Module2

Chapter1and 2.
Verification Guidelines and Data Types

Text book: SV for Verification

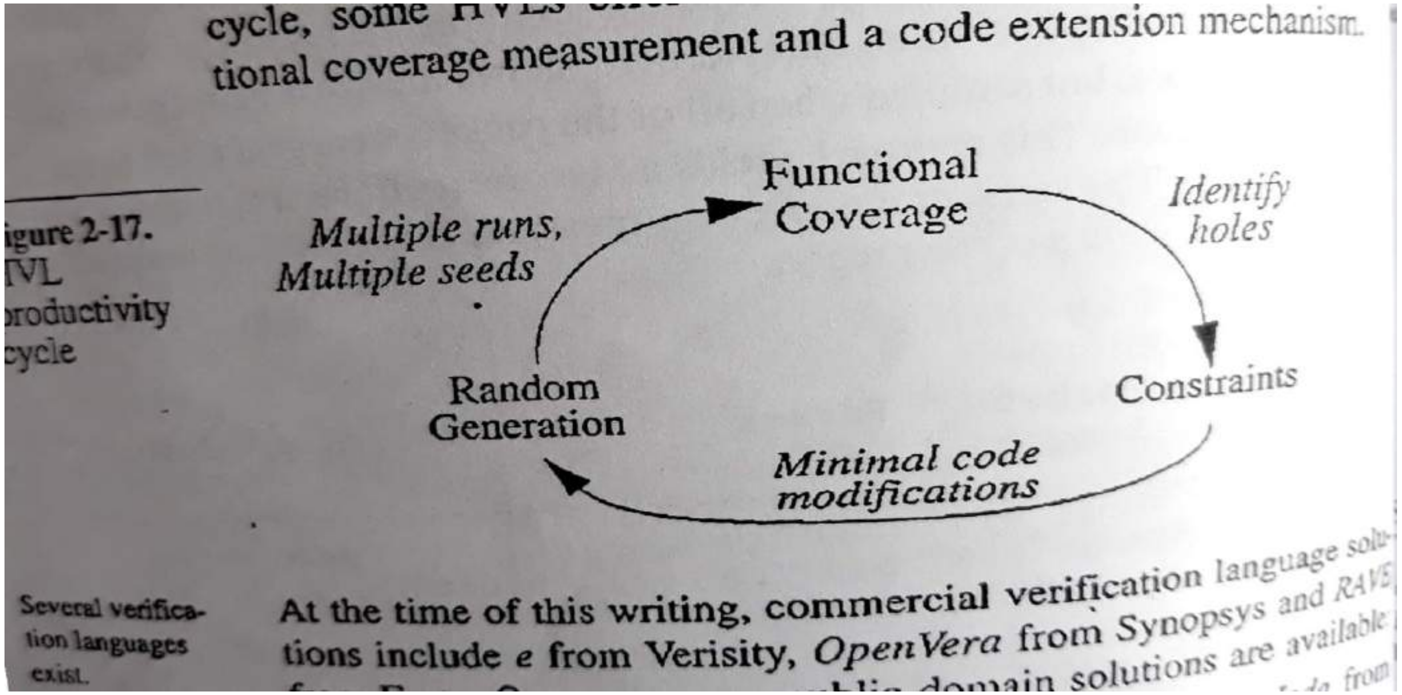
Chris Spear. Greg Tumbush

Verification languages

- VHDL and Verilog are not verification languages. Verilog was designed for describing low level hardware structures and VHDL was designed for very large system design. They can’t be used for verification.
- HVLs support higher level of abstraction and object orientedness.
- HVLs maintain important concepts necessary to interact with hardware – time, concurrency and instantiation.
- HVL can automate verification

HVL productivity Cycle- main benefit

- HVLs ability to generate random stimulus, collecting functional coverage to identify holes and ability to add constraints.


cycle, some HVLs
tional coverage measurement and a code extension mechanism.

igure 2-17.
VL
roductivity
ycle

Functional Coverage
Identify holes
Constraints
Minimal code modifications
Random Generation
Multiple runs, Multiple seeds

Several verifica-
tion languages
exist.

At the time of this writing, commercial verification language solu-
tions include *e* from Verisity, *OpenVera* from Synopsys and *RAVE*
domain solutions are available

Typical Features of Verification Languages

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures:- object oriented programming and transaction level modeling
- Multithreading and inter process communication
- Support for HDL types such as Verilog’s 4-state values
- Tight integration with event-simulator for control of the design

HVLs

- Commercial Verification languages: e from Vericity, openvera from synopsis, RAVE from Forte
- Open-source verification languages: System C from Cadence, Jeda from Juniper Networks, System Verilog from Accellera

SystemVerilog's roots

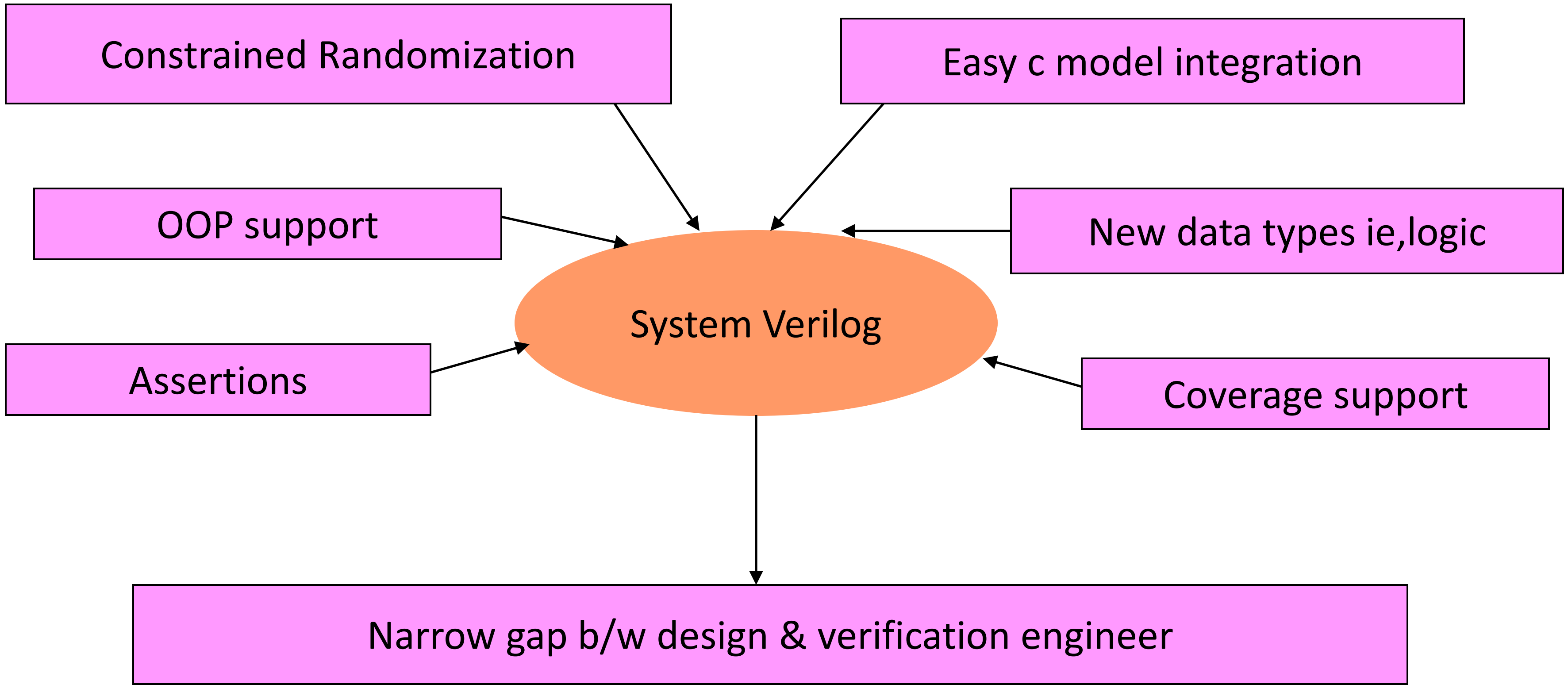
- Instead of re-inventing the wheel, Accellera relied on donations of technology from a number of companies.
  - **High-level modeling constructs** : Superlog language developed by Co-Design
  - **Testbench constructs** : Open Vera language and VCS DirectC interface technology by Synopsys
  - **Assertions** : OVA from Verplex, ForSpec from Intel, Sugar (renamed PSL) from IBM, and OVA from Synopsys

What is SystemVerilog?

- ❑ System verilog is the superset of verilog
- ❑ It supports all features of verilog plus add on features
- ❑ It’s a super verilog

- ❑SystemVerilog is a hardware description and Verification language(HDVL)
- ❑ SystemVerilog has an extensive set of enhancements to IEEE 1364 Verilog-2001 standards
- ❑ It has features inherited from Verilog HDL,VHDL,C,C++
- ❑ In 2005, System Verilog became IEEE standard.

Why SystemVerilog?



Verification Process

- How you plan to verify your particular design ?
- How this influences the test bench structure. ?
- Ex-Just as all houses have kitchens, bedrooms, and bathrooms,
  - ✓ all test benches share some common structure of stimulus generation and response checking.

Important principle: as a verification engineer is:

- ✓ "Bugs are good."
- ✓ Don't shy away from finding the next bug,
- ✓ Keep track of the details of each bug found.
- ✓ Each bug found before tape-out is one fewer that ends up in the customer's hands.
- ✓ At each stage in the design, the cost of fixing a bug goes up by a factor of 10, so find those bugs early and often.

The Verification Process

- What is the goal of verification? "Finding bugs," (partly correct).
- Bugs are the discrepancy between specification and the actual design.

- ➢ Hardware designer creates a device that performs a particular task
- ✓ Reads the hardware specification for a block,
- ✓ Interprets it through the human language description,
- ✓ Creates the corresponding logic in a machine-readable form, usually RTL code.
- ➢ Ambiguities in the original document, missing details, or conflicting descriptions.

- ➢ Verification engineer is to make sure the device can accomplish that task successfully.
- ✓ must also read the hardware specification, create the verification plan, and follow it to build tests showing the RTL code correctly implements the features.
- ➢ The design is an accurate representation of the specification.

The Verification Process

Reasons for a design to fail during verification :

- ✓ Incorrect/Insufficient Specifications
- ✓ Misinterpretation and Misunderstandings
- ✓ Incorrect interaction between IPs/Cores
- ✓ **Unexpected behavior of the system**

Importance of Verification

- ✓ Bug escapes to silicon can be costly including re-spin
- ✓ 70 % of design cycles is spent in verifying design
- ✓ Ever increasing complexity of designs makes this harder
- ✓ Hence Verification is always on critical path for any product design

Testing at Different Levels

Each level of Verification will be suited for a specific objective:

Lower levels-

- ✓ need to create tests that generate stimuli from all the surrounding blocks
- ✓ have better controllability and visibility
- ✓ Faster

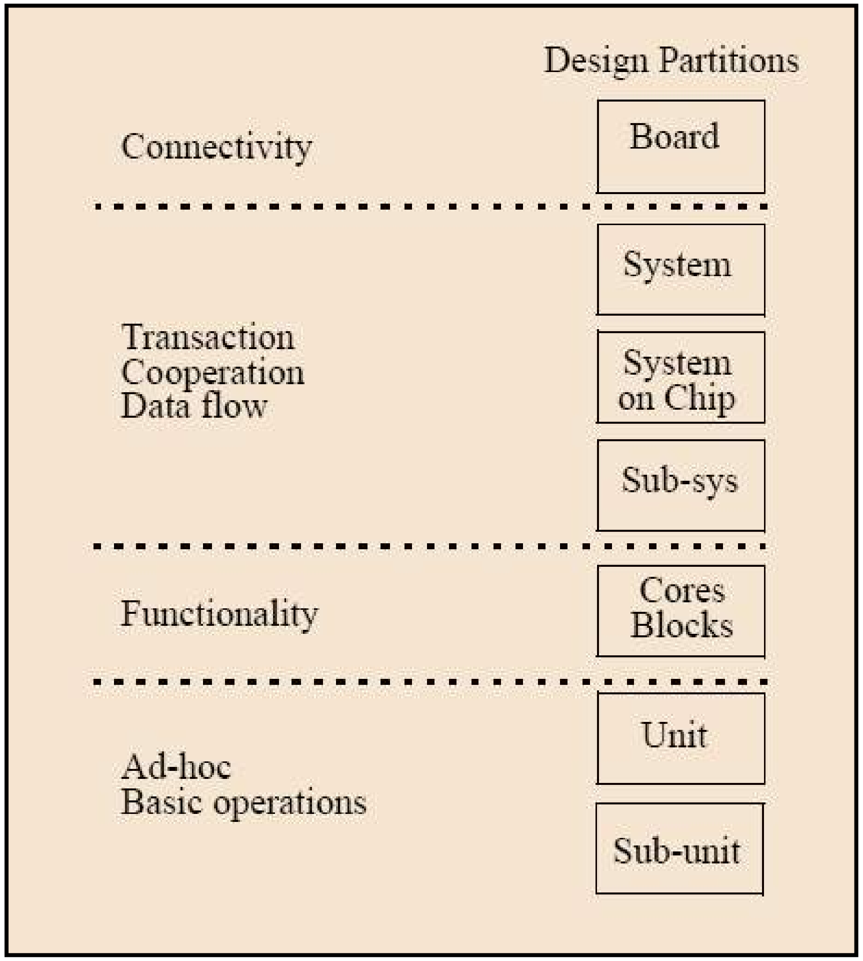
Block level-

- ✓ Verification helps designs to be verified independently and in parallel

System Level-

Verification focuses more on interactions and integration phase.

- ✓ the entire system is tested,
- ✓ but the simulation performance is greatly reduced.



The Verification Plan and Approaches

- ➢ The verification plan is derived from the hardware specification documents
- ✓ Contains a description of what features need to be exercised under what conditions.
- ✓ Techniques to be used:

  Directed or random testing, assertions, HW/SW co-verification,

  Emulation, formal proofs, and use of verification IP.

- ➢ White Box
  - ✓ Intimate knowledge of design implementation
  - ✓ Full visibility and observability

Verification Methodology Manual

- VMM developed by Janick Bergeron and others at Qualis Design.
- VMM techniques were originally developed for use with the OpenVera language and were extended in 2005 for System Verilog.
- OVM and UVM are the newer methodologies used for verifying 100 million gate design with many communication protocols, complex error handling and library of IPs.

Basic Test bench Functionality

The purpose of a test bench is to determine the correctness of the DUT.

This is accomplished by the following steps.

- ✓ Generate stimulus
- ✓ Apply stimulus to the DUT
- ✓ Capture the response
- ✓ Check for correctness
- ✓ Measure progress against the overall verification goals

- ➢ Some steps are accomplished automatically by the test bench, while others are manually determined by you.

Directed Testing

In directed tests approach,

- ✓ look at the hardware specification,
- ✓ write a verification plan with a list of tests, (each of which concentrated on a set of related features).

  ex: UART features – Clear -to –send(CTS) pin must be asserted when UART can accept a new word to transmit via the CPU interface.

  Data Terminal Ready(DTR) pin must be asserted when there is a received word ready to be read by the CPU interface

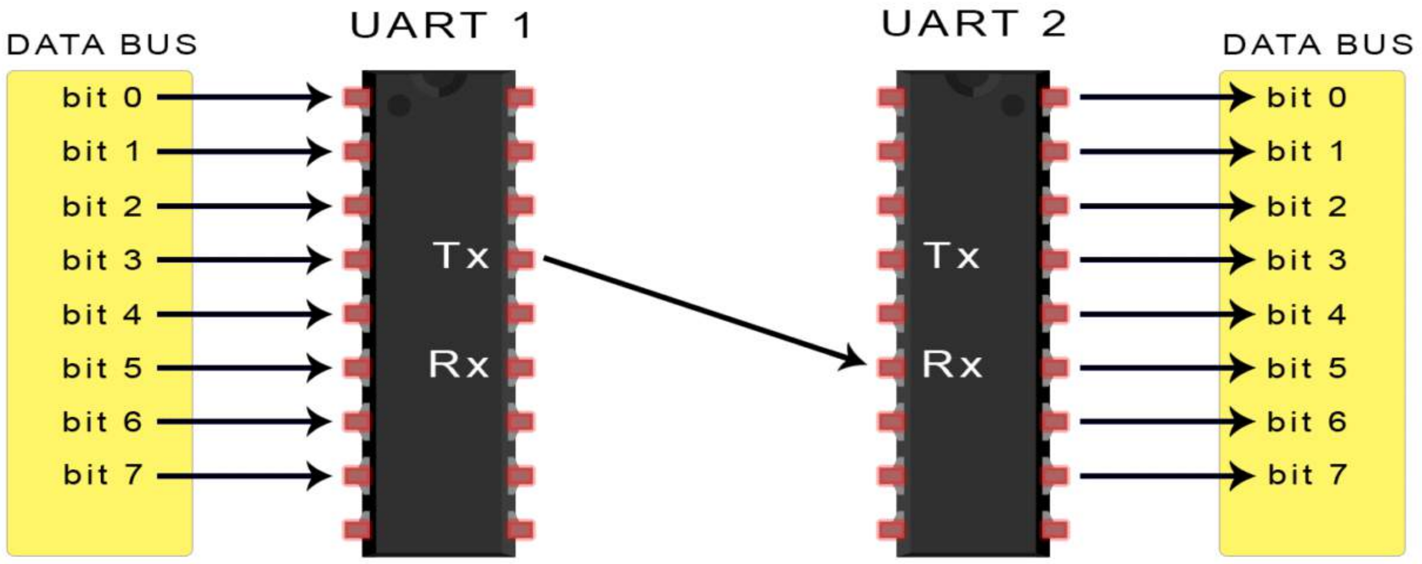
- ✓ With this plan, you write stimulus vectors that exercise these features in the DUT.
- ✓ Simulate the DUT with these vectors
- ✓ Manually review the resulting log files and waveforms to make sure the design does what you expect.

- ➢ Once the test works correctly, you check it off in the verification plan and move to the next one.
- ➢ Given ample time and staff, directed testing is sufficient to verify many designs.

UART

- UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols
- Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.
- UART, the mode of transmission is in the form of a packet. The piece that connects the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines. A packet consists of a start bit, data frame, a parity bit, and stop bits.

UART


DATA BUS
bit 0
bit 1
bit 2
bit 3
bit 4
bit 5
bit 6
bit 7
UART 1
Tx
Rx
UART 2
Tx
Rx
DATA BUS
bit 0
bit 1
bit 2
bit 3
bit 4
bit 5
bit 6
bit 7

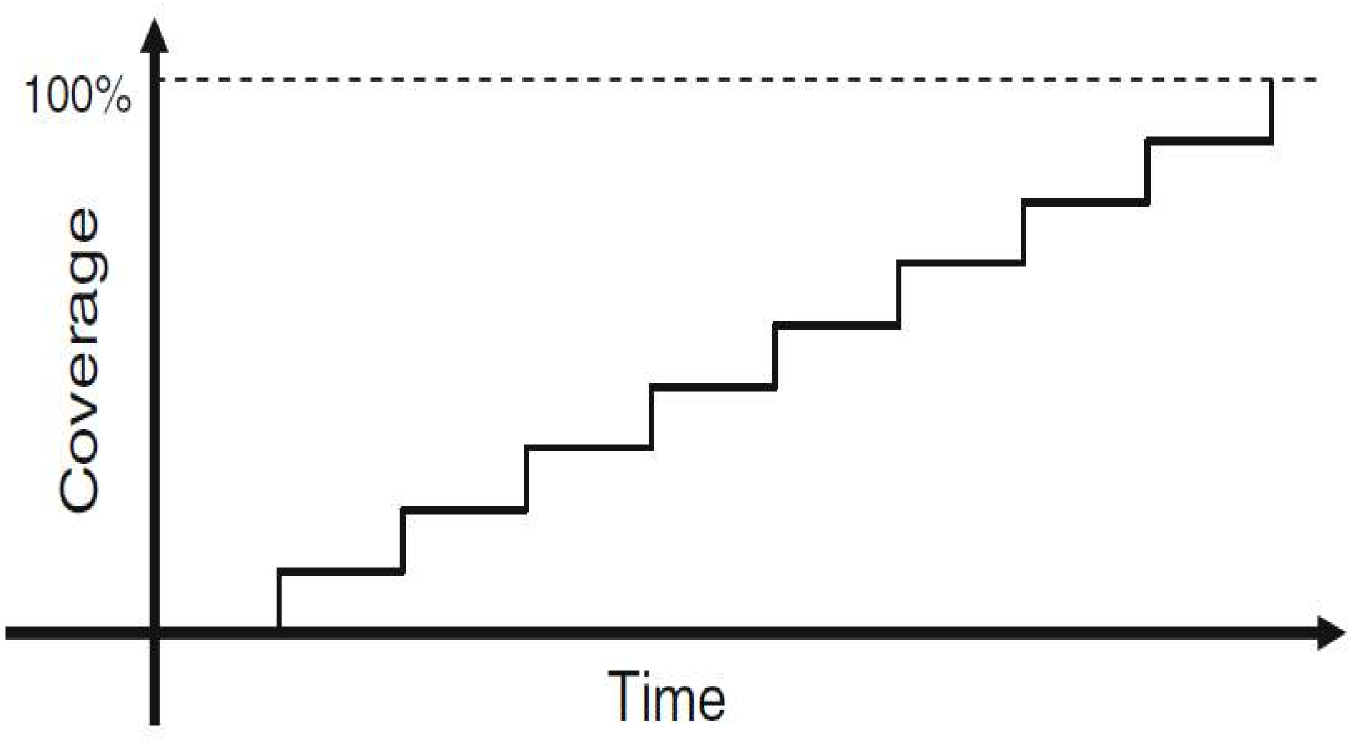
Directed Testing

- It is an incremental approach with steady progress as in Fig1.
- We should have the necessary time or resources to carry out the directed testing approach.
- Forward progress comes, but the slope remains the same.
- If the design complexity doubles, it takes twice as long to complete or requires twice as many people to implement it.

- Example:

  To verify every combination of inputs for a 32-bit adder, simulations would still be running years after the project should have shipped.

Fig1. Directed test progress over time



- Fig2 shows the total design space and features that are covered by directed test cases. You need to write tests that cover all the features and find bugs.

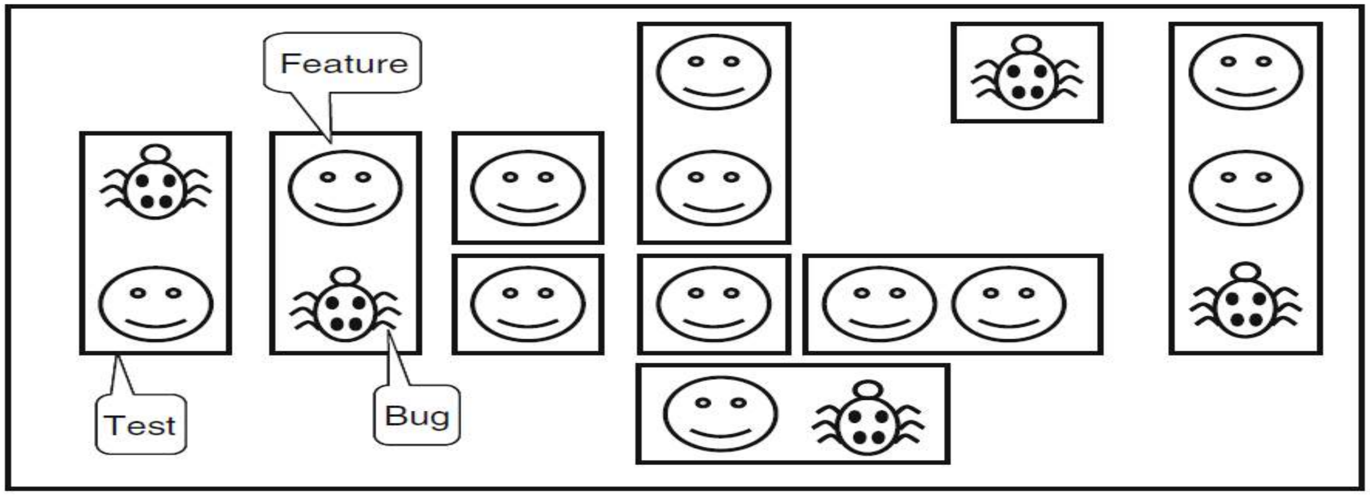
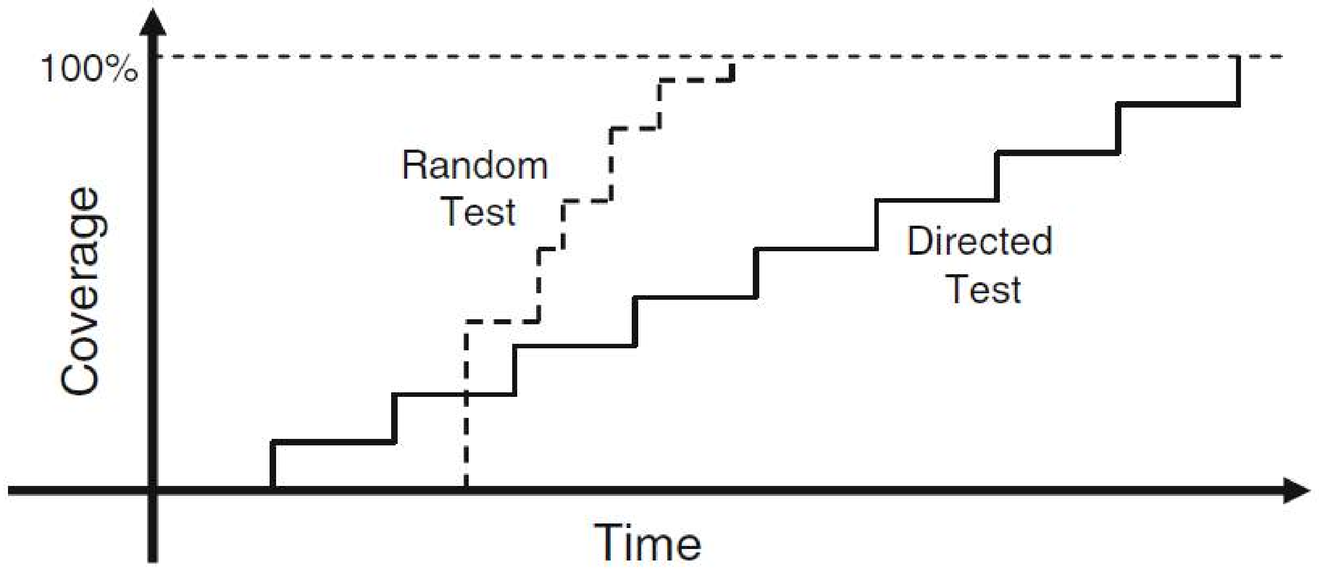


Fig. Directed test coverage

Random Testing

- Random stimulus is crucial for exercising complex designs.
- A directed test finds the bugs you expect to be in the design, whereas a random test can find bugs you never anticipated.
- Random stimuli needs functional coverage to measure verification progress.
- Furthermore, need an automated way to predict the results — generally a scoreboard or reference model.
- A layered testbench helps you control the complexity by breaking the problem into manageable pieces.
- Transactors provide a useful pattern for building these pieces.
- With appropriate planning, a common test bench infrastructure can be shared by all tests and does not have to be continually modified.



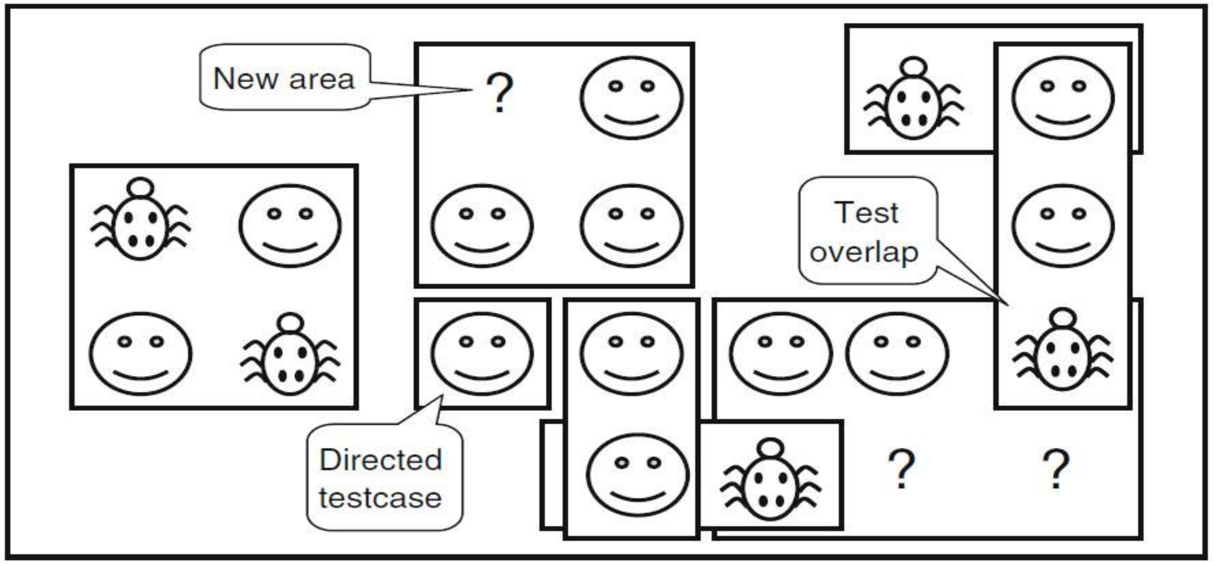
Constrained-random test progress over time vs. directed testing

- Every random test shares common test bench, as opposed to directed tests where each is written from scratch.
- Constrained-random test bench is now finding bugs faster than the many directed ones.
- The last few bugs may only be found with directed tests, but the vast majority of bugs will be found with random tests.

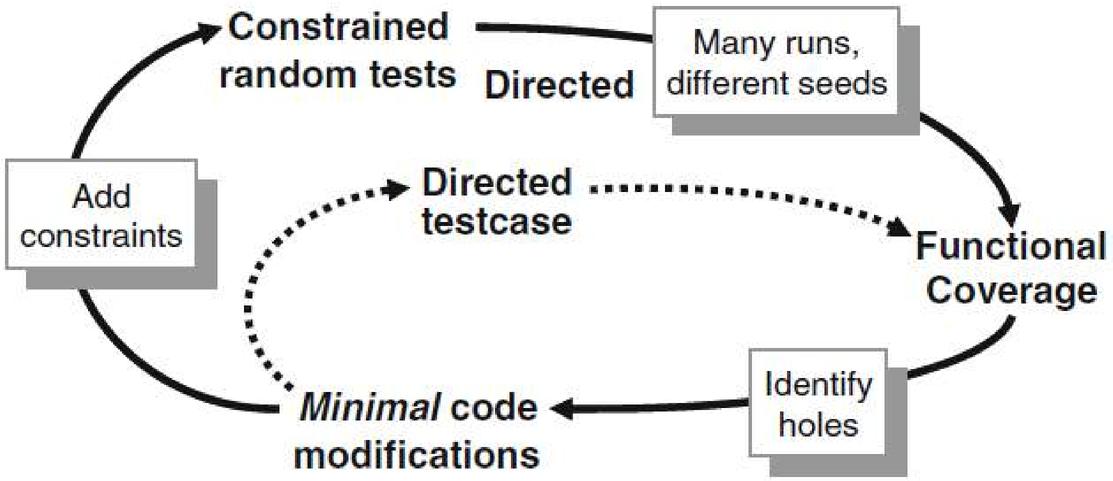
A random test bench can always be turned into directed tests.

Directed Test bench can never be turned into a true random test bench.

Constrained-Random Stimulus



- Productivity Cycle



.5 Coverage convergence

Constrained Random Test

- Figure shows the coverage over the total design space.
- Random test covers a wider space than a directed test. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate. If these new areas find a bug, you are in luck!

What should you Randomize?

- Think broadly about all design inputs, when you think of randomizing the stimulus to a design:- device configuration, environment configuration, input data, protocol exceptions, errors and violations and delays.

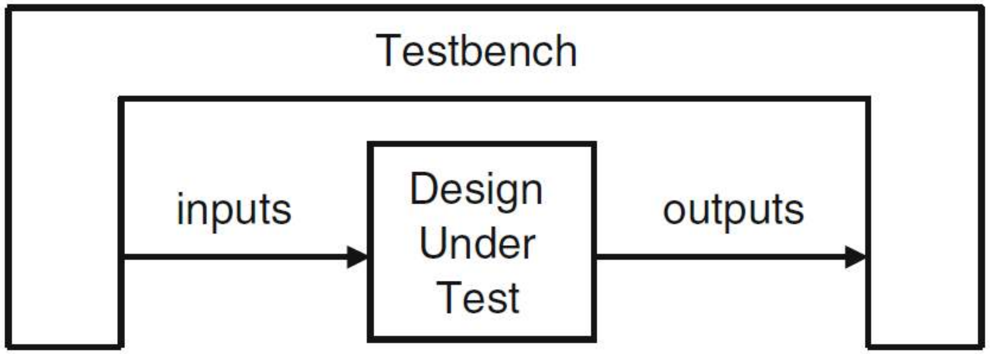
Coverage

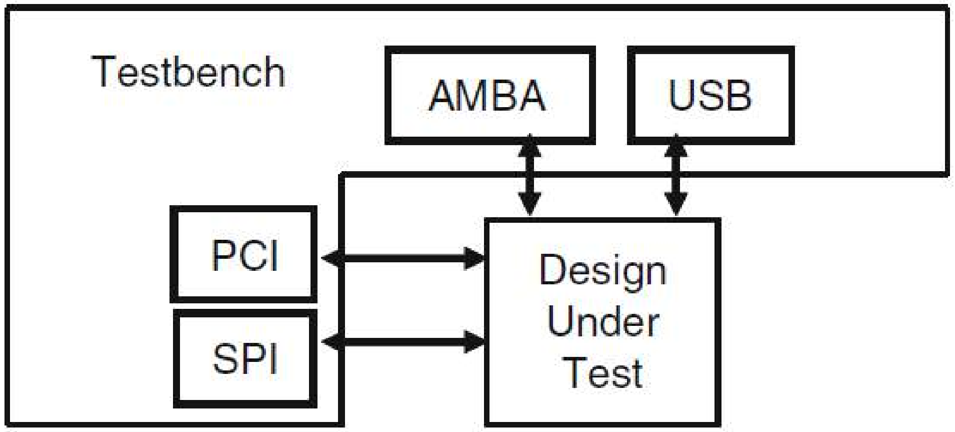
- What is Coverage?
  - Coverage is the metric of completeness of verification.
- Why coverage?
  - Verification is based on samples
    - Need to know that all areas of the design have been verified
- Functional Coverage and Code Coverage
- Functional Coverage -Covers the functionality of the DUT- ADD, MUL,ROL, MOV,etc
- Code Coverage- Covers all the statements, branch operation, conditional statements, toggling of inputs, State Coverage.

Test bench Components

In simulation, the test bench wraps around the DUT, just as a hardware tester connects to a physical chip.

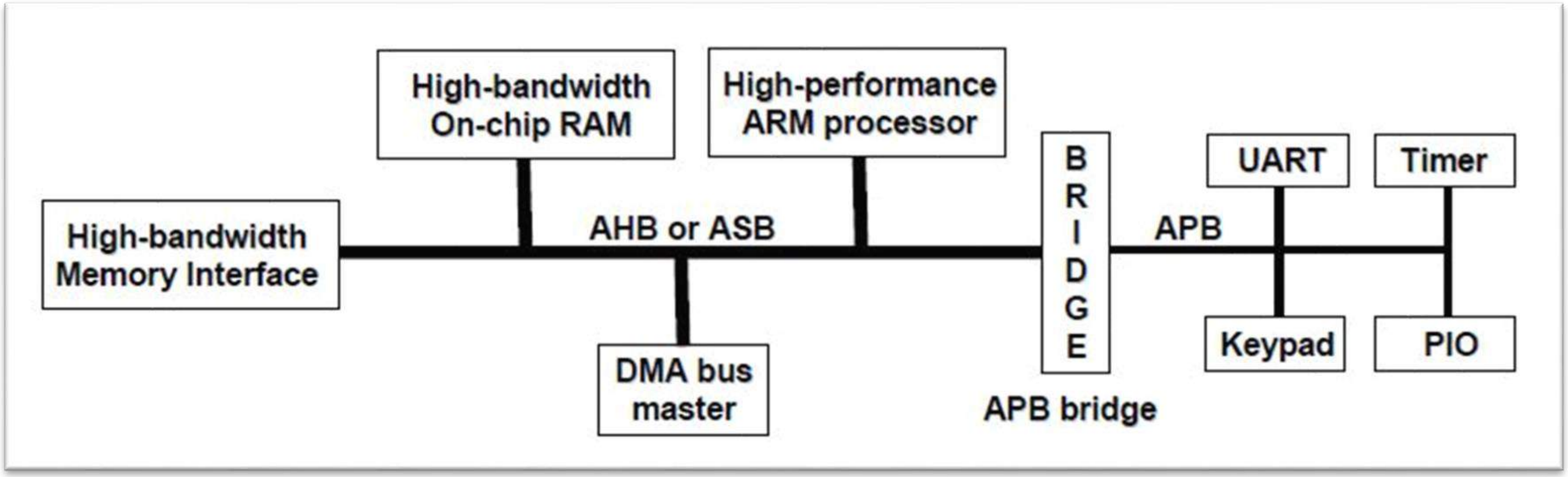
- ✓ It is comprised of many Bus Functional Models (BFM)
- ✓ They look like real components, but they are part of the testbench, not the RTL design.
- ✓ Build equivalent components in your testbench that can generate stimulus and check the response,
- ✓ These are not detailed, non-synthesizable models,
- ✓ High level transactors that obey the protocol, and execute more quickly.





AMBA system

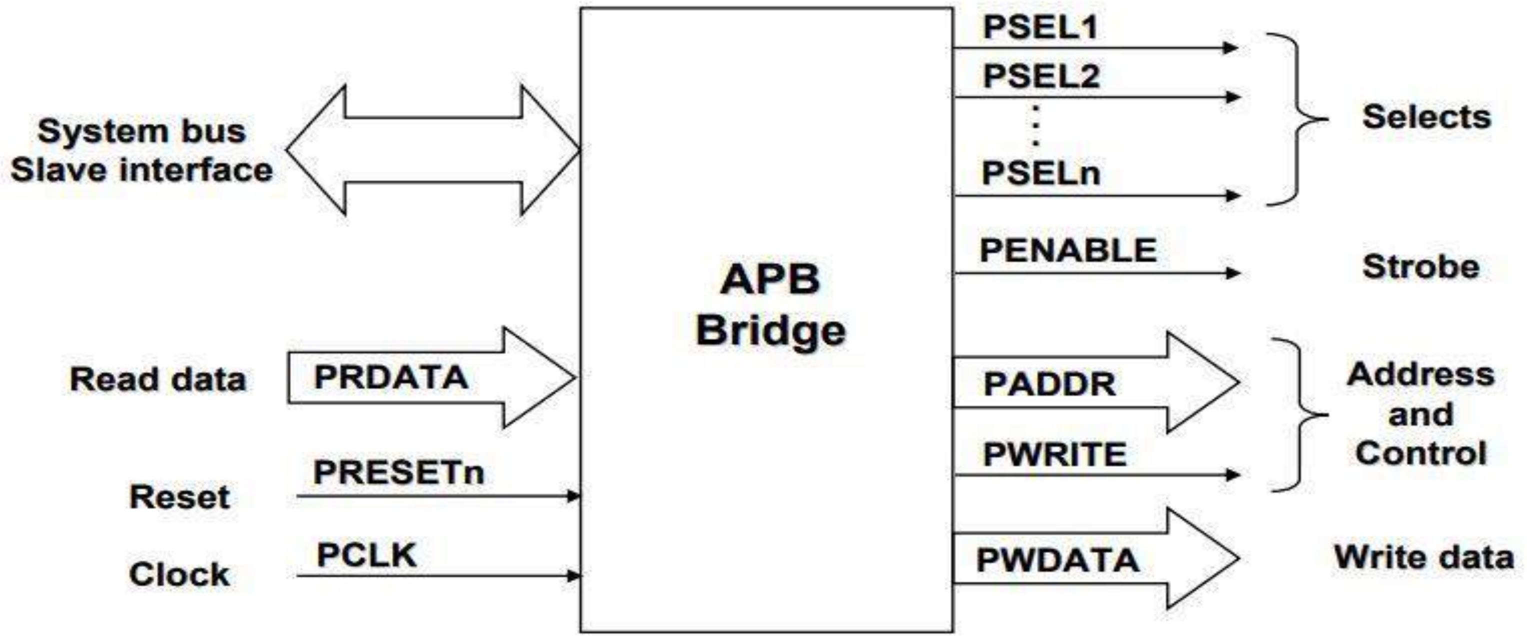
- The ARM advanced microcontroller bus architecture(AMBA) is an open-standard, onchip interconnect specification for the connection and management of functional blocks in SOC designs


High-bandwidth On-chip RAM
High-performance ARM processor
High-bandwidth Memory Interface
AHB or ASB
DMA bus master
BRIDGE
APB bridge
APB
UART
Timer
Keypad
PIO

APB Bridge Interface

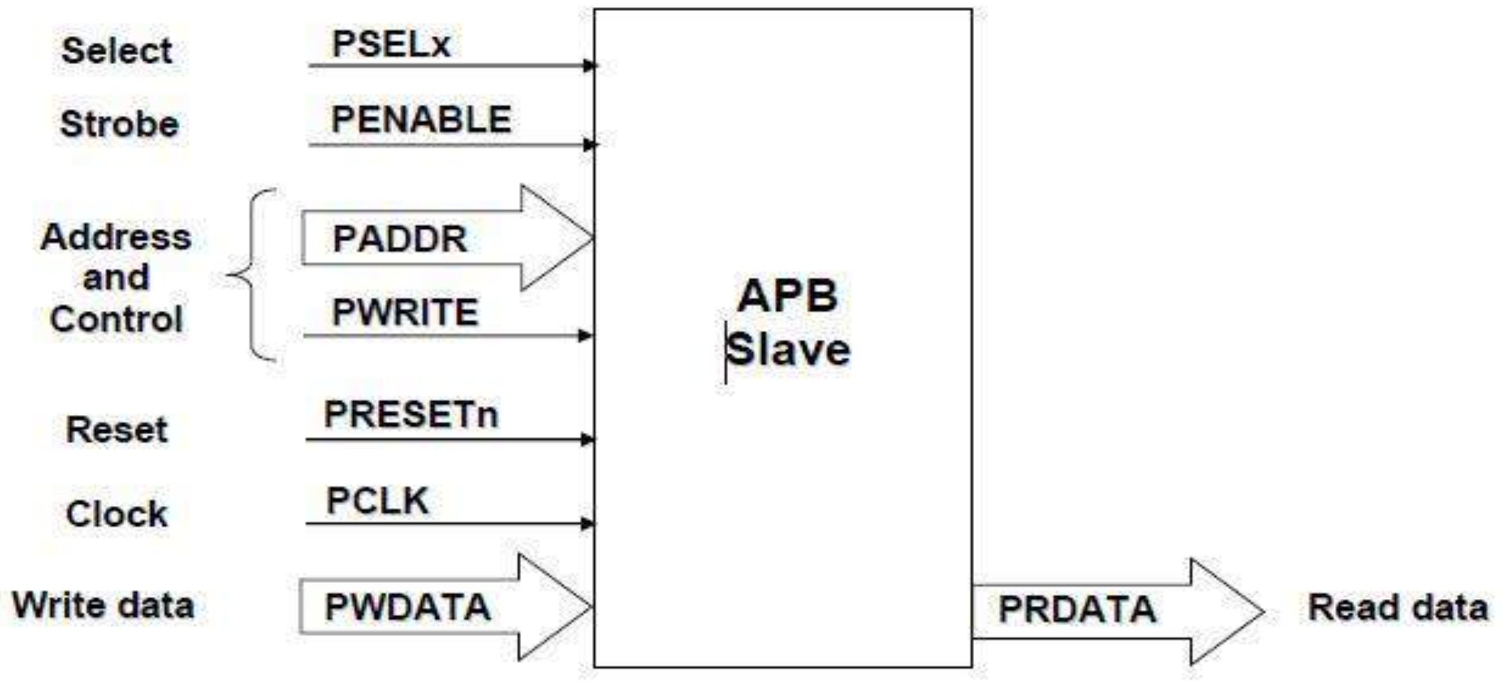
- APB bridge is the only bus master on the AMBA APB.

.

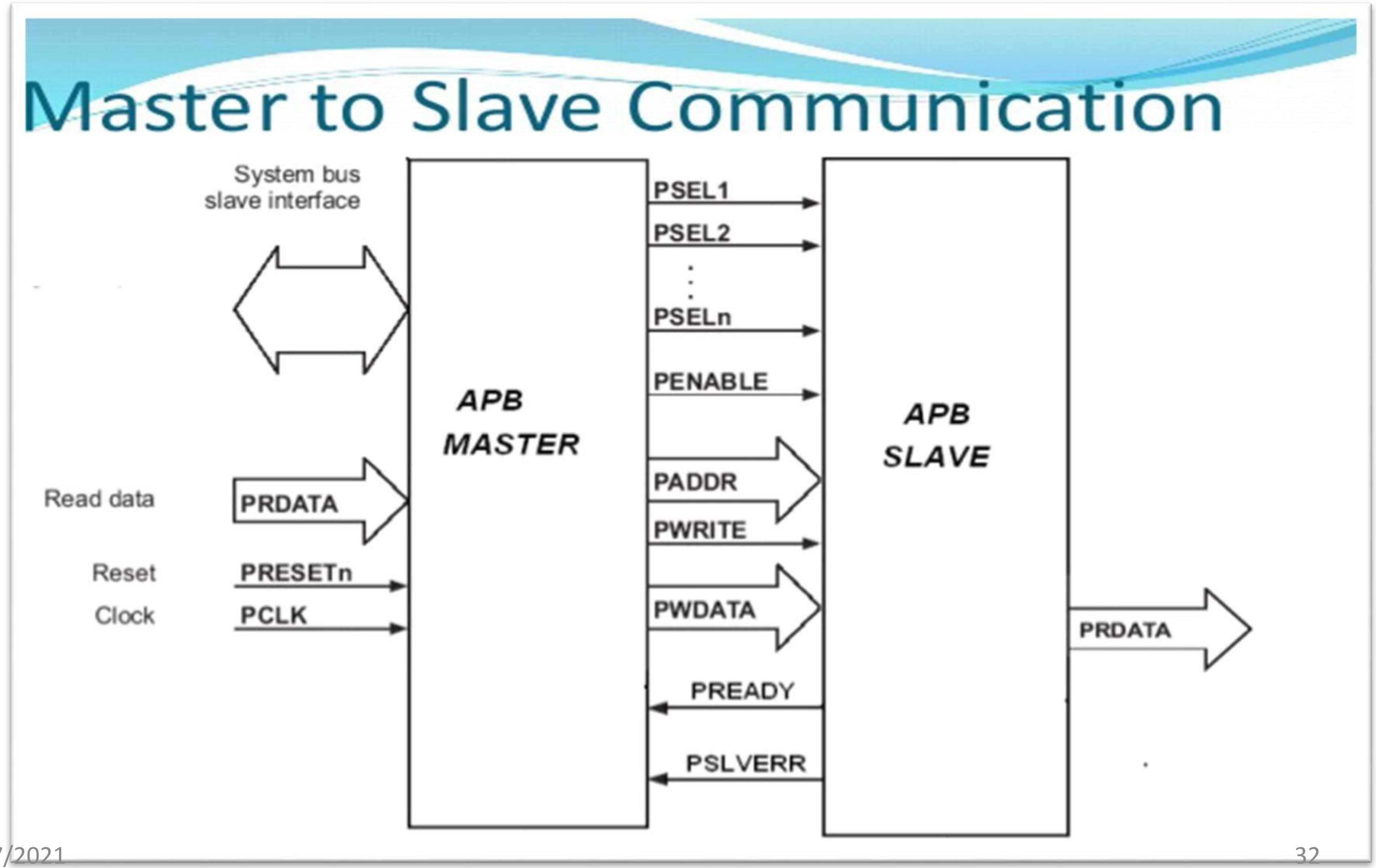


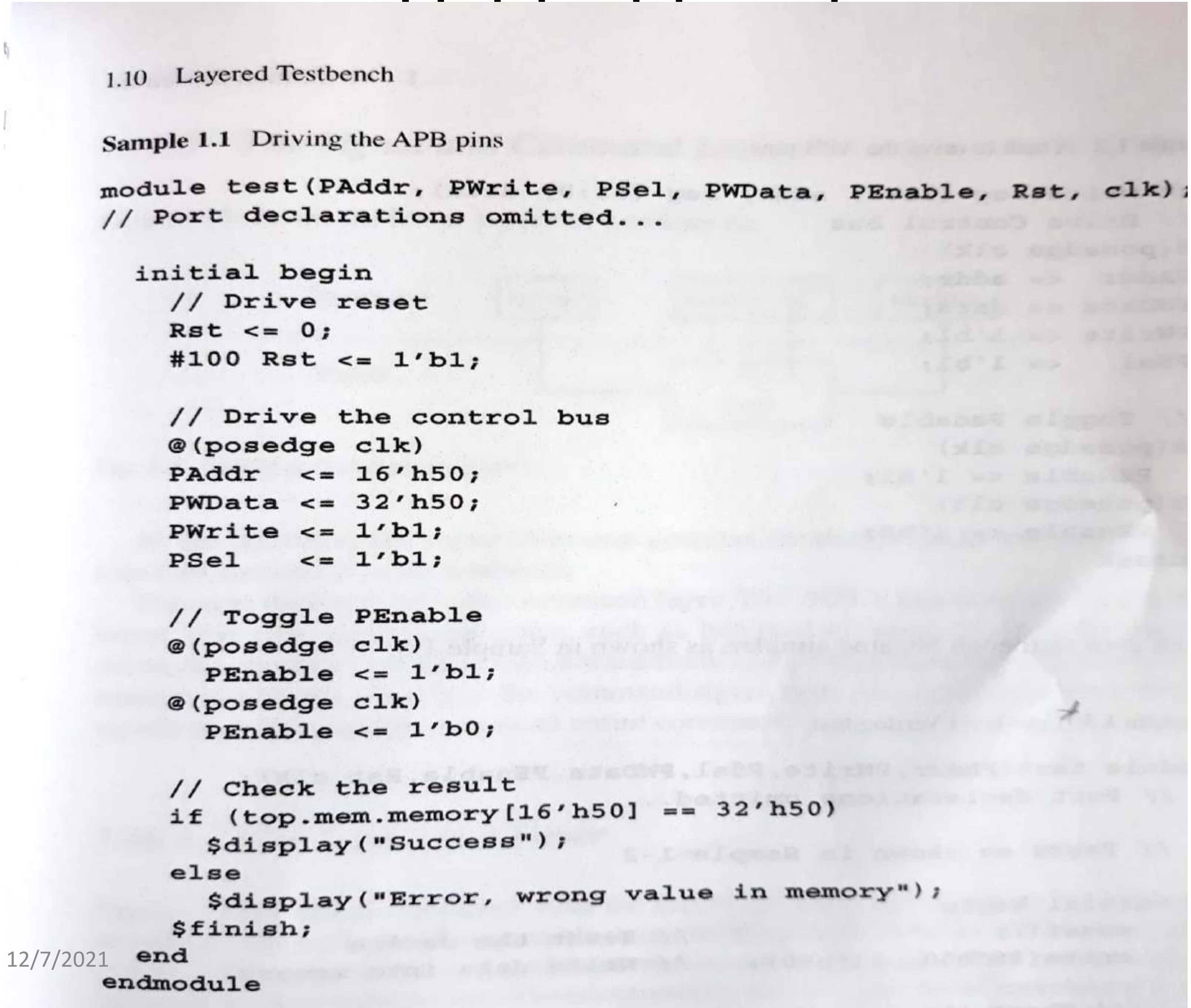
APB Slave Interface

- APB slaves have a simple, yet flexible, interface.



MASTER-SLAVE COMMUNICATION (APB)





Sample 1.1 Driving the APB pins

```
module test(PAddr, PWrite, PSel, PWData, PEnable, Rst, clk);
// Port declarations omitted...

  initial begin
    // Drive reset
    Rst <= 0;
    #100 Rst <= 1'b1;

    // Drive the control bus
    @(posedge clk)
    PAddr  <= 16'h50;
    PWData <= 32'h50;
    PWrite <= 1'b1;
    PSel   <= 1'b1;

    // Toggle PEnable
    @(posedge clk)
      PEnable <= 1'b1;
    @(posedge clk)
      PEnable <= 1'b0;

    // Check the result
    if (top.mem.memory[16'h50] == 32'h50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

Flat test bench

- Individual values are assigned one after the other and verified in the test bench.
- Length of the test bench is more.

Example 1-2 A task to drive the APB pins

- task write(reg [15:0] addr, reg [31:0] data);
- // Drive Control bus
- @(posedge clk)
- PAddr <= addr;
- PWData <= data;
- PWrite <= 1'b1;
- PSel <= 1'b1;
- // Toggle Penable
- @(posedge clk)
- PEnable <= 1'b1;
- @(posedge clk)
- PEnable <= 1'b0;
- endtask

Example 1-3 Low-level Verilog test(using task)

- module test (PAddr, PWrite, PSel, PRData, Rst, clk);
- // Port declarations omitted...
- initial begin
- reset(); // Reset the device
- write(16’h50, 32’h50); // Write data into memory
- // Check the result
- if (top.mem.memory[16’h50] == 32’h50)
- $display("Success");
- else
- $display("Error, wrong value in memory");
- $finish;
- end
- endmodule

Layered Testbench Components

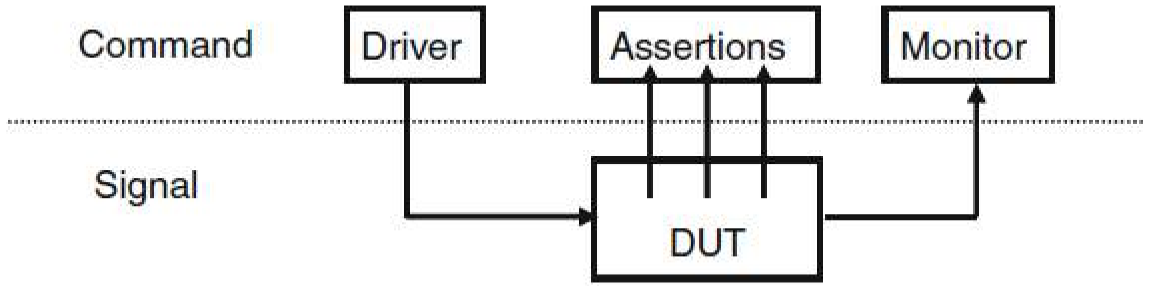
It divides the code into smaller pieces that can be developed separately.

Signal layer:

- ✓ It contains the design under test and the signals that connect it to the test bench.

Command layer:

- ✓ The DUT's inputs are driven by the driver that runs single commands, such as bus read or write.
- ✓ The DUT's output drives the monitor that takes signal transitions and groups them together into commands.
- ✓ Assertions looks at individual signals and also changes across an entire command.

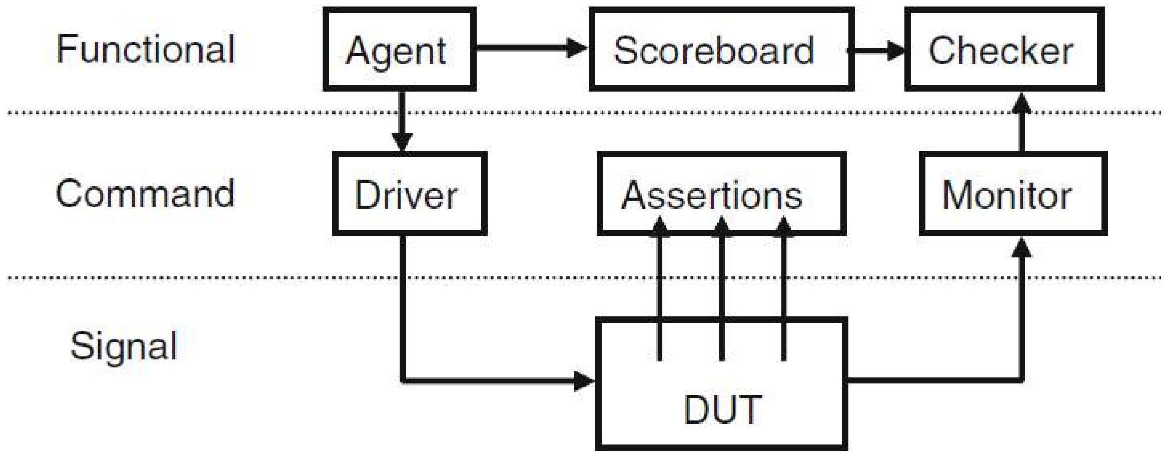


Signal and command layers

Layered Testbench Components

The Functional Layer:

- ✓ The functional layer feeds down into the command layer.
- ✓ The agent block (called the transactor) receives higher-level transactions such as DMA read or write and breaks them into individual commands or transactions.
- ✓ These commands are also sent to the scoreboard that predicts the results of the transaction.
- ✓ The checker compares the commands from the monitor with those in the scoreboard.



Testbench with functional layer added

The Scenario Layer

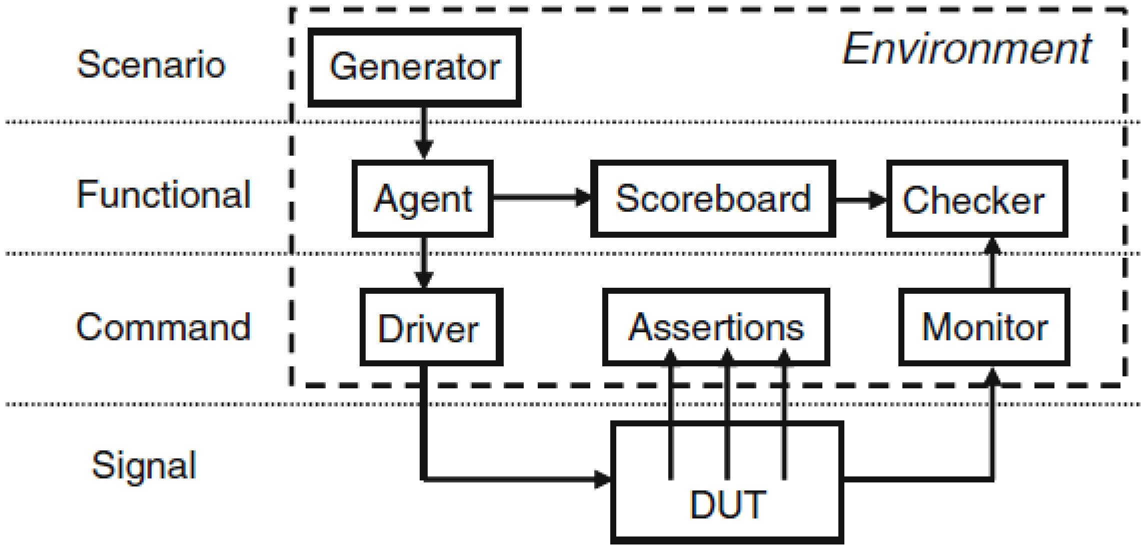
The functional layer is driven by the generator in the scenario layer, as shown in Figure

Scenario? Helps to make sure that this device accomplishes its intended task.

MP3 player →Functionality – can concurrently play music from its storage, download new music from a host and respond to input from the user (adjusting the volume and track controls).

Each of these operations is a scenario.

- Downloading a music file takes several steps, such as control register reads and writes to set up the operation, multiple DMA writes to transfer the song, and then another group of reads and writes such as track size and memory location.

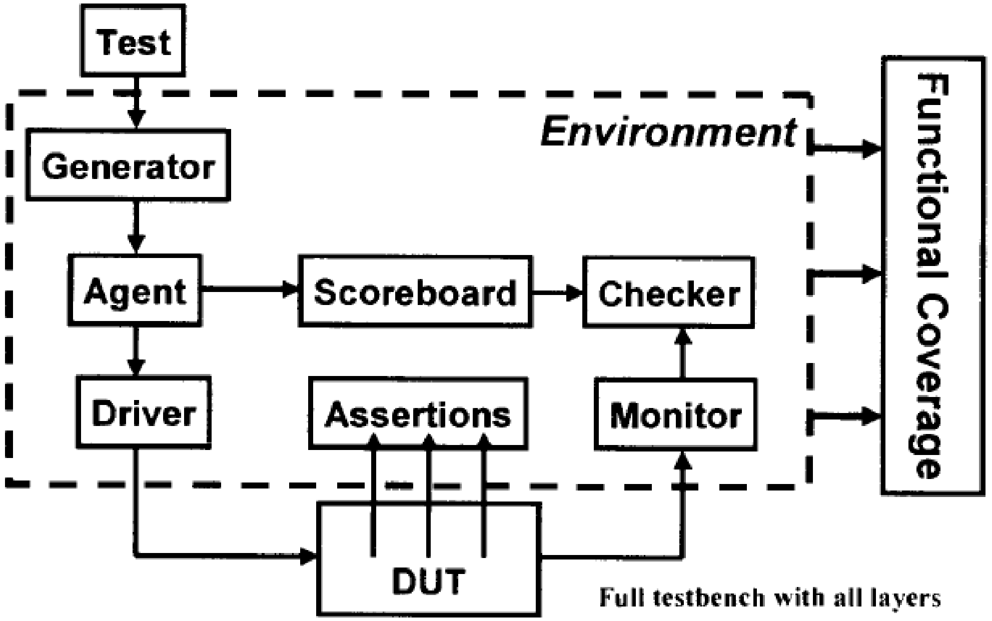


Testbench with scenario layer added

The Test Layer and Functional Coverage- https://www.edaplayground.com/x/3X7H

Test layer: Top of the testbench.

- The test contains the constraints to create the stimulus.
- Functional coverage measures the progress of all tests in fulfilling the verification plan requirements.
- The functional coverage code changes through the project as the various criteria complete. (As modified this layer is not part of the environment).
- ir



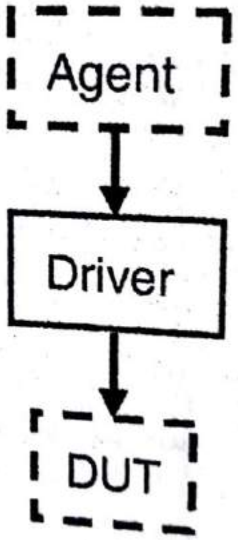
Full testbench with all layers

Creating a simple Driver

- The driver shown in the fig receives commands from the agent.
- Driver may inject errors or delays, it then breaks the command down in to individual signal changes such as bus requests and handshakes.

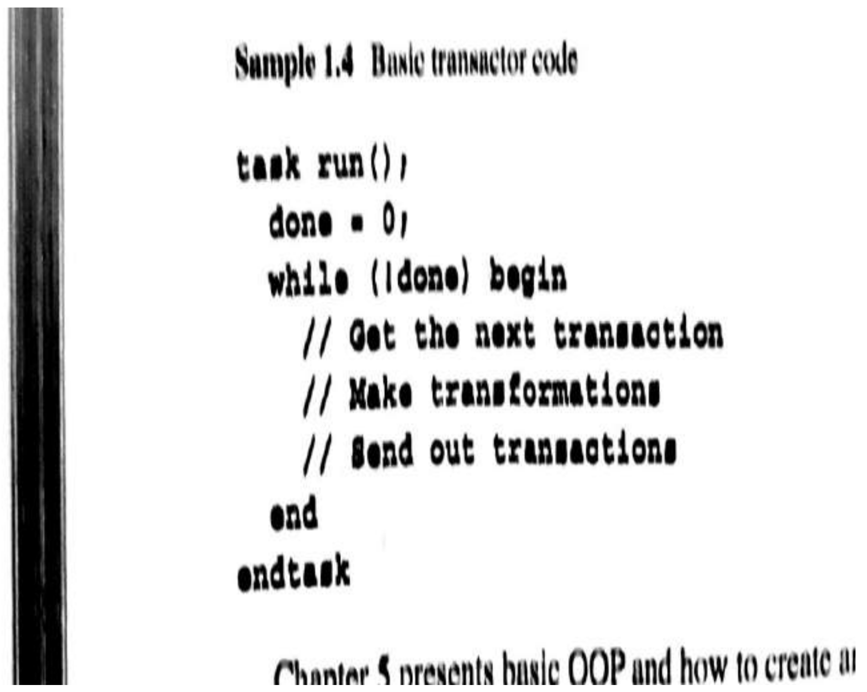
Creating a simple Driver

ok at one of the blocks, the driver.



s for the driver

vn in Fi



Sample 1.4 Basic transactor code

```
task run();
  done = 0;
  while (!done) begin
    // Get the next transaction
    // Make transformations
    // Send out transactions
  end
endtask
```

Chapter 5 presents basic OOP and how to create a

Simulation Environment Phases

- The 3 primary phases are –1. build, 2.run and 3. wrap-up.
- These are a subset of the many phases in UVM and VMM
- The Build phase: is divided in to generate configuration, build environment, reset the DUT and configure the DUT.
- Generate Configuration: randomize the configuration of the DUT and the surrounding environment.
- Build environment: allocate and connect the test bench components based on the configuration.
- Configure the DUT: Based on generated configuration from the first step, load the DUT command registers.

- The Run phase: is where the test actually runs.

Steps are: i)start environment- run the test bench components such as BFMs and stimulus generators.

ii) Run the test: start the test and then wait for it to complete.

3. The Wrap-up phase has two steps;

i) Sweep; after the lowest layer completes, you need to wait for the final transactions to drain out of the DUT.

ii). Report; once the DUT is idle, sweep the test bench for lost data. Sometimes the score board holds transactions that never came out, because they were dropped by the DUT. Create the final report.

SV features: Global declarations

- Allows global variables, type definitions, functions and other information to be declared, that are shared by all levels of hierarchy in the design.

Relaxed data type rules

- Allow variable types to be used in almost any context and make it much easier to write hardware models without concern about which data type class to use
- Net data types (such as wire, wand, wor)
- Variables (such as reg, integer)

Built-in Data Types

- **Adds several new data types, which allow hardware to be modeled at more abstract levels, using data types more intuitive to C programmers**
  - **class** — an object-oriented dynamic data type, similar to C++ and Java.
  - **byte** — a 2-state signed variable, that is defined to be exactly 8 bits.
  - **shortint** — a 2-state signed variable, that is defined to be exactly 16 bits.
  - **int** — a 2-state signed variable, similar to the "int" data type in C, but defined to be exactly 32 bits.
  - **longint** — a 2-state signed variable, that is defined to be exactly 64 bits.
  - **bit** — a 2-state unsigned data type of any vector width.
  - **logic** — a 4-state unsigned data type of any vector width, equivalent to the Verilog "reg" data type.
  - **shortreal** — a 2-state single-precision floating-point variable, the same as the "float" type in C.
  - **void** — represents no value, and can be specified as the return value of a function.
- **User defined types**
  - Typedef unsigned int uint;
    uint a, b;

Data Types (Cont..)

- **Enumerated types(set of values)**
  - **enum** {red, green, blue} RGB;
  - Built in methods to work with
- **Structures and unions;**

```
Struct {
bit[15:0 ]opcode;
Logic[23:0] address
} IR;
IR.opcode = 1  or IR = {5, 200};
```

- 2 State Type; improves simulator performance and reduce memory

Logic Type

```
module logic_data_type(input logic rst_h);

parameter CYCLE = 20;

logic q, q_l, d, clk, rst_l;

initial begin

clk <= 0; // Procedural assignment

forever #(CYCLE/2) clk = ~clk;

end

assign rst_l = ~rst_h; // Continuous assignment

not n1(q_l, q); // q_l is driven by gate

my_dff d1(q, d, clk, rst_l); // d is driven by module

endmodule
```

- Logic Type data:

We can use logic type of data to find netlist bugs as this type can only have a single driver. A signal with multiple driver declared as logic type will definitely give compilation error.

ex- bidirectional bus should be declared as a wire or tri.

SystemVerilog Data Types

- Literal value assignments in Verilog
  - In verilog a vector can be easily filled with all Xs, all Zs or all zeros

```
parameter SIZE=64;
reg [SIZE-1:0] data;
data=0;
data='bz;
data='bx;
data=64'hFFFFFFFFFFFFFFFF;
```

Fills all bits of data with zero
Fills all bits of data with Zs
Fills all bits of data with Xs
Fills all bits of data with 1s

Filling a vector with literal value in Verilog

Redefine size to 128 and fill the vector with 0s,Zs,Xs and 1s

```
parameter SIZE=128;
reg [SIZE-1:0] data;
data=0;
data='bz;
data='bx;
data=128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
```

Fills all bits of data with zero
Fills all bits of data with Zs
Fills all bits of data with Xs
Fills all bits of data with 1s

Filling a vector with literal value by changing the size in Verilog

Each of the assignments in the above example are scalable except filling the vector with all ones

SystemVerilog Overview

- **New data types**
  - Two-state: better performance, reduced memory usage
  - Queues, dynamic and associative arrays and automatic storage: reduced memory usage, built in support for searching and sorting
  - Unions and packed structures: allows multiple views of the same data
  - Classes and structures: support for abstract data structures
  - Strings: built in string support
  - Enumerated types: code is easier to write and understand

SystemVerilog Data Types

- Literal value assignments in SystemVerilog
  - SystemVerilog enhances assignment of a literal value in two ways:
    - ➢ Allows specifying a fill value without having to specify a radix of binary, octal or hexadecimal
    - ➢ The fill value can also be a logic 1

```
parameter SIZE=64;
logic [SIZE-1:0] data;
data='0;
data='z;
data='x;
data='1;
```

`data='0;`	Fills all bits of data with zero
`data='z;`	Fills all bits of data with z
`data='x;`	Fills all bits of data with x
`data='1;`	Fills all bits of data with 1s

Filling a vector with literal value in SystemVerilog

SystemVerilog Built in Data Types

- The 4-state `logic` type
  - Verilog has two basic data types: `wires` and `regs`
    - Verilog uses `reg` type as a general purpose variable for modeling hardware behavior in `initial` and `always` procedural blocks
      - Context of use of `reg` determines if the hardware represented is combinational logic or sequential
    - Verilog uses `wire` to connect different parts of the design
  - SystemVerilog uses the more intuitive `logic` keyword to represent a general purpose, hardware centric data-type
    - Can be driven by continuous assignments, gates and modules
    - Cannot have multiple drivers (for example for a bidirectional bus use a wire)

`reg` is replaced by `logic`

SystemVerilog Built in Data Types

- The 4-state `logic` type
  - Logic has 4 states

States are 0,1,z,x

```
logic   [31:0] r;              // 4-states
```

  - Some example declarations using the `logic` type

```
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
```

SystemVerilog Data Types

- Two state data types : bit, byte, shortint, int, longint
  - Can only have two states 0 or 1 (x and z are mapped to a 0)
  - Used to improve simulation performance and reduce memory usage
  - Keep away from the DUT (since x or z values are converted to a 0 or 1)

Type	Description	Example
bit	User defined size, unsigned	`bit b; bit [31:0] b32;`
byte	8 bits, signed	`byte b8;`
shortint	16 bits, signed	`shortint s;`
int	32 bits, signed	`int i;`
longint	64 bits, signed	`longint l;`

```
logic [7:0] x

byte x
```

Are they the same?

no

(Byte is signed so can count up to 127 and not 255 as might be expected)

SystemVerilog Data Types

- Two state simulation semantics
  - 4-state begins simulation with `x` and a 2-state begins simulations with a `0`
  - 2-state variables cannot represent an uninitialized state
  - Can legally assign 4-state values to 2-state variables. `X` and `Z` values are mapped to 0 in two state types
  - Use `$isunknown` operator to check if any bit of an expression is x or z

```
logic a; bit b;
b=a;
```

x or z values get converted to a logic "0"

"**`$isunknown`**" operator returns 1 if any bit of the expression is x or z

example:

```
 if($isunknown(iport)==1
     $display (" 4-state variable detected");
```

Returns a "1" on detecting an `x` or a `z` (can be any bit)

Fixed Size Arrays

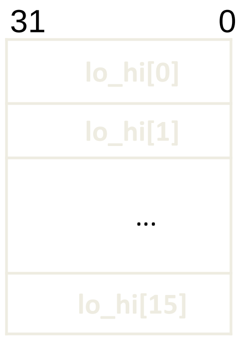
- Fixed Size Arrays
  - Multiple dimensions supported
  - Out-of-bounds write ignored
  - Out-of-bounds read returns x, even for 2-state
  - `byte`, `shortints` and `ints` are stored in a single longword
  - `longint` is stored in two longwords

```
type name [constant];
```

Fixed Size Arrays

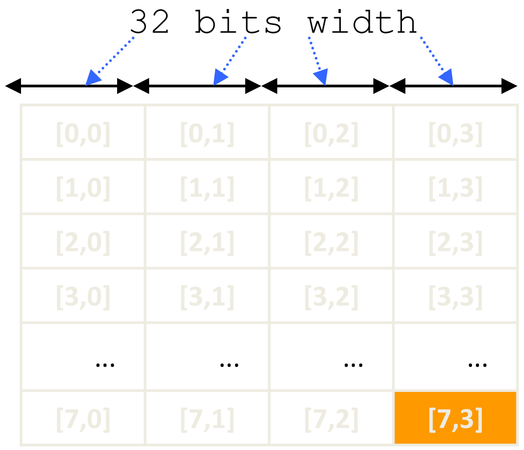
- Single Dimension Array

```
int lo_hi[0:15];        //16 ints [0]..[15]
int lo_hi[16]; //16 ints [0]..[15]
```



- Multidimensional Array

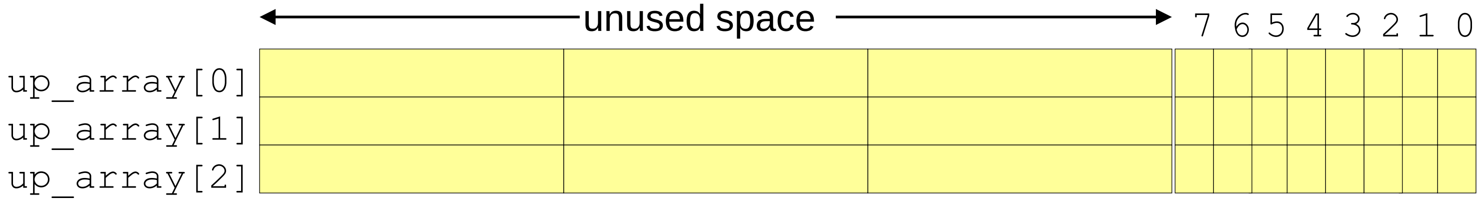
```
int array [0:7][0:3]; //Verbose Declaration,
8 entries by 4.
int array [8][4];     //Compact Declaration
array[7][3]=1;
```



Fixed Size Arrays

- Fixed Size Arrays / Unpacked Arrays
  - Fixed Size arrays are stored on a longword boundary (32 bits)

```
bit [7:0] up_array[3];
```

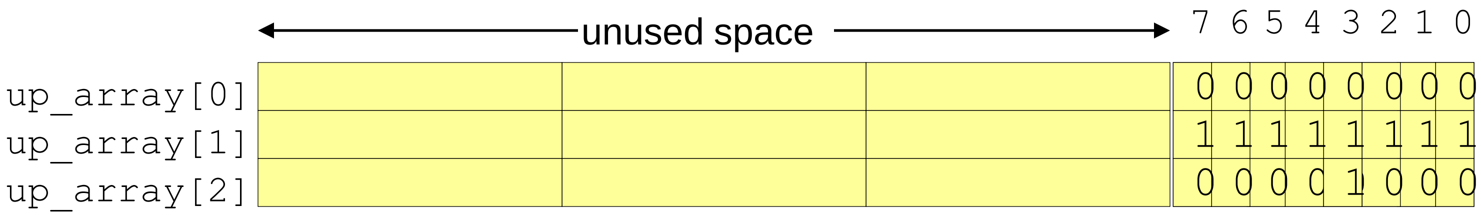


Total memory consumed = 3*32 = 96 bits

Fixed Size Arrays

- Fixed Size Arrays / Unpacked Arrays- stores the values in lower portion of the word.

```
up_array[0]='0;
up_array[1]='1;
up_array[2]='0;
up_array[2][3]=1;
```



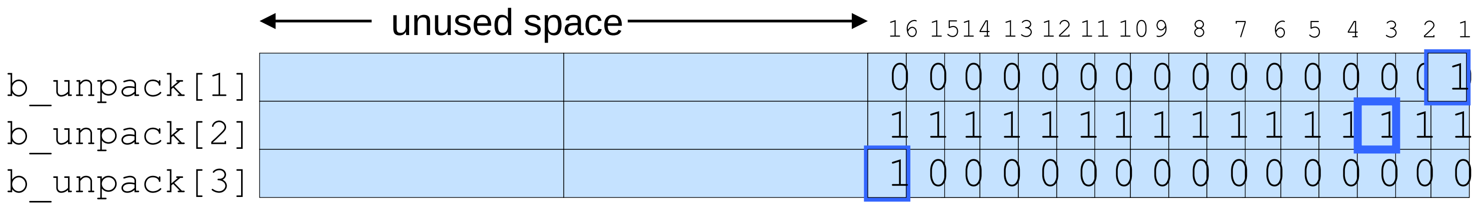
Fixed Size Arrays

- Quiz

```
bit[16:1] b_unpack [1:3];
b_unpack[0]='0;
b_unpack[1]='0;
b_unpack[2]='1;
b_unpack[3]='0;
b_unpack [1][1]=b_unpack [2][3];
b_unpack [3][16]=1;
i=b_unpack [3][0];
```

Out of bound writes are ignored

Out of bound reads return x

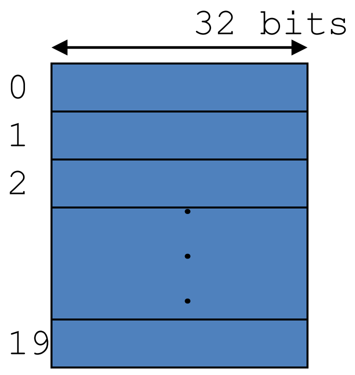

unused space
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
b_unpack[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
b_unpack[2] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
b_unpack[3] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Fixed Size Arrays

- Simplified unpacked array declaration

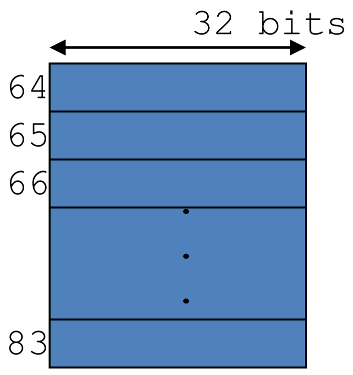
```
int array[20];
```

an array with elements from 0 to 19



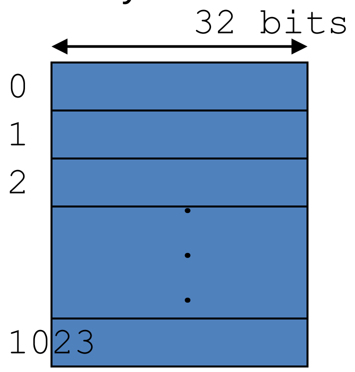
```
int array[64:83];
```

an array with addresses 64 to 83



```
logic [31:0] data [1024];
```

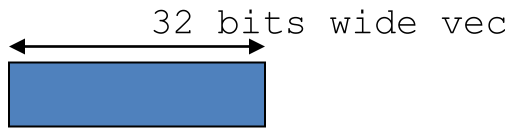
32 bit wide array with 0 to 1023 elements



```
logic [32] d;
```

✗

Not an array



Initializing an array

```
int ascend[4] = '{0,1,2,3};  // Initialize 4
elements
int decend[5];
int md[2][3] = '{'{1,2}, '{3,4,5}};
descend = '{4,3,2,1,0};                // Set 5
elements
descend[0:2] = '{5,6,7};     // Set first 3 elements
ascend = '{4{8}};                  // Four values of 8
```

Initializing Unpacked Arrays

- The array literal '{ }, 'n{ }
  - Can initialize all or some values

```
int array1 [4]='{3,2,4,2};
```

array1

```
array1 [0:2] ='{5,6,7};
```

array1

```
array1 = '{{3}, '{3{8}}};
```

array1

```
int md [2][3]='{'{0,1,2}, '{3,4,5}};
```

md

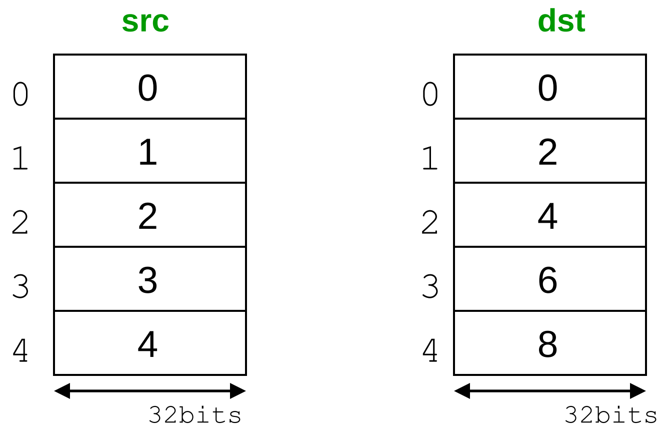
Basic Array Operations

- `for` Operation
  - Most common way of manipulating an array
  - Variable `i` is declared local to the for loop
  - $size returns the size of the array

i is local to the `for` loop

Returns size of array

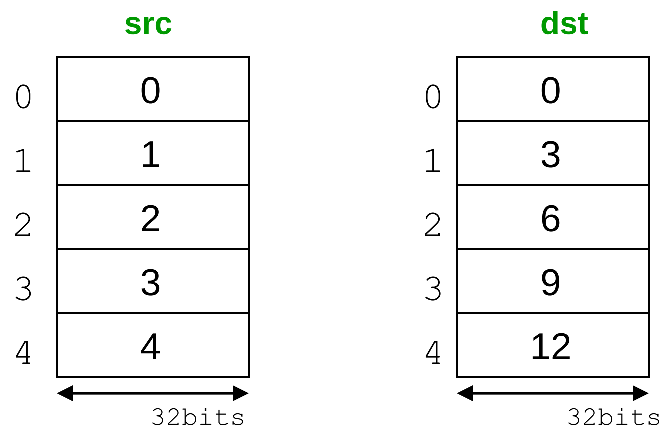
```
initial begin
   bit [31:0] src[5], dst[5];
   for (int i=0; i<$size(src); i++)
        src[i]=i;
        dst[i]=i*2;
end
```



Basic Array Operations

- `foreach` Operation
  - Specify the array name and index in square brackets and SystemVerilog steps through each element
  - foreach loop iterates through each index starting from 0.
  - The index variable is automatically declared and is local to the loop

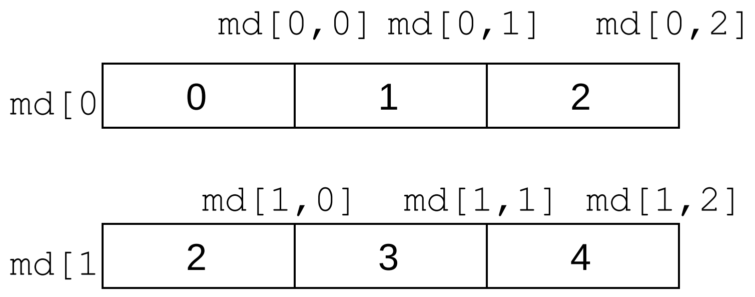
```
initial begin
   bit [31:0] src[5], dst[5];
   int md[2][3];
   foreach(src[i])
        src[i]=i;
   foreach(dst[j])
        dst[j]=src[j]*2+j;
   foreach (md[i,j]);
        md[i,j]=2*i+j;
end
```



Basic Array Operations

- foreach Operation

```
initial begin
   bit [31:0] src[5], dst[5];
   int md[2][3];
   foreach(src[i])
       src[i]=i;
   foreach(dst[j])
       dst[j]=src[j]*2+j;
   foreach (md[i,j]);
       md[i,j]=2*i+j;
end
```



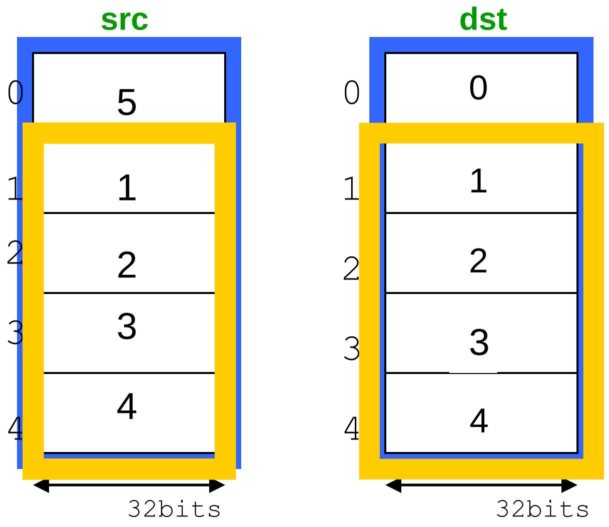
Basic Array Operations

- Copy and Compare
  - Can perform copy and compare without loops (only == and != operations)
  - Cannot perform aggregate arithmetic operations such as addition on arrays

compare operation

copy operation

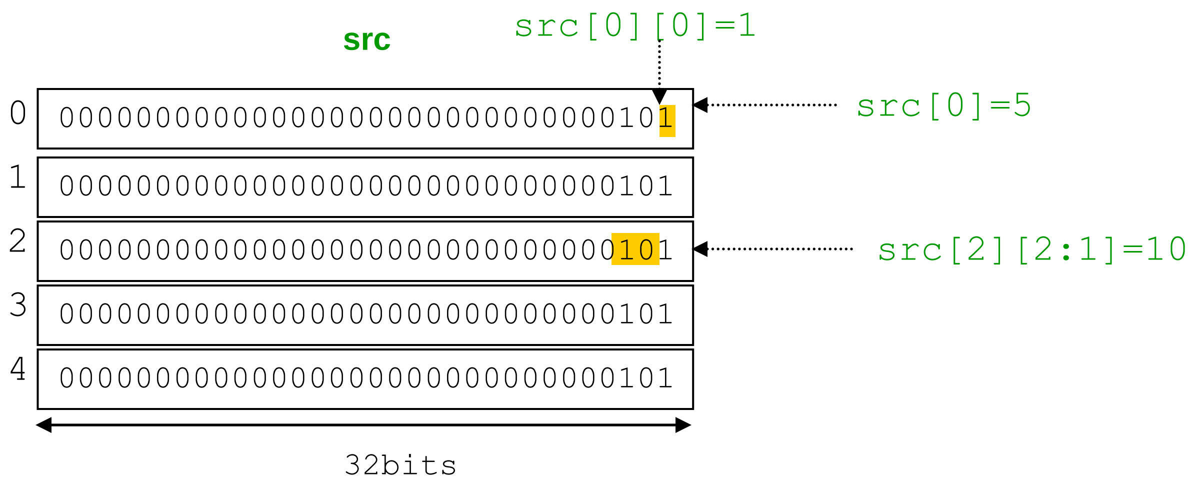
```
initial begin
   bit [31:0] src[5] = '{0,1,2,3,4},
                 dst[5] = '{5,4,3,2,1};
   if(src==dst)
        $display("src==dst");
   else
        $display("src!=dst");
   dst=src;
   src[0]=5;
   if(src[1:4]==dst[1:4])
         $display("src==dst");
   else
         $display("src!=dst");
end
```



Basic Array Operations

- Bit and Array subscripts

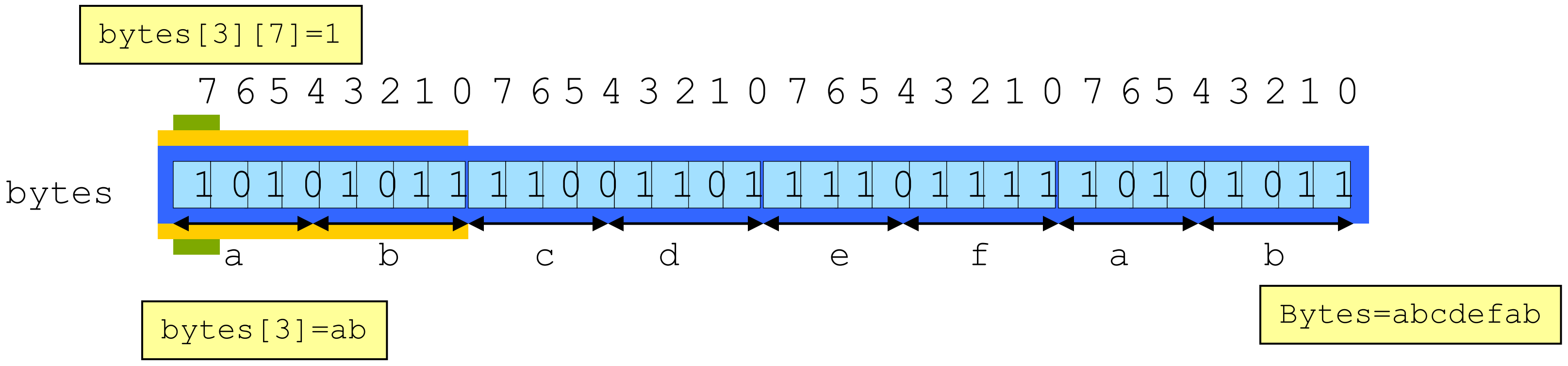
```
initial begin
   bit [31:0] src[5] = '{5{5}},
   $displayb(src[0], src[0][0],src[2][2:1]);
end
```



Packed Arrays

- Packed array
  - Treated as a both an array and a single value
  - Stored as contiguous bits
  - Packed bit and word dimensions should be specified

```
bit  [3:0][7:0]bytes;
bytes=32'habcd_efab;
$displayh(bytes
          bytes[3]
          bytes[3][7]);
```



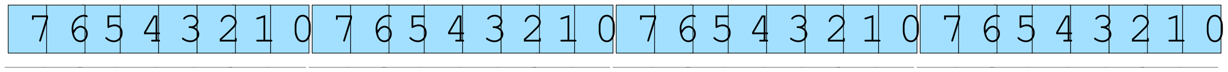
```
bytes: Shows all 32 bits //abcd_efab
bytes[3]: shows the most significant byte //ab
bytes[3][7]: shows the most significant bit //1
```

Packed Array

- Packed array initialization
  - Packed arrays can be initialized at declaration using simple assignments
  - 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;
logic [3:0][7:0] b = {16’hz, 16’h0};
logic [3:0][7:0] c = {16{2’b01}};
```

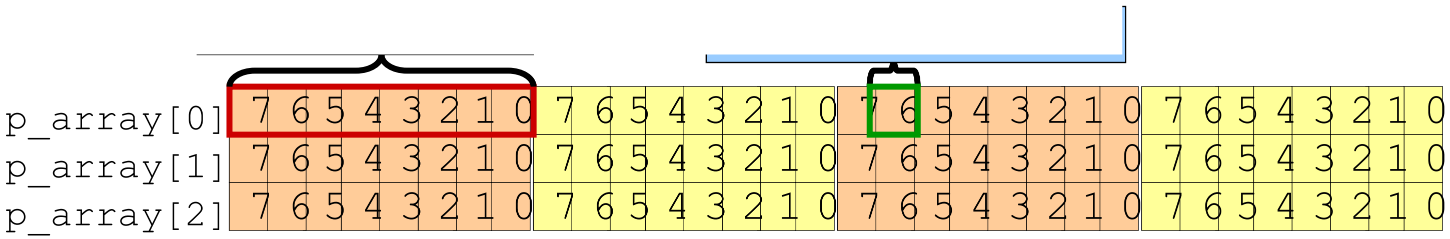
Vector assignment
Concatenate operator
Replicate operator



Mixed Arrays

- Mixing packed and unpacked array
  - Common to have combination of packed and unpacked array
  - p_array is an unpacked array of three packed elements
  - which dimension is unpacked?

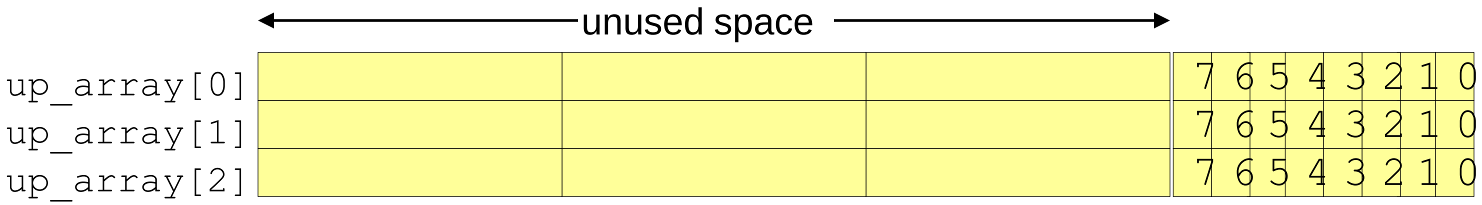
```
bit [3:0][7:0]p_array[0:2];
```



p_array[0] 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
p_array[1] 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
p_array[2] 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

- Unpacked array

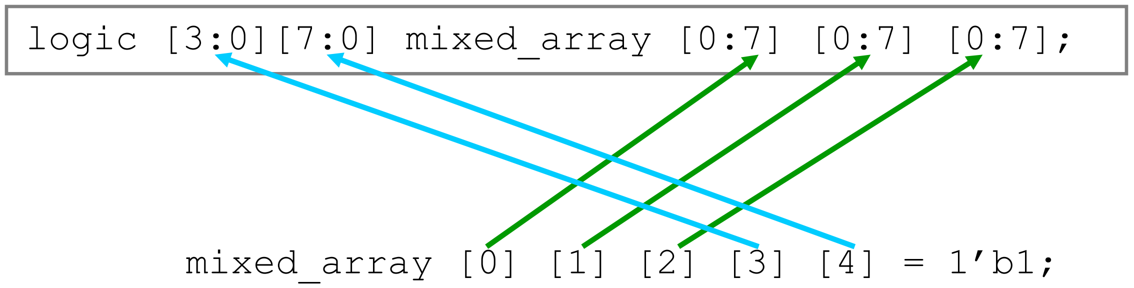
```
bit [7:0] up_array[3];
```



unused space
up_array[0] 7 6 5 4 3 2 1 0
up_array[1] 7 6 5 4 3 2 1 0
up_array[2] 7 6 5 4 3 2 1 0

Mixed Arrays

- Indexing arrays of arrays
  - When indexing mixed arrays, unpacked dimensions are referenced first from the left-most to the right-most dimension
  - Packed dimensions are referenced second from the left-most dimension to the right-most dimension



```
logic [3:0][7:0] mixed_array [0:7] [0:7] [0:7];
```

```
mixed_array [0] [1] [2] [3] [4] = 1’b1;
```

Packed vs Unpacked Arrays

- Using packed or unpacked arrays
  - Helpful in referencing memory as a byte or a longword
  - If you need to wait for a change in an array, you have to use a packed array
  - Only fixed arrays can be packed
  - Dynamic arrays, associative arrays and queues cannot be packed

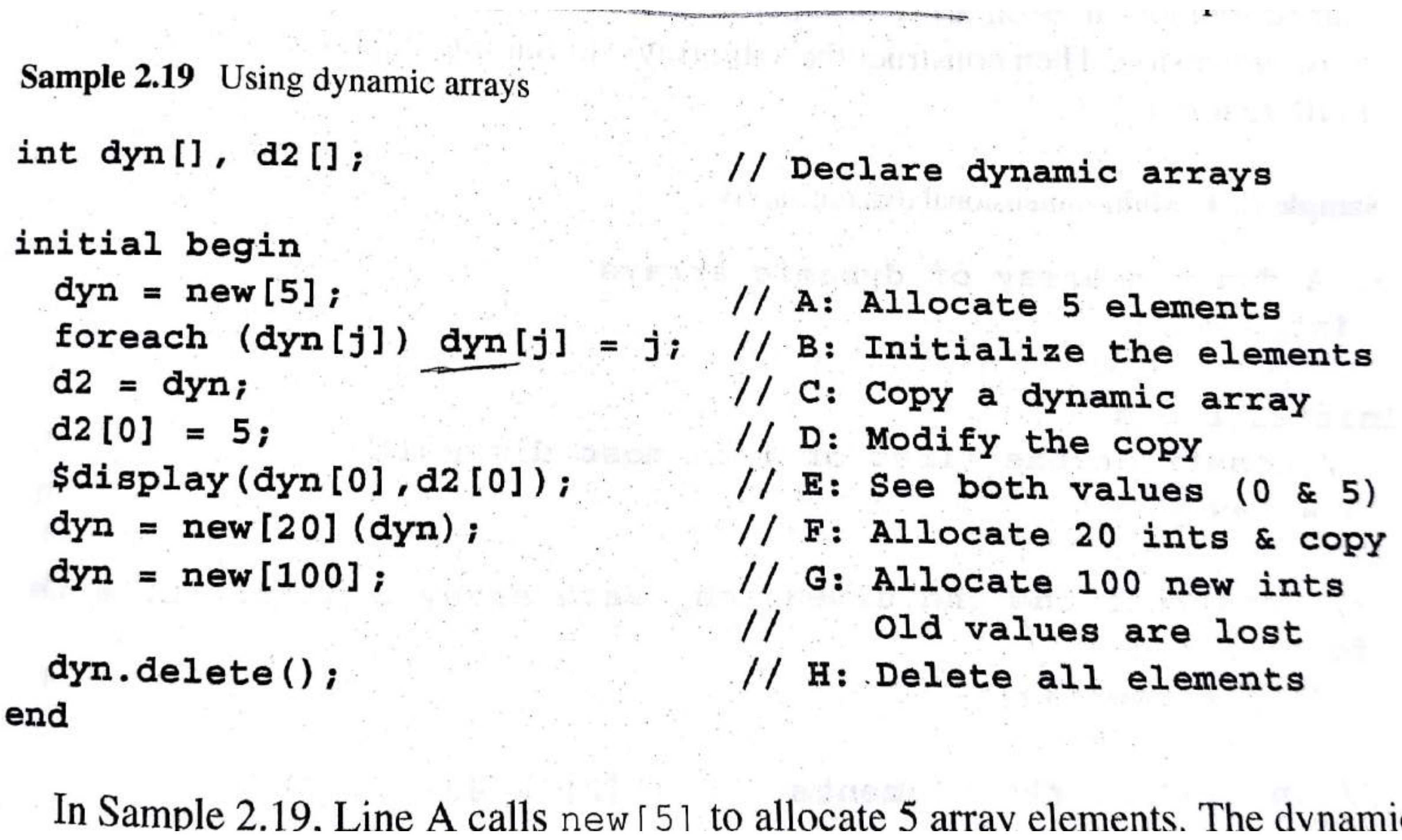
Dynamic Arrays

What if you don't know the size of the array before run time?

- Dynamic Arrays
  - Array size is provided at run-time and not compile-time
  - Can be allocated and resized during simulation
  - Declared with empty subscripts [ ]
  - `new[]` operator is used to allocate space, passing the number of entries
  - Can also pass name of an array to copy values
  - Fixed size arrays can be assigned to dynamic arrays as long as they have the same base type
  - $size function returns the size of a fixed-size or dynamic array

Syntax for dynamic arrays

```
data_type name_of_dynamic_array[];
name_of_dynamic_array=new[number of elements]
```



Sample 2.19 Using dynamic arrays

```
int dyn[], d2[];                        // Declare dynamic arrays

initial begin
  dyn = new[5];                         // A: Allocate 5 elements
  foreach (dyn[j]) dyn[j] = j;          // B: Initialize the elements
  d2 = dyn;                             // C: Copy a dynamic array
  d2[0] = 5;                            // D: Modify the copy
  $display(dyn[0],d2[0]);               // E: See both values (0 & 5)
  dyn = new[20](dyn);                   // F: Allocate 20 ints & copy
  dyn = new[100];                       // G: Allocate 100 new ints
                                        //    Old values are lost
  dyn.delete();                         // H: Delete all elements
end
```

In Sample 2.19. Line A calls `new[5]` to allocate 5 array elements. The dynamic

QUEUES

- System Verilog introduces a new data type, the queue, which provides easy searching and sorting in a structure that is as fast as a fixed-size array but as versatile as a linked list.
- Like a dynamic array, queues can grow and shrink, but with a queue you can easily add and remove elements anywhere.

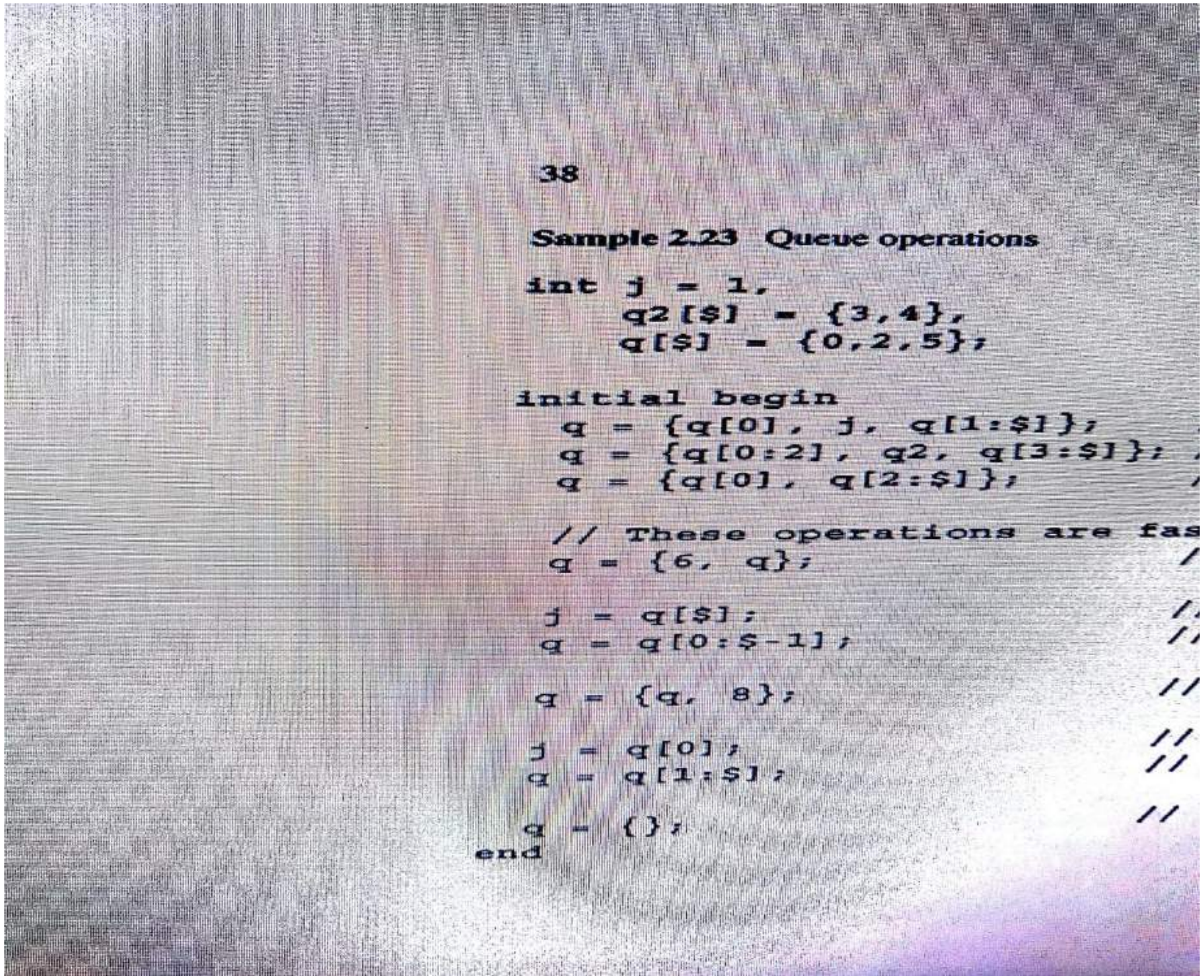
Queues

`type name [$];`

- Can provide easy sorting and searching
- Allocates extra space to quickly add extra elements
- Does not need the new[ ] operator
- Push and pop operations are efficient
- Can add and remove elements from anywhere
- Can copy contents of fixed or dynamic arrays to the queue

Sample 2.22 Queue methods

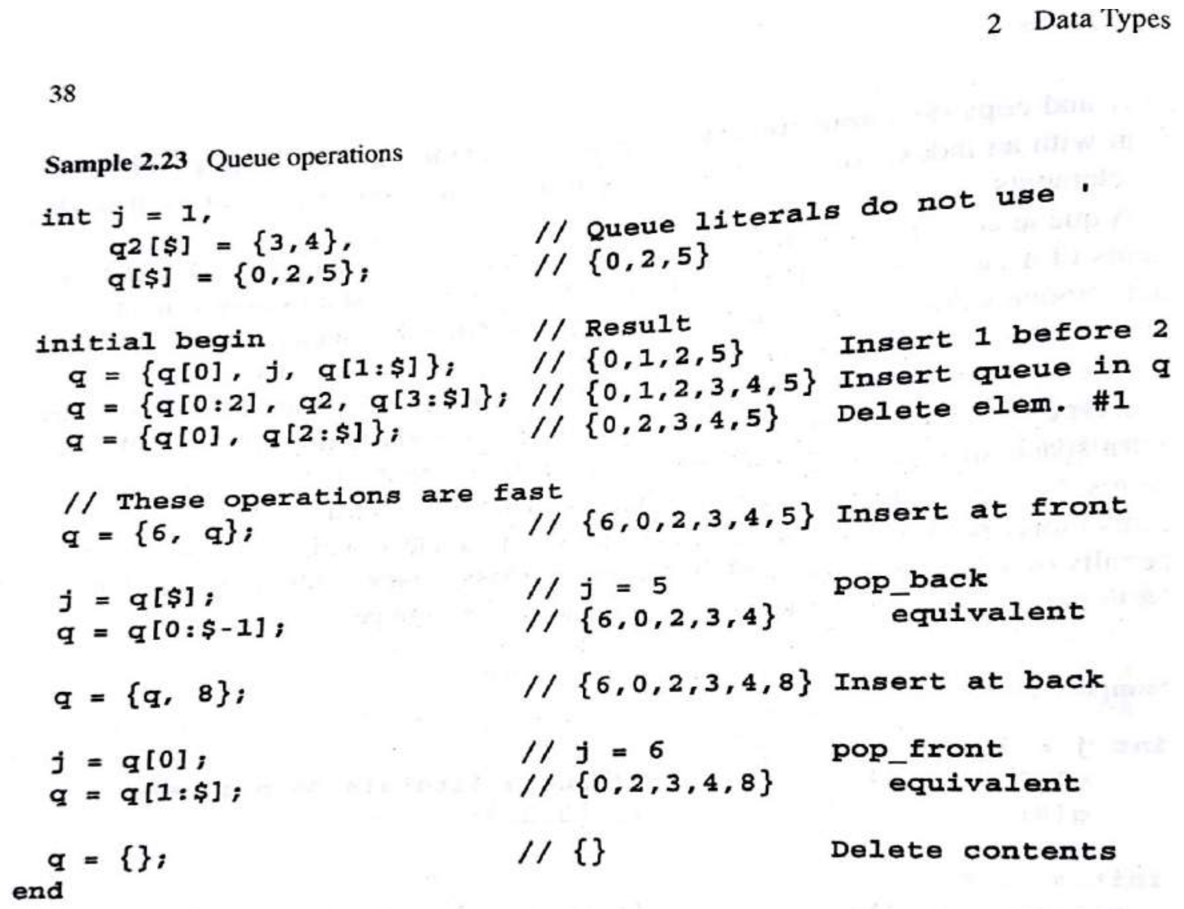
```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use '
    q[$] = {0,2,3};         // {0,2,3}

initial begin
  q.insert(1, j);          // {0,1,2,3}  Insert j before ele #1
  q.delete(1);             // {0,2,3}    Delete element #1

  // These operations are fast
  q.push_front(6);         // {6,0,2,3}  Insert at front
  j = q.pop_back;          // {6,0,2}    j = 3
  q.push_back(8);          // {6,0,2,8}  Insert at back
  j = q.pop_front;         // {0,2,8}    j = 6
  foreach (q[i])
    $display(q[i]);        //            Print entire queue
  q.delete();              // {}         Delete queue
end
```



38

Sample 2.23 Queue operations

```
int j = 1,
    q2[$] = {3,4},
    q[$] = {0,2,5};

initial begin
  q = {q[0], j, q[1:$]};
  q = {q[0:2], q2, q[3:$]};
  q = {q[0], q[2:$]};

  // These operations are fas
  q = {6, q};                    /

  j = q[$];                      //
  q = q[0:$-1];                  //

  q = {q, 8};                    //

  j = q[0];                      //
  q = q[1:$];                    //

  q = {};                        //
end
```



Sample 2.23 Queue operations

```
int j = 1,                      // Queue literals do not use '
    q2[$] = {3,4},              // {0,2,5}
    q[$] = {0,2,5};

initial begin                   // Result
  q = {q[0], j, q[1:$]};        // {0,1,2,5}       Insert 1 before 2
  q = {q[0:2], q2, q[3:$]};     // {0,1,2,3,4,5}   Insert queue in q
  q = {q[0], q[2:$]};           // {0,2,3,4,5}     Delete elem. #1

  // These operations are fast
  q = {6, q};                   // {6,0,2,3,4,5}   Insert at front

  j = q[$];                     // j = 5           pop_back
  q = q[0:$-1];                 // {6,0,2,3,4}       equivalent

  q = {q, 8};                   // {6,0,2,3,4,8}   Insert at back

  j = q[0];                     // j = 6           pop_front
  q = q[1:$];                   // {0,2,3,4,8}       equivalent

  q = {};                       // {}              Delete contents
end
```

Array Reduction Methods

- A basic array reduction method takes an array and reduces it to a single value.
- Array reduction methods (Can be used on unpacked arrays: fixed, dynamic, queue, associative)
  - `sum, product, and, or` and `xor`
  - a.sum of a single bit array returns a single bit value unless stored in a wider value or compared to a wider value

```
bit on[10]
int summ;

initial begin
   foreach (on[i])
     on[i]=i; //0,1,0,1,0,1,0,1,0,1
     $display(on.sum) //1
     summ=on.sum; //summ=5
end
```

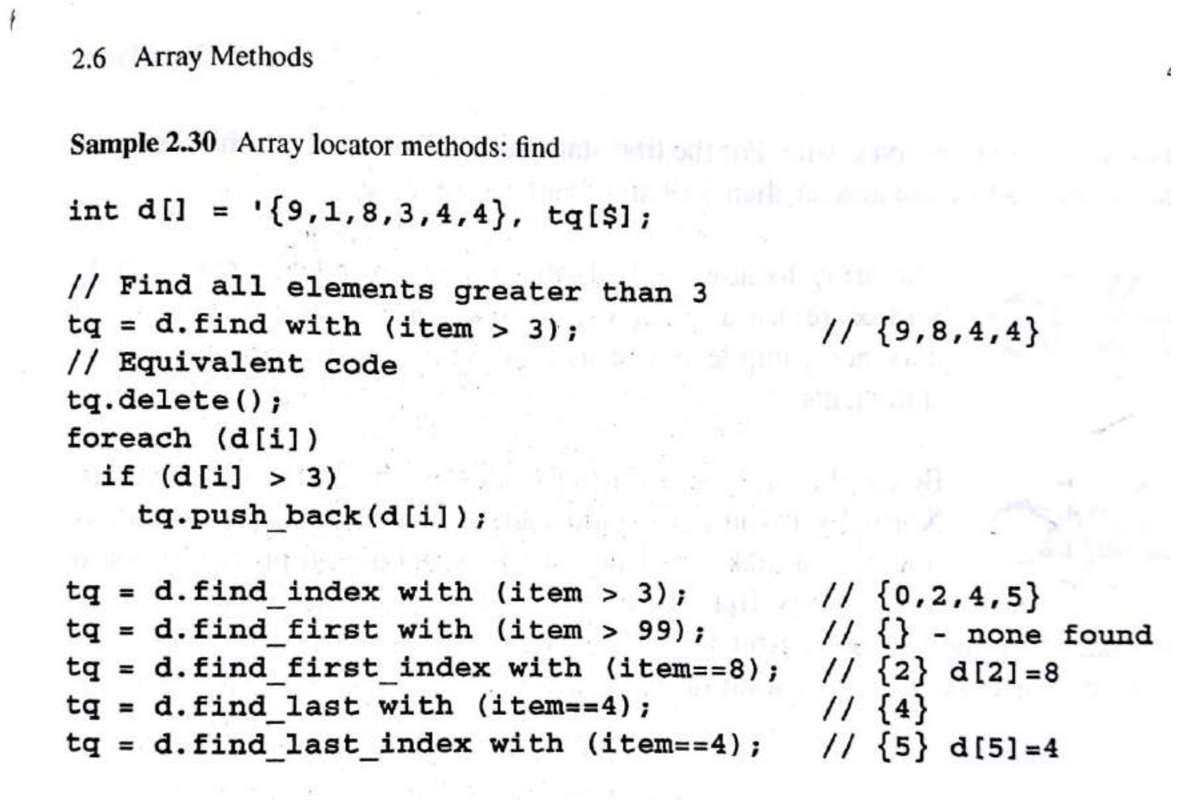
Array Methods

- Array locator methods
  - sort, rsort, reverse, find_first, find_first_with_index
  - Unique- method returns a queue of unique values from the array- duplicate values are not included

```
int q[$] = {1,3,5,7}, tq[$];
int d[]  = '{9,1,8,3,4};
int f[6] = '{1,6,2,6,8,6};

$display(q.sum, q.product);          // 16 105
tq = q.min();                        // {1}
tq = q.max();                        // {7}
tq = f.unique;                       // {1,6,2,8}
tq = d.find with (item > 3);         // {9,8,4}
tq = d.find_index with (item > 3);   // {0,2,4}
```

Array Methods

- Array locator methods
  - find_first, find_first_with_index



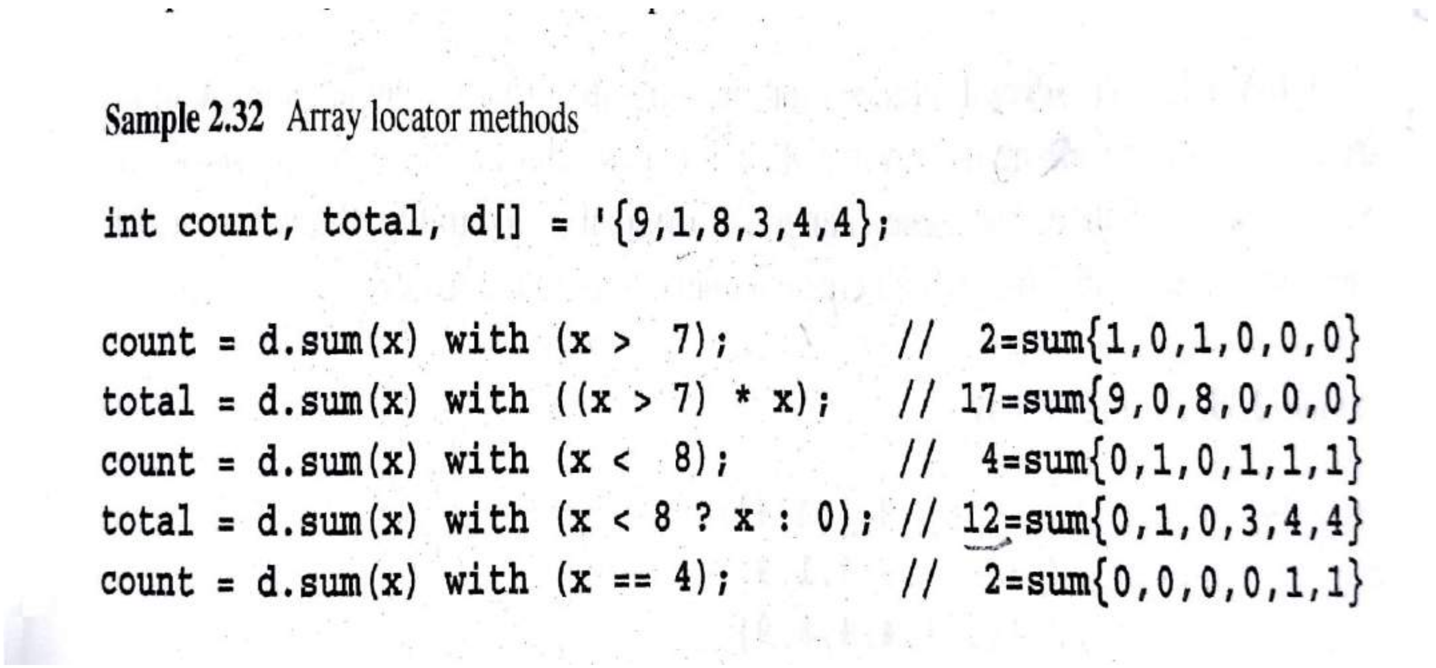
2.6 Array Methods

Sample 2.30 Array locator methods: find

```
int d[] = '{9,1,8,3,4,4}, tq[$];

// Find all elements greater than 3
tq = d.find with (item > 3);                // {9,8,4,4}
// Equivalent code
tq.delete();
foreach (d[i])
 if (d[i] > 3)
   tq.push_back(d[i]);

tq = d.find_index with (item > 3);          // {0,2,4,5}
tq = d.find_first with (item > 99);         // {} - none found
tq = d.find_first_index with (item==8);     // {2} d[2]=8
tq = d.find_last with (item==4);            // {4}
tq = d.find_last_index with (item==4);      // {5} d[5]=4
```

Array Methods

- Array locator methods

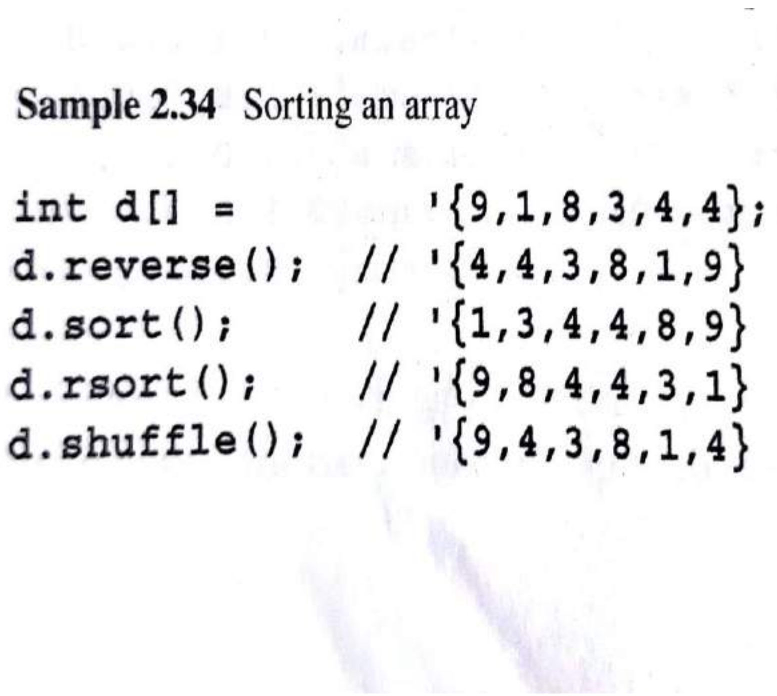


Sample 2.32 Array locator methods

```
int count, total, d[] = '{9,1,8,3,4,4};

count = d.sum(x) with (x >  7);          //  2=sum{1,0,1,0,0,0}
total = d.sum(x) with ((x > 7) * x);     // 17=sum{9,0,8,0,0,0}
count = d.sum(x) with (x <  8);          //  4=sum{0,1,0,1,1,1}
total = d.sum(x) with (x < 8 ? x : 0);  // 12=sum{0,1,0,3,4,4}
count = d.sum(x) with (x == 4);          //  2=sum{0,0,0,0,1,1}
```

Array Methods

- Array Sorting and Ordering
  - sort, rsort, reverse, shuffle



Sample 2.34 Sorting an array

```
int d[] =          '{9,1,8,3,4,4};
d.reverse();    // '{4,4,3,8,1,9}
d.sort();       // '{1,3,4,4,8,9}
d.rsort();      // '{9,8,4,4,3,1}
d.shuffle();    // '{9,4,3,8,1,4}
```

Associative array

- Associative array Stores entries in a sparse matrix
- Associative arrays allocate the storage only when it is used, unless like in the dynamic array we need to allocate memory before using it
- In associative array index expression is not restricted to integral expressions, but can be of any type- Accessed with integer, or string index, single dimension

Associative Arrays

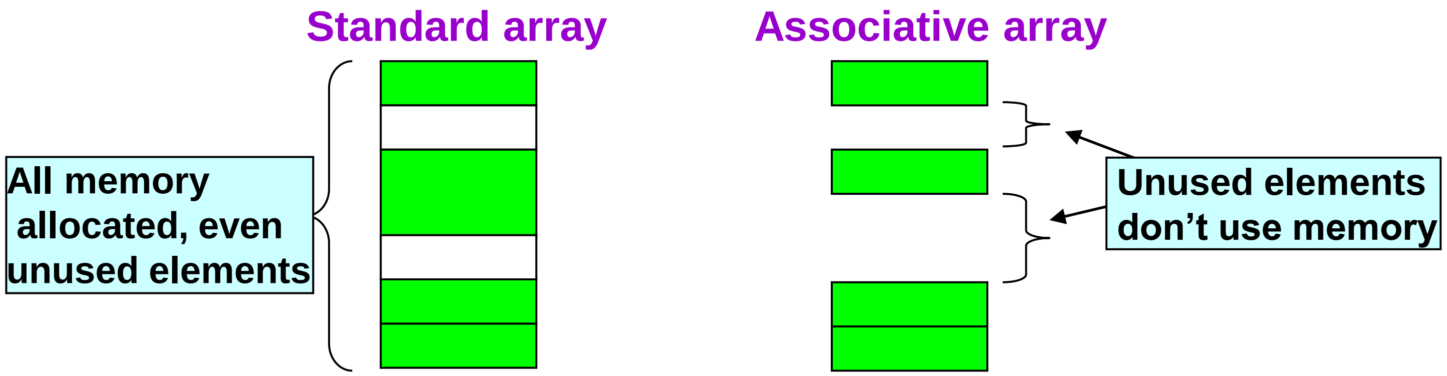
type name [*];

- Array declaration,

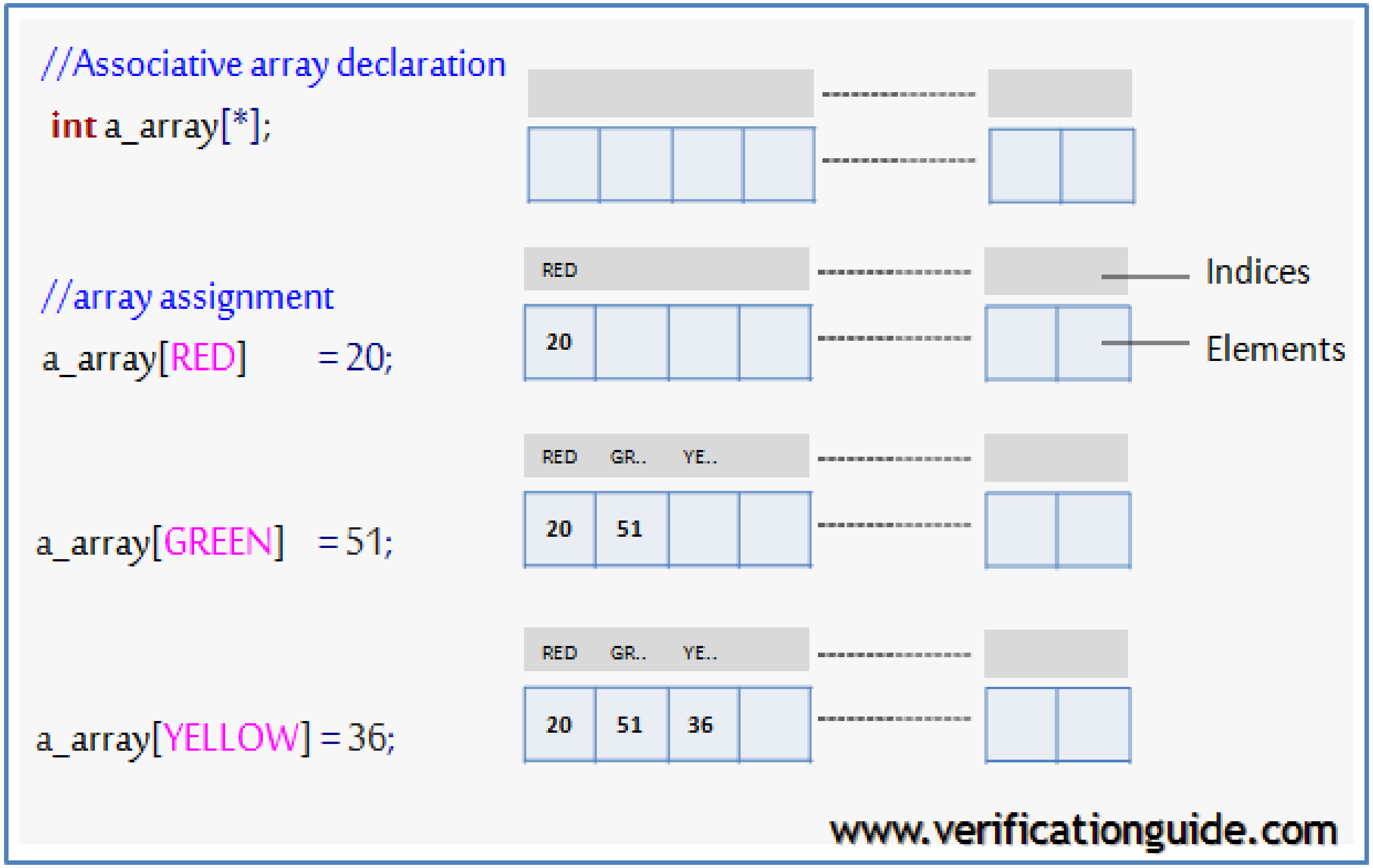
    data_type array_name [index_type];

where:
data_type – data type of the array elements.
array_name – name of the associative array.
index_type – data-type to be used as an index, or *.

* indicates the array is indexed by any integral expression of arbitrary size.


Standard array
Associative array
All memory allocated, even unused elements
Unused elements don't use memory

- Array examples:
- int a_array1[*] ;   // associative array of integer (unspecified index)
- bit [31:0] a_array2[string]; // associative array of 32-bit, indexed by string
- ev_array [myClass];   //associative array of event,indexed by class



```
//Associative array declaration
 int a_array[*];

//array assignment
a_array[RED]     = 20;

a_array[GREEN]   = 51;

a_array[YELLOW] = 36;
```

Associative Arrays

- Operations
  - `foreach` is used with associative arrays
  - Reading from an element that has not been allocated results in a x for 4-state and a 0 for 2-state
  - **num(),** `first, next, prev, delete,` **last() methods are used.**

Structures

- Structures can be created using the `struct` statement
  - A structure is a collection of variables and/or constants that can be accessed separately or as a whole
  - Allows logical groups of signals to be combined together (example: control signals of a bus protocol)

```
struct {
    int           a,b;
    logic [7:0]   opcode;
    logic [23:0]  address;
    bit           error;
} Instruction_Word
```

  - The entire collection can be referenced using the name of the struct

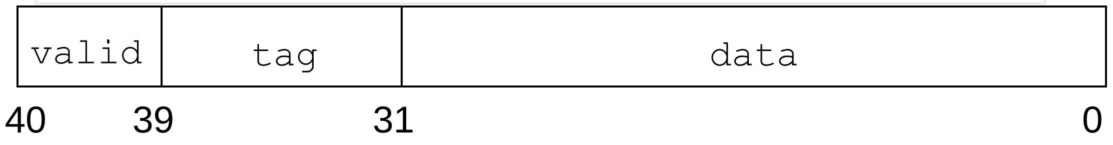
```
<structure.name> <variable.name>
```

```
Instruction_Word.address=24'hF00000;
```

Structures

- Packed structures
  - Structures are unpacked by default
    - Layout varies from tool to tool
  - Structures can be declared as packed structures using `packed` keyword
    - Packed structure stores all elements as contiguous bits
    - First member of the structure is the left-most field of the vector

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



Structures

- Packed structures
    - The members of a packed structure can be referenced by either the name of the member or by part select of the vector

```
data_word.tag = 8'hf0;
data_word [39:32]=8'hf0;
```

- Operations on packed structures
    - Assigning values to a packed structure

```
data_word =  `{1,8'hff, 1024}
```

    - Logical operations that can be performed on vectors can be performed on packed structures

Structures

```
• module struct_tb;
•
•   typedef struct packed {
•     bit [7:0]  addr;
•     bit        valid;
•     bit [31:0] data;
•   } mem_pkt;
•
•   mem_pkt pkt;
•
•   initial begin
•
•     // Initializing Struct
•     pkt = '{8'h6, 1'b1, 32'hC001_0FAB};
•     $display ("pkt = %p", pkt);
•
•     // Change the struct field value
•     pkt.addr = 8'h8;
•     $display ("pkt = %p", pkt);
•
•     // Change the struct field value
•     pkt.data = 32'hFFF0_0FFF;
•     $display ("pkt = %p", pkt);
•   end
• endmodule
```

-
  ```
  pkt = '{addr:'h6, valid:'h1, data:'hc0010fab}
  pkt = '{addr:'h8, valid:'h1, data:'hc0010fab}
  pkt = '{addr:'h8, valid:'h1, data:'hfff00fff}
  ```

Enumerated Types

- An enumerated type defines a set of named values. The simplest enumerated type declaration contains a list of constant names and one or more variables
- enum { red, green, blue, yellow, white, black } Colors;

Here, colors are defined to be variable of the unnamed enumerated int type that includes the members red, green, blue, yellow, white, black.

- The actual values are defaulted to integers starting at 0 and then increase. in the above example by default variable will get the default value of 0,1,2,3,4,5 respectively from red.

Enumerated Types

- Enumerated type values
  - By default the labels of an `enum` are `int`
    - The first label in the enum list is represented with a value 0, the second label with 1 and so on

```
          0      1
enum {red, green, blue} RGB
```

  - SystemVerilog allows the value for each label to be explicitly declared
    - Label values must be unique

```
enum {ONE =1 , FIVE =5, TEN =10} state;
```

    - If each label is unspecified, the value representing each label will be incremented by 1 from the previous label

```
enum {A=1, B, C, X=24, Y, Z} list1;
```

```
enum {A=1, B, C, D=3} list2;
```

Enumerated Types

- Base type of enumerated types
  - By default the labels of an `enum` are `int`
  - SystemVerilog allows base type to be explicitly defined

```
enum bit {TRUE, FALSE} Boolean;
```

```
enum logic [1:0] {WAITE, LOAD, READY} state;
```

- Enum value size
  - ◆ Assigned label values should match the size of base type
  - ◆ Assigned label values should match the base types

```
 enum logic [1:0]
        {WAITE=3'b001,
         LOAD=3'b010,
         READY=3'b100} state;
```

```
enum logic [2:0]
       {WAITE=3'b001,
        LOAD=3'b010,
        READY=3'b100} state;
```

Enumerated Types

- Enum value size
  - Cannot have more labels in the enumerated list than the base type size can represent
  - It is legal to assign x or z to a four state type
    - The label next to the x or z assigned label should have an explicitly defined value

```
enum logic {A=1’b0, B, C} list3;
```

```
enum logic {ON=1’b1, OFF=1’bz}out;
```

```
enum logic [1:0] {WAITE,
                   ERR=2’bxx,
                  LOAD,
                  READY} state;
```

Package in SV

- Packages provide a mechanism for storing and sharing data, methods, functions, property etc that can be reused in multiple other modules or programs.
- Packages provide ways to have common code to be shared across multiple modules. SystemVerilog provides package support to help share following-parameters
- data
- type
- task
- function
- sequence
- property

- **Package Declaration**
- **package definesPkg;**

TypeDefs declaration

typedef enum {FALSE, TRUE} bool;

typedef struct { 10 bit [7:0] addr; 11 bit [7:0] data;

bit wr; 13 } mem_s;

endpackage

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		
!=	logical inequality	&	unary and
===	case equality	~&	unary nand
!==	case inequality	|	unary or
==?	wildcard case equality	~|	unary nor
!=?	wildcard case inequality	^	unary exclusive
<<	shift left	~^	unary exclusive nor
>>	shift right	?:	conditional (ternary)
{}	concatenation		

Assignment:

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