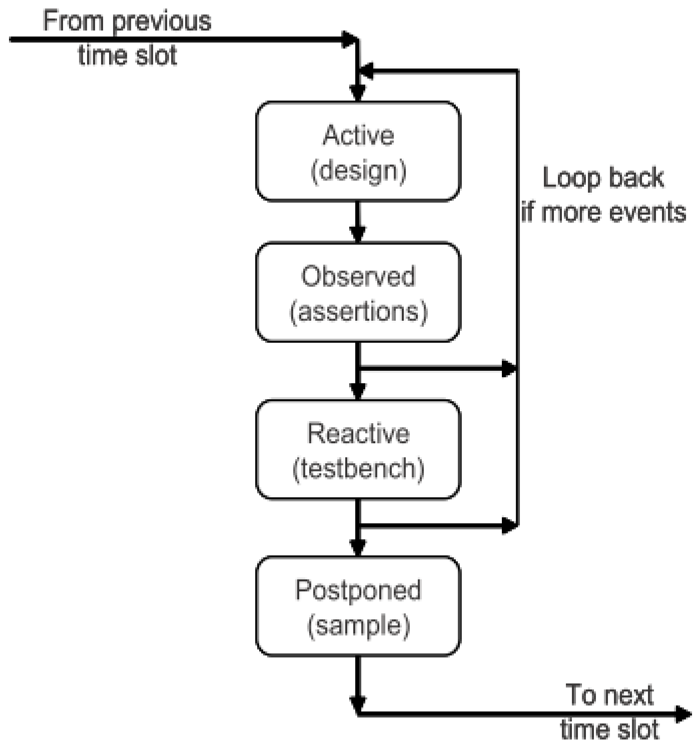
```
typedef enum { low, mid, high } AddrType;
class MyBus;
rand bit[7:0] addr;
rand AddrType atype;
constraint addr_range {
(atype == low ) -> addr inside { [0:15] };
(atype == mid ) -> addr inside { [16:127] };
(atype == high) -> addr inside { [128:255] };
// same as:
// if (atype == low) addr inside { [0:15] };
// if (atype == mid) addr inside { [16:127] };
// if (atype == high) addr inside { [128:255] };
}
endclass
```

Appendix: Program Block

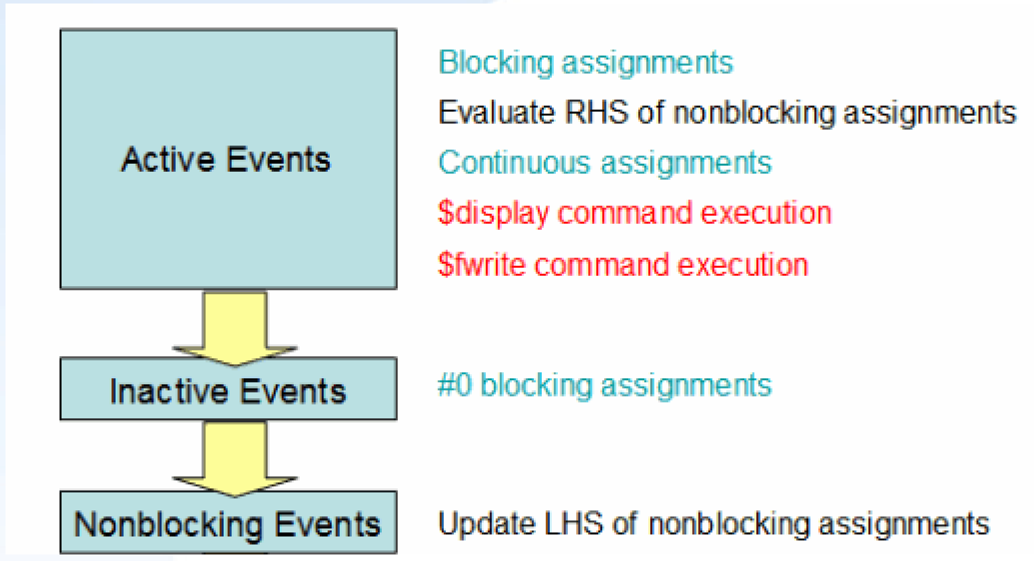
- **Purpose: Identifies verification code**
- **A program differs from a module**
  - Only initial blocks allowed
  - Execute in reactive region

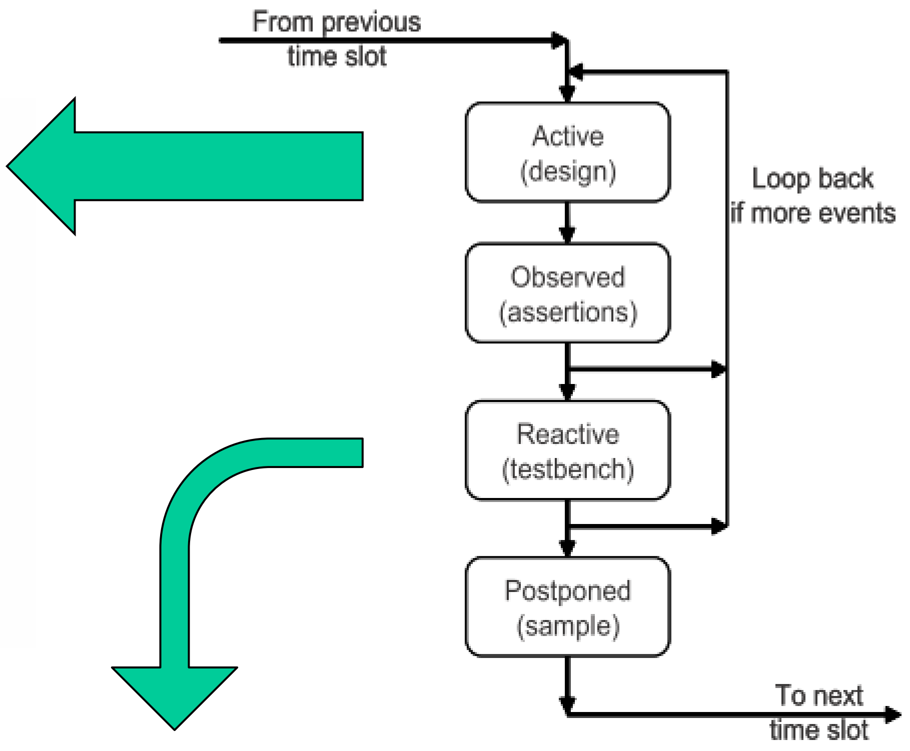


```
program name (<port_list>);
  <declarations>;  // type, func, class, clocking...
  <continuous_assign>
  initial <statement_block>
endprogram
```



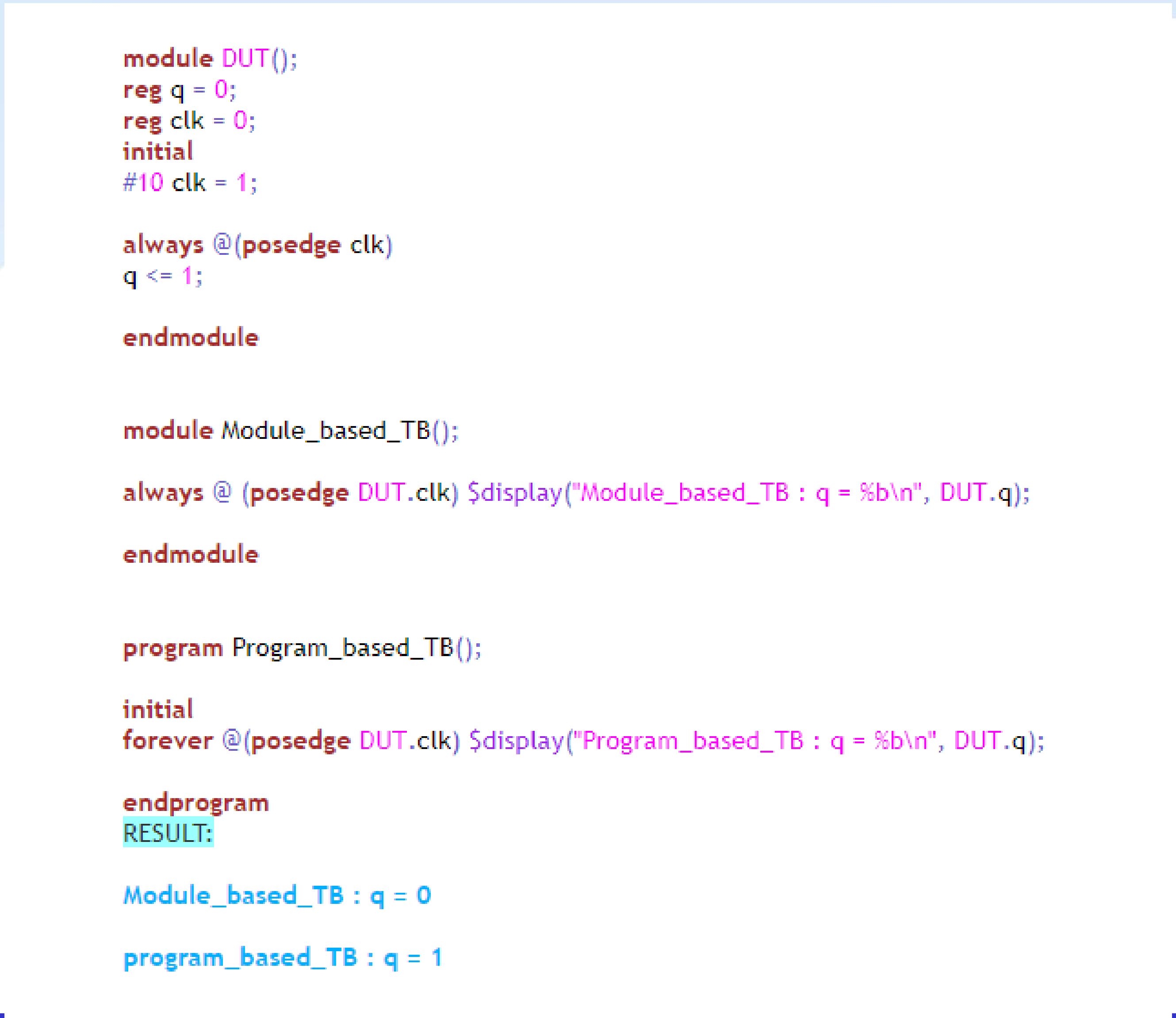
Appendix: Program Block


Active Events
Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
$fwrite command execution
Inactive Events
#0 blocking assignments
Nonblocking Events
Update LHS of nonblocking assignments



Program block will be executed In reactive region

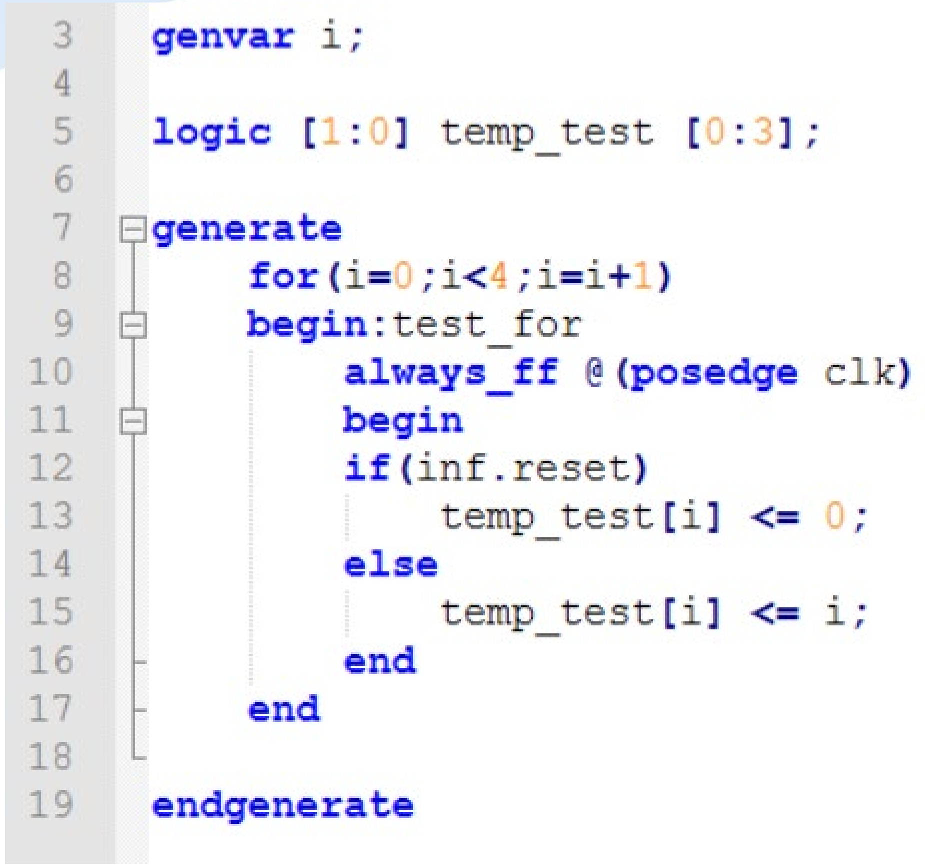
Appendix: Program Block



```
module DUT();
reg q = 0;
reg clk = 0;
initial
#10 clk = 1;

always @(posedge clk)
q <= 1;

endmodule


module Module_based_TB();

always @ (posedge DUT.clk) $display("Module_based_TB : q = %b\n", DUT.q);

endmodule


program Program_based_TB();

initial
forever @(posedge DUT.clk) $display("Program_based_TB : q = %b\n", DUT.q);

endprogram
RESULT:

Module_based_TB : q = 0

program_based_TB : q = 1
```

Appendix: Generate

- ✓ Useful for vector or array



```
genvar i;

logic [1:0] temp_test [0:3];

generate
    for(i=0;i<4;i=i+1)
    begin:test_for
        always_ff @(posedge clk)
        begin
        if(inf.reset)
            temp_test[i] <= 0;
        else
            temp_test[i] <= i;
        end
    end

endgenerate
```

Appendix : More about randomize()

- ✓ **When randomize() executes, three events occur:**
  - pre_randomize() is called
  - Randomization is executed
  - post_randomize() is called
- ✓ **pre_randomize()**
  - Optional
  - Set/Correct constraints before randomization
- ✓ **post_randomize()**
  - Optional
  - Make corrections after randomization

```
function void pre_randomize ();
  $display ("This will be called just before randomization");
endfunction

function void post_randomize ();
  $display ("This will be called just after randomization");
endfunction
```

Reference

- ✓ ASIC World (http://www.asic-world.com/systemverilog/index.html)
- ✓ electroSofts (http://electrosofts.com)
- ✓ AsicGuru.com (http://www.asicguru.com/home/6/)
- ✓ Verification Guide (http://www.verificationguide.com/p/systemverilog-tutorial.html)
- ✓ Project VeriPage (http://www.project-veripage.com/index.php)
- ✓ Language Reference Manual (http://www.ece.uah.edu/~gaede/cpe526/SystemVerilog_3.1a.pdf)